Performance Prediction for Simple CPU and Network Sharing

Shreenivasa Venkataramaiah **Jaspal Subhlok**

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

Abstract

Performance of virtually all parallel and distributed applications deteriorates when a CPU or a communication link has to be shared, but the extent of the slowdown is application dependent. In our experiments with the NAS benchmarks, the slowdown due to congestion on a single link varied from negligible to 120 percent. Estimation of performance of an application under given network conditions is of central importance for resource selection and resource management in shared computing environments. This paper develops a framework to model the performance of applications with CPU and link sharing. The methodology is based on monitoring the application behavior and resource usage on a controlled testbed. The procedure does not require access to the source code or the libraries. We demonstrate that the performance of applications in simple scenarios of network and CPU sharing can be predicted fairly accurately. For the NAS benchmark suite, we observed that the average error in predicting the execution time in different resource sharing scenarios was in the range of 2-6% and the maximum error was below 12%.

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 conditions, is important for resource selection and for achieving good and predictable performance. The goal of this research is automatic development of performance profiles that can estimate application execution behavior under different network conditions.

This research is motivated by the problem of resource selection in shared heterogeneous environments which can be stated as follows: "What is the best set of nodes and links on the network for the execution of a given application under current network conditions?" A solution to this problem requires the following steps:

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

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[23] and Remos[11]. Various algorithms and systems to map and schedule applications onto a network have been proposed, such as [2, 4, 16]. However, these efforts assume that the applications fit a known simple profile. 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 [17]. The goal is to automatically develop application profiles to estimate performance in different resource availability scenarios. We believe that this is a critical *missing piece* in successfully tackling the larger problem of automatic resource selection.

