The Case of the Missing Supercomputer Performance:
Achieving Optimal Performance on the 8,192 Processors of ASCI Q

Fabrizio Petrini Darren J. Kerbyson Scott Pakin

Performance and Architecture Laboratory (PAL)
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory
Los Alamos, New Mexico, USA

{fabrizio,djk,pakin}@lanl.gov

Abstract

In this paper we describe how we improved the effective performance of ASCI Q, the world's second-fastest supercomputer, to meet our expectations. Using an arsenal of performance-analysis techniques including analytical models, custom microbenchmarks, full applications, and simulators, we succeeded in observing a serious—but previously undetected—performance problem. We identified the source of the problem, eliminated the problem, and "closed the loop" by demonstrating up to a factor of 2 improvement in application performance. We present our methodology and provide insight into performance analysis that is immediately applicable to other large-scale supercomputers.

1 Introduction

"[W]hen you have eliminated the impossible, whatever remains, however improbable, must be the truth."
— Sherlock Holmes, *Sign of Four*,
Sir Arthur Conan Doyle

Users of the 8,192-processor ASCI Q machine that was recently installed at Los Alamos National Laboratory (LANL) are delighted to be able to run their applications on a 20 Tflop/s supercomputer and obtain large performance gains over previous supercomputers. We, however, asked the question, "Are these applications running as fast as they *should* be running on ASCI Q?" This paper chronicles the approach we took to accurately determine the performance that should be observed when running SAGE [9], a compressible Eulerian hydrodynamics code consisting of ~150,000 lines of Fortran + MPI code; how we proposed and tested numerous hypotheses as to what was causing a discrepancy between prediction and measurement; and how we finally identified and eliminated the problem.

As of April 2003, ASCI Q exists in its final form—a single system comprised of 2,048 HP ES45 AlphaServer SMP nodes, each containing four EV68 Alpha processors and interconnected with a Quadrics QsNet network [16]. ASCI Q was installed in stages and its performance was measured at each step. The performance of individual characteristics such as memory, interprocessor communication, and full-scale application performance were all measured and recorded. Performance testing began with the measurement on the first available hardware worldwide: an eight-node HP ES45 system interconnected using two rails of Quadrics in March 2001 at HP in Marlborough, Massachusetts. The first 128 nodes were available for use at LANL in September 2001. The system increased in size to 512 nodes in early 2002 and to two segments of 1,024 nodes by November 2002. The peak processing performance of the combined 2,048-node system

© 2003 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SC'03, November 15–21, 2003, Phoenix, Arizona, USA
Copyright 2003 ACM 1-58113-695-1/03/0011 ... $5.00

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

is 20 Tflop/s and is currently listed as #2 in the list of the top 500 fastest computers.[1]

The ultimate goal when running an application on a supercomputer such as ASCI Q is either to maximize work performed per unit time (weak scaling) or to minimize time-to-solution (strong scaling). The primary challenge in achieving this goal is complexity. Large-scale scientific applications, such as those run at LANL, consist of hundreds of thousands of lines of code and possess highly nonlinear scaling properties. Modern high-performance systems are difficult to optimize for, as their deep memory hierarchies can incur significant performance loss in the absence of temporal or spatial access locality; multiple processors share a memory bus, potentially leading to contention for a fixed amount of bandwidth; network performance may degrade with physical or logical distances between communicating peers or with the level of contention for shared wires; and, each node runs a complete, heavyweight operating system tuned primarily for workstation or server workloads, not high-performance computing workloads. As a result of complexity in applications and in supercomputers it is difficult to determine the source of suboptimal application performance—or even to determine if performance *is* suboptimal.

Ensuring that key, large-scale applications run at maximal efficiency requires a methodology that is highly disciplined and scientific, yet is still sufficiently flexible to adapt to unexpected observations. The approach we took is as follows:

1. Using detailed knowledge of both the application and the computer system, use performance modeling to determine the performance that SAGE ought to see when running on ASCI Q.

2. If SAGE's measured performance is less than the expected performance, determine the source of the discrepancy.

3. Eliminate the cause of the suboptimal performance.

4. Repeat from step 2 until the measured performance matches the expected performance.

Step 2 is the most difficult part of the procedure and is therefore the focus of this paper.

While following the above procedure the performance analyst has a number of tools and techniques at his disposal as listed in Table 1. An important constraint is that time on ASCI Q is a scarce resource. As a result, any one researcher or research team has limited opportunity to take measurements on the actual supercomputer. Furthermore, configuration changes are not always practical. It often takes a significant length of time to install or reconfigure software on thousands of nodes and cluster administrators are reluctant to make modifications that may adversely affect other users. In addition, a complete reboot of the entire system can take several hours [11] and is therefore performed only when absolutely necessary.

The remainder of the paper is structured as follows. Section 2 describes how we determined that ASCI Q was not performing as well as it could. Section 3 details how we systematically applied the tools and techniques shown in Table 1 to identify the source of the performance loss. Section 4 explains how we used the knowledge gained in Section 3 to achieve our goal of improving application performance to the point where it is within a small factor of the best that could be expected. Section 5 completes the analysis by re-measuring the performance of SAGE on an optimized ASCI Q and demonstrating how close the new performance is to the ideal for that application and system. A discussion of the insight gained in the course of this exercise is presented in Section 6. We contrast our work to others' in Section 7. Finally, we present our conclusions in Section 8.

2 Performance expectations

Based on the Top 500 data, ASCI Q appears to be performing well. It runs the LINPACK [3] benchmark at 68% of peak performance, which is well within range for machines of its class. However, there are more accurate methods for determining how well a system is actually performing. From the testing of the first ASCI Q hardware in March 2001, performance models of several applications representative of the ASCI workload were used to provide an expectation of the performance that should be achievable on the full-scale system [7, 9]. These performance models are parametric in terms of certain basic, system-related features such as the sequential processing time—as measured on a single processor—and the communication network performance.

In particular, a performance model of SAGE was developed for the express purpose of predicting SAGE's

[1] http://www.top500.org

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

TABLE 1: Performance analysis tools and techniques

Technique	Description	Purpose
measurement	running full applications under various system configurations and measuring their performance	determine how well the application actually performs
microbenchmarking	measuring the performance of primitive components of an application	provide insight into application performance
simulation	running an application or benchmark on a software simulation instead of a physical system	examine a series of "what if" scenarios, such as cluster configuration changes
analytical modeling	devising a parameterized, mathematical model that represents the performance of an application in terms of the performance of processors, nodes, and networks	rapidly predict the expected performance of an application on existing or hypothetical machines

performance on the full-sized ASCI Q. The model has been validated on many large-scale systems—including all ASCI systems—with a typical prediction error of less than 10% [10]. The HP ES45 AlphaServer nodes used in ASCI Q actually went through two major upgrades during installation: the PCI bus within the nodes was upgraded from 33 MHz to 66 MHz and the processor speed was upgraded from 1 GHz to 1.25 GHz. The SAGE model was used to provide an expected performance of the ASCI Q nodes in all of these configurations.

The performance of the first 4,096-processor segment of ASCI Q ("QA") was measured in September 2002 and the performance of the second 4,096-processor segment ("QB")—at the time, not physically connected to QA—was measured in November 2002. The results of these two sets of measurements are consistent with each other although they rapidly diverge from the performance predicted by the SAGE performance model, as shown in Figure 1 for weak-scaling (i.e., fixed per-node problem size) operation. At 4,096 processors, the time to process one cycle of SAGE was *twice* that predicted by the model. This was considered to be a "difference of opinion" between the model and the measurements. Without further analysis it would have been impossible to discern whether the performance model was inaccurate—although it has been validated on many other systems—or whether there was a problem with some aspect of ASCI Q's hardware or software configuration.

