DOI: 10.14529/jsfi160101

NR-MPI: A *N*on-stop and Fault *R*esilient *MPI* Supporting Programmer Defined Data Backup and Restore for E-scale Super Computing Systems

***Guang Suo*[1], *Yutong Lu*[1], *Xiangke Liao*[2], *Min Xie*[1], *Hongjia Cao*[1]**

Fault resilience has became a major issue for HPC systems, particularly, in the perspective of future E-scale systems, which will consist of millions of CPU cores and other components. MPI-level fault tolerant constructs, such as ULFM, are being proposed to support software level fault tolerance. However, there are few systematic evaluations by application programmers using benchmarks or pseudo applications. This paper proposes NR-MPI, a *N*on-stop and Fault *R*esilient *MPI*, supporting programmer defined data backup and restore. To help programmers write fault tolerant programs, NR-MPI provides a set of friendly programming interfaces and a state transition diagram for data backup and restore. This paper focuses on design, implementation and evaluation of NR-MPI. Specifically,this paper puts emphases on failure detection in MPI library, friendly programming interface extending for NR-MPI and examples of fault tolerant programs based NR-MPI. Furthermore, to support failure recovery of applications, NR-MPI implements data backup interfaces based on double in-memory checkpoint/restart. We conduct experiments with both NPB benchmarks and Sweep3D on TH supercomputer in NSCC-TJ. Experimental results show that NR-MPI based fault tolerant programs can recover from failures online without restarting, and the overhead is small even for applications with tens of thousands of cores.

Keywords: Message passing interface, fault tolerant MPI, NR-MPI, Application-level Checkpoint/Restart..

Introduction

Large scale scientific applications have been the main driving force for high-performance computing. Scientists need to analyze ever-larger data set and to run ever-larger simulations, which drives the scale of high-performance computers, growing to millions of processor cores. In the future, extreme scale high-performance computers will consist of even more cores. From the top500 [1] list of November 2015, there are 3120000 cores in the rank of 1 supercomputer. With the expansion of computer systems, the failure rate is also increasing. So the mean time between failures (MTBF) is decreasing. However, many scientific applications need to run for weeks or even months. Therefore, the MTBF of these computers is becoming significantly shorter than the execution time of many current scientific applications. To support the execution of such applications, fault tolerance is imperative [2].

The lack of appropriate resilience solutions is a major problem at exascale. Currently the MPI forum is working on a new fault tolerant MPI standard. Additional MPI-level constructs will be added into the future MPI standard. The most promising way is User Level Failure Migration(ULFM) [3]. However, there is still lack of enough experimental results to proof both the usability and performance of ULFM.

In this paper, we present NR-MPI, a Non-stop and Fault Resilient MPI. The semantics of NR-MPI is derived mainly from FT-MPI and ULFM. The programming interface, which is designed for iteration based on scientific parallel applications, is less complicated than FT-MPI

[1]State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, Hunan Province, China

[2]Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha, Hunan Province, China

and ULFM. The interface is more suitable for programmers, who just put emphasis on which data to be backuped and when to backup, to convert a parallel application into a fault tolerant one. The failure detector of NR-MPI is different from ULFM. ULFM detects the process failures in the MPI library, while NR-MPI relies on external failure detector which is usually integrated with process manager or resource manager. We implemented some fault tolerance of ULFM based on MPICH. The implementation of NR-MPI is based on MPICH [4]. NR-MPI has no runtime overhead when there are no failures.

This paper focuses on the design, implementation and evaluation of NR-MPI, which is implemented on top of ULFM. The MPI Forum has not reached a consensus on the principles of a resilient MPI, although ULFM is discussed a lot. However, we think that the following issues are important when implementing a fault tolerant MPI.

- How to detect failures in a MPI library.
- How to recover the state of a MPI library based on ULFM.
- How to recover the lost application data after failures using NR-MPI.
- What programming interfaces are needed in order to reduce the complexity of fault tolerant programming.
- How to convert a non-fault tolerant program into a fault tolerant one by NR-MPI.

The rest of this paper is organized as follows: Section 1 presents the design and implementation of NR-MPI. Section 2 shows the usage of programming interface based an example algorithm. Section 3 evaluates the performance of NR-MPI with NPB benchmarks. Section 4 gives the related work of NR-MPI. In Section 4.2, we conclude this paper and discuss the future work.

1. Design and Implementation of NR-MPI

NR-MPI is designed on top of ULFM and implemented based on MPICH [4]. Traditional ULFM is implemented based on OpenMPI. We use MPICH instead of OpenMPI because the software stack of the high performance network is on top of MPICH. The fault tolerant RMS is modified based on SLURM [5].

1.1. Failure Model

In this paper, we assume a failure model in which fail-stop failures can occur anytime in any process during a parallel execution. There are two types of failure models: fail-stop model and byzantine model. Fail-stop failures can be detected more easily. In fact, byzantine failures can be detected by error checking based on ABFT, which can also be implemented based on NR-MPI.

1.2. The Fault Tolerant RMS of NR-MPI

The Resource Management System, for example SLURM, is developed to manage and monitor parallel applications running on a cluster of computers. It is designed to coordinate a global and consistent system state upon failures. According to roles and locations, the RMS can be divided into two parts: Resource Manager and Process Manager, shown in fig. 1. Furthermore, we add Failure Arbiter (FA) and Failure Detector (FD) to them respectively. The functions of the fault tolerant RMS are: fault tolerant resource management, failure detection and notification.

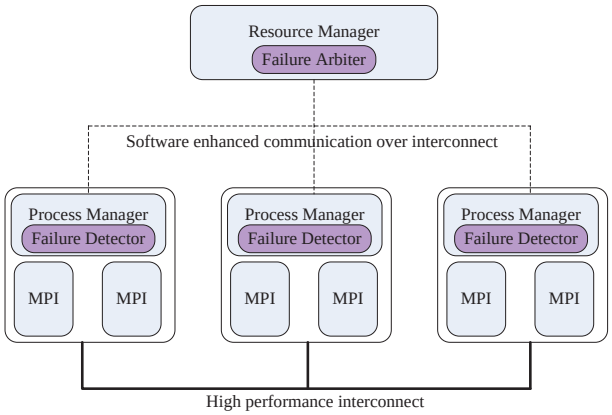

Figure 1. Structure of the resource management system of NR-MPI

For NR-MPI, failures are detected by the fault tolerant RMS. To detect fail-stop failures, we add FDs in Process Managers of RMS in computing nodes and add FA in Resource Manager. FDs can detect process failures of parallel jobs using the `SIGCHLD` signal. FA uses a periodic heartbeat to detect failures from FDs. In this way, FA can detect all process failures of parallel jobs. There are two advantages, if the FDs and FAs are integrated into RMS. Firstly, they can be light weighted so as to not interfere with the performance of jobs. Secondly, FD and FA can make use of the fault tolerant techniques already implemented in the RMS. For example, there are two active Resource Managers on line. One is active while the other one is standing by, so that failures of one Resource Manager don't influence the availability of the system.

