IEICE TRANS. INF. & SYST., VOL.E95–D, NO.3 MARCH 2012

PAPER

WBC-ALC: A Weak Blocking Coordinated Application-Level Checkpointing for MPI Programs

Xinhai XU[†a)], ***Student Member***, **Xuejun YANG**[†], ***and*** **Yufei LIN**[†], ***Nonmembers***

SUMMARY As supercomputers increase in size, the mean time between failures (MTBF) of a system becomes shorter, and the reliability problem of supercomputers becomes more and more serious. MPI is currently the de facto standard used to build high-performance applications, and researches on the fault tolerance methods of MPI are always hot topics. However, due to the characteristics of MPI programs, most current checkpointing methods for MPI programs need to modify the MPI library (even operating system), or implement a complicated protocol by logging lots of messages. In this paper, we carry forward the idea of Application-Level Checkpointing (ALC). Based on the general fact that programmers are familiar with the communication characteristics of applications, we have developed BC-ALC, a new portable blocking coordinated ALC for MPI programs. BC-ALC neither modifies the MPI library (even operating system) nor logs any message. It implements coordination only by the *Barrier* operations instead of any complicated protocol. Furthermore, in order to reduce the cost of fault-tolerance, we reduce the synchronization range of the barrier, and design WBC-ALC, a weak blocking coordinated ALC utilizing group synchronization instead of global synchronization based on the communication relationship between processes. We also propose a fault-tolerance framework developed on top of WBC-ALC and discuss an implementation of it. Experimental results on NPB3.3-MPI benchmarks validate BC-ALC and WBC-ALC, and show that compared with BC-ALC, the average coordination time and the average backup time of a single checkpoint in WBC-ALC are reduced by 44.5% and 5.7% respectively.

key words: *Application-Level Checkpointing, weak coordinated, MPI, fault tolerance, consistency*

1. Introduction

In recent years, with the performance improvement of supercomputers, the reliability problem has become more and more serious. On the one hand, with the development of manufacturing technology, the single chip integrates more and more transistors, and it is more vulnerable to transient faults because of high temperature [1]; on the other hand, in order to obtain higher performance, the scale of supercomputers keeps growing. For example, the top ten supercomputers in the top500 list of Jun. 2011 have more than 100 k nodes (cores) on average [2]. However, the reliability of a parallel computing system is inversely proportional to its degree of parallelism [3]. Some researchers have pointed out that the mean time between failures (MTBF) of future Exascale supercomputers will reduce to days or even hours. However, in order to improve the precision of results, some applications need to run several days even on the fastest supercomputer [4], and the execution times of these applications are much longer than MTBF. So fault tolerance methods are necessary for guaranteeing the execution of these applications.

In the distributed and parallel computing area, Checkpoint/Restart (C/R), i.e., checkpointing methods are widely used fault tolerance methods [5], [6]. These methods periodically save the checkpoint data, namely the states of the computation, to stable storage. If some failure happens in system, the program will roll back to an appropriate checkpoint, and the computation is restarted after recovering the state. Generally, checkpointing can be classified along two orthogonal dimensions.

Firstly, according to the level of saving states, checkpointing methods can be classified into two kinds: System-level checkpointing (SLC) and Application-level checkpointing (ALC) [7]. SLC stores all the states of the whole system periodically. It is a transparent method for the programmer, but its implementation relies on the concrete hardware and software environments. ALC reduces the fault-tolerance overheads with the help of programmers, who appoint the location and data of checkpoint. Checkpointing function codes are inserted into the original program, so ALC can satisfy the requirements of different hardware and operation systems.

Secondly, according to the manner of maintaining consistency, checkpointing methods can be classified into coordinated checkpointing and uncoordinated checkpointing [8]. Coordinated checkpointing needs to harmonize all processes to generate a consistent global state. It simplifies the recovery procedure, but its overhead expands as the system scale grows. In uncoordinated checkpointing, each process saves its checkpoint data independently, so each process can save its state at a convenient time. However, uncoordinated checkpointing may lead to the domino effect [9]. Furthermore, according to the manner of coordination, coordinated checkpointing can be classified into blocking coordinated checkpointing and non-blocking coordinated checkpointing. Blocking techniques bring all processes to a stop before taking a global checkpoint. Non-blocking coordinated checkpointing implements coordination by exchanging special markers or control tokens between processes. In order to form a global checkpoint, a global coordination protocol is used to orchestrate the saving of the states of individual processes and the contents of certain messages.

The Message Passing Interface (MPI) [10] is currently the de facto standard used to build high-performance appli-

Manuscript received September 5, 2011.
Manuscript revised November 7, 2011.

[†]The authors are with National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha, China.
a) E-mail: xuxinhai@nudt.edu.cn

DOI: 10.1587/transinf.E95.D.786

cations. How to perform fault-tolerance for MPI programs with C/R is always a hot topic. Many researches [11]–[14] have designed and optimized the checkpointing for MPI programs from different points of view. Nonetheless, in order to get a global consistency checkpoint, the methods above either modify the MPI library (even operating system) or implement a coordination protocol by logging lots of messages. These methods are complicated to implement and will bring heavy fault-tolerance overheads. Based on the ideas of ALC and blocking coordinated checkpointing, this paper develops BC-ALC, a blocking coordinated ALC for MPI programs. This method need not modify the MPI library or operating system, and programmers only need to appoint the positions of checkpoints in an MPI program. A global consistency checkpoint can be obtained by synchronization, implemented by *Barrier* operations, without any complicated coordination protocol.

The classical blocking coordinations are usually implemented by global barriers. However, the synchronization overhead of global barriers is very expensive, and all processes must backup data together after the global barrier, which will exacerbate the pressure on I/O bandwidth. So the blocking coordinated checkpointing based on global barrier is considered as a poor scalable checkpointing method. In order to reduce the overheads of synchronization and backup, based on the analysis of the communication characteristics of MPI programs, this paper develops WBC-ALC, a weak blocking coordinated ALC using group barrier to implement synchronization.

