Optimizing All-to-All Collective Communication by Exploiting Concurrency in Modern Networks

Vinod Tipparaju
Pacific Northwest National Laboratory
Richland, WA
vinod@pnl.gov

Jarek Nieplocha
Pacific Northwest National Laboratory
Richland, WA
jarek.nieplocha@pnl.gov

ABSTRACT

The paper proposes a novel approach for optimizing performance of all-to-all collective communication by taking advantage of concurrency available in modern networks such as Infiniband or Quadrics. Using the MPI AllGather operation as an example, we describe how network concurrency can be exploited in an optimized implementation of this operation. For example, compared to leading MPI implementations for a 32-KB message on 128 processors, our new algorithm yields a 65% improvement on the Infiniband at Virginia Tech and an 89% improvement on the Quadrics cluster at Pacific Northwest National Laboratory.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed programming, parallel programming.

General Terms

Algorithms, Theory, Verification, Design and Performance.

Keywords

Collective communication, Network concurrency, All-to-All, MPI_AllGather.

1. INTRODUCTION

Advances in commodity microprocessors and the advent of affordable high-performance gigabyte networks have made clusters a clear choice and widely used alternative to massively parallel processors (MPPs). High-performance networks have developed significantly over the past several years. Many of these networks are capable of remote direct memory access (RDMA) communication that involves movement of data between processor memories without memory copies or remote host processor involvement. Current high-speed networks deploy powerful communication processor(s) in the network interface card (NIC) to handle processing of multiple incoming and outgoing messages without interrupting the host processor. For example, the Quadrics Elan4 network interconnect can do two overlapping DMAs and allows multiple outstanding read/write transactions from and to the network interconnect. Switches used in the modern interconnects also have developed significantly in the last several years, facilitating multiple communication paths between network endpoints and hence roviding increased levels of concurrency and redundancy communication between network endpoints [3]. As a result, the modern networks are very capable of handling simultaneous and concurrent data movements. These developments lead to a question: can the collective communication algorithms we have been using on these networks take full advantage of the concurrency in the network? We believe that some implementations of collectives based on multicast are steps in the right direction[1] but there also are other opportunities for performance improvements based on network concurrency. This paper explores one of them.

This paper discusses how all-to-all collective communication algorithms can be envisioned to take more advantage at the *algorithmic level* of the concurrency in several high-performance networks. We use the All-Gather collective communication operation as an example of such algorithms. This operation is one of the most "network intensive" collective communication operations known. It requires every process to receive data from every other process at the end of the operation. All-gather is a popular collective communication routine included in the MPI standard and provided in many collective communications libraries [4-6] and various implementations of MPI [7, 8]. For example, it is useful in applications that rely on replicated data approaches when individual processors update a part of a data structure and then exchange it globally with the others.

Our approach uses multiple asynchronous RDMA operations to better exploit the network concurrency. We use the term *concurrency index* to characterize the number of outstanding concurrent messages and how it can be used to determine the all-gather pattern. The paper describes the new baseline algorithm and then its modification for more efficient handling of larger messages. The experimental results collected on two platforms, one with Infiniband and the other with Quadrics Qsnet-II, are discussed.In addition to the experimental evaluation reported in the current paper, our approach already has been deployed successfully in the context of real-world applications. In particular, it has been used in a computational chemistry simulation code for updating replicated data structures as a part of the iterative simulation procedure for computing polarizable water potentials [9] and for development of a new implementation of the Numerical Aerodynamic Simulation (NAS) Conjugate Gradient benchmark [10].

[1] Techniques that use multicast to implement collective communication are network-specific and are applicable for networks such as Quadrics without native multicast interfaces.

(c) 2005 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SC|05 November 12-18, 2005, Seattle, Washington, USA

(c) 2005 ACM 1-59593-061-2/05/0011…$5.00

2. BACKGROUND AND RELATED WORK

Multiple programming environments and communication libraries for parallel computing rely on collective communications [7, 11-14]. Collective communication is significant for many applications and attracts extensive attention regarding their efficient implementation on modern networks. These efforts to accelerate collective operations at the implementation level can be categorized as 1) based on point-to-point operations (we called them "traditional"), 2) based on multicast, and 3) other advanced NIC capabilities.

Over the last several years, many methods have been described for implementing collective communication algorithms. Among the more common of these algorithms are binary/binomial and other tree-based algorithms and the pairwise exchange-based algorithms [5, 15, 16]. Many topology-specific collective communication algorithms, including the ones to AllGather also have been discussed. Papers [17, 18] focus on SMP-based cluster hierarchy to implement collective communication operations. Paper [16] advocates automatically tuned collectives based on network topology. The primary focus in these efforts is on other aspects of the implementation rather than exploitation the network concurrency at the algorithmic level.

A few papers have used network concurrency to optimize collective communication. Work by Susumu et al. [19] uses asynchronous communication and parallelism for implementing the gather operation. Although this method exploits concurrency, the communication time still is modeled to be on the order of log(n), where n is the number of processes. Work by Mamidala [22] et al. talks about adaptive methods to accomplish AllReduce by sending/receiving messages farther down the communication tree and hence utilizing network concurrency.

IRecently, the multicast capabilities network has been used for all-to-all communication primarily by Dr. Panda's group at Ohio State University. Paper [20] describes the NIC-based AllGather algorithm in which every NIC does a concurrent Gather. This method involves n*log(n) messages. Paper [21] talks about using multicast for All-Reduce (has much less data transmission than All-Gather and involves computation). NIC-based methods have been used for other collective operations [22] [23]. These efforts use NIC and multicast to implement traditional collective algorithms efficiently. However, they differ to a great degree from our approach, which is implemented without relying on these capabilities and is based on multiple asynchronous RDMA messages and therefore applicable to networks without multicast or with very limited or impractical NIC-level programmability.

