UTILIZING THE MULTI-THREADING TECHNIQUES TO IMPROVE THE TWO-LEVEL CHECKPOINT/ROLLBACK SYSTEM FOR MPI APPLICATIONS *

Yuan Tang
Parallel Processing Institute
Software School
Fudan University, P.R.China
yuantang@fudan.edu.cn

Yunquan Zhang
Lab. of Parallel Computing
Institute of Software
Chinese Academy of Science
zyq@mail.rdcps.ac.cn

Abstract

With the increasing number of processors in modern HPC (High Performance Computing) systems, there are two emergent problems to solve. One is scalability, the other is fault tolerance. In our previous work, we proposed an MPI operation level checkpoint/rollback system. The main benefits of the system is that it offers the opportunity to employ in-memory (disk-less) checkpoint/rollback techniques which has demonstrated a much better performance over its on-disk counterpart, and the opportunity to have a concurrent two level recover-and-continue MPI system [1] which has been proven to have a high efficiency. To the scope of my knowledge, this is the first concurrent two-level checkpoint/recovery system in use. With the coming of Multi-core era, it's time to utilize the Multi-threading techniques to improve the performance of in-memory checkpointing algorithm. In this paper, we present two versions of MPI operation level checkpoint/rollback system, one is of single-threaded, the other is of Multi-threaded. Also, we provide an in-depth performance analysis between these two approaches to illustrate the benefits of Multi-threading techniques on multi-core platform. With the progress of our work, a picture of the hierarchy of future generation fault tolerant HPC system is gradually unrolled.

keywords: FT-MPI, Recover-and-Continue, Stop-and-Restart, Multi-core, Multi-threading Programming Model, Checkpoint, Rollback

*This paper is partly supported by National 863 Plan under contract NO.2006AA01A125 and NO.2006AA01A102

1 Background

The main goal of HPC is pursuing high performance. Restricted by the physical law, the performance of a single computing core is very limited. The HPC community has the trend to include more and more processors in one system. Today's number 1 in the top500 list, the IBM BlueGene/L - eServer, is composed of $212,992$ processors. And the systems with more processors are in development [19] [12]. With this trend of increasing number of processors integrated in one system , there are two emergent problems to solve. One is scalability, i.e. whether the performance of HPC system could grow at the same pace as the increasing number of processors. The other is fault tolerance. From current using experiences on high-end machines, a $100,000$ processor machine will experience multiple failures every hour [14].

The MPI Specification 1.2, the most popular parallel programming model, especially for large scale HPC systems, has not specified an efficient and standard way to process failures. Currently, MPI gives the user two options in processing failures. The first one, which is also the default mode of MPI, is to immediately abort the application. The second option is just returning the control back to the user application without requiring that subsequent operations succeed, nor that they fail. In short, according to the current MPI specification, an MPI program is not supposed to continue in case of an error. While most systems are much more robust, even though partial node failure/unavailability are much more frequent. In most cases the partial failure will be recovered very soon. So, there is a mismatch between hardware and the (non fault tolerant) programming model of MPI. There is a request for the programming model of MPI to include the capability of processing partial system failure/unavailability.

In [15], we extended the MPI specification in this

978-0-7695-3352-0/08 $25.00 © 2008 IEEE
DOI 10.1109/HPCC.2008.58

direction by constructing a systematic framework for the recovery procedures, communicator modes, message modes etc.(refer to [15] for details)

These extensions not only specify how the implementations of MPI library handles failures at system level, but provide the normal MPI application developers various recovery tradeoff between performance and cost. Also, an implementation of this extension, which is named FT-MPI [14], is available at http://icl.cs.utk.edu/ftmpi.

The main difference between FT-MPI's approach and lots of other fault tolerant parallel systems is that FT-MPI adopts a programming model of recover-and-continue other than stop-and-restart which is the tradition in lots of other fault tolerant parallel systems [11] [20] [17] [18] [21].

