Network Performance Variability in NOW Clusters

Jeffrey J. Evans
Department of Electrical and
Computer Engineering Technology
Purdue University
401 N. Grant St.
West Lafayette, IN 47907
jje@purdue.edu

Cynthia S. Hood
Department of Computer Science
Illinois Institute of Technology
10W 31^{st} Street
Chicago, Illinois 60616
hood@iit.edu

Abstract

Performance management of clusters and Grids poses many challenges. Sharing large distributed sets of resources can provide efficiencies, but it also introduces complexity in terms of providing and maintaining adequate performance. Current application requirements focus on the amount of resources needed without explicitly characterizing the performance required from those resources. In clusters and Grids, inconsistent or highly variable application run-time is an indication of systemic inconsistency, with ramifications for those running the application and those managing the resources. We are focusing on the contribution of the interconnection network to application run-time variability. This work presents experimental results characterizing parallel application run-time sensitivity to communication performance variability using an Application Communication Emulator (ACE).

1. Introduction

Performance management of clusters and Grids poses many challenges. The sharing of a large distributed set of resources can provide efficiencies, but it also introduces complexity in terms of providing and maintaining adequate performance. Current application requirements focus on the amount of resources needed without explicitly characterizing the performance required from those resources. In this paper, we focus on communication performance and demonstrate a high variability. For some applications, this variability may have a significant negative impact.

The requirements of each application in terms of computation, communication, and storage are unique. Generally the application designer will specify the number of processors, along with an estimated duration of the job. As in many clusters, our work assumes that each processor will be used by a single job. Additionally, the network can be specified in clusters equipped with multiple interconnection systems.

High-performance interconnects are generally used by applications that have strict communication requirements. When the specified requirements are consistent, it is desirable that large *systems* of resources operate in a consistent manner. Inconsistent or highly variable run-time of applications is an indication of systemic inconsistency. This continues to be an issue [7, 8, 15] with implications for those running the application as well as those managing the resources.

We are focusing on the contribution of the interconnection network to application run-time variability. The long-term goal is to determine how to best manage network performance in clusters and Grids. To do this we must understand how much variability exists in communication performance, and then determine how this impacts the overall system performance and application run-times. The work in [6, 7, 8] provides motivation for further analysis into the application's perception of sensitivity to communication cost variability.

This work presents experimental results characterizing parallel application run-time sensitivity to communication performance variability. We introduce an *Application Communication Emulator* (ACE) intended to behave like a parallel application, emulating one or more applications concurrently in a controlled and repeatable manner. Because ACE operates from the perspective of the application it is independent of the interconnect system.

The remainder of this paper is organized as follows. Background on parallel application design and communication cost modeling is presented in section 2. Section 3 examines work related to application performance, scheduling, network performance and communication cost profiling. We then briefly introduce the ACE tool and present results from our preliminary evaluation in section 4. Finally, we summarize and target areas for future work in section 5.

2. Background

Parallel programs are distributed among many processes which are then distributed across several processors (compute nodes). Our work focuses on those where a one-to-one relationship of processes to processors exists. Communication between processes then reduces to communication between processors, requiring the services of a communications "stack" and an interconnection network.

The time for P processors to execute a program can be expressed as [9]

$$T = \frac{1}{P}\left(\sum_{i=0}^{P-1} T_{comp}^{i} + \sum_{i=0}^{P-1} T_{comm}^{i} + \sum_{i=0}^{P-1} T_{idle}^{i}\right) \quad (1)$$

where i is the individual process(or), T_{comp} is the compute time, T_{comm} corresponds to the communication time, and T_{idle} is the idle time.

The communication time T_{comm} is "the time it takes to send and receive messages" [9]. From the application's perspective, message communication can also be divided into components relating to CPU and memory system interaction (including the NIC) and network interaction.

The modeling of communication cost in parallel programs has been developed into a well-known linear 2-parameter model:

$$T_{msg} = \alpha + \beta n, \quad (2)$$

where α is the startup time, β is the transfer time of a unit of n data (bits, bytes, words, etc.). The startup time α is generally assumed to be independent of message size.

