RC24664 (W0810-052) October 9, 2008
Other

IBM Research Report

Asynchronous Task Dispatch for High Throughput Computing for the eServer IBM Blue Gene® Supercomputer

Amanda Peters[1], Alan King[2], Tom Budnik[1], Pat McCarthy[1], Paul Michaud[3], Mike Mundy[1], James Sexton[2], Greg Stewart[1]

[1]IBM Systems and Technology Group
Rochester, MN 55901

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[3]IBM Software Group
Houston, TX 77056

Asynchronous Task Dispatch for High Throughput Computing for the eServer IBM Blue Gene® Supercomputer

Amanda Peters[1], Alan King[2], Tom Budnik[1], Pat McCarthy[1],
Paul Michaud[3], Mike Mundy[1], James Sexton[2], Greg Stewart[1]

[1]IBM Systems and Technology Group
Rochester, MN 55901
{apeters, tbudnik, pjmccart, mmundy, gregstew}@us.ibm.com

[2]IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{kingaj, sextonjc} @us.ibm.com

[3]IBM Software Group
Houston, TX 77056
{pkmichaud} @us.ibm.com

High Throughput Computing (HTC) environments strive "to provide large amounts of processing capacity to customers over long periods of time by exploiting existing resources on the network" according to Basney and Livny [1]. A single Blue Gene/L rack can provide thousands of CPU resources into HTC environments. This paper discusses the implementation of an asynchronous task dispatch system that exploits a recently released feature of the Blue Gene/L control system – called HTC mode – and presents data on experimental runs consisting of the asynchronous submission of multiple batches of thousands of tasks for financial workloads. The methodology developed here demonstrates how systems with very large processor counts and light-weight kernels can be configured to deliver capacity computing at the individual processor level in future petascale computing systems.

1. Introduction

The goal of High Throughput Computing (HTC) as stated by Basney and Livny [2] is "to provide large amounts of processing capacity to customers over long periods of time by exploiting existing resources on the network". A single Blue Gene/L rack has thousands of processors, so in principle, a Blue Gene/L computer has the potential to vastly increase the available resources.

Blue Gene is the result of an IBM supercomputing project begun over five years ago, dedicated to building a new family of supercomputers optimized for bandwidth, while minimizing electrical power and floor space requirements. The design of the Blue Gene/L supercomputer was focused on simplicity and scalability. [3] These aims came from the needs of the initial target applications for large scale computer simulations of biological processes like protein folding. These applications were developed using the High Performance Computing (HPC) model and are predicated on the use of cooperating nodes in a synchronous runtime and are often utilizing MPI. This required the development of a massively parallel supercomputer fine tuned for HPC applications. The major challenge fell in the mapping of a fixed size simulation onto such a massively parallel system. [4]

However, as Blue Gene/L moves from the domain of a research-only vehicle to that of IBM product offering, new user requirements are introduced as the fields of usage broaden.

In the finance industry, focus is on both applying computational power to increasingly complex workloads, and also on reliability and fault tolerance [5]. Work loads such as risk management and portfolio analysis, require large amounts of asynchronous calculations as well as high reliability and fast recovery.

In 2006, the responses we received during a year-long campaign of finance industry briefings on the capabilities of the Blue Gene supercomputers gave us a view of the deep concerns of the industry on the tradeoffs involved in a business model that requires growth in computing capacity on the order of 30% annually.

The IBM term for the reliability of data centers is Reliability, Availability, and Serviceability (RAS). RAS issues dominate the concerns of IT managers in the financial industry, since any interruption in the processing of risk management workloads may affect the ability of a financial firm to manage their trading risks in the marketplace. There are two basic workflows involved. First, daily risk management reports are generated during a tightly choreographed time window between close of trading on one day and market reopen on the next day. These risk management reports are used by senior management to set high-level trading policies, and by traders to set parameters for the numerical processes used to evaluate positions and trades. Second, an increasing fraction of trading activity is automated, especially including trades performed to manage portfolio risk.