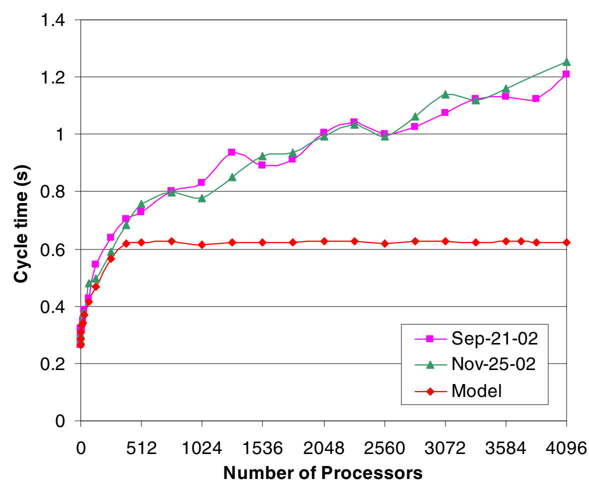


Figure 1: Expected and measured SAGE performance

MYSTERY #1

SAGE performs significantly worse on ASCI Q than was predicted by our performance model.

In order to identify why there was a difference between the measured and expected performance we performed a battery of tests on ASCI Q. A revealing result came from varying the number of processors per node used to run SAGE. Figure 2 shows the difference between the modeled and the measured performance when using 1, 2, 3, or all 4 processors per node. Note that a log scale is used on the x axis. It can be seen

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

that the only significant difference occurs when using all four processors per node thus giving confidence to the model being accurate.

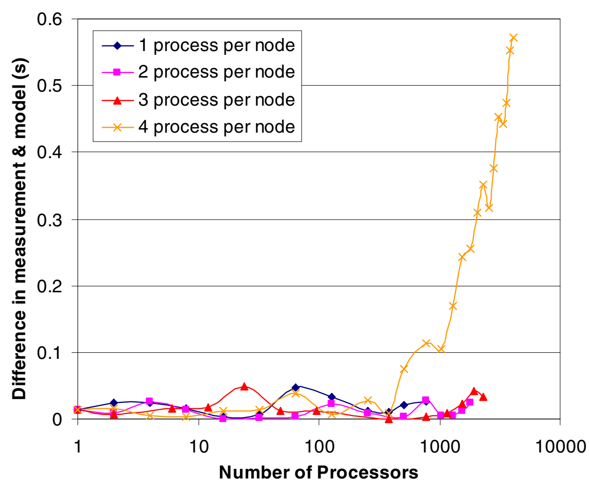


Figure 2: Difference between modeled and measured SAGE performance when using 1, 2, 3, or 4 processors per node

It is also interesting to note that, when using more than 256 nodes, the processing rate of SAGE was actually better when using three processors per node instead of the full four, as shown in Figure 3. Even though 25% fewer processors are used per node, the performance can actually be greater than when using all four processors per node. Furthermore, another crossover occurs at 512 nodes, after which two processors per node also outperform four processors per node.

Like Phillips et al. [17], we also analyzed application performance variability. Each computation cycle within SAGE was configured to perform a constant amount of work and could therefore be expected to take a constant amount of time to complete. We measured the cycle time of 1,000 cycles using 3,584 processors of one of the ASCI Q segments. The ensuing cycle times are shown in Figure 4(a) and a histogram of the variability is shown in Figure 4(b). It is interesting to note that the cycle time ranges from just over 0.7s to over 3.0s, indicating greater than a factor of 4 in variability.

A profile of the cycle time when using all four processors per node, as shown in Figure 5, reveals a number of important characteristics in the execution of SAGE. The profile was obtained by separating out the time taken in each of the local boundary exchanges

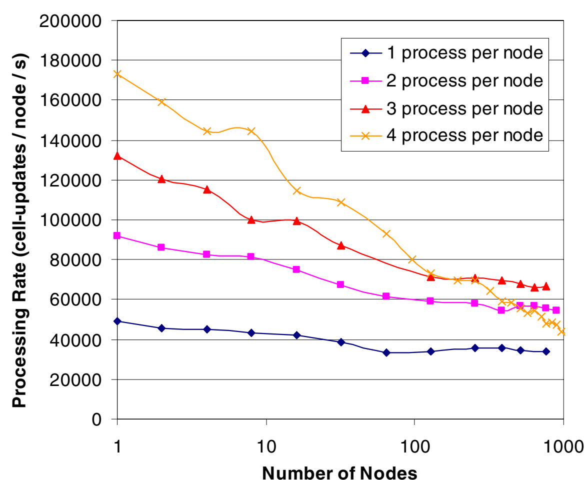


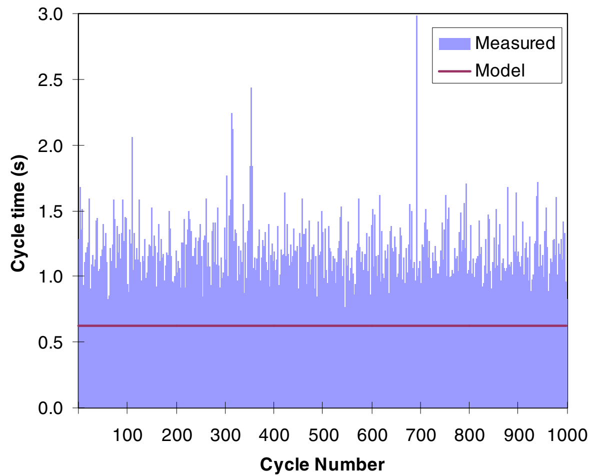
Figure 3: Effective SAGE processing rate when using 1, 2, 3, or 4 processors per node

(get and put) and the collective-communication operations (allreduce, reduction, and broadcast) on the root processor. The overall cycle time, which includes computation time, is also shown in Figure 5. The time taken in the local boundary exchanges appears to plateau above 500 processors and corresponds exactly to the time predicted by the SAGE performance model. However, the time spent in allreduce and reduction increases with the number of processors and appears to account for the increase in overall cycle time with increasing processor count. It should be noted that the number and payload size in the allreduce operations was constant for all processor counts, and the relative difference between allreduce and reduction (and also broadcast) is due to the difference in their frequency of occurrence within a single cycle.

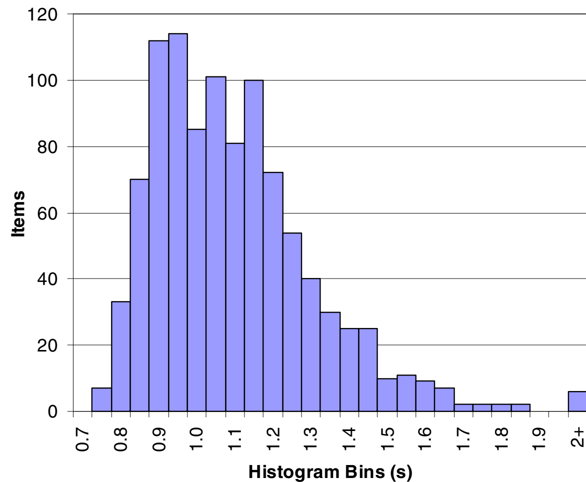
To summarize, our analysis of SAGE on ASCI Q led us to the following observations:

- There is a significant difference of opinion between the expected performance and that actually observed.
- The performance difference occurs only when using all four processors per node.
- There is a high variability in the performance from cycle to cycle.
- The performance deficit appears to originate from the collective operations, especially allreduce.

