



# Failure detection algorithm for Fail-Lagging model applied to HPC

Yingjun Ye<sup>1</sup> · Yongdong Zhang<sup>1</sup> · Weicai Ye<sup>1</sup>

Accepted: 30 January 2022 / Published online: 27 March 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

## Abstract

It is essential to use fault tolerance techniques on exascale high-performance computing systems, but this faces many challenges such as higher probability of failure, more complex types of faults, and greater difficulty in failure detection. In this paper, we designed the Fail-Lagging model to describe HPC process-level failure. The failure model does not distinguish whether the failed process is crashed or slow, but is compatible with the possible behavior of the process due to various failures, such as crash, slow, recovery. The failure detection in Fail-Lagging model is implemented by local detection and global decision among processes, which depend on a robust and efficient communication topology. Robust means that failed processes do not easily corrupt the connectivity of the topology, and efficient means that the time complexity of the topology used for collective communication is as low as possible. For this purpose, we designed a torus-tree topology for failure detection, which is scalable even at the scale of an extremely large number of processes. The Fail-Lagging model supports common fault tolerance methods such as rollback, replication, redundancy, algorithm-based fault tolerance, etc. and is especially able to better enable the efficient forward recovery mode. We demonstrate with large-scale experiments that the torus-tree failure detection algorithm is robust and efficient, and we apply fault tolerance based on the Fail-Lagging model to iterative computation, enabling applications to react to faults in a timely manner.

**Keywords** Failure model · Fault tolerance · Failure detection · HPC

---

✉ Yongdong Zhang  
Lnszyd@mail.sysu.edu.cn

<sup>1</sup> Guangdong Province Key Laboratory of Computational Science, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, Guangdong, China

## 1 Introduction

The application scenario of large-scale supercomputing is becoming more and more extensive. With the development of technology, the computing power of supercomputers is constantly increasing, and the scale has been able to reach tens of millions of cores. Scaling up the system increases computing power, but also shortens the Mean Time Between Failure (MTBF), which means that the system will have a higher probability of encountering faults. Without fault tolerance mechanisms, applications can waste a lot of time in the event of failure, causing unbearable financial losses to users and providers of computing services.

Current high-performance computing (HPC) platforms mostly use distributed storage and message passing in parallel. If the HPC application is unable to identify abnormally slow or crashed processes in time during operation, once the processes need to communicate with each other, the normal processes will continuously wait for the response from the abnormal processes, resulting in the task progress not being able to advance further. If the application is large, with millions of computer cores, every second wasted can cost a lot of money. Failure detection is essential for fault-tolerance mechanisms, allowing applications to detect failure in time and take recovery measures. However, it is very difficult to implement failure detection on large-scale HPC systems, and the difficulties include:

- The large number of processes makes detection very costly and difficult. Some detection methods used on small-scale distributed systems are not scalable to cases where the number of processes is large, for example, using a master process or a manager to monitor the activity of each process in real time; using an all-to-all approach to connect processes that each process monitors each other by receiving and sending heartbeats. These methods require tremendous cost, and the latency resulting from intensive communication and managing a large amount of processes may cause some errors in the detection results.
- It is impossible to distinguish whether the failed process is crashed or very slow with the asynchronous assumption, and it can only suspect but not conclude the failure of the process. In HPC, researchers strengthen asynchronization to partial synchronization [7, 10, 18] so as to be able to detect failure using a timeout-based approach, even so, the setting of the timeout value remains a tricky issue. Too short timeout settings are prone to misclassification; too long timeout settings degrade efficiency.
- The variety of faults in HPC systems makes it difficult to find a suitable model to describe them. Fail-Stop [21] is the failure model used by most HPC fault-tolerant techniques, in which processes either work or stop. The model is simple and intuitive, but is not sufficient to cover the multifarious types of faults in HPC. The Byzantine [19] can cope with various types of faults, but implementing Byzantine fault tolerance requires high redundancy costs. For this reason, researchers are working to explore a more suitable failure model, seeking a balance between the effectiveness and cost of Fail-Stop and Byzantine models.

- There may be hard-to-detect faults or “false negative” processes. Such faults can cause applications to run much longer than anticipated, with multiple runs struggling to complete a valid calculation. Some famous supercomputers such as Sunway and Tianhe-2 users have encountered this type of error [24]. The users are more likely to encounter such errors and waste a lot of money when calling computing resources with millions of cores or more.
- Failed processes can break the connectivity of the original communication structure. Results of fault detection require fault propagation with the help of communication topologies. The topologies employed by HPC can be reconnected to maintain connectivity. However, as the number of processes increases, the number of failed processes as a proportion of the total number of processes does not decrease, so that maintaining topology connectivity will be more expensive and challenging.

The implementation of failure detection on HPC should be as accurate, efficient, low cost, scalable to exascale, and avoid “false negative” detection results, which cannot be fully satisfied by existing methods, which motivates us to propose a new failure model to explore a more suitable solution for exascale HPC. We focus on the running state of processes, and the stable performance of HPC applications depends on the proper execution of each process. We are concerned about the running state of the processes, because the stable performance of HPC applications depends on the proper execution of each process. Process failure can be due to a variety of errors. Failed processes may stop running, run slowly, fail to communicate properly, resume from an unresponsive state, etc. Static data errors can be corrected by checksum without affecting the speed of the process, which is out of the scope of this paper.

The response speed of abnormal processes will be significantly slower than other normal processes, or even no response, and the failed processes will affect the progress of other processes during the communication phase, thus causing blocking. The application needs to find and isolate the abnormal processes in time to avoid being dragged down. To this end, we design the Fail-Lagging model where once the processes appear to be severely slower than others, regardless of crash or slow, they are judged as “lagging” by others. Normal processes will skip all “lagging” processes, take appropriate fault-tolerant measures and continue to work.

The Fail-Lagging model provides a novel solution idea for failure detection and fault-tolerant recovery for HPC. Failure detection no longer uses heartbeats and uses an idea similar to loose synchronization, replacing periodic communication with the overhead of collective communication. Loose synchronization [2] means that processes do not have to wait for all processes to arrive before they can pass the barrier. Processes that have arrived can pass the barrier if they meet the release conditions and no longer wait for the late processes, at which point the unreached processes are determined to be failed processes by the other processes. In this way, non-arrived processes, regardless of the crash, performance failure or other reasons, are uniformly treated as “Lagging” processes, simplifying the definition of faults to the greatest extent possible and adapting to the complicated types of faults in HPC systems.

Fail-Lagging model blurs process crashes and slowness and does not require a perfect failure detector, only a strong failure detector, as long as all the correct processes are ensured to be able to perform appropriately. Some processes may recover from failure, but are already considered as failed processes because of their slow operation. For those processes that are still able to run, they are also allowed to be retrieved and then re-engage in the computation when they resume running when certain conditions are met.

The detection algorithm is implemented in a torus-tree topology, where processes detect adjacent processes according to the torus-tree topology, count the number of arrived processes, and the arrived processes collectively decide whether the late processes are lagging or not. The torus-tree topology does not require a master to manage all processes, and the connectivity of the topology can be maintained if any process fails.

The detection algorithm is implemented in the torus-tree topology, where processes detect adjacent processes according to the torus-tree, count the number of arrived processes, and the arrived processes collectively decide whether the late processes are lagging or not. The torus-tree topology does not require a master to manage all processes, and the connectivity of the topology can be maintained if any process fails. As the scale of processes increases, the probability and proportion of failed processes per unit time increases, but the proportion of failures that the torus-tree can withstand does not decrease, and the time complexity grows logarithmically, two features that ensure the scalability of the communication topology.

