Empirical Evaluation of Shared Parallel Execution on Independently Scheduled Clusters

Mala Ghanesh Sathish Kumar Jaspal Subhlok

Department of Computer Science, University of Houston, Houston, TX 77204

Abstract

Parallel machines are typically space shared, or time shared such that only one application executes on a group of nodes at any given time. It is generally assumed that executing multiple parallel applications simultaneously on a group of independently scheduled nodes is not efficient because of synchronization requirements. The central contribution of this paper is to demonstrate that performance of parallel applications with sharing is typically competitive for independent and coordinated (gang) scheduling on small compute clusters. There is a modest overhead due to uncoordinated scheduling but it is often compensated by better sharing of resources. The impact of sharing was studied for different numbers of nodes and threads and different memory and CPU requirements of competing applications. The significance of the CPU time slice, a key parameter in CPU scheduling, was also studied. Application characteristics and operating system scheduling policies are identified as the main factors that influence performance with node sharing. All experiments are performed with NAS benchmarks on a Linux cluster. The significance of this research is that it provides evidence to support flexible and decentralized scheduling and resource selection policies for cluster and grid environments.

1. Introduction

Shared networks, varying from workstation clusters to computational grids, are an increasingly important platform for high performance computing. Such distributed computation environments are normally space shared, i.e., one application executes on a set of nodes to completion before the nodes are assigned to another application. Gang scheduling [6] is a technique used to provide fair sharing with space scheduling whereby all threads belonging to an application are simultaneously scheduled on a set of nodes and simultaneously swapped out. In both of these models, only a single application executes on a group of processors at a given time. We will refer to this basic approach as *gang scheduling* in this paper. This is the most common usage model for parallel computing but it has significant shortcomings in terms of performance and flexibility.

- **Performance:** Gang scheduling can lead to underutilization of nodes in two ways. First, in some instances, the number of available nodes may not match the number of nodes needed to execute an application. In such a situation, many nodes must stay idle while enough nodes become available to execute a parallel application. Second, a single application may not use computation and communication resources on the nodes efficiently. As an example, for some applications, the CPU is idle a large percentage of the time because of communication, I/O, user interactions, or inherent load imbalance. These aspects are discussed in more detail in [8].
- **Flexibility:** Gang scheduling implies control of all nodes by a single resource scheduler or queuing system. This becomes increasingly difficult for grid computations that may span several clusters controlled by different organizations. For example, the model of self scheduling of applications based on information about available resources pioneered by the AppleS project [4] is difficult to implement if a single resource manager controls access to all nodes in a system.

The problems associated with gang scheduling can be avoided by allowing multiple parallel applications to time share nodes based on local operating system scheduling on each node. The main reason such independent scheduling is rarely used for high performance applications is the implicit assumption that most parallel applications cannot execute efficiently if individual threads are scheduled independently by the operating systems on the nodes. There are good reasons for such behavior. In general, a pair of processes has to execute concurrently to communicate or synchronize. This is intuitively clear for blocking communication, but it is often the case for non-blocking commu-

Contact E-mail: *jaspal@uh.edu*

nication also for implementation specific reasons such as buffer reservation messages and flow control. Independent scheduling of threads of a parallel application implies that when one thread is swapped out by the operating system for time sharing, other threads on other nodes can get blocked. Since every thread is assigned a CPU time slice independent of others, application execution can deteriorate dramatically because of the multiplicative effect potentially caused by every context switch on every executing node. Intuitively, independent scheduling is expected to be effective for coarse-grained applications but not for fine-grained applications with frequent communication or synchronization operations.

The main result of this paper is that for a broad range of applications on small clusters, independent scheduling is at least competitive with gang scheduling. That is, the application performance is often similar whether multiple parallel applications timeshare a set of nodes individually controlled by the operating systems, or if all threads of one application are scheduled collectively on groups of nodes with gang scheduling. In many cases, sharing with independent scheduling performs better because it offers better utilization of resources. We also present results that relate the performance of shared execution with independent scheduling to the key factors that it depends on: number of threads and nodes, memory requirements, operating system time slice quantum, and the number of sharing applications on a node.