Appears in LACSI Symposium 2002

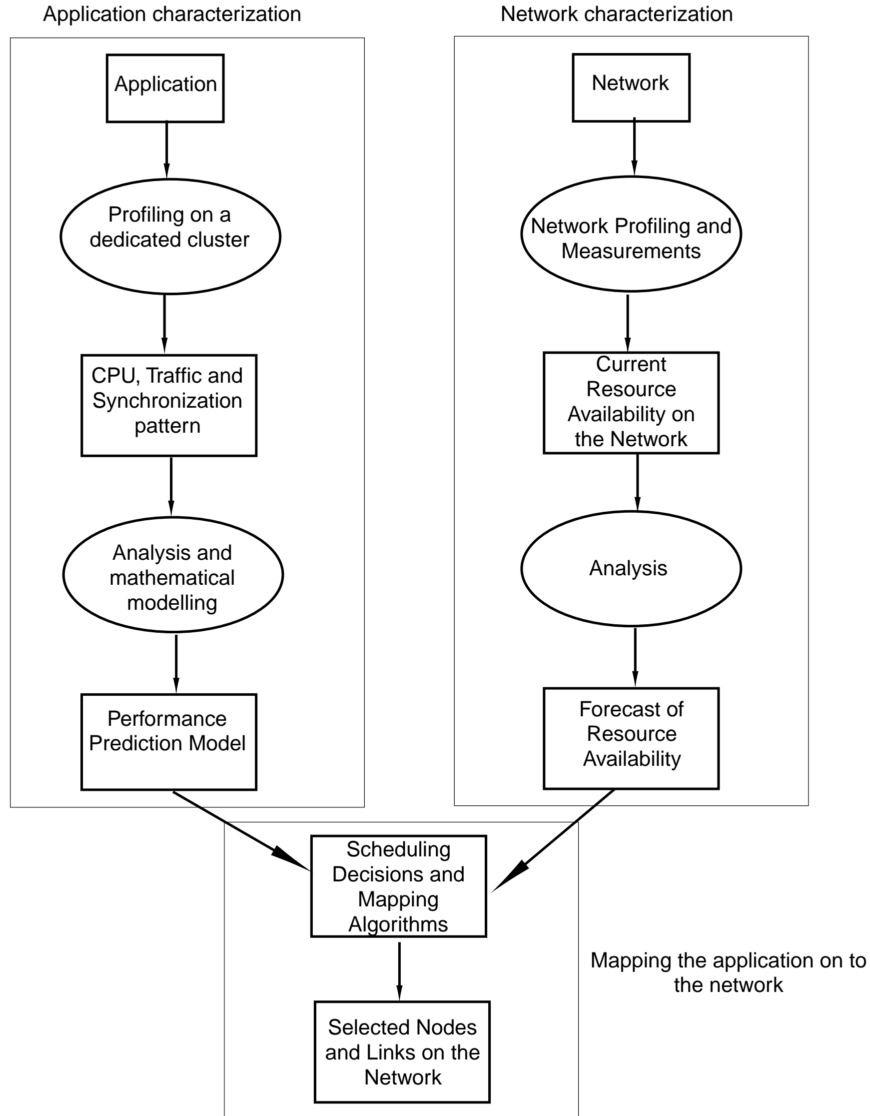

Figure 1: Framework for resource selection in a network computing environment

This paper introduces a framework to model and predict the performance of parallel applications in simple network sharing scenarios, such as shared nodes and links. Our approach is to measure 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. 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. We present measurements of the performance of the NAS benchmark programs that demonstrate that our methodology can predict the execution time with link and node sharing fairly accurately. In terms of the overall framework for resource selection, this research contributes and validates an application characterization module, albeit with limitations.

2 Modeling performance with resource sharing

We present the basic results that we have developed for prediction of performance with shared CPUs and network links. The basis for the performance prediction model for an application is a set of measurements made while executing the application on a dedicated testbed. The model is provided with the existing availability of network resources as input and it predicts application performance under those conditions. The prediction

model for a given application is based on the measured computation and communication characteristics of the application.

- *Computation:* Application processes switch between CPU usage (busy) and non-usage (idle) phases during normal execution. (We use the terms "idle" and "sleep" interchangeably in this paper). The average time duration for a CPU busy phase labeled *busytimephase* and the average time duration for a CPU idle phase labeled *idletimephase* are assumed to be known.

- *Communication:* The sequence and size of messages exchanged between every pair of nodes executing the application are assumed to be known. For a given pair of nodes, the number of messages exchanged between them is labeled *nummessages* and the average size of the messages is labeled *avgmsgsize*.

The procedure for computing these application characteristics is outlined later in this paper.

2.1 Execution with one shared 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? We shall assume that the CPU idle phases of the application are "short" in normal execution. We will quantify "short" later in this discussion.

The basic question we have to address is what happens to application execution when there is a single competing CPU process on one of the nodes. Suppose the application repeatedly executes for *busytimephase* seconds and then sleeps for *idletimephase* seconds during normal execution. The execution time with a competing load depends on how the scheduler arbitrates the CPU between the application process and the competing load process. The details of CPU scheduling are complex and vary significantly, even between different Unix systems, say FreeBSD and Linux. However, the goal of the scheduler in such a situation is to be fair and provide equal CPU time to the two processes. If neither of the processes enters a sleep phase, the scheduler simply gives alternate equal time slices to the two processes. In practice, when our application process is scheduled and reaches a sleep phase (typically waiting to receive a message), it will be evicted and not be able to completely use its time slice. The scheduler uses a sophisticated priority update mechanism to allocate the application process a larger share of the CPU if it wakes up in the near future. Under the assumption that sleeps are "short", the application will always wake up in time to reclaim the share of the CPU time it could not use because of its sleeps. The result is that our application and the competing CPU process, both receive a fair share of the CPU even in the presence of sleeps.

Now what happens to the total execution time of an application due to sharing? The results for different CPU busy-idle patterns is illustrated in Figure 2. We discuss two cases separately:

- $busytimephase <= idletimephase$: In this case there is no increase in the execution time. The reason is that every *busytimephase* has a larger compensating *idletimephase*. Hence the competing process gets equal or more than its share of the CPU when our application is "sleeping". In other words, since our application process is sleeping for more time than it is busy, it always preempts the competing process whenever it is ready to execute, and incurs no delays due to CPU sharing.

