Volunteer Computing on Clusters

Deepti Vyas and Jaspal Subhlok

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

Abstract. Clusters typically represent a homogeneous, well maintained pool of high-end computation resources. This makes them particularly attractive for volunteer computing, where unused compute cycles are utilized for scientific guest applications. Cluster nodes are not idle as often as public PCs, but they are frequently underutilized while actively executing parallel applications. Hence, fully exploiting clusters for volunteer computing requires the ability to efficiently and invisibly steal the unused cycles at a fine grain from the currently running host applications, without slowing them down. In this paper we present measurements on a production compute cluster that show long periods of CPU and memory underutilization patterns that could be used to execute guest applications. Our experiments with NAS benchmarks show that, under the best configuration of Linux, cycles can be stolen with only a 3.6% average slowdown of the host application. This was accompanied by an overall improvement in the system throughput of 38%, when progress of the guest applications was included. We introduce simple guidelines on using clusters for volunteer computing. We also argue for the support of zero priority processes in OS schedulers which could virtually eliminate the impact of volunteer computing on host applications.

1 Introduction

Volunteer computing, also referred to as public-resource computing or global computing, is based on exploiting unused cycles on ordinary desktop computers. The concept was pioneered by SETI@home [1], and is being increasingly employed to solve important real life problems. BOINC [2, 3], a framework to support volunteer computing, is being used by a variety of scientific simulation projects such as protein folding, climate prediction, and biomedical computing. Condor [4] pioneered the employment of idle periods on organizational desktop systems for useful computing. We use the term volunteer computing for all scenarios where a low priority guest application can run on unused resources without significantly impacting high priority host applications. Examples of other projects with similar goals include Entropia [5], OpenMosix [6], and GridMP [7]. Availability of computation and storage resources that can be effectively employed for volunteer computing has been studied in [8, 9].

A growing source of computation power today is compute clusters consisting of 10s to 1000s of processors. In addition to the high performance computing centers, it is becoming increasingly common for individual computational scientists and research groups to maintain their own clusters. In our estimate the combined compute power of all clusters on our campus (University of Houston) is comparable to the combined compute power of all desktops on campus, and we believe this is not uncommon.

Computation clusters are particularly attractive for volunteer computing for a number of reasons.

- Clusters are typically built from high end computing and communication components.
- Clusters typically offer a homogeneous and well maintained pool of processors.
- While many supercomputing centers are heavily used, many clusters are also frequently idle, although the usage of a typical cluster node is certainly higher than a typical home PC. A recent study [10] of one group of clusters for scientific research found that their average usage varied between 7% and 22%.

In this paper we empirically demonstrate the following additional properties of cluster behavior that are relevant to volunteer computing.

1. CPU usage on clusters is frequently not close to the maximum while they are executing parallel scientific applications. The reason is that synchronization delays are fundamental to parallel processing, and increase as a fixed size problem is scaled up to a larger number of processors. For illustration, our experiments with NAS class B parallel benchmarks on 4 nodes show that their average CPU utilization varied from 53% to 100% as listed in Table 1. Further, the average speedup from 4 nodes (8 threads) to 8 nodes (16 threads) was 1.51 implying that the added 4 nodes were used only half as efficiently as the first 4 nodes. Other classes of applications, such as sparse matrix computations, are fundamentally more prone to synchronization delays due to load imbalance. We report on measured usage of a production cluster at the University of Houston that shows average CPU utilization of 64% even though applications are running on the nodes almost the entire time.
2. Usage of cluster nodes shows significant predictability, i.e., computation behavior in the recent past is a good predictor of the usage in the near future. The reason is that clusters are typically employed for long running scientific applications, and node usage for a single application is usually similar over the course of execution.

Benchmark	BT	CG	EP	FT	LU	MG	SP
CPU utilization (%)	90	65	100	53	94	73	81

Table 1. Average CPU utilization of Class B NAS benchmarks on 4 cluster nodes

Sometimes, techniques like backfilling [11, 12] and interstitial computing [13] are used to increase the cluster utilization by scheduling small jobs on idle nodes. Since free cycles are available on many clusters only at a fine grain, a cluster is far more attractive for volunteer computing if guest applications can execute when CPU and memory are being underutilized, not just when the nodes are idle. Scheduling support for such fine-grained cycle stealing has been studied in [14–16]. However, the impact of resource sharing within a cluster node is difficult to predict although related research has addressed some aspects [17, 18].

