A Novel and Efficient Algorithm-Level Fault Tolerance Technique and Implementation Postprint

Authors: Wang Rui, Yao Erlin, Chen Mingyu, Tan Guangming

Date: 2016-06-08T00:00:00+00:00

Abstract

As the scale of high-performance computing systems continues to expand, node failures become increasingly frequent. Traditional fault tolerance techniques are mostly based on checkpointing. However, the overhead of checkpointing techniques increases with system scale, and at the Exaflops scale, its fault tolerance efficiency struggles to meet system requirements. Algorithmic failure recovery techniques offer higher efficiency compared to checkpointing. However, this technique is still based on a stop-and-wait model. For large-scale systems, the stop-and-wait model significantly impacts program parallel efficiency. This paper proposes a non-stop-and-wait algorithmic-level fault tolerance strategy—the hot replacement strategy. When node failures occur during program execution, instead of stopping to recover data on failed nodes, redundant nodes replace failed nodes, allowing computation to continue. The final correct result can be obtained through a linear transformation. To demonstrate the effectiveness of the proposed scheme, we implemented fault-tolerant High Performance Linpack (HPL) by leveraging the fault tolerance features of MPICH and evaluated the performance of the scheme. Experimental results show that even at small scales, the performance of our scheme is significantly superior to algorithmic failure recovery techniques.

Full Text

Preamble

Vol. 9 No. 6
Information Technology Newsletter

A Novel Efficient Algorithm-Level Fault Tolerance Technique and Implementation
Wang Rui, Yao Erlin, Chen Mingyu, Tan Guangming

Abstract

As high-performance computing systems continue to scale, node failures are becoming increasingly frequent. Traditional fault tolerance techniques rely primarily on checkpointing. However, checkpointing overhead grows with system scale, and its efficiency will be inadequate for exascale systems. Algorithmic failure recovery techniques offer higher efficiency than checkpointing, yet they still employ a stop-and-wait model that significantly impacts parallel efficiency in large-scale systems. This paper proposes a non-stop-wait algorithm-level fault tolerance strategy—the hot-replacement strategy. When a node fails during program execution, rather than stopping to recover the lost data, a redundant node replaces the failed node, allowing computation to continue uninterrupted. The correct final result can then be obtained through a linear transformation. To demonstrate the effectiveness of our approach, we have implemented a fault-tolerant version of the High Performance Linpack (HPL) benchmark by leveraging the fault tolerance features of MPICH, and we have evaluated its performance. Experimental results show that even at small scales, our approach significantly outperforms algorithmic failure recovery techniques.

Keywords: high-performance computing; checkpoint; algorithmic fault tolerance; Exaflops

1.1 Significance of Fault Tolerance Research

As the computational capability of high-performance computing systems continues to grow, system scale has been increasing exponentially over time. According to the latest TOP500 statistics, the world' s fastest supercomputer now contains processors numbering in the hundreds of thousands [2]. Following current technological trends, next-generation exascale computers will exceed one million cores [1]. With the development of these next-generation supercomputers, system reliability issues will become increasingly prominent.

On one hand, as system scale increases, the mean-time-to-interrupt (MTTI) becomes shorter. Garth Gibson and colleagues at Carnegie Mellon University, based on their analysis of failure data from Los Alamos National Laboratory' s supercomputers over a decade, found that failure frequency is proportional to the number of processors in the system [11, 12], as shown in Figure 1 [Figure 1: see original paper]. The three curves in the figure represent different statistical predictions corresponding to scenarios where the number of processor cores per board doubles every 18, 24, and 30 months (Figure 2 [Figure 2: see original paper] follows the same convention).

On the other hand, high-performance applications continue to grow in data scale, computational complexity, and execution time. For many large-scale scientific computations, the MTTI of high-performance computing systems is already shorter than program execution time. Therefore, efficient fault tolerance mechanisms must be employed to improve system reliability and meet application requirements.

1.2 Current State of Fault Tolerance Research

Traditional fault tolerance techniques are primarily based on checkpointing, which periodically saves system state by writing checkpoints to disk; if a failure occurs, the system rolls back to the previous checkpoint and resumes computation. However, as the gap between computational capability and disk I/O bandwidth in high-performance computing systems continues to widen, checkpointing efficiency will be inadequate for exascale systems.

