An Efficient In-Memory Checkpoint Method and its Practice on Fault-Tolerant HPL

Xiongchao Tang, Jidong Zhai, Bowen Yu, Wenguang Chen, Weimin Zheng, and Keqin Li, *Fellow, IEEE*

Abstract—Fault tolerance is increasingly important in high-performance computing due to the substantial growth of system scale and decreasing system reliability. In-memory/diskless checkpoint has gained extensive attention as a solution to avoid the IO bottleneck of traditional disk-based checkpoint methods. However, applications using previous in-memory checkpoint suffer from little available memory space. To provide high reliability, previous in-memory checkpoint methods either need to keep two copies of checkpoints to tolerate failures while updating old checkpoints or trade performance for space by flushing in-memory checkpoints into disk. In this paper, we propose a novel in-memory checkpoint method, called self-checkpoint, which can not only achieve the same reliability of previous in-memory checkpoint methods, but also increase the available memory space for applications by almost 50 percent. To validate our method, we apply self-checkpoint method to an important problem: High-Performance Linpack (HPL) with fault tolerance. We implement a scalable and fault tolerant HPL based on this new method, called SKT-HPL, and validate it on two large-scale systems. Experimental results with 24,576 processes show that SKT-HPL achieves over 95 percent of the performance of the original HPL. Compared to the state-of-the-art in-memory checkpoint method, it improves the available memory size by 47 percent and the performance by 5 percent.

Index Terms—Fault tolerance, fault-tolerant HPL, in-memory checkpoint, memory consumption

1 Introduction

The substantial growth of system scale in High Performance Computing (HPC) makes fault tolerance increasingly important. A long-running HPC application can last for hours, or even days on an HPC system. Unfortunately, a large-scale system's mean time between failures (MTBF) may be too short to afford a complete fault-free run. For example, large-scale systems such as Blue Waters and Titan have failures every day [25], [29]. This problem becomes even worse as systems scale up towards exascale computing.

Significant research efforts have been made trying to overcome this problem. A series of algorithm-based fault-tolerant (ABFT) applications [8], [10], [34], [37] have been proposed. The main idea is to modify an application's algorithm for fault tolerance [21]. A recent study, called redMPI [17], employs a strategy of redundant execution to increase system reliability. All computation and communication are duplicated in redMPI. Thus, if there is any copy that survives a failure, the program can tolerate the failure and continue running. However, since everything is duplicated, the system efficiency is relatively low (no more than 50 percent).

Although ABFT and redundant execution provide a certain fault tolerance for HPC applications, their ability to tolerate faults highly depends on underlying runtime libraries. Message Passing Interface (MPI) is a *de-facto* standard for HPC applications. In the ABFT and redundant execution, it is assumed that MPI programs can be suspended after a *node failure*. A node failure means that a node is permanently lost, such as power down and network disconnect. Based on our observation, almost all current MPI implementations force the whole program to abort after a node failure is detected. On most MPI runtime, none of ABFT or redundant execution methods can tolerate a permanent node loss, which is quite common on a real HPC system [14], [30]. Some fault-tolerant MPI implementations [5], [12] support ABFT and redundant execution, with additional performance overhead.

Checkpoint-and-Restart (CR) is a classic strategy [13] that works under extreme cases such as a permanent node loss. Traditional CR methods save checkpoints to underlying storage systems, and leads to significant performance degradation for applications. Since memory has higher bandwidth than disks, in-memory/diskless CR methods effectively reduce the overhead of saving checkpoints, and have gained extensive attention [19], [28], [42].

Although the in-memory CR significantly reduces checkpoint overhead, it poses a new challenge for applications. Unlike disks, memory is a kind of relatively scarce resource. Checkpoints saved in memory occupy some space so that there is less available memory for applications. What's more, to maintain high reliability, in-memory checkpoint needs to maintain at least two copies of checkpoints [42]. A second checkpoint is used to tolerate failures when updating an old checkpoint. This results in only one third of

- *X. Tang, J. Zhai, B. Yu, W. Chen, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {txc13, yubw15}@mails.tsinghua.edu.cn, {zhaijidong, cwg, zwm-dcs}@tsinghua.edu.cn.*
- *K. Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.*

Manuscript received 26 May 2017; revised 6 Oct. 2017; accepted 30 Nov. 2017. Date of publication 8 Dec. 2017; date of current version 9 Mar. 2018.
(Corresponding author: Jidong Zhai.)
Recommended for acceptance by Y. lu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2781257

memory left for applications. A solution is to use memory as a cache of disks, and flush in-memory checkpoints into disks periodically. Several multi-level checkpoint frameworks have already been proposed [3], [25]. However, copying checkpoints from memory into disks involves disk operations, which is much slower than memory operations and introduce additional overhead. So multi-level checkpoint frameworks loses the performance advantage of in-memory checkpoint.

To address this problem, we propose a novel method for in-memory checkpoint, called *self-checkpoint*. Our method not only achieves the same reliability of in-memory checkpoint using two copies of checkpoints, but also increases available memory for applications by almost 50 percent. We use the term *workspace* to call the memory space where the original data is stored. The core idea of our method is to make the workspace of applications also as a checkpoint. For in-memory checkpoint methods, when updating a checkpoint, we need to copy data from the workspace of applications into checkpoint. It means that the data in the workspace and the updated checkpoint are the same, and both of them are in memory. Based on this observation, we propose a novel encoding mechanism, self-checkpoint. With the self-checkpoint, there is no need to save multiple checkpoints in memory, thus more memory is available for applications. Specifically, more available memory has different meanings to different programs. For some programs, more available memory means that the program can run for larger problem sizes with the same nodes. For some others, they can solve fixed-size problems with fewer nodes.

To verify and evaluate the self-checkpoint method, we apply it to a challenging problem, fault-tolerant HPL. *High-Performance Linpack (HPL)* is a prominent benchmark used in the TOP500 ranking list [1] of HPC systems. However, a future large-scale system may not afford a complete HPL test because of decreasing system reliability. Despite previous efforts on fault tolerant HPL, existing approaches either fail to tolerate a permanent node loss on a real system (algorithm-based fault tolerant methods) [34], [37], or introduce too much overhead (traditional CR methods saving checkpoints in disks) [13], [32], thus they are not practical for a performance benchmark.

In-memory checkpoint is a promising solution for fault tolerant HPL. However, HPL has several characteristics that make it even more difficult for in-memory checkpoint. First, the memory usage of HPL is configurable, and generally larger memory is much better for performance. For this reason, HPL needs as much memory as possible for high performance, while checkpoint itself should use as little memory as possible. Second, as the consequence of high memory usage, it will take a long time to flush checkpoints from memory into disks. So multi-level checkpoint methods are not suitable for fault-tolerant HPL. Third, HPL has a big memory footprint. Almost every byte is modified between two checkpoints. As a result, incremental checkpoint methods [2], [27], [33] are not efficient for this problem.

To this end, we implement a fault-tolerant HPL based on the self-checkpoint mechanism, called SKT-HPL.[1] It can not only achieve very high performance but also tolerate a permanent node loss. We evaluate SKT-HPL on two large systems, Tianhe-1A and Tianhe-2 (ranked TOP#2 in TOP500 list). Experimental results show that with 24,576 processes on Tianhe-2, the self-checkpoint method improves the available memory size by 47 percent. SKT-HPL achieves over 95 percent of the original HPL's performance, and 5 percent higher than using previous in-memory checkpoint methods. We also perform powering-off experiments to validate SKT-HPL can tolerate a real node failure.

In summary, we make the following contributions in this work.

- We propose a novel in-memory checkpoint method, self-checkpoint, which can not only keep the same reliability of in-memory checkpoint using two copies of checkpoints, but also significantly increase the available memory space of applications by almost 50 percent.
- We apply our proposed self-checkpoint method to an important problem, fault-tolerant HPL. We implement a scalable and node failure tolerant HPL (SKT-HPL) on real HPC systems.
- We evaluate SKT-HPL on two large-scale systems, Tianhe-1A and Tianhe-2. Results show that SKT-HPL achieves over 95 percent of the original HPL's performance and improves the memory usage over the state-of-the-art in-memory checkpoint by 47 percent.

A preliminary version of this work has been published in PPoPP [31].

2 Basic In-Memory Checkpoint Framework

In this section, we discuss the model of failure and recovery used in this paper. Also, we describe some general fundamental techniques for in-memory checkpoint methods. Our work is constructed on this basic framework.

2.1 The Model of Failure and Recovery

In this paper, a node means a physical machine. Nodes are connected through network. An MPI parallel program consists of many processes, which are running on many nodes. There are multiple processes running on a single machine. In our model, a node failure is that a node loses connection with others, thus the processes running on it are also lost. For most MPI programs, a node failure leads to the abortion of the whole program, i.e., all processes on all nodes abort. The task of recovery is to restart all processes and continue their work. Although we need to restart all processes, we only need to restart failed nodes or replace them by spare nodes. Other nodes or *healthy nodes* require no additional operation. Fig. 1 shows the model of failure and recovery.

2.2 Protecting Data with Encoding

For in-memory checkpoint methods, since checkpoints are saved in volatile memory, an error-correcting code is necessary to encode in-memory checkpoints. Calculating error-correcting codes for all processes in a large-scale system is prohibitive due to large communication overhead, so we apply a group encoding strategy [3], [25], [39], [40] to reduce the communication overhead. We partition all the processes into a number of small groups and build error-correcting codes for each group separately. To further reduce

[1] Source code and documents of self-checkpoint can be downloaded from https://github.com/thu-pacman/self-checkpoint.git

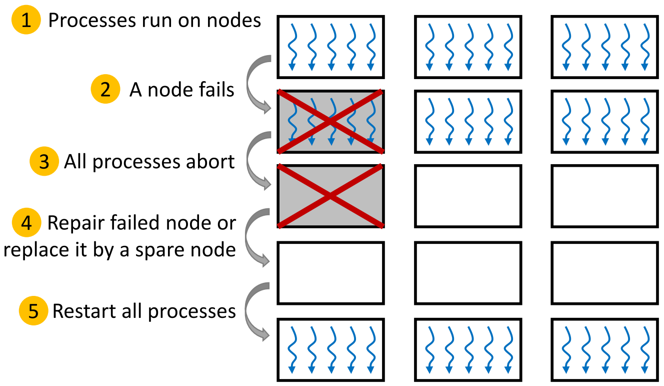


Fig. 1. The model of failure and recovery. A node is a physical machine which many processes run on. All processes abort after a node failure.

communication contention during building error-correcting codes within a group, we perform a stripe-based encoding for each process. The basic idea is that we partition each process data into $N-1$ stripes (each group has N processes), and then each process of the group is in charge of building an error-correcting code for partial stripes. This method can effectively avoid single-node network contention during encoding.