Based on the communication topology of Resource Manager and Process Manager, the communication topology of failure detecting system is a tree too. Root of the tree is FA; inter-mediate and leaf nodes are FDs. Shared memory is used for the communication between FD and MPI processes in the same node. Software enhanced communication network for high performance computing used for the communications between FD and FD (or FA), so that the heartbeat and failure notification messages can be unfailingly transferred. In addition, to recover sucessfully, the failure notification messages should be received by all the MPI processes of a parallel job in the same order.

During MPI communications, the NR-MPI library needs to check the failure notification messages by reading the contents of shared memory, when sending or receiving data. For example, based on MPICH, NR-MPI checks the failure notification messages in progress engine.

The contents of the failure notification message are the list of failed ranks of the parallel job. This failure information can help the programmers to recover from process failures, node failures and network failures. When a node crashes, the message contains all the ranks belonging to the program on that node. When a part of network fails, the message contains all the ranks of processes which FA can't communicate with due to the network failure.

1.3. Recovery of NR-MPI Library

Main job of recovering the NR-MPI Library is to recover the communicators. We take MPI_COMM_WORLD as an example, shown in fig. 2, to explain how to recover it inside the library. The world communicator recovery is to repair the corrupted attributes. In fact, old communication context is OK for the repaired communicator. So recovering group and virtual

connection table is the main job. It can be done together by using mainly existing MPI procedures.

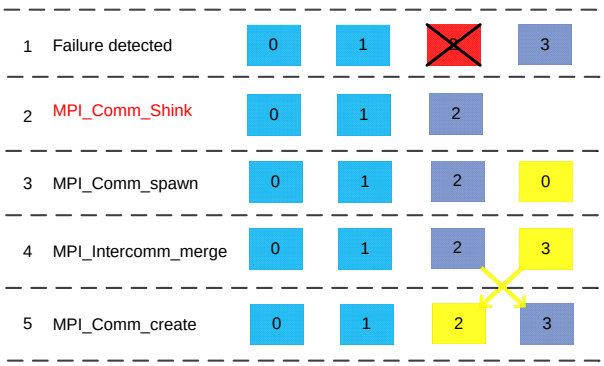

Figure 2. Recovery process of a communicator

The steps of world communicator are as follows.

(1)Failure detection. When failures occur, the other alive ranks enter the failure recovery process as soon as they receive failure information from FDs.

(2)Shinking the communicator. The alive ranks shink the failed ranks from world communicator by calling MPI_Comm_shink. The MPI_Comm_shink operation is defined in ULFM. It will exclude the failed ranks from a failed communicator.

(3)Spawning replacements. The world communicator spawns processes as replacements for the failed ones by calling MPI_Comm_spawn. Note that the spawned processes are in a different process group. They communicate via an inter-communicator.

(4)Merging the inter-communicator. Constructing a new world group from the old world group and the spawned group. Note that the order of new world group is different from the world group before the failures occur.

(5)Reordering rank in a new world group. This can be done by calling MPI_Comm_create.

Most of the steps, except MPI_Comm_shink, of world communicator recovery are based on existing MPI standard version 2. Meanwhile, the recovery process does not manipulate virtual connection table directly. So the complexity of implementing NR-MPI is relatively low.

1.4. State Transition Diagram with Failures

Upon failures, the lost application data also need to be recovered. C/R, ABFT, application level data backup via MPI communications or combination of the above can be used to recover the lost data. In this paper, we don't assume a specific data backup and restore technique. Instead, we define a data backup and recovery protocol for NR-MPI, so that the data backup and recovery techniques defined by programmers can be integrated with NR-MPI as a whole.

During the execution of a NR-MPI parallel program, any processes may fail. In addition, the failures may occur when recovering MPI data, or application data. Meanwhile, a failure is recovered, if and only if all processes recovered their MPI data and application data. The data to recover is different for different kinds of processes at different state. To help programmers implementing NR-MPI parallel programs, we define a state transition diagram of a NR-MPI process, shown in fig. 3. There are 8 states in the diagram, defined in Table 1.

For any MPI parallel programs (including NR-MPI based programs), regardless of failures, they have four states: INIT, RUNNING, FINISH, and ABORT. The other 4 states are the

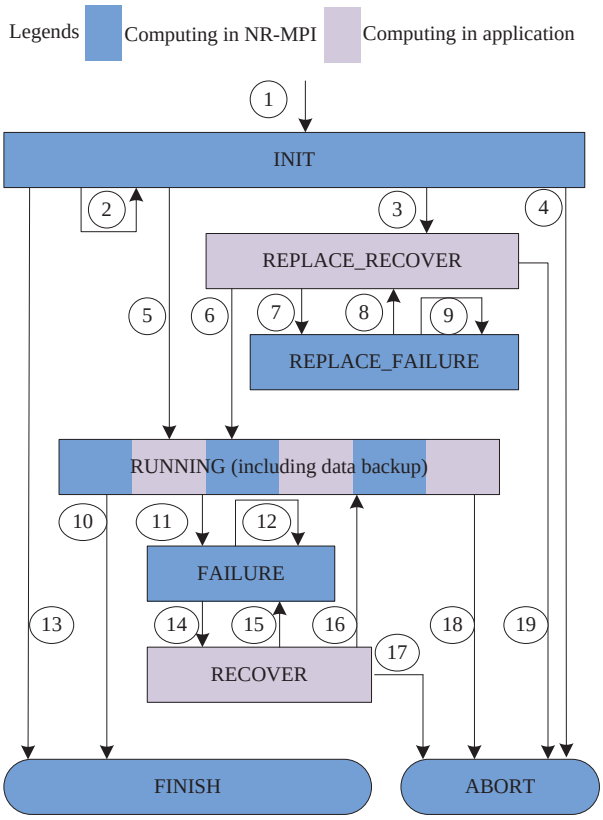

Figure 3. State transition diagram of a NR-MPI process

states to recover MPI core data, MPI extend data and application data for normal processes and replacements. The RUNNING state has two colors because it can be in both MPI library and programmers' codes.