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM



(a) Variability



(b) Histogram

Figure 4: SAGE cycle-time measurements on 3,584 processors

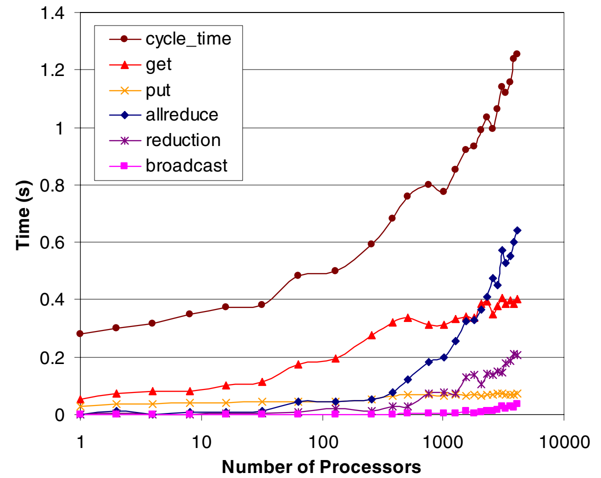


Figure 5: Profile of SAGE's cycle time

It is therefore natural to deduce that improving the performance of allreduce, especially when using four processors per node, ought to lead to an improvement in application performance. In Section 3 we test this hypothesis.

3 Identification of performance factors

In order to identify why application performance such as that observed on SAGE was not as good as expected, we undertook a number of performance studies. To simplify this process we concerned ourselves with the examination of smaller, individual operations that could be more systematically analyzed. Since it appeared that SAGE was most significantly affected by the performance of the allreduce collective operation several attempts were made to improve the performance of collectives on the Quadrics network.

3.1 Optimizing the allreduce

Figure 6 shows the performance of the allreduce when executed on an increasing number of nodes. We can clearly see that a problem arises when using all four processors within a node. With up to three processors the allreduce is fully scalable and takes, on average, less than 300 μs. With four processors the latency surges to more than 3 ms. These measurements were obtained on the QB segment of ASCI Q.

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

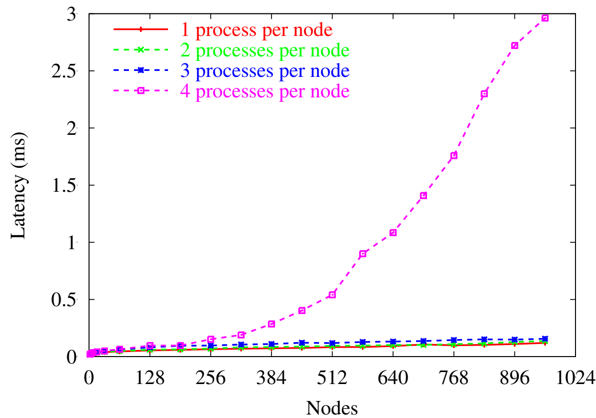


Figure 6: allreduce latency as a function of the number of nodes and processes per node

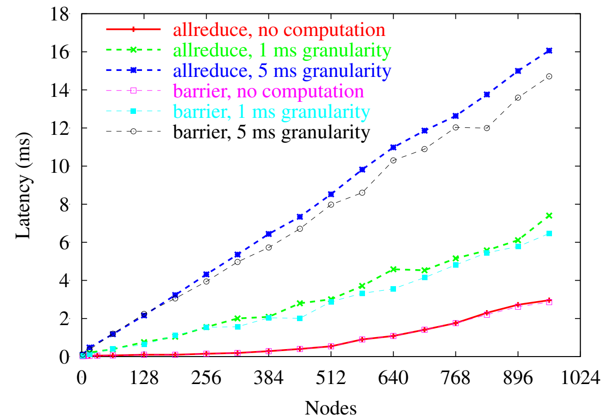


Figure 7: allreduce and barrier latency with varying amounts of intervening computation

Because using all four processors per node results in unexpectedly poor performance we utilize four processors per node in the rest of our investigation. Figure 7 provides more clues to the source of the performance problem. It shows the performance of the allreduce and barrier in a synthetic parallel benchmark that alternately computes for either 0, 1, or 5 ms then performs either an allreduce or a barrier. In an ideal, scalable, system we should see a logarithmic growth with the number of nodes and insensitivity to the computational granularity. Instead, what we see is that the completion time increases with both the number of nodes and the computational granularity. Figure 7 also shows that both allreduce and barrier exhibit similar performance. Given that the barrier is implemented using a simple hardware broadcast whose execution is almost instantaneous (only a few microseconds) and that it reproduces the same problem, we concentrate on a barrier benchmark later in this analysis.

We made several attempts to optimize the allreduce in the four-processor case and were able to substantially improve the performance. To do so, we used a different synchronization mechanism. In the existing implementation the processes in the reduce tree poll while waiting for incoming messages. By changing the synchronization mechanism from always polling to polling for a limited time (100 μs, determined empirically) and then blocking, we were able to improve the latency by a factor of 7.

At 4,096 processors, SAGE spends over 51% of its time in allreduce. Therefore, a sevenfold speedup in allreduce ought to lead to a 78% performance gain in SAGE. In fact, although extensive testing was performed on the modified collectives, this resulted in only a marginal improvement in application performance.

Mystery #2

> Although SAGE spends half of its time in allreduce (at 4,096 processors), making allreduce seven times faster leads to a negligible performance improvement.

We can therefore conclude that neither the MPI implementation nor the network are responsible for the performance problems. By process of elimination, we can infer that the source of the performance loss is in the nodes themselves. Technically, it is possible that the performance loss could be caused by the interaction of multiple factors. However, to keep our approach focused we must first investigate each potential source of performance loss individually.

3.2 Analyzing computational noise

Our intuition was that periodic system activities were interfering with application execution. This hypothesis follows from the observation that using all four processors per node results in lower performance than when using fewer processors. Figures 3 and 6 confirm this observation for both SAGE and allreduce performance. System activities can run without interfering with the application as long as there is a spare processor available in each node to absorb them. When there is no spare processor, a processor is temporarily taken from the application to handle the system activity. Doing so

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

may introduce performance variability, which we refer to as "noise". Noise can explain why converting from strictly polling-based synchronization to synchronization that uses a combination of polling and blocking substantially improves performance in the allreduce, as observed in Section 3.1.

To determine if system noise is, in fact, the source of SAGE's performance variability, as well, we crafted a simple microbenchmark designed to expose the problems. The microbenchmark works as shown in Figure 8: each node performs a synthetic computation carefully calibrated to run for exactly 1,000 seconds in the absence of noise.

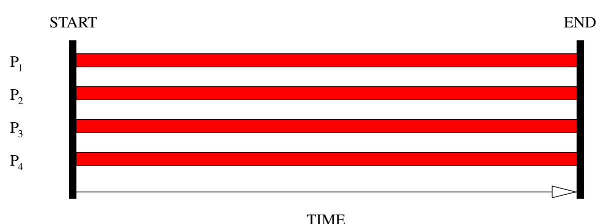


Figure 8: Performance-variability microbenchmark

The total normalized run time for the microbenchmark is shown in Figure 9 for all 4,096 processors in QB. Because of interference from noise the actual processing time can be longer and can vary from process to process. However, the measurements indicate that the slowdown experienced by each process is low, with a maximum value of 2.5%. As Section 2 showed a performance slowdown in SAGE of a factor of 2, a mere 2.5% slowdown in the performance-variability microbenchmark appears to contradict our hypothesis that noise is what is causing the high performance variability in SAGE.

MYSTERY #3

Although the "noise" hypothesis could explain SAGE's suboptimal performance, microbenchmarks of per-processor noise indicate that at most 2.5% of performance is being lost to noise.

