Technical Comparison between several representative checkpoint/rollback solutions for MPI programs

Yuan Tang
Innovative Computing Laboratory
Department of Computer Science
University of Tennessee Knoxville, U.S.A
yuantang@cs.utk.edu

ABSTRACT

With the increasing number of processors in modern HPC(High Performance Computing) systems (65536 in current #1 IBM BlueGene/L), there are two emergent problems to solve. One is Scalability, that is, whether the performance of HPC system could grow at the pace of the number of the processor. The other is fault tolerance. Concluding from the current experiences on the top-end machines, a 100,000-processor machine will experience a process failure every few minutes. Currently, there're following two representative fault tolerance solution of MPI program : $MPICH-V/V2/CL+condor$, which employs un-coordinated checkpoint approach and is a user level checkpoint/restart library; $LAM/MPI+BLCR$ provides coordinated, system level checkpoint/restart; $FT-MPI$ will be a user directive fault tolerant MPI package. In this paper, we compare some technical details between these solutions.

Categories and Subject Descriptors

B.8.1 [**PERFORMANCE AND RELIABILITY**]: Reliability, Testing, and Fault-Tolerance;
D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Parallel programming*;

Keywords

HPC, MPI, Fault Tolerance, checkpoint and rollback, performance comparison, modelling and prediction

1. METHODOLOGY

In order to utilize MPICH-V2's checkpoint/rollback features, users must set the checkpoint interval (-xw-checkpoint ${time _in_sec}) on command line, then the checkpoint/rollback actions will all be done by system. For LAM/MPI, the checkpoint/ restart process are started by user separately on command line by "cr_checkpoint ${pid_mpirun}" and "cr_restart context.${pid _mpirun}", respectively.

Every time to take a checkpoint, MPICH-V2 snap shot only one MPI process' image as well as the message-in-fly in random order. LAM/MPI will first drain all the message-in-fly (synchronization/ coordination). Then, LAM/MPI will checkpoint all MPI processes' images plus that of MPIRUN but excluding any messages.

In the rest of this paper, we firstly try running a synthesized benchmark – do_coll_reset, which is easy to adjust the message size, message frequency, computation time, and so on to reflect needs. Based on the run and collection of data of do_coll_reset , we will try establish a performance prediction metric. Finally, we will try using this metric to predict the performance of some real benchmarks, say NPB, and compare it with the real run data.

1.1 do_coll_reset

The do_coll_reset application performs contiguous collective communication among all the processes. There're some unit computation/ sleep distributed evenly between these communication. There're in total 4 parameters in do_coll_reset which could be adjusted for modelling purpose : msgSize, TAG_COLL (number of operation, including communication and computation), r_comm2all_num (number ratio of communication to all, which decides the message frequency), SLEEP_TIME (the duration time of unit computation).

1.2 Sample run of do_coll_reset

1.2.1 1st run

In this sample run, we set:

- $msgSize = 1000 \times sizeof(int)$
- $SLEEP\_TIME = 0.00001 seconds$
- $TAG\_COLL = 30,000$

Then, we set :

- $msgSize = 10,000 \times sizeof(int)$
- $SLEEP\_TIME = 0.0001 seconds$
- $TAG\_COLL = 30,000$

First, we need some explanation of Table 1 and Table 2:

MPICH-V2				
r_comm2all_num	0.05	0.35	0.65	0.95
T_total (no ckpt)	111.97 (111.9)	80.8 (79.17)	49.00 (46.58)	21.71 (19.58)
# ckpt	56	40	21	8
single overhead	0.0012	0.0407	0.1152	0.2662
T_com (no ckpt)	1.26 (1.17)	7.57 (6.33)	11.71 (10.01)	15.48 (13.05)
Time_% (no ckpt)	1.13% (1.04%)	9.37% (8%)	24.09% (21.51%)	69.59% (66.65%)
#_%	4.79%	35.34%	65.44%	95.00%
LAM/MPI				
r_comm2all_num	0.05	0.35	0.65	0.95
T_total (no ckpt)	124.65 (111.63)	84.73 (76.02)	46.41 (41.50)	10.26 (8.98)
# ckpt	88	61	33	8
single overhead	0.1479	0.1427	0.1487	0.16
T_com (no ckpt)	3.45 (0.67)	7.5 (3.91)	7.34 (4.33)	4.92 (3.97)
Time_% (no ckpt)	2.76% (0.60%)	8.85% (5.15%)	15.84% (10.45%)	48.07% (44.19%)
#_%	4.79%	35.34%	65.44%	95.00%

