

# Checkpointing and Migration of parallel processes based on Message Passing Interface

Zhang Youhui, Wang Dongsheng, Zheng Weimin  
*Department of Computer Science,  
Tsinghua University, China.*

## Abstract

This paper presents a Checkpoint-based Rollback Recovery and Migration System for Message Passing Interface, ChaRM4MPI, for Linux Clusters. Some important fault tolerant mechanisms are designed and implemented in this system, which include coordinated checkpointing protocol, synchronized rollback recovery, process migration, and so on. Owing to ChaRM4MPI, the node transient faults can be recovered automatically, and the permanent fault can also be recovered through checkpoint mirroring and process migration techniques. Moreover, users can migrate MPI processes from one node to another manually for load balance or system maintenance. ChaRM4MPI is a user-transparent implementation and introduces a little running time overhead.

## 1. Introduction

Linux Clusters offer a cost-effective platform for high-performance, long-running parallel computations and have been used widely in recent years. However the main problem with programming on clusters is the fact that it is prone to change. Idle computers may be available for computation at one

moment, and gone the next due to load, failure or ownership [1]. The probability that clusters will fail increases with the number of nodes. During normal computing, one abnormal event is likely to cause the entire application to fail. To avoid this kind of time waste, it is necessary to achieve high availability for clusters. Checkpointing & Rollback Recovery (CRR) and Process Migration offer a low overhead and full solution to this problem [1][2].

CRR, which records the process state to stable storage at regular intervals and restarts the process from the last recorded checkpoint upon system failure, is a method that avoids the waste of computations accomplished prior to the occurrence of the failure. Checkpointing also facilitates process migration, which suspends the execution of a process on one node and subsequently resumes its execution on another. However, Checkpointing a parallel application is more complex than just having each processor take checkpoints independently. To avoid domino effect and other problems upon recovery, the state of inter-process communication must be recorded, and the global checkpoint must be consistent. And there are some related CRR software, including MPVM [3], Condor [4], Hector [5].

This paper presents a Checkpoint-based Rollback Recovery and Migration System for Message Passing Interface, ChaRM4MPI, for Linux Clusters. In this system some important fault tolerant mechanisms are designed and implemented, which include coordinated checkpointing protocol, synchronized rollback recovery, process migration, and so on.

Using ChaRM4MPI, the node transient faults can be recovered automatically, and the permanent fault can also be recovered through checkpoint mirroring and process migration techniques. If any node that is running the computation drops out of the clusters, the computation will not be interrupted. Moreover, users can migrate MPI processes from one node to another manually for load balance or system maintenance. ChaRM4MPI is a user-transparent implementation, which means it is needless for users to modify their source codes.

Some MPI parallel applications are tested on ChaRM4MPI, which include the sorting programs and LU decomposition developed by NASA, an overfall simulation and so on. We study the extra time overhead introduced by CRR mechanism, and the result is fairly satisfied. The average extra overhead per checkpointing is less than 2% of the original running time.

In the full text, the coordinated CRR and migration protocol is described first, and then we introduce the work involved in modifying MPICH4 in detail. At last, testing results will be presented and analyzed.

## **2. Protocols**

One of the major components of ChaRM4MPI is its facility for transparent checkpointing and restarting a process, possibly on a different node. In order to achieve this goal, we use some techniques employed by *libckpt* [6] and *CoCheck* [7], and then implement a portable, low-overhead, user-transparent process CRR library *libcsm* [8].

### **2.1 CRR protocol**

Based on *libcsm*, ChaRM4MPI implements a parallel applications CRR mechanism. To avoid domino effect, orphan messages, lost messages and other problems upon recovery, the state of inter-process communication must be recorded; also the global checkpoint must be consistent. We employ a coordinated CRR protocol in ChaRM4MPI, which is relatively easy to implement and has low expenditure in modern parallel systems, like COCs, because of the high rate of network communication. Readers who are interested in the protocol can study [9].