The processes take fault-tolerant recovery measures based on the results of failure detection. When the number of processes is large, it is expensive to back up data, save the running state of processes, restart, and other methods. The approach of disk rollback is difficult to be scaled to the case of large number of processes and short MTBF. In general, faults of HPC are spatially localized [15], i.e., they are insignificant in proportion but concentrated in distribution. If the failed processes can be identified and isolated at the application level, allowing normal processes to continue running, the frequency of application restarts can be reduced and unnecessary storage costs can be minimized. To achieve such a fault-tolerant pattern, this paper initially conceptualizes a scheme of data intersection storage, fault-tolerant collective communication, and process task substitution to support the requirements of application running across failure, and initially applies it to iterative computing.

## 2 Background and motivation

### 2.1 Failure model

Failure model is critical to the design of fault tolerance mechanisms. At present, most fault-tolerant methods are established based on the Fail-Stop [21]. However, researchers are aware of the limitations of the fail-stop model; failure can be permanent, transient, and unstable. It is clearly not the best solution to terminate the execution of the application directly [11]. The Byzantine model [19] can cope with any type of failure and is suitable for redundancy and replication fault-tolerant solutions,

but it is expensive and cannot be achieved with more redundancy in extreme scales. Some of the failure models under exploration try to provide an intermediate model between the two extreme models of Byzantine and Fail-Stop. For example, the Fail-Stutter [5] and Fail-Slow [14] that take into account performance failures. In the distributed field, there is a Crash-Recovery [16] that considers the possibility of recovery after processes crash. We wish to explore a simple and adaptable failure model for HPC that does not make fault tolerance too difficult and can accommodate more complex failure scenarios. We will explore a simple and adaptable fault model for HPC that does not make failure detection and fault tolerance excessively challenging, while in a position to adapt to complex HPC scenarios.

## 2.2 Failure detection

In the distributed field, Chandra and Toueg proposed the first unreliable failure detector [8]. Many fault tolerance methods rely on the powerful characteristics of the perfect failure detector (see, e.g., [22]). Using a non-all-to-all heartbeat detection method, it is difficult to complete fault propagation effectively if the coverage topology is corrupted. For this reason we want to design a failure detection algorithm that can repair the communication topology in case of encountering a failure. The detector satisfies at least the eventual strong completeness.

George Bosilca et al. [25]. proposed a failure detector based on the ring network overlay, where the processes can reconnect to repair the ring in case of failure and complete fault propagation via the hypercube topology [13], which provides a remarkably enlightening solution for the implementation of communication-based failure detection. The results of the detection have to be propagated to each process. Although hypercubes have high communication efficiency, the number of failed processes that a hypercube consisting of  $n$  processes can tolerate in a single detection is  $\log n$  [6]. As the scale of the process increases, the percentage of faults that can be tolerated tends to zero due to  $\lim_{n \rightarrow \infty} \frac{\log n}{n} = 0$ , which is a significant limitation at exascale and larger. To find a suitable topology, we designed the torus-tree [23], which is easy to repair, robust, and has high communication efficiency.

## 2.3 Fault tolerance

The main common fault tolerance techniques for HPC are checkpointing/restart [11], replication and redundancy [12], and algorithm-based fault tolerance [9]. Each of these fault-tolerance techniques has its own advantages, and current research on HPC is increasingly tending to use a combination of technologies to achieve fault tolerance. However, the current version of MPI does not support continuous delivery in case of failure yet, hindering the availability of many technologies for direct use. For this reason, User-Level Fault Mitigation (ULFM) [20] is proposing an extension to MPI that would allow control to be handed over to the user when encountering faults, rather than immediately terminating the application, thus enabling application to run across faults. Future expansion to larger applications will require more

cross-fault capability to break through the bottleneck of increasing failure frequency as scale increases.

## 2.4 Motivation

This paper presents a natural idea of fault tolerance from the perspective of an application:

1. The application performs failure detection at the appropriate time.
2. In case of failure, when the number of processes reaches an user-defined percentage during the detection phase, the processes that have not arrived are temporarily identified as “failed” processes.
3. Normal processes will skip the failed process and continue to execute the task.
4. The input data required for fault-tolerant recovery need to be saved in advance.
5. The failed process may recover and can be retrieved and reused in subsequent calculations.

This idea of fault tolerance is best suited for Bulk Synchronous Parallelism (BSP), where failure detection, fault-tolerant recovery, data backup, and other related operations are done in one super step. The idea of failure detection algorithm is somewhat similar to loose synchronization [2], a concept that was earlier proposed in applications such as distributed computing and network systems. The main reason why it has not been used on HPC systems is that the previously implemented approach requires a manager to accomplish the detection task and ensure consistency among nodes, which can be an extremely high cost for HPC systems.

In order to implement the above fault-tolerant idea, this paper designs Fail-Lagging model, implements failure detection using the definition of failure by the model, implements fault-tolerant recovery using the results of failure detection, and explores a fault-tolerant model that can empower applications to run across failures on large-scale systems.

## 3 Fail-Lagging model

The failure model proposed in this paper is applicable to HPC platforms. The timeout mechanism can be used to determine if a process fails, but the asynchronous model needs to be strengthened with the assumption of partial synchronization, i.e., the assumption of “unbounded message latency” is strengthened with the assumption of “unknown upper time limit for computation and network.” The communication speed of HPC system components tends to be quite fast and should not normally experience too long message delays. The assumption of partial synchronization for HPC is reasonable. Processes cannot use the global clock to determine timeout. They can only infer the state of the process based on the local clock and communication. Processes follow the rules of MPI, each process is able to communicate with

any process, and the communication topology is used to specify the communication routes of the processes at the time of collective communication.

The Fail-Lagging model is used to deal with non-fatal faults, that is, various possible causes such as memory leaks, message omissions, performance failure, etc. that cause processes to crash, deadlock, slow down, failure to communicate, etc. Applications that fail to detect these faults in time can easily suffer from a run time that is far longer than expected or even fails to produce results. The application has to be able to identify processes that slow down the overall running schedule in a timely and efficient manner. Our model describes the definition of failed processes, detection methods and fault tolerance schemes.

### 3.1 Definition of "Lagging" process

Failure definition and detection are important parts of the failure model. Fail-Lagging uses failure detectors with epoch numbers, that is similar to the Crash-Recovery model [1], since it takes into account the possible recovery of the failed processes. For the sake of description, we define as follows:

The set  $P = \{p_1, \dots, p_n\}$  consists of  $n$  processes, and the undirected graph  $G(V, E)$  represents the communication topology used by the set of processes  $P$  in the detection phase,  $V$  is the set of points corresponding to the bijection of the set of processes  $P$ , and  $E$  is the set of edges indicating that two processes on this communication topology are able to directly communicate with each other communication.

Since the global clock is not available, processes obtain their respective local time, and usually take time fragments as the basis for timing, e.g., start timing after entering the detection phase. The time progress can be described by the macroscopic time  $t$ , and the set of local time read by the processes individually is denoted as  $D(t) = \{d_1(t), \dots, d_n(t)\}$ .  $D(t)$  has the following properties:

- $t = \max_i(D(t))$ . That is,  $t$  starts timing with the first process that enters the current detection phase.
- $d(t + \Delta t) = d(t) + \Delta t$

In the detection of epoch  $k$ , using the set  $A(k, t) = \{a_1(k, t), \dots, a_n(k, t)\}$  denotes the actual arrival state of the process at time  $t$ ,  $Arr_i(k, t)$  denotes that process  $p_i$  has arrived at time  $t$ ,  $\neg Arr_i(k, t)$  then denotes that process  $p_i$  has not arrived at time  $t$ . For the set  $A(k, t)$  we have:

$$a_i(k, t) = I_{Arr_i(k, t)} = \begin{cases} 1 & Arr_i(k, t) \\ 0 & \neg Arr_i(k, t) \end{cases} \quad (1)$$

Similarly,  $A'(k, t) = \{a'_1(k, t), \dots, a'_n(k, t)\}$  denotes the arrival status of the process being detected.

Lagging processes are identified based on the fact that when a sufficient number of processes have reached the detection phase and all have waited long enough, if there are still processes that have not arrived, the processes that have arrived determine the processes that have not arrived as "Lagging" processes.

