LA-UR- 03 -3116

Approved for public release;
distribution is unlimited.

C. 1

Title: THE CASE OF THE MISSING SUPERCOMPUTER PERFORMANCE: ACHIEVING OPTIMAL PERFORMANCE ON THE 8,192 PROCESSORS OF ASCI Q

Author(s): Fabrizio Petrini, 154601, CCS-3
Darren J. Kerbyson, 176262, CCS-3
Scott Pakin, 179752, CCS-3

Submitted to: SuperComputing Conference 2003
Phoenix, AZ

Los Alamos
NATIONAL LABORATORY

99

LAUR & JAAP

SuperComp 2003

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

{fabrizio,djk,pakin}@lanl.gov

May 2, 2003

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 undetectable—performance problem. We identified the source of the problem, eliminated the problem, and "closed the loop" by demonstrating improved application performance. We present our methodology and provide insight into performance analysis that is immediately applicable to other large-scale cluster-based 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, 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 [8]. 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 is 20 Tflop/s and will be listed as #2 in the list of the top 500 fastest computers.[1]

The ultimate goal when running an application on a parallel 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 clusters are difficult to optimize for, as their deep memory hierarchies can incur orders-of-magnitude performance loss in the absense 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-

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

TABLE 1: Performance analysis tools and techniques

Technique	Meaning	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 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 reboot of the entire system can take several hours [6] 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 remeasuring 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, and we present our conclusions in Section 7.

2 Performance expectations

Based on the Top 500 results, ASCI Q appears to be performing well. It runs the LINPACK [1] benchmark at 75% of peak performance, which is well within range for machines of that 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 [3, 4]. 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 a large-scale hydrodynamic application, SAGE, was developed for the express purpose of predicting the performance of SAGE 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% [5]. The HP ES45 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. The results of the two measurements are consistent with each other although they rapidly diverge from the performance predicted by the SAGE performance model (Figure 1). At 4,096 processors, SAGE's cycle time was *twice* the time 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.

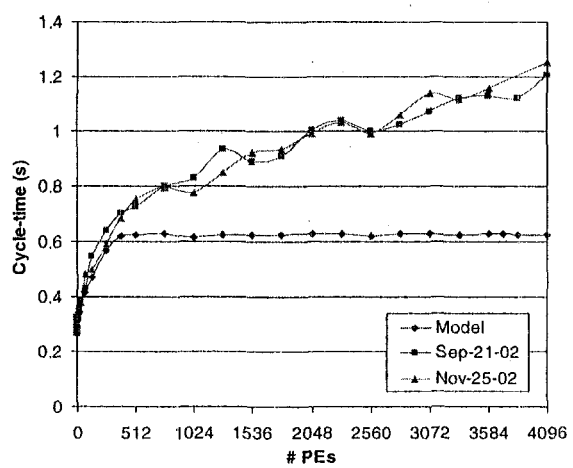


Figure 1: SAGE performance: expected and measured

MYSTERY #1

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

In order to identify why there was a difference in performance between the measured and expected performance, we performed a battery of tests on ASCI Q. The most 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 4 processors per node. Note that a log scale is used on the *x* axis. It can be seen that the only significant difference occurs when using all four processors per node thus giving confidence to the model being accurate.

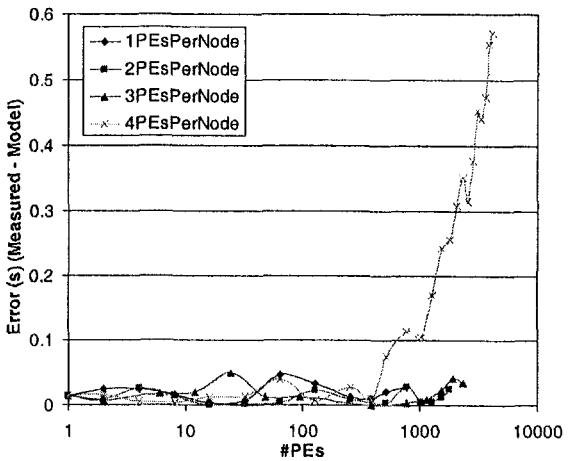


Figure 2: Error in modeled and measured SAGE performance when using 1, 2, 3, or 4 processors per node

It is also interesting to note that, above approximately 1,500 processors, 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. Phillips et al. observed similar behavior on a molecular-dynamics application running at the Pittsburgh Supercomputing Center using similar hardware to that in ASCI Q [9]. They 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 [their] low level communication primitives will overcome this problem".

