In-Memory Checkpointing for MPI Programs by XOR-Based Double-Erasure Codes*

Gang Wang, Xiaoguang Liu, Ang Li, and Fan Zhang

Nankai-Baidu Joint Lab, College of Information Technical Science,
Nankai University, 94 Weijin Road, Tianjin, 300071, China
wgzwp@163.com, liuxg74@yahoo.com.cn,
{megathere,zhangfan555}@gmail.com

Abstract. Today, the scale of High performance computing (HPC) systems is much larger than ever. This brings a challenge to fault tolerance of HPC systems. MPI (Message Passing Interface) is one of the most important programming tools for HPC. There are quite a few fault-tolerant extensions for MPI, such as MPICH-V, StarFish, FT-MPI and so on. Most of them are based on on-disk checkpointing. In this paper, we apply two kinds of XOR-based double-erasure codes - RDP (Row-Diagonal Parity) and B-Code to in-memory checkpointing for MPI programs. We develop scalable checkpointing/recovery algorithms which embed erasure code encoding/decoding computation into MPI collective communications operations. The experiments show that the scalable algorithms decrease communication overhead and balance computation effectively. Our approach provides highly reliable, fast in-memory checkpointing for MPI programs.

Keywords: MPI, fault-tolerant, erasure codes, collective communication.

1 Introduction

Today, the scale of High performance computing (HPC) systems is much larger than ever. 54.8% of Top500 systems are composed of 129-512 processors in June 2003. In November 2007, 53.6% of Top500 systems are composed of 1025-2048 processors. The fastest system in 2007 consists of more than twenty thousand processors [1].

A challenge to the systems of such a scale is how to deal with hardware and software failures. Literal [2] reports the reliability of three leading-edge HPC systems: LANL ASCIS Q (8192 CPUs) has a MTBI (Mean Time Between Interrupts) of 6.5 hours, LLNL ASCI White (8192 CPUs) has a MTBF (Mean Time Between Failures) of only 5.0 hours, and Pittsburgh Lemieux (3016 CPUs) has a MTBI of 9.7 hours. In order to deal with such frequent failures, some fault tolerance mechanisms must be considered in both hardware and software design.

* This paper is supported partly by the National High Technology Research and Development Program of China (2008AA01Z401), RFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000).

M. Ropo et al. (Eds.): EuroPVM/MPI 2009, LNCS 5759, pp. 84–93, 2009.

MPI (Message Passing Interface) [3] is one of the most important programming tools for HPC. Fault-tolerant MPI is obviously necessary for users to run jobs reliably on large HPC systems. Some fault-tolerant extensions had been developed for MPI [4, 5, 6, 7]. Their common fault tolerant method is checkpointing. We have studied MPI in-memory checkpointing technique based on erasure codes [10]. In this paper, we apply two kinds of XOR-based double-erasure codes - RDP and B-Code to in-memory checkpointing. Scalable checkpointing/recovery algorithms combing erasure code encoding/decoding and collective communication operations are developed. The experiments show that the scalable algorithms decrease communication overhead and balance computation effectively.

2 Fault Tolerant MPI and Checkpointing

Today's HPC applications typically run for several days, even several weeks or several months. To deal with the frequent system failures, checkpointing technique is generally used. Checkpoints - the states of processes are written into nonvolatile storage (often hard disk) periodically. If a process failure occurs, the last checkpoint is read and then all (surviving and re-spawned) processes roll back to the latest state. The current MPI specification doesn't refer to any fault tolerance strategy. Therefore, quite a few extensions had been developed to help programmers handle failures more conveniently and efficiently. Cocheck [4] is the first checkpointing extension for MPI. It sits on top of the message passing library. Its shortcoming is global synchronization and implementation complexity. Starfish [5] implements checkpointing at a much lower level. It uses strict atomic group communication protocols, and thus avoids the message flush protocol of Co-Check. MPICH-V uses a coordinated checkpoint and distributed message log hybrid mechanism [6]. Several MPICH-V architectures with different features are developed.