The determination of failure is implemented with the help of local timing and global counting among processes that have arrived. In order to explain the conditions in detail, the following parameters are listed:

- $R_{\text{release}}$  is the minimum release ratio, assuming that there are  $M$  processes participating in this detection. The number of processes that have arrived at the detection phase must be at least  $R_{\text{release}} \times M$  before they can be released.
- $T_{\text{release}}$  is the minimum release time, the number of arrived processes exceeds  $R_{\text{release}} \times M_k$ , but is less than  $M_k$ . If all arrived processes are waiting for more than  $T_{\text{release}}$ , they can pass the detection phase. That is, in the detection epoch  $e$ , if  $\neg \text{Arr}_i(k, t) \neq \emptyset$ , condition *for*  $\forall p_i \in \text{Arr}_i(k, t), d_i(k, t) > T_{\text{release}}$  must be met to determine failure of processes.
- $T_{\text{timeout}}$ . If execution exceeds  $T_{\text{timeout}}$  during the detection phase, the detection is time-out. The program exits the failure detection and reports an error.  $T_{\text{timeout}}$  is used to avoid some situations that the detector cannot handle that lead to exceeding the expected detection time. Generally  $T_{\text{timeout}}$  will be much longer than the time necessary for the collective communication to complete. In order to ensure that the program can complete the detection as much as possible, the  $T_{\text{timeout}}$  can also be extended appropriately using dynamic adaptive methods.
- $T_{\text{recon}}$  is used for reconnection. The process looks for the next valid process to connect when the neighboring processes do not respond for more than  $T_{\text{recon}}$  time. In fact, this parameter is related to the connection policy and some topologies with a higher degree can do without this parameter.

The result of one detection is first defined from a global perspective. For ease of illustration, it is first assumed that the processes arriving at the detection phase can form a connectivity graph. It is guaranteed that each arriving process can achieve collective communication to obtain arrival information of other processes. A simple proof can be made according to 1.

**Theorem 1** *The processes on the connectivity graph are able to get the set of all processes on the connectivity graph.*

**Proof** The set of processes  $G$  is connected and for  $\forall p \in G, \forall p \neq q \in G$ ,  $p$  and  $q$  have communication paths and  $p$  can obtain information about  $q$ . Therefore,  $\forall p \in G$  can obtain the set of processes  $G - \{p\}$  through communication, so  $\forall p \in G$  can hold the set of processes  $G$ .  $\square$

Arrived processes have to maintain the topology connectivity by means of communication, which depends on the initial topology and the reconnection strategy. How to maintain topological connectivity is an important scientific issue that will be discussed in the next section. For now, it is assumed that processes can obtain global information based on the connectivity graph. The detection results in the following three cases:



- In the detection of one epoch, the total number of processes that participate in the detection is  $M_k$ . All processes arrive and all processes pass the detection directly, which can be expressed as:

$$\text{Success } (D(k, t), A(k, t)) \Leftarrow \sum_{i=1}^{M_k} I_{(d_i(k,t) < T_{\text{timeout}})} \cdot a_i(k, t) = M_k \tag{2}$$

$\text{Success}(D(k, t), A(k, t))$  indicates that the result is that all processes arrive and pass. The right side of the arrow indicates the condition that the detection result is fault-free.

- If there are non-arrived processes, but the number of late processes is low enough to satisfy the release condition, i.e., there are Lagging processes as a result of the failure detection, this can be expressed as:

$$\begin{aligned} \text{Pass } (D(k, t), A(k, t)) \Leftarrow & \left( \sum_{i=1}^{M_k} I_{(d_i(k,t) \geq T_{\text{release}})} = \sum_{i=1}^{M_k} a_i(k, t) \right) \\ & \wedge \left( \sum_{i=1}^{M_k} I_{(d_i(k,t) < T_{\text{timeout}})} \cdot a_i(k, t) \geq R_{\text{release}} \cdot M_k \right) \end{aligned} \tag{3}$$

In this case, the arrived processes have to perform collective communication to obtain the complete detection result, i.e., the Lagging processes set  $L = \{p_j | p_j \in P, a'(j) = 0\}$

- Detection timeout, that is, the processes fail to complete the detection within  $T_{\text{timeout}}$ . Processes will exit the detector one after another because of timeout, which is:

$$\text{Timeout } (D(k, t), A(k, t)) \Leftarrow \sum_{i=1}^{M_k} I_{(d_i(k,t) \geq T_{\text{timeout}})} \cdot a_i(k, t) > (1 - R_{\text{release}}) \cdot M_k \tag{4}$$

The detector will time out once condition 4 is satisfied. This is because if only a small number of processes time out. These processes are determined to be Lagging processes. When the number of arrivals is larger than RM, the detection does not time out, although a few “false positives” are generated. However, once the number of non-arrived processes is too large and exceeds the maximum tolerated by the detector. Arrived processes never satisfy the release condition 3.

### 3.2 Detecting “Lagging” processes

A connectivity graph is required for processes to obtain global information as a communication topology. To count the number of arrived processes in the absence of a manager, it is necessary to complete collective communication among the arrived processes, which is equivalent to performing All-Reduce among the arrived processes. Failed processes break the original topology. Normal processes have to maintain the connectivity of the topology. The processes have to confirm the arrival

of the neighboring processes in the communication graph to establish connection with them. If an adjacent process does not arrive, it needs to find a new process to establish a connection, which ensures that the arrived processes can form a connectivity graph.

Each process enters the detection phase and performs the following actions, which are described in Fig. 1 and as follows:

- \* Process establishes connection with adjacent processes based on the current communication topology. If an adjacent process response time out and this location needs to be reconnected, the process will find a valid process to replace it. It will continuously wait for responses from unresponsive processes, so use non-blocking communication.
- \* Processes perform counting while establishing connection. The counting operation is equivalent to a multi-gather operation that counts the number of currently arrived processes according to the connectivity graph composed by the currently arrived processes.
- \* During the detection phase, the process needs to complete two operations, process connection and process count, to obtain local detection information and global information, respectively. Task T1 builds the connectivity graph, and task T2 in fact not only counts the number of arrived processes, but also accumulates the value of  $I_{(d_i(k,t) \geq T_{release})}$ , which can be done in the same collec-

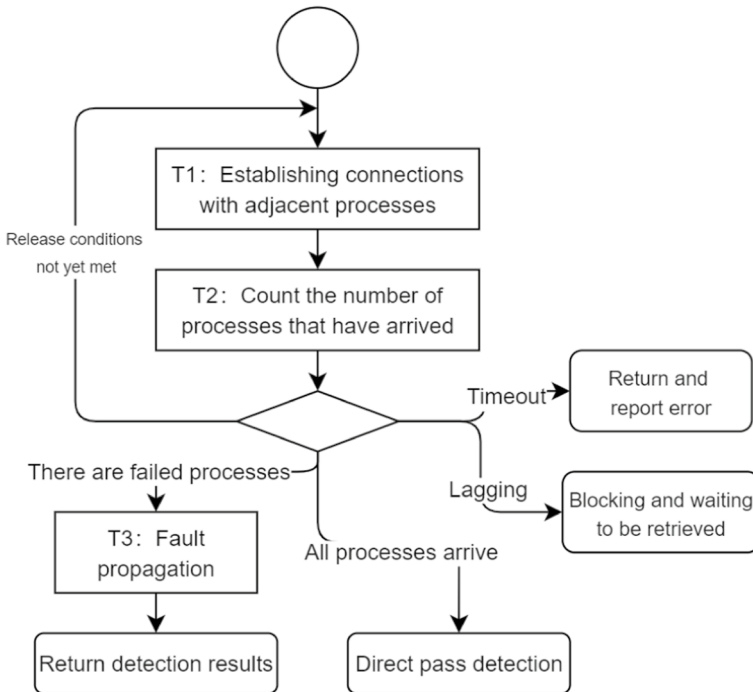


Fig. 1 Failure detection flow chart

tive communication. In this way, after the normal process completes tasks T1 and T2, the result obtained is  $\sum_{i=1}^{M_k} a'_i(k, t) \leq \sum_{i=1}^{M_k} a_i(k, t)$ . According to the results process do that:

- The release condition is not met, recount and continue to wait for other processes to arrive.
- As long as all processes arrive, that is  $\sum_{i=1}^{M_k} a'_i(k, t) = M_k$ , the process can pass the failure detection directly.
- If there are failed processes and  $\sum_{i=1}^m a'_i(t) = \sum_{i=1}^m I_{(d_i(t) \geq T_{\text{release}})}$ , the processes can pass the detection, but they have to perform an additional collective communication, i.e., task T3, to multi-gather the set of Lagging processes.