Based on analysis of data from the Computer Failure Data Repository (CFDR) [13], Gibson et al. predict that following current technological trends, checkpointing efficiency in future supercomputing systems will approach zero [8, 11], as shown in Figure 2. System-level checkpointing is transparent to users, but the overhead becomes staggering at large scales. The main costs of checkpointing include: (1) rollback upon node failure invalidates computation performed during the rollback period; (2) periodic checkpoint writes to disk, where external storage bandwidth is far lower than memory bandwidth; and (3) the stop-and-wait fault tolerance model, where a single node failure causes the entire system to halt and wait for repair.

J. S. Plank et al. proposed diskless checkpointing [14], which uses high-speed memory instead of slow disk to store checkpoints and encodes data from compute nodes to store in additional redundant nodes. Diskless checkpointing offers better scalability than traditional checkpointing but still incurs rollback and stop-and-wait overhead. Algorithm-based fault tolerance (ABFT) recovery [3-5, 15-18] goes further by eliminating rollback entirely. Although this technique is not application-transparent or universal, it covers a broad range of applications, including ScaLAPACK [4, 15], HPL [16], PCG solvers [5, 17], and iterative linear solvers in PETSc [18]. The primary advantage of ABFT recovery is that it requires no additional time to save checkpoints—redundant data is updated synchronously during computation, eliminating the need for rollback. Moreover, since redundant data resides in memory, the scheme avoids slow external storage I/O. Experiments show that this technique outperforms both traditional checkpointing [5, 17] and diskless checkpointing [18].

However, ABFT recovery still employs a stop-and-wait fault tolerance model, where a single process failure during execution forces all running processes to halt and wait for recovery of the failed process' s data. According to Amdahl' s law, program speedup is limited by the serial portion of the program. For large-scale systems, the stop-and-wait model significantly impacts parallel efficiency.

Research into non-stop-wait, application-aware fault tolerance strategies represents a new direction for future fault tolerance development. The main challenges include: (1) application-level failure awareness requires deep understanding of application algorithms, complicating implementation; (2) designing encoding schemes that ensure both efficiency and numerical stability remains an open problem; and (3) non-stop-wait fault tolerance models demand greater support from the fault tolerance system—current MPI implementations' fault tolerance

features are mostly checkpoint-based, and implementing application-level fault tolerance requires new MPI support.

1.3 Main Contributions of This Paper

Building upon research in ABFT recovery, this paper proposes a novel efficient algorithm-level fault tolerance scheme—ABFT hot-replacement. When a node fails during program execution, this scheme avoids stop-and-wait recovery of lost data by instead replacing the failed node with redundant data, allowing computation to continue immediately. At the end of execution, the correct result can be obtained from the intermediate result through a simple linear transformation.

To verify the correctness of our approach, we have implemented a fault-tolerant HPL by combining our scheme with the new fault tolerance features of MPICH2 and evaluated its performance. Experimental results demonstrate that even at small scales, our approach offers clear performance advantages over ABFT recovery.

1.4 Paper Organization

This paper consists of six chapters. Chapter 1 introduces the research significance and current state of fault tolerance techniques. Chapter 2 reviews related work, explains ABFT recovery, and briefly introduces MPICH2' s support for algorithm-level fault tolerance. Chapter 3 proposes the non-stop-wait ABFT hot-replacement scheme and discusses handling multiple failures. Chapter 4 details the implementation of the ABFT hot-replacement scheme. Chapter 5 evaluates the performance of the ABFT hot-replacement scheme and verifies its correctness. Chapter 6 outlines future work.

2 Related Work

This chapter first introduces the basic concepts of algorithm-based fault tolerance, then elaborates on the fault tolerance strategy of ABFT recovery, and finally discusses the MPI support required for implementing algorithm-level fault tolerance techniques.

2.1 Algorithm-Based Failure Recovery Techniques

The concept of Algorithm-Based Fault Tolerance (ABFT) [7] was first proposed by K. H. Huang and J. A. Abraham in 1984 for detecting, locating, and correcting transient errors in large-scale integrated circuits. The core idea of ABFT is to first encode the original data through a transformation, then redesign the algorithm workflow so that redundant data is updated synchronously during computation, enabling error detection and correction. This scheme is not universal, as not all applications can synchronize redundant and computational data updates. ABFT techniques primarily target applications involving specific

matrix operations, such as matrix addition, multiplication, inner products, LU decomposition, and transposition.