Our work is based on FT-MPI [7] which is another important MPI fault tolerant implementation. It extends some of the semantics of MPI for allowing the application the possibility to survive process failures. FT-MPI provides four "communicator modes" to deal with the status of MPI objects after recovery and two "communication modes" to deal with on the fly messages. FT-MPI handles failures typically in three steps: 1) the run-time environment discovers failures; 2) all surviving processes are notified about these events; 3) the MPI library and the run-time environment are recovered, and then the application recovers its data itself. So, programmers are responsible for checkpoints recording in the fault-free mode and application data recovery after MPI system level recovery.

Apparently, disk write/read operations are the major source of overhead in on-disk checkpoint systems [8]. Diskless checkpointing technique, or so-called in-memory checkpointing, stores checkpoints in the main memory instead of hard disks. Because checkpoints are stored in the address space of the local process, process/node failures induce checkpoint loss. Generally erasure coding techniques are used to protect checkpoints. An erasure code for storage systems is a scheme that encodes the content on n data disks into m check disks so that the system is resilient to any m disk failures [11]. In checkpointing context, processes correspond to disks, n working processes (responsible for original HPC tasks and checkpoints maintenance) correspond to n

data disks, and *m* redundant processes (dedicated for check information maintenance) correspond to *m* check disks. It seems that diskless checkpointing introduces extra communication and computation overhead. But in fact, in order to tolerate node failures, on-disk checkpointing often relies on reliable distributed or parallel storage systems which generally are also based on erasure codes [8].

There are two kinds of in-memory checkpointing techniques: one treats checkpoint data as bit-streams [8] and the other treats checkpoint data as its original type [9] (generally floating-point numbers in scientific applications). The advantage of the former is that it can introduce erasure coding schemes smoothly and has no round-off errors. The advantage of the latter is that it is suitable for heterogeneous architectures and can eliminate local checkpoints for some applications. Our work uses the former strategy. However it can be translated into the latter strategy easily.

3 Erasure Codes

The most common used erasure codes in diskless checkpointing are mirroring and simple parity [8]. They are also the most popular codes used in storage systems known as RAID-1 and RAID-4/5 [12]. As the size of HPC systems becomes larger and larger, multi-erasure codes are necessary to achieve high reliability. But unfortunately, there is no consensus on the best coding technique for $n, m > 1$.

The best known multi-erasure code is Reed-Solomon code [13]. It is the only known multi-erasure code suitable for arbitrary *n* and *m*. RS code is based on Galois Field, thus the computational complexity is a serious problem. A real number field version of RS code has been used in diskless checkpointing [9].

Another kind of multi-erasure is so-called parity array codes. They arrange the data and parity symbols into an array, and divide symbols into overlapping parity groups, hence the name. Array codes are inherently XOR-based, thus are far superior to RS codes in computational complexity. In this paper, we focus on RDP (Row-Diagonal Parity) - a double-erasure horizontal code with dependent symbols [14]. "*Horizontal*" means that some disks contain only data symbols and the others contain only parity symbols. Its opposite - "*Vertical*" means that the data and parity symbols are stored together. Fig 1.a shows the 6-disk RDP code. A standard RDP code with $(p+1)$ disks can be described by a $(p-1)*(p+1)$ array. The parity groups are organized along horizontal and skew diagonal directions. D_{ij} denotes the data symbol that participates in the i^{th} horizontal parity P_i and the j^{th} diagonal parity Q_j. D_i denotes the data symbol

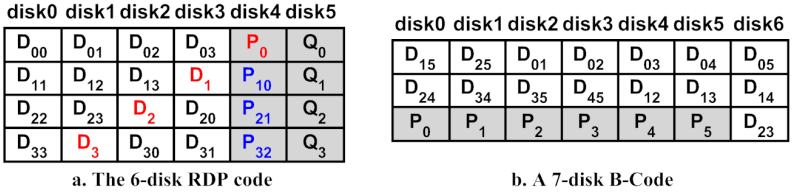

Fig. 1. RDP code and B-Code

that participates in the only i^{th} horizontal parity. P_{ij} denotes the horizontal parity that also acts as a data member of another diagonal parity Q_j - they are "dependent parity symbols". RDP achieves optimal encoding performance and good decoding performance. Note that p must be a prime number. But this limitation can be removed by deleting some data disks (assuming they contain nothing but zeros). Another advantage of RDP is that it has been generalized to more than 2 erasures [15].