\* If the process is notified by another process that it is timed out during the connection, the process actively goes into blocking and waits for being retrieved.  
 \* If the time spent of the detection phase exceeds  $T_{\text{timeout}}$ , the detector returns and reports errors.

At this point for process  $p_j$ , if  $(a'_j(k, t) = 0 \wedge a_j(k, t) = 1) = 1$ , an isolated point is created. This is one of the manifestations of false positives, and another is that the process has a performance failure and fails to arrive on time in this epoch. The result of a false positive does not affect the judgment of other processes, and other processes that are on the connectivity graph can still pass the failure detection after meeting the release conditions. Even if a false positive result is generated, the detection result is still valid if the release conditions are met.

**Theorem 2** *As long as  $\sum_{i=1}^m a'_i(t) = \sum_{i=1}^m I_{(d_i(t) \geq T_{\text{release}})}$  is satisfied, condition 3  $Pass(D(k, t), A(k, t))$  can eventually be satisfied even if there are isolated points.*

**Proof** The processes get the Lagging process set after T3 and will send message notifications to the processes belonging to set  $L = \{p_j | p_j \in P, a'(j) = 0\}$ . The messages will be stored in the message cache to which the process belongs.

In this way,  $\forall p \in L$  that encounters  $(a'(k, t) = 0 \wedge a(k, t) = 1) = 1$  will enter blocking during this detection. It also means that the normal processes on the connectivity graph judge the processes that become isolated points as “Lagging” processes based on the current decision and let them change their state into blocking (processes in the MPI communication domain can communicate directly, and the communication topology used for failure detection only constrains the communication rules). Thus, we have:  $\sum_{i=1}^{M_k} a'_i(k, t) = \sum_{i=1}^{M_k} I_{(d_i(k,t) \geq T_{\text{release}})} \Rightarrow Pass(D(k, t), A(k, t))$  □

### 3.3 Handling “Lagging” processes

Each process that completes failure detection holds the set of late processes, and the late processes are identified as “Lagging.” The treatment of failed processes follows:

- The normal process no longer actively communicates with the Lagging process, but can probe messages from the Lagging process. This ensures that the “Lagging” process does not interfere with the normal process, while the “Lagging” process is able to be informed of its own status even though and go into blocking waiting for other processes to retrieve it.
- The missing data and tasks of the “Lagging” process are replaced by the normal process. There are two ways to implement this, either by redundancy or by the processes involved in the computation additionally taking on the tasks of the “Lagging” process.
- The “Lagging” process, due to its lagging schedule, has already been replaced by other processes even if it resumes operation. Therefore, it cannot join the calculation directly after resuming, but needs to wait for other processes to retrieve and reuse it.
- The processes that pass the detector need to exclude all “Lagging” processes and reconstruct the topology for operations such as collective communication.

If the number of failed processes is small and the application can execute across failure, it directly assigns the tasks of the “Lagging” process to other normal processes. If the number of failed processes is too high, or if all data backups of a process are lost and cannot be recovered, the application can exit in time to report to the user and reduce losses.

### 3.4 Retrieving recovery processes

Lagging processes may recover from unstable states, or they may become isolated for various reasons such as message omissions, network failure, and so on. If no measures are taken to retrieve these lagging processes, they may become less and less available over a long period of time. Even with the data backup algorithm, the additional tasks to be performed by the available processes increase dramatically, raising the cost of fault tolerance and possibly leading to new errors. Therefore, the Fail-Lagging model allows reusing the recovered Lagging processes.

Processes belonging to the lagging process set behave in two ways: (1) They are unresponsive and do not work. (2) Resume running from the delay. Table 1 shows the interaction of the processes in different states. “Normal” indicates a process that is running normally. “Lagging” indicates a process that is determined to be failed. “Recovery” indicates a process that has recovered from failure, but is still a “Lagging” process because the “Recovery” process is lagging behind other

**Table 1** Interaction of processes with different states

	Normal	Lagging	Recovery
Normal	Yes	No	Yes
Lagging	No	No	Yes
Recovery	No	No	Yes

normal processes. Table 1 indicates whether the process in the row direction state will communicate with the process in the column direction state.

Retrieving recovery processes requires collective communication, just like detecting the lagging processes. Figure 2 briefly depicts a pickup activation process with the gray process table lagging processes. For the normal processes, there are three phases of retrieving the recovered process:

1. Local response, which can probe messages from the lagging process during runtime. If a message is probed, the lagging process is notified to enter blocking and wait to be retrieved.
2. Multi-gathering recovered processes. The process responds to the recovered process, but other processes do not know that. It is necessary to multi-gather the restored processes with collective communication.

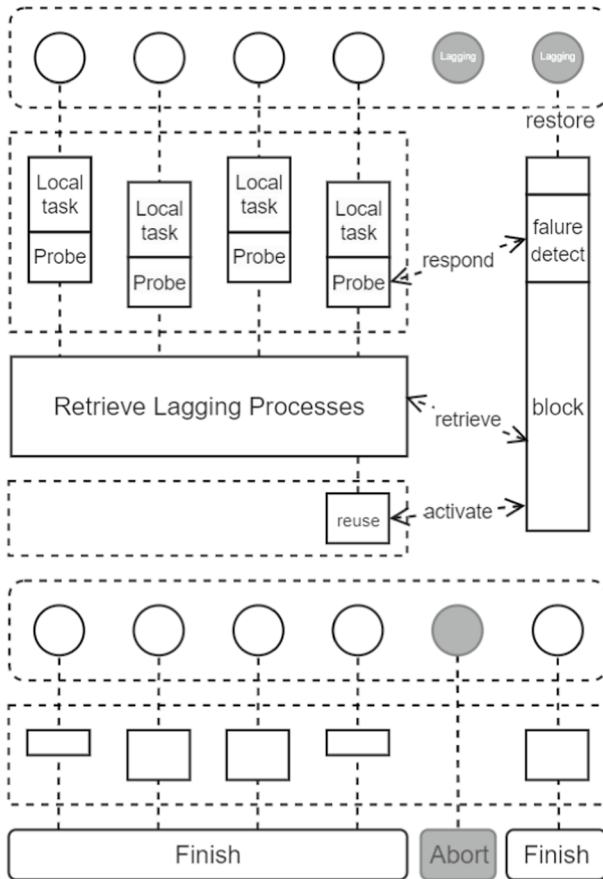


Fig. 2 Retrieving, activating, and reusing lagging processes

3. Activate and reuse the process. After the process is activated, it needs to get to the stage of the task it should perform, skip to that stage, and get the required data. This has different solutions on different applications, and the difficulty varies from one application to another. The iterative method modifies the number of iterations to skip to the same position as the other processes. Some applications skip directly the code that does not need to be executed, while others need to determine the required task to be executed by using the task pointer.

Retrieval operation should not be too frequent; otherwise, it will slow down the whole application performance. The process, in turn, needs to respond as promptly as possible to the lagging process that restores, allowing the lagging process to enter blocking and wait to be retrieved. This operation only requires the process to locally perform a probe operation to detect whether there is a message from the lagging process.

