Building Algorithmically Nonstop Fault Tolerant MPI Programs

Rui Wang, Erlin Yao, Mingyu Chen, Guangming Tan
State Key Laboratory of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
{wangrui2009, yaoerlin, cmy, tgm}@ict.ac.cn

Pavan Balaji, Darius Buntinas
Mathematics and Computer Science,
Argonne National Laboratory
{balaji, buntinas}@mcs.anl.gov

***Abstract*—With the growing scale of high-performance computing (HPC) systems, today and more so tomorrow, faults are a norm rather than an exception. HPC applications typically tolerate fail-stop failures under the stop-and-wait scheme, where even if only one processor fails, the whole system has to stop and wait for the recovery of the corrupted data. It is now a more-or-less accepted fact that the stop-and-wait scheme will not scale to the next generation of HPC systems.**

Inspired by the previous stop-and-wait algorithm-based fault tolerance (ABFT) recovery technique, we propose in this paper a nonstop fault tolerance scheme at the application level and describe its implementation. When failure occurs during the execution of applications, we do not stop to wait for the recovery of the corrupted node; instead, we replace it with the corresponding redundant node and continue the execution. At the end of execution, the correct solution can be recovered algorithmically at a very low cost.

In order to implement the scheme, some new fault-tolerant features of the Message Passing Interface (MPI) have been investigated and utilized in the MPICH implementation of MPI. We also describe a case study using High Performance Linpack (HPL) with these new features and evaluate the performance of both our new scheme and ABFT recovery. Experimental results show the advantage of our new scheme over ABFT recovery even in a small scale.

Keywords: **Nonstop; Algorithm-Based Fault Tolerance; High Performance Linpack; MPICH**

I. Introduction

The largest systems in the world today already scale to hundreds of thousands of cores. With plans under way for exascale systems to emerge within the next decade, we will soon have systems comprising more than a million processing elements. As researchers work toward architecting these enormous systems, it is becoming increasingly clear that, at such scales, resilience to hardware faults is going to be a prominent issue that needs to be addressed.

While the peak performance of contemporary high-performance computing (HPC) systems continues to grow exponentially, it is getting more and more difficult for scientific applications to achieve high performance because of the increasing failures in these systems. Today's long-running scientific applications typically tolerate system failures by using checkpoint-restart techniques, where the application (or the system) occasionally checkpoints the state of the executing processes to a reliable storage [1], [2]. With the increasing gap between the compute power of these systems and the capability of the storage systems, the scalability of checkpointing solutions in next-generation HPC systems is in doubt. For example, based on a balanced system model and statistics from the Computer Failure Data Repository (CFDR) [3], Gibson et al. predicted that the effective application utilization of checkpoint/restart-based fault tolerance will keep dropping to zero under current technology trends [4], [5]. Such behavior has been predicted by other researchers as well [6].

Diskless checkpointing has been proposed to improve the scalability of checkpointing [1], [7]. However, methods such as algorithm-based fault tolerance (ABFT) recovery [8], [7] promise even better scalability. The ABFT method has been studied in many applications, such as ScaLAPACK [9], [10], HPL [11], preconditioned conjugate gradient (PCG) solver [8], [7], and iterative methods in PETSc [12]. Unlike system-level checkpointing, using ABFT recovery to handle faults is not transparent to applications. However, it does have several advantages. First, this technique introduces no explicit stoppage to perform a checkpoint. The redundancy is computed synchronously with the main computing process; thus, no rollback of work is needed when failure occurs. Second, the redundancy is completely in-memory and does not rely on the capability of the storage system. Previous experimental results show that this technique has better performance than checkpointing [8], [7], [12].

However, ABFT recovery is still based on the stop-and-wait scheme, where even if only one process fails during the execution of applications, all processes alive have to stop and wait for the recovery of the failed process. According to Amdahl's law, the maximum speedup of an application is limited by its sequential proportion. Thus, the stop-and-wait scheme will affect the parallel efficiency of applications under large scale.

To ease this problem, we propose a new nonstop algorithm-based fault tolerance scheme called ***ABFT hot-replacement.*** When failure occurs during the execution of the application, we do not stop to wait for the recovery of corrupted data; rather, we replace it with the corresponding redundant data and continue the execution. At the end of execution, our scheme can recover the correct solution algorithmically at a low cost.

In order to implement this new scheme, some new fault

tolerance features of Message Passing Interface (MPI) implementations are required, for which we utilized the MPICH implementation of MPI.

This paper also describes a case study using High Performance Linpack (HPL) and the MPICH implementation of MPI and evaluates its performance. Together with those new features, some additional supports at application level such as message redundancy and virtual ranks are implemented, which may be addressed in future version of MPICH. Experimental results show an observable advantage of the new scheme over ABFT recovery even in a small scale.

The rest of paper is organized as follows. Section II briefly introduces the conventional ABFT method and fault-tolerant MPI implementations. Section III proposes our new algorithm-based fault tolerance scheme, ABFT hot-replacement. In Section IV, we introduce the new fault tolerance features of MPICH used to implement the proposed scheme. In Section V, we demonstrate how to incorporate fault tolerance into HPL using the proposed scheme. Section VI presents some experimental results and evaluations. Section VII concludes this paper and discusses future work.

II. Background and Related Work

In this section, we first briefly introduce the conventional algorithm-based fault tolerance method. Then the background and related work of fault-tolerant MPI are given.