The main points of recover-and-continue are, when some processes are found failed/unavailable, the other processes neither exit nor migrate. Instead, they stay in their original processor/memory places and try re-spawning the failed processes and re-building the communicator. From the system's point of view, the main benefit of this approach is to significantly reduce the RTE (Run Time Environment) recovery costs(see [16]). Also, it provides the opportunity to employ the in-memory (disk-less) checkpoint/rollback techniques, which has demonstrated much higher performance than its on-disk counterpart [1]. More importantly, we have the opportunity to establish a framework of concurrent multiple level checkpoint/rollback [1].

However, this previous work did not cover the data sections in user application. In order to be really fault-tolerant, users' application have to have their own checkpoint/restart codes.

Based on the specification extension, we proposed an MPI operation level checkpoint/rollback system [1]. The initial test results have proven the high efficiency of our two level checkpoint/rollback system. The in-memory (diskless) level checkpoint/rollback process does demonstrate their advantages under certain conditions.

The test results of this initial implementation also reveal two main factors that may be the performance bottleneck of in-memory checkpointing algorithm: that is, the memory capacity and the interconnection performance. The test results reveals that the in-memory checkpointing overhead consists of two main parts: memory copy and collective communication. And the collective communication dominates the overall costs.

Because the in-memory checkpointing algorithm relies heavily on the underlying interconnect to conduct the algorithm (details in Section 2.1), the checkpoint size becomes very crucial to the overall performance.

With the coming of Multi-core era, we think that it will be a good idea to utilize the multi-threading programming model to overlap the checkpointing process with the computation/communication of the main thread. After implemented the idea, we tested it and found the solution really worked. The Multi-threaded checkpointing/rollback system has demonstrated approximately one order of magnitude performance improvement over its single-threaded counterpart.

2 Current Implementation of FT-MPI checkpoint/rollback system

Based on the above specification proposal and current implementation of FT-MPI, in our previous work [1] we implemented one single-threaded version of MPI operation level checkpoint/rollback library.

However, we found that the overall checkpointing costs are dominated by the collective communication [1]. With the coming of Multi-core era, it's time to exploit the multi-threading techniques to improve the performance. We decide to overlap the expensive collective communication overhead in checkpointing with the main computation thread. So we separated the original in-memory checkpointing procedure to two steps: first we perform a local memcpy() to copy the data_needs_ckpt to local memory; secondly we signal the background checkpointing thread to collaborate with its counterparts on remote nodes to checkpoint the data to remote memory. By this approach, the user's application will no longer waiting for the lengthy checkpointing procedure (especially, the collective communication) to complete. The call to the checkpointing routine returns immediately after just one local memcpy().

2.1 The imem-m-rep algorithm

No matter in single-threaded version or in Multi-threaded version, we use an in-memory-m-replication algorithm to store one copy of checkpoint in local memory on each node. This algorithm is designed to tolerate any $m, where 0 \leq m \leq n-1$, processes failure/unavailability simultaneously. Figure 1 demonstrates when the nof equals 2, i.e. $m = 2$, how the imem-m-rep algorithm works.

1. When the MPI_Ckpt_here() routine starts, every process (assume its rank is i) first copies the data_needs_ckpt into local_data_copy
2. For Multi-threaded version, at this point the main computation thread i returns after signaling the background checkpointing thread to run. For single-threaded version, because there is no separate checkpointing thread, the main thread will continue its job in checkpointing.

3. Every checkpointing thread i sends the data from local_data_copy to the ckpt_buf of process $(i+1)\%n$
4. If $m > 1$, every process i will repeatedly send the data received from its PREVIOUS (process $(i-1+n)\%n$) to its NEXT (process $(i+1)\%n$).
5. The send/receive pipeline keeps going until the repeated times equals m.
6. When the repeated times equals m, the checkpointing threads stop and write one bit in corresponding data structure to signal the end of current round of checkpointing.

