Performance Estimation for Scheduling on Shared Networks

Shreenivasa Venkataramaiah Jaspal Subhlok

Department of Computer Science
University of Houston
Houston, TX 77204
{shreeni,jaspal}@cs.uh.edu
www.cs.uh.edu/~jaspal

Abstract

This paper develops a framework to model the performance of parallel applications executing in a shared network computing environment. For sharing of a single computation node or network link, the actual performance is predicted, while for sharing of multiple nodes and links, performance bounds are developed. The methodology for building such a shared execution performance model is based on monitoring an application's execution behavior and resource usage under controlled dedicated execution. The procedure does not require access to the source code and hence can be applied across programming languages and models. We validate our approach with experimental results with NAS benchmarks executed in different resource sharing scenarios on a small cluster. Applicability to more general scenarios, such as large clusters, memory and I/O bound programs and wide are networks, remain open questions that are included in the discussion. This paper makes the case that understanding and modeling application behavior is important for resource allocation and offers a promising approach to put that in practice.

1 Introduction

Shared networks, varying from workstation clusters to computational grids, are an increasingly important platform for high performance computing. Performance of an application strongly depends on the dynamically changing availability of resources in such distributed computing environments. Understanding and quantifying the relationship between the performance of a particular application and available resources, i.e., how will the application perform under given network and CPU conditions, is important for resource selection and for achieving good and predictable performance. The goal of this research is automatic development of application performance models that can estimate application execution behavior under different network conditions.

This research is motivated by the problem of resource selection in the emerging field of grid computing [11, 12]. The specific problem that we address can be stated as follows: "What is the best set of nodes and links on a given network computation environment for the execution of a given application under current network conditions?" Node selection based on CPU considerations has been dealt effectively by systems like Condor [13] and LSF [27] but network considerations make this problem significantly more complex. A solution to this problem requires the following major steps:

1. *Application characterization:* Development of an application performance model that captures the resource needs of an application and models its performance under different network and CPU conditions.

2. *Network characterization:* Tools and techniques to measure and predict network conditions such as network topology, available bandwidth on network links, and load on compute nodes.

3. *Mapping and scheduling:* Algorithms to select the best resources for an application based on existing network conditions and application's performance model.

Figure 1 illustrates the general framework for resource selection. In recent years, significant progress has been made in several of these components. Systems that characterize a network by measuring and predicting the availability of resources on a network exist, some examples being NWS[26] and Remos[14]. Various algorithms and systems to map and schedule applications onto a network have

Appears in the Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing, June 2003, Seattle, WA

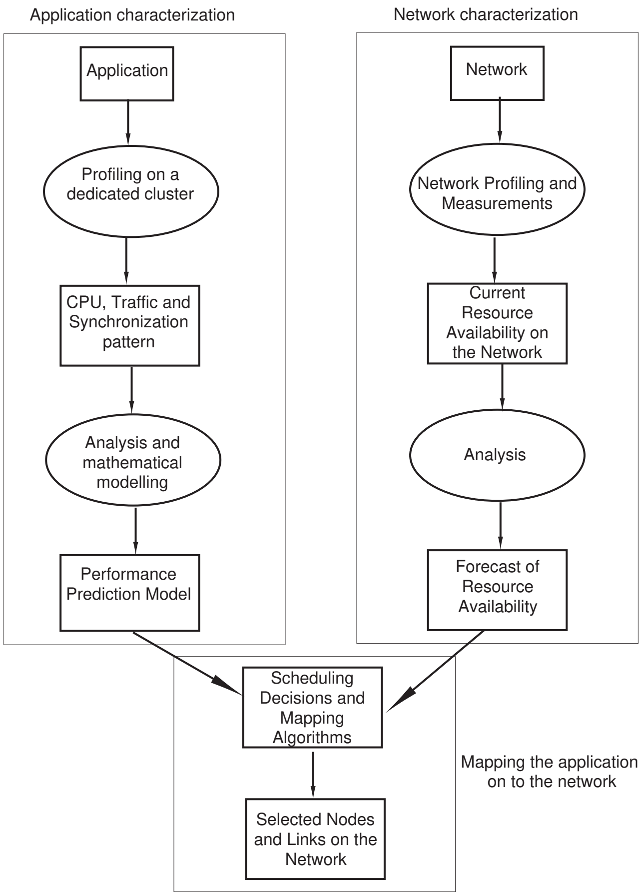

Figure 1. Framework for resource selection in a network computing environment

been proposed, such as [4, 5, 18, 21, 24]. In general, these projects target specific classes of applications and assume a simple, well defined structure and resource requirements for their application class. In practice, applications show diverse structures that can be difficult to quantify. Our research is focused on application characterization and builds on earlier work on dynamic measurement of resource usage by applications [19]. The goal is to automatically develop application performance models to estimate performance in different resource availability scenarios. We believe that this is an important *missing piece* in successfully tackling the larger problem of automatic scheduling and resource selection.

