Front. Comput. Sci., 2023, 17(4): 174105
https://doi.org/10.1007/s11704-022-2096-3

REVIEW ARTICLE

Software approaches for resilience of high performance computing systems: a survey

Jie JIA (✉)[1,2], Yi LIU[1,2], Guozhen ZHANG[1,2], Yulin GAO[1,2], Depei QIAN[1,2]

1 School of Computer Science and Engineering, Beihang University, Beijing 100191, China
2 Sino-German Joint Software Institute, Beihang University, Beijing 100191, China

Abstract With the scaling up of high-performance computing systems in recent years, their reliability has been descending continuously. Therefore, system resilience has been regarded as one of the critical challenges for large-scale HPC systems. Various techniques and systems have been proposed to ensure the correct execution and completion of parallel programs. This paper provides a comprehensive survey of existing software resilience approaches. Firstly, a classification of software resilience approaches is presented; then we introduce major approaches and techniques, including checkpointing, replication, soft error resilience, algorithm-based fault tolerance, fault detection and prediction. In addition, challenges exposed by system-scale and heterogeneous architecture are also discussed.

Keywords resilience, high-performance computing, fault tolerance, challenge

1 Introduction

In the past decades, the performance of high-performance computing (HPC) systems has been increasing continuously. At the same time, the system scale has also grown rapidly. E.g., according to the TOP500 list of supercomputers, the number of processor cores of Frontier and Fugaku [1] reaches 8,730,112 and 7,630,848 respectively. Future HPC systems consist of tens of thousands of nodes and tens of millions of processor cores homogeneously/heterogeneously, exposing much more severe challenges to system resilience.

Theoretically, the reliability of a HPC system depends on the reliability of individual components and the number of components. Considering that the reliability of components is relatively stable under current manufacturing techniques, it can be concluded that the bigger the system scale becomes, the higher the probability of failure is. According to statistics, Peta-scale HPC systems' MTBF (Mean Time Between Failures) is about 15 hours [2]. On the one hand, considering the rapid increase of system scale, the MTBF of exa-scale HPC systems will be several hours or even less. On the other hand, due to the long-running characteristic of HPC applications, the execution of HPC applications will more likely be interrupted by different kinds of failures. Therefore, it is crucial to provide resilience approaches to guarantee parallel programs' correct execution and completion.

Resilience involves multi-layers of HPC systems. Various methods and systems have been proposed, either in hardware, software, or hybrid, in which the software/hybrid approaches mainly focus on system-/application-level reliability. This paper surveys software resilience approaches for HPC systems. The main contributions of this paper are as follows:

1) This paper provides a comprehensive and systematic survey of existing software resilience approaches for HPC systems. The approaches are introduced and summarized according to a classification that covers not only traditional fault-tolerant methods but also newer techniques such as silent data corruption and ABFT methods. We believe this survey can help researchers understand the progress and the overall picture of HPC software resilience.

2) This paper discusses challenges for software resilience approaches regarding recent developments of HPC systems, mainly in scalability and heterogeneous architecture. In addition, we discuss the emerged challenge of the fail-slow fault, which arises along with the scaling-up of supercomputers and needs more attention in the future.

The rest of this paper is organized as follows. Section 2 introduces the background of resilience. Section 3 presents a taxonomy of software approaches for HPC resilience. Sections 4 – 8 introduce different resilience approaches, including checkpointing, replication, soft error resilience, algorithm-based fault tolerance (ABFT), fault detection and prediction. Finally, Section 9 summarizes the challenges of current resilience approaches and concludes the paper.

2 Background

2.1 Resilience problem of HPC systems

With the scaling up of HPC systems in recent years, the reliability has been descending continuously. Research [3] has demonstrated that the MTBI (Mean Time Between Interrupt) will decrease along with increasing nodes in a HPC system. Besides, the growth of system scale is accompanied by the

Received February 16, 2022; accepted September 6, 2022

E-mail: jj@buaa.edu.cn

increase of clock frequency, and the number of processor cores and sockets in one node [4]. Currently, the world's state-of-the-art exa-scale supercomputer Frontier has more than 9,400 CPUs and 37,000 GPUs. However, with more components equipped in a supercomputer, the reliability will get worse. $e^{-\lambda_i t}$ is the exponential distribution of failure probability for one component in parallel execution is formulated as Eq. (1), where R_p is the reliability of a HPC system, m is the number of components connected in parallel in a HPC system [5]. Furthermore, the development of the processors also brings shrinking of manufacturing techniques which causes more soft errors [6].

$$R_p = 1 - \prod_{n=1}^{m} \left(1 - e^{-\lambda_i t}\right). \quad (1)$$

A study on the failure of large-scale HPC systems in the past decade demonstrates that the newer generation of HPC systems might have smaller MTBF than the previous generation, mainly due to the increased number of nodes [7]. However, the failure rate can also increase, even ruling out the impact of the number of nodes in the system. The detailed comparison is shown in Table 1. The table shows that the average MTBF of several Peta-scale HPC systems, XT5, XK6, and XK7, is about 15 hours. Particularly, if we compare the MTBF of Jaguar XT4 and XT5, both of which are built with AMD processors but with a different number of nodes, we can see that the MTBF decreases along with the increase of system scale. According to [2], the probability of application-failure increases by 20x when scaling applications from 10,000 to 22,000 nodes.

2.2 Malfunctions of HPC systems

The cause of malfunctions in a HPC system could be very complicated. In the system layer, previous studies usually use the terminology of faults, errors, and failures to describe the malfunction [8–10]. The comparison of these concepts is shown in Table 2. This section will briefly discuss the impact of malfunctions and the most common types of errors and failures.

The frequency and type of failures can be different in different HPC systems [7,11–13]. Table 3 summarizes the most common abnormal states in HPC systems. Overall, the abnormal states of HPC systems can be divided into two classes. One class is commonly called the fail-stop failure, which indicates failures that cause the system or application to stop working. The other is the soft error that makes the system or application produce incorrect results instead of crashing or stopping working. This kind of error is also called the fail-continue error.

From [14,15], one can see that the distribution of failures and errors can be varied among different HPC systems. Software-related errors dominate in some HPC systems like Blue Waters [16]. But in other HPC systems such as Jaguar XT4, Jaguar XT5, Jaguar XK6, and Titan [17], hardware-related errors are equally or more dominant across systems. As we look inside a HPC system, the failure events are proven to have temporal recurrence properties [18–20]. Except for temporal recurrence, spatial recurrence properties also exist. Some nodes are easier to fail due to hardware impairment or an imbalanced load schedule.

3 Classification of resilience approaches

Table 4 gives the classification of typical resilience methods and their characteristics.

Both checkpointing and replication can deal with almost all fail-stop failures while keeping transparency to applications. However, replication approach needs to consume at least 2x resources, and therefore is not very attractive for large-scale systems and applications. In addition, checkpointing can be used together with other resilience methods to deal with soft errors. Due to the above reasons, checkpointing is currently

Table 1 The MTBF of different HPC systems [7]

System	MTBF(hours)	Cores	Nodes
Jaguar XT4	36.91	31,328	7,832
Jaguar XT5	22.67	149,504	18,688
Jaguar XK5	8.93	298,592	18,688
Titan XK7	14.51	560,640	18,688

Table 2 The terminology of HPC malfunctions

Concept	Explaining
Fault	Root cause of an error, usually physical defects or software bugs
Error	Deviation from the expected result
Failure	Fails to deliver correct service

Table 3 Abnormal states of HPC systems

Class	Meaning	Typical examples	Explanation
Fail-stop failure	Hardware and/or software stop working	Kernel panic	Kernel error from which the operating system cannot quickly recover.
		Node heartbeat fault	Exception when accepting the heartbeat from other nodes.
		Traps	Segmentation faults, trap invalid opcode.
		GFS failure	Failure of the global file system.
		Scheduler	Internal bugs of job scheduler.
		Acc failure	Failure of accelerators or co-processors.
		Storage failure	Storage system fails to work.
		Node hardware failure	Node fails due to power/cooling-system error, damage of hardware components, etc.
		Interconnect conjunction	Network connection is congested.
Soft error / Fail-continue error	System still works but the execution of application incorrect	SDC	Undetected silent data corruption.
		CFE	Control flow error.
		MCE	Memory check exception.