- $busytimephase > idletimephase$: In this situation, the competing process cannot get its entire fair share of the CPU when the application process is idle. Hence the application will observe an increased execution time due to processor sharing. Consider a segment consisting of an *idletimephase* followed by a *busytimephase*. The CPU must be shared for *busytimephase* - *idletimephase* seconds and the corresponding execution time will be doubled. Hence the increase in execution time will be *busytimephase* - *idletimephase*. Therefore, the fraction by which the overall execution time will increase is:

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

The above cases are also illustrated in Figure 2.

We now revisit the assumption that the idle times must be short. The reason is that schedulers remember past usage only over a short window of time. A scheduler attempts to give equal CPU time to two competing processes by allocating a higher fraction of CPU in the near future to a process that had to relinquish its time slice because it entered a sleep phase. However, a scheduler does not carry this "credit" indefinitely. If a process enters a very long sleep phase, it will not be fully compensated for the share of the CPU it did not use. The length of this sliding *epoch* during which the scheduler attempts to give a fair share of the CPU to all processes is implementation specific. For the standard FreeBSD implementation [9], this period is roughly 5**load-average* seconds. We omit

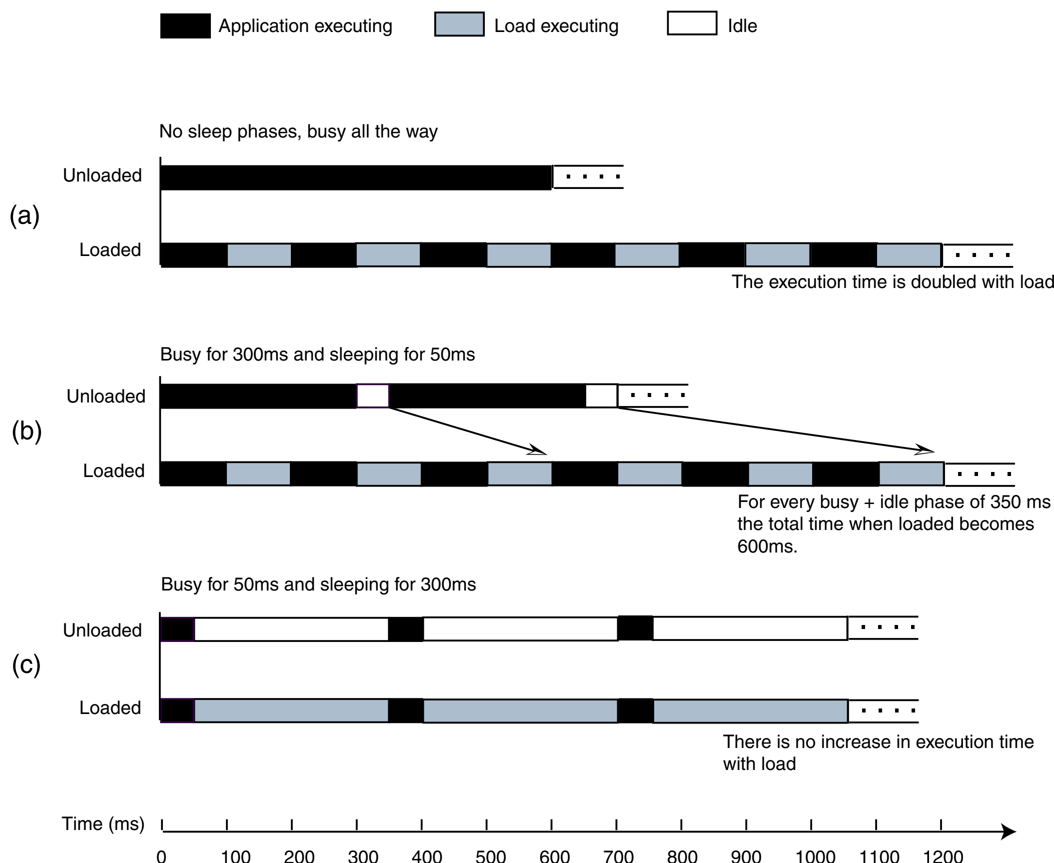

Figure 2: CPU usage during application execution on a dedicated compute node and with a competing load. (a) The application uses the CPU continuously. The execution time is doubled with load. (b) The application uses the CPU for 300 milliseconds and sleeps for 50 milliseconds. The execution time increases by the factor (300-50/350) with a competing load. (c) The applications uses the CPU for 50 milliseconds and then sleeps for 300 milliseconds. The application execution time is unaffected by a competing load.

the analysis for the cases when the sleeps are large, but it is discussed in [21]. In most parallel applications, the sleep phases are short from this standpoint.