Competition places enormous pressure on financial industry IT departments to reduce costs by automating trading of securities and to support the increasingly complex calculations for more profitable new generations of securities. As IT managers deploy additional computers to meet the demands of their businesses, they are deeply familiar with the fact that the more programs that are running, the higher is the probability that one will fail during any particular time window.

The solution adopted by the industry to the RAS and performance tradeoff is to deploy large numbers of homogeneous single or dual processor computers in form factors that can be densely stacked in data centers and networked into a grid. This grid solution fits the workload type, which consists mostly of large numbers of "pleasantly parallel" operations that can easily be distributed onto a grid, and also contributes to a RAS approach based on the capability to shift workload from a failed device to any of a large number of identical devices. Consequently, an IT department in a large bank currently is typically managing thousands of single or dual processor blade-type computers, and installing new capacity at a 30% annual rate.

The reaction of the financial industry to the Blue Gene design was that while the machine can conveniently deliver a very large number of processors, it must do so in such a way as to not increase the risk of failure or reduce the ability to recover.

The designers of the Blue Gene computer did contemplate the fact that a system with thousands of processors is much more likely to suffer a failure during the execution of a long-running workload. Consequently, the control system has protocols to detect node failures and to route around them if possible, and the nodes themselves are manufactured to be robust to mechanical failures and easy to replace. While customer data shows that a Blue Gene computer has a far-lower mean time to failure than an equivalent grid of computers (largely because Blue Gene nodes have no disk) the operating mode of the Blue Gene system was designed for HPC workloads, in which a large number of processors are harnessed using MPI to perform a more-or-less single operation. While it is possible to develop software to mimic a grid-type environment using MPI, this would not meet the RAS requirements of financial workloads. This is primarily due to the fact that when an HPC program fails the entire MPI partition will fail, bringing down all the healthy nodes and jobs as well.

This paper discusses the first phase of the enablement of a model for High Throughput Computing (HTC) on Blue Gene/L. Our implementation enables the support of applications that run asynchronously and increases the availability of the system. In this paper we report on experiments demonstrating how HTC mode can reliably supply many thousands of individual processors into a grid pool. We developed a task dispatch program, and tested its performance on independent task dispatch to over 8000 nodes with a latency overhead of around 1 millisecond per task. Dispatcher performance scales perfectly from 2 racks to 4 racks, with declining overheads as more processors are added. In addition, each process runs independently of each other in HTC mode, and so is resilient to soft-failures caused by abnormal application exits.

HTC mode of the Blue Gene/L operating system is reviewed in Section 2. Section 3 covers the implementation of a basic asynchronous dispatch subsystem for HTC mode of operation. Section 4 discusses the results of some large scale experiments with the task dispatcher. Finally, Section 5 summarizes our experiences with HTC mode on Blue Gene/L, its advantages and disadvantages, and discusses some possible extensions.

2. HTC mode

In HTC mode, the compute nodes in a partition are running independent programs that do not communicate with each other. A launcher program (running on a compute node) requests work from a dispatcher that is running on a remote system that is connected to the functional network. Based on information provided by the dispatcher, the launcher program spawns a worker program that performs some task.

When running in HTC mode there are two basic operational differences over the default HPC mode. First, after the worker program completes, the launcher program is reloaded so it can handle additional work requests. The launcher program is cached so that it can be reloaded onto the node automatically by the control system. Second, if a compute node encounters a soft error, such as a parity error, the entire partition is not terminated. In HPC mode, the processors are cooperating on one task relying on the supporting networks for message passing so a single node failure will cause the entire job to halt and the whole partition will need to be rebooted. In HTC mode, the processors are running asynchronous independent tasks without inter-nodal communication, so one node failure does not need to bring the entire partition down. Rather, the control system attempts to restart the single compute node while other compute nodes continue to operate. The control system will poll hardware on a regular interval looking for compute nodes in a reset state. If a failed node is discovered and the node failure is not due to a network hardware error, a software reboot is attempted to recover the compute node. A diagram of the Blue Gene/L Compute Chip is shown below. If the failure falls on the chip anywhere outside the network dedicated hardware (circled), a reboot will be attempted.

