HPC-Colony: Services and Interfaces for Very Large Systems

Sayantan Chakravorty
Celso L. Mendes
Laxmikant V. Kalé

University of Illinois
{schkrvrt,cmendes,kale}@cs.uiuc.edu

Terry Jones

Lawrence Livermore National Lab.
trj@llnl.gov

Andrew Tauferner
Todd Inglett
José Moreira

IBM
{ataufer,tinglett,jmoreira}@us.ibm.com

ABSTRACT
Traditional full-featured operating systems are known to have properties that limit the scalability of distributed memory parallel programs, the most common programming paradigm utilized in high end computing. Furthermore, as processor counts increase with the most capable systems, the necessary activity to manage the system becomes more of a burden. To make a general purpose operating system scale to such levels, new technology is required for parallel resource management and global system management (including fault management). In this paper, we describe the shortcomings of full-featured operating systems and runtime systems and discuss an approach to scale such systems to one hundred thousand processors with both scalable parallel application performance and efficient system management.

1. INTRODUCTION
The HPC-Colony project, a collaboration between Lawrence Livermore National Laboratory, the University of Illinois and IBM, funded by the Department of Energy's Office of Science under the FastOS program, is focused on services and interfaces for systems with 100,000+ processors. 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 (OS) and runtime systems that were originally designed for much smaller processor counts. Under the HPC-Colony project, we are researching and developing system software to enable general purpose operating and runtime systems for hundreds of thousands of processors. Our strategy to redress these issues relies on new technology from the areas of parallel resource management, fault tolerance and global system management.

Parallel resource management is focused on assisting application developers on very large numbers 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 resource" includes balancing CPU time, network utilization and memory usage across the entire machine.

Reliability and fault tolerance become major concerns in systems with thousands of processors because the overall reliability of those systems decreases with a growing number of system components. Hence, large systems are more likely to incur a failure during execution of a long-running application. Traditional checkpoint/restart techniques remain valid solutions for this problem, but in some cases the overheads associated with restarting an application may be excessive: if one has to restart execution from a previous checkpoint on a large processor set due to the failure in just one processor, there is a severe waste of the available computing capability. We are developing alternative techniques that allow tolerating faults while imposing minimal overhead, and investigating the effectiveness of proactive approaches to fault handling.

Global system management is focused on assisting those who administer machines that consist of very large numbers of processors. We are enhancing operating system support for parallel execution by providing coordinated scheduling and improved management services for very large machines. Moreover, we believe machines with tens of thousands of processors place additional requirements on system services and interfaces related to efficiencies in system administration. This paper also reports our initial investigations regarding those operating system features that are critical in supporting large scale parallelism.

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

2. RESOURCE MANAGEMENT ISSUES
Resource management is a major function of a sequential OS; for a parallel machine, its OS must manage resources such as processors, memories and communication infrastructure. A major challenge in scaling applications to ultrascale

machines is efficient resource management. Today, application developers carefully specify the data to be housed on each processor, and which parts of the computation to perform on each processor in each phase of the application. However, with ever larger processor counts and ever more complex applications, programmers are finding it increasingly difficult to manually balance resource usage. By delegating these low-level resource management tasks to sophisticated OS/runtime software, parallel programs can be made to run more efficiently, as resource usage is better balanced. In addition, we believe this will allow programs to be created more efficiently. Such an automated balancing process requires minimal manual work, thereby reducing programmer time and effort. In the HPC-Colony project, we are developing infrastructure and strategies for automated management of parallel resources such as CPU time, network utilization and memory use.

OS processes are too coarse-grained to allow accurate load balancing. Our approach for automating resource management decomposes the parallel application into sub-process migratable work units such as user-level threads and parallel objects. For example, in Adaptive MPI (AMPI - our MPI implementation based on Charm++) [1], regular MPI applications are run using several user-level threads per physical processor. These threads, or parallel objects, can be migrated between physical processors. The parallel OS/runtime measure the CPU load and communication patterns of each object, and the obtained measurements are used by an application independent load balancer. The load balancer then decides when and where to redistribute work by migrating objects. We have used this approach to efficiently scale high-performance applications to several thousand processors (e.g. [2]).