This paper focuses on fine-grained cycle stealing on Linux, which is the operating system of choice for cluster computing. We demonstrate that execution of low priority guest applications only have a small impact on regular host applications. We also discuss how various system and application factors affect the slowdown of host applications. This information, along with the fact that cluster usage shows significant predictability, helped us develop guidelines for employing volunteer computing on clusters that can minimize the impact on host applications while maximizing the benefit to guest applications. We argue that fine-grain cycle stealing on clusters with negligible impact on host applications is possible, but would require simple changes to the Linux scheduler.

The paper is organized as follows. Section 2 presents results on CPU and memory utilization of a production cluster. Section 3 presents results on cycle-stealing on a Linux cluster and its dependence on system and application factors. Section 4 outlines our approach to volunteer computing on clusters and recommends beneficial changes to OS schedulers, and section 5 contains conclusions.

2 Utilization of clusters

The study presented in this section empirically measures the CPU and memory utilization on cluster nodes when they are busy executing scientific applications. Performance data was collected from a Beowulf cluster at the High Performance Computing Center at University of Houston, one of the most busy clusters on campus. The cluster consisted of 30 Intel Xeon dual processor nodes, running Linux (2.4.21 SMP kernel) with 2Gb RAM. The nodes were interconnected with a Gigabit ethernet network.

The data was collected over a period of 1 month and measurements were made at 5 minute intervals. The information was gathered from various files under the `/proc` file system of each node. CPU and memory utilization of representative nodes is plotted in Figure 1. Several small groups of nodes had very similar usage patterns. The nodes plotted in Figure 1 were not selected randomly, but chosen to represent different patterns. Figure 2 shows a zoomed in CPU utilization representing the first 12 hours of the periods covered in Figure 1 for two of the nodes. The graphs are in descending order of average CPU utilization within each figure.

Following are the main observations from this study of cluster utilization:

- The CPU utilization often shows fluctuation from point to point, as seen in Figure 2 which zooms in on the beginning part of the first two graphs in Figure 1. However, CPU utilization shows remarkable stability when it is considered over windows of several points. The average CPU utilization typically stays in a very narrow band from hours to days, and even weeks, in some cases, as seen in Figure 1. We presume this is a result of the same or similar applications running on the same group of nodes for extended periods of time.
- While nodes show long periods where CPU utilization is high, they also show long periods when CPU utilization is moderate or low. The average CPU utilization of a node varied between 25% to 85% with a mean around 64% and median around 65%.

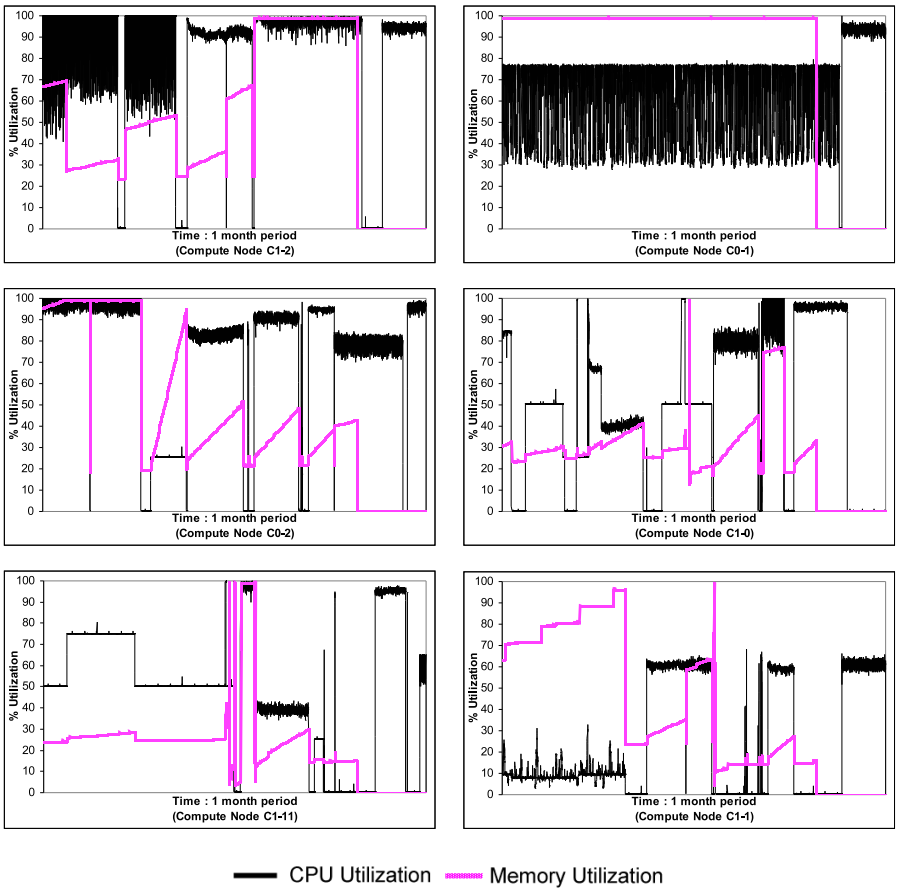

Fig. 1. CPU and Memory utilization of sample nodes of a busy cluster plotted every 5 minutes over a period of 1 month (14 Jun 2005 to 16 Jul 2005)

