Office of Science

HPC-Colony: Services and Interfaces to Support Systems With Very Large Numbers of Processors

2006 Annual Report
January 2006 through December 2006

Principal Investigators

Terry Jones
Lawrence Livermore National Laboratory
P.O. Box 808, L-561
Livermore, CA 94551
Tel: 925-423-2685
Fax: 925-424-2477
Email: trj@llnl.gov

Laxmikant Kale
Univ. of Illinois, Urbana-Champaign
1304 W. Springfield Avenue
Urbana, IL 61801
Tel: 217-333-5827
Fax: 217-333-3501
Email: kale@cs.uiuc.edu

José Moreira
International Business Machines
Thomas J. Watson Research Center
1101 Kitchawan Rd
Yorktown Heights NY 10598-0218
Tel: 914-945-3987,
fax: 914-945-4121
Email: jmoreira@us.ibm.com

Investigators

Celso Mendes	UIUC	cmendes@cs.uiuc.edu
Sayantan Chakravorty	UIUC	schkrvrt@uiuc.edu
Andrew Tauferner	IBM	ataufer@us.ibm.com
Todd Inglett	IBM	tinglett@us.ibm.com

Disclaimer

Auspices Statement

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

1. Introduction

The HPC-Colony Project, a collaboration with Lawrence Livermore National Laboratory, the University of Illinois at Urbana-Champaign and IBM, is focused on services and interfaces for very large numbers of processors.

Advances in parallel systems in the last decade have delivered phenomenal progress in the overall capability available to a single parallel application. Several systems with peak capability of over 100TF are already available and systems are expected to exceed 1PF within a few years. Despite these impressive advances in peak performance capability, the sustained performance of these systems continues to fall as a percentage of the peak capability. Initial analysis suggests that key architectural bottlenecks (in hardware and software) are responsible for the lower sustained performance and some architectural change of direction may be necessary to address the declining sustained performance.

In this proposal we focus on addressing software architectural bottlenecks, in the areas of operating system and runtime systems. While the trend towards larger processor counts benefits application developers through more processing power, it also challenges application developers to harness ever-increasing numbers of processors for productive work. Much of the burden falls to operating systems and runtime systems that were originally designed for much smaller processor counts. Under the Colony project, we are researching and developing system software to enable general purpose operating and runtime systems for tens of thousands of processors. Difficulties in achieving a balanced partitioning and dynamically scheduling workloads can limit scaling for complex problems on large machines. Scientific simulations that span components of large machines require common operating system services, such as process scheduling, event notification, and job management to scale to large machines. Today, application programmers must explicitly manage these resources. We address scaling issues and porting issues by delegating resource management tasks to a sophisticated parallel OS. Our definition of "managing resources" includes balancing CPU time, network utilization, and memory usage across the entire machine. We believe a consistent environment that provides newly necessary technology (such as fault tolerance) will also provide important efficiencies in system administration.

1.1 Purpose

The primary objective of the Colony Project is to develop technologies that enable application scientists to easily scale applications to computing platforms comprised of tens of thousands to hundreds of thousands of compute cores. This will be accomplished by addressing several problem areas that are known to be key factors when scaling applications to tens of thousands of processors. First, ***by providing*** a smart runtime system to quickly and dynamically make cpu and memory and interconnect resource management adjustments, we remove the burden of achieving applications that are highly tuned and load-balanced for a particular execution instance (*i.e.* a particular input datasets and machine platform combination). Second, ***by providing*** a full complement of system services including the entire Linux system call set, we ease the challenge of developing portable applications since lightweight kernels frequently incorporate only a small subset of the POSIX calls prevalent in typical large scientific applications. Third, ***by providing*** fundamental changes to the Linux kernel that reduce variability in context switch times and provide for parallel-aware scheduling across the entire machine, we remove the negative impact of synchronizing collectives on bulk-synchronous applications. Fourth, ***by providing*** fault tolerance mechanisms that utilize our unique migration abilities in conjunction with in-memory techniques for minimal overhead, we eliminate the necessity for costly frequent application-driven check-points. Our research utilizes full implementations of these technologies on systems consisting of tens of thousands of processors.

