Exploring the Relationship Between Parallel Application Run-Time Variability and Network Performance in Clusters

Jeffrey J. Evans
Department of Electrical and
Computer Engineering Technology
Purdue University
1415 Knoy Hall
West Lafayette, IN 47907
jjevans@tech.purdue.edu

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

William Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois 60439
gropp@mcs.anl.gov

Abstract

Highly variable parallel application execution time is a persistent issue in cluster computing environments, and can be particularly acute in systems composed of Networks of Workstations (NOWs). We are looking at this issue in terms of consistency. In particular, we are focusing on network performance and using techniques from fault management to attain consistency. Our analysis of run-time variability from logs and experiments expose important issues related to systemic inconsistency in NOW clusters. The characterization of application sensitivity can be used to set network performance goals, thereby defining operational requirements. Network performance depends on the virtual topology imposed by the scheduler's allocation of nodes and the communication patterns of the set of running applcations. Therefore it is important to look at both the network and the cluster's centralized node mapper (scheduler) as critical subsystems.

1. Introduction

A central idea in cluster computing is that the entire computing environment (the applications, the resources they use, and any centralized allocation and control entities) must be thought of as a *"system"*. Moreover, it is desirable that the system operate in a consistent manner. Inconsistent, or highly variable run-time of applications executing on clusters continues to be an issue, particularly on systems composed of networks of workstations (NOWs) [1]. Highly variable run-time carries both technological and economic ramifications.

Parallel applications running on clusters are generally submitted as batch jobs. Users request a number of nodes from a centralized scheduler/resource manager that allocates nodes based upon centralized access policies and queueing disciplines. Variability in application run-time is often attributed to variations in the input data set or the design of the application itself. When the input data set remains constant, highly variable run-time performance forces the developer or user to "pad" estimates of run-time in order to ensure completion of the job [18]. This approach of padding run-time estimates may work to undermine the efficiency of the scheduler due to large numbers of job terminations at unexpected times.

Computing using clusters is rapidly evolving from the laboratory to the business environment. A ramification of job padding is the potential of unused (idle) nodes and the resulting loss of revenue. Additionally, continued refinement of applications to achieve "optimum" performance results in increased development cost and lack of portability. As this evolution progresses, it will quickly become unacceptable for providers of cluster resources to deliver inconsistent performance of those resources while reaping finan-

cial benefits of extended application run-times. Moreover, it will also become unacceptable for providers not to control the performance of their resources or even explain why inconsistency is occurring relative to them.

We are focusing on the contribution of the interconnection network to application run-time variability in clusters of NOWs. The network in these systems has no direct control over the virtual topology imposed by the scheduler at run-time. This topology along with the communication patterns of each application running on the cluster at any given time complicates the efficient transfer of information (messages) between nodes. Moreover, the idea of network adaptability is also further complicated since adaptive routing schemes do not take into account other cluster subsystems, and therefore the ramifications of network adaptation.

We present an analysis of issues related to parallel application run-time variability from logs and experiments on the Chiba City Linux NOW cluster located at Argonne National Laboratory [21]. We show that inconsistent application performance is not limited to the application itself. Other cluster subsystems, such as the interconnection network and scheduler may also contribute to overall system performance degradation. This systemic performance degradation ultimately degrades the performance of one or more parallel applications simultaneously executing on the cluster.

The remainder of this paper is organized as follows. In section 2, we present background information on parallel application design with emphasis on modeling communication cost. Section 3 examines work related to various aspects of performance variability in high performance computing. We then present our preliminary experiments and results in section 4. Finally, we summarize our thoughts and target areas for future work in section 5.

2. Background

Parallel programs are distributed among many processes which are then distributed across several compute nodes. Several processes may be allocated to a single compute node but in many cases a one-to-one relationship is established of processes to processors. Communication between processes then reduces to communication between processors, requiring the services of the interconnection network. Several parallel applications execute simultaneously in NOW clusters. Therefore there are several factors to consider and balance to ensure that applications do not excessively "interfere" with each other.

Perhaps the most important area of consideration is to examine how parallel programs are analyzed, modeled and designed. Traditionally parallel programs (applications) have been developed with a single goal - maximum performance. In this case maximum performance is defined as

Definition 2.1 ***Maximum Performance.*** *Solving (or at least terminating) a computational problem in the smallest amount of real (wall clock) time.*

2.1. Parallel Application Design

A methodology developed in [11] commonly used to analyze, model, and design parallel programs (sometimes referred to as "*codes*"), follow four distinct yet interdependent stages:

1. *Partitioning*. The decomposition of the computation into small tasks. The intent is to expose opportunities for parallel execution. A good partition divides the problem into small pieces of *computation* and *data*. The ordered consideration of data, then computation is commonly known as *domain decomposition*. Alternatively, *functional decomposition* reverses the order of consideration.