Our previous work has shown that automated load balancers can provide significant performance improvement by reassigning parallel objects to physical processors based on measured load and communication patterns [2, 3]. When the load varies slowly, it is possible to use a centralized scheme, in which all measured performance data is brought to one processor, where a heuristic strategy decides a new reassignment. However, this centralized scheme does not scale to very large processor count machines. Not only will centralized data collection and decision making be too slow, but also a single processor may not have the memory to accommodate all the measured data. We are exploring fully distributed strategies, such as neighborhood averaging [4], and extending them to support machines with a large number of processors. Distributed balancers rely on load information from "neighboring" processors, where the neighborhood relationship may be defined by a virtual topology.

We are also exploring the impact of physical topology on such distributed balancers. On very large machines, communication between nearby processors creates less network congestion than communication between distant processors. A dense virtual topology (such as a hypercube, or even more dense graphs) leads to quicker convergence to a balanced state. However, if the physical topology is a 3D grid (as it is likely on extremely large machines), a dense virtual topology may cause communicating objects to migrate to physically distant processors, leading to network congestion.

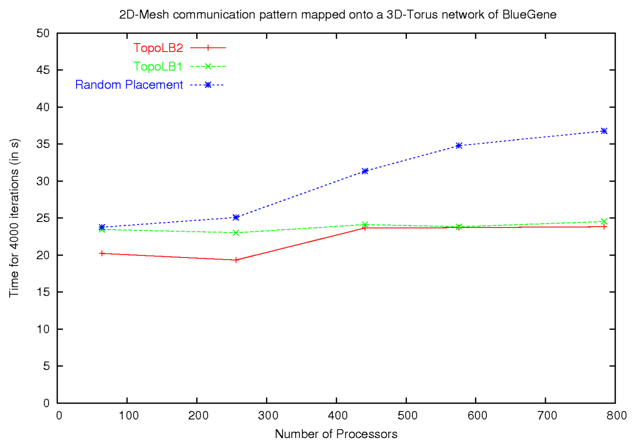

Figure 1: 2D Jacobi performance on Blue Gene/L.

The trade-off between agility, congestion, and communication overhead are all targets of our current analysis and of strategies we are designing to simultaneously optimize both metrics. Under this study, we recently created a new load balancing strategy that considers the number of network hops to be traversed by each communicated byte, and tries to balance the load across processors by distributing objects so that the computational load is balanced and the communication traffic is minimized [5]. Our new topology-based load balancers minimize network congestion by intelligently and adaptively mapping objects to processors during execution of an application.

Figure 1 illustrates the use of topology-based load balancing with execution of a 2D-Jacobi operation on Blue Gene/L. In this example, the application's 2D-mesh structure is mapped into Blue Gene/L's 3D torus interconnect. TopoLB1 and TopoLB2 are task mapping strategies that are applied after the load across processors is balanced. The two mapping schemes are derived according to an analytical model. TopoLB1 uses a first order approximation to the model solution, while TopoLB2 employs a more precise approximation. Both schemes achieve significantly better performance than what would have been achieved by a random mapping strategy, in special for large machine configurations.

We also plan to leverage and extend ongoing work at Illinois on strategies that combine the benefits of centralized and distributed schemes. In these hierarchical, multilevel schemes, a small amount of aggregated information is initially exchanged between all processors. More detailed information is then exchanged within a smaller set of processors. This pattern goes on until detailed per-object information is exchanged only with a processor's immediate neighbors. Compared to most typical fully distributed neighbor-based load balancing strategies, the exchange of global information should allow much better responsiveness and overall balance.

Another resource management aspect that is critical to our migration capability, and consequently to load balancers in general, is memory allocation. We need a consistent support for memory reservation and allocation, in a global form across the processors of a given system, such that threads can migrate with minimal cost. Charm++ has implemented such support with a scheme named "isomalloc", but such scheme cannot be installed on certain systems (e.g. Blue- Gene/L), where alternative schemes have been created. Thus, we plan to develop a new scheme that would be both efficient and portable. We have been interacting with other members of the FastOS community, looking for a permanent solution to this issue.

3. FAULT TOLERANCE

As systems increase in complexity, fault tolerance becomes an increasingly important aspect of system design. Even for today's large machines, the hardware and software fault rates are high enough to impact system utilization. Applications are slowed down by the overhead of periodic checkpoint-restart. Current environments (particularly MPI) frequently lose work in the presence of faults. We started our work on fault tolerance techniques in this project by integrating our previous developments of checkpoint/restart and diskless checkpointing for Charm++ and AMPI codes [6, 7]. Our checkpointing approach leverages the migration capabilities of Charm++, and implements checkpointing as a migration into disk. As an optimization step, we have implemented memory-based checkpointing [7], which is much faster because all the saved state is kept in memory, not in disk. This memory-based scheme may be useful for applications that do not have a large memory footprint.