These results suggest that scheduling and resource allocation models that are decentralized or employ concurrent scheduling [8] are perhaps more valuable than is currently believed. For example, models where applications independently select execution nodes based on best effort network and CPU information [4, 5, 13, 15] provided by tools like NWS [16] and Remos [11] are promising because a loosely controlled execution environment can provide acceptable application performance. This is especially important for grid environments that employ multiple distributed resources [7, 9, 10] since fine coordination of multiple clusters in different administrative domains can be difficult to impossible. This research essentially promotes the value of flexible resource selection mechanisms that allow independent scheduling of nodes for parallel computing.

2. Performance of independent and gang scheduling

We discuss the expected performance when a long running application has to fairly share a set of nodes with other applications. For our discussions we will focus on the performance of an *application of interest* that has to share nodes with a *competing application* or a *competing load*. In this paper we assume that the competing load is always CPU-hungry so that the results can be interpreted in a meaningful way. For introductory discussion in this section, we assume single processor executing nodes, the number of application threads equals the number of nodes, and there is one competing load thread on every node. Many of these assumptions will be relaxed later in this paper. We will also present results for multiple CPU nodes and for multiple threads per node. We now discuss node sharing performance for gang scheduling and for uncoordinated independent scheduling.

In the case of gang scheduling, all threads associated with one application will execute simultaneously for an interval of time and then be swapped out for an equal interval of time. The application execution time will double as compared to execution without sharing, in addition to the overhead of context switching. Assuming that the time for which one application executes is much larger than the context switch overhead, the overall execution time (same as turnaround time in this context) will approximately double for a long running application. In this paper we employ such a "doubled" execution time as the reference execution time with gang scheduling. A real implementation will have additional overheads that are ignored. Hence our estimates are essentially optimistic execution times for gang scheduling and we will treat them as such. However, we also believe that gang scheduling overheads should be relatively small in a good implementation and hence our optimistic estimates should be realistic for long running applications.

We now consider the case of independent scheduling. We restate that we are considering the case where one thread of the application of interest and the competing load is assigned to every node. The scheduler on each node will attempt to assign equal time slices to the application of interest and competing loads, independent of scheduling on other nodes. If the application of interest is compute intensive with no communication, then the threads on each node will simply take twice as long to complete and the overall execution time of the application of interest will double. However, when an application has a significant communication component, estimating performance is much more complex. For a pair of threads to perform a synchronous data exchange, both must be actively executing at the same time, but since each thread is scheduled independently, it is difficult to predict synchronization waits. The communication delays due to uncoordinated scheduling can have a cascading effect on the performance of the entire application because of data and control dependencies.

Following is a discussion of the main factors that determine the performance of an application with sharing of independently scheduled nodes. We separately discuss the role of the node scheduling policy and application characteristics.

2.1. Node scheduling policy

If the application threads on each node of a workstation cluster were scheduled in a rigid round-robin fashion, a parallel application will get slowed down by an unacceptably large factor, and theoretically may never complete execution. The reason is that a pair of processes on different nodes that need to perform a synchronous data exchange may never be scheduled at the same time. In practice, this means potentially long delays on every communication step. Fortunately, such rigid policies are not used in practice. CPU schedulers make an effort to provide a fair share of CPU to all active processes. If a thread is blocked waiting for a communication operation, it is swapped out, but its priority in the waiting queue increases. Hence, it is likely to be immediately scheduled in the future when the thread on another node with which it needs to communicate becomes active. This feature significantly reduces the slowdown due to communication and synchronization waits. Many parallel applications follow the Bulk Synchronous Processing (BSP) model where all nodes repeatedly perform a computation operation followed by a communication operation. In such cases the processing nodes often self synchronize, where threads of the same application are scheduled on all nodes at about the same time because of the lock-step nature of execution. Related work has analyzed this behavior [1, 3].