### **2.2 Migration protocol**

CRR mechanism enables process migration. Checkpointing generally implies that the process state is saved to local stable storage. Note that by transporting a process state to and recovering the process on another node, we accomplish process migration. Like process checkpointing, process migration has to avoid lost messages. Therefore we design and implement a quasi-asynchronous migration protocol [10], which has lower overhead than other migration protocols. In our protocol, although migrated processes (MPs) and non-migrated processes (NMPs) are interrupted at the beginning of migration, NMPs are allowed to execute during most time of the migration. NMPs only do some coordination at the beginning and the end of the migration.

### 3. ChaRM4MPI System

#### 3.1 Architecture

ChaRM4MPI (Figure 1), is based on MPICH4, an MPI implementation developed at Mississippi State University and Argonne National Laboratories that employs P4 library as the device layer. Our work is focused on the follows:

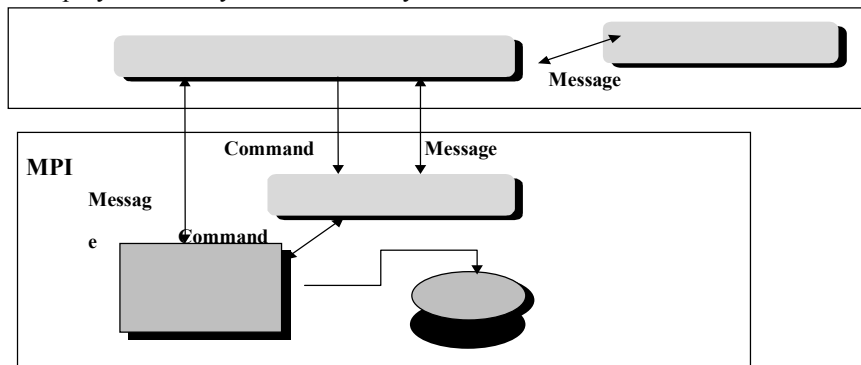


Figure 1 the Architecture of ChaRM4MPI

- 1) A coordinated CRR and migration protocol for parallel applications is designed, which have been described in Section 2.
- 2) One management process, *manager*, is implemented as the coordinator of parallel CRR and process migration. It can operate users' checkpoint /migration commands received through a GUI.
- 3) We modify source codes of P4 listener. On reception of commands from the coordinator, P4 listener will interrupt computing processes by signals. The most difficult task is how to design this procedure carefully to avoid causing any dead lock.
- 4) Some signal handlers are integrated with P4 master to deal with CRR/migration signals, which employ *libesm* to take checkpointing and accomplish CRR/migration protocols.
- 5) We modify the startup procedure of MPICH4, so all MPI processes will register themselves to the coordinator. In this way, the latter can maintain a global process-info-table to manage the whole system.

In ChaRM4MPI, computing processes in one node save their checkpoints information and in-transit messages to local disk. Checkpointing to local disk can recovery any number of node transient faults. Moreover it uses RAID like

checkpoint mirroring technique to tolerate one or more node permanent faults. Each node uses a background process to mirror the checkpoint file and related information to other nodes besides its local disk. When some node fails, the recovery information of application processes running on the node will be available on other nodes. Therefore the application process that ran on the fault node would be resumed on the other corresponding node where the checkpoint information is saved, and go on running from the checkpoint.

### **3.2 Implementation of CRR**

MPICH is based on a two-level approach. The higher level consists of the actual MPI functions and the part of the MPI library that maintains information such as the number of tasks in a program, any virtual topologies that may be in use, etc. Communication between tasks is passed to the second level.

The second level of MPICH is the device layer where data that has been constructed and formatted by the first layer is actually sent from task to task. The most commonly used is the device based on P4 library, which is designed to allow communication based on TCP/IP sockets between tasks. The scope of this paper only includes P4 device used in COCs.

#### **3.2.1 The Structure of MPICHP4**

MPICHP4 tasks actually consist of two separate tasks, a “master” and a “listener”. The master is a task that performs its part of parallel program while the listener is primarily a dormant one that only handles the requests from other tasks to establish communication with the master. Moreover, the first launched master is called as *Big Master* while the others spawned by *Big Master* are called as *Remote Master*.