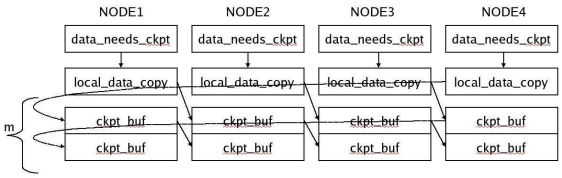

Figure 1. How imem-m-rep-ckpt works – Assume $m = 2$

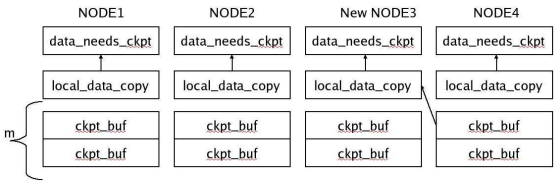

Figure 2. How imem-m-rep-rollback works – Assume the Process on Node 3 is newly re-spawned

For the Multi-threaded version, the overall checkpointing overhead sensed by the main thread is just that of a local memcpy(). The very expensive costs of global collective communication is well hidden from the user's application by the overlap. This approach works the best on Multi-core platform equipped with multiple network interface card as we will see soon.

With this checkpointing algorithm, the rollback algorithm is straight forward as illustrated in Figure 2(refer to [1] for details).

Obviously, this imem-m-rep algorithm could tolerate any $m, 0 \leq m \leq n-1$, number of simultaneous processes failure/unavailability.

The main disadvantage of in-memory (diskless) checkpoint/rollback algorithm is the memory consumption issue. Memory is a very scarce resource in large scale scientific computing. So we provide the user an alternative approach to checkpoint to/rollback from stable disk. The on disk checkpoint/rollback algorithm is very similar to that of LAM/MPI, so we omit the details here (interested readers could refer to [2] [3] for details).

3 Test Data

Due to page limitation, we only attach the most typical and interesting testing results here.

3.1 1st Testing Platform – PPI-XEON

Our first testing platform (PPI-XEON) is a cutting edge Multi-Core platform. It is of a single node, Dell PowerEdge 2950 server. The hardware/software parameters are listed below:

- Hardware:
  - 2 x Broadcom NetXtreme 1000Gbit/s Ethernet port
  - Computing nodes:
    - 2 x Intel Xeon 4-core 1.6GHz Processor
    - 2 GBytes Main Memory (ECC DIMM)
    - 146 GBytes Hard disk (10k rpm SAS)
- Software:
  - LINUX (kernel version: 2.6.20-2925.9.fc7xen)
  - gcc version 4.2.1, glibc version 2.6

Due to the hardware limitation, all our tests are performed by 4 processes on 4 different hard-cores, with the nof set to 1. The ratio of on-disk checkpointing to in-memory checkpointing was set to 1:1000 (which means we take 1 stable disk checkpoint automatically every 1000 in-memory checkpoints). The benchmark is a synthetic benchmark only for the purpose of measuring checkpointing/rollback overhead.

The Testing Data on PPI-XEON could be viewed at Figure 3. In Figure 3, we could see that no matter how small or how large the checkpoint size is, the Multi-threaded version always outperforms the single-threaded version by an order of magnitude. If we look into these performance data, we could find out that the checkpointing overhead sensed by end user (application developer) of Multi-threading version consists of just one local memory copy plus some thread overhead (mainly. signals and mutual exclusive lock) , while the overhead of Single-threaded version consists of one local memory copy and $m, (0 \leq m \leq n)$ collective communications. Because the collective communication of both the main

thread and the checkpointing thread are so intensive that the communication saturate the underlying network. So, the very high costs of collective communication dominates the overall costs of Single-threaded checkpointing. Obviously, for Multi-threaded version, because the very expensive collective communication for checkpointing is well hidden from the main thread, it wins.