Fig. 2 shows an example to illustrate the above encoding method used in our system, which is similar to the encoding mechanism used in RAID-5 [26]. Suppose that there are four processes, P_0, P_1, P_2, and P_3, in each group. Each process partitions its local data into three stripes and allocates four empty slots. After filling its own three stripes into the slots, each process has an empty slot for the checksum of stripes from the other processes within the same group. For example, we calculate the checksum of A_1, A_2, and A_3 from P_1, P_2, and P_3, then store this checksum A_S into the empty slot of P_0. Other checksums, B_S, C_S, and D_S, are built in the same way. A general encoding method is listed below

$$X_S = X_1 + X_2 + \cdots + X_{n-1}. \tag{1}$$

X is a stripe in each process. The operator "+" can be either a numerical sum or a logical exclusive-or. Note that the encoding time for each group does not change with the system scale and only depends on the group size, which makes our approach more scalable for a large-scale system.

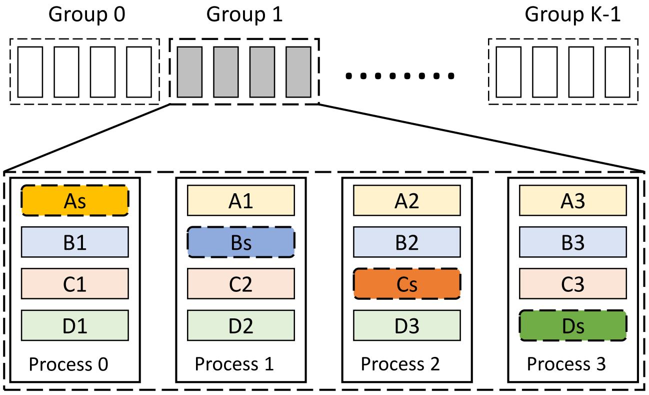


Fig. 2. Data encoding. Processes in a large-scale application are partitioned into a number of small groups. We use a stripe-based encoding in each group. A_S, B_S, C_S, and D_S in dashed dark blocks are checksums for A_i, B_i, C_i, and D_i respectively.

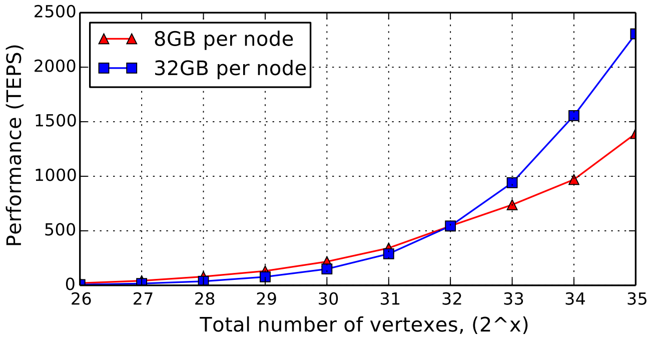


Fig. 3. Performance of distributed BFS with small and big memory configurations on Sunway-TaihuLight. The big memory configuration has $4\times$ larger memory space than the small configuration, for each computing node (8GB versus 32 GB on each node). In the largest case (vertexes=2^{35}), the small memory configuration is run with 32,768 processes while the big configuration is just run with 8,192 processes. For the same problem size, $1/4$ nodes with more memory can obtain much better performance than more nodes with less memory on each node.

Using above encoding, each group can tolerate a single node failure. We can apply more complex encoding methods, such as RAID-6 [23] and Reed-Solomon [35], to tolerate more node failures. For a higher degree of fault tolerance, in-memory checkpoint methods can also be combined with a multi-level checkpoint framework [3], [11], [25].

2.3 Keeping Checkpoints in Memory

Data saved in in-memory checkpoints needs to be accessible after applications restart. However, in most modern operating systems like Linux and Windows, a normal memory region will be freed after its owner exits. The data saved in the freed memory region is no longer accessible. To keep the data always in memory, one method is to mount an in-memory file system (e.g., `ramfs` and `tmpfs` in Linux) and write the checkpoint into that file system. Alternatively, Linux provides a shared memory mechanism (SHM).[2]

A memory region allocated through SHM will not be automatically freed by OS, even though no process is attached to it. In our framework, we use the shared memory mechanism to allocate memory regions for checkpoints. With this mechanism, all the checkpoint data in healthy nodes is still accessible after a node failure.

3 Importance of More Available Memory

The core idea of self-checkpoint is to use less memory for fault-tolerance, and leave more memory for applications. As mentioned in Section 1, more available memory has different meanings to different applications. In extreme cases, available memory space determines whether an application can be launched. For some applications, they can solve fixed-size problems with fewer nodes. Solving a problem with less nodes has potential performance benefits for network-intensive applications. For some others, more available memory means that programs can run for larger problem sizes with the same nodes.

Distributed breadth-first search (BFS) is an example of the former case. As shown in Fig. 3, if we run BFS on less

[2] More information about SHM can be found at http://man7.org/linux/man-pages/man2/shmget.2.html

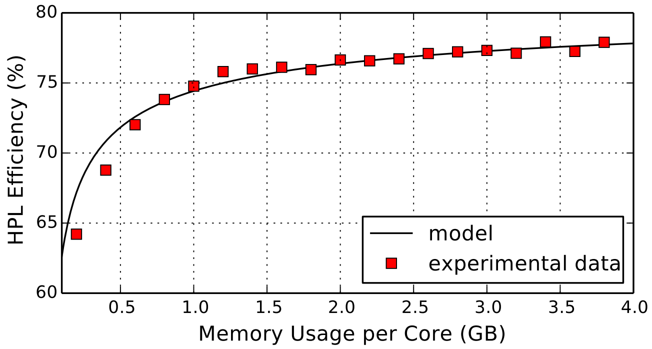


Fig. 4. Fitting of the efficiency model with experimental data. Experiments are conducted with 192 MPI ranks on a local cluster with Xeon E5-2670(v3) and 100 Gbps Infiniband network.

nodes with sufficient memory, then the performance can be much better than on more nodes but each with less memory. The data are obtained with *big* and *small* memory space configurations on Sunway-TaihuLight supercomputer. The *big* configuration provides 32 GB memory on each node, which is $4\times$ larger than the *small* configuration (8 GB on each node). With a *big* configuration, the number of nodes is only $1/4$ of with *small* configuration, so the network communication overhead is significantly reduced. Fig. 3 shows that more available memory can lead to much performance improvement.

HPL belongs to the latter case. It has an adjustable problem size, thus larger problems can be solved with more available memory. Also, HPL has a characteristic that it can achieve better performance with a larger problem size. Therefore, we use HPL as an example to show the potential performance benefits of more available memory.

In the rest of this section, we present the derivation of HPL efficiency model and give our observation on the relationship between available memory space and HPL efficiency. Our model indicates that more available memory leads to higher performance in HPL. This also confirms our opinion that an in-memory checkpoint method should occupy as little space as possible.

For a given system, the *efficiency* of HPL is a ratio between HPL test performance and the system's theoretical peak performance. For example, if the test performance of HPL is 80 GFlops and the system's theoretical peak performance is 100 GFlops, the efficiency is 80 percent.

The kernel of HPL is to solve a dense linear equation $Ax = b$, where A is an $N \times N$ matrix. The computational complexity of HPL is $O(N^3)$ and its communication volume and memory access are $O(N^2)$. Therefore, if we omit the linear and constant computational work in HPL, the main work of HPL can be modeled by an $O(N^3)$ part plus an $O(N^2)$ part.

Since the total computational work in HPL is $O(N^3)$, the theoretical execution time without including any communication or memory access overhead can be modeled as γN^3. Considering various performance overhead and loss, the actual execution time of HPL can be modeled as $\alpha N^3 + \beta N^2$, where $\alpha > \gamma$. And βN^2 is an estimation for memory access and communication overhead. Therefore, the HPL efficiency $E(N)$ can be calculated as follows:

$$E(N) = \frac{\gamma N^3}{\alpha N^3 + \beta N^2} = \frac{N}{aN + b}. \tag{2}$$

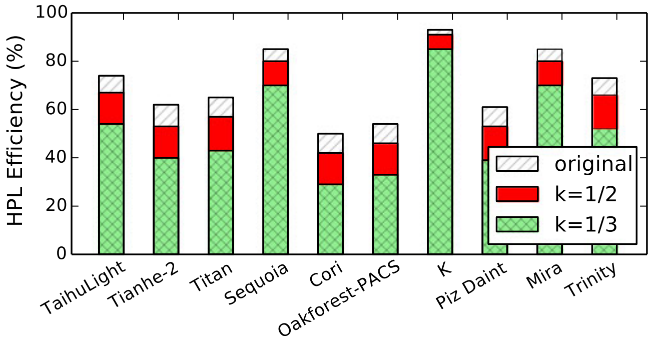


Fig. 5. Modeled HPL efficiency of the top 10 supercomputers with different available memory space. The green mesh bars represent officially reported performance. The hatched and red bars are results only using one third and half of the memory size respectively, drawn by our model.

