Performance Evaluation of High-Speed Interconnects using Dense Communication Patterns[1]

R. Fatoohi, K. Kardys, S. Koshy, S. Sivaramakrishnan
Computer Engineering, San Jose State Univeristy, San Jose, California, USA, Email: rfatoohi@sjsu.edu

J. S. Vetter
Oak Ridge National Lab, Oak Ridge, Tennessee, USA

Abstract

We study the performance of high-speed interconnects using a set of communication micro-benchmarks. The goal is to identify certain limiting factors and bottlenecks with these interconnects. Our micro-benchmarks are based on dense communication patterns with different communicating partners and varying degrees of these partners. We tested our micro-benchmarks on five platforms: an IBM system of 68-node 16-way Power3, interconnected by a SP switch2; another IBM system of 264-node 4-way Power PC 604e, interconnected by a SP switch; a Compaq cluster of 128-node 4-way ES40/EV67 processor, interconnected by an Quadrics interconnect; an Intel cluster of 16-node dual-CPU Xeon, interconnected by an Quadrics interconnect; and a cluster of 22-node Sun Ultra Sparc, interconnected by an Ethernet network. Our results show many limitations of these networks including the memory contention within a node as the number of communicating processors increased and the limitations of the network interface for communication between multiple processors of different nodes.

1. Introduction

Most computer designers of network interconnects focus on the point-to-point communication performance of the network. They normally present performance data in terms of bandwidth and latency of single communication link. There have been a number of theoretical studies on the impact of the network topologies and communication patterns on the performance of high-performance computing systems but few empirical studies at scale. Most of the benchmarking efforts are based on simple routines that measure latency and bandwidth aspects of a network and ignoring other factors such as the multiplicity of communication sessions between multiple nodes and the locality of these sessions.

There have been several recent studies on performance of high-performance networks using different benchmarks. Bell, et al. [1] used a set of micro-benchmarks for measuring bandwidth, latency, and software overhead on five supercomputing networks: Cray T3E, IBM SP, Quadrics, Myrinet, and Gigabit Ethernet. They showed the benefits of overlapping communication with computation and analyzed their results using a variant of LogP model [3]. Liu, et al. [8] developed another set of benchmarks to evaluate three cluster interconnects: InfiniBand, Myrinet, and Quadrics. Their benchmarks include traditional measures, such as latency and bandwidth, as well as other features such as user-level access for performing communication and remote direct memory access, and showed the performance impact of these features. Finally, Kurmann, et al. [7] developed yet another set of micro-benchmarks to find the best cost performance point for networks in PC clusters. They designed their benchmark to evaluate the performance of interconnect for specific communication patterns. They also experimented with several applications and concluded that many applications are not very sensitive to full bisection bandwidth.

Our work was inspired by Kurmann's work [7] so we used some of their communication patterns. Our goal is to develop a methodology in evaluating interconnects without knowing their topologies or the mapping of the processes onto the processors. Our focus is on dense communication patterns in order to identify certain limiting factors and bottlenecks of these interconnects. So we designed our benchmark in such a way that we stress the network with multiple messages from different nodes to identify hot spots within the network. Our micro-benchmark avoids point-to-point as well as collective communication

[1] Part of this work was performed under the auspices of the U.S. Dept. of Energy by University of California LLNL under contract W-7405-Eng-48.

Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)
1530-2016/05 $20.00 © 2005 IEEE

routines (for interested readers, please refer to mpptest for a point-to-point communication benchmark and Pallas for both point-to-point and collective communication benchmarks). In addition to different communication patterns, we employ different message sizes and number of nodes to determine the sensitivity of the network to these changes.

2. Experimental Platforms

We ran our algorithms on five platforms consisting of two IBM SP (Scalable PowerParallel) systems: Frost and Blue and three clusters of machines: TC2K, Pengra, and CE. All these machines except CE are located at Lawrence Livermore National Laboratory (LLNL), Livermore, California. The CE (Computer Engineering) cluster is located at San Jose State Univeristy (SJSU), San Jose, California. All machines run different versions of the Unix operating system.

The IBM SP Frost system at LLNL is composed of 68 Symmetric MultiProcessor (SMP) nodes, each with 16 processors totaling 1088 processors with a peak performance of 1.6 Tflops. Each processor is an IBM RS/6000 POWER3-II "Nighthawk-2" at 375 MHz.

All Frost nodes are connected by an IBM SP switch2, a proprietary IBM interconnect [6]. It is a bidirectional, multistage interconnection network with 2 GB/s peak node-to-node bandwidth. The switch consists of two basic hardware elements: switch board and communication adapter. There is one switch board per an SP frame (SP frame contains 16 nodes on Frost). The switch board contains eight switch chips wired as a bidirectional 4-way to 4-way crossbar. Also, every Frost node has two switch adapters.