Like Phillips et al., we also analyzed application performance variability. Each computation cycle within SAGE performs 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 (token_get and token_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

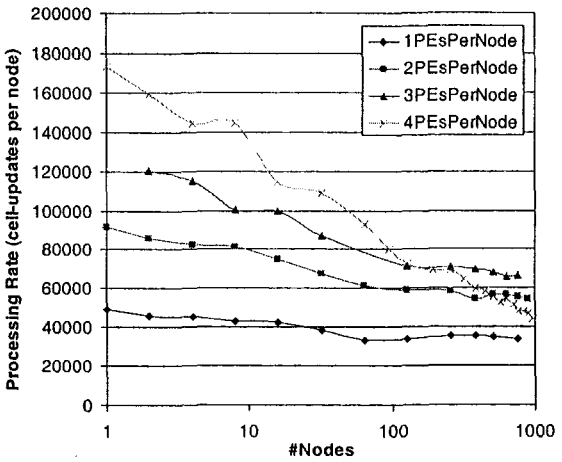
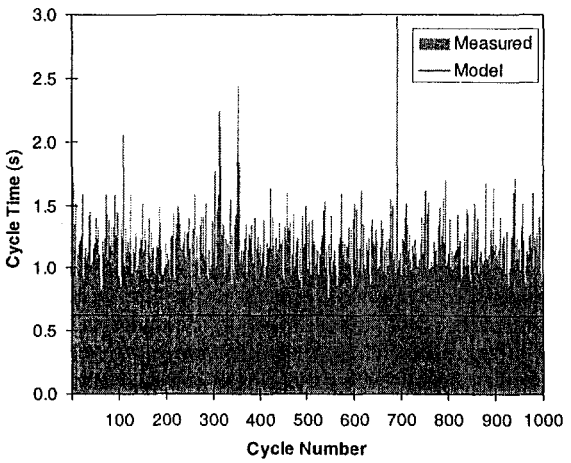
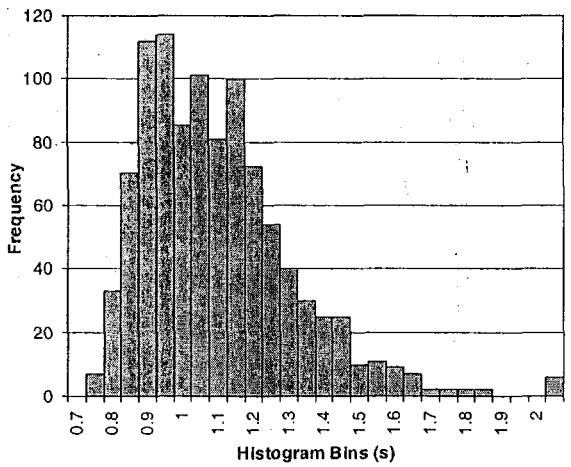


Figure 3: Effective SAGE processing rate when using 1, 2, 3, or 4 processors per node



(a) Variability



(b) Histogram

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

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.

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 effected 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

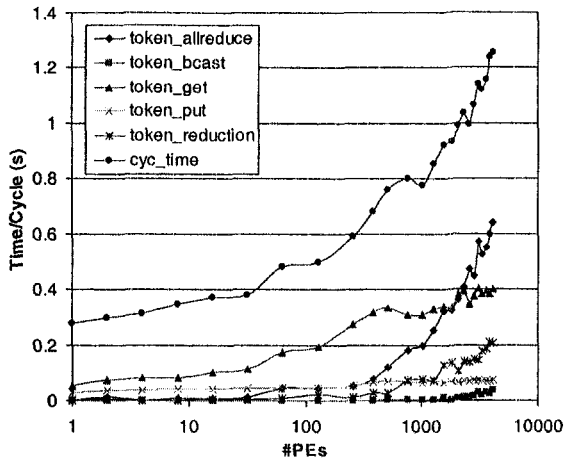


Figure 5: Profile of SAGE cycle time

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.

