Understanding the Causes of Performance Variability in HPC Workloads

David Skinner and William Kramer
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

Abstract.

While most workload characterization focuses on application and architecture performance, the variability in performance also has wide ranging impacts on the users and managers of large scale computing resources. Performance variability, while secondary to absolute or optimal performance itself, can significantly detract from both the overall performance realized by parallel workloads and the suitability of a given architecture for a workload. In making choices about how to best match an HPC workload to an HPC architecture most examinations focus primarily on application performance, often in terms nominal or optimal performance. A practical concern which brackets the degree to which one can expect to see this performance in a multi-user production computing environment is the degree to which performance varies. Without an understanding of the performance variability exhibited by a computer for a given workload, in a practical sense, the effective performance that can be realized is still undetermined. In this work we examine both architectural and application causes of variability, quantify their impacts, and demonstrate performance gains realized by reducing variability.

Introduction and Motivation

Variability of parallel application performance has broad implications for how much useful work can be produced by a particular HPC system. Factors leading to changes in performance occur over multiple time scales and originate both from within applications and externally. As a result variability in runtime performance is as strongly tied to the workload and HPC architecture as it is to any specific application. This motivates the need for a better understanding of workload characteristics and the interplay between individual applications on a parallel computer. This work examines performance variations in parallel applications on time scales of years to microseconds, with a focus on understanding the causes of performance variability in multi-user production environments. While the focus is on IBM SP systems, comparisons are made across several architectures, interconnects and operating systems. Where possible, specific causes for the variation measured are identified. In so far as performance variability is the result of some sort of contention, decreasing variability nearly always equates with increased performance. An evaluation of actions taken to achieve both of these goals is presented.

The characterization of HPC workloads and their representative applications' performance has been the topic of many and on going studies. Performance benchmark timings of parallel HPC applications may be taken either on dedicated hardware or in the context of day to day use. For a large scale computing center such as NERSC the latter means the timings may be influenced by hundreds of other parallel jobs, system administrative activity, and contention for shared resources. All of these may to lead to changes in execution times. It is this more complicated production context that is arguably more important in setting expectations about how long a scientific calculation will take to complete.

Variability in runtimes leads to many negative impacts all of which make HPC systems less productive. The first impact is less work overall done by the system. Runtime variability is inherently bad for performance since variations in runtime proceed upward from some best case runtime, i.e., variation is never toward better than optimal performance. The longer a task takes, the more time it takes to get usable results for analysis. Since some applications have a strict order of processing steps (i.e. in climate studies year 1 has to be simulated before year 2 can start), we can not directly overcome this slowdown via increased parallelism, and longer time steps mean slower progress.

Variability decreases the efficiency of HPC parallel computers in that cycles are lost to both job failure and complex job scheduling. Jobs fail through incorrect estimation of the batch queue requirements. System scheduling becomes less effective because users must be overly conservative in requesting batch time. Most scheduling software relies on user-provided run estimates, or times assigned by default values, to schedule work. When a cautious user over estimates run time, the job scheduler operates on poor information and can cause inefficient scheduling selections on system. Any job that overruns the requested batch queue time will suffer a loss of productive time elapsed since the last checkpoint, or in the case of code which do not check point, the entire run may be lost.

For the above reasons variability in runtime is a quantity that we seek to minimize insofar as it does not impact the mean application performance. In very general terms we may summarize the goal of this study algebraically by first identifying a workload, W, and expressing the realized performance, P, as the optimal or nominal performance P_0 less the performance lost due to variability, V.

$$P(W) = P_0(W) - cV(W,\ldots)$$

This is the performance expression which accounts for the performance loss due to variability. It is written here for a fixed compute resource. Note that without the context of a particular workload such an expression of performance is not possible. I.e., we identify performance as having meaning only within the context of both workload and computer. The overall goal of the present work is to identify the factors "…" which influence V and if possible the impact extent and parallel scaling of these factors.

In order to do so, it is important to be able to detect, measure, and address the causes of performance variability. Keeping both goals in mind we will concentrate on how the shape of the distribution of runtimes for a task or application is influenced by a variety of factors and events occurring on the system.

Understanding the parallel scaling of factors leading to performance variation is a chief concern of those who use and maintain large scale parallel computers. Large scale HPC resources are often built from thousands of smaller systems. Since the majority testing and performance analysis is done on the test systems much smaller than production machines it's common to encounter variability induced performance loss that goes unseen on smaller systems. As we will demonstrate below the performance impact of variability at high concurrency can be quite large, becoming the dominant impediment to parallel scaling in some cases.