Competition for network resources is modeled as a scaling factor S on βn [9] representing the number of processors concurrently communicating. The scaling factor S is useful for single messages, but does not account for, and would be impractical to use as a scaling of the overall communication time of a parallel application. Moreover, the scaling factor does not account for the cost of network adaptation (re-routing). Therefore, where m is the number of messages, T_{comm} can be expressed as

$$T_{comm} = \sum_{i=0}^{m} T_{msg}^{i} = \sum_{i=0}^{m} \alpha + \beta n. \quad (3)$$

Other models have been developed extending the parameters of the linear model. In [24] and [23] a hyperbolic model is presented and evaluated. This model of the time t for a message size of m (bytes) follows a form bearing a strong resemblance to equation (2). The model is somewhat superior in the case of small messages, however the simple linear model of equation (2) is noted as superior for large messages [24].

3. Related Work

The majority of work in run-time variability examines a particular perspective in isolation of all others. From the perspective of the application, performance tuning and steering concepts have been attempted to aid in post-mortem application development and optimized execution. This generally involves instrumenting the application itself to study time perturbations [18], kernel [27] and program [14] tuning for maximum performance, and dynamic object management [28].

Centralized scheduler research attempts to address run-time variability in a proactive manner by trying to optimize node location while maintaining its operational objective of maximum node utilization. Most schedulers currently available however work on specific systems, and were not designed specifically for use on clusters [3]. Effort has been put forth investigating communication-aware [19], contiguous task mapping [15] and selective techniques [25]. The problems of requirements gathering, dissemination, and integration into a centralized scheduler still remain.

Work related to network performance has traditionally focused on raw bandwidth performance, and routing reachability and adaptability. The Network Systems Lab at IIT is also exploring routing reachability and adaptability in a hierarchical manner [1, 2]. Moving forward this work promises to integrate monitoring and routing in a more systemic and global context.

Research continues in network performance, congestion and traffic control, and adaptive routing. Most work attempts to optimiize the fine details of moving bits of information from point A to point B. In high performance networks this requires specialized fine-grained techniques such as sophisticated clocks [4] and faster sampling rates [22] of monitors.

Special nodes to temporarily off-load data during congestion periods [5] or marking of non-uniform traffic prior to entry into the network [16] have been proposed. Both techniques however require additional and modified hardware. Schemes such as overlays [21] and exchanges [26] have also been proposed. These however have the same effect of reducing network capacity or adding to the idle time of the process and application, potentially contributing to run-time variability of applications that are sensitive to these perturbations.

Communication performance from the perspective of message passing libraries has been studied. Tools have been developed which focus on measuring maximum performance [11, 13, 20]. The difficulties and pitfalls of obtaining reproducible measurements are discussed in [10].

There has been little work published that characterizes a parallel application's run-time sensitivity to network/communication performance. A study on the Intel

Paragon [30] is the closest work to our own in this regard. Two approaches were used to degrade network bandwidth. Messages were either exchanged multiple times or synthetic perturbations were applied to communications routines.

4. Experimental Framework and Results

To enable performance management of interconnection networks, it is necessary to understand the impact of network performance on the overall system performance. Our goal is to develop a way of characterizing application sensitivity to network performance, so the interconnection network can be managed appropriately. Sufficiently instrumenting a parallel application to get precise performance information can lead to overwhelming amounts of data as well as measurement bias. Such probing can adversely affect data movement (cache) efficiency, which can alter the original performance of the application itself. Therefore we propose an approach that minimizes instrumentation within the application itself.

4.1. Application Communication Emulation

Most communication benchmark programs attempt to quantify the maximum performance of the communication subsystem. Typically this is done by executing a number of communication exchanges in rapid succession, with no "compute" phase. Once completed, statistics based on collective hardware-based timing measurements are calculated and presented [11, 13]. Our method of characterizing variability which we call "application communication emulation" relies on a two phase approach motivated from the notion of compute / communicate cycles [9].

The first phase determines the communication cost for phase two by measuring message exchanges, then calculating communication time using the 2-parameter linear cost model previously described. While most benchmarking tools strive to determine maximum performance, the ACE benchmark calculates an average value based on all participating processes. During the performance measurement, processes communicate without any temporal spacing for computation (i.e. 100% communication load). Each process presents their perception of communication cost using linear interpolation to determine β and α.