An important aspect of a node scheduling policy is the CPU time slice quantum for which a process executes before the CPU is reassigned to another process in the waiting queue. We will discuss the performance aspects of different CPU time slice quanta and their relationship to shared application performance along with experimental results.

2.2. Application characteristics

The extent of slowdown of an application due to CPU sharing depends to a large extent on the basic execution characteristics. We discuss the major relevant application features:

- *Communication volume and frequency:* If a parallel application is compute intensive and does not have a significant amount of communication, there will be no significant impact of asynchronous scheduling of threads on different nodes. Slowdown of such an application with sharing will be similar for independent scheduling and gang scheduling. In general, the additional slowdown due to communication and synchronization is likely to be higher for fine grain applications and for applications that exchange significant amounts of data.
- *Communication and synchronization pattern:* Beside the rate of messages and bytes exchanged by nodes, the communication pattern is also an important factor that determines the slowdown due to node sharing. For example, if one node only communicates with one or two of its logical neighbors, the communication and synchronization related slowdown is likely to be much less than when each node communicates with every other node. Similarly, applications with a regular communication pattern are likely to perform better with sharing than applications with an irregular communication pattern.
- *CPU utilization:* Some parallel applications have relatively low processor utilization when executing exclusively on a cluster. When such an application must share the CPU with another application, the competing application will get the bulk of its fair share of the CPU from the times that the CPU would have gone idle otherwise. Hence the slowdown of the application of interest is likely to be relatively low.

3. Experiments and results

We performed a set of experiments with Class-B NAS benchmark suite [2] on a small cluster to measure the slowdown associated with node sharing with independent scheduling. The computation cluster used for the experiments is a 100Mbps fast Ethernet based test bed of 1.8GHz Pentium Xeon Duos running Linux and MPICH implementation of MPI. We used the following NAS benchmarks in our experiments: BT (Block Tridiagonal), CG (Conjugate Gradient), EP (Embarrassingly Parallel), IS (Integer Sort), LU (LU decomposition), MG (Multigrid) and SP (Scalar Pentadiagonal). Each of the NAS codes was compiled with *g77* or *gcc* for 4, 8, or 16 threads and executed on 4 or 8 Dual processor nodes. SP and BT benchmarks used 9 threads instead of 8 because of the nature of the codes and the number of executing nodes was adjusted as appropriate.

The results in this paper compare the measured slowdown due to node sharing with independent scheduling with the estimated slowdown for node sharing with gang scheduling. Slowdown for an application is defined as:

$$PercentageSlowdown = \frac{(B-A)*100}{A}$$

where A is the execution time in dedicated mode and B is the execution time in shared mode.

The gang scheduling estimates are based on the simple concept that an application on a single CPU node will take twice as long to execute if it has to share the CPU with another application as it will have the CPU on all nodes half the time. However, in our experiments we are using dual processor compute nodes. We describe the two kinds of experiments that were conducted and how the gang scheduling slowdown was estimated.

1. *One application thread per node:* In this set of experiments, one application thread per node was executed for reference dedicated execution. For shared execution with independent scheduling, this application thread was executed simultaneously with 2 competing load threads on each node. (Note that no significant slowdown is expected with one competing load since each program can get one dedicated CPU.) In this scenario, the slowdown with gang scheduling is estimated to be 50% since every thread including the application thread will be scheduled $2/3^{rd}$ of the time and idle $1/3^{rd}$ of the time. We also state that 50% is the *nominal* expected slowdown for independent scheduling since that is the slowdown for a dedicated compute intensive application based on fair CPU sharing.
2. *Two application threads per node:* In this set of experiments, two application threads per node were executed for reference dedicated execution. For shared execution with independent scheduling, these application threads were executed simultaneously with one competing load thread on each node. Once again, the slowdown with gang scheduling is estimated to be 50% since every thread, including the application threads, will be scheduled $2/3^{rd}$ of the time and idle $1/3^{rd}$ of the time. We again state that 50% is the *nominal* expected slowdown for independent scheduling since that is the slowdown for a dedicated compute intensive application based on fair CPU sharing.[1]