2. *Communication*. Defining the communication required to coordinate task execution.

3. *Agglomeration*. Once the task and communication structures are defined, a performance evaluation is done with respect to requirements and implementation costs. At this point tasks may be combined into larger tasks to improve performance or reduce development costs.

4. *Mapping*. Finally, each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximum processor utilization with minimum communication cost. Mapping is often determined at run-time by a centralized *scheduler* with the assistance of specialized *resource managers* and/or *load-balancing* algorithms.

Clearly, perfect balance of these goals is not always possible or desired. Other factors, such as development cost and portability often influence this balance. At one extreme (minimum task size, maximum number of tasks) interprocess(or) maximum communication may be implied, although the size of the data (message size) is reduced or even minimized. Conversely, more agglomeration may result in fewer interprocess(or) communications but the size of the data (message size) may grow as a result.

The mapping process is clearly a critical aspect of achieving maximum performance. The static mapping of tasks is useful for those codes that decompose nicely into structured communication formations, taking advantage of certain direct interconnection topologies, such as trees, grids, or meshes. When communication patterns become unstructured static mapping is less useful. Moreover, when parallel applications execute on NOW's tasks (nodes) are

generally mapped dynamically using centralized schedulers and resource managers. The mapping aspect of parallel program design then becomes disjoint from the other design aspects.

2.2. Parallel Application Communication Modeling

The concept of modeling communication cost in parallel programs is developed in [11] and [13] with a linear 2-parameter model:

$$T_{msg} = s + rn, \tag{1}$$

where s is the startup time, r is the transfer time of a unit of data (bits, bytes, words, etc.) and n is the number of units being transferred. The startup time s is generally assumed to be independent of message size.

When there is competition for network resources a scaling factor S is introduced to begin to account for the number of processors needing to send concurrently over the same wire

$$T_{msg} = s + rnS. \tag{2}$$

Equation 2 does not account for the cost of contention, whether it be for message retransmission due to collision or the cost of network adaptation (re-routing). Parallel applications such as finite difference (commonly used in atmospheric modeling) and many other single program multiple data (SPMD) problems operate synchronously. This means processors tend to send and receive messages at approximately the same time. In these applications processes cannot proceed until their messages have been received. The impact of prolonged or continued contention for these applications is a gradual skewing of application synchronization, resulting in unpredictable run-time. Other problems such as searches and matrix construction algorithms operate asynchronously. Therefore it is assumed that processors executing this type of application will seldom compete for network bandwidth.

The scaling factor S can be useful if the parallel application is the only one running on the cluster. If however, more than one application is executing simultaneously (a common practice on NOW's) it becomes apparent that modeling communication performance in this manner for the purpose of performance estimation may be less effective. Moreover, we have not yet accounted for *where* the tasks are mapped within the cluster.

There has been considerable effort put forth in achieving maximum performance of network hardware in terms of raw bandwidth. Moreover, several techniques have been developed at the sub-application level to achieve communication abstraction and latency improvement. These include special message passing libraries such as MPI [15] and PVM [12]. Additionally, techniques have been developed to all but eliminate kernel intervention of the communication process with so-called "zero copy" mechanisms.

3. Related Work

The issue of run-time variability has been studied from several different perspectives. From the perspective of the application, performance tuning and so-called "steering" concepts have been attempted to aid in post-mortem application development and optimized (tuned) execution. Research in centralized schedulers has attempted to address run-time variability in a proactive manner by trying to optimize the location of nodes while maintaining the scheduler's overall objective of maximum node utilization. Work related to network performance has traditionally focused on raw bandwidth performance, reachability, and adaptability. We are looking at the problem at the "communication" level from the perspectives of the network and application.

3.1. Application Performance Management

Several areas related to performance management have been studied in the context of (wide area) Grids and (local area) clusters. Work in the area of parallel program application performance prediction such as [26] provide methods and assessment of application program stability by instrumenting the application code and stimulating it using a time perturbation technique and creating program execution graphs.

Program tuning [35] and steering [17], [38], [37] is an area that shows promise toward adapting parallel applications to run more efficiently (maximizing performance) during program execution. Simply, *"program steering permits users to control program execution in terms of program abstractions familiar to them"* [37]. The concept does however require learning a toolkit (middleware), then instrumenting the application with steerable objects. A limitation as noted by the authors is that some programs are more steerable than others. Before a program can be tuned or steered it is necessary for the parallel program to communicate "what" can be tuned or steered, which is considered in [20]. As part of the Active Harmony [36] software architecture for computational object management in dynamic environments, it is intended to allow applications to advertise tuning options to a higher level decision making entity.