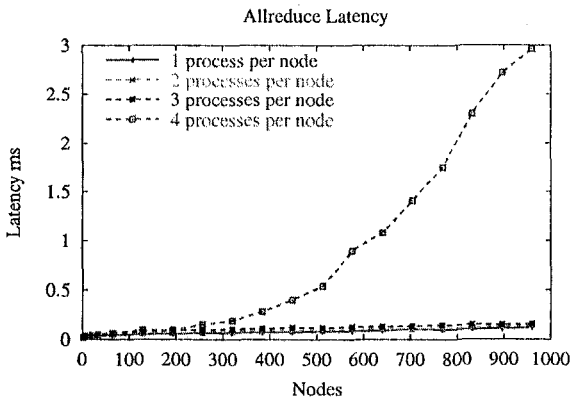


Figure 6: allreduce latency as a function of the number of nodes and processes per node

Figure 7 provides more clues to our analysis. 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 linearly 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 will consider a barrier benchmark later in the analysis.

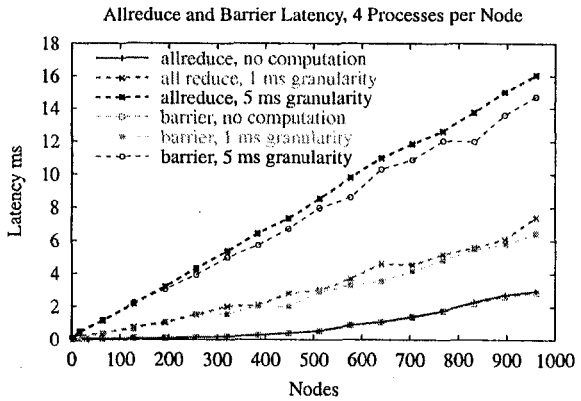


Figure 7: allreduce and barrier latency with varying amounts of intervening computation

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 to poll for a limited time (100 µs) and then block, 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.

3.2 Analyzing the 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 to absorb them. When there is no spare processor,

the application must relinquish one of its processors to the system activity. Doing so may introduce performance variability, which we refer to as "noise".

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

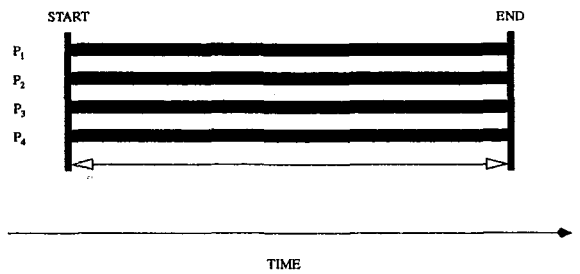


Figure 8: Performance-variability microbenchmark

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 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.

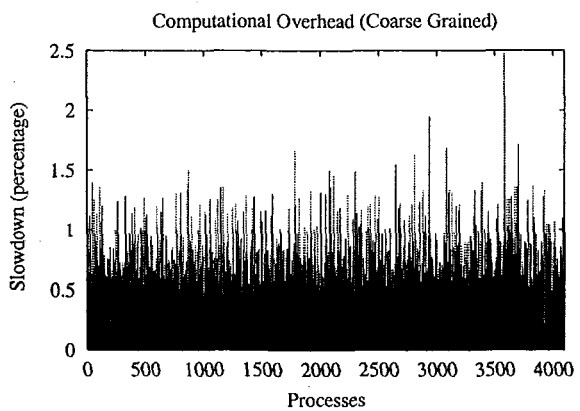


Figure 9: Results of the performance-variability microbenchmark

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 to run for exactly 1 ms in the absense 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.

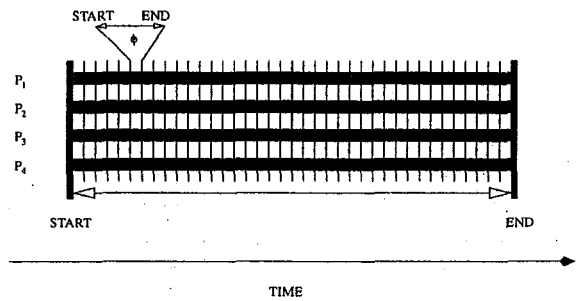


Figure 10: Performance-variability microbenchmark, second attempt

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 attempt 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 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.

