BGW-Day Report: HPC-Colony's Resource Management Techniques for Large Systems

Laxmikant V. Kalé
(kale@cs.uiuc.edu)

October 19, 2006

1 Introduction

As part of the HPC-Colony project sponsored by the Department of Energy's FastOS program (http://www.hpc-colony.org), we are conducting research on resource management techniques for systems with very large numbers of processors. In particular, our group at the Parallel Programming Laboratory at the University of Illinois (http://charm.cs.uiuc.edu) is studying load balancing and fault tolerance techniques for that class of systems. During the recent BGW-Day, we had a chance to test the scalability of some of our approaches in those areas. This document describes the tests we conducted, the results we obtained, and the conclusions that we could draw from such experiments.

The remainder of this report is organized as follows. In Section 2 we briefly describe our advanced load balancing schemes and the experiments that we conducted to test them. Section 3 presents our fault tolerance approaches and their observed performance on BGW. Section 4 shows scalability results for a real scientific application that is based on those techniques, running on large BGW configurations. Finally, Section 5 contains our conclusions and a summary of our achievements during BGW-Day.

2 Load Balancing Techniques

Load balancing on peta-scale machines is a highly challenging problem, due to the large scale of the parallel system and the complexity in applications. In the context of Charm++, in order to exploit such peta-scale machines and take advantage of the load balancing capability of Charm++, applications typically involve a number of migratable objects that is by far larger than the physical system size. The complexity of load balancing algorithms thus leads to significant overhead from load balancing both in terms of CPU time and of memory requirements.

2.1 Load Balancing Scheme

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 [4].

2.2 Load Balancing Experimental Results

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 BGW. The test program creates a large number of objects having size 512KBytes. 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 1.

nPEs	4096	8192	16384
Mem	6.8MB	22.57MB	22.63MB

Table 1: Memory Usage of the HybridLB

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. Table 2 shows these results.

nPEs	8192	16384
Time	14.70s	23.16s

Table 2: Load balancing time of the HybridLB

We found that the load balancing process itself is not as efficient as we expected. We captured performance traces on BGW and dumped performance logs for post-mortem analysis. Through this analysis, conducted 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.

3 Fault Tolerance

Traditionally, checkpointing and restart has been one of the most widely used techniques for fault tolerance in large parallel applications. By periodically saving application status to permanent storage, the execution can be restarted from the last checkpoint if system faults occur. This scheme is available for Charm++ and AMPI applications [5]. However, in large applications, the costs involved in saving checkpoints to disk may be too high. We have implemented two other schemes for fault tolerance in parallel applications, based on diskless checkpointing and message-logging. We tested both schemes on BGW, and report the observed results in the next subsections.

3.1 In-Memory Checkpointing

Our in-memory checkpoint scheme [5, 6] adopted the idea of diskless checkpointing, which checkpoints data in memory. It uses a *coordinated* checkpoint strategy. In order to handle one fault at a time — a common case scenario, each checkpoint of an object needs be stored in the memory of two different processors. This *double-checkpointing* ensures the availability of one checkpoint in case the other is lost. Since accessing memory is much faster than accessing disk, the potentially low checkpoint overhead and faster restart allows us to achieve better performance than traditional disk-based checkpoint schemes. Sending checkpoint data to the memory of other processors takes advantage of the high speed interconnect, resulting in much lower overhead compared with disk-based checkpointing. With the distributed nature of the checkpoint protocol, our checkpoint approach scales well when the number of processors increases.

3.1.1 In-Memory Checkpointing Results

We used a 7-point stencil with 3-D domain decomposition, written in MPI, to test the performance of the in-memory checkpoint scheme. For a particular number of processors, we varied the amount of data per processor. We took multiple checkpoints for every run and report the average time for a checkpoint. Table 3 shows the average time for a checkpoint for varying amounts of data per processor on 8192 and 20480 BGW processors.

3.2 Message Logging

We bring together the ideas of sender-based pessimistic message logging and object-based virtualization to develop a fault tolerance protocol that provides fast recovery from faults [1]. Object-based virtualization [3] encourages the user to view his computation as a large number of interacting objects. These objects are also referred to as virtual processors. Virtual processors can interact only by sending messages to each other. The user decomposes his computation into virtual processors without caring about the number of physical processors present. The runtime system is

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 3: Time taken for in-memory checkpoints with different amounts of data per BGW processor

responsible for mapping the virtual processors to physical processors. Messages between different objects are delivered by the runtime system without the user being aware of the objects' physical location.