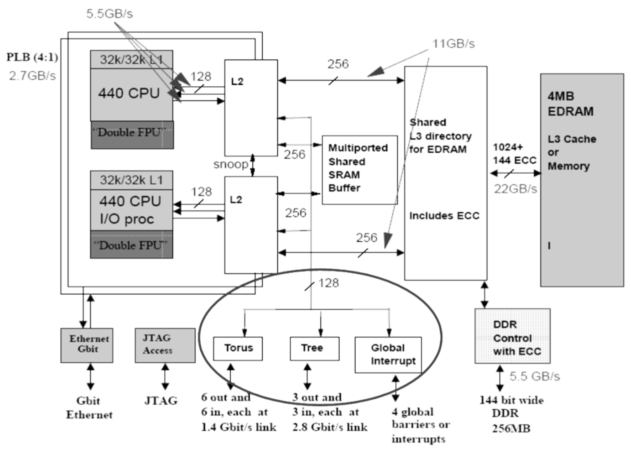

Figure 1. Overview of the Blue Gene/L Compute Chip (BLC) [3]

On a remote system that is connected to the functional network, which could be a Service Node or a Front End Node, there is a dispatcher that manages a queue of jobs to be run. There is a client/server relationship between the launcher program and the dispatcher program. After the launcher program is started on a compute node, it connects back to the dispatcher and indicates it is ready to receive work-requests. When the dispatcher has a job for the launcher, it responds to the work-request by sending the launcher program the job related information (such as the name of the worker program executable, arguments, and environment variables). The launcher program then spawns off the worker program. When the worker program completes, the launcher is reloaded and repeats the process of requesting work from the dispatcher.

2.1 Applications perspective: MPI or HTC

As previously stated, the Blue Gene/L architecture targeted MPI applications for optimal execution. These applications are characterized as Single Program Multiple Data (SPMD) with synchronized communication and execution. The tasks in an MPI program are cooperating to solve a single problem. Because of this close cooperation, a failure in a single node, software or hardware, requires the termination of all nodes.

HTC applications have different characteristics. The code executing on each node is independent of work being done on another node and communication between nodes is not required. At any specific time, each node is solving its own problem. As a result, a failure on a single node, software or hardware, will not necessitate the termination of all nodes. Initially, in order to run these applications on a Blue Gene/L, a port to the MPI programming model was required. This was done successfully for several applications, but was not an optimal solution in some cases. Some MPI applications may benefit by being ported to the HTC model. In particular, some pleasantly parallel MPI applications may be good candidates for HTC mode because they do not require communication between the nodes and the failure of one node does not invalidate the work being done on other nodes. A key advantage of the MPI model is a reduction of extraneous booking by the application. An MPI program will be coded to handle data distribution and minimize IO by having all IO done by one node (typically rank 0) and distribute the work to the other MPI ranks. When running in HTC mode, the application data needs to be manually split up and distributed to the nodes. This porting effort may be justified to achieve better application reliability and throughput than could be achieved with an MPI model.

2.2 Restarting the launcher program

In the default HPC mode, when a program ends on the compute node the Compute Node Kernel (CNK) sends a message to the IO node that reports how the node ended. The message indicates if the program ended normally or by signal and the exit value or signal number, respectively. The IO node then forwards the message to the control system. When the control system has received messages for all of the compute nodes, it then ends the job. In HTC mode, CNK handles a program ending differently depending on what program ended. CNK records the path of the program that is first submitted with the job which is the launcher program. When a program other than the launcher program (or the worker program) ends, CNK records the exit status of the worker program, and then reloads and restarts the launcher program. If the worker program ended by signal, CNK generates an event to record the signal number that ended the program. If the worker program ended normally, no information is logged.