3.2. Scheduling and Resource Management

The function of a scheduler in a cluster environment is to allocate (map) nodes to jobs. So we say that the scope of the scheduler is system wide. For example, a cluster scheduler generally is responsible for allocating nodes within a cluster

to jobs. Closely related to schedulers are resource managers whose focus is component related. What we mean is that a resource manager is more concerned with individual components, such as a single compute node or a portion of an interconnection network. For instance, a resource manager typically handles activities such as moving a parallel program's application code to the nodes, then beginning the execution of the program. Overall, solving the problem of efficiently scheduling groups of tasks to a set of machines forming a system is *NP-complete* [28], [9], [23].

The objectives of each subsystem are clear. For cost effectiveness, the scheduler must always try to fully utilize the nodes within the system. The objective of a resource manager is to utilize each system component effectively. An example of this might be to match compute node capability in a cluster with high performance network resources with an application that can most benefit from these resources. Interestingly, most schedulers available today work well only on certain systems and were never designed specifically for use on clusters [4].

In cluster environments, distributed techniques such as [9] propose local scheduling based on coscheduling techniques. Coscheduling attempts to ensure that no process will wait for a nonscheduled process for synchronization or communication and will minimize waiting time at synchronization points. This work focuses on issues related strictly with the nodes, such as its architecture, and fails to address issues related to the interconnection network explicitly.

Work related to centralized cluster scheduling include communication-aware task mapping strategies [28], contiguous allocation strategies with [18], [22] or without aggressive backfilling [24], [32]. At this time however there is no evidence of combining the gathering of communication requirements or the converse (application run-time sensitivity to communication performance) with the integration of centralized scheduling and resource management systems. In [28], network awareness is achieved by understanding application communication requirements, either by measurement or estimation. Additionally, the available network resources must be characterized and a criterion must be developed to determine the suitability of network resource allocation to each application allocated. A mapping technique is developed based on these requirements but there is no mention on how these requirements are actually gathered or how the resulting mapping would be integrated with a centralized scheduler.

3.3. Network Performance and Adaptability

Research continues in the areas related to high performance network performance, contention, congestion and traffic control and adaptive routing. The vast majority of work attempts to understand 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 [5] and faster sampling rates [31].

Other work such as [25] studies the effects of latency, overhead, and bandwidth in cluster architectures in the context of the LogP model [7], where it is noted that the ability to predict network performance in the face of communication imbalance (burstiness) is difficult at best. Another ongoing research project [2] involves the combination of network monitoring and adaptability which could be applied to our work. The Active Mapper and Monitor for Myrinet (AM3) uses a hierarchical approach to route discovery, traffic monitoring, and adaptability of a high performance Myrinet network.

An area closely related to network performance and adaptability is that of congestion and traffic control. In one case [6], special nodes are used to remove data from the network temporarily. Another approach is to handle nonuniform traffic that is known *a priori* differently from uniform traffic by marking the traffic prior to entry into the network [19]. This approach does require modifications to current hardware.

Other approaches that attempt to address high performance network congestion include using what are commonly called overlays, in this case called multirail networks [30] and exchange schemes [34] that take into account the network topology to compute a communication schedule that is a contention free complete exchange of information in the cluster. Yet another mechanism proposed in [3] assumes the existence of virtual channels and estimates network traffic locally based on a percentage of free virtual channels available.

All of the above work considers network performance, but not overall system performance. A lack of systemic consideration exposes the potential for interactions that may contribute to highly variable parallel application execution time. This is due to the fact that some parallel applications are sensitive to excessive or continued time "perturbations" in their execution [26]. One particularly nagging question arises that when the network is experiencing congestion, *how does the network know that adapting in a particular way will not induce more network congestion?*

3.4. Network Performance Variability

In [8], the topic of hot spot contention is explored in shared-memory multiprocessor systems. In this context, hot spot contention is described as a phenomenon whereby several processors request access to a critical data structure, or more generally to a single critical memory module, independent of data structure. Once hot spot contention begins, interconnection points (network buffers, links) also become

congested. This leads to a ”tree-saturation” phenomenon in multistage interconnection networks. Since some or all processors are connected to a subset of these switches, hot-spot congestion can effect normal (non hot-spot) traffic, degrading performance of all applications involved.