The IBM Blue system at LLNL is composed of 264 SMP nodes, each with four processors totaling 1056 processors with a peak performance of 722 Gflops. Each processor is an IBM PowerPC 604e at 332 MHz. All Blue nodes are connected by an IBM SP switch. It is similar to IBM SP switch2 except that the peak node-to-node bandwidth is 300 MB/s.

The Compaq TC2K system at LLNL is composed of 128 SMP nodes, each with four processors totaling 512 processors with a peak performance of 681 Gflops. Each processor is an AlphaServer system ES40, with the EV67 processor which contains the Alpha 21264 chip at 667 MHz.

The Pengra system at LLNL is composed of 16 dual-CPU nodes with a peak performance of 141 Gflops. Each CPU is a 2.2 GHz Intel Xeon and uses the Intel E7500 chipset.

Nodes of both TC2K and Pengra are connected by a Quadrics Network (QsNet) [9]. QsNet is a bidirectional multistage interconnection network with a transmission bandwidth of 400 MB/s in each direction. It consists of two hardware building blocks: a programmable network interface, Elan, and a switch, Elite. The Elan network interface connects a processing node via the PCI bus and a multistage network. In addition to generating and accepting packets, Elan provides local processing power, through two processing engines, to implement high-level, message passing protocols, such as MPI. The Elite switch is an 8-way bidirectional crossbar switch. QsNet connects Elite switches in a quaternary fat-tree topology.

The CE cluster at SJSU is composed of 22 Sun Ultra Sparc nodes, each with a CPU of 333 MHz. The nodes are connected by a 10/100 Ethernet network. The CE cluster is used here mainly for code development and testing on a bus-based network.

3. Algorithms

We implemented several algorithms to study the limitations of a set of interconnects.

3.1. Congested-controlled AAPC

In all-to-all personalized communication (AAPC), each process sends a distinct message to every other process. AAPC, also called total exchange, is widely used in many algorithms, such as Fast Fourier Transform (FFT), matrix transpose, and sorting. There are many implementations of AAPC on parallel computers, some of them have been optimized on specific architectures and can be invoked as library routines. A simple implementation of AAPC, where all processes communicate with all other processes simultaneously, may lead to network congestion on most networks. Grama, et al. [5] describe AAPC implementations on ring, mesh, and hypercube networks based on multiple steps where in each step every process sends a message to its neighbor. Kurmann, et al. [7] describe a phased AAPC algorithm that attempts to reduce network congestion.

A phased AAPC algorithm, also called congestion-controlled AAPC, proceeds as follows: In the first phase, each process sends data to its next higher neighbor (based on its rank) and receives data from its next lower neighbor. In the next phase, each process sends data to its next but one higher neighbor and receives data from next but one lower neighbor and proceeds such away till the last phase where every process sends data to its lower neighbor and receives data from its higher neighbor. Each phase is separated by global synchronization using barriers, which may add some overhead. A pseudo code representation of the algorithm is shown below:

Algorithm 1: Congested-controlled AAPC

for stride = 1 to Size - 1 do

1. *Start timer*
2. *Concurrently send bytes to (myRank + stride) % Size & receive bytes from (myRank - stride) % Size*
3. *End timer*
4. *Wait for barrier*
5. *Do all-to-one reduction to find maxTime*

Where *Size* is number of processes within the communicator and *maxTime* is the maximum time taken by all processes in each phase.

3.2. Simple pair-wise communication

In this algorithm, a set of processes communicates in pairs. All pairs send and receive data in parallel and at full duplex. The algorithm proceeds in phases such that the stride (hop) or the distance between the communicating processes increases in each phase until it reaches its maximum (*Size* –1), as shown below:

Algorithm 2: Simple pairwise

for stride = 1 to Size - 1 do

1. *Start timer*
2. *Concurrently send bytes to & receive bytes from (myRank + stride) % Size and (myRank - stride) % Size*
3. *End timer*
4. *Wait for barrier*
5. *Do all-to-one reduction to find maxTime*

The first phase of the algorithm is repeated in the last phase (*Size* –1) while the second phase is repeated in the next to last phase (*Size* -2) and so on. In the middle phase (*Size* /2), process 0 pairs with process *Size*/2 twice as well as other pairs; i.e., there are *Size*/2 pairs. For example, consider an eight process case, phase 1 has the following pairs (0,1), (1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,0); phase 2 has (0,2), (1,3), (2,4), (3,5), (4,6), (5,7), (6,0), (7,1); phase 3 has (0,3), (1,4), (2,5), (3,6), (4,7), (5,0), (6,1), (7,2), phase 4 has (0,4), (1,5), (2,6), (3,7), (4,0), (5,1), (6,2), (7,3), and so on.