Here we have $a = \alpha/\gamma > 1$ and $b = \beta/\gamma$. In this model, given an invariant system and fixed process mapping, the parameters a and b are independent of the problem size.

Fig. 4 shows that this model fits well with real experimental data on a local cluster. The dots in Fig. 4 represent the real data when executing 192 MPI ranks on a local cluster with different problem sizes. The line is fitted with our model. In Section 7.5, we will validate this model in two larger systems.

From above efficiency model, we can get an interesting finding for the HPL performance. For a given system, the HPL performance increases with the input problem size.

Next, we analyze the efficiency of HPL when reducing available memory space. Suppose that a system has efficiency e_1 for the problem size of N_1 with the total available memory, then we have

$$e_1 = E(N_1) = \frac{N_1}{aN_1 + b}. \tag{3}$$

Thus the value of b is

$$b = \frac{(1 - ae_1)N_1}{e_1}. \tag{4}$$

If only partial memory is available for HPL, and ratio is k $(0 < k < 1)$, then the problem size N_2 will be $\sqrt{k}N_1$. Combining Equations (2), (3) and (4), we have the efficiency for the problem size of N_2 as

$$e_2 = \frac{\sqrt{k}e_1}{1 - (1 - \sqrt{k})ae_1} > \frac{\sqrt{k}e_1}{1 - (1 - \sqrt{k})e_1}. \tag{5}$$

The last step is because $a > 1$. Equation (5) gives a lower bound of HPL efficiency for different problem sizes.

To make the relationship between available memory space and HPL efficiency more clear, Fig. 5 shows the efficiency of HPL for the top 10 supercomputers in the latest TOP500 list with different available memory space according to our model. We can find that these systems can achieve higher performance using more memory and improve 11.96 percent of the efficiency on average from one third of the memory to half of the memory. In consequence, it would be much better if an in-memory checkpoint method uses less memory space and leaves more for applications.

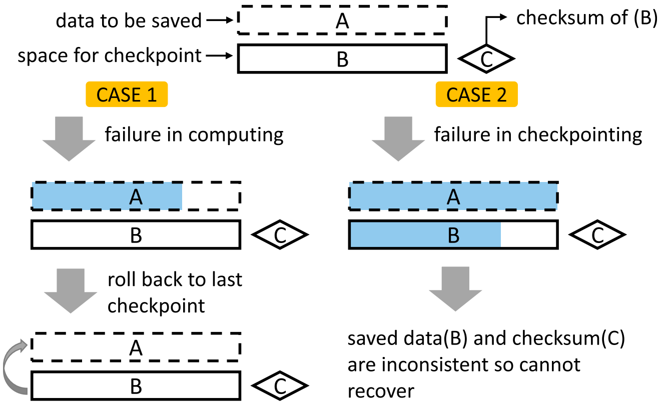


Fig. 6. The strategy of single in-memory checkpoint. This method cannot recover data from a failure during checkpoint updating. Shading means a part of memory space is updated to newer data.

4 Self-Checkpoint Mechanism

4.1 Methodology

In this section, we elaborate our proposed self-checkpoint mechanism. To demonstrate its advantage over previous approaches, we first give a short introduction to single checkpoint and double checkpoint mechanisms.

Fig. 6 shows the strategy of single in-memory checkpoint. The main advantage of the single checkpoint strategy is its low memory consumption. Almost half of the memory can be used for useful computation. The user's data of the original program is represented by rectangle A. B stands for the memory space for checkpoint, and C is the checksum of B. The single checkpoint strategy can handle a node failure during the program's computation by rolling back the program using the checkpoint B and checksum C (CASE 1 in Fig. 6). However, the single checkpoint cannot tolerate a node failure while updating a checkpoint or a checksum (CASE 2 in Fig. 6), because at this time, the checkpoint B and checksum C are in an inconsistent state.

To tolerate the node failure during checkpointing, a straight-forward solution is to maintain two copies of checkpoints and overwrite the old one when making a new checkpoint, as shown in Fig. 7. Since there are two checkpoints and at least one is available when updating, this strategy can tolerate a node failure at any time. This strategy is widely used in traditional checkpoint methods, which use disks and have sufficient storage space for checkpoints. The state-of-the-art in-memory checkpoint systems [41], [42] also adopt this strategy. However, this strategy has high memory consumption and significantly reduces available memory space for useful computation (only 1/3), which is prohibitive in a system with limited memory resource. Another solution is to flush the old checkpoint into a permanent device like hard-disks, but it will lose the performance advantage of in-memory checkpoint.

To address above problems, we propose a novel *self-checkpoint* strategy, which can significantly increase the available memory space while tolerating a node failure during checkpointing. Instead of maintaining two copies of checkpoints as shown in Fig. 7, we store two copies of checksums in memory. This method is inspired by an observation that a checksum is much smaller than a checkpoint in a group encoding method. More specifically, a checksum is only $1/(N-1)$ of the checkpoint size when a group has N processes. Fig. 8 illustrates an ideal case using the self-checkpoint strategy to handle different situations of failures.

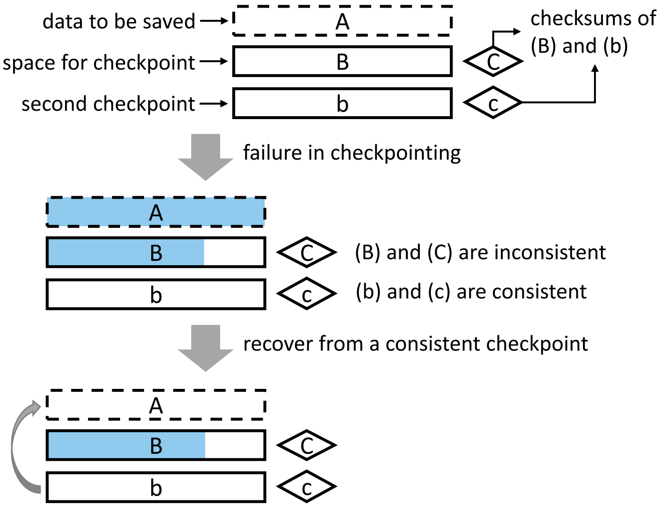


Fig. 7. The strategy of double checkpoints. While updating checkpoints, at least one checkpoint and its checksum which are consistent can be used for recovery. Its main disadvantage is that too much memory space is wasted. Shaded areas are updated to newer data.

When making a new checkpoint, a checksum D is first calculated for the user's data A, and then the memory space of A and D is flushed into B and C. As shown in CASE 1 of Fig. 8, when a node failure is detected while calculating the checksum D, we can recover the program from checkpoint B and checksum C. How do we recover the program if a node failure is incurred while updating checkpoints? As shown in CASE 2 of Fig. 8, we can recover the program using the user's data A and its checksum D. *In other words, the memory space for computation itself is served as a checkpoint*. Therefore, we call our proposed method a self-checkpoint mechanism.

As discussed in Section 2.3, we use shared memory to keep data in memory. In principle, all variables of a program can be allocated in shared memory. However, there are some strategies to reduce the modification of source code. Big arrays in C/C++ code are often allocated in heap, by calling library functions. Therefore, a function wrapper can be used to easily change the locations of these variables. For example,

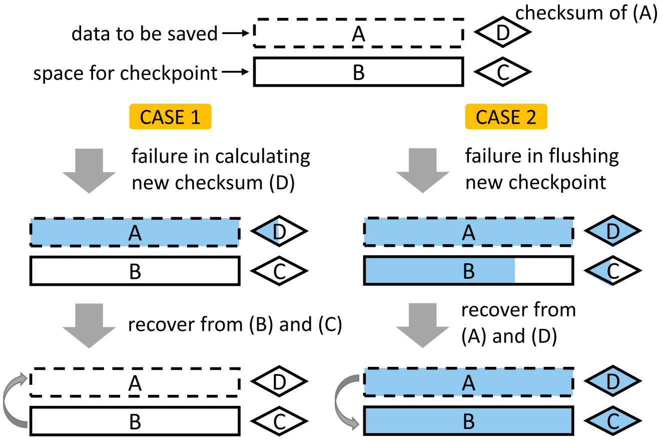


Fig. 8. The strategy of self-checkpoint. In CASE 1, if there is a failure during calculating a new checksum, the program can recover from B and C. In CASE 2, if there is a failure while updating checkpoints, the program can recover from A and D.

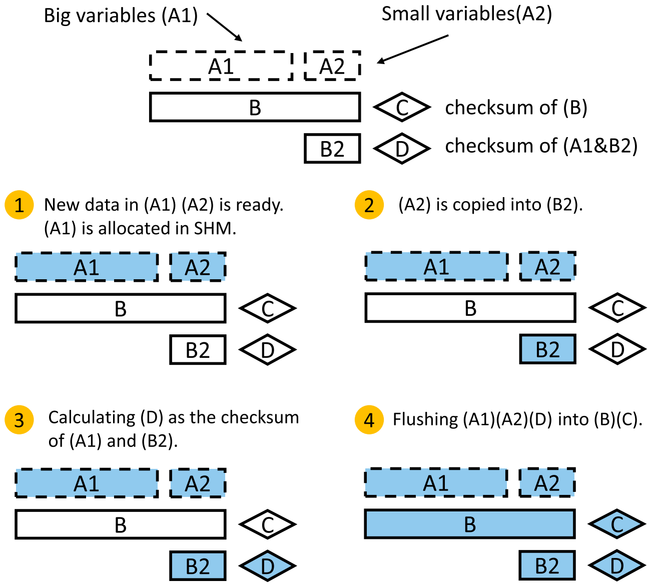

Big variables (A1)
Small variables(A2)
A1
A2
B
C
checksum of (B)
B2
D
checksum of (A1&B2)
1 New data in (A1) (A2) is ready.
(A1) is allocated in SHM.
2 (A2) is copied into (B2).
3 Calculating (D) as the checksum of (A1) and (B2).
4 Flushing (A1)(A2)(D) into (B)(C).

