IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 7, 2020, accepted February 18, 2020, date of publication February 24, 2020, date of current version March 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2975832

FT-PBLAS: PBLAS-Based Fault-Tolerant Linear Algebra Computation on High-performance Computing Systems

YANCHAO ZHU, YI LIU, AND GUOZHEN ZHANG
School of Computer Science and Engineering, Sino-German Joint Software Institute, Beihang University, Beijing 100191, China
Beijing Key Laboratory of Network Technology, Beihang University, Beijing 100191, China

Corresponding author: Yanchao Zhu (zyc0627cool@gamil.com)

This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB0200100, and in part by the Natural Science Foundation of China under Grant 91530324.

ABSTRACT As high-performance computing (HPC) systems have scaled up, resilience has become a great challenge. To guarantee resilience, various kinds of hardware and software techniques have been proposed. However, among popular software fault-tolerant techniques, both the checkpoint-restart approach and the replication technique face challenges of scalability in the era of peta- and exa-scale systems due to their numerous processes. In this situation, algorithm-based approaches, or algorithm-based fault tolerance (ABFT) mechanisms, have become attractive because they are efficient and lightweight. Although the ABFT technique is algorithm-dependent, it is possible to implement it at a low level (e.g., in libraries for basic numerical algorithms) and make it application-independent. However, previous ABFT approaches have mainly aimed at achieving fault tolerance in integrated circuits (ICs) or at the architecture level and are therefore not suitable for HPC systems; e.g., they use checksums of rows and columns of matrices rather than checksums of blocks to detect errors. Furthermore, they cannot deal with errors caused by node failure, which are common in current HPC systems. To solve these problems, this paper proposes FT-PBLAS, a PBLAS-based library for fault-tolerant parallel linear algebra computations that can be regarded as a fault-tolerant version of the parallel basic linear algebra subprograms (PBLAS), because it provides a series of fault-tolerant versions of interfaces in PBLAS. To support the underlying error detection and recovery mechanisms in the library, we propose a block-checksum approach for non-fatal errors and a scheme for addressing node failure, respectively. We evaluate two fault-tolerant mechanisms and FT-PBLAS on HPC systems, and the experimental results demonstrate the performance of our library.

INDEX TERMS Algorithm-based fault tolerance, HPC systems, node failure, matrix multiplication, linear algebra computations.

I. INTRODUCTION

With the scaling up of high performance computing (HPC) systems in recent years, resilience has become a major challenge. Currently, supercomputers generally have tens of thousands of processors; e.g., the number of cores in Summit [1] and Sunway TaihuLight [2] is 2,414,592 and 10,649,600, respectively. As the number of hardware components increases, failures occur more frequently on average. Statistics show that the MTBF, the mean time between failures, of the currently most powerful supercomputers has been reduced to several hours. This situation will become worse in the future due to the foreseeable increase of the number of processors and nodes in HPC systems.

To ensure reliable execution of applications in HPC systems, various kinds of fault-tolerant techniques have been proposed, which can be simply classified into hardware and software approaches. Among software approaches, the checkpoint-restart method [5]–[8] is the most popular one, and it is extensively used in current HPC systems. However, the checkpoint-restart technique faces challenges in current massive parallel systems, especially to its scalability;

The associate editor coordinating the review of this manuscript and approving it for publication was Utku Kose.

e.g., at the checkpoint time, all the nodes need to be synchronized for some particular applications, and the volume of the checkpoint data also impacts the I/O infrastructure. Another extensively used software-based technique is replication [9], [10], which uses replication in different levels (e.g., the process-level) to ensure the reliable execution of applications and consequently consumes a large amount of resources. Apart from the above application-independent software-based techniques, algorithm-based approaches, i.e., algorithm-based fault tolerance (ABFT), provide another way to achieve the correct execution of applications. ABFT ensures the reliability of basic algorithms through mathematical principles, which help with fault tolerance through data validation, error detection and data recovery; then, the higher-level applications receive indirect fault-tolerance support through these fault-tolerant basic algorithms. Although the ABFT approach is algorithm-dependent, its efficiency and lightweight characteristics make it attractive in the era of peta-scale and exa-scale systems.

Many engineering and scientific applications rely on some fundamental algorithms, such as matrix computation and fast Fourier transform (FFT). Therefore, it is possible to provide algorithm-based fault-tolerant mechanisms at a low level and to make these mechanisms application-independent. However, previous ABFT approaches face two challenges when they are used in HPC systems. First, most previous approaches aim at fault-tolerance in integrated circuits (ICs) or at the architecture level rather than in massive parallel systems; e.g., the first ABFT approach for matrix multiplication [11] uses checksums of rows and columns to detect and recover from errors, which is suitable for IC chips. However, matrix computations in HPC systems are usually block-based, that is, matrices are partitioned into blocks and assigned to different processes. The second challenge is fatal errors caused by node failure. Node failure occurs frequently in current HPC systems and often causes nodes to crash, which renders traditional checksum-based systems ineffective.

This paper proposes a fault-tolerant library for linear algebra computations called FT-PBLAS, which can be regarded as the fault-tolerant version of PBLAS. We propose a block-based approach and a lightweight scheme to deal with node failure, and we incorporate them into the extensively used PBLAS library. The main contributions of this work include the following:

- To the best of our knowledge, FT-PBLAS is the first library that supports fault-tolerant linear algebra computations in HPC systems. The library provides a series of fault-tolerant versions of interfaces in PBLAS and supports underlying error detection and recovery for both non-fatal errors and node failure.
- We propose a block-checksum-based approach for fault-tolerant matrix computations in HPC systems. The approach uses a block checksum instead of a traditional row-column checksum to detect computation errors, which is not only more suitable for HPC systems but also better in terms of computational complexity.
- We propose a fault-tolerant matrix computation scheme to deal with errors caused by node failure. The scheme ensures the execution and completion of large iteration-based matrix algorithms such as Cannon 12], even in node failure cases.

The rest of this paper is organized as follows: Section II introduces the fault models of our work. Section III introduces our proposed block-checksum approach. Section IV presents our lightweight method for addressing node failure. Section V presents the implementation based on PBLAS. Section VI evaluates the proposed method and presents the experimental results. Section VII presents related work and section VIII concludes the paper.

II. FAULT MODEL

We propose our fault-tolerant mechanisms under two different fault models. As these fault models produce different results, the corresponding fault-tolerant mechanisms are discussed separately.

Most previous ABFT methods focus on the correctness of the calculation results, which are affected by silent errors during computations, e.g., bit inversion caused by cosmic rays. This kind of error does not impact the execution of an application, but it can lead to a wrong calculation result. Our block-checksum approach, which is discussed in next section, aims at solving this kind of error in HPC systems.

The second fault model is a typical fail-stop model, which is caused by non-silent errors, i.e., crash cases. Under this model, an application cannot be executed completely if the error occurs. Section IV presents a fault-tolerant mechanism for node failure, which helps ensure the execution of a matrix computation even if this kind of error occurs.

III. THE BLOCK-CHECKSUM APPROACH

As one kind of typical and complex linear algebra computation, matrix multiplication is a focus for many researchers. Previous studies of fault-tolerant matrix multiplication use checksums for each row and column of a matrix to check computational errors, which is not suitable for current HPC systems. The traditional row-column method detects error through comparing all the checksums before and after the matrix multiplication; then, it re-calculates the corresponding data, the coordinates of which can also be found by comparing checksums. However, with the scaling up of matrices in HPC systems, this fine-grained fault-tolerant method will have faster growth of redundant data, which certainly impacts the efficiency of the whole matrix multiplication.

Matrix computations generally divide matrices into pieces, instead of rows or columns, and allocate these pieces to different nodes (processes) for parallel computation. In this situation, we propose a block-checksum approach, which uses a checksum of blocks instead of rows and columns to check for computational errors in HPC systems [14]. This approach reduces the number of redundant operations,

i.e., calculations and comparisons of checksums, because it detects and recovers errors in blocks instead of individual data values. Although the approach needs more overhead to recover data, it is more suitable for improving the performance of HPC systems overall, as the recovery overhead is manageable and can be negligible compared to the overall number of matrix multiplication operations. In addition, our approach also considers the procedure and algorithm of matrix multiplication in PBLAS, in which each process performs several steps of multiplication and addition, and some communications among processes occur between those steps. To minimize the impact on overall computation, our approach focuses on each multiplication step in each process so that error correction can be accomplished in processes without communication.

A. PRINCIPLE AND PROCEDURE

We take matrix multiplication as an example to present the principle of our block-checksum approach.

As mentioned above, the block-checksum approach is designed for block-based matrix computations. We first set k as the size of each block (the method of setting the k value will be introduced in the following sections), and then the sum of all elements in each block will be the checksum to detect the occurrence of errors during the calculation. When errors are detected, the corresponding process will re-calculate each block with an error.