We are extending our investigation to more ambitious message-logging schemes to tolerate system faults [8]. The major goals of this approach are (a) to provide a fast recovery from faults, without wasting computation done by processors that have not faulted, and (b) to impose low overhead on the forward path (i.e. when there are no failures). In one of the schemes of our investigation, a sender processor keeps a log of all messages that are sent, until the execution reaches a checkpoint where it is safe to discard those messages. If a failure occurs and one of the processors has to be replaced before that checkpoint, all messages sent to the failing processor must be replayed by the senders. We created a prototype version of Charm++ that contains this scheme. Important issues of our current research in this area are the extra communication overhead imposed by maintaining message logs, the checkpointing frequency, optimization opportunities arising from processor virtualization, and others.

Based on the hypothesis that, on current systems, some faults can be predicted, we developed a new technique to proactively handle system faults. By leveraging the migration capabilities of Charm++, we migrate objects from a processor where faults are imminent. To be effective, this approach requires that faults be predictable. Processor manufacturers, such as Intel, are building infrastructure to detect transient errors inside processor chips and notify the OS [9]. Furthermore, recent studies have demonstrated the feasibility of predicting the occurrence of faults in large-scale systems [10] and of using these predictions in system management strategies [11]. Hence, it is possible, under current technology, to act appropriately before a system fault becomes catastrophic to an application. For faults that are not predictable we can revert back to traditional fault recovery schemes, like checkpointing and message logging.

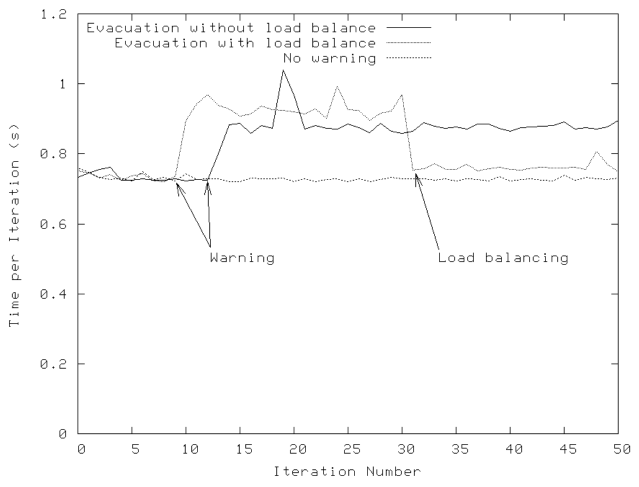

Figure 2: Time per iteration for a 150^3 Sweep3d problem on 32 Xeon processors

Our preliminary tests with this evacuation technique were very encouraging. Based on our initial results, the time to evacuate all objects from a processor depends only on the local dataset size and on the speed of the interconnection network. The scheme scales well with the number of processors, and, in contrast to other techniques, it imposes no overhead when there are no faults. In our tests, we executed real applications, such as the Sweep3d code, on large configurations of a Linux cluster, and simulated faults by sending external signals to tasks running on one or more processors.

One critical issue in our evacuation scheme is the need to apply some form of load balancing after a processor is evacuated, since the surviving processors may become unbalanced due to the migration of load from the failing processor. For some applications, any generic load balancer may be sufficient in this phase; for others, where communication locality is a sensitive issue, we developed a special form of load balancer which preserves the proximity between objects that communicate frequently. As an example of this technique, Figure 2 shows the iteration times for Sweep3d with a fault in one of the processors. In one of the executions, load balance is applied after the evacuation. After load balance is applied, the loss in performance becomes proportional to the loss in computational power due to the failure.