This paper introduces a framework to model and predict the performance of parallel applications with CPU and network sharing. The framework is designed to work as a tool on top of a standard Unix/Linux environment. Operating system features to improve sharing behavior have been studied in the MOSIX system [3]. The techniques employed to model performance with CPU sharing have also been studied in other projects with related goals [1, 25]. This paper generalizes authors' earlier work [23] to a broader class of resource sharing scenarios, specifically loads and traffic on multiple nodes and communication links. For more complex scenarios, it is currently not possible to make accurate predictions, so this research focuses on computing upper and lower bounds on performance. A good lower bound on performance is the characteristic that is most useful for the purpose of resource selection.

The approach taken in this work is to measure and infer the core execution parameters of a program, such as the message exchange sequences and CPU utilization pattern, and use them as a basis for performance modeling with resource sharing. This is fundamentally different from approaches that include analysis of application code to build a performance model that have been explored by many researchers, some examples being [6, 9]. In our view, static analysis of application codes has fundamental limitations in terms of the program structures that can be analyzed accurately, and in terms of the ability to predict dynamic behavior. Further, assuming access to source code and libraries inherently limits the applicability of this approach. In our approach, all measurements are made by system level probes, hence no program instrumentation is necessary and there is no dependence on the programming model with which an application was developed. Some of the challenges we address are also encountered in general performance modeling and prediction for parallel systems [7, 10, 16].

We present measurements of the performance of the NAS benchmark programs to validate our methodology. In terms of the overall framework for resource selection shown in Figure 1, this research contributes and validates an application characterization module.

2 Overview and validation framework

The main contribution of this paper is construction of application performance models that can estimate the impact of competing computation loads and network traffic on the performance of parallel and distributed applications. The performance estimation framework works as follows. A target application is executed on a controlled testbed, and the CPU and communication activity on the network is monitored. This system level information is used to infer program level activity, specifically the sequence of time slots that the CPU spends in compute, communication, and idle modes, and the size and sequence of messages exchanged between the compute nodes. The program level information is then used to model execution with resource sharing. For simpler scenarios, specifically sharing of a single node or a single network link, the model aims to predict the actual execution time. For more complex scenarios that involve sharing on multiple nodes and network links, the model estimates upper and lower bounds on performance.

The input to an application performance model is the expected CPU and network conditions, specifically the load average on the nodes and expected bandwidth and latency on the network routes. Computing these is not the subject of the paper but is an important component of any resource selection framework that has been addressed in related research [8, 14, 26].

We have developed a suite of monitoring tools to measure the CPU and network usage of applications. The CPU monitoring tool would periodically probe (every 20 milliseconds for the reported experiments) the processor status and retrieve the application's CPU usage information from the kernel data structures. This is similar to the working of the UNIX *top* utility and provides an application's CPU busy and idle patterns. The network traffic between nodes is actively monitored with *tcpdump* utility and application messages are reassembled from network traffic as discussed in [17]. Once the sequence of messages between nodes is identified, the communication time for the message exchanges is calculated based on the benchmarking of the testbed. This yields the time each node CPU spends on computation, communication and synchronization waits.

In order to validate this shared performance modeling framework, extensive experimentation was performed with MPI implementation of Class A NAS Parallel benchmarks [2], specifically the codes EP (Embarrassingly Parallel), BT (Block Tridiagonal solver), CG (Conjugate Gradient), IS (Integer Sort), LU (LU solver), MG (Multigrid), and SP (Pentadiagonal solver). The compute cluster used for this research is a 100Mbps Ethernet based testbed of 500 MHz, Pentium 2 machines running FreeBSD and MPICH implementation of MPI. Each of the NAS codes was compiled with *g77* or *gcc* for 4 nodes and executed on 4 nodes. The computation and communication characteristics of these codes were measured in this prototyping phase. The time spent by the CPUs of executing nodes in different activities is shown in Figure 2. The communication traffic generated by the codes is highlighted in Figure 3 and was verified with a published study of NAS benchmarks [20]. The details of the measured execution activity, including the average duration of the busy and idle phases of the CPU, are presented in Table 1.

In the following sections we will discuss how this information was used to concretely model the execution behavior of NAS benchmarks with compute loads and network traffic. We will present results that compare the measured execution time of each benchmark under different CPU and network sharing scenarios, and how they compare with the estimates and bounds computed by the application performance model.