Table 4 Classification of typical resilience approaches

Resilience method	Checkpointing	Replication	Soft error resilience	ABFT	Fault detection and prediction
Redundancy data	System memory or application data space	Process data and message	N/A	Checksum of algorithm	N/A
Recovery method	Failure-rollback	Forward recovery	Error-restart	Error-restart	N/A
Overhead/cost	Medium	High	Medium	Low	Low
Generality	Systems and applications	Systems and applications	Systems and applications	Applications	Systems and applications
Ease of use or deployment	Easy	Easy	Hard	Hard	Medium
Limitation	Scalability	Resource consumption and scalability	Soft error only	Algorithm-dependent	Rely on other recovery methods

the most widely used resilience approach in HPC systems. However, the scalability of checkpointing is not satisfied for large-number of processes. The algorithm-based fault-tolerance, or ABFT, is the most lightweight approach but unfortunately algorithm-dependent, which means different schemes must be proposed for different algorithms. Soft error approaches are dedicated to soft error. Most of them focus on how to detect the soft error, and depend on checkpointing to recover the error. Similarly, fault detection and predication act as an alarm or trigger. Once a fault is detected or predicated, corresponding failure-recovery or proactive actions can be performed.

4 Checkpointing

Checkpointing, or checkpointing/restart (CR), records the snapshots of applications & systems during the execution of applications. In case of failure, it uses the previously saved checkpoint file to resume application execution. Checkpointing can be triggered periodically or by specific events such as a function call in the application or an exception. CR is important for programs that need to run a long time to avoid restarting from the beginning.

The checkpointing concept can be applied at all levels: system-level checkpointing, user-level checkpointing, and application-level checkpointing. System-level checkpointing is implemented by the kernel. It is convenient to use and provides resilience for all applications. User-level checkpointing is typically implemented in parallel libraries. The user-level resilience doesn't modify the kernel; thus, it is easier to migrate to other HPC systems. Application-level checkpointing means that the user needs to add checkpoint and restart functions in their applications. Usually, it means a lot of engineering. But it also allows the user to design more efficient resilience checkpointing mechanisms for their applications. The comparison of them is shown in Table 5.

In addition to the classic approaches, there are researches on emerging challenges. One of them is how to support checkpointing on heterogeneous architecture, e.g., GPUs. The other is how to leverage multi-level checkpointing. We will discuss those new areas in section 4.4 and 4.5 respectively.

4.1 System-level checkpointing

System-level checkpointing is implemented by the OS kernel. It is transparent for users and needs no modification to user codes. The synchronization between user and kernel space is avoided because the system-level checkpointing is usually implemented in the kernel. Many teams focus on transparently supporting checkpoints for user applications. They typically need to modify kernel modules and user-shared libraries to save an entire state [24,25].

BLCR (Berkeley Lab Checkpoint/Restart) is a famous system-level checkpointing tool for multi-threaded programs. It allows CR through libcr.so and modified Linux kernel. Once BLCR is fully initialized, it will spawn a callback thread to block system calls in the kernel. When checkpointing, the callback process sends broadcast signals to allow all threads to enter kernel space. After that, all threads will reach three barriers sequentially to achieve a consistent state. Upon reaching each barrier, relevant data is saved. Finally, each thread returns to the user space and continues execution.

Although BLCR does not support CR of network state, there are some system-level checkpointing works [26,27] based on BLCR, using BLCR to save system state within the node. They use a global controller acting as a coordinator to control numerous processes so as to support CR of MPI programs. However, this will trigger two serious overheads: the time spent coordinating between processes via messages (coordination overhead); and the time spent saving the state of entire systems to non-volatile storage (memory overhead).

Non-blocking checkpointing try to reduce the coordination

Table 5 Comparison of different checkpointing level

Checkpointing level	System-level	User-level	Application-level
Explanation	Operating system in charge of checkpointing.	A user-level library is responsible for checkpointing and links to applications	The application itself is in charge of checkpointing.
Typical systems	BLCR [21]	DMTCP [22]	FTI [23]
Checkpointing data	Status of entire system	Status of entire application	user-specified application status
Overhead	High	Medium	Low
Transparency	Transparent to applications	Application needs to be loaded or linked with checkpoint library	Application needs to be modified
Portability	Low	Medium	High

overhead by not reaching a globally consistent state. However, in practice, the domino effect among processes becomes unacceptable on restart. Non-blocking checkpointing on HPC parallel programs incurs an overhead close to the blocking approach.

Incremental checkpointing is one of the efficient ways to reduce the memory overhead. There are two kinds of incremental checkpointing, page-based and hash-based. Page-based incremental checkpointing requires OS support to find dirty memory pages [28]. This approach has two limitations. Firstly, it is difficult to meet the requirement for hardware and kernel support on some systems; Secondly, this approach cannot track the dirty bytes in a page. The hash-based incremental checkpointing, on the other hand, finds dirty memory blocks by calculating the hash value of blocks at a smaller granularity [29]. This will take more overhead, obviously. Although incremental checkpointing save memory usage, it consumes more computation in checkpointing, which can be regarded as a space-time tradeoff.

4.2 User-level checkpointing

In user-level checkpointing, the application is either compiled & linked with a modified library that integrates the checkpoint implementation, or loaded using a checkpoint utility to track the processes & threads of the application. The attraction of user-level checkpointing is that there is no need to modify the source code, while the checkpoint overhead is smaller than the system-level checkpointing.

One kind of user-level checkpointing directly modifies the linking parallel library, such as MPI. Thus, users can transparently use the checkpointing when linked with the library. A lot of research teams focus on modifying the MPI library [30–36]. Those methods track the message passed by the MPI library. Typically, there are several components in the implementation, including a single process checkpointing library, a process management system, and a library handling checkpointing requests during asynchronous message transportation.

User-level checkpointing support for inter-process communication (IPC) gets and saves the connections of the communication layer on user space, as well as the messages being delivered by these connections. When restoring, on user space, the inter-process connections are first restored and then resend the messages being delivered. Since MPI is implemented based on IPC, tools supporting IPC, such as DMTCP, DejaVu [37], actually support MPI as well. Further, although some user-level checkpointing tools were not originally proposed for HPC systems, they can still be used on HPC with the support for IPC and MPI libraries.

DMTCP (Distributed MultiThreaded CheckPointing) is a user-level checkpointing tool implemented with libraries. Its own library enables CR support for various applications such as C/C++, python. DMTCP instruments the system call in order to record system calls (fork, exec, ssh, etc.) of the user program. DMTCP uses a global coordinator to which all worker processes connect with sockets. The DmtcpCoordinator sends checkpoint barrier signals to all Dmtcpworkers to achieve a globally consistent state and save checkpoints at runtime. Based on DMTCP, [38] adds the support for Infiniband interconnect which is widely used in HPC with lower runtime overhead. [39] support checkpointing and restarting for HPC applications with combinations of all MPI implementations and interconnect by designing a single code with a split-process approach that allows a process to run two different programs.

The user-level checkpoint software mentioned above needs to recover all nodes' processes during recovery, which costs a large performance overhead. For MPI, Reinit [40–42] uses the method of storing checkpoints on both the local node and other nodes in user space, so that in the case of an error on a few nodes, error nodes can be restarted using checkpoints from other non-error nodes. Instead of DMTCP requiring all nodes to be restored during recovery, the overhead of checkpoints is reduced.

4.3 Application-level checkpointing

Application-level checkpointing (ALC) is controlled by the source code. The kernel is not aware of the checkpointing and restarting procedure. The ALC has a smaller performance overhead and storage overhead compared to system-level checkpointing and better scalability, because only the essential variables and data structures are saved. In addition, the application has more control over when to trigger the checkpointing. Compared with the system-level and user-level checkpointing, ALC is easier to migrate to different platforms. However, users are totally aware of checkpointing and need to modify the source code.

Early researchers [31,38,39] typically design a preprocessor and a checkpointing library. They are aimed at MPI or OpenMP applications on HPC, where code that saves variables and memory is inserted into the source program by compilation. They typically allow the programmer to decide the time of checkpointing by inserting function calls for setting checkpoints. One typical ALC library for HPC is FTI [23]. The library provides many application programming interfaces(APIs) for checkpointing. APIs provided by those libraries are fine-grained. The user needs to place the API call in the right place. For MPI support, FTI redesigned a set of interfaces based on the MPI interface and MPI communicator. These FTI interfaces can store MPI process and the information of message passing as checkpoints using Reed-Solomon (RS) encoding based on the MPI call. When the node is faulty, the data lost by the faulty node is recovered by means of RS decoding by the set checkpoint.

Bronevetsky et al. [43] further extends FTI by analyzing the reuse of data structure at the compilation time. In this way, the checkpoint states can be incremental. Only the dirty data structure needs to be saved.

Typically inserting checkpoint-related code into the application's source code requires extensive engineering efforts. Some researchers try to develop more easily used tools to help insert the checkpointing code. The Domain Specific Language (DSL [44]) automatically inserts the checkpointing code, with which the performance is comparable to the manually modified version. The user only needs to write a high-level specification on when and where to checkpoint.