A process recovering from the lagging state does not know the state it is in until it enters the detection phase. After entering the detector, it will communicate and interact with other processes, at which point it will be notified that it is lagging, and therefore get stuck in blocking waiting to be retrieved. To reuse the recovered lagging processes and get the recovered processes back into the computation, a collection operation needs to be performed to let all processes know which lagging processes are recovered and to update the collection of lagging processes. The process responsible for activating and re-enabling the recovered process then sends the required data to the recovered process to jump to a phase consistent with the other normal processes.

#### 4 Torus-tree-based failure detection algorithm

In Sect. 3, when describing the idea of failure detection, it is assumed that the processes that reach the detection phase can compose a connectivity graph. This ensures that processes can use the connectivity of the topology to complete collective operations. Failed processes will damage the connectivity of the topology, and the efficiency of collective communication depends on the structure of the topology, which means that the topology used by the detector must be both robust and efficient. But these two features are often contradictory. Topologies that are robust and easy to repair usually communicate inefficiently, such as the ring. Highly efficient topologies tend to be more fragile, such as the tree.

In exascale systems, the number of processes called can be in the millions of cores or more. In such a large scale, the MTBF of the system will be significantly shorter. The percentage of failed processes, although small, does not get smaller as the system scales, which makes it infeasible to apply repair algorithms for many topologies to larger scales. For example, hypercubes and sibling trees [4] cannot guarantee the topology connectivity 100% when the fault percentage exceeds a certain value. For this reason we design the torus-tree [23].

### 4.1 Torus-tree

A torus-tree is a composite structure. We define a torus by  $T(d, K), K = \{k^1, k^2, \dots, k^d\}$ ,  $d$  is the dimension of the torus.  $K = \{k^1, k^2, \dots, k^d\}$  represents the vector of nodes in each dimension, and the total number of nodes on each  $T(d, K)$  torus is  $a$ . The ring is equivalent to an one-dimensional torus. Suppose there are  $b$  torus of exactly the same size and shape, and the size of the torus is  $a$ . Each node on the torus can be marked with unique coordinates for each position, and the same coordinates form a tree, so that a tree is formed, resulting in a torus-tree with node size  $a \cdot b$ . Figure 3 illustrates the torus-tree, also known as the 1D torus-tree and 2D torus-tree, with the dashed lines indicating the edges of the topology.

The robustness and repairability of the torus-tree structure benefit from the torus, because the ring is the easiest topology to repair, and remains ring-shaped after repair. Each dimension of the torus is a ring, and the rings in each dimension are able to complete communication independently and concurrently, thus improving communication efficiency to some extent. Moreover, the degree of each node on the torus  $T(d, K)$  is equal to  $2d$ , and node failure will not easily disrupt the connectivity. The high communication efficiency of the torus-tree is due to the tree shape, where each ring completes the collective communication concurrently, and then completes the collective communication concurrently through the tree topology.  $d$ -dimensional torus-tree has a time complexity of  $O\left(\sum_{i=1}^d |k^i| + \frac{n}{\prod_{i=1}^d |k^i|}\right)$

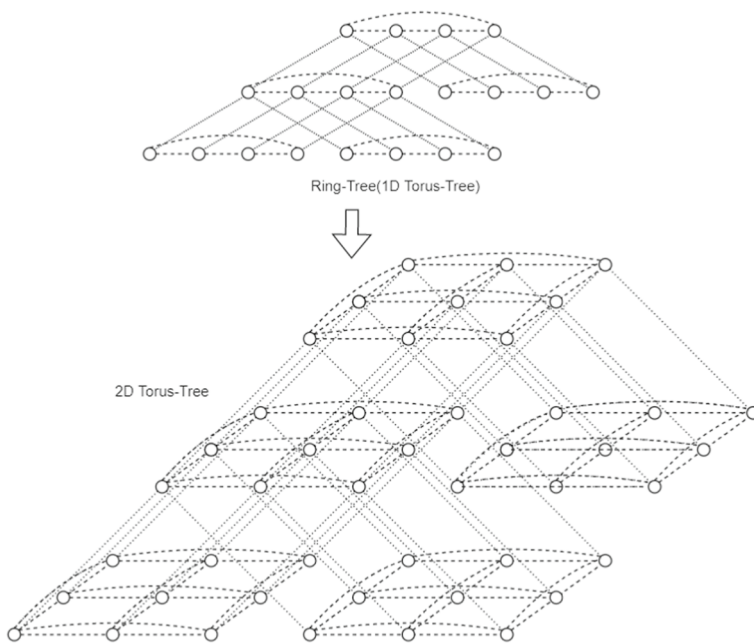


Fig. 3 Examples of torus-tree

### 4.2 Failure detection algorithm

The detection algorithm follows the flow described in Fig. 1. The process enters the detection phase and establishes connection with neighboring processes on the topology. The detector comes with epoch, which needs to be confirmed with the current epoch when establishing connection. A connection is established using two pairs of send and receive, sending an empty message and Epoch, respectively. The reasons are:

1. To avoid errors and delays caused by MPI messages stored in the cache, and the process can reconfirm the arrival status of the process after receiving the message.
2. If a process time out, it can find a new valid process to connect to, and it does not affect the message reception of the originally connected object. And it can try to connect multiple processes at the same time.
3. Processes that have recovered from a failure can actively enter blocking. The process judged as lagging may recover from the fault, but its progress has lagged behind the normal process. When the process resumes and enters the connection phase of the detector, it may receive the connection information of the process from the cache, and it needs to confirm again at this time.

For the sake of illustration, the detection algorithm is first described with the simplest ring, and then directly expanded to the torus-tree based on topological features.

According to Fig. 4. Processes can start performing counting after there are processes successfully connected in both directions of the ring, counting according to the currently reserved connection status. Processes in the ring can judge their own position in the current ring based on their own logical coordinates and the logical coordinates of neighboring processes. Processes in the ring can determine their own

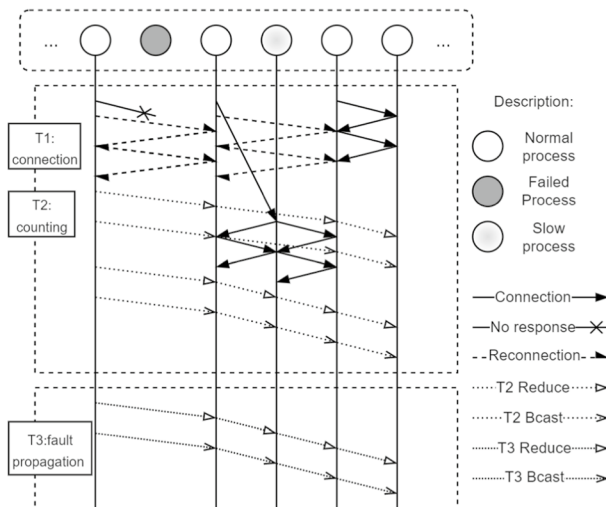


Fig. 4 Connction, counting, and multi-gather lagging processes on ring



position in the current ring based on their own logical coordinates and the logical coordinates of neighboring processes. From the process with the smallest current coordinate number, messages are passed and added up in turn, which is equivalent to reduce, and the process with the largest coordinate number can get the number of processes on the current ring. The processes then broadcast with the process with the largest current coordinate as the root.

The communication topology among processes is determined at the beginning. However, as the application runs, processes may fail, at which point the communication topology is changed by process reconnection. Reconnection is done to repair and maintain the connectivity of the communication topology, so that all surviving processes form a maximally connected subgraph of the initial topology. In this way, processes can complete collective operations such as counting, fault propagation and retrieve recovered processes on the current connectivity graph.

The ring is easy to implement this operation. Multi-reduce is equivalent to reduce then broadcast, and multi-gather is equivalent to gather then broadcast, and the root processes are the processes with the largest coordinates, following linear propagation.