To the best of our knowledge, this is the first work investigating the development of checkpointing methods for MPI programs without modifying MPI library or logging MPI messages. The contributions of this paper are summarized below:

- We introduce BC-ALC, a new portable blocking coordinated ALC without modifying MPI library or logging any message, for MPI programs.
- We design WBC-ALC, a weak blocking coordinated ALC using group barrier instead of global barrier to implement synchronization for MPI programs to reduce fault-tolerance overheads.
- We present a WBC-ALC-based fault-tolerance framework for providing reliability guarantees for MPI programs, and describe an implementation of this framework.
- We demonstrate the validations of BC-ALC and WBC-ALC by making experiments on NPB3.3-MPI benchmarks. Experiment results illustrate that programmers can use BC-ALC or WBC-ALC easily in MPI programs. Compared with BC-ALC, the average coordination time and the average backup time of a single checkpoint in WBC-ALC are reduced by 44.5% and 5.7% respectively.

The rest of the paper is organized as follows. Section 2 introduces the blocking coordinated ALC method for MPI programs to overcome the difficulties in ALC of MPI programs. Section 3 reduces the synchronization range of BC-ALC, and introduces the basic idea and synchronization mechanism of WBC-ALC. Section 4 describes a fault-tolerance framework developed based on WBC-ALC for MPI programs. Section 5 presents an implementation of WBC-ALC. Section 6 describes our evaluation methodology, demonstrates the correctness of BC-ALC and the effectiveness of WBC-ALC. Section 7 reviews the related work. Finally, Sect. 8 concludes the paper.

2. Blocking Coordinated ALC for MPI

In this section, we first introduce the difficulties in ALC of MPI programs, then we present the basic idea of BC-ALC, a high portable blocking coordinated ALC, and describe its mechanism of blocking coordination.

Same as most previous researches on checkpointing, we assume the communications between processes are reliable. Therefore, we will concentrate on the faults of computing nodes, and assume the fail-stop fault model [15].

2.1 Difficulties in ALC of MPI Programs

As shown in Fig. 1, we describe a parallel program with checkpointing as follows: the program is constituted by n processes, denoted by $P_0, P_1, \ldots, P_{n-1}$; during the normal execution of P_i $(0 \le i < n)$, there are m_i checkpoints, denoted by $C_{i,j}$ $(0 \le i < n, 0 < j \le m_i)$; these checkpoints divide the normal execution of P_i into $m_i + 1$ epochs, denoted by $E_{i,j}$ $(0 \le i < n, 0 \le j \le m_i)$. And we call the set of j-th epoch from each process *system epoch* j, denoted by E_j.

Based on the description above, we give the following definitions:

Definition 1: **Global checkpoint** is a checkpoint set $L = \{C_{0,k_0}, C_{1,k_1}, \ldots, C_{n-1,k_{n-1}}\}$, which contains one and only one checkpoint from each process.

Definition 2: Given a global checkpoint $L = \{C_{0,k_0}, C_{1,k_1}, \ldots, C_{n-1,k_{n-1}}\}$ and a message M, which is sent from $E_{a,b}$ to $E_{c,d}$, M is a **late message** for L if $b < k_a$ and $k_c \le d$, and an **early message** for L if $k_a \le b$ and $d < k_c$.

For example, in Fig. 1 M_1 is a late message for L' and M_2 is an early message for L'.

Definition 3: Given a global checkpoint L, L is a **consistent global checkpoint** if there is no early message for L, a **transitless global checkpoint** if there is no late message for L, and a **strong-consistent global checkpoint** if there is no late or early message for L.

Apparently, there is no message crossing any strong-consistent global checkpoint, so a system does not have to log any message at any strong-consistent global checkpoint.

[13] analyzed the difficulties in ALC of MPI programs, and the difficulties are as follows:

- Delayed state-saving: different from SLC, which may

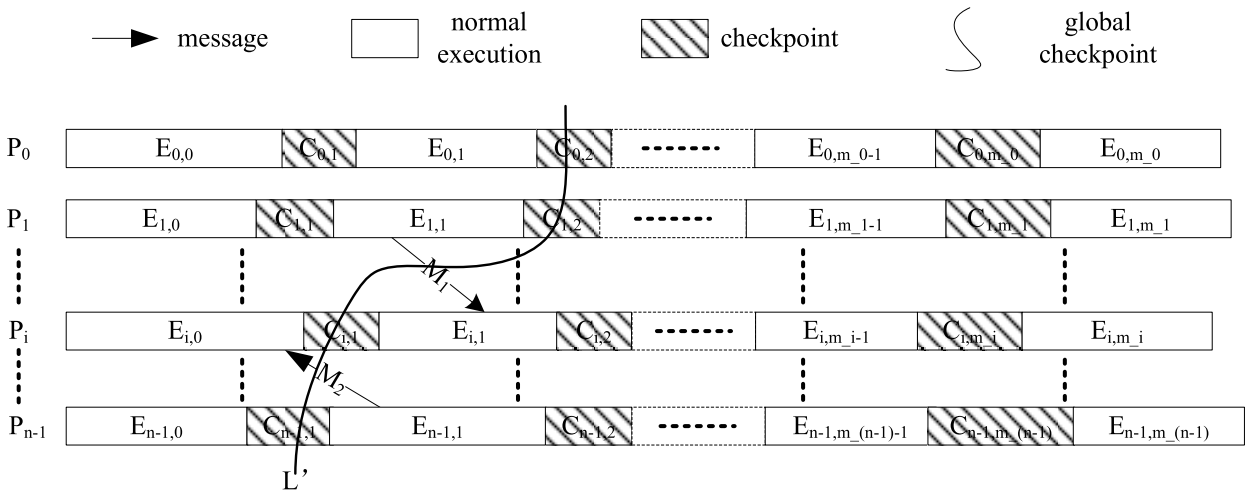

Fig. 1 The basic model of ALC.

