

LA-UR-PENDING

*Approved for public release;
distribution is unlimited.*

Title: **Identifying and Eliminating the Performance Variability on
the ASCI Q Machine**

Author(s): Adolfy Hoisie, CCS-3
Darren J. Kerbyson, CCS-3
Scott Pakin, CCS-3
Fabrizio Petrini, CCS-3
Harvey J. Wasserman, CCS-3
Juan Fernandez-Peinador, CCS-3

Submitted to:

Los Alamos
NATIONAL LABORATORY

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Identifying and Eliminating the Performance Variability on the ASCI Q Machine

Adolfy Hoisie, Darren J. Kerbyson, Scott Pakin, Fabrizio Petrini, Harvey J. Wasserman, Juan Fernandez-Peinador

Performance and Architecture Lab (PAL)
CCS-3
Los Alamos National Laboratory

Summary

This report is split into several sections:

- 1 – examines the performance of SAGE on QB
- 2 – examines the factors that contribute to the achievable application performance on each node
- 3 – details the performance factors across the system
- 4 – Gives results from a detailed model of the performance factors
- 5 – Gives an indication of possible application performance that may be achievable if the identified performance factors are removed
- 6 – Recommendation to improve the performance, decrease its variability and overall improving the system software
- 7 – An appendix presenting a Quadrics network debugging scenario which we have utilized consistently on QB in order to reveal potential hardware problems. We strongly recommend that the system people include this procedure in their standard running scripts and run it periodically. First time we utilized it we found many Quadrics switches that were not reporting errors to the RMS database and were not operating optimally for a variety of reasons. We guess that this situation may exist on QA too, as we never had a chance of debugging the network on that machine.

The analysis is quite complex in this work. It flows as follows:

First we measure the performance of SAGE and note 1. its departure from the accurate modeled performance (i.e., the expected performance), and 2. its variability, seemingly random. We also compare the performance of SAGE when using 1,2,3 and 4 processors per node against the model. The performance degradation and its variability are present when using 4 processors per node only. This is presented in section 1.

With that in mind, we proceed to identifying the factors that contribute to the performance degradation observed, in section 2. For instance, profiling SAGE in terms of the communication kernels it utilizes, we note that apparently `allreduce` is responsible for the performance degradation and variability, while all the other communication kernels exhibit expected performance. Furthermore, since `allreduce` is composed of a `reduce` and a `broadcast`, we analyze the performance of each of these 2 kernels to observe that in fact the performance of `reduce` is suboptimal and variable.

Next logical step was to attempt to improve the performance of `reduce`, both algorithmically and by optimizing the runtime environment, as detailed in section 2.1. The overall improvement of

reduce, a whopping factor of 7 though didn't seem to lead to overall improvements in the performance of SAGE.

We then proceeded with a groundbreaking analysis of the computational nodes, the first of its kind that we are aware of for large-scale parallelism, beginning in section 2.2.

We designed and implemented a set of benchmarks to reveal and quantify the level of noise in the nodes. We quantify all the perturbations in the system in terms of frequency and duration, assign them to system processes (kernel or daemons), and link these perturbations to "types" of nodes with selected functionality in the system. Many interesting observations are made here: all nodes are affected by one or more types of periodic events, the frequency and duration of these events is widely different, the I/O clusters and `rms` impose a coarse-grain periodicity, etc. Knowing all these events that contribute to performance degradation and variability allowed us to write an accurate discrete-event simulator for analyzing these issues. Of course, since the proper functioning of the system requires these processes to run on it, further analysis would not be possible without the simulator as experiments are not possible.

With the simulator we analyze the "what if" scenarios, when one or more sources of noise are eliminated, in section 4. There are many counterintuitive observations here, for example that the positive impact from removing the seemingly bigger sources of noise is small. In this section we reveal the "real" source of variability: the compounded effect of the system events leading to serious synchronization problems at large scale. Hence, in fact, the degradation in the performance of reduce is due to synchronization delays and not to communication delays!

In section 5 we relate back this quantified noise into the SAGE model showing that it accounts for all the performance variability described in section 1.

In the last section we make some recommendation for improving the situation, although it is now crystal clear that a 100% solution is not achievable, since not all system activity can be removed.

We have identified the root causes of the performance degradation and variability: system activity on the nodes (kernel and daemons), and not network traffic delays.

We claim that with this report we completely and conclusively clarified the problems of Q variability that have been widely noticed on the system but not comprehensively analyzed before this work.

1. SAGE Performance

SAGE, version 20001220 was executed on the 1024 node QB system. The scaling behavior of SAGE running test case “timing.input” was recorded for a number of test cases.

- Scaling up to 4096 PEs using 1 Rail (Figure 1.1)
- Scaling up to 4096 PEs using 2 Rails (Figure 1.2)
- Scaling when using either 1 PE, 2 PEs, 3PEs, or 4PEs per node (Figure 1.3, 1.4, and 1.5)
- Performance breakdown of communication components (Figure 1.6)
- 1000 cycle run to examine variability on 3584 PEs (Figure 1.7, and Figure 1.8)

In all cases the performance of SAGE was recorded using the cycle time metric – the time taken to execute a single cycle of timing.input.

From these sets of measurements the components that add to the variability of the performance of SAGE at higher PE counts were identified. As will be seen there is an unexpected increase in the time spent in the *allreduce* collective communications with scaling (especially at and above 1024 PEs).

1.1 Scaling using 1 Rail

The performance of SAGE using 1-rail of QB is shown in Figure 1.1. It can be seen that the performance is almost identical to that earlier in September. It should also be noted that the performance of SAGE is significantly above that expected as given by the CCS-3 performance model. At 1024 PEs the performance is 14% worse than expected rising to 70% worse at 4096 PEs.

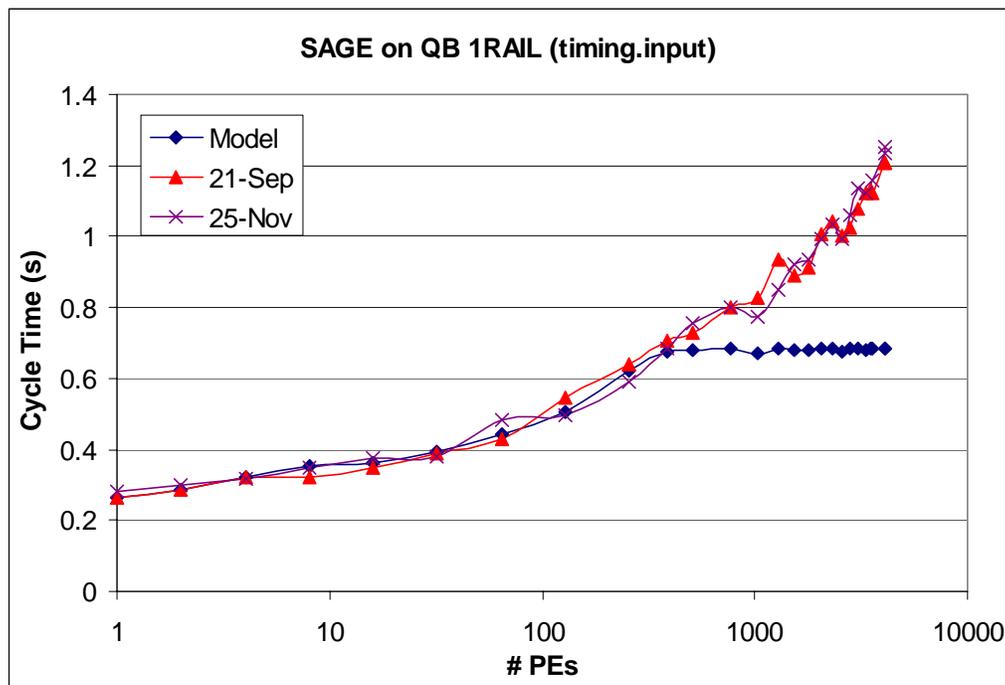


Figure 1.1 – SAGE scaling using 1 Rail (timing.input)

1.2 Scaling using 2 Rails

The performance of SAGE was also measured when using 2-Rails and is compared to the performance on 1 RAIL in Figure 1.2. It can be seen that the 2-Rail performance is slightly better than using 1-Rail at 64 PEs and above. This actually corresponds to an improvement expected from the performance model. However, above 1024PEs, the performance is still significantly worse than expected – the same situation as when using 1-rail.