The launcher program can retrieve the exit status of the worker program using a CNK system call. Since no message is sent to the control system indicating that a program ended, the job continues running. The effect is to have a continually running program on the compute nodes. To reduce the load on the file system, the launcher program is cached in memory on the IO node. When CNK requests to reload the launcher program it does not need to be read from the

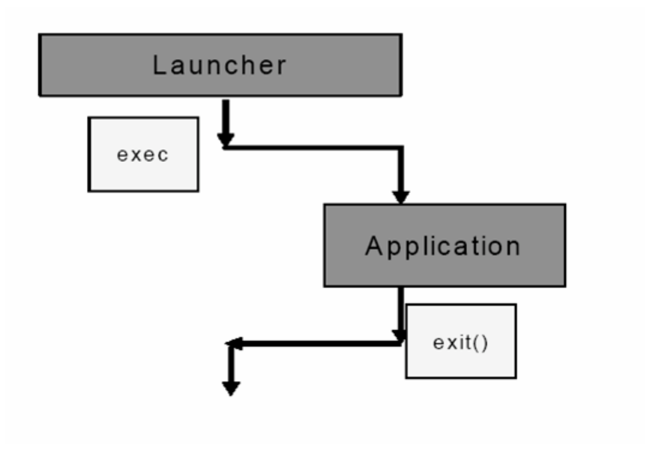

Figure 2. Launcher run sequence. [8]

file system but can be sent directly to the compute node from memory. Since the launcher is typically a small executable it does not require much additional memory to cache it.
After the launcher program ends, CNK reports that a program ended to the control system, as it does for HPC mode. This allows the launcher program to cleanly end on the compute node and for the control system to end the job.

2.3 IO considerations

Running in HTC mode changes the IO patterns compared to HPC mode. When a program reads and writes to the file system it is typically done with small buffer sizes. While the administrator can configure the buffer size, the most common sizes are 256KB or 512KB. When running in HTC mode, loading the worker program requires reading the complete executable into memory and sending it over the collective network to a compute node. An executable is at least several megabytes and can be many megabytes.

The Blue Gene/L design was not optimized for applications that load large files. Internal tracing showed that the greatest amount of overhead was spent sending the executable over the collective network. It took 0.001131 seconds to read a 5MB executable from the file system, but it took 1.535258 seconds for the IO node to inject the packets into the collective network. This includes the time to read the packet for the next message but it accounts for most of the time.

To achieve the best IO performance a low compute node to IO node ratio is preferred. Blue Gene/L supports compute node to IO node ratios of 8-1, 32-1, and 64-1. The additional IO demands of HTC mode from loading executables is best handled by a system with a 8-1 compute node to IO node ratio. The smaller number of compute nodes per IO node allows for the best utilization of the collective network. When running in HTC mode the variability in the length of work requests does allow computation and IO to be overlapped.

3. Asynchronous task dispatch subsystem

This section covers an implementation of an asynchronous task dispatch subsystem. The dispatch subsystem was implemented in C and is available as an example from the online Blue Gene/L Technical Knowledge Base.

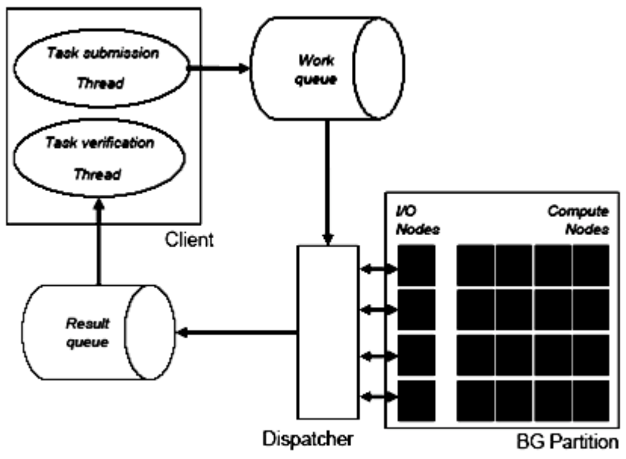

Figure 3. Asynchronous task dispatch subsystem.