2.1 Network concurrency in current networks

Current trends on high-speed interconnects include the availability of a communication processor in the NIC, which allows the implementation of high-level messaging libraries without explicit intervention of the main central processing unit (CPU). The Myricom Myrinet network [24], Infiniband [17], and Quadrics QsNet [3] are a few examples of networks that offer RDMA capabilities with the assistance of the processors they have available on the NIC. These networks also offer a good amount of concurrency in data transmission and offer excellent bidirectional bandwidth. We focus on the Mellanox Infiniband Interconnect and Quadrics QsNetII network for experimental evaluation of highly parallel All-Gather algorithms proposed in this paper.

QsNetII is the latest generation of the Quadrics interconnect. It consists of two application-specific integrated circuits (ASICs)—Elan4 (Figure 1) and Elite4. The Elan4 communication processor forms the interface between a high-performance multistage network and a processing node containing one or more CPUs. Elite4 switch components are capable of switching eight bi-directional communications links. Each communications link carries data in both directions simultaneously at 1.3 GBytes/s. The link bandwidth is shared between two virtual channels. It uses wormhole switching with two virtual channels per physical link, source-based routing and adaptive routing. The network has support for zero-copy RDMA transactions and the hardware support for collective communication [25]. The functional units of

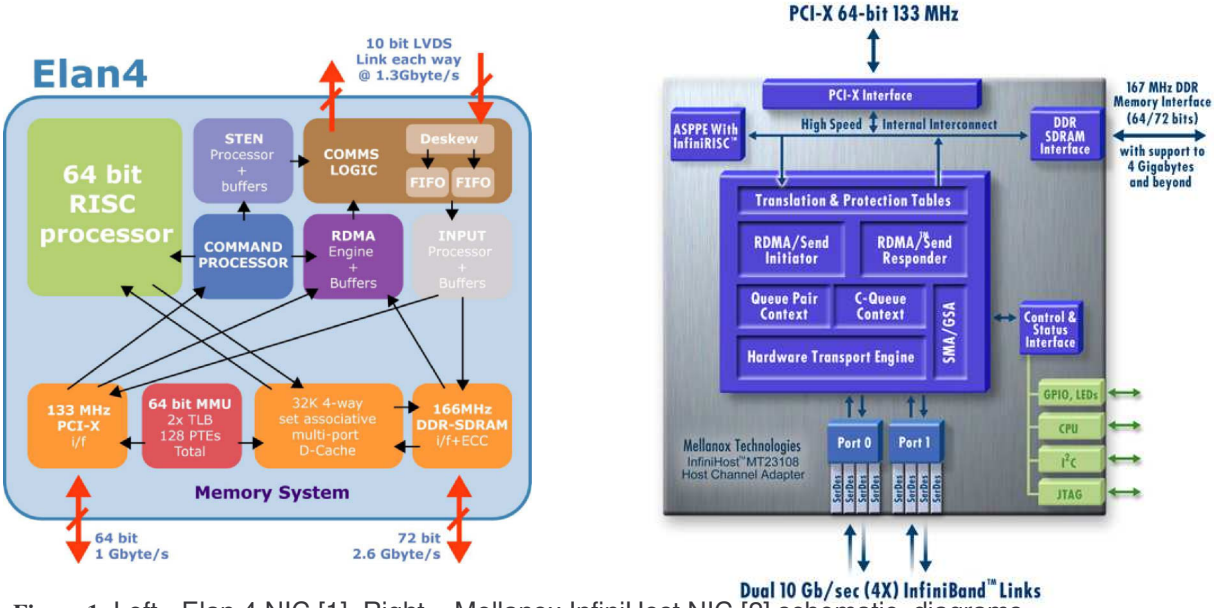

Figure 1: Left - Elan 4 NIC [1], Right – Mellanox InfiniHost NIC [2] schematic diagrams

the Elan4 are interconnected using multiple independent 64-bit data buses. The use of separate paths increases concurrency and reduces data transfer latency. The user processes can perform remote read/write memory operations by issuing DMA commands to the Elan4. The DMA engine processes up to two DMAs to overlap the start-up/finish latency and maintain full PCI-X read bandwidth.

Infiniband architecture (IBA) defines a switched communications fabric, allowing many devices to communicate concurrently with high bandwidth and low latency in a protected, remotely managed environment. A node can employ multiple paths through the IBA fabric. IBA hardware off-loads from the CPU much of the I/O communications operation. This allows multiple concurrent communications without the traditional overhead associated with communicating protocols. In the current work, we used the Mellanox Cougar HCA cards.

3. ALGORITHMS AND THEORY

For all-to-all communication, pairwise and butterfly algorithms are known to be a better choice because their complexity is on the order of log(n) (logarithm of n with base 2) where n is the number of processors. The Argonne National Laboratory MPICH [7] implementation uses a pairwise exchange algorithm. It should be noted that the times modeled in the following equations under the assumption that the switch bisection bandwidth is sufficiently large so that it can be ignored from the analysis.

In the equations, the symbol o represents a constant overhead per step. Time taken by traditional send/receive-based pairwise exchange algorithms can be represented by Equation (1).

$$T_1 = o\sum_{i=0}^{(\log n)-1} t(m \cdot 2^i) \qquad (1)$$

The symbol $t(m)$ in Equation (1) stands for time taken to prepare, initiate and transmit a message of size m. This is a log(n) step exchange, and in step i a message of size $m*2^i$ is exchanged. By using blocking messages, time for the exchange is twice the time it takes to transfer a unidirectional message of size $m*2^i$. For log(n) steps, this is what is shown as T_1 in Equation (1).

