# Self-Checkpoint: An In-Memory Checkpoint Method Using Less Space and Its Practice on Fault-Tolerant HPL


Xiongchao Tang Jidong Zhai Bowen Yu Wenguang Chen Weimin Zheng

Department of Computer Science and Technology
Tsinghua University, Beijing, China
{txc13,yubw15}@mails.tsinghua.edu.cn {zhaijidong,cwg,zwm-dcs}@tsinghua.edu.cn

## 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%. To validate our method, we apply the self-checkpoint to an important problem, fault tolerant HPL. 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% 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% and the performance by 5%.

***Keywords*** Fault Tolerance; In-Memory Checkpoint; Fault-Tolerant HPL; Memory Consumption



*PPoPP '17* Feb. 4–8, 2017, Austin, Texas, USA.

ACM 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3018743.3018745

## 1. Introduction

The substantial growth of system scale in High Performance Computing (HPC) makes fault tolerance increasingly important. A long-running HPC application can run 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 everyday [23, 28]. 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 [9, 10, 32, 36] have been proposed. The main idea is to modify an application's algorithm for fault tolerance [20]. 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 exists 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%).

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, 29]. Some fault-tolerant MPI implementations [5, 6, 12] may 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 much higher performance than disks, in-memory or diskless CR methods have gained extensive attention recently, which can effectively reduce the overhead of saving checkpoints [18, 27, 38].

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 [38]. A second checkpoint is used to tolerate failures when updating an old checkpoint. This results in only one third of 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, 23]. However, copying checkpoints from memory into disks also introduces additional overhead thus loses the 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%. 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) [32, 36], or introduce too much overhead (traditional CR methods saving checkpoints in disks) [13, 30], 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, 26, 31] 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%. SKT-HPL achieves over 95% of the original HPL's performance, and 5% 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%.
- 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% of the original HPL's performance and improves the memory usage over the state-of-the-art in-memory checkpoint by 47%.

The remainder of this paper is organized as follows. Section 2 gives basic in-memory checkpoint framework. Section 3 describes the approach of self-checkpoint mechanism. Section 4 discusses the importance of more available memory. Section 5 describes the implementation of SKT-HPL. Section 6 presents our experimental results. Section 7 describes related work. Section 8 gives the conclusion.

## 2. Basic In-Memory Checkpoint Framework

In this section, we describe some general fundamental techniques for in-memory checkpoint methods. Our work is also constructed on this basic framework.

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

### 2.1 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, 23] 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 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.

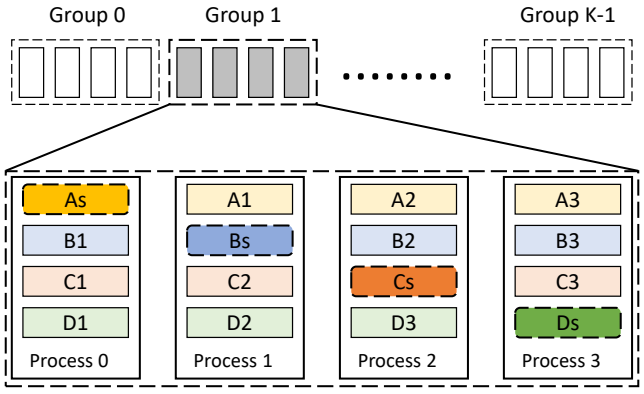


**Figure 1.** Data encoding. Processes in a large-scale application are partitioned into a number of small groups. We use a stripe-based encoding in each group. $A_S, B_S, C_S$, and $D_S$ in dashed dark blocks are checksums for $A_i$, $B_i$, $C_i$, and $D_i$ respectively.

Figure 1 shows an example to illustrate above encoding method used in our system, which is similar with the encoding mechanism used in RAID-5 [24]. 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.

$$X_S = X_1 + X_2 + ... + X_{n-1} \quad (1)$$

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.

