Congestion Avoidance on Manycore High Performance Computing Systems

Miao Luo Dhabaleswar K. Panda
Ohio State University
{luom, panda}@cse.ohio-state.edu

Khaled Z. Ibrahim Costin Iancu
Lawrence Berkeley National Laboratory
{kzibrahim,cciancu}@lbl.gov

ABSTRACT

Efficient communication is a requirement for application scalability on High Performance Computing systems. In this paper we argue for incorporating proactive congestion avoidance mechanisms into the design of communication layers on manycore systems. This is in contrast with the status quo which employs a reactive approach, e.g. *congestion control mechanisms are activated only when resources have been exhausted. We present a core stateless optimization approach based on open loop end-point throttling, implemented for two UPC runtimes (Cray and Berkeley UPC) and validated on InfiniBand and the Cray Gemini networks. Microbenchmark results indicate that throttling the number of messages in flight per core can provide up to 4X performance improvements, while throttling the number of active cores per node can provide additional 40% and 6X performance improvement for UPC and MPI respectively. We evaluate inline (each task makes independent decisions) and proxy (server) congestion avoidance designs. Our runtime provides both performance and performance portability. We improve all-to-all collective performance by up to 4X and provide better performance than vendor provided MPI and UPC implementations. We also demonstrate performance improvements of up to 60% in application settings. Overall, our results indicate that modern systems accommodate only a surprisingly small number of messages in flight per node. As Exascale projections indicate that future systems are likely to contain hundreds to thousands of cores per node, we believe that their networks will be underprovisioned. In this situation, proactive congestion avoidance might become mandatory for performance improvement and portability.*

Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: Computer-Communication Networks—*General*; D.4.4 [**Software**]: Operating SystemsCommunications Management[Network Communication]

Keywords

Congestion, Avoidance, Management, High Performance Computing, Manycore, Multicore, InfiniBand, Cray

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.

1. INTRODUCTION

The fundamental premise of this research is that contemporary or future networks for large scale High Performance Computing systems are likely to be underprovisioned with respect to the number of cores or concurrent communication requests per node. In an underprovisioned system, when multiple tasks per node communicate concurrently, it is likely that software or hardware networking resources are exhausted and congestion control mechanisms are activated: the performance of a congested system is usually lower than the performance of a fully utilized yet uncongested system.

In this paper we argue that, to improve performance and portability, HPC runtime implementations need to employ novel node level congestion avoidance mechanisms. We present the design of a proactive congestion avoidance mechanism using a network stateless approach: end-points limit the number of messages in flight using only local knowledge, without global information about the state of the interconnect. The communication load is allowed to reach close to the threshold where congestion might occur, after which it is throttled. Our node level mechanism is orthogonal to the traditional congestion control mechanisms [12, 2] deployed for HPC, which reason in terms of the overall network (switch) load rather than the Network Interface Card load. Our work makes the following contributions:

- We are the first to present evidence that on existing multicore systems maximal network throughput cannot be achieved when all cores are active.
- We propose, implement and evaluate mechanisms to handle node level congestion, ranging from completely distributed control to coordinated control.
- We describe the end-to-end experimental methodology to empirically derive the control algorithms for node level congestion control.

The rest of this paper is organized as follows. In Section 2 we discuss related work and in Section 3 we present our experimental setup. In Section 4 we describe microbenchmarks to understand the variation of network performance with the number of messages in flight, to recognize congestion and to derive heuristics used for message throttling. As shown, techniques to limit the number of in-flight messages per core can improve performance up to 4X. In addition, restricting the number of cores concurrently active per node from 32 to 16 provides additional performance improvements of up to 40% on our InfiniBand testbed. For MPI on Cray Gemini,

restricting the number of active cores per "node" from 48 to 12 provides as much as 6X performance improvement.

We then present several designs for congestion avoidance mechanisms implemented in two Unified Parallel C [23] runtimes on different networks: the Berkeley UPC [9] implementation on InfiniBand and the Cray UPC implementation on Cray XE6 systems with the Gemini interconnect. In Section 5.1 we discuss the design of a message admission control policy using rate or count based metrics. In Section 5.2 we present *inline* and *proxy* based implementations of the admission control policy. With *inline* mechanisms, each task is responsible for managing its own communication requests using either task or node level information. With *proxy* based mechanisms, any communication request can be initiated and managed by any task: intuitively this scheme provides communication servers that perform node-wide communication management on behalf of client pools.

In Sections 6 and 7 we discuss the performance implications of the multiple runtime designs considered. In Sections 8 and 9 we demonstrate how our congestion avoiding runtime is able to provide both performance and performance portability for applications. We transparently improve all-to-all collective performance by 1.7X on InfiniBand system and 4X on Cray Gemini systems, using a single implementation. In contrast, obtaining best performance on each system requires completely different hand tuned implementations. Furthermore, our automatically optimized implementation performs better than highly optimized third party MPI and UPC all-to-all implementations. For example, using 1,024 cores on the InfiniBand testbed, our implementation provides more than twice the bandwidth of the best available library all-to-all implementation for 1,024 byte messages. We also demonstrate 60% performance improvements for the HPCC RandomAccess [3] benchmark. When running the NAS Parallel Benchmarks [15, 1], our runtime improves performance by up to 17%.

As Exascale projections indicate that future systems are likely to contain hundreds to thousands of cores per node, we believe that their networks are likely to be underprovisioned. In this situation, proactive congestion avoidance might become mandatory for performance and performance portability.

2. RELATED WORK

"but I always say one's company, two's a crowd..."

Congestion control in HPC systems has received a fair share of attention and networking, transport or runtime layer techniques to deal with congestion inside high speed networks have been thoroughly explored. Congestion control mechanisms are universally provided at the networking layer. For example, the IB Congestion Control mechanism [2] specified in the InfiniBand Architecture Specification 1.2.1 uses a closed loop reactive system. A switch detecting congestion sets a *Forward Explicit Congestion Notification* (FECN) bit that is preserved until message destination. The destination sends a backward ECN bit to the message source, which will temporarily reduce the injection rate. Dally [12] pioneered the concept of wormhole routing and his work has been since extended with congestion free routing alternatives on a very large variety of network topologies. For example, Zahavi et al [27] recently proposed a fat-tree routing algorithm that provides a congestion-free all-to-all shift pattern for the InfiniBand static routing.

A large body of research proposes algorithmic solutions, rather than runtime approaches. Yang and Wang [25, 26] discussed algorithms for near optimal all-to-all broadcast on meshes and tori. Kumar and Kale [18] discussed algorithms to optimize all-to-all multicast on fat-tree networks. Dvorak et al [13] described techniques for topology aware scheduling of many-to-many collective operations. Kandalla et al [16] discussed topology aware scatter and gather for large scale InfiniBand clusters. Thakur et al [22] discussed the scalability of MPI collectives and described implementations that use multiple algorithms in order to alleviate congestion in data intensive operations such as all-to-all.

A common characteristic of all these approaches is that they target congestion in the network core (or switches): the low level mechanisms use reactive flow control while the "algorithmic" approaches use static communication schedules that avoid route collision. Systems with enough cores per node to cause NIC congestion have been deployed only very recently and we believe that our study is the first to propose solutions to this problem.