Traditional message passing-based pairwise algorithms tend to use unidirectional network bandwidth. Pairwise algorithms that use the non-blocking point-to-point communication can in principle better utilize the bidirectional bandwidth and hence have a potential to give better performance than algorithms based on blocking messages. Time taken by these algorithms can be approximated as shown in Equation (2). The time per step is a factor of the bidirectional bandwidth. So, for a step i, the time taken can be represented by $T(m \cdot 2^i)$, $T(m)$ is the time taken to prepare, initiate, and transmit a message of size m when communication is bidirectional. Hence, the T_2 in Equation (2)

$$T_2 = o\sum_{i=0}^{(\log n)-1} T(m \cdot 2^i) \qquad (2)$$

represents the sum over log(n) terms for each of these steps. It should be noted that $t(m)$,which was introduced earlier, represents time similar to that of $T(m)$ but for a unidirectional message.

However using the bidirectional bandwidth is just the first step in the attempt to use available concurrency. As discussed in Section 2, many current-generation NICs and switches are capable of concurrent processing of messages. But this feature offered by many networks is rarely considered while designing collective communication algorithms. Here we describe two methods that use network concurrency to reduce the time taken by traditional pairwise collective communication algorithms, particularly All-Gather.

We introduce a term we call *concurrency index (C)*, which indicates how many concurrent exchanges the algorithms described below perform. Figure 2 shows two ways of implementing concurrent exchange on 16 processors. The traditional pairwise All-Gather takes 4 steps to complete the All-Gather operation such that every processor has every other processors data. In Figure 2, the encircled regions show the exchange occurring in that group. The dotted lines connecting the encircled regions show the nonblocking messages sent during each step of the exchange with in each group. Steps are indicated with numbers. Lines connecting groups show messages exchanged in a nonblocking manner by every process in each group, between the groups, for each step of the exchange with in the group. The left side of Figure 2 shows the 16 processes divided into two groups. This is a good way to do concurrent

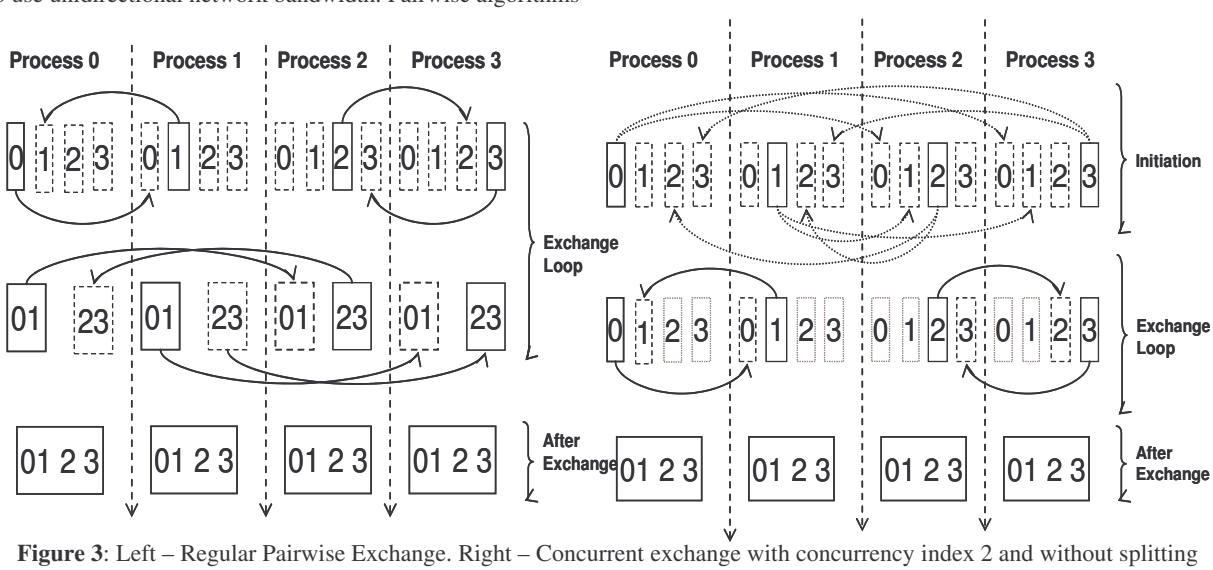

Figure 3: Left – Regular Pairwise Exchange. Right – Concurrent exchange with concurrency index 2 and without splitting

1-59593-061-2/05 $20.00 © 2005 IEEE

exchange if the network supports algorithms using two concurrent messages. Each group, in each step of the exchange, does an exchange with its *pair* process. In addition, it sends a nonblocking message to a corresponding *pair* process in the other group. Steps are indicated with numbers. For example, in step 1, Process 0 sends a message to Process 1. It also initiates a nonblocking message to the peer process of Process 1 in the other group, which is Process 9. On the right side, the 16 processes are divided into four groups. This is ideal if the network is able to support algorithms with four concurrent messages. Here, each process, in each step of its exchange, exchanges data with a *peer* process in the same group and simultaneously sends nonblocking messages to corresponding pair processes in all the other three groups. This corresponds to the new method described below.

3.1 Concurrent Exchange Algorithm-Baseline

Unlike the traditional pairwise method, the concurrent exchange algorithm involves only log (n/C) steps for n processes, where C is the *concurrency index*. This method communicates in such a way that with sufficient concurrency in the network, it would incur much less communication overhead than the traditional pairwise approach. In this method, the number of processes is divided into C groups. Each process in a group has a peer process in every other group. Processes in every group do a log(n/C) step pairwise exchange. Before entering the log(n/C) step exchange loop, every process initiating nonblocking *put* of its data to its *peer* process. Time for this is τ. After entering the loop, for each of the log(n/C) steps, every process initiates a nonblocking send of the data it has both to its *pair* process in its group and the corresponding *peer-of-pair* processes in every other group. The *pair* process is in the same group as the process itself while the *peer-of-pair* is the corresponding pair process in the other group. This requires a notification mechanism that notifies the remote process of the completion of a send done to it. With sufficient concurrency, the time taken by this method to complete can be approximated as shown in Equation (3). Because there are only log(n/C) steps in the critical path, the time for the All-Gather exchange is $T(m*2^i)$ per step, where $T(m)$ is the time to transmit a message of size m, bidirectionally. In addition, each step has the initiation cost of a nonblocking message to C-1 groups. This is shown as the left side of the summation in Equation (3). Hence time per step is: $T(m*2^i)+\tau$. Sum over log(n/C) steps for this is shown as T_3 in Equation (3). τ is the time to prepare and initiate a message.