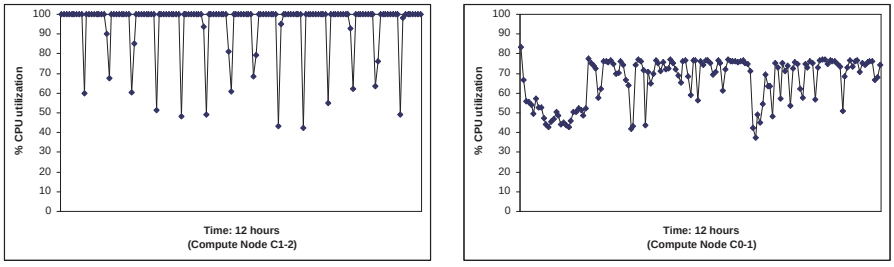

Fig. 2. CPU utilization for selected nodes plotted every 5 minutes over a period of 12 hours

– The memory utilization either stays steady or slowly increases linearly and then drops, over extended periods of time. The memory usage does not exhibit the short term fluctuations of CPU usage. However, we should point out that the reported memory utilization does not necessarily reflect the active set of pages and may include memory that has been released by the application, but is pending release at the system level.
– The average memory utilization can be close to 100% for a node for extended periods of time, but it is frequently around or well below 50% for extended periods of time. The average memory utilization of the nodes varied between 30% and 90% with a mean utilization around 52% and median utilization around 44%.

The main conclusion from this study, that is relevant to volunteer computing, is that cluster nodes show long and predictable periods of low CPU and memory utilization. The implication is that a substantial fraction of resources are available for volunteer computing, and when a scenario with good resource availability is identified, it is likely to continue for hours to days. The reason for such behavior is that clusters are typically employed for long running scientific applications. Hence, even though this study was limited, we expect the conclusions to be valid for other clusters employed for parallel scientific computing.

As pointed out earlier, the particular cluster that was monitored is known to be heavily utilized. The purpose was to investigate available resources while applications are running. Of course, if a cluster node is idle, it is an even more attractive option for volunteer computing (although perhaps not as predictable). The usage of clusters is likely to be higher than the average desktop, and indeed major supercomputing centers are known to be very busy. However, our observation is that smaller clusters often have considerable idle periods. A recent study of a 5 cluster research environment observed that the average time a system was busy ranged from 7.3% to 22% and a large fraction of jobs had a very small memory requirement [10].

We summarize this discussion as follows:

1. Many clusters nodes are idle and not running any applications a substantial fraction of the time. Of course, these can be directly exploited for volunteer computing.
2. When cluster nodes are busy running applications, a substantial fraction of the memory and CPU resources are often not utilized for extended periods of time. These idle resources can be exploited with fine-grain cycle stealing making clusters even more attractive for volunteer computing.

3 Fine grain cycle stealing on clusters

A critical consideration in making a cluster available for volunteer computing is how a high priority host application will be affected when a low priority guest application is stealing unused cycles for execution. Ideally, there should be no impact at all; the guest process should be scheduled only when the host process is blocked, and the guest process should be evicted as soon as the host process is ready to execute again. However, this is difficult to achieve for fine-grained cycle stealing when a host and guest

application are executing concurrently because one of the goals of commercial operating system schedulers is to prevent starvation of low priority processes. Research has shown that it is possible to construct schedulers where the impact on the host application is negligible [14, 16]. However, we are most interested in volunteer computing with mainstream operating systems since installing a new scheduler is not likely to be acceptable. All our experimentation is on Linux since that is the dominant cluster operating system.

The goal of the experiments was to see how to best run guest applications on Linux with minimum impact on host applications. Dependence on system factors such as priority mechanism and scheduler versions, as well as dependence on characteristics of host and guest applications, are also analyzed.

3.1 Experimental setup

Our experimental environment consists of a ten-node cluster. Each node has 1GB of main memory and dual Pentium Xeon processors running at 1.8 GHz. The nodes are connected through a 1 Gbps ethernet switch. The cluster was running Rocks 4.0 Beta and a MPICH 1.2.6 version of MPI. This configuration is representative of small and midsize clusters employed for scientific computing.