Another array code used in our work is B-Code [16]. It's a double-erasure vertical code. B-Code has no prime size limitation because it is constructed by perfect one-factorizations (P1F) of complete graphs. There is a widely believed conjecture in graph theory: every complete graph with an even number of vertices has a P1F [17]. Fig 1.b shows a 7-disk B-Code. D_{ij} denotes the data symbol that participates in parities P_i and P_j. The number of disks p is an odd number. The last disk contains $(p\text{-}1)/2$ data symbols. Every other disk contains $(p\text{-}1)/2\text{-}1$ data symbols and 1 parity symbol. We use B_{2n+1} denotes this kind of B-Codes. Deleting the last disk produces a B-Code with even number of disks. This kind of B-Codes is denoted by B_{2n}. That is to say, B-Code exists for all sizes if P1F conjecture holds. B-Code achieves optimal encoding and decoding performance. Another advantage of B-Code is its inherent distributed structure. Because the parities are scattered over all disks, communication and computation workload are naturally distributed evenly.

We chose RDP and B-Code because they both have good encoding/decoding performance and perfect parameter flexibility - they can be applied to any number of MPI processes - this is obviously significant for the practice.

4 Scalable Checkpointing and Recovery Algorithms

4.1 Encoding and Decoding

FT-MPI system is responsible for failure detection and MPI environment rebuilding. We just need to add checkpointing and recovery functions into user applications. In our scenario, checkpointing and recovery are essentially erasure codes encoding and decoding respectively. So we first examine RDP and B-Code encoding/decoding.

RDP encoding is straightforward. The data symbols in the same row (with the same in-disk offset) are XORed into the horizontal parity symbol in the same row, and then the data symbols and the horizontal parity symbol in the i^{th} skew diagonal (the sum of the row index and the column index equals to i, $0 \le i \le p\text{-}2$) are XORed into the i^{th} diagonal parity symbol.

Unlike RDP, B-Code doesn't guarantee regular structure. So we can't determine the two parity symbols of a data symbol according to its row and column indices directly. We store the mapping from the data symbols to the parity symbols into a table g. $g[i, j, k]$ stores the index of the l^{th} parity symbols of the i^{th} data symbol in the j^{th} disk. Given a B-Code instance, we can calculate g easily. Therefore encoding is performed by XORing every data symbol into all of its parity symbols according to g.

Decoding is somewhat complicated. Both RDP and B-Code encoding can be described by a matrix-vector multiplication, therefore decoding is just a matrix inversion followed by a matrix-vector multiplication [18]. But this method inherently has

non-optimal computational complexity and is hard to be parallelized. A better way is chained decoding.

Single-erasures in a RDP coding system are easy to recover. If one of the parity disks fails, decoding is just (half) encoding. Otherwise, the failed data disk is recovered through the horizontal parity disk. Double-erasures involving the diagonal parity disk are also trivial - another failed disk is recovered through horizontal parity groups and then the diagonal parity disk is re-encoded. The most complicated double-erasures are those excluding the diagonal parity disk. In this case, the horizontal parity disk is regarded as a data disk. Observing Fig 1.a, we can see that each data disk except the first one touches only (p-2) diagonal parities (touching a parity means that contains symbols belonging to the parity), and every data disk misses different diagonal parity. Thus, a pair of data disks including the first data disk touches (p-2) diagonal parities twice and another diagonal parity once; a disk pair excluding the first data disk touches (p-3) diagonal parities twice and other 2 diagonal parities once. Anyway, parity group(s) losing only one symbol always exists. We can start decoding from this kind of parities, and then recover the lost symbols using horizontal and diagonal parity groups alternatively. For example, if a double-erasure (disk0, disk1) occurs in the RDP coding system shown in Fig 1.a, we recover D_{00} by XORing all surviving symbols in the parity group Q_0, then recover D_{01} using P_0, and then recover D_{11} using Q_1, and so on, finally recover D_3 using P_3. If a double-erasure excluding the first data disk occurs, decoding goes along two paths. Because RDP has a very regular structure, we can deduce starting points and directions of the decoding chains easily.