Figure 1 shows the principle of our approach, where matrix A and B are multiplied with a checksum calculation and error check. First, the two matrices are divided into pieces, with each piece consisting of a block and its checksum. Our algorithm will detect possible errors in blocks by these block-checksums. The gray part in Figure 1 indicates the checksum of the corresponding block. All of the block checksums will be encoded into checksum matrices before computation and then be used for calculating the checksum result. When the multiplication is complete, the final result is encoded again to compare with the checksum result. Under certain criteria, our approach can detect an error and re-calculate the error block or finish the entire calculation if no error occurred.

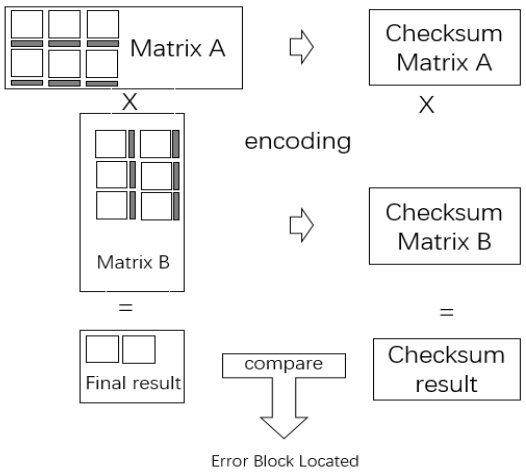


FIGURE 1. Block-checksum approach for matrix multiplication.

According to our scheme, to accomplish the matrix multiplication $A \times B$, we first need to encode matrix A and matrix B to generate the checksum. When the block has been divided, the matrix A is represented as:

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,k} & a_{1,k+1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{k,1} & \cdots & a_{k,k} & a_{k,k+1} & \cdots & a_{k,n} \\ a_{k+1,1} & \cdots & a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,k} & a_{m,k+1} & \ldots & a_{m,n} \end{bmatrix}$$

For matrix A, we first divide the matrix into w blocks, with each block consisting of k rows, then calculate the sum of each column in each block and put the checksum into the checksum matrix A_{check}. That is, each row of the checksum matrix A_{check} corresponds to one block of matrix A. We also add an extra row (the last row) in A_{check} in which each element is the sum of the corresponding column.

$$A_{check} = \begin{bmatrix} ac_{1,1} & \cdots & ac_{1,n} \\ \vdots & \ddots & \vdots \\ ac_{w,1} & \cdots & ac_{w,n} \\ ac_{w+1,1} & \ldots & ac_{w+1,n} \end{bmatrix} \text{ and } \mathrm{w} = \left\lceil \frac{m}{k} \right\rceil \quad (1)$$

$$ac_{i,j} = \sum_{p=i*k-k+1}^{i*k+1} a_{p,j}\,(i \le w)\; ac_{i,j} = \sum_{p=1}^{m} a_{p,j}\,(i > w) \quad (2)$$

Matrix B is processed in the same way:

$$B = \begin{bmatrix} b_{1,1} & \cdots & b_{1,k} & b_{1,k+1} & \cdots & b_{1,q} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & \cdots & b_{k,k} & b_{k,k+1} & \cdots & b_{k,q} \\ b_{k+1,1} & \cdots & b_{k+1,k} & b_{k+1,k+1} & \cdots & b_{k+1,q} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,k} & b_{n,k+1} & \ldots & b_{n,q} \end{bmatrix}.$$

The checksum matrix for matrix B is encoded as follows:

$$B_{\text{check}} = \begin{bmatrix} bc_{1,1} & \ldots & bc_{1,t} & bc_{1,t+1} \\ \vdots & \ddots & \vdots & \vdots \\ bc_{n,1} & \ldots & bc_{n,t} & bc_{n,t+1} \end{bmatrix} \text{ And } \mathrm{t} = \left\lceil \frac{q}{k} \right\rceil \quad (3)$$

$$bc_{i,j} = \sum_{p=i*k-k+1}^{i*k+1} a_{i,p}\,(j \le t)\; ac_{i,j} = \sum_{p=1}^{q} a_{i,p}\,(j > t)\,. \quad (4)$$

The result of matrix multiplication is $C{=}A{\times}B$.

$$C = \begin{bmatrix} c_{1,1} & \cdots & c_{1,q} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \cdots & c_{m,n} \end{bmatrix}$$

The result checksum matrices are calculated as:

$$C_{rc} = [cr_1, \ldots cr_w], cr_i = \sum_{j=1}^{n} ac_{i,j} \times bc_{j,t+1} \quad (5)$$

$$C_{cc} = [cc_1, \ldots cc_t], cc_i = \sum_{j=1}^{n} ac_{w+1,j} \times bc_{j,i} \quad (6)$$

C_{rc} means the sums of the data in the same block row calculated by block checksums, and C_{cc} means the sums of the data in the same block column. They help locate the coordinates of the error block through the following comparisons if an error occurs.

To check if there is an error during the multiplication, we need to calculate the checksums for the matrix C and compare them with the data in the result checksum matrix (i.e., C_{rc} and C_{cc}). An error block will be located at the intersection of mismatching block checksums. The checksums of matrix C are calculated as follows:

$$cr_i^* = \sum_{j=i*k-k+1}^{i*k+1} \sum_{h=1}^{q} c_{j,h}, \text{ and } cc_i^* = \sum_{j=i*k-k+1}^{i*k+1} \sum_{h=1}^{m} c_{h,j} \quad (7)$$

Error detection is performed by comparing the above newly calculated checksums with the elements in the result checksum matrix. One question that should be considered is how to distinguish floating-point rounding errors from computational errors. To solve this problem, we set an appropriate error bound $\mathcal{E}$ (which is discussed in the next sub-section). The final formula for error detection is as follows:

$$\left|cc_i^* - cc_i\right| < \varepsilon_i, \text{ and } \left|cc_i^* - cc_i\right| < \varepsilon_i \quad (8)$$

Proof: From the matrix multiplication theorem, we know that

$$c_{i,j} = \sum_{u=1}^{n} a_{u,j} * b_{u,j} \quad (9)$$

For any row block checksum, we have:

$$\begin{aligned} cr_i^* &= \sum_{j=i*k-k+1}^{i*k+1} \sum_{h=1}^{q} c_{j,h} = \sum_{j=i*k-k+1}^{i*k+1} \sum_{h=1}^{q} \sum_{u=1}^{n} a_{j,u} * b_{u,h} \\ &= \sum_{u=1}^{n} \sum_{i*k-k+1}^{i*k+1} \sum_{h=1}^{q} a_{j,u} * b_{u,h} \\ &= \sum_{u=1}^{n} \sum_{i*k-k+1}^{i*k+1} a_{j,u} * \sum_{h=1}^{q} b_{u,h} \end{aligned} \quad (10)$$

From equations (9) and (10), the block checksum is

$$\begin{aligned} cr_i^* &= \sum_{u=1}^{n} \sum_{j=i*k-k+1}^{i*k+1} a_{j,u} * bc_{u,t+1} \\ &= \sum_{u=1}^{n} bc_{u,t+1} * \sum_{j=i*k-k+1}^{i*k+1} a_{j,u} = \sum_{u=1}^{n} bc_{u,t+1} * ac_{i,u} = cr_i \end{aligned} \quad (11)$$

Algorithm 1 Kernel for Encoding the Checksum of Matrix A

Input:A[m][n], k
Output: AC[w+1][n]

$$w \leftarrow \frac{m}{k}$$

for i = 0 → n do
  p, q, asum, bsum=0;
  /*Initialize: *asum* stores all data sums, while *bsum* stores the sum of the data in a block. *p* and *q* are used to judge whether the calculation is still in the block.
  */
  for j = o → m
    if p<k
      bsum = + A[j][i];
      p++;
    else
      AC[q][i] = bsum;
      asum = + bsum;
      bsum, p=0;
      q++;
  /*judge whether the sum for the corresponding block has been calculated. Store the sum and do the same with the rest of the data.
  */
 end for
end for
Output AC.

The algorithm for the encoding procedure is given in Algorithm 1:

As soon as an error is detected, the algorithm will enter the recovery procedure. After locating the intersection of the mismatching block checksum, the algorithm will re-calculate all the data in the block, as the block checksum cannot give the exact coordinate where the error occurred. After the re-calculation, the application will check the block checksum again to ensure correctness.

B. BLOCK SIZE AND ROUNDING ERROR

Many studies have given a variety of methods to distinguish rounding errors and soft errors, as this is an inescapable problem in floating-point calculations.

In our block-checksum approach, there are two factors that need to be determined: the block size k and the rounding error bound $\mathcal{E}$. Generally, the larger the block we select is, the smaller the time and space overhead and the lower the accuracy; the opposite is also true. That is, block size is inversely correlated with accuracy due to the effect of error-counteraction; e.g., if there are two errors in one block, one result is larger than the accurate value and the other is smaller. Therefore, when the two values are added, the errors will counteract each other in producing the checksum, and errors may not be detected. To achieve a balance between accuracy and efficiency, we propose an approach that combines our

simplified method for block-size calculation with a classic method for rounding-error-bound calculation.