These values are collected from each process, averaged, and redistributed prior to the start of phase two. This way all processes use the same basis for determining their compute time. This approach is consistent with the use of single (global) values for α and β, the fundamental basis for developing communication cost when designing parallel programs [9].

The per-cycle compute time is derived based on the communication time and the user-selectable communication "load", which is then used to compute an estimated overall "run"-time of the emulated application. Working from our definition of parallel run-time (equation (1)), an emulation *run*-time is

$$T_{run} = \sum_{i=0}^{c-1} T_{comp} + \sum_{i=0}^{c-1} T_{comm} \tag{4}$$

where c is the number of compute/communicate cycles.

The second phase performs communication exchanges in a manner that "emulates" a real parallel application "run" (i.e. a series of compute/ communicate "cycles"). This phase also measures the actual compute and communication times during the run. The resulting "Application Communication Emulator" (ACE) system performs communication exchanges in a manner that emulates a set of parallel applications, hence its name. ACE uses the Message Passing Interface [12] for message passing and acquisition of per process timing information.

One or more *emulated applications* (EAs) are created using MPI *communicators*. Each EA acts independently per user input in terms of communication exchange type (synchronous, asynchronous) and pattern (point-to-point, circular, cartesian, etc.). Users also provide the number of messages (compute/communicate iterations), a range of message sizes, and a communication load (in percent from 1 to 100). Users can also elect to ignore running phase one, using default values for α and β communication cost coefficients based on workstation parameters illustrated in [9].

The computation phase of each *cycle* (equation (5)) consists of "busy waiting" for a predetermined period of time, as defined by the communication time and load. By performing the computation using a hardware-based timed loop, little to no data is moved through the memory system of the workstation, drastically reducing compute phase variability. Compute phase statistics are also collected for validation purposes. Any variation in run-time should therefore be attributable to variations in communication time. [1] The compute time is derived from the notion that the compute/communicate cycle time is simply

$$T_{cycle} = T_{comp} + T_{comm} \tag{5}$$

The tool attempts to emulate actual applications, so the values of T_{comp} and T_{cycle} must be derived. Our approach is from the perspective of communication load, or the percentage of time an application spends communicating. As such we first must determine the expected communication

1 The exception is in the event of a system administrator or inadvertent user gaining access to a compute node and performing operations in addition to those of the emulator.

time, then calculate the compute time. Using the measured communication time T_{comm} and the percentage of the total time that processes communicate L, we calculate the compute time as

$$T_{comp} = \frac{T_{comm} * 100}{L} - T_{comm}. \quad (6)$$

Memory buffer allocation for communication is performed "apiori" to eliminate undesired memory exchanges during a run. It is also noteworthy to mention that the overall run-time is strictly bound to the sequence of compute/communicate cycles of the slowest EA (MPI communicator). Each process keeps track of its own runtime, communication, compute, and barrier (synchronization) time statistics in simple (small) data structures. The collection of these data at the completion of a run and subsequent statistical calculations are not included in the emulated run-time.

4.2. Cluster Testbeds

Our preliminary evaluations of the ACE framework were performed on two different clusters. Both clusters operate on the premise that when a node allocation is made, the user "owns" the node, meaning that no other users can get to the nodes to otherwise disturb them.

Chiba City is a 512 CPU Linux cluster located at Argonne National Laboratory [17]. It is comprised of 256 Dual Pentium III 500MHz systems, each with 512MB of RAM and 9GB of local disk space, configured into 8 computing *"towns"*. There are also storage, visualization, and cluster management towns, each equipped with added functional capacity. Multiple interconnection networks are available for use in the cluster. All systems in the cluster are interconnected with a high performance 64bit Myrinet switched network. Additionally, an Ethernet is used for file service and management functions. All compute and visualization nodes are interconnected using fast (100Mbps) Ethernet while all other nodes are interconnected with switched gigabit Ethernet. Our preliminary evaluations were performed on both (Myrinet and Fast Ethernet) interconnect systems.