Table 1: Run of do_coll_reset with TAG_COLL=30,000; tm_sleep_us(0.00001); msgSize=1000Int

MPICH-V2				
r_comm2all_num	0.05	0.35	0.65	0.95
T_total (no ckpt)	117.64 (113.79)	106.11 (89.25)	83.50 (71.85)	75.04 (63.93)
# ckpt	54	23	13	9
single overhead	0.0712	0.7330	0.8961	1.2344
T_com (no ckpt)	5.83 (2.68)	32.61 (17.57)	43.62 (34.27)	64.79 (54.26)
Time_% (no ckpt)	4.9% (2.35%)	30.62% (19.7%)	52.24% (48.03%)	86.42% (84.97%)
#_%	4.79%	35.34%	65.44%	95.00%
LAM/MPI				
r_comm2all_num	0.05	0.35	0.65	0.95
T_total (no ckpt)	124.74 (112.21)	90.97 (82.18)	63.61 (57.29)	43.79 (38.76)
# ckpt	87	59	42	28
single overhead	0.1440	0.1489	0.1504	0.1796
T_com (no ckpt)	3.91 (1.56)	14.31 (10.10)	25.43 (21.03)	38.65 (33.81)
Time_% (no ckpt)	3.13% (1.39%)	15.74% (12.3%)	40.06% (36.46%)	88.18% (87.23%)
#_%	4.79%	35.34%	65.44%	95.00%

Table 2: Run of do_coll_reset with TAG_COLL=30,000; tm_sleep_us(0.0001); msgSize=10,000Int

- r_comm2all_num is the number ratio of communication to all. This number decides the message frequency in the 'do_coll_reset'. In 'do_coll_reset', we employ the pseudo random number generator to make the communication and computation operation distributed evenly across the run. For example, if r_comm2all_num = 0.65, 65% of all the operation will be communication.
- # ckpt is the total number of checkpoint taken during the run. In MPICH-V2, we set '-xw-checkpoint 60', while in LAM/MPI, we set 'sleep 1' in between consecutive cr_checkpoint operation.
- T_total (no ckpt) is the total time of the run. The number without parenthesis is the time taken with checkpoint. The number in parenthesis is the time taken without any checkpoint.
- single overhead is the number of $(T\_total - T\_total\_no\_ckpt)/(\#ckpt)$
- T_com (no ckpt) is the total communication time taken with/ without checkpoint, respectively.
- Time_% (no ckpt) is the ratio of communication time to total with/ without checkpoint, respectively.
- #_% is the ratio of actual number of communication operation to all.
- all the numbers listed in Table 1 and Table 2 are the average of at least 4 runs.

In both Table 1 and Table 2, we could see the number of 'single overhead' (the average overhead of taking one checkpoint) of MPICH-V2 and LAM/MPI will cross at some point. Intuitively, if the total communication time is less than some threshold, the MPICH-V2's single overhead is much smaller, else the LAM/MPI will win. As we know, the total communicatin time depends on both the message size and frequency. In Table 1 and Table 2, we have different message size and frequency and the cross point are at different 'r_comm2all_num's. If we use only one metric to describe it, it should be 'T_com' without any checkpoint. That is, if the 'T_com' without any checkpoint is less than about 10-11 seconds, the single overhead of MPICH-V2 will be smaller than LAM/MPI. The reason of selecting 'T_com' is due to the message cumulative effects of MPICH-V2. Also, in both Table 1 and Table 2, we could notice that without checkpoint, the total run time of LAM/MPI is always a little bit smaller than MPICH-V2. The more percentage of communication, the larger the gap will be.

Table 3 and Table 4 provide a zoom-in picture of Table 1 and Table 2.