To achieve fine-grained cycle stealing, the guest applications were run simultaneously with host applications, but at a lower priority using the UNIX *nice* mechanism. The execution times of the host and guest applications were measured when run individually (dedicated mode), and when run simultaneously (shared mode). Percentage *slowdown*, defined as the percentage increase in execution time when executing in shared mode as compared to dedicated mode, is used to quantify and compare the effect of sharing in different scenarios.

NAS Class B parallel benchmarks were used as host applications and guest applications. Unless otherwise noted, the experiments were run on 4 (dual processor) nodes, and each node ran 2 threads of the host application at normal priority (nice = 0) and 1 thread of the guest application at lowest priority (nice = 19). NAS benchmark EP (Embarrassingly Parallel) was used as the default guest application. The EP program has virtually no communication, and hence it represents a sequential compute intensive application.

3.2 Slowdown on Linux

We study the slowdown of host applications when running with a guest application on Linux, and examine how the slowdown can be minimized. The slowdown for the NAS benchmarks running as host applications, with the compute intensive EP benchmark as the guest application, is shown in Figure 3. Results are shown for Linux 2.4 and 2.6 kernels, as well as "2.6(tuned)", that will be explained later in this section.

The slowdown of the host application on Linux 2.4 kernel was relatively high when running concurrently with a minimum priority guest application, averaging 25% for the benchmark suite. This validates similar observations in [14, 16]. As seen from Figure 3, the 2.6 kernel performs significantly better than the 2.4 kernel in this regard. The average slowdown is reduced from approximately 25% to 16%, but is still simply too

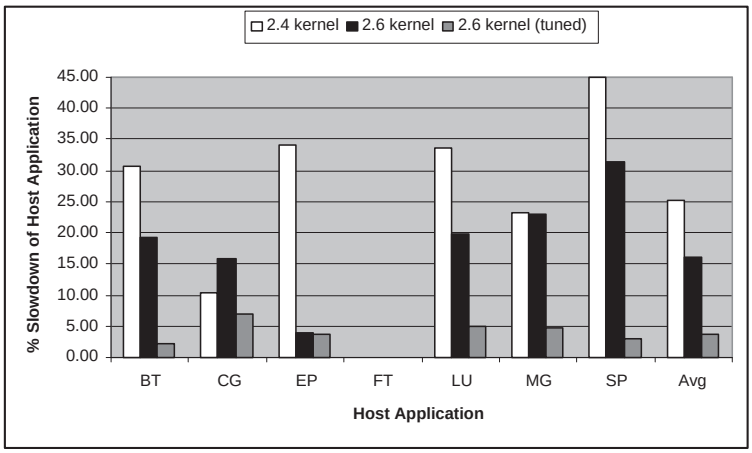

Fig. 3. Comparison of percent slowdown of the host application in shared mode when executing on different Linux kernels

large to be acceptable. This was surprising since the new O(1) scheduler in the 2.6 kernel was designed to respect the *nice* priorities more strictly.

Detailed investigation revealed the following. Unlike the 2.4 kernel, the 2.6 kernel has separate run queues for each of the two processors on a single node. In our scenario, one queue will have two processes, and the other queue will have one process, since there are 3 active processes (two host processes and one guest process). In some situations, both the host processes would get assigned to the same processor queue, with the one guest process assigned to the other processor's run queue. Clearly, this would lead to a nominal 50% slowdown of the host processes. The situation will eventually get corrected as the queues are periodically "load balanced". However, the default load balancing frequency is 200 milliseconds, implying that a phase of 50% slowdown could last for a significant amount of computing time. In order to mitigate this effect, we decreased the period between the invocation of the kernel load balancer to 10 milliseconds. Linux kernel 2.6 with this setting is referred to as "kernel 2.6 tuned" in Figure 3. We observe a dramatically reduced slowdown of the host application - down from an average of 16% to 3.6%. We believe that these are the lowest reported slowdowns for host applications when sharing the processors with a guest application on a widely deployed cluster operating system.

We would like to point out that "tuning" the Linux 2.6 kernel as discussed above technically contradicts our goal that an unmodified mainstream operating system should be employed. However, the tuning we have done is to mitigate the impact of an undesirable and unexpected side-effect of a new Linux feature. Hence we consider it to be a "performance bug fix" and expect that it will not be needed with continued development of Linux.

In the results discussed above, the host was assigned normal priority (nice = 0) while the guest was assigned the lowest priority available on the system (nice = 19).

The lowest priority for the guest is expected to yield the least slowdown for the host, and this was validated. However, it would appear logical that the host application should be assigned the highest priority (nice = -20), rather than normal priority (nice = 0), to minimize the slowdown. The measured slowdown with normal and highest priority for the host is shown in Figure 4.