The torus-tree also performs similar operations. Processes enter the detection phase and establish connection with neighboring processes in the torus-tree. Counting and fault propagation follow concurrent communication first in the torus direction and then from the tree direction to finalize the detection. If adjacent processes in the torus direction fail, the torus need to be reconnected. If neighboring processes in the tree direction fail, the trees do not need to be reconnected, and the missing data will be replenished by the neighboring processes on the torus direction. The detection algorithm is described in algorithm 1

---

**Algorithm 1:** Torus-Tree Based Failure Detection Algorithm
 

---

**Require:** Detector epoch  $e$

- 1: //Tours size is  $a$ , tree size is  $b$ ,  $n = ab$  processes in communicator.
- 2: //Detect failure.
- 3: **while** Termination conditions not met **do**
- 4:   Insert timing during execution
- 5:   //T1:establish connection.
- 6:   **for all**  $r$  in torus  $T(d, K)$ ,  $r$  is the ring **do**
- 7:     Connecting two adjacent processes on the ring
- 8:   **end for**
- 9:   Connecting adjacent processes on the tree
- 10:   //T2:counting."ind" represents the coordinates of the process.
- 11:   //Each dimension in the ring direction counting in turn.
- 12:   **for**  $r = 1$  to  $d$  **do**
- 13:      $g_r(ind) := \sum_{i=1}^{k^r} g_{r-1}(i)$ ,  $g_1(ind) = \sum_{i=1}^{k^1} a_i(e, t)$
- 14:      $h_{r-1}(ind) := \sum_{i=1}^{k^r} g_{r-1}$ ,  $h_1(ind) = \sum_{i=1}^{k^1} I_{(d_i(t) \geq T_{release})}$
- 15:   **end for**
- 16:   //Compute  $\sum_{i=1}^b a'_i(k, t)$  and  $\sum_{i=1}^{M_e} I_{(d_i(t) \geq T_{release})}$  concurrently in the tree direction
- 17:    $\sum_{i=1}^{M_e} a'_i(e, t) := \sum_{i=1}^b g_d(i)$ ,  $\sum_{i=1}^{M_e} I_{(d_i(t) \geq T_{release})} := \sum_{i=1}^b h_d(i)$
- 18:   **if**  $\sum_{i=1}^{M_e} a'_i(e, t) = M_e$  **then**
- 19:     **return** SUCCESS
- 20:   **else if**  $\sum_{i=1}^m a'_i(t) = \sum_{i=1}^m I_{(d_i(t) \geq T_{release})}$  and s.t. Pass(D(e,t),A(e,t)) **then**
- 21:     **break**
- 22:   **else if** Timeout **then**
- 23:     **return** TIMEOUT
- 24:   **else**
- 25:     **continue**
- 26:   **end if**
- 27: **end while**
- 28: //T3:Fault propagation
- 29: Multi-gather the set of lagging processes based on the current connectivity path.
- 30: **return** PASS

---

## 5 Experiments and discussions

The experiments in this section are mainly designed to verify and compare the effectiveness of the detection algorithms, as well as to give reference suggestions on the selection of parameters. Meanwhile, this paper initially designs a fault-tolerant system based on Fail-Lagging model and also applies the system to some computing examples to verify the effectiveness of the fault-tolerant algorithm. The experiments are conducted in different cluster environments and supercomputing environments, including ordinary clusters, Tianhe-2, and Chengdu Dawning supercomputers. In this paper, we focus on the problem of fault tolerance, which needs to be adapted to various compilation environments, so no restrictions are imposed on the compilation environments.

## 5.1 Repairability and detection success rate

The key to effective failure detection lies in the connectivity of the communication topology. That is, even if there are failed processes, the normal processes can remain connected to each other to achieve a strong detector. The number of failed processes as a percentage affects the repairability of the communication topology. The larger the  $\mu$ , the lower the probability of success of  $r$ . However, it is important to ensure that  $\mu$  is in the range less than a certain range and  $R(\mu)$  is 1, which means that it is definitely repairable. Since it is very difficult to calculate the probability of  $R(\mu)$  directly, the experiments estimate the repairability of the detector by a Monte Carlo method that randomly generates faults according to the probability of failure and allows the detector to detect them, counting the number of successful detections.

Experiments are conducted to compare the variation in repairable probability of sibling-tree, ring-tree, and torus-tree at the scale of 10,000 processes. Since the proportion of faults tolerated by a single hypercube is  $\frac{\log n}{n}$ , it cannot be used to implement the fault detector in this paper. Therefore, the only thing that can be done for comparison is the sibling tree topology. The fault rate is incremented from 0.01 to 0.6, the ring size is 20 for the ring-tree, and the ring size is  $10 \times 5$  for the torus-tree, 10,000 simulations are performed for each fault rate, the detection success rate of each detector is counted, and the ratio is derived, and the results are obtained as in Fig. 5. The experimental results show that the torus-tree can withstand a certain percentage of failures.

## 5.2 Efficiency of the torus-tree

Time complexity of the torus-tree is  $O\left(\sum_{k=1}^d |k^i| + \frac{n}{\prod_{i=1}^d |k^i|}\right)$ . This section examines whether the communication efficiency is the same as the theoretical value for the detection algorithms implemented in various topologies. We increment the process size from 1200 to 12,000 and compare the running time of ring-tree,

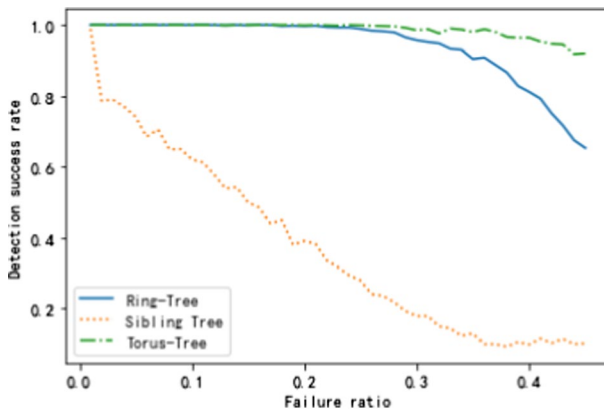


Fig. 5 Detection success rate of different topologies

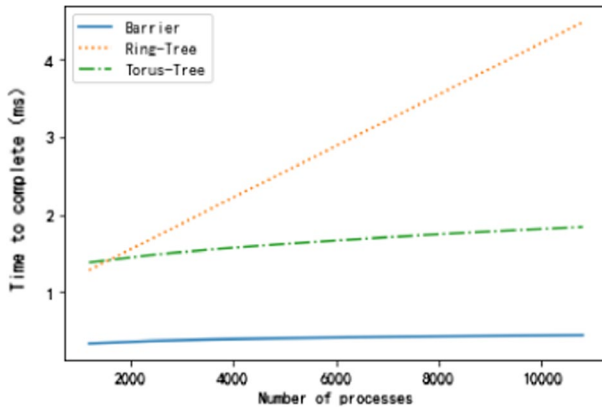


Fig. 6 Efficiency of detection algorithms

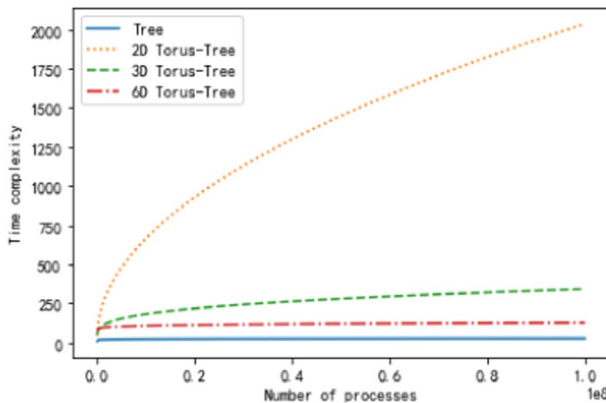


Fig. 7 Detection algorithm time complexity

torus-tree topology and standard MPI\_Barrier, and experimentally run 100 times to take the average value. MPI\_Barrier uses a communication algorithm with time complexity  $O(\log n)$  by default.