A. Conventional Algorithm-Based Fault Recovery Method

The algorithm-based fault tolerance technique first proposed by Huang and Abraham [13] [14] uses the encoded data to detect and correct instant errors in certain matrix operations at the system level. This technique was further developed by Chen and Dongarra to tolerate fail-stop failures that occurred during the execution of HPC applications [15] [9]. This latter technique is also the basis of the ABFT recovery method mentioned in this paper. The idea of ABFT is to encode the original matrices using real number codes to establish a checksum type of relationship between data, and then redesign algorithms to operate on the encoded matrices in order to maintain the checksum relationship during the execution.

Assume there will be a single process failure. Since it's hard to locate which process will fail before the failure actually occurs, a fault-tolerant scheme should be able to recover the data on any process. In the conventional ABFT method, it is assumed that at any time during the computation the data D_i on the ith process P_i satisfies

$$D_1 + D_2 + \cdots + D_n = E, \tag{1}$$

where n is the total number of processes and E is data on the encoding process. Thus, the lost data on any failed process can be recovered from Eq. (1). Suppose P_i fails. Then the lost data D_i on P_i can be reconstructed by

$$D_i = E - (D_1 + \cdots + D_{i-1} + D_{i+1} + \cdots + D_n). \tag{2}$$

In practice, this kind of special relationship is by no means natural. However, it is possible to design applications to maintain such a special checksum relationship throughout the computation, and this is one purpose of ABFT research.

B. Fault-Tolerant MPI

Although fault tolerance has not been addressed by the MPI standard as of 2.2, work has been done previously to improve the fault tolerance of MPI implementations and applications using checkpoint-rollback recovery (e.g., [16], [17] and [18]) and application-level fault tolerance (e.g., [19], [20] and [21]). Checkpoint-based fault tolerance is attractive because it is provided transparently to the application. No action on the part of the application is required to detect or handle faults. However, checkpointing-based fault tolerance has drawbacks, as mentioned in Section I, and is not suitable in all situations.

To support application-level fault tolerance, the MPI implementation must itself be able to tolerate faults. Furthermore, it must provide features to allow the application to proceed in a meaningful manner after a fault has occurred. This issue has also been studied previously (e.g., [19], [20] and [21]). Collective operations in FT-MPI [19] are required, in the presence of a failed process, to complete with the same result as if there were no failed processes; otherwise the operation must abort. The fault-tolerant features we added to MPICH differ from FT-MPI in that we decided to relax the restriction on collective operations in order to reduce the overhead during failure-free operation. In our implementation, collectives will return an error code at those processes where the result may be invalid, but they also may return successfully if the result is known to be unaffected by the failure.

In MPI/FT [21] failed processes are always replaced from a pool of spare processes. When the pool of spare processes is exhausted, the application is aborted. Our implementation does not yet have the ability to restart failed processes; instead, the decision on whether to continue is left to the application. Since spare processes are not always feasible, we feel that allowing the application to decide how to continue gives the application more flexibility.

III. A Nonstop Algorithm-Based Fault Tolerance Scheme

In this section, a nonstop algorithm-based fault tolerance scheme, called ABFT hot-replacement, is proposed.

A. Hot-Replacement Scheme

For the simplicity of presentation, we assume there will be only one process failure. However, it is straightforward to extend the results here to multiple failures by simply using multilevel redundancy or regenerating the encoded data. Suppose that at any time during the computation the data D_i on the ith process P_i satisfies

$$D_1 + D_2 + \cdots + D_n = E. \tag{3}$$

If the ith process failed during the execution, we replace it with the encoding process E and continue the execution instead of stopping all the processes to recover the lost data D_i. Note

that this kind of transformation can be effective only when there is an encoding relationship among the data.

From the global view, the original data is

$$D = (D_1 \cdots D_{i-1} D_i D_{i+1} \cdots D_n), \tag{4}$$

and the transformed data (after replacement) is

$$D' = (D_1 \cdots D_{i-1} E D_{i+1} \cdots D_n). \tag{5}$$

We can establish a relationship between the transformed data and the original data as

$$D' = D \times T, \tag{6}$$

and T can be represented as an $n \times n$ matrix in the following form:

$$T = \begin{pmatrix} 1 & & & 1 & & & \\ & \ddots & & \vdots & & & \\ & & 1 & 1 & & & \\ & & & 1 & & & \\ & & & 1 & 1 & & \\ & & & \vdots & & \ddots & \\ & & & 1 & & & 1 \end{pmatrix}, \tag{7}$$

where the elements omitted in the diagonal and the ith column are all 1 and the other elements omitted are 0. We can see that T is a nonsingular matrix.

If operations on the data are linear transformations (e.g., matrix operations such as decomposition), the relationship $D' = D \times T$ will always be kept. At the end of computation, the original correct solution based on D can be recomputed through the intermediate solution based on D'. And this recomputation is actually a transformation related to T.

One can see that the encoding relationship $D' = D \times T$ cannot be maintained under all HPC applications. However, for a class of them, including matrix computations involving linear transformations, such as matrix decomposition, matrix-matrix multiplication, and scalar product, the encoding relationship can be maintained. A simple example on HPL is given in the next section.

