OAK RIDGE
NATIONAL LABORATORY
MANAGED BY UT-BATELLE
FOR THE DEPARTMENT OF ENERGY

Linux OS Jitter Measurements at Large Node Counts using a Blue Gene/L

November 30, 2009

Prepared by
Terry Jones
Colony Project Principle Investigator

UT-BATTELLE

.

Computer Science and Mathematics Division

LINUX OS JITTER MEASUREMENTS AT LARGE NODE COUNTS USING A BLUE GENE/L

Terry Jones, Oak Ridge National Laboratory
Andrew Tauferner, International Business Machines
Todd Inglett, International Business Machines

Date Published: November 30, 2009

CONTENTS

LIST OF FIGURES

Figure   **Page**

ACKNOWLEDGEMENTS

ABSTRACT

We present experimental results for a coordinated scheduling implementation of the Linux operating system. Results were collected on an IBM Blue Gene/L machine at scales up to 16K nodes. Our results indicate coordinated scheduling was able to provide a dramatic improvement in scaling performance for two applications characterized as bulk synchronous parallel programs.

1. INTRODUCTION

As part of the HPC-Colony project[1] sponsored by the DOE's FastOS program[2], we are conducting research on operating system (OS) issues on extremely large systems. In particular, we are actively investigating scaling issues associated with system software including certain coordination aspects of our smart runtime system and operating system. One key aspect of our work is concerned with correcting problems associated with operating system jitter[3]· [Jones03a, Jones03b, Chakravorty06] Recently, we had a chance to test the scalability of some of our approaches in those areas. We were able to start our tests on configurations with a few thousand BlueGene/L processors, and then increase the test configuration to the full machine size, so that the scaling behavior of our techniques can be assessed. This technical report describes the tests we conducted, the results we obtained, and the conclusions we could draw from such experiments.

The remainder of this report is organized as follows. In Section 2 we briefly describe our motivation. Section 3 describes the experiments that we conducted and provides the scalability results on large Blue Gene/L configurations. Finally, Section 4 contains our conclusions and a summary of our achievements during BGW-Day.

---

[1] HPC Colony Project: http://www.hpc-colony.org
[2] DOE office of Advanced Scientific Computing Research FastOS Program http://www.fastos2.org/
[3] Note that operating system jitter is also commonly referred to as operating system noise or operating system interference

2.   TECHNICAL BACKGROUND AND EXPERIMENT MOTIVATION

2.1   IMPORTANCE OF OPERATING SYSTEM SCALABILITY

Operating and runtime systems provide mechanisms to manage system hardware and software resources for the efficient execution of large scale scientific applications. They are essential to the success of both large scale systems and complex applications and must track changes in programming paradigms and computer architectures.

We are in a time of swiftly changing computer architectures and this has affected operating systems in multiple ways. One of the more notable emerging computer architecture trends is the advent of wide-scale parallelism – especially in the High Performance Computing (HPC) domain. Within the next 10 years, exascale computers with unprecedented processor counts and complexity will require significant new levels of scalability and fault management. Advances in parallel systems in the last decade have delivered phenomenal progress in the overall capability available to a single parallel application. According to the November 2009 Top500 survey, sixty-two systems now have a peak capability of over 100TF and five systems have a peak exceeding 1 PF. This is accomplished with ever increasing node counts and five systems now have total core counts exceeding 100,000. [Top500]

Under such core counts, full-featured operating systems such as Linux have shown a deleterious effect on parallel programs. Full-featured operating systems incorporate many small tasks as independent threads to perform common tasks including hardware-health-monitoring, parallel file system activity, and so on. It has been demonstrated that these independent activities result in an operating system jitter or interference that causes scalability issues. [Jones03a, Jones03b, Ferreira08] For example, at the Association for Computing Machinery (ACM) Supercomputing07 conference, Aroon Nataraj of the Tau team at the University of Oregon reported 23% to 32% increase in runtime for parallel applications running at 1024 nodes and 1.6% noise operating system noise. [Nataraj07]

2.2   HOW OPERATING SYSTEM JITTER AFFECTS PARALLEL APPLICATIONS

Parallel programs are frequently implemented in the Bulk-Synchronous Single-Program-Multiple-Data (SPMD) programming model. For this programming model, computation consists of one or more *cycles* or *timesteps* (see 'repeat' cycle in Figure 1).
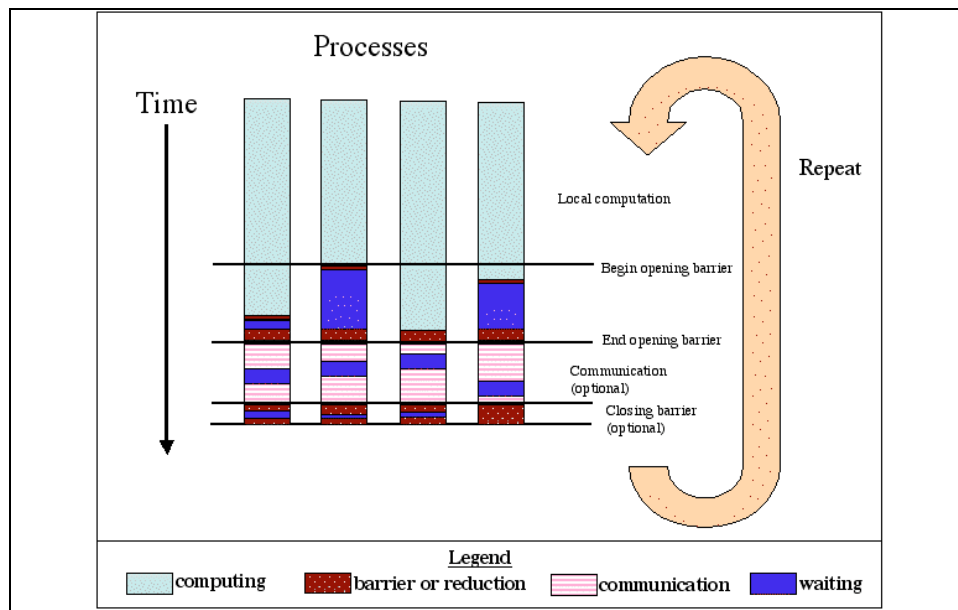