To accommodate users who otherwise would not be able to use the various supercomputers and task-specific clusters on the Purdue University campus, machines have been "recycled" from instructional labs as they are replaced with newer models. These recycled machines have been made into a nearly 1,000 node Linux cluster. The machines range in capability, but multiple groups of 48 nodes are matched by processor speed, memory, and storage capability. We used two groups of 48 nodes whose machines are 450MHz Pentium II's, each with 256MB of RAM and 9GB of local disk storage. The interconnect is Fast (100Mbps) Ethernet with Gigabit Ethernet aggregation links across the (nearly) flat topology.

4.3. Preliminary Evaluations

Our overall goal is to use ACE as a means to characterize actual parallel application codes for their sensitivity to network performance. For this work we focus on testing the functionality of ACE in isolation, in (nearly) completely benign conditions. Message sizes of 1KB (1024 bytes) and 1MB (1048576 bytes) were used, which by most definitions falls into the category of large messages.

Communication load, or more precisely the percentage of time a parallel program spends communicating, depends on the application. Some parallel applications communicate less than one percent of the time. Other parallel programs [29, 30] can communicate over fifty percent of the time. As such we have chosen to study communication load in the range of ten to eighty percent. The range of node allocation sizes is from 4 to 32. When two EAs are used the size of each EA is 1/2 of the entire node allocation.

We begin by evaluating simple point-to-point communications, performing either blocking or non-blocking exchanges between pairs of nodes using a single EA. The predicted run-time is derived from the communication time as measured by the benchmark component (phase one) of the framework just prior to the run. The emulated compute time is calculated from the average communication time of all the node pairs and the communication load. Once T_{comm} and T_{comp} are established we can use the number of iterations (cycles) to determine the predicted run-time. The run-time error is then the difference between the predicted and actual (measured) run-time.

It is important to note that for all tests the runs for each communication load with each node allocation size were performed in sequence. This means that tests performed using 1KB messages and subsequent 1MB messages, used the same nodes. Additionally, a series of tests (one for each communication load) was conducted on each message size. Thus an entire suite of tests, for example 16 nodes, 1KB and 1MB messages, each running individual communication load tests from 10 to 80%, used the same node set.

Equally important, a new benchmark (phase one) was performed for each test point (an emulated application run at a particular communication load - phase two). Thus new communication parameter data α and β is acquired and new calculations performed for communicate, compute, and overall run-time for each run. This is important as it provides a direct measure of the most up-to-date communication (network) performance.

After each compute/communicate cycle (equation (5)), all processes perform a
MPI_Barrier() operation to ensure that the next cycle begins in synchronization. This is especially important with collective (Alltoall) communication. In order to provide confidence in our measurements and predictions, statistics are

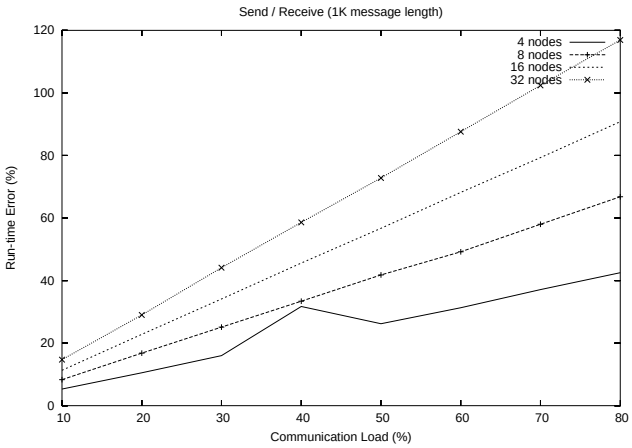

Figure 1: Run-Time Prediction Error - Blocking Send(1K)

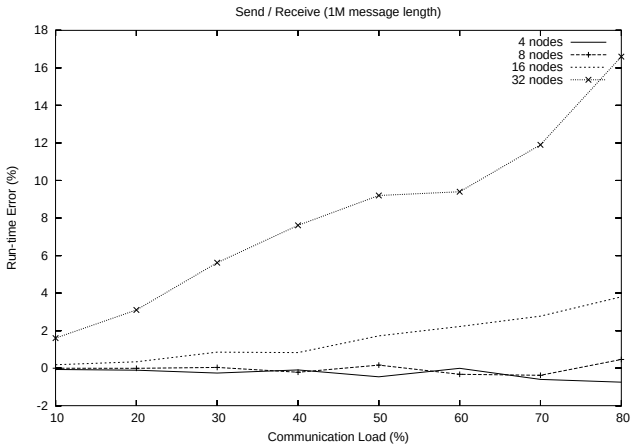

Figure 2: Run-Time Prediction Error - Blocking Send (1M)

maintained for all aspects of the run which includes communication, the mock computation, and the MPI_Barrier() time.