The work closest related to ours in the HPC realm has been performed for MPI implementations and mostly on single core, single threaded systems. In 1994 Brewer and Kuszmaul [7] discuss how to improve performance on the CM-5 data network by delaying MPI message sends based on the number of receives posted by other ranks. Chetlur et al [10] propose an active layer extension to MPI to perform dynamic message aggregation on unicores. Pham [20] also discusses MPI message aggregation heuristics on unicore clusters and compares sender and receiver initiated schemes. These techniques use a message aggregation threshold and timeouts with a result equivalent to message rate limitation within a single thread of control. In this paper we advocate for node wide count based message limitation and empirically compare it with rate limitation extended to multi-threaded applications. Furthermore, in MPI information about the system wide state is available to feedback loops (closed control) mechanisms by matching Send and Receive operations. Since in one-sided communication paradigms this type of flow control is not readily available, our techniques use open loop control with heuristics based only on node local knowledge.

2.1 Node Level Proactive Congestion Avoidance

The basic premise of our work is that manycore parallelism breeds congestion and additional techniques for congestion management are required in such clusters. First, the Network Interface Card is likely to be underprovisioned with respect to the number of cores per node and techniques to avoid NIC congestion are required. Second, congestion inside the network proper is likely to become the norm, rather than the exception and congestion control mechanisms are likely to become even harder to implement.

While the former reason is validated throughout this paper, the latter is more subtle. With more cores per node, the likelihood of any node sending messages to multiple nodes at any time is higher, thus making "all-to-all" patterns the norm. These patterns are dynamic, while the whole body of work in algorithmic scheduling [25, 26, 18, 13, 16, 22] addresses only static patterns. Second, the low level congestion control mechanisms [2] already require non-trivial extensions to handle multiple concurrent flows and to deal with runtime

software artifacts such as multiplexing processes, pthreads on multiple endpoints or "interfaces".

In this paper we argue that proactive congestion avoidance mechanisms are required in conjunction with reactive congestion control mechanisms. In a reactive approach, congestion control is activated when resources are exhausted or performance degrades below an acceptable threshold. In a proactive approach, traffic is policed such that, ideally, congestion never occurs. We propose several designs incorporated into a software layer interposed between applications and their runtime. In order to provide scalability, we explore only designs where congestion is managed at endpoints (nodes in the systems), using open loop control without any knowledge of the state of the network core or the system load. All of our implementations are designed to avoid first and foremost congestion at the Network Interface Card, rather than network core congestion. By throttling traffic at endpoints, we also alleviate congestion in the core.

3. EXPERIMENTAL SETUP

We use two large scale HPC systems for our evaluation. Trestles is a 324 compute nodes cluster at the San Diego Supercomputing Center. Each compute node is quad-socket, each with a 8-core 2.4 GHz AMD Magny-Cours processor, for a total of 32 cores per node and 10,368 total cores for the system. The compute nodes are connected via QDR InfiniBand interconnect, fat tree topology, with each link capable of 8 GB/s (bidirectional). Trestles has a theoretical peak performance of 100 TFlop/s.

NERSC's Cray XE6 system, Hopper, has a peak performance of 1.28 Petaflops/sec and 153,216 cores organized into 6,384 compute nodes made up of two twelve-core AMD 'MagnyCours'. Hopper uses the Cray 'Gemini' interconnect for inter-node communication. The network is connected in a mesh topology with adaptive routing. Each network interface handles data for the attached node and relays data for other nodes. The "edges" of the mesh network are connected to each other to form a "3D torus." The Gemini message latency is $\approx 1\mu s$ and two 24 core compute nodes are attached to the same NIC, thus 48 cores share one Gemini card.

All the software described in this paper is implemented as a thin layer interposed between applications and runtimes for the Unified Parallel C (UPC) language. On the InfiniBand network we use the Berkeley UPC runtime [9], version 2.12.2. BUPC is free software and it uses for communication the GASNet [6] layer which provides highly optimized one-sided communication primitives. In particular, on InfiniBand GASNet uses the OpenIB Verbs API. On the Cray system, we use the Cray UPC compiler, version 5.01 within Cray Compiling Environment (CCE) 7.4.2. The Cray UPC runtime is built using the DMAPP[1] layer. We also experiment with MPICH, Cray MPI and OpenMPI.

4. NETWORK PERFORMANCE CHARACTERIZATION

We explore the variation of network performance using a suite of UPC microbenchmarks that vary: i) the number of active cores per node; ii) the number of messages per core; iii) the number of outstanding messages per core; and iv) the message destination. We consider bi-directional traffic, i.e. all cores in all nodes perform communication operations and we report the aggregate bandwidth. We have performed experiments where each core randomly chooses a destination for each message, as well as experiments where each core has only one communication partner. Both settings provide similar performance trends and in the rest of this paper we present results only for the latter.

Figure 1(a) shows the aggregate node bandwidth on InfiniBand when increasing the number of cores. Each core uses blocking communication and we present three runtime configurations ('Proc', 'Hyb' and 'Pth') that are characterized by increasing message injection overheads as first indicated by Blagojevic et al [5]. The series labeled 'Proc' shows results when running one process per core, the series labeled 'Hyb' shows one process per socket with `pthreads` within the socket and the series 'Pth' shows one `process` per node. In general, best communication performance [5] is obtained when threads within a process are mapped on the same socket, rather than spread across multiple sockets. For lack of space, we only summarize the performance trends without detailed explanations.

For all message sizes, the throughput with 'Pth' keeps degrading when adding more sockets. With 'Hyb', the throughput slightly increases up to two sockets active, after which it reaches a steady state. With 'Proc', which has the fastest injection rate, the throughput increases up to two sockets, after which it drops dramatically. In the best configuration, 'Proc' has 3X better throughput than 'Hyb' and 15X better throughput than 'Pth'. The performance difference between best and steady state 'Proc' throughput is roughly 2X. Similar behavior is observed across all message sizes.

These trends are a direct result of congestion in the networking layers. When running `pthreads`, runtimes such as BUPC or MPI use locks to serialize access to the networking hardware: the larger the number of threads, the higher the contention and the higher the message injection overhead. The UPC 'Proc' configuration running with one process per core[2] on InfiniBand, does not use any locks to mitigate the network accesses and the drop in throughput is caused by either low level software (NIC driver) or hardware. Since one process per core provides the best default performance for UPC and it is the default for MPI, the results presented in the rest of this paper are for this particular configuration.

Figure 1(a) indicates that there is a temporal aspect to congestion and throughput drops when too many endpoints inject traffic concurrently. We refer to this as *Concurrency Congestion* (CC) and informally define its threshold measure as the number of concurrent transfers from distinct endpoints (with only one transfer per endpoint) that maximize node bandwidth. For example, any 'Proc' or 'Pth' run with more than 20, respectively four, cores active at the same time is said to exhibit *Concurrency Congestion*. On Cray Gemini, congestion is less pronounced and it occurs when more than 40 of the 48 cores per NIC are active. We believe that ours is the first study to report this phenomenon.

Avoiding CC requires throttling the number of active endpoints and Figure 1(b) shows the performance expectations of a simple optimization approach. Assuming N cores per node, there are N messages to be sent and we plot the speedup when using only subsets of C cores to perform the communication. There are $\frac{N}{C}$ rounds of communication and we

[1]Distributed Memory Application API for Gemini

[2]One thread per process, rather.

plot $\frac{T(N)-T(C)*\frac{N}{C}}{T(N)}$. Positive values on the z axis indicate performance improvements. For example, on InfiniBand using 16 cores to send two stages of 16 messages each is up to 37% faster than allowing each core to send its own message.

Optimized applications use non-blocking communication primitives to hide latency with communication-communication and communication-computation overlap. Figure 2 shows results for a two node experiment where we assume each core has to send a large number of messages. The figure plots the relative differences between communication strategies: N outstanding messages compared with $\frac{N}{D}$ rounds of communication, each with D outstanding messages: $\frac{T(D)*\frac{N}{D}-T(N)}{T(N)}$. In the first setting each core initiates N non-blocking communication requests then waits for completion, while in the second it waits after initiating only D requests.