1.3 Run of NPB's BT

In this subsection, we list the running results of NPB 2.4's BT on both LAM/MPI and MPICH-V2.

Table 5 tells us that without checkpoint, the LAM version of BT, no matter class A or B, are always a little bit faster. While with checkpoint taken, the LAM version of BT, especially the time overhead of taking a single checkpoint will be longer than that of MPICH-V2. And the larger the problem set size, the larger the gap will be.

In Table 6, we could see the difference of application run time image between MPICH-V2 and LAM are very little. Figure 1(a) and Figure 1(b) illustrate the difference of checkpoint size. The MPICH-V2 takes only one MPI process' image at one time, and the size keep increasing as the run continues. The LAM/MPI everytime takes all the MPI processes' image as checkpoints, and the checkpoint size keeps almost constant. The LAM curve in Figure 1(a) and Figure 1(b) illustrate the checkpoint size of one MPI process.

MPICH-V2			
r_comm2all_num	0.05	0.10	0.35
T_total (no ckpt)	117.64 (113.79)	122.28 (108.76)	106.11 (89.25)
# ckpt	54	48	23
single overhead	0.0712	0.2816	0.7330
T_com (no ckpt)	5.83 (2.68)	15.27 (5.02)	32.61 (17.57)
Time_% (no ckpt)	4.9% (2.35%)	12.43% (4.62%)	30.62% (19.7%)
#_%	4.79%	9.88%	35.34%
LAM/MPI			
r_comm2all_num	0.05	0.10	0.35
T_total (no ckpt)	124.74 (112.21)	121.35 (106.89)	93.45 (82.56)
# ckpt	87	97	75
single overhead	0.1440	0.1490	0.1452
T_com (no ckpt)	3.91 (1.56)	6.30 (3.09)	16.40 (10.80)
Time_% (no ckpt)	3.13% (1.39%)	5.19% (2.89%)	17.55% (13.08%)
#_%	4.79%	9.88%	35.34%

Table 3: Run of do_coll_reset with TAG_COLL=30,000; tm_sleep_us(0.0001); msgSize=10,000Int

MPICH-V2				
r_comm2all_num	0.45	0.55	0.60	0.65
T_total (no ckpt)	73.70 (67.02)	58.77 (56.72)	56.45 (51.71)	49.27 (45.85)
# ckpt	36	27	27	21
single overhead	0.1855	0.0759	0.1755	0.1628
T_com (no ckpt)	12.57 (6.92)	10.13 (8.70)	11.68 (9.35)	12.36 (9.40)
Time_% (no ckpt)	17.05% (10.33%)	17.26% (15.37%)	20.74% (18.09%)	25.12% (20.50%)
#_%	45.23%	55.29%	60.29%	65.44%
LAM/MPI				
r_comm2all_num	0.45	0.55	0.60	0.65
T_total (no ckpt)	73.32 (64.65)	60.22 (53.38)	53.75 (47.37)	47.40 (41.35)
# ckpt	75	49	44	38
single overhead	0.1156	0.1395	0.145	0.1592
T_com (no ckpt)	8.91 (4.80)	8.73 (6.49)	7.99 (4.77)	8.09 (3.67)
Time_% (no ckpt)	12.16% (7.43%)	14.52% (12.16%)	14.89% (10.08%)	17.03% (8.89%)
#_%	45.23%	55.29%	60.29%	65.44%

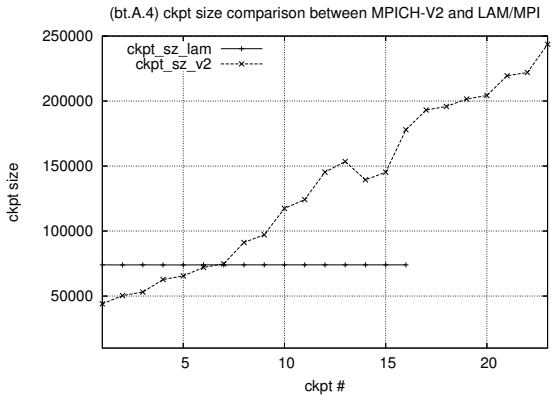
Table 4: Run of do_coll_reset with TAG_COLL=30,000; tm_sleep_us(0.00001); msgSize=1000Int