MPICHP4 uses *process tables* to track tasks in a parallel program. Each individual task maintains a *process table* with information about itself and every other tasks. For example, each *process table* has a copy of the MPI process ID (which is independent of the UNIX process ID) and socket number of every other task. *Process table* also tracks the connections that have been made with others. *Process table* is designed to be static. Once the table is sent to all tasks,

the list of process IDs and hostnames is never altered. Checkpointing and migration tasks require the ability to modify *process table* held by each task. An individual task does not open a communication channel with any other task at startup. Instead, a dynamic connection scheme is used, i.e., communication is only established on a point-to-point basis when a specific task requests it.

### 3.2.2 Modification of Startup Procedure

In ChaRM4MPI, all tasks will send some information including MPI process ID, Unix Process ID, listening socket number and running hostname, to *manager* just on startup. Therefore, *manager* can notify tasks to take checkpointing or to migrate and coordinate CRR/migration procedures based on this information. Similarly, a task will send *exit\_msg* to *master* on exit.

Message Name	Description
CHKPOINT_MSG	To notify tasks to take a global-consistency checkpoint
RECOVER_MSG	To notify tasks to rollback to the last saved checkpoint files
MIG_MSG	To notify a task to migrate to another host
REFRESH_MSG	To send the new process table to all tasks(after checkpointing and migration)

Table 1. Messages sent by *manager* to tasks

### 3.2.3 Modification of listener

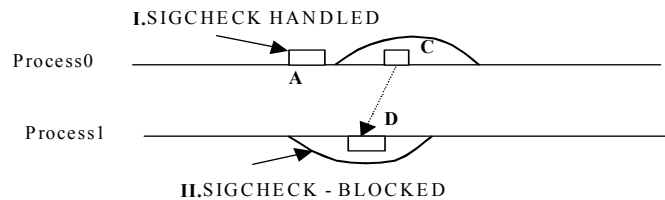
*Listener* is modified to act as the coordinator to cooperate with *manager* to implement CRR protocol as described in [9]. In other words, *listener* deals with not only requests from any other tasks but also messages sent by *manager*.

It is apparent that some extra asynchronous events must be created in addition to the original one---*connection event*. So special methods have to be employed to avoid potential dead-lock.

- On *connection event*, *listener* sends SIGUSER1 to interrupt *master* in the unmodified version. Now SIGUSER2 (called SIGCHECK) is used for *checkpointing event* and SIGSOUND (called SIGRECOVER) for *recovery event*. Moreover, the handler of signals has been designed carefully to avoid

breaking the communication between tasks.

- After startup, connections among tasks will be established at first. In contrast with the unmodified version, this method introduces unnecessary



connections, and then some OS resource will be wasted. However, it simplifies signal-handlers of *listener*, and CRR-relative events will not happen when any connection-request is in process.

- Although the above methods are employed, one type of deal-lock will still take place, which is described in Figure 2. Task 0 will send a common message to task 1 at point C, and they both mask SIGCHECK to avoid breaking communication. However, before sending task 0 is interrupted by SIGCHECK at point A and calls the proper signal handler to enter CRR process, while task 1 is waiting for this message from task 1 at D and masks all signals. It is apparent that the whole CRR process is blocked.

Therefore, the signal handler is improved to send a special message to all others at first. At the same time reception handlers are modified, then they will check the message-buffer first and unmask signals to deal SIGCHECK signal if detecting this special message. The whole process is showed as Figure 3.

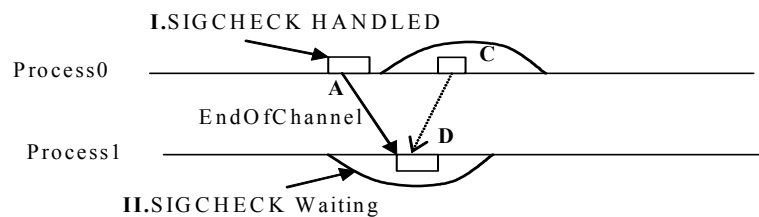


Figure 3 Solution for dead lock

In fact this special message is also used to flush message-pipes to implement CRR protocol.