Using above encoding, each group can tolerate a single node failure. We can apply more complex encoding methods, such as RAID-6 [21] and Reed-Solomon [33], to tolerate more node failures. For a higher degree of fault tolerance, in-memory checkpoint methods can be also combined with a multi-level checkpoint framework [3, 11, 23].

### 2.2 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.

### 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. Self-Checkpoint Mechanism

### 3.1 Methodology

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

Figure 2 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

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

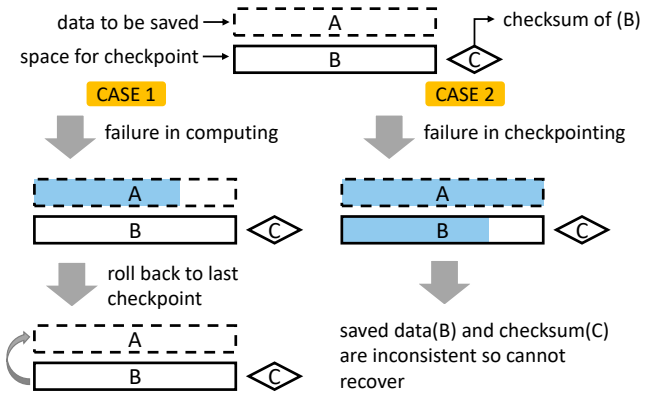


**Figure 2.** The strategy of single in-memory checkpoint. This method cannot recover data from a failure during checkpoint updating.

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 Figure 2). However, the single checkpoint cannot tolerate a node failure while updating a checkpoint or a checksum (CASE 2 in Figure 2), because at this time, the checkpoint $B$ and checksum $C$ are in an inconsistent state.

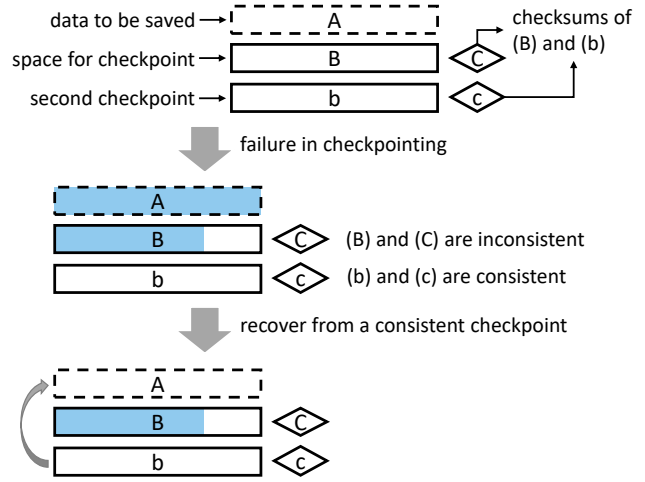


**Figure 3.** The strategy of double checkpoints. While updating checkpoints, at least one checkpoint and its checksum which are consistent can be used for recovery. Its main disadvantage is that too much memory space is wasted.

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 Figure 3. 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 [37, 38] 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 resources. Another solution is to flush the old checkpoint into a permanent device like hard-disks, but it will lose the performance advantage of the in-memory checkpoint.

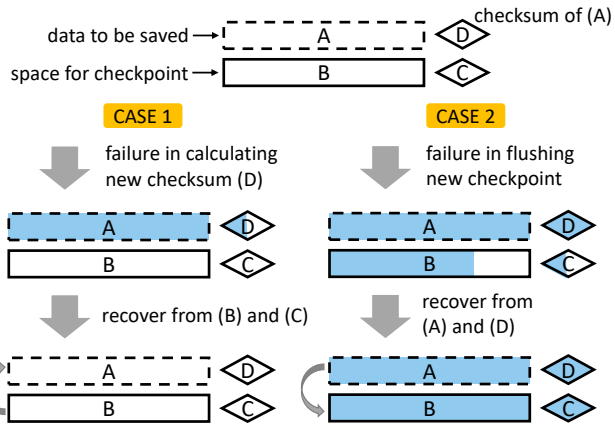


**Figure 4.** The strategy of self-checkpoint. In CASE 1, if there is a failure during calculating a new checksum, the program can recover from $B$ and $C$. In CASE 2, if there is a failure while updating checkpoints, the program can recover from $A$ and $D$

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 Figure 3, 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. Figure 4 illustrates an ideal case using the self-checkpoint strategy to handle different situations of failures.