We developed a sender-based pessimistic message logging protocol that works along with object-based virtualization. In sender-side message logging, the messages sent to the receiver and the sequence in which they are processed by the receiver are stored on the senders. This reduces the overhead of pessimistic message logging and also removes the need for an idealized stable storage. We treat the virtual processors, and not the physical processors, as the communicating entities that send and receive messages. Since an object's state is modified only by the messages it receives, we can apply the PWD assumption to virtual processors instead of physical processors. After a crash, if a virtual processor re-executes messages in the same sequence as before, it can recover its exact pre-crash state. Therefore, we run the sender-based message logging protocol with the objects as participating entities instead of physical processors.

Virtualization affords us a number of potential benefits with respect to our message logging protocol. It is the primary idea behind faster restarts since it allows us to spread the work of the restarting processor among other processors. The facility of runtime load balancing can be utilized to restore any load imbalances produced by spreading the work of a processor among other processors. Virtualization also makes applications more latency-tolerant by overlapping communication of one object with the computation of another. This helps us hide the increased latency caused by the sender-side message logging protocol.

Although combining virtualization with message logging provides a number of advantages, it requires significant extensions to a basic sender-side message logging protocol. These extensions are primarily required for the case where a virtual processor sends a message to another one on the same physical processor (say processor A). We record some data about such a message in the memory of another physical processor (say B) that we refer to as the *buddy* of processor A. If A crashes, the data is fetched from its buddy B so that the virtual processors on A can reprocess the same messages in the same sequence as before the crash. In fact, a processor's buddy should be chosen such that the chances of both nodes failing at the same time are as low as possible.

We implement this message logging protocol in the Charm++/AMPI runtime system. Since AMPI is an implementation of MPI on top of Charm++, it lets traditional MPI codes to take advantage of our message logging protocol without being modified.

3.2.1 Message-Logging Experimental Results

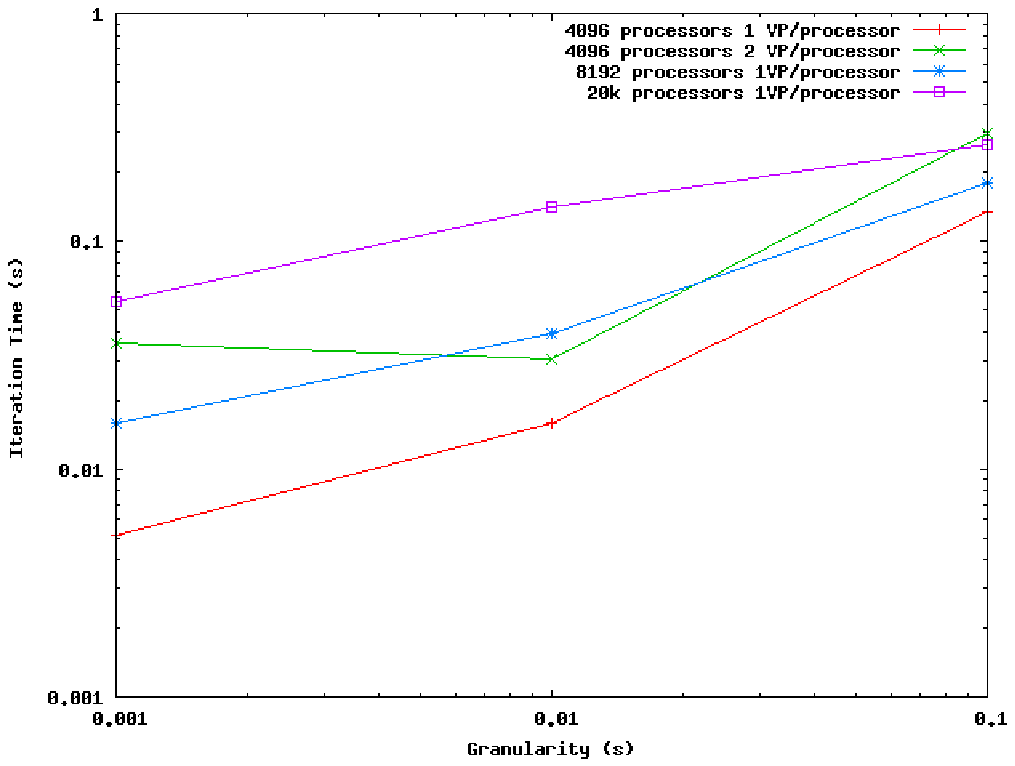

Figure 1: Iteration time of 7-point stencil on BGW for different levels of granularity

To study the performance of our message logging protocol on different numbers of processors, we used a 7-point stencil with 3-D domain decomposition written in MPI. The stencil program was written so that in every iteration each MPI thread (virtual processor) exchanges messages with its neighbors and then performs computations for a fixed amount of time. The amount of time a virtual processor 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 using 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, with one processor per BGW node. These 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.