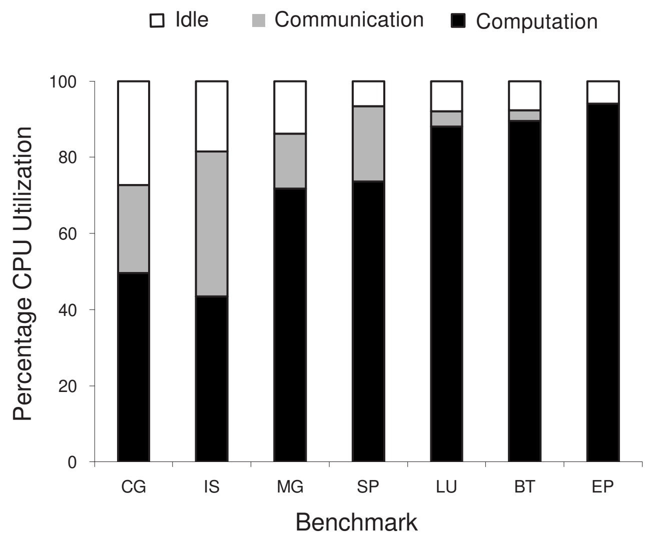

Figure 2. CPU usage during execution of NAS benchmarks

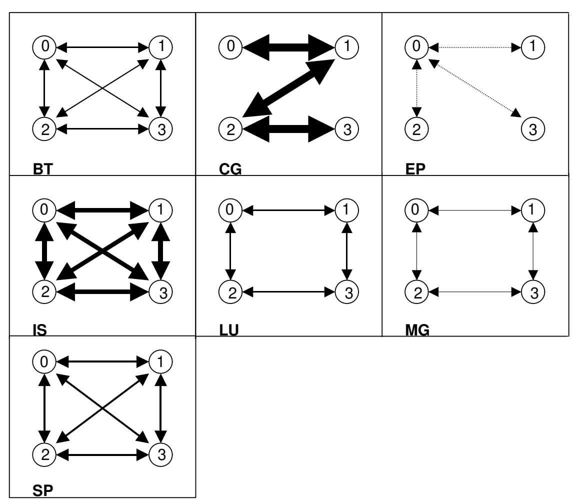

Figure 3. Dominant communication patterns during execution of NAS benchmarks. The thickness of the lines is based on the generated communication bandwidth.

Benchmark	Execution time - On a dedicated system (seconds)	Percentage CPU Time			Average CPU Time (milliseconds)		Messages on one link	
		Busy		Idle	Busy Phase	Idle Phase	Number	Average size (KBytes)
		Compuat-ion	Communicat-ion					
CG	25.6	49.6	23.1	27.3	60.8	22.8	1264	18.4
IS	40.1	43.4	38.1	18.5	2510.6	531.4	11	2117.5
MG	43.9	71.8	14.4	13.8	1113.4	183.0	228	55.0
SP	619.5	73.7	19.7	6.6	635.8	44.8	1606	102.4
LU	563.5	88.1	4.0	7.9	1494.0	66.0	15752	3.8
BT	898.3	89.6	2.7	7.7	2126.0	64.0	806	117.1
EP	104.6	94.1	0	5.9	98420.0	618.0	0	0

Table 1. Measured execution characteristics of NAS benchmarks

3 Modeling performance with CPU sharing

3.1 CPU scheduler on nodes

We assume that the node scheduler assigns the CPU to active processes fairly as follows. All processes on the ready queue are normally given a fixed execution time slice in round robin order. If a process blocks during execution, it is removed from the ready queue immediately and the CPU is assigned to another waiting process. A process gains priority (or collects credits) for some time while it is blocked so that when it is unblocked and joins the ready queue again it will receive a higher share of the CPU in the near future. The net effect is that each active process receives approximately equal CPU time even if some processes block for short intervals during execution. In our experience, this is qualitatively true at least of most Unix based systems, even though the exact CPU scheduling policies are complex and vary significantly among operating systems.

3.2 CPU shared on one node

We investigate the impact on total execution time when one of the nodes running an application is shared by another competing CPU intensive process. The basic problem can be stated as follows: If a parallel application executes in time T on a dedicated testbed, what is the expected execution time if one of the nodes has a competing load?

Suppose an application repeatedly executes on a CPU for *busytimephase* seconds and then sleeps for *idletimephase* seconds during dedicated execution. When the same application has to share the CPU with a compute intensive load, the scheduler will attempt to give equal CPU time slices to the two competing processes. The impact on the overall execution time due to CPU sharing depends on the values of *busytimephase* and *idletimephase* as illustrated in Figure 4 and explained below for different cases:

