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

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

Date: 2017-03-10T00:00:00+00:00

Abstract

As high-performance computing systems continue to expand in scale, node failures are becoming increasingly frequent. Traditional fault tolerance techniques are predominantly based on checkpointing. However, the overhead of checkpointing increases with system scale, and its fault tolerance efficiency at the Exaflops scale struggles to meet system requirements. Algorithm-based failure recovery techniques offer higher efficiency compared to checkpointing, yet they still rely 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 algorithm-level fault tolerance strategy—the hot replacement strategy. When node failures occur during program execution, rather than stopping to recover data on failed nodes, redundant nodes replace the failed nodes, enabling computation to continue. The final correct result can be obtained through a linear transformation. To validate the effectiveness of the proposed scheme, we implemented fault-tolerant High Performance Linpack (HPL) by integrating MPICH' s fault tolerance features and evaluated the scheme' s performance. Experimental results demonstrate that even at small scales, our scheme' s performance is significantly superior to algorithm-based failure recovery techniques.

Full Text

Preamble

Vol.9 No.6 Information Technology Letters 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 are predominantly based on checkpointing. However, the overhead of checkpointing grows with system scale, making its fault tolerance efficiency inadequate for exascale systems. Algorithm-based fault recovery techniques offer higher efficiency than checkpointing, yet they still rely on a stop-and-wait model that significantly impacts parallel efficiency in large-scale systems. This paper proposes a non-stop-and-wait algorithm-level fault tolerance strategy—the hot-replacement strategy. When a node failure occurs during program execution, rather than stopping to recover data on the failed node, a redundant node replaces it, allowing computation to continue immediately. The final correct result can then be obtained through a linear transformation. To demonstrate the effectiveness of our approach, we implemented a fault-tolerant version of the High Performance Linpack (HPL) benchmark by leveraging the fault tolerance features of MPICH and evaluated its performance. Experimental results show that even at small scales, our scheme significantly outperforms algorithm-based fault recovery techniques.

Keywords: high-performance computing; checkpoint; algorithm-based fault tolerance; Exaflops

1.1 Significance of Fault Tolerance Research

As the computational capability of high-performance computing systems increases, system scale has grown exponentially over time. According to the latest Top500 supercomputer rankings, 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]. Alongside 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 ten years of failure data from Los Alamos National Laboratory supercomputers, found that failure frequency in supercomputer systems is proportional to the number of processors [11, 12], as shown in [Figure 1: see original paper].

[Figure 1: see original paper] The relationship between mean-time-to-interrupt and processor count [11, 12]. The three curves represent different statistical predictions corresponding to models where the number of processor cores per board doubles every 18, 24, and 30 months respectively (the same applies to [Figure 2: see original paper] without separate annotation).

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 mean-time-to-interrupt of high-performance computing systems is already shorter than program execution time. Therefore, efficient fault tolerance methods must be employed to improve system reliability and meet application requirements.

1.2 Current Status of Fault Tolerance Research

Traditional fault tolerance techniques are primarily based on checkpointing, which periodically saves system state by writing checkpoints to disk; upon failure, 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. predicted that following current technological trends, the fault tolerance efficiency of checkpointing will approach zero in future supercomputer systems [8, 11], as shown in [Figure 2: see original paper].

[Figure 2: see original paper] Gibson et al.'s prediction of checkpointing efficiency [11].

System-level checkpointing is transparent to users, but the overhead becomes staggering at large scales. The main costs of checkpointing are: (1) rollback upon node failure invalidates all computation performed since the last checkpoint; (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 pause 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. While diskless checkpointing offers better scalability than traditional checkpointing, it 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 greatest 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 have shown that this technique outperforms both traditional checkpointing [5, 17] and diskless checkpointing [18].

However, ABFT recovery still relies on the stop-and-wait fault tolerance model, where even a single process failure causes all running processes to pause 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-and-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-and-wait fault tolerance models demand greater support from fault tolerance systems—existing MPI implementations' fault tolerance features are mostly checkpoint-based, requiring new MPI support for application-level fault tolerance.

1.3 Main Work and Contributions

Building upon research into ABFT recovery, this paper proposes a novel efficient algorithm-level fault tolerance scheme—Algorithm-Based Fault Tolerance Hot-Replacement (ABFT hot-replacement). When a node failure occurs during program execution, this scheme avoids stop-and-wait recovery of lost data by instead replacing it with redundant data, allowing computation to continue immediately. At program completion, the correct result can be obtained from the intermediate result through a simple linear transformation.

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

1.4 Paper Organization

This paper consists of six chapters: Chapter 1 introduces the research significance and current status 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-and-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 validates its correctness; and Chapter 6 outlines future work.