Similar to other works [15], [16], Our approach uses the reciprocal distribution of mantissa bits to calculate the rounding error bound for a block checksum. The principle of the method is based on the fact that matrix multiplication consists of multiple steps of multiplication and addition, and the rounding error bound can be obtained by calculating expectation and variance during those steps.

In arbitrary data sets, the mantissas of floating-point numbers tend to follow a reciprocal distribution, which, in the case where the base $B = 2$, is represented as:

$$\mathrm{r}(\mathrm{x}) = \frac{1}{x \cdot \ln(2)}, \quad x \in [\frac{1}{2}, 1) \tag{12}$$

where $r(x)$ is the probability that the floating-point mantissa is x. After several calculation steps, we calculate the variance of a single result.

$$\begin{aligned} \sigma\ (\mathrm{S}) &= \sqrt{Var_{Prod}\ (S) + Var_{sum}(S)} \\ &\leq \sqrt{\frac{n \cdot (n+1) \cdot (2n+1) + 2n}{24}} \cdot y.2^{-t} \end{aligned} \tag{13}$$

In the above formula, n refers to the size of the matrix multiplied; y refers to the largest absolute value among the data in matrix A and matrix B; and t depends on the bits of floating-point in the hardware. In our block-checksum approach, the block checksum is the summation of $k \times k$ data points. After this addition, we obtain the variance of the block checksum:

$$\sigma\ (\mathrm{RS}) \leq k \cdot m \cdot \sqrt{\frac{n \cdot (n+1) \cdot (2n+1)}{24}} \cdot y.2^{-t}$$

and

$$\sigma\ (\mathrm{CS}) \leq k \cdot q \cdot \sqrt{\frac{n \cdot (n+1) \cdot (2n+1)}{24}} \cdot y.2^{-t} \tag{14}$$

where k refers to the size of the block we used for the block checksum m refers to the number of rows in matrix A, and q refers to the number of columns in matrix B. Formula 14 can be used to determine the value of k. Then, there is the formula for the bound of the rounding error, which is based on principles of numerical algorithms [17]:

$$\varepsilon \leq \gamma_n |\,|A|\,|_\infty |\,|B|\,|_\infty, \tag{15}$$

where $\gamma_n = \frac{nu}{1-nu}$,u is the unit roundoff error of the target machine, and n is the common dimension of the matrix.

As a result, the value of rounding error bound calculated through reciprocal distribution is much smaller than that calculated through the other methods. By solving the formula below, we can get a suitable value of the block size k.

$$\begin{aligned} \sigma\ (\mathrm{CS}) &\leq k \cdot q \cdot \sqrt{\frac{n \cdot (n+1) \cdot (2n+1)}{24}} \cdot y.2^{-t} \\ &\leq \gamma_n |\,|A|\,|_\infty |\,|B|\,|_\infty \end{aligned} \tag{16}$$

In the above formula, some necessary data, e.g., y, needs to be determined during the calculation. However, to determine the value of k in fewer steps, we employ a simplified method: before multiplication starts, the process extracts data from some of the input matrices randomly, calculates the needed values of these data rather than all data and finally obtains the recommended range for k. By simplifying the calculation of k before the matrix multiplication or offline, our method causes little performance loss. However, since the value of k is determined by only two methods for error threshold judgment, it cannot be guaranteed to be optimal. Therefore, we can also set a default value for k if necessary.

C. OVERHEAD ANALYSIS

A fault-tolerant mechanism involves extra computations, which influence both performance and scalability. This sub-section analyzes the computational complexity of our block-checksum approach and compares it with that of the previous ABFT [11] approach based on the row-column checksum.

In the encoding procedure, both our approach and ABFT need to traverse the whole matrix to calculate the checksum, which corresponds to an overhead of$O(N^2)$, where N refers to the scale of the matrices for multiplication. In the computing procedure and error-detecting procedure, the computational complexity of our approach is $O(N^2)$ and $O(N)$, respectively, which is only $1/k$of that of ABFT. However, the error location of our method is a coordinate of a block which includes k^2 data, instead of a coordinate of a single data. Although it take some more time to recover the computational error than the ABFT, our block-checksum approach has little impact on the accuracy of error detection. As a result, the overhead for recovery is of complexity $O(N)$ for both ABFT and our approach.

In conclusion, the overall complexity of our approach is $O(N^2)$. Considering that the complexity of matrix multiplication is $O(N^3)$, our approach imposes only a limited performance impact on overall computation, especially when the matrices become larger, as in HPC systems. Compared to the previous row-column-checksum approach, our approach has fewer operations in its computing and detecting procedures.

For example, consider conducting a 128000×1280000 matrix multiplication on a cluster with 256 nodes, each of which runs 16 processes. There will be a 20000×20000 submatrix multiplication for each process. The number of fault-tolerant operations is shown in Table 1.

IV. FAULT-TOLERANT MATRIX COMPUTATION FOR NODE FAILURE

A. PROBLEM ANALYSIS

The block-checksum approach can ensure the detection and recovery of computational errors caused by non-fatal errors (e.g., soft errors) in HPC systems; however, it cannot deal with errors caused by node failure. Node failure often implies a crash or a suspension of the node system, and it occurs frequently in current HPC systems due to their large number of processors and nodes. In this situation, all of the processes

TABLE 1. Number of fault-tolerant operations (128000 × 128000 matrix multiplication, 256 nodes, 16 processes/node).

Method	Row-col checksum			Block checksum (block size = 20)		
Operations	Additions	Multiplications	Comparisons	Additions	Multiplications	Comparisons
Encoding	400 million			400 million		
Matrix mult	800 million	800 million		40 million	40 million	
Detecting	400 million		40000	400 million		2000

in the failure node stop running, and neither a computation result nor a checksum can be returned for error-detection.

The situation may be more complex for practical matrix algorithms such as the Cannon algorithm [12], which is a classic and extensively used algorithm for large matrix multiplication, To solve the problem of insufficient memory for storing the blocks of a large matrix, the algorithm divides the matrix into smaller blocks and performs matrix multiplication through multiple iterations on those blocks. During each iteration, blocks are computed and exchanged among processes. If a node failure occurs during the execution of the algorithm, not only is the result of the failed node lost but also the data blocks that are essential for subsequent computations are lost; as a result, the process of the algorithm cannot continue.

To address the above problem, we propose an error detection and recovery scheme on the basis of the Cannon algorithm to deal with errors caused by node failure.

B. FAULT-TOLERANT SCHEME FOR NODE FAILURE

Our fault-tolerant scheme for node failure consists of two parts: error detection and error recovery. Error detection is performed along with computation, and we use a two-dimensional state table (as shown in Figure 2) to trace the status of the processes that compute over the blocks of the matrix. The state table is organized by the column and row index of blocks corresponding to the partitions of the matrix, in which the element *(i, j)* indicates the status of the process that compute over *block(i, j)*: 1 means the process is alive (i.e., working normally), and 0 means the process is down (i.e., there is a node failure).

Process state / Column index / Row index	1	2	...	n
1	1	1	...	1
...	...	...	...	...
m	1	1	...	1

FIGURE 2. Two-dimensional state table.

The procedure of this fault-tolerant mechanism for each iteration is described as follows:

(a) The Algorithm starts. Initialize the state table by setting the status of each process to 1 (normal) and the MPI environment; each process obtains the appropriate data blocks according to the Cannon algorithm.

(b) Perform multiplication and addition. Judge whether the iteration is complete. If it is complete, go to step d; if not, then each process obtains state information from the state table. If the data is 1, the process starts to obtain the data for the next iteration from the corresponding process and then goes to step c. If the process state information is 0 (meaning the process cannot execute normally), the process acquires data from the storage system, then iterates step b.

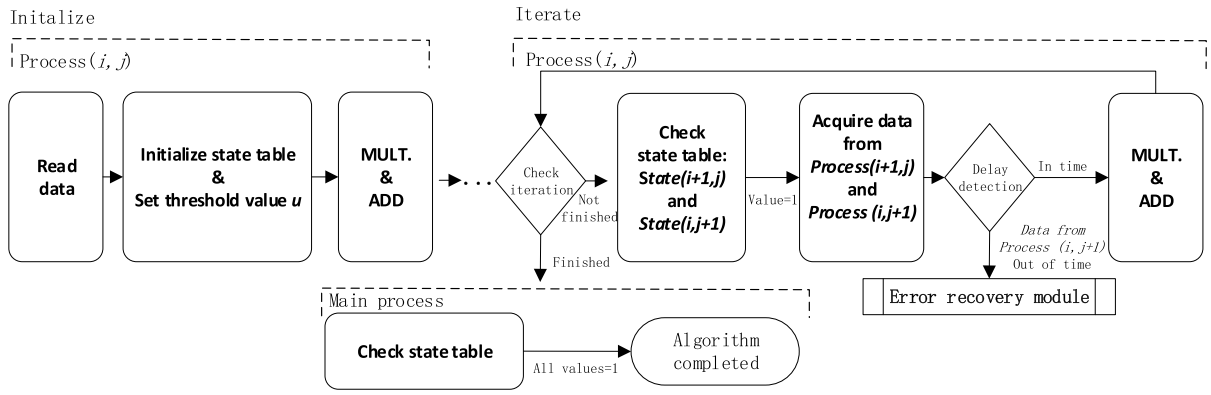
(c) Compare the time with a preset threshold. If the process completes the data transfer within the threshold time, then go to step b. If not, consider the node of the corresponding process to be down; then, update the information of this process in the state table and broadcast this information to the other processes. Acquire data from the storage system and go to step b.