Ba [45] goes further by analyzing the source code and providing an interactive command for user decisions. This method specially optimizes the memory overhead and supports multiple languages, including C/C++/MPI/OpenMP. The tool is built on top of the ROSE compiler [46]. Ba supports analyzing the checkpointing-and-restart logic, changes of for-loop code, order of function calls, etc.

Craft [47] provides a library for checkpointing specific data types, and the user can extend the data types supported. In addition, Craft uses the asynchronous checkpointing mechanism to reduce overhead and features node-level checkpointing.

4.4 Heterogeneous checkpointing

Heterogeneous architecture has been widely used in HPC systems. Unfortunately, the checkpointing with accelerators such as GPUs is often neglected. To our knowledge, CheCUDA [48] is the first work at checkpointing on heterogeneous systems. Due to the fact that we cannot suspend the GPU kernel, when saving checkpoints, it first waits for the GPU kernel to complete, then transfers the data from GPUs to the host and uses BLCR to preserve the status of the host and GPUs.

Garg et al. [49] research transparent checkpointing of GPUs and newer RDMA networks. They proposed a new transparent checkpointing framework based on split processes that use the hardware virtual memory of the host to decouple the computation state from the external subsystem context. The isolation between the application process and external computation systems makes it possible for transparent checkpointing.

Based on DMTCP, CRUM [50] and CRAC [51] are the latest work on Nvidia GPU checkpointing, which implement user-level checkpointing and supports CUDA's Unified Memory Architecture(UMA) architecture. CRUM uses a proxy process. CRUM first intercepts all CUDA calls, sends the CUDA calls to the proxy process, and the proxy process actually communicates with the GPU. In this way, the program's data excludes the CUDA context. CRAC takes a novel method of split process. The address space is divided into two parts, the upper half is the CUDA application, and the lower half is the GPU and kernel drivers. Upon checkpoint, only the upper half memory is saved. On restart, reinitialize the lower half and then the lower half restores the upper half memory from the checkpoint file. We could see that CRAC can achieve a much lower overhead than CRUM. However, CRAC only support single-node heterogeneous checkpointing.

Heterogeneous systems need to save all GPU data to the host when checkpointing. If multiple GPUs checkpoint simultaneously, it will cause a bursty buffer and degrade the performance. GPU-snapshot [52] addresses this problem by using the idea of partitioning GPU memory, combining the idea of incremental and asynchronous checkpointing to reduce the total amount and peak of IO. Different from GPU-snapshot, Heterocheckpoint [53] uses pre-copy and non-volatile memory (NVM) to address the problem, which can reduce the bandwidth impact by 60%.

4.5 Multi-level checkpointing

The different levels of checkpointing are not incompatible. They can work together to achieve an optimal solution in terms of overhead and error rates. Vaidya [54] proposes a two-level checkpointing model. The first level is a single process failure tolerance scheme. The processes periodically take checkpoints that need not be consistent with each other. The second level is the global checkpointing scheme. The processes coordinate with each other and ensure that their states saved on the stable storage are consistent with each other. The first level with lower overhead handles the more probable process errors, and the second with higher overhead tolerates the less likely system failures.

Haines et al. [55] combines the advantages of system-level and application-level checkpointing for the applications exhibiting data parallelism. System-level checkpointing can detect hardware failure and kernel failure. The application-level checkpointing is used as a complement for reliability when the system-level checkpointing fails to work.

Many studies have conducted a fine-grained analysis of the efficiency, frequency, and other checkpointing issues. Di [56] gives a theoretical analysis on the best checkpointing configuration with two levels. One checkpoint deals with soft errors like transient memory errors. Another checkpoint deals with hardware crashes. Benoit [57] further extends the theoretical analysis to the situation of multiple levels and derives the optimal checkpointing frequency in each level. An error at level l destroys all lower level checkpoints (from 1 to $l-1$) and requires a rollback to a checkpoint at level l or higher. The overhead of a k-level checkpointing HPC system can be described in Eq. (2), where λ_l and C_l are the error rate and checkpointing cost at level l, respectively. Based on Eq. (2), Benoit also proposes a dynamic programming algorithm for choosing the optimal checkpointing levels considering the actual situation.

$$o = \sum_{l=1}^{k} \sqrt{2\lambda_l C_l}. \tag{2}$$

5 Replication

The replication technique is a space-time tradeoff technique that increases resources used with decreased recovery time. It uses multiple redundant copies of process/computation to achieve resilience, enabling the program to finish in close to normal execution time. Replication is one of the few technologies that can handle both fail-stop and fail-continue errors. It handles fail-stop errors by executing each copy independently and handles fail-continue errors by comparing the results of copies. Replication is easy to understand and can significantly improve the mean time to interrupt (MTTI) of applications [58]. It is mostly used for reliability assurance of vital parts of large-scale computing. There are two major types of replication: process-pairs and N modular redundancy (NMR).

Process-pairs are at the process level. Process-pairs consist of two types of processes: a primary process and a backup process, where the backup process is a clone of the primary

process. The processes of each process-pair run simultaneously on different processors. Note that the primary process also needs to transfer messages to the backup process through MPI. By doing so, the backup process can continue to work when the primary process crashes. This allows the application to continue executing even if the primary process fails. In principle, process-pairs consume twice the number of resources than the un-replicated execution. The output of a process-pair is usually dependent on the primary process. Sometimes, although the backup process is correct with the incorrect primary output, the consequence is still wrong. Therefore, in some areas, process-pairs can be combined with checkpointing. When the output of the primary process and the backup are different, we can roll back to the nearest checkpoint and restart.

NMR is at the computation level. NMR creates N identical modules (usually $N = 3$) that run simultaneously, and determines the output by a voting principle. Therefore, the result with the highest amount of N modules becomes NMR's final output. As we can see, the larger the N is, the more reliable the system is. NMR is widely used to detect fail-continue failures [59], such as the silent data corruption (SDC). NMR can improve reliability significantly but with the highest cost among available resilience approaches.

Replication is widely used for resilience efforts in MPI applications. rMPI [60] transparently provides replication for MPI applications. rMPI library is inserted between an application and an MPI library that uses the MPI profiling interface at the linked time. In the case of dual redundancy, the program is started on $2n$ computing nodes, and n denotes the original execution scale. The application can see rank 0 to rank n–1, and rMPI uses the remaining nodes to conduct redundancy. Each redundant node is maintained by rMPI and duplicates the work of its active partner. To avoid the divergence results from the non-deterministic MPI functions (e.g., receive operations with wildcard, probe operation for the incoming messages), rMPI library coordinates and duplicates messages between redundant nodes and forces an active node and its mirror to return the same values for the MPI function to the application. However, synchronization protocol and additional messages cause very serious overhead.

MMPI [61] proposes a protocol set of redundant execution for MPI applications. It has different replication partitioning and comparison schemes. MMPI depends on cumbersome source code modification for implementing these redundancy protocols.

The biggest disadvantage of redundancy is that it consumes more resources and is less efficient. The hot issue of replication for HPC is to reduce the volume of replication. Hussain [62] demonstrates through experiments that partial replication typically yields higher performance with different node failure rates than full replication. Elliott et al. [63] combine partial replication with checkpointing. They proved by fault injection within MPI applications that the proper degree of redundancy and checkpointing frequency achieves the maximum performance. In their experiments, 2x redundancy worked excellently. George [64] adaptively and dynamically alters the number of replicated process sets based on the fault prediction to minimize energy usage. However, the partial replication scheme needs to be customized according to the HPC applications, and optimal partial replication is an NP-hard problem. So partial replication cannot make good generality.

6 Soft error resilience

The soft error, also called fail-continue error, is a kind of fault different from the fail-stop errors. That is, the system and application will not crash or stop running in the case of soft errors. Instead, they produce incorrect outputs. The soft error is generally caused by bit-flipping in memory or processor due to radiation or hardware error. The rate of bit-flipping can increase significantly due to the adoption of fabrication miniaturization, aging of silicon, and the dynamic power management cycles [65,66]. According to the error behavior in programs, the soft errors can be divided into two types: the control flow error (CFE) and the silent data corruption (SDC).

6.1 Control flow error

CFEs are caused by the transient and permanent faults that occurred in hardware components (e.g., program counter, address circuits or memory subsystem), which will affect the correctness and predictability of instruction execution sequence in processors.