B. A Simple Example on HPL

HPL is a popular benchmark for supercomputing and is widely used for ranking supercomputers on Top500 [22]. It is used to solve linear system

$$Ax = b \tag{8}$$

using Gaussian elimination with partial pivoting (GEPP). We assume process P_i fails during the execution. After the replacement with redundancy, the linear systems becomes

$$A'y = b. \tag{9}$$

At the end of execution, it gives the solution y as an intermediate result. Since all the operations in GEPP are linear transformations, the relationship

$$A' = A \times T \tag{10}$$

can always be kept during the execution. Combining Eq. (8), (9), and (10), we obtain the correct solution x by

$$x = T \times y. \tag{11}$$

Assume the transformation matrix T is the same as Eq. (7). Substitute Eq. (7) into Eq. (11), giving

$$\begin{cases} x_j = y_i + y_j, & if\ 1 \le j \ne i \le q \\ x_i = y_i \end{cases} \tag{12}$$

It shows that the cost of recomputing the solution x is $O(n)$, whereas the cost of ABFT recovery to handle a single process failure is $O(n^2)$. Because of the simplicity of T, the correct solution can be recovered algorithmically at a low cost.

In the real implementation of fault-tolerant HPL, however, the encoding of data matrix and the transformation matrix T will be more complex because of its block-cyclic data distribution. We will discuss these issues in Section V.

C. Advantages

Compared with ABFT recovery, ABFT hot-replacement introduces the following two advantages.

First, consider there is only one process failure in a system, say P_i. With ABFT hot-replacement, the other processes in the system do not need to stop and wait for the recovery of the data on P_i. This nonstop scheme improves the efficiency of large-scale systems, since the wasted CPU cycles during the stop-and-wait period increase with the system scale.

Second, since T is a very sparse matrix, the cost of recomputing the correct solution by using T can be much lower than that of the recovery of failed process directly, especially for those computation-intensive or communication-intensive applications where each process is allocated with a large set of data. This can be seen from the simple example of HPL.

In the case shown above, because of the maintenance of checksum relationship, we were able to process the lost data on any failed process without periodically checkpointing. In practice, however, this kind of special relationship is by no means natural. Nevertheless, one can design an application to maintain such a special checksum relationship throughout the computation, by using the checksum matrices of the original ones as the input matrices. To demonstrate the feasibility of the proposed ABFT hot-replacement method, we will apply it to the fault tolerance of HPL in Section V.

IV. Fault Tolerance in MPICH

While the MPI Forum is currently working on adding fault-tolerance features to the standard, [23], the current MPI standard (2.2) does not yet define the behavior of an MPI library in the presence of process or communication failures. We have identified the minimum set of fault-tolerance features required of an MPI implementation in order to support our fault tolerant algorithm as follows.

- Node failure will not cause the entire job to abort.
- The MPI implementation should provide a mechanism to query for failed processes.

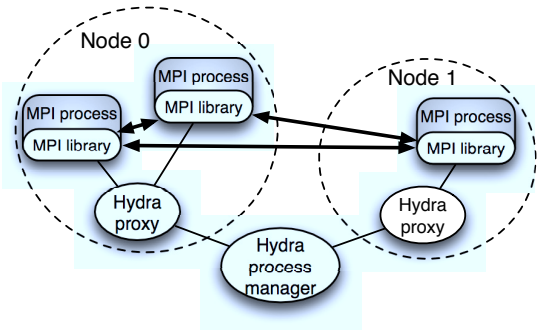

Fig. 1. Logical diagram showing MPI processes, MPICH libraries and Hydra process manager and proxies

- Communication operations involving a failed process will not hang and will eventually complete.
- Communication operations will return an error code when it is affected by a failed process. This is needed to determine whether to re-send or re-receive messages.

These requirements are a subset of the features proposed by the MPI Forum's fault tolerance working group. We have implemented these requirements and included them in the 1.3.2 release of MPICH2. In this section we describe the failure detection mechanism and the API to query for failed processes followed by the behavior of point-to-point and collective communication when failed processes are involved. Note that since MPI fault-tolerance features have not been standardized and are still in the process of being defined, the MPICH fault-tolerance features described here may differ from the features ultimately included in the MPI standard, and so may change in the future.

A. Process failure detection and query

Although process failure can sometimes be inferred from communication failure by the MPI library, this is not a reliable way to detect failed processes. For instance, sending a message to a failed process may result in a timeout or an error reply from the operating system at the remote node, but a passive receiver may wait for a message which will never arrive from a failed process. For this reason, we modified the Hydra process manager to detect failed processes. Figure 1 shows the Hydra process manager and proxies in relation to the MPI processes. When launching an application Hydra starts a proxy at each node, which, in turn, spawns the MPI processes. When an MPI process terminates, the proxy will receive a `SIGCHLD` signal and will then notify the process manager. If the process terminated before calling `MPI_Finalize()`, the process is considered to be a failed process. Hydra can be configured to abort the entire job when a failed process has been detected. This is often useful for non-fault tolerant applications in order to prevent runaway processes. For fault tolerant applications, however, when Hydra detects a failed process, it does not abort the job, but adds the process to its list of failed processes, then notifies the MPI processes by having the proxies send the processes a `SIGUSR1` signal.

The MPICH library catches the `SIGUSR1` signal and queries Hydra for the list of failed processes then closes all connections to the failed processes. The MPI process can get the failed process list by querying the `MPICH_ATTR_FAILED_PROCESSES` attribute on `MPI_COMM_WORLD`. The list consists of an array of integers representing the ranks of failed processes, terminated by `MPI_PROC_NULL`.