(d) Finish the work of the current process. The main process starts checking the state table. If all the data are 1, the algorithm ends. If any processes are found with state information 0, the main process re-assigns the tasks of these processes to other processes that execute the program correctly. After all the results are calculated, the algorithm ends.

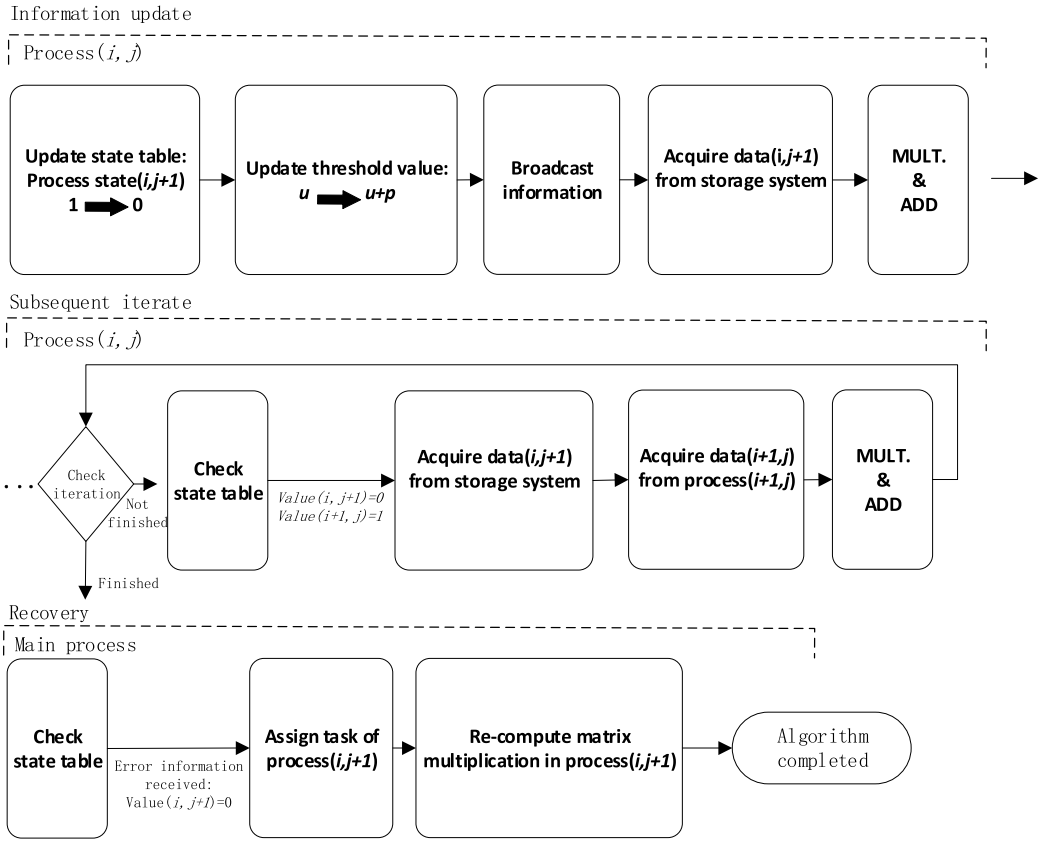
To be more specific, we use *process(i, j)* to represent the process that computes over *block(i, j)* of the matrix. Figure 3 illustrates our algorithm with an example of *process(i, j)*. Part A in the figure shows the whole procedure of our fault-tolerant scheme for node failure. The algorithm starts with an initialization step, and a timeout threshold u is used to judge the node failure error. During each iteration, the process exchanges data with neighboring processes, and any timeout of data transfer will be regarded as node failure, which will trigger the error recovery shown in part B. In the figure, we also give an error example in which the data transfer from *process*$(i, j+1)$ runs out of time; then, the process will enter the error recovery module as shown in part B.

To start the error recovery, the process updates the state table to mark the status of the failed process (i.e., it sets their values to 0), and the timeout threshold is also updated to avoid misjudging failures during error recovery; after that, the process broadcasts the information to the other processes and reloads the lost data from the storage system directly, then finishes the current iteration. During the subsequent iterations, the process obtains data from the storage system, as the value of *process*$(i, j+1)$ in the state table is 0.

After all of the iterations are completed, the main process checks the state table. If all of the values in the table are 1, the algorithm will finish immediately because no failure occurred. However, for this example, the main process will find that the process *state*$(i, j+1)$ has the value 0, so it



A. Procedures in process (i, j) of the fault-tolerant algorithm for node failure



B. Error recovery module of process (i, j)

FIGURE 3. Fault-tolerant matrix multiplication for node failure.

re-assigns the task of *process*$(i, j+1)$ to the processes that execute it correctly. After the re-computing of the corresponding block completes normally, we finally obtain all the results and finish the whole procedure.

From the above description, we can see that the error detection incurs only a little overhead; on the other hand, the error recovery takes a long time to finish the whole calculation. The overhead of error recovery mainly comes from two sources: first, we have to wait for a certain amount of time to check whether the relevant process is down, as it may be affected by network fluctuations, and we cannot get hardware information from the platform system, as this scheme is deployed at the application level. The updating and broadcasting of the information of failed processes also take some time. Second, some time is also needed to re-calculate the tasks of failed nodes.

It should be noted that the state table of the scheme is useful for achieving fault tolerance in upper-level applications. For example, matrix decomposition needs many instances of matrix multiplication, but our scheme ensures that each node failure is detected only once, e.g., through the repeated writing and reading of the state table in storage system between multiplications, which avoids wasting time with the repeated detection of node failure in multiple matrix multiplications.

This approach enables HPC matrix multiplication to continue to execute when node failure error occurs, and compared to the overall matrix operation, it only takes a little additional overhead when no node failure occurs. This fault-tolerant matrix computation method for addressing node failure achieves a good balance between functionality and performance.

V. IMPLEMENTATION BASED ON PBLAS

We implemented our block-checksum-based approach and the scheme for node failure on the basis of PBLAS. PBLAS is the cluster-version of a popular linear algebra computation library, BLAS, and it is used extensively in HPC systems and other software packages such as ScaLAPACK [4]. To guarantee ease of use and compatibility, our library provides fault-tolerant version interfaces corresponding to the original interfaces of PBLAS.

Similar to BLAS, PBLAS also supports three levels of linear algebra operations: vector-vector (level-1), matrix-vector (level-2) and matrix-matrix (level-3). Among those operations, matrix multiplication is the most complex operation and has the highest computational complexity. Therefore, we focus on the implementation and interface of level 3 (matrix-matrix) operations in the following section. Considering that the default MPI error handling mechanism for node failure contradicts our mechanism, the relevant measures are also presented in part C.

A. ARCHITECTURE OF THE LIBRARY

Figure 4 shows the architecture of FT-PBLAS. Both the block-checksum mechanism and the scheme for node failure are implemented in the underlying programs, and by providing a fault-tolerant version of interfaces (introduced in the next sub-section), HPC applications can invoke this library to accomplish transparent fault-tolerant vector and matrix computations.

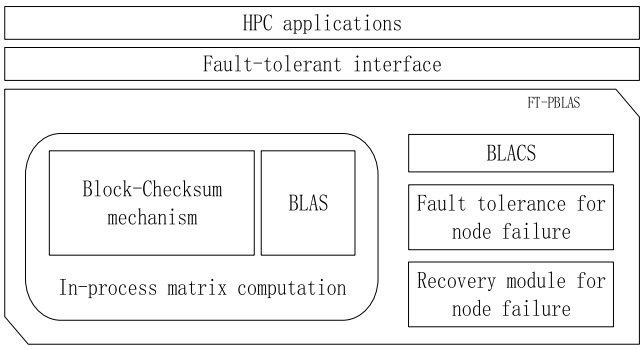


FIGURE 4. Architecture of the FT-PBLAS.

Considering that PBLAS is built on BLAS and the communication interface BLACS (basic linear algebra communication subprograms), we still utilize these functions to implement fault-tolerant mechanisms. The block-checksum mechanism is combined with the original BLAS, and fault-tolerant interfaces are incorporated by adding parameters and renaming methods. At the same time, we utilize the communication functions of BLACS to broadcast and update the information of the state table among processes.