<u>Variability over long time scales</u>

The majority of the data reported here comes from "Seaborg", the 10 Teraflop/s IBM SP at NERSC. This machine has 380 16 way POWER3 nodes connected with a colony switch. Some of the data comes from tests conducted in order to answer specific questions other data is collected as part of routine procedures. This section shows how system events that impact application performance change throughout the lifetime of the machine.

For a sufficiently large parallel compute resource operating in a diverse production environment it is difficult to unambiguously identity a precise normative state in terms of level of contention or overall functioning. Instead it is useful to utilize a compact set of short running benchmarks that serve as a barometer as to system functioning. The timescales on which influences on performance act span many orders of magnitude, with some changes coming only on the timescale of months or years. Changes in such a course grained barometer provide evidence of how the system and/or its workload is evolving over very long intervals of days to years.

In order to examine factors that change throughout a parallel machine over time the concept of node index is useful. We will define an arbitrary index whose values are important only in that they span the entire machine, providing a machine coordinate through which we examine properties across the machine.

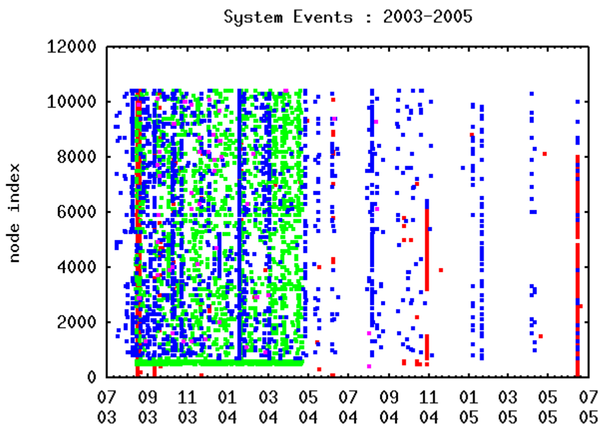

Fig 1. A chronology of filesystem and process events that could impact parallel applications on seaborg. Blue = unaccounted for unix process, Green = filesystem memory buffers low, Red and Purple = significant filesystem or node problem

As seen in Fig. 1, the space of system events that may impact parallel applications is not homogenous within the machine over long time scales. Events ranging from minor interruptions to outright node failures occur sporadically within this 6080 CPU parallel machine. While outright system failure is an exceptional case in which measuring performance is not directly meaningful, many system events do not cause failure, but rather introduce unexpected delays in some part of the parallel application. It’s important to note that these events may occur inhomogeneously both in time and within the machine itself. Certain events occurring regularly, e.g. persistently low filesystem buffers (dense green line near the bottom in Fig. 1) occur only on login nodes, where no parallel applications are run, but only sporadically on batch compute nodes. While these types of events do impact application performance, e.g. slowing filesystem access, they occur so infrequently that they are rarely considered in performance analysis studies.

Once identified, the causes of these events can be investigated in detail and when possible improvements are made to system software or configuration to eliminate them. This is evidenced in the case of the filesystem buffer under-runs (green events) and stray unix processes (blue events) in Fig. 1. In both cases system level changes were made to prevent or decrease the frequency of such events. In these cases performance improvements are the result of diligently monitoring and addressing variability at a system administrative level.

A similar chronology of performance variability at the application level is visible in the scaled wallclock times presented in Fig 2. Here the ratio of observed time to the average time over the course of more than three years shows wallclock times varying, in some cases, by a factor of two for the LU NBP benchmark and by just 20% for the less synchronizing EP NPB. These measurements were made with unchanging set of binaries (no recompilation was done). Many factors drive the fluctuations seen in Fig.2. Changes in operating systems and associated system software, internal interconnect loads, batch queue configurations, application libraries (e.g. MPI) are among the influences on these timings. Some of these changes can be clearly identified as coinciding with changes in the average performance or the variance therein.

- Jan 2002 average performance increase following system software upgrades

- April 2003 increase in variability after doubling the number of nodes on the switch.