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-

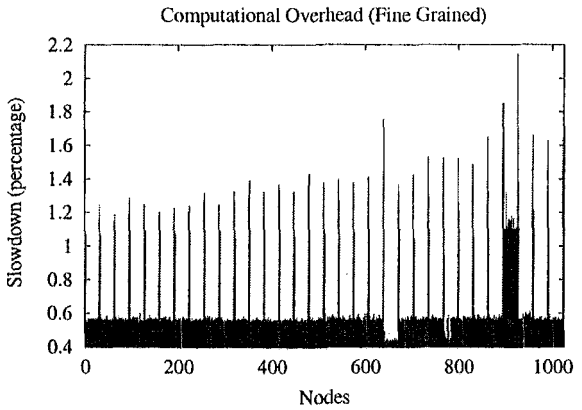


Figure 11: Results of the per-node performance-variability microbenchmark

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 are slower than the others. This pattern repeats for each cluster of 32 nodes.

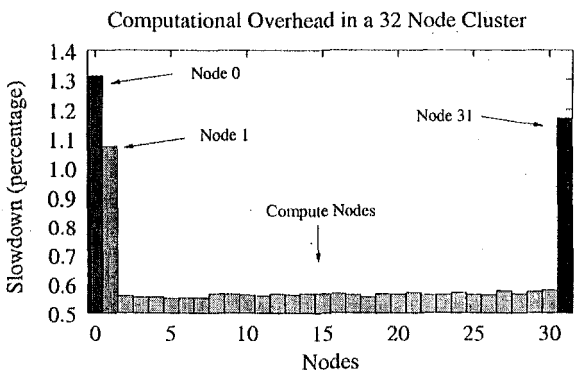


Figure 12: Slowdown per node within each 32-node cluster

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 contains one such histogram for each of four groupings of nodes: nodes 0, 1, 2–30, and 31 of each 32-node cluster. These histograms 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 2–30 of a cluster we can identify two events that happen regularly every 30 seconds and whose durations are 16 and 19 ms. This means that a slice of computation that should take 1 ms occasionally takes 16 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.

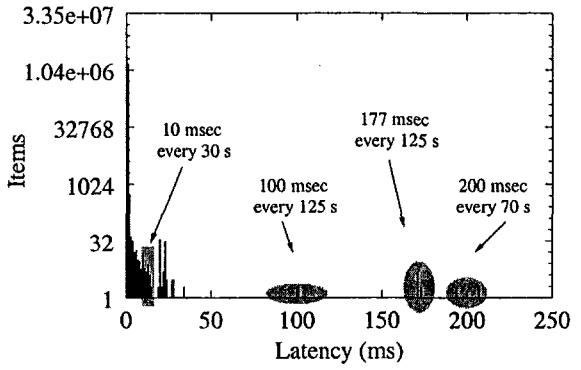
Using a number of techniques on QB and other parallel machines, we were able to identify the source of most activities. As a general rule, these activities happen at regular intervals. The two events that take 16 and 19 ms on each node 2–30 are generated by Quadrics's resource management system, RMS [10], 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 hundreds of microseconds. Other dæmons that implement the parallel file system and TruCluster are the source of the noise on nodes 0 and 31.

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 clear guidelines to identify and eliminate the sources of noise.

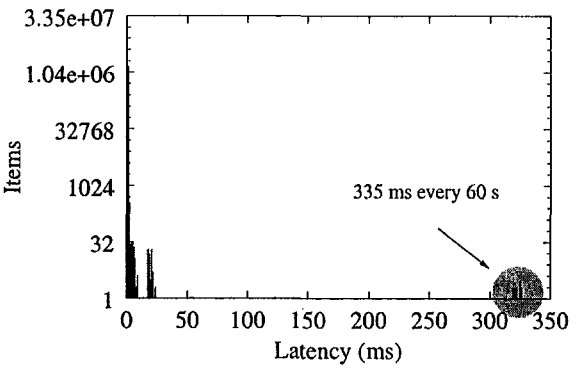
3.3 Effect on system performance

Figure 14(a) provides some 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. For example, if only one process out of 4,096 experiences a delay of 100 ms, on an application that barrier-synchronizes every 1 ms, then the whole application will run 100 times slower!