B. INTERFACE OF THE LIBRARY

As mentioned in previous sections, FT-PBLAS provides a series of fault-tolerant versions of interfaces corresponding to the original interfaces of PBLAS. To distinguish the fault-tolerant interfaces from the original ones, we add the prefix "*FT_*" to all of the fault-tolerant interfaces. In addition, some new parameters are added to specify parameters for error detection and recovery; e.g., the parameter *BS* is used to specify the block size, but this can also be set to a default value by using our built-in automatic block selection mechanism.

Based on the operation of matrix multiplication introduced previously, the vector-vector and matrix-vector operations can be regarded as matrix operations with a lower latitude and complexity. Therefore, we mainly discuss the fault-tolerant interfaces in Level 3, as listed in Table 2.

In addition to the fault-tolerant versions of the original interfaces, we added three new interfaces to facilitate the node failure approach and cooperate with high-level applications, as shown in Table 3.

C. WORK WITH MPI ENVIRONMENT

In order to ensure the completion of the application and its efficiency with MPI environment, MPI will monitor all the processes during the execution of program. When some processes fail, caused by node failure or other reasons during the execution of the program, MPI will handle the corresponding error according to the preset mechanism when the error is monitored. Generally MPI terminates the executing program as the default predefined error handle is MPI_ERROR_ARE_FATAL, which means to bring down the whole computation when error occurs [18]. Considering that a failed MPI process usually leads to the failure of the whole parallel program due to some information which will never been received from the failed process, to bring down the whole program is the safest and most efficient measure while facing the node failure error.

However, even though our proposed fault-tolerant mechanism could deal with the node failure during the execution of the matrix multiplication, the current MPI error handling mechanism will hinder this mechanism which uses MPI. To ensure our mechanism without hindrance, a function, i.e., MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler) [18], which is used to set the error handle for a specific communicator, will be called during the initialization

TABLE 2. Level 3 interfaces of FT-PBLAS and PBLAS.

Level 3 PBLAS interface	FT-PBLAS interface	operations
P*GEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC)	FT_P*GEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC, BS)	$C \leftarrow \alpha A \times B + \beta C$ A, B: common matrix
P*SYMM(SIDE, UPLO, M, N, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC)	FT_P*SYMM(SIDE, UPLO, M, N, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC, BS)	$C \leftarrow \alpha A \times B + \beta C$ A, B: symmetric matrix
P*HEMM(SIDE, UPLO, M, N, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC)	FT_P*HEMM(SIDE, UPLO, M, N, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC, BS)	$C \leftarrow \alpha A \times B + \beta C$ A, B: Hermitian matrix
P*SYRK(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, BETA, C, IC, JC, DESCC)	FT_P*SYRK(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, BETA, C, IC, JC, DESCC, BS)	$C \leftarrow \alpha A \times A^T + \beta C$
P*HERK(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, BETA, C, IC, JC, DESCC)	FT_P*HERK(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, BETA, C, IC, JC, DESCC, BS)	$C \leftarrow \alpha A \times A^H + \beta C$
P*SYR2K(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC)	FT_P*SYR2K(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC, BS)	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
P*HER2K(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC)	FT_P*HER2K(UPLO, TRANS, N, K, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BETA, C, IC, JC, DESCC, BS)	$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C$
P*TRSM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB)	FT_P*TRSM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, IA, JA, DESCA, B, IB, JB, DESCB, BS)	$B \leftarrow \alpha A^{-1} B$

TABLE 3. Interface for supporting the state table.

Interface for state table	Operations
CHECKPROCESS (*numA,*numB)	Query the status of the process that computes the corresponding block
CHECKERROR()	Returns a list of the block numbers of processes in which an error has occurred
SETTABLEVALUE (*numA,*numb, VALUE)	Update the state table

TABLE 4. Experimental environment.

Item	Configurations
Node machine	Intel Xeon E5 processor * 2 64 GB memory
Operating system	Red Hat 4.4.7-4
PBLAS version BLACS version	ScaLAPACK 2.0.2
Network	MPICH-3.1.3

of the MPI environment. Another predefined error handle MPI_ERROR_RETURN, which means only to return the error code while error occurs, will be set for the communicator of our matrix multiplication, so that the execution of our algorithm will not be terminated by node failure error. As a result, our fault-tolerant matrix multiplication for node failure could work with MPI environment normally simply by setting a new error handle.

Previous researches have proposed some mechanisms with MPI for node failures, there are two main methods: global restart [19] and local recovery via a ULFM interface [20]. Although these methods could restore the MPI environment efficiency, the applications must be re-executed. Even though some necessary fault-tolerant measures are used to reduce the performance overhead of the re-execution (e.g., FT-MPI [38]), it is still less efficient compared to our fault-tolerant mechanism.

VI. EVALUATION

A. EXPERIMENT STEP

We evaluate our FT-PBLAS on the Tianhe-2 system [31], major configurations of which are listed in Table 4. The number of computing nodes used in our experiments ranges from 4 to 128, with each node assigned 32 processes (i.e., 1 process/processor core). Therefore, the number of processes ranges from 128 to 4096.

Our experiments mainly evaluate the overhead of the two fault-tolerant approaches and the FT-PBLAS library. In principle, the overhead can be investigated according to two aspects: the overhead when no error occurs, and the overhead when there is error (i.e., the overhead of error recovery). Accordingly, our experiments are composed of two parts—experiments without error and experiments with error—both of which use the computation time of matrix multiplication as the metric. That is, the metric is the execution time of the function *PDGEMM()* and its fault-tolerant version *FT_PDGEMM()* in PBLAS and FT-PBLAS, respectively. in the function calls to *PDGEMM()* and *FT_PDGEMM()*, we set the parameters alpha and beta to 1 and 0, respectively, which transforms the function into a simple matrix multiplication. we use a simple program to pre-load the data and invoke these two interfaces to test performance.

B. PERFORMANCE AND OVERHEAD WITHOUT ERROR

1) BLOCK-CHECKSUM APPROACH

To better demonstrate the performance of the two methods of our fault-tolerant mechanism, we test them one by one.

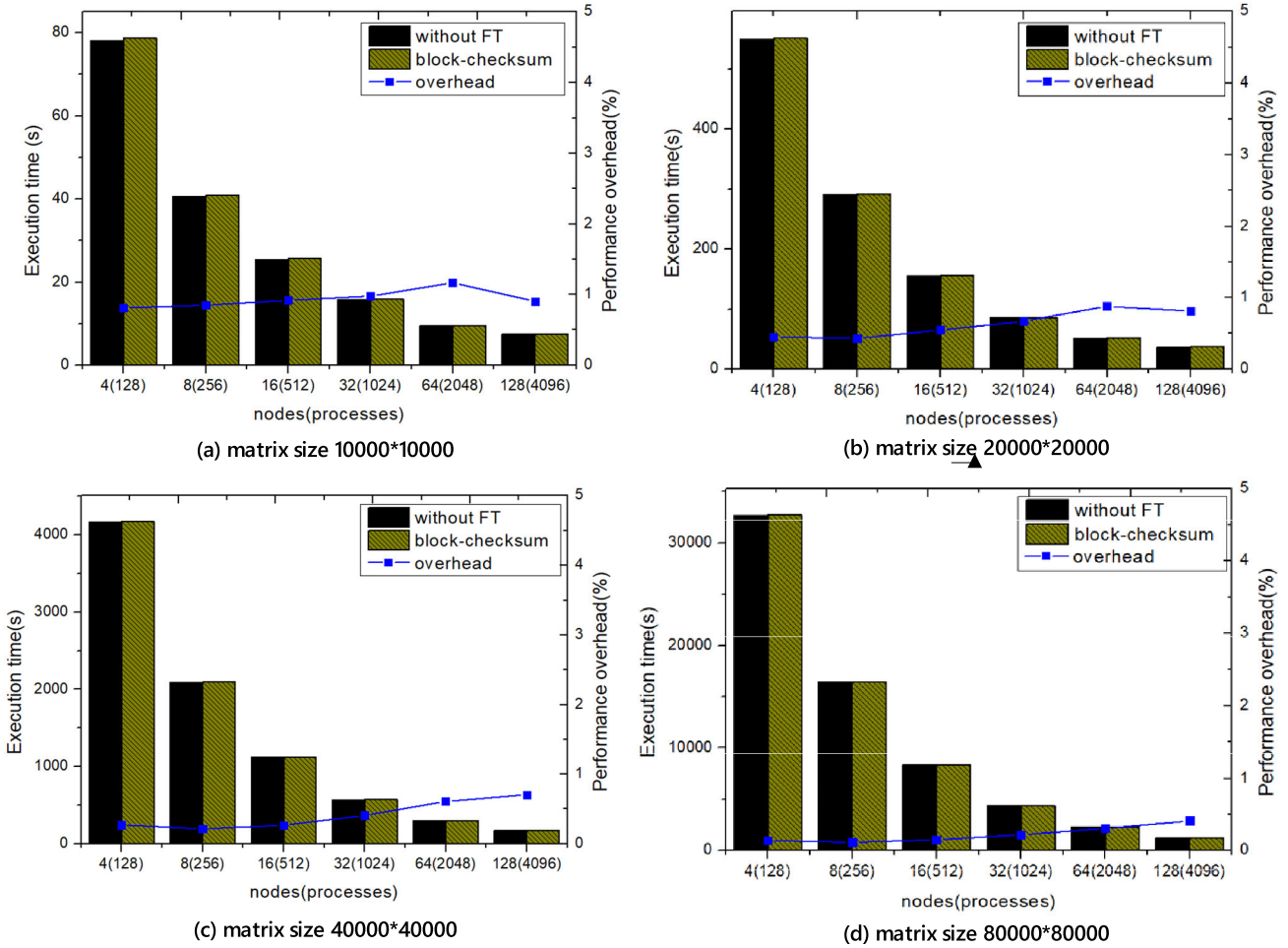


FIGURE 5. Performance of the block-checksum approach without error.