Communication performance from the perspective of message passing libraries has been studied. The difficulties and pitfalls of obtaining reproducible measurements are discussed in [13]. While the focus of this work was to illustrate issues involved with performance for a single application, we note that several of the perils cited have relevance to our work.

For example, depending on how communication connections are setup the “first communication between two processes can take far longer than subsequent communications”. Additionally, when trying to obtain reproducible measurements for a given application it is important not to ignore contention with unrelated applications or jobs. This is precisely the type of behavior we are interested in, however we are interested in how application run-time is affected by this contention. Finally, message-passing accomplishes both the transfer of data and a synchronization. The synchronization (or handshake) indicates that the data are available. We need to ensure that this is taken into account.

4. Experiments and Results

It is generally difficult to acquire sufficient direct data reflecting the overall execution of parallel programs. Sufficiently instrumenting a parallel application can lead to overwhelming amounts of data that must then be analyzed. Instrumenting and gathering large amounts of data can also lead to the altering of the original performance of the application itself. Moreover, the nature of concurrent processing suggests that no two executions of the same parallel program using the same input data set will execute exactly the same way with the same run-time. It is precisely this idea that makes it so difficult to precisely predict how long a parallel program will take to run.

In most cases, jobs are submitted to clusters in such a way that a request is made to the scheduler for some number of nodes for some period of (wall clock) time. It has been reported in [10] and [18] that in fact these requests overestimate the actual wall clock execution time by factors of 2 to 5. Therefore the requested time for nodes is actually the sum of an estimated run-time plus some amount of “padding”. This padding not only complicates the task of the scheduler, but it also demonstrates the ineffectiveness of accurately predicting the run-time of parallel programs. Conversely, we see that parallel programmers do have some idea of how long their program should take to execute, yet there seems to be a wide variation in overestimation thus it is difficult to determine the accuracy of the estimate portion.

Our preliminary investigation of run-time variability is the analysis of run-time logs produced by the scheduler/resource management system (Maui/PBS) [33],[27] of the Chiba City [21] Linux NOW cluster at Argonne National Laboratory. The Maui scheduler has gained wide acceptance as one of the most advanced schedulers currently available to the High Performance Computing (HPC) community. Maui is what is known as a batch scheduler, which determines when and where submitted jobs should run in a sequential fashion [18].

The Maui scheduler handles both static and automatic control of the batch job process. By automatic we mean that Maui automatically distributes the required input and executable files out to the allocated nodes and begins execution of the job. If an application (or script) terminates before the scheduled timeout of the job, Maui will automatically close down the job by retrieving the output files for the submitter, then relinquishing the nodes back to being available. Alternatively, users can reserve nodes for a period of time and manually push their job files out to each reserved node, run the job, then manually pull the resulting output back. Here the user does not need to specifically estimate the run-time of the job. Rather, the user must estimate how long it will take to run the job as well as the time to push any executable and data files to the assigned nodes and also collect any output data from the nodes when the job finishes. In either case (more so in the manual case) it is possible for there to be idle nodes for long periods of time.

Maui operates on an iterative basis, where events trigger the beginning of a new iteration. These events include a job or resource state-change, a reservation boundary, an external command, or a timeout. Maui supports the notions of job class, QoS, job credentials, and throttling policies. It is worth noting that the concept of QoS in the scheduler context has nothing to do with communication or network QoS [33]. Instead, QoS applies to special privileges such as improved queue time, access to additional resources, or exemptions from certain constraining policies.

Two policies for backfill are implemented in Maui, where backfill is a scheduling optimization allowing better (high utilization) of resources by running jobs out of order. This concept *“essentially fills holes in the node space”*. The backfill policies are called “soft” and “hard” policy backfill respectively. They differ by virtue of soft and hard “throttling policies” which limit resources such as processors, jobs, nodes, and memory. One of the drawbacks of backfilling cited is job subset delay, which strongly relates to inaccurate estimation of run-time performance by the submitter, a problem discussed earlier.

To gain a better understanding parallel application run-time variability we performed experiments consisting of communication tests and event log analysis. These initial experiments were aimed at confirming or refuting the fol-

lowing hypotheses:

1. Parallel applications running on NOW clusters exhibit a high degree of run-time variability.
2. Requested node times for these applications significantly overestimate (pad) the actual run-time.
3. Linear communications models used for application design and run-time estimation purposes may not accurately reflect "real" communications.