Figures 1 and 2 show the run-time error on the y-axis (%) vs. communication load (%) for an emulated application that performs point-to-point blocking send / receive exchanges. For these tests, 10,000 iterations were used for 1KB messages and 1,000 iterations for the 1MB message size. In the interest of space we present results from tests performed on the Chiba City cluster using the Myrinet interconnect. For these tests, overall run times ranged from a few seconds to several minutes per test. The same tests using the Ethernet interconnect exhibited run-times ranging from minutes to over 24 hours.

The reason for this large range of run-times should be clear. The compute time T_{comp} for each cycle is derived based on the measured communication time T_{comm} and the communication load. Since the communication time is constant, T_{comp} and the cycle time T_{cycle} will necessarily grow as the load shrinks (i.e. a run using a 10% communication load will take far longer than a run specifying a 80% load).

Both figures 1 and 2 suggest that the error in run-time prediction grows as a function of both the number of nodes and the communication load.

Figure 3 illustrates how ACE experiences and can expose network performance variability. The 1M message size set of runs captured in figure 3 exhibits a run-time prediction error of nearly two orders of magnitude. Investigating the run-time logs of ACE we discovered that in this case the two sets of experiments used slightly different sets of nodes. However while the sets were different, they both contained over half of their nodes from the same set. In other words, over half of the nodes in the two node allocations were the same. This led to investigating the logs for differences such as compute or communication times.

First we examined compute time statistics noting that

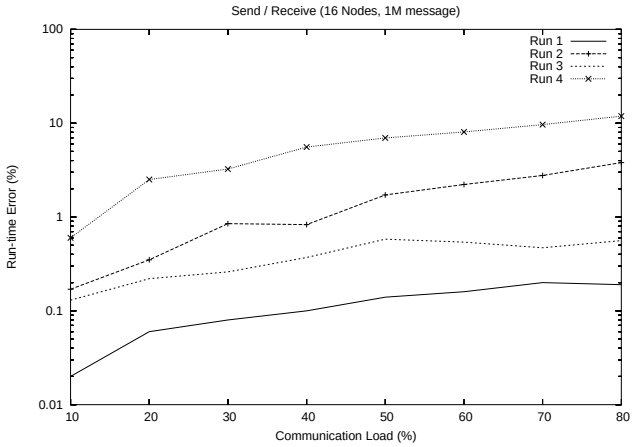

Figure 3: Blocking Send Run-Time Variability (16 Nodes - 1M)

they were nearly identical (within 1% of calculated). Their standard deviations, on the order of 0.005% is what we would expect from the ACE compute module. Next we examined the values of β and α produced from the pre-run benchmark test in order to verify the computed communication times for each run. We note that run 3 had $\beta = 0.01057\mu s/B$ while run 4 $\beta = 0.01097\mu s/B$, a 3 percent difference. The values of α were also quite close with run $3 = 75.9\mu s$ and run $4 = 80.3\mu s$. These values of β and α, when used to compute T_{comm} produce values of 22.336ms and 23.173ms for runs 3 and 4 respectively. These are within 5% of the run-time mean communication time values observed in Table 1. This is significant as it suggests that the run-time prediction error between the runs should also be on the order of 3 percent, not the order of magnitude that was observed.

Table 1 documents additional statistics of these runs. We note striking differences in the communication time statis-

tics. First we note that the mean values are within $\pm 3\%$. However the standard deviations are different by a factor of nearly 70. Examining the barrier time statistics we see similar differences between the means (a factor of 6) and standard deviations (a factor of 3). These differences however still dwarf those of the communication times during the runs. We attribute these anomalies to network performance since the machines themselves are of identical capability (CPU, memory, OS, libraries, etc.).