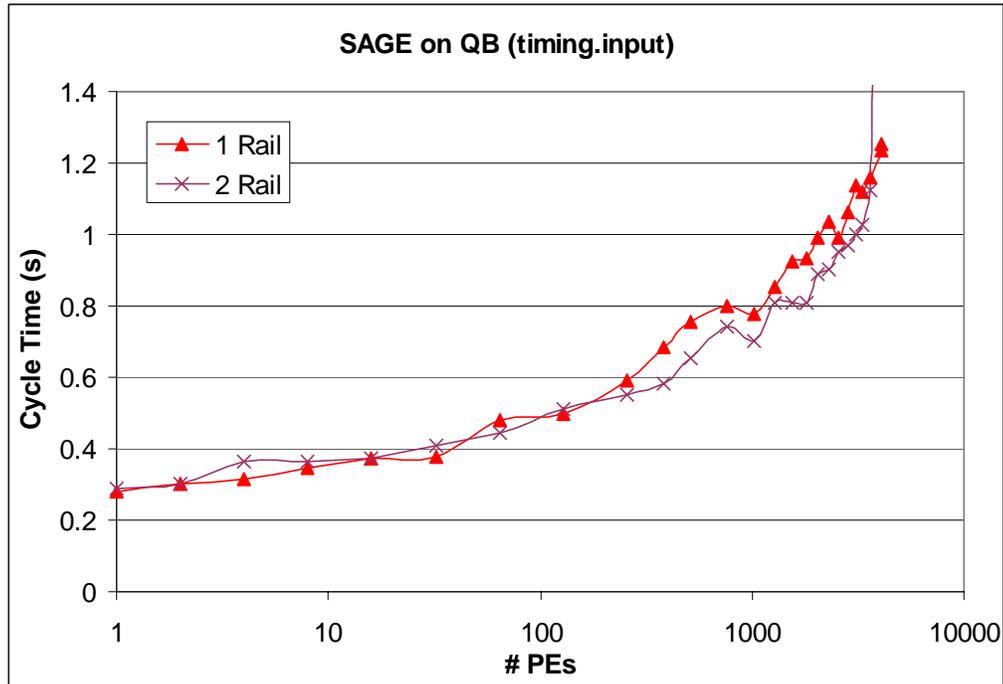


Figure 1.2 – Comparison of the SAGE scaling using 1 and 2 Rails (timing.input)

1.3 Scaling using 1, 2, 3, and 4 PEs per node

In order to investigate the performance of SAGE further, we measured the performance when using a different number of PEs per node. In this analysis, we varied the number of PEs per node: 1, 2, 3, or 4. The measured performance is shown in Figure 1.3.

It can be seen that the performance of SAGE gets worse as more PEs per node are used. When using 1, 2, or 3 PEs per node the scaling behavior is reasonable. However, when using all 4 PEs per node the performance is significantly worse.

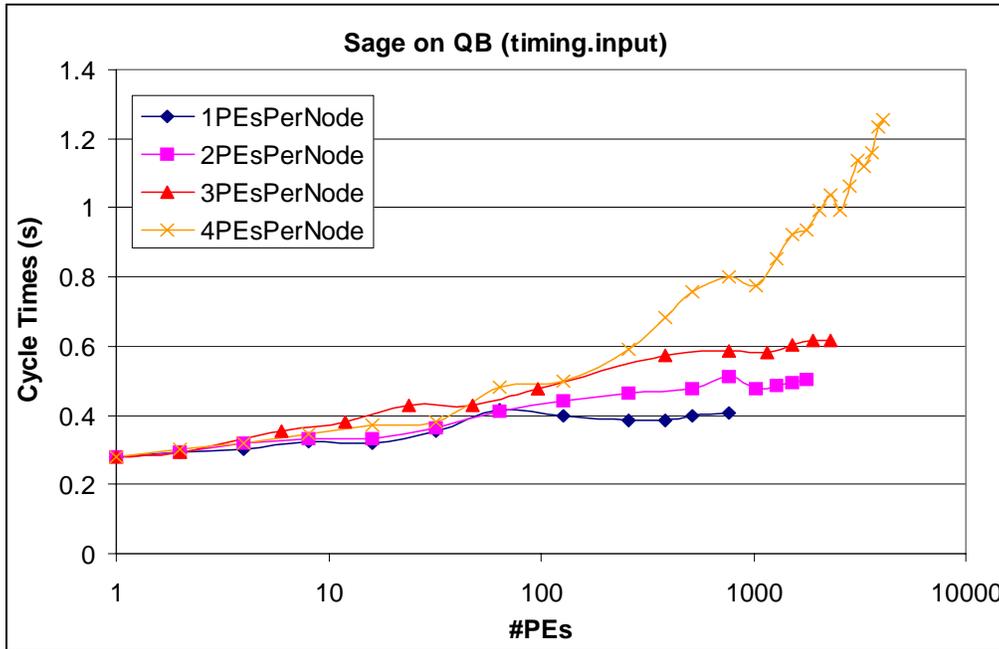


Figure 1.3 – SAGE scaling using 1PE, 2PEs, 3PEs, or 4PEs per node

The difference between the measured performance for using 1PE, 2PEs, 3PEs, and 4PEs per node is compared to that expected from the performance model in Figure 1.4. Note that only the difference between the measured performance and that expected by the model is shown in Figure 1.4. It can be seen that the measured data is very close to that expected when using 1PE, 2PEs, and 3PEs per node. However there is a large deviation when using 4PEs per node indicating a performance problem.

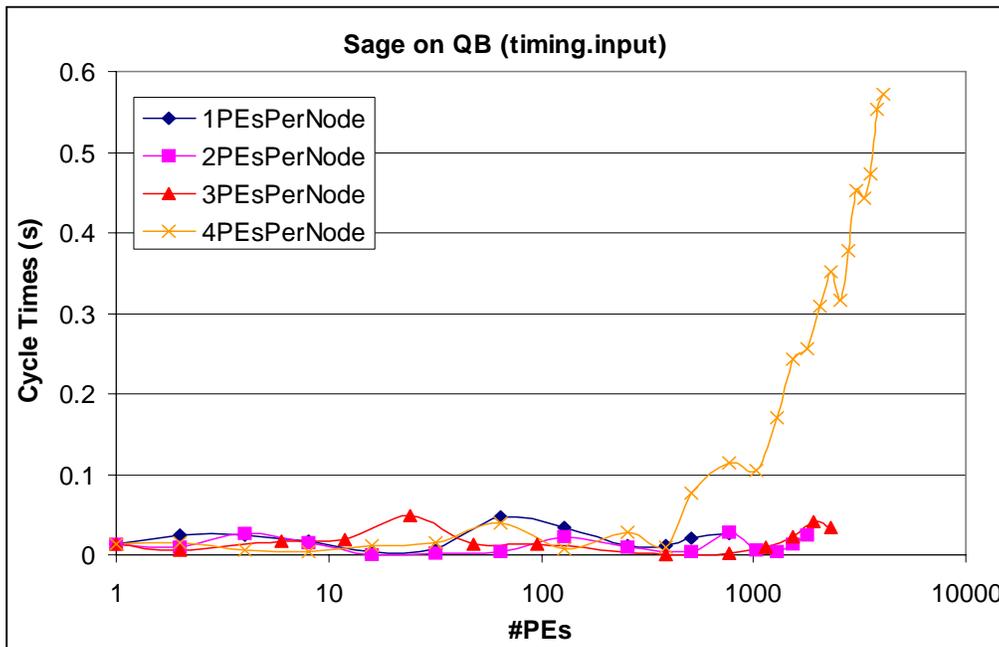


Figure 1.4 – Difference between measurement and model for 1PE, 2PEs, 3PEs, and 4PEs per node.