4 Scientific Applications

Many of the enhancements that we are adding to the Charm++ infrastructure are directly available to the various scientific applications based on Charm++. One of these applications is a cosmological simulator, developed in collaboration with the University of Washington (Prof. Thom Quinn), under funding from an NSF grant (NSF-ITR-0205611). This new simulator, named ChaNGa (Charm++ N-body Gravity), had been previously scaled on up to one thousand processors. During BGW-Day, we were able to run the code using the entire machine (twenty thousand nodes). These executions achieved very good performance, as we describe in this section.

4.1 Cosmology Code

ChaNGa [2] is a new N-body cosmological simulator that can be used to study the formation of galaxies and planets. The code utilizes the Barnes-Hut tree topology to compute gravitational forces. We leverage the object-based virtualization inherent in the Charm++ runtime system to obtain automatic overlapping of communication and computation time, as well as to perform automatic runtime measurement-based load balancing. ChaNGa advances the state-of-the-art in N-Body simulations by allowing the programmer to achieve higher levels of resource utilization with moderate programming effort. In addition, as confirmed by our experimental results, the use of Charm++ has enabled ChaNGa to efficiently scale on large machine configurations.

4.2 Cosmological Simulation Results

We executed ChaNGa on BGW configurations of various sizes. In all of our tests, we used BGW in "Co-Processor" mode, so that each processor could utilize the maximum amount of memory and of network bandwidth available in a node. Initially, we used a cosmological dataset containing 50 million particles. Figure 2 shows the code performance for the gravity calculation phase in one iteration of the execution. The code scales well on up to two thousand processors, but there are no gains beyond that point. The reason for this is that with more than two thousand processors, there is no longer a total overlap between computation and communication, and the overhead imposed by the communication dominates the execution.

In our second set of tests, we used ChaNGa with a much larger dataset, consisting of 700 million particles. The observed performance on BGW is shown on Figure 3. One can see that there is still gain when twenty thousand processors are used, which is the maximum configuration where we can use the machine in "Co-Processor" mode. Thus, we can confidently claim that we can benefit from using all the available nodes in BGW for this cosmological simulator.

5 Conclusion and Final Comments

Our experimental results on BGW enabled us to verify that our approaches for resource management are indeed applicable for large systems such as Blue Gene. In the load balance area, our

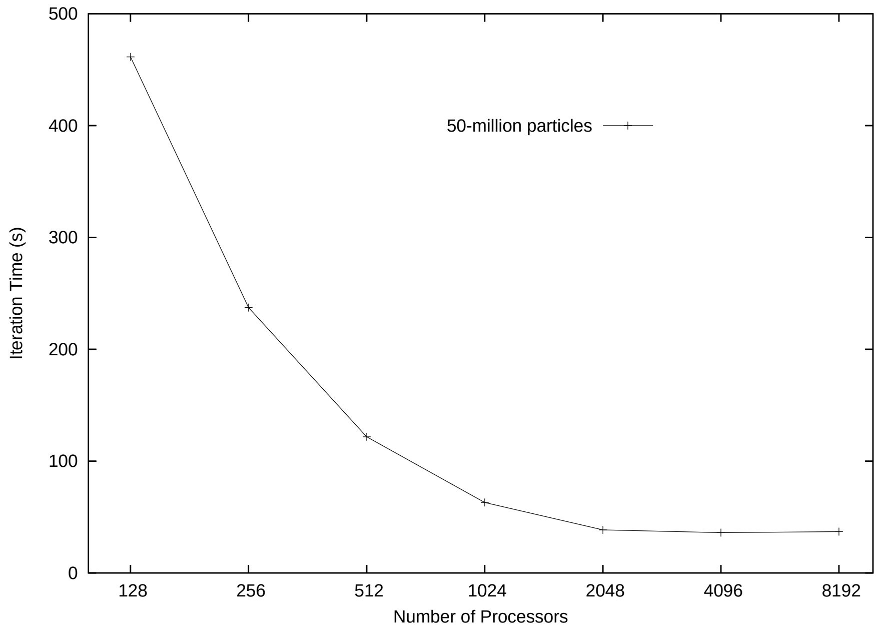

Figure 2: ChaNGa performance on BGW with 50-million particle dataset

hybrid load balancing scheme achieved much more limited use of memory than traditional centralized schemes across a significant number of processors, which makes agressive load balancing optimization possible on machines like BlueGene. The experiments also helped us identify performance bottlenecks of our new load balancing strategy for further improvement in the parallel efficiency.