B-Code decoding also can be done in chained method. "1-missing" parities and decoding chains are determined according to the mapping table g. For example, the two decoding chains in the double-erasure (disk0, disk1) in the B-Code system shown in Fig 1.b are "$D_{34}\rightarrow D_{24}\rightarrow D_{25}\rightarrow D_{15}\rightarrow P_1$" and "$P_0$".

4.2 Checkpointing and Recovery Algorithms

The last section outlines the basic RDP and B-Code encoding/decoding methods. In fault-tolerant MPI context, they should be translated into a distributed style, particularly should be merged into MPI collective communication primitives. In this section, we only focus on checkpointing systems based on standard RDP codes and B_{2n}. Those based on shortened RDP codes and B_{2n+1} can be dealt with in a similar way. Moreover, we only concern with double-erasures excluding the diagonal parity disk, because other double-erasures and single-erasures are much easier.

A plain idea is appointing a root process (generally a redundant process or a re-spawned process) to execute all encoding/decoding computation. Fig 2.a depicts plain RDP checkpointing algorithm. The first redundant process gathers all checkpoint data from n working processes, then calculates horizontal and diagonal parities locally, and finally sends diagonal parities to the second redundant process. This algorithm can be transformed into recovery algorithm simply by exchanging the role of the re-spawned processes and the redundant processes and replacing local encoding by local decoding. Suppose each working process contributes s bytes checkpoint data and linear time gather algorithm [19] is used. Because per data word cost of RDP encoding/decoding is about 2 XOR operations, the time complexity of plain RDP

checkpoint/recovery algorithm is $O(sn)$. The algorithm also requires an extra buffer of size $O(sn)$ which is obviously induced by local encoding/decoding.

Fig 2.b shows the plain B-Code Checkpoint algorithm. The first working process collects checkpoint data, then encodes them into parities, and finally scatters parities over all processes. This algorithm also can be converted into recovery algorithm. The time complexity and extra memory requirement are $O(sn)$ too.

Apparently, the plain algorithms lead to serious load imbalance and unnecessary communication traffic. In order to distribute computation workload evenly and minimize communication traffic, a natural idea is embedding encoding/decoding into MPI collective communication operations. This idea is actually feasible. The basic operation in encoding and decoding is XORing data from all processes into parity at a single destination process. This is in fact a typical *all-to-one reduction* [19]. Therefore a RDP checkpointing can be accomplished by two successive reductions. Fig 3.a shows a checkpointing procedure in a fault-tolerant system based on the 6-disk RDP code. Horizontal parity reduction is trivial. Diagonal parity reduction requires a buffer preprocessing. The i^{th} process shifts its buffer to the up by i steps (with wraparound) to align data with parity. Because every process except the first one misses a diagonal parity, the corresponding area in the buffer is zeroed. If logarithmic time reduction algorithm [19] is used, the time complexity is $O(s\log n)$. Total extra memory requirement is also $O(sn)$, but per process extra memory requirement is only $O(s)$.