Table 2 summarizes the interactions between the states in fig. 3. State Transfer 2, 9 and 12 are self-to-self, indicating that failures occur in INIT, FAILURE and REPLACE_FAILURE. It means that the stand-by process, the replacements and the normal processes find failures when they recover MPI core data respectively.

When data recovery is via MPI communications, failures can be detected in progress engine (the module that probes upcoming messages and sends queued messages) of MPI library. However, when there is no interaction between processes during data recovery, the processes won't enter REPLACE_RECOVER, so the state transfer 7, 8, 9 and 15 won't be triggered either.

1.5. Extending Interface of NR-MPI

Many parallel applications are iteration based, such as linear solvers and PDE based applications. We focus on fault tolerance for iteration based applications. To help programmers, NR-MPI provides a convenient programming interface which can reduce the burden of programmers, listed in Table 3.

NR_Register, NR_Backup, NR_Recover and NR_Need_-backup are used to backup and restore the application data based on the double in memory checkpoint [6]. Moreover, to prevent the damages of failures during the data backup and restore, we use ping-pong buffers to backup application data. NR_Backup and NR_Recover are simple examples for application level data backup and restore. There can be any data backup and restore techniques, and programmers can implement their own methods by overriding the 4 functions. NR_Get_state and NR_Set_state are

Table 1. STATE DEFINATION

States	Descriptions
INIT	When a process calls MPI_Init, it enters INIT state. In INIT state, the processes, including standby processes, initialize their internal MPI data. At the end of INIT, the normal processes can exit, while the standby processes are waiting and processing failure notification messages. The standby processes exit INIT when they are selected as replacements upon failures or the program exits.
RUNNING	If the return status of MPI_Init indicates that initialization is OK, or if a replacement process recovers its lost data successfully, the process enters RUNNING state. A process can do user computation, user defined MPI communication, and backup application data (via C/R, ABFT, or MPI communication) in RUNNING state.
FINISH	When a process calls MPI_Finalize, it enters FINISH state. FINISH state is an absorbing state.
ABORT	Whenever a process finds unrecoverable failures, it enters ABORT state. To simplify the diagram, we only draw 4 state transfers to ABORT. ABORT is also an absorbing state.
FAILURE	When a normal process in RUNNING state finds failures in MPI library, it enters FAILURE state. This state is hidden in MPI library; the work in this state is to recover MPI core data
RECOVER	After recovering MPI core data, a process enters RECOVER state. The work in this state is to recover MPI extend data and application data by programmers. If successful, the programmers should alter the state of the local process to RUNNING explicitly.
REPLACE_ RECOVER	When a replacement process exits from INIT, it needs to recover MPI extend data and the lost application data. If recovering successfully, programmers need to alter the state of the local process to RUNNING explicitly.
REPLACE_ FAILURE	When failure occurs during REPLACE_RECOVER, the process enters REPLACE_FAILURE. This state is hidden in MPI library; the work in this state is to recover the MPI core data.

used to get and set the state of the local process. NR_Get_failure_ranks is used to query failed rank set in NR_Recover.

1.6. Implementation of the NR-MPI

The structure of NR-MPI is shown in fig. 4. Clearly, NR-MPI is in the middle of the software stack. The fault tolerant RMS is modified based on slurm-2.4.0-rc1[XX]. Many different MPI implementations have been developed to support different supercomputers efficiently. NR-MPI can integrate with a wide range of existing MPI implementations. In this paper, NR-MPI is modified based on mpich2-1.4.1p1[21], integrated with state management module, data backup and restore module, failure detecting module, and failure recovery module. Their functions are as follows: state management module provides a state managing interface for programmers. From state management module, programmers can query or set its own fault tolerant state, which is essential for NR-MPI. Data backup and restore module provide the programming interface based on mutual data backup to save the application data. Failure detecting module can query failures

Table 2. STATE TRANSFER DEFINATION

ID	State Transfer Descriptions
1	When a progress calls MPI_Init.
2	When failures occur during the creation of MPI core data for a replacement process. Or when stand-by processes detect failures.
3	When a replacement exits from MPI_Init, it enters REPLACE_RECOVER.
4,17,18,19	When an unrecoverable failure occurs.
5	When a normal process exits from MPI_Init, it enters RUNNING.
6	When a replacement finishes recovering its app data.
7	When a replacement encounters failures during the process of recovering app data, it enters REPLACE_FAILURE.
8	When a replacement recovers its MPI core data.
9	When a replacement encounters failure during the process of recovering of its MPI library data
10	When a normal process calls MPI_Finalize.
11	When a normal process encounters a failure.
12	When a failed normal process encounters failure during the process of recovering of MPI core data
13	When all normal processes enter FINISH, the stand-by processes enter FINISH.
14	When a failed normal process finishes recovering its MPI library data
15	When a normal process encounters failures during the process of recovering of app data
16	When a normal process finishes recovering the app data.

Table 3. PROGRAMMING INTERFACE

Interface	Descriptions
NR_Register	register the data to be backed up
NR_Backup	backup the registered data
NR_Recover	recover the lost data which had been backed up
NR_Need_backup	determine whether data backup is needed
NR_Get_state	get the state of the caller process
NR_Set_state	set the state of the caller process
NR_Get_failure_ranks	get the set of ranks failed in last failure

of the system efficiently from the external failure detector. When the processes are spawned by process manager, the process manager creates a shared memory used to pass failure notification messages and adds FD_SHMID=shmid into the environmental variables of spawned processes. Failure recovery module can recover the corrupted MPI core data into a consistent state.

2. NR-MPI USAGE EXAMPLE

Based on NR-MPI, the programmers just add two sections to a non-fault tolerate program to make it fault tolerant, without changing other codes of the program. The function of data backup segment is to save the application data periodically in case of failures, while the function of data restore segment is to restore the lost application data of failed processes.