B. Point-to-point communication

When a message is sent to or a receive is posted for a message from a failed process, the send or receive function returns an `MPI_ERR_OTHER` error code, or, in the case of nonblocking operations, the request is completed with the error code set. Additionally, wildcard receives, i.e., using `MPI_ANY_SOURCE`, posted to communicators with a failed process will also be completed with an error. When notified of a failed process, MPICH scans the list of posted receives and queued sends, and completes the ones involving the failed process with an error.

Handling wildcard receives as described above requires knowing that a communicator contains a failed process. The communicator data structure has a list of every process it contains, however, to conserve memory, MPICH does not keep track of the reverse, i.e., a list of every communicator of which a particular process is a member. Determining whether a communicator contains failed processes requires scanning the list of processes in that communicator. This can be costly, especially if an application uses many communicators.

To address this, a communicator is only checked for a failed process only when a wildcard receive is encountered, either when one is posted or when scanning the list of posted receives. Each communicator has a flag which is set whenever the list of processes is scanned and it is determined that it contains a failed process. If the flag is set, then the list of processes does not need to be scanned again, since once a process fails, it cannot be repaired. However, if the flag is not set, it is possible that the communicator was not previously checked for a failed process or that a process has failed since the last time the communicator was checked. In this case, we use a timestamp to indicate the last time the list of processes was scanned. If a process failure notification has been received since the list of processes was scanned, the list is scanned again, otherwise we know that there cannot be any failed processes in the communicator. This reduces the number of communicators that need to be scanned and the number of times a particular communicator needs to be scanned.

C. Collective communication

Collective operations, by definition, require the participation of all processes of the communicator. The current MPI standard does not define the result of a collective communication operation when a member of the communicator has failed. Rather than try to define the result of a collective operation with missing processes, we decided first, make sure that the collective operation does not hang, and additionally, that the

collective operation will return an error at all processes where the result may be invalid.

Failed processes may cause a collective operation to hang unless the collective operation has been implemented to handle process failures. Consider a broadcast operation that is implemented by logically arranging the processes in a tree and forwarding the message over the tree using point-to-point messages. If the broadcast algorithm is implemented so that when a send or receive operation returns an error, the process will immediately abort the broadcast operation, then that process's children may hang. A child process will wait trying to receive the message from its parent, but if the parent has aborted the broadcast it will never receive the message, and will not get an error because the parent has not failed. The solution we applied is to continue the collective operation even when a failure is detected. This will prevent processes from hanging, but may result in some, but not all, processes getting invalid results from the collective operation. For instance, in the broadcast operation, the ancestors of a failed process would receive an invalid message; however, the other processes will receive the message correctly.

It is important for each process of an application to tell whether a collective operation produced a valid result. Because only the processes directly receiving from or sending to the failed process will detect the failure, the other ancestors of the failed process will need to be notified that the received data is invalid. To do this we mark the messages carrying invalid data by using a different tag value. During a collective, when a receive operation fails, all subsequent send operations will set the tag to mark the message as containing invalid data. Processes will use a wildcard tag, i.e., `MPI_ANY_TAG`, to receive the collective messages, then check the tag to determine whether the data is valid. If a process receives a message marked as containing invalid data, the process will continue performing the collective operation, but will mark any subsequent messages it sends as containing invalid data. When a process completes the collective operation it will return an error code if a send or receive operation failed, or if it had received invalid data at any point during the operation. In this way an error code will be returned by the collective function at any process where the collective produced an invalid result.

As of MPICH2 version 1.3.2, no mechanism is provided to be able to repair a communicator, or to allow a collective operation to be performed correctly while excluding any failed processes in a communicator. Such features are currently being investigated in this paper at the application level and we expect to include them in future versions of MPICH.

V. Incorporating Fault Tolerance into High Performance Linpack

With the support of the MPICH implementation of MPI, we incorporate fault tolerance into HPL using ABFT hot-replacement scheme proposed in Section III. Our implementation focuses on handling a single process failure occurred in the main part of HPL, that is factorization, row broadcast and update phases. However, it's possible to extend the result here to multiple process failure cases by rebuilding checksum in background and maintaining multi-level redundancies.

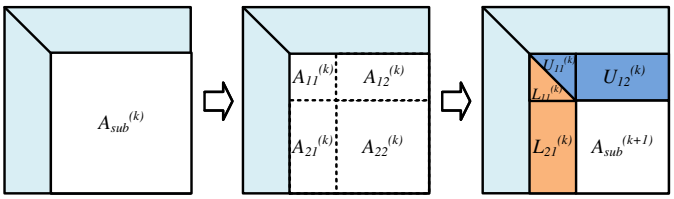

Fig. 2. Algorithm of HPL using left-looking LU factorization.

In this section, we first present an overview of HPL. Second, we briefly introduce the encoding of matrix and how to locate a failure. Then we give a detailed presentation on the failure handling in factorization and update phases, using hot replacement and background recovery technique. For better robustness, we also describe failure handling in row broadcasting by proposing a new message broadcasting mechanism.