Fig. 1: Bulk-Synchronous SPMD Model of parallel application. Each process of a parallel job executes on a separate processor and alternates between computation and communication phases. Adapted from Dusseau96.

Each cycle may contain one or more ***synchronizing collective*** operations – an operation in which a set of processes (frequently every process) participates and no single process can continue until every process has participated. Examples of synchronizing collective operations from the Message Passing Interface (MPI) interface are MPI_Barrier, MPI_Allreduce, and MPI_Allgather. For example, a parallel application designed to simulate climate may use the MPI_Allreduce operation to find the maximum pressure present in an array distributed over all nodes; note that the overall maximum pressure cannot be determined until every node has contributed its maximum. Synchronizing collective operations pose serious challenges to scalability since a single instance of a laggard process will block progress for every other process.

Synchronizing collectives are common in parallel applications. Even though today's most prevalent operating systems, including Linux, do not include synchronizing collectives -- operating systems may determine the scalability of a parallel application running in user-space if the parallel application contains synchronizing collectives. When we speak of the ***scalability*** of an operating system, we are referring to the operating system's ability to support a parallel application without introducing scaling issues for the parallel application. Adverse performance associated with synchronizing collectives would seem to restrict their usage, but unfortunately synchronizing collective operations are required for a large class of parallel algorithms. [Gupta91]

Operating systems impact synchronizing collectives in the following way. A *cascading effect* results when one laggard process impedes the progress of every other process. The cascading effect has significant operating system implications and proves especially detrimental in an HPC context: while operating systems may be considered very efficient in a serial context, even minimal system and/or daemon activity proves disastrous due to the cascading effect in the large processor count parallel environment common in HPC centers. When interruptions occur on a subset of the computer nodes used for a parallel application during a synchronizing collective (e.g. an interruption for operating system activity such as a buffer flush), the degree of overlap is a key component in determining the performance impact of the interruption event on the synchronizing collective operation.

Fig. 2: Coordinated and uncoordinated schedulings. The above figure depicts two schedulings of the same eight-way parallel application. In the lower depiction, co-scheduling increases the efficiency of the parallel application as indicated by the larger amount of time periods where progress can be made across the entire 8-task parallel application. The top legend is blue; the middle legend is red, and the bottom legend is green. Adapted from Jones03a.

Figure 2 graphically portrays **two** separate runs of an eight-way parallel application with time as the x-axis. In the top instance, system activity (denoted as dark-red rectangles) occurs at purely random times. As a result, operations that require all eight processors to make progress are able to go forward only when medium-blue is present across all eight processors vertically (at one point in time). The light-green rectangles show those periods in time when the application is running across all 8 processors. In the bottom portrayal, the same amount of system activity occurs (there is the same total amount of red) but it is largely overlapped. This means much more time is available for parallel activities requiring all processors, as shown by the larger green rectangles.

For clusters comprised of nodes with more than one core, both inter- and intra-node overlap is an issue. Notice that if the eight cores in Figure 2 are spread across two 4-core nodes, it is desirable to ensure overlap between nodes as well as on-node. The bottom run shows very good on-node overlap of operating system interference, but does not fully achieve cross-node overlap of operating system interference.

2.3 COLONY'S APPROACH TO IMPROVING OPERATING SYSTEM SCALABILITY

Today, the primary operating system approach for mitigating parallel scalability issues is the ***lightweight kernel***. Lightweight kernels tradeoff generality for efficiencies in resource utilization, improved scalability for large-scale bulk-synchronous applications, and simplicity of design. Resource utilization differences between lightweight kernels and Linux-like full-featured operating systems include memory footprint and overhead CPU cycles. Entire lightweight kernels may be 3 MB or less. Overhead CPU cycles are reduced through the removal of various services like those provided by most daemons and those associated with socket communication or asynchronous file I/O. Examples of a lightweight kernel include the Puma operating system on Advanced Simulation & Computing (ASC) Red [Wheat94], the Catamount operating system on ASC Red Storm, and the IBM Compute Node Kernel (CNK) operating system on ASC BlueGene/L [Almasi03]. This technical report includes results from the CNK kernel for comparison purposes.