It is also interesting to note the effective performance that can be achieved when using less than the maximum number of PEs per node on SAGE. This is illustrated in Figure 1.5. When using more than approximately 1024 PEs a higher performance can be achieved using 3PEs/node than 4PEs/node. The exact cross-over point is dependant upon the exact configuration of the problem being processed by SAGE.

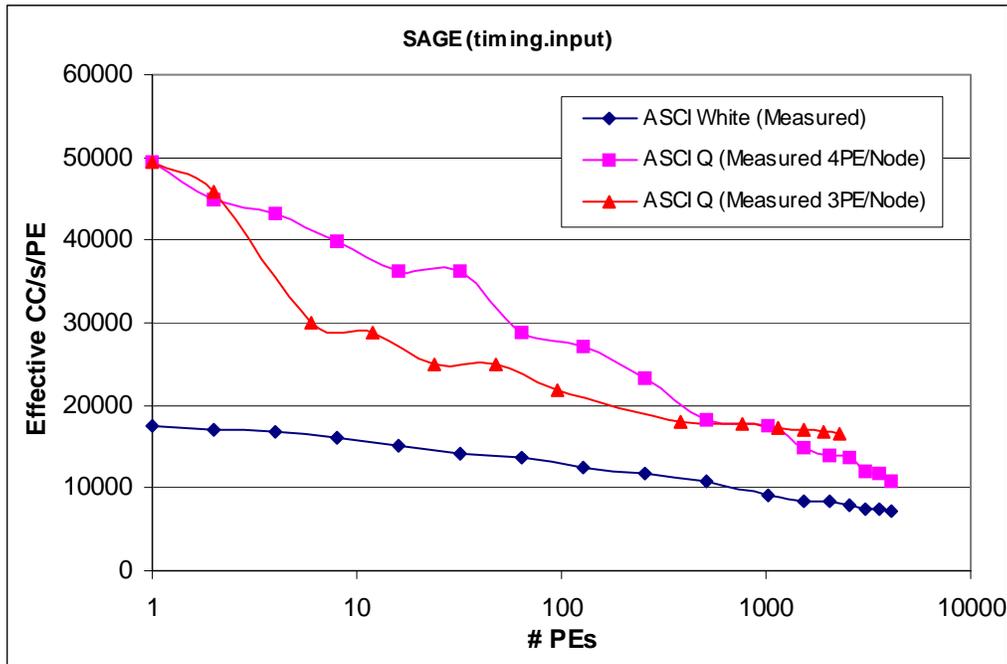


Figure 1.5 – Relative performance on SAGE when using 3PEs/node and 4PE/node

1.4 Time breakdown (1-rail scaling)

In order to understand why the performance of SAGE is not as expected, the time spent in the different communication aspects of SAGE were recorded in a scaling study using one rail. The communication components are separated into `token_get` and `token_put` (for the data gathers and scatters between spatially neighboring processors), and `token_allreduce`, `token_bcast`, and `token_reduction` (collective operations). The time taken in each of these components is shown in Figure 1.6.

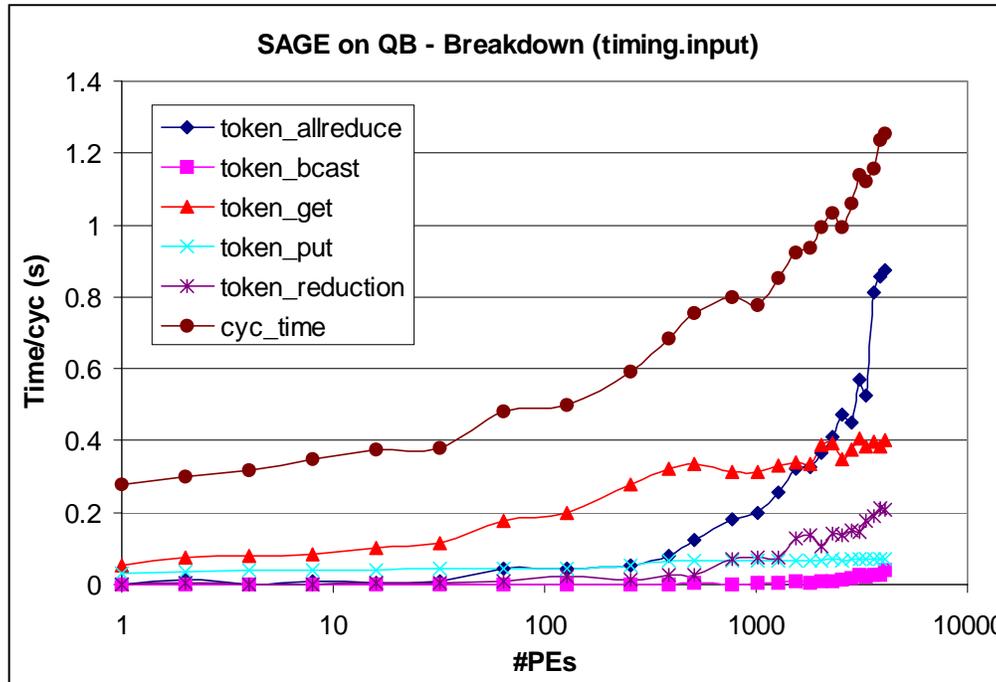


Figure 1.6 – Component times in SAGE scaling (timing.input)

The scaling of the gather-scatter operations (`token_put` and `token_get`) are as expected. However the scaling of the collective operations, in particular the `token_allreduce`, are much worse than expected when using more than 256 PEs. This can be clearly seen when considering the `cycle_time` in comparison to the model expectations in Figure 1.1. **The difference between the model and the measurements can be attributed to the `token_allreduce` time** in Figure 1.6.

1.5 Performance Variability of SAGE

In a further performance study, SAGE was executed for 1000 cycles using `timing.input` on 3584 PEs. The time taken for each cycle was recorded along with the time components as used in Figure 1.6. Figure 1.7 shows the cycle time for all cycles along with the performance model estimate. The distribution of the cycle time is shown in a histogram in Figure 1.8.

The cycle time for SAGE when using `timing.input` should be constant as all cycles effectively perform the same processing. However, as can be seen in Figure 1.7 there is considerable variability in the performance from cycle to cycle which can be attributed to the `allreduce` operations.

The histogram shown in Figure 1.8 indicates that the best cycle time observed over all the cycles is ~0.75s, which is very close to the performance model prediction of 0.68s. This indicates that in a very small percentage of cycles the performance obtained is as expected, but in the majority of cycles the performance is worse, sometimes by more than a factor of 3!

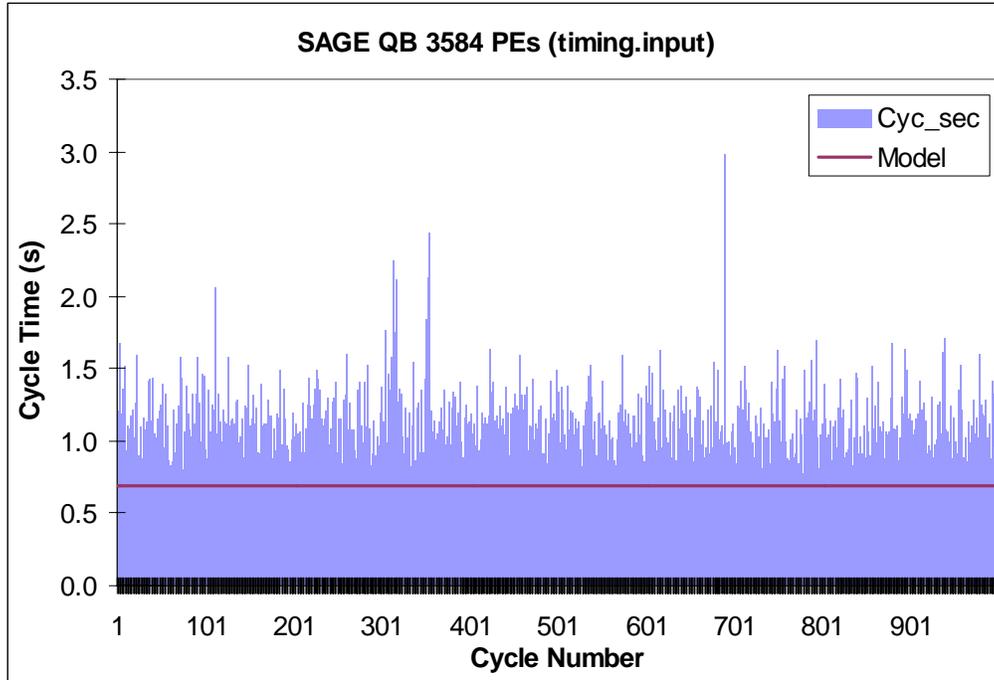


Figure 1.7 – Cycle time for 1000 cycles of SAGE (timing.input)