Sticking to our assumption that noise is somehow responsible for SAGE's performance problems we refined our microbenchmark into the version shown in Figure 10. The new microbenchmark was intended to provide a finer level of detail into the measurements presented in Figure 9. In the new microbenchmark, each node performs 1 million iterations of a synthetic computation, with each iteration carefully calibrated

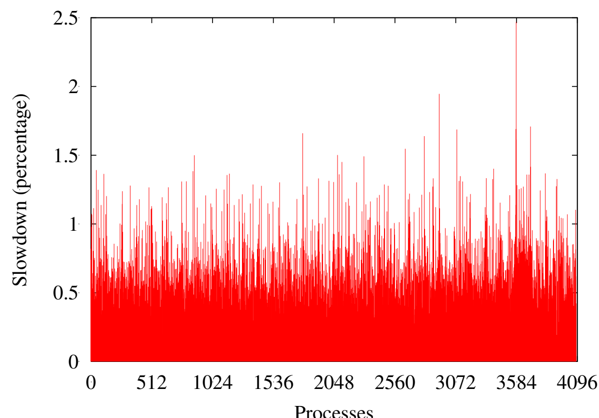


Figure 9: Results of the performance-variability microbenchmark

to run for exactly 1 ms in the absence of noise, for an ideal total run time of 1,000 seconds. Using a small granularity, such as 1 ms, is important because many LANL codes exhibit such granularity between communication phases. During the purely computational phase there is no message exchange, I/O, or memory access. As a result, the run time of each iteration should always be 1 ms in a noiseless machine.

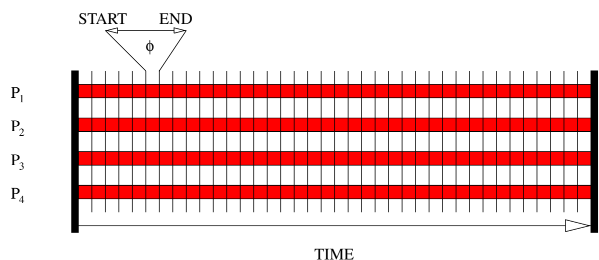


Figure 10: Performance-variability of the new microbenchmark

We ran the microbenchmark on all 4,096 processors of QB. However, the variability results were qualitatively identical to those shown in Figure 9. Our next step was to aggregate the four processor measurements taken on each node, the idea being that system activity can be scheduled arbitrarily on any of the processors in a node. Our hypothesis is that examining noise on a per-node basis may expose structure in what appears to be uncorrelated noise on a per-processor basis. Again, we ran 1 million iterations of the microbenchmark, each with a granularity of 1 ms. At the end of each iteration we measured the actual run time and for each iteration that took more than the

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

expected 1 ms run time, we summed the unexpected overhead. The idea to aggregate across processors within a node led to an important observation: Figure 11 clearly indicates that there is a regular pattern to the noise across QB's 1,024 nodes. Every cluster of 32 nodes contains some nodes that are consistently noisier than others.

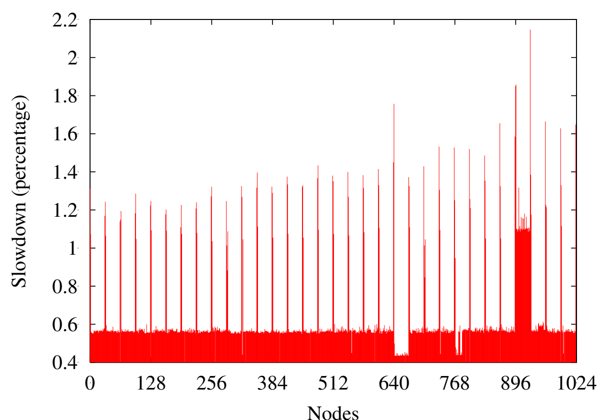


Figure 11: Results of the performance-variability microbenchmark analyzed on a per-node basis

> FINDING #1
>
> Analyzing noise on a per-node basis instead of a per-processor basis reveals a regular structure across nodes.

Figure 12 zooms in on the data presented in Figure 11 in order to show more detail on one of the 32-node clusters. We can see that all nodes suffer from a moderate background noise and that node 0 (the cluster manager), node 1 (the quorum node), and node 31 (the RMS cluster monitor) are slower than the others. This pattern repeats for each cluster of 32 nodes.

In order to understand the nature of this noise we plot the actual time taken to perform the 1 million 1 ms computations in histogram format. Figure 13 shows one such histogram for each of the four groupings of nodes: nodes 0, 1, 2–30, and 31 of a 32-node cluster. Note that the scale of the x axis varies from graph to graph. These graphs show that the noise in each grouping has a well-defined pattern with classes of events that happen regularly with well-defined frequencies and durations. For example, on any node of a cluster we can identify two events that happen regularly every 30 seconds and whose durations are

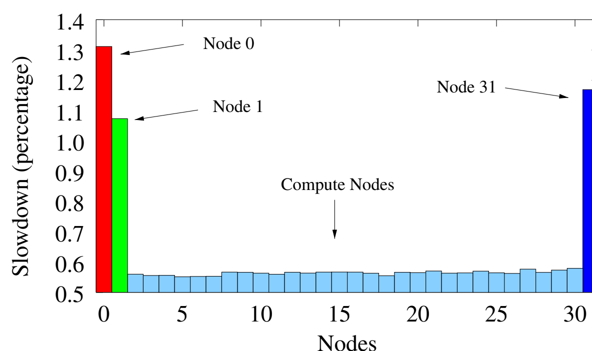


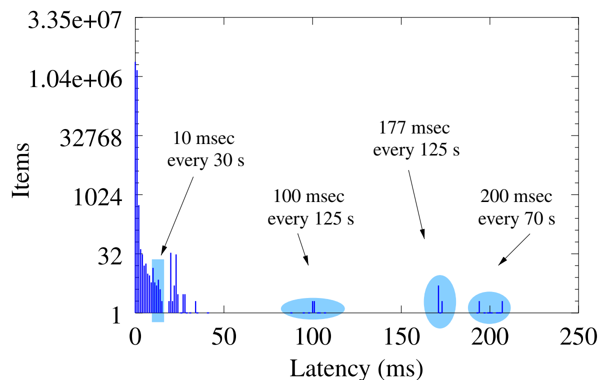
Figure 12: Slowdown per node within each 32-node cluster

15 and 18 ms. This means that a slice of computation that should take 1 ms occasionally takes 16 ms or 19 ms. The process that experiences this type of interruption will freeze for the corresponding amount of time. Intuitively, these events can be traced back to some regular system activity as dæmons or the kernel itself. Node 0 displays four different types of activities, all occurring at regular intervals, with a duration that can be up to 200 ms. Node 1 experiences a few heavyweight interrupts—one every 60 seconds—that freeze the process for about 335 ms. On node 31 we can identify another pattern of intrusion, with frequent interrupts (every second) and a duration of 7 ms.

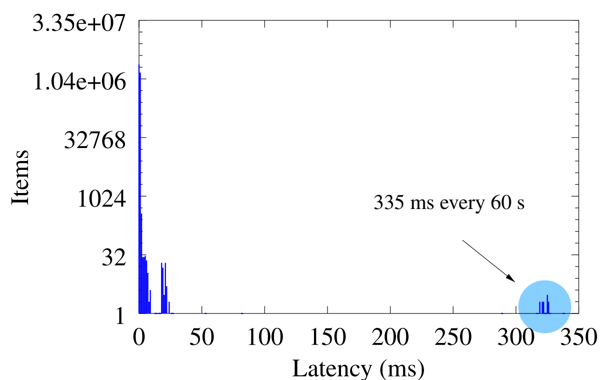
Using a number of techniques on QB we were able to identify the source of most activities. As a general rule, these activities happen at regular intervals. The two events that take 15 and 18 ms on each node are generated by Quadrics's resource management system, RMS [18], which regularly spawns a dæmon every thirty seconds. A distributed heartbeat that performs cluster management, generated at kernel level, is the cause of many lightweight interrupts (one every 125 ms) whose duration is a few hundred microseconds. Other dæmons that implement the parallel file system and TruCluster, HP's cluster management software, are the source of the noise on nodes 0 and 1. Table 2 summarizes the duration and location within each 32-node cluster of the various types of noise.