$$T_3 = (C-1)\tau + o\sum_{i=0}^{\log(n/C)-1}(T(m\cdot 2^i)+\tau) \qquad (3)$$

Dotted lines on the right side of Figure-3 represent nonblocking operations. Figure-3 shows how this algorithm works for four processes for *concurrency index* $\boldsymbol{C}$ = 2. the left side of Figure 3 represents the traditional pairwise exchange algorithm. The right side of Figure 3 shows how the new algorithm described here works. The encircled step on the right side represents the step before the exchange were each process initiates a nonblocking send of its data to its *peer* process in the other group. The cost of this step is constant and is less than 5 microseconds for most of the networks (GM2, Infiniband, Elan4) on which we measured it. The cost of doing this, however, shows up in the critical path for very small messages.

This method works well only for short and medium-length messages. The improvement that can be obtained, time-wise, can be represented as a difference between the time for All-Gather using the pairwise method (T_2) and the new method (T_3). This can be seen in Equation (4) as T_2-T_3.

$$T_2 - T_3 = o\sum_{i=\log\frac{n}{C}}^{\log n-1} T(m\cdot 2^i) + (\log\frac{n}{C}+1)\cdot\tau \qquad (4)$$

When the message size is very large we need a special provision to account for the way the NIC handles consecutive asynchronous messages. Hence, for very large messages, we propose a modification to the current algorithm.

3.2 Concurrent Exchange with Message Splitting

We describe the method for *concurrency index* of 2. This method has the same number of steps as in a pairwise method but instead of sending whole message to a pair process, we split the message into two halves and send the second half of the message to what we call the *co-pair* process. We define the *co-pair* process as a process at the same distance as the *pair* process but in the opposite direction. Hence in this method, two pairwise exchanges happen simultaneously in two directions. The objective of this method is to employ the unused links in a pairwise exchange method and hence offer more concurrency in the exchange. Figure 4 shows this exchange on four processors. In the first step of the exchange loop, each process sends one half of the data it has to its *pair* process and the other half of its data to the *co-pair* process. In step two, each process has four halves of the data. It now sends half of it to its new *pair* and the other half to its new *co-pair*. The cost of this method when doing two simultaneous exchanges can be approximated as T_4, shown in Equation 5.

The tradeoff this method involves is that for each step, it may split

$$T_4 = o\sum_{i=0}^{(\log n)-1} T\left(\frac{m}{2}2^i\right) + \tau \qquad (5)$$

the message into many chunks; hence, on networks with no

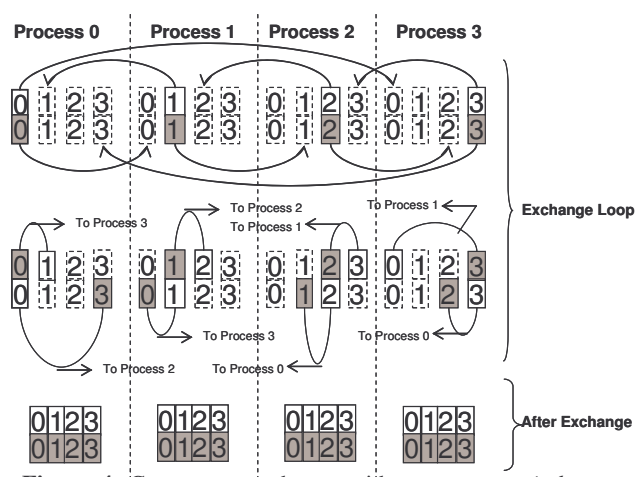

Figure 4: Concurrent exchange with *concurrency index* of 2 and data splitting. Data on each process has been divided into two halves.

1-59593-061-2/05 $20.00 © 2005 IEEE

noncontiguous transfer support, this would involve multiple message transfer initiation penalties. On networks with noncontiguous data transfer support, like Infiniband and Elan-4, the transmission of these chunks can be handled as a single message.

4. EXPERIMENTAL RESULTS

We have implemented algorithms described in Section 3 and evaluated them experimentally. The experiments were performed on two clusters:

1) *Infinband Cluster*: This is a cluster of MAC OS-X nodes at Virginia Tech. Each node has dual 2.3-GHz PowerPC 970FX processors with 4 GB RAM and one Mellanox Cougar InfiniBand 4x HCA.

2) *Elan4 Cluster*: This is a Linux cluster at PNNL. The nodes we were using had dual 1.5-GHz Intel IA64 Madison processors with 6 GB RAM and the Quadrics Elan-4 interconnect.

All the experiments were performed while running on only one processor per each dual-processor node.

4.1 Concurrency and Overhead

The proposed method relies on availability of concurrency for Infiniband and Quadrics networks. To evaluate the available network concurrency, we ran two experiments, both on eight processors. The first experiment involved all the participating processors doing puts in a ring, and we timed the completion of the put. In Figure 5 (left side) this time is compared to the time required to do two puts of the same size in two different directions and waiting for them to complete. It can be seen that for messages up to 16 KB, doing two puts to different destinations takes much less than twice the time taken to complete one put to a single destination. This confirms concurrency in the network. This ability applies not only to the NIC but also to the switch. The results in Figure 5 (left) provide a motivation for splitting large messages as described in Section 3.2.

On the right side of Figure 5, time taken to do one put is compared to the time taken for initiating two puts to two different processes but waiting for only one of them to complete. This is done to measure the overhead incurred when additional nonblocking puts are pending, as done in the new method. It can be seen from Figure 5 that there is higher overhead on the Infiniband network whereas the overhead on Quadrics is very limited. This experiment was done on only two destinations because only a concurrency index of 2 is evaluated in experimental evaluations in the balance of this paper.