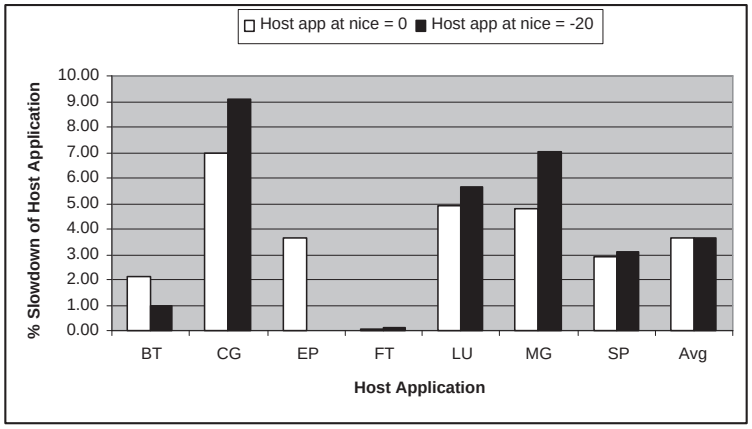

Fig. 4. Comparison of percent slowdown of the host application when running at normal priority (nice = 0) vs. when running at the highest priority (nice = -20) in shared mode

The slowdown was reduced dramatically with a higher priority when the host application was EP (the only application in the suite with no communication). Surprisingly the slowdown *increased* significantly for some of the communicating applications, in particular, CG (Conjugate Gradients) and MG (Multigrid). The average slowdown across the benchmark suite was virtually the same. The reasons for the higher slowdown for some applications are not understood and need to be investigated further. Related work has shown an increase in slowdown for some communicating applications when a larger time slice is given to all applications [18]. However, an increase in priority should result in a larger time slice only for the host application, so there is no apparent reason for its slowdown. Overall, there seems to be little benefit in raising the priority of the host applications.

Linux also supports a *realtime* priority level which appears attractive for host jobs for volunteer computing. However, this priority level blocks interrupts that are necessary for execution of parallel programs. Most applications in our benchmark were unable to complete execution with realtime priority.

3.3 Impact on cluster throughput

The goal in volunteer computing is for a guest application to make progress without any significant negative impact on the host application. Until now we have focused

on analyzing the impact on the host application. We now study the progress of guest applications. However, instead of directly reporting on the performance of guest applications, we report on the increase in system throughput, which is a measure of the overall benefit of fine-grained cycle stealing, as a consequence of a guest application executing in addition to the host application. Any increase in throughput is due to the work that is accomplished by the guest application, after any negative impact on the host application has been accounted for.

We define *normalized throughput* as the number of units of work completed per unit time on the cluster. The normalized throughput when a cluster is executing a single application is always considered to be 1. In shared mode both the host application and the guest application run simultaneously. Depending on the rate at which the host and guest applications proceed while sharing nodes, the normalized throughput can be greater than or less than 1. The normalized throughput of the cluster in shared mode is represented as follows:

$$\textit{Normalized throughput} = \frac{Th_D}{Th_S} + \frac{Tg_D}{Tg_S}$$

where

Th_D: Execution time of the host application in dedicated mode

Tg_D: Execution time of the guest application in dedicated mode

Th_S: Execution time of the host application in shared mode

Tg_S: Execution time of the guest application in shared mode

Figure 5 shows the percentage increase in the normalized throughput of the system when each host application is run in shared mode with EP as the guest application, as compared to dedicated execution of the host application.

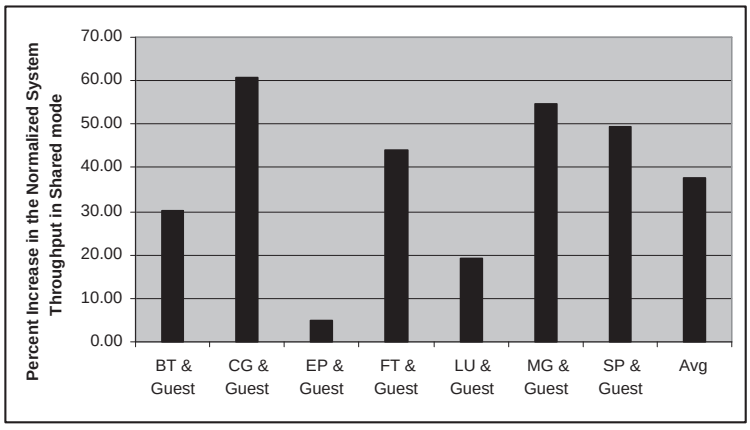

Fig. 5. Percent increase in the normalized system throughput with different benchmarks as host applications

We observe that there is a significant system throughput improvement that averages 38% for the benchmark suite. This comes at the cost of a relatively low 3.6% slowdown of the host applications. This demonstrates that a significant number of unused CPU cycles are available when the host application is executing in dedicated mode, and the guest application was able to utilize them successfully in shared mode.