1.2 Approach

Among the decisions investigators researching HPC operating system and runtime system issues have faced historically are tradeoffs related to capability versus performance. One may choose to utilize an

operating system design developed for much smaller general purpose machines such as Unix derivatives as a starting point. While this approach does provide many highly desired features including support for a broad range of services and interfaces, the lack of any parallel awareness confines what may be managed by the system software stack. For instance, scheduling issues can dramatically impact scalability of common parallel algorithms. One may choose a lightweight operating system approach such as a custom micro-kernel. A primary goal of this approach is to remove services and interfaces from compute nodes, thereby eliminating the difficulties arising from a lack of parallel awareness. The challenge faced by this kind of approach is borne by the parallel application developer. They may face fixed memory constraints, reduced programming interfaces (many times, a partial POSIX programming interface), and reduced programming tools for development.

Our approach is based on a general purpose Linux kernel and supporting interfaces and services. Important benefits of this approach include the benefits from over 20 years of lessons on Unix-derivative operating systems, the large number of people contributing to Linux, and the support for many features typically missing from lightweight approaches (e.g., full POSIX, program development tools, sophisticated memory capabilities, and so on). A primary goal will be to remove performance and scalability problems found in general purpose operating systems.

Our deliverables will be software implementations of the Linux kernel, new system services, and a parallel runtime, Charm++. The technology will be applicable to cluster-based supercomputers with large numbers of processors; BlueGene/L type supercomputers will be used extensively in our work and will provide a demonstration platform. The interfaces in our software implementations will be generic. They can be implemented in other operating systems, used by other runtime systems on other platforms, and will support a wide spectrum of applications. In addition to improving the performance of flagship applications, we will develop performance metrics and test cases to quantify the benefits of our strategies and to provide a means for comparison with other approaches.

To investigate our strategies, we are implementing our strategies on IBM BlueGene type architectures. To our knowledge, BlueGene machines are the largest node count machines currently utilized by the HPC community (e.g. 128,000+ processors). Our environment differs from other BlueGene environments in that we will provide a fully functional Linux on BlueGene compute nodes.

2. Project Highlights From Calendar Year 2006

2.1 Linux on BlueGene Compute Nodes

We continue to develop an all-Linux solution for Blue Gene. The ability to execute Linux on the compute nodes of Blue Gene will significantly expand the range of applications for that platform. In 2006 we completed our design for a compute node Linux. We also got a first prototype Linux solution for Blue Gene compute nodes operational. The solution includes the ability to run MPI programs on the compute nodes with Linux. We here report our initial experience with this prototype solution.

We have previously reported results that quantify the impact of TLB misses on the performance of parallel applications in Blue Gene. More recently, we have done a more detailed study of the difference in performance observed when running the same application using either Linux or the lightweight Compute Node Kernel (CNK) on the Blue Gene compute nodes.

Figure 1(a) shows the performance of the NAS Parallel Benchmark IS on Blue Gene, running both under Linux and CNK. We observe a large performance gap (~2x) between Linux and CNK. The lower Linux performance can be explained by the large number of TLB misses that happens with Linux, as shown in Figure 1(b). That figure shows number of TLB misses per node, for different numbers of compute nodes, when operating with a standard Linux page size of 4 KiB. In comparison, CNK configures the TLB of the compute nodes so that there are no TLB misses during execution.

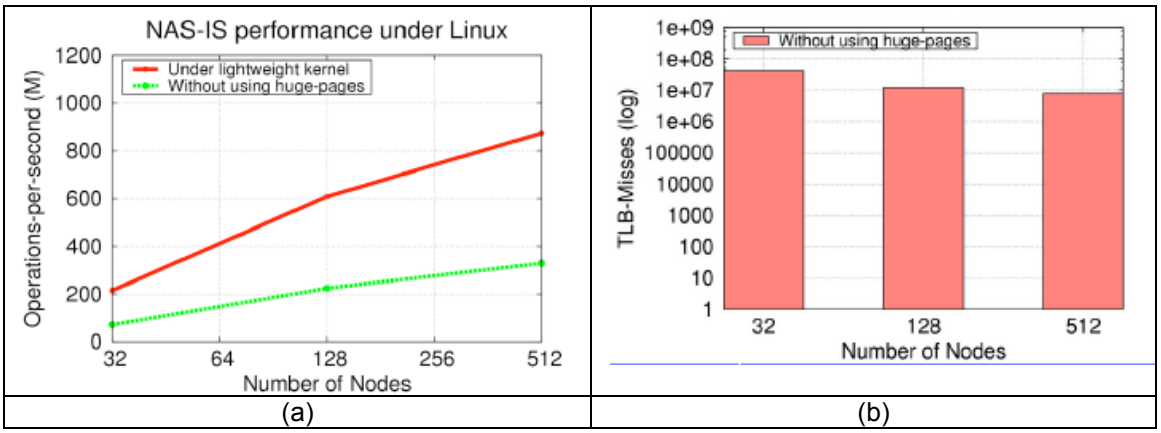

Figure 1: Performance of NAS Parallel Benchmark under Linux and CNK (a). The lower performance under Linux can be explained by the large number of TLB misses (b) under Linux. (There are no TLB misses under CNK.)