3.1. Performance across different numbers of threads and nodes

In order to analyze the impact of node sharing with independent scheduling, NAS benchmarks were executed with and without artificial competing load applications. For these experiments, the competing load is a synthetic CPU intensive program that uses little memory and has no communication or I/O. The slowdown for each benchmark due to competing loads was measured for each of the following scenarios:

- 1 application and 2 load threads per node on 4 nodes.
- 1 application and 2 load threads per node on 8/9 nodes.
- 2 application and 1 load thread per node on 4/5 nodes.
- 2 application and 1 load thread per node on 8 nodes.

The above combinations represent the minimum number of load threads needed to create competition for CPU resources. The reference unshared execution uses the same configuration without the load threads and the estimated slowdown with gang scheduling in every case is 50%. The results are presented in Figure 1. We point out the general observations and continue discussion relating to specific benchmarks, CPU and memory loads, and CPU time slices, in the remainder of this section.

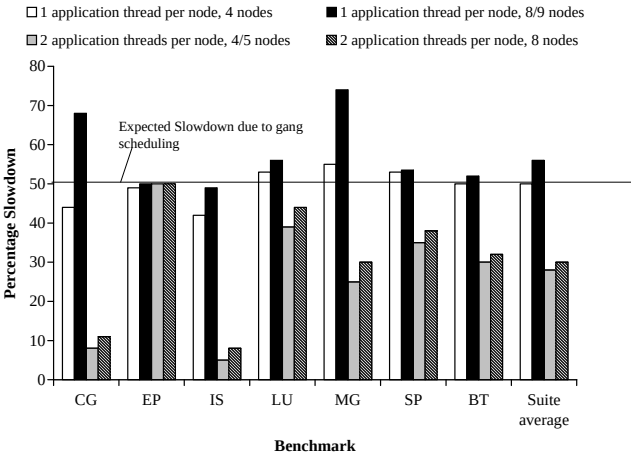

Figure 1. Slowdown of the NAS benchmarks due to competing compute loads.

- *Slowdown with independent scheduling is generally less than or comparable to the estimated slowdown with gang scheduling*. A slowdown less than 50% reflects better performance than gang scheduling. The average slowdown for the entire benchmark suite (rightmost group of bars) is below or around 50% for different combinations of threads and nodes. The implication is that, in most cases, the additional overhead due to communication between asynchronous threads with independent scheduling is overcome with better CPU utilization. That is, often the competing loads derive a large part of their CPU usage during times when the benchmark application is blocked on synchronization waits which does not affect the performance of the benchmark.
- *Slowdown is greater for the case of 1 application thread per node than for the case of 2 application threads per node.* This is clearly observed for the case of 4 nodes as well as 8 nodes from Figure 1. The reason is as follows. In the case of 1 application thread, we have added 2 load threads, and in the case of 2 application threads, we have added 1 load thread. The load threads need the CPU 100% of the time while the CPU demand of application threads varies. Hence there is

1 This analysis assumes that an application with 2 threads out of a total 3 on the node will be awarded the system 2/3rd of the time in gang scheduling. While this is debatable, the analysis does compare gang scheduling and independent scheduling fairly and meaningfully.

more slowdown when 2 CPU hungry load threads are competing with an application thread as compared to the case of only 1 CPU hungry load thread.