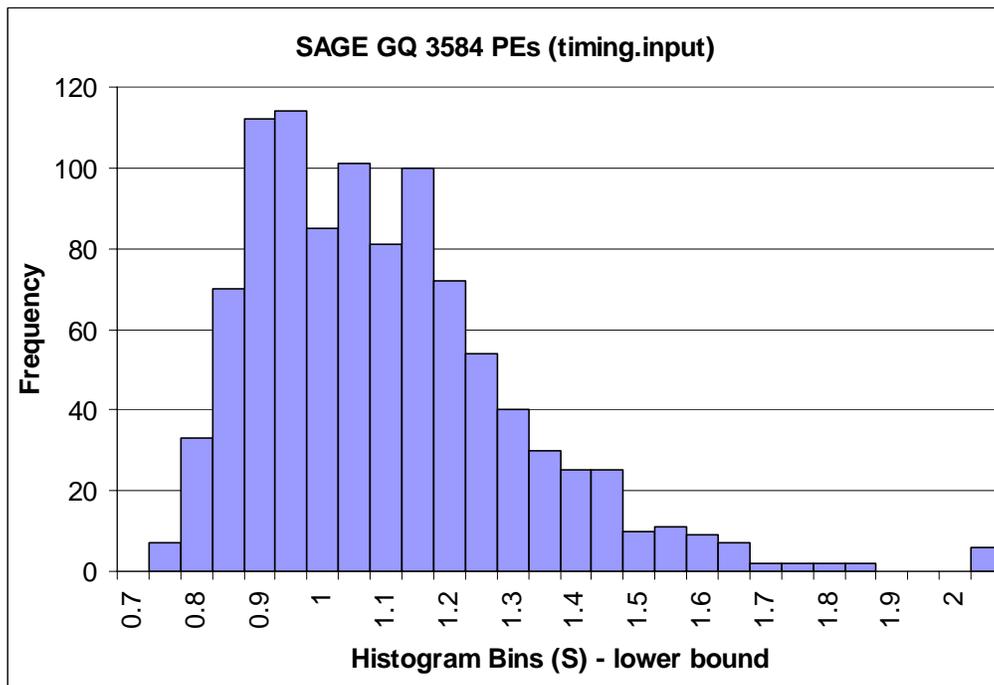


Figure 1.8 – Histogram of the cycle times for 1000 cycles of SAGE (timing.input)

1.6 Summary of SAGE performance

The performance of SAGE is less than expected. The cycle time is above expectation at 1024 PEs and above (Figure 1.1) when using 4 PEs per node. The performance is 14% worse than expected at 1024 PEs and 70% worse than expected at 4096 PEs.

When using a lower number of PEs per node the performance is as expected (Figure 1.4). There is no performance degradation on larger PE counts.

The degradation is not constant across cycles but varies significantly (Figure 1.7). This seems to correspond to performance variability reported elsewhere.

The degradation in performance and the performance variability most likely result from the collective operations (specifically `allreduce`).

In the following sections, the performance degradation will be analyzed in a comprehensive manner, quantified and solutions for improvements presented.

2. Identification of Performance Factors

In order to identify why application performance such as that observed on SAGE is 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. Since it appeared that SAGE was adversely affected by the performance of the `allreduce` collective operation several attempts were made to improve the performance of collectives on the Quadrics network.

2.1 Optimizing the Allreduce

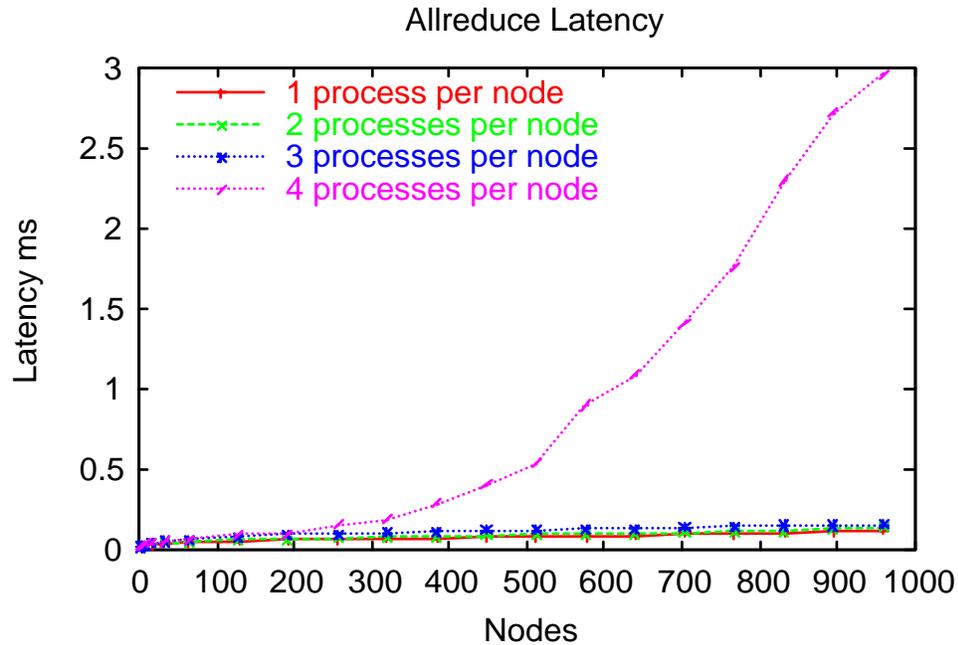


Figure 2.1 `allreduce` latency, varying the number of nodes and processes per node

Figure 2.1 shows the performance of the `allreduce` when executed on an increasing number of nodes. The graph clearly shows that the problem arises when we use all four processors per node. With up to three processes the `allreduce` is fully scalable and takes, on average, less than 300 μ s. With four processors the latency surges to more than 3ms on the full QB machine.

We made several attempts to optimize the `allreduce` in the four-processes case, and we were able to substantially reduce the worst case. In order to do that 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 μ s) and then block, we were able to improve the latency by a factor of 7.

Extensive testing was made on the modified collectives but resulted in only a marginal improvement in performance. **So we were not able to link the performance variability problems to glitches in either the MPI implementation or the network.** Having thus ruled out

MPI and the network as sources of performance loss we now proceed to scrutinize the nodes themselves.

2.2 Analyzing the Computational Noise

A further test was designed to examine the computational performance of each processor within a node. The goal of this test was to narrow down the search space and to come up with a simple benchmark to expose the variability problems. The benchmark works as follows. Each node performs a sequence of synthetic computations, each one carefully calibrated at the granularity of 1ms, for a total run time of 1000 seconds. Using a small granularity is important because both SAGE and Sweep3D may display such granularity. During this purely computational phase, there is no message exchange or I/O and the benchmark is run on all 4096 processors of QB. As shown in Figure 2.2 we take two types of measurements. In the coarse-grained experiment (Figure 2.2(a)) we measure the entire run time of each process, which in the ideal case should be 1000 seconds. In the fine-grained experiment (Figure 2.2(b)) we measure the run time of each single chunk of computation, which should always be 1ms in a noiseless machine. N.B.: On 2DEC2002 this test initially identified a node containing processors running at 1GHz, i.e., a node that did not successfully get upgraded to 1.25GHz!