- October 2003 decrease in variability and increase in performance coincides with steps taken to remove asymmetries in the machine (vide infra)

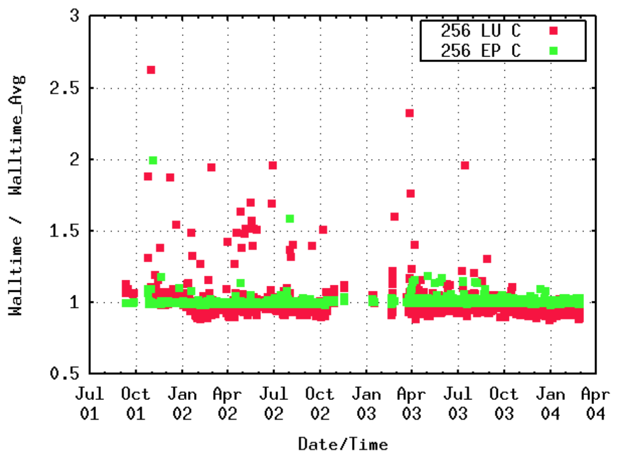

Fig 2. The ratio of observed wallclock time to the average time for EP and LU class C 256 way NAS NPB benchmarks. The average is computed over the entire span of the data.

The data shown in Fig. 2, while presenting a chronology of variability, do not accurately describe the underlying distribution of runtimes so much as simply demonstrating the outliers. The distributions are shown in Fig. 3 for a total of twelve thousand runs conducted over three years. The data marked "new" contains only data after October 23 2003 when changes, detailed below, were implemented.

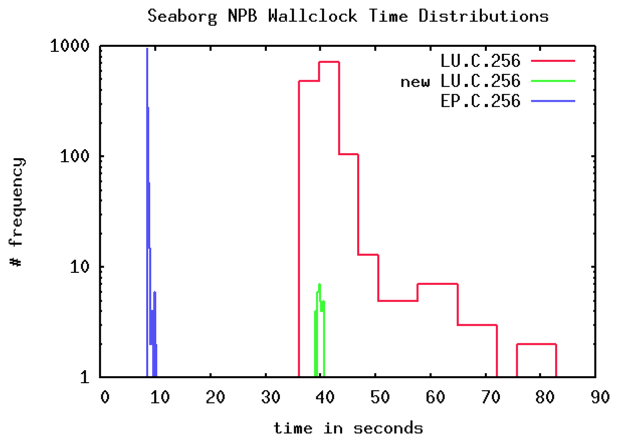

Fig 3. Wallclock timing distributions for EP (blue) and LU class C 256 way NAS NPB benchmarks. The LU data show performance variability prior to (red) and after (green) improvements in system configuration.

It is interesting to note that the distribution of EP timings, which span the same time frame as the LU timings, is relatively unchanged throughout. That performance variation is tied to application and workload characteristics is not unusual. When resource contention is involved it often gives some indication as to what resource is being contended for. Indeed the LU and EP benchmarks are quite different in that the former involves significant inter-task communication while the later involves very little.

In this section we have presented a high level view of performance variability over long time scales. In the sections that follow we present examinations into performance variability factors at finer time scales and using more tightly controlled tests. By looking at a fine grained time scale we are able to identify some of the causes that underlie the above course grained changes.

While we are principally concerned with the IBM SP architecture, where possible comparison across architectures is made. The next section in particular examines overall system architecture and runtime variability.

Overall System Architecture:

One of the fundamental contributors to variability is the hardware and software architecture. Almost all current large scale systems consist of a set of nodes connected by an internal interconnect, run by different software. Different architectures show different susceptibility to variability[i].

Four systems with different architectural features were examined:

- Cray T3E - The oldest system is the Cray T3E at the NERSC, placed into service in 1997 consisted of 696 CPUs (or PEs – Processing Elements), each with 256 MB of local memory. The PEs are connected by a network arranged in a 3-dimensional Torus with low latency and relatively high bandwidth but static routing. The processors are Alpha EV-57 running at 450 MHz.
- IBM SP - The next system is a 6,656 processor IBM-RS/6000-SP at NERSC called *Seaborg*. It is composed of a 380 compute nodes containing 16 Power 3+ processors connected to each other with a high-bandwidth, switching network known as the “Colony” switch in a Omega topology. A full instance of AIX runs on every node. Each node has two switch adapters.
- Compaq SC - The *Lemieux* Compaq SC system at the Pittsburgh Supercomputer Center (PSC) is composed of 750 Compaq Alphaserver ES45 nodes and a separate front end node. Each computational node contains four 1-GHz processors capable of two Flop/s per cycle and runs theTru64 Unix operating system. Intel - LBNL’s *Alvarez* commodity cluster of 85 two-way SMP Pentium III nodes connected with Myrinet 2000, another Fat Tree. The CPUs are xSeries 330, running at 866 MHz with 1GB SDRAM. Each node runs the Linux RedHat distribution.