2.2 Linux Modifications For HPC Scaling

Since the excessive number of TLB misses is a major source of performance degradation under Linux, we started investigating how to avoid or reduce those misses. The first solution we investigated is the use of the "Huge TLB file system" feature of Linux. This is an in-memory file system that uses 16 MiB pinned pages to map virtual to physical memory. Files in the "Huge TLB file system" can be memory mapped into the address space of a process. When a process accesses memory in that region, the translation goes through the pinned 16 MiB entries in the TLB, and therefore there are no TLB misses for that region of memory.

In order to evaluate the impact of reducing the number of TLB misses we modified the IS benchmark so that most of the accesses were done against memory mapped through 16 MiB pages. The performance results are shown in Figure 2(a). We observe that performance under Linux (magenta line) is able to match performance under CNK. (We note that both data and heap segments had to be placed in a large page region to get the most benefit.) Figure 2(b) is a direct measurement of TLB misses and it shows the great reduction in the number of misses.

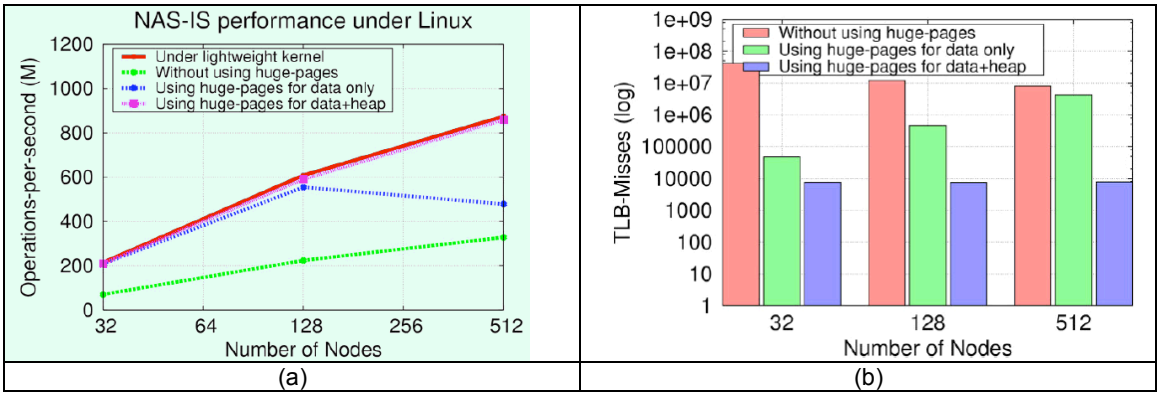

Figure 2: Performance of NAS Parallel Benchmark under Linux with "Huge TLB file system". The application was slightly modified to place both data and heap segments in a memory mapped file with 16 MiB TLB entries. We observe a drastic reduction in the number of TLB misses (b) and an improvement in performance under Linux (a), matching the performance under CNK.

Another well-known source of performance degradation when running parallel applications on a Linux-based system is system noise, usually introduced by system services (*e.g.*, daemons) running on the same nodes as the computation. To quantify the impact of this noise on the performance of Blue Gene, we ran a simple synthetic benchmark. This benchmark consists of 10,000 iterations. Each of

those iterations consists of 1ms of work followed by a barrier across all processes in the parallel execution. We ran the benchmark in four different configurations: (i) under CNK, (ii) under Linux with nothing else in the node, (iii) under Linux with a daemon that wakes every second for 1 ms, with the wakeup coordinate across nodes and (iv) under Linux with the same daemon, but the "phase" of each daemon is randomly spread over a second. Results are shown in