When making a new checkpoint, a checksum $D$ is firstly 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 Figure 4, when a node failure is detected when calculating the checksum $D$, we can recover the program with the 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 Figure 4, we can recover the program using the user's data $A$ and its checksum $D$. In other words, the user's memory space for computation itself is served as a checkpoint. Therefore, we call our proposed method a self-checkpoint mechanism.

As discussed in Subsection 2.3, we use SHM to keep data in memory. For dynamic variables allocated in heap, we can change their allocation. For example, a malloc statement `p=malloc(...)` is transferred into an SHM statement `p=shmget(...)`. For statically allocated variables, such as

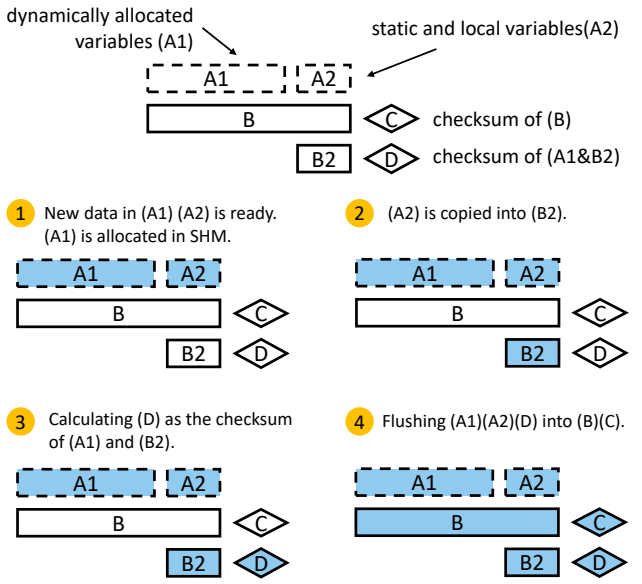


**Figure 5.** The complete workflow of self-checkpoint strategy. Most data is allocated in the shared memory, represented by $A1$ and little data is stored in the user's space, represented by $A2$. $D$ is the checksum of $A1$ and $B2$.

global and local variables, we can also manually transfer related code to use shared memory. In our experience, large data structures are usually dynamically allocated in heap. For example, almost all data structures of HPL are dynamically allocated. As a result, if the size of statically allocated variables (A2 in Figure 5) is small, such as loop iterators or other scalar variables, we can allocate a small second-buffer (B2 in Figure 5) for simplicity.

Figure 5 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. Most of the user's data $A1$ is stored in the 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 Subsection 3.2. Section 6 gives the detailed performance data of the self-checkpoint mechanism.

### 3.2 Analysis of Memory Usage

Selecting a suitable strategy for group partitioning is important for the performance of the self-checkpoint mechanism. In this subsection, 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 Figure 5) and only partial local variables in the user's space ($A2$ in Figure 5), so the size of $A2$ and $B2$ is negligible.

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

**Table 1.** The memory usage of the self-checkpoint mechanism for each part in Figure 5. 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 Figure 5) and $B$ is also $M$. Due to the negligible size, B2 is omitted for simplicity. According to the encoding method described in Subsection 2.1, 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

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

Similarly, we can calculate the available memory space of the double checkpoint method shown in Figure 3.

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

And the available memory with the single checkpoint shown in Figure 2 is

$$U_{single} = \frac{M}{M+MN/(N-1)} = \frac{N-1}{2N-1} \tag{4}$$

We illustrate the available memory space of different in-memory checkpoint methods with several typical group sizes in Figure 6. 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 group size, our method has more available memory space than the double checkpoint, up to nearly 50%.