Each of these events can be characterized by a tuple $\langle F, L, E, P \rangle$ that describes the frequency of the event F, the average duration L, the distribution E, and the placement (the set of nodes where the event is generated) P. As will be discussed in Section 3.4, this characterization is accurate enough to closely model the noise in the system and is also able to provide

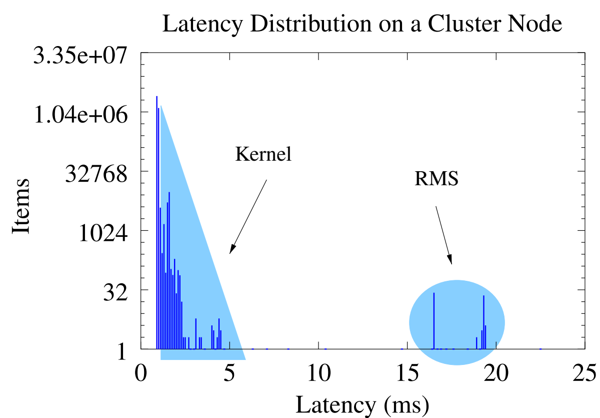
Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM



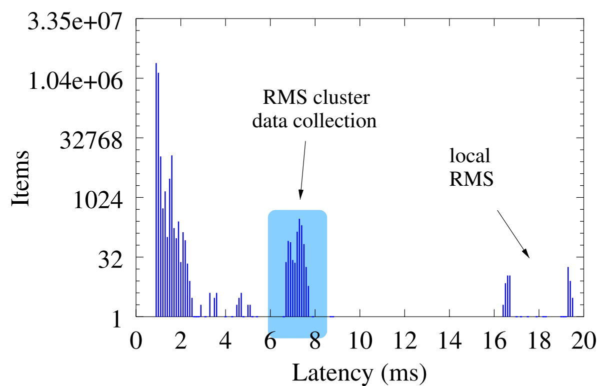
(a) Latency distribution on node 0



(b) Latency distribution on node 1



(c) Latency distribution on nodes 2–30



(d) Latency distribution on node 31

Figure 13: Identification of the events that cause the different types of noise

TABLE 2: Summary of noise on each 32-node cluster

Source of noise	Duration (ms)	Location (nodes)			
		0	1	2–30	31
Kernel	0–3	✔	✔	✔	✔
RMS dæmons	5–18	✔	✔	✔	✔
TruCluster dæmons	>18	✔	✔		

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

clear guidelines to identify and eliminate the sources of noise.

3.3 Effect on system performance

Figure 14(a) provides intuition on the potential effects of these delays on applications that are fine-grained and bulk-synchronous. In such a case, a delay in a single process slows down the whole application. Note that even though any given process in Figure 14(a) is delayed only once, the collective-communication operation (represented by the vertical lines) is delayed in every iteration. When we run an application on a large number of processors, the likelihood of having at least one slow process per iteration increases. Consider, for example, an application that barrier-synchronizes every 1 ms. If, on each iteration, only one process out of 4,096 experiences a 100 ms delay, then the whole application will run 100 times slower!

While the obvious solution is to remove any type of noise in the system, in certain cases it may not be possible or cost effective to remove dæmons or kernel threads that perform essential activities as resource management, monitoring, parallel file system, etc. Figure 14(b) suggests a possible solution that doesn't require the elimination of the system activities. By coscheduling these activities we pay the noise penalty only once, irrespective of the machine size. An indirect form of dæmon coscheduling based on global clock synchronization was implemented by Mraz on the IBM SP1 [14]. We recently developed a prototype of an explicit coscheduler as a Linux kernel module [4, 5, 15] and we are in the process of investigating the performance implications.

3.4 Modeling system events

We developed a discrete-event simulator that takes into account all the classes of events identified and characterized in Section 3.2. This simulator provides a realistic lower bound on the execution time of a barrier operation. We validated the simulator for the measured events, and we can see from Figure 15 that the model is close to the experimental data. The gap between the model and the data at high node counts can be explained by the presence of a few especially noisy (probably misconfigured) clusters.

Using the simulator we can predict the performance gain that can be obtained by selectively removing the sources of the noise. For example, Figure 15 shows that with a computational granularity of 1 ms, if we remove the noise generated by either node 0, 1 or 31, we only get a marginal improvement, approximately 15%. If we remove all three "special" nodes—0, 1 and 31—we get an improvement of 35%. However, the surprise is that the noise in the system dramatically reduces when we eliminate the kernel noise on all nodes.

FINDING #2

On fine-grained applications, more performance is lost to short but frequent noise on all nodes than to long but less frequent noise on just a few nodes.

4 Eliminating the sources of noise

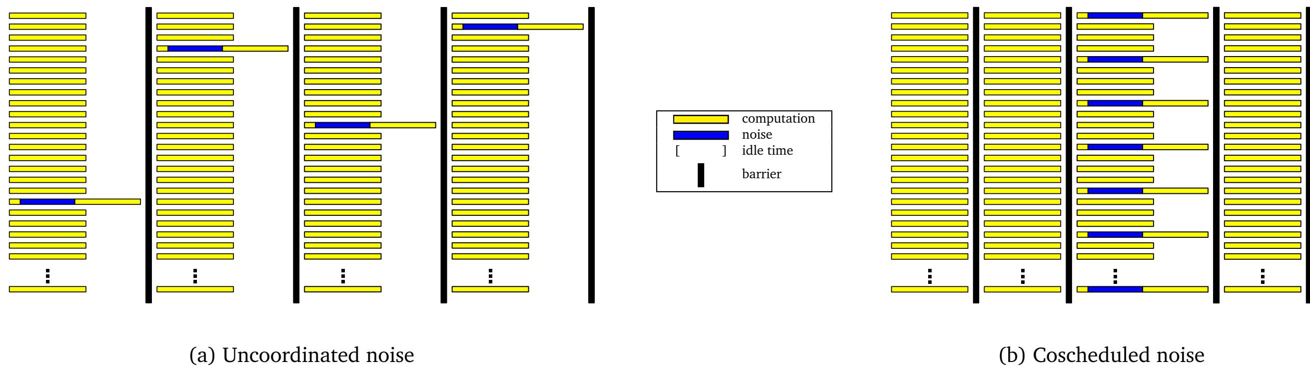
It is not generally feasible to remove *all* the noise in a system. For example, TruCluster performs two types of heartbeats at kernel level: one runs for 640 µs every 125 ms and the other runs for 350 µs every 600 ms. Removing either of these would require substantial kernel modifications. Using our methodology of noise analysis we were able to determine that, when running medium-grained applications, the first type of heartbeat accounts for 75% of performance lost to kernel activity while the second accounts for only 4% of lost performance. Time is therefore better spent eliminating the first source of noise than the second when running medium-grained applications.

Based on the results of our noise analysis, in January 2003 we undertook the following optimizations on ASCI Q:

- We removed about ten dæmons (including `envmod`, `insightd`, `snmpd`, `lpd`, and `niff`) from all nodes.
- We decreased the frequency of RMS monitoring by a factor of 2 on each node (from an interval of 30 seconds to an interval of 60 seconds).
- We moved several TruCluster dæmons from nodes 1 and 2 to node 0 on each cluster, in order to confine the heavyweight noise to this node.

It was not possible for us to the modify the kernel to eliminate the two noise-inducing heartbeats described above. However, our noise analysis indicated that the optimizations we did perform could be expected to improve SAGE's performance by a factor of 2.2.