4.2 All-Gather Performance

We implemented the All-Gather operation using network concurrency as described in Section 3. In addition, we

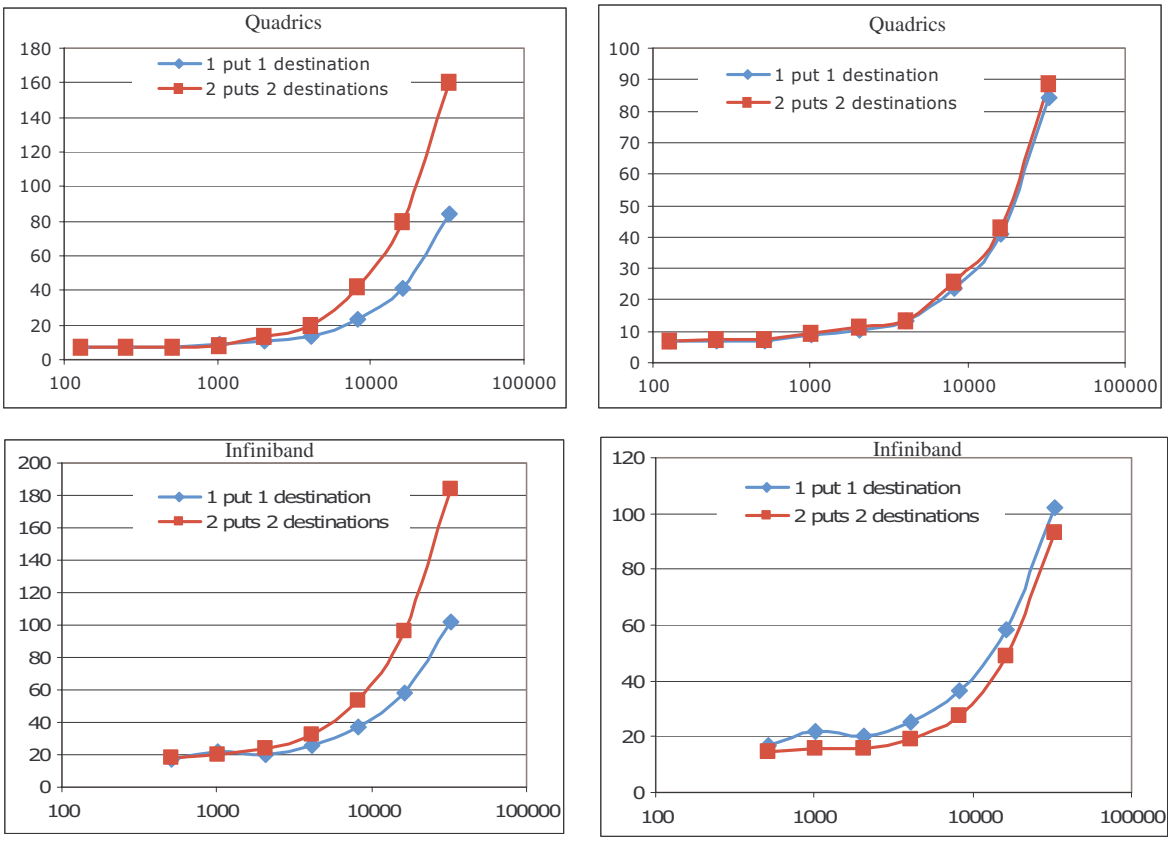

Figure 5: Network concurrency tests (left): Time to initiate and complete 1 put vs. time to initiate and complete 2 puts of the same size to different destinations. Overhead test (right): Time to initiate and complete 1 put vs. time to initiate 2 puts to 2 destinations and complete the first