Control-flow checking (CFC) is a critical method for monitoring program execution flow by dividing the program into basic blocks and assigning a pre-computed signature for each block. Afterward, obtain the runtime signature during program execution and compare these two signatures to determine whether an illegal jump happens. The schemes are used to generate signatures for basic blocks by various methods. For example, generate a unique global static signature for every basic block or compute given the block code in compile time, then re-compute a runtime signature during program execution. Suppose one CFE happens during program execution, resulting in an illegal jump inside a basic block. In that case, the sequence of instructions executed differs from the one obtained at compile-time, such that divergence appears between the pre-computed and the re-computed signatures. Therefore, this CFE can be detected at the end of a basic block by comparing signatures. The CFC can be realized by hardware-based or software-based approaches. Since hardware-based approaches are beyond the scope of this survey, we only introduce software-based approaches.

The software-based error detection technique employs encoded signatures (SWTES) [67] assumes that the program is divided into several blocks containing basic blocks and partition blocks. It defines and considers seven types of CFEs between these blocks during the program execution. This technique is based on the signature assignment for each block and signature comparison at the end of these blocks. It consists of two primary parts, i.e., the labeling algorithm and the signature generating step. The block signature self-checking (BSSC) [68] method assigns a signature for each basic block and checks for control flow at the end of the block. It calls a subroutine at the start of the block used to push the address of its first instruction as the signature to the

top of the stack or store it in a static variable. Another subroutine is invoked at the end of the block to compare the embedded signature and the one preserved by the entry routine. The control-flow error detection through assertions (CEDA) [69] method allows extra instructions embedded into the program automatically at compile-time for constantly updating the run-time signature, which will be compared with the pre-assigned signature. Due to the different ways of calculating signatures, CEDA inserts fewer instructions and brings less overhead but higher efficiency.

After detecting a CFE, control should be transferred back to the block where the illegal branch happens. However, it is not enough to just correct the CFE because the program may fail due to the data corruption caused by the CFEs.

The automatic correction of CFEs (ACCE) divides the program code into functions with one or more basic blocks. ACCE uses CEDA to detect CFEs, automatically executes a predefined function termed as error-handler, and transfers the control to this function, next transfers to the basic block where the illegal jump happens. Automatic correction of control-flow errors with duplication (ACCED) is an extension of ACCE. It is used to detect data corruption and correct it by duplicating instructions. However, not all data corruptions can be detected, and this method results in serious performance overhead (more than 100%).

Although there are some researches [70] for automatic detection & correction of CFEs, the complexity and overhead of CFEs correction make them not applicable to large-scale HPC systems. The feasible solution is combining CFE detection approaches with checkpointing. Considering checkpointing as a CFEs correction approach, we can roll back to the previous checkpoint after detecting CFEs.

6.2 Silent data corruption

In silent data corruption (SDC), bit-flipping occurs in application data (e.g., a matrix) which will not cause abnormal behavior but produce error results. Therefore, SDC is more difficult to detect than the CFE.

Given the definition of SDC, it cannot be detected at low-level hardware or software stacks. One approach to detect the SDC is outlier detection. The idea of this approach is: higher-level software can discover outliers by leveraging properties of the data dynamics, and these outliers indicate the appearance of data corruption. [71] proposes an SDC detection meth-od based on data monitoring, by which it learns normal dynamic of their datasets and quickly locate abnormalities. [72] utilizes historical data to build a trend model of data change for predicting the data point's value in the next iteration(temp-oral or spatial neighbor point next to the current one) and then compares the predictive value with the actual value derived from program execution. Therefore, SDC can be determined if the difference between the two values is beyond a given threshold.

Another approach combines the replication mechanism with message verification to detect SDC. Most of them are implemented in MPI-layer. This kind of approach assumes that any data corruption in a process will cause contaminated messages which will be transmitted to other processes. Therefore, data corruption can be detected by comparing the messages of the equivalent ranks from different replications. Typical works include rMPI [60], MMPI [61], VoplexMPI [73] and MR-MPI [74]. rMPI and MMPI are introduced in Section 5.

Berrocal [75] proposes a partial replication mechanism to reduce the overhead of full replication in SDC detection. It consists of two adaptive algorithms. In one of them, the number of the processes to be replicated is invariant through the whole execution. In another, this number can change dynamically. To improve the overall recall rate and, meanwhile, lead to a small overhead on time and space, the processes expected to be replicated should be chosen carefully according to their data behaviors.

The third approach tracks access to memory pages and uses the hash-value of pages to detect data corruption in those pages. To some extent, this approach is a software-implemented page-level memory checking mechanism. Typical works include LIBSDC [76], FlipSphere [77], and Mini-ckpt [78].

LIBSDC [76] protects against SDC in memory at the system level. It intercepts access operations to memory pages and triggers signals if these pages are within the scope of protection. The handler of the signal is responsible for computing a hash value of the memory page, then comparing this value with the last known good one to see whether divergence appears. This method is only suitable for DRAM but not for other components (e.g., CPU, registers, and logical devices). Besides, it lacks the corresponding mechanism to determine whether the divergence is caused by SDC or calculation operations.

Based on LIBSDC, FlipSphere [77] traces page accesses by using MMU and removes the least recently used page permissions when a new page is accessed, to provide software-based level protection.

Mini-ckpt [78] considers that applications cannot continue if SDC happens in the memory occupied by the operating system and causes kernel panic. For avoiding rollback, recovery, and calculation loss, mini-ckpt records the kernel state information of the running application to non-volatile memory. Afterward, restart the kernel and recover the target application's execution state. This method requires modifications to the operating system kernel. The overhead it introduced relies on the procedure of the error handler after a kernel panic.

Here, we make a comparison of checkpoint/restart, replication and ABFT in SDC challenge, shown in Table 6, and ABFT will be presented in the next section.

Table 6 Software solutions for SDC challenge

Approach	Advantages	Disadvantages
Checkpoint/restart	No hardware features required, less or no program modification	Requires large storage space and high time overhead
Replication	Simple and straightforward	High overhead, including running time, computing resources
ABFT	Low-overhead	Required program code modifications, and poor portability

7 Algorithm-based fault tolerance

Considering that for most scientific applications, the execution process is multiple calls of a limited number of underlying algorithms due to their generality. The algorithm-based fault tolerance (ABFT) indicates resilience approaches dedicated to specific algorithms. It is based on the relationship between the input data and checksum of a particular algorithm to detect and correct errors. If the checksum is incorrect, an error has occurred. The first ABFT scheme [79] uses a row-column checksum to detect and correct errors in matrix operations such as multiplication, transpose, and LU decomposition on multiprocessor systems. Compared to checkpointing and replication, ABFT can ensure resilience with the lowest cost. Unfortunately, it is algorithm-specific, which requires extra effort to design different schemes for different algorithms. Compared with checkpointing or replication, ABFT is lower in overhead and easier in computation complexity. However, ABFT can only deal with errors with limited corruption.

7.1 ABFT for linear algebra

On the basis of ABFT for matrix multiplication, [80–82] extend ABFT to matrix decomposition and basic linear algebra operations such as LU, QR, Cholesky. These schemes are for offline problems. Offline means that the detection and correction of errors occur after the matrix operation and can be used for general matrix decomposition problems without considering specific algorithms. Although offline ABFT techniques usually have a minimal runtime overhead, they cannot prevent the propagation and accumulation of errors. When the number of failures exceeds the checksum correction capability, matrix re-computation is required, so ABFT is insufficient to provide fault tolerance for large-scale long-running applications.

Therefore, [83,84] focus on online variants of matrix multiplication. Online means that fault detection and correction are performed continuously during the computation, not only at the end of the computation. Online variants need to design ABFT schemes based on specific matrix algorithms to detect and correct errors timely to prevent error propagation. For example, [83] uses the iterative method for the Krylov subspace to dynamically verify the orthogonal relationship and residuals of the program during the calculation.

Besides ABFT for dense linear algebra, recent work also concerns sparse matrix. Many scientific applications on HPC usually involve sparse linear algebra, so it is important to develop the ABFT scheme for sparse linear algebra. Due to the different data structures of sparse matrices (e.g., CSR, COO), traditional ABFT schemes are not suitable for sparse matrices. Recent researches reduce the runtime cost of error detection and correction for the data structures and calculation characteristics of the sparse matrix for SpMV [85], PCG [86], etc.

It should be pointed out that many ABFT studies mainly focus on fault tolerance at the integrated circuit level rather than HPC systems. In HPC systems, most parallel matrix algorithms, such as Cannon, Fox, High Performance Linpack (HPL), etc., divide the matrix into blocks and assign blocks to different nodes and processes to achieve scalability and flexibility. Therefore, we cannot directly use traditional ABFT mechanisms based on checksums of rows and columns on HPC systems.