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM



(a) Uncoordinated noise

(b) Coscheduled noise

Figure 14: Illustration of the impact of noise on synchronized computation

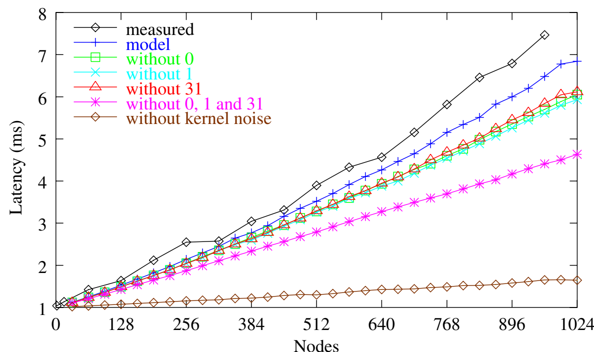


Figure 15: Simulated vs. experimental data with progressive exclusion of various sources of noise in the system

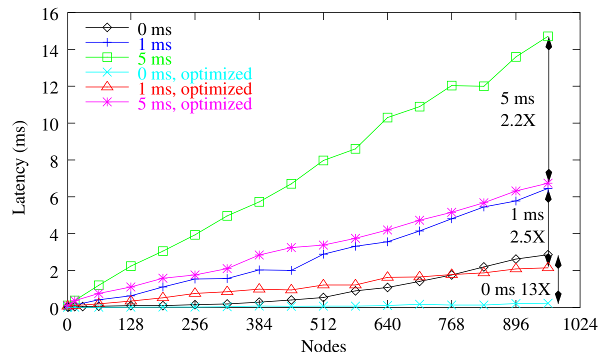


Figure 16: Performance improvements obtained on the barrier-synchronization microbenchmark for different computational granularities

As an initial test of the efficacy of our optimizations we used a simple benchmark in which all nodes repeatedly compute for a fixed amount of time and then synchronize using a global barrier, whose latency is measured. Figure 16 shows the results for three types of computational granularity—0 ms (a simple sequence of barriers without any intervening computation), 1 ms, and 5 ms—and both with the noise-reducing optimizations, as described above, and without, as previously depicted in Figure 7.

We can see that with fine granularity (0 ms) the barrier is 13 times faster. The more realistic tests with 1 and 5 ms, which are closer to the actual granularity of LANL codes, show that the performance is more than doubled. This confirms our conjecture that performance variability is closely related to the noise in the nodes.

Figure 16 shows only that we were able to improve the performance of a microbenchmark. In Section 5 we discuss whether the same performance optimizations can improve the performance of applications, specifically SAGE.

5 SAGE: Optimized performance

Following from the removal of much of the noise induced by the operating system the performance of SAGE was again analyzed. This was done in two situations, one at the end of January 2003 on a 1,024-node segment of ASCI Q, followed by the performance on the full sized ASCI Q at the start of May 2003 (after the two individual 1,024-node segments had been connected together). The average cycle time obtained is shown in Figure 17. Note that the performance obtained in September and November 2002 is repeated

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

from Figure 1. Also, the performance obtained in January 2003 is measured only up to 3,716 processors while that obtained in May 2003 is measured up to 7,680 processors. These tests represent the largest-size machine on those dates but with nodes 0 and 31 configured out of each 32-node cluster. As before, we use all four processors per node.

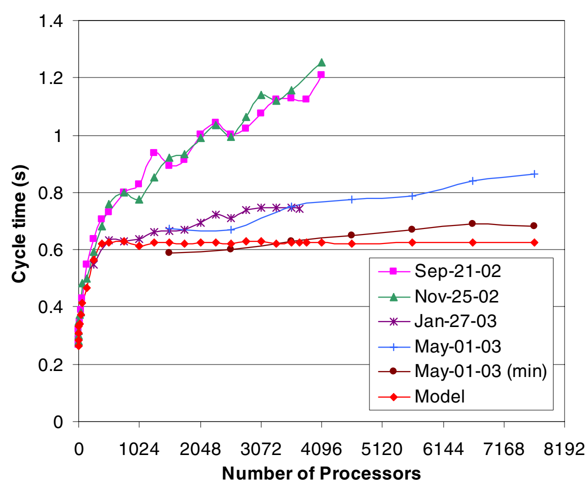


Figure 17: SAGE performance: expected and measured after noise removal

It can be seen that the performance obtained in January and May is much improved over that obtained before noise was removed from the system. Also shown in Figure 17 is the minimum cycle time obtained over 50 cycles. It can be seen that the minimum time very closely matches the expected performance. The minimum time represents those cycles of the application that were least effected by noise. Thus it appears that further optimizations may be possible that will help reduce the average cycle time down towards the minimum cycle time.

The effective performance for the different configurations tested prior to noise removal and after is listed in Table 3. Listed are the cycle time for the different configurations. However, the total processing rate across the system should be considered in comparing the performance as the number of usable processors varies between the configurations. The achieved processing rate of the application that is the total number of cell-updates per second is also listed. This is derived from the cycle time as the processor count $\times$ cells per processor $\div$ cycle time. The cells per processor in all the SAGE runs presented here was 13,500 cells. Note that the default performance on 8,192 processors is an extrapolation from the 4,096 processor performance using a linear performance degradation observed in the measurements of September/November 2002. An important point is that the best observed processing rate with nodes 0 and 31 removed from each cluster is only 15% below the model's expectations.

> FINDING #3
>
> We were able to *double* SAGE's performance by removing noise caused by several types of dæmons, confining dæmons to the cluster manager, and removing the cluster manager and the RMS cluster monitor from each cluster's compute pool.

We expect to be able to increase the available processors by just removing one processor from each of node 0 and 31 of each cluster. This will allow the operating system tasks to be performed without interfering with the application, while at the same time increase the number of usable processors per cluster from 120 (30 out of 32 usable nodes) to 126 (with only two processors removed). This should improve the processing rate by a further 5% just by the increase of the usable processors by 6 per cluster while not increasing the effect of noise.

6 Discussion

In the previous section we saw how the elimination of a few system activities benefited SAGE when running with a specific input deck. We now try to provide some guidelines to generalize our analysis.

To estimate the potential gains on other applications we provide insight on how the computational granularity of a balanced bulk-synchronous application correlates to the type of noise. The intuition behind this discussion is the following: while any source of noise has a negative impact on the overall performance of an application, a few sources of noise tend to have a significant impact. As a rule of thumb, the computational granularity of the application is deemed to "enter in resonance" with noise of a similar harmonic frequency and duration.

In order to explain this correlation, consider the barrier microbenchmark described in Section 4 and running on the optimized ASCI Q configuration. For each of three levels of computational granularity—0, 1, or 5 ms between successive barriers—we analyze the measured barrier-synchronization latency for the

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

Table 3: SAGE effective performance after noise removal

Configuration	Usable processors	Cycle time	Processing rate (10^6 cell updates/sec.)	Improvement factor
Unoptimized system	8,192	1.60	69.1	*—N/A—*
3 processes/node	6,144	0.64	129.3	1.87
Without node 0	7,936	0.87	123.1	1.78
Without nodes 0 and 31	7,680	0.86	120.6	1.75
Without nodes 0 and 31 (best observed)	7,680	0.68	152.5	2.21
Model	8,192	0.63	178.4	2.58

largest node count available when we ran these experiments (960 nodes). The total amount of system noise is, of course, the same for all three experiments.[2] The goal of these experiments is to categorize the relative impact of each of the three primary sources of ASCI Q's noise (kernel activity, RMS dæmons, and TruCluster dæmons) on the barrier microbenchmark's performance.