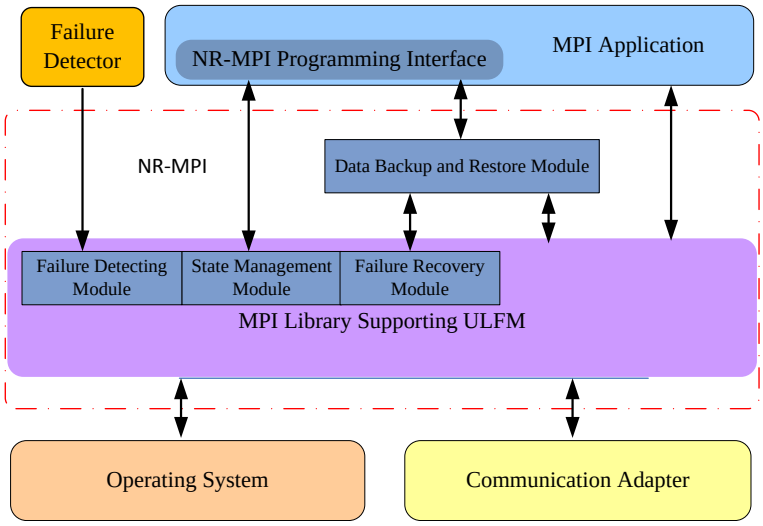

Figure 4. Structure of NR-MPI

We take the conjugate gradient (CG) algorithm as an example to illustrate the usage of the programming interface, shown in fig. 5. NR-CG is the fault tolerance version of CG algorithm based on NR-MPI. To support fault tolerance, we added data backup and restored codes. Lines of 19∼29(data restore segment) and 39∼42(data backup segment) are the additional codes.

From the algorithm of NR-CG (lines 33∼38), we can see that vector x and iteration index j are the data to be backed up. So after initialization of MPI library, NR-CG registers x and j to NR-MPI (line 11∼12). Then, NR-CG sets the return position env, so that NR-CG can switch from MPI library to the position user defined in the program after the world communicator has been recovered in a failure (line 13∼14). The callback function *cg_callback* is called by NR-MPI after recovering the MPI core data. In line 15, NR-CG gets the current state of itself. Then, it sets x and j_start based on the program state. If the state is RECOVER or REPLACE_RECOVER, the program needs to restore the application data. For the two states, the codes are the same. However, they may be different for ABFT. Lines 39∼42 are used to back up data. Usually it is not necessary to back up data per iteration, so NR_Need_backup is called to determine whether the backup is needed.

In this example, NR-CG uses longjmp to get to the user-defined position in the program. In fact, NR-MPI can also return FAILURE status like the interfaces defined in FT-MPI, so that setjmp and longjmp can be omitted. We didn't follow the style of returning error codes, because more modifications are needed. Furthermore, programmers can call any MPI routines after recovering MPI core data. For example, MPI_Comm_create can be used to create a new communicator, which doesn't contain the recovered ranks, like shrink operation in FT-MPI and OpenMPI.

After receiving notification from FDs, the survival processes recover MPI core data before calling *cg_callback*. In *cg_callback*, the default action of *cg_callback* is jumping to line 13. *longjmp* can clear the calling stack of MPI library. At this moment, the state of the program is RECOVER. So lines 26∼29 are executed by the survival processes, they will help the failed processes to restore their lost application data, when they find their partner is failed. Meanwhile, the replacements exit from INIT and run to line 15. Their states are REPLACE_RECOVER, so they get x and j_start from their partners (lines 22∼25). At last, they set their states to RUNNING.

```
1   void cg_callback(jmp_buf env){
2       if (env) longjmp(env, 1);
3       else MPI_Abort();
4   }
5   int main(){
6       int j, j_start, j_end, rank, jhandle, xhandle, error, state;
7       float a, b, c;
8       float x[L], r[L], b[L], p[L], A[L,L];
9       MPI_Init();
10      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11      jhandle = NR_Register(&j, partner);
12      xhandle= NR_Register(&x, partner);
13      error = setjmp(env);
14      NR_Set_callback(cg_callback, &env);
15      NR_Get_state(&state);
16      if (state == RUNNING){
17          j_start=0;
18          x=initial value; }
19      if (state == ABORT) {
20          MPI_Abort(MPI_COMM_WORLD, error);
21          return 0;}
22      if (state == REPLACE_RECOVER) {
23          NR_Recover(jhandle, &j_start);
24          NR_Recover (xhandle, &x);          Data Restore
25          NR_Set_state(RUNNING); }           Section
26      if (state == RECOVER) {
27          NR_Recover (jhandle, &j_start);
28          NR_Recover (xhandle, &x);
29          NR_Set_state(RUNNING); }
30      r=b - A x
31      p= r
32      for (j = j_start,…, j_end) {
33          c=(r, r)
34          a=c/(A p, p)
35          x= x +a p
36          r= r - a p
37          b=(r, r)/c
38          p= r + b p
39          if (NR_Need_backup(j)){
40              NR_Backup( jhandle);           Data Backup
41              NR_Backup(xhandle);            Section
42          }
43      }
44      MPI_Finalize();
45      return 0;
46  }
```

```
sum = 0;
for(i=1; i<L; i++)
    sum+=r[i]*r[i];
MPI_Allreduce(&sum, &c,
        MPI_FLOAT, 1, MPI_SUM,
        MPI_COMM_WORLD);
```

Figure 5. Algorithm of NR-CG based on NR-MPI

Figure 6 shows an execution process of NR-CG upon a failure. There are 5 processes in NR-CG, 4 of them are normal process and the other one is the stand-by process. During execution, P0 and P2 backup the application data of each other, and P1 and P3 backup the application data of each other. At phase 1, the 4 processes backup their application data successfully. At phase 2, after backing up their application data, P2 crashed. Then the event, which P2 is crashed,

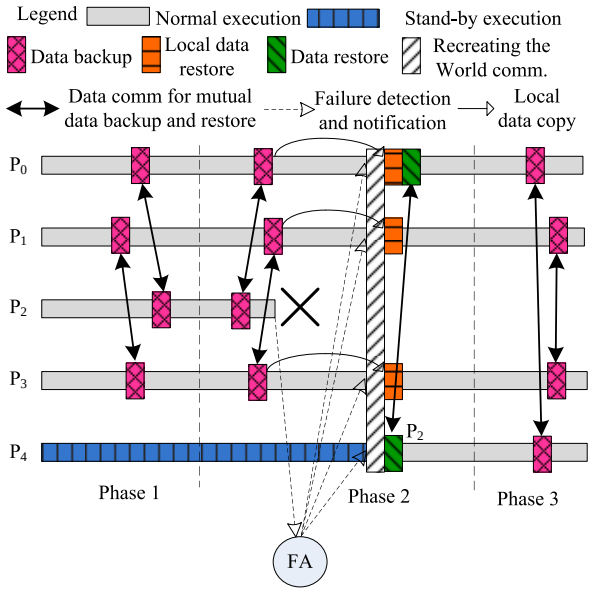

Figure 6. Execution example of NR-CG

is detected by FA. FA notifies the other 3 processes of the failure. The recovery process is as follows:

(0)P0, P1, P3 spawn a new process, which is named P4.