Fig. 9. The complete workflow of self-checkpoint. Most data is allocated in the shared memory, represented by $A1$ and little data is stored in the user's space, represented by $A2$. D is the checksum of $A1$ and $B2$.

a malloc statement `p=malloc(...)` is transferred into an SHM statement `p=shmget(...)`. A program may also have some small variables allocated in stack, such as loop iterators and temporal variables. It can introduce a lot of work to change all the code to use shared memory. Therefore, as a trade-off, small local variables ($A2$ in Fig. 9) are still allocated in regular memory. We use a small second-buffer ($B2$ in Fig. 9) to save their copies for simplicity.

Fig. 9 illustrates the complete workflow of the self-checkpoint mechanism. $A1$ denotes the data which is stored in the shared memory. $A2$ denotes the data which is stored in the user's space and we need additional checkpoint space for $A2$, but the size of $A2$ can be much smaller than $A1$. The workflow of the self-checkpoint mechanism is listed as below:

1) Store most of the user's data $A1$ in shared memory.
2) Copy the data of $A2$ into $B2$.
3) Calculate D, which is the checksum of $A1$ and $B2$.
4) Copy ($A1$, $A2$) into B, and copy D into C.

Before the last step, we can recover the program from the old checkpoint B and checksum C. Once we get the checksum D, the program can be recovered from $A1$, $B2$, and D. In summary, a single node failure can always be tolerated with our proposed self-checkpoint mechanism.

More detailed analysis about the memory usage is discussed in Section 4.2. Section 7 gives the detailed performance data of the self-checkpoint mechanism.

4.2 Analysis of Memory Usage

Selecting a suitable strategy for group partitioning is important for the performance of the self-checkpoint mechanism. In this section, we discuss the relationship between the group size and memory usage for different in-memory checkpoint methods.

Suppose that each group has N processes, and each process needs M units of memory for the user's computation. In the self-checkpoint mechanism, as we save most of data in the shared memory ($A1$ in Fig. 9) and only partial local variables in the user's space ($A2$ in Fig. 9), so the size of $A2$ and $B2$ is negligible.

TABLE 1
The Memory Usage of the Self-Checkpoint Mechanism for Each Part in Fig. 9

Item	$A1$+$A2$	B	C	D	Total
Size	M	M	$\frac{M}{N-1}$	$\frac{M}{N-1}$	$\frac{2MN}{N-1}$

The group size is N.

Table 1 lists the memory usage for each part of the self-checkpoint mechanism. Each process uses M of memory size for the original work ($A1$ and $A2$ in Fig. 9) and B is also M. Due to the negligible size, B2 is omitted for simplicity. According to the encoding method described in Section 2.2, a checksum has the size of $M/(N-1)$. Therefore, the total memory usage of the self-checkpoint mechanism is the sum of $A1$, $A2$, B, C, and D, so it is $2MN/(N-1)$. The available memory for application with the self-checkpoint mechanism is

$$U_{self} \approx \frac{M}{2MN/(N-1)} = \frac{N-1}{2N}. \tag{6}$$

Similarly, we can calculate the available memory space of the double checkpoint method shown in Fig. 7

$$U_{2ckpt} = \frac{M}{M+2MN/(N-1)} = \frac{N-1}{3N-1}. \tag{7}$$

And the available memory with the single checkpoint shown in Fig. 6 is

$$U_{single} = \frac{M}{M+MN/(N-1)} = \frac{N-1}{2N-1}. \tag{8}$$

We illustrate the available memory space of different in-memory checkpoint methods with several typical group sizes in Fig. 10. The single checkpoint has the least memory consumption and most available memory space among three methods, but it is not fully fault tolerant and cannot recover from a failure during checkpointing updating. The double checkpoint method is fully fault tolerant but has much less available memory (less than 1/3). The available memory of our self-checkpoint method is slightly less than the single checkpoint but much higher than the double checkpoint. At the same time, our method is fully fault tolerant. For a large

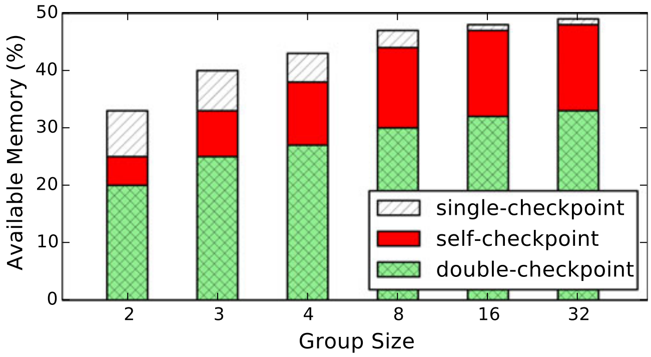


Fig. 10. The memory usage of different in-memory checkpoint methods with group sizes of 2, 3, 4, 8, and 16.

group size, our method has more available memory space than the double checkpoint, up to nearly 50 percent.

4.3 Grouping Strategy

In this section, we give our strategies in determining suitable group partitioning in a large-scale system. From Fig. 10, we can find that a larger group size always has more available memory, but a larger group is not good for the data encoding. The communication time during the data encoding is positively correlated with the group size (at least $O(log(N))$ for N nodes). Therefore, a smaller group size introduces much lower communication overhead. Furthermore, a small group size is also beneficial for the system's reliability. Currently, our system only tolerates a single node failure in each group. The more processes a group has, the more likely more than one process will fail. In an extreme case, if a group includes the whole system, only a single failure can be tolerated. If each group has only two processes, the system can tolerate failures for half of the processes at the same time.

As a consequence, a large group is good for the memory space, but increases the communication overhead and is more likely to fail. Conversely, a small group encodes the checkpoint quickly, but less memory space is left for useful computation. As shown in Fig. 10, The available memory of a group with 16 processes is 47 percent and is close to the upper bound of 50 percent. We find that in a large system a larger group size provides little benefit for available memory space but causes much overhead in communication. As a result, we select the group size of 16 for our experiments .

For better performance and reliability, an appropriate process mapping strategy should be considered. To tolerate a permanent node loss, processes within a group must be distributed onto different physical nodes. At the same time, for better communication performance, a group tends to select some neighboring nodes. But for high reliability, a group should also spread its nodes as far as possible to tolerate a single rack or switch failure. Based on previous studies [14], [30], as most failures in HPC systems are single node failures, and rack and network failures are minor, we give high priority to the performance in our current grouping strategy. Exploring more mapping strategies within one group is left for future work.

4.4 Supporting Accelerators

Accelerators like Graphics Processing Units (GPU) and Xeon Phi are increasingly important for high performance computing. There are no fundamental obstacles to use self-checkpoint on a supercomputer with accelerators. However, since we cannot assume all accelerators support the shared memory mechanism, self-checkpoint does not save data inside the memory of an accelerator.

For many applications, accelerators are used to solve some critical kernels, and data are transferred between main memory and accelerators over iterations. For these applications, self-checkpoint can be used without extra code modification. If an application has some data remaining in accelerators during a whole run, then we need to explicitly transfer data back to main memory for fault tolerance.

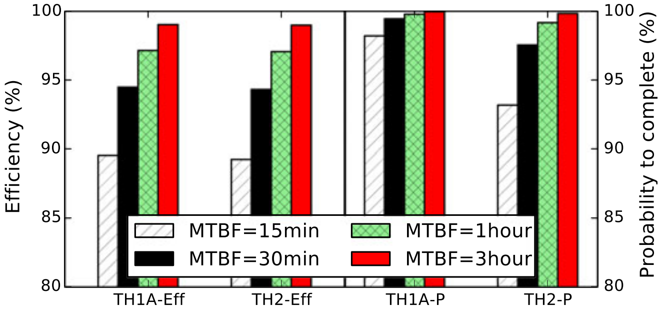


Fig. 11. The left side shows efficiency compared to fault free case. The right side is probability to complete successfully. All results use 256K nodes under different MTBFs. Even under a short MTBF, self-checkpoint still has high efficiency and high reliability.

5 Theoretical Analysis for Large Scale

In this section, we give a theoretical analysis for the performance and the reliability of applications using self-checkpoint that run on an even larger HPC system under short MTBF.

5.1 Performance Under Different MTBFs

We use a checkpoint interval based on the following estimation formula [9]:

$$\tau = \sqrt{2M\delta}\left[1 + \frac{1}{3}\left(\frac{\delta}{2M}\right)^{\frac{1}{2}} + \frac{1}{9}\left(\frac{\delta}{2M}\right)\right] - \delta, \tag{9}$$

where δ is the time cost for a checkpoint, and the optimal interval to next checkpoint is τ.

Based on that formula, we calculate the total time used for checkpoints during a whole run, called T_c. We call the time for an application without any failure T_h. If MTBF is M , there will be $(T_h + T_c)/M$ failures during the test, and each recovery costs $R + \delta$, where D is the failure detection time. The efficiency is

$$eff = \frac{T_h M}{(T_h + T_c)(M + D + \delta)}. \tag{10}$$

We analyze the cases in which an application runs on a system with 256,144 nodes and MTBF=15 min, 30 min, 1 hour, and 3 hours. The results are shown in Fig. 11. The left part of Fig. 11 illustrates the relative performance under different MTBFs, compared to a fault-free case. The overhead introduced by the checkpoints is less than 10 percent when MTBF is longer than 30 minutes. As a redundant execution uses half of the processors, its efficiency is no more than 50 percent, which is far lower than ours.