Our research has focused on providing portable performance. To maximize the set of applications and activities available, we chose a ***full-featured kernel*** – in our case Linux. Unlike lightweight kernels, full-featured kernels support the full POSIX Applications Programming Interface (API) including calls for spawning new processes (fork or clone), socket calls, memory mapping calls, and others not normally found on lightweight kernels. They also support multiple threads and daemons (a typical paradigm used by helpful programming development and debugging tools). In choosing a full-featured kernel, our primary challenge becomes one of achieving high scalability.

The Colony Linux kernel achieves high scalability through ***coordinated scheduling*** techniques and other strategies aimed at reducing operating system overhead. Coordinated scheduling (also referred to as ***parallel aware scheduling***) seeks to reduce the impact of operating system jitter. This is accomplished by increasing the overlap of interruption activity (e.g., increasing the overlap of 'red activity' in figure 2). Colony establishes two alternating intervals for activity across the entire parallel computer. During the longer interval, the parallel application is scheduled (e.g., the 'blue activity' in figure 2). This is accomplished by modifying the Linux scheduler to favor the parallel application with a high scheduler priority. During a shorter interval, other necessary activities such as health-monitoring daemons, parallel file system daemons, and so on, are scheduled (e.g., the 'red activity' in figure 2). During this period, the normal Linux algorithms are used allowing delayed operating system activities to make progress. In this way, interfering interruptions from daemon activity are minimized. [Jones03b]

2.4 EXPERIMENT MOTIVATION AND NEW TECHNOLOGY TO BE EVALUATED

Our previous work incorporated coordinated scheduling through the use of a separate daemon. [Jones03b] While that approach was a success, we were not able to achieve the tight time tolerances we desired. This technical report presents our first results of incorporating coordinated scheduling into the actual operating system kernel scheduler – an approach that permits much tighter time tolerances.

Earlier Colony kernel work focused on reducing the impact of translation lookaside buffer (TLB) misses (a small but frequent performance detriment). Previous results for Big-pages with the Colony Linux kernel show very good MPI_allreduce scaling for today's standards (e.g. compared to the ASC Purple machine), but substantially below the BlueGene lightweight kernel (CNK). The results presented here include our parallel aware scheduler that we designed for improved Linux scalability together with our memory system changes.

Our tests were designed to measure the performance of our coordinated scheduling kernel strategies. We conducted an assessment of our schemes on our parallel aware scheduler, a typical scheduler, and the lightweight IBM CNK. We envision a computational environment where full Linux is one of a number of choices for compute node system software, thus realizing a portable environment for large node count supercomputers. [Jones07] In order to realize this vision, the "noise" or "operating system jitter" usually attributed to full Linux must not be prohibitive.

Operating system jitter may stem from many sources including timer decrement interrupts, TLB activity, and interference from daemons and threads other than the parallel application running on the system. The Colony Linux kernel mitigates the negative impact through a variety of techniques including a large-page TLB design and parallel aware co-scheduling. The tests we performed provide a baseline for strategies we are developing to provide excellent portability together with excellent scalability.

3.   RESULTS

3.1   EXPERIMENT COMPONENTS

Two simple HPC applications were used during these experiments. Both are MPI programs that perform reduction operations and measure node to node communications performance. As a result they are good indicators of OS noise or interference, making them well suited to evaluation of the kernels used in these experiments.

These applications were slightly modified to allow them to identify themselves to the kernel as the HPC task. The kernel creates two files in /proc that applications can utilize to specify scheduling parameters as well as to identify the HPC application. Only one of the kernels evaluated in these experiments leveraged that information. A cleaner approach would have been to rely on the BlueGene I/O daemon to identify the HPC application as it was launched but this work was not possible in the available time. Both approaches would work well and the one used is not significant for our purposes.

In addition, three different kernels were used for these experiments: IBM's light-weight kernel known as CNK and two Linux kernels created by the Colony project. Both of the Linux kernels used 64KB memory pages instead of the standard 4KB pages to minimize TLB miss effects. One of the Linux kernels also incorporated "coordinated scheduling" to dynamically modify scheduler behavior. Coordinated scheduling allocates computing resources to the HPC applications in an advantageous way (possibly at the expense of background daemons and other tasks which may be delayed).

Finally, a separate "noise application" was developed to provide a tunable level of noise with characteristics similar to that observed on HPC systems.

3.1.1   Computation Facilities

All experiments were carried out on the Blue Gene / L machine located at IBM's T.J. Watson Research Center during the 2008 BGW day. This is a 20K node machine; 16K nodes were available for Colony.

3.1.2   Application 1: Allreduce

The first application is called allreduce. As the name suggests it performs multiple all reduce operations among all the available compute nodes. Consecutive reduction operations are performed, reporting the time per task for each operation. The average elapsed operation time is reported at the end of each run; it is the value that we focused on for these experiments.

3.1.3   Application 2: glob

The second application is called glob. The glob application is similar to allreduce in that it performs reduction operations. However, it measures performance over a range of node counts. The node counts cover the powers of two from two to the maximum number of nodes available. For each node count 500,000 operations are performed. The operation size is 32 bytes -- the size of a typical synchronizing collective operation (e.g., finding the min or max for two double-precision objects).