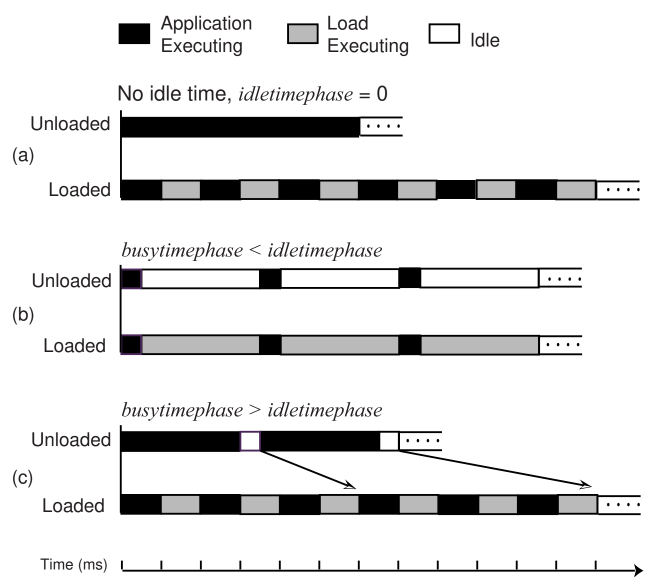

Figure 4. Relationship between CPU usage pattern during dedicated execution and execution pattern when the CPU has to be shared with a compute load.

- $idletimephase = 0$: The CPU is always busy without load. The two processes get alternate equal time slices and the execution time doubles as shown in Figure 4(a).

- $busytimephase < idletimephase$: There is no increase in the execution time. Since the CPU is idle over half the time without load, the competing process gets more than its fair share of the CPU from the idle CPU cycles. This is illustrated in Figure 4(b)

- $busytimephase > idletimephase$: In this situation, the competing process cannot get its entire fair share of the CPU from idle cycles. The scheduler gives equal time slices to the two processes. This case is illustrated in Figure 4(c). The net effect is that every cycle of duration $busytimephase+idletimephase$ now executes in $2 * busytimephase$ time. Alternately stated, the execution time will increase by a factor of:

$$\frac{busytimephase - idletimephase}{busytimephase + idletimephase}$$

Once the CPU usage pattern of an application is known, the execution time with a compute load can be estimated based on the above discussion. For most parallel applications, the CPU usage generally follows a pattern where $busytimephase > idletimephase$. A scheduler provides fairness by allocating a higher fraction of CPU in the near future to a process that had to relinquish its time slice because it entered an idle phase, providing a smoothing effect. For a parallel application that has the CPU busy $cpubusy$ seconds, and idle for $cpuidle$ seconds on aggregate during execution, the execution time often simply increases to $2 * cpubusy$ seconds. This is the case for all NAS benchmark programs. The exception is when an application has long intervals of low CPU usage and long intervals of high CPU usage. In those cases, the impact on different phases of execution has to be computed separately and combined. Note that the execution time with two or more competing loads, or for a given UNIX *load average*, can be predicted in a similar fashion.

In order to validate this approach, the execution characteristics of the NAS programs were computed as discussed and the execution time with sharing of CPU on one node was estimated. The benchmarks were then executed with a load on one node and the predicted execution time was compared with the corresponding measured execution time. The results are presented in Figure 5. There is a close correspondence between predicted and measured values for all benchmarks, validating our prediction model for this simple scenario. It is clear from Figure 5 that our estimates are significantly more accurate than the naive prediction that the execution time doubles when the CPU has to be shared with another program.

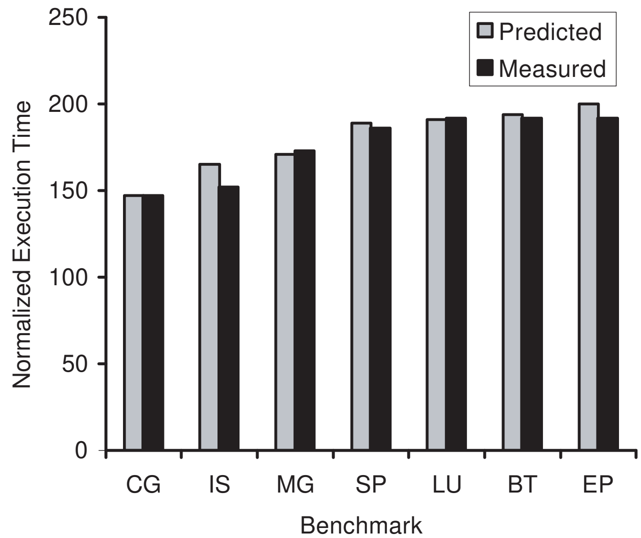

Figure 5. Comparison of predicted and measured execution times with a competing compute load on one node. The execution time with no load is normalized to 100 units for each program.