A. High Performance Linpack Overview

The main algorithm of HPL solves $Ax = b$ using GEPP. It first computes the LU factorization of the $n \times (n+1)$ coefficient matrix $[A|b]$, producing $[A|b] = [[L,U]L^{-1}b]$. Then the solution x can be obtained by solving the upper triangular system $Ux = L^{-1}b$. The pseudo code for HPL using left-looking LU factorization with look-ahead $depth = 0$ is given as follows. For better presentation, we also depict it in Figure 2. The colored part of the matrix in Figure 2 is the part that has been computed, while the uncolored part is the trailing submatrix A_{sub}. The part of the matrix that changes color is the part that is computed by one iteration. We denote matrices in the kth iteration with a superscript (k). At the very beginning ,we assume ${A_{sub}}^{(0)} = A$. Then we could see that in the kth iteration, $A_{sub}^{(k)}$ is divided into 4 submatrices: ${A_{11}}^{(k)}$, ${A_{12}}^{(k)}$, ${A_{21}}^{(k)}$ and ${A_{22}}^{(k)}$. At the end of the iteration, it produces the new trailing submatrix ${A_{sub}}^{(k+1)}$. We could also notice that the trailing submatrix shrinks after every iteration. For more details, we refer the reader to [24].

Each process generates its local random matrix A
; operate on each trailing submatrix ${A_{sub}}^{(k)}$
for $k = 0, 1, \ldots$
  ; factorization phase
  factorize ${A_{11}}^{(k)}$ and ${A_{21}}^{(k)}$ as ${L_{11}}^{(k)} \cdot {U_{11}}^{(k)}$ and ${L_{21}}^{(k)} \cdot {U_{11}}^{(k)}$
  ; row broadcast phase
  broadcast ${L_{11}}^{(k)}$, ${L_{21}}^{(k)}$ and pivoting information right
  ; update phase
  perform row swaps and calculate ${U_{12}}^{(k)} = {{L_{11}}^{(k)}}^{-1} \cdot {A_{12}}^{(k)}$
  update the trailing submatrix ${A_{sub}}^{(k+1)} = {A_{22}}^{(k)} - {L_{21}}^{(k)} \cdot {U_{12}}^{(k)}$
solve $U \cdot x = L^{-1} \cdot b$ to obtain x

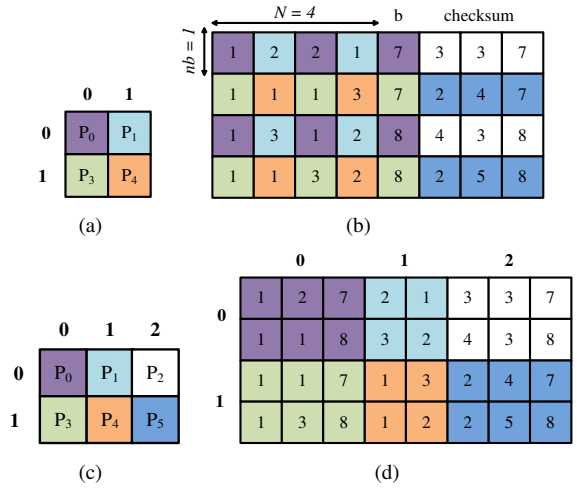

Fig. 3. Two-dimensional process grid with $P = 2$ and $Q = 2$ and the data distribution of an example matrix with redundancy: (a) original process grid, (b) original matrix with redundancy from global view, (c) process grid with redundancy, (d) original matrix with redundancy from distributed/local view.

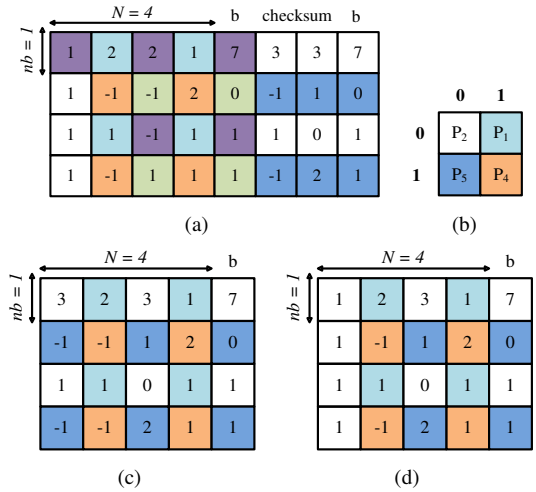

Fig. 4. Perform ABFT hot-replacement on the example matrix after $P3$ failed: (a) global matrix before failure, (b) rearranged process grid after hot-replacement, (c) global matrix after hot-replacement, (d) global matrix after recovery.

B. Matrix Encoding

The encoding of the matrix is similar to that described in [9]. In HPL, the random matrix A is distributed onto a two-dimensional $P \times Q$ process grid according to the block-cyclic scheme. To tolerate a single-process failure, we dedicate additional P processes and organize the total $P \times Q + P$ processes as a $P \times (Q + 1)$ process grid. The redundant P processes are distributed on the $(Q+1)$th column of the new process grid, holding the checksum encoding, which can be obtained by adding all local matrices on the first Q process columns. Figure 3(a) shows the original layout of a process grid with P = 2 and Q = 2. Figure 3(c) is the corresponding process grid with redundant processes $P2$ and $P5$, where $P2$ holds the checksum encoding of $P0$ and $P1$, and $P5$ does similarly. We show the encoding of an example matrix with redundancy in Figure 3(b) and denote the local matrices of each process in Figure 3(d). In Figure 3, we distinguish different processes in different colors. The matrix order N we use is 4 and the block size nb is 1. Based on the consideration of simple implementation, we also keep a copy of column b on the redundant processes.