Table 1: Blocking Exchange Run-Time Variability (1MB Message)

Run	3	4
$T_{comm}\ \bar{x}$	23.3ms	24.0ms
$T_{comm}\ \sigma$	$38.4\mu s$	$2.625ms$
$T_{comm}\ \sigma^2$	$1.474ns$	$6.9\mu s$
$T_{barr}\ \bar{x}$	$526\mu s$	$3.25ms$
$T_{barr}\ \sigma$	$908\mu s$	$2.70ms$
$T_{barr}\ \sigma^2$	$824ns$	$7.2\mu s$

4.4. Collective Communications and Multiple EAs

Our preliminary experiments confirmed our objective to provide tight control over compute time as derived from communication benchmark measurements. They also exposed highly variable emulated application run-time prediction errors as well as many cases where the prediction error was nearly zero. Our next series of experiments focuses on the addition of two capabilities; collective communications exchanges and multiple EAs.

Collective operations are used frequently in parallel programs such as those solving computational fluid dynamics and N-body particle problems. These operations perform scatter, gather, reduction, and all to all (Alltoall) communications among processes. The ACE framework allows for the implementation of any MPI operation using any exchange mechanism. Here we focused on examining the Alltoall collective operation and the creation of two EAs running independently. EA combinations included blocking send/receive with Alltoall, non-blocking send with Alltoall, and Alltoall with Alltoall. The same test parameters used to characterize operations on a single EA were replicated for the 2-EA experiments.

As with the single EA testing, multiple EAs can run a benchmark of the pattern and exchange semantics they will use for their run. The run does not begin in earnest until all EAs have completed their benchmarks. The benchmarks are run independently by each EA and in parallel, which intuitively increases the likelihood of some amount of run-time prediction error due to communication variability. We are currently evaluating the effects of this approach to determine if it is more or less advantageous to perform multiple EA benchmarks in parallel.

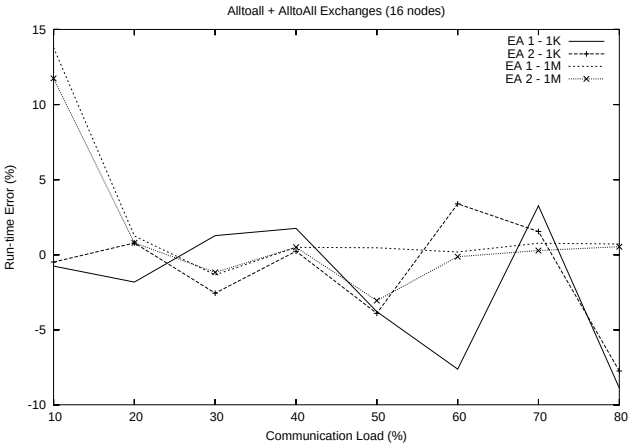

Figure 4: Collective Exchange + Collective Exchange (16 Nodes 1K - 1M)

Figure 4 is one example that captures run-time variability behavior of each of two EAs, using 16 total nodes, both executing collective (Alltoall) message exchanges. We see significant variability in this operational scenario, even at light communication load.

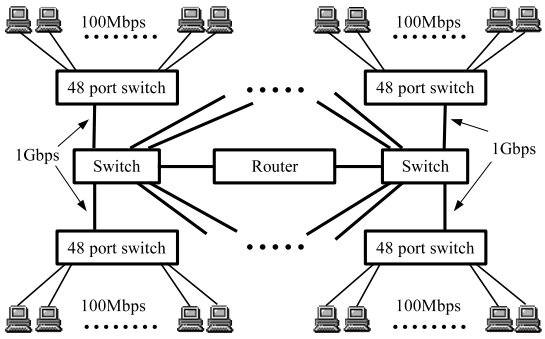

Figure 5: Purdue Linux Cluster (Partial Network Topology)

Figure 5 illustrates a partial network topology of the Purdue University Linux Cluster. The scheduler on this cluster allows the user to specify which cluster group(s) to use for a job. This is done so users can select sets of nodes whose capabilities are matched for CPU, memory, etc. when job requests exceed 48 (the primary grouping). We ran our tool with the same configuration settings used on Chiba City. For multiple EA runs however, we selected cases where both

EAs were associated with the same group (switch) and others where two groups were "spanned". In some cases we chose the span for the greatest number of hops, namely those that must go through the router.