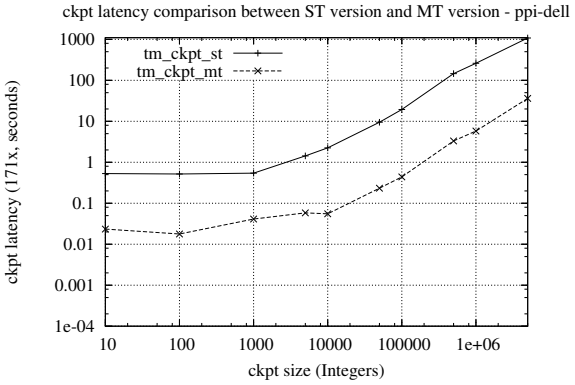

Figure 3. Performance Comparison between Single-threaded checkpoint and Multi-threading version – PPI-XEON; tm_ckpt_st is the checkpointing overhead of Single-threaded version; tm_ckpt_mt is that of Multi-threading version

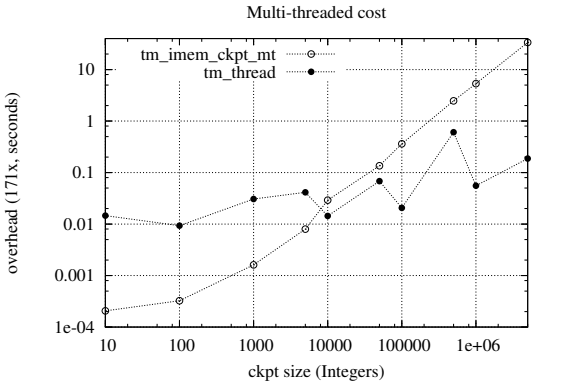

Figure 4. Thread cost versus imem-ckpt overhead – PPI-XEON; tm_imem_ckpt_mt is the real checkpointing overhead, excluding thread overhead, of Multi-threaded version; tm_thread is the corresponding thread overhead

Figure 4 tells us that if the checkpoint size is too small, the thread costs (mainly, signals and mutual exclusive locks) introduced by Multi-threading version is

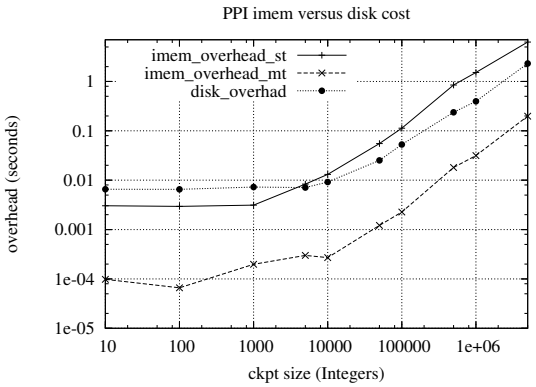

Figure 5. in-memory versus on-disk overhead – PPI-XEON ; imem_overhead_st is the real checkpointing overhead of Single-threaded version; imem_overhead_mt is that of Multi-threading version

a little higher than the actual checkpointing overhead.

Compared the overhead of in-memory checkpointing with that of on-disk algorithm, we could see that if the checkpoint size increases, the overhead of in-memory checkpoint increases as well. After some threshold point, the high costs of large message collective communication beat the benefits of in-memory algorithm.

3.2 2nd Testing Platform – FDU-ITANIUM

Our second testing platform (FDU-ITANIUM) is a real cluster. It consists of 32 dual-Itanium2-processor nodes. The main hardware/software parameters are listed below:

- Hardware:
  - 32 dual-Itanium2 1.3GHz nodes in total
  - Myrinet 2000 Network Interface (LANai 10.0)
  - Computing nodes:
    - Dual Intel Itanium2 1.3GHz Processor
    - 2 GBytes Main Memory per node (PC2100 DDR SDRAM)
    - 36 GBytes Hard disk (10k rpm SCSI)
- Software:
  - LINUX (kernel version: 2.4.18-e.31smp)
  - gcc version 2.96, glibc version 2.2.x

In order to compare with the testing results on PPI-XEON, we employ the same synthetic benchmark as above and still use only 4 processes (one process on one node and every node has a dual-Itanium2 processors).