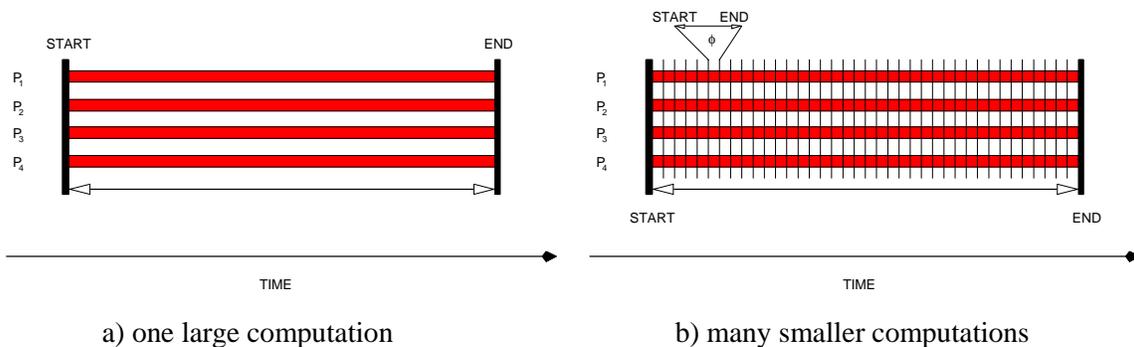


Figure 2.2 Measurement of the time taken to perform a computation task on each processor.

The total normalized run time for the coarse grained measurement is shown in Figure 2.3 for all of the processors in the system. Due to interference from non-application activities within each node, the processing time can be longer and can vary from process to process. The aggregate overhead experienced by each process is low, with a maximum delay of 2.5% (which amount to a slowdown of 25 seconds over a total runtime of 1000 seconds).

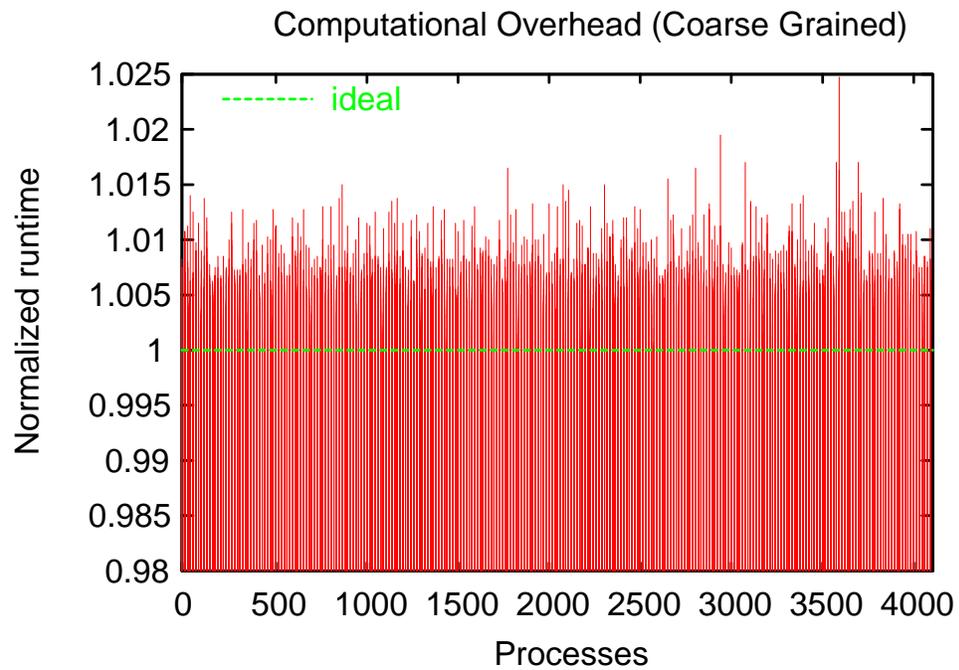


Figure 2.3 – Average time to process a 1ms computation task on each processor on QB

A particular aspect of these results is that they are analyzed on a per-process basis. This does not show the true nature of the performance effects that are occurring. By analyzing the data on a per node basis we can gather more insight into the performance variability problem. Figure 2.4 shows the results of the fine grained experiment. In this experiment we run 1 million iterations, each with a granularity of 1ms. As in the previous experiment, we do not perform any communication or I/O. At the end of each iteration we measure the actual run time, and for each iteration that takes more than twice the expected run time, we sum the unexpected overhead, expressed as the actual run time minus the threshold, for each node. Looking at the graph we can see that the noise has a regular pattern. Every partition of 32 nodes contains some nodes that are consistently noisier than others.

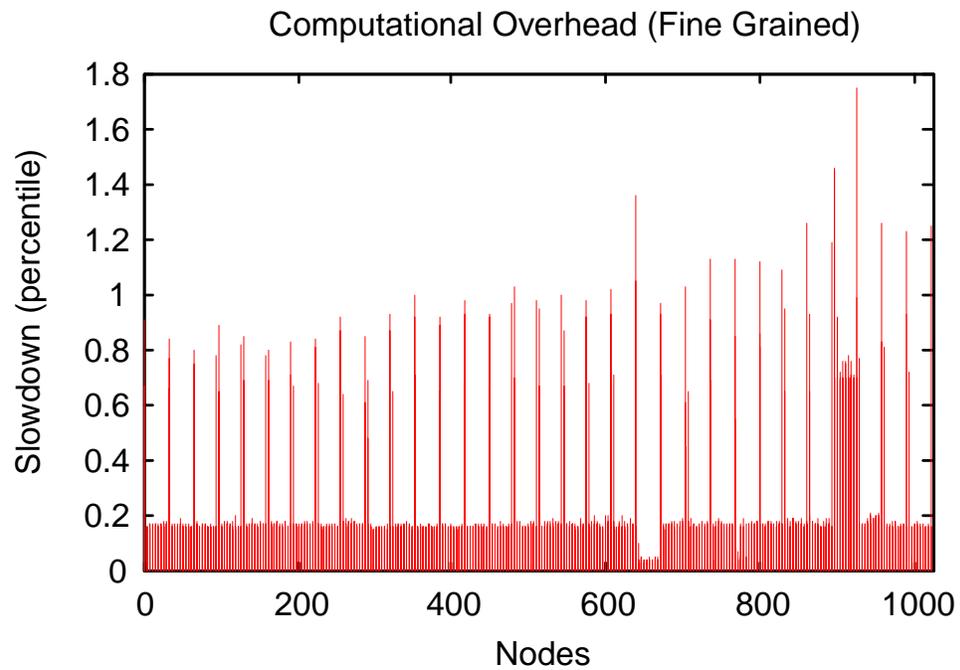


Figure 2.4 – Variation in computation time on a per-node basis.

Figure 2.5 expands the previous graph in order to pursue more detail on one of the 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 every 32 nodes. Therefore, in the rest of this section, “0” means “0+32k”, “1” means “1+32k”, and “31” means “31+32k”, where k is a nonnegative integer.