3.3. Cumulative pair-wise communication

In this algorithm, the number of communicating pairs in the network is increased during successive phases of communication. In the first phase only two processes communicate with each other (0,1). In the second phase, the number of communicating processes increases to four (0,1), (2,3), and that number increases to six in phase three (0,1), (2,3), (4,5) and so on, as shown below:

Algorithm 3: Cumulative pairwise

for phase = 1 to Size / 2 do

1. *Start timer*
2. *If (myRank < 2 × phase)*
3. *Concurrently send bytes to & receive bytes from (myRank + 1) if myRank is even or (myRank – 1) if myRank is odd*
4. *End timer*
5. *Wait for barrier*
6. *Do all-to-one reduction to find maxTime*

3.4. Random pair-wise communication

In this algorithm, the communicating processes are determined at runtime, randomly. This is done by using a random-shuffling algorithm, which shuffles the process array (*processArray*) in each phase. Here, initially *processArray* consists of process ranks in the ascending order of their ranks. The communicator is split up into two equal parts, and communication takes place between the two halves of the communicator. At the end of each phase, where the number of phases (*maxPhase*) is chosen at runtime, the process array is shuffled so that the two halves, for the next phase, contain different processes, as shown below:

Algorithm 4: Random pairwise

for phase = 1 to maxPhase do

1. *Divide processArray into two halves*
2. *Partner = processArray [((Size/2) + myRank) % Size]*
3. *Start timer*
4. *Concurrently send bytes to & receive bytes from partner*
5. *End timer*
6. *Wait for barrier*
7. *Do all-to-one reduction to find maxTime*
8. *Shuffle processArray randomly*

4. Implementations and Results

We implemented the four algorithms, described in section 3, on different configurations of the five platforms described in section 2. The code was written in C with MPI calls for all our implementations. For each algorithm, there are several implementation options including: the message size, the number of repetitions in each routine, and type of communication: blocking or non-blocking send and receive.

We ran the four algorithms on LLNL machines (Frost, Blue, TC2K, and Pengra) through a batch system. The execution environment was not dedicated: i.e., there were other jobs running while our jobs were

executing. We ran some jobs more than once when we noticed some abnormality on the timing results. But, in general, the results were consistent across multiple runs. The batch system at LLNL schedules submitted jobs based on the available resources, job sizes, time limits and job priority. The user has no control over which processor, a node, or a set of nodes that the job will run on - it is up to the scheduler. However, the user can specify that all processes of a job to be executed on a single node. On the CE machines, we used a simple machine configuration file to run the jobs.

We used seven message sizes to run the four algorithms: 1K, 3K, 10K, 30K, 100K, 300K, and 1000K bytes. We set the number of repetitions to 1000 to get meaningful results. We also used the algorithms in the non-blocking mode since we found out that in some cases the blocking mode caused the jobs to hang due to buffer space limitations. In addition, we set the number of phases that random shuffling can be performed in Algorithm 4 to 10.

We tested the four algorithms on three different configurations: 1) multiple nodes with a single process per node, 2) multiple processes on a single node, and 3) multiple processes on multiple nodes. Not all possible combinations of algorithms, platforms and configurations are reported here for several reasons. Given the amount of data generated by these experiments, we focus on the results that have some significance for the sake of brevity. Also, there were cases that the results did not meet our quality insurance standards. In addition, we were also limited by the number of nodes that we were able to run using the batch system at LLNL. In our case, the largest number of nodes we used is four nodes on Pengra and eight nodes on each of Frost, Blue, TC2K, and CE.

In all cases, we measured the maximum time taken by all processes. Then we calculated the bandwidth by multiplying the message size by the number of sends and receives that each process performs (it is two for algorithms 1, 3 and 4 while it is four for algorithm 2) and dividing the result by the measured time.

4.1. Single process per node

We implemented the four algorithms on eight nodes of Frost, Blue, TC2K, and CE and four nodes of Pengra and running one process per node. We used the configuration to find out inter-node network limitation.

Figures 1 through 3 present the results of implementing algorithm 1 (congested-controlled AAPC) on Frost, Pengra, and CE (Frost and Blue results are very similar). The results show different phases of the algorithm have a minimum impact on the bandwidth; i.e., as we increase the distance between the communicating nodes, the bandwidth remains about the same. An interesting case is the middle phase (phase 4 for the eight node configuration) since here each process sends and receives messages to the same partner in a full-duplex mode. We noticed a minor bandwidth change in the middle phase compared to the others. The results also show an increase in bandwidth as the message size increases. The results for the bus-based CE cluster show more fluctuation than the others due to the nature of an Ethernet network.