Our in-memory checkpointing technique provided a checkpointing overhead that increased proportionally to the amount of state in each processor. We were able to demonstrate the use of this protocol in a fault-free scenario on up to twenty thousand BGW nodes. Meanwhile, our message-logging scheme provided an overhead in execution time that scaled well on large numbers of processors when the computation granularity was large. When that granularity was small, there was a significant performance penalty when we scaled the machine size; for applications with this level of granularity, message logging may not be the best fault tolerance approach.

In addition to our basic resource management tests, we successfully ran a Cosmology code based on Charm++. These tests utilized effectively all the BGW nodes when the dataset size was sufficiently large. Such large datasets are typical in production executions of interest to astronomers. Hence, we confirmed that our cosmological simulator running on a machine such as BGW can be a powerful resource to the Cosmology community.

In general, we were very pleased with the results of the tests conducted on BGW. Some of our

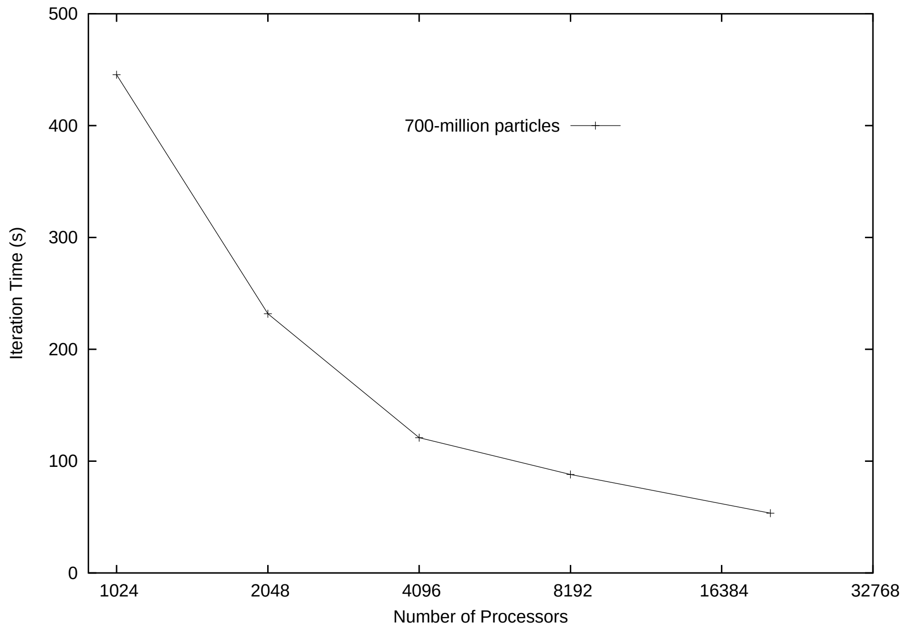

Figure 3: ChaNGa performance on BGW with 700-million particle dataset

tests on BGW did not succeed and we had to abort their execution, in order to use the time during the day for other tests; these errors might have been due to bugs in our code that only manifest in large machine configurations, and we are studying more carefully those issues. Nevertheless, most of our tests completed execution in a normal fashion. The results that we obtained validated some of our approaches, and confirmed that some of the tested techniques may not be applicable for all applications (such as message logging and small-granularity codes). Overall, the opportunity to conduct these tests on a large machine such as BGW was a very valuable step in our research. Furthermore, the simple fact that we obtained performance gains in a real application using the entire machine gives us even more evidence of the powerful capabilities of Charm++ as a basis for high performance computing on large systems.

Acknowledgments: We are very grateful to IBM for offering us time on BGW and for giving us the opportunity to test our codes on the full system. In particular, we thank Fred Mintzer and his team at IBM for the support provided during BGW-Day.

References

[1] Sayantan Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[2] Filippo Gioachin, Amit Sharma, Sayantan Chackravorty, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Scalable cosmology simulations on parallel machines. In *7th International Meeting on High Performance Computing for Computational Science (VECPAR)*, July 2006.

[3] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *First Intl. Workshop on Productivity and Performance in High-End Computing (HPCA 10)*, Madrid, Spain, February 2004.

[4] Gengbin 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] 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 Computing Systems*, 40(2), April 2006.

[6] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, pages 93–103, San Dieago, CA, September 2004.