Zhu [87] proposes a matrix fault-tolerant mechanism for large-scale parallel systems based on block-checksum. He divides the matrix into blocks and adds checksums in each block. These blocks are then put into the parallel matrix multiplication. In the mathematical operations, the computation of block-checksum is 1/k of the original row-column ABFT, where k is the size of the block. However, the larger the k, the more prone to rounding errors. To achieve a balance between accuracy and efficiency, he proposes an approach to selecting the appropriate size of blocks. Furthermore, the blocked-based checksum is used to implement FT-PBLAS [88], a fault-tolerant version of PBLAS.

7.2 ABFT for fail-stop errors

Most traditional ABFTs need to maintain the relationship of checksum during the computation, making them only suitable for fail-continue errors. Because the relationship is not available for fail-stop failures [89]. By using redundant checksum in memory during computation, researchers [90,91] can recover the failed processor when processing fail-stop failures, and realize diskless checkpointing. For instance, [90] sums the vector data of all processes according to a certain weight to get vector y, and saves vector y to another process; When a process X crashes, the data of X can be recovered by the vector y. In the case of a single process crash, diskless checkpoints can avoid storage overhead and greatly reduce the time required for rollback. It is possible to recover data from multiple processes based on mathematical methods.

Davies et al. [92] demonstrate that the checksum is maintained at each step of the computation for the right-looking LU factorization algorithm. Based on this, they propose a scheme to tolerate the failure-stop Errors of Cholesky decomposition based on HPL's two-dimensional block-cyclic distribution. After one process crashes, it can recover data from the original matrix and U. Although the scheme is specific to matrix operations, it can offer lower overhead than diskless checkpoints.

7.3 ABFT on heterogeneous architecture

Similar to traditional homogeneous systems, heterogeneous systems often have errors. There have been some studies on ABFT schemes for heterogeneous systems in recent years. These researches fall into two aspects: porting the ABFT scheme from CPU to GPU and increasing the reliability of ABFT on GPU.

Chen [93] proposes an online enhanced ABFT Cholesky decomposition scheme specially designed for heterogeneous system with GPUs. Checksums verification is the key operation in ABFT, and it is relatively expensive on GPU due to its low-performance efficiency. Since GPU kernels can run in parallel, Chen uses multi-concurrent CUDA kernels to accelerate the checksum verification. For the two one-sided matrix decomposition algorithms, including LU and QR, Chen further optimized the kernel of their encoding checksum

according to the GPU architecture, which provide a better ABFT solution [94].

GPUs commonly perform lots of floating-point computations, and their results are inevitably prone to rounding errors. Therefore, ABFT for GPU-based systems has to deal with rounding errors in checksums to avoid false positives. A-ABFT [95] uses a probabilistic model for rounding errors occurring in floating point operations to determine error bounds for the comparison of ABFT checksums for matrix multiplication.

8 Fault detection and prediction

Fault detection aims to find faults and perform fault recovery in time in order to prevent further propagation of the fault and reduce the impact on resilience. For example, by using a combination of fault detection and checkpointing, we can roll back to the latest checkpoint as soon as an error is detected.

The goal of fault prediction is to predict faults in advance and make proactive operations (e.g., process migration, save checkpoints). Even better, fault prediction can be used with forward recovery resilience techniques (replication, ABFT), which can counteract the impact of faults by redundant calculations.

8.1 Heartbeat-based fault detection

The most commonly used node-level fault detection technique is the Heartbeat. Each computing node sends heartbeat information to each other or to a controller node periodically to confirm it is alive. Suppose the heartbeat from a node is not received for the pre-defined period. In that case, the node will be regarded as failed, and the controller node may start error handling, e.g., removing the node from the available resources, restarting the job in the failed job.

Many fault detectors are working on the heartbeat mechanism. They usually communicate in an all-to-all way and use the heartbeat information to broadcast fault information within the HPC system. Since the all-to-all communication approach has a scalable problem in large-scale systems, [96] proposes a scalable heartbeat fault detection algorithm based on the gossip protocol, which uses random communication to propagate information instead of all-to-all communication.

8.2 Statistics-based fault detection and prediction

In HPC systems, different kinds of monitoring & performance data are used for system monitoring and management, such as CPU temperature, fan speed, voltage. Deviation of these data often indicates an abnormal state or oncoming failure. For instance, if the CPU temperature continues to rise and exceed a certain threshold, the node is likely to fail in a short time. Modeling these data according to statistics can help us carry out fault detection and prediction.

A widely used method is to measure the performance of a node while executing similar programs and compare them with other nodes. If its performance deviation exceeds the threshold, a fault is considered to occur in that node [97]. [98] models the normal activities of nodes by using the runtime data and then realizes fault detection by comparing the current running state of nodes with the normal activity model. [99] calculates the node system log generation frequency (SG) within a fixed time interval and compares it with similar nodes. If the SG parameter deviates from most nodes by more than a certain threshold, the node is considered to fail. [100] proposed a rule-based fault prediction method, which uses daemon process on each node to read sensors and compares it with the historical value in the system log to predict fault.

8.3 ML-based fault detection and prediction

HPC systems produce various kinds of performance data and system logs that are suitable for machine learning techniques to perform fault detection and prediction.

Various supervised machine learning (ML) algorithms, like neural networks, are trained to detect and recognize system failures on HPC [101,102]. Supervised machine learning fault detection is efficient and accurate in detecting known faults, but it is difficult to find unknown faults due to the fact that the training data is manually annotated. Dani [103] proposes an unsupervised anomaly detection approach that utilizes the k-means clustering algorithm to analyze logs and find active & failed nodes. [103] relies on the node itself to detect faults and logs the fault, which also limits the types of faults that can be detected.

Fault prediction is performed mainly by classification techniques, such as support vector machines (SVM), decision trees, and logistic regression. [104,105] use SVM to analyze data and predict disk faults. [106] utilizes the decision tree and logistic regression to predict disk faults, in which the decision tree is the primary classifier while the logistic regression is used to avoid overfitting. Desh [13] uses the long short-term memory (LSTM) neural networks for predicting node-level failures, in which the LSTM is trained with the log event chain that caused the failure as well as the time interval between the log event chain; Experiments show that it can predict the failure 3 minutes in advance.

AIOps (Artificial intelligence for IT Operations) combines big data, ML and advanced operation technologies, and has already started to be used on HPC. During the training phase, it uses ML to learn the normal working patterns of HPCs from their massive data. At runtime, AIOps collects real-time data and analyzes it to detect and predict faults and make automatic intelligent decisions. As a result, AIOps can detect and avoid various faults faster than operation teams, reducing the frequency of failures and shortening HPCs' MTTR (Mean Time To Repair).

8.4 MPI-based fault detection

Most applications in HPC systems rely on MPI for inter-process communications, so it is possible to detect faults of parallel programs in MPI-layer. Marmot [107] and Umpire [108] utilize the MPI profiling interface to detect program faults (e.g., deadlocks), but they may have problems in scalability and performance for large-scale applications. To improve scalability, SRFD [109] introduces a scalable runtime fault detection mechanism that uses MPI library functions to obtain abnormal runtime information and logically constructs a fault detection tree (FDT) for each process. Depending on the detection period and data structure, Kharbas [110] uses random detection and ring-based periodic detection at the MPI

communication layer to detect faults in MPI programs. These fault detection mechanisms can be embedded in the MPI application without any modification to the MPI application.

8.5 Correlation-based fault prediction

The basic idea of correlation-based fault prediction is to estimate the probability of the next fault within a short period by finding the correlation between faults according to the time and type. The system log is usually the main research direction of correlation-based fault prediction. [111] analyzes BlueGene/L's RAS logs and found the time characteristics (50% of network failures and 35% of I/O failures occurred within 30 minutes of the last failure) and spatial characteristics (6% of nodes suffered 61% of network failures) of failures. Based on this, they proposed a strategy: when a job submits to HPC after one fatal event or two non-fatal events, it is highly likely to fail.

Gainaru [112,113] combine signal processing to analyze the logs and introduce two fault prediction methods; They extract three kinds of signal events from the logs and carry out anomaly detection based on the signals; By mining the correlation between signals and the spatiotemporal correlation, more than 50% of faults could be predicted in one minute in advance.

Pelaez [114] proposes a distributed online node fault prediction scheme to improve scalability. The idea is to divide the data into regions and assign each region to a specific work node. Then the work node conducts cluster analysis to get clusters and outliers. This method provides good scalability with high accuracy and ensures low data transmission and RAM usage.