3.3 CPU shared on multiple nodes

We now consider the case where the CPUs on all nodes have to be shared with a competing load. Additional complication is caused by the fact that each of the nodes is scheduled independently, i.e., there is no coordinated (or gang) scheduling. This does not impact the performance of local computations but can have a significant impact on communication. When one process is ready to send a message, the receiving process may not be executing, leading to additional communication and synchronization delays. It is virtually impossible to predict the exact sequence of events and arrive at precise performance predictions in this case [1]. Therefore, we focus on developing upper and lower bounds for execution time.

During program execution without load, the CPU at any given time is computing, communicating or idle. We discuss the impact on the time spent on each of these modes when there is a competing load on all nodes.

- *Computation:* The time spent on local computations with load can be computed exactly as in the case of a compute load on only one node that was discussed earlier. For most parallel applications, this time doubles with fair CPU sharing.

- *Communication:* The CPU time for communication is first expected to double because of CPU sharing. Completion of a communication operation implemented over the networking (TCP/IP) stack requires active processing on sender and receiver nodes even for asynchronous operations. Further, when one process is ready to communicate with a peer, the peer process may not be executing due to CPU sharing since all nodes are scheduled independently. The probability that a process is executing at a given point when two processes are sharing the CPU is 50%. If a peer process is not active, the process initiating the communication may have to wait half a CPU time slice to start communicating. A simple analysis shows that the communication time could double again due to independent scheduling. However, this is the statistical worst case scenario since the scheduler will try to compensate the processes that had to wait, and because pairs of processes can start executing in lock-step in the case of regular communication. Hence, the communication time *may* increase by up to a factor of 4.

- *Idle:* For compute bound parallel programs, the CPU is idle during execution primarily waiting for messages or signals from another node. Hence, the idle time occurs while waiting for a sequence of computation and communication activities involving other executing nodes to complete. The time taken for computation and communication activities is expected to increase by a factor of 2 and 4, respectively with CPU sharing. Hence, in the worst case, the idle time may increase by a factor of 4.

Based on this discussion, we have the following result. Suppose *comptime, commtime* and *idletime* are the time spent by the node CPUs computing, communicating, and idling during execution on a dedicated testbed. The execution time is bounded from above by:

$2 * comptime + 4 * (commtime + idletime)$

The execution time is also loosely bounded from below by:

$2 * (comptime + commtime)$

which is the expected execution time when the CPU is shared on only one node.

For validation, the higher and lower bounds for execution time for the NAS benchmarks with a single load process on all nodes were computed and compared with the measured execution time under those conditions. The results are charted in Figure 6.

We observe that the measured execution time is always within the computed bounds. The range between the bounds is large for communication intensive programs, particularly CG and IS. For most applications, the measured values are in the upper part of the range, often very close to the predicted upper bound. The main conclusion is that the above analysis can be used to compute a meaningful upper bound for execution time with load on all nodes and independent scheduling. We restate that a good upper bound on execution time (or a lower bound on performance) is valuable for resource selection.

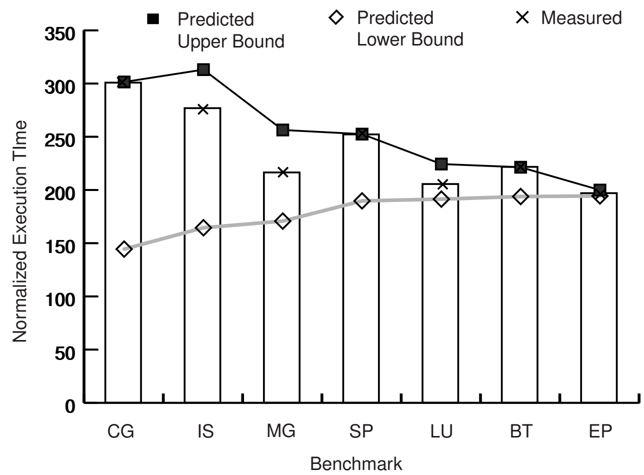

Figure 6. Comparison of predicted and measured execution times with a competing compute load on all nodes.

4 Modeling performance with communication link sharing

4.1 One shared link

We study the impact on execution time if a network link has to be shared or the performance of a link changes for any reason. We assume that the performance of a given network link, characterized by the effective latency and bandwidth observed by a communicating application, are known. We want to stress that finding the expected performance on a network link is far from trivial in general, even when the capacity of the link and the traffic being carried by the link are known.

The basic problem can be stated as follows: If a parallel application executes in time *T* on a dedicated testbed, what is the expected execution time if the effective latency and bandwidth on a network link change from *L* and *B* to *newL* and *newB*, respectively.

The difference in execution time will be the difference in the time taken for sending and receiving messages after the link properties have changed. If the number of messages traversing this communication link is *nummsgs* and the average message size is *avgmsgsize*, then the time needed for communication increases by:

$$[(newL + avgmsgsize/newB) - (L + avgmsgsize/B)] * nummsgs$$

We use this equation to predict the increase in execution time when the effective bandwidth and latency on a communication link change.