5.2 Reliability Analysis of SKT-HPL

In this section, we analyze the reliability of the self-checkpoint and compare it with the single-checkpoint. We assume that the failure obeys a *Poisson-Process*, then the time between two failures x follows an exponential distribution

$$f(x) = \lambda e^{-\lambda x}, \lambda = \frac{1}{MTBF_{system}}. \tag{11}$$

If the whole system has N_t processes and a group has N_g processes, the MTBF of a single group is

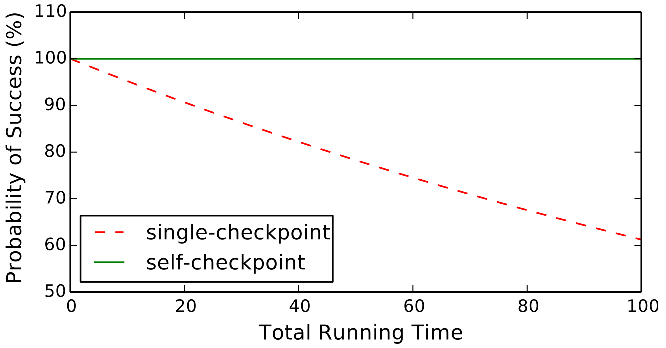


Fig. 12. Simulation results for the reliability of the self-checkpoint and single-checkpoint. Results show that the self-checkpoint is more reliable for a long-running program while its available memory is as much as using single-checkpoint.

$$MTBF_{group} = \frac{N_t}{N_g} MTBF_{system}. \tag{12}$$

The probability of a failure occurs during checkpoint updating is

$$P_1 = 1 - e^{\alpha}, \alpha = -\frac{checkpoint\_time}{MTBF_{system}} \times checkpoints. \tag{13}$$

A fault-tolerant application using the single-checkpoint method cannot handle this situation. Although the self-checkpoint can tolerate a failure during checkpoint updating, it fails if a second failure occurs in the same group during recovery. The probability of this situation is

$$P_2 = 1 - e^{\beta}, \beta = -\frac{recovery\_time}{MTBF_{group}} \times failures. \tag{14}$$

Therefore, the success probabilities for the single-checkpoint and self-checkpoint are

$$P_{single} = (1 - P_1) \times (1 - P_2) = e^{\alpha+\beta} \tag{15}$$

$$P_{self} = (1 - P_2) = e^{\beta}. \tag{16}$$

We consider a situation with following configurations: The whole system has $N_t = 160,000$ processes and each group has processes $N_g = 16$. Based on the data of Fig. 19, both the checkpoint time and recovery time are assumed to be 20 seconds; Checkpoints are done every 13.6 minutes, based on the estimation method of [18]; A failure occurs every $MTBF_{system}$, which is 5 hours. A simulation result is presented in Fig. 12.

Based on above simulation results, the self-checkpoint has a probability of over 99 percent to successfully complete an application executing for 100 hours. This means that an application can successfully finish in most cases. In contrast, an application with the single-checkpoint has a probability of 39 percent to fail for such a long-running application. In conclusion, the self-checkpoint achieves high reliability while keeping large available memory space.

6 SKT-HPL Implementation

In this section, we discuss some details of implementing SKT-HPL on a large-scale system.

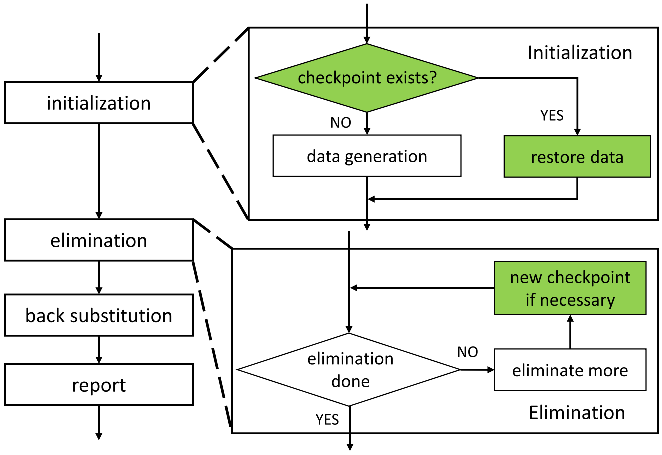


Fig. 13. The workflow of SKT-HPL. Checkpoints are made at the end of a certain iteration during the elimination step. If a node failure is detected, SKT-HPL restores the data from the checkpoint in the initialization step.

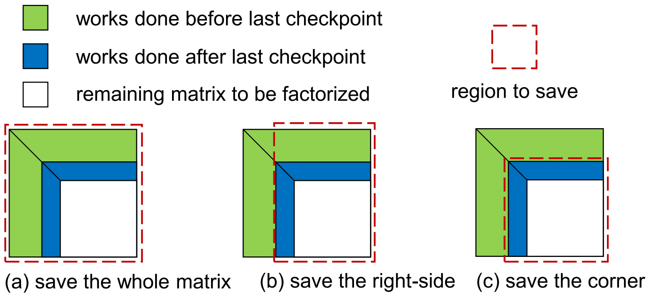
6.1 SKT-HPL Overview

The kernel of HPL is to solve a linear equation $Ax = b$, where A is an $N \times N$ matrix. This equation is solved by Gaussian Elimination with Partial Pivoting (GEPP). In general, HPL can be divided into the following four steps:

1) *Initialization.* The first step is to initialize MPI runtime, generate and partition data. The coefficient matrix A and vector b are allocated at runtime and filled by random numbers.
2) *Elimination.* The original equation $Ax = b$ is then transformed into an upper triangle equation $Ux = y$. This is the most time-consuming part in HPL, and its computational complexity is $O(N^3)$. This step is done in a loop that iterates over all rows of A.
3) *Back Substitution.* This step finally obtains the solution of $x = U^{-1}y = A^{-1}b$. Solving the upper triangle equation $Ux = y$ is much easier, and the computational complexity is only $O(N^2)$.
4) *Report.* After solving above equation, HPL verifies the solution of x and then reports the final performance. The computational complexity of this step is $O(N^2)$.

Fig. 13 illustrates the workflow of SKT-HPL. The white blocks stand for operations in the original HPL as described above, and the shaded ones are added by SKT-HPL. SKT-HPL makes checkpoints during the elimination step at the end of loop iterations, the main computational step in HPL. After a node failure is detected, some necessary data structures need to be rebuilt in the initialization step and SKT-HPL can restore the program's data from the checkpoint. SKT-HPL do not make checkpoints for the other steps, because they normally take far less time than MTBF and even less than a typical checkpoint interval.

Although the current implementation of SKT-HPL does not use accelerators, self-checkpoint can be used to implement fault-tolerant HPL that supports accelerators. Accelerators usually have data in their own memory, but operations like checkpointing and recovering are normally performed in main memory. As a result, updated data in accelerators' memory should be explicitly transferred back to main memory before making a new checkpoint.


🟩 works done before last checkpoint
🟦 works done after last checkpoint
□ remaining matrix to be factorized
region to save

(a) save the whole matrix (b) save the right-side (c) save the corner

Fig. 14. Incremental checkpoints in SKT-HPL. Data in blue and white need to be saved. The region in the red dashed block is saved for different checkpoint strategies: (a) Saves the whole matrix, (b) Saves the right side continuous region, and (c) Only saves the right bottom corner.

6.2 Failure Detection and Restart

During an SKT-HPL test, a daemon runs on a master node that is assumed not to fail. Since the master node is a single node, its MTBF is very long and this assumption is reasonable. Deploying the daemon on a reliable distributed system is an alternative choice. If one MPI process aborts, the daemon can detect it by checking the return value of `mpirun` command or the output of a job management system in a typical supercomputer.

To recover the program, SKT-HPL should restart and put each process in the right position. Most MPI implementations support specifying the layout of processes. Normally a ranklist file is used to assign each process to a certain node. After all the MPI ranks exit due to a node failure, the daemon checks the connection for each node in the ranklist. Lost nodes are replaced by other healthy nodes, which can be spare nodes or repaired nodes.

Next, the daemon restarts SKT-HPL according to a new ranklist file. All the processes that ran on healthy nodes continue to run on the same nodes and just attach to the checkpoints saved in their local memory. The processes that ran on the failed node will be restarted on a fresh node where there is no checkpoint. SKT-HPL can skip the generation of matrix A and b, because they are already in the checkpoints. However, some data structures need to be rebuilt in the initialization step.

6.3 Optimization

Besides the general approach described in Section 4, we make special optimization for HPL.

For SKT-HPL, a checkpoint only needs to save the updated part of matrix A and vector b. Since the size of vector b is so small, we simply save the whole b in each checkpoint. We discuss the checkpoint size of matrix A in this section.

Matrix A is eliminated from the top left corner to the bottom right corner. Fig. 14 presents three strategies to save data. A straightforward strategy is to save the whole matrix in every checkpoint, as shown in strategy (a) in Fig. 14, which produces checkpoints with a constant size of N^2. Strategy (b) saves the right side continuous part. If the tailing matrix is $k \times k$, it produces a checkpoint of size $N \times k$. Strategy (c) in Fig. 14, which saves the corner submatrix, produces the smallest checkpoints of size k^2. We mark S_x as the total size of all checkpoints for strategy (x) during a fault-free execution. If a checkpoint is made after eliminating each row, the total size of all checkpoints can be calculated for each strategy.

The checkpoints produced by this strategy are between the strategies (a) and (c). If a checkpoint is made after eliminating each row, during a fault-free execution, the total size of all checkpoints in the optimal case (c) is

$$S_{corner} = \sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}. \tag{17}$$

And the total data size for method (b) is

$$S_{side} = \sum_{i=1}^{N} N \times i = \frac{N^2(N+1)}{2}. \tag{18}$$

The size using strategy (a) is also given here

$$S_{whole} = \sum_{i=1}^{N} (N \times N) = N^3. \tag{19}$$

We can calculate their ratios as below:

$$S_a : S_b : S_c \approx 3 : 1.5 : 1. \tag{20}$$

This reduces communication traffic to half or one third by applying an incremental checkpoint strategy.

SKT-HPL does not need to make a checkpoint after each iteration, but this conclusion is still correct. For example, a checkpoint is made after eliminating R rows of an $N \times N$ matrix. It is equivalent to making a checkpoint after each iteration for a matrix with order N/R.