Algorithm-based failure recovery techniques [3-5, 15-18] represent further advancement by Z. Chen and J. Dongarra et al. to address node failures in high-performance computing applications. These techniques encode data from compute nodes (typically through redundant summation) and store it in additional redundant nodes, designing parallel algorithms that synchronize updates to redundant and compute node data during computation, thereby enabling recovery of data from failed nodes.

Consider single-node failure. Since we cannot know which node will fail beforehand, any fault tolerance scheme must provide mechanisms to recover data from any arbitrary node. Assume there are n compute nodes, with data D_i on each compute node and data E on the redundant node. During computation, the data on each node satisfies:

$$D_1 + D_2 + \cdots + D_n = E$$

If node i fails, data D_i can be recovered using:

$$D_i = E - D_1 + \cdots + D_{i-1} + D_{i+1} + \cdots + D_n$$

However, in practical applications, maintaining this redundant sum relationship is not inherent but requires specific design. How to design algorithms that preserve the redundant sum relationship during computation is a key problem in ABFT research.

2.2 MPI Support for Algorithm-Level Fault Tolerance

To implement algorithm-level fault tolerance, MPI implementations must provide mechanisms for effective user participation in fault tolerance. Our implementation uses the MPICH2-trunk-r7834 package, whose main fault tolerance features are:

1. Node failure does not cause entire program termination. This is ensured by adding the `-disable-auto-cleanup` option when running programs with the `mpirun` command.
2. Algorithm-level fault tolerance schemes rely on underlying failure detection. Therefore, MPI implementations must provide mechanisms for processes to query which nodes have failed. MPICH2 adds the `MPICH_ATTR_FAILED_PROCESS` attribute to the `MPI_COMM_WORLD` communicator. Processes can obtain failure information by querying this attribute' s value.

3. Communication operations affected by failed nodes return error codes, primarily used to determine whether messages need to be resent and re-received. It should be noted that to reduce collective operation overhead in failure-free scenarios, MPICH2 relaxes restrictions on collective communications. For example, a barrier operation containing failed processes cannot guarantee that all processes return error codes, and a broadcast operation containing failed processes cannot guarantee that all processes correctly receive messages.

It should be noted that MPICH2-trunk-r7834 does not support dynamic communicator adjustment in the event of node failure, but the newer MPICH2-1.4 can do so. In our implementation, communicator adjustment must be implemented at the application level, which we will address in Chapter 4.

3 Scheme Design

This chapter proposes a novel efficient algorithm-level fault tolerance technique, discusses solutions for single and multiple failures, and finally presents the corresponding fault tolerance design for HPL.

3.1 Hot-Replacement Strategy

For simplicity, we first consider single failure scenarios. Assume that during computation, data D_i on compute node i and data E on the redundant node satisfy:

$$D_1 + D_2 + \cdots + D_n = E$$

Once node i fails, rather than having all "alive" processes stop and wait for recovery of data on the failed process, we replace the failed node with the redundant node, allowing computation to continue immediately.

From a global perspective, before failure occurs, data on n compute nodes is $D_1, D_2, \ldots, D_{i-1}, D_i, D_{i+1}, \ldots, D_n$. After replacement, it becomes $D_1, D_2, \ldots, D_{i-1}, E, D_{i+1}, \ldots, D_n$. Let $D' = D \times T$, then T can be represented as an $n \times n$ matrix:

$$T = \begin{pmatrix} 1 & 0 & \cdots & 1 & \cdots & 0 \\ 0 & 1 & \cdots & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 1 \end{pmatrix}$$

where diagonal elements and elements in column i (omitted) are 1, and all others are 0. As shown in (6), T is a non-singular matrix. If all operations on

data D are linear transformations (such as matrix LU decomposition, etc.), the relationship $D' = D \times T$ can be maintained throughout computation. At the end of computation, the correct result based on original data D can be obtained from the intermediate solution based on D'. This process is essentially a linear transformation based on T.

However, this encoding relationship $D' = D \times T$ cannot be maintained in all high-performance computing applications, but it can be preserved in a class of matrix operations involving linear transformations, such as matrix addition, multiplication, LU decomposition, inner products, and transposition.