Under this encoding scheme, the row checksum relationship (matrix L excluded) can be maintained during the computation, after each update phase [11].

C. Failure Location

Since failure occurrence is unpredictable, we do not know the time and the location of the failure before it actually occurs. To obtain this information, every process should query the new `MPICH_ATTR_FAILED_PROCESSES` attribute on `MPI_COMM_WORLD` after every phase. Since different processes may finish the same phase at different time, we need to do a barrier operation before the failure checking operation to make sure that all processes get the same result.

D. Hot Replacement Technique

If a process failure is discovered in the failure checking after a phase, all processes alive will enter a uniform entry to perform failure handling using the ABFT hot-replacement method proposed in Section III. The redundant process column will replace the dead process column, and the computation will continue. Consider the example in Figure 3. We assume the failed process is $P3$ and the failure is discovered after the first update phase, thus we replace the redundant process column with the dead one, i.e. the 1st column. Figure 4(a) is the supposed matrix after the first update phase without failure. Figure 4(b) is the rearranged grid after hot-replacement and Figure 4(c) is the corresponding global matrix. Note that the submatrix with cross lines in Figure 4 denotes matrix L.

In our implementation, we replace the whole column rather than a single process upon process failure. Since processes inside a column need to do some row swaps during the computation, only replacing the dead process by a redundant process will make the transformation matrix T difficult to derive.

Moreover, this hot-replacement is not physical but can be done rapidly. It is implemented by exchanging the relative parameters in the data structure of process grid and does not involve any internode data transmissions.

Note that the replacement can be done only when the checksum relationship is satisfied, that is, after the update phase. Thus, if a process failure is found after a factorization or row-broadcasting phase, all processes alive should continue execution until they finish the updating. After that, the hot-replacement could be performed.

Another issue is how to update the status of communicators after the replacement. A communicator is used in MPI to denote a set of active processes. An ideal fault-tolerant MPI implementation should be able to dynamically adjust a communicator to add new or remove failed processes. In our

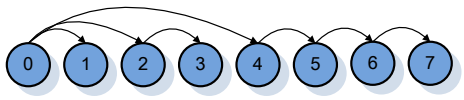

Fig. 5. The way how 2rinM method works

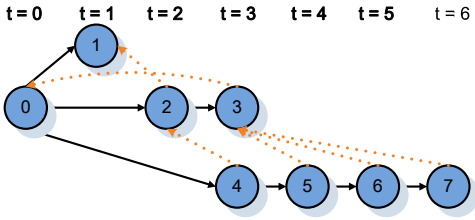

Fig. 6. Message transmissions of 2rinM method with father-son relationship in chronological sequence

implementation, we address this issue in the application level by designating a virtual rank to each process. Messages are sent or received by using these virtual ranks. We wrote two macros to do the mappings between the original rank of a process in `MPI_COMM_WORLD` and its virtual rank according to the status of the process grid.

E. Background Recovery Technique

It can be inferred from Figure 4(c) that the matrix U we get at last will no longer be upper-triangular, since the submatrix that has been factorized before the failure occurrence is replaced by redundancy. This will make it difficult to solve $Ux = L^{-1}b$. To overcome this issue, we do not replace the already factorized data but, instead, recover them onto the corresponding redundant processes. Figure 4(d) shows the global view of the example matrix after the recovery.

We note that the factorized data is not for immediate use (only used in solving $Ux = b$). Thus, recovering the data could be performed in background to achieve better performance. We overlap the communication needed for the recovery with the succeeding update phases, which consist mainly of matrix-matrix multiplications. We divide the data to recover into several parts according to the amount of data to be transferred in every succeeding update phase. Then we use nonblocking `MPI_Isend` and `MPI_Irecv` to overlap the communication and the computation. Note that as the calculation takes place, the data to update decreases; thus the data to send or receive during the update phase also decreases.

F. Failure Handling in Row Broadcasting

The methods discussed thus far can handle failures in the factorization or update phases. But for the row broadcast phase, the procedure may be a little tricky. All row broadcasting methods provided by HPL are based on message forwarding. Thus, if a process fails during a row broadcast phase, processes that indirectly receive messages from the failed process will be blocked. For example, suppose P_1 forwards messages from P_0 to P_2 and P_0 failed. Then P_1 will get notified of the failure by the error code of a MPI receive operation while P_2 will block.

Here we need a new robust message broadcasting mechanism. It should meet the following conditions:

- None of the processes will block if a failure occurs.
- Either all nonfailed processes receive the message successfully or none of them receive the message.

In this section, we present a robust message broadcasting mechanism for the commonly used row broadcast method in HPL: the increasing-2-ring-modified method (2rinM method).

Figure 5 illustrates how the 2rinM method works in a `row_com` with size $Q = 8$. Messages are first sent from the root P_0 to P_1. The $Q - 2$ processes left are divided into two groups: processes P_2 to $P_{(Q+1)/2-1}$ are in one group and processes $P_{(Q+1)/2}$ to P_{Q-1} in another. Messages are then sent to processes P_2 and $P_{(Q+1)/2}$, which act as sources of two increasing-one rings.