The general result is as follows. Suppose an application executes as a sequence of busy and idle CPU phases with average durations of *busytimephase* and *idletimephase* over all the epochs during execution. An epoch length is implementation dependent with a representative value of 5**load-average* for FreeBSD. If the same application is executed with a competing CPU bound process, then:

If $busytimephase <= idletimephase$, the execution time of the application is unchanged.

If $busytimephase > idletimephase$, the execution time increases by a factor:

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

Execution time with two or more competing loads or for a given load average can be predicted in a similar fashion, but a discussion is omitted for brevity.

2.2 Execution with one shared communication link

We analyze the impact on execution time if a network link has to be shared or the performance of a link changes for any reason. The performance of a network link is characterized by the latency and bandwidth observed by an application communicating over the link. 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 is the difference in 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:

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

This equation will be used to predict the increase in execution time when the effective bandwidth and latency on a communication link change.

3 Framework for performance prediction

The procedure employed for performance prediction proceeds as follows. The target application is executed on a controlled testbed and system level activity generated by it is measured. These measurements are processed to infer program level activity such as CPU busy-idle cycles and message exchange sequences. The program level activity constitutes the parameters for the general performance prediction model developed in Section 2. The result is a customized model that can predict the performance of a specific application under given network conditions.

3.1 Application message exchange sequence

Our framework requires the knowledge of the size and number of messages exchanged between any two executing nodes. A simple way to determine the message sequence is to instrument the code, or use a profiling library. However, one of the goals of this project is to be able to determine the message exchange pattern without access to the source code or special libraries. To achieve this, the traffic between all pairs of nodes is monitored with *TCPDump* utility available on Unix systems, and the TCP packet sequence is analyzed to determine the application level message sequence. This procedure is not the topic of this paper but has been validated in [14]. We simply state that, in our experience, the two methods give functionally identical results. In this paper we present and use the message sizes obtained from using an MPI profiling library since they are exact. However, employing the TCP packet analysis method yields virtually identical results.

3.2 CPU utilization pattern

Our framework requires the knowledge of an application's sequence of CPU busy and idle phases while executing on a testbed. For this purpose, a CPU probe program based on Unix *top* utility was developed. The CPU was probed every 20 milliseconds to check whether our application was actively executing. This provides the sequence and length of busy and idle phases with a granularity of 20 milliseconds. For reference, a typical value of CPU time slice given by the scheduler to a process is 100 milliseconds. Our CPU probe can also obtain the CPU utilization as measured by the Unix kernel over a specified interval of time. This gives us two methods to measure the CPU utilization of an application. As a practical matter, we used results from direct probing, but if there was a significant difference between the results from the two methods, the specific experiment was repeated.

CPU probing provides us the lengths of the CPU busy and idle phases as well as the overall CPU utilization. The next goal is to determine the fraction of time spent by the CPU on computation and communication. As noted earlier, we can determine the sequence of messages exchanged by a pair of nodes. Next, we experimentally determine the observed user level latency, say *L*, and bandwidth, say *B*, on our testbed. The time to send or receive a message of size *msgsize* can then be computed as *L* + *msgsize/B*. Hence the total time to send all messages, and the fraction of time spent in communication, can be determined.

3.3 Prediction of execution time with CPU and link sharing

The final goal is to predict the execution time under given network conditions. We developed the theoretical basis for such prediction in section 2. The parameters of this performance prediction model are CPU busy and idle sequence, CPU percentage utilization for computation and communication, and the sizes and sequence of messages exchanged between pairs of nodes. We have explained how each of these is obtained, hence our prediction model is complete.