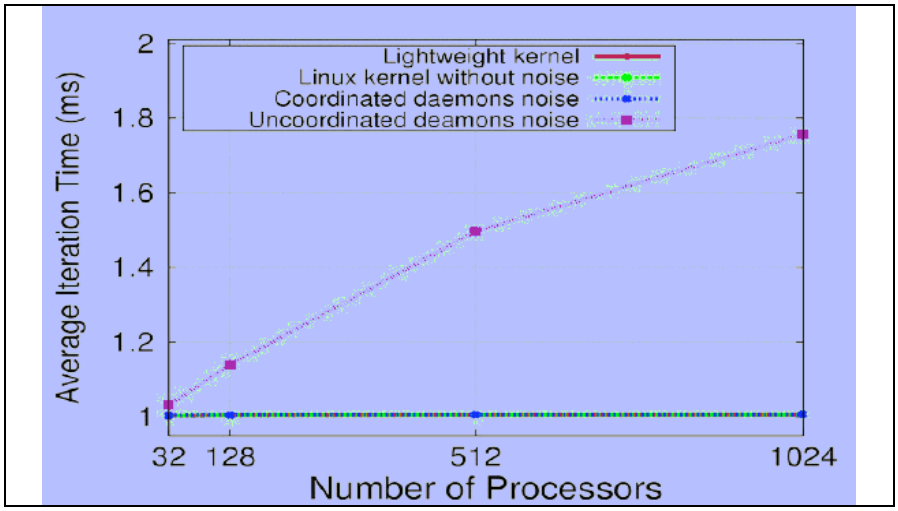

Figure 3: Performance of synthetic benchmark under CNK (lightweight kernel) and Linux with or without noise. "Linux with noise" means there is an additional daemon process in each node that wakes up for 1ms every second. The daemons in different nodes can either wake up all the same time (coordinated noise) or each daemon can have its own "phase" randomly distributed over a second (uncoordinated noise). (The red, green and blue lines are all on top of each other.)

We observe that significant performance degradation happens only when the daemons are uncoordinated. In other words, we do not have to eliminate all the daemons in a compute node Linux, but it is important to ensure that the daemons across the nodes get to execute all at the same time. This lets the application proceed uninterrupted for a long period of time.

We note that the experiments reported above (large pages, system noise) have been very important in quantifying the performance impact of an all-Linux solution for Blue Gene. There are still unresolved issues that need to be addressed in the solutions to the performance impact. For example, how can we deploy a 16 MiB TLB entry solution in a manner that is transparent to the application programmer? Also, how do large pages affect the functionality of other tools (*e.g.*, debuggers) that interact with processes using those large pages? In the system noise front, we are still investigating the proper way to coordinate daemon execution.

2.3 Fault Tolerance Techniques

We continued our research on fault tolerance techniques for large systems by working on three different schemes: in-memory checkpointing, proactive fault tolerance and message-logging. All of these schemes are being developed in the context of Charm++ and AMPI, and thus they will become available for many MPI codes. We now describe our recent activities for each of these schemes.

Our in-memory checkpoint scheme adopts the idea of diskless checkpointing, which checkpoints data in memory. It uses a coordinated checkpoint strategy. This reporting period, we conducted additional tests to assess the effectiveness of this technique on a large machine such as BlueGene/L. In these tests, we used a 7-point stencil with 3-D domain decomposition, written in MPI. For a particular number of processors, we varied the amount of data per processor. We took multiple checkpoints for every run and measured the average time for a checkpoint. Table 1 shows the average time for a checkpoint for varying amounts of data per processor on 8,192 and 20,480 processors of BlueGene/L at IBM's T.J.Watson. These results show that the checkpointing overhead increases proportionally to

the amount of state in each processor, as one would expect, even for a large BlueGene/L configuration. This fact confirms that this technique is useful for this class of system.

Number of Processors	Data per Processor (kB)	Checkpoint Time(s)
8192	80	.1613006
8192	300	.1626386
8192	1200	.1801744
8192	5000	.2370764
8192	20000	.4174428
20480	80	.3777704
20480	20000	.6924555

Table 1: Time taken to perform in-memory checkpoints with different amounts of data per BG/L processor

Our proactive fault tolerance scheme is based on the hypothesis that, some faults can be predicted. We leverage the migration capabilities of Charm++, to evacuate objects from a processor where faults are imminent. During this reporting period, besides conducting additional validation tests with this scheme (as reported in [CHAKR2006]), we also integrated this proactive scheme to the regular Charm++ codebase. Thus, the technique is now available for Charm++ users willing to test the evacuation approach. Currently, the evacuation is triggered by an incoming signal, and we assume that such signal could be derived from some condition external to the application.

In our message-logging schemes to tolerate system faults, the major goals are (a) to impose low overhead on the forward path (i.e. when there are no failures) and (b) to provide a fast recovery from faults, without wasting computation done by processors that have not faulted. During this reporting period, we continued to enhance a prototype version of Charm++ that employs this message-logging technique, and tested this scheme on large systems.