Figure 18 presents the analysis of our barrier experiments. For each graph, the x axis indicates the duration of an individual occurrence of system noise. The y axis shows the cumulative amount of barrier performance lost to noise, expressed both in absolute time and as a percentage of the total performance lost to noise. (A running total is used because the noise distribution is tail-heavy and would otherwise make the graphs unreadable.) The curves are shaded to distinguish the different sources of noise. As presented by Table 2, instances of noise with a 0–3 ms duration are always caused by kernel activity; instances of noise with a 5–18 ms duration are always caused by RMS dæmons; and, instances of noise with a greater-than-18 ms duration are always caused by TruCluster dæmons. Note that these categories and durations are specific to ASCI Q; noise on other systems will likely stem from other sources and run for differing lengths of time.

Although the *amount* of noise is the same for all three sets of measurements, the *impact* of the noise is clearly different across the three graphs in Figure 18. When the barrier microbenchmark performs back-to-back barriers (Figure 18(a)), the majority of the performance loss—66%—is caused by the high-frequency, short-duration kernel noise; when there is 1 ms of intervening computation between barriers (Figure 18(b)), the largest single source of performance loss—40%—is caused by the medium-frequency, medium-duration RMS dæmons; and, when the barrier microbenchmark performs 5 ms of computation between barriers (Figure 18(c)), more performance is lost to the low-frequency, long-duration TruCluster dæmons—52%—than to all other sources of noise combined.

[2] The total amount of system noise is equal to the superimposition of the data in all four graphs of Figure 13 weighted by the number of nodes represented by each graph.

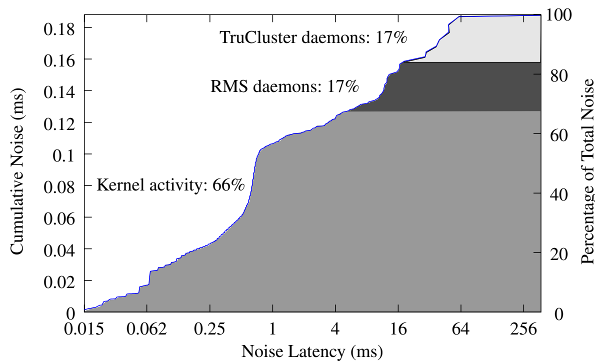
Finding #4

Substantial performance loss occurs when an application resonates with system noise: high-frequency, fine-grained noise affects only fine-grained applications; low-frequency, coarse-grained noise affects only coarse-grained applications.

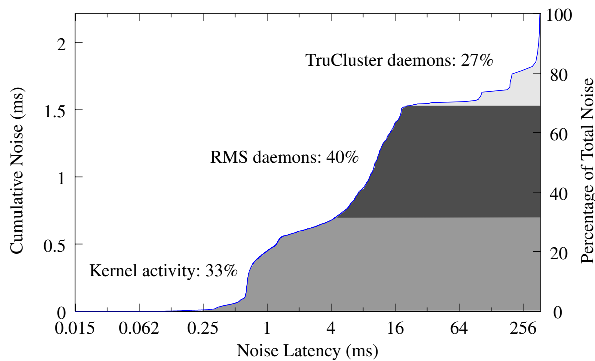
Given that there is a strong correlation between the computational granularity of an application and the granularity of the noise, we make the following observations:

- Load balanced, coarse-grained applications that do not communicate often (e.g., LINPACK [3]) will see a performance improvement of only a few percent from the elimination of the noise generated by node 0. Such applications are only marginally affected by other sources of noise. Intuitively, with a coarse-grained application the fine-grained noise becomes coscheduled as illustrated in Figure 14(b).
- Because SAGE is a medium-grained applications it experiences a substantial performance boost when the medium-weight noise on node 31 and on the cluster nodes is reduced.
- Finer-grained applications, such as deterministic Sn-transport codes [7] which communicate very frequently with small messages, are expected to be very sensitive to the fine-grained noise.

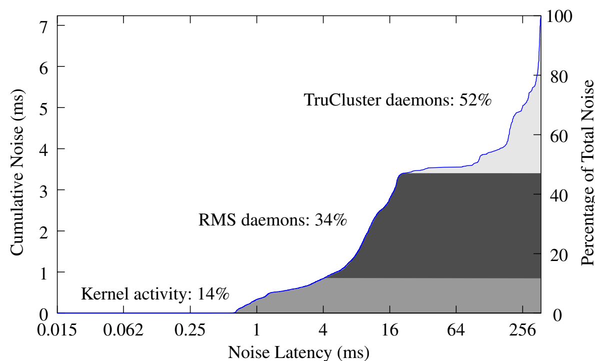
Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM



(a) No intervening computation



(b) 1 ms of intervening computation



(c) 5 ms of intervening computation

Figure 18: Cumulative noise distribution for barrier synchronizations with different computational granularities

7 Related work

The discovery of system noise in parallel systems is not new. Our contribution includes the quantification of this noise on a large-scale supercomputer, the use of performance models to quantify the gap between measured and expected performance, the characterization of noise as a collection of harmonics, the use of a discrete-event simulator to evaluate the contribution of each component of the noise to the overall application behavior, and the correlation of the computational granularity of an application to the granularity of the noise.

Many researchers have observed performance variability in large-scale parallel computers. For instance, Kramer and Ryan [12] as well as Srivastava [19] found that application run times may vary significantly. Kramer and Ryan were able to attribute some of the performance variability to the mapping of processes to processors on the Cray T3E. However, they also observed performance variability on the embarrassingly parallel (EP) benchmark [1] running on the Pittsburgh Supercomputing Center's AlphaServer cluster. Because EP performs very little communication and should therefore be robust to both processor mapping and network performance, Kramer and Ryan concluded that noise within the nodes was the source of the performance variability.

Srivastava [19] and Phillips et al. [17] ran experiments—also on the Pittsburgh Supercomputing Center's AlphaServer cluster—and noticed that leaving idle one processor per node reduces performance variability. Phillips et al. concluded that "the inability to use the 4th processor on each node for useful computation" is a major problem, and they conjectured that "a different implementation of [the] low level communication primitives will overcome this problem". However, in more recent work the authors investigated techniques to eliminate "stretches" ("noise" in our terminology) and discovered that fine-tuning the communication library and using blocking receives in their application alleviates the performance penalty caused by operating-system interference [8].

In what may be the most similar work to ours, Mraz [14] observed performance variation in point-to-point communication in the IBM SP1, determined that this variation was caused by a variety of factors—dæmons, interrupts, and other system activity—and analyzed multiple techniques to reduce the performance loss. He found that raising the priority of user

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

applications above that of the system dæmons reduced the coarse-grained noise. Raising the priority further also reduced the fine-grained noise but at the cost of system stability lost to priority inversion (e.g., the system hangs on the first page fault if the application runs at a higher priority than the operating system's page-fault handler). Mraz concluded that globally synchronizing the system clocks gave the best results overall as it generally caused the dæmons to run in a coscheduled fashion and did not degrade system stability. (The technique of coscheduling via global clock synchronization was also patented by Grice and Hochschild [6].)

Hard real-time systems are designed to execute workloads in a consistent amount of time from run to run. As a result, the effects of noise on traditional time-shared systems are well known to researchers in the area of hard real-time systems. For example, the measurement and analysis of many short, purely sequential computations has also been used as a means to identify system effects by Monk et al. [13]. This work was primarily aimed at analyzing high performance embedded real-time systems. They noted that the effect of the operating system was platform-specific and could significantly delay short sequential computations.

Burger et al. [2] took a different approach to performance variability analysis than the previously mentioned works. Rather than *observe* performance variability on existing systems, they instead *injected* noise into a simulated system and measured the impact of this noise on the performance of various parallel-application kernels. They found that when the noise was not coscheduled, it caused a performance degradation of up to 800% in tightly coupled kernels.