The testing data on FDU-ITANIUM could be viewed at figure 6 and Figure 7. These two figures tell approximately the same story as that on PPI-XEON.

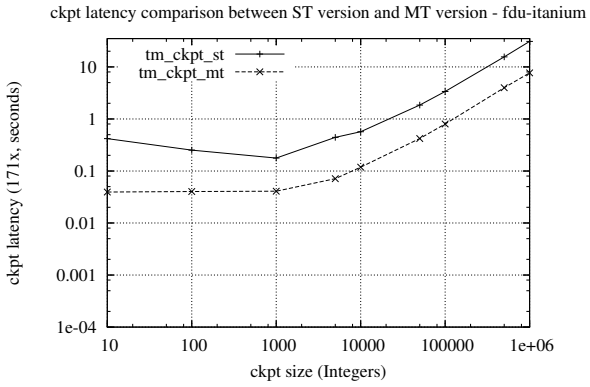

Figure 6. Performance Comparison between Single-threaded checkpoint and Multi-threading version – FDU-ITANIUM; tm_ckpt_st is the checkpointing overhead of Single-threaded version; tm_ckpt_mt is that of Multi-threading version

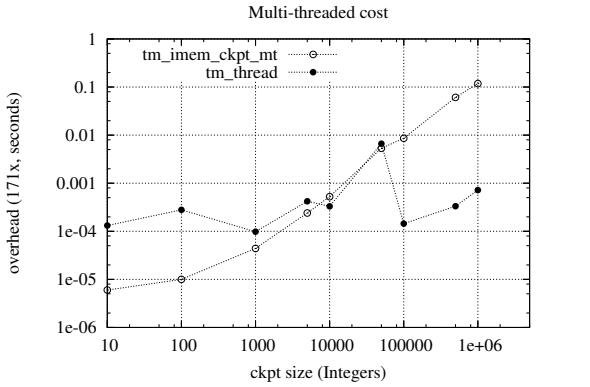

Figure 7. Thread cost versus imem-ckpt overhead – FDU-ITANIUM; tm_imem_ckpt_mt is the real checkpointing overhead of Multi-threading version; tm_thread is the corresponding thread overhead

However, Figure 8 tells a little different story from its PPI-XEON counterpart. Figure 8 tells us that the

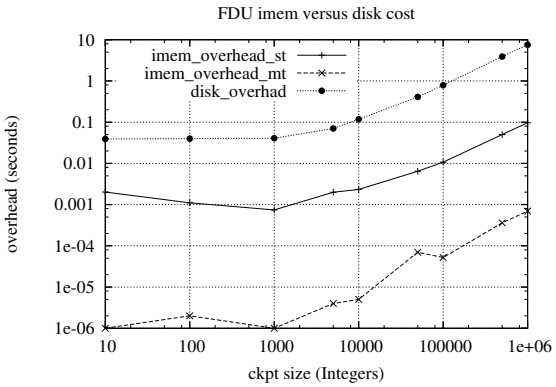

Figure 8. in-memory versus on-disk overhead – FDU-ITANIUM; imem_overhead_st is the real checkpointing overhead of Single-threaded version; imem_overhead_mt is that of Multi-threading version

choice of which checkpointing algorithm to choose relies heavily on the condition of network interface. If the network interface is a high performance one (which should have high bandwidth, low latency, and low congestion ratio, etc.), like the Myrinet 2000 on our FDU-ITANIUM platform, for a very wide range of checkpoint size, the in-memory algorithm could have a much better performance over its on-disk counterpart.

3.3 Some conclusions

From above testing results, we summarize some most important points below:

- The in-memory checkpointing algorithm depends heavily on the network conditions, both its performance and its traffic load. So which algorithm to choose, in-memory or on-disk, depends on the memory capacity, interconnect performance, and the performance of stable disk system. In one word, which checkpointing algorithm to choose is and should be performance driven.