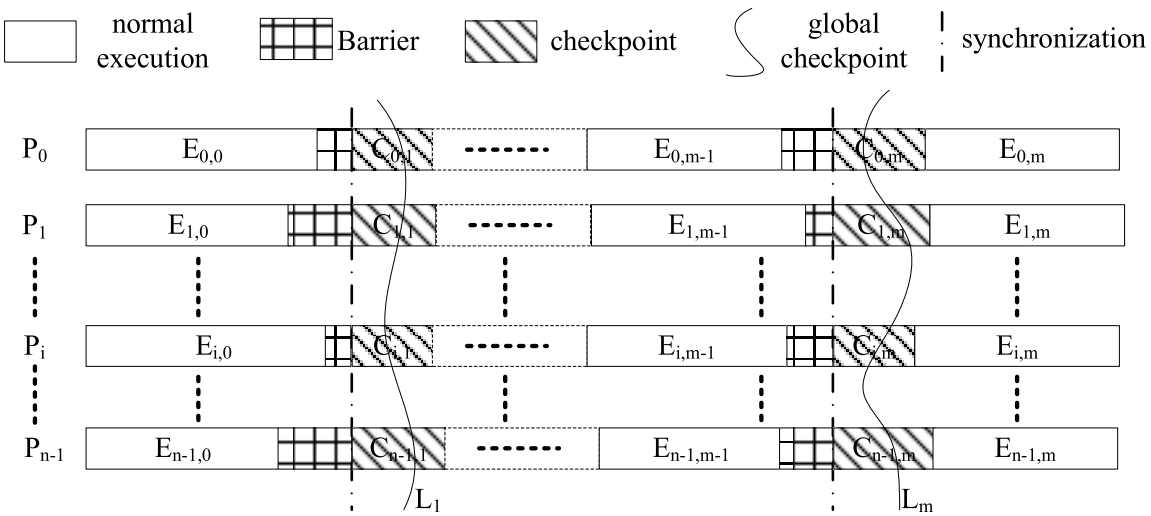

Fig. 2 BC-ALC model.

be taken at any time during a program's execution, ALC can only be taken when a program runs to the positions of checkpoints.

- Handling late and early messages: there may be late and early messages in system when taking checkpoints, so in order to recover the system state correctly, these messages must be carefully dealt with.
- Non-FIFO message delivery at application level: in an MPI application, a process P can use tag matching to receive messages from Q in a different order than as they were sent.
- Collective communication: there are many kinds of collective communications in MPI programs, and these communications involve a group of processes. If some processes make a collective communication call before taking their checkpoints, and others after, the restart from this global checkpoint may lead deadlock.
- Problems Checkpointing MPI Library State: there are two kinds of MPI calls: one is used to manage the MPI environment, such as *MPI_Init*, *MPI_Comm_rank*, *MPI_Comm_split*, etc; the other is used to operate the variables, such as *MPI_Send*, *MPI_Allreduce* and so on. All the states of these MPI calls must be stored when taking checkpoints.

2.2 Blocking Coordination Mechanism

For the difficulties of ALC in MPI programs, we introduce BC-ALC, a new blocking coordinated ALC for MPI programs. As shown in Fig. 2, in BC-ALC, all processes must synchronize together via global barrier before backup, and thus form a global checkpoint. If some failure happens in system, all processes just roll back to their last checkpoint, i.e., the system recovers from the last global checkpoint.

As shown in Fig. 2, we modifies the basic model of ALC into the BC-ALC model. Firstly, each process must take a global barrier before taking checkpoint, so during the

normal execution, all processes take checkpoints the same times, i.e., $m_0 = m_1 = \ldots = m_{n-1}$. We denote the number as m for short. Secondly, system can only recover from the global checkpoint $L_i = \{C_{0,i}, C_{1,i}, \ldots, C_{n-1,i}\}, (0 < i \leq m)$.

ALC does not know the exact send/receive time of a specific message, and message delivery at application level does not follow the FIFO rule, so we analyze the communication relationship among processes based on application level communications, which are implemented by communication primitives. In MPI programs, there are two kinds of communications: point-to-point communication and collective communication. The former involves two processes, which respectively call *Send* primitive and *Receive* primitive to accomplish the data transfer. For this kind of communication, there is an application level communication from the process that calls *Send* primitive to the process that calls *Receive* primitive. A collective communication involves a group of processes, all of which call the same collective communication primitive to exchange data. For the collective communications, we conservatively assume that there is an application level communication between any two processes in the process group.

Apparently, messages are caused by application level communications, so we make the following definition:

Definition 4: Given an application level communication T from process P_i to P_j, the message active area of T is an primitive quadruple $(I_{i,x}, I_{i,y}, I_{j,v}, I_{j,z})$, in which primitive $I_{i,x}$ and $I_{i,y}$ respectively start and complete the *Send* operation of T in P_i, and primitive $I_{j,v}$ and $I_{j,z}$ respectively start and complete the *Receive* operation of T in P_j.

In non-blocking point-to-point communications, $I_{i,x}$ ($I_{j,v}$) and $I_{i,y}$ ($I_{j,z}$) are implemented by two different primitives, which usually appear in pairs, such as *MPI_Isend* (*MPI_Irecv*) and *MPI_Wait*. In blocking point-to-point communications, $I_{i,x}$ ($I_{j,v}$) and $I_{i,y}$ ($I_{j,z}$) are implemented by the same primitive, which takes charge of starting and completing the operation together, such as *MPI_Send* (*MPI_Recv*). In collective communications, all of the four primitives in the message active area are the same primitive, which is called twice by P_i and P_j.

As shown in Fig. 3, for a message active area $(I_{i,x}, I_{i,y}, I_{j,v}, I_{j,z})$, from the application level, process P_i can send messages at any time between t_x and t_y, and process P_j can receive messages at any time between t_v and t_z. From the system level, between $\min\{t_x, t_v\}$ and $\max\{t_y, t_z\}$, there

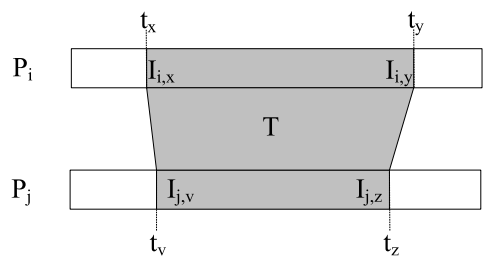

Fig. 3 Message active area.

are lots of information about the messages in the system, including message queues, message buffers and the network. Consequently, in order to avoid storing the information of messages at checkpoints, for any application level communication T with message active area $(I_{i,x}, I_{i,y}, I_{j,v}, I_{j,z})$, all the primitives $I_{i,x}$, $I_{i,y}$, $I_{j,v}$ and $I_{j,z}$ must be in the same system epoch.

Based on the general fact that programmers are familiar with the communication characteristics of applications, programmers can guarantee that no message active area will cross any global checkpoint in BC-ALC by inserting application level checkpoints at appropriate positions. This is not a tough work for programmers as we know from the experiment results shown in Sect. 6. Apparently, with the programmers' effort, no message will cross the global checkpoints after global barriers, so all the global checkpoints L_i $(0 < i \leq m)$ are strong-consistent ones. Consequently, BC-ALC overcomes all the difficulties except "Problems Checkpointing MPI Library State". To tackle this difficulty, we only need to deal with the MPI calls which manage the MPI environment, and it can be done at implementation level, as described in Sect. 5.