The experiment is run in the Tianhe-2 environment, and the experimental results are shown in Fig. 6. The ring size of the ring-tree is taken as 1% of the total number of processes, and the ring size of the torus-tree is also taken as 1% of the total number of processes, and the time unit is ms. Under the experiment at the scale of 10,000 cores, the completion time of the torus-tree is about 1 ms more time consuming than that of the Barrier algorithm.

Assume that the process size expands to 100 million processes and keeps the torus size at 1%. Compare the time complexity of 2-d, 3-d, and 6-d torus-tree, respectively, as shown in Fig. 7, the 6-d torus-tree topology will have better communication efficiency.

### 5.3 Selection of torus size

The choice of torus size and dimensionality is based on the total number of processes and the proportion of system failure. The problem of balancing topological robustness and efficiency is the optimization problem, and the optimization objective can be measured simply by the robustness-efficiency ratio  $H(\mu, s) = R(\mu) \cdot G(s)$ .  $R(\mu)$  measures the reparability of the topology; the higher the probability of reparability, the more robust it is. The more robust the topology, the more effective the detector is.  $G(s)$  measures the time complexity, communication topology, and number of processes are related. The higher the time complexity, the worse the efficiency and the lower the score.

We use the metric  $\bar{H}(\mu, s) = \bar{R}(\mu) \cdot \frac{1}{\sqrt{s + \log(\frac{n}{s})}}$  to measure the torus-tree.  $R(\mu)$  measures the reparability of the topology.  $G(s)$  measures the time complexity,  $s$  is the ratio of the size of the torus to the size of the total process, and the size of each dimension of the torus is the same by default.

Take the 2D torus-tree as an example. When the torus size growth step is 0.01, the experimental results are obtained as Fig. 8, by observing that the optimal torus ratio is roughly around 0.01, when the restorability and communication efficiency are optimal.

Further refining the growth step of the torus size, the experimental results are obtained in Fig. 9, and it can be seen that if the probability of failure is not high (less than 20%), the torus size is best detected at about 1% of the total number of processes. In fact, for high-performance computing, 1% is already a very high percentage of faults. If the approximate percentage of system faults is estimated to be known in advance, and the percentage of faults occurring within the MTBF time does not exceed 1%, the lower the percentage of faults, the smaller the torus size can be as a percentage, and the more efficient the detector can be. The principle is to keep both the torus size and dimension as small as possible while ensuring the detection success rate.

Fig. 8 Step size of 0.01

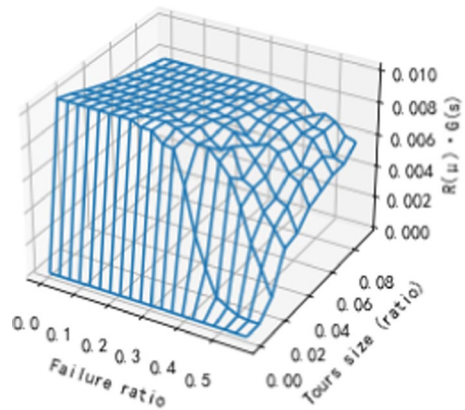
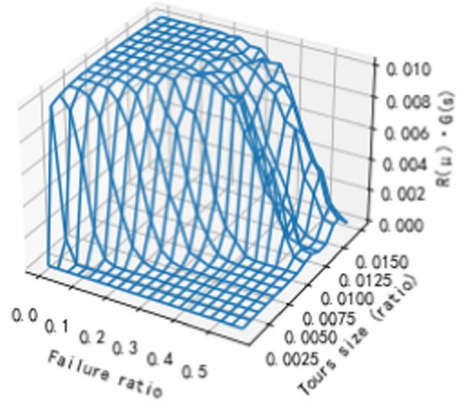


Fig. 9 Step size of 0.002



### 5.4 Fault tolerance for SART parallel iteration

The SART iterative algorithm is commonly used in CT image reconstruction. The problem  $Ax = b$  is solved by the iterative Formula (5),  $x_j^k$  denotes the value  $j = 1, 2, \dots, n$  with coordinates  $j$  in the  $k$ th iteration of the solution vector,  $n$  is the dimension of the vector,  $\lambda$  and  $\beta$  are the parameters related to the iteration step.

$$x_j^{k+1} \leftarrow x_j^k - \frac{\lambda}{\beta A_{+j}} \sum_{i=1}^m \frac{a_{ij}}{A_{i,+}} (Ax^k - b)_i, A_{i,+} = \sum_{j=1}^n a_{ij}, A_{+j} = \sum_{i=1}^m a_{ij} \quad (5)$$

During the computation,  $A_{i,+}$  and  $A_{+j}$  are fixed values that can be computed when the sparse matrix is read in and then broadcast to all processes. Using the checker-board parallel approach. Decomposition of the iterative Formula (5) as:

1. Compute  $u = Ax^k - b$ . This step requires the multiplication of sparse matrices and dense vectors, Allreduce in the row direction.
2. Compute  $t_i = \frac{u_i}{A_{i,+}}$ .  $A_{i,+}$  is stored as a constant, this step does not require collective communication, the process can calculate locally.
3. Compute  $x_j^{k+1} = x_j^k - \frac{\lambda}{\beta A_{+j}} \sum_{i=1}^m a_{ij} \cdot t_i$ . In fact, for  $\sum_{i=1}^m a_{ij} \cdot t_i$ , it is the inner product of a column of  $A$  with a column vector of  $t$ . This is written in matrix form as  $x^{k+1} = x^k - \frac{\lambda}{\beta A_{+j}} A^T t$ . Computing  $A^T t$  requires multiplication of sparse matrices and dense vectors, Allreduce in the column direction.

After completing the vector update, the iteration error is calculated and Allreduce according to the row direction. During an iterative computation, three Allreduce operations will be performed. Allreduce in the row direction when computing  $u = Ax^k - b$ , and in the column direction when computing  $A^T t$ . Allreduce in the row direction when computing the error.

Fault-tolerant computing adds the following operations to each iteration:

- Failure Detection. The frequency of failure detection is determined by the user. The detection function can be called once in several iterations, or multiple times in one iteration.
- Data Backup. The data backup scheme uses a row Shift and column Shift backup matrix, which is a call to `MPI_Cart_shift`. This ensures that fault-tolerant All-reduce operations can be executed in both the row and column directions.
- Fault-tolerant collective communications. If there are failed processes, the collective functions of MPI will not be available. Fault-tolerant collective communication can be implemented by splitting the collective communication into point-to-point communication, where the communication location of the failed process is communicated by the process holding the backup data of the failed process instead.
- Fault-tolerant computing. The computation is substituted by the process that holds the backup data of the failed process. Since redundant recovery is not used, efficiency is reduced during subsequent fault-tolerant calculations.
- Process retrieval and reuse. After the failed process is restored, it can rejoin the computation after obtaining the current iteration steps and the new set of failed processes.

The iterative algorithm is modified as described above to allow the application to cope with a certain level of failure. We simulate process failure for experiments and all eventually get the correct solution.

## 6 Conclusion and future work

This paper is devoted to the problem of fault tolerance and failure detection of applications on exascale HPC systems. The success rate of running applications on exascale systems will be greatly reduced. If the application fails to react to the failure in time, it can cause financial losses. It is hard to detect process failures because many times failed processes just behave as running abnormally slow and do not crash or report errors. To overcome this difficulty, we propose the Fail-Lagging model to describe how the application determines and reacts to process faults, and design a failure detection algorithm for fail-lagging model. As the failure process can break the communication path among processes, we also design the torus-tree communication structure to implement the failure detection with both robustness and efficiency.