(1)P4 finds that the replacement of P2 is itself. Then P0, P1, P4 and P3 recover their MPI core data (creating a new world communicator).

(2)After recovering MPI core data, the 4 processes begin to recover the application data. P0, P1 and P3 recover their local application data to the last backing up version.

(3)P0 finds that its partner (P2) has failed by calling NR_Get_failure_ranks, so it sends application data of P2 to help it to recover. Meanwhile, the new P2 receives its lost application data.

(4)All of the four processes recover their data to the latest backing up version. And the NR-MPI parallel program recovers from a failure.

3. EXPERIMENTAL EVALUATION

To use NR-MPI, we have modified benchmarks from NAS Parallel Benchmarks [7] (version 3.3) and Sweep3D [8]. Our experimental platform, configuration of which is shown in Table 4, is TH-1A [9, 10], deployed in National Supercomputer Center in Tianjin.

Table 4. CONFIGURATION OF EXPERIMENT PLATFORM

Component	Configuration
CPU	X5670 CPU, 2.93GHz, 6 core, 2 CPUs per node
Memory	48GB
Compiler	Intel C++ Compiler 11.1
Interconnect	bi-bandwidth 160Gbps, MPI bandwidth 6340MB/s

There are two steps to modify a non-fault tolerant program. Firstly, to indentify main iteration and application data to backup. Secondly, to add data restore segment and data backup

segment before and inside the main iteration, if necessary. Moreover, if a program has several phases, that is to say, it has several main iterations. For each one of the main iterations, the programmers need to identify application data, to add data restore and to backup segments respectively. So the modification complexity is very low. For example, we have added only a few additional codes to CG in NPB-3.3 to enable fault tolerant.

3.1. Runtime Overhead of Fault Resilient NPB Benchmarks

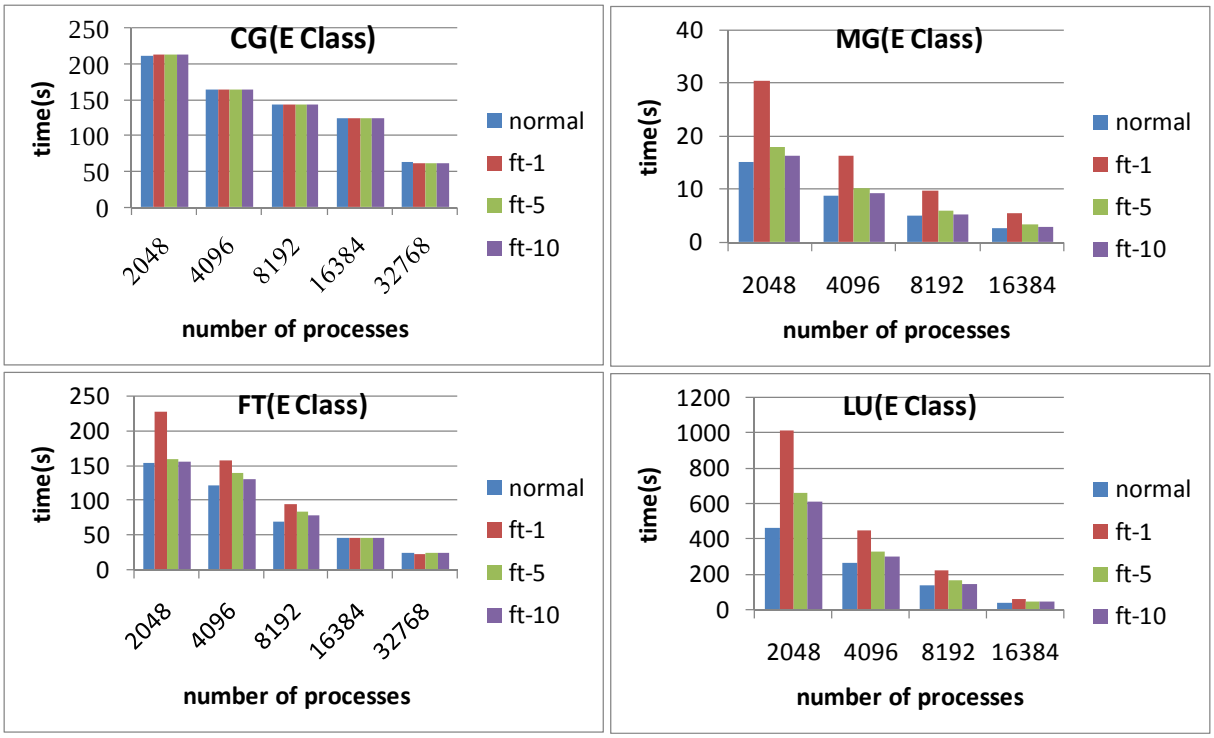

Figure 7. Runtime overheads of E class NPB benchmarks without failures

When there is no failure, the overhead of a MPI program without data backup and restore is the failure detection overhead. In our implementation, the failure detection overhead is almost zero based on experiments. So we don't provide the results of failure detection overhead. fig. 7 presents the execution time of non-fault tolerant and fault tolerant NPB benchmarks with NR-MPI. For fault tolerant version of the benchmarks, the application data is backed up every 1,5 and 10 iterations, using double in memory checkpoint. Normal results are the experimental results without data backup. ft-1 is the experimental result, in which the application data is backed up every 1 iteration. ft-5 is the experimental result, in which the application data is backed up every 5 iteration. ft-10 is the experimental results in which the application data is backed up every 10 iteration. The iteration intervals are different for different benchmarks, so time interval of data backup is different for different benchmarks.