4. GLOBAL SYSTEM MANAGEMENT

The problem of OS interference has recently received considerable interest from the HPC community. Studies have concluded that asynchronous events within the operating system such as timer decrement interrupts or daemon activity can have a cascading negative impact on parallel application performance [12, 2, 13]. We address this problem with parallel aware scheduling. In our solution, we went a step further in synchronizing the various Linux images running in a Blue Gene/L machine. We leverage the fast hardware barrier network of Blue Gene/L, to synchronize an entire 64-rack system to within less than 1 microsecond. We note that we synchronize the internal operations of the various Linux images to within 1 microsecond, so that they all program their timer interrupts at the same time and take them

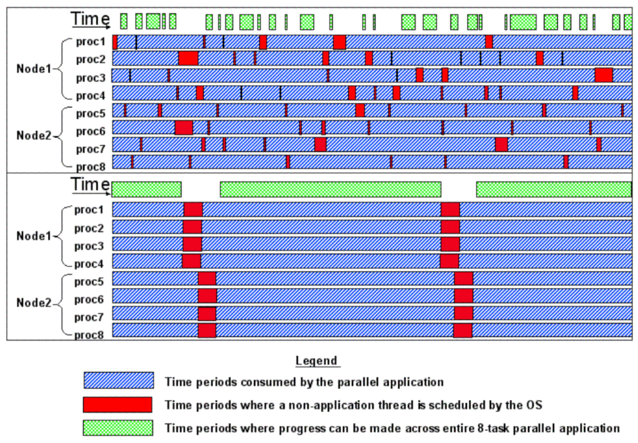
Time
Node1
proc1
proc2
proc3
proc4
Node2
proc5
proc6
proc7
proc8
Time
Node1
proc1
proc2
proc3
proc4
Node2
proc5
proc6
proc7
proc8
Legend
Time periods consumed by the parallel application
Time periods where a non-application thread is scheduled by the OS
Time periods where progress can be made across entire 8-task parallel application

Figure 3: Parallel Aware Scheduling

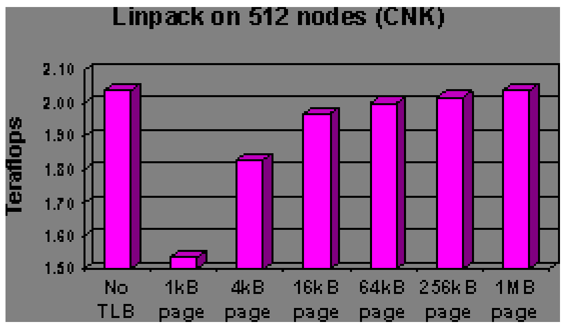

Figure 4: Linpack execution time on Blue Gene/L with different page sizes. The performance with static TLB (leftmost bar) can be recovered with large enough page sizes.

at the same time (within 1 microsecond). So far, we can only synchronize Linux images running on I/O nodes (because compute nodes do no yet run Linux). That has been enough to verify the validity of the approach. Using the fast hardware barrier network, we mitigate the impact of random short-lived interruptions (such as timer decrement processing and periodic daemon activity) by a parallel aware scheduling designed to globally coordinate large processor count SPMD bulk-synchronous programming styles.

Our studies for this parallel aware scheduling scheme indicate improved performance of fine-grained synchronous collective activities such as barriers and reductions. This is accomplished by cycling the process priority of the tasks between a favored and unfavored value at periodic intervals across the entire program's working set of processors. The actual priorities, favored priority duty cycle, and adjustment period are obtained at job launch. Setting a process priority to a fixed favored priority value causes the operating system to assign a processor to this process (assuming there are no higher priority processes already running), and hence the operating system puts the application task into a running state. Similarly, setting a process priority to a fixed unfavored priority causes the operating system to assign some other process to the processor if there are processes with more favored priority waiting to be run.

Figure 3 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. [14]

Parallel aware scheduling techniques have demonstrated impressive performance improvements for full featured OS. Measurements with the Miranda parallel instability code [15] indicate that parallel aware scheduling across the machine can dramatically improve variability in runtimes (standard deviation decreased from 108.45 seconds to 5.45 seconds) and total wallclock runtime (mean decreased from 452.52 seconds to 254.45 seconds). These results were obtained on an AIX Power5 based system with an IBM HPS interconnect.

5. OPERATING SYSTEM OPTIONS

In this section, we discuss two issues that have a significant impact on the scalability of operating systems for large parallel machines: memory management and file systems.

Modern processors perform memory management through translation lookaside buffers (TLBs). The entries in the TLBs are controlled by the operating system. TLB misses during execution of a program can be a source of severe application performance degradation.