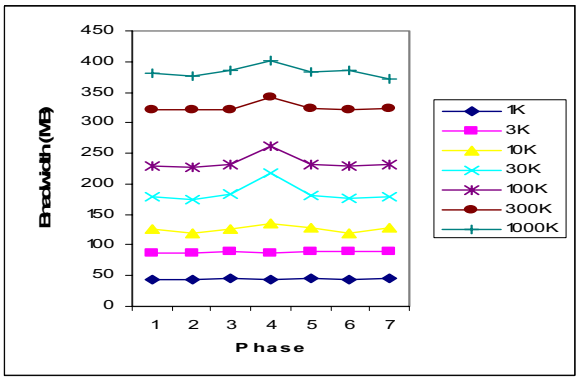

Figure 1. Congested-controlled AAPC on Frost (8 processes on 8 nodes)

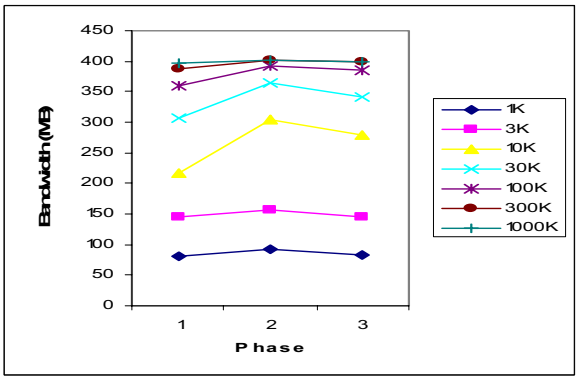

Figure 2. Congested-controlled AAPC on Pengra (4 processes on 4 nodes)

Figure 4 shows the results of implementing algorithm 2 (simple pairwise) on Frost. As for algorithm 1, algorithm 2 shows similar results on the other machines – mainly stable bandwidth results except for minor differences for phase 4.

Figures 5 and 6 present the results of implementing algorithm 3 (cumulative pairwise) on Frost and TC2K. The results on Frost, as well as on Blue, show no performance degradation as the number of communicating nodes increases, while TC2K shows some degradation. Consistently, we noticed a drop of about 20% to 30% in bandwidth as the number of pairs

doubled from one pair to two regardless of the total number of nodes employed.

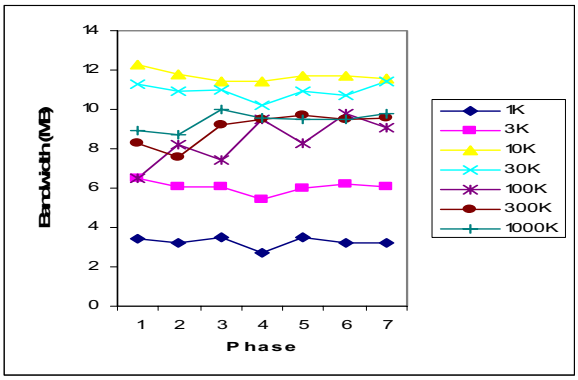

Figure 3. Congested-controlled AAPC on CE (8 processes on 8 nodes)

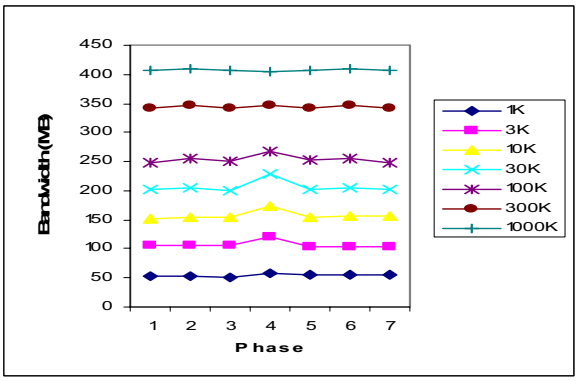

Figure 4. Simple pairwise on Frost (8 processes on 8 nodes)