The throughput improvement is the lowest for EP, LU (LU Matrix Factorization), and FT (Fast Fourier Transforms) benchmarks. We recall from Table 1 that these are the benchmarks that show the highest CPU utilization in dedicated execution, all of them over 90%. Hence fewer CPU cycles were available for the guest application in these cases. If these applications were removed from the suite, the average increase in system throughput would be 52%. This is relevant, since execution of the guest applications can be managed to avoid periods of high CPU usage or other system activity.

3.4 Parallel guest applications

Volunteer computing with communicating parallel guest applications is an important challenge [19] that can be met more effectively with clusters. In order to investigate this possibility, we performed a set of experiments with the CG benchmark, which is the most communication intensive application in the NAS benchmark suite, as the guest application. This is in contrast to EP which has negligible communication. “Tuned” Linux 2.6 kernel, as discussed earlier, was employed in these experiments. Figure 6 presents a comparison of the slowdown of the host application with CG and EP as guest applications.

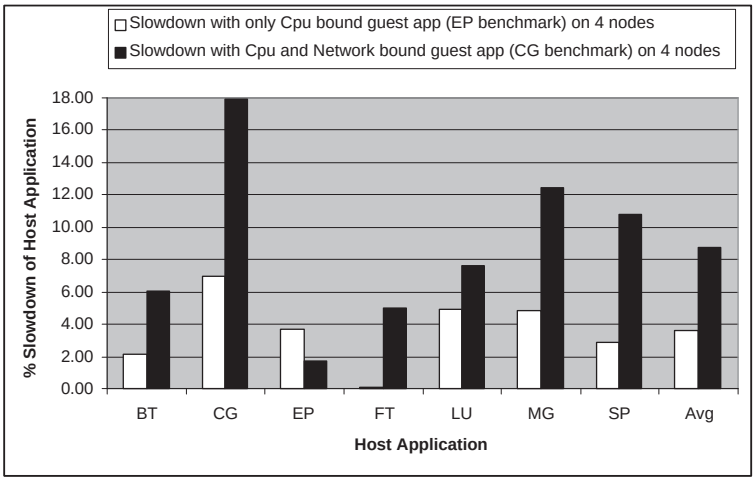

Fig. 6. Comparison of percent slowdown of the host application when running with the guest application EP vs. when running with the guest application CG

We observe that the slowdown for all host applications, with the exception of EP, is considerably higher when CG is the guest application. As seen in Figure 6, the average percentage slowdown of the host application when running with CG as guest, as compared to EP as guest, increases from 3.6% to 9%. One obvious reason is that EP being completely CPU bound just competes for CPU with the host application, while CG being communication intensive competes for CPU and network resources. However, a possibly more significant factor is the impact of the synchronization structure of the host and guest applications. CG being a communication intensive guest application is frequently blocked for communication, and hence cannot use the free CPU cycles when the host application itself is blocked for communication. As a result, the dynamic priority of the guest application rises and it is more likely to force an eviction of the host application later. We note that host application EP slows down less with CG as guest versus EP as guest. This is not surprising as they do not compete for communication resources. Further, unlike the case of EP as guest, CG as guest will sometimes not claim a proportional share of the CPU time since it can be blocked on communication.

The conclusion is that high communication parallel applications are not suitable for execution as guests on the current Linux operating system. However, parallel applications with moderate or low communication may be appropriate for volunteer computing.

3.5 Scalability

One of the factors that exacerbates the slowdown of a host application in shared execution is synchronization. When one node is slowed down due to sharing, it can have a cascading slowdown effect on the others. This effect is likely to be larger when the number of executing threads is higher. In order to investigate this, we compared the slowdown associated with execution on 4 nodes (8 threads) and 8 nodes (16 threads). The results are plotted in Figure 7.

The primary observation is that the the slowdown is slightly higher for a larger number of threads; the average slowdown was 3.6% for 8 threads and 4.5% for 16 threads. While this is encouraging, more experiments are needed to establish the impact of guest applications on large clusters.