3. Weak Blocking Coordinated ALC

Although all the global checkpoints L_i $(0 < i \leq m)$ are strong-consistent ones in BC-ALC, the overheads of global barriers before taking checkpoints are very high. In order to reduce these overheads, this section introduces the basic idea behind WBC-ALC, and describes the optimized coordination mechanism of WBC-ALC.

3.1 Basic Idea behind WBC-ALC

From Fig. 4 (b), it is not difficult to find out that there is only one communication between P_0 and P_1 in system epoch E_0, so once P_0 and P_1 have completed the computation in E_0, the two processes need not rollback if P_2 occurs a failure in E_0. So P_0 and P_1 can take their checkpoints after synchronizing with each other at the end of $E_{0,0}$ and $E_{1,0}$ respectively. Similarly, P_2 can take checkpoint at the end of $E_{2,0}$ without synchronizing with any other process.

Based on the analysis above, we reduce the synchronization range of the barrier in BC-ALC: as shown in Fig. 4 (c), at the end of a epoch, a process can take checkpoint after synchronizing with the processes that have communicated during this epoch. If some failure happens in system, all the processes rollback to their last checkpoints, and recompute from the global checkpoint. We call this blocking coordinated ALC with group synchronization instead of global synchronization *weak blocking coordinated ALC (WBC-ALC)*.

Different from BC-ALC, processes in WBC-ALC only synchronize with a part of processes. So processes are not necessarily running in the same system epoch, neither in the normal execution nor after the recovery. Compared with BC-ALC, there are three advantages in WBC-ALC:

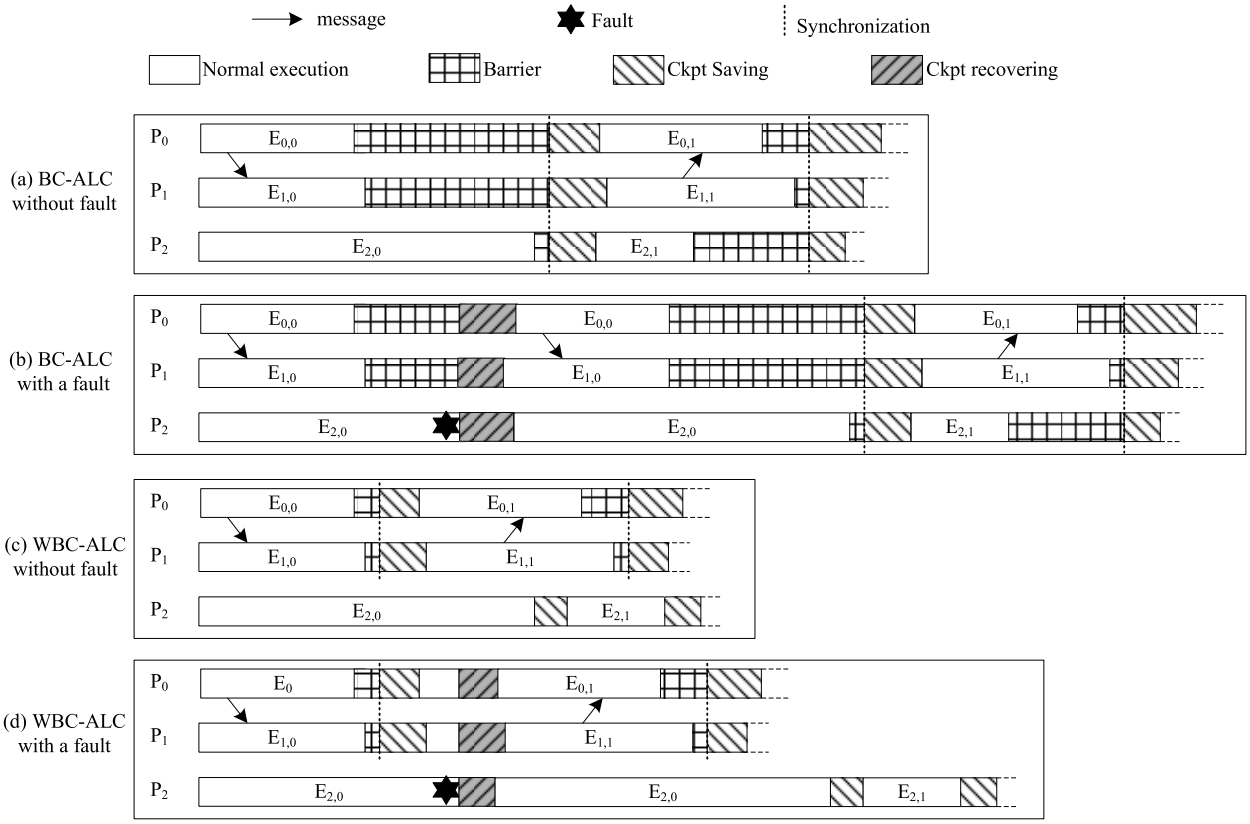

Fig. 4 BC-ALC v.s. WBC-ALC.

- The overhead of group synchronization is lower than that of global synchronization.
- Group synchronization provides the processes the chance to take checkpoints at different time, which eases the pressure on I/O bandwidth, and thus can reduce the overhead of backup.
- With group synchronization, processes execute asynchronously: on the one hand, different processes can compute the epochs belong to different system epoches; on the other hand, a process's checkpoint can be taken while another process is performing the normal computation. So the asynchronism can reduce the total overhead of fault-tolerance.

3.2 Coordination Mechanism of WBC-ALC

The key difference between BC-ALC and WBC-ALC is changing global synchronization into group synchronization. This section describes the coordination mechanism of WBC-ALC. Since BC-ALC ensures that all primitives of a message active areas are in the same system epoch, we make the definitions as follows:

Definition 5: In a system with a process set $Proc = \{P_0, P_1, \ldots, P_{n-1}\}$, if P_i and P_j satisfy one of the following conditions, P_i is **communication-related** with P_j in system epoch E_k. We denote this relation as $< P_i, P_j >_k$.

1. $i = j$.
2. There is an application level communication between P_i and P_j in E_k.
3. There is a process P_x that satisfies the two relations $< P_i, P_x >_k$ and $< P_x, P_j >_k$.

It is not difficult to prove that the communication relation $<,>_k$ is reflexive, symmetric and transitive, so $<,>_k$ is an equivalence relation.