To allow data sharing both within an application and across applications in a large parallel system, it is common to have all compute nodes in a machine sharing the same file system. Current parallel file systems have been demonstrated to scale to a few thousand compute nodes. It is yet unknown how they behave, or even if they work, at the next level of scalability, from ten thousand to a hundred thousand compute nodes.

In the Blue Gene/L supercomputer, these issues were addressed through a custom lightweight kernel for the compute nodes [16]. Familiar programming interfaces are provided through GNU glibc runtime support and basic file I/O support. The compute nodes implement much of the POSIX standards but are not fully POSIX compliant. While the compute node kernel provides an environment similar to that of many other operating systems, it is different in some key areas.

The compute node kernel simplifies memory management by providing a simple, flat, fixed-size address space without paging. Since there are no virtual addresses there is no address translation to impact the computational workload. Memory is statically mapped through the TLB to avoid any misses that would steal CPU cycles from the computational workload.

Figure 4 illustrates the impact of TLB management on the execution of the Linpack benchmark on 512 nodes of Blue Gene/L. The left most bar shows the performance when the compute node kernel with static TLB is used. The other bars show the performance of dynamic TLB management with different page sizes. We see that (1) the page size of

4 KB used in most Linux systems today causes severe performance degradation and (2) performance can be recovered using large page sizes of 64 or 256 KB. However, benchmarks with truly random memory access (NAS IS or HPCC RandomAccesss) see significant performance impact even with large page sizes.

The Blue Gene/L compute node kernel provides a set of system calls that deliver basic file I/O functionality. These I/O operations are not executed by the compute nodes directly. The I/O requests are shipped to the I/O node associated to that compute node for execution. The results are shipped back to the compute nodes. Since the I/O nodes run Linux, Blue Gene's compute nodes can rely on Linux's I/O infrastructure to correctly handle the I/O requests. There are many compute nodes associated with a single I/O node. This multiplexing of I/O reduces scaling issues. The largest Blue Gene/L system today has 64K compute nodes redirecting I/O via only 1024 I/O nodes.

Despite its disadvantages for scalability, there are reasons to run a full operating system (i.e., Linux) on compute nodes. The lightweight kernel of Blue Gene/L lacks key functionality that is increasingly important to a growing number of HPC applications, including general process and thread creation, full server sockets, shared memory segments, and memory mapped files.

Our goal in the HPC-Colony project is to address the scaling issues discussed here (memory management and file system), and others, so that we can run a full Linux operating system in machines with hundreds of thousands of processors, like Blue Gene/L.

6. RELATED WORK

Most of the dynamic load balancing strategies can be classified as centralized [17, 18] or fully distributed methods [19, 20]. In centralized strategies, a dedicated "central" processor gathers global information about the state of the entire machine and uses it to make global load balancing decisions. On the other hand, in a fully distributed strategy, each processor exchanges state information with other processors in its neighborhood. Fully distributed strategies have been proposed typically in the context of non-iterative tasks. These include the neighborhood averaging scheme (ACWN) [21], and a set of strategies proposed in [22]. In [22], several distributed and hierarchical load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model (GM), Dimension Exchange Method (DEM) and a Hierarchical Balancing Method (HBM).

The load balancing strategies cited above are applicable for problems with continuous creation of tasks, such as space-state searching and branch-and-bound problems. They are less suited for scientific applications with an iterative scheme. Such iterative applications require load balancing that moves data and tasks during the computation based on the most recent load. Examples of this scheme include DRAMA [23], Zoltan [24] and Chombo [25].

Various strategies for balancing the load on specific topologies and/or specific task graphs have been studied. Ercal et al [26] provide a divide and conquer solution in the context of a hypercube topology called Allocation by Recursive Mincut or ARM. Bianchini and Shen [27] consider mesh network topology. Fang, Li and Ni [28] study the problem of 2-D convolution on mesh, hypercube and shuffle exchange topologies only.

The techniques for fault tolerance in message-passing environments can be broadly divided in two classes: checkpointing schemes and message-logging schemes. In checkpoint-based techniques, the application status is periodically saved to stable storage, and recovered when a failure occurs. Representative examples of this class are CoCheck [29], Starfish [30] and Clip [31]. Meanwhile, in message-logging techniques, the central idea is to retransmit one or more messages when a system failure is detected. Message-logging can be optimistic [32], pessimistic [33, 34, 35, 36] or causal [37]. In all of these proposed fault-tolerant solutions, some corrective action is taken in reaction to a detected failure. In contrast, with our proactive approach [38], fault handling consists in migrating a task from a processor where failures are imminent. Thus, no recovery is needed.