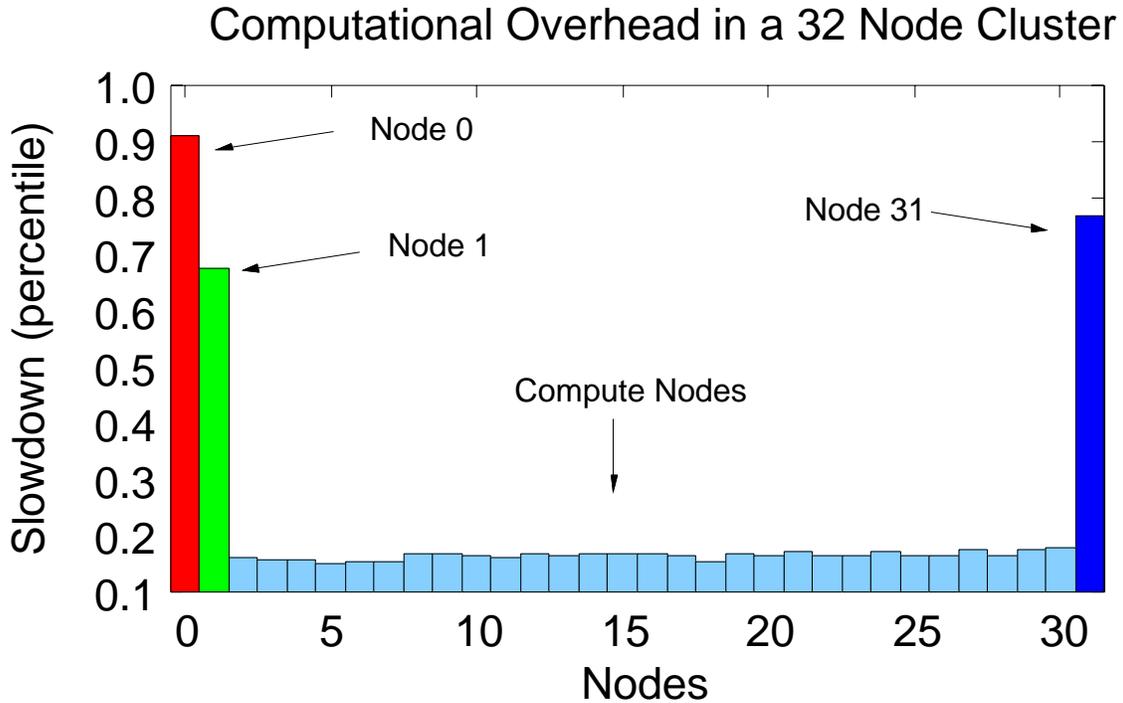
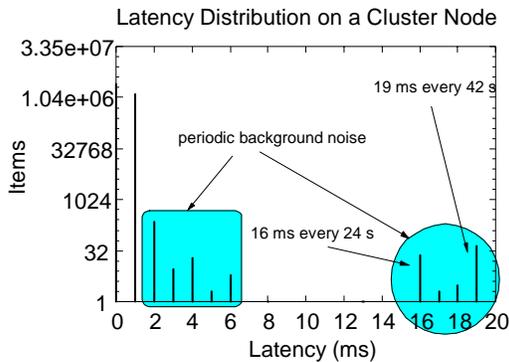


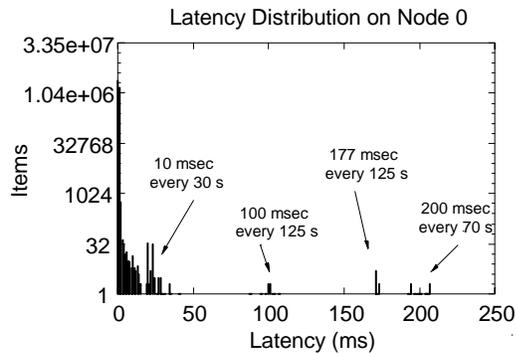
Figure 2.5– Examination of the overhead per node within each 32-node partition.

In order to understand the nature of this noise, we plot the latency distribution for four classes of nodes: the standard cluster node, node 0, 1 and 31. In Figure 2.6 we can see that the noise on each node has a well defined pattern, with classes of events that happen regularly with well defined frequencies and durations.

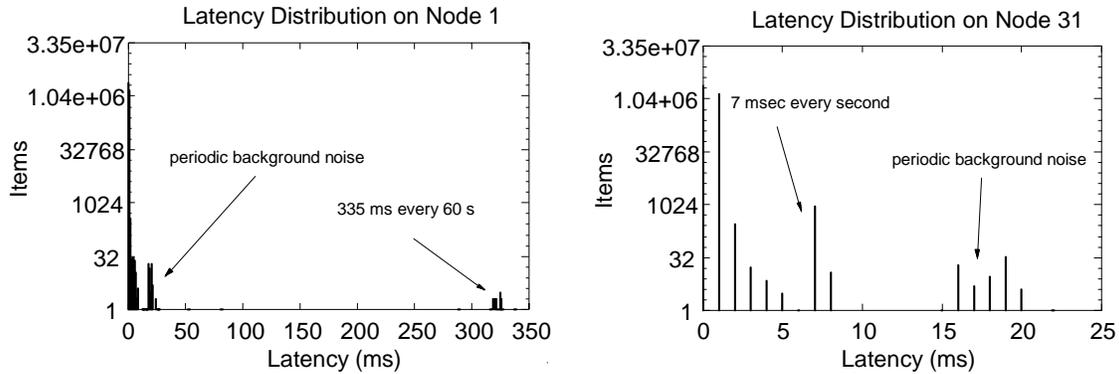
For example, on any cluster node we can identify two events that happen regularly every 24 and 42 seconds, and whose duration is, respectively, 16 and 19ms. This means that a slice of computation that should take 1ms occasionally takes 16 or 19ms. The process that experience this type of interruption will freeze for the corresponding amount of time. Intuitively, these events can be traced back to some regular system activity as daemons or the kernel itself.



a) on a 'normal' cluster node



b) on Node 0 in a cluster



c) on Node 1 in a cluster

d) on Node 31 in a cluster

cluster

Figure 2.6 – Identification of the events that cause delay on the different types of nodes.

Node 0 displays four different types of activities, all happening at regular intervals, with a duration that can be up to 200 ms. Node 1 experiences a few heavyweight interrupts, every 60 seconds, that freeze the process for about 335 ms (two orders of magnitude larger than the cycle time of many ASCII codes).

On node 31 we can identify another pattern of intrusion, with frequent interrupts (every second) and a duration of 7ms. Table 2.1 summarizes the characteristics of the main components of the noise in the system.

Node ID (within a 32-node partition)	Event Period (seconds)	Event Duration (milli-seconds)
All	24	16
All	42	1.9
0	30	10
0	125	100
0	125	177
0	70	200
1	60	335
31	1	7

Table 2.1 – Summary of Identified Events – periods and durations across nodes within each partition.

3. Effect on System Performance

Figure 3.1 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 job. Notice that even though any given process in Figure 3.1 is delayed only once, the collective-communication operation (represented by the black lines) is delayed in every iteration. When we run a job 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 4096 experiences a delay of 100 ms, on a job that barrier synchronizes every 1ms, then the whole application will run 100 times slower.

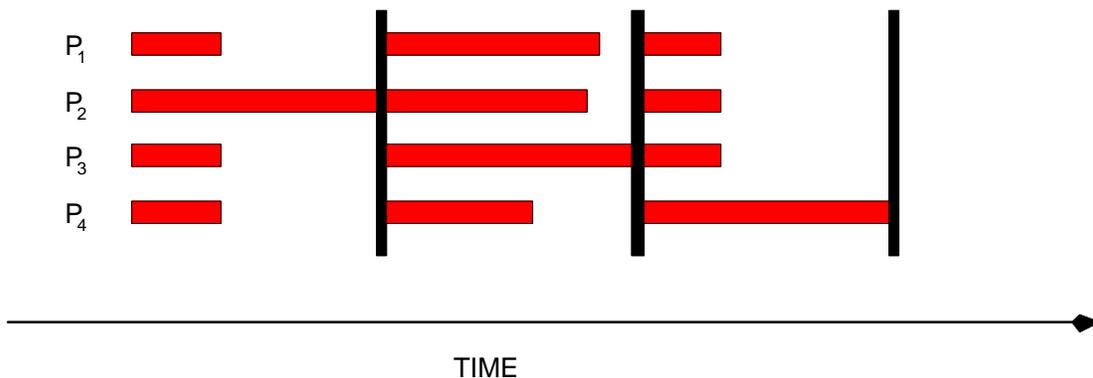


Figure 3.1: Example of how a slow process delays the entire application

Figure 3.2 shows the performance of the `allreduce` and the barrier in a synthetic parallel benchmark that computes for either 0, 1, or 5ms and performs an `allreduce` or a barrier at the end of each compute step. In an ideal, scalable, system we should not see any difference in the run time of the collective communication. **What we see is that the completion time increases linearly with the number of nodes and with the computational granularity. This is due to skew caused by noise within the node, not by any problems in either the network or the communication library.** Larger grain sizes induce longer latencies because they increase the likelihood that background load will strike during at least one process's compute time. The graph below also shows that both `allreduce` and `barrier` exhibit similar performance. Given that the barrier is executed using the hardware broadcast, whose execution is almost instantaneous (a few μ s), the only reason for this performance degradation is the skew accumulated within the processing nodes.