2 Related Work

This chapter first introduces the fundamental 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 Fault Tolerance Recovery

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 is to first encode the original data through a transformation, then redesign the algorithm flow so that redundant data is updated synchronously during computation, enabling error detection and correction during execution. This scheme is not universal, as not all applications can synchronously update redundant and computational data. ABFT techniques primarily target applications involving specific matrix operations, such as matrix addition, multiplication, inner products, LU decomposition, and transposition.

ABFT recovery [3-5, 15-18], advanced by Z. Chen and J. Dongarra et al., extends ABFT to handle node failures in high-performance computing applications. This technique encodes data from compute nodes (typically through redundant summation) and stores it on additional redundant nodes, designing parallel algorithms that synchronously update data on both redundant and compute nodes during computation, thereby enabling recovery of data on failed nodes.

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

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

If node i fails, its data Di can be recovered using:

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

However, in practical applications, this redundant sum relationship is not inherent but requires specific design. How to design algorithms that maintain 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 users to effectively participate in fault tolerance. Our implementation uses the MPICH2-trunk-r7834 software 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 node failure information by querying this attribute.
3. Communication operations affected by failed nodes return error codes, primarily used to determine whether messages need to be resent and rereceived. 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 upon node failure, but the newer MPICH2-1.4 can do so. In our implementation, communicator adjustment must be implemented at the application layer, which we will address in Chapter 4.

3 Design

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

3.1 Hot-Replacement Strategy

For simplicity, we first consider single failure. Assume that during computation, data Di 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 "live"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 $D = (D_1, D_2, \ldots, D_n)^T$. After replacement, it becomes $D' = (D_1, \ldots, D_{i-1}, E, D_{i+1}, \ldots, D_n)^T$. Let $D' = T \times D$, 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 the equation, T is a non-singular matrix. If all operations on data D are linear transformations (such as matrix LU decomposition, etc.), then the relationship $D' = T \times D$ can be maintained throughout computation. At the end of execution, the correct result for the original data D can be obtained from an intermediate solution based on D' . This process is essentially a linear transformation based on T.

However, this encoding relationship $D' = T \times D$ 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 Multiple Failure Handling

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

Another novel approach is to accelerate redundant sum reconstruction in the background—that is, 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 sum 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 applying multiple hot-replacements to compute intermediate results, followed by iterative transformations.

3.3.1 HPL Overview

[Figure 3: see original paper] HPL computation 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 uses Gaussian Elimination with Partial Pivoting to solve linear systems Ax=b. It first computes the LU decomposition of the n×(n+1) coefficient matrix [A|b] to obtain [A|b] = [[L,U] L-1b], then solves Ux = L-1b for x, where U is an upper triangular matrix and L is a lower triangular matrix, as shown in [Figure 3: see original paper]. [Figure 4: see original paper] shows the pseudocode for the HPL program using LU decomposition. For more information, see [6].

3.3.2 Fault Tolerance Design

We first consider single node failure. As described in §3.3.1, HPL solves linear systems Ax=b using Gaussian Elimination with Partial Pivoting. Assume process Pi fails during execution. After replacing the failed data with redundant data, the linear system becomes A' y = b. At program completion, we obtain intermediate solution y. Since Gaussian Elimination with Partial Pivoting involves only linear transformations, the relationship A' = AT 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:

$$x_j = \begin{cases} y_j - y_i, & j \neq i \\ y_i, & j = i \end{cases}$$

This shows that 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 Ti for the ith failure:

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

In actual implementation, due to HPL' s 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, since transformation matrix T is very 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 validate correctness, the next chapter details the implementation of ABFT hot-replacement in HPL.

4 Implementation

We have currently implemented a fault-tolerant HPL using ABFT hot-replacement for single node failure. The following implementation focuses on single node failure. 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; if node failure occurs outside this window, all HPL processes terminate.

4.1 Matrix Encoding

Matrix encoding uses a scheme similar to reference [4]. In HPL, random matrix A is distributed in block-cyclic pattern across a 2D P×Q process grid. To handle single node failure, we add P processes, forming a new (P+1)×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: see original paper] shows the 2D process grid and distribution of encoded data. For example, a 2×2 process grid [FIGURE:5(a)] becomes a new grid with redundant processes [FIGURE:5(b)].

Under 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

Failure timing is unpredictable, so we cannot know when or which node will fail before it occurs. 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 each 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 "live" processes enter a unified entry point to handle the failure using ABFT hot-replacement—that is, replacing the failed process column with the redundant process column to continue

computation. Using the process grid in [FIGURE:5(b)] as an example, if process P3 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 be completed quickly.

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

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

4.3 Background Recovery

After hot-replacement, the U matrix obtained after the final trailing matrix update is no longer upper triangular, making it difficult to solve 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. This has the added benefit 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 in subsequent update phases. For large recovery data volumes, we may need to split the 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.