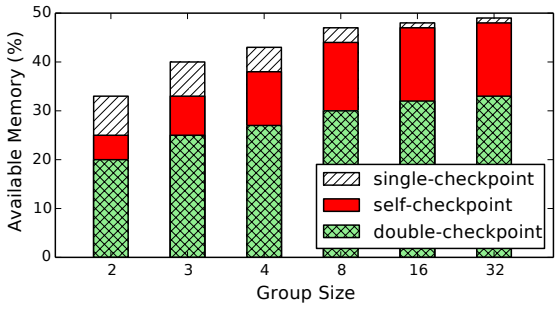


**Figure 6.** The memory usage of different in-memory checkpoint methods with group sizes of 2, 3, 4, 8, and 16.

### 3.3 Grouping Strategy

In this subsection, we give our strategies in determining suitable group partitioning in a large-scale system. From Figure 6, 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. 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. In our experiments, we select the group size of 16. The available memory of a group with 16 processes is 47% and is close to the upper bound of 50%. We find that in a large system a larger group size provides little benefit for available memory space but causes much overhead in communication.

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, 29], 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. 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, more available memory means that programs can run for larger problem sizes with the same nodes. For some others, they can solve fixed-size problems with fewer nodes.

HPL belongs to the former case. It has an adjustable problem size, thus larger problem 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%.

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 $O(N^2)$ [25]. 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 $\gamma N^3$. 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 $\beta N^2$ is an estimation for memory access and communication overhead. Therefore, the HPL efficiency $E(N)$ can be calculated as follows:

$$E(N) = \frac{\gamma N^3}{\alpha N^3 + \beta N^2} = \frac{N}{aN + b} \tag{5}$$

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.

Figure 7 shows that this model fits well with real experimental data on a local cluster. The dots in Figure 7 represent the real data when executing 192 MPI ranks on a local cluster with different problem sizes. The line is fitted with our model.

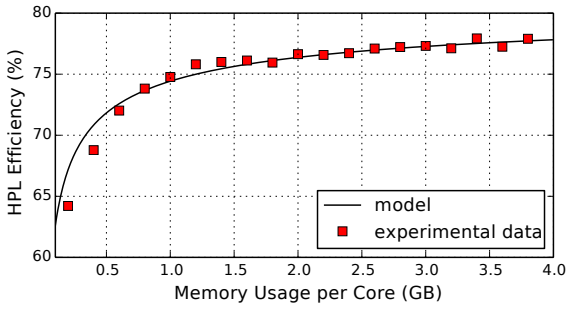


**Figure 7.** Fitting of the efficiency model with experimental data. Experiments are conducted with 192 MPI ranks on a local cluster with Xeon E5-2670(v3) and 100Gbps Infiniband network.

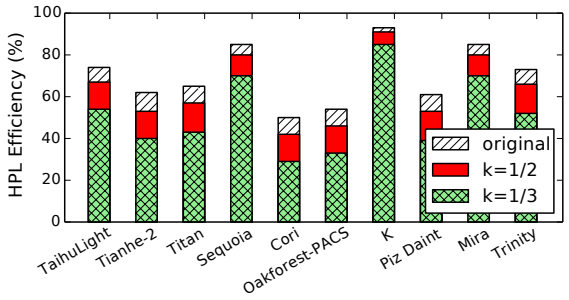


**Figure 8.** Modeled HPL efficiency of the top 10 supercomputers with different available memory space. The green mesh bars represent officially reported performance. The hatched bars and red bars are results only using one third and half of the memory size respectively, drawn by our model.

In Subsection 6.5, 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 an efficiency $e_1$ for the problem size of $N_1$ with the total available memory, then we have

$$e_1 = E(N_1) = \frac{N_1}{aN_1 + b} \tag{6}$$

Thus the value of $b$ is

$$b = \frac{(1 - ae_1)N_1}{e_1} \tag{7}$$

If only partial memory is available for HPL, and ratio is $k(0 < k < 1)$, then the problem size $N_2$ is $\sqrt{k}N_1$. Combining Equation 5, 6 and 7, we have the efficiency for the problem size of $N_2$ as

$$e_2 = \frac{\sqrt{k}e_1}{1 - (1 - \sqrt{k})ae_1} > \frac{\sqrt{k}e_1}{1 - (1 - \sqrt{k})e_1} \tag{8}$$

The last step is because $a > 1$. Equation 8 gives a lower bound of HPL efficiency for different problem sizes.