The scheduling of processes onto processors of a parallel machine has been and continues to be an important and challenging area of research. Previous research activities [39, 40] have not resulted in capabilities available to production super-computing facilities in which a single job consisting of thousands of cooperating processes occupies a dedicated portion of the computing complex. Since the machines are typically SMP nodes, a node is assigned as many processes as there are processors on the node, and each process acts as if it has exclusive use of the processor. In this environment, fair share CPU scheduling and demand-based co-scheduling required for networks of workstations (NOWs) are not necessary. Other work has focused on communication scheduling strategies, or the elimination of daemon processes [41]. Our work focuses on collaborative scheduling of the job processes both within a node and across nodes, so that fine grain synchronization activities can proceed without having to experience the overhead of making scheduling requests.

7. CONCLUSION, STATUS AND PLANS

Traditional operating systems have severe limitations when applied to large scale machines with many thousands of processors. In our HPC-Colony project, described in this article, we are addressing some of those limitations to provide new features and capabilities to operating and runtime systems for that class of machines. In particular, we have been working in the areas of resource management, fault tolerance and global system management. We are using Blue Gene/L as a test platform, and are developing new strategies both at the system and at the application level.

In resource management, we have extended our Charm++ infrastructure with additional load balancing schemes. Our recent results have shown that the topological information can be efficiently explored for load balancing. Meanwhile, in fault tolerance, our preliminary results with the use of a proactive migration approach have indicated that this technique can enable the application to tolerate failures while avoiding the overhead of extra computation due to those failures.

Further demonstrations of the mitigating effect of large memory page size on TLB miss are planned for the Blue Gene/L system. Preliminary tests with Linpack have shown encouraging results. Investigation of parallel I/O strategies on Blue Gene/L is also under consideration.

During the upcoming year, we plan to demonstrate booting a Linux kernel on Blue Gene/L compute nodes, and to continue our integration of our fault tolerance work based on message logging, so that it becomes a regular component in the Charm++ distribution. In addition, we intend to drive our fault evacuation mechanism with decisions based on real fault indicators that will be available with the upcoming PAPI-4 toolkit [42]. In the resource management area, we will continue to develop and deploy the hybrid schemes for load balancing. We will expand our studies of parallel aware scheduling to include Blue Gene/L compute nodes. We will also continue to pursue a permanent solution to the memory reservation and allocation issue.

Acknowledgments
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's Office of Science under the FastOS program [1] . Supplemental funding for people and/or machine time provided by IBM, LLNL, and the University of Illinois at Urbana-Champaign. Part of this work was supported by the National Center for Supercomputing Applications under grant CCR050006S and utilized the NCSA tungsten system.

8. REFERENCES

[1] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, (College Station, Texas), pp. 306–322, October 2003.

[2] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of SC 2002*, (Baltimore, MD), September 2002.

[3] R. K. Brunner and L. V. Kalé, "Handling application-induced load imbalance using parallel objects," in *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pp. 167–181, World Scientific Publishing, 2000.

[4] G. Zheng, *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[5] T. Agarwal, A. Sharma, and L. V. Kalé, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.

[6] C. Huang, "System support for checkpoint and restart of charm++ and ampi applications," Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[7] G. Zheng, L. Shi, and L. V. Kalé, "Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *2004 IEEE International Conference on Cluster Computing*, (San Dieago, CA), September 2004.

[8] S. Chakravorty and L. V. Kale, "A fault tolerant protocol for massively parallel machines," in *FTPDS Workshop for IPDPS 2004*, IEEE Press, 2004.

[9] P. Apparao and G. Averill, "Firmware-based platform reliability." Intel white paper, October 2004.

[10] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *Proceedings og the ACM SIGKDD, Intl. Conf. on Knowledge Discovery Data Mining*, pp. 426–435, August 2003.

[11] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam, "Fault-aware job scheduling for BlueGene/L systems," Tech. Rep. RC23077, IBM Research, January (2004).

[12] T. Jones, J. Fier, and L. Brenner, "Observed impacts of operating systems on the scalability of applications," Tech. Rep. UCRL-MI-202629, Lawrence Livermore National Laboratory, March 2003.