The diagram in Figure 2 shows the logical relationships. Our design is motivated by the principle that the client is ultimately responsible for ensuring successful completion of the job, following [2].

3.1 Clients

The client implements a task submission thread that publishes task submission messages onto a work queue, and a task verification thread that listens for task completion messages. Clients are assumed to be responsible citizens of the dispatch system, as in the ”Ethernet” analogy discussed in [2]. When the dispatch system informs the client that the task has terminated, and optionally supplies an exit status, the client is then responsible for resolving task completion status and for taking actions, including re-launching tasks. The client is expected to behave itself, "Ethernet” style, in such a way as to maintain the stability and the fair allocation of the dispatch system resources.

3.2 Message Queues

The design is based on publishing and subscribing to message queues. Clients publish task submission messages onto a single work queue. Dispatcher programs subscribe to the work queue and process task submission messages. Clients subscribe to task completion messages.

Messages consist of text data comprising the work to be performed, a job identifier, a task identifier, and a message type. Job identifiers are generated by the client process and are required to be globally unique. Task identifiers are unique within the job session. The message type field for task submission messages is used by the dispatcher to distinguish work of high priority versus work of normal priority. Responsibility for reliable message delivery belongs to the message queuing system.

3.3 Dispatcher program

The dispatcher program responsibilities are as follows. First it pulls a task submission message off the work queue. Then it waits on a socket for a launcher connection, and reads the launcher id from the socket. It writes the task into the socket, and the association between task and launcher is stored in a table. The table stores the last task dispatched to the launcher program, so this connection is an indication that the last task has completed and the task completion message can be published back to the client.

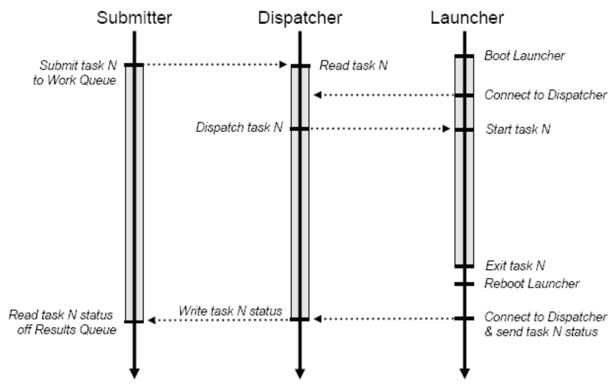

Figure 4. Interaction diagram for asynchronous HTC dispatcher program. [8]

The interaction diagram for the duration of task N is depicted in Figure 3. The intention of this design is to optimize the launcher program. The dispatcher program spends little time between connect and dispatch, so latency volatily would mainly be due to the waiting-time for dispatcher program connections. Publishing the task messages closer to the compute nodes will go a long way to eliminating launcher program latencies.

One might think that the launcher program would preferentially publish the completion message. But this mechanism is not more efficient than the implementation just described, since any publication involves a network transaction. The first thing the launcher program does after rebooting is connect to the dispatcher program, so it may as well pass the completion information to the dispatcher program at that time. Faster methods of spawning completion messages will require direct connections between the executing task and the client.

This still leaves open the question of how the task completion message is generated. The solution we implemented was to have the dispatcher generate the task completion message. To assist task status resolution, the CNK stores the exit status of the last running process in a buffer. After launcher program restart, the contents of this buffer can be written to the dispatcher and stored in the task completion message.

3.4 Launcher program

The launcher program is intentionally kept very simple. Arguments to the launcher program describe a socket connection to the dispatcher. When the launcher program starts, it connects to this socket, writes its identity into the socket, and waits for a task message. Upon receipt of the task message the launcher parses the message and calls the execve system call to execute the task. When the task exits (for whatever reason), the CNK restarts the launcher program again. The launcher program is not a container for the application, so no matter what happens to the application, the launcher program will not fail to restart.