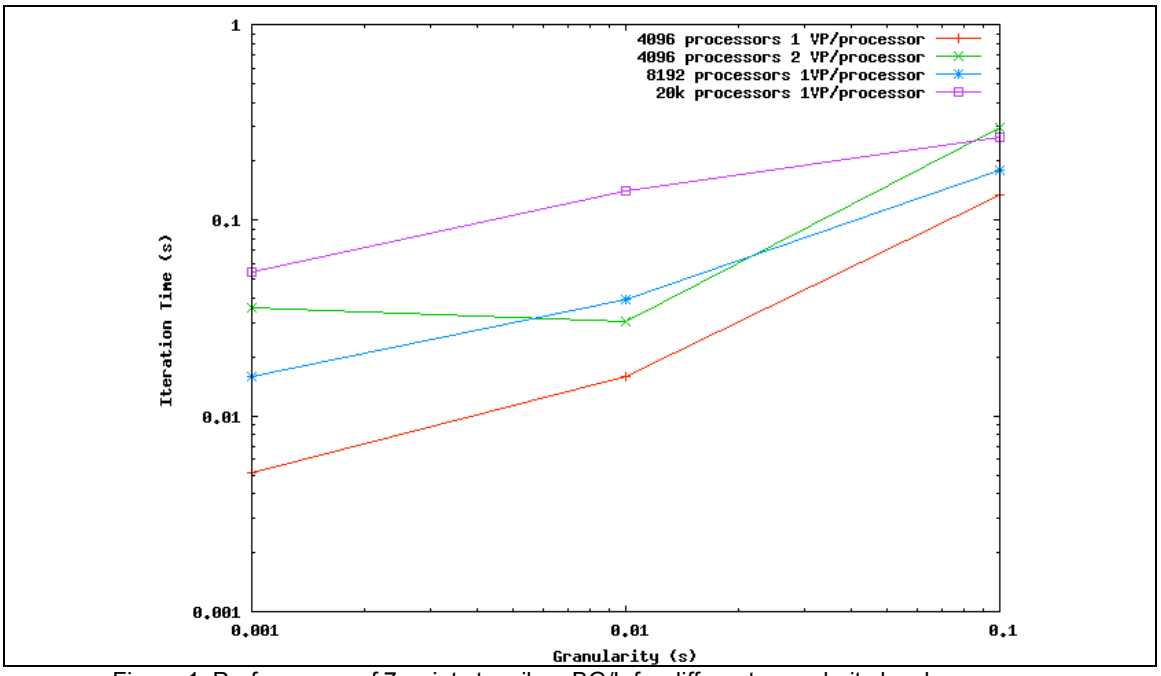

Figure 1: Performance of 7-point stencil on BG/L for different granularity levels

As an example of our tests, we used the same 7-point stencil with 3-D domain decomposition written in MPI, as before. The stencil program was written so that in every iteration each MPI task exchanges messages with its neighbors and then performs computations for a fixed amount of time. The amount of time a task computes in each iteration is called its granularity. Because our message-logging protocol creates additional messages between processors, it is important to ensure that these extra messages do not become a bottleneck in the execution, even in the case of a fault-free scenario. To verify that, we conducted executions of the 7-point stencil employing our message logging protocol,

and obtained the data presented in Figure 1. Those plots show the average iteration time for the 7-point stencil application for different values of granularity. We show results on four, eight and twenty thousand processors, using one processor per node of the BlueGene/L system. In the ideal case, with no extra penalty for communication or fault tolerance protocol, the iteration time would be the same as the granularity. The results in our plots confirm that our protocol scales to a large number of processors for applications with large granularity. When the granularity is smaller, there is a significant performance penalty on larger numbers of processors.

Number of Processors	4096	8192	16384
Memory Used (MB)	6.8	22.57	22.63

Figure 2: Memory Usage of the Hybrid Load Balancer

2.4 Global Resource Management

We have developed a new hybrid load balancing algorithm (HybridLB) that is designed for scientific applications with persistent computation and communication patterns. We use the automatic run-time instrumentation in Charm++ for collecting load data. HybridLB utilizes a load balancing hierarchical tree to distribute tasks across processors. This new algorithm takes advantage of Charm++'s processor virtualization idea for applications with fine-grained parallelism, and it takes communication into account for achieving satisfactory load balancing decisions. More details about this algorithm can be found at [ZHENG2005].

One of the most important goals of the HybridLB is to reduce the usage of the memory taken by the load balancing algorithms, including the memory space for the load database. For centralized load balancing schemes where a global load database is constructed on a particular processor, the load database can easily exceed the memory capacity of a node of BG/L.