## 4. Performance

Two types of performance are tested, which include *checkpointing time overhead* and *migration time overhead*. Four scientific computing programs are selected to run on ChaRM4MPI, and we use a cluster of 4 PowerPCs running AIX4.1 OS that are connected by 100Mbps Ethernet.

### 4.1 Checkpointing Time Overhead

Table 2. Performance with and without Checkpointing(s:second, MB:Megabyte)

PROGRAM	CPU NUMBER	RUNNING TIME WITHOUT CHECKPOINT (S)	RUNNING TIME WITH CHECKPOINT (S)	RUNNING TIME OVERHEAD (%)	CHECKPOINT INTERVAL (S)	NUMBER OF CHECKPOINT	TIME OVERHEAD per CHECKPOINT
Integer Sorting(i s.W.4)	4	35.06	41.32	17.9	20	1	17.9
LU decomposition(lu. W.4)	4	219.82	235.77	7.26	60	4	1.81
overfall simulation n 100	4	592.28	664.26	12.2	60	10	1.22
overfall simulation n 300	4	1214.96	1251.19	3.0	180	7	0.429

On Table 1, we can get that in ChaRM4MPI system, the consistent checkpointing adds not much overhead to the running time of the application programs, especially for those running longer. The average extra overhead per checkpointing is less than 2% except for the first program. Of course it is unnecessary to employ CRR for this program with such short running time.

### 4.2 Migration Time Overhead

Table 3. Performance of Migration

Size of process state space	0 922MB	1.971MB	3.019 MB	4.067 MB	5.116 MB
Migration Overhead(s)	5.603	4.540	5.460	6.947	8.436
Transfer Time(s)	1.752	2.334	3.427	4.665	5.914
Size of process state space	6.329 MB	7.212 MB	8.262 MB	9.393 MB	10.604 MB
Migration Overhead(s)	10.068	10.213	11.372	14.394	15.197
Transfer Time(s)	6.983	7.840	8.794	10.120	11.752

IS.W.4 is selected because the size of its process state space can be changed based on different arguments, and the results are showed in Table 2.

## References

- [1]. Elnozahy E N, Johnson D B, Wang Y M. *A Survey of Rollback Recovery Protocols in Message-Passing System*. Technical Report. Pittsburgh, PA: CMU-CS-96-181. Carnegie Mellon University, Oct 1996
- [2]. Elnozahy E N. *Fault tolerance for clusters of workstations*. Banatre M and Lee P (Editors), chapter 8, Springer Verlag, Aug. 1994.
- [3]. J. Casas, Dan Clark, et. al. *MPVM: A Migration Transparent Version of PVM*, Technical Report, Dept. of Comp. Sci. & Eng., Oregon State Institute of Sci. & Tech.
- [4]. M. J. Litzkow, et. al. *Condor— A Hunter of Idle Workstations*, In Proc. of the 8th IEEE Intl. Conf. on Distributed Computing Systems, pp104-111, June 1988
- [5]. S. H. Russ, B. Meyers, C.--H. Tan, and B. Heckel, *User--Transparent Run--Time Performance Optimization*, Proceedings of the 2nd International Workshop on Embedded High Performance Computing, Associated with the 11th International Parallel Processing Symposium (IPPS 97), Geneva, April 1997.
- [6]. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. *Libckpt: Transparent checkpointing under Unix*. In Proceedings of the 1995 USENIX Technical Conference, pages 213-224, January 1995.
- [7] Stellner G, Pruyne J. *Resource Management and Checkpointing for PVM*. In Proc. 2th Euro. PVM Users Group Meeting. Lyon, Lyon: Editions Hermes, Sept.1995:31-136.
- [8] Dan Pei, *Research and Implementation of Checkpointing and Process*

*Migration for Network of Workstations*. Master Degree Thesis, Computer Science Department, Tsinghua Univ. July 2000.

[9] G. Stellner, *CoCheck: Checkpointing and Process Migration for MPI*, In 10th Intl. Par. Proc. Symp., Apr. 1996.

[10] Pei Dan Wang, Dongsheng, Zhang Youhui, Shen Meiming, Quasi-asynchronous Migration: A Novel Migration Protocol for PVM Tasks, ACM Operating Systems Review 33(2): 5-15(April 1999).