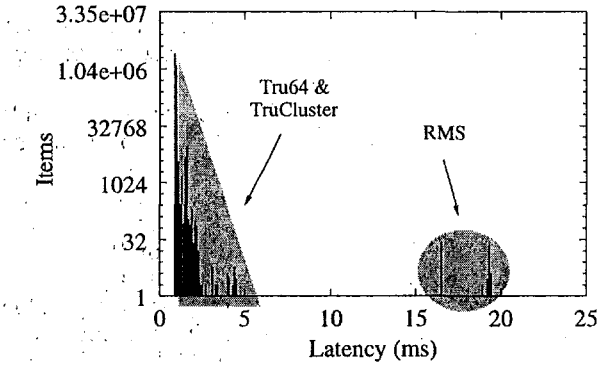
While the obvious solution is to remove any type of noise in the system, in certain cases it may not be possible or cost effective to remove dæmons or kernel threads that perform essential activities as resource management, monitoring, parallel file system, etc. Figure 14(b) suggests a possible solution that doesn't require the elimination of the system activities. By coscheduling these activities we pay only once, irrespective of the machine size. We recently developed a prototype coscheduler as a Linux kernel module [2, 7] and we are in the process of investigating the performance implications.



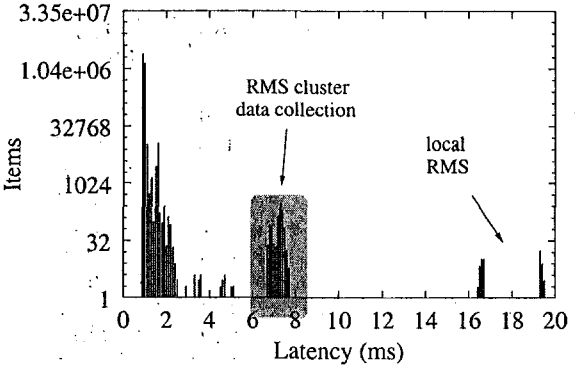
(a) Latency distribution on node 0



(b) Latency distribution on node 1

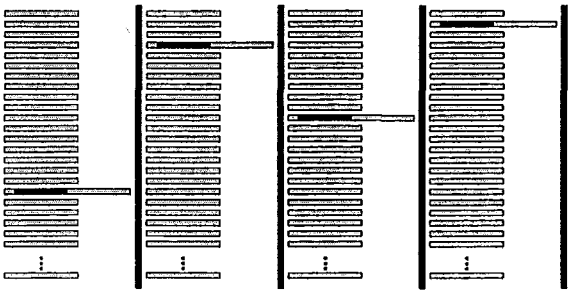


(c) Latency distribution on nodes 2–30

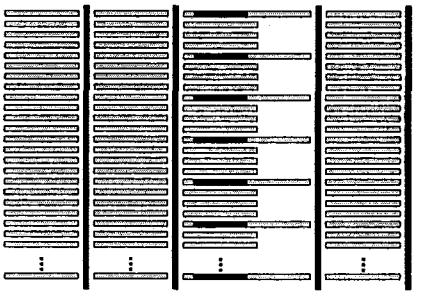


(d) Latency distribution on node 31

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



(a) Uncoordinated noise



(b) Coscheduled noise

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

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 background noise on "ordinary" nodes 2–30.

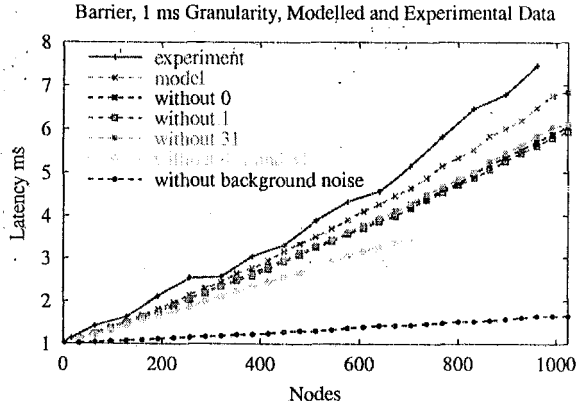


Figure 15: Simulated vs. experimental data with progressive exclusion of various sources of noise in the system

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 a few nodes.