For prediction of execution time in a new environment, the load average on the nodes and the effective latency and bandwidth on the links must be known. For the cases where only one node has competing loads, or a single link is shared, the expected execution time can be directly computed from the model in section 2. When one node has a competing load *and* one of the links connecting it is shared, then the model is applied as follows. First, the expected execution time is computed assuming only that a link is shared, that is, the node sharing is ignored. Along with the increase in execution time for this scenario, the increase in the average CPU busy time, and the increase in CPU utilization, are also computed. The increase in computation time due to competing loads is then computed using the predicted CPU behavior with a

shared link as obtained above as the starting point. The result is the predicted value of execution time with a busy link and a busy node.

We only consider the cases when one node has competing loads, and additionally when one connecting link to this node is shared. We will discuss the issues in generalization of this method later in this paper.

4 Experiments and results

For the validation of our prediction framework, we conducted a set of experiments with NAS Parallel benchmarks [1]. The codes used are EP (Embarrassingly Parallel), BT (Block Tridiagonal solver), CG (Conjugate Gradient), IS (Integer Sort), LU (LU solver), MG (Multigrid), and SP (Pentadiagonal solver). Experiments were performed on a testbed of 500 MHz, Pentium 2 nodes running FreeBSD. The nodes are connected by a 100 Mbps switched network. Experiments were performed with the MPI implementation of benchmarks and the environment runs the MPICH implementation of MPI. All benchmarks were compiled with class A size for 4 nodes and executed on 4 nodes. *g77* and *gcc* (Fortran 77 and C compilers from GNU) were used for compilation. The end-to-end latency and bandwidth observed on this network were 400 microseconds and 70Mbps, respectively. The *dummynet* toolkit [12] was employed to control the nominal bandwidth on network links, e.g., to make a 100Mbps link act like only 10Mbps was available.

The programs were first executed with no other activity on the testbed, and measurements were made to infer the various program properties as discussed in earlier sections. The summary of results is presented in Figure 3. We observe that the programs show a wide diversity in all aspects of resource utilization.

Subsequently, the same benchmarks were run on the same testbed under the following three scenarios:

1. A synthetic CPU intensive process was run on one of the nodes during benchmark execution.
2. The available bandwidth on one of the links was reduced to a nominal 10Mbps with *dummynet* during benchmark execution.
3. A synthetic CPU intensive process was run on one node and the bandwidth on a connecting link was reduced to 10Mbps during benchmark execution.

For each of these scenarios, the execution time was predicted using the framework developed in this paper, and the results were compared with measured execution time. The results are charted in Figure 4.

The figure shows normalized predicted and measured execution times based on a value of 100 for execution with no load for each benchmark. We observe that the predicted and measured values are fairly close. The prediction error is in the range of 0-7.8% with an average of 2.3% for execution with one compute load, 0.7-7.4% with an average of 2.0% for execution with one busy link, and 0.9-11.8% with an average of 5.7% for execution with one compute load and one busy link. We consider the prediction errors low given the nature of the measurements and the methodology. Hence the basic conclusion is that the framework developed in this paper can predict the execution time effectively for such scenarios.

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.

- **Loads and traffic on all nodes and links:** We presented results when only one node and/or link is facing competition for resources. To estimate execution time accurately when several nodes and links have different resource availability will require an analysis of the synchronization structure of the program This is currently being researched in this project.
- **Execution on a different network from where an application was prototyped:** If the relative execution speed between the two types of nodes, and the latency and bandwidth of the new network can be inferred, a prediction can be performed. This task may be trivial, e.g., when moving between nodes of similar architectures, but may require additional measurements on the new network, e.g., if the new nodes have a fundamentally different architecture such as a vector node.
- **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. However, our model does not account for sharing of bandwidth by different communication streams within the application which can be an important factor in wide area computing.
- **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

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

Figure 3: Measured execution characteristics of NAS benchmarks

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 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 Related Work

This research is in the context of shared metacomputing environments pioneered by Globus [7] and Legion [8]. These systems provide support for a wide range of functions, such as resource location and reservation, authentication, and remote process creation mechanisms. Our research is in the broad area of resource selection and management in shared clusters and metacomputing environments.