3.2 Handling Multiple Failures

For multiple failure scenarios, one approach is to use multi-level redundancy. However, this solution cannot effectively address the problem for next-generation high-performance systems because redundancy will eventually be exhausted.

Another novel approach is to accelerate redundant sum reconstruction in the background—that is, to re-encode data after replacement and store it on new redundant nodes. Reconstructing the redundant sum involves substantial communication and requires participation from compute nodes. We propose "background accelerated reconstruction," which aims to "free" compute nodes from redundant reconstruction work as early as possible by adding nodes or network resources to accelerate reconstruction in the background, allowing redundant nodes to "catch up" with compute nodes quickly. The correct result can then be obtained by performing multiple hot-replacements to compute intermediate results, followed by iterative transformations.

3.3.1 HPL Overview

[Figure 3: see original paper] HPL Computational Flow Diagram

```
Each process generates its local random matrix A
for k = 0, 1, ···
    calculate broadcast and ( ) kL and pivoting information right;
    perform row swaps and calculate ( ) 1
    update the trailing sub-matrix (
solve Ux L b-= to obtain x;
```

[Figure 4: see original paper] HPL Program Pseudocode

HPL is the benchmark used for the TOP500 supercomputer rankings. Its main algorithm solves linear systems $Ax = b$ using Gaussian Elimination with Partial Pivoting. It first computes the LU decomposition of the $n \times (n + 1)$ coefficient matrix $[A|b]$ to obtain $[A|b] = [[L, U]L^{-1}b]$, then solves $Ux = L^{-1}b$ for x, where U is an upper triangular matrix and L is a lower triangular matrix, as shown

in Figure 3. Figure 4 shows the pseudocode for the HPL program using LU decomposition. For more information, see [6].

3.3.2 Fault Tolerance Design

Consider single-node failure first. As described in §3.3.1, HPL solves linear equations $Ax = b$ using Gaussian Elimination with Partial Pivoting. Assume process P_i fails during execution. After replacing the failed data with redundant data, the linear system becomes $A'y = b$. At the end of program execution, we obtain intermediate solution y. Since Gaussian Elimination with Partial Pivoting involves only linear transformations, the relationship $A' = A \times T$ can be maintained throughout execution.

Combining equations (7), (8), and (9), the solution x can be obtained by:

$$x = T^{-1}y$$

Assuming transformation matrix T has the form shown in (6), substituting (6) into (10) yields:

$$x_i = \begin{cases} y_i - y_j & \text{if } i \neq j \\ y_i & \text{if } i = j \end{cases}$$

As shown above, the cost of computing x from y is $O(n)$, significantly lower than the $O(n^2)$ overhead of ABFT recovery for single-node failure.

For multiple failures, x can be obtained iteratively from intermediate solution y. Considering n failures with transformation matrix T_i for the i-th failure:

$$x = T_1^{-1}T_2^{-1} \times \cdots \times T_n^{-1}y$$

In actual implementation, due to HPL's use of 2D block-cyclic data distribution, matrix encoding and transformation matrices become more complex, which we will address in the next chapter.

3.4 Advantages

Compared with ABFT recovery, our scheme offers several advantages. First, the hot-replacement strategy enables instantaneous failure switching without stop-and-wait recovery of failed nodes. Second, because transformation matrix T is extremely sparse, the overhead of computing the final solution from intermediate results is minimal, far lower than the cost of recovering data on failed nodes in ABFT recovery. Based on these two points, ABFT hot-replacement theoretically outperforms ABFT recovery. To verify correctness, the next chapter details the implementation of ABFT hot-replacement in HPL.

4 Implementation

We have currently implemented a fault-tolerant HPL using the ABFT hot-replacement scheme for single-node failure. The following implementation details focus on single-node failure scenarios. As described in §3.3.1, HPL' s main component consists of a series of elimination, row broadcast, and update phases, which account for the vast majority of HPL's total execution time. Our work targets fault tolerance during this period; therefore, if node failure occurs outside this window, all HPL processes will terminate.

4.1 Matrix Encoding

Matrix encoding uses a scheme similar to reference [4]. In HPL, random matrix A is distributed across a 2D $P \times Q$ process grid in block-cyclic fashion. To handle single-node failure, we add P processes, forming a new $(P+1) \times Q$ grid with the original processes. The redundant P processes are placed in column $Q+1$ of the new grid, storing encoded data from the first Q columns. Encoding information can be obtained by summing local data from processes in the first Q columns. Figure 5 [Figure 5: see original paper] shows the 2D process grid and encoded data distribution. For example, a $P = 2, Q = 2$ grid (Figure 5(a)) becomes a new grid with redundant processes (Figure 5(b).