We evaluated our new HybridLB with the Lb test program on BlueGene/L at IBM's T.J.Watson The test program creates a large number of objects having size 512 KBytes. These objects communicate in a 2D-mesh virtual topology similar to a 2D stencil computation. The hierarchical tree used by HybridLB is built as following: every 1024 processors form a load balancing domain at level 1, and 8 such domains in an 8K-processor simulation form the level-2 load balancing domain; meanwhile, in a 16K-processor simulation case, there are 16 such level-2 domains. We used greedy-based load balancing algorithms (GreedyLB) at level 1 and used a refinement-based load balancing algorithm at level 2. We ran this test program and measured the memory usage by the HybridLB load balancer. The results are shown in Table 2.

For comparison, we ran the same test with our centralized load balancing strategy. Indeed, the centralized scheme ran out of memory when constructing the global load database. We also measured the time spent in load balancing by HybridLB, as shown in Table 3. We found that the load balancing process itself is not as efficient as we expected. We captured performance traces on BlueGene/L and dumped logs for post-mortem analysis. Through this analysis, done with Projections (our performance analysis tool), we found that the multicast—sending 1024 messages down a hierarchical tree—is responsible for most of the overhead.

Number of Processors	8192	16384
LB Time	14.70s	23.16s

Table 3: Load Balancing time of the HybridLB

2.5 Other Activities

As a complement to our research work, the HPC-Colony project conducted bi-weekly conference calls. These teleconferences have been very useful to maintain our team in synchrony with the project partners, and to exchange valuable information regarding issues about existing large systems, in special the BlueGene/L platform. We had two face-toface meetings with our project partners, one during the FastOS meeting in May at Boston, and another in November during SC'2006.

In October, our team utilized the 20 rack BlueGene machine at T.J. Watson Research Center to

perform fault tolerance and scalable resource management research. This was done through submitting a winning proposal to the Blue Gene Consortium ***BGW Days*** competition. A full report of our tests is available.

Meanwhile, using resources from other sources, we have been conducting tests with Charm++ based applications on the BlueGene/L platform. These tests provide a synergistic complement to our efforts in the HPC-Colony project, as they give us a chance to test the new products of our developments (e.g. new load balancers) with real scientific codes on a large system.

We maintain an external website as well as a "project internal" website at UIUC. The URLs for those sites are the following:

- Colony project's URL: http://www.hpc-colony.org
- UIUC site for Colony project: http://charm.cs.uiuc.edu/grants/colony.html

3. Plans for the Upcoming Year

During this final year of our HPC-Colony project, we are continuing the development of our fault tolerance work based on message-logging and its integration to the Charm++ distribution. In addition, we are conducting tests where we compare the effectiveness of the various fault tolerance strategies available in Charm++.

Meanwhile, we continue to develop new schemes for load balancing, based on observed computation and communication characteristics of real applications. According to recent tests with real scientific applications, there are situations where our existing load balancers are not able to fully balance the load. These cases may arise, for example, when there is work outside the regular objects, such as in chare "groups" (a group is a special type of chare that has a unique instance in each processor). For these cases, we are creating new policies that take the groups into account for the load balancing decisions.

4 Publications and Talks

4.1 2006 Publications

- Tarun Agarwal, Amit Sharma and Laxmikant V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines, Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006, Greece, April 2006.
- Gengbin Zheng, Chao Huang and Laxmikant V. Kalé. Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++. ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Systems, 40(2), April 2006.
- 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.
- Sayantan Chakravorty, Celso L. Mendes and Laxmikant V. Kalé. Proactive Fault Tolerance in MPI Applications via Task Migration, Accepted for HiPC2006, Bangalore, India, December 2006.
- Sayantan Chakravorty, Laxmikant V. Kalé. A Fault Tolerance Protocol with Fast Fault Recovery. Accepted for the IEEE International Parallel and Distributed Processing Symposium 2007, California, March 2007.

4.2 2006 Talks

- Terry Jones, Operating System Interference Effects at Extreme Scale, SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, February 2006.

5. Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Funding provided by the U.S. Department of Energy Office of Science program.

Thanks to George Almasi, Jose Castanos and Edi Shmueli of the IBM T.J. Watson Research Center for implementing the prototype Linux solution for the Blue Gene compute nodes. In particular, Edi Shmueli performed measurements shown in sections 2.1 and 2.2 this report.

Finally, thanks to the Blue Gene Consortium for providing access to a large Blue Gene machine for system software research.