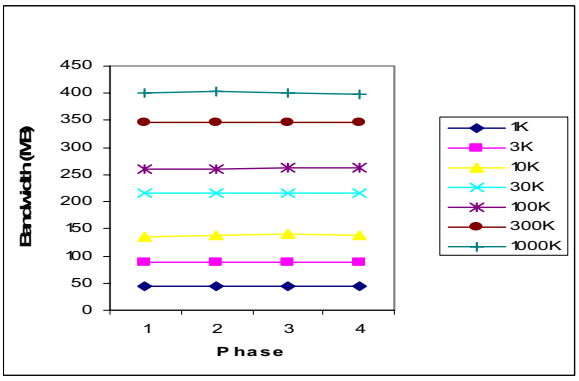

Figure 5. Cumulative pairwise on Frost (8 processes on 8 nodes)

Figure 7 presents the results of implementing algorithm 4 (random pairwise) on Frost (the Blue and Pengra results are similar to Frost results, the TC2K results are noisy with few outliers, and CE results are similar to CE results for other algorithms). The results, on Frost, as well as on Blue, are very stable regardless to which nodes are paired to the others.

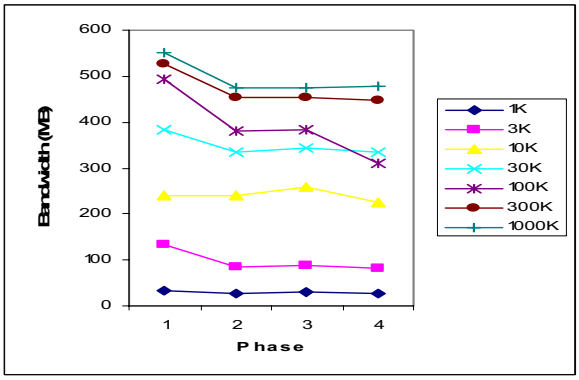

Figure 6. Cumulative pairwise on TC2K (8 processes on 8 nodes)

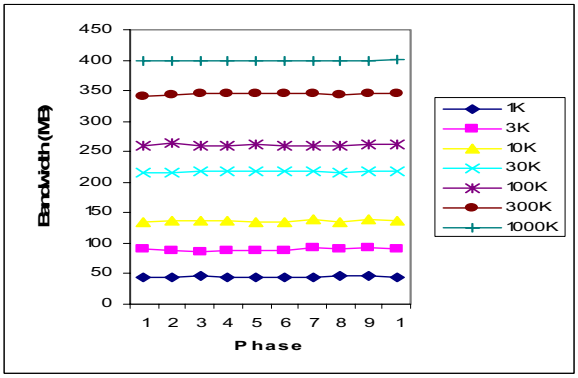

Figure 7. Random pairwise on Frost (8 processes on 8 nodes)

4.2. Multiple processes on a single node

We implemented the four algorithms on Frost, Blue and TC2K (Pengra is a dual-CPU machine while CE nodes have a single CPU each). Here we show samples of these results. Figure 8 presents the results of implementing algorithm 1 on 16 processors of Frost. The results show no dependency on the phase value, as it would be expected. Similar results were obtained in implementing algorithms 2 and 4 (not shown here) on these machines with no phase or pairing dependency, except for some small glitches.

The most interesting results on a single node are that of algorithm 3 to observe the achieved bandwidth as we increase the number of processes. Figures 9 and 10 present the results of implementing algorithm 3 on Frost and TC2K. The Frost results show a significant drop in bandwidth for messages larger than 3K bytes as the number of communicating processors increases. Actually, the drop in bandwidth is about 50% for messages of size 100K bytes and above (from 600 MB/s for two pairs to 300 MB/s for eight pairs). A small drop was observed on Blue (not shown here) as

the number of pairs doubled from one pair to two. Also, a larger drop in bandwidth for all messages was observed on TC2K. These results show the bandwidth limitation of all SMPs, for larger messages. As Bland, et al. [2] observed in evaluating a 32-way IBM p690 machine, different demands on the memory subsystem can cause performance degradation for large messages, especially in simultaneous swapping of messages between 16 pairs of distant processors.

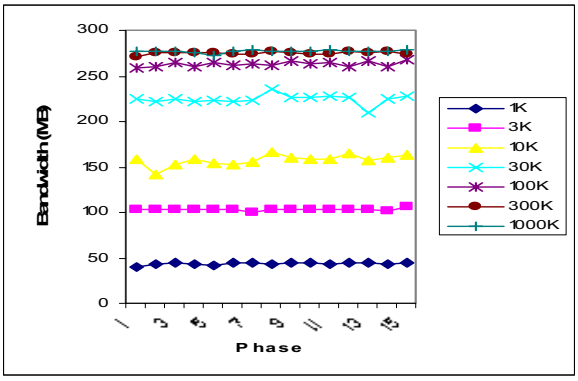

Figure 8. Congested-controlled AAPC on Frost (16 processes on single node)