Intuitively, congestion occurs whenever performing multiple rounds of communication is faster than a schedule that initiates all the messages at the same time. On the InfiniBand system this occurs whenever there is more than one message outstanding per core and best throughput is obtained using blocking communication. Increasing the number of outstanding messages per core decreases throughput, e.g. two outstanding messages per core deliver half the throughput of blocking communication. Asymptotically, the throughput difference is as high as 4X. The Cray Gemini network can accommodate a larger number of messages in flight and there is little difference between one and two outstanding messages per core; with more than two messages per core throughput decreases by as much as 2X. These results indicate that when all cores within the node are active, best throughput is obtained when using blocking communication.

On both systems, reducing the number of active cores determines an increase in the number of outstanding messages per core that provides best throughput, e.g. on InfiniBand with *one* core active, best throughput is observed with 40 outstanding messages.

Intuitively, the behavior with non-blocking communication illustrates the spatial component of congestion, i.e. there are limited resources in the system and throughput drops when space is exhausted in these resources. For the purposes of this study, we refer to this *Rate Congestion*. As with CC, we define the threshold for *Rate Congestion* as the number of outstanding messages per node that maximize throughput. Note than while CC distinguishes between the traffic participants, RC does not impose any restrictions.

The behavior reported for the UPC microbenchmarks is not particular to one-sided communication or caused solely by implementation artifacts of GASNet or DMAPP. Similar behavior is shown in Figure 3 for MPI on both systems. Note the very high speedup (250% on InfiniBand and 700% on Gemini) of MPI throughput when restricting the number of cores: Overall, it appears that MPI implementations exhibit worse congestion than the UPC implementations.

Increasing the number of nodes participating in traffic and varying the message destinations does not change the trends observed using a two node experiment. For lack of space we do not include detailed experimental results but note that on both systems, best throughput when using a large number of nodes is obtained for workloads that have a similar or lower number of outstanding messages per node than the best number required for two nodes. Workloads with small messages tend to be impacted less at scale than workloads containing large messages. In summary, congestion happens first in the Network Interface Card, with secondary effects inside the network when using large messages.

5. CORE STATELESS CONGESTION AVOIDANCE

As illustrated by our empirical evaluation, network throughput drops when tasks initiate too many communication operations. We interpose a proactive congestion avoidance layer between applications and the networking layer that allows traffic to be injected into the network only up to the threshold of congestion.

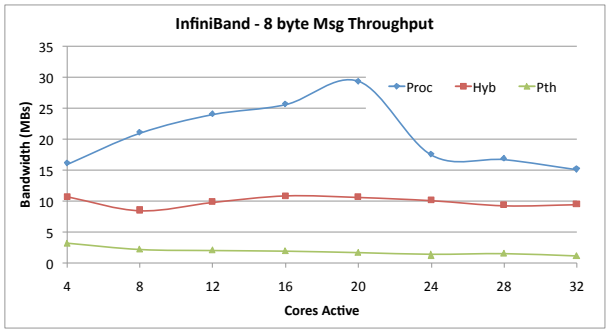
Our implementation redefines the UPC level communication calls and it is transparently deployed for the BUPC and the Cray runtimes. While the UPC language specification allows only for blocking communication, e.g. `upc_memget()`, all existing implementations provide non-blocking communication extensions. Avoiding *Concurrency Congestion* requires instrumenting the blocking calls, while avoiding *Rate Congestion* requires instrumenting the non-blocking calls.

The first component is an admission control policy that determines whether a communication request can be injected into the network. In order to provide scalability, we explore core stateless policies: communication throttling decisions are made using open-loop control[3] with only task or node knowledge and without any information about the state of the outside network. In contrast, previous work [7, 10, 20] tries to correlate MPI Send and Receive events. We derive the congestion thresholds and heuristics to drive the admission control policy from the results reported by the microbenchmark described in Section 4.

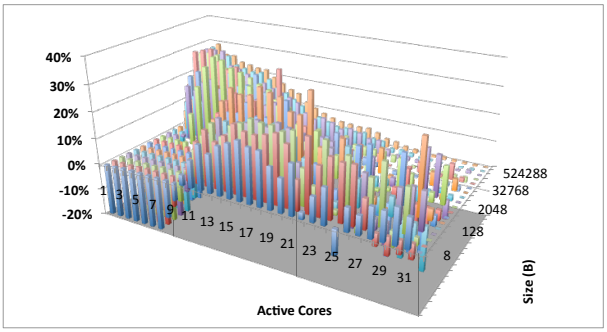
For each communication call, e.g. `upc_memget`, our implementation consults the admission control policy. To address both *Rate* and *Concurrency Congestion* we present a count based policy that uses node level information for message access. Mostly for comparison with delay based techniques (and for the few scenarios where node level network state information is not available to endpoints), we design a rate based policy that allows each endpoint to inject traffic based only on knowledge about its own history. While providing the most scalable runtime design, rate based admission is expected to be able to avoid only *Rate Congestion*.

We have implemented the admission control policy using multiple designs. In the inline design, each task is directly responsible for managing its own communication operations, e.g. the task that initiates a `upc_memget` is also the only one capable of deciding it has completed. Note that this is the functionality implicitly assumed and supported by virtually all contemporary communication layers and runtimes. In the proxy based design, a set of communication "servers" manage client requests. We provide a highly optimized implementation that uses shared memory between tasks (even processes) and allows any task to initiate and retire any communication operation on behalf of any other task. To our knowledge, we are the first to present results using this software architecture within a HPC runtime. The approach is facilitated by GASNet, which provides a communication library for Partitioned Global Address Space Languages. We wish to thank the GASNet developers for providing the modifications required to enable any task to control any communication request.

[3]No monitoring or feedback loop.

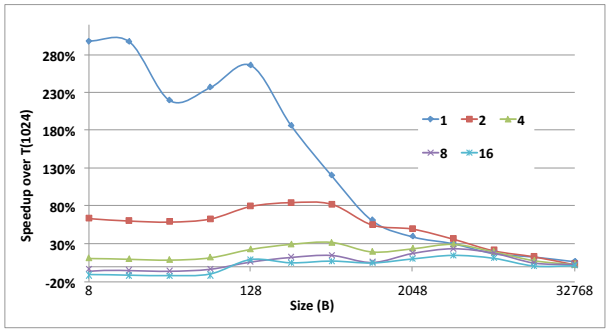


(a) Variation of node throughput with the number of active cores

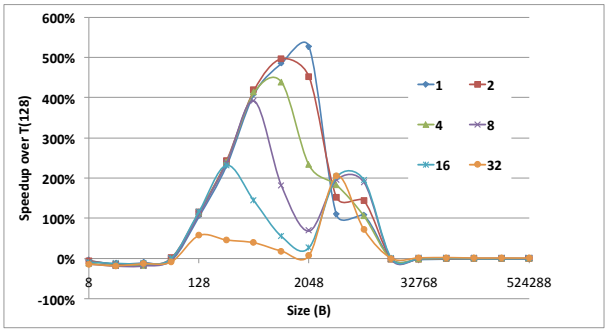


(b) Throughput Improvement when Restricting Active Cores: InfiniBand

Figure 1: *Concurrency Congestion: node throughput drops when multiple cores are active at the same time. We assume each core has one message to send and we plot the speedup of using cores/x rounds of communication over all cores active.*