Definition 6: In a system with a process set $Proc = \{P_0, P_1, \ldots, P_{n-1}\}$, given a communication relation $<,>_k$, we denote the equivalence class of P_i under $<,>_k$ as $[P_i]_k = \{P_j | < P_i, P_j >_k\}$, and call $[P_i]_k$ the **communication related process group** of P_i in E_k.

Definition 7: In a system with a process set $Proc = \{P_0, P_1, \ldots, P_{n-1}\}$, given a communication relation $<,>_k$ and a partition of $Proc \prod = \{S_0, S_1, \ldots, S_{v-1}\}$, $\prod$ is a **consistent partition** of system epoch E_k if for any communication related process group $[P_i]_k$ $(0 \le i < n)$, there is a S_j $(0 \le j < v)$ so that $[P_i]_k \subseteq S_j$.

Based on the definitions above, the coordination mechanism of WBC-ALC is: in a system with a process set $Proc = \{P_0, P_1, \ldots, P_{n-1}\}$, given a consistent partition of system epoch $E_k \prod = \{S_0, S_1, \ldots, S_{v-1}\}$, all the processes belong to the same S_j $(0 \le j < v)$ synchronize with each other before taking checkpoints. We implement this mechanism by splitting the processes of the system into v groups

corresponding to $\prod$, and synchronizing the processes in the same group by group barriers.

Theorem 1: In WBC-ALC, if some failure happens in the system, the global checkpoint, constituted by the last checkpoints of all processes, is a strong-consistent global checkpoint.

Proof: Let the last checkpoint of P_i $(0 \leq i < n)$ be $C_{i,last_i}$, and denote $L_{last} = \{C_{0,last_0}, C_{1,last_1}, \ldots, C_{n-1,last_{n-1}}\}$. If L_{last} is not a strong-consistent global checkpoint, there must be a message M that crosses L_{last}. Assume M is caused by the application level communication T, whose message active area is $(I_{a,x}, I_{a,y}, I_{b,v}, I_{b,z})$. On the one hand, in BC-ALC/WBC-ALC, programmers must ensure that all the four primitives belong to the same system epoch, denoted by E_k, so $I_{a,x}$ and $I_{a,y}$ are on the same side of L_{last} and so do $I_{b,v}$ and $I_{b,z}$; On the other hand, since M crosses L_{last}, there must be two primitives among $\{I_{a,x}, I_{a,y}, I_{b,v}, I_{b,z}\}$ on different sides of L_{last}. Consequently, $I_{a,x}$ and $I_{b,z}$ must reside on the different sides of L_{last}, i.e., $last_b < k \leq last_a$ or $last_a < k \leq last_b$. However, based on Definition 5, P_a is communication-related with P_b in system epoch E_k, i.e., $< P_a, P_b >_k$, so P_a and P_b belong to the same communication related process group $[P_a]$ ($[P_b]$). According to the coordination mechanism of WBC-ALC, P_a and P_b must synchronize with each other before taking checkpoints at the end of E_k, i.e., $last_a < k, last_b < k$ or $k \leq last_a, k \leq last_b$. Apparently, $last_b < k \leq last_a$ or $last_a < k \leq last_b$ is opposite to $last_a < k, last_b < k$ or $k \leq last_a, k \leq last_b$, so L_{last} must be a strong-consistent global checkpoint.

Based on Theorem 1, in WBC-ALC, each process only need to maintain one checkpoint without saving any information of messages.

In sum, given consistent partitions for all system epochs, BC-ALC can be improved into WBC-ALC by substituting global barriers with group ones. The method of partitioning is described in Sect. 5.1.

4. Framework of WBC-ALC

Carrying forward the idea of application level ALC, we make use of the programmer's understanding of the applications to help accomplish the fault-tolerance function of WBC-ALC. This section presents the compiler-directed fault tolerance framework developed on top of WBC-ALC. In Sect. 4.1, we describe our programming method used to facilitate fault tolerance of WBC-ALC. In Sect. 4.2, we briefly introduce the functionalities of all key components in our framework.

4.1 Programming Method

As shown in Fig. 5, in order to facilitate fault tolerance of WBC-ALC, programmers need to modify the original MPI programs as follows:

Firstly, programmers should define *nCkpt*, the number of application level checkpoints in the source code, by the compiler directive "#define nCkpt N". Note that *nCkpt* represents statically the number of checkpoints inserted in each process. During program execution, a particular application level checkpoint may be executed several times.

```
#define nCkpt  N
main(){
   ……
   #CKPT_x
   ……
   #CKPT_y
   ……
}

#ColorInit(int* ColorArray, int myrank){
   // code written by programmer
   ……
}
```

Fig. 5 Programming method of WBC-ALC.

Secondly, programmers should appoint the positions of the *i*-th application level checkpoints by the compiler directive "#CKPT_i". Notice that the positions of application level checkpoints must satisfy the blocking coordination mechanism in Sect. 2.2. It is not a hard work for programmers to find the positions, and Sect. 6 will show that we can insert application level checkpoints in all the MPI-version programs of NPB benchmark easily. For some complicated MPI programs, we have implemented a source-to-source pre-compiler ALEC to identify the safe checkpointing regions [16]. ALEC firstly constructs the CFG of MPI programs, and then conservatively matches the MPI communication primitives and finds all potential message active areas over the CFG. A global checkpoint is safe, i.e., strong-consistent if the virtual line that connects each checkpoint location of this global checkpoint in the CFG does not cross any potential message active area. Based on a live-variable analysis for MPI programs, ALEC can also find out the data which really need to be saved at checkpoints [17].

Thirdly, in order to implement the coordination mechanism of WBC-ALC, for each system epoch, we need the communication relations among all processes to split these processes into corresponding process groups. So the programmers have to implement a function named "*ColorInit*" to afford the information. There are two parameters in *ColorInit*: one is *myrank*, an integer stands for the ID of the calling process; the other is *ColorArray*, an array of *N* integers, in which *ColorArray*[*i*] is the *color* value of the process P_{myrank} at the *i*-th application level checkpoint. Each process calls *ColorInit* to initialize its *ColorArray*, and all the processes with same *color* value of *ColorArray*[*i*] are in the same group at the *i*-th application level checkpoint.