In our implementation, we designate a father node for each nonroot node. The messages of a father node should be transferred ahead of its son nodes. Each nonroot node sends the error code of its `MPI_Recv` to its father node. If its `MPI_Recv` failed, it should also initiate another `MPI_Recv` waiting to receive re-sent messages from its father node. As a father node, it receives the error code from its corresponding son nodes and then determines whether to resend a message again. We denote in Figure 6 the chronological sequences of message transmissions of Figure 5. The dashed arrows in Figure 6 point from son nodes to their father nodes, showing one possible father-son relationship. In this way, we can ensure that all processes will receive the messages successfully no matter which nonroot node is failed.

But what if the root P_0 fails before the first message is received successfully? If so, all processes will hang there except those that receive messages directly from the root node. To overcome this problem, we separate the first message transmission out of the row broadcast phase. The fault-tolerant mechanism presented above is performed only after the root P_0 send messages to P_1 successfully.

VI. Experimental Evaluation and Analysis

We performed three sets of experiments to answer the following questions about our ABFT hot-replacement method:

- How does the ABFT hot-replacement method perform?
- Is the ABFT hot-replacement method correct?
- How does the timing of process failures affect performance?

The experiments were performed on two different platforms. The first two sets of experiments were performed on a smaller cluster of 17 nodes. Each node of the cluster has four quad-core 2.2GHz AMD Opteron processors, connected by Gigabit Ethernet. The third set of experiments was performed on a larger cluster of 8 blades, 10 Intel Xeon X5650 processors each, 960 cores in total. In this cluster, nodes in the same blade are connected by InfiniBand, while different blades are connected with each other by a single InfiniBand cable. All nodes of the two clusters are running Linux operating systems. The MPICH package we used on both platforms is

TABLE I
EXPERIMENT CONFIGURATIONS ON THE SMALLER CLUSTER

Matrix Order	Number of nodes	Processes per node	Process grid with redt.
10,000	6	1	2 by 3
20,000	1	16	4 by 4
30,000	2	16	4 by 8
40,000	4	16	8 by 8
50,000	12	16	16 by 12
60,000	16	16	16 by 16

TABLE II
HPL EXECUTION TIME (IN SECONDS) USING DIFFERENT METHODS

Matrix Order	No Failures	Hot-Rep.	Rec.
10,000	577.74	591.61	602.73
20,000	1608.15	1677.14	1723.37
30,000	2601.03	2791.12	2821.43
40,000	3150.11	3358.38	3497.08
50,000	3433.09	3479.26	3577.77
60,000	4708.54	4722.44	4858.23

mpich2-trunk-r7834, and all the timings were measured by `MPI_Wtime`.

A. Performance of Different Algorithm-Based Fault Tolerance Methods

The first set of experiments was designed to compare the performance of the two different fault tolerance methods: ABFT recovery and ABFT hot-replacement. We incorporated both of them into HPL to handle a single process failure. Process failure was simulated by killing a process at a certain period of time after HPL started running.

The order of random matrix A and the number of computation nodes used in this set of experiments are listed in Table I. The matrix orders we used are of 10^4 magnitude. Table II reports the execution time of HPL using the two methods with one process failure and HPL without failure. The corresponding overheads of the two fault tolerance methods are shown in Figure 7. These overheads were calculated against HPL without failure. We can see that to handle a single-process failure, the execution time of HPL using ABFT hot-replacement is about 10–200 s shorter than that of HPL using ABFT recovery. The proportion of this part is about 1–5% of the execution time of HPL without failures. It should also be noted that the overhead of ABFT methods is affected by the mappings between processes and physical cores. If processes in the same row or column are mapped to physical cores in different nodes, the communication cost will be higher than that they all mapped inside the same node. Consider the fifth experiment where the matrix order is 50,000. One process row contains 12 processes, while one node has 16 physical cores. We can infer that there must be processes in certain rows are not mapped inside the same node. This could also explain why the overhead of ABFT methods in this experiment is higher than that of the last one, though the amount of data allocated to each process in this experiment is less.

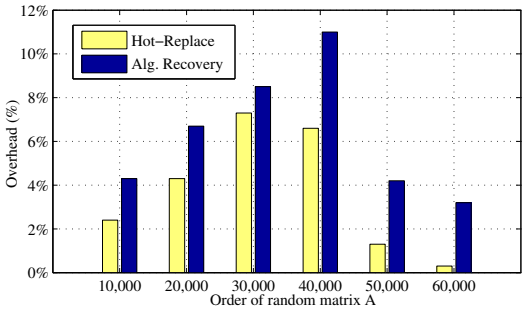

Fig. 7. Overhead of different ABFT methods (in seconds)

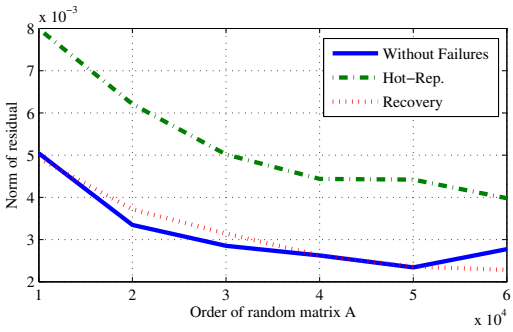

Fig. 8. Norm of residuals of HPL using different ABFT methods

B. Numerical Impact of Round-Off Errors in Different Algorithm-Based Fault Tolerance Methods

As discussed in Section II-A, ABFT techniques are based on floating-point arithmetic encodings and may therefore introduce round-off errors in HPL. Moreover, the ABFT hot-replacement method involves matrix transformation when replacing the dead process column by the redundant process column, which will make the round-off errors even greater. The experiments reported in this section were designed to measure the numerical impact of the round-off errors by checking the norm of residuals.