To make the relationship between available memory space and HPL efficiency more clear, Figure 8 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% 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.

## 5. SKT-HPL Implementation

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

### 5.1 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, and 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 $O(N^2)$.

Figure 9 illustrates the workflow of SKT-HPL. The white blocks stand for operations in the original HPL as described above, and the shadow 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.

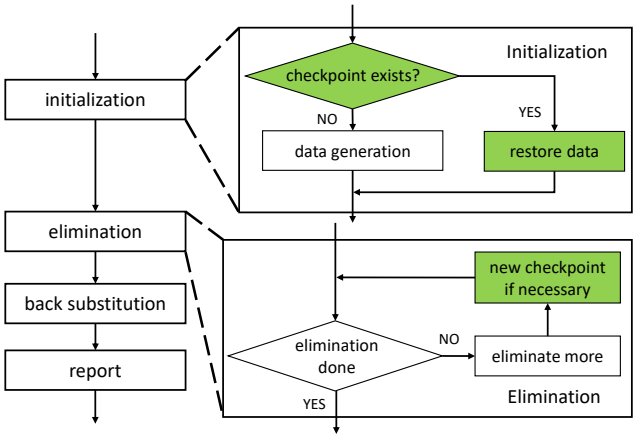


**Figure 9.** The workflow of SKT-HPL. Checkpoints are made at the end of a certain iteration during the elimination step. If a node failure is detected, SKT-HPL restores the data from the checkpoint in the initialization step.

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.

### 5.2 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. With the same configure file, matrix $A$ and $b$ are always the same since the HPL test uses a fixed random seed. 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.

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

**Table 2.** Node configuration of Tianhe-1A and Tianhe-2.

## 6. Evaluation

### 6.1 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 CPUs and larger memory size than its predecessor, Tianhe-1A. However, each processor core of Tianhe-1A has more memory than that of Tianhe-2 (4GB/core vs. 2.4GB/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% 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.

### 6.2 Comparison with State-of-the-Art Methods

In this subsection, we compare SKT-HPL with state-of-the-art methods for fault-tolerant HPL, including an algorithm-based fault tolerant HPL (ABFT-HPL) [36], a traditional checkpoint-and-restart method (BLCR) [19], and a multi-level checkpoint system (SCR) [23]. As some experiments have special requirements, e.g., mounting ramfs, installing SSD (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. 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, and the group size is set to 8. Table 5.2 shows experimental results.

| | 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 | 234240 | 2338.64 | - | 3669.81 (0 chkpt) | 4.00 | 100.00% | NO |
| ABFT | 212224 | 2208.55 | - | 2885.00 (0 chkpt) | 3.28 | 78.61% | NO |
| BLCR+HDD | 234240 | 2338.64 | 295.20 | 2661.83 (3 chkpt) | 4.00 | 72.53% | YES |
| BLCR+SSD | 234240 | 2338.64 | 111.92 | 3209.08 (3 chkpt) | 4.00 | 87.45% | YES |
| SCR+Memory | 129280 | 426.18 | 4.33 | 3380.02 (0 chkpt) | 1.22 | 92.10% | YES |
| SKT-HPL | 154880 | 709.84 | 6.21 | 3467.64 (1 chkpt) | 1.75 | 94.49% | YES |

**Table 3.** Comparison between different methods of fault-tolerant HPL.

In our experiment, ABFT-HPL [36] 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% of the original performance.

BLCR [19] 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, the HPL performance is very poor, only achieving 72.53% of the original HPL, shown with BLCR+HDD in Table 5.2. 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% 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 [23] is a state-of-the-art multi-level checkpoint system developed by Lawrence Livermore National Laboratory (LLNL), 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% 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% of the original HPL performance.