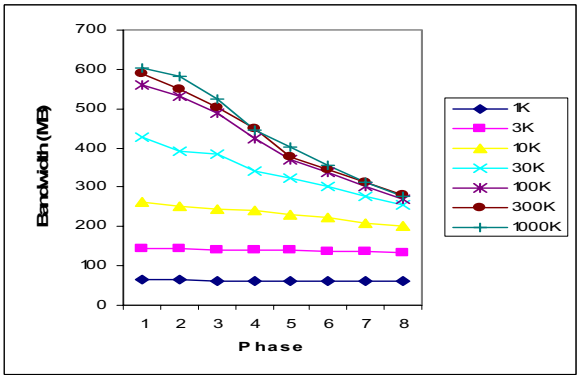

Figure 9. Cumulative pairwise on Frost (16 processes on single node)

4.3. Multiple processes on multiple nodes

We implemented the four algorithms on different multiple nodes of Frost, Blue and TC2K using multiple processes per node. We used several configurations but here we present the results of 32 processes on eight nodes. We also implemented 64 processes on four nodes and 128 processes on eight nodes of Frost.

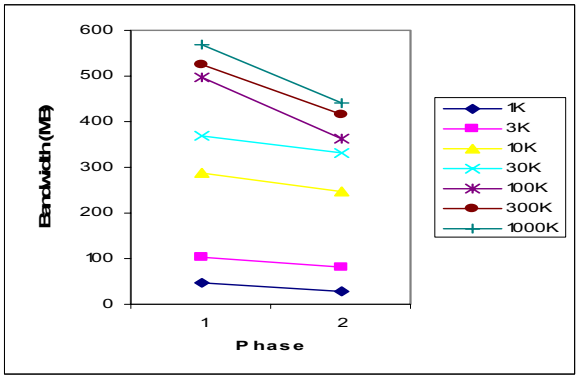

Figure 10. Cumulative pairwise on TC2K (4 processes on single node)

Figure 11 presents the results of implementing algorithm 1 on eight nodes of Frost using 32 processes (similar results obtained on Blue). These results show a drop in bandwidth as we move away from nearby to farther apart communication. For example, we noticed a drop of about 40% between phase 1 and phase 16 (the middle phase) on both machines. This drop can mainly be attributed to the number of network interfaces that are shared between the processors of a single node. The four processors of a Blue node share one network interface while the sixteen processors of a Frost node share two interfaces. Here each processor gets only a portion of the available bandwidth when multiple processors on one node send messages to processors on another node. This degradation is more apparent in phase 16 since most communication is between processes located on different nodes than in phase 1 where most communication is between processes within a node. Similar observations were reported by Dunigan [4].

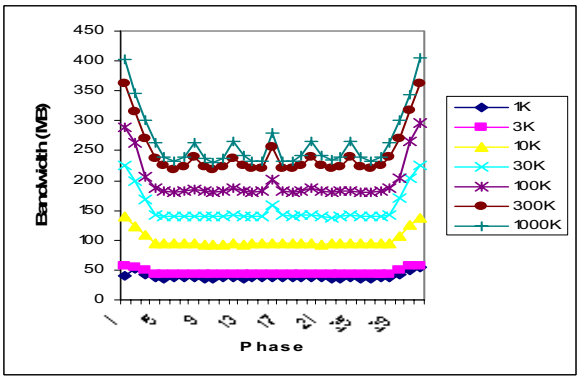

Figure 11. Congested-controlled AAPC on Frost (32 processes on 8 nodes)

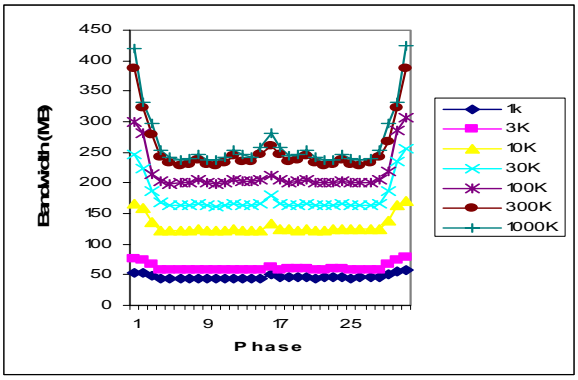

Figure 12. Simple pairwise on Frost (32 processes on 8 nodes)