4.2 The Framework

Figure 6 depicts our checkpoint-based fault-tolerance

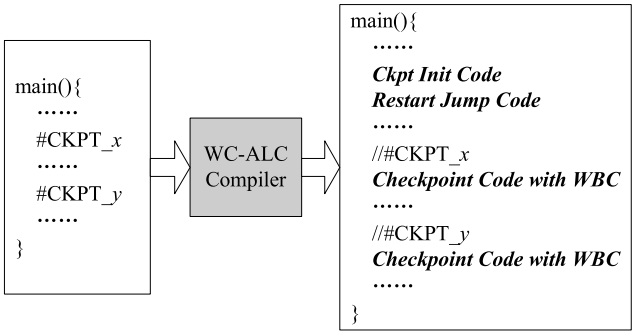

Fig. 6 Framework of WBC-ALC.

framework. An MPI program with the compiler directives shown in Fig. 5 is compiled by a source-to-source WBC-ALC compiler into another program with all required fault-tolerance code, which can be categorized into three types:

- **Ckpt Init Code**. After initializing the MPI environment, we generate code to initialize the variables for checkpointing.
- **Restart Jump Code**. After *Ckpt Init Code*, we generate code to switch the program's context to the last checkpoint saved during last execution if the program is restarted after some failure.
- **Checkpoint Code with Weak Blocking Coordination**. For each compiler directive "#CKPT_x", we generate code to take checkpoint after group synchronization.

5. Implementation of WBC-ALC

5.1 ColorInit

Function *ColorInit* computes the color values at every checkpoint for each process, and it is the key of implementing weak blocking coordination. We inherit the idea of ALC, and leave the work to the programmers. Programmers can implement the function directly based on the well understanding of the application, or analyze the colors at different checkpoints one by one in the following steps:

- In Step 1, for an application level checkpoint C, programmers find out the statement set *S(C)*, constituted by all the statements between C and C's predecessor checkpoints. Notice that due to the branch and jump statements, there may be several predecessor checkpoints of C.
- In Step 2, programmers have to find out $Pair(C)$, the communication process pair set of checkpoint C. If there exists a communication between process P_x and P_y in *S(C)*, the process pair (P_x, P_y) belongs to $Pair(C)$. There are three methods to obtain $Pair(C)$:
  1. Profile method: programmers can pre-run the program and log the communication object processes of the communication statements in *S(C)* for each process. In order to implement this function, programmers can insert code at each communication primitive to log communication objects. Also, the tools TAU and Vampir can be used to implement this function.
  2. Static analysis method: based on an understanding of the code or with the help of the compiler, programmers can obtain $Pair(C)$ by static analysis. Firstly, based on an analysis of control dependence of communication statements [17], [18] in *S(C)*, programmers can obtain each process's communication statements, which will be executed by the process in *S(C)*. Secondly, by traversing each process's communication statements, programmers can obtain each process's expressions of communication objects. Thirdly, by computing each process's expressions of communication objects, programmers can obtain the ID set of all communication object processes for each process. Finally, for each process P_x and its communication object process P_y, the process pair (P_x, P_y) belongs to $Pair(C)$.
  3. Default method: for the programs which can not be analyzed by the Profile or Static analysis method, programmers can assume that each process communicates with all others. Apparently, this is a conservative method, and the weak blocking coordination reverses to the basic blocking coordination.
- Step 3 is to form an undirected graph $G(C)$ based on $Pair(C)$. The nodes of $G(C)$ represent the processes of the system, and the edge between two nodes represents the communication process pair of corresponding processes.
- The final step is to assign the color values of processes based on $G(C)$. All the color values of the processes in a connected subgraph of $G(C)$ must be the same.

In sum, based on the understanding of communication characteristics of MPI programs, programmers can implement the function *ColorInit* with the help of profiling or compiler static analysis. In most practical MPI programs, processes communicate regularly, so it is not difficult for programmers to define the color values of processes in *ColorInit*, as shown in Sect. 6.

5.2 Ckpt Init

As shown in Fig. 7, in order to record the color values at all application level checkpoints, we declare *ColorArray*, an array of *nCkpt* integers, and initialize it by calling function *ColorInit*. Meanwhile, in order to perform weak blocking coordination by group barriers, we declare *CommTeam*, an array of *nCkpt* integers, to store the handles of communicators of group barriers, and initialize the array by calling *MPI_Comm_split*.

```
// Ckpt Init Code
   int* ColorArray = malloc(nCkpt*sizeof(int));
   ColorInit(ColorArray, myrank);

   MPI_Comm* CommTeam = malloc(nCkpt*sizeof(MPI_Comm));
   for(i=0;i<nCkpt;i++){
      MPI_Comm_split(MPI_COMM_WORLD,
            ColorArray[i], myrank, CommTeam[i], ierr);
   }
```

Fig. 7 Codes of Ckpt Init.

```
// Restart Jump Code
if(State == Restart){
   Read LastCkptNum from checkpointdata;
   Switch( LastCkptNum ){
      case 1: goto CKPT_1;
      case 2: goto CKPT_2;
      ……
   }
}
```

Fig. 8 Codes of Restart Jump.

```
// Checkpoint Code for "#CKPT_x"
MPI_Barrier(CommTeam[x], ierr);
Backup_Checkpoint_Data();

CKPT_x:
   if(State == Restart){
      Recover_Checkpoint_Data();
      State = Normal;
   }
```

Fig. 9 Codes of Checkpoint.

5.3 Restart Jump

Similar as most previous checkpointing methods, in WBC-ALC, if some failure happens in system, user or system restarts the program with "*Restart*" state, which is recorded in variable *State*. As shown in Fig. 8, after initializing variables and the MPI environment, processes check the state of this execution. If *State* is *Restart*, each process reads the checkpoint ID from its last checkpoint data, and jumps to the code of corresponding checkpoint. Notice that because processes do not backup the MPI environment at checkpoints, the *Restart Jump Code* must be inserted after the initialization code of MPI environment.

5.4 Checkpoint with Weak Blocking Coordination

As shown in Fig. 9, in order to take checkpoints with weak blocking coordination, for a compiler directive "#define nCkpt N", beside transforming it into "*Backup_Checkpiont_Data*", the function of saving data, we also need to perform the group synchronization by *MPI_Barrier*. Of course, we also add recovery code at the end of the checkpoint code.

6. Experiments

We demonstrate the validation of WBC-ALC (BC-ALC) and the superiority of WBC-ALC over BC-ALC using the MPI-version NPB benchmarks. Section 6.1 introduces the benchmarks and experiment platform used. Section 6.2 describes the methodology used for evaluating this work. Section 6.3 presents and analyzes our results.