A number of resource management systems support the selection of computation resources, some examples being Condor [10] and LSF(Load Sharing Facility). However, the selection of communication resources can be just as important and that introduces several new challenges. A number of systems have been developed to measure and forecast network and CPU availability [5, 11, 15, 23]. The main goal of our research is to be able to predict application behavior once a forecast of the CPU and network availability is known. Hence, this research complements network measurement and prediction research. An alternate approach is integrated network measurement and adaptation systems such as [4, 20] that are designed for a particular class of applications.

Several projects have addressed application scheduling on shared networks [2, 16, 22]. They target specific classes of applications and assume a simple, well defined structure and resource requirements for their application class. The focus of this research is to quantitatively measure the resource requirements of applications so that more precise algorithms can be used to match the application needs and available resources.

The pattern of usage of MPI in NAS benchmarks has been reported in [19] based on instrumenting the MPI library. A key feature of our approach is that all measurements are made at the system level and hence no instrumentation is necessary. This research also relates to task scheduling with CPU and communication constraints [3, 18] and performance prediction for parallel systems [6, 13].

7 Concluding remarks

This paper demonstrates that detailed measurements of the resources that an application needs and uses can be used 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 ex-

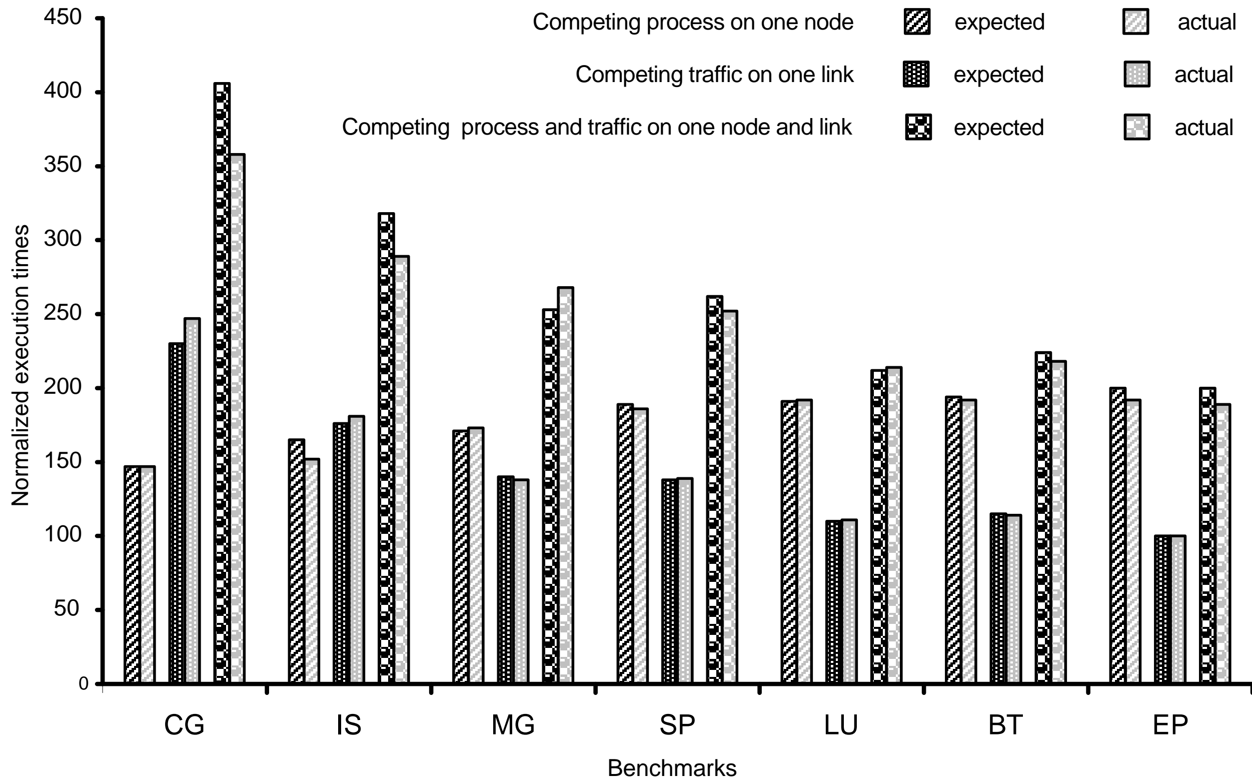

Figure 4: Comparison of predicted and measured execution times in different resource sharing scenarios. The execution time with no load is normalized to 100 units for each program.

periments, we observed that the framework was remarkably accurate in predicting the execution time of the programs in the NAS parallel benchmark suite with competing loads and traffic.