- *Slowdown is greater for larger number of nodes.* It is clear from Figure 1 that slowdown is higher for 8 nodes than for 4 nodes. In order to gain insight into the impact of the size of a cluster, we ran the benchmarks on a separate cluster on 4, 8 and 16 nodes with one thread per node - the range of nodes over which the benchmarks scale well and run in a reasonable amount of time. Since this experiment was done on a separate cluster from all others, the results are shown at the end of the paper, to avoid confusion, in Figure 5. It is clear that slowdown with independent scheduling increases as the number of nodes is increased from 4 to 16 across all benchmarks. The increase is faster for some applications, such as CG, IS and MG, and very little for others, particularly EP and IS. A larger number of nodes applied to the same computation implies more frequent global communication and synchronization operations since the application executes faster. Also, each operation typically involves a larger number of nodes. These factors have the potential of making shared execution slower for independent scheduling but are not relevant for gang scheduling. The conclusion is that performance of independent scheduling deteriorates slowly with cluster size. Hence the approach is suitable for small to midsize clusters.

3.2. Performance across NAS benchmarks

We observe from Figure 1 that the slowdown varies widely across the programs in the NAS benchmark suite. To understand this, we measured the basic runtime characteristics of NAS programs during execution on a dedicated testbed of 4 nodes. Vampir profiling library [12] was used to monitor messages sent by each executing node and CPU probes that we have developed [14] were used to measure the average CPU utilization, i.e., the percentage of time the CPU was busy executing the application. The results are presented in Table 1. We now point out a few observations that relate the shared performance of NAS benchmarks to their execution characteristics.

From Figure 1 we see that EP benchmark exhibits around 50% slowdown in all cases. Since EP is a compute bound program with no communication, it is expected that it will show the same slowdown whether gang scheduling or independent scheduling (or any other fair way of sharing the CPU among threads) is used. From Table 1 we see that the CPU utilization for EP is nearly 100%.

Let us focus on the numbers corresponding to two application threads per node in Figure 1. We observe that CG and IS show little slowdown, in the range of 5-10%, for execution on 4 or 8 nodes. We also see from Table 1 that IS and CG show the lowest CPU utilization for dedicated execution, around 42% and 59%, respectively. The explanation is that the single competing load in these cases is able to get most of its fair share of the CPU during the times the application threads would have been blocked for synchronization. Hence, the application suffers little slowdown. In general, we observe a strong correlation between the CPU utilization for dedicated execution shown in Table 1 and the slowdown for the case of 2 application threads and one load thread shown in Figure 1. Note that the above correlation is not apparent for the cases where a single application thread is executing with two load threads, also shown in Figure 1. As discussed previously, load threads need the CPU 100% of the time, so there is much more competition for the CPU when there are 2 load threads.

Bench mark	Exec Time (Sec)	Communication		Computation	
		Rate of msg dispatch from each node (msgs/s)	Rate of data dispatch from each node (KB/s)	Avg % CPU Utilization	Memory Utilization (MB)
CG	417.2	19.1	2700	59.4	110
EP	228.6	0	0	99.5	0
IS	81.2	1.3	4333	42.3	121
LU	511.7	98.8	577	91.6	50
MG	37.0	89.3	3020	69.0	114
SP	881.3	5.4	1343	77.2	90

Table 1. Execution characteristics of the NAS benchmarks: 4 threads run on 4 nodes

For the cases of 1 application thread and 2 load threads, the slowdown is the highest for CG, LU and MG. We observe from Table 1 that these three benchmark programs also have the highest frequency of message exchange, and CG and MG are among the programs with the highest volume of data exchange. It is apparent that, in this case, the slowdown is dominated by the overheads of message and data exchange. As noted earlier, CG and MG exhibit the maximum increase in slowdown going from 4 to 8 nodes. Clearly, the frequency and volume of communication is a key factor that determines performance with independent scheduling and its scalability.

3.3. Performance across CPU time slice quanta

An important aspect of processor scheduling for parallel applications is the nominal CPU time slice quantum given to an application for execution by the operating system. When multiple jobs are in the ready queue, an application may execute for the entire time slice quantum or it may be swapped out before the end of the time slice if it is blocked or if another application with a higher priority joins

the ready queue. A larger time slice quantum implies that a thread may have to wait for a longer time for another thread with which it needs to communicate to be scheduled. However, it also means that a message exchange is less likely to be interrupted because of a thread being swapped out after completing a time slice quantum.