System		LU	FT
Cray T3E	Number of Runs	119	118
	Mean Wallclock Run Time (sec)	305.2	106.5
	Standard Dev (sec)	47.8	12.1
	Coefficient of Variance based on wallclock tim	15.58%	11.33%
	Coefficient of Variance based on accounted tim	0.6%	0.93%
IBM SP	Number of Runs	165	210
	Mean Run Time (sec)	74.6	41.5
	Standard Dev (sec)	3.4	2.4
	Coefficient of Variance	4.58%	5.70%
Compaq SC	Number of Runs	359	371
	Mean Run Time (sec)	42.8	30.6
	Standard Dev (sec)	1.9	1.0
	Coefficient of Variance	4.53%	3.18%
Intel Cluste	Number of Runs	71	119
	Mean Run Time (sec)	408.7	90.7
	Standard Dev (sec)	10.7	1.0
	Coefficient of Variance	2.62%	1.07%

Table 1 shows the basic statistics for the test runs. Including some of the special tests discussed below, over 2,500 test runs were made.

The class C benchmarks were not run on the BlueGene/L system but a similar examination of the class B FT NPB was conducted and a comparison of the runtime distributions is shown in Fig 4. This architecture is also based on a microkernel and shows a tighter distribution of runtimes than the AIX based POWER3 system to which it is compared.

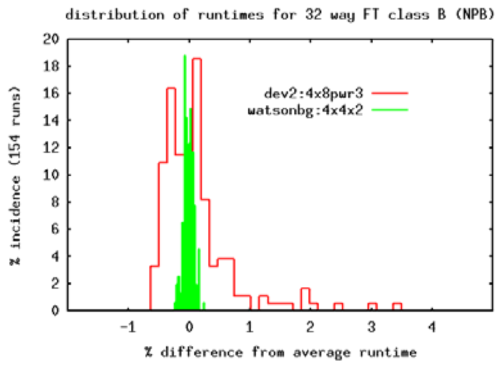

Fig. 4 Relative distributions of runtimes for the class B FT NPB compared between an AIX / IBM SP cluster (in red) and a microkernel based BlueGene Prototype (in green).

We now move from the high level architectural view of operating system, interconnect, and application to a detailed examination of specific causes of performance variability.

<u>Causes of Variability I: SMP Resource Contention</u>

In most large parallel systems, one task of one application runs on each CPU. This task requires not only the CPU as a resource in order to make application progress, but memory access, and will periodically require access to the communication fabric interfaces.

The impact of intra-SMP contention between tasks within a node for CPU and memory is shown in Fig 5. In this experiment the given number of synchronized tasks each fill a separate vector of the given size with a scalar value. This operation requires CPU and memory bandwidth for each task to make progress. The frequency and magnitude of variation depend on the concurrency and the data size. The data size determines essentially which resource might be contended for (L1 cache, L2 cache, main memory etc.) Concurrency controls the frequency of contention.

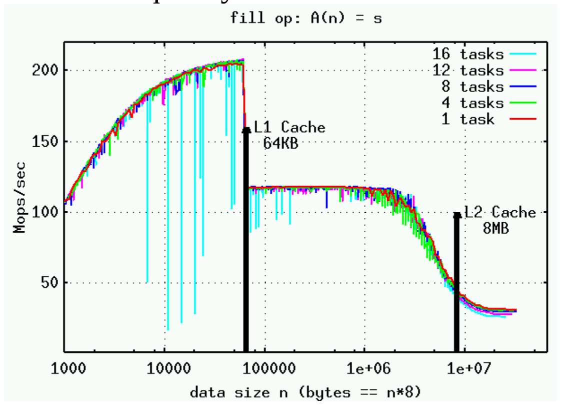

Fig 5. The impact of contention within an SMP node. The given number of synchronized tasks each fill a vector of a given size with a scalar value.

On a fully packed SMP executing such a fill operation requires all caches closest to the CPU. If a non application process is to be scheduled it must necessarily displace one or more of the application processes from a CPU and cache. The result is contention between application code and system activity. This is demonstrated in Fig 5 by the large variations in the blue lines for small data sizes.

This sort of contention occurring at the very smallest measurable timescales leads to processing scheduling and resource locking problems which impact codes which require synchronization between tasks. Work is underway to address CPU scheduling through co-scheduling in order to minimize the temporal cross section of the aggregate time spent outside of application code.

The impact of this low level contention on higher concurrency codes is realized in the subsequent analysis that examines such synchronization.

<u>Causes of Variability II: MPI</u>

Outside of CPU and memory applications contention may occur in communications between and within nodes through messaging.