9 Challenges and conclusion

9.1 Challenges

● Challenges on scalability

The development of HPC systems not only promotes the performance but also increases the number of processors/cores/nodes. To make full use of the computing resources of the HPC system, there will be massive processes/threads in parallel programs. The increase in the number of nodes and processes/threads poses challenges to the scalability of current resilience approaches.

The challenge on scalability of checkpointing mainly lies in two aspects. Firstly, the globally coordinated checkpointing requires synchronization among processes to achieve a globally consistent state before saving the checkpoint. For large-scale HPC systems, this kind of global communication is time-consuming and inefficient. Secondly, massive processes/threads will generate massive checkpoint data that needs to be saved to the storage-system. When thousands of nodes save checkpoints to shared storage simultaneously, the I/O sub-system will experience severer overloading. Therefore, checkpointing requires further efforts on reducing data volume, optimal checkpoint intervals, and uncoordinated checkpoints to improve scalability.

● Challenges of heterogeneous architecture

Heterogeneous architecture has been widely employed by HPC systems to achieve high-density of performance and to improve power efficiency, which makes software resilience approaches more complicated. Taking GPU as an example, popular GPUs currently have an independent memory space, which needs extra handling for checkpointing, SDC, and ABFT. Other challenges include synchronization between CPU and GPU, coordination of stream processors, saving and restoring GPU status.

Traditional checkpoint approaches and tools are mainly for CPU systems and take no account of heterogeneous systems. Although there are already some checkpointing approaches for CPU-GPU, they have limitations more or less. For instance, checkpointing can only be done between kernel-functions of GPU rather than inside a kernel-function; runtime system of GPU is not open, which brings challenges and limitations to saving & restoring the status of GPU. Furthermore, in addition to GPU, future HPC systems will employ more kinds of heterogeneous architectures to improve power-efficiency, which will bring more challenges to checkpointing.

● Challenges of emerged fail-slow fault

Traditionally, the system faults are classified into fail-stop and fail-continue (i.e. soft error). Current resilience approaches are designed for either or both of them. However, with the scaling-up of HPC systems, there are thousands of millions of processors/components in a system, and researchers have found a new kind of fault, called fail-slow [115], which indicates the abnormal state of slow working of processors/nodes/components. Causes of fail-slow faults may be related to instable connections between components, aging components, etc.

Although the fail-slow faults do not make applications crash or generate incorrect results, they affect system performance and often cause some strange phenomena. E.g., an application runs quickly in most times, but becomes very slow occasionally, which brings troubles not only to the users but also to system management and maintenance. At present, there is not much research on fail-slow, especially on how to detect and locate it.

9.2 Conclusion

Accompanied with the growth in scale and complexity of HPC systems, resilience has become one of the key challenges for large-scale HPC systems. To tackle this problem, various kinds of methods, systems and tools have been proposed. This paper gives a comprehensive survey of software resilience approaches for HPC systems. Firstly, we present a classification of those methods. Then we summarize the advantages and disadvantages of each method, and introduce the most popular resilience approaches as well as their recent progress and works.

Specifically, as HPC systems become more complex and heterogeneous, we also track the latest work that meets new challenges in each section. We believe our work can help researchers retrieve the main work for the resilience of HPC systems and quickly catch up with the recent advances in each classification.

Acknowledgements The research presented in this paper has been supported by the GHFund A (No. ghfund202107010337).

References

1. Dongarra J. Report on the fujitsu fugaku system. University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06, 2020
2. Di Martino C, Kramer W, Kalbarczyk Z, Iyer R. Measuring and understanding extreme-scale application resilience: a field study of 5, 000, 000 HPC application runs. In: Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2015, 25–36
3. Hursey J, Squyres J M, Mattox T I, Lumsdaine A. The design and implementation of checkpoint/restart process fault Tolerance for open MPI. In: Proceedings of 2007 IEEE International Parallel and Distributed Processing Symposium. 2007, 1–8
4. Cappello F, Geist A, Gropp B, Kale L, Kramer B, Snir M. Toward exascale resilience. The International Journal of High Performance Computing Applications, 2009, 23(4): 374–388
5. Egwutuoha I P, Levy D, Selic B, Chen S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. The Journal of Supercomputing, 2013, 65(3): 1302–1326
6. Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J, et al. Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep, 2008, 15: 181
7. Gupta S, Patel T, Engelmann C, Tiwari D. Failures in large scale systems: Long-term measurement, analysis, and implications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2017, 44
8. Radojkovic P, Marazakis M, Carpenter P, Jeyapaul R, Gizopoulos D, Schulz M, Armejach A, Ayguade E A, Bodin F, Canal R, et al. Towards resilient EU HPC systems: A blueprint. PhD thesis, European HPC resilience initiative, 2020
9. Avizienis A, Laprie J C, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 2004, 1(1): 11–33
10. Mukherjee S. Architecture Design for Soft Errors. San Francisco: Morgan Kaufmann, 2008
11. Tan L, DeBardeleben N. Failure analysis and quantification for contemporary and future supercomputers. 2019, arXiv preprint arXiv: 1911.02118
12. Shoji F, Matsui S, Okamoto M, Sueyasu F, Tsukamoto T, Uno A, Yamamoto K. Long term failure analysis of 10 peta-scale supercomputer. In: Proceedings of HPC in Asia Session at ISC 2015. 2015
13. Das A, Mueller F, Siegel C, Vishnu A. Desh: deep learning for system health prediction of lead times to failure in HPC. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing. 2018, 40–51
14. Di Martino C, Kalbarczyk Z, Iyer R K, Baccanico F, Fullop J, Kramer W. Lessons learned from the analysis of system failures at petascale: the case of blue waters. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2014, 610–621
15. El-Sayed N, Schroeder B. Reading between the lines of failure logs: understanding how HPC systems fail. In: Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2013, 1–12
16. Bode B, Butler M, Dunning T, Hoeer T, Kramer W, Gropp W, WenMei H. The blue waters super-system for super-science. In: Contemporary High Performance Computing: From Petascale toward Exascale, 339–366. Chapman and Hall/CRC, 2013
17. Bland B. Titan - Early experience with the titan system at oak ridge national laboratory. In: Proceedings of 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. 2012, 2189–2211
18. Bautista-Gomez L, Gainaru A, Perarnau S, Tiwari D, Gupta S, Engelmann C, Cappello F, Snir M. Reducing waste in extreme scale systems through introspective analysis. In: Proceedings of 2016 IEEE International Parallel and Distributed Processing Symposium. 2016, 212–221
19. Tiwari D, Gupta S, Vazhkudai S S. Lazy checkpointing: exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2014, 25–36
20. Tiwari D, Gupta S, Gallarno G, Rogers J, Maxwell D. Reliability lessons learned from GPU experience with the Titan supercomputer at Oak Ridge leadership computing facility. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2015, 1−12
21. Hargrove P H, Duell J C. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. Journal of Physics: Conference Series, 2006, 46: 494–499
22. Ansel J, Arya K, Cooperman G. DMTCP: Transparent checkpointing for cluster computations and the desktop. In: Proceedings of 2009 IEEE International Symposium on Parallel & Distributed Processing. 2009, 1−12
23. Bautista-Gomez L, Tsuboi S, Komatitsch D, Cappello F, Maruyama N, Matsuoka S. FTI: High performance fault tolerance interface for hybrid systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. 2011, 1−12
24. Zhong H, Nieh J. Crak: Linux checkpoint/restart as a kernel module. Technical Report, Citeseer, 2001
25. Osman S, Subhraveti D, Su G, Nieh J. The design and implementation of zap: a system for migrating computing environments. ACM SIGOPS Operating Systems Review, 2002, 36(S1): 361–376
26. Sankaran S, Squyres J M, Barrett B, Sahay V, Lumsdaine A, Duell J, Hargrove P, Roman E. The LAM/MPI checkpoint/restart framework: system-initiated checkpointing. The International Journal of High Performance Computing Applications, 2005, 19(4): 479–493
27. Wang C, Mueller F, Engelmann C, Scott S L. Hybrid checkpointing for MPI jobs in HPC environments. In: Proceedings of the 16th International Conference on Parallel and Distributed Systems. 2010, 524−533
28. Sancho J C, Petrini F, Johnson G, Frachtenberg E. On the feasibility of incremental checkpointing for scientific computing. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium. 2004, 58
29. Agarwal S, Garg R, Gupta M S, Moreira J E. Adaptive incremental checkpointing for massively parallel systems. In: Proceedings of the 18th Annual International Conference on Supercomputing. 2004, 277−286
30. Bosilca G, Bouteiller A, Cappello F, Djilali S, Fedak G, Germain C, Herault T, Lemarinier P, Lodygensky O, Magniette F, Neri V, Selikhov A. MPICh-V: toward a scalable fault tolerant MPI for volatile nodes. In: Proceedings of 2002 ACM/IEEE Conference on Supercomputing. 2002, 29
31. Bronevetsky G, Marques D, Pingali K, Stodghill P. Automated application-level checkpointing of MPI programs. In: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2003, 84−94