4.1. Application Run-Time Variability

Maui scheduler log files from the years 2000, 2001, and 2002 were examined. Each job is represented by an event entry similar to that shown in figure 1. Each log file contains event records of the completed job activity from a single day. Each job completion event constitutes a record in the file. Note that in figure 1 there are indications of which nodes were assigned to the job (`ccn152:ccn151:...`) and what interconnection network was used (`[myrinet]`).

```
35248      0          12         binns       futures
28800      Completed  [default:1]1034179264  1034179277
1034179277 1034181293 [NONE]     [NONE]      [NONE]
>=         0          >=         0           [myrinet]
1034179264 12         1          [NONE]      [RESTARTABLE]
mcs-mml    [NONE]     [NONE]     0           239162.01
DEFAULT    1          0          0           0
0          2140000000 ccn152:ccn151:ccn149:ccn148:ccn116:
ccn115:ccn113:ccn112:ccn111:ccn110:ccn109:ccn108
0          [NONE]     [NONE]     [DEFAULT]   [NONE]
[NONE]
```

Figure 1. Maui run-time log entry (2002)

The event logs of a set of over 4000 jobs executed during 8 randomly selected days between August to December of 2002 were examined. Of the 4209 jobs completed, 1318 (about 31%) of the jobs had requested more than one processor. Of the 2891 remaining single processor jobs, 1982 (about 69%) of those jobs "appear" to have been multiprocessor jobs. We deduce this by examining groups of completed job events where a single user requested more than one node at the same time (or within seconds of each other) and the run-time is similar (same order of magnitude). We therefore conclude that these multi-processor jobs were submitted to the scheduler in a manual fashion as previously described. Table 1 clarifies this point.

Returning our focus back to the 1318 multiprocessor jobs we examined the run-time of each job. Jobs that lasted less than 10 seconds were considered outliers and removed from the data set. For the remaining 1146 jobs, the requested time was compared against the actual run-time. Results of this comparison are shown in table 2. In this case

Table 1. Job Distribution

Jobs	Numproc > 1	Numproc = 1	Numproc = 1 & Multiproc
4209	1318 (31%)	2891 (69%)	1982

the majority of jobs (91%) are overestimated by a factor of 2 or greater.

Several interesting event sequences were also observed, where we define an "event sequence" as a series of "Completed" events close in time where the same user has requested the same number of nodes asking for the same amount of wall clock time. We deduce from the information contained in these sequences that the user ran the same application with presumably the same or very similar data set two or more times in a row, with subsequent runs following closely (within minutes) in time. By similar data we mean that the user did not in all likelihood change the size of the problem. If the user had changed the size of the problem then presumably the requested time of the job would have been adjusted to accommodate the change. These sequences allow us to identify potential cases of run-time variability, however we do not have enough information to determine the precise cause of the observed system and application behavior. Table 3 illustrates a sampling of sequences where the run-time variability is fairly high.

Table 2. Requested vs. Actual Run-time

Jobs	Req. $< 2x$ Act.	$\geq 2x$ to $\leq 5x$	Req. $> 5x$ Act.
1146	92 (8%)	101 (9%)	953 (83%)

Table 3 also illustrates the issue of overestimation of application run-time. We observe that the requested time of the nodes far exceeds the actual run-time (by a factor of 2 to over 10). In fact, in the best cases where applications are known to be stable and exhibit less than 5% communication overhead the logs revealed that the requested node times were still on the order of 2 to 5 times the actual run-time, as also stated in [10] and [18].

4.2. Non-Deterministic Node Allocation

As discussed in section 2, the parallel program design process consists of a "mapping" phase, where tasks are mapped to specific nodes in order to minimize communications contention. Schedulers in clusters of NOWs allocate nodes dynamically at run-time, effectively removing

Table 3. Run-time Event Sequences

User	Req. (sec.)	Actual (sec.)	Req. Nodes	Node Allocation
A	28000	416	12	111 - 100
A	28000	207	12	111 - 100
A	28000	2016	12	152, 151, 149, 148, 116, 115, 113-108
B	18000	1426	72	224-218, 216, 210-201, 154, 151, 130, 119, 118, 116, 115, 113-105, 103-98, 96, 95, 93-85, 81-74, 72, 71, 69, 67-65, 32, 31, 29-27, 25, 23
B	18000	131	72	223-218, 216, 210-207, 205-201, 154, 151, 130, 119, 118, 116, 115, 113-105, 103-101, 98, 96, 95, 93-85, 81, 80, 78, 77, 75, 74, 72, 71, 66, 65, 32, 31, 29-27, 25, 23, 20, 19, 16, 13, 12, 10, 7-4
C	21600	511	32	224 - 222, 210, 207, 203, 154, 151, 130, 119, 118, 116, 115, 113 - 105, 103 - 98, 96, 95, 93, 92
C	21600	187	32	224 - 218, 216, 215, 213 - 201, 154, 151, 130, 119, 118, 116, 115, 113 - 111
D	3600	1184	6	167 - 164, 162, 161
D	3600	431	6	183 - 178

the mapping step from the parallel program designer. If the scheduler does not use or have the capability of taking advantage of network topology, then intuitively the effectiveness of the mapping step is diminished from the perspective of the application. Moreover, we are concerned that the combination of not taking advantage of network topology and scheduler backfilling may inadvertently produce congestion spaces when several parallel applications execute simultaneously on the cluster.