Contrary to intuition, we observed runs where run-time variability was greater on runs where the nodes were connected to a single switch. In some cases of heavy communication load, runs that spanned switches and even those that traversed the router performed more consistently. We speculate that since most users try to limit their jobs to less than 48 nodes traffic across switches and the router is minimal. Therefore in situations where communication load is high, we speculate that the local switch buffers may be heavily loaded, causing communication variability.

5. Summary and Future Work

We are studying run-time variability at the communication level from the perspective of the application focusing on the network (and tightly coupled communication library). This work attempts to gain insight into this relationship, as motivated by our previous work [7, 8].

We have designed an approach and framework for examining network and communication performance impacts on the run-time of parallel applications. The ACE framework provides this capability by executing one ore more "emulated parallel applications" (EAs) by taking advantage of MPI communicators.

Testing and validation activities on one experimental and one production NOW cluster has produced insightful results. Predicting run-time performance accurately and consistently remains an issue, even when the compute time of an EA is tightly controlled. We have shown the usefulness of capturing dispersion statistics in order to gain insight into run-time variability when benchmarked communication coefficients and average communication times are nearly identical, implying run-time consistency.

This work also illustrates that measuring communication performance only moments prior to program execution provides no guarantee that performance will not degrade (or improve) during the actual running of the application. Intuitively, static communications cost models that do not account for the performance variability aspects of the network subsystem can result in increasing application runtime variability as the number of compute/communicate cycles, nodes, or communication load grows. For an actual parallel application this imprecision manifests itself as a variability in the ratio of computation to communication.

We are currently using the ACE framework as a means to characterize other parallel applications in terms of their sensitivity to network performance. Without instrumenting the application we hope to quantify sensitivity based on the combination of the run-time of the application and measured dispersion statistics collected by ACE. Quantifying sensitivity in this manner has the potential for more appropriate resource allocation and management. Having the ability to characterize acceptable network performance will facilitate system adaptation in the face of unacceptable performance. To remedy poor network performance, the network may adapt it's routes or the application may be migrated to another part of the cluster with better network performance.

In systems such as clusters and Grids, central control elements such as schedulers and resource managers know which applications are running and where. Network management systems however do not have this knowledge. The communication performance ramifications of the virtual communication topology created by the scheduler are generally unknown to any of the subsystems. Reactive operations such as application or network adaptation can be counter-productive to the system as a whole. Ultimately we hope to establish simple operating parameters that can be used in a cooperative way among cluster (and Grid) subsystems to maintain systemic operating and performance consistency.

6. Acknowledgements

This work was supported in part by the U.S. Department of Energy, under Contract W-31-109-Eng-38 and NSF 9984811. The authors also wish to thank the research computing department of Information Technology at Purdue (ITaP) for their valuable insights and use of their Linux clusters.

References

[1] S. Baik and C. Hood. Decentralized route generation method for adaptive source routing in system area networks. In *The 8th World Multi-Conference on Systemics, Cybernetics and Informatics (To appear*, July 2004.

[2] S. Baik, C. Hood, and W. Gropp. Prototype of AM3: Active Mapper and Monitoring Module for the Myrinet Environment. In *Proceedings of the HSLN Workshop*, Nov. 2002.

[3] B. Bode, D. Halstead, R. Kendall, and Z. Lei. The Portable Batch Scheduler and the Maui Scheduler on linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[4] S. Chakravarthi, A. Pillai, J. Padmanabhan, M. Apte, and A. Skjellum. A fine-grain synchronization mechanism for QoS based communication on Myrinet. Submitted to the International Conference on Distributed Computing, 2001, 2001.

[5] S. Coll, J. Flich, M. Malumbres, P. Lopez, J. Duato, and F. Mora. A first implementation of in-transit buffers on myrinet gm software. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 1640–1647, 2001.

[6] J. J. Evans, S. Baik, C. S. Hood, and W. Gropp. Toward understanding soft faults in high performance cluster networks. In *Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management*, pages 117–120, March 2003.