4. Experimental results

This section discusses our experiments driving computational workloads through our asynchronous task dispatch subsystem. The test workloads are derived from a financial problem set. Each test is configured to reflect a realistic use case.

4.1 Financial Portfolio Risk Calculations

Portfolio risk calculations are run by broker-dealers, market-makers, and investment banks to meet financial regulatory reporting requirements and to feed automated trading programs. A risk calculation integrates portfolio cash flows over probability distributions that are calibrated to market data and/or stressed by market and economic scenarios. These integrations are almost always performed by Monte Carlo simulation.

Risk calculation workloads for regulatory reporting are typically run in a carefully synchronized overnight batch run, but there is an emerging competitive trend to perform risk calculations in "real-time" to support automated trading programs. Batch workloads consist of large numbers of tasks that are distributed on local grids using commercial or home-grown grid services middleware. A typical batch workload configuration allocates around 45 minutes of work to a single node. The executables are generally large – around 50MB – because they are built to handle any conceivable portfolio and are typically pricing several thousand instruments. A node running such a workload would be unavailable for additional work during the computation, so this model of workload distribution would not be suitable for a real-time environment. HTC mode on a Blue Gene/L offers fast access to very large numbers of nodes, so we focused on repackaging finance workloads into smaller units. When a single instrument simulation is packaged into an executable its size may be quite small. For example, an implementation of a stochastic mesh method to price American options, due to Broadie and Glasserman [7], has an executable size of 1.5MB and has a run time that is approximately 28 seconds.

For our experimentation with HTC mode we configured the options pricing method to be a function that accepts a single input string and writes a single output string. This function was wrapped into a main program that implements a socket connect-write-receive protocol similar to that of the launcher, and contains a control loop that responds to sequences of task messages from a client. The program essentially implements a stateful session under the control of the client.

In the experiments we perform, the client first requests the dispatcher to launch the sessions on the Blue Gene nodes. Each session opens a connection to the dispatcher and identifies itself as a session bound to a client id. The dispatcher then places the session into the list of available resources for task messages from that client. Sessions are terminated by the client.

4.1.1 Results

Experiments were run on the BGW computer at IBM's Thomas J Watson Research Center. This is a Blue Gene/L system with 1024 dual-core nodes, and one IO Node for every 32 Compute Nodes. The HTC partitions were booted in Virtual Node mode. A single rack will boot 2048 launchers, one per core. Two racks will boot 4096 launchers, and four racks 8192. When running on Blue Gene we have exclusive access to the HTC partition, however the Front End Node that hosts the dispatcher is typically a busy machine.

#Sessions	#Tasks	Load Time (s)	Overhead/ Task (ms)
256	1024	73.44	0.0182
512	2048	116.76	0.0104
1024	4096	111.66	0.0071
2048	8192	193.5	0.005
256	2048	60.31	0.0169
512	4096	77.12	0.0108
1024	8192	120.93	0.0058
2048	16384	166.46	0.0035

Table 1. The results of 4 and 8 tasks per session. Both sets were run on 2048 processors (one rack) with 8 dispatchers.

Each row in the table represents a single run. Each run specifies the number of sessions to start, followed by a list of tasks of a given size. Tasks are distributed by the dispatcher to the sessions on a first-come first-served basis; there is no requirement to synchronize so that each node performs a given number of tasks. The client blocks until all sessions connect, so the per-task timings are conservative. (Better elapsed times would be observed if task dispatch started after the first session connects.)

The first set of experiments, displayed in Table 1, was run on sub-rack numbers of partitions. The first batch submits an average of 4 tasks per session; the second submits 8 tasks per session. Each batch consists of runs of 256, 512, 1024, and 2048 sessions. The first interesting point to note is the high variability of the time to load the executables. Load time is measured at the client, from beginning of session dispatch to the time when all sessions requested have connected. It is clear that these timings are not scaling particularly well for sub-rack sized partitions. The second interesting point to note is the predictability of the overhead per task. Overhead is computed by subtracting load time from the elapsed time for the entire run. Overhead per task is computed simply by dividing the number of tasks into the overhead. It is clear that the overhead per task declines as the numbers of tasks increases becomes quite predictable. For example the overhead per task for 8192 tasks is approximately 5 or 6 milliseconds per task whether there are 1024 or 2048 sessions.