- If we choose in-memory checkpointing algorithm, the Multi-threaded version always outperforms the single-threaded version by an order of magnitude, especially on a Multi-core/Multi-processor system equipped with multiple network interface cards, because the Multi-threaded version overlaps very well the checkpointing thread with the computation/communication of main thread.

References

[1] Y. Tang, G. Fagg, and J. Dongarra. Proposal of MPI operation level checkpoint/rollback and one implementation. In Proceedings of IEEE CCGrid 2006, pages 27–34, 2006.

[2] S. Sankaran, J. Squyres, B. Barrett, A. Lumsdaine, et al. Checkpoint-restart support system service interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.

[3] S. Sankaran, J. Squyres, B. Barrett, A. Lumsdaine, et al. The LAM/MPI Checkpoint/Restart framework: System-initiated checkpointing. International Journal of High Performance Computing Applications, 19(4):479-493, Winter 2005.

[4] J. Duell, P. Hargrove, and E. Roman, The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Berkeley Lab, 2002

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volative Nodes. In Proceedings of ACM/IEEE SC'2002, Baltimore, MD, Nov. 2002.

[6] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madsin, Computer Science Department, April 1997.

[7] H. Jung, D. Shin, H. Han, J.W.Kim, H.Y.Yeom, and J. Lee. Design and Implementation of Multiple Fault-Tolerant MPI over Myrinet (M^3) In Proceedings of ACM/IEEE SC'2005, Seattle, WA, Nov. 2005.

[8] Myricom Myrinet Software and Customer Support. http://www.myri.com/scs/, 2003

[9] R.T.Aulwes, D.J.Daniel, N.N.Desai, R.L.Graham, L.D.Risinger, and M.W.Sukalski, M.A.Taylor, T.S.Woodall Architecture of LA-MPI, a network-fault-tolerant MPI In Proceedings of International Parallel and Distributed Processing Symposium, Santa Fe, NM, Apr. 2004

[10] Q. Gao, W. Yu, W. Huang, D.K.Panda Application-Transparent Checkpoint/Restart for MPI programs over InfiniBand in Proceedings of the International Conference on Parallel Processing, 2006.

[11] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In Proceedings of Supercomputing Symposium, pages 379–386, 1994.

[12] J. Dongarra. An overview of high performance computers, clusters, and grid computing. 2nd Teraflop Workbench Workshop, March 2005.

[13] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. Harness and fault tolerant mpi. Parallel Computing, 27(11):1479–1495, 2001.

[14] G. E. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 346–353, London, UK, 2000. Springer-Verlag.

[15] G. E. Fagg, E. Gabriel, G. Bosilca, and et al. Extending the mpi specification for process fault tolerance on high performance computing systems. Proceedings of the ISC2004, June 2004.

[16] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesiva-Grbovic, and J. J. Dongarra. Process fault tolerance: semantics, design and applications for high performance computing. The International Journal of High Performance Computing Applications, 19(4):465–477, 2005.

[17] E. Godard, S. Setia, and E. L. White. Dyrect: Software support for adaptive parallelism on nows. In IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, pages 1168–1175, London, UK, 2000. Springer-Verlag.

[18] V. K. Naik, S. P. Midkiff, and J. E. Moreira. A checkpointing strategy for scalable recovery on distributed parallel systems. In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM), pages 1–19, New York, NY, USA, 1997. ACM Press.

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

[20] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In Proceedings, 10th European PVM/MPI Users' Group Meeting, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[21] S. S. Vadhiyar and J. Dongarra. Srs: A framework for developing malleable and migratable parallel applications for distributed systems. Parallel Processing Letters, 13(2):291–312, 2003.

[22] N. H. Vaidya. A case for two-level recovery schemes. IEEE Trans. Comput., 47(6):656–666, 1998.