In this encoding scheme, the "row-sum" relationship for matrix U can be maintained during computation—that is, preserved after each update phase [16].

4.2 Failure Detection and Hot-Replacement

First, we present the overall performance of optimized Double-precision General Matrix Multiply (DGEMM) on heterogeneous systems. Benchmark programs cannot predict when failures will occur, so before failure happens, we don't know when or which node will fail. This information can be obtained by periodically (after each phase) querying the `MPICH_ATTR_FAILED_PROCESSES` attribute of `MPI_COMM_WORLD`.

Since different processes finish the same phase at different times, a barrier operation is required before querying to ensure consistent results.

If a process failure is detected after a phase, all "alive" processes enter a unified entry point to handle the failure using the ABFT hot-replacement scheme—that is, replacing the failed process column with the redundant process column to allow computation to continue. Using the process grid in Figure 5(b) as an example, if process P_3 fails, the grid after hot-replacement is shown in Figure 5(c).

Hot-replacement here is not physical replacement but is achieved by exchanging certain member variables in process grid-related data structures. It involves no data communication and can therefore be completed rapidly.

It should be noted that hot-replacement can only occur when the "row-sum" relationship for U is preserved—that is, after the update phase completes. If process failure is detected after an elimination phase, the remaining "alive" processes must complete row broadcast and update before hot-replacement can occur.

Another issue is how to update communicator state after hot-replacement. Communicators represent groups of "alive"processes. Since MPICH2 currently cannot dynamically adjust communicators (e.g., add or remove processes), we address this at the application level by assigning each process a virtual rank. Processes communicate using virtual ranks. We have written two macros to implement mapping between virtual ranks and process ranks in `MPI_COMM_WORLD` based on processor grid status information.

4.3 Background Recovery

After hot-replacement, we observe that U is no longer an upper triangular matrix after the final trailing matrix update completes, complicating the solution of $Uy = L^{-1}b$ for y. One solution is to not replace already-decomposed data with redundant data but instead recover it on the corresponding redundant process. Another benefit is that this data is not needed immediately but only in the final back-substitution phase, so its recovery can proceed in the background.

We use non-blocking point-to-point communication to overlap the communication required for recovery with the computation of subsequent update phases. For large recovery data volumes, we may need to split background recovery data into multiple portions. It should be noted that as computation progresses, the amount of data requiring updates continuously decreases, so the amount of communication data partitioned each time should also decrease accordingly.

4.4 Message Fault Handling

The above methods suffice for handling node failures during elimination and update phases. For row broadcast phases, special handling is required. The six row broadcast methods provided by HPL are all based on message forwarding. If a process fails during broadcast, processes that do not communicate directly with it will hang. For example, if process P_1 forwards a message from process P_0 to process P_2 and P_0 fails, P_1 can learn of P_0' s failure through the error code returned by `MPI_Recv`, but P_2 cannot obtain this information because its parent node P_1 remains "alive." Therefore, for process failures during broadcast, we need a robust message broadcast mechanism that satisfies two conditions:

1. Either all nodes successfully receive the message, or all return an error signal.

This section customizes a "robust" broadcast method for HPL' s commonly used increasing-2-ring-modified (2rinM) broadcast pattern.

[Figure 6: see original paper] Message Transmission Path in 2rinM Mode

In 2rinM mode, the message transmission path is shown in Figure 6. Process 0 first sends to process 1, then the remaining $Q-1$ processes are divided into two groups: processes 2 to $(Q+1)/2-1$ as one group, and processes $(Q+1)/2$ to $Q-1$ as another. The message is then forwarded sequentially from processes 2 and $(Q+1)/2$ as source nodes.