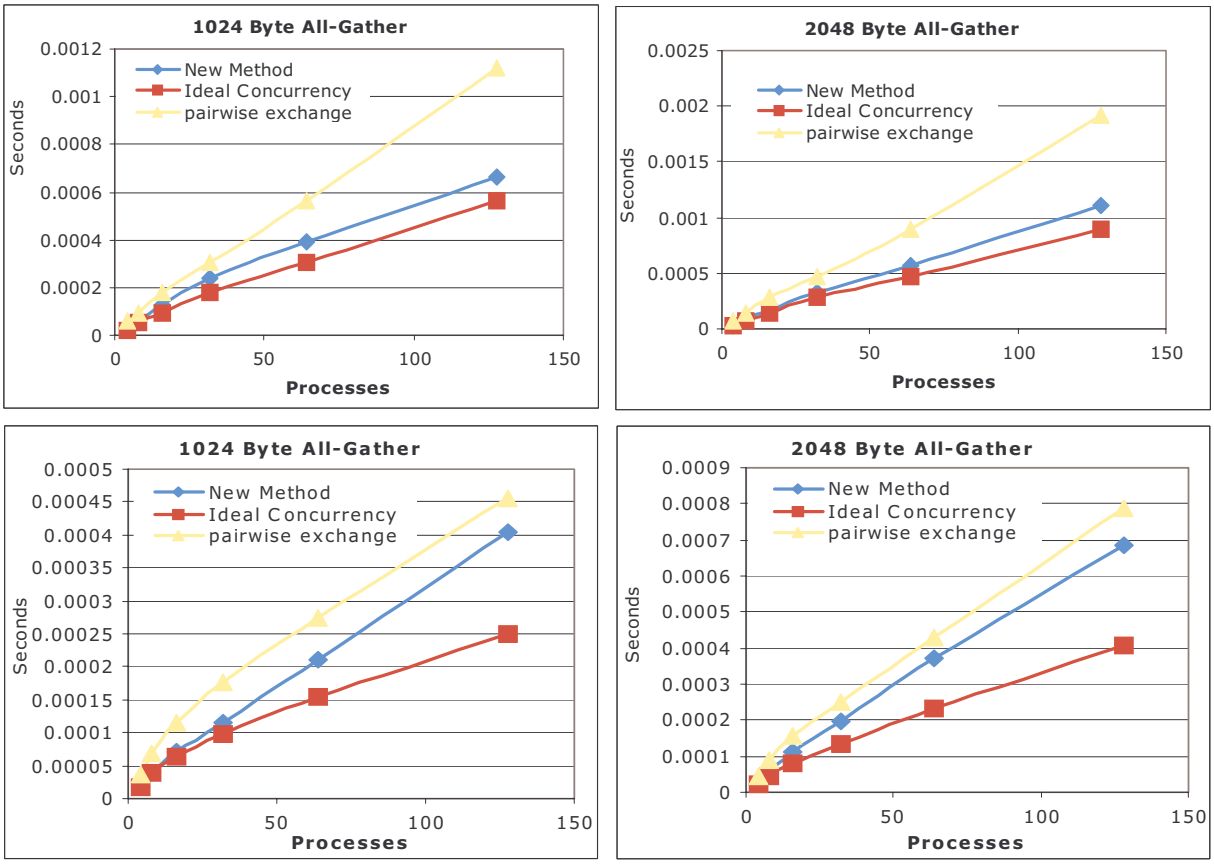

Figure 6: Ideal vs. Actual time in comparison to pairwise exchange Top: Infiniband, Bottom: Elan4

implemented a nonblocking message-passing version of the MPI_AllGather operation to correspond to the complexity described in Equation (2). We did this for two reasons: 1)MPI_AllGather on quadrics was not based on pairwise exchange, and 2)we wanted to give a contrast to the pairwise exchange-based MPI_AllGather in MVAPICH that we compared against to understand the effectiveness of their implementation. From a theoretical standpoint, assuming a negligible cost of initiating a message and sufficient network concurrency in the network, the new algorithm should give a performance improvement as represented by Equation (4), with concurrency index of 2. This corresponds to the example shown in Figures 3 and 4 in Section 3 with a concurrency index of 2.

Figure 6 shows the estimated and actual performance for a 1024-byte exchange on 128 processors of the Infiniband and Quadrics networks. The estimated performance was calculated by setting the values for the corresponding message size and number of processes in Equation (2). It can be seen from Figure 6 that the actual performance is close to that of the expected performance for Infiniband. We believe that the cost of descriptor processing overhead in the Infiniband Verbs API layer (see Figure 5 and Section 4.1) is in part responsible for the mismatch. The descriptor processing time is constant and thus adds a constant overhead. Unlike the Infiniband clusters where the measurements obtained in consecutive runs were very close, the results on the Quadrics-based cluster appeared to have substantial variability between runs (we use average timing from multiple runs to produce the graphs). We attribute these variations to the system "noise" caused by daemons that interrupt the processes in the job. This noise has been observed by some other experiments run on this cluster and has been measured; however, such analysis is beyond the scope of this work. Because collective operations amplify the performance degradations caused by system noise, we believe this factor is responsible for the larger discrepancy between theoretical and measured values.

Figure 7 shows the execution time of the proposed algorithm compared to the MPI_AllGather operation and the pairwise All-Gather algorithm we implemented measured on the Infiniband cluster. For each processor count, we plot $\frac{T_{proposed\ method}}{T_{MPI_AllGather}} \cdot 100\%$ and $\frac{T_{proposed\ method}}{T_{pairwise}} \cdot 100\%$ quantities. The values less than 100% indicate that the proposed method is faster than the reference implementation. Figure 7 shows this for messages ranging from 512 to 32768 bytes. Note that on 128 processors, when each processor does an exchange of 32768 bytes, the total amount of data transferred to each processor is ~4MB (127*32768). The MPI_AllGather on the Infiniband cluster, a part of MVAPICH from Ohio State University, delivers very good performance (better than the pairwise algorithm) and has been implemented with less overhead. Still it can be clearly seen that our method, for all the tested processor cases and message sizes delivers superior performance due to its concurrency and asynchronous nature, .