	V2-bt.A.4	V2-bt.B.4	LAM-bt.A.4	LAM-bt.B.4
T_total ()	192.16 (160.38)	759.74 (687.97)	215.7 (155.39)	1682.8 (673.36)
# ckpt	22	27	42	121
single	1.4148	2.6235	1.4358	8.3077
T_com	35.76 (11.38)	80.31 (45.83)	33.16 (7.20)	434.12 (23.28)
ratio	19.66% (7.09%)	10.56% (6.65%)	15.37% (4.63%)	25.8% (3.45%)

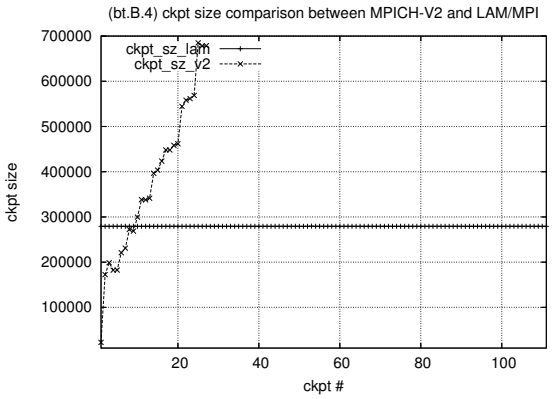
Table 5: Comparison of NPB2.4's BT between MPICH-V2 and LAM/MPI

	V2-bt.A.4	V2-bt.B.4	LAM-bt.A.4	LAM-bt.B.4
VmSize	$82808KB$	$298836KB$	$91680KB$	$307828KB$
VmRSS	$74580KB$	$280376KB$	$75260KB$	$281072KB$
VmHeap	$81588KB$	$297620KB$	$89444KB$	$305596KB$
VmStk	$28KB$	$28KB$	$24KB$	$24KB$
VmExe	$1148KB$	$1144KB$	$484KB$	$480KB$
VmLib	$0KB$	$0KB$	$1636KB$	$1636KB$

Table 6: Comparison of NPB2.4's BT between MPICH-V2 and LAM/MPI



(a) ckpt size comparison of bt.A.4 between V2 and LAM



(b) ckpt size comparison of bt.B.4 between V2 and LAM

Figure 1: ckpt size comparison between V2 and LAM

2. REFERENCES

[1] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamos, CA, USA, 2002. IEEE Computer Society Press.

[2] Y. Chen, K. Li, and J. S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. In *SC97: High Performance Networking and Computing*, San Jose, CA, USA, 1997.

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

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

[5] 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.

[6] 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.

[7] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications. *International Journal of High Performance Computing Applications*, 11(3):224–236, August 1997.

[8] 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.

[9] W. Gropp and E. Lusk. Fault tolerance in mpi programs. *submitted to a special issue of the Journal High Performance Computing and Applications*.

[10] Y. Kim, J. Plank, and J. Dongarra. Fault tolerant matrix operations for networks of workstations using multiple checkpointing. 1997.

[11] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, 1994.

[12] V. K. Naik, S. P. Midkiff, and J. E. Moreira. A checkpointing strategy for scalable recovery on distributed parallel systems. pages 1–19, 1997.

[13] T. Organization. System processor counts/systems in top500 list nov. 2004. November 2004.

[14] T. Organization. System processor counts/systems in top500 list june 2005. June 2005.

[15] J. Plank, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing, 1995.

[16] J. S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-likesystems. 1996.

[17] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. pages 213–223, January 1995.

[18] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on reed-solomon coding. (CS-03-504), April 2003.

[19] J. S. Plank, Y. Kim, and J. J. Dongarra. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43(2):125–138, 1997.

[20] J. S. Plank, K. Li, and M. A.Puening. Diskless checkpointing. 1997.

[21] S. Vadhiyar and J. Dongarra. Srs - a framework for developing malleable and migratable parallel applications for distributed systems. 2002.

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