All experiment configurations are the same with the first one, listed in Table I. Figure 8 reports the norm of residuals at the end of each computation for the two ABFT methods with a single process failure, compared with the nonfailure case. One can see that the ABFT recovery method introduces a little round-off error into HPL whereas the ABFT hot-replacement method almost doubles the norm of residuals of HPL.

C. Performance of HPL using Hot-Replacement at Different Time Period

The third set of experiments was designed to evaluate the performance of the HPL using ABFT hot-replacement method when failure occurs at different times. The experiments were performed on the larger cluster with 8 blades. We used 75 of the nodes, 900 cores in total. We initiated one process per core and organized the 900 processes into a 30×30 grid. The matrix order we used was 200,000. The time of failure occurrence were controlled by the sleep function in python

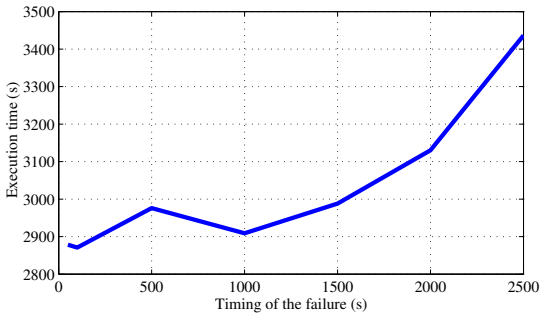

Fig. 9. Execution time of HPL using the ABFT hot-replacement method while varying the timing of the failure

scripts.

Figure 9 shows the execution time of HPL using the ABFT hot-replacement method while varying the timing of the failure. We can see that after 1,000 s, the total execution time increases radically. The reason is that, with the execution, the amount of factorized data is increasing. At the same time, the amount of data to be recovered, if a failure occurs, is also increasing. When that amount becomes too large to overlap the background recovery and the succeeding update phases, the total execution time will increase. However, the execution time of HPL using ABFT hot-replacement in the worst case should approach that of HPL using ABFT recovery theoretically.

VII. Conclusions and Future Work

In this paper, a new nonstop algorithm-based fault-tolerant scheme is proposed for a class of HPC applications, including matrix computations involving linear transformations. HPL has been modified to use the new fault-tolerant scheme with the support of MPICH's new fault tolerance features. Experimental results indicate that the proposed scheme is more efficient than existing algorithmic recovery methods.

Future work may include the following three directions. First, the proposed nonstop algorithm-based fault tolerance scheme should be implemented at large scale and under real circumstances, including handling multiple failures. Second, the round-off errors introduced by this fault-tolerant method when handling multiple failures at large scale should be further investigated. Third, this approach should be extended to the fault tolerance of other HPC applications.

Acknowledgments

This research was supported partly by the National Basic Research Program of China (973) 2011CB302502; by the National Natural Science Foundation of China under grants 60925009, 61003062, and 60921002 and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract #DE-AC02-06CH11357; by the DOE grant #DE-FG02-08ER25835.

References

[1] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, 1998.

[2] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, pp. 303–312, February 2006.

[3] "The computer failure data repository sites." http://cfdr.usenix.org. Last accessed February 2011.

[4] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, 2007.

[5] G. A. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," *CTWatchQuarterly*, vol. 3, November 2007.

[6] H. Naik, R. Gupta, and P. Beckman, "Analyzing checkpointing trends for applications on petascale systems," in *the Proceedings of the Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, (Vienna, Austria), Sep. 22 2009.

[7] 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*, pp. 213–223, 2005.

[8] Z. Chen and J. Dongarra, "Highly scalable self-healing algorithms for high performance scientific computing," *IEEE Transactions on Computers*, July 2009.

[9] Z. Chen and J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, December 2008.

[10] 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 2010.

[11] 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*, pp. 303–304, 2011.

[12] Z. Chen., "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2011.

[13] K. H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, pp. 518–528, June 1984.

[14] J. Anfinson and F. T. Luk, "A linear algebraic model of algorithm-based fault tolerance," *IEEE Trans. Comput.*, vol. 37, pp. 1599–1604, 1988.

[15] 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*, p. 76, 2006.

[16] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in *in Proceedings, LACSI Symposium, Sante Fe*, pp. 479–493, 2003.

[17] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in *In Workshop on Dependable Parallel, Distributed and Network-Centric Systems(DPDNS), in conjunction with IPDPS*, 2007.

[18] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols," *Future Generation Computer Systems*, vol. 24, no. 1, pp. 73 – 84, 2008.

[19] G. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (J. Dongarra, P. Kacsuk, and N. Podhorszki, eds.), vol. 1908 of *Lecture Notes in Computer Science*, pp. 346–353, Springer Berlin / Heidelberg, 2000.

[20] W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.

[21] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu, "MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware," *Cluster Computing*, vol. 7, pp. 303–315, 2004.

[22] "Top 500 supercomputing sites." http://www.top500.org. Last accessed February 2011.

[23] "MPI-Forum fault-tolerance working group." https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_users_guide. Last accessed February 2011.

[24] "HPL benchmark sites." http://www.netlib.org/benchmark/hpl. Last accessed February 2011.