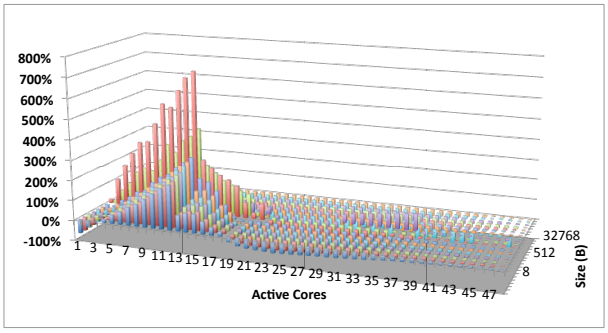


(a) Throughput Variation with Msg/Core - InfiniBand

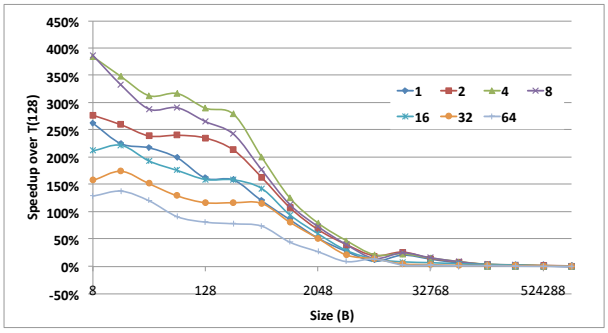


(b) Throughput Variation with Msg/Core - Gemini

Figure 2: *Rate Congestion: node throughput drops when cores have multiple outstanding messages. We assume each core has to send 1,024 messages and we plot the speedup of using 1,024/x rounds of communication with x = (1,2,4..) messages over sending 1,024 messages at the same time.*



(a) Throughput Improvement when Restricting Active Cores: MPI on Gemini



(b) Throughput Variation with Msg/Core - InfiniBand MPI Isend/Irecv with 1,2, 4 ... outstanding messages at a time compared with 128 outstanding messages

Figure 3: *Concurrency and Rate Congestion in MPI. Note that MPI (two-sided) exhibits even larger performance degradation, up to 600%, than UPC (one-sided).*

5.1 Admission Control Policies

Count Limiting: In the count based approach, each node has a predetermined set of tokens. Any task has to acquire enough tokens before being allowed to call into the low level communication API. The number of tokens can either be fixed, *i.e.* one token per request, or can be dynamically specified based on the message size; larger messages are associated with more tokens. Completion of a request involves relinquishing the tokens consumed at posting. Given that we provide a single token pool per node, unfairness is certainly a concern as it reduces the performance of SPMD programs by increasing the synchronization time. To reduce the likelihood of such behavior, we use a ticket-based token allocation that guarantees a first-come first-serve policy. Specifically, threads that are denied network access are given a numbered ticket with the tokens requested. Completion of requests is associated with activation of the next ticket in-line.

We have explored both one token per message and "size proportional" allocations. Achieving optimal performance requires to use for each message a number of tokens proportional to its size. We have implemented a benchmark that performs a guided sweep of different token allocation strategies and synthesizes a total number of tokens per node as well as the number of tokens required by any size. As we explore a multidimensional space $[nodes, cores\_per\_node, msgs\_per\_core, msg\_size]$, this is a very labor intensive process which we plan to automate in future work. The parameters determined by the offline search are then used to specialize the control avoidance code.

Rate Limiting: In the rate limiting approach, tasks are not allowed to call into the low level API for certain periods of time in an attempt to throttle the message injection rate. If the time difference between the last injection time-stamp and the current time is less than a specified threshold, the thread waits spending its time trying to retire previously initiated messages. To determine the best self-throttling injection rates we implemented a benchmark that for each message size sweeps over different values of injection delay using a fine-grained step of 0.5 μs. For each message size we select the delay that maximizes node throughput.

5.2 Admission Control Implementations

Inline Admission Control: We implement two variants of inline enforcement of the admission control policy. The first variant, which is referred to in the rest of this paper as *inline rate throttling* (IRT) uses rate control heuristics based on task local knowledge with synchronous behavior with respect to message injection. IRT is provided mostly for comparison with the related optimization [7, 10, 20] approaches. The second variant, referred to as *inline token throttling* (ITT) uses count limiting heuristics based on node-wide information and it has synchronous behavior with respect to message injection. We did not implement IRT with node knowledge due to the lack of synchronized node clocks.

Proxy Based Admission Control: We implement a proxy based admission control that provides non-synchronous behavior with respect to message injection at the application level. The design is presented in Figure 4. We group tasks in pools and associate a communication server with each task pool. Besides being a client, each application level task can act as a server. Each task has an associated request

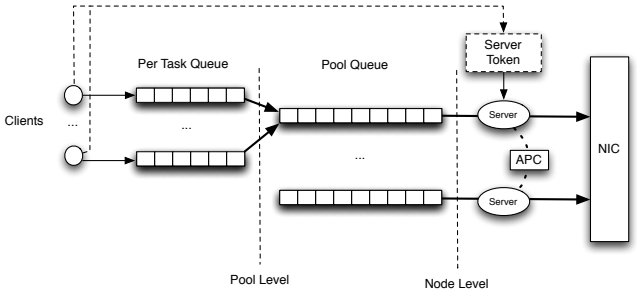


Figure 4: *Software architecture of the Proxy implementation. An admission control policy layer can be easily added behind the servers.*

queue and whenever it wants to perform a communication operation, it will place a descriptor in its "Per Task Queue". Afterwards, the task tries to grab the "Server Token" associated with its pool. If the token is granted, the task starts acting as a server, polls all the queues in the pool and initiates and retires communication operations. When a task masquerades as a server, it serve queues in a round-robin manner, starting with its own. This approximates the default best-effort access to the NIC provided by the underlying software layers. If a token is denied, control is returned to the application and the request is postponed.

Another possibility we have still to experiment with is to have a server issue all the messages within a queue before proceeding to the next queue. This strategy has the effect of minimizing the number of active routes when a task has only one communication partner within a "scheduling" window,

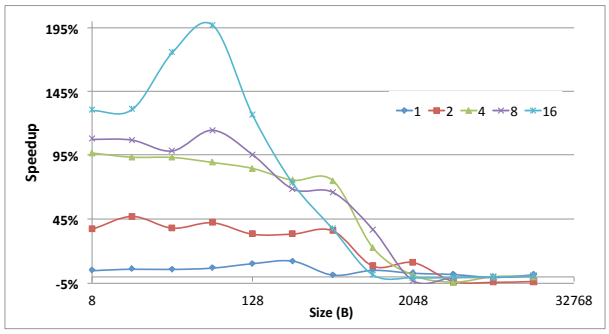
Such a software architecture is able to avoid both types of congestion. In addition, having the requests from multiple tasks aggregated into a single pool increases the opportunity of more aggressive communication optimizations such as message coalescing or reordering. *Concurrency Congestion* is clearly avoided by controlling the number of communication servers. The question remains whether the overhead and throttling introduced by the proxy indirection layer is able to prevent *Rate Congestion* by itself or supplementary avoidance mechanisms are required behind the server layer. *Rate Congestion* avoidance might require controlling the number of requests in flight per server.

In the experimental evaluation section, this implementation is referred to as *Proxy*.

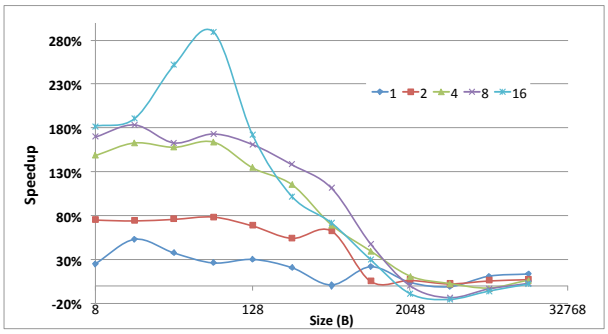
6. MICROBENCHMARK EVALUATION

Figure 5 shows the performance when running the microbenchmark described in Section 4 on top of our congestion avoiding runtime on InfiniBand. We plot the speedup of congestion avoidance over the default runtime behavior. The parameters controlling the behavior of congestion avoidance are obtained using sweeps as described in Sections 5.1 and 5.2.