Measuring the roundtrip transit time for small messages on seaborg reveals statistically distinct modes. The first mode is dominant accounting for 99.45% of all messages sent and received. The median transit time for this mode corresponds to the nominal latency of 11 microseconds. Analysis of the slower modes is increasingly complex, but the second most significant mode is reasoned to be a result of the 10 millisecond interval at which process scheduling takes place.

Tests were conducted on POWER3 and colony in full production mode as well as on a dedicated system running no workload other than the test at hand.

The slower messaging modes show strong overlap with from unix processes. In the case of shared memory the overall variability from the nominal latency occurs for a mixture of reasons, which may with further analysis be identified.

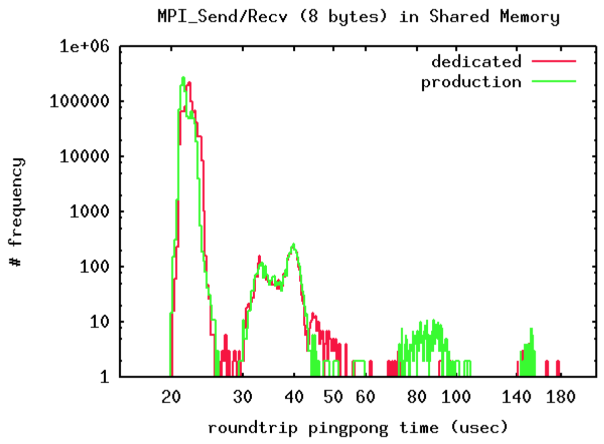

Fig. 6 Distribution of intra-node roundtrip MPI_Send/MPI_Recv times through shared memory in dedicated and production modes.

Conducting the same experiments for short MPI messages over the colony switch fabric shows a dominant mode with the expected 36 microsecond round trip latency. The manner in which variability in messaging times arise is however different from the single node shared memory case. Instead of distinct modes the dominant mode is shifted by 2.5 microseconds and a broad range of transit times in the range of 70 to 180 microseconds occurs.

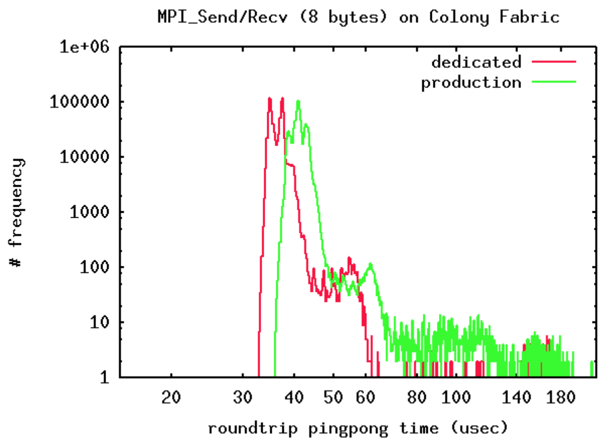

Fig. 7 Distribution of intra-node roundtrip MPI_Send/MPI_Recv times through the colony switch fabric in dedicated and production modes.

In a production environment 998 out of 1 million transits requiring more than 50 usec. The system in dedicated mode has 1/3 as many .Nearly same holds for colony MPI so contention is likely in this case a result of GPFS or other daemons requiring resources within the node.

In order to estimate the impact on applications of the minor modes observed in this study one must model the cumulative impact of infrequent slow messages on the MPI implemented in an application.

The simplest model of a fully synchronizing MPI Collective operation for N tasks would entail each task sending or receiving a number of message proportional to the concurrency. Any variability leading to a slowdown in point-to-point communication would be reflected in a collective slowdown that increases with concurrency.

It should be noted that the long time tails seen in the above distributions are similar to the features seen in distributions of application timings. There is a nominal runtime which varies downward very slightly but can drift upward (slower) to a significant greater degree.

Causes of Variability III: Kernel Process Scheduling

In their default mode of operation AIX and other Unix operating system kernels interrupt running code to schedule new processes for running in a round robin manner. For instance, in the AIX kernel there is an interrupt for each CPU at 10 millisecond intervals to schedule new work on each CPU. In AIX this interval is tunable via the schedtune utility. Given that many HPC applications have a lock step program flow which is highly synchronous, it is natural to consider if there is a benefit in process scheduling happening less frequently.

To test this question we adjusted the AIX kernels process scheduling interval and examine the resultant impact on colony fabric MPI latency. The second dominant mode in AIX/PWR3 colony MPI latency is seen to be directly related to this kernel interval. We conducted experiments on POWER3/AIX5.1 to determine the optimal value for the clock tick interval. The impact of lengthening the clock tick is to shift the messaging time to lower (better) values and decrease the magnitude of the second mode. A 4x clock tick setting leads to the number of message times greater than 40 microseconds decreasing from 0.2% to 0.06% and an average messaging time one microsecend less.