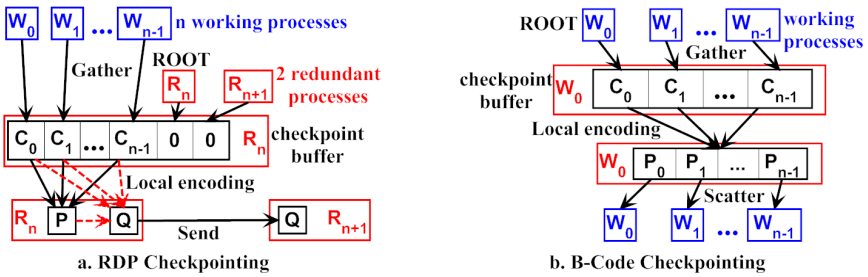

Fig. 2. Plain Checkpointing Algorithm

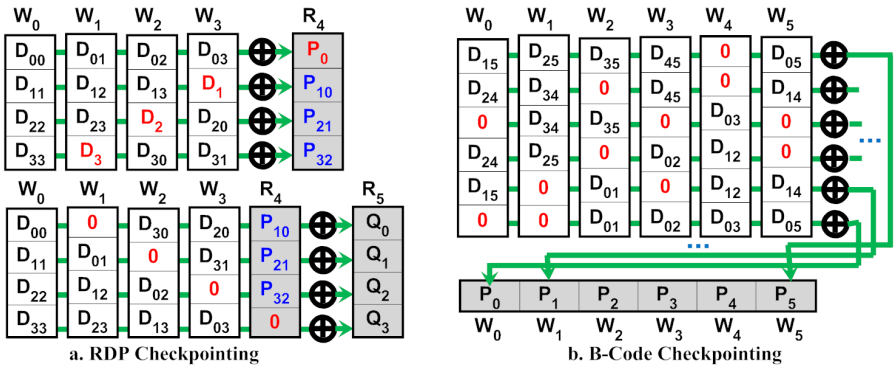

Fig. 3. Scalable Checkpointing Algorithm

Fig 3.b illustates the scalable B-Code checkpointing algorithm. This example is based on the B-Code shown in Fig 1.b. Buffers must be preprocessed too. Every data packet (data symbol) is duplicated and the two copies are aligned with its two parities respectively. The buffer areas corresponding to missing parities are zeroed too. Unlike RDP, B-Code uses *all-to-all reduction* [19] instead of all-to-one reduction because parities should be distributed across all processes. The time complexity is $O(s+\log n)$. Per process extra memory requirement is $O(s)$.

Designing scalable recovery algorithms has an obstacle - chained decoding is inherently a serial procedure. So we must reorganize the computation to exploit its potential concurrency. The notations so-called *syndromes* are used here. $\hat{P}_i$ and $\hat{Q}_i$ denote the XOR sum of all surviving symbols in P_i and Q_j respectively. They equal to the sum of all lost symbols in the same parity group. Observing the double-erasure (disk0, disk1) in a 6-disk RDP coding system, $\hat{Q}_0$ is just D_{00}, D_{01} is decoded by XORing $\hat{P}_0$ and D_{00}, $D_{00}=\hat{Q}_1$ XOR D_{01}, and so on. Although each step depends on the last one, but in fact syndromes can be calculated independently. So we can calculate all syndromes by two all-to-one reduction, then the root process (one of the re-spawned process) performs decoding locally (this step is not suitable for parallelization because of high communication overhead), and finally the root process sends recovered checkpoint data to another re-spawned process. This is an $O(s\log n)$ algorithm. The space complexity is $O(s)$. Scalable B-Code recovery algorithm is similar. But only one all-to-one reduction is executed to calculate syndromes.

5 Experimental Evaluation

We implemented our algorithms in FT-MPI. For RDP, the scalable algorithms were used. For B-Code, we chose the plain algorithms. We also implemented RAID-4 and RAID-5 based checkpointing for comparison. RAID-4 checkpointing and RAID-4/5 recovery were implemented by an all-to-one reduction. RAID-5 checkpointing was implemented by an all-to-all reduction.

We implemented in-memory checkpointing at the application level. We designed a common interface for erasure code based checkpointing. The checkpointing and recovery algorithms were implemented as helper functions. When checkpointing or recovery needs to be performed, the interface function packs user checkpoint data into a buffer first, and then calls the helper function to execute actual checkpointing or recovery. This framework simplifies new codes incorporating. We can simply implement a set of helper functions for a new code.

We mainly tested the performance overhead of our XOR erasure codes based fault tolerance approach using the Preconditioned Conjugate Gradient (PCG) application. The underlying PCG parallel program with FT-MPI is from Innovative Computing Laboratory (ICL) at University of Tennessee. Namely, it is just the one used in [9]. The input instance used is BCSSTK23 [20] which is a 3134*3134 symmetric matrix with 45178 nonzero entries.

All experiments were performed on a cluster of 8 single-core 3.06 GHz Intel Celeron nodes. The nodes are connected with a Gigabit Ethernet. Each node has 512MB of memory and runs Red Hat Enterprise Linux AS release 4 (Nahant Update 2). The FT-MPI runtime library version is 1.0.1. We ran the PCG program for 100000

iterations. For PCG with checkpoint, we took checkpoint every 25 iterations. That is to say, 4000 checkpoints were taken in each run. For PCG with recovery, we simulated a single or double process failure by killing the first process or the first two processes at the 50000-th iteration. MPI_Wtime was used to measure the execution time. Each data point is the average of 3 runs.