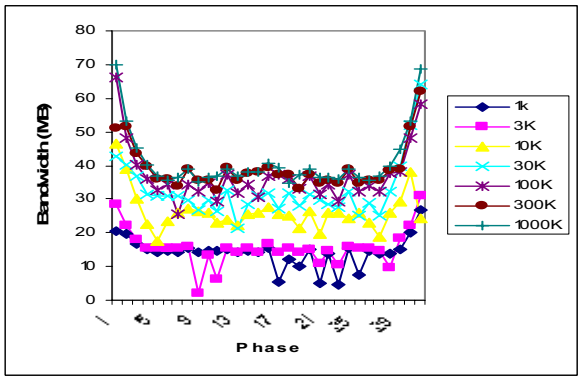

Figure 13. Simple pairwise on Blue (32 processes on 8 nodes)

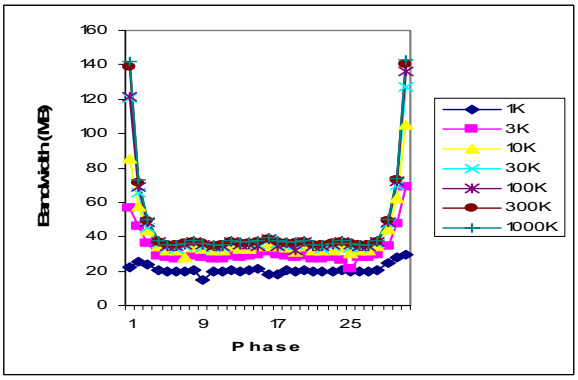

Figure 14. Simple pairwise on TC2K (32 processes on 8 nodes)

Figures 12 through 14 present the results of implementing algorithm 2 on eight nodes of Frost, Blue and TC2K using 32 processes. We noticed similar drop in bandwidth (as in algorithm 1) on Frost and Blue (about 40%) as the distance between the communicating nodes increases. On TC2K, we noticed even a larger drop (by a factor 3) as the distance between the communicating nodes increases. As for IBM machines, TC2K suffers from the same network interface limitation as there is only one network interface per node. We also observed more noises and outliers on Blue than on the other two machines.

Figures 15 and 16 present the results of implementing algorithm 3 on eight nodes of Frost and TC2K using 32 processes (similar results obtained on Blue). We noticed a small drop in bandwidth as the number of communicating nodes increases. Also, these drops happen only when the number of pairs increases from one to two for large messages.

Figure 17 presents the results of implementing algorithm 3 on four nodes of Frost using 64 processes. The results show a drop by a factor of more than two as the number of communicating pairs increases from one to eight for large messages. Beyond eight pairs, the bandwidth remains constant for all message sizes.

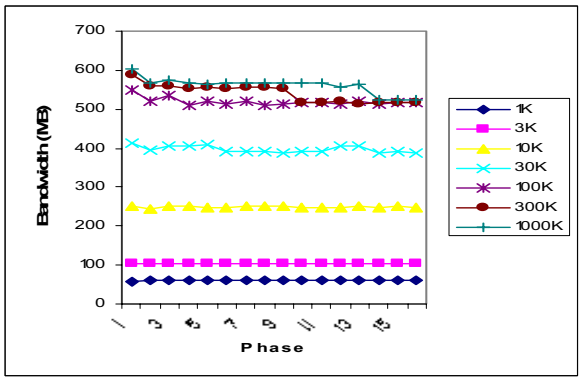

Figure 15. Cumulative pairwise on Frost (32 processes on 8 nodes)

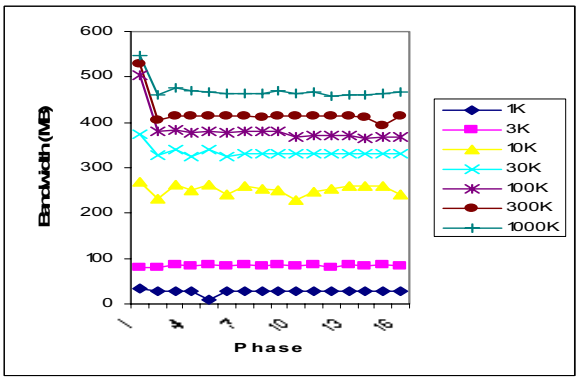

Figure 16. Cumulative pairwise on TC2K (32 processes on 8 nodes)

Finally, Figures 18 presents the results of implementing algorithms 1 on eight nodes of Frost using 128 processes (similar results obtained with algorithm 2). These results are similar to the results of 32 processes on four nodes but with a larger drop in bandwidth. This again shows the limitation of the IBM SP switch2 for a large number of processes and multiple processes per node.