1-59593-061-2/05 $20.00 © 2005 IEEE

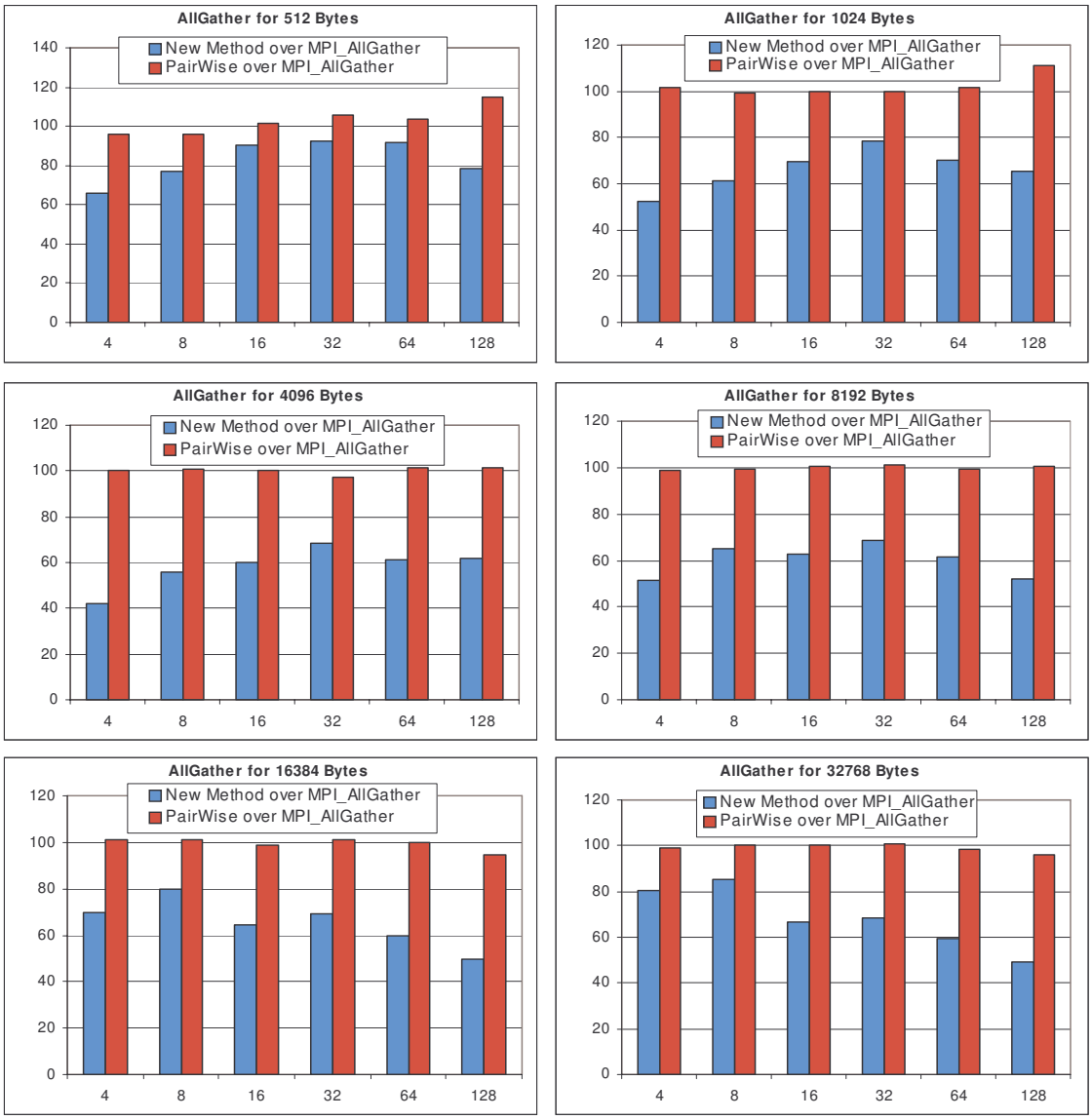

Figure 7: Execution time of the new method as a percentage of the execution time of MPI_AllGather and execution time of the pair-wise algorithm on Infiniband

The method clearly shows a performance gain even for very large All-Gather operations. For larger message sizes and higher processor count, the performance advantage tends to be higher. It ranges from 7% to 58% over the MPI_AllGather and Pairwise algorithm.

Figure 8 shows the same quantities for the Elan-4 cluster. For this experiment, we used the MPI implementation from Quadrics (QSNETlibs Version 1.8). On that cluster, the new method shows even a higher performance improvement over MPI_AllGather than on Infiniband. It was measured to range from 19% to 88% over MPI_AllGather and from 11 to 66% over the pairwise algorithm. For very small messages (<128 bytes) we did not observe any performance improvement over traditional flat tree or pairwise exchange based methods. We attribute this to our implementation of these algorithms and the overead of initiating multiple messages.

5. CONCLUSION AND FUTURE WORK

The paper described a novel algorithm for optimizing performance of all-to-all collective communication. It attempts to take advantage of concurrency available in modern networks such as Infiniband or Quadrics and relies on multiple nonblocking RDMA operations. In particular, the technique has been deployed in the MPI_AllGather operation, one of the most communication-intensive collective operations known. The experimental results on the Infiniband and Quadrics Qnet-II clusters showed substantial performance advantage of the proposed algorithm over the MPI_AllGather implementations in MPI and the pairwise All-Gather algorithm. Our future work will focus on applying the