This application uses sub-communicators so the performance on CNK is sub-optimal for node counts less than the maximum for the block. The performance at the maximum is significantly better than the other node counts because under those node counts BlueGene's collective hardware cannot be utilized. We will focus on the results for the maximum node count at each block size.

3.1.4   Kernel 1: Blue Gene/L Compute Node Kernel

CNK is a flexible, lightweight-kernel for BlueGene/L compute nodes capable of running user MPI applications at large scale. It is a "Linux-like" operating system from the application's point-of-view. This is achieved by supporting a large subset of Linux compatible system calls. A portion of CNK's system calls required custom implementations (such as memory management system calls), but the majority of system calls are "function shipped" to a program called CIOD running on the I/O Node. This program serves as a proxy, replaying those system calls on a real Linux kernel and shipping the result back to CNK. This allows CNK to achieve precise semantics of Linux system calls without the costly code volume and effort needed to keep in-sync with Linux semantics. CNK has two process modes that can be used to execute applications: coprocessor mode and virtual node mode. In coprocessor mode, the application runs on 1 core, the 2nd core being used for communications offloading. In virtual node mode, 2 applications are executed, each on its dedicated core. It should be noted, however, that CNK does not support dynamic process creation.

One of CNK's principal design points was to avoid OS noise. It runs one process at a time; therefore it does not need to perform time-slicing or preemptive multitasking. It runs with a (TLB) memory map that is statically chosen when the job is loaded. This static memory map completely avoids TLB misses that can randomly degrade performance, particularly in large scale systems like BlueGene. The only instances in which the kernel interrupts the application is when a hardware event occurs, a debugger request needs processing, or an asynchronous signal is injected by the user.

3.1.5   Kernel 2: Colony Linux Kernel with unmodified Scheduler

The Colony Linux kernel is based on Linux version 2.6.16 from kernel.org. The original kernel source was updated to support BlueGene/L hardware. A console driver and RAS driver were added in addition to various changes to support the BlueGene/L platform. The default 4KB pages were replaced with 64KB pages.

3.1.6   Kernel 3: Colony Linux Kernel with Coordinated Scheduler

The differences between this kernel and the kernel described in section 3.1.5 involve the scheduler. Two /proc interfaces were created and the scheduler was modified to give priority to the HPC applications in a coordinated fashion.

3.1.7   Noise Generation

The Colony Linux kernel described in section 3.1.6 normally generates little noise because there aren't many daemons or other tasks running. A more realistic supercomputer environment might have various daemons running for various purposes (e.g. health of system monitoring, parallel file system activity, and so on). These daemons will increase noise and undoubtedly impact HPC applications. To assess the impact of noise we developed a simple program that was run in the background to create noise on every compute node during some of the experiments.

The noise program repeatedly calculates pi using Leibniz' method. The number of terms in each calculation of pi is chosen at random. An upper-bound determined the maximum time spent on a calculation. After each calculation the noise application performs a sleep for 500,000 microseconds.

By varying the upper-bound we were able to adjust the ratio of time spent doing calculations to sleeping for the noise application. For example, an average 10,000 microsecond calculation followed by the fixed 500,000 microsecond sleep after each calculation was termed 2% Noise Overhead (the net effect of a competing serial CPU-intensive job would be a 2% slowdown to the serial CPU intensive job). The noise application spends the majority of its time sleeping but the random amount of work that it does while calculating pi is a good source of noise.

The source code for our noise generation daemon is found in Appendix A.

3.2   OPERATION OF THE COLONY KERNEL

The Colony Linux kernel maintains two classes of runnable tasks: the parallel application, and everything else. The parallel application cycles between phases of favored-priority and regular-priority. During favored-priority, all other tasks and/or daemons are starved in preference to the parallel application. The cycling between favored-priority and regular-priority is accomplished concurrently across the entire machine. Therefore, it is impossible for daemon activity to interrupt *any* task of the parallel application during an epoch of favored-priority. A full duration of favored-priority and regular-priority is termed a single *period*. The duration for the favored cycle is termed the *duty cycle*.

The /proc interfaces, /proc/hpc/period and /proc/hpc/duty_cycle allow the HPC application to tune scheduler behavior as well as identify itself to the kernel. These values represent the total time per HPC application time slice plus the non-HPC application time slice. The period and duty cycle are measured in processor cycles.   The maximum period of 0xffffffff represents approximately 6 seconds on BlueGene/L's 700,000,000 cycles/second processor clock.   The duty cycle represents the number of cycles that will be devoted to the HPC application. Therefore, a duty cycle of 0x7fffffff represents 50% of the cycles in a period.

Fig. 3. Measuring the effect of "Duty Cycle".

Figure 3 shows the effect of the duty cycle on the Allreduce application. These measurements were taken on 16,384 nodes.   The performance of the Allreduce application increased as the duty cycle was increased. The average increase in performance was 3.33% for each increase in the duty cycle. Further increases in the duty cycle were not reasonable due to the limited amount of cycles that would remain for the non-HPC tasks. At a duty cycle of 0xf4000000, the remaining 0x0bfffff cycles leave a window of opportunity of only a few scheduler ticks for the non-HPC tasks to run. Increases beyond 0xf4000000 showed little or no application performance improvement.   At 0xfc000000 the performance increase versus 0xf4000000 was less than 0.26 %.   Increases in duty cycle beyond 0xfc000000 showed no improvement in HPC application performance.