Thanks to the self-checkpoint mechanism, SKT-HPL has 1.75GB memory for the HPL test, which is 43% higher than the SCR method. Among these methods, SKT-HPL achieves the best performance for the HPL test and it is 94.49% 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.

### 6.3 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 those nodes and remove those nodes from the resource pool of the 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.

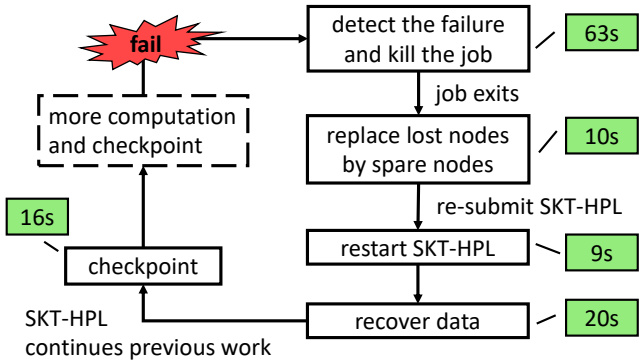


**Figure 10.** Time for each phase during a work-fail-detect-restart cycle. All results are measured on Tianhe-2 with 24,576 MPI processes. The time value for each phase is presented in the small green rectangles.

We further measure the time for each phase during a work-fail-detect-restart cycle, as shown in Figure 10. 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 seconds 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).

### 6.4 Performance of SKT-HPL

We perform the original HPL test on both systems with typical configurations. We obtain 15.55 TFLOPS (86.38% 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 power. Instead, we run HPL for minutes and record its actual FLOPS value. The performance on Tianhe-2 is 367.04 TFLOPS (84.94% of the theoretical peak performance), with 24,576 processes.

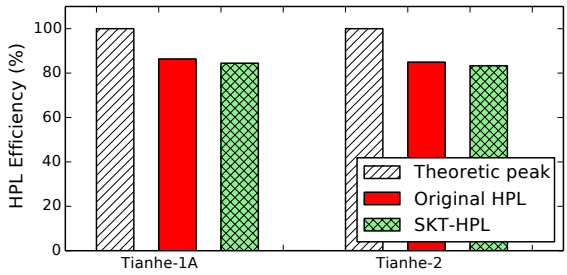


**Figure 11.** The efficiency of the original HPL and SKT-HPL (without making checkpoint). The original HPL uses full memory and SKT-HPL uses near half of the memory.

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% of the total memory on both systems respectively. SKT-HPL obtains 15.21 TFLOPS (97.81% of the original HPL) on Tianhe-1A and 351.60 TFLOPS (95.79% of the original HPL) on Tianhe-2. Figure 11 shows the performance of the original HPL and SKT-HPL.

### 6.5 Benefits of Self-Checkpoint

To verify the efficiency models presented in Section 4 and demonstrate the benefits of the self-checkpoint mechanism, we run SKT-HPL with different memory size on two large systems. Figure 12 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 behavior between problem size and HPL efficiency. Using the self-checkpoint can get 5% higher performance than using the double-checkpoint on Tianhe-2 because of its much more available memory (44% vs. 30%).

### 6.6 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). The local overwriting

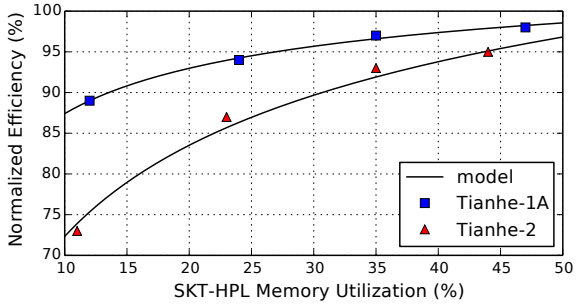


**Figure 12.** The relationship between Memory space used for computation and the normalized efficiency. The memory space and efficiency are compared with a typical run of the original HPL with full memory. The impact of memory space is more significant on Tianhe-2 than on Tianhe-1A.