Non-deterministic node allocation effects can be seen in table 3. User A clearly experiences a significant run-time increase when the allocation of nodes is more widely distributed around the cluster as shown between runs 2 and 3. It is less clear, but it appears that user B experiences an improvement when the allocation becomes apparently more widely distributed. The jobs submitted by user C on the other hand showed a marked improvement in run time when the allocation of nodes is tighter.

In the context of Chiba City the Maui scheduler assigns nodes in "reverse node number order". The physical distribution of nodes to switches is not in node order however. This is presumably done to ensure "all-to-all" maximum bisection bandwidth among the nodes in the cluster. The cluster topology is flat, with 256 compute nodes and 88 16-port switches. In the case of user A in table 3, the effect is that communications between the set of nodes utilizes 14 switches. Had the scheduler selected a specific "set" of nodes the number of switches involved could have been reduced to as little as 3. A more detailed examination of the network's routing tables or tests using different node assignments must be performed in order to determine if any congestion avoidance advantage is possible.

4.3. Communication Variability

As discussed previously in section 2, communication cost in parallel programs is most often described using linear $s + rn$ models. From the perspective of the application we would expect to see little variation in communication time for a given message size. Several tools exist for evaluating MPI performance such as MPIBench [16], mpptest [14], and SKaMPI [29] to distinguish the $s+rn$ relationship between inter-process, but intra-processor vs. inter-process, inter-processor communications. Many of these tools rely on finely granular global clocking mechanisms and are less useful to test the intra-processor (many processes on a single processor) communications for comparison purposes.

We wish to study the statistical variation of communication cost, a feature lacking in most performance specific tools. We are currently running experiments on Chiba City to ascertain the required timing accuracy and a reasonable development strategy for the first iteration of a tool to better understand the network contribution to communication variability.

We have executed a series of experiments to learn about the nature of communications on NOWs. These simple experiments expose communication variability as a function of message size. Tests were run on 1, 2, 4, and 8 node configurations. In all cases multiple processes communicated with each other in pairs, similar to the implementation of MPIBench [16]. The main difference was that rather than communicating as rapidly as possible we intentionally slowed down our "application communication emulation". Communications were slowed such that the run time was on the order of several hours to collect data on roughly several thousand messages. We also limited this set of experiments to blocking send operations.

Figure 2 illustrates the behavior observed on a single node. Message data in this case was only collected up to 64KB sized messages. Figures 3, 4, and 5 illustrate message variability behavior for 2, 4, and 8 nodes, particularly as the message size grows beyond 10KB. Data points that were clearly indicative of connection setup as described in [14] were considered outliers and removed from the data sets and subsequent statistical calculations.

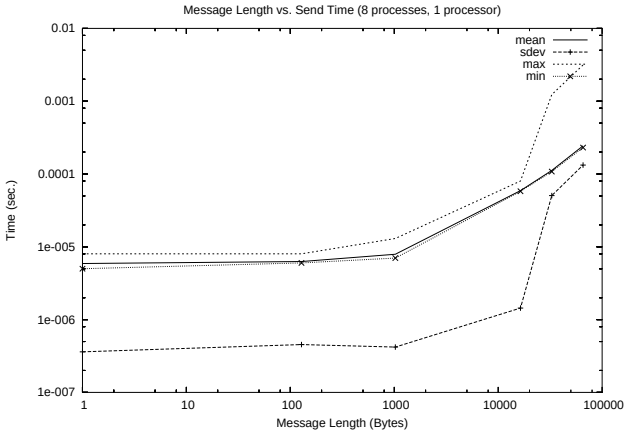

Figure 2. Blocking Send (1 Node, 8 Processes)