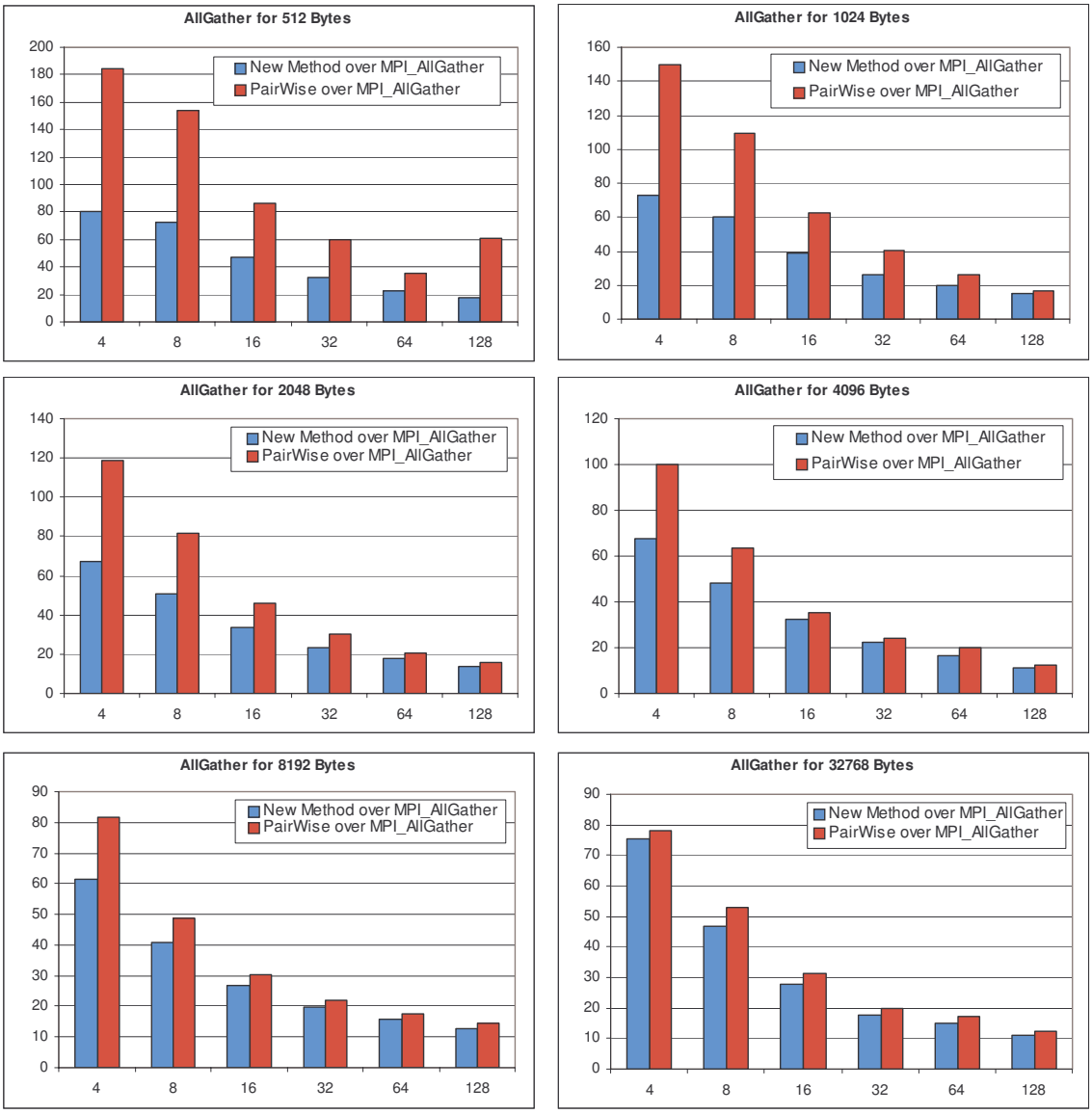

Figure 8: Execution time of the new method as a percentage of the execution time of MPI_AllGather and execution time of the pair-wise algorithm on Quadrics Qsnet-II

described methodology to other collective all-to-all communication operations. We will also investigate the best ways to use these algorithms for small (8 to 128 byte) messages. Finally, we plan to incorporate these algorithms into collective communication in the ARMCI communication library [26].

6. REFERENCES

[1] "Elan4 NIC Diagram."http://www.quadrics.com/Quadrics/QuadricsHome.nsf/DisplayPages/79CEA2573DC8909880256D88004A5C01/$File/QuadricsHotChipsOld1.ppt

[2] "Infinihost NIC Diagram."http://www.hd3d.com/diagram02.html

[3] J. Beecroft, D. Addison, F. Petrini, and M. McLaren, "QsNetII: An Interconnect for Supercomputing Applications," Quadrics.http://doc.quadrics.com/Quadrics/QuadricsHome.nsf/DisplayPages/7B383554432E5F4D80256EAD0010AA83/$File/QsNet+Hot+chips+paper1.pdf

[4] N. Arnold, K. Thilo, E. B. Henri, and M. Jason, *Object-based collective communication in Java*. Palo Alto, California, United States: ACM Press, 2001.

[5] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D. G. Payne, and J. Watts, "Interprocessor collective communication library (InterCom)," 1994.

[6] B. Vasanth, B. Jehoshua, C. Robert, E. Pablo, H. Alex, H. Ching-Tien, K. Shlomo, and S. Marc, "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 154-164, 1995.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message

passing interface standard," *Parallel Computing*, vol. 22, pp. 789-828, 1996.
[8] L. Jiuxing, "Designing high performance and scalable mpi over infiniband," 2004, pp. 186
[9] S. Xantheas and G. Fanourgakis, "Polarizable Water Potentials," 2005, pp. Personal Communication
[10] Z. Yeliang, V. Tipparaju, J. Nieplocha, and S. Hariri, "Parallelization of the NAS Conjugate Gradient Benchmark Using the Global Arrays Shared Memory Programming Model," 2005.
[11] S. Baden, P. Collela, D. Shalit, and B. Van Straalen, "Abstract Kelp," presented at International Conference on Computational Science, San Francisco, CA, 2001.
[12] R. Bariuso and A. Knies, *SHMEM's User's Guide*: Cray Research, Inc., 1994.
[13] A. Basumallik, S.-J. Min, and R. Eigenmann, "Towards OpenMP execution on software distributed shared memory systems," presented at Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI'02), 2002.
[14] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems," presented at RTSPP of IPPS/SDP'99, 1999.
[15] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware Efficient Supercomputing," *IEEE Transactions on Computers*, vol. C-34, pp. 892-901, 1985.
[16] S. V. Sathish, E. F. Graham, and D. Jack, *Automatically tuned collective communications*. Dallas, Texas, United States: IEEE Computer Society, 2000.
[17] "InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000."www.infinibandta.org
[18] V. Tipparaju, J. Nieplocha, and D. Panda, "Fast collective operations using shared and remote memory access protocols on clusters," presented at Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 2003.
[19] S. Susumu, M. Hiroyuki, N. Shigeki, and H. Jun-ichi, *Scatter and gather operations on an asynchronous communication model*. Como, Italy: ACM Press, 2000.
[20] W. Yu, D. Buntinas, and D. Panda, "Scalable and High Performance NIC-Based Allgather over Myrinet/GM," presented at IEEE Cluster Computing, San Diego, CA, 2004.
[21] A. Mamidala, J. Liu, and D. K. Panda, "Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms," presented at IEEE Cluster Computing, San Diego, CA, 2004.
[22] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan, "Broadcast/multicast over Myrinet using NIC-assisted multidestination messages," *Network-Based Parallel Computing, Proceedings*, vol. 1797, pp. 115-129, 2000.
[23] W. Yu, D. Buntinas, and D. K. Panda, "High performance and reliable NIC-based multicast over Myrinet/GM-2," presented at Parallel Processing, 2003. Proceedings. 2003 International Conference on, 2003.
[24] N. J. Boden, D. Cohen, E. Felderman, A. E. Kulawick, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, vol. 15, pp. 29-36, 1995.
[25] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie, "Hardware- and Software-Based Collective Communication on the Quadrics Network," presented at IEEE International Symposium on Network Computing and Applications, Boston, 2001.
[26] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High Performance Remote Memory Access Communication: The ARMCI Approach," *IJHPCA Special Issue on ACT Tools*, 2005.