4 Eliminating the sources of noise

On January 25, 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 dæmons from nodes 1 and 2 to node 0 on each cluster, in order to confine the heavyweight noise to this node.

As an initial test of the efficacy of these optimizations we used a simple benchmark in which all nodes 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 and without the noise-reducing optimizations described above.

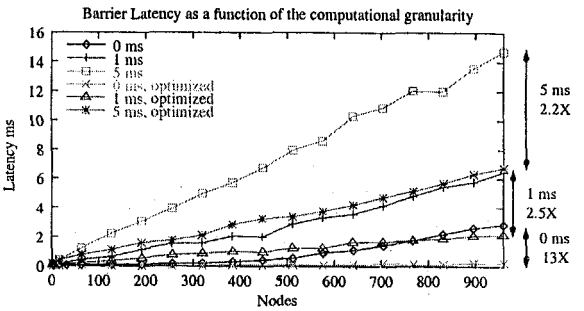


Figure 16: Performance improvements obtained on the barrier-synchronization microbenchmark for different computational granularities

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 improvements 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 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.

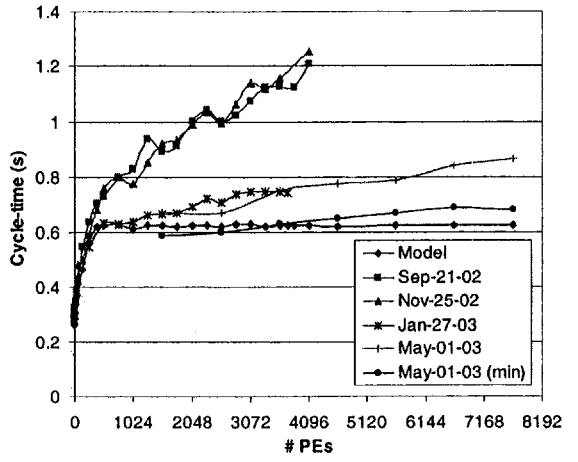


Figure 17: SAGE performance: expected and measured after noise removal

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 2. 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 × cells per processor ÷ 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.

FINDING #3

The performance of SAGE has substantially improved. The best observed processing rate with nodes 0 and 31 removed from each cluster is only 15% below the model's expectations.

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 have seen how the elimination of a few system activities benefited SAGE with a specific input deck. We now try to provide some guidelines to generalize our analysis.

In order 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 the application, a few sources of noises 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 benchmark of Figure 16 for the three optimized configurations with 0, 1 and 5 ms of computational granularity. For each of these cases we analyze the barrier-synchronization latency for the largest node count. For example, in such a configuration the barrier takes 0.19 ms, 2 ms, and 7 ms respectively. In each case we consider the cumulative latency distribution, as shown in Figure 18. Each graph describes how different sources of noise affect the barrier-synchronization latency.

Figure 18(a) shows the results for a sequence of barriers without any computation (which represents an extreme case of fine grained computation). We can see that 66% of the delay is caused by fine-grained noise, generating computational holes of less than 4 ms. The heavyweight, but less frequent, noise generated by node 0 in each cluster impacts the barrier latency by only 17%.

With 1 ms of computational granularity, the impact of the fine-grained noise is reduced to only 33% while the relative effect of the heavyweight noise grows to 27%, as shown in Figure 18(b). The primary source of degradation is the medium-grained noise generated by RMS on node 31 and on each cluster node.

Finally, we can see that with 5 ms of computational granularity (Figure 18(c)), more than half of the barrier latency is caused by node 0, while 33% is caused by RMS on node 31 plus the cluster nodes.

TABLE 2: 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 PEs/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

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 can make the following observations:

- Load balanced, coarse-grained applications that do not communicate often (e.g., LINPACK [1]) 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 fine-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 [3] which communicate very frequently with small messages, are very sensitive to the fine-grained noise.

7 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 sizeable 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 incabable 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. Our methodology employs exotic tools such as analytical performance models and application-specific microbenchmarks. 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.