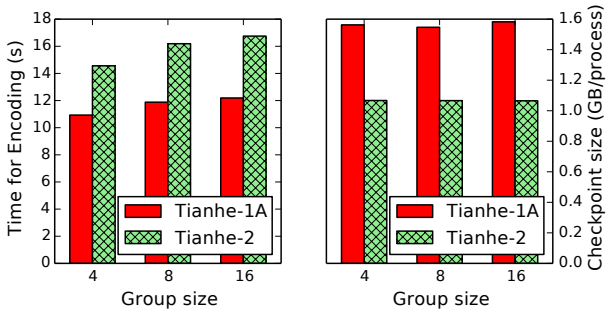


**Figure 13.** The encoding time of calculating checksums (left) and checkpoint size (right) with different group sizes. Encoding time grows slowly with group size. Checkpoint size is not very sensitive to group size.

time is normally less than one second, and is insignificant compared with the communication time used for the encoding. Figure 13 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 2, the checkpoint size is similar for different group sizes. The encoding time on both systems increases slowly with the group size.

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. Related Work

Checkpoint-and-restart (CR) [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. As traditional CR methods save checkpoints to a parallel file system [30] 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 [27]. Plank et al. proposed an incremental diskless checkpoint system [26] to reduce memory consumption. In their method, the size of a checkpoint buffer does not need to be equal to workspace. Instead, a buffer should be large enough to save the original version of data modified during a checkpoint interval. This incremental checkpoint method is good for applications with small memory footprint. It is a double-checkpoint method that requires two buffers for checkpoint, while one buffer is needed in our self-checkpoint method. 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 [37, 38]. This scheme can only use one third of the memory, making it more suitable for applications with little memory consumption.

Besides in-memory checkpoint, multi-level CR models have been proposed, such as SCR [23], 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 (ABFT) for classic matrix operations such as matrix-matrix multiplication and LU decomposition [20]. Yao et al. proposed a fault tolerant HPL [35]. Besides HPL, some other algorithms such as iterative methods and QR decomposition have also been studied with ABFT [7, 34]. Fault tolerant applications based on ABFT usually have low overhead. Chen et al. proposed Online-ABFT [8], which can detect soft errors in the widely used Krylov subspace iterative methods. Li et al. [22] 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 its data.

To the best of our knowledge, no MPI runtime can tolerate node failure with negligible overhead. MPICH-V [5], MPICH-V2 [6], 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. However, their 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 also 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%.

## 8. Conclusion

To reduce the memory usage of in-memory checkpoint, we propose a new strategy, called self-checkpoint. The self-checkpoint 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%. 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% than the state-of-the-art in-memory checkpoint. Moreover, SKT-HPL achieves over 95% of the original HPL's performance, and is 5% higher than using previous in-memory checkpoint methods.

## Acknowledgments

We would like to thank our shepherd Prof. Narayanasamy and the anonymous reviewers for their insightful comments and suggestions. Also, we thank Yi Yang, Zhen Zheng, Heng Lin, Haojie Wang for their valuable feedback and suggestions. This work is partially supported by National Key Research and Development Program of China 2016YFB0200100, National High-Tech Research and Development Plan of China (863 project) No.2015AA01A301, National Natural Science Foundation of China No.61232008, Tsinghua University Initiative Scientific Research Program, and Microsoft Research Asia Collaborative Research Program FY16-RES-THEME-095. The corresponding author of this paper is Jidong Zhai (Email: zhaijidong@tsinghua.edu.cn).

## References


[1] top500 website. http://top500.org/.

[2] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286. ACM, 2004.

[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 *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063427.

[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 *Springer-Link*, pages 477–488. Springer Berlin Heidelberg, Aug. 2012. URL http://link.springer.com/chapter/10.1007/978-3-642-32820-6_48. DOI: 10.1007/978-3-642-32820-6_48.

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, Nov. 2002. doi: 10.1109/SC.2002.10048.