4 Discussion

The following is a list of observations that are relevant for volunteer computing on clusters, based on the results in this paper and related research:

- Clusters show diverse usage patterns - many clusters are frequently idle.
- When a cluster is actively executing an application, a substantial fraction of the CPU and memory resources are often not used.
- The usage pattern of a cluster node can be similar for hours to days.
- If a host application uses most of the available CPU resources, there is little benefit from running a guest application simultaneously.
- Only guest applications that are sequential or have low communication requirements can generally execute with minimal effect on the host applications.

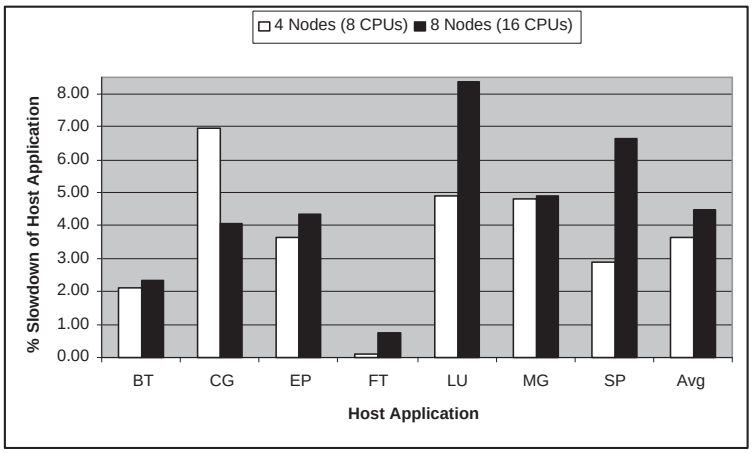

Fig. 7. Comparison of percent slowdown of the host application when running on 4 nodes vs. 8 nodes

– We have shown in related work [18] that memory usage has little relation to performance with sharing, but when the combined memory requirement of all executing threads approaches the total system memory, the performance can deteriorate sharply. While these results were collected for applications with equal priority, it is reasonable to conclude that both host and guest applications will not be able to execute effectively when their combined memory requirement exceeds available memory.

Based on these observations we present a set of guidelines for volunteer computing on clusters. Note that the above observations are based on a recent Linux release. Operating system support for zero priority processes that do not compete for resources can considerably ease the task of volunteer computing on clusters, and is discussed later in this section.

4.1 Guidelines for volunteer computing on clusters

In the scenarios in which we performed our experiments, the slowdown of the host applications with a sequential guest application averages around 3.6% and the improvement in throughput (indicating the progress of the guest application) averages around 38%. However, in individual cases, the slowdown can be higher and improvement in throughput significantly lower. With the following basic considerations, we can limit the use of volunteer computing to scenarios where the cost is minimized and the benefit is maximized:

1. Only consider sequential applications and parallel applications with a low communication bandwidth as guest applications. While this condition cannot be enforced by a system, such applications will get very poor service and hence the procedure

should be self correcting. Note that current volunteer computing frameworks are generally applicable only to "embarrassingly parallel" or "bag of tasks" applications.
2. Monitor the CPU, memory, and network usage on volunteered cluster nodes. Consider invoking a guest application only when a stable usage pattern emerges.
3. If the usage pattern shows CPU or network or memory usage above preset thresholds (say 85% for CPU) then do not invoke the guest application.
4. Before invocation, verify that the available memory exceeds the memory requirement of the guest application by a significant threshold.
5. If the resource usage pattern of the machine shows a significant change, suspend the guest application and restart after examining the criteria listed above.

Employing these guidelines will reduce the scenarios when volunteer computing can be applied on a cluster node, but also reduce the cost and increase the benefits when it is applied. Typically the resources available for volunteer computing exceed the demand, and the challenge is to exploit those resources effectively. Hence, eliminating potentially unattractive nodes is not a major concern.

4.2 Case for zero priority processes

Ideally a guest application should only use idle resources and have no impact on the host applications. Our experiments demonstrate that the latest version of the Linux operating system allows the guest applications to execute with a small impact on executing the host applications, but it is not negligible. Volunteer computing on clusters will be considerably simplified with support for *zero priority* processes that would not consume any resources that other processes can potentially use. Such a zero priority process will never be scheduled so long as a higher priority process is able to execute, and would immediately relinquish the CPU when a higher priority process is ready to execute. Developing such schedulers is technically feasible and prototypes have been demonstrated in other research [14, 16]. However, widely deployed operating systems have a concept of fairness that implies that even the lowest priority process must get a certain share of resources and should not starve. Support for zero priority processes, that never compete for resources, can be done without compromising other design goals of an operating system, although a detailed discussion is beyond the scope of this paper. Fairness and starvation are not issues if a process is explicitly designated as zero priority, except to ensure freedom from deadlocks and any other unintended consequences.