From fig. 7, it can be found that the runtime overheads of NR-MPI parallel programs are different from different benchmarks. For NR-CG, the runtime is almost independent from the backup interval. The reason is that the data amount to be backed up is really small compared with the communications during the execution. However, for NR-MG, NR-FT and NR-LU, the overheads are higher than NR-CG. Take NR-LU as an example, the runtime of ft-1 is 122% higher than normal execution for 2048 processes. However, when the backup interval increases,

the overhead due to the data backup is decreasing. For NR-LU, the runtime of ft-10 is by 33 higher than the normal execution for 2048 processes. NR-MG, NR-FT and NR-LU have higher runtime overhead, because the data to be backed up is larger. For example, when normal execution, the LU benchmark only exchanges the surface data of a data cube. However, NR-LU needs to backup the entire cube every 1, 5 or 10 iterations. So the runtime overhead of data backup is relatively higher than NR-CG. When the number of processes increases, the runtime overhead also decreases. For example, the runtime overhead of NR-LU is 15%, when the backup interval is 10 iterations for 16384 processes. The runtime overhead of NR-FT is 2%, when the backup interval is 10 iterations for 32768 processes.

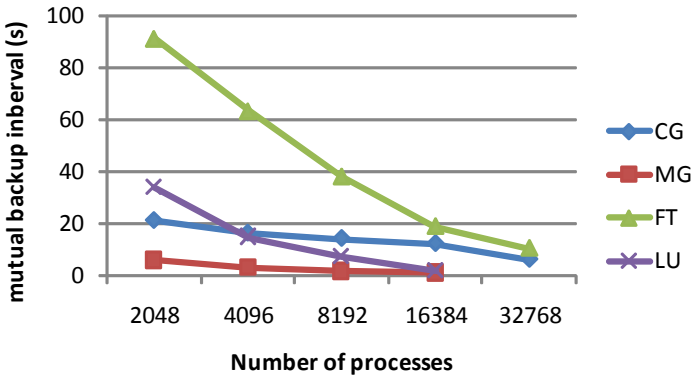

Figure 8. Backup intervals of E class NPB benchmarks

The backup interval (similar to the checkpoint intervals) is another important criterion for NR-MPI parallel programs, because it can be used to measure the expected lost computation due to one failure. Fig. 8 shows the backup of ft-10 intervals of E class NPB benchmarks. It can be found that the intervals are really small. For example, the highest interval is 90 seconds, for FT.E.2048, which means the average lost of computation is 45 seconds. For the other benchmarks, except NR-FT, the highest interval is 20 seconds.

3.2. Overall Time Overhead due to One Failure

NR-MPI is similar to SCR in data backup and restore. For example, SCR also supports mutual file backup when checkpoint files are stored in local disk or memory disk. In addition, when using SCR, the programmers also need to specify the variables to be saved, just like backing up key application data, when programmers use NR-MPI.

However, there are still two differences between SCR and NR-MPI. Firstly, the data backup and restore efficiency of NR-MPI is higher than SCR, because SCR uses file system interface to save and restore data, which needs additional copies between programs and the operating systems. Meanwhile, SCR usually uses TCP/IP-based communication channels. It cant use fast interfaces, such as RDMA, to accelerate communication directly. Secondly, in a failure, SCR needs restarting jobs, which is a time consuming operation in large parallel systems.

Table 5 gives the overall timing analysis of SCR and NR-MPI based on Sweep3d. We have modified two versions of Sweep3d: SCR-Sweep3d and NR-Sweep3d. The two versions save the same data during execution using different interfaces. The grid size of Sweep3d is 1024x1024x 16384 and the parallelism is 16384. The iteration count is 10, and data is saved or checkpointed per iteration.

Table 5. OVERAL TIMING ALALYSIS OF SCR AND NR-MPI

Time (seconds)	SCR- Sweep3d	NR-Sweep3d
Time to detect failure	60	60
Time to backup data	3	2.5
Rebuild or route data (SCR)	1	0
Time to re-launch job(SCR)	10	0
Time to recover MPI library.(NR-MPI)	0	0.1
Time to read application data	5	2
Average lost computation	11	11
Total	90	75.6

The time to detect a failure is two times of the heartbeat interval. In the experimental system, it is about 60s. The way to backup application data is different for the two versions. One is via SCR interfaces, while the other one is via MPI communications. So NR-Sweep3d is faster than SCR-Sweep3d for backing up data. Rebuilding or routing data is specific for SCR, so do restarting jobs. While recovering MPI library is only needed by NR-MPI. The time to read application data after failures is different for the two versions. All processes of SCR-Sweep3d need to read some application data from parallel file systems except the application data in local memory disk, while only the replacement processes of NR-Sweeep3d need to receive lost application data and read lost application data from file systems. The average lost of computation is the same for the two versions, because they all backup data per iteration. In conclusion, NR-Sweep3d is better than SCR-Sweep3d.

4. RELATED WORK

4.1. Fault Tolerant Techniques

In our previous work [11], we all also present NR-MPI, which is implemented based on MPICH. The failure recovery of MPI library is based on our own fault tolerant MPI constructs. In this paper, the failure recovery of MPI library is based on ULFM. In fact, there is few performance difference between this work and previous work.

There are several fault tolerant techniques, such as Checkpoint/Restart [12] (C/R for short), Message logging [13, 14], process-level replication [15] and forward recovery [16]. C/R, which has been used widely in previous high performance computers, periodically saves the state of a computation to stable storages, such as parallel file systems. However, C/R requires a restart of the entire parallel job even when only one process of the job failed. In a restart, all processes of a parallel job need to be reloaded. Then, all processes read the latest checkpoint to recover a consistent state. Both the overheads of checkpoint file I/O and restart overhead are unbearable for the current large scale systems, letting alone for the future extreme large scale systems. Log based methods, such as MPICH-V [17], can recover the process to its initial state and roll it forward by re-playing the messages before failures in the same order they were delivered before the crash. However, the main limitation is the necessity to log all messages of the execution. Process-level replication of parallel executions, such as MrMPI [18] and RedMPI [19], employs replicated processes performing the same task. If a process fails its a replication can take over its execution. Thus, redundant copies can decrease the overall failure rate. One major replication

overhead comes from the management of extra messages required for replication. For a double-replication execution, when a process sends a message to another process, four communications of that message take place. Forward recovery, such as FT-MPI (and OpenMPI [20, 21]) and User-Level Failure Mitigation (ULFM), allows applications continue running after a failure, while standard MPI does not provide any specification of the behavior of an MPI application after a failure. FT-MPI allows the semantics and associated failure modes to be completely controlled by the applications. FT-MPI required programmers to recover all the application state after failures. In addition, Harness [22], a distributed virtual machine, is used as the runtime system, in the initial implementation of FT-MPI. Both the programming interface and runtime system of FT-MPI are too complicated to use. ULFM allows the application to get notifications of errors and to use specific functions to reorganize the execution for forward recovery. However, ULFM provides only the basic interface and new semantics to enable applications and libraries to repair the state of MPI and tolerate failures. It is just like an assemble language for fault tolerance and is a little complicated for programmers to write fault tolerant applications.