For the purpose of validation, the available bandwidth on one of the links on our 100Mbps Ethernet testbed was reduced to a nominal 10Mbps with *dummynet* [15] tool. The characteristics of the changed network were measured and the information was used to predict the execution time for each NAS benchmark program. The programs were then executed on this modified network and the measured and predicted execution time were compared. The results are presented in Figure 7.

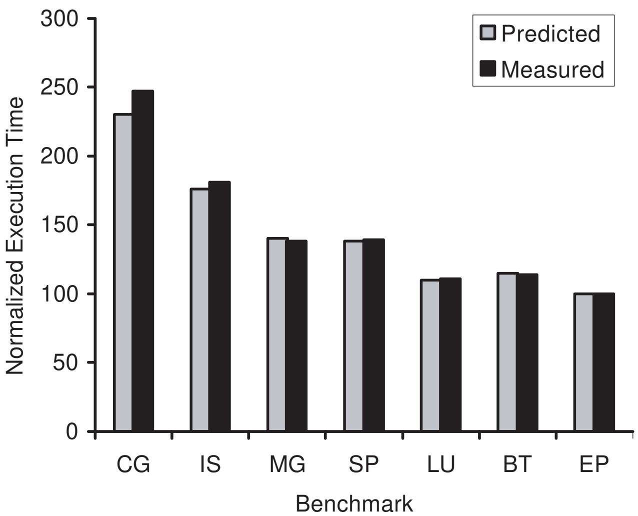

Figure 7. Comparison of predicted and measured execution times with bandwidth reduced to 10 Mbps from 100 Mbps on one communication link.The execution time with no load is normalized to 100 units for each program.

We observe that the predicted and measured values are fairly close demonstrating that the prediction model is effective in this simple scenario of resource sharing.

4.2 Multiple shared links

When the performance of multiple communication links is reduced, the application performance also suffers indirectly because of synchronization effects. As in the case of load on all nodes, we discuss how the time the CPU spends on computation, communication, and idle phases during execution on a dedicated testbed, changes due to link sharing.

- *Computation:* The time spent on local computations remains unchanged with link sharing.
- *Communication:* The time for communication will increase as discussed in the case of sharing of a single link. The same model can be used to compute the increase in communication time.
- *Idle:* As discussed earlier, the idle time at nodes of an executing parallel program occurs while waiting for a sequence of computation and communication activities involving other executing nodes to complete. Hence, in the worst case, the idle time may increase by the same factor as the communication time.

We introduce *commratio* as the factor by which the time taken to perform the message exchange sequences on the executing nodes are expected to slow down due to link sharing. (the largest value is used when different nodes perform different sequences of communication.) That is, the total time to physically transfer all messages in an application run is expected to change by a factor *commratio*, not including any synchronization related delay. This *commratio* is determined by two factors.

1. The messages sequence sent between a pair of nodes including the size of each message.
2. The time to transport a message of a given size between a pair of nodes.

The message sequences exchanged between nodes is computed ahead of time as discussed earlier in this paper. The time to transfer a message depends on application level latency and bandwidth between the pair of nodes. A network measurement tool like NWS is used to obtain these characteristics. For the purpose of experiments reported in this paper, the effective latency and bandwidth was determined by careful benchmarking ahead of time with different message sizes and different available network bandwidths. The reason for choosing this way is to factor out errors in network measurements in order to focus on performance modeling.

We then have the following result. Suppose *comptime, commtime* and *idletime* are the time spent by the node CPUs computing, communicating, and idling during execution on a dedicated testbed. An upper bound on the application execution time due to a change in the characteristics of all links is:

$$comptime + commratio * (commtime + idletime)$$

A corresponding lower bound on execution time is:

$$comptime + commratio * commtime + idletime$$

For validation, the bounds for execution time for the NAS benchmarks with nominal available bandwidth reduced from 100Mbps to 10Mbps were computed and compared with the measured execution times under those conditions. The results are charted in Figure 8.

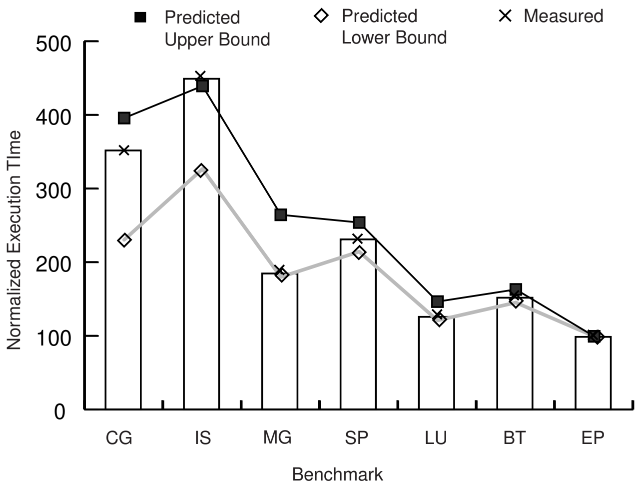

Figure 8. Comparison of estimated and measured execution times with bandwidth limited to 10 Mbps on all communication links.