There are additional requirements for the size of matrix A. As mentioned above, SKT-HPL uses an incremental checkpoint strategy. Therefore, all processes must have the same partition size as matrix A. For example, to solve a matrix of $N \times N$ with $P \times Q$ MPI ranks, we eliminate NB rows in each iteration during the elimination step. The matrix order N must be an integral multiple of both $P \times NB$ and $Q \times NB$ at the same time, as described by the following formula:

$$(NB \times P) \mid N \textbf{ and } (NB \times Q) \mid N. \tag{21}$$

6.4 Calculating Checksums

We use `MPI_Reduce` to calculate checksums of checkpoints, taking full advantage of underlying well-tuned MPI library.

```
MPI_Reduce (datatype, operator, ...)
```

Supported by most MPI implementations, both *exclusive or* (XOR) and *numeric addition* (SUM) encoding methods are supported in our approach:

```
MPI_Reduce (MPI_LONG_LONG, MPI_BXOR, ...)
MPI_Reduce (MPI_DOUBLE, MPI_SUM, ...)
```

On some platforms, the logical XOR operation is much faster than the numerical SUM. Our implementation uses XOR by default, but SUM is also supported.

7 Evaluation

7.1 Methodology

We perform our evaluation on two large-scale HPC systems, Tianhe-1A and Tianhe-2 [1]. The configuration of a single node for both platforms is listed in Table 2. Table 2 shows that Tianhe-2 has more powerful CPU and larger

TABLE 2
Node Configuration of Tianhe-1A and Tianhe-2

	Tianhe-1A node	Tianhe-2 node
CPU	Dual Xeon X5670 ×6 cores @2.93 GHz	2Dual Xeon E5-2692(v2) 2×12 cores @2.20GHz
Peak Performance	140 GFLOPS	422 GFLOPS
Memory	48 GB	64 GB
P2P Bandwidth	6.9 GB/s	7.1 GB/s

memory size than its predecessor, Tianhe-1A. However, each processor core of Tianhe-1A has more memory than that of Tianhe-2 (4 GB/core versus 2.4 GB/core). The bandwidth of point-to-point communication is similar to both systems. Besides these two large systems, we also use a local cluster connected by EDR Infiniband network for experiments that need to power-off computing nodes. Each node is equipped with 2-way Xeon E5-2670 v3 processors (24 processor cores in total) and 64GB of memory size.

Compared to a double-checkpoint method, self-checkpoint provides more available memory for applications. It is a general method and not tied to any specified application. However, some usual benchmarks such as NPB are fixed-size, or have multiple sizes that differ too much. As a result, 50 percent more available memory does not enable a larger-size problem, so they are not proper for demonstrating our idea. Instead, we use HPL, which has an adjustable problem size, to verify our idea.

7.2 Comparison with State-of-the-Art Methods

In this section, we compare SKT-HPL with state-of-the-art methods for fault-tolerant HPL, including an algorithm-based fault tolerant HPL (ABFT-HPL) [37], a traditional checkpoint-and-restart method (BLCR) [20], and a multi-level checkpoint system (SCR) [25]. As some experiments have special requirements, e.g., mounting ramfs, installing SSD [43] (Solid-State Drive), and powering-off computing nodes, which are not allowed on supercomputing centers, we perform these experiments on our local cluster. We compare different methods for fault-tolerant HPL and report their performance. Since BLCR is a system level tool and automatically saves all data, the optimization in Section 6.3 cannot be applied. For a fair comparison, we disable the optimization in SKT-HPL. Every time of making a checkpoint, all methods will save all data. Moreover, to validate the reliability of different methods under a permanent node failure, we power off a computing node during HPL tests. All tests of fault-tolerant HPL are run with 128 MPI processes. Each process has 4GB of memory space. Limited by the scale of this local cluster, which has 8 nodes, the group size is set to 8. Table 3 shows experimental results.

In our experiment, ABFT-HPL [37] fails to tolerate a power-off event, because MPI runtime cannot recover the user program's data structures after a node loss. The overhead of such fault-tolerant algorithms is inversely proportional to the number of processes. So its performance is not good in this small-scale experiment and it only achieves 78.61 percent of the original performance.

BLCR [20] is a classic checkpoint-and-restart framework. We perform our experiments on both hard disk drives (HDDs) and SSD devices. When the checkpoints are written into hard disk drives, its performance only achieves 72.53 percent of the original HPL, shown with BLCR+HDD in Table 3. Therefore, it is not practical to use the traditional CR method as a performance benchmark.

When replacing the HDDs with SSD devices, the BLCR method gets much better performance for the HPL test. The performance of BLCR+SSD achieves 87 percent of the original HPL. In our experiments, both BLCR+HDD and BLCR +SSD write checkpoints into local devices. It would be much slower if a distributed file system is used.

SCR [25] is a state-of-the-art multi-level checkpoint system , which can write checkpoints to RAM, Flash, or disk of computing nodes in addition to a parallel file system. In our experiments, we only present its best performance of writing checkpoints into RAM. Because SCR needs to save double in-memory checkpoints to tolerate a node failure during checkpoint updating, there is only 1.22GB available memory (30.5 percent of the total size) for each process to do the HPL test. Therefore, the problem size SCR solves is the smallest among these methods. Since our local cluster has very large memory size per core, it also achieves 92.1 percent of the original HPL performance.

Thanks to the self-checkpoint mechanism, SKT-HPL has 1.75 GB memory for the HPL test, which is 43 percent higher than the SCR method. Among these methods, SKT-HPL achieves the best performance for the HPL test and it is 94.49 percent of the original performance. There are two main reasons for the best performance of SKT-HPL. One is that it has much shorter checkpoint time than traditional checkpoint methods. The other is that it has much more available memory space than previous in-memory checkpoint methods.

7.3 Validation on Large-Scale Systems

Validation experiments on Tianhe-1A and Tianhe-2 are performed by manually removing several computing nodes during SKT-HPL tests. We kill the SKT-HPL processes of

TABLE 3
Comparison Between Different Methods of Fault-Tolerant HPL

	Problem Size	Runtime (sec.) (no checkpoint)	Checkpoint Time (sec.)	GFLOPS and Checkpoints (checkpoint per 10min)	Available Memory(GB)	Normalized Efficiency	Recover after node powered-off?
Original HPL	234,240	2,338.64	-	3,669.81 (0 chkpt)	4.00	100.00%	NO
ABFT	212,224	2,208.55	-	2,885.00 (0 chkpt)	3.28	78.61%	NO
BLCR+HDD	234,240	2,338.64	295.20	2,661.83 (3 chkpt)	4.00	72.53%	YES
BLCR+SSD	234,240	2,338.64	111.92	3,209.08 (3 chkpt)	4.00	87.45%	YES
SCR+Memory	129,280	426.18	4.33	3,380.02 (0 chkpt)	1.22	92.10%	YES
SKT-HPL	154,880	709.84	6.21	3,467.64 (1 chkpt)	1.75	94.49%	YES

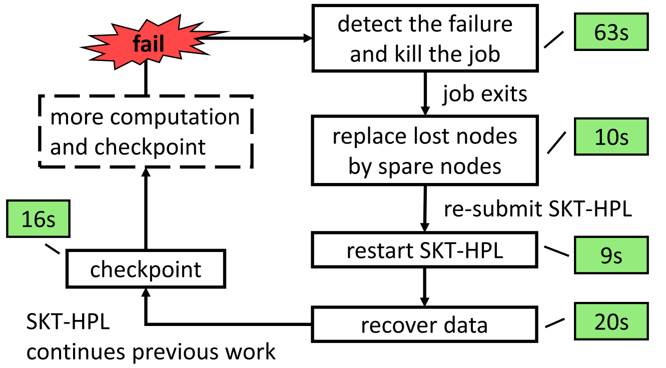

fail
detect the failure and kill the job
63s
job exits
more computation and checkpoint
replace lost nodes by spare nodes
10s
re-submit SKT-HPL
16s
checkpoint
restart SKT-HPL
9s
recover data
20s
SKT-HPL continues previous work

Fig. 15. Time for each phase during a work-fail-detect-restart cycle. All results are measured on Tianhe-2 with 24,576 MPI processes. The time value for each phase is presented in the small green rectangles.

those nodes and remove those nodes from the resource pool of job management system. Those nodes are permanently lost since SKT-HPL can no longer launch processes on them.

In our experiments, SKT-HPL is able to replace the lost nodes using spare nodes, recover the lost data, continue running, and finally pass verification. We therefore argue that SKT-HPL can tolerate real permanent node losses on two systems.

We further measure the time for each phase during a work-fail-detect-restart cycle, as shown in Fig. 15. The time for detecting a failure depends on underlying job management systems. The failure detection time varies largely on Tianhe-1A, and it is about 30 seconds on average, while the detection time on Tianhe-2 is about 63 seconds. The time for replacing lost nodes and restarting SKT-HPL is about 10 and 9 seconds respectively. The recovery process is similar to that used to calculate the checksum. But due to some additional computation, the recovery time (20 seconds) is a little longer than that to make a checkpoint (16 seconds).

7.4 Performance of SKT-HPL

We perform the original HPL test on both systems with typical configurations. We obtain 15.55 TFLOPS (86.38 percent of the theoretical peak performance) with 1,536 processes on Tianhe-1A. On Tianhe-2, we do not run HPL from the beginning to the end, since it consumes too much time and energy. Instead, we run HPL for minutes and record its actual FLOPS value. The performance on Tianhe-2 is 367.04 TFLOPS (84.94 percent of the theoretical peak performance), with 24,576 processes.

To analyze the performance of SKT-HPL, we test SKT-HPL on both systems with near half of the memory, no checkpoint is written. The group size for Tianhe-1A is 16 and 8 for Tianhe-2. Therefore, with the self-checkpoint mechanism, the available memory is 47 and 44 percent of the total memory on both systems respectively. SKT-HPL obtains 15.21 TFLOPS (97.81 percent of the original HPL) on Tianhe-1A and 351.60 TFLOPS (95.79 percent of the original HPL) on Tianhe-2. Fig. 16 shows the performance of the original HPL and SKT-HPL.