The purpose of the first set of experiments is to measure the performance overhead of taking checkpoints. For RAID-4, 6 working processes and 1 redundant process are invocated. For RDP, 6 working processes and 2 redundant processes are used. For RAID-5 and B-Code, 6 working processes are used. Fig 4 shows the execution time of PCG w/wo checkpoint, and the overhead of taking checkpoint. As expected, the single-erasure codes have lower overhead than the double-erasure codes. RAID-5 has higher overhead than RAID-4. The reason is that the checkpoint data size s is relatively small which favor all-to-one reduction than all-to-all reduction. The overhead of the plain B-Code algorithm is obviously higher than that of the scalable RDP algorithm. It seems that the checkpoint overhead of our approach is considerably worse than the result published in [9]. But please note, the checkpoint interval in our experiments is only about 30ms, while that in [9] ranges from 24s to 330s. It takes our algorithms 0.875ms~2.5ms to take a checkpoint. While the time of the single- and double-erasure algorithms presented in [9] ranges from 205ms to 500ms. After stripping out the impact of system size and input size, our bitwise, XOR-based approach is still superior to floating-point arithmetic coding approach used in [9].

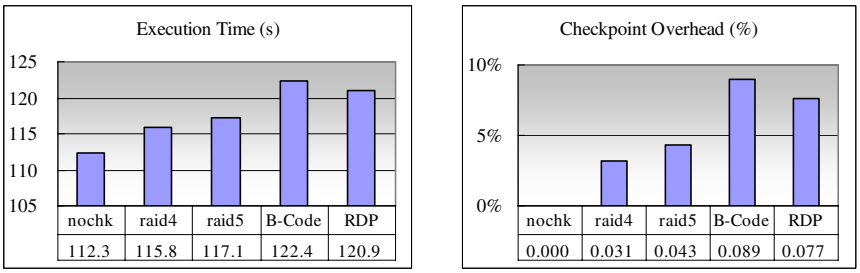

Fig. 4. PCG Checkpointing Overhead

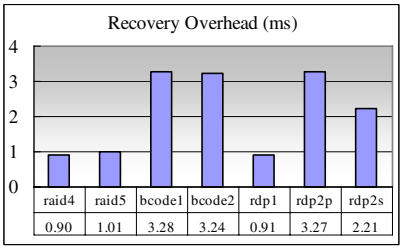

Fig. 5. PCG Recovery Overhead

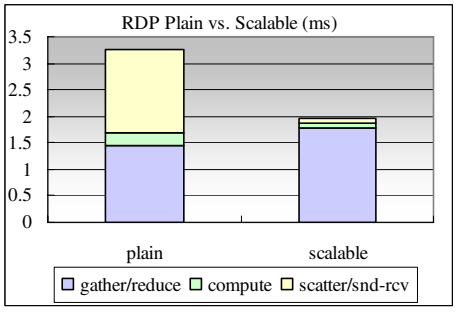

Fig. 6. RDP Plain vs. Scalable

Fig 5 shows the overhead of recovery. “bcode1” and “bcode2” denote single- and double-failure recovery based on B-Code respectively. “rdp1” denotes RDP based single-failure recovery. “rdp2p” and “rdp2s” denote RDP plain and scalable double-failure recovery respectively. We can see that the three single-failure recovery algorithms have the lowest overhead, because they only perform one all-to-one reduction. Compared with the plain algorithm, the scalable RDP double-failure recovery algorithm decreases overhead remarkably. It is also superior to the plain B-Code algorithm. It is worth while to note that our recovery algorithms are almost as fast as their checkpointing buddies. However the recovery algorithms presented in [9] are much slower than their corresponding checkpointing algorithms..

Fig 6 shows the detailed comparison between the plain RDP double-failure recovery algorithm and the scalable algorithm. Obviously, although the reduction step of the scalable algorithm is slightly slower than the gather step of the plain algorithm because the former undertakes numerous XOR operations, but the next two steps of the scalable algorithm are much faster than those of the plain algorithm.