Based on the temporal relationship of message transmission, the message passing in Figure 6 can be represented as a tree structure shown in Figure 7 [Figure 7: see original paper]. We assign a parent node to each non-root node. Parent node assignment must satisfy two conditions: (1) the parent node' s message transmission precedes the child node' s; (2) message transmissions between parent and child nodes have no direct dependencies. Parent and child nodes handshake after each `MPI_Recv` to decide whether to resend and re-receive messages—that is, each non-root node sends the return code of `MPI_Recv` to its parent node. If `MPI_Recv` returns an error, it must wait for the parent node to resend the message. For a parent node, it receives return codes from child nodes and determines whether to resend messages. Using Figure 7 as an example, we assign parent nodes as shown, with dashed lines indicating child-to-parent relationships. This method ensures that for single-node failure, failure of any non-root node does not affect successful message delivery.

[Figure 7: see original paper] Tree Diagram of Message Transmission in 2rinM Mode (dashed lines indicate parent-child relationships)

But what if root node 0 fails before the first message transmission succeeds? In this case, all nodes not directly receiving from node 0 would hang. To solve this problem, we separate the first message transmission from the row broadcast, applying the above mechanism only after the first message transmission succeeds.

5 Scheme Validation

This chapter evaluates the performance and rounding error of the ABFT hot-replacement scheme through experiments addressing three questions:

1. How does the performance of different algorithm-level fault tolerance schemes compare?
2. What is the rounding error introduced by the algorithm-level fault tolerance scheme?
3. How does the timing of failure affect performance?

The first two experiments were conducted on the Dawning 5000A platform using 16 nodes, each with four quad-core 2.2GHz AMD Opteron processors and 64GB of shared memory, connected via Gigabit Ethernet. The third experiment was conducted on the "Super Dragon" platform consisting of 8 blades, each with 10 Intel Xeon X5650 processors totaling 960 cores. Nodes within the same blade are connected via InfiniBand, while blades are connected by a single InfiniBand

link. Both platforms run Linux. The MPICH2 package used was MPICH2-trunk-r7834. All timing statistics were obtained using the `MPI_Wtime` function.

5.1 Performance Comparison of Algorithm-Level Fault Tolerance Schemes

The first experiment compares the performance of two different algorithm-level fault tolerance schemes. We integrated both ABFT recovery and ABFT hot-replacement into HPL to handle single-node failure. Node failure was simulated by forcibly terminating a process.

Table 1 . Experimental Configuration on D5000A Platform

Matrix Order	Nodes	Processes per Node
10,000	16×12	16×16
20,000		
30,000		
40,000		
50,000		
60,000		

Table 2 . HPL Execution Time

Matrix Order	No Failure	Hot-Replacement	Recovery
10,000			
20,000			
30,000			
40,000			
50,000			
60,000			

In this experiment, the order of random matrix A and compute node usage are shown in Table 1. The matrix order is on the order of 10^4. Table 2 shows HPL execution times for no failure, hot-replacement, and recovery scenarios. Figure 8 [Figure 8: see original paper] shows the overhead of both schemes relative to the no-failure case. Table 2 shows that for single-node failure handling, ABFT hot-replacement reduces total HPL execution time by 10-200 seconds compared to ABFT recovery—equivalent to 1-5% of HPL' s total execution time without failures. Overhead in both schemes varies not only with the amount of data allocated to each process but also with the mapping relationship between processes and processor cores.

[Figure 8: see original paper] Overhead of Different Algorithm-Level Fault Tolerance Schemes

5.2 Rounding Error Comparison of Algorithm-Level Fault Tolerance Schemes

Algorithm-level fault tolerance introduces rounding error through arithmetic encoding of floating-point numbers to construct redundant data. Additionally, ABFT hot-replacement replaces failed data with redundant data involving matrix transformations, further exacerbating rounding error. The second experiment measures this rounding error by examining HPL' s error check results:

$$\|Ax - b\|_\infty / (n \cdot \|A\|_\infty \cdot \|x\|_\infty + \|b\|_\infty) \leq \epsilon$$

where n is matrix dimension and ϵ is machine precision.

This experiment uses the same data scale and node configuration as the first experiment (Table 1). Figure 9 [Figure 9: see original paper] shows HPL error check results for single-node failure handling using ABFT hot-replacement and ABFT recovery, as well as the no-failure case. The results show that ABFT recovery introduces relatively small rounding error, while ABFT hot-replacement introduces rounding error roughly twice that of the no-failure case.