7.5 Benefits of Self-Checkpoint

To verify the efficiency models presented in Section 3 and demonstrate the benefits of the self-checkpoint mechanism, we run SKT-HPL with different memory size on two large systems. Fig. 17 shows test results and fitting results by our model. The squares and triangle dots represent the measured results of Tianhe-1A and Tianhe-2 respectively. Results show that our efficiency models can fit the test results very well and also verify the nonlinear relationship between problem size and HPL efficiency. Using the self-checkpoint can get 5 percent higher performance than using the double-checkpoint on Tianhe-2 because of its much more available memory (44 versus 30 percent).

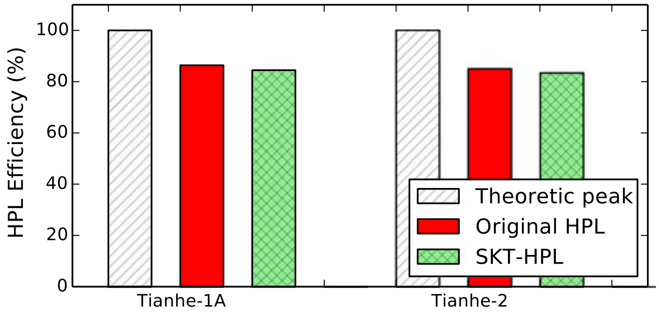


Fig. 16. The efficiency of the original HPL and SKT-HPL (without making checkpoint). The original HPL uses full memory and SKT-HPL uses near half of the memory.

7.6 Overhead of Building Checkpoints

The time cost of building checkpoints in the self-checkpoint mechanism includes two parts, calculating checksums or encoding (network communication) and overwriting old checkpoints (local memory copying). Overwriting time is less than one second in experiments, and is insignificant compared with the communication time for encoding.

As mentioned in Section 6, we use `MPI_Reduce` to encode checkpoints. There are two alternatives. One is XOR and the other is numeric sum. Fig. 18 shows the numeric sum is better than XOR when a subgroup has more than 4 nodes. Numeric sum has 15 percent speedup over XOR when a subgroup has 16 nodes. Notably, this phenomenon depends on particular systems, and this is an option in our implementation of SKT-HPL. By default, we use XOR for bit-wise correctness.

Fig. 19 shows the encoding time and checkpoint size for different group sizes. As a checkpoint is close to half of the memory as shown in Equation (6), the checkpoint size is similar for different group sizes. The encoding time on both systems increases slowly with the group size.

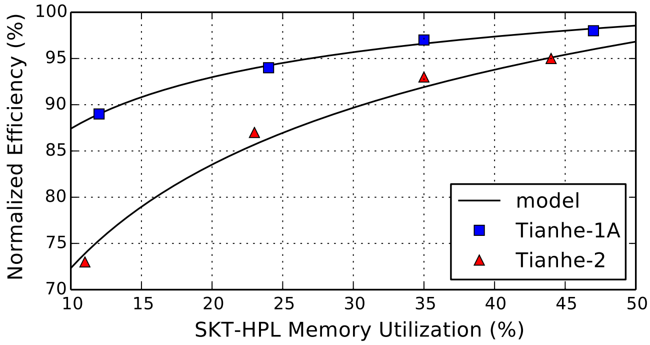


Fig. 17. The relationship between Memory space used for computation and the normalized efficiency. Memory space and efficiency are compared with a typical run of the original HPL with full memory. The impact of memory space is more significant on Tianhe-2 than on Tianhe-1A.

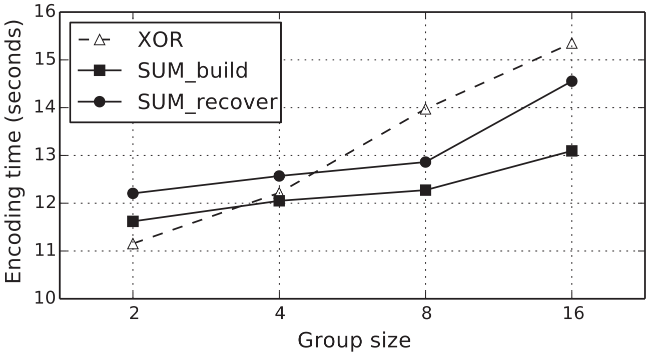


Fig. 18. Encoding time with two schemes and different subgroup sizes on Tianhe-1A. The numeric sum method is asymmetric, since recovery time is longer than building time. Both are shorter than the XOR method.

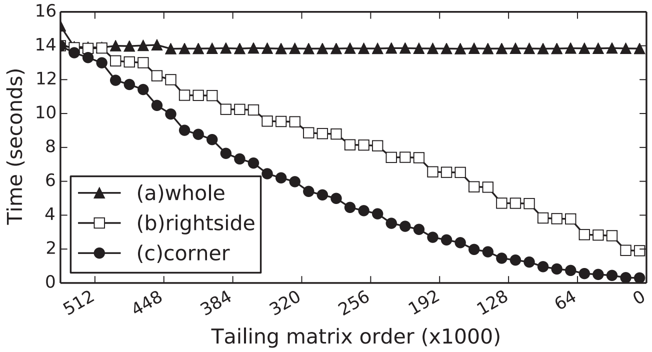


Fig. 20. Checkpoint time of three strategies on Tianhe-1A. Using 1,536 processes and problem size $N = 552,960$. The time includes both encoding time and memory copying time.

From Table 2 we know that the point-to-point communication performance of Tianhe-2 is better than Tianhe-1A. But the encoding time of Tianhe-2 is much longer even with smaller checkpoints than Tianhe-1. It is because a network port of Tianhe-2 is shared by 24 processes, while in Tianhe-1A one port is only shared by 12 processes. As a result, the bandwidth per process of Tianhe-1A is much higher.

7.7 Decreasing Checkpoint Time

Fig. 20 shows the checkpoint time for three strategies discussed in Section 6.3. Obviously, the best choice is the strategy (c) in Fig. 14. Therefore, saving the corner submatrix has the least overhead.

The experimental result fits the analysis in Section 6.3, but there are still some differences. The time curves are not perfectly smooth but with some flat segments on them. It is because SKT-HPL requires that each MPI rank gets the same size part of checkpoint, as described in Section 6.3. During the execution, SKT-HPL copies a little more data than theoretical requirement.

8 Related Work

Checkpoint-and-restart [13] is a classic fault tolerance method, which saves the intermediate states of an application (i.e., a checkpoint) into a reliable storage, and recovers data from a checkpoint after a failure [38]. As traditional CR methods save checkpoints to a parallel file system [32] and introduce large storage overhead, they are only suitable for medium-scale systems.

Diskless or in-memory checkpoint, which saves checkpoints in memory and uses error-correcting codes to protect data, was proposed since it has much lower overhead than disk-based CR and is a potential solution for large-scale systems [28]. Ibtesham et al. compress the data before making a checkpoint [22], so that the size of checkpoint is smaller. Plank et al. proposed an incremental diskless checkpoint system [27] to reduce memory consumption. In this method, only data updated will be saved in a new checkpoint. The impact of compression and incremental checkpoint depends on the characteristics of applications. In worst cases, i.e., data cannot be compressed and all data has been updated, they cannot save any memory space. Our method is not an alternative of compression checkpoint or incremental checkpoint. In fact, they are orthogonal. Compression or incremental checkpoint tell us *what to save* and our method answers *where and how to save*. To improve scalability, Zheng et al. proposed an in-memory checkpoint scheme using a buddy system, which is scalable by dividing nodes into many two-node groups [41], [42]. This scheme can only use one third of the memory, making it more suitable for applications with little memory consumption.

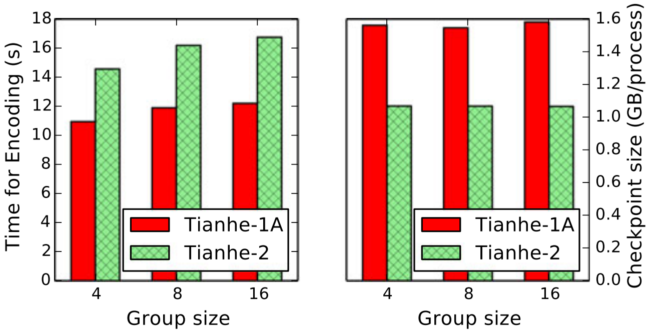


Fig. 19. The encoding time of calculating checksums (left) and checkpoint size (right) with different group sizes. Encoding time grows slowly with group size. Checkpoint size is not very sensitive to group size.

Besides in-memory checkpoint, multi-level CR models have been proposed, such as SCR [25], 3D-PCRAM [11], and FTI [3]. Multi-level CR saves checkpoints to fast devices like memory, PCRAM, and local SSD in a short interval, and to slower devices (e.g., global file system) in a long interval. These studies focus on a general CR framework for parallel applications, while our method focuses on improving the available memory space of in-memory checkpoint. Therefore, we can integrate our method into a multi-level CR framework for better performance.

Some numerical algorithms can obtain redundancy by pre-processing the original input data. Huang and Abraham studied algorithm-based fault tolerance for classic matrix operations such as matrix-matrix multiplication and LU decomposition [21]. Yao et al. proposed a fault tolerant HPL [37]. Besides HPL, some other algorithms such as iterative methods and QR decomposition have also been studied with ABFT [6], [36]. Fault tolerant applications based on ABFT usually have low overhead. Chen et al. proposed Online-ABFT [7], which can detect soft errors in the widely used Krylov subspace iterative methods. Li et al. [24] coordinated ABFT and error-correcting code (ECC) for main memory, to improve performance and energy efficiency of ABFT-enabled applications. ABFT methods highly rely on underlying MPI runtime. If the MPI runtime cannot recover from a failure, ABFT has no chance to recover data.