The scope of this paper is limited to competition for resources on a single link and/or a single node. However the methods employed in this research can form the basis for more general prediction frameworks. We believe that prototyping and modeling application execution characteristics is an important component of resource selection and resource management for shared computing environments and this paper makes a clear contribution in that direction.

8 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 Texas Advanced Technology Program under grant number 003652-0424 and the University of Houston's Texas Learning and Computation Center.

We also wish to thank other members of our research group, in particular Amitoj Singh, Srikanth Goteti and Mala Ghanesh, for their contributions to this research.

References

[1] BAILEY, D., HARRIS, T., SAPHIR, W., VAN DER WIJNGAART, R., WOO, A., AND YARROW, M. The NAS Parallel Benchmarks 2.0. Tech. Rep. 95-020, NASA Ames Research Center, December 1995.

[2] BERMAN, F., WOLSKI, R., FIGUEIRA, S., SCHOPF, J., AND SHAO, G. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing '96* (Pittsburgh, PA, November 1996).

[3] BHATT, P., PRASANNA, V., AND RAGHAVENDRA, C. Adaptive communication algorithms for distributed heterogeneous systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing* (Chicago, IL, July 1998).

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

[5] DINDA, P. Statistical properties of host load in a distributed environment. In *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (Pittsburgh, PA, May 1998).

[6] FAHRINGER, T., SCHOLZ, B., AND SUN, X. 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).

[7] FOSTER, I., AND KESSELMAN, K. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications 11*, 2 (1997), 115–128.

[8] GRIMSHAW, A., AND WULF, W. The Legion vision of a worldwide virtual computer. *Communications of the ACM 40*, 1 (January 1997).

[9] LEFFLER, S., MCKUSICK, M., KARELS, M., AND QUARTERMAN, J. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1990.

[10] LITZKOW, M., LIVNY, M., AND MUTKA, M. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems* (San Jose, California, June 1988).

[11] LOWEKAMP, B., MILLER, N., SUTHERLAND, D., GROSS, T., STEENKISTE, P., AND SUBHLOK, J. A resource query interface for network-aware applications. In *Seventh IEEE Symposium on High-Performance Distributed Computing* (Chicago, IL, July 1998).

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

[13] SCHOPF, J., AND BERMAN, F. Performance prediction in production environments. In *12th International Parallel Processing Symposium* (Orlando, FL, April 1998), pp. 647–653.

[14] SINGH, A. Discovery of message passing sequences in parallel applications based on network measurements. Master's thesis, University of Houston. In preparation.

[15] STEMM, M., SESHAN, S., AND KATZ, R. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems* (Monterey, CA, June 1997).

[16] SUBHLOK, J., LIEU, P., AND LOWEKAMP, B. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Atlanta, GA, May 1999), pp. 163–172.

[17] SUBHLOK, J., VENKATARAMAIAH, S., AND SINGH, A. Characterizing NAS benchmark performance on shared heterogeneous networks. In *11th International Heterogeneous Computing Workshop* (April 2002).

[18] SUBHLOK, J., AND VONDRAN, G. Optimal latency–throughput tradeoffs for data parallel pipelines. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures* (Padua, Italy, June 1996), pp. 62–71.

[19] TABE, T., AND STOUT, Q. The use of the MPI communication library in the NAS Parallel Benchmark. Tech. Rep. CSE-TR-386-99, Department of Computer Science, University of Michigan, Nov 1999.

[20] TANGMUNARUNKIT, H., AND STEENKISTE, P. Network-aware distributed computing: A case study. In *Second Workshop on Runtime Systems for Parallel Programming (RTSPP)* (Orlando, March 1998).

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

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

[23] WOLSKI, R., SPRING, N., AND PETERSON, C. Implementing a performance forecasting system for metacomputing: The Network Weather Service. In *Proceedings of Supercomputing '97* (San Jose, CA, Nov 1997).