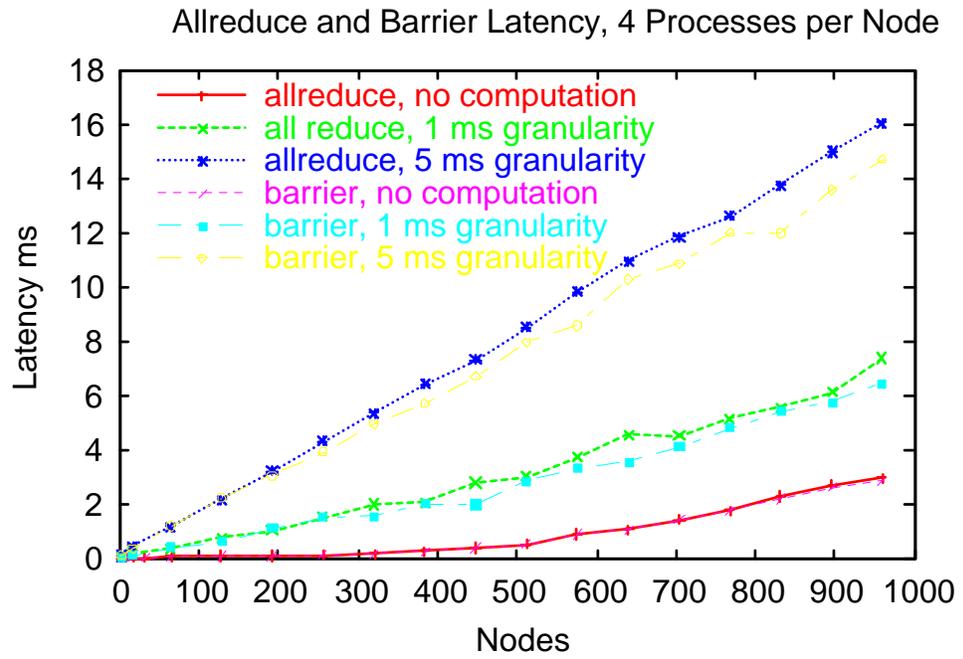


Figure 3.2: allreduce and barrier latency with varying amounts of intervening computation

4. Modeling System Events

From the events we identified in Section 2, we have developed a discrete-event simulator that takes into account all the classes of events identified in the previous section and provides a realistic lower bound on the execution time. We validated the simulator for the measured events, and we can see from Figure 4.1 below 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—partitions in the second half of the cluster.

Using the simulator we can predict the performance gain that can be obtained by selectively removing the sources of the noise. For example, 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 noise in the system dramatically reduces when we eliminate the background noise on the compute nodes. Although Table 2.1 showed that the background jobs running on all nodes tends to have an order of magnitude shorter duration than those running on only the cluster-manager or quorum nodes, the short period and sheer number of nodes running those background jobs cover the first-order effect of the observed performance loss.

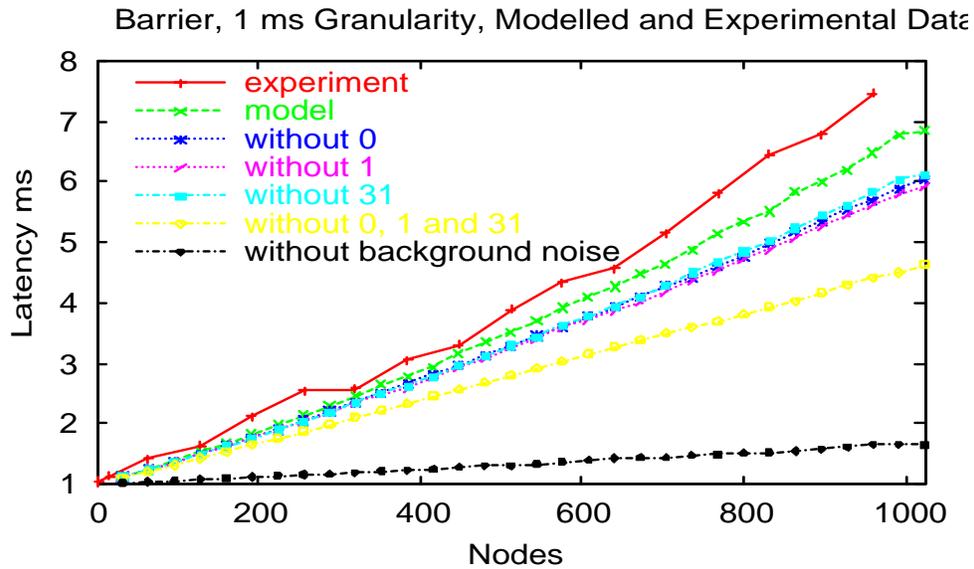


Figure 4.1. Simulated vs. experimental data with progressive exclusion of various sources of noise in the system.

The table below shows the actual overhead incurred by the different cases for two computation granularities of 1ms and 5ms. The reduction in percent indicates the effective performance improvement that would be possible if the events on the indicated nodes were removed. It can be seen that the reduction in time if the events were removed from only a single type node within each partition would be between 10.5 and 17.1%, whereas removing all the background events would cause a reduction in time of between 77.6% and 88.9%.

Events Modeled	1ms Computation			5ms Computation		
	Latency (ms)	Overhead (ms)	Reduction (%)	Latency (ms)	Overhead (ms)	Reduction (%)
All	6.84	5.84	-	18.76	13.76	-
Without Node 0	6.05	5.05	13.5	16.82	11.82	14.2
Without Node 1	5.93	4.93	15.6	16.40	11.40	17.1
Without Node 31	6.12	5.12	12.4	17.32	12.32	10.5
Without Nodes 0, 1 & 31	4.63	3.63	37.8	13.18	8.18	40.5
Without all background events	1.65	0.65	88.9	8.09	3.09	77.6

Table 4.1 – Performance summary of `allreduce` on 1024 nodes.

5. Possible Application Performance

From the insight into the performance of the events in Section 4, we have used this to indicate what could be the improvement on SAGE. By assuming that the increased cycle time measured in over that of the model expectation (as shown in Figure 1.1) we are able to provide an estimate of what the performance of SAGE may be if the events identified and characterized in Section 4 could be removed.

Figure 5.1. shows the performance of SAGE, using the timing.input deck. The upper curve is the current measured performance and the lower curve is the ideal expected performance from the CCS-3 performance model. It can be seen that the curves in between correspond to removing the events from Node 0, Node 1, Node 31, Node 0 & 1 & 31, and all background events. The improvements in performance correspond to the improvement in performance simulated as detailed in Section 4 and listed in Table 4.1. It should be noted that Figure 5.1 shows the maximum possible performance improvement that could be obtained.