There will always be some performance impact due to guest jobs - some factors, e.g., the overhead of warming the cache after a context switch, cannot be eliminated. However, with a well designed implementation of zero priority processes and a good model for volunteer computing, we believe that the slowdown of host jobs can be made negligible, possibly well below 1%, which increases the appeal of volunteer computing dramatically.

5 Conclusions

Computation clusters present a vast and attractive resource of unused compute cycles that can be used for volunteer computing. Based on a study of a production cluster, we

show that long periods of significant CPU and memory underutilization are common. However, utilizing these free cycles for guest applications, while other applications are executing, is a challenge. Based on our experiments on the most recent version of Linux, we show that these cycles can be exploited with only a small slowdown of the host applications. The contribution of this paper is to present evidence that clusters are attractive for volunteer computing and can be used efficiently for that purpose. The paper also offers guidelines on how slowdown of the host applications can be minimized and cluster throughput maximized for volunteer computing. Additional discussion emphasizes that simple support for zero priority processes will make the case of clusters for volunteer computing more compelling.

6 Acknowledgments

The staff at the High Performance Computing Center at University of Houston provided us with access to the computation clusters that made this project possible. We would like to thank all members of our research group, in particular, Tsung-I Huang and Qiang Xu, for their contributions to this work. We would also like to thank the Linux kernel developers and Rocks developers for their support. In particular, we want to thank Con Kolivas for providing and improving the smp nice patch. Finally, the anonymous referees made several suggestions that helped improve this paper.

This material is based upon work supported by the National Science Foundation under Grant No. ACI-0234328 and Grant No. CNS-0410797. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Support was also provided by University of Houston's Texas Learning and Computation Center.

References

1. Anderson, D., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: An experiment in public-resource computing. Communications of the ACM **45**(11) (2002)
2. BOINC. (http://boinc.berkeley.edu/)
3. Anderson, D.: BOINC: A system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing. (2004) 4–10
4. Litzkow, M., Livny, M., Mutka, M.: Condor - a hunter of idle workstations. In: 8th International Conference on Distributed Computing Systems. (1988) 104–111
5. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: architecture and performance of an enterprise desktop grid system. Journal of Parallel and Distributed Computing **63**(5) (2003) 597–610
6. OpenMosix. (http://openmosix.sourceforge.net/)
7. Grid MP. (http://ud.com/solutions/deploy/mp_enterprise.htm)
8. Anderson, D., Fedak, G.: The computation and storage potential of volunteer computing. In: Sixth IEEE International Symposium on Cluster Computing and the Grid. (2006)
9. Kondo, D., Taufer, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating desktop grids: An empirical study. In: International Parallel and Distributed Processing Symposium (IPDPS'04). (2004)

10. Li, H., Groep, D., Wolters, L.: Workload characteristics of a multi-cluster supercomputer. In: 10th International Workshop on Job Scheduling Strategies for Parallel Processing. Springer Verlag (2004) 176–193
11. Zhang, Y., Franke, H., Moreira, J., Sivasubramaniam., A.: Improving parallel job scheduling by combining gang scheduling and backfilling techniques. In: 14th International Parallel and Distributed Processing Symposium. (2000)
12. Feitelson, D.G., Weil, A.M.: Utilization and predictability in scheduling the IBM SP2 with backfilling. In: 12th International Parallel Processing Symposium. (1998) 542–546
13. Kleban, S.D., Clearwater, S.H.: Interstitial computing: Utilizing spare cycles on supercomputers. In: IEEE International Conference on Cluster Computing. (2003)
14. Ryu, K., Hollingsworth, J.: Linger longer: fine-grain cycle stealing for networks of workstations. In: ACM/IEEE Conference on Supercomputing. (1998) 1–12
15. Ryu, K., Hollingsworth, J.: Resource policing to support fine-grain cycle stealing in networks of workstations. IEEE Transactions on Parallel and Distributed Systems **15**(10) (2004) 878–892
16. Stiehr, G.: Using fine-grained cycle stealing to improve throughput, efficiency and response time on a dedicated cluster while maintaining quality of service. Master's thesis, Washington University (2004)
17. Weinberg, J., Snavely, A.: Symbiotic space-sharing on SDSC's DataStar system. In: 12th Workshop on Job Scheduling Strategies for Parallel Processing. (2006)
18. Ghanesh, M., Kumar, S., Subhlok, J.: Empirical evaluation of shared parallel execution on independently scheduled clusters. In: 1st International Workshop on Grid Performability. (2005)
19. Acharya, A., Edjlali, G., Saltz, J.: The Utility of Exploiting Idle Workstations for Parallel Computation. In: ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. (1997)