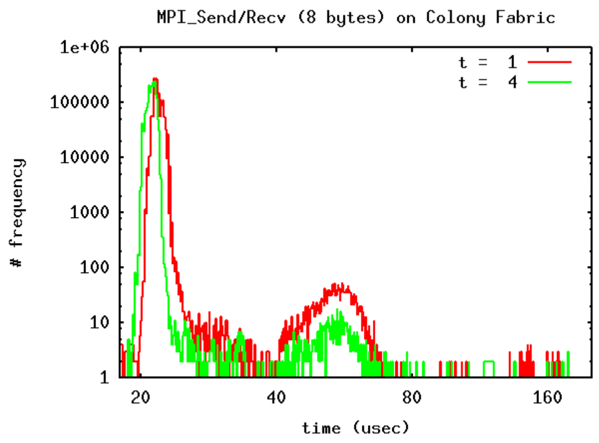

Figure 8. The central moment and second mode of the inter node MPI latency are shown to improve by making the AIX kernel's process scheduling interval less frequent.

An arguably more important benefit is the reduction in the number of very slow messages in the second mode of the distribution. These slow messages, while infrequent, have a compounded impact at high concurrency leading to significant scalability problems. Since MPI collectives are built from a number point to point messages which often scale logarithmically (MPI_Allreduce) or quadratically (MPI_Alltoall) with concurrency the impact of this kernel process scheduling

Causes of Variability IV: Cross application contention

One of the most complex manifestations of performance variability on large scale parallel computers involves the contention between applications. Diagnosis of cross application contention is difficult for both statistical reasons and in that it requires a system context as opposed to application context of the factors leading to performance degradation. For instance in Fig. 9 we show the MPI_Allreduce performance for a 1290 way application that optimally, in the sense of $P_0(W)$, should show constant timings per iteration for its reduction step. In practice, as shown in Fig. 9, we on occasion observe a sustained 2-3X increase in MPI_Allreduce timings that commence at times unrelated the flow of the program itself. Fig. 10 shows a different performance variation for the same application. The distinct, and in the case of Fig. 10 discrete, nature of these performance variations points to causes outside both the application itself and the normative state of the parallel computer. In order to determine the cause for this performance variation we must go outside the context of this specific application and examine the other workload coincident with the slowdown.

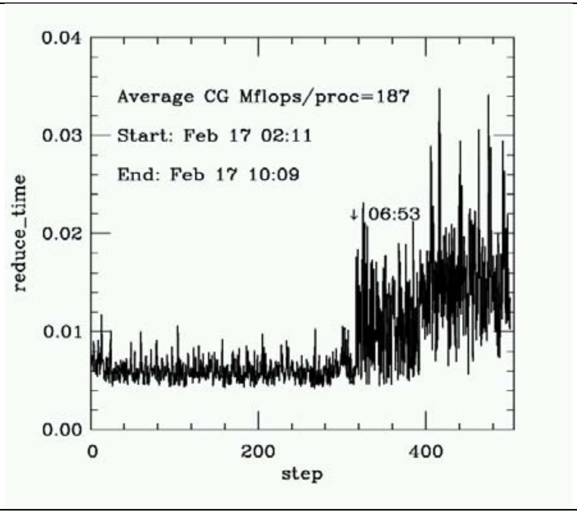	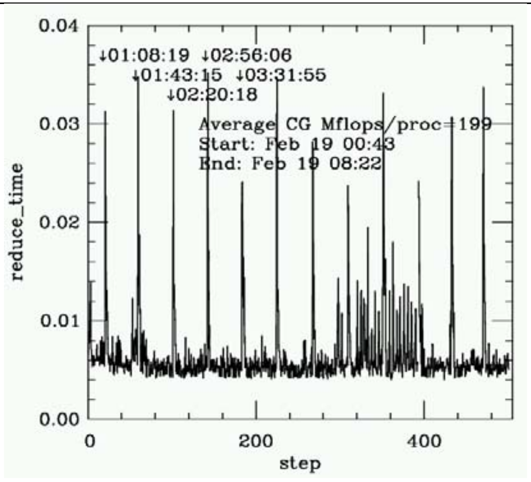
Fig. 9 Systematic change in application performance.	Fig. 10 Periodic change in application performance.