In our future work, we will gradually promote the following items. (1) To further improve the fault-tolerant collective communication. (2) Further standardize the operation of the retrieval recovery process and how the recovery processes are reused. (3) Design more mature data backup and recovery algorithms so that algorithm-based fault tolerance can be better supported. (4) Abstract a more universal and common fault-tolerance solution.

**Acknowledgements** The authors would like to thank Jianfeng Zheng for interesting discussions related to this work. The work is partially supported by Key-Area Research and Development of Guangdong Province

(No.2021B0101190003). The work is also partially supported by Guangdong Province Key Laboratory of Computational Science at the Sun Yat-sen University (2020B1212060032).

## References

1. Aguilera MK, Chen W, Toueg S (1998) Failure detection and consensus in the crash-recovery model. In: Kutten S (ed) Distributed Computing, 12th International Symposium, DISC '98, Andros, Greece, September 24–26, Proceedings, Lecture Notes in Computer Science, vol 1499, pp 231–245. Springer. <https://doi.org/10.1007/BFb0056486>
2. Albrecht JR, Tuttle C, Snoeren AC, Vahdat A (2006) Loose synchronization for large-scale networked systems. In: Adya A, Nahum EM (eds) Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30–June 3, pp 301–314. USENIX. <http://www.usenix.org/events/usenix06/tech/albrecht.html>
3. Angskun T, Bosilca G, Dongarra JJ (2007) Binomial graph: a scalable and fault-tolerant logical network topology. In: Stojmenovic I, Thulasiram RK, Yang LT, Jia W, Guo M, de Mello RF (eds) Parallel and Distributed Processing and Applications, 5th International Symposium, ISPA 2007, Niagara Falls, Canada, August 29–31, 2007, Proceedings, Lecture Notes in Computer Science, vol 4742, pp 471–482. Springer. [https://doi.org/10.1007/978-3-540-74742-0\\_43](https://doi.org/10.1007/978-3-540-74742-0_43)
4. Angskun T, Fagg GE, Bosilca G, Pjesivac-Grbovic J, Dongarra JJ (2006) Scalable fault tolerant protocol for parallel runtime environments. In: Mohr B, Träff JL, Worringer J, Dongarra JJ (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17–20, Proceedings, Lecture Notes in Computer Science, vol 4192, pp 141–149. Springer. [https://doi.org/10.1007/11846802\\_25](https://doi.org/10.1007/11846802_25)
5. Arpaci-Dusseau RH, Arpaci-Dusseau AC (2001) Fail-stutter fault tolerance. In: Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20–23, Elmau/Oberbayern, Germany, pp 33–38. IEEE Computer Society. <https://doi.org/10.1109/HOTOS.2001.990058>
6. Bosilca G, Bouteiller A, Guermouche A, Hérault T, Robert Y, Sens P, Dongarra JJ (2016) Failure detection and propagation in HPC systems. In: West J, Pancake CM (eds) Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13–18, pp 312–322. IEEE Computer Society. <https://doi.org/10.1109/SC.2016.26>
7. Bosilca G, Bouteiller A, Guermouche A, Hérault T, Robert Y, Sens P, Dongarra JJ (2018) A failure detector for HPC platforms. *Int J High Perform Comput Appl* 32(1):139–158. <https://doi.org/10.1177/1094342017711505>
8. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. *J ACM* 43(2):225–267. <https://doi.org/10.1145/226643.226647>
9. Chen Z, Dongarra JJ (2008) Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans Parallel Distrib Syst* 19(12):1628–1641. <https://doi.org/10.1109/TPDS.2008.58>
10. Dwork C, Lynch NA, Stockmeyer LJ (1984) Consensus in the presence of partial synchrony (preliminary version). In: Kameda T, Misra J, Peters JG, Santoro N (eds) Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27–29, pp 103–118. ACM. <https://dl.acm.org/citation.cfm?id=1599406>
11. Egwutuoha IP, Levy D, Selic B, Chen S (2013) A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J Supercomput* 65(3):1302–1326. <https://doi.org/10.1007/s11227-013-0884-0>
12. Ferreira K, Stearley J, Laros JH, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges PG, Arnold D (2011) Evaluating the viability of process replication reliability for exascale systems. In: SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12. <https://doi.org/10.1145/2063384.2063443>
13. Graham N, Harary F, Livingston M, Stout QF (1993) Subcube fault-tolerance in hypercubes. *Inf Comput* 102(2):280–314. <https://doi.org/10.1006/inco.1993.1010>
14. Gunawi HS, Suminto RO, Sears R, Gollhofer C, Sundararaman S, Lin X, Emami T, Sheng W, Bidokhti N, McCaffrey C, Srinivasan D, Panda B, Baptist A, Grider G, Fields PM, Harms K, Ross RB, Jacobson A, Ricci R, Webb K, Alvaro P, Runesha HB, Hao M, Li H (2018) Fail-slow at scale:



- evidence of hardware performance faults in large production systems. *ACM Trans Storage (TOS)* 14(3):1–26. <https://doi.org/10.1145/3242086>
15. Gupta S, Tiwari D, Jantzi C, Rogers JH, Maxwell D (2015) Understanding and exploiting spatial properties of system failures on extreme-scale HPC systems. In: 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22–25, pp 37–44. IEEE Computer Society. <https://doi.org/10.1109/DSN.2015.52>
  16. Hurfin M, Mostéfaoui A, Raynal M (1998) Consensus in asynchronous systems where processes can crash and recover. In: The Seventeenth Symposium on Reliable Distributed Systems, SRDS 1998, West Lafayette, Indiana, USA, October 20–22, Proceedings, pp 280–286. IEEE Computer Society. <https://doi.org/10.1109/RELDIS.1998.740510>
  17. Hursey J, Graham RL (2011) Building a fault tolerant mpi application: a ring communication example. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp 1549–1556. <https://doi.org/10.1109/RELDIS.1998.740510>
  18. Kharbas K, Kim D, Hoefler T, Mueller F (2012) Assessing HPC failure detectors for MPI jobs. In: Stotzka R, Schiffers M, Cotronis Y (eds) Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2012, Munich, Germany, February 15–17, pp 81–88. IEEE. <https://doi.org/10.1109/PDP.2012.11>
  19. Lamport L, Shostak RE, Pease MC (1982) The byzantine generals problem. *ACM Trans Program Lang Syst* 4(3):382–401. <https://doi.org/10.1145/357172.357176>
  20. Losada N, González P, Martín MJ, Bosilca G, Bouteiller A, Teranishi K (2020) Fault tolerance of MPI applications in exascale systems: the ULFM solution. *Future Gener Comput Syst* 106:467–481. <https://doi.org/10.1016/j.future.2020.01.026>
  21. Schlichting RD, Schneider FB (1983) Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans Comput Syst (TOCS)* 1(3):222–238. <https://doi.org/10.1145/357369.357371>
  22. Sloan J, Kumar R, Bronevetsky G (2013) An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24–27, pp 1–12. IEEE Computer Society. <https://doi.org/10.1109/DSN.2013.6575309>
  23. Ye Y, Zhang Y, Ye W (2021) An application-level failure detection algorithm based on a robust and efficient torus-tree for HPC. In: 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCLOUD/SocialCom/SustainCom), New York City, NY, USA, September 30–Oct. 3, pp 484–492. IEEE. <https://doi.org/10.1109/ISPA-BDCLOUD-SocialCom-SustainCom52081.2021.00073>
  24. Zhai J, Chen W (2018) A vision of post-exascale programming. *Front Inf Technol Electron Eng* 19(10):1261–1266. <https://doi.org/10.1631/FITEE.1800442>
  25. Zhong D, Bouteiller A, Luo X, Bosilca G (2019) Runtime level failure detection and propagation in HPC systems. In: Hoefler T, Träff JL (eds) Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI 2019, Zürich, Switzerland, September 11–13, pp 14:1–14:11. ACM. <https://doi.org/10.1145/3343211.3343225>

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.