4.4 Message Fault Handling

The above methods suffice for node failures during elimination and update phases. For the row broadcast phase, special handling is required. All six row broadcast methods provided by HPL are based on message forwarding. If a process fails during broadcast, processes that do not communicate directly with it will hang. For example, if process P1 forwards messages from process P0 to process P2 and P0 fails, P1 can detect P0's failure through the return error code of `MPI_Recv`, but P2 cannot obtain this information because its upstream node P1 remains "live." Therefore, for process failures during broadcast, we need a robust message broadcast mechanism satisfying two conditions:

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

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

[Figure 6: see original paper] Message transmission path in 2rinM mode

In 2rinM mode, the message transmission path is shown in [Figure 6: see original paper]. 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.

[Figure 7: see original paper] Tree diagram of message transmission in 2rinM mode (dashed lines indicate parent-child relationships)

Based on the temporal order of message transmission in [Figure 6: see original paper], we represent it as a tree in [Figure 7: see original paper]. We assign a parent node to each non-root node that must satisfy two conditions: (1) the parent node' s message transmission precedes the child node' s; (2) message transmissions between parent and child have no direct dependency. 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 also waits for the parent node' s retransmission. For a parent node, it receives return codes from child nodes and determines whether to retransmit messages. Using [Figure 7: see original paper] as an example, we assign parent nodes as shown, with dashed arrows from child to parent. This method guarantees that for single node failure, any non-root node failure does not affect successful message delivery.

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

5 Validation

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

1. What is the performance of the algorithm-level fault tolerance scheme?
2. What rounding errors does the algorithm-level fault tolerance scheme introduce?
3. How does failure timing affect performance?

The first two experiment groups were conducted on the Dawning 5000A platform using 16 nodes, each with four quad-core 2.2GHz AMD Opteron processors sharing 64GB memory, connected via Gigabit Ethernet. The third group was conducted on the "Chaolong" blade platform with 8 blades, each containing

10 Intel Xeon X5650 processors (960 cores total). Nodes within a blade are connected via InfiniBand, with two blades connected by a single InfiniBand link. Both platforms run Linux. The MPICH2 package used was MPICH2-trunk-r7834. All timing was measured using the `MPI_Wtime` function.

5.1 Performance Comparison of Algorithm-Level Fault Tolerance Schemes

Experimental configuration on D5000A platform
HPL execution time

The first experiment group compares 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, simulated by forcibly terminating a process.

In this group, the order of random matrix A and compute node usage are shown in . Matrix order is at the 10 level. HPL execution times for ABFT recovery, ABFT hot-replacement, and failure-free scenarios are shown in . Overheads of both schemes relative to failure-free execution are shown in [Figure 8: see original paper].

[Figure 8: see original paper] Overhead of different algorithm-level fault tolerance schemes

As shown in , 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 failure-free HPL execution time. Overhead in both schemes varies not only with the amount of data allocated per process but also with the mapping relationship between processes and processor cores.

5.2 Rounding Error Comparison of Algorithm-Level Fault Tolerance Schemes

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

$$\frac{\|Ax-b\|_\infty}{\|A\|_\infty\|x\|_\infty+\|b\|_\infty} \le \epsilon \cdot N$$

where N is matrix dimension and is machine precision.

This group uses the same data scale and node configuration as the first group (). [Figure 9: see original paper] shows HPL error check results for ABFT hot-replacement, ABFT recovery, and failure-free scenarios. The results show that

ABFT recovery introduces relatively small rounding errors, while ABFT hot-replacement introduces rounding errors roughly twice those of the failure-free case.

[Figure 9: see original paper] Error comparison among different fault tolerance methods

5.3 Impact of Failure Timing on Hot-Replacement Performance

The third experiment group evaluates how failure timing affects ABFT hot-replacement performance. Conducted on the "Chaolong-1" platform using 75 nodes (900 cores), we launched one process per core, forming a 30×30 process grid. Matrix order 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: see original paper]. 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 under ABFT hot-replacement will not exceed that of ABFT recovery.

6 Future Work

Future work has three potential directions:

The first direction is designing a background accelerated redundant sum reconstruction scheme to handle multiple node failures. The greatest challenge is overhead—reconstructing redundant sums involves substantial communication and requires compute node participation. We need efficient schemes to "free" compute nodes from reconstruction work as early as possible, adding nodes or network resources to accelerate background reconstruction and enabling redundant nodes to "catch up" quickly. Another issue is rounding error from the hot-replacement strategy. [Figure 11: see original paper] shows rounding error versus failure count. Experiments indicate 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 future work. Additionally, implementing multiple failures requires not only application-layer modifications but also new MPI support, such as dynamic communicator adjustment.

[Figure 11: see original paper] Relationship between rounding error and failure count

The second direction is applying the scheme to larger-scale supercomputers and implementing more complete failure handling. The third direction is extending the hot-replacement strategy to more high-performance computing applications.

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

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