Figure 5(a) shows the impact of IRT for microbenchmark settings with an increasing number of operations per task, i.e. for the series "2" each task issues two non-blocking operations at a time. As expected, IRT it is not able to avoid *Concurrency Congestion* (series "1"). When tasks issue a larger number of transfers IRT provides a maximum of 2X performance improvements. The largest improvements are observed for messages shorter than 2KB.

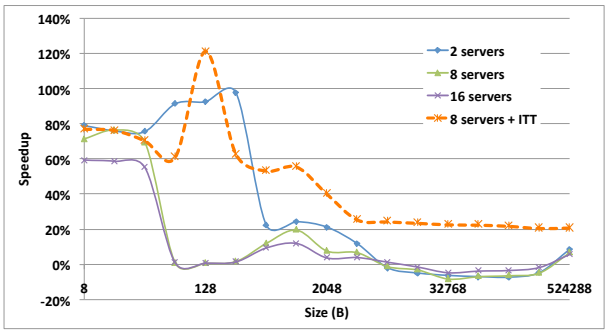


(a) Inline Rate Throttling on InfiniBand



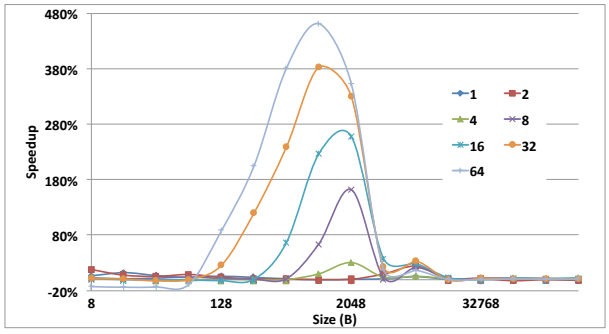
(b) Inline Token Throttling on InfiniBand

(c) Proxy on InfiniBand(1 op per Core)

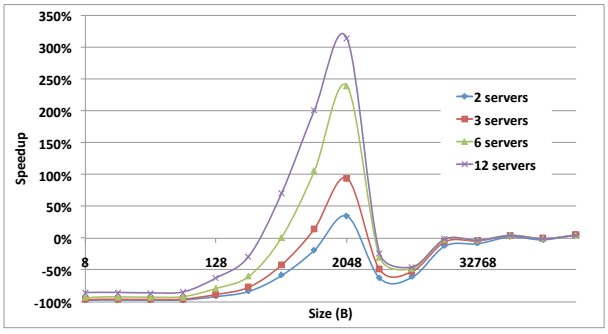


(d) Proxy on InfiniBand(256 op per Core)

Figure 5: *Performance impact of Rate (IRT), Token (ITT) and Proxy congestion avoidance on InfiniBand. We plot the speedup of applying our congestion avoidance while increasing the number of outstanding messages per core (1, 2, 4, 8, 16).*

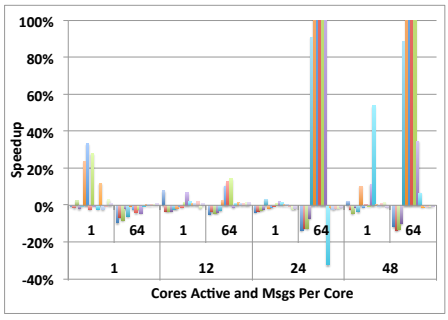


(a) Inline Token Throttling on Gemini

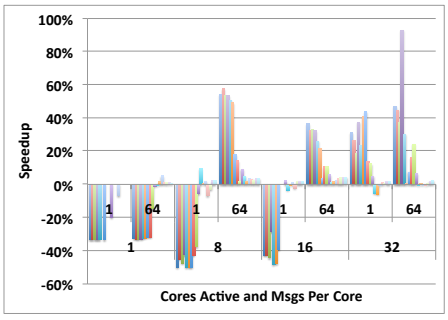


(b) Proxy on Gemini(64 op per Core)

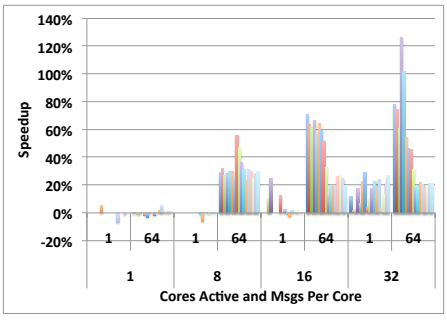
Figure 6: *Performance impact of Token (ITT) and Proxy congestion avoidance on Gemini. We allow an increasing number of outstanding messages per core and plot the speedup of applying our congestion avoidance mechanisms.*



(a) UNOPTIMIZED ITT on Gemini



(b) UNOPTIMIZED Proxy on InfiniBand



(c) Optimized Predictor on InfiniBand

Figure 7: *Overhead of Congestion Avoidance with Active Cores. We plot the speedup for an increasing number of active cores, number of messages per core and message sizes. We vary the message size from 8B to 512KB in increasing powers of 2 and there is one data point per 8, 16, ..., 512KB.*

Figure 5(b) shows that ITT avoids both types of congestion and we observe as much as 3X speedups. Note the 50% speedup obtained when throttling blocking communication.

Figures 5(c) and (d) show the impact of the Proxy implementation for settings where we allow one and 256 outstanding operations per core. We plot the speedup obtained when using two, four, eight and 16 servers per node and observe speedups as high as 1.6X. When the degree of communication concurrency per core is low, e.g. blocking communication, Proxy performs best when the number of active servers (16) is close to the concurrency congestion threshold. When the communication concurrency per core is high (256 outstanding messages), Proxy by itself cannot prevent *Rate Congestion* and best performance is obtained with two servers. The series "8 serves + ITT" shows that implementing an additional admission control layer behind the servers enables the Proxy design to handle both Rate and Concurrency Congestion.

Figure 6 shows the performance on Gemini. ITT is able to improve performance whenever tasks issue more than two outstanding messages and by as much as 5X when issuing 64 outstanding messages. The Proxy implementation improves performance for workloads with at least four outstanding messages per core.

Our approach delays message injection and it might decrease throughput when the communication load is below the congestion thresholds. Figure 7(a) and Figure 7(b) show the impact of **unoptimized ITT** on Gemini and **unoptimized Proxy** on InfiniBand throughput when increasing the number of active sockets per node and the number of messages per core. In this case we are using a predictor independent of the message size, i.e. one token per message. For short messages, ITT on Gemini decreases throughput by at most 10% independent of core concurrency. The data indicates that, although it introduces only a low overhead, it is be beneficial to disable our congestion avoidance mechanism for certain message sizes when only a subset of cores is active.

7. BUILDING A CONGESTION AVOIDANCE POLICER

The microbenchmarks presented throughout the paper indicate that congestion avoidance should be driven by the following control parameters: i) concurrency congestion threshold ($CCT[size]$) measured as the number of cores that when active decrease throughput of blocking communication; ii) node congestion threshold ($NRCT[size]$) measured as the total number of outstanding messages per node that "maximizes" throughput; iii) core congestion threshold ($CRCT[active\_cores][size]$) measured as the number of non-blocking operations per core that "maximize" throughput at a given core concurrency. Intuitively, these parameters capture the minimal amount of communication parallelism required to saturate the network interface card.