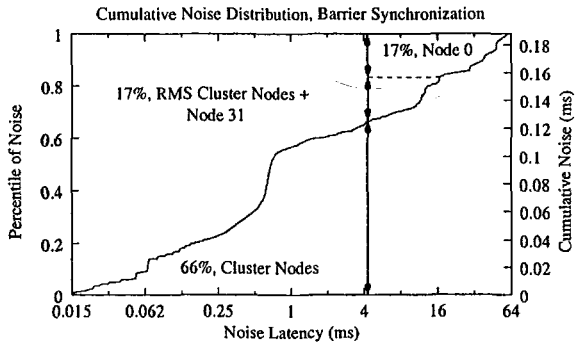
The primary contribution of our work is the methodology presented in this paper. While other researchers have observed application performance anomalies, we are the first to determine how fast an application could *potentially* run, investigate even those components of a system that would not be expected to significantly degrade performance, and propose alternate system configurations that dramatically reduce the sources of performance loss. A secondary contribution includes 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 [1] 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.

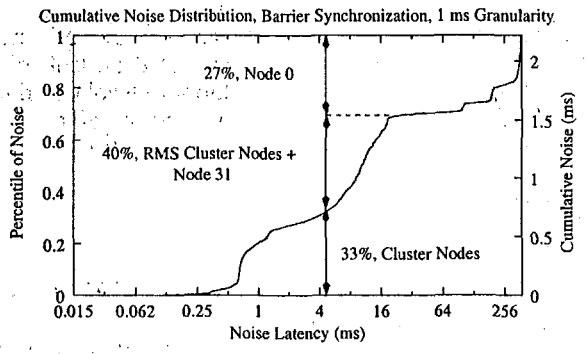
> "Nobody reads a mystery to get to the middle. They read it to get to the end. If it's a letdown, they won't buy anymore. The first page sells that book. The last page sells your next book."
>
> — Mickey Spillane

References

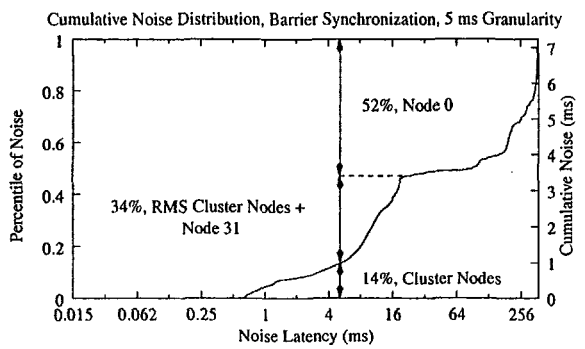
[1] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, Computer Science



(a) Barrier synchronizations with no intervening computation



(b) 1 ms



(c) 5 ms

Figure 18: Cumulative noise distribution for different computational granularities

Department, University of Tennessee, Knoxville, Tennessee, 1989. Available from `http://www.netlib.org/benchmark/performance.ps`.

[2] E. Frachtenberg, F. Petrini, et al. STORM: Lightning-fast resource management. In *Proceedings of SC2002*. Baltimore, Maryland, Nov. 16–22 2002. Available from `http://sc-2002.org/paperpdfs/pap.pap297.pdf`.

[3] A. Hoisie, O. Lubeck, et al. 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, Aug. 21–24, 2000. Available from `http://www.c3.lanl.gov/par_arch/pubs/icpp.pdf`.

[4] D. J. Kerbyson, H. J. Alme, et al. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of SC2001*. Denver, Colorado, Nov. 10–16, 2001. Available from `http://www.sc2001.org/papers/pap.pap255.pdf`.

[5] D. J. Kerbyson, A. Hoisie, et al. Use of predictive performance modeling during large-scale system installation. *Parallel Processing Letters*, 2003. World Scientific Publishing.

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

[7] F. Petrini and W. 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`.

[8] F. Petrini, W. Feng, et al. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, Jan./Feb. 2002. ISSN 0272-1732. Available from `http://www.computer.org/micro/mi2002/pdf/m1046.pdf`.

[9] J. C. Phillips, G. Zheng, et al. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC2002*. Baltimore, Maryland, Nov. 16–22 2002. Available from `http://www.sc-2002.org/paperpdfs/pap.pap277.pdf`.

[10] Quadrics Supercomputers World Ltd. *RMS Reference Manual*, Jun. 2002.