The version of the Linux operating system we used (Redhat 7.2, kernel version 2.4.7-10) has a default time slice quantum of 50 milliseconds that was used for the results presented so far. In Figure 2, we present results with varying time slice quanta, specifically 30, 50, 100, and 200 milliseconds. We verified that the execution time for all the benchmarks without a competing load was virtually identical for all values of time slice quanta.

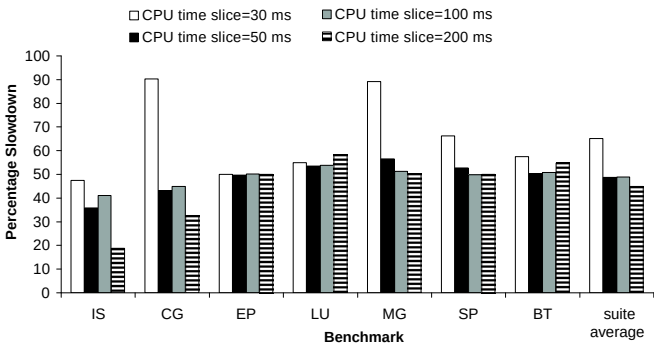

Figure 2. Slowdown for various CPU time slice values: 4 benchmark threads run on 4 nodes.

We note that 30 ms time slice quantum has the worst performance for all benchmarks. The performance for 50ms, 100ms, and 200 ms are close to each other for most programs. The best performance is achieved with a 50ms time slice quantum for some applications and a 200ms time slice quantum for others. The average performance is the best for a 200ms time slice, although it is only slightly better than the average performance for 50ms and 100ms time slices. On LU and BT benchmarks the trend is towards worse performance as the time slice is increased from 100ms to 200ms. However, IS and CG, two of the more communication intensive programs, show the best performance with a time slice of 200ms.

On the whole, it appears that the choice of 50ms to 100ms time slice quantum common in operating systems is a reasonable one for time sharing parallel programs (even though such programs are unlikely to have been a design consideration). Some communication intensive programs, however, may achieve better sharing performance with a larger time slice quantum. There appears to be little scope of benefit with a time slice smaller than 50ms on this hardware. We also observed that when different time slice quanta were used on different executing nodes, the sharing performance was similar to the lower performing of the two time slice quanta, but we have omitted the results for brevity.

3.4. Performance across memory loads

The competing loads employed in the experiments presented in this paper so far consisted of repeated numerical computations without significant memory usage. We separately conducted a series of experiments with loads that allocated a significant amount of memory and periodically scanned the entire data space. The slowdown of different benchmark programs due to competing loads with different memory requirements is presented in Figure 3. Each benchmark program itself has a fixed memory requirement in this suite of experiments, and those are listed in Table 1.

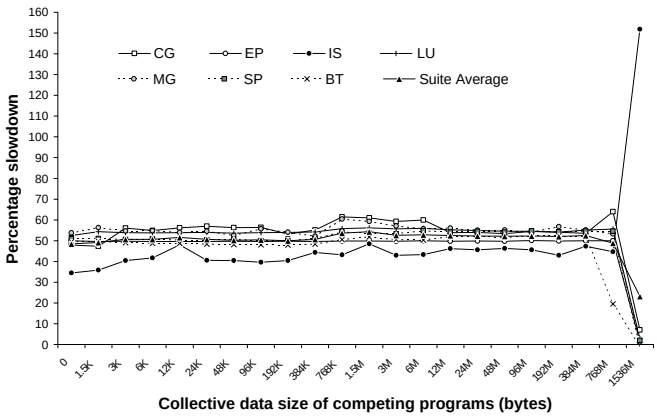

Figure 3. Slowdown under different memory load conditions. 4 benchmark threads run on 4 nodes.