32. Graham R L, Choi S E, Daniel D J, Desai N N, Minnich R G, Rasmussen C E, Risinger L D, Sukalski M W. A network-failure-tolerant message-passing system for terascale clusters. International Journal of Parallel Programming, 2003, 31(4): 285–303
33. Woo N, Choi S, Jung h, Moon J, Yeom H Y, Park T, Park H. MPICH-GF: providing fault tolerance on grid environments. In: Proceedings of the 3rd IEEE//ACM International Symposium on Cluster Computing and the Grid (CCGrid2003), the Poster and Research Demo Session. 2003
34. Zheng G, Shi L, Kale L V. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: Proceedings of 2004 IEEE International Conference on Cluster Computing. 2004, 93−103
35. Zhang Y, Wong D, Zheng W. User-level checkpoint and recovery for LAM/MPI. ACM SIGOPS Operating Systems Review, 2005, 39(3): 72–81
36. Buntinas D, Coti C, Herault T, Lemarinier P, Pilard L, Rezmerita A, Rodriguez E, Cappello F. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols. Future Generation Computer Systems, 2008, 24(1): 73–84
37. Ruscio J F, Heffner M A, Varadarajan S. DejaVu: transparent user-level checkpointing, migration, and recovery for distributed systems. In: Proceedings of 2007 IEEE International Parallel and Distributed Processing Symposium. 2007, 1−10
38. Cao J, Arya K, Garg R, Matott S, Panda D K, Subramoni H, Vienne J, Cooperman G. System-level scalable checkpoint-restart for petascale computing. In: Proceedings of the 22nd International Conference on Parallel and Distributed Systems. 2016, 932−941
39. Garg R, Price G, Cooperman G. MANA for MPI: MPI-agnostic network-agnostic transparent checkpointing. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. 2019, 49−60
40. Laguna I, Richards D F, Gamblin T, Schulz M, De Supinski B R, Mohror K, Pritchard H. Evaluating and extending user-level fault tolerance in MPI applications. The International Journal of High Performance Computing Applications, 2016, 30(3): 305–319
41. Chakraborty S, Laguna I, Emani M, Mohror K, Panda D K, Schulz M, Subramoni H. EREINIT: scalable and efficient fault-tolerance for bulk-synchronous MPI applications. Concurrency and Computation: Practice and Experience, 2020, 32(3): e4863
42. Georgakoudis G, Guo L, Laguna I. Reinit^{++}: evaluating the performance of global-restart recovery methods for MPI fault tolerance. In: Proceedings of the 35th International Conference on High Performance Computing. 2020, 536−554
43. Bronevetsky G, Marques D J, Pingali K K, Rugina R, McKee S A. Compiler-enhanced incremental checkpointing for OpenMP applications. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2008, 275−276
44. Arora R, Bangalore P, Mernik M. A technique for non-invasive application-level checkpointing. The Journal of Supercomputing, 2011, 57(3): 227–255
45. Ba T N, Arora R. A tool for semi-automatic application-level checkpointing. In: Technical Posters at the International Conference for High Performance Computing, Networking, Storage and Analysis. 2016, 16–20
46. Quinlan D, Liao C. The ROSE source-to-source compiler infrastructure. In: Proceedings of the Cetus Users and Compiler Infrastructure Workshop. 2011, 1−3
47. Shahzad F, Thies J, Kreutzer M, Zeiser T, Hager G, Wellein G. CRAFT: a library for easier application-level checkpoint/restart and automatic fault tolerance. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(3): 501–514
48. Takizawa H, Sato K, Komatsu K, Kobayashi H. CheCUDA: a checkpoint/restart tool for CUDA applications. In: Proceedings of 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies. 2009, 408−413
49. Garg R. Extending the domain of transparent checkpoint-restart for large-scale HPC. Northeastern University, Dissertation, 2019
50. Garg R, Mohan A, Sullivan M, Cooperman G. CRUM: checkpoint-restart support for CUDA's unified memory. In: Proceedings of 2018 IEEE International Conference on Cluster Computing. 2018, 302−313
51. Jain T, Cooperman G. CRAC: Checkpoint-restart architecture for CUDA with streams and UVM. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. 2020, 1−15
52. Lee K, Sullivan M B, Hari S K S, Tsai T, Keckler S W, Erez M. GPU snapshot: checkpoint offloading for GPU-dense systems. In: Proceedings of the ACM International Conference on Supercomputing. 2019, 171−183
53. Kannan S, Farooqui N, Gavrilovska A, Schwan K. HeteroCheckpoint: efficient checkpointing for accelerator-based systems. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2014, 738−743
54. Vaidya N H. A case for two-level distributed recovery schemes. In: Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. 1995, 64−73
55. Haines J, Lakamraju V, Koren I, Krishna C M. Application-level fault tolerance as a complement to system-level fault tolerance. The Journal of Supercomputing, 2000, 16(1−2): 53−68
56. Di S, Robert Y, Vivien F, Cappello F. Toward an optimal online checkpoint solution under a two-level HPC checkpoint model. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(1): 244–259
57. Benoit A, Cavelan A, Le Fèvre V, Robert Y, Sun H. Towards optimal multi-level checkpointing. IEEE Transactions on Computers, 2017, 66(7): 1212–1226
58. Ferreira K, Stearley J, Laros J H, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges P G, Arnold D. Evaluating the viability of process replication reliability for exascale systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. 2011, 1−12
59. Wu P, Ding C, Chen L, Gao F, Davies T, Karlsson C, Chen Z. Fault tolerant matrix-matrix multiplication: Correcting soft errors on-line. In: Proceedings of the 2nd Workshop on Scalable Algorithms for Large-Scale Systems. 2011, 25−28
60. Fiala D, Mueller F, Engelmann C, Riesen R, Ferreira K, Brightwell R. 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. 2012, 1−12
61. Wang Z, Yang X, Zhou Y. MMPI: a scalable fault tolerance mechanism for MPI large scale parallel computing. In: Proceedings of the 10th IEEE International Conference on Computer and Information Technology. 2010, 1251−1256
62. Hussain Z, Znati T, Melhem R. Partial redundancy in HPC systems with non-uniform node reliabilities. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. 2018, 566−576
63. Elliott J, Kharbas K, Fiala D, Mueller F, Ferreira K, Engelmann C. Combining partial redundancy and checkpointing for HPC. In: Proceedings of the 32nd International Conference on Distributed Computing Systems. 2012, 615−626
64. George C, Vadhiyar S. Fault tolerance on large scale systems using

adaptive process replication. IEEE Transactions on Computers, 2015, 64(8): 2213–2225