[13] P. Terry, A. Shan, and P. Huttunen, "Improving application performance on hpc systems with process synchronization," *Linux Journal*, pp. 68–73, November 2004.

[14] T. Jones, S. Dawson, R. Neely, W. Tuel, L.Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, , and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Proceedings of Supercomputing'03*, (Phoenix, AZ), November 2003.

[15] A. W. Cook and W. H. Cabot, "Large scale simulations with miranda on Blue Gene/L," Tech. Rep. UCRL-PRES-200327, Lawrence Livermore National Laboratory, 2003.

[16] J. Moreira et al, "Blue Gene/L programming and operating environment," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 367–376, 2005.

[17] Y.-C. Chow and W. H. Kohler, "Models for dynamic load balancing in homogeneous multiple processor systems," in *IEEE Transactions on Computers*, vol. c-36, pp. 667–679, May 1982.

[18] L. M. Ni and K. Hwang, "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," in *IEEE Trans. on Software Eng.*, vol. SE-11, 1985.

[19] A. Corradi, L. Leonardi, and F. Zambonelli, "Diffusive Load Balancing Policies for Dynamic Applications," in *IEEE Concurrency*, pp. 7(1):22–31, 1999.

[20] A. Ha'c and X. Jin, "Dynamic Load Balancing in Distributed System Using a Decentralized Algorithm," in *Proc. of 7-th Intl. Conf. on Distributed Computing Systems*, April 1987.

[21] A. Sinha and L. Kalé, "A load balancing strategy for prioritized execution of tasks," in *International Parallel Processing Symposium*, (New Port Beach, CA.), pp. 230–237, April 1993.

[22] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, September 1993.

[23] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J.-M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw, "Dynamic load balancing of finite element applications with the DRAMA Library," in *Applied Math. Modeling*, vol. 25, pp. 83–98, 2000.

[1] `http://www.cs.unm.edu/~fastos`

[24] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New challenges in dynamic load balancing," *Appl. Numer. Math.*, vol. 52, no. 2–3, pp. 133–152, 2005.

[25] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo Software Package for AMR Applications Design Document," 2003. http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf.

[26] F. Ercal, J. Ramanujam, and P. Sadayappan, "Task allocation onto a hypercube by recursive mincut bipartitioning," in *Proceedings of the third conference on Hypercube concurrent compu ters and applications*, (New York, NY, USA), pp. 210–221, ACM Press, 1988.

[27] R. P. B. Jr. and J. P. Shen, "Interprocessor traffic scheduling algorithm for multiple-processor networks.," *IEEE Trans. Computers*, vol. 36, no. 4, pp. 396–409, 1987.

[28] Z. Fang, X. Li, and L. M. Ni, "On the communication complexity of generalized 2-d convolution on array processors," *IEEE Trans. Comput.*, vol. 38, no. 2, pp. 184–194, 1989.

[29] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium*, pp. 526–531, 1996.

[30] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations," *Cluster Computing*, vol. 6, pp. 227–236, July 2003.

[31] Y. Chen, J. S. Plank, and K. Li, "Clip: A checkpointing tool for message-passing parallel programs," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pp. 1–11, 1997.

[32] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 204–226, 1985.

[33] G. E. Fagg and J. J. Dongarra, "Building and using a fault-tolerant MPI implementation," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 353–361, 2004.

[34] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam, Y. Dandass, and M. Apte, "Mpi/fttm: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing," in *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, p. 26, IEEE Computer Society, 2001.

[35] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, "MPI-FT: Portable fault tolerance scheme for MPI," *Parallel Processing Letters*, vol. 10, no. 4, pp. 371–382, 2000.

[36] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging programming via processor virtualization," in *Proceedings of Supercomputing'03*, (Phoenix, AZ), November 2003.

[37] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526–531, 1992.

[38] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in MPI applications via task migration," 2006. Submitted to publication.

[39] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Third International Conference on Distributed Computing Systems*, pp. 22–30, May 1982.

[40] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "Dynamic co-scheduling on workstation clusters," Tech. Rep. 1997-017, Digital Systems Research Center, March 1997.

[41] F. Petrini, D.J.Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proceedings of Supercomputing'03*, (Phoenix, AZ), November 2003.

[42] K. London, S. Moore, D. Terpstra, and J. Dongarra, "Support for simultaneous multiple substrate performance monitoring," October 2005. Poster Session at LACSI Symposium 2005.