The duty cycle and period are examined by the scheduler during each scheduler tick. Based on the CPU timebase and the specified period and duty cycle, the scheduler determines if it should favor the HPC parallel task or all other tasks. The idea is that the HPC task should be given the vast majority of CPU cycles for a given period while other tasks receive some limited number of cycles at regular intervals. The HPC parallel tasks are run with real time priority and FIFO policy that prevents background daemons and other tasks from interrupting it. The non-HPC tasks are allowed to run during small windows of time by reducing the HPC task's priority and the scheduling policy, allowing other tasks a chance to run. To assure the favored and regular intervals occur at the same time across nodes, the Colony Linux kernel utilizes a hardware heartbeat present with BlueGene/L machines that provides a minimal amount of discontinuity among the nodes. This leads to the scheduler ticks happening in concert on all nodes. As a result the HPC task interruptions at the end of the duty cycle are also synchronized.

3.3   EXPERIMENT PROCEDURES

All applications and kernels were evaluated at various node counts to look for scaling issues. Those experiments at 1K and 2K node counts were run on a BlueGene/L system in Rochester while the remainders of tests were run on the IBM TJ Watson BlueGene/L system. A maximum of 16K nodes out of a total of 20K nodes of the T.J. Watson BlueGene/L system were available for these tests.

The following node counts were utilized: 1024, 2048, 4096, 8192, and 16384. All blocks used a 128:1 compute node to IO node configuration. We believe that these experiments could be scaled to much higher node counts and plan to pursue such experiments in the future. For most node counts a baseline measurement was taken by booting the standard BlueGene/L system software (CNK kernel) to run glob and allreduce. Data from CNK at 8096 nodes was not collected due to time constraints. Since glob takes a significant amount of time to run, CNK data was also not collected for 4096 nodes

Figure 4 illustrates how the maxTermsPerCalc parameter of the Noise daemon determines the amount of noise present during each experiment. This parameter allowed us to easily choose noise levels between 0 and 35% (see section 3.1.7 for description of noise overhead).
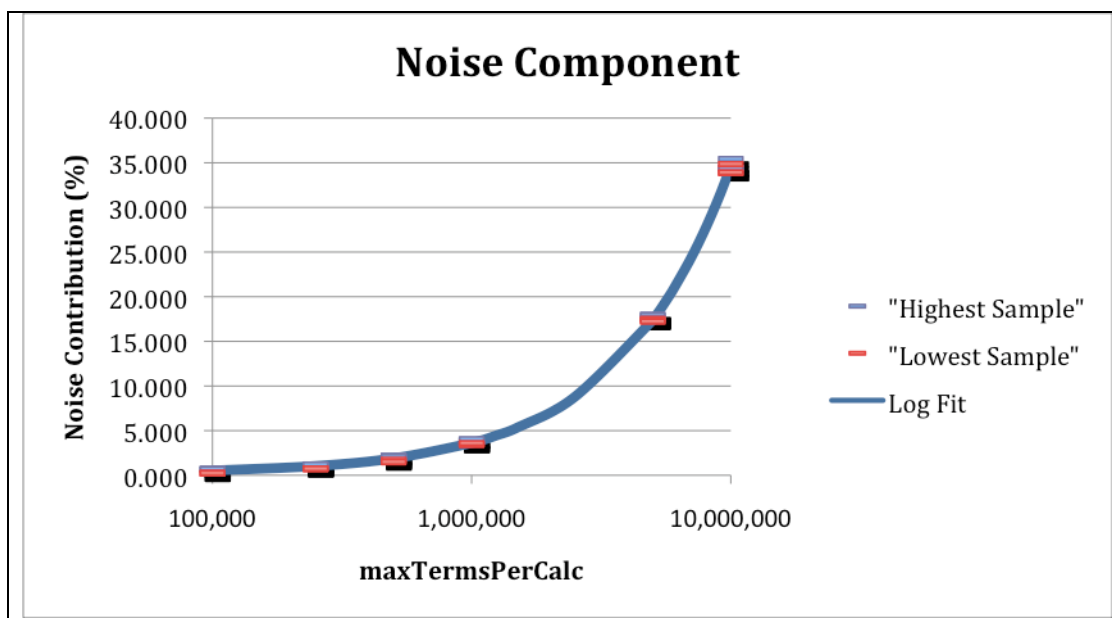

Fig. 4. Noise injection made possible by the noise daemon.

10

For each Colony kernel, both glob and allreduce were run with and without the noise application running in the background.

To examine the effect of the duty cycle this parameter was varied for the allreduce application at full scale only. Since the period and duty cycle were passed as environment parameters during program launch a variety of values could be easily tested in a relatively short time. The duty cycle values chosen were: 0x000f4000, 0x00f40000, 0x0f400000, and 0xf4000000.