The final sets of experiments are run on multiple rack sized partitions. The first run is on two racks, or 4096 processors, and the second is on four racks, or 8192 processors. Both runs are shown in Table 2. The runs show practically identical time to load. Overhead per task still shows a tendency to decline with increasing numbers of tasks; perhaps this is due to some overhead amortization in the initialization of the client sessions.

#Disp	#Sess.	#Tasks	Load	Overhead
16	4096	16384	182.8s	0.0015ms
32	8192	32796	182.85s	0.0011ms

Table 2. The results with 16 and 32 dispatchers, and partition sizes of 4096 and 8192 processors.

The bottom line is that on a very large run of 32,796 tasks on 8192 processors, we obtain an average overhead of 1.1 millisecond per task.

5. Conclusions and Future Directions

We have shown in our results that HTC mode on Blue Gene/L provides large amounts of processing capacity to a wide range of applications. We have presented data for financial workloads which demonstrates Blue Gene's capability of running thousands of independent tasks. This is significant because it enables a new class of applications to execute on the Blue Gene/L architecture without significant porting effort. Application reliability is significantly improved by allowing single node software or hardware failures to occur without causing application termination on all nodes.

The asynchronous task dispatch subsystem described in this paper shows a scalable design capable of managing thousands of tasks. Running in HTC mode produced different IO patterns then typically seen in HPC applications on Blue Gene/L , therefore best performance is achieved by using a high IO to compute node ratio. Work is currently underway to explore further optimization of the HTC mode, such as persistent memory across worker program executions, mixing HTC and HPC applications within the same partition, and providing tighter integration of HTC mode with the Blue Gene/L control system. We are also continuing to refine our dispatcher control flow seeking efficiencies in the message queuing system.

Acknowledgements

The authors thank Nick Goracke for his HTC testing assistance. We would also like to thank Fred Mintzer and David Singer for their work with administration of the Watson Blue Gene/L. As well as Carl Obert, Sam Ellis, John Thomas, and Susan Lee for their continued support.

References

[1] J. Basney and M. Livny, "Deploying a high throughput computing cluster," in High Performance Cluster Computing: Architectures and Systems, Volume 1 (R. Buyya, ed.), Prentice Hall PTR, 1999.

[2] D. Thain and M. Livny, "The ethernet approach to grid computing.," in High-Performance Distributed Computing, pp. 138–151, IEEE Computer Society, 2003.

[3] Gara, M. A. Blumrich, et al. "Overview of the Blue Gene/L system Architecture." IBM Journal of Research and Development. Vol. 49, no. 2/3, March/May 2005.

[4] Allen, F. G. Almasi, et al. "Blue Gene: a vision for protein science using a petaflop supercomputer." IBM Systems Journal. Vol. 40, no. 2, February 2001.

[5] Chakravorty, S. C Mendes, et al. "HPC-Colony: services and interfaces for very large systems". ACM SIGOPS Operating Systems Review. Vol. 40, no. 2, April 2006.

[6] IBM System Blue Gene Solution Delivers Competitive Advantage to Finance and Securities Firms http://www03.ibm.com/servers/deepcomputing/pdf/fssindustrybrief.pdf

[7] M. Broadie and P. Glasserman, "A stochastic mesh method for pricing high-dimensional American options," Journal of Computational Finance, vol. 7, pp. 35–72, 2004.

[8] G. Mullen-Schultz and C. P. Sosa, "IBM System Blue Gene Solution: Application Development", RedBook, SG24-7179-04, Poughkeepsie, NY, June 21, 2007: http://www.redbooks.ibm.com/redbooks/pdfs/sg247179.pdf