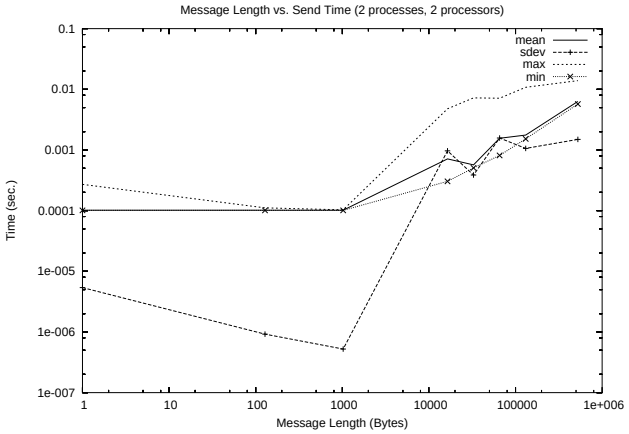

Figure 3. Blocking Send (2 Nodes, 2 Processes)

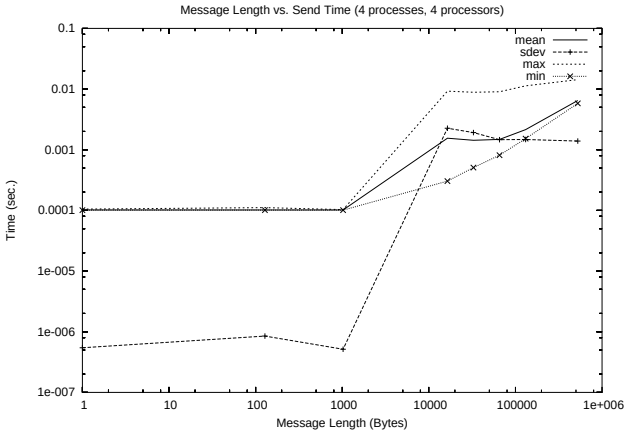

Figure 4. Blocking Send (4 Nodes, 4 Processes)

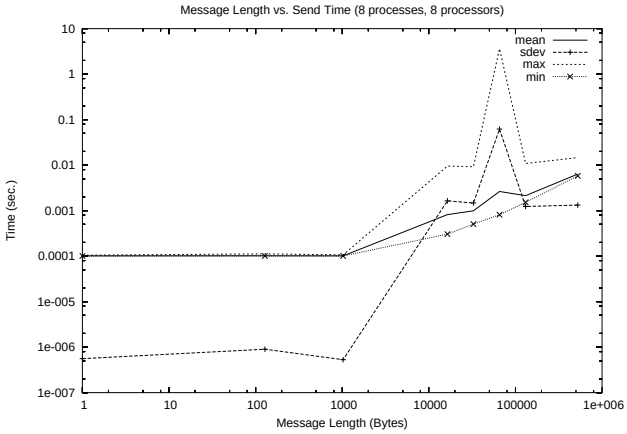

Figure 5. Blocking Send (8 Nodes, 8 Processes)

The results of these simple experiments are both interesting and surprising. Message sizes used for the 2, 4, and 8 node experiments were 1, 128, 1K, 16K, 32K, 64K, 128K, and 512K bytes. Notice in figure 2 as the message size increases (especially above 1KB) the mean value rises at a very near linear rate. This performance would be expected according to the $s + rn$ model. As we look at the graphs of multiprocessor communications we begin to see unexpected behavior.

In figures 3, 4, and 5 we can see sections where the mean message time (the solid line) is actually non-monotonic. In figures 3 and 5 two message sizes (32KB and 128KB) exhibited average message times that were less than the next smaller message size (16KB and 64KB respectively), which was unexpected. A degree of non-monotonicity can be expected due to a number of possible causes, such as a change in cache operation or communication protocol [14]. Generally however we would expect this to occur at one point. In these experiments only the minimum value (dashed "x" line) exhibited strictly monotonic behavior over the range of message sizes tested.

The maximum values (the dotted line) were observed to exhibit widely varying behavior across the range of experiments. Both the maximum value and standard deviation (dashed "plus" line) varied significantly across the 16KB to 128KB message sizes. After initially plotting these results we went back to investigate any obvious explanation for these results. Surprisingly, there was no indication that that the maximum value or standard deviation variations were caused by several excessively long connection estab-

lishment events. To the contrary, the data contributing to this were indeed scattered about the collected data in a random fashion.

These results illustrate the potential of subsystem limitations that may contribute to "systemic" inconsistency in NOW clusters. The limitations of Application communication cost modeling, scheduler effects, and interprocessor communications combine to degrade system performance, resulting in highly variable application run-time behavior. Our analysis and experiments are ongoing, however these early results suggest that run-time variability remains an issue in NOW clusters. The results also serve to motivate continued research in this area.

5. Summary and Future Work

We are in the early stages of our research to better understand the relationship between network performance and parallel application run-time sensitivity. This work provides the background and motivation to investigate using fault management techniques to strive toward systemic performance consistency in high performance computing. Parallel applications using high performance networks do so because maximum performance is expected, and therefore assumed. The interconnection network tries to move data as fast and efficiently as possible from point A to point B. In the presence of congestion the network could autonomously decide to re-route traffic to reduce or avoid congestion, but is this the "correct" decision to make?