[6] 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 *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 25–, New York, NY, USA, 2003. ACM. ISBN 1-58113-695-1. doi: 10.1145/1048935.1050176.

[7] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0552-5. doi: 10.1145/1996130.1996142.

[8] Z. Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442533. URL http://doi.acm.org/10.1145/2442516.2442533.

[9] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065973. URL http://doi.acm.org/10.1145/1065944.1065973.

[10] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing*, pages 162–171. ACM, 2011.

[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 *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 57:1–57:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654117.

[12] A. Duarte, D. Rexachs, and E. Luque. An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI. In B. Mohr, J. L. Trff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 4192 in Lecture Notes in Computer Science, pages 150–157. Springer Berlin Heidelberg, Sept. 2006. ISBN 978-3-540-39110-4 978-3-540-39112-8. URL http://link.springer.com/chapter/10.1007/11846802_26.

[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. *The Journal of Supercomputing*, 65(3):1302–1326, Sept. 2013. ISSN 0920-8542, 1573-0484. doi: 10.1007/s11227-013-0884-0.

[14] N. El-Sayed and B. Schroeder. Reading between the lines of failure logs: Understanding how HPC systems fail. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[15] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 1908 in Lecture Notes in Computer Science, pages 346–353. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41010-2, 978-3-540-45255-3.

[16] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063443.

[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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.

[18] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed Diskless Checkpoint for Large Scale Systems. pages 63–72. IEEE, 2010. ISBN 978-1-4244-6987-1. doi: 10.1109/CCGRID.2010.40.

[19] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[20] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.

[21] C. Jin, H. Jiang, D. Feng, and L. Tian. P-code: A new raid-6 code with optimal properties. In *Proceedings of the 23rd international conference on Supercomputing*, pages 360–369. ACM, 2009.

[22] D. Li, Z. Chen, P. Wu, and J. S. Vetter. Rethinking Algorithm-based Fault Tolerance with a Cooperative Software-hardware Approach. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 44:1–44:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503226. URL `http://doi.acm.org/10.1145/2503210.2503226`.

[23] A. Moody, G. Bronevetsky, K. Mohror, and B. De Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov. 2010. doi: 10.1109/SC.2010.18.

[24] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116. ACM, 1988.

[25] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. `http://www.netlib.org/benchmark/hpl/`.

[26] J. S. Plank and K. Li. Faster checkpointing with N+1 parity. In *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*, pages 288–297, June 1994. doi: 10.1109/FTCS.1994.315631.

[27] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10): 972–986, 1998.

[28] Y. Robert. Fault-tolerance techniques for computing at scale. *CCGrid2014*, 2014.

[29] B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct. 2010. ISSN 1545-5971. doi: 10.1109/TDSC.2009.4.

[30] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, Mar. 2007. doi: 10.1109/IPDPS.2007.370307.

[31] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid full/incremental checkpoint/restart for mpi jobs in hpc environments. In *International Conference on Parallel and Distributed Systems*, 2011.

[32] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas. Building algorithmically nonstop fault tolerant MPI programs. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–9. IEEE, 2011.

[33] S. B. Wicker and V. K. Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[34] P. Wu and Z. Chen. Ft-scalapack: Correcting soft errors on-line for scalapack cholesky, qr, and lu factorization routines. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 49–60, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7. doi: 10.1145/2600212.2600232.

[35] E. Yao, M. Chen, R. Wang, W. Zhang, and G. Tan. A New and Efficient Algorithm-Based Fault Tolerance Scheme for A Million Way Parallelism. *arXiv preprint arXiv:1106.4213*, 2011.

[36] 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. pages 438–448. IEEE, May 2012. ISBN 978-1-4673-0975-2, 978-0-7695-4675-9. doi: 10.1109/IPDPS.2012.48.

[37] G. Zheng, L. Shi, and L. V. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *IEEE International Conference on Cluster Computing*, pages 93–103, Sept 2004.

[38] G. Zheng, X. Ni, and L. V. Kal. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.