Figure 5 portrays how noise level affects the Allreduce program. We collected "noise-impacted" results with two separate noise levels. The first value was selected to be a typical noise level consistent with full-featured operating systems in a typical supercomputing center. For this value, 2% noise was used. The second value was chosen to be a "moderate level of noise" indicative of the amount of background activity one might encounter with an active parallel file system with buffered I/O, or perhaps a separate program development daemon such as a profiling or memory tool. Due to the non-linear fashion of performance impact in the presence of noise, we limited the "moderate" setting to 21%.
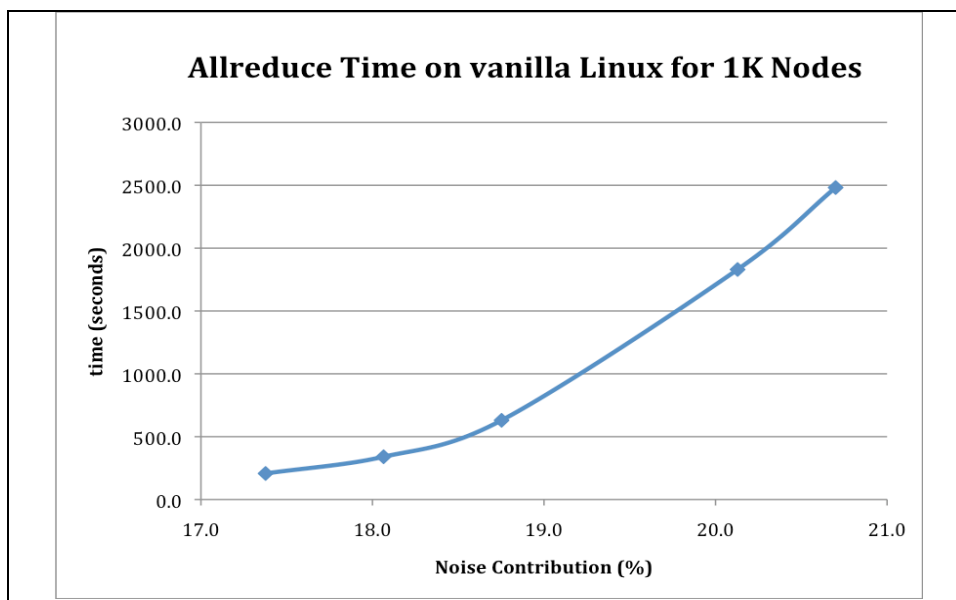

Fig. 5. Measuring scaling performance of Allreduce in the presence of OS Jitter.

Both Allreduce and GLOB are able to perform all their work within this window of 0xbffffff cycles if the number of terms in its calculation is kept below approximately 21% (maxTermsPerCalc = 5.9 million terms). Beyond 21% yields rapidly increasing Allreduce and GLOB run times.

3.4   OVERALL MEASUREMENTS

Our tests were designed to measure the effectiveness of our coordinated scheduler strategy – a parallel aware scheduler that we developed for improved Linux scalability in the presence of noise or jitter. The coordinated scheduler dramatically improved performance results.

We also measured the coordinated scheduler for added overhead when no jitter is present. The behavior of the unmodified scheduler without the noise program running and the Colony coordinated scheduler (with and without noise) were almost identical. The Colony Linux kernel was successful at mitigating the effect of the noise program and prevented any significant increase in Allreduce run time.

3.5 COLONY KERNEL BENEFIT DURING MODERATE NOISE

To determine the impact under moderate noise, we utilized the noise daemon to provide a 21% level of noise and ran it concurrently on the same nodes as a parallel application. The well understood Allreduce and GLOB parallel scaling codes were used and results were collected for up to 8192 processors. As we predicted, a moderate level of noise caused tremendous variability in common MPI operations for Linux without enhanced scheduling; measurements for individual operations ranged from about 4 microseconds to over 9900 microseconds. Next we turned on the parallel-aware coordinated scheduling and the variability for individual operations dramatically reduced -- the overall runtime for one of the tests reduced from 3175.7 seconds to 1.73 seconds!

Fig. 6. Measuring scaling performance of Allreduce in the presence of 21% OS Jitter.

The CNK kernel demonstrated the best scaling performance in most cases; the exception was glob performance of glob for processor counts less than the full partition count. Here, Colony with coordinated scheduling performed best of the three. The reasons for this are not fully understood but possible explanations are given below. GLOB showed nearly identical behavior where there was no noise interference or the when the Colony Linux kernel was used. Again the Colony Linux kernel was able to eliminate almost all the impact of the noise application.

Fig. 7. Measuring scaling performance of GLOB in the presence of 21% OS Jitter.

3.6   COLONY KERNEL BENEFIT DURING LOW NOISE

We then reduced the level of noise from 'moderate' to 'low' (2% noise). Under low noise conditions, the parallel-aware coordinated scheduling typically resulted in about a 10% improvement in overall runtimes.