6.1 Benchmarks and Platform

We select the NPB3.3-MPI benchmarks to evaluate WBC-ALC (BC-ALC). These benchmarks consist of five parallel kernels EP, MG, CG, FT and IS, and three simulated applications LU, SP and BT. In our experiments, the problem sizes of all these benchmarks are Class *C*, and each benchmark is computed by 64 processes.

Our platform is a cluster with eight nodes, and each node is equipped with two 2.93 G Intel Xeon X5670 CPUs and 24 GB RAM. The interconnection is the same as described in [19], and the simplex point-to-point bandwidth is 80 Gb/s. Notice that although there are 6 cores in Xeon X5670 CPU, we only allocate 4 processes on it. All these benchmarks are executed in Redhat5.5.

6.2 Evaluation Methodology

For each benchmark, we analyze its communication characteristic and insert application level checkpoints into the program at appropriate positions according to the coordination mechanism of BC-ALC. In order to demonstrate the validation of BC-ALC, for each benchmark, during the execution of its MPI program with application level checkpoints, we interrupt it randomly and restart it with "*State=Restart*". If the results of this program after restarting can pass the results test, BC-ALC is validate for the benchmark. Besides, we also evaluate the overheads (coordination time and backup time per checkpoint) of BC-ALC.

In order to implement WBC-ALC, based on the analysis of the communication characteristics of benchmarks, we give the group information of each application checkpoint by implementing function "ColorInit". In the similar way with BC-ALC, we demonstrate the validation of WBC-ALC and examine the effect of the weak blocking coordination optimization.

Since the optimization of checkpoint data is not the focus of this paper, we directly use the method in [17] to obtain the checkpoint data at each checkpoint. And at a given checkpoint, the checkpoint data in BC-ACL are the same as those in WBC-ALC.

6.3 Results and Analysis

6.3.1 Overheads of BC-ALC

In order to implement BC-ALC in MPI programs, programmers need to follow the programming method, described in Sect. 4.1, to insert application level checkpoints into the original programs with compiler directive "#CKPT_i". The positions of the checkpoints must satisfy the requirement of the coordination mechanism of BC-ALC. Based on the analysis of the eight benchmarks, we find a common characteristic: for each benchmark, the major body of the corresponding MPI program is a major loop, and an application level communication that starts in an iteration of this loop must completes in the same iteration. So for each program, we can insert an application level checkpoint at the beginning or end of its major loop's body.

Concretely, due to the communication characteristics of each benchmark, we insert checkpoints at different positions in different programs. For EP, CG, MG and LU, we only insert checkpoints at the end of their major loops' bodies. For BT and SP, the bodies of their major loops can both be split into four steps: three steps for computations and communications in X/Y/Z dimension respectively and one step for global computations and communications. So we insert four checkpoints at the end of the four step respectively. For FT, the computations and communications of the major loop is organized based on the allocation of processes. In implementation of FT, we map the processes onto a two-dimensional *Mesh* topology, so we insert two checkpoints into the program. The major loop's body of IS is separated by several global collective communications, so we insert two checkpoints: one before these communications and the other at the end of the loop's body.

Based on the method in [17], programmers give the checkpoint data at each application level checkpoint directly. Experiment results show that BC-ALC is validate for all the benchmarks. Table 1 shows the average coordination time and average backup time of a single checkpoint for these benchmarks. The average backup time is directly proportional to the size of the checkpoint data.

Table 1 Overheads per checkpoint in BC-ALC.

Prog.	Cktp Size (B)	Coordination Time (S)	Backup Time (S)
EP	144	2.297E-4	5.258E-5
MG	56M	3.942E-4	5.561E-1
CG	12M	1.163E-4	1.742E-1
FT	81M	2.060E-1	1.605
IS	18M	2.701E-1	2.115E-1
LU	7.8M	5.746E-3	2.240E-2
SP	25M	5.055E-2	2.274E-1
BT	8.1M	6.422E-3	2.462E-2

6.3.2 Optimization of WBC-ALC

To implement fault-tolerance by WBC-ALC, we must define the color values of application level checkpoints by function "ColorInit". There is no communication in the major loop of EP, so for the unique checkpoint in EP, the color value of a process is equal to its process ID *myrank*. For BT and SP, the first three checkpoints are at the ends of X/Y/Z dimension respectively, so the color values of these checkpoints are the process's coordinate in X/Y/Z dimension correspondingly; the last checkpoint is after the global computations and communications, so its color value is a constant 0, which stands for that all processes are in the same communication group. In the major loop of FT, there are two communication groups split from global communicators, and these two groups are just those needed for the two checkpoints, so we use them for group barriers directly. For IS, there is no communication before the first checkpoint, so the color value of the first checkpoint is equal to the process's *myrank*; there are several global collective communications before the second checkpoint, so the color value of the second checkpoint is a constant 0. For MG, CG, and LU, the communications in their major loops are complicated, so we conservatively assume that all processes communicate with all others. The color value of each process is then constantly 0, and the weak blocking coordination reverses to the basic blocking coordination.

Experimental results show that WBC-ALC is validate for all the benchmarks based on splitting groups above. As shown in Fig. 10, compared to BC-ALC, the coordination time of a single checkpoint in WBC-ALC is reduced by 44.5% on average and 99.1% on maximum. There are two reasons for this improvement: on the one hand, group barrier reduces the synchronization range of the barrier; on the other hand, group barrier makes processes in different groups execute asynchronously. Beside the coordination time, WBC-ALC also reduces the backup time of a single checkpoint by 5.7% on average and 31.2% on maximum, and the percentage is larger when there are more checkpoint data.

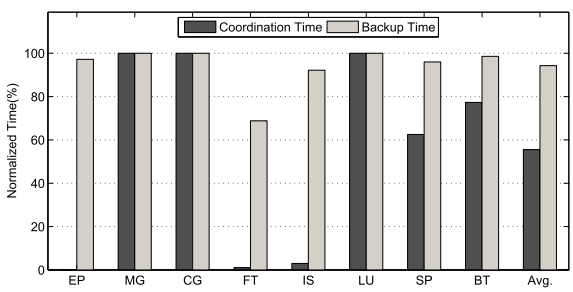

Fig. 10 Normalized overheads per checkpoint by WBC-ALC over BC-ALC.