First, we disable the fault-tolerant module for node failure and evaluate the block-checksum approach with the default block size value 4. This experiment aims to evaluate the overhead of the block-checksum approach by comparing it with the original PBLAS. We run the program with different sizes of matrices and different parallel scales, and the results are shown in Figure 5. From the figure we can see that, under different sizes of matrices, the total execution time decreases with the increase of the number of nodes, but the difference between our method and the original matrix multiplication is hard to see when no error occurs. The blue curve, which represents the percentage of time spent on overhead, is always at a low level. Compared with the complexity of matrix multiplication itself, the overhead of our redundancy operations is diluted at this massive computational scale.

We also compare the performance of our block-checksum approach with that of the row-column checksum method. To implement parallel matrix multiplication on multiple nodes with the row-column checksum method, we assign whole rows and columns to different nodes, despite the fact that this is not a normal method for performing matrix computations on large-scale parallel systems. In Figure 6, the extra performance overhead represents the percentage of the time it takes to perform fault-tolerant operations. As shown in the figure, the ratio of the row-column method grows faster than the block-checksum method as the size of the matrix in each process increases (the node number decreases), however, the overhead of the block-checksum approach grows much more slowly than that of the row-column method for both sizes of matrices. The result also confirms the fact that the size of the block influences performance.

Then, we did further experiments; Figure 7 presents the experimental data calculated with different sizes of the checksum blocks. In the figure, n refers to the size of the block. Due to the method of the algorithm, the larger the size of the checksum block is, the less overhead it takes. However, the larger size reduces the accuracy of locating errors, and it may take more time to recover the error data. So, if we take the misdiagnosis rate into account, a larger block size does not produce a more accurate detection; in other words, an appropriate block size is necessary, which can be calculated by our simple method.

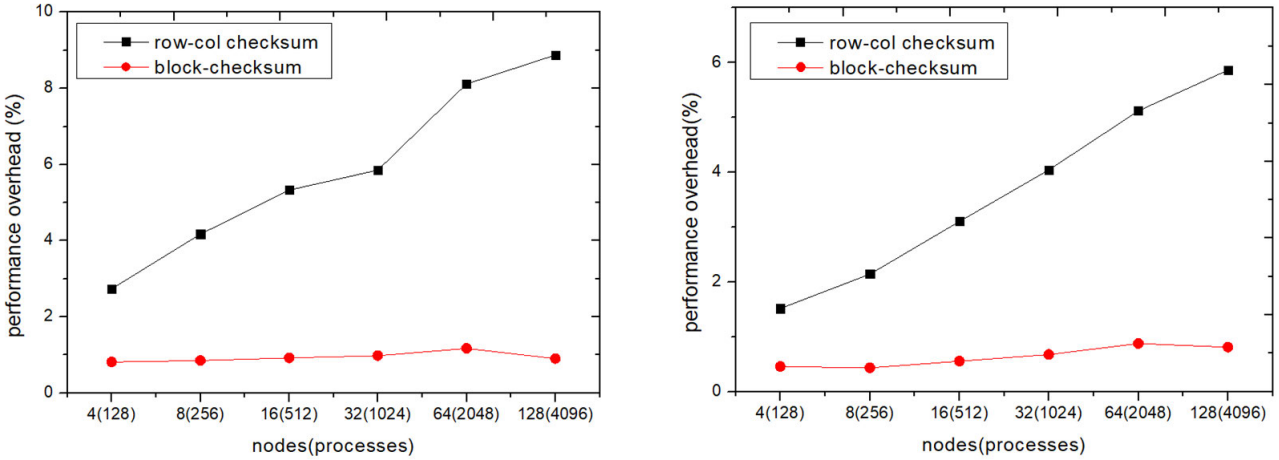


FIGURE 6. Overhead comparison between two FT methods.

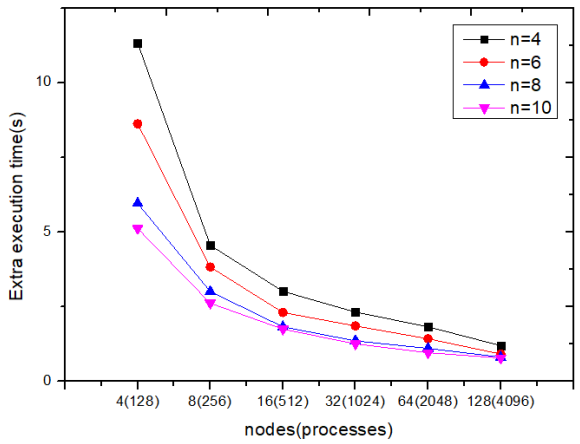


FIGURE 7. Extra execution time under different sizes of checksum blocks with matrix size 40000∗40000.

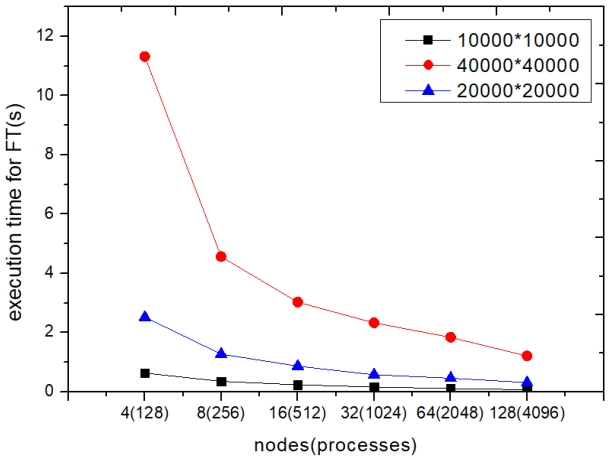


FIGURE 8. Performance of the block-checksum approach on fault tolerance.

Figure 8 presents the execution time for fault tolerance under different scales of matrices. It can be seen that the extra execution time (fault tolerance) reduces significantly as the scale of the matrix decreases (and increases with parallel scales), as the overhead for fault-tolerant calculation is $O(N^2)$. However, because the extra time is combined with three parts—encoding time, calculation time and checking time—it does not reduce perfectly.

2) FAULT-TOLERANT MATRIX COMPUTATION OF NODE FAILURE

In this experiment, we disable the block-checksum module and enable the fault-tolerant module for node failure. We also run the program with different sizes of matrices and different numbers of computing nodes without error, and we compare the performance of our method with that of the original matrix multiplication; the results are shown in Figure 9.

As discussed in previous sections, our method for node failure has little influence on the original algorithm when no error occurs. The experimental results demonstrate this point, as shown in Figure 9; the overhead percentage (blue curve) of time spent on overhead is always at a lower level no matter what the matrix size is or how much greater the parallelism is.

C. PERFORMANCE AND OVERHEAD WITH ERROR

As the main objective of our fault-tolerant method is to ensure reliable matrix multiplication, we perform some experiments to verify the effectiveness of the two approaches. Although both fault-tolerant approaches can ensure program execution, they aim at solving different problems. Therefore, the fault-tolerant function tests are also performed separately. We injected errors through various kinds of artificial methods and explored further issues through the performance results.