In order to make sense of the performance seen in Fig. 9 and 10 we look outside of the application itself to the other workload which coincides with the performance changes. Fig 11. shows that these performance changes coincide with GPFS read activity initiated from a different parallel application. In this case the contention leading to performance changes is for resources shared between applications, namely the parallel filesystem and interconnect.

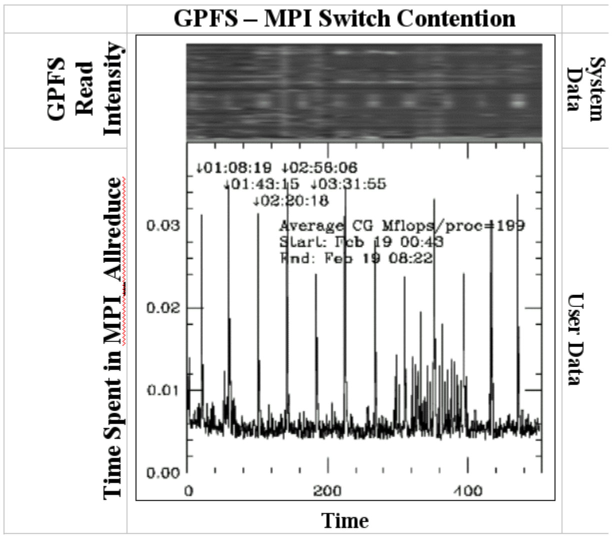

Figure 11. The coincidence of application performance changes in one application (lower) and the GPFS read activity in another application (upper). The read activity in the upper panel is represented as a machine index versus time. The nodes responsible for the read activity are from a different parallel job that than the one shown in the lower panel.

In the case of Fig 11. the most directly accessible means on lessening performance variability was to examine and improve the I/O strategy in the application that induced the slowdown in the application shown in Figures 9 and 10.

Causes of Variability V: System Activity

NERSC's IBM SP was doubled in size in early 2003. Due to previous experience with variability performance issues, great care was taken to assure all known causes of variability were eliminated. Some of these include

- All new hardware was identical to the existing hardware
- In order to assure the software configurations between the new and old nodes was identical, an exact image of all software was made and copied onto the new nodes.
- All system administrative procedures were identical.

In order to enable testing of the new nodes they were separated into a queue used for testing and an early user program. During this test period there were reports of the new nodes performing somewhat better – about 10% - than the old hardware. All reports involved parallel MPI codes. No serial performance differences were detected. At other times, the new and old sets of nodes showed very similar parallel performance. After the new nodes were integrated with the existing nodes comparisons of parallel jobs between old and new nodes became extremely difficult due to do IBM's LoadLeveler scheduling implementation but the difference in performance between new and old nodes continued to be a concern.

Old/new node performance differences continued. Periodic observations and testing provided inconsistent results and at times no asymmetry could be measured. The systems were audited and not cause was determined.

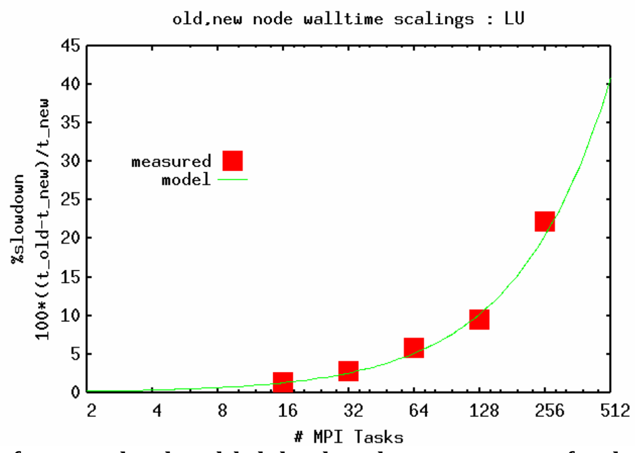

Figure 12: Comparison of measured and modeled slowdown between two sets of nodes in a parallel computer.

Over time a body of data and timings was established and it was found that parallel jobs ran slower, in proportion to their concurrency, on old nodes. The degree of the difference depends on the concurrency and the amount of synchronization in the MPI calls used in the code. A test case employed in the resolution of this issue is the NAS parallel benchmark LU because it was turned out to be a fast, reliable probe that coincided with the performance difference of full scale applications.

Since serial codes show no measurable difference the parts of the parallel codes that involve synchronization are implicated. Interruptions at the OS level or at the switch adapter level can have a minimal impact on serial processes, but compound when many concurrent processes are interrupted. If a linear model of frequent short interruptions on each node is extrapolated, the old nodes have half the performance of the new nodes (for LU decomposition) at a concurrency of 1250 tasks. Everything observed from the testing shows that synchronization of parallel jobs were impacted by delays proportional to concurrency. However it was not known if these interruptions were from hardware or software.