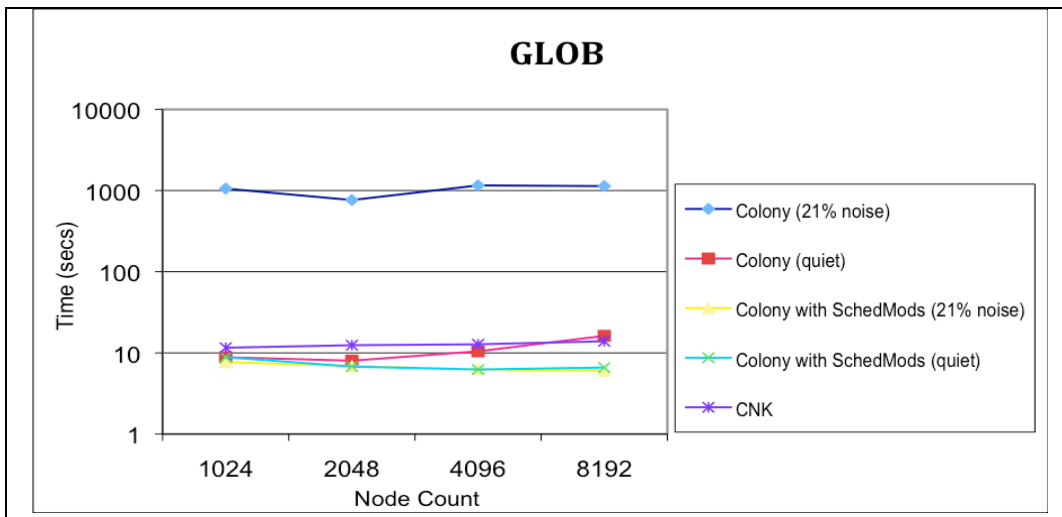**Fig. 8. Measuring scaling performance of Allreduce in the presence of 2% OS Jitter**.

On average the glob application performed just over 12% better on the Colony Linux kernel with coordinated scheduling versus typical Linux scheduling during small amounts of noise.

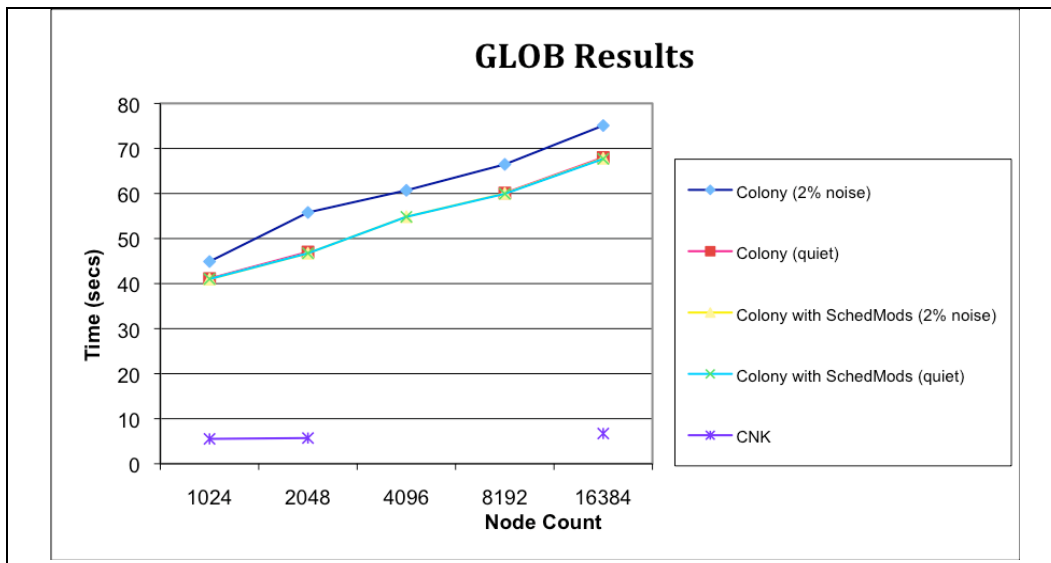**Fig. 9. Measuring scaling performance of GLOB in the presence of 2% OS Jitter**.

3.7   OVERHEAD OF COORDINATED SCHEDULING

We measured the overhead associated with our coordinated scheduling by making a series of measurements with no noise. The overhead was very low; in fact, many times the coordinated scheduling kernel actually ran faster than the kernel with standard scheduling.

13

4. CONCLUSION & JUDGEMENTS OF NEED

Our experimental results enabled us to verify that our approach of full Linux for large systems such as Blue Gene is scalable enough to be a valid option. The results that we obtained indicate that scaling for parallel aware Linux is noticeably improved over standard Linux scheduling. Indeed, some tests ran quicker on parallel aware coordinated scheduling Colony than on IBM's lightweight CNK kernel. The modifications to Linux proved to be a benefit when OS noise was present and, in fact, were better than the IBM CNK lightweight kernel in a few instances.

Since we chose to design our Colony Linux Kernel from a base 2.6.16.27 kernel, its coordinated scheduling capability should have wide application. Our results show it is possible *to allow* useful daemons while still providing a scalable system software infrastructure. We were able to demonstrate scalability while *allowing* the 'noise' or 'interference' levels associated with useful daemons (e.g. asynchronous I/O daemons, health of system daemons, and even other more invasive daemons for the HPC environment including program development, analysis, and debugging daemons).

Together, these results provide ground-breaking evidence that Linux modified with an effective coordinated scheduling scheme can provide excellent HPC scalability.

The overall benefit from coordinated scheduling during moderate noise was dramatic -- the overall runtime for one of the tests reduced from 3175.7 seconds to 1.73 seconds. Similarly, the coordinated scheduling benefit during low noise was very beneficial typically resulted in a 10% to 12% improvement in overall runtimes. The measured overhead of coordinated scheduling was very low; in fact, many times the coordinated scheduling kernel actually ran faster than the standard kernel.