[7] J. J. Evans, S. Baik, J. Kroculick, and C. S. Hood. Network Adapability in Clusters and Grids. In *Proceedings from the Conference on Advances in Internet Technologies and Applications (CAITA)*. IPSI, July 2004.

[8] J. J. Evans, C. S. Hood, and W. D. Gropp. Exploring the relationship between parallel application run-time variability and network performance in clusters. In *Workshop on High Speed Local Networks (HSLN) from the Proceedings of the 28th IEEE Conference on Local Computer Networks (LCN)*, pages 538–547, October 2003.

[9] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Company, 1995.

[10] W. Gropp. An introduction to performance debugging for parallel computers (mcs-p500-0295). In *Proc. of the ICASE/LaRC Workshop on Parallel Numerical Algorithms*, to appear. ftp://info.mcs.anl.gov/pub/tech_reports/reports/P500.ps.Z.

[11] W. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. Technical Report ANL/MCS-P755-0699, Argonne National Laboratory, 1999.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 2nd edition, 1999.

[13] D. Grove and P. Coddington. Precise MPI performance measurement using MPIBench. Technical report, Adelaide University, Adelaide SA 5005, Australia, 2001.

[14] W. Gu, G. Eisenhauer, and K. Schwan. Falcon: On-line monitoring and steering of parallel programs. In *Ninth International Conference on Parallel and Distributed Computing and Systems (PDCS'97)*, Oct. 1997.

[15] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the $m$aui scheduler. In *7th Workshop on Job Scheduling Strategies for Parallel Processing*. SIGMETRICS 2001, ACM, June 2001.

[16] M. Jurczyk. Traffi c control in wormhole-routing multistage interconnection networks. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, volume 1, pages 157–162, 2000.

[17] A. N. Laboratory. Chiba City, the Argonne scalable cluster. Online Document, 1999. http://www-unix.mcs.anl.gov/chiba/.

[18] C. Mendes and D. Reed. Performance stability and prediction. In *IEEE International Workshop on High Performance Computing (WHPC'94)*, March 1994.

[19] J. M. Orduna, F. Silla, and J. Duato. A new task mapping technique for communication-aware scheduling strategies. In *International Conference on Parallel Processing Workshops*, pages 349–354, 2001.

[20] R. Reussner, P. Sanders, L. Prechelt, and M. Muller. SKaMPI: a detailed, accurate MPI benchmark. In *PVM/MPI*, pages 52–59, 1998.

[21] F. P. S. Coll, E. Frachtenberg and L. G. A. Hoisie. Using multirail networks in high-performance clusters. In *Proceedings of the 2001 IEEE International Conference on Cluster Computing*, pages 15–24, 2001.

[22] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 39–46, September 2002.

[23] I. Stoica, F. Sultan, and D. Keyes. Evaluating the hyperbolic model on a variety of architectures. Technical Report TR-96-34, ICASE, NASA Langley Research Center, 1996.

[24] I. Stoica, F. Sultan, and D. Keyes. A hyperbolic model for communication in layered parallel processing environments. *Journal of Parallel and Distributed Computing*, 39(1):29–45, 1996.

[25] V. Subramani, R. Kettimuthu, S. Srinivasan, and J. Johnston. Selective buddy allocation for scheduling parallel jobs on clusters. In *Proceedings of the International Conference on Cluster Computing*, pages 107–116, September 2002.

[26] A. T. C. Tam and C. L. Wang. Contention-free complete exchange algorithm on clusters. In *Proceedings of the IEEE International Conference on Clusters*, pages 57–64, Nov.-Dec 2000.

[27] A. Tamches and B. P. Miller. Using dynamic kernel instrumentation for kernal and application tuning. *International Journal of High-Performance Computing Applcations*, 13(3):263–276, Fall 1999.

[28] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings from the Conference on High Performance Networking and Computing*, 2002.

[29] University of California, San Diego. Enzo - cosmological simulation code. Online Document, 2003. http://cosmos.ucsd.edu/enzo/.

[30] P. H. Worley, A. C. Robinson, D. R. Mackay, and E. J. Barragy. A study of application sensitivity to variation in message passing latency and bandwitdh. In *Concurrency: Practice and Experience*, volume 10, pages 387–406. John Wiley & Sons, April 1998.