In addition, there are some other hybrid ways. M^3 [23] is a user-transparent checkpoint system for fault tolerant MPI without restarting jobs. Coordinated checkpoint is used to recover lost data upon failures. Job pause service [24], Adaptive MPI [25], StarFish [26] and LAM/MPI [27] are also similar with M^3, using system level checkpoint to recover MPI runtime system. User-Directed Fault Tolerance (UDFT) [28, 29], on top of standard MPI, provides the user directed support for application level algorithmic fault tolerance.

4.2. Data Recovery for HPC

C/R is the most typical technique for fault tolerance in previous researches. There are two types of C/R. Traditional C/R can tolerant the whole system failures by writing checkpoint data periodically to stable storages, such as parallel file systems, and restarting from the latest checkpoint. While diskless C/R [30] saves the states of the processes into the memory directly, eliminating the overhead of writing data to stable storages. Diskless C/R is faster, but traditional C/R can recover from more serious failures. C/R can be implemented at two levels: application-level checkpoint [31] and system-level checkpoint [32]. SCR [33] reduces the checkpoint and restart overheads by caching checkpoint data in the memory or local storage of the compute node. However, the spatial overhead of SCR is triple size of the checkpoints at least. This is a huge overhead for future supercomputers which have a lower memory/processor ratio. In addition, SCR also needs a restart of the entire parallel job. Usually, the overheads of restarting jobs are directly proportional to the parallelism of jobs. Checkpoint-on-Failure Protocol [34] can reduce checkpoint overhead by eliminating the overhead of customary periodic checkpointing. Using NR-MPI, Checkpoint-on-Failure can reduce restart overhead furthermore.

Algorithm-based Fault Tolerance (ABFT) can also be used to recover lost data due to failures. Huang and Abraham [35] developed the ABFT technique to detect, locate and correct soft failures. For many matrix operations, the checksum relationship in the input checksum matrices is still held in the final computation results. Therefore, the soft failures can be detected by checking the checksum relationship in the final computation results. If some processes fail during computation, the data is lost. Based on the checksum relationship, the lost data can be rebuilt using the data of the survival processes. For example, Davies [36] proposed an algorithm-based recovery scheme for the HPL benchmark, based on the checksum relationship of the right-looking LU factorization algorithm. The checksum is maintained at every step of the

computation. Chen [37] found that, for many iterative methods, if the data partitioning scheme satisfies certain conditions, the iterative methods will maintain enough inherent redundant information for the accurate recovery of the lost data. Yang [38] proposed a new application-level fault-tolerant approach for parallel applications called the Fault-Tolerant Parallel Algorithm, which provides fast self-recovery upon failures. In a failure, all survival processes re-compute the workload of the crashed processes in parallel. However, it requires programmers to redesign algorithms.

Conclusion

This paper proposes a convenient, scalable and efficient fault tolerant MPI, named NR-MPI. By the notifications from fault tolerant RMS, failures are internally and automatically recovered by the NR-MPI runtime system. On the one hand, NR-MPI eliminates the restarting job overhead by automatic online recovering MPI communication states after failures for the survival processes. On the other hand, NR-MPI reduces the complexity of fault tolerant MPI by designing new semantics of MPI. For example, duplicate messages and termination of programs do not to be detected any more. We carried detailed experiments to evaluate NR-MPI. The experimental results, from 2048 processes to 32768 processes, show that NR-MPI could be scalable soundly.

NR-MPI also has limitations. Firstly, not all failures can be recovered. Not enough communicator contexts, not enough replacements, or the death of the two partners backing up data each other can cause of unsuccessful recovery. Secondly, programmers need to modify programs to using NR-MPI. Thirdly, upon failures, programmers have to rollback to a consistent position in the NR-MPI parallel programs. However, the lost computation due to rollback is controlled by programmers. Fourthly, NR-MPI assumes that the crashed processes are necessary for the parallel job, so it reinitializes replacements upon failures. In fact, if replacements are not needed, they can be excluded from the recovered world communicator by calling MPI_Comm_create.

In the future, our work focuses on: 1)lazy allocation-based failure recovery, which requires the survival processes to spawn replacements when necessary. 2)more flexible data recovery algorithms to reduce the memory overheads of double in memory checkpoint. 3)combining NR-MPI with the new de facto message passing standard, to support fault tolerance for a broad range of extreme scale applications.

Acknowledgment

The authors thank the engineers in NSCC-TJ for their efforts on this work.

This work is supported in part by the National High Technology Research and Development 863 Program of China under grant 2012AA01A301 and 2012AA01A309, and the National Natural Science Foundation of China under grants 61120106005, 61272140, 61272141 and 61025009.

References

1. Top500, Top500 lists. http://www.top500.org, 2012.

2. F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer and M. Snir. Toward Exascale Resilience: 2014 update. *Supercomputing Frontiers and Innovations*, Vol. 1 no. 1, 2014

3. W. Bland. User Level Failure Mitigation in MPI, *Euro-Par 2012: Parallel Processing Workshops Lecture Notes in Computer Science,* vol.7640, p.499-504, 2013.

4. W. Gropp, MPICH2: A New Start for MPI Implementations, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, p.37–42, 2002.

5. M. Jette and M. Grondona, SLURM: Simple Linux Utility for Resource Management, in *Proceedings of ClusterWorld Conference and Expo*, San Jose, California, 2003.