Our approach to addressing parallel application run-time sensitivity comes from the area of fault management. The fault management paradigm is based on monitoring a system to verify it is performing to some performance objective or requirement. When a situation arises where this is compromised (or will shortly be compromised) a corrective action is undertaken for the benefit of the operation of the system.

While exploring the relationship between application run-time sensitivity and network performance we have also exposed three distinct yet inter-related cluster "subsystems", namely the scheduler/resource manager, the interconnection network, and the "set" of applications executing on the cluster at any given moment. We plan on investigating other potential relationships between these subsystems with the purpose of combining them in a comprehensive NOW cluster management framework.

The results of our experiments suggest that run-time variability continues to be an issue in high performance computing. Moreover, we show that run-time estimates continue to be conservatively padded to insure that the scheduler does not time out and terminate the application prematurely. Additionally, our results suggest that network topology may not be sufficiently considered when jobs are submitted, which also may contribute to run-time variability. Our preliminary findings also indicate that linear n-parameter models traditionally used for modeling communication cost may be less useful for larger message sizes and when multiple applications run simultaneously on the same system, a more common occurrence with today's large high performance NOW clusters. These issues when taken collectively may go to undermine consistent performance of the system as a whole.

Our future work promises to further expose the network contribution to parallel application run-time variability. It is recognized however, that subsystems acting autonomously to correct perceived performance degradation may be in fact contributing to systemic performance degradation. It is therefore important to recognize that a comprehensive approach should be considered that addresses the objectives and requirements of each critical subsystem (network, scheduler, and applications) in a collective manner to achieve more consistent performance.

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.

References

[1] T. Anderson, D. Culler, and D. Patterson. A case for networks of workstations: NOW. In *IEEE Micro*, pages 54–64. IEEE, Feb. 1995.

[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] E. Baydal, P. Lopez, and J. Duato. A congestion control mechanism for wormhole networks. In *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing*, pages 19–26, 2001.

[4] 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.

[5] S. Chakravarthi, A. Pillai, J. Padmanabhan, M. Apte, and A. Skjellum. A fi ne-grain synchronization mechanism for qos based communication on myrinet. Submitted to the International Conference on Distributed Computing, 2001, 2001.

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

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings*

of the Fourth ACM SIGPLAN Symposium on Princples and Practices of Parallel Programming, May 1993.

[8] S. P. Dandamudi. Reducing hot-spot contention in shared-memory multiprocessor systems. *IEEE Concurency*, 7(1), Jan.-Mar. 1999.

[9] X. Du and X. Zhang. Coordinating parallel processes on networks of workstations. *Journal of Parallel and Distributed Computing*, 46:125–135, 1997.

[10] D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfi lling. In *12th Intl. Parallel Processing Symp.*, pages 542–546, 1998.

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

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[13] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] P. J. Keleher, J. K. Hollingsworth, and D. Perkovic. Exposing application alternatives. In *International Conference on Distributed Computing Systems*, pages 384–392, 1999.

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

[22] D. A. Lifka. The ANL/IBM SP scheduling system. Technical Report MCS-P498-0395, Argonne National Laboratory, 1995. ftp://info.mcs.anl.gov/pub/tech_reports/reports/P498.ps.Z.

[23] B. Lowekamp, D. O'Hallaron, and T. Gross. Direct network queries for discovering network resource properties in a distributed environment. In *Proceedings of the 8th IEEE Symposium on High-Performance Distributed Computing (HPDC-8)*, pages 38–46, August 1999.

[24] J. Mache, V. Lo, and S. Garg. Job scheduling that minimizes network contention due to both communication and I/O. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS'00*, pages 457–463, May 2000.

[25] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 85–97, June 1997.

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

[27] OpenPBS.org. The Portable Batch System, 1998. http://www.OpenPBS.org/.

[28] 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.

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

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

[31] 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.

[32] 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.

[33] Supercluster.org. The Maui Scheduler, 2000. http://www.supercluster.org/maui/.

[34] 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.

[35] 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.

[36] 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.

[37] J. Vetter and K. Schwan. Progress: A toolkit for interactive program steering. In *Proceedings of the International Conference on Parallel Processing*, August 1995.

[38] J. S. Vetter and D. A. Reed. Real-time performance monitoring, adaptive control, and interactive steering of computational grids. *International Journal of High Performance Computing Applications*, 14(4):357–366, Winter 2000.