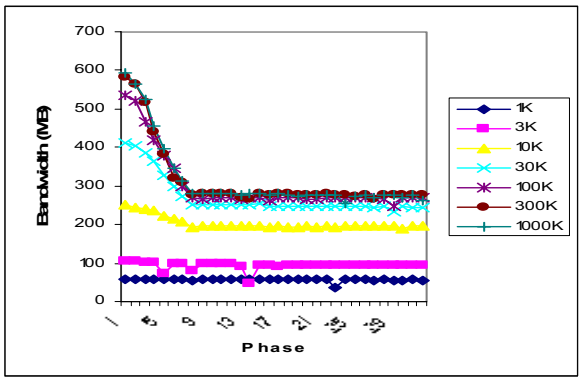

Figure 17. Cumulative pairwise on Frost (64 processes on 4 nodes)

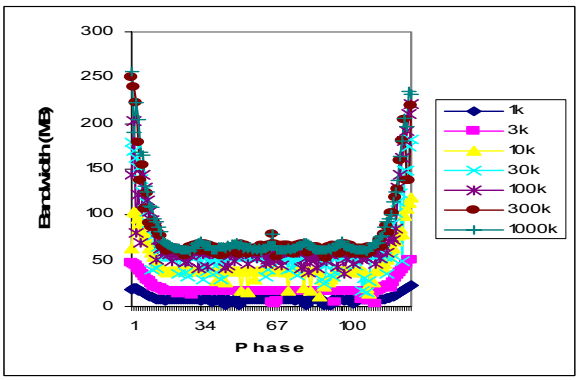

Figure 18. Congested-controlled AAPC on Frost (128 processes on 8 nodes)

5. Concluding Remarks and Future Work

In testing network bottlenecks between nodes (single process per node tests), we found that different communication patterns have no impact on the IBM SP switches and the Quadrics switch. However, the cumulative pairwise test shows a drop in bandwidth for the Quadrics switch as the number of pairs doubled.

In testing communication bottlenecks within an SMP (multiple processes on a single node tests), the cumulative pairwise tests showed the limitation of the SP switch2 on Frost where for large messages (over 10k bytes) the achieved bandwidth dropped by about one half as the number of communicating processes increased from two to eight. We also noticed a drop in bandwidth on both Blue and TC2K as the number of communicating processes doubled. This reflects the limitation of SMP machines mainly for large messages.

In testing communication bottlenecks of the whole system (multiple processes on multiple nodes tests), we noticed a drop in bandwidth on all tested platforms (Frost, Blue and TC2K) as the distance between communicating nodes increases. That drop is about 40% on IBM based switches and up to 300% on TC2K for 32 processes on eight nodes. The drop is even more significant as we increase the number of processes (128 processes on eight nodes of Frost). We also noticed a significant drop in bandwidth as the number of communicating processes increases from two to eight on Frost. The number of network interfaces per node might be the limiting factor since multiple processors of a single node share the interface bandwidth.

This is our first step in evaluating performance of overall network interconnects. By selecting specific dense communication patterns, setting up a procedure to evaluate SMP machines, and testing them on five different platforms, we hope that we can start a formal procedure to evaluate interconnects. Our methodology requires no prior knowledge of the network topology nor the mapping of processes onto processors.

We plan to test other systems and networks as well, such as Columbia, IBM ASC Purple and Blue Gene/L, Cray X1, and InfiniBand based clusters. We also plan to add more algorithms to our benchmarks. Moreover, we are testing some communication intensive applications and correlating their results with our communication-only tests.

6. References

[1] C. Bell, et al. *An Evaluation of Current High-Performance Networks*, International Parallel and Distributed Processing Symposium (IPDPS'03), 2003.

[2] A. Bland, et al. *Early Evaluation of the IBM p690,* Supercomputing 2002, 2002.

[3] D. Culler, et al. *LogP: Towards a realistic model of parallel computation.* Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pg 1–12, 1993.

[4] T. Dunigan, *ORNL Compaq Alpha/ IBM SP evaluation*, Oak Ridge National Lab, Oct. 2001.

[5] A. Grama, et al., *Introduction to Parallel Computing,* 2nd ed., Addison Wesley, 2003.

[6] IBM Corp., *IBM e server pSeries: SP Switch and SP Switch2 Performance, Ver. 8,* Feb. 2003.

[7] C. Kurmann, F. Rauch, & T. Stricker, *Cost/Performance Tradeoffs in Network Interconnects for Clusters of Commodity PCs,* Proc. workshop on Communication Architecture for Clusters, Nice, 2003.

[8] J. Liu, et al. *Micro-Benchmark Level Performance Comparison of High-Speed Cluster Interconnects,* Proc. 11th Symp. On High Performance Interconnects, Stanford, 2003.

[9] F. Petrini, et al., *The Qudarics Network (QsNet): High-Performance Clustering Technology,* IEEE Micro, 22 (1), pp. 46 – 57, 2002.