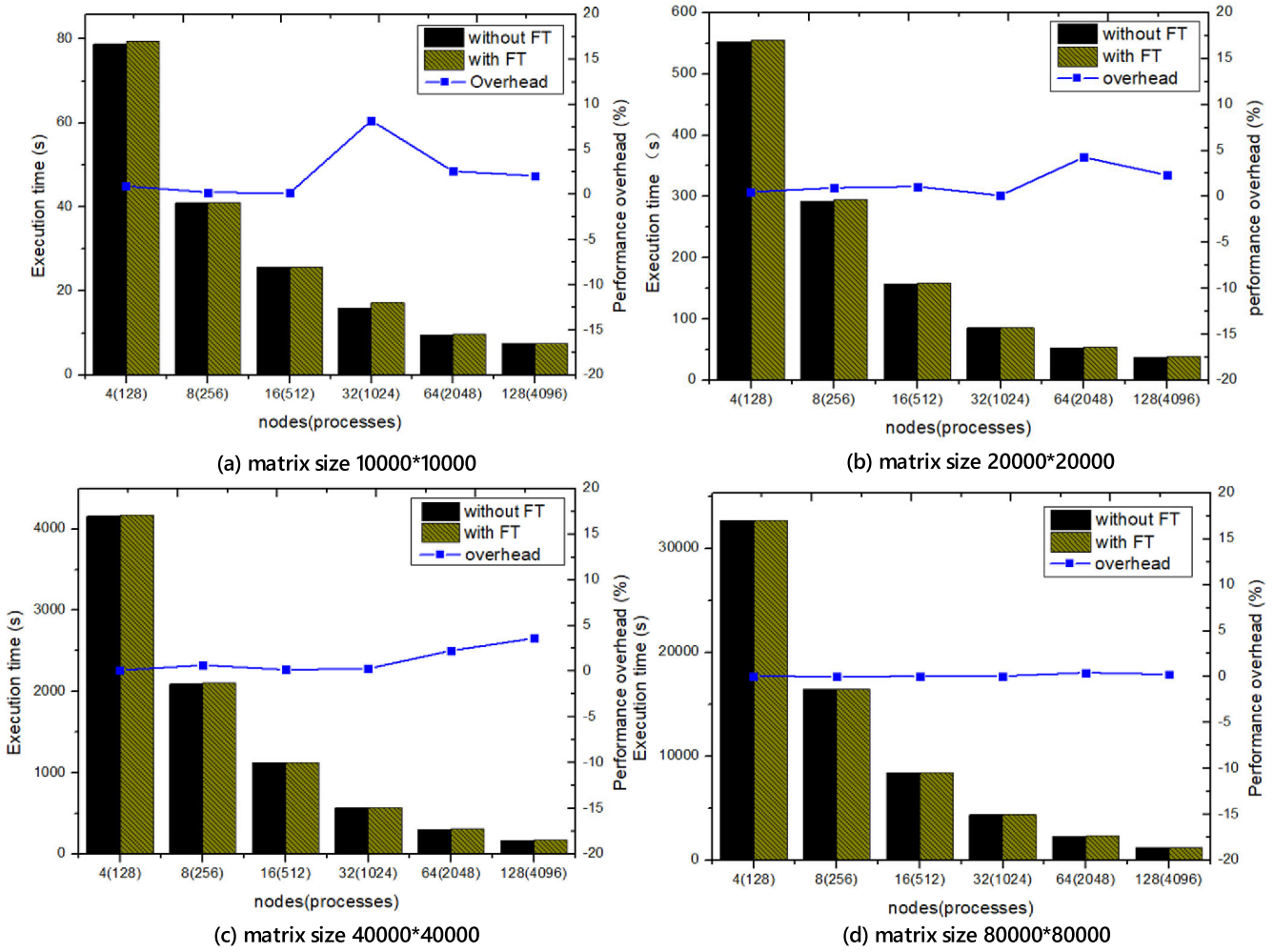


FIGURE 9. Performance of the fault-tolerant method for node failure without error.

1) BLOCK-CHECKSUM APPROACH

The block-checksum approach mainly aims at detecting non-fatal soft errors, such as bit-flipping caused by universal radiation or certain calculation errors, which generally lead to wrong results. We simulate this kind of error by inserting statements to modify some data during computation, and then check the final result to verify whether the fault-tolerant mechanism works correctly. In addition, the total execution time is recorded and compared with the normal execution time (i.e., when there is no error); the results are shown in Figure 10.

As shown in Figure 10, the overhead of error detection and recovery for non-fatal errors are fairly low compared to the high computational overhead of matrix multiplication.

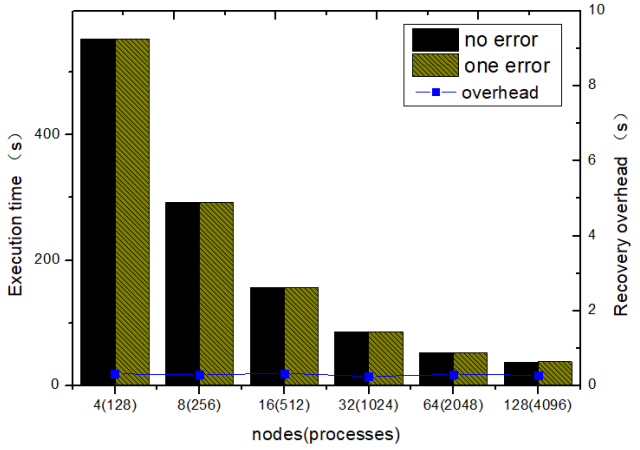


FIGURE 10. Performance of the block-checksum mechanism with error (matrix size: 20000∗20000).

2) FAULT-TOLERANT MATRIX COMPUTATION FOR NODE FAILURE

Node failure is usually in the form of crashes or suspension of the node system; in that case, all of the processes in the node stop working. Therefore, we injected this kind of error by killing all of the processes on a particular node. Considering that the overall execution time is related to the time of error occurrence, we divide the whole execution

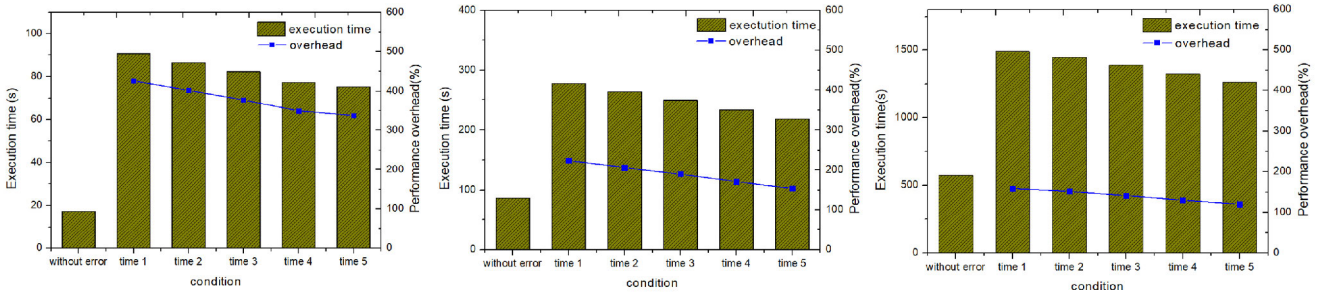


FIGURE 11. Performance of the fault-tolerant method for node failure with error.

procedure into five equal time segments and test the overall execution time by injecting an error in each of those time segments separately.

Figure 11 shows the overall execution time with one node failure. The blue curve indicates the ratio of extra execution time to the time without error. Compared to the normal execution time (i.e., when there is no error), the execution time with error is at least doubled, because error recovery involves re-computation of the tasks on the failed node. Another phenomenon is that the earlier the error occurs, the longer the overall execution time is; this is because processes need to load the data that was lost in the failed node from the storage systems multiple times. In any case, this method ensures the completion of the program even if a node fails, and in addition, most computing nodes are free to allocate to other applications in the error recovery process. Considering that as the matrix scales up, the execution time for computation increases faster than the additional time consumed by the fault-tolerant operation of the different data achieving ways, as a result, the overhead ratio decreases as the matrix size increases.

VII. RELATED WORK

Fault-tolerance and resilience are classic topics for HPC systems and have been studied for a long time. Researchers have proposed various kinds of hardware and software techniques. Among software approaches, checkpoint-restart and replication are two popular and extensively used techniques [30]. However, in the era of peta- and exa-scale systems, there will be tens of thousands of processes in the system, which challenges the scalability of both checkpoint-restart and replication. In this situation, the algorithm-based fault-tolerant (ABFT) approach becomes attractive.

The ABFT was first proposed in [11] to support fault-tolerance for matrix multiplication by using checksums of the rows and columns of matrices to detect errors in computation. Further studies focused on searching for a more accurate way to distinguish rounding errors and computation errors or a more efficient way to detect and recover from errors. [13] presents a highly efficient online approach; instead of detecting errors offline when computation is finished, it transforms matrix multiplication into another algorithm that is more like an iterative algorithm. Reference [16] uses a graphics processing unit (GPU) or co-processors to perform fault-tolerant calculations to reduce the impact of matrix operations. Reference [17] uses a weighted checksum to achieve fault-tolerant matrix multiplication, which reduces time and space overhead but is only suitable for small-scale matrix multiplication. Reference [22] uses an arithmetic and logic unit (ALU) to check the operation of floating-point multiplication. A simplified error analysis (SEA) approach for ABFT is introduced in [23]. A-ABFT calculates the range of rounding errors through the probability distribution of floating-point tails [15].