We note that all measured execution times are within the bounds, except that the measured execution time for IS is marginally higher than the upper bound. As expected, the range covered by the bounds is larger for communication intensive programs CG, IS and MG. In the case of IS, the measured value is near the upper bound implying that synchronization waits are primarily related to communication in the program, while the measured execution time of MG is near the lower bound indicating that the synchronization waits are primarily associated with computations on other nodes. The main conclusion is that this analysis can be used to compute meaningful upper and lower bounds for execution time with sharing on all communication links.

We have used the execution time from a representative run of the application for each scenario for our results. In most cases the range of execution time observed is small, typically under 1%. However, for applications with high rate of message exchange, significant variation was observed between runs. For example, in the case of LU running with competing loads on all nodes, the difference between the slowest and fastest runs was around 10%.

5 Limitations and extensions

We have made a number of assumptions, implicit and explicit, in our treatment, and presented results for only a few scenarios. We now attempt to distinguish between the fundamental limitations of this work and the assumptions that were made for simplicity.

- **Estimation of network performance:** For estimation of performance on a new or changed network, we assume that the expected latency and bandwidth on the network links are known and can be predicted for the duration of the experiments. The results of performance estimation can only be as good as the prediction of network behavior. Estimation of expected network performance is a a major challenge for network monitoring tools and accurate prediction is often not possible. However, this is orthogonal to the research presented in this paper. Our goal is to find the best performance estimates for given network characteristics.

- **Asymmetrical computation loads and traffic:** We have developed results for the cases of equal loads on all nodes and equal sharing on all links. This was done for simplicity. The approach is applicable for different loads on different nodes and different available bandwidth on different links. The necessary input for analysis is the load average on every node and expected latency and bandwidth on links. In such situations, typically the slowest node and the slowest link will determine the bounds on application speed. The modeling is also applicable when there is sharing of nodes as well as links but we have omitted the details due to lack of space. More details are described in [22].

- **Asymmetrical applications:** We have implicitly assumed that all nodes executing an application are following a similar execution pattern. In case of asymmetrical execution, the approach is applicable but the analysis has to be done for each individual node separately before estimating overall application performance. Similarly, if an application executes in distinctly different phases, the analysis would have to be performed separately for each phase.

- **Execution on a different architecture from where an application performance model was prototyped:** If the relative execution speed between the prototyping and execution nodes is fixed and can be determined, and the latency and bandwidth of the executing network can be inferred, a prediction can be performed. This task is relatively simple when moving between nodes of similar architectures, but is very complex if the executing nodes have a fundamentally different cache hierarchy or processor architecture as compared to prototyping nodes.

- **Wide area networks:** All results presented in this paper are for a local cluster. The basic principles are designed to apply across wide area networks also although the accuracy of the methodology may be different. An important issue is that our model does not account for sharing of bandwidth by different communication streams within an application. This is normally not a major factor in a small cluster where a crossbar switch allows all nodes to simultaneously communicate at maximum link speed. However, it is an important consideration in wide area networks where several application streams may share a limited bandwidth network route.

- **Large systems:** The results developed in this paper are independent of the number of nodes but the experimentations was performed only on a small cluster. How well this framework will work in practice for large systems remains an open question.
- **Memory and I/O constraints:** This paper does not address memory bound or I/O bound applications. In particular, it is assumed that sufficient memory is available for the working sets of applications even with sharing. In our evaluation experiments, the synthetic competing applications do not consume significant amount of storage and hence the caching behavior of the benchmarks is not affected with processor sharing. Clearly more analysis is needed to give appropriate consideration to storage hierarchy which is critical in many scenarios.
- **Different data sets and number of nodes than the prototyping testbed:** If the performance pattern is strongly data dependent, an accurate prediction is not possible but the results from this work may still be used as a guideline. This work does not make a contribution for performance prediction when the number of nodes is scaled, but we conjecture that it can be matched with other known techniques.
- **Application level load balancing:** We assume that each application node performs the same amount of work independent of CPU and network conditions. Hence, if the application had internal load balancing, e.g., a master-slave computation where the work assigned to slaves depends on their execution speed, then our prediction model cannot be applied directly.

6 Conclusions

This paper demonstrates that detailed measurement of the resources that an application needs and uses can be employed to build an accurate model to predict the performance of the same application under different network conditions. Such a prediction framework can be applied to applications developed with any programming model since it is based on system level measurements alone and does not employ source code analysis. In our experiments, the framework was effective in predicting the execution time or execution time bounds of the programs in the NAS parallel benchmark suite in a variety of network conditions.