6. G. Zheng, L. Shi and L.V. Kal, FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI, *Proc. Sixth IEEE Int',l Conf. Cluster Computing* (Cluster ',04), p.93-103, Sept. 2004.

7. D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, and A. Woo, The NAS Parallel Benchmarks 2.0, NASA Ames Research Center, Moffett Field, CA 2002.

8. A. Hoisie, O. Lubeck, and H. Wasserman, Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications, *International Journal of High Performance Computing Applications*, vol.14, p.330-346, 2000.

9. M. Xie, Y. Lu, K. Wang, L. Liu, H. Cao, and X. Yang, Tianhe-1A Interconnect and Message-Passing Services, *IEEE Micro*, vol.32, p.8-20, 2012.

10. X. Yang, X. Liao, K. Lu, Q. Hu, J. Song, and J. Su, The TianHe-1A Supercomputer: Its Hardware and Software, *Journal of Computer Science and Technology*, vol. 26, p.344-351, 2011.

11. G. Suo, Y. Lu, X. Liao, M. Xie, and H. Cao, NR-MPI: a Non-stop and Fault Resilient MPI, In *Proceedings of the 19th IEEE International Conference on Parallel and Distributed Systems(ICPADS 2013)*, Seoul, Korea, p.190-199, 2013

12. R. Koo and S. Toueg, Checkpointing and Rollback-Recovery for Disitributed Systems, *IEEE Transactions on Software Engineering*, vol.13, p.23–31, 1987.

13. A. Bouteiller, P. Lemarinier, G. Krawezik and F. Cappello, Coordinated Checkpoint versus Message Log for Fault Tolerant MPI, in *Proc. Fifth IEEE Int',l Conf. Cluster Computing* (Cluster ',03), p.242, 2003.

14. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Toward a scalable fault tolerant mpi for volatile nodes, In *Proceedings of SC 2002.* IEEE, 2002.

15. K. Ferreira, J. Stearley, I. J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, Evaluating the viability of process replication reliability for exascale systems, in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2011, p.1–12.

16. G. E. Fagg and J. Dongarra, FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World, in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, UK, UK, 2000, p.346–353.

17. A. Bouteiler, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V project: A multiprotocol automatic fault tolerant MPI, *The International Journal of High Performance Computing Applications*, vol.20, p.319-333, 2006.

18. C. Engelmann and S. Bohm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, p.31C38, 2011.

19. D. Fiala, F. Mueller, C. Engelmann, and R. Riesen, Detection and correction of silent data corruption for large-scale high-performance computing. In *Parallel & Distributed Processing Workshops & Phd Forum IEEE International Sympos*, 7196(5), p.2069-2072, 2011.

20. W. Bland, Enabling Application Resilience with and without the MPI Standard, in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (ccgrid 2012), Washington, DC, USA, 2012, p.746–751.

21. J. Hursey and R. Graham, Building a Fault Tolerant MPI Application: A Ring Communication Example, in *16th International Workshop on Dependable Parallel, Distributed and Network-Centric Systems* (DPDNS) held in conjunction with *the 25th IEEE International Parallel and Distributed Processing Symposium* (IPDPS), Anchorage, Alaska, 2011.

22. M. Beck, J. J. Dongarra, G. E. Fagg, G. A. Geist, P. Gray, J.s Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. L. Scott, and V. Sunderam. HARNESS: A Next Generation Distributed Virtual Machine, *Future Generation Computer Systems*, 15(5-6):571-582, 1999.

23. H. Jung, D. Shin, H. Han, J. W. Kim, H. Y. Yeom, and J. Lee, Design and Implementation of *M*ultiple Fault-Tolerant *M*PI over *M*yrinet(M^3), in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005, p.32–46.

24. C. Wang. Transparent Fault Tolerance for Job Healing in HPC Environments, PhD thesis, North Carolina State University, 2009.

25. C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI, In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing* (LCPC 2003), LNCS 2958, p.306-322, College Station, Texas, October 2003.

26. A. Agbaria and R. Friedman, Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations, In *8th IEEE International Symposium on High Performance Distributed Computing,* 1999.

27. S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing, *International Journal of High Performance Computing Applications,* 19(4):479-493, Winter 2005.

28. R. Wang, E. Yao, P. Balaji, D. Buntinas, M. Chen and G. Tan. Building Algorithmically Nonstop Fault Tolerant MPI Programs, In *Proceedings of the 18th IEEE International Conference on High Performance Computing.* (HiPC 2011), December 2011, Bangalore, India.

29. Z. Wu, R. Wang, W. Xu, M. Chen, E. Yao, Supporting User-directed Fault Tolerance over Standard MPI, in *2012 IEEE 18th International Conference on Parallel and Distributed Systems* (ICPADS), p.696-697, 17-19 Dec. 2012.

30. J. S. Plank, K. Li, and M. A. Puening, Diskless Checkpointing, *IEEE Trans. Parallel Distrib. Syst.,* vol. 9, p.972–986, 1998.

31. J. P. Walters and V. Chaudhary, Application-Level checkpointing techniques for parallel programs, in *ICDCIT'06,* Berlin, Heidelberg, 2006, p. 221–234.

32. J. S. Plank, M. Beck, G. Kingsley, and K. Li, Libckpt: transparent checkpointing under Unix, in *TCON'95,* Berkeley, CA, USA, 1995, p.18–18.

33. A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System, in *SC '10*, Washington, DC, USA, 2010, p.1–11.

34. W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, J. Dongarra. A Checkpoint-on-Failure protocol for algorithm-based recovery in standard MPI, In *18th Euro-Par,* LNCS, vol. 7484, p.477-489. 2012.

35. K. Huang and J. A. Abraham, Algorithm-Based Fault Tolerance for Matrix Operations, *IEEE Trans. Comput.,* vol. 33, p.518–528, 1984.

36. D. Fiala, Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing, in *IPDPSW '11*, Washington, DC, USA, 2011, p.2069–2072.

37. Z. Chen, Algorithm-based recovery for iterative methods without checkpointing, in *HPDC '11,* New York, NY, USA, 2011, p.73–84.

38. X. Yang, Y. Du, P. Wang, H. Fu, and J. Jia, FTPA: Supporting Fault-Tolerant Parallel Computing through Parallel Recomputing, *IEEE Trans. Parallel Distrib. Syst.,* vol.20, p.1471–1486, 2009.