Algorithm 1 shows the pseudo code for the control decisions in our mechanism. We give priority to dealing with *Concurrency Congestion* and then we try to avoid *Rate Congestion*. The "procedures" avoid_* use internally a count based predictor for both ITT and the Proxy implementation. For Proxy we add an admission control layer behind the servers. All parameters, including the tokens per node, are determined by iteratively executing the microbenchmarks

Algorithm 1 Pseudo code for congestion avoidance.

```
1: procedure MSG_INIT( size : In, dest : In)
2:     if active_cores < CCT[size] then
3:                                   ▷ no concurrency congestion
4:         if active_node_msgs < NRCT[size] AND
   active_msg < CRCT[active_cores][size] then
5:             inject(size, dest)                ▷ no congestion
6:         else
7:             avoid_RC(size,dest)
8:                                   ▷ rate congestion detected
9:         end if
10:    else
11:        avoid_CC_RC(size,dest)
12:                ▷ concurrency and rate congestion detected
13:    end if
14:
15: end procedure
```

using manual guidance. At this point, the predictor we synthesize in practice contains thresholds that are independent of the message size, i.e. one token per message. For Proxy, we also search for the best server configuration. Our results on the InfiniBand cluster indicate that eight servers per node produce good results in practice, while on Gemini, 24 servers per node are required. This amounts to a ratio of four, respectively two tasks per server.

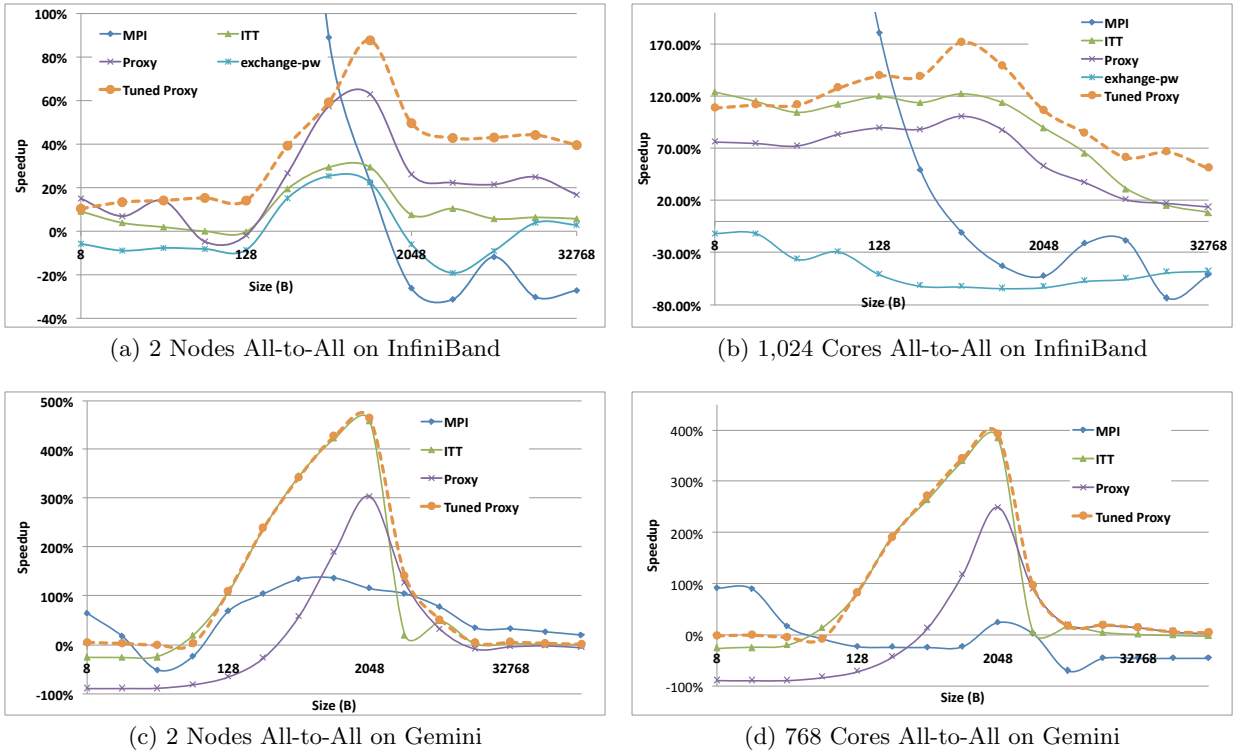
The behavior on InfiniBand using the tuned predictors is shown in Figures 7(c). When varying the number of active cores, messages per core and size of the message, our implementation improves performance in most of the cases. In very few cases it introduces a small overhead, at most 4%. We are still investigating the behavior of ITT when using eight cores with blocking communication. Similar trends are observed on Gemini. When comparing with Figure 7(a) and 7(b) which show unoptimized ITT and Proxy performance, tuning reduces drastically the number of configurations where performance is lost. For the configurations where our implementation actually slows down the microbenchmark execution, the predictor causes an average slowdown of 2% across varying core concurrencies, message sizes and messages per core.

As we have only partially processed a large volume of experimental data, we believe that we can further tune the predictors and improve the performance of our mechanisms.

8. ALL-TO-ALL PERFORMANCE

All-to-all communication is widely used in applications such as CPMD [11], NAMD [21], LU factorization and FFT. MPI [22, 17] and parallel programming languages such as UPC [19] provide optimized implementations of all-to-all collective operations. Most if not all of the existing implementations use multiple algorithms selected by message size. Bruck's algorithm [8] is used for latency hiding for small messages and it completes in $log(P)$ steps, where P is the number of participating tasks. For medium message sizes an implementation overlapping [22] all the communication operations is used. In this implementation, tasks use non-blocking communication and initiate $P-1$ messages. For large message sizes, a pairwise exchange [22] algorithm is used where pairs of processors "exchange" data using blocking communication.

To demonstrate the benefits of our congestion avoidance runtime, we compare the performance of a single algorithm all-to-all against the performance of library implementations.



(a) 2 Nodes All-to-All on InfiniBand

(b) 1,024 Cores All-to-All on InfiniBand

(c) 2 Nodes All-to-All on Gemini

(d) 768 Cores All-to-All on Gemini

Figure 8: *Impact of IRT, ITT and Proxy congestion avoidance on all-to-all performance. We plot the speedup of MPI and that of single implementation running on top of congestion avoidance (IRT, ITT,Proxy). The performance baseline is a overlapped implementation in UPC.*

Our baseline implementation is the overlapping "algorithm" with each processor starting to communicate with $MYTHREAD+1$. In Figure 8 we plot the speedup of multiple implementations over the baseline implementation running on the native UPC runtime layers. For reference, the series labeled MPI presents the performance of the MPI_Alltoall on the respective system. The performance of the UPC library all-to-all is similar to MPI and not shown. On both systems the library calls implement Bruck's algorithm for small messages. The series labeled "exchange-pw" presents the performance of a handwritten pairwise exchange implementation in UPC. We have also implemented pairwise exchanges in MPI, the results are similar to exchange-pw and omitted for brevity. The series labeled "ITT" and "Proxy" show the performance of the overlapping algorithm with a runtime that implements ITT and Proxy congestion avoidance respectively. These implementations are not tuned and use a simple count based predictor enabled for all core concurrencies and message sizes. The series labeled "Tuned Proxy" shows the behavior of a tuned implementation of Proxy and it illustrates the additional benefits after a significant effort to mine the experimental data.