The immediate observation from Figure 3 is that the variation in slowdown is relatively small when the load memory requirement is changed, until we approach the total available memory on the system, which is nominally 1 Gigabyte. When the combined memory requirement of all threads reaches the point where virtual memory has to be employed, the change in performance is drastic. Interestingly, for most benchmarks, the slowdown *decreases* dramatically which appears to be counterintuitive. We believe that the reason is that the load threads are swapped out frequently due to page faults giving the benchmark programs a much larger fraction of the CPU time.

Figure 4 zooms in on the impact of changed memory requirement in the range of cache capacities, which is 256K per CPU for the L2 cache on these nodes. When a competing application is using the cache extensively, the benchmark will find the cache "cold" when it is scheduled. This should result in a slowdown but the practical impact seems

to be relatively small. There is a distinct increase in the slowdown across most applications as memory usage of competing applications approaches and exceeds the cache capacity, which corresponds to 512K point in the graph. Some benchmarks show significant performance variations at other points, but an analysis is beyond the scope of this paper and we believe it is related to application specific memory access patterns.

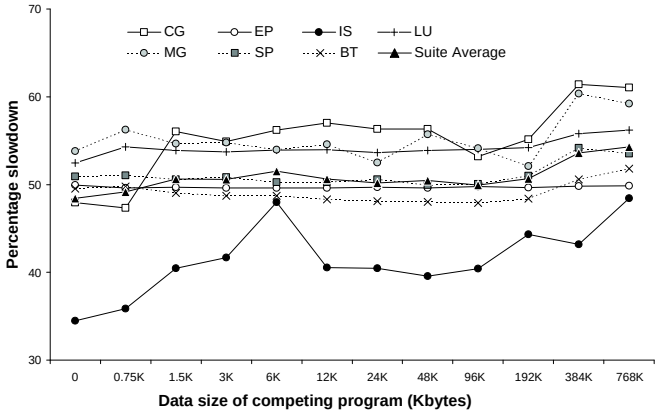

Figure 4. Slowdown under memory loads around cache sizes. 4 benchmark threads run on 4 nodes.

4. Summary and discussion

We summarize the main results of this paper and discuss their significance.

- Sharing a set of nodes with independent scheduling is competitive with gang scheduling for small clusters.
- An application's CPU utilization and communication volume and frequency are the key factors that determine performance with sharing. A lower CPU utilization during dedicated execution implies better sharing performance, while frequent and large message exchanges imply worse sharing performance with independent scheduling.
- The CPU time slice quantum assigned by the operating system is an important factor for shared execution. Common values in the range of 50ms to 100ms are reasonable choices. A lower CPU time slice quantum value uniformly deteriorates performance while the impact of a higher one is application dependent.
- The memory competition is a major factor only if the combined memory requirement of all the threads approaches the total available memory.

Performance of applications with uncoordinated time-sharing on a small cluster of nodes is competitive with gang scheduling for a variety of parallel applications including fine-grained and communication intensive computations represented by the NAS benchmarks. This contradicts with the common assumption that space scheduling or gang scheduling is essential for getting reasonable performance for most parallel applications. The performance in this context also reflects the cluster throughput.

We have not performed actual experiments with gang scheduling, and instead used optimistic estimates without overheads. Our main point is that independent scheduling yields competitive performance as compared to gang scheduling, hence using an optimistic estimate for gang scheduling only makes it stronger. We have used synthetic CPU intensive competing loads for our experiments and analysis. The actual applications may not be as CPU hungry and provide better sharing performance and that has been our empirical experience. Hence, using CPU intensive competing loads is a conservative assumption.

We believe that one of the key reasons for relatively good performance with independent scheduling is that application threads over multiple nodes become automatically coordinated with standard operating system scheduling policies. An executing application process may get swapped out prematurely if it blocks waiting for a peer process to be scheduled, but it gains priority, and is likely to get scheduled immediately again once its peer process is also scheduled. Such a mechanism implicitly leads to scheduling synchronization among application threads which greatly helps communication performance.