65. Quinn H, Graham P. Terrestrial-based radiation upsets: a cautionary tale. In: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. 2005, 193−202
66. Schroeder B, Pinheiro E, Weber W D. DRAM errors in the wild: a large-scale field study. Communications of the ACM, 2011, 54(2): 100–107
67. Sedaghat Y, Miremadi S G, Fazeli M. A software-based error detection technique using encoded signatures. In: Proceedings of the 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems. 2006, 389−400
68. Miremadi G, Harlsson J, Gunneflo U, Torin J. Two software techniques for on-line error detection. In: Proceedings of the 22nd International Symposium on Fault-Tolerant Computing. 1992, 328−335
69. Vemu R, Abraham J. CEDA: control-flow error detection using assertions. IEEE Transactions on Computers, 2011, 60(9): 1233–1245
70. Zarandi H R, Maghsoudloo M, Khoshavi N. Two efficient software techniques to detect and correct control-flow errors. In: Proceedings of the 16th Pacific Rim International Symposium on Dependable Computing. 2010, 141−148
71. Gomez L B, Cappello F. Detecting silent data corruption through data dynamic monitoring for scientific applications. ACM SIGPLAN Notices, 2014, 49(8): 381–382
72. Berrocal E, Bautista-Gomez L, Di S, Lan Z, Cappello F. Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. 2015, 275−278
73. LeBlanc T, Anand R, Gabriel E, Subhlok J. VolpexMPI: an MPI library for execution of parallel applications on volatile nodes. In: Proceedings of the 16th European Parallel Virtual Machine / Message Passing Interface Users' Group Meeting. 2009, 124−133
74. Engelmann C, Boehm S. Redundant execution of HPC applications with MR-MPI. In: Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks. 2011, 31−38
75. Berrocal E, Bautista-Gomez L, Di S, Lan Z, Cappello F. Toward general software level silent data corruption detection for parallel applications. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(12): 3642–3655
76. Fiala D, Ferreira K B, Mueller F, Engelmann C. A tunable, software-based DRAM error detection and correction library for HPC. In: Proceedings of European Conference on Parallel Processing. 2012, 251−261
77. Fiala D, Mueller F, Ferreira K B. FlipSphere: a software-based DRAM error detection and correction library for HPC. In: Proceedings of the 20th International Symposium on Distributed Simulation and Real Time Applications. 2016, 19−28
78. Fiala D, Mueller F, Ferreira K, Engelmann C. Mini-Ckpts: surviving OS failures in persistent memory. In: Proceedings of 2016 International Conference on Supercomputing. 2016, 7
79. Huang K H, Abraham J A. Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers, 1984, C-33(6): 518–528
80. Luk F T, Park H. Fault-tolerant matrix triangularizations on systolic arrays. IEEE Transactions on Computers, 1988, 37(11): 1434–1438
81. Luk F T, Park H. An analysis of algorithm-based fault tolerance techniques. Journal of Parallel and Distributed Computing, 1988, 5(2): 172–184
82. Bouteiller A, Herault T, Bosilca G, Du P, Dongarra J. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. ACM Transactions on Parallel Computing, 2015, 1(2): 10
83. Chen Z. Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. ACM SIGPLAN Notices, 2013, 48(8): 167–176
84. Tao D, Song S L, Krishnamoorthy S, Wu P, Liang X, Zhang E Z, Kerbyson D, Chen Z. New-sum: a novel online ABFT scheme for general iterative methods. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing. 2016, 43−55
85. Schöll A, Braun C, Kochte M A, Wunderlich H J. Efficient algorithm-based fault tolerance for sparse matrix operations. In: Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2016, 251−262
86. Shantharam M, Srinivasmurthy S, Raghavan P. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In: Proceedings of the 26th ACM International Conference on Supercomputing. 2012, 69−78
87. Zhu Y, Liu Y, Li M, Qian D. Block-checksum-based fault tolerance for matrix multiplication on large-scale parallel systems. In: Proceedings of the 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems. 2018, 172−179
88. Zhu Y, Liu Y, Zhang G. FT-PBLAS: PBLAS-based fault-tolerant linear algebra computation on high-performance computing systems. IEEE Access, 2020, 8: 42674–42688
89. Chen Z, Dongarra J. Algorithm-based fault tolerance for fail-stop failures. IEEE Transactions on Parallel and Distributed Systems, 2008, 19(12): 1628–1641
90. Roche T, Cunche M, Roch J L. Algorithm-based fault tolerance applied to P2P computing networks. In: Proceedings of the 1st International Conference on Advances in P2P Systems. 2009, 144−149
91. Hakkarinen D, Wu P, Chen Z. Fail-stop failure algorithm-based fault tolerance for Cholesky decomposition. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(5): 1323–1335
92. Davies T, Karlsson C, Liu H, Ding C, Chen Z. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In: Proceedings of the International Conference on Supercomputing. 2011, 162−171
93. Chen J, Li S, Chen Z. GPU-ABFT: optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs. In: Proceedings of 2016 IEEE International Conference on Networking, Architecture and Storage. 2016, 1−2
94. Chen J, Li H, Li S, Liang X, Wu P, Tao D, Ouyang K, Liu Y, Zhao K, Guan Q, Chen Z. Fault tolerant one-sided matrix decompositions on heterogeneous systems with GPUs. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. 2018, 854−865
95. Braun C, Halder S, Wunderlich H J. A-ABFT: autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units. In: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2014, 443−454
96. Ranganathan S, George A D, Todd R W, Chidester M C. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. Cluster Computing, 2001, 4(3): 197–209
97. Gabel M, Schuster A, Bachrach R G, Bjørner N. Latent fault detection in large scale services. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks. 2012, 1−12
98. Wu L, Luo H, Zhan J, Meng D. A runtime fault detection method for HPC cluster. In: Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies. 2011, 68−72
99. Ghiasvand S, Ciorba F M. Anomaly detection in high performance

computers: a vicinity perspective. In: Proceedings of the 18th International Symposium on Parallel and Distributed Computing. 2019, 112−120

100. Egwutuoha I P, Chen S, Levy D, Selic B, Calvo R. Cost-oriented proactive fault tolerance approach to high performance computing (HPC) in the cloud. International Journal of Parallel, Emergent and Distributed Systems, 2014, 29(4): 363–378
101. Borghesi A, Libri A, Benini L, Bartolini A. Online anomaly detection in HPC systems. In: Proceedings of 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems. 2019, 229−233
102. Borghesi A, Molan M, Milano M, Bartolini A. Anomaly detection and anticipation in high performance computing systems. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(4): 739–750
103. Dani M C, Doreau H, Alt S. K-means application for anomaly detection and log classification in HPC. In: Proceedings of the 30th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems. 2017, 201−210
104. Zhu B, Wang G, Liu X, Hu D, Lin S, Ma J. Proactive drive failure prediction for large scale storage systems. In: Proceedings of the 29th Symposium on Mass Storage Systems and Technologies. 2013, 1−5
105. Fulp E W, Fink G A, Haack J N. Predicting computer system failures using support vector machines. In: Proceedings of the 1st USENIX Conference on Analysis of System Logs. 2008, 5
106. Ganguly S, Consul A, Khan A, Bussone B, Richards J, Miguel A. A practical approach to hard disk failure prediction in cloud platforms: big data model for failure management in datacenters. In: Proceedings of the 2nd International Conference on Big Data Computing Service and Applications. 2016, 105−116
107. Krammer B, Bidmon K, Müller M S, Resch M M. MARMOT: an MPI analysis and checking tool. Advances in Parallel Computing, 2004, 13: 493–500
108. Vetter J S, De Supinski B R. Dynamic software testing of MPI applications with Umpire. In: Proceedings of 2000 ACM/IEEE Conference on Supercomputing. 2000, 51
109. Gao J, Yu K, Qing P. A scalable runtime fault detection mechanism for high performance computing. In: Proceedings of the 2nd Information Technology, Networking, Electronic and Automation Control Conference. 2017, 490−495
110. Kharbas K, Kim D, Hoefler T, Mueller F. Assessing HPC failure detectors for MPI jobs. In: Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. 2012, 81−88
111. Liang Y, Zhang Y, Sivasubramaniam A, Jette M, Sahoo R. BlueGene/L failure analysis and prediction models. In: Proceedings of the International Conference on Dependable Systems and Networks. 2006, 425−434
112. Gainaru A, Cappello F, Snir M, Kramer W. Fault prediction under the microscope: a closer look into HPC systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 2012, 1−11
113. Gainaru A, Cappello F, Kramer W. Taming of the shrew: modeling the normal and faulty behaviour of large-scale HPC systems. In: Proceedings of the 26th International Parallel and Distributed Processing Symposium. 2012, 1168−1179
114. Pelaez A, Quiroz A, Browne J C, Chuah E, Parashar M. Online failure prediction for HPC resources using decentralized clustering. In: Proceedings of the 21st International Conference on High Performance Computing. 2014, 1−9
115. Gunawi H S, Suminto R O, Sears R, Golliher C, Sundararaman S, et al. Fail-slow at scale: evidence of hardware performance faults in large production systems. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies. 2018, 1−14

Jie Jia is a PhD candidate in School of Computer Science and Engineering, Beihang University, China. She is currently working on the fault tolerance of large-scale parallel applications. Her research interests include high performance computing, checkpointing, distributed and parallel computing.

Yi Liu is a professor in School of Computer Science and Engineering, and Director of the Sino-German Joint Software Institute (JSI) at Beihang University, China. In 2000, he completed PhD in Department of Computer Science of Xi'an Jiaotong University, China. His research interests include computer architecture, HPC and new generation of network technology.

Guozhen Zhang received his PhD from the School of Computer Science and Engineering, Beihang University, China. He is currently working on program debugging and fault tolerance of large-scale parallel applications. His research interests include HPC, computer architecture, distributed and parallel computing.

Yulin Gao received his master degree from the School of Computer Science and Engineering, Beihang University, China. His research interests include HPC, fault tolerance.

Depei Qian is a professor at the School of Computer Science and Engineering, Beihang University, China. He received his master degree from University of North Texas, USA in 1984. He is an academician of Chinese Academy of Sciences and a fellow of China Computer Federation. His research interests include innovative technologies in distributed computing, high performance computing, and computer architecture.