On the InfiniBand network, "Tuned Proxy" provides best performance and we observe speedups as high as 90% and 170% for 512 byte messages on two nodes and 1024 cores respectively. Furthermore, our implementation is faster than any all-to-all deployed on the system for medium to large messages. For example, "Tuned Proxy" is roughly 5X faster than the MPI library at 1KB messages. We omit any rate throttling results (IRT) since IRT provides only modest performance improvements.

On Gemini, our congestion avoiding runtime provides again the best performance. The MPI library is not as well tuned on Gemini and our implementation is as much as 6X faster than MPI for medium sized messages. ITT provides better performance than Proxy and IRT provides the least improvements. Except for small messages where MPI uses Bruck's algorithm, ITT is faster than any communication library deployed on the Cray, by as much as 25% for 2KB messages when using 768 cores.

These results indicate that our congestion avoiding runtime is able to improve performance and provide performance portability. We have obtained best performance on two systems using one implementation when compared against multi-algorithm library implementations.

9. APPLICATION BENCHMARKS

We evaluate the impact of our congestion avoiding runtime on several application benchmarks written by outside researchers. The HPCC RandomAccess benchmark [3] uses fine grained communication, while the NAS Parallel Benchmarks [1] are optimized to use large messages. Fine-grained communication is usually present in larger applications during data structure initializations, dynamic load balancing, or remote event signaling.

The current UPC language specification does not provide non-blocking communication primitives and all publicly available benchmarks use blocking communication. On the other hand, both BUPC and Cray provide nonblocking extensions. We have modified each benchmark implementation to exploit as much communication overlap as possible. All the performance models and heuristics described in this

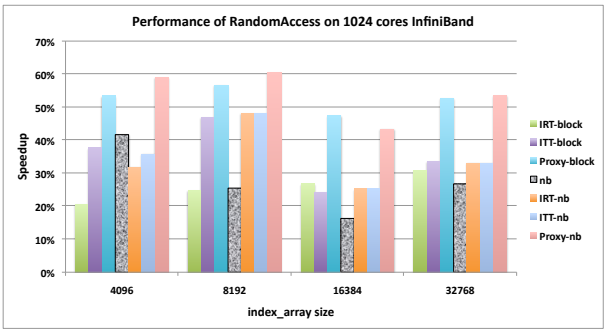


Figure 9: *RandomAccess on 1,024 cores InfiniBand. We plot the speedup relative to a baseline implementation using blocking communication.*

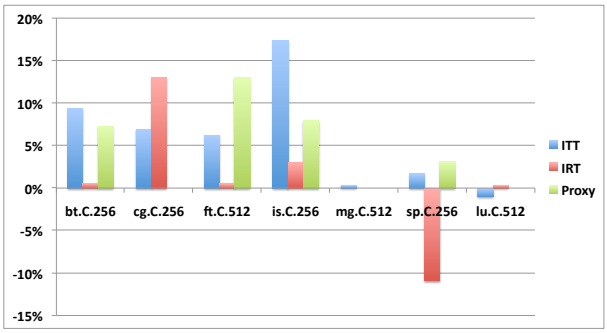


Figure 10: *The NAS Parallel Benchmarks on InfiniBand. We plot the speedup relative to a baseline implementation using blocking communication.*

paper have been implemented in a thin layer between the application and the runtimes for Berkeley UPC and Cray UPC, which is transparent to the application developer.

RandomAccess: The RandomAccess benchmark is motivated by a growing gap in performance between processor operations and random memory accesses. This benchmark intends to measure the peak capacity of the memory subsystem while performing random updates to the system memory. The benchmark performs random read/modify/write accesses to a large distributed array, a common operation in parallel hash table construction or distributed in-memory databases. The amount of work is static and evenly distributed among threads at execution time. Figure 9 presents the results on InfiniBand when using 1,024 cores. We plot the speedup relative to a baseline implementation that uses only blocking communication primitives. The x-axis plots the number of indirect references per thread. The message size for every single operation is 16 byte. The first three bars (IRT-block, ITT-block, Proxy-block) plot the speedup observed when running the baseline implementation with congestion avoidance and illustrate the capability of our runtime to avoid *Concurrency Congestion*. Proxy is able to provide speedup as high as 57%. The series labeled "nb" plots the performance of a hand optimized implementation in which the inner loops are unrolled and communication is pipelined and overlapped with computation and other communication. This is the de facto communication optimization strategy that is able to improve performance by as much as 40%. The series IRT-nb, ITT-nb and Proxy-nb show the additional performance improvements of congestion avoidance for *Rate Congestion* and Proxy-nb is able to provide as much as 60% speedup. As indicated by Figure 2, the small messages in RandomAccess do not generate congestion on Gemini and our runtime does not affect its performance.

The behavior of RandomAccess illustrates an interesting performance inversion phenomenon: an implementation with blocking communication and congestion avoidance is able to attain better performance than an implementation hand optimized for communication overlap. Best performance is obtained by the implementation optimized for overlap and using congestion avoidance.

NAS Benchmarks: All implementations are based on the official UPC [1, 15] releases of the NAS benchmarks, which we use as a performance baseline. For brevity we do not provide more details about the NAS Parallel Benchmarks, for a detailed description please see [4, 14]. The benchmarks exhibit different characteristics. FT and IS perform all-to-all communication. SP and BT use scatter-gather communication patterns. SP issues requests (Put) to transfer a variable number of mid-size contiguous regions. The requests in BT (Put and Get) vary from small to medium sizes. In MG, the communication granularity varies dynamically at each call site. CG uses point to point communication with constant message sizes. For all benchmarks, the count and granularity of messages varies with problem class and system size. Vetter and Mueller [24] indicate that large scientific applications show a significant amount of small to mid-size transfers and all the benchmark instances considered in this paper exhibit this characteristic.

Figure 10 presents the results on InfiniBand. As discussed, the implementations that use `upc_memput` show no performance improvement. Best performance improvements are observed for the communication intensive benchmarks (CG, FT, IS) and we observe as much as 17% speedup for IS.

10. DISCUSSION

Most of the previous work [2, 12, 27, 25, 26, 18, 13, 16] addresses congestion in the core (switches) of HPC networks. As our experimental evaluation shows, the advent of multicore processors introduces congestion at the edge of these networks and mechanisms to handle *Concurrency Congestion* are required for best performance on contemporary hardware. Our count based heuristic can handle both *Rate* and *Concurrency Congestion* and it has been easily incorporated into software architectures using either task level (ITT) or node level (Proxy) mechanisms. While ITT is simple to implement and provides good performance, we favor in the long run the Proxy with ITT design which allows for further optimizations such as coalescing and reordering of communication operations. We also believe that the admission control policy heuristics can be further improved.

All the experimental results illustrate the challenges of writing performance portable code in a multi-system, hybrid-programming model environment and we have shown that our congestion avoiding runtime provides both performance and performance portability. The current optimization dogma advocates for exposing a large concurrency and hiding latency (with multi-threading or other optimizations) by overlapping communication with other work. Our experiments indicate that each system supports only a very limited amount of communication concurrency without significant performance degradation. Our techniques allow developers to ex-

pose the maximum "logical" concurrency at the application level and throttle it at runtime for optimal performance. Also, note that without congestion avoidance, our evaluation indicates that overlap is becoming harder to achieve with portability on manycore systems.

Examining the design tradeoffs of congestion avoidance mechanisms, and in general application optimization tradeoffs, we see two main design criteria: 1) optimizing for overlap; and 2) optimizing for throughput. As overlap requires fast message injection and throughput requires throttling and delays, these two have contradictory requirements. The status-quo in runtime and optimizations design favors overlap and fast injection. For the systems examined in this paper we observe a performance inversion between injection speed and throughput: the networking layers allowing the fastest injection rate observe the highest throughput degradation. We have re-implemented all of our microbenchmarks using the vendor APIs OpenIB Verbs and DMAPP. While calling the native API provides the fastest injection rate, those benchmarks achieve lower throughput than either GASNet, UPC or MPI. The detailed results are omitted for brevity. We believe that increasing the number of cores per node will require a shift towards optimizations for throughput using new approaches and performance metrics. Our congestion avoiding runtime samples points in the space of throughput oriented designs and we believe the Proxy design can provide both fast injection/overlap and throughput.