To the best of our knowledge, no MPI runtime can tolerate node failure with negligible overhead. MPICH-V2 [5] and RADICMPI [12] are implemented based on message logging and checkpoints. Thus the performance overhead is not trivial. Some MPI runtime environments such as Intel MPI can keep running after a process is aborted, instead of forcing all processes to exit. But the aborted process is permanently lost and cannot be recovered. FT-MPI [15] extends the semantic of MPI by trying to repair MPI data structures and restart lost processes after a failure. Based on our experiments, neither Intel MPI nor FT-MPI can restart the lost processes after a node being powered off. Bland et al.'s work [4] shows that a standard MPI runtime can tolerate permanent node losses with the help of network configuration. This method introduces overhead to MPI library, and it is not practical for non-privileged users to change the configuration of supercomputers.

The idea of redundant execution uses multiple processes for a logical MPI rank. Both computation and communication are duplicated. Ferreira et al. proposed a prototype system rMPI [16]. Fiala et al. proposed redMPI [17], which can not only tolerate fail-stop errors, but also detect and correct silent data corruption. Similar to ABFT methods, redundant execution is good at detecting soft-errors but cannot tolerate node failures. Unlike ABFT, a redundant execution model has no requirement on algorithms. Nevertheless, its overhead is much heavier than ABFT. Redundant execution only uses half of the CPU and memory and has an efficiency less than 50 percent.

9 Conclusion

To reduce the memory usage of in-memory checkpoint, we propose a new strategy, called self-checkpoint. It not only achieves the same reliability of in-memory checkpoint using two copies of checkpoints, but also increases available memory space for applications by almost 50 percent. Based on the self-checkpoint mechanism, we further implement SKT-HPL, a fault-tolerant HPL, which can not only tolerate a real node failure on a large-scale supercomputer, but also achieve very close performance with the original HPL. Experimental results show that with 24,576 processes on Tianhe-2, the self-checkpoint method improves the available memory size by 47 percent than the state-of-the-art in-memory checkpoint. Moreover, SKT-HPL achieves over 95 percent of the original HPL's performance, and is 5 percent higher than using previous in-memory checkpoint methods.

Acknowledgments

The authors sincerely thank the anonymous reviewers for their valuable comments and suggestions. The authors thank Heng Lin, and Zhen Zheng for their efforts to improve this paper. This work is supported by National Key Research and Development Program of China 2017YFB1003103, NSFC projects 61232008 and 61722208, Tsinghua University Initiative Scientific Research Program, and Microsoft Research Asia Collaborative Research Program FY16-RES-THEME-095.

References

[1] top500 website. (2017). [Online]. Available: http://top500.org/

[2] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proc. Annu. Int. Conf. Supercomput.*, 2004, pp. 277–286.

[3] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 32.

[4] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI," in *Proc. Eur. Conf. Parallel Process.*, Aug. 2012, pp. 477–488.

[5] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *Proc. ACM/IEEE Conf. Supercomput.*, 2003, pp. 25–25.

[6] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 73–84.

[7] Z. Chen, "Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 167–176.

[8] Z. Chen, et al., "Fault tolerant high performance computing by a coding approach," *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2005, pp. 213–223.

[9] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comput. Syst.*, vol. 22, pp. 303–312, 2006.

[10] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: A fault tolerant implementation without checkpointing," in *Proc. Annu. Int. Conf. Supercomput.*, 2011, pp. 162–171.

[11] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, Art. no. 57.

[12] A. Duarte, D. Rexachs, and E. Luque, "An intelligent management of fault tolerance in cluster using RADICMPI," in *Proc. Eur. Parallel Virtual Mach./Message Passing Interface Users Group Meet.*, Sep. 2006, pp. 150–157.

[13] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *J. Supercomput.*, vol. 65, no. 3, pp. 1302–1326, Sept. 2013. ISSN 0920–8542, 1573-0484.

[14] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how HPC systems fail," in *Proc. Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2013, pp. 1–12.

[15] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Proc. Eur. Parallel Virtual Mach./Message Passing Interface Users Group Meet.*, 2000, pp. 346–353.

[16] K. Ferreira, et al., "Evaluating the viability of process replication reliability for exascale systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 44.

[17] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–12.

[18] E. Gelenbe, "On the optimum checkpoint interval," *J. ACM*, vol. 26, no. 2, pp. 259–270, Apr. 1979. ISSN 0004–5411.

[19] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *Proc. IEEE/ACM Int. Conf. Cluster Cloud Grid Comput.*, 2010, pp. 63–72.

[20] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for Linux clusters," *J. Physics: Conf. Series*, vol. 46, 2006, Art. no. 494.

[21] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, no. 6, pp. 518–528, Jun. 1984.

[22] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *Proc. Int. Conf. Parallel Process.*, 2012, pp. 148–157.

[23] C. Jin, H. Jiang, D. Feng, and L. Tian, "P-code: A new RAID-6 code with optimal properties," in *Proc. Annu. Int. Conf. Supercomput.*, 2009, pp. 360–369.

[24] D. Li, Z. Chen, P. Wu, and J. S. Vetter, "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–12.

[25] A. Moody, G. Bronevetsky, K. Mohror, and B. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–11.

[26] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1988, pp. 109–116.

[27] J. S. Plank and K. Li, "Faster checkpointing with N+1 parity," in *Proc. IEEE 24th Int. Symp. Fault- Tolerant Comput.*, Jun. 1994, pp. 288–297.

[28] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998.

[29] Y. Robert, "Fault-tolerance techniques for computing at scale," *Proc. IEEE/ACM Int. Conf. Cluster Cloud Grid Comput.*, 2014.

[30] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans. Depend. Secure Comput.*, vol. 7, no. 4, pp. 337–350, Oct. 2010. ISSN 1545–5971.

[31] X. Tang, J. Zhai, B. Yu, W. Chen, and W. Zheng, "Self-checkpoint: An in-memory checkpoint method using less space and its practice on fault-tolerant HPL," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2017, pp. 401–413.

[32] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "A job pause service under LAM/MPI+BLCR for transparent fault tolerance," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–10.

[33] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Hybrid checkpointing for MPI jobs in HPC environments," in *Proc. 2010 IEEE 16th Int. Conf. Parallel Distrib. Syst.*, 2010, pp. 524–533.

[34] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas, "Building algorithmically nonstop fault tolerant MPI programs," in *Proc. Int. Conf. High Perform. Comput.*, 2011, pp. 1–9.

[35] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*. Hoboken, NJ, USA: Wiley, 1999.

[36] P. Wu and Z. Chen, "FT-ScaLAPACK: Correcting soft errors on-line for scalapack cholesky, QR, and LU factorization routines," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 49–60.

[37] E. Yao, R. Wang, M. Chen, G. Tan, and N. Sun, "A case study of designing efficient algorithm-based fault tolerant application for exascale parallelism," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 438–448.

[38] X. Ye, et al., "An anomalous behavior detection model in cloud computing," *Tsinghua Sci. Technol.*, vol. 21, no. 3, pp. 322–332, 2016.

[39] J. Zhang, J. Chen, J. Luo, and A. Song, "Efficient location-aware data placement for data-intensive applications in geo-distributed scientific data centers," *Tsinghua Sci. Technol.*, vol. 21, no. 5, pp. 471–481, 2016.

[40] Y. Zhao, H. Jiang, K. Zhou, Z. Huang, and P. Huang, "DREAM-(L) G: A distributed grouping-based algorithm for resource assignment for bandwidth-intensive applications in the cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3469–3484, Dec. 2016.

[41] G. Zheng, L. Shi, and L. V. Kale, "FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2004, pp. 93–103.

[42] G. Zheng, X. Ni, and L. V. Kal, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *Proc. IEEE/IFIP Int. Conf. Depend. Syst. Netw. Workshops*, 2012, pp. 1–6.

[43] K. Zhou, S. Hu, P. Huang, and Y. Zhao, "LX-SSD: Enhancing the lifespan of NAND flash-based memory via recycling invalid page," in *Proc. Int. Conf. Massive Storage Syst. Technol.*, 2017, pp. 1–13.

Xiongchao Tang received the BS degree from Tsinghua University, in 2013. He is currently working toward the PhD degree in the Institute of High-Performance Computing, Tsinghua University. His major research interests include performance analysis and optimization for parallel programs, performance benchmarking and performance variance detection for large-scale high-performance computing systems, heterogeneous programming, and fault tolerance.

Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China, in 2003 and the PhD degree in computer science from Tsinghua University, in 2010. He is currently an assistant professor in the Department of Computer Science and Technology, Tsinghua University. His research interests include high-performance computing, compilers and performance analysis, and optimization of parallel applications.

Bowen Yu received the BS degree from Northwestern Polytechnical University, in 2015. He is working toward the PhD degree in the Institute of High-Performance Computing, Tsinghua University. His major interests include MPI, and large-scale systems in machine learning.

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University, in 1995 and 2000, respectively. He was the CTO in Opportunity International Inc. from 2000 to 2002. Since January 2003, he has joined Tsinghua University. He is currently a professor and an associate head in the Department of Computer Science and Technology, Tsinghua University. His research interests include parallel and distributed computing and programming model.

Weimin Zheng received the BS and MS degrees from Tsinghua University, in 1970 and 1982, respectively. He is currently a professor in the Department of Computer Science and Technology, Tsinghua University. He is currently the director in China Computer Federation (CCF). His research interests include parallel and distributed computing, compiler technique, grid computing, and network storage.

Keqin Li is a SUNY distinguished professor of computer science in the State University of New York. He is also a distinguished professor of Chinese National Recruitment Program of Global Experts (1000 Plan), Hunan University, China. He was an Intellectual Ventures endowed visiting chair professor in the National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China, during 2011-2014. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things, and cyber-physical systems. He has published more than 530 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*, and the *IEEE Transactions on Sustainable Computing*. He is a fellow of the IEEE.