It is expected that sharing performance will be different for different computation environments. However, we expect the patterns to be similar unless the processing paradigm is fundamentally different. For example, we assume that when a process is blocked for communication it is removed from the ready queue of processes. Use of busy-waiting would yield different results. Also, although NAS benchmarks represent a large class of scientific computations, there are other very fine grain applications that may perform poorly with independent scheduling. The results in this paper were obtained on a fast Ethernet but we have not observed any qualitative difference with gigabit Ethernet. However, the challenges of sharing are different for other interconnects such as Myrinet. Finally, we have presented results for a small cluster. It appears that the performance with independent scheduling gradually deteriorates for larger clusters but more experiments and simulations are needed to study that relationship.

5. Concluding remarks

The main result of this paper is that sharing of a small cluster by multiple parallel applications with independently scheduled nodes is competitive with sharing with a gang scheduling paradigm. The main reasons are better CPU utilization when multiple applications compete for it, and a lower overhead for communication with asynchronous processing than is generally believed. The results support a flexible approach to scheduling clusters where multiple applications may be mapped to the same set of nodes for improved performance. They also support the use of application controlled resources selection which is important for grid computing where centralized control of resources may not be feasible.

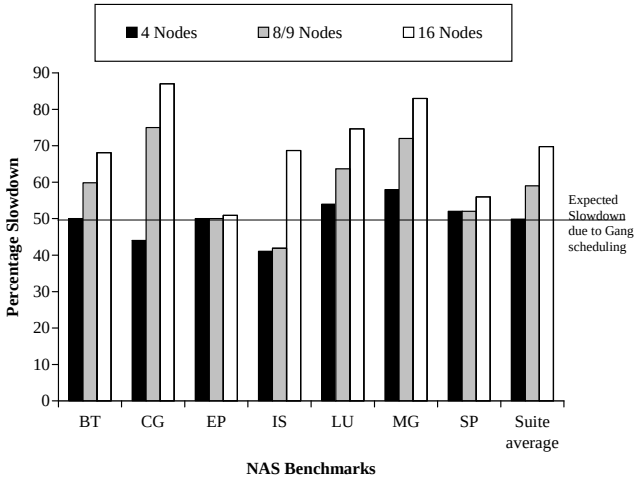

Figure 5. Scaling of slowdown of NAS benchmarks due to competing loads

6. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. ACI-0234328 and CNS-0410797. Support was also provided by the Department of Energy through Los Alamos National Laboratory (LANL) contract No. 03891-99-23 , and by University of Houston's Texas Learning and Computation Center.

References

[1] ARPACI-DUSSEAU, A., CULLER, D., AND MAINWARING, A. 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] 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.

[3] BARAK, A., AND LA'ADAN, O. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems 13*, 4–5 (1998), 361–372.

[4] 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).

[5] 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.

[6] FEITELSON, D., AND RUDOLPH, L. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing* (1992), 306–318.

[7] FOSTER, I., AND KESSELMAN, C. The Globus project: a status report. *Future Generation Computer Systems 15*, 5–6 (1999), 607–621.

[8] FRACHTENBERG, E., FEITELSON, D., PETRINI, F., AND FERNANDEZ, J. Flexible coscheduling:mitigating load imbalance and improving utilization of hetwrogeneous resources. In *International Parallel and Distributed Processing Symposium* (April 2003).

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

[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] NAGEL, W., ARNOLD, A., WEBER, M., HOPPE, H., AND SOLCHENBACH, K. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 12*, 1 (1996), 69–80.

[13] 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.

[14] VENKATARAMAIAH, S., AND SUBHLOK, J. Performance prediction for simple CPU and network sharing. In *LACSI Symposium 2002* (October 2002).

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

[16] 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).