The microbenchmark results in Section 4 indicate that congestion is observable independently of the implementation, i.e. GASNet, MPI or native APIs, or the communication paradigm, i.e. one-sided in GASNet and two-sided in MPI. On the InfiniBand system we have experimented with multiple NIC resource knobs controlled by software: the settings used in this study provide the best default performance. The MPI implementations (Cray MPI, MPICH, OpenMPI) seem to be affected even more than the one-sided runtimes (GASNet and Cray UPC). Thus, deploying similar mechanisms into MPI implementations is certainly worth pursuing. The implicit flow control provided by MPI Send and Receive operations allows for extensions using closed loop control techniques.

Our congestion avoidance mechanisms implement a core stateless approach where decisions are made at the edge of the network (nodes), without global state information about actual congestion in the network core (switches). The results indicate that we can provide good performance at scale, but the question remains how close to optimal we can get and whether mechanisms using global state can do better. While we do not have conclusive evidence, our conjecture is that addressing congestion at the edge of the network is likely to provide similar or better performance than global state mechanisms at scale. Another question is that of fairness when not all the nodes in the system use congestion avoiding runtimes. Our experiments were run on capacity systems and the benchmarks were competing directly against applications using unmodified runtimes. This indicates that a congestion avoiding runtime competes well with greedy traffic participants.

This work also raises the question whether the runtime can displace the algorithm. Previous work proposes application algorithmic changes that affect the communication schedule to reduce the chance of route collision. Our implementation throttles communication operations and implicitly reduces the chance of collisions. Furthermore, Proxy can be extended with node-wide message reordering and coalescing optimizations and mechanisms to avoid route collision can be provided at that level. Understanding the tradeoffs between these alternatives is certainly important and is the subject of future work.

Finally, we believe that our open loop runtime congestion avoidance mechanisms are orthogonal to the vendor provided closed loop congestion control mechanisms. In all of our experiments the vendor congestion control mechanisms (e.g. IB CCA) were enabled. However, the question remains if there are any undesired interactions between the two mechanisms.

11. CONCLUSION

Efficient communication is required for application scalability on contemporary High Performance Computing systems. One of the most commonly employed and advocated optimization techniques is to hide latency by overlapping communication with computation or other communication operations. This requires exposing a large degree of communication concurrency within applications.

In this paper, we show that contemporary networks or runtime layers are not very well equipped to deal with a large number of operations in flight and suffer from congestion. We distinguish two types of congestion: *Rate Congestion* happens when tasks inject too many concurrent messages, while *Concurrency Congestion* happens when too many cores are active at the same time. We propose a runtime design using proactive congestion avoidance techniques: a thin software layer is interposed between the application and the runtime to limit the number of concurrent operations. This approach allows the communication load to increase to the point where native congestion control mechanisms might have been triggered, without actually triggering them.

We implement a congestion avoiding runtime for one-sided communication on top of two UPC runtimes for two networks: InfiniBand and Cray Gemini. We discuss heuristics to limit the number of messages in flight and present implementations using either task inline or server based mechanisms. Our runtime is able to provide performance and performance portability for all-to-all collectives (2x improvements), fine grained application benchmarks (60% improvements), as well as implementations of the NAS Parallel Benchmarks (up to 17% improvements).

As Exascale projections indicate that future systems are likely to contain hundreds to thousands of cores per node, we believe that their networks are likely to be underprovisioned and applications are likely to suffer from both *Rate Congestion* and *Concurrency Congestion*. In this situation, proactive congestion avoidance might become mandatory for performance and performance portability.

12. REFERENCES

[1] The GWU NAS Benchmarks. http://threads.hpcl.gwu.edu/sites/npb-upc.

[2] The InfiniBand Specification. Available at http://www.infinibandta.org.

[3] V. Aggarwal, Y. Sabharwal, R. Garg, and P. Heidelberger. HPCC Randomaccess Benchmark For Next Generation Supercomputers. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick. Hybrid PGAS Runtime Support for Multicore Nodes. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, 2010.

[6] D. Bonachea. GASNet Specification, v1.1. Technical Report CSD-02-1207, University of California at Berkeley, October 2002.

[7] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *IPPS'94*, pages 858–867, 1994.

[8] J. Bruck, S. Member, C. tien Ho, S. Kipnis, E. Upfal, S. Member, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 298–309, 1997.

[9] Berkeley UPC. Available at http://upc.lbl.gov/.

[10] M. Chetlur, G. D. Sharma, N. B. Abu-Ghazaleh, U. K. V. Rajasekaran, and P. A. Wilsey. An Active Layer Extension to MPI. In *PVM/MPI*, 1998.

[11] Available at http://www.cpmd.org/.

[12] W. J. Dally and C. L. Seitz. The Torus Routing Chip. *Distributed Computing*, pages 187–196, 1986.

[13] V. Dvorak, J. Jaros, and M. Ohlidal. Optimum Topology-Aware Scheduling of Many-to-Many Collective Communications. *International Conference on Networking*, 0:61, 2007.

[14] A. Faraj and X. Yuan. Communication Characteristics in the NAS Parallel Benchmarks. In *14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, November 2002.

[15] H. Jin, R. Hood, and P. Mehrotra. A Practical Study of UPC with the NAS Parallel Benchmarks. *The 3rd Conference on Partitioned Global Address Space (PGAS) Programming Models*, 2009.

[16] K. C. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda. Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case studies with Scatter and Gather. In *IPDPS Workshops'10*, pages 1–8, 2010.

[17] R. Kumar, A. Mamidala, and D. K. Panda. Scaling alltoall Collective on Multi-Core Systems. *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[18] S. Kumar and L. V. KalÃl'. Scaling All-to-All Multicast on Fat-tree Networks. In *ICPADS'04*, pages 205–214, 2004.

[19] R. Nishtala, Y. Zheng, P. Hargrove, and K. A. Yelick. Tuning Collective Communication for Partitioned Global Address Space Programming Models. *Parallel Computing*, 37(9):576–591, 2011.

[20] C. D. Pham. Comparison of Message Aggregation Strategies for Parallel Simulations on a High Performance Cluster. In *In Proceedings Of The 8th International Symposium On Modeling, Analysis And Simulation Of Computer And Telecommunication Systems, August-September*, 2000.

[21] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular Simulation on Thousands of Processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[22] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *IJHPCA*, pages 49–66, 2005.

[23] UPC Language Specification, Version 1.0. Available at http://upc.gwu.edu.

[24] J. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. *Proceedings of the 2002 International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[25] Y. Yang and J. Wang. Efficient All-to-All Broadcast in All-Port Mesh and Torus Networks. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 290–, Washington, DC, USA, 1999. IEEE Computer Society.

[26] Y. Yang and J. Wang. Near-Optimal All-to-All Broadcast in Multidimensional All-Port Meshes and Tori. *IEEE Trans. Parallel Distrib. Syst.*, 13:128–141, February 2002.

[27] E. Zahavi, G. Johnson, D. J. Kerbyson, and M. L. 0003. Optimized InfiniBand^{TM} Fat-Tree Routing For Shift All-To-All Communication Patterns. *Concurrency and Computation: Practice and Experience*, 22(2):217–231, 2010.