8 Conclusions

To increase application performance, one traditionally relies upon algorithmic improvements, compiler hints, and careful selection of numerical libraries, communication libraries, compilers, and compiler options. Typical methodology includes profiling code to identify the primary performance bottlenecks, determining the source of those bottlenecks—cache misses, load imbalance, resource contention, etc.—and restructuring the code to improve the situation.

This paper describes a figurative journey we took to improve the performance of a sizable hydrodynamics application, SAGE, on the world's second-fastest supercomputer, the 8,192-processor ASCI Q machine at Los Alamos National Laboratory. On this journey, we discovered that the methodology traditionally employed to improve performance falls short and that traditional performance analysis tools alone are incapable of yielding maximal application performance. Instead, we developed a performance-analysis methodology that includes the analysis of artifacts that degrade application performance yet are not part of an application. The strength of our methodology is that it clearly identifies all sources of noise and formally categorizes them as "harmonics"; it quantifies the total impact of noise on application performance; and, it determines which sources of noise have the greatest impact on performance and are therefore the most important to eliminate. The net result is that we managed to almost *double* the performance of SAGE without modifying a single line of code—in fact, without even recompiling the executable.

The primary contribution of our work is the methodology presented in this paper. While other researchers have observed application performance anomalies, we are the first to determine how fast an application could *potentially* run, investigate even those components of a system that would not be expected to significantly degrade performance, and propose alternate system configurations that dramatically reduce the sources of performance loss.

Another important contribution is our notions of "noise" and "resonance". By understanding the resonance of system noise and application structure, others can apply our techniques to other systems and other applications.

The full, 8,192-processor ASCI Q only recently became operational. Although it initially appeared to be performing according to expectations based on the results of LINPACK [3] and other benchmarks, we determined that performance could be substantially improved. After analyzing various mysterious, seemingly contradictory performance results, our unique methodology and performance tools and techniques enabled us to finally achieve our goal of locating ASCI Q's missing performance.

> "Nobody reads a mystery to get to the middle. They read it to get to the end. If it's a letdown, they won't buy anymore. The first page sells that book. The last page sells your next book."
>
> — Mickey Spillane

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

Acknowledgments

Shedding light on the interactions among the myriad components of a large-scale system would not have been possible without the light shed by the helpful interactions with many people. In particular, we would like to thank Eitan Frachtenberg for first noticing the effect of noise on Sweep3D; Juan Fernandez for pioneering the performance-debugging methodology in the Buffered Coscheduling framework; David Addison for his help in optimizing the allreduce; Manuel Vigil and Ray Miller for providing access to ASCI Q and coordinating our experiments with the ASCI Q team; Amos Lovato, Rick Light, and Joe Kleczka for helping diagnose the sources of noise, preparing our experiments, and providing system support; and finally, Ron Green, Niraj Srivastava, Malcolm Lundin, Mark Vernon, and Steven Shaw for operational support during the experiments.

This work was supported by the ASCI program at Los Alamos National Laboratory. Previous work related to the methodology applied in this paper was supported in part by LDRD ER 2001034ER, "Resource Utilization and Parallel Program Development with Buffered Coscheduling". Los Alamos National Laboratory is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36.

References

[1] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995. Available from `http://www.nas.nasa.gov/Research/Reports/Techreports/1995/PDF/nas-95-020.pdf`.

[2] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed shared-memory multiprocessors. In *Proceedings of Supercomputing '94*, Washington, D.C., November 14–18, 1994. Available from `ftp://ftp.cs.utexas.edu/pub/dburger/papers/TR_1244.pdf`.

[3] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, Computer Science Department, University of Tennessee, Knoxville, Tennessee, 1989. Available from `http://www.netlib.org/benchmark/performance.ps`.

[4] Juan Fernandez, Fabrizio Petrini, and Eitan Frachtenberg. BCS MPI: A new approach in the system software design for large-scale parallel computers. In *Proceedings of SC2003*, Phoenix, Arizona, November 15–21, 2003.

[5] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-fast resource management. In *Proceedings of SC2002*, Baltimore, Maryland, November 16–22, 2002. Available from `http://sc-2002.org/paperpdfs/pap.pap297.pdf`.

[6] Donald G. Grice and Peter H. Hochschild. Resource allocation synchronization in a parallel processing system. United States patent 5,600,822, International Business Machines Corporation, Armonk, New York, February 4, 1997. Available from `http://patft.uspto.gov/`.

[7] Adolfy Hoisie, Olaf Lubeck, Harvey Wasserman, Fabrizio Petrini, and Hank Alme. A general predictive performance model for wavefront algorithms on clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing (ICPP-2000)*, Toronto, Canada, August 21–24, 2000. Available from `http://www.c3.lanl.gov/par_arch/pubs/icpp.pdf`.

[8] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Proceedings of the Terascale Performance Analysis Workshop, International Conference on Computational Science (ICCS 2003)*, volume 2660 of *Lecture Notes in Computer Science*, pages 23–32, Melbourne, Australia, June 2–4, 2003. Springer-Verlag. Available from `http://charm.cs.uiuc.edu/papers/namdPerfStudy.pdf`.

[9] Darren J. Kerbyson, Hank J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman, and Michael Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of SC2001*, Denver, Colorado, November 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap255.pdf`.

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM

[10] Darren J. Kerbyson, Adolfy Hoisie, and Harvey J. Wasserman. Use of predictive performance modeling during large-scale system installation. *Parallel Processing Letters*, 2003. World Scientific Publishing Company. Available from http://www.c3.lanl.gov/par_arch/pubs/KerbysonSPDEC.pdf.

[11] Ken Koch. How does ASCI actually complete multi-month 1000-processor milestone simulations? In *Proceedings of the Conference on High Speed Computing*, Gleneden Beach, Oregon, April 22–25, 2002. Available from http://www.ccs.lanl.gov/salishan02/koch.pdf.

[12] William T. C. Kramer and Clint Ryan. Performance variability of highly parallel architectures. In *Proceedings of the International Conference on Computational Science (ICCS 2003)*, volume 2659 of *Lecture Notes in Computer Science*, pages 560–569, Melbourne, Australia, June 2–4, 2003. Springer-Verlag. Available from http://www.nersc.gov/~kramer/papers/variation-short.pdf.

[13] Leonard Monk, Richard Games, John Ramsdell, Arkady Kanevsky, Curtis Brown, and Perry Lee. Real-time communications scheduling: Final report. Technical Report MTR 97B0000069, The MITRE Corporation, Center for Integrated Intelligence Systems, Bedford. Massachusetts, May 1997. Available from http://www.mitre.org/tech/hpc/docs/rtcs_final.pdf.

[14] Ronald Mraz. Reducing the variance of point to point transfers in the IBM 9076 parallel computer. In *Proceedings of Supercomputing '94*, pages 620–629, Washington, D.C., November 14–18, 1994. Available from http://www.computer.org/conferences/sc94/mrazr.ps.

[15] Fabrizio Petrini and Wu-chun Feng. Improved resource utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 16:123–144, 2001. Available from http://www.c3.lanl.gov/~fabrizio/papers/paa00.ps.

[16] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002. ISSN 0272-1732. Available from http://www.computer.org/micro/mi2002/pdf/m1046.pdf.

[17] James C. Phillips, Genbing Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC2002*, Baltimore, Maryland, November 16–22 2002. Available from http://www.sc-2002.org/paperpdfs/pap.pap277.pdf.

[18] Quadrics Supercomputers World Ltd. *RMS Reference Manual*, June 2002.

[19] Niraj Srivastava. Performance variability study. Technical report, Hewlett-Packard Company, November 5, 2002.

Proceedings of the ACM/IEEE SC2003 Conference (SC'03)
1-58113-695-1/03 $ 17.00 © 2003 ACM