6 Conclusion and Future Work

In this paper, we made a preliminary attempt to applied XOR-based erasure codes to in-memory checkpointing for MPI programs. Our main contributions include: 1) introduces two XOR-based double-erasure codes - RDP and B-Code into fault-tolerant MPI scenario; 2) incorporates encoding/decoding computation into MPI collective communication primitives. Our fault-tolerant approach is resilient to any two node/process failures. The experiments show that the scalable algorithms decrease communication overhead and balance computation effectively. Our approach is superior to related work in checkpointing and recovery overhead.

Evaluating our approach in larger systems using more different MPI applications is the principle work in the future. Optimizing computation and communication further is another important work. A possible way is to design new collective communication operations because some encoding/decoding processes conform to *all-to-k* patterns instead of standard all-to-one or all-to-all. Using multi-erasure codes to tolerate more than two failures is obviously a valuable work. In addition, translating our approach into floating-point arithmetic style will improve its applicability. For MPI applications with local communication patterns, we plan to try erasure codes with good locality.

References

1. `http://www.top500.org`
2. Wu-Chun, F.: The Importance of Being Low Power in High Performance Computing. Cyberinfrastructure Technology Watch Quarterly 1(3), 12–21 (2005)
3. Message Passing Interface Forum: MPI: A Message Passing Interface Standard. Technical report, University of Tennessee (1994)
4. Stellner, G.: CoCheck: Checkpointing and Process Migration for MPI. In: 10th International Parallel Processing Symposium, Honolulu, USA, pp. 526–531 (1996)

5. Agbaria, A., Friedman, R.: Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In: 8th IEEE International Symposium on High Performance Distributed Computing, Redondo Beach, California, USA, pp. 167–176 (1999)
6. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, pp. 1–18 (2002)
7. Fagg, G.E., Dongarra, J.: FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: 7th European PVM/MPI Users' Group Meeting, Balatonfüred, Hungary, pp. 346–353 (2000)
8. Plank, J.S., Li, K., Puening, M.A.: Diskless checkpointing. IEEE Trans. Parallel Distrib. Syst. 9(10), 972–986 (1998)
9. Chen, Z., Fagg, G., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., Dongarra, J.: Fault Tolerant High Performance Computing by a Coding Approach. In: 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, IL, USA, pp. 213–223 (2005)
10. Liu, X.G., Wang, G., Zhang, Y., Li, A., Xie, F.: The Performance Of Erasure Codes Used In FT-MPI. In: 2nd International Forum on Information Technology and Applications, Chengdu, China (2005)
11. Plank, J.S.: Erasure Codes for Storage Applications. Tutorial. In: 4th Usenix Conference on File and Storage Technologies, San Francisco, CA, USA (2005)
12. Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., Patterson, D.A.: RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys 26(2), 143–185 (1994)
13. Plank, J.S.: A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. Software - Practice & Experience 27(9), 995–1012 (1997)
14. Corbett, P., English, B., Goel, A., Grcanac, T., Kleiman, S., Leong, J., Sankar, S.: Row-Diagonal Parity for Double Disk Failure Correction. In: 3rd USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, pp. 1–14 (2004)
15. Blaum, M.: A Family of MDS Array Codes with Minimal Number of Encoding Operations. In: 2006 IEEE International Symposium on Information Theory, Washington, USA, pp. 2784–2788 (2006)
16. Xu, L., Bohossian, V., Bruck, J., Wagner, D.G.: Low-Density MDS Codes and Factors of Complete Graphs. IEEE Trans. on Information Theory 45(6), 1817–1826 (1999)
17. Colbourn, C.J., Dinitz, J.H., et al.: Handbook of Combinatorial Designs, 2nd edn. CRC Press, Boca Raton (2007)
18. Plank, J.S.: The RAID-6 Liberation Codes. In: 6th USENIX Conference on File and Storage Technologies, San Francisco, USA, pp. 97–110 (2008)
19. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison Wesley, Edinburgh Gate (2003)
20. `http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc3/bcsstk23.html`