To our knowledge, this is the first effort in the specific direction of building a model to estimate application performance in different resource sharing scenarios, and perhaps, this paper raises more questions than it answers. Some of the direct questions about the applicability of this approach are discussed (but not necessarily answered) in the previous section. Different application, network, processor and system architectures raise issues that affect the applicability of the simple techniques developed in this paper. However, our view is that most of those problems can be overcome with improvement of the methodology that was employed.

More fundamentally, the whole approach is based on the ability to predict the availability of networked resources in the near future. If resource availability on a network changes in a completely dynamic and unpredictable fashion, no best effort resource selection method will work satisfactorily. In practice, while future network state is far from predictable, reasonable estimates of the future network status can be obtained based on recent measurements. The practical implication is that the methods in this paper may only give a rough estimate of the expected performance on a given part of the network, since the application performance estimate is, at best, as good as the estimate of the resource availability on the network. However, these performance estimates are still a big improvement over current techniques that either do not consider application characteristics, or use a simplistic qualitative description of an application such as master-slave or SPMD. Even an approximate performance prediction may be able to effectively make a perfect (or the best possible) scheduling decision by selecting the ideal nodes for execution.

In summary, the ability to predict the expected performance of an application on a given set of nodes, and using this prediction for making the best possible resource choices for execution, is a challenging problem which is far from solved by the research presented in this paper. However, this paper makes a clear contribution toward predicting application performance or application performance bounds. We believe this is an important step toward building good resource selection systems for shared computation environments.

7 Acknowledgments

This research was supported in part by the Los Alamos Computer Science Institute (LACSI) through Los Alamos National Laboratory (LANL) contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California. Support was also provided by the National Science Foundation under award number NSF ACI-0234328 and the University of Houston's Texas Learning and Computation Center.

We wish to thank other current and former members of our research group, in particular, Mala Ghanesh, Amitoj Singh, and Sukhdeep Sodhi, for their contributions. Finally, the paper is much improved as a result of the comments and suggestions made by the anonymous reviewers.

References

[1] A. Arpaci-Dusseau, D. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *SIGMETRICS' 98/PERFORMANCE' 98 Joint Conference on the Measurement and Modeling of Computer Systems*, June 1998.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report 95-020, NASA Ames Research Center, December 1995.

[3] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996.

[5] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Trans. Softw. Eng.*, 24(5):376 – 390, May 1998.

[6] M. Clement and M. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Computing*, 23(10):1405–1420, 1997.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, May 1993.

[8] P. Dinda and D. O'Hallaron. An evaluation of linear models for host load prediction. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, August 1999.

[9] T. Fahringer, R. Basko, and H. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 347–56, Washington, DC, July 1992.

[10] T. Fahringer, B. Scholz, and X. Sun. Execution-driven performance analysis for distributed and parallel systems. In *2nd International ACM Sigmetrics Workshop on Software and Performance (WOSP 2000)*, Ottawa, Canada, Sep 2000.

[11] I. Foster and K. Kesselman. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[12] A. Grimshaw and W. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[14] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[15] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), Jan 1997.

[16] J. Schopf and F. Berman. Performance prediction in production environments. In *12th International Parallel Processing Symposium*, pages 647–653, Orlando, FL, April 1998.

[17] A. Singh and J. Subhlok. Reconstruction of application layer message sequences by network monitoring. In *IASTED International Conference on Communications and Computer Networks*, November 2002.

[18] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–172, Atlanta, GA, May 1999.

[19] J. Subhlok, S. Venkataramaiah, and A. Singh. Characterizing NAS benchmark performance on shared heterogeneous networks. In *11th International Heterogeneous Computing Workshop*, April 2002.

[20] T. Tabe and Q. Stout. The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan, Nov 1999.

[21] H. Tangmunarunkit and P. Steenkiste. Network-aware distributed computing: A case study. In *Second Workshop on Runtime Systems for Parallel Programming (RTSPP)*, Orlando, March 1998.

[22] S. Venkataramaiah. Performance prediction of distributed applications using CPU measurements. Master's thesis, University of Houston, August 2002.

[23] S. Venkataramaiah and J. Subhlok. Performance prediction for simple CPU and network sharing. In *LACSI Symposium 2002*, October 2002.

[24] J. Weismann. Metascheduling: A scheduling model for metacomputing systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[25] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared unix systems on the computational grid. *Cluster Computing*, 3(4):293–301, 2000.

[26] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The Network Weather Service. In *Proceedings of Supercomputing '97*, San Jose, CA, Nov 1997.

[27] S. Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Orlando, FL, April 1992.