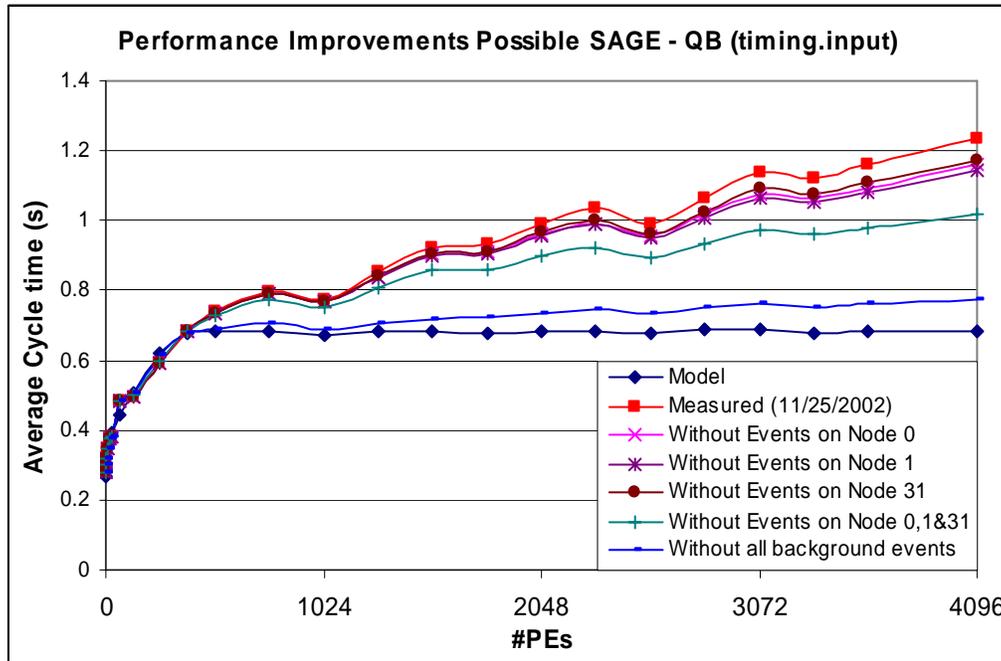


Figure 5.1 – Possible SAGE performance improvement.

A summary of the performance improvements are shown in Table 5.1. for the case of using 1024 nodes (4096 PEs). The performance is shown as a % increase over that of the ideal case provided by the CCS-3 model. The current measurements are 76% greater (run-time is 76% longer) than the ideal case. If all the background events could be removed then the performance would only be 11% greater than the ideal.

	Average Cycle time (s)	% Greater than Ideal Model
Current Measurement	1.24	76%
Without Node 0	1.16	65%
Without Node 1	1.15	64%
Without Node 31	1.17	67%
Without Nodes 0, 1 & 31	1.02	46%
Without all background events	0.78	11%
Model (Ideal)	0.68	-

Table 5.1 – Summary of SAGE performance if background events could be removed (1024 nodes)

A summary of the performance changes that will result from removing all the background noise, configuring out nodes and using only three processors per node is shown in Table 5.2. Also shown in Table 5.2 is the effect of using only 3PEs/node (which results from measurements). A positive % figure indicates an increase in performance, whereas a negative figure indicates a decrease. All these figures are based on using the timing.input deck on SAGE.

It can clearly be seen that the best situation is to use 3PEs/node when using 512nodes or above. The effect of configuring out node 0, or node 1, or node 31, or all three together, results in only a marginal change in performance.

Configuration	256Nodes	512Nodes	768Nodes	1024Nodes
3PEs/node	-1%	+23%	+39%	+47%
Without Node 0	-2%	0%	+3%	+3%
Without Node 1	-2%	0%	+3%	+5%
Without Node 31	-2%	0%	+1%	+2%
Without Nodes 0,1,&31	-6%	+5%	+10%	+15%
4PEs/node (without all background events)	+13%	+35%	+49%	+60%

Table 5.2 – Summary of SAGE performance on configuring out nodes per partition, or using 3PEs/node.

6. Conclusions and Recommendations

We have analyzed the performance of ASCI QB and its variability. We have identified the root causes of the performance degradation and variability: system activity on the nodes, and not network traffic delays. The analysis is qualitative and quantitative, the first of this kind that we are aware of anywhere. The sources of performance degradation and variability are identified, quantified and plugged back into the performance model for SAGE. We show that these performance degradation sources account for more than 90% of the difference between the measured data and the modeled (expected) data for the performance of SAGE.

In this way, having gained the confidence that the analysis is comprehensive (i.e., we took into account all sources of performance degradation and variability) we use an accurate discrete event simulator developed for this work to assess the effect of various sources of delays considered on the overall performance of the code. We found out that in fact, counter intuitively, most of the degradation is due to the compound effect of the small delays in the compute nodes, and not to the big delays induced by the I/O cluster nodes.

At a strictly practical level the table below summarize the sources of performance degradation and instability and suggests means to improve or outright fix the problems.

Node	Problem	Possible Action
0	Cluster Management & I/O	1) Configure out Node 0 when system is used for multiple jobs (Capacity mode) 2) No action when system used for a single job that performs bulk-synchronous I/O
1	Cluster Management (quorum node)	Same actions as for Node 0
31	RMS data collection	1) Reduce frequency of collection
Other	O/S (tru64)	1) Reduce the length of the O/S scheduling quantum (for example from 10ms down to 1ms). 2) Reduce the activity of kernel threads and other demons 3) Co-scheduling of system activities – by synchronizing system activities across all nodes, the effect of system intrusions can be minimized.
	RMS data monitoring	1) Reduce frequency of monitoring 2) Reduce number of different events monitored – especially those with high latency (e.g. those that access CFS).

Nearly all of these suggested actions have been implemented and tested on one of the CCS-3 Linux cluster test-beds. It has been demonstrated that almost all of the overheads on the compute nodes can be removed and hence resulting in near optimal application performance.

One further suggestion is to provide a dual-boot facility on QB enabling either Linux or Tru64 to be used as the O/S. This may not be entirely straightforward but the potential performance benefit makes this a worthwhile enterprise and a path that should be fully explored.

This work is a qualitative and quantitative example of the effect of the “weakest link in the chain” effect (described by the Amdahl’s Law) on extreme-scale parallel machines. Small, unsynchronized system activity of many kinds gets compounded leading to effects sometime orders of magnitude bigger on large scale system.

Undoubtedly, all possible sources of system activity that could be reduced should be reduced. But

most importantly, **on future generation system software for large scale architectures, system activity on the nodes should be synchronized, for example by a heartbeat in the network. In this way, the effect of the small delays is overlapped and not additive.**

Acknowledgements

We acknowledge the help from Joe Kleczka, Amos Lovato, Malcom Lundin, David Addison , and the active involvement of Manuel Vigil and Ray Miller. Thanks to the Quadrics folks in Bristol for their prompt help.

7. Appendix: Methodology for Quadrics network debugging

Fabrizio Petrini, David Addison

We strongly advise that a procedure should be put in place to monitor the embedded controllers regularly or else the network error database will be an incomplete record of the network status.

Also, the network error database should also be queried regularly, perhaps several times a day to be more proactive on network issues.

Methodology for Quadrics network debugging

In order to expose the problems, we found these two aggressive tests useful.

1) complement traffic

```
prun -N 1024 -p full -Rrailmask=[1-2] tping -p 1023 -fdping 0 4m
```

this test uses pairs of nodes to flood the network with messages and keeps all the links active in both directions at the same time. Network problems can usually be suspected when some of the pairs report much lower bandwidths (e.g 30-50MB/s instead of 200MB/s)

The command line works on 1024 nodes, but it is possible to use the same test on a smaller number of nodes, as long this number is a power of two. In the general case, just replace 1024 with NODES

```
-N NODES ..... -p NODES-1 with NODES power of 2
```

2) global exchange traffic

```
prun -N 1024 -p full -Rrailmask=[1-2] tping -fgex -n100 64k
```

This test also floods the network and stresses the Elans. It is possible to run the global exchange on any number of nodes.

The complement traffic deterministically exposed a set of problems on the second rail, while the global exchange exposes another set of problems on the first rail.

The errors that are logged by the RMS database can be viewed by running

```
/usr/opt/rms/diag/neterror -H hours
```

where number is the number of hours.

3) In order to troubleshoot the cable/switch failures, it is possible to use the following script

```
/users/fabrizio/CABLETEST/QB-CABLETEST
```

We wrote this script today for the purpose of debugging the network, and handed it over to Malcom Landon of Raytheon and Joe Klescka. *We recommend using this script periodically for debugging purposes.*

The script runs cable tests every 64 nodes, covering both rails the whole QB. It can also be adapted to run the same tests on QA, by simply changing

```
QB=qb`expr $NODE + 1024`      with
```

```
QA=qa`expr $NODE`
```

and replacing all the occurrences of qb with qa.

Surprisingly, all the "standard" benchmarks completed successfully, without displaying any substantial functional or performance problems in the network.