Process	#calls	#calls	Asymmetry
Name	old	new	ratio
#spget_sy	191817	6864	27.945367
#fcistm	192121	7188	26.728019
#lssrc	194608	7780	25.013882
#basename	385701	15550	24.803923
#odmget	193918	8481	22.864992
#ksh	390625	20129	19.406081
#rm	197514	12449	15.865853
#sed	397999	29482	13.499729
ksh	206206	23159	8.903925

By looking at UNIX process accounting logs it was possible determine the control work staion (CWS) on certain nodes ran several processes (lssrc, spget, odmget) and other system administrative commands up to 27 times more often than other nodes. This led to analysis of the problem management subsystem (pman) and found that while four definitions were deactivated using the system management GUI, they were still running. A problem in the implementation meant pman required explicit deletion of definition, rather than deactivation as was documented.

Once the definitions were deactivated, applications ran with very low variability regardless of how many new and old nodes it used.

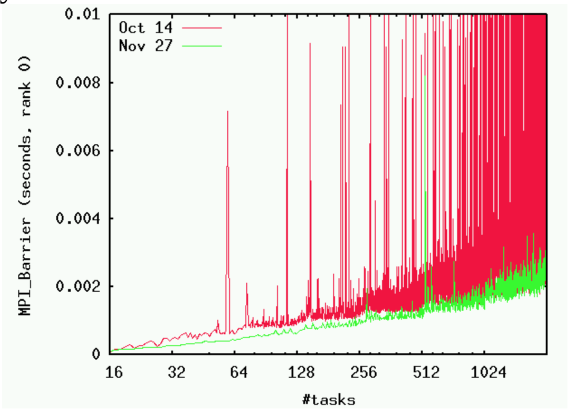

Figure 13: The parallel scaling of MPI_Barrier after leaving and MPI_Barrier is compared before (red) and after (green) changes where made to lessen synchronization variability within the machine.

Furthermore, applications run at high concurrency with much less variability. Resolving this issue lead to a measurable improvement for synchronizing MPI codes at high concurrency. In normal operation, jobs use a combination of old and new nodes. Thus, the end result is that all codes see a benefit of faster and more consistent run times, particularly those codes at higher concurrency. The performance gains shown earlier in Fig 2. and Fig 3. are a result of the changes described in this section.

Conclusion:

Many factors lead to variability in HPC application runtime on parallel architectures. Variability in performance is inherently tied to contention for resources between applications and operating system.

Events that occur as a machine evolves over years and at the microsecond timescale lead to the performance distributions observed in production environments. Among these are:

- OS/Kernel intrusion on applications
- Transaction processing oriented kernel scheduling algorithms
- Inter-Application contention for shared resources
- Concurrency as a magnifier of synchronization penalties

In general application level interruptions due to resource contention and process scheduling are at the root of runtime variability. It is hoped that by illustrating the specific mechanisms by which variability is introduced and by quantifying the impact of variability on parallel codes that system architects might better understand which elements of design impact HPC goals and that developers of parallel applications might better understand how to mitigate the impact of variability.

It may be possible to reduce the causes of variability (I-V) detailed above down to a smaller subset of fundamental causes, the profiling and monitoring required to do so would likely introduce system load that could skew the analysis. The detail and scope of performance monitoring which best suits HPC computing is an active and open area of investigation.

References:

1) WWC-7: IEEE 7th Annual Workshop on Workload Characterization
http://www.iiswc.org/wwc7/

2) Performance Variability of Highly Parallel Architectures, William T.C. Kramer and Clint Ryan, High Performance Distributed Computing, June 2003, Melbourne, Australia

3) HATS and HAGS in RSCT:
http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/tips0090.html?Open

4) IEEE Transactions on Parallel and Distributed Systems archive Volume 12 , Issue 9 (September 2001) Yanyong Zhang, Anand Sivasubramaniam José Moreira Hubertus Franke, “http://portal.acm.org/citation.cfm?id=506022&dl=ACM&coll=portal”

5) "The case of the Missing Supercomputer Performance" Fabrizio Petrini, Darren J. Kerbyson, Scott Patkin , SC2003, “http://www.scconference.org/sc2003/paperpdfs/pap301.pdf”

6) “Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System” Jones, Tuel, Maskell http://www.sc-conference.org/sc2003/paperpdfs/pap136.pdf

7) http://www.nacse.org/perfdb/