For the problem of node failure in HPC systems, researchers also propose a variety of solutions. The main ideas can be classified into checkpoint-restart and replication approaches. The checkpoint-restart method records the status of processes and/or systems periodically during program execution and restarts the execution from the nearest checkpoint whenever a fatal error occurs. This method requires synchronization among all of the processes at the time of checkpointing and produces a large volume of checkpoint data, which impacts the I/O system. Further studies aim at optimizing redundant resources, such as encoding data to compress the checkpoints [24]. Traditional replication is mostly used for verifying the correctness of computations, but for node failure errors, it entails using redundant equipment to replace failed equipment. However, due to the loss of data in the faulty equipment, this method often works together with the checkpoint method. Related research has also tried to improve the efficiency of error diagnosis [26]–[28], such as by using a daemon process [25]. Many of these methods rely on the MPI environment [32]–[34]. Concerning the algorithm itself, researchers have proposed the hardware fault-tolerant algorithm [29]. This method relies on FT-MPI [38] for error location and can recover lost data through a coding method based on matrix multiplication characteristics. However, as multiple processes can run on one node, a number of processes will stop working at the same time when an error occurs, and this limits the generality of this method. Most of the fault-tolerant methods for node failure need to be supported by the computing platform itself [35]–[37].

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented the FT-PBLAS, a library for fault-tolerant parallel linear algebra computations on HPC systems. The library is implemented on the basis of the extensively used PBLAS and supports underlying fault-tolerant mechanisms by employing a block-checksum approach for non-fatal errors and a scheme for node failure. Compared to previous work, the block-checksum approach loosens the accuracy of error detection to reduce the expenditure of time on tolerance. In addition, in order to balance the accuracy and efficiency of this algorithm, we also present a method to obtain an appropriate block size, which also has a low overhead. A scheme of fault-tolerant matrix computation for node failure has also been developed to guarantee the reliability of the library. It uses a state table to ensure program execution even if a node fails.

REFERENCES

[1] *Summit-IBM Power System AC 922, IBM POWER9 22C, NVDIA Volta GV100, Dual-Rail Mellannox EDR Infiniband*. Accessed: Feb. 2020. [Online]. Available: https://www.top500.org/system/179397

[2] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The sunway TaihuLight supercomputer: System and applications," *Sci. China Inf. Sci.*, vol. 59, no. 7, Jun. 2016, Art. no. 072001.

[3] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, Mar. 1990.

[4] L. S. Blackford, J. Choi, and A. Cleary, "ScaLAPACK: A linear algebra library for message-passing computers," in *Proc. SIAM Conf. Parallel Process. Sci. Comput.*, 1997, pp. 1–16.

[5] L. Wan, Q. Cao, F. Wang, and S. Oral, "Optimizing checkpoint data placement with guaranteed burst buffer endurance in large-scale hierarchical storage systems," *J. Parallel Distrib. Comput.*, vol. 100, pp. 16–29, Feb. 2017.

[6] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *J. Phys., Conf. Ser.*, vol. 46, pp. 494–499, Sep. 2006.

[7] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–12.

[8] T. Li, M. Shafique, J. A. Ambrose, J. Henkel, and S. Parameswaran, "Fine-grained checkpoint recovery for application-specific instruction-set processors," *IEEE Trans. Comput.*, vol. 66, no. 4, pp. 647–660, Apr. 2017.

[9] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, Jun. 2002.

[10] D. A. Adams, R. R. Nelson, and P. A. Todd, "Perceived usefulness, ease of use, and usage of information technology: A replication," *MIS Quart.*, vol. 16, no. 2, p. 227, Jun. 1992.

[11] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vols. C-33, no. 6, pp. 518–528, Jun. 1984.

[12] H.-J. Lee, P. J. Robertson, and A. B. J. Fortes, "Generalized Cannon's algorithm for parallel matrix multiplication," in *Proc. 11th Int. Conf. Supercomput. (ICS)*. New York, NY, USA: Association for Computing Machinery, 1997, pp. 44–51.

[13] Y. Kim and J. J. Dongarra, "Fault tolerant matrix operations for parallel and distributed systems," Ph.D. dissertation, Univ. Tennessee, Knoxville, TN, USA, 1996.

[14] Y. Zhu, Y. Liu, M. Li, and D. Qian, "Block-Checksum-Based fault tolerance for matrix multiplication on large-scale parallel systems," in *Proc. IEEE 20th Int. Conf. High Perform. Comput. Commun., IEEE 16th Int. Conf. Smart City, IEEE 4th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Jun. 2018, pp. 172–179.

[15] C. Braun, S. Halder, and H. J. Wunderlich, "A-ABFT: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 443–454.

[16] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithmic based fault tolerance applied to high performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 410–416, 2009.

[17] J. N. Higham, *Accuracy and Stability of Numerical Algorithms*. Philadelphia, PA, USA: SIAM, 1996.

[18] *Open MPI Document*. Accessed: Feb. 2020. [Online]. Available: https://www.open-mpi.org/doc/current/

[19] A. D. Selvakumar, P. M. Sobha, and G. C. Ravindra, "Design, implementation and performance of fault-tolerant message passing interface (MPI)," in *Proc. 7th Int. Conf. High Perform. Comput. Grid Asia Pacific Region*, 2004, pp. 120–129.

[20] K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using MPI-ULFM," in *Proc. 21st Eur. MPI Users' Group Meeting (EuroMPI/ASIA)*, 2014, pp. 51–56.

[21] V. Stefanidis and K. Margaritis, "Algorithm based fault tolerant matrix operations for parallel and distributed systems: Block checksum methods," in *Proc. 6th Hellenic-Eur. Conf. Comput. Math. Appl.*, 2003, pp. 767–773.

[22] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham, "Algorithm-based fault tolerance on a hypercube multiprocessor," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1132–1145, Sep. 1990.

[23] S. Dutt and F. T. Assaad, "Mantissa-preserving operations and robust algorithm based fault tolerance for matrix computations," *IEEE Trans. Comput.*, vol. 45, no. 4, pp. 408–424, Apr. 1996.

[24] L. Bautistagomez, "FTI: High performance fault tolerance interface for hybrid systems," in *Proc. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–32.

[25] G. Zhang, Y. Liu, H. Yang, and D. Qian, "A lightweight and flexible tool for distinguishing between hardware malfunctions and program bugs in debugging large-scale programs," *IEEE Access*, vol. 6, pp. 71892–71905, 2018.

[26] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, and F. Qin, "MC-checker: Detecting memory consistency errors in MPI one-sided applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2014, pp. 499–510.

[27] B. Zhou, J. Too, and M. Kulkarni, "WuKong: Automatically detecting and localizing bugs that manifest at large system scales," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 131–142.

[28] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree, "Large scale debugging of parallel tasks with AutomaDeD," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2011, pp. 1–10.

[29] C. J. D. Zizhong, *Algorithm-Based Fault Tolerance for Fail-Stop Failures*. Piscataway, NJ, USA: IEEE Press, 2008.

[30] P. I. Nitin and T. N. Vijaykumar, "FaultHound: Value-locality-based soft-fault tolerance," *ACM SIGARCH Comput. Archit. News*, vol. 2015, vol. 43, no. 3, pp. 668–681.

[31] (2018). *Nation Supercomputer Center in Guangzhou Home Page*. [Online]. Available: http://www.nscc-gz.cn/

[32] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, "Deadlock detection in MPI programs," *Concurrency Comput., Pract. Exper.*, vol. 14, no. 11, pp. 911–932, 2002.

[33] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with umpire," in *Proc. ACM/IEEE SC Conf. (SC)*, Nov. 2000, p. 51.

[34] B. Krammer, T. Hilbrich, and V. Himmler, "MPI correctness checking with marmot," in *Proc. Int. Workshop Parallel Tools High Perform. Comput.*, 2008, pp. 61–78.

[35] Center for High Throughput Computing, University of Wisconsin Madison. (Mar. 2018). *HTCondor Version 8.7.7 Manual*. [Online]. Available: http://research.cs.wisc.edu/htcondor/manual/v8.7/condor-V8_ 7_7-Manual.pdf

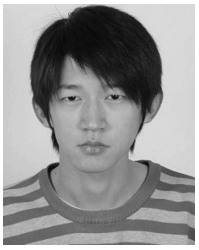
[36] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "DejaVu: Transparent user-level checkpointing, migration, and recovery for distributed systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–10.

[37] M. Schulz, G. Bronevetsky, and R. Fernandes, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs," in *Proc. ACM/IEEE SC2004 Conf.*, Nov. 2004, p. 38.

[38] G. E. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Proc. Eur. PVM/MPI Users Group Meeting Recent Adv. Parallel Virtual Mach. Message Passing Interface*, 2000, pp. 346–353.

YANCHAO ZHU is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Beihang University. He focuses on algorithm-based fault tolerance for high performance computing systems. His research interests include high-performance computing, distributed computing, and parallel computing.

YI LIU received the Ph.D. degree from the Department of Computer Science, Xi'an Jiaotong University, in 2000. He is currently a Professor with the School of Computer Science and Engineering and the Director of the Sino-German Joint Software Institute, Beihang University, China. His research interests include computer architecture, high-performance computing, and new generation of network technology.



GUOZHEN ZHANG is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Beihang University. He focuses on program debugging of large-scale parallel applications. His research interests include high-performance computing, program debugging, distributed computing, and parallel computing.