In addition, we note that the Colony Kernel with coordinated scheduling provides multiple benefits as a lightweight CNK alternative. Obvious benefits include (a) a full POSIX API; (b) access to a huge list of development tools like symbolic debuggers, memory allocation debuggers, performance profilers, and so on; (c) and easier transitions between platforms for code teams. Other benefits might not be so obvious: during our testing we noted a number of MPI operations that consistently ran faster on Linux than on CNK. This points to areas of potential improvement in the CNK MPI implementation that would have been hard to uncover without a CNK alternative such as Colony.

We plan to collect additional measurements of the Colony Linux Kernel with coordinated scheduling on larger node counts. In addition, we plan to incorporate a software solution that obviates the need for hardware time-synchronization heartbeat. Finally, we plan to enhance Colony's ability to support adaptive runtimes (including the Charm++ runtime) and overlay communication systems.

5. REFERENCES

[Almasi03]   George Almasi, Ralph Bellofatto, Calin Cascaval, Jose G. Castanos, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Jose E. Moreira, Alda Sanomiya, and Karin Strauss. An Overview of the BlueGene/L System Software Organization, Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003) Aug. 2003, Klagenfurt, Austria.

[Chakravorty06] Sayantan Chakravorty, Celso L. Mendes, Laxmikant V. Kalé, Terry Jones, Andrew Tauferner, Todd Inglett, and José Moreira. HPC-Colony: Services and Interfaces for Very Large Systems. ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Systems, 40(2), April 2006.

[Ferreira08]   Kurt Ferreira, Ron Brightwell, and Patrick Bridges. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection, International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'08), Austin, TX, November 2008.

[Gupta91]   A. Gupta, A. Tucker, and S. Urushibara, The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications, In Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 120-132, May 1991.

[Jones03a]   Terry Jones, Jeff Fier, Larry Brenner, Observed Impacts of Operating Systems on the Scalability of Applications. LLNL Technical Report UCRL-MI-20269, March 5, 2003.

[Jones03b]   Terry Jones, Shawn Dawson, Rob Neely, William Tuel, Larry Brenner, Jeff Fier, Robert Blackmore, Pat Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts, Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System. Proceedings of Supercomputing 2003, Phoenix, AZ, November 2003.

[Jones07]   Terry Jones, Andrew Tauferner, and Todd Inglett. HPC System Call Usage Trends, the 8th LCI International Conference on High Performance Computing, South Lake Tahoe, CA, May 2007.

[Nataraj07]   Nataraj, A., Morris, A., Malony, A. D., Sottile, M., and Beckman, P. 2007. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing - Volume 00 (Reno, Nevada, November 10 - 16, 2007). SC '07. ACM, New York, NY, 1-12. DOI= http://doi.acm.org/10.1145/1362622.1362662

[Top500]   http://www.top500.org

[Wheat94]   S. R. Wheat, A. B. Maccabe, R. E. Riesen, D. W. van Dresser, and T. M. Stallcup, "PUMA: An Operating System for Massively Parallel Systems,'" Journal of Scientific Programming, vol. 3, no.4, Winter 1994, (special issue on operating system support for massively parallel systems) pp 275-288.

APPENDIX A.
NOISE DAEMON SOURCE

APPENDIX A.1 NOISE DAEMON SOURCE

```c
// noise.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define SPRN_TBRL 0x10c
static inline unsigned int getTimebaseL(void)
{
        unsigned tbl;

        asm volatile ("mfspr %0,%1" : "=r"(tbl) : "i"(SPRN_TBRL));

        return tbl;
}


int main(int argc, char** argv)
{
        unsigned int maxTermsPerCalc = 1000000;

        if (argc == 2)
                maxTermsPerCalc = atoi(argv[1]);

        srandom(getTimebaseL());

        while (1) {
                unsigned int term;
                unsigned int termsPerCalculation;
                long double pi = 0.0;
                long double denominator = 1.0;

                termsPerCalculation = random() % maxTermsPerCalc + 10;

                for (term = 0; term < termsPerCalculation; term++, denominator += 2.0) {
                        long double quotient = 4.0 / denominator;

                        if (term & 1)
                                // Odd
                                pi -= quotient;
                        else
                                // Even
                                pi += quotient;
        }

//        printf("%.20Lf (%d terms)\n", pi, termsPerCalculation);
                usleep(500000);

        }

        return 0;
}
```

Fig. Appendix A.1. Noise Daemon source.

INTERNAL DISTRIBUTION

1. Terry Jones, PI
2. Tammy Darland, APT Group Records
3. ORNL Office of Technical Information and Classification

EXTERNAL DISTRIBUTION

4. Andrew Tauferner, International Business Machines, IBM Systems & Technology Group, Rochester, MN 55901
5. Todd Inglett, International Business Machines, IBM Systems & Technology Group, Rochester, MN 55901
6. José Moreira, International Business Machines, IBM Research, 1101 Kitchawan Rd, Yorktown Heights, NY, 10598-0218
7. Fred Mintzer, International Business Machines, IBM Research, 1101 Kitchawan Rd, Yorktown Heights, NY, 10598-0218