[Figure 9: see original paper] Error Comparison of Different Fault Tolerance Methods

5.3 Impact of Failure Timing on Hot-Replacement Performance

The third experiment evaluates how failure timing affects ABFT hot-replacement performance. Conducted on the "Super Dragon 1" platform, we used 75 nodes totaling 900 cores. We launched one process per core, organizing these 900 processes into a 30×30 grid. The matrix order in this experiment was 20,000, with failure timing controlled by the `sleep` function in a Python script.

[Figure 10: see original paper] Impact of Failure Timing on Performance

Results are shown in Figure 10. We observe that when failure occurs after 1,000 seconds of computation, total HPL execution time increases sharply. This is because as computation progresses, the amount of data requiring background recovery also increases. When this data volume grows to the point where it cannot be fully overlapped with subsequent update phases, additional overhead is incurred. However, theoretically, the maximum overhead of ABFT hot-replacement will not exceed that of ABFT recovery.

6 Future Work

Future work may proceed in three directions. The first direction involves designing background accelerated redundant sum reconstruction schemes to handle

multiple node failures. The greatest challenge of this approach is overhead—reconstructing redundant sums involves substantial communication and requires compute node participation. We need to design efficient schemes that "free" compute nodes from redundant reconstruction work as early as possible, adding nodes or network resources to accelerate background reconstruction and allowing redundant nodes to "catch up" quickly. Another issue is the rounding error introduced by the hot-replacement strategy. Figure 11 [Figure 11: see original paper] shows how rounding error varies with failure count. Experiments show that when failure count exceeds 10, results often fail HPL' s error check. Selecting more numerically stable encoding schemes or correcting final results through empirical values may become part of our future work. Additionally, implementing multiple failure handling requires not only appropriate modifications at the application level but also new MPI support, such as dynamic communicator adjustment.

[Figure 11: see original paper] Relationship Between Rounding Error and Failure Count

The second research direction involves applying the scheme to larger-scale supercomputers and implementing more comprehensive failure handling. Furthermore, extending the hot-replacement strategy to more high-performance computing applications is also a topic for future research.

References

[1] The International Exascale Software Project. http://www.exascale.org.
[2] Top 500 supercomputing sites. http://www.top500.org. Last accessed February 2011.
[3] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In Proceedings of the 20st IEEE International Parallel and Distributed Processing Symposium, page 76, 2006.
[4] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. IEEE Transactions on Parallel and Distributed Systems, 19(12), December 2008.
[5] Z. Chen, G. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Building fault survivable mpi programs with ft-mpi using diskless checkpointing. In Proceedings for ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 213–223, 2005.
[6] HPL benchmark sites. http://www.netlib.org/benchmark/hpl.
[7] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers, C-33(6):518–528, June 1984.
[8] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. Journal of Physics: Conference Series, 78, 2007.
[9] MPI-Forum fault-tolerance working group. https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_users_guide.

[10] Franck Cappello. Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. International Journal of High Performance Computing Applications, 23:212-226, August 2009
[11] G. Gibson, B. Schroeder, and J. Digney. Failure tolerance in petascale computers. CTWatchQuarterly, 3(4), November 2007.
[12] G. Gibson, Reflections on Failure in Post-Terascale Parallel Computing, Keynote at Int. Conf. on Parallel Processing, Xi' An China, 2007.
[13] The computer failure data repository sites. http://cfdr.usenix.org.
[14] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. IEEE Trans. Parallel Distrib. Syst., 9(10):972–986, 1998.
[15] D. Hakkarinen and Z. Chen. Algorithmic Cholesky factorization fault recovery. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium, Atlanta, GA, USA, April
[16] T. Davies, C. Karlsson, H. Liu, and Z. Chen. Algorithm-based recovery for HPL. In Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 303–304, 2011.
[17] Z. Chen and J. Dongarra. Highly scalable self-healing algorithms for high performance scientific computing. IEEE Transactions on Computers, July 2009.
[18] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. Proceedings of the 20th ACM International Symposium on High-Performance Parallel and Distributed Computing, June 2011.

Author Biographies

Wang Rui: Master' s student, State Key Laboratory of Computer Architecture
Yao Erlin: Assistant Researcher, State Key Laboratory of Computer Architecture
Chen Mingyu: Researcher, State Key Laboratory of Computer Architecture, cmy@ict.ac.cn
Tan Guangming: Associate Researcher, State Key Laboratory of Computer Architecture

Note: Figure translations are in progress. See original paper for figures.