7. Related work

In order to implement blocking coordinated checkpointing, hardware blocking was used on the IBM SP-2 to take system-level checkpoints; Software blocking techniques exploit barriers - when processes reach a global barrier, each one saves its own state on stable storage [13]. Blocking coordinated checkpointing with global barrier has been well used in OpenMP programs [20]. However, this checkpointing method has not been used in MPI programs.

In coordinated checkpointing, the procedure of coordination is costly. In order to reduce this cost, minimal checkpoint coordination implements a protocol to reduce the range of coordination by only harmonizing the processes which it has communicated with since the last checkpoint [21]. Xavier Besseron developed an optimized coordinated protocol using a dataflow graph model for KAAPI applications [22]. In this optimized coordinated protocol, a process only synchronizes with its communication objects based on the dataflow graph. However, this protocol is only used for KAAPI applications.

For send-deterministic MPI applications, [11] developed an uncoordinated checkpointing to avoid domino effect by implementing a complicated protocol based on the characteristic of send-deterministic. However, most of the researches of checkpointing for MPI programs are focusing on coordinated checkpointing. [12] compared blocking with non-blocking coordinated checkpointing for large-scale fault tolerant MPI programs. The authors found out that for high-speed networks, the blocking implementation gives the best performance for sensible checkpoint frequency.

The above researches of checkpointing for MPI are all SLC, which must modify the MPI library or operating system. Bronevetsky developed a automated ALC for MPI programs [13]. A coordination layer is added between the application and the MPI library, and a coordination protocol is implemented. Bronevetsky also improved the fault-tolerance method for collective operations in MPI programs [14]. However, Bronevetsky's methods must intercept all calls to the MPI library and save lots of information of MPI library.

8. Conclusion

For the difficulties in ALC of MPI programs, we introduce BC-ALC, a new portable blocking coordinated ALC for MPI programs, and with the reduction of synchronization range in blocking coordination, we develop WBC-ALC. Based on this new method, we have developed a compiler-directed fault tolerance framework for MPI programs and an implementation for it. Our experimental results obtained on NPB3.3-MPI benchmarks demonstrate that programmers can use BC-ALC and WBC-ALC easily, and these two methods are valid. Compared to BC-ALC, WBC-ALC can reduce the fault-tolerance overhead effectively.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (NSFC) No.60921062, 61120106005.

References

[1] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," Proc. International Conference on Dependable Systems and Networks, pp.389–398, June 2002.
[2] Top500. http://www.top500.org/lists/2011/06
[3] C.-D. LU, Scalable Diskless Checkpointing for Large Parallel Systems. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 2005.
[4] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs," Proc. 2004 ACM/IEEE Conference on Supercomputing, 2004.
[5] E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," ACM Comput. Surv., vol.34, no.3, pp.375–408, 2002.
[6] J.S. Plank, "An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance," Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.
[7] E. Roman, "A survey of checkpoint/restart implementations," Berkeley Lab Technical Report (publication LBNL-54942), July 2002.
[8] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," ACM Trans. Comput. Syst., vol.3, no.1, pp.63–75, 1985.
[9] G.-M. Chiu and C.-R. Young, "Efficient rollback-recovery technique in distributed computing systems," IEEE Trans. Parallel Distrib. Syst., vol.7, no.6, pp.565–577, June 1996.
[10] The Message Passing Inteface (MPI) Standard. http://www-unix.mcs.anl.gov/mpi
[11] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic message passing applications," 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS2011), Anchorage, USA, 2011.
[12] 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," Proc. 2006 ACM/IEEE Conference on Supercomputing, Tampa, Florida, Nov. 2006.
[13] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," Proc. Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, June 2003.
[14] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in application-level fault-tolerant MPI," Proc. 17th Annual International Conference on Supercomputing, San Francisco, CA, USA, June 2003.
[15] R.D. Schlichting and F.B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," ACM Trans. Comput. Syst., vol.1, no.3, pp.222–238, 1983.
[16] P. Wang, Y. Du, H. Fu, X. Yang, and H. Zhou, "Static analysis for application-level checkpointing of MPI programs," 10th IEEE International Conference on High Performance Computing and Communications (HPCC2008), Dalian, China, 2008.
[17] X. Yang, P. Wang, H. Fu, Y. Du, Z. Wang, and J. Jia, "Compiler-assisted application-level checkpointing for MPI programs," 28th International Conference on Distributed Computing Systems, Beijing,

China, June 2008.
[18] M.M. Strout, B. Kreaseck, and P.D. Hovland, "Data-flow analysis for MPI programs," Proc. 2006 International Conference on Parallel Processing (ICPP06), pp.175–184, Washington, DC, USA, 2006.
[19] M. Xie, Y. Lu, L. Liu, H. Cao, and X. Yang, "Implementation and evaluation of network interface and message passing services for TianHe-1A supercomputer," Proc. 19th Annual Symposium on High-Performance Interconnects, Santa Clara, CA, USA, Aug. 2011.
[20] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, USA, Oct. 2004.
[21] N.S. Bowen and D.K. Pradhan, "Processor- and memory-based checkpoint and rollback recovery," Computer, vol.26, no.2, pp.22–32, Feb. 1993.
[22] X. Besseron and T. Gautier, "Optimized coordinated checkpoint/rollback protocol using a dataflow graph model," Workshop APRETAF: Algorithmes Parallèles, Répartis Et Tolérance Aux Fautes, Grenoble, France, Jan. 2009.

Xinhai Xu Born in 1984. Received his B.S. and M.S. degree in computer science from the National University of Defense Technology (NUDT) in 2006 and 2008. He is now a Ph.D. candidate in Computer Science from School of Computer Science at NUDT. His research interest lies in high-performance computing and fault tolerance. His email address is xuxinhai@nudt.edu.cn

Xuejun Yang Born in 1963. Received his M.S. and Ph.D. degree in computer science from the National University of Defense Technology (NUDT) in 1986 and 1991. He is now a professor in the National Laboratory for Parallel and Distributed Processing, School of Computer, NUDT. His research interests include supercomputer architecture, parallel and distributed operating systems, parallel language, and compilers. His email address is xjyang@nudt.edu.cn

Yufei Lin Born in 1985. Received her B.S. degree in automation from the University of Science and Technology of China (USTC) in 2006 and M.S. degree in computer science from the National University of Defense Technology (NUDT) in 2008. She is now a Ph.D. candidate in Computer Science from School of Computer Science at NUDT. Her research interest lies in high-performance computing and evaluation. Her email address is linyufei@nudt.edu.cn