Checkpoint and Recovery for Parallel Applications with Dynamic Number of Processes

Nam Thoai and Doan Viet Hung
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology, Vietnam
{nam,dvhung}@cse.hcmut.edu.vn

Abstract

This paper presents a checkpoint and recovery (C&R) protocol to support fault-tolerance for PVM (Parallel Virtual Machine). The protocol helps to mask fail-stop failures from an application. The C&R activities are transparent and do not require any change in the PVM library nor operating system.

In PVM, an application can change the number of processes during execution. This paper focuses on solving problems raised by the dynamic spawn and the asynchronous exit of tasks in PVM. The proposed protocol is a non-blocking one, so it reduces side-effect of checkpoint activities of original programs.

1. Introduction

Distributed systems today are popular and have become main platforms to run computational science applications. To meet the requirements of applications, these systems need to be more powerful and, therefore, become more complex. The higher complexity both in hardware and software make these systems more susceptible to failure. Moreover, for long running applications, a large amount of computing work will be lost when a failure happens. Therefore, fault-tolerance becomes an important requirement in distributed systems.

PVM [6] is a popular parallel distributed computing environment that assists users to create parallel applications. One main advantage of PVM over other parallel distributed computing environments is its ability to support applications to change the number of processes during execution. PVM enables applications to spawn new processes as well as enables a process exit without waiting for other processes. This characteristic gives developers more freedom in programming and may help applications to utilize computational resources better. However, it makes C&R more difficult.

This paper presents a transparent, non-blocking checkpointing protocol to support fault-tolerance for PVM applications even if the applications have dynamic number of processes. The protocol is implemented as a user-level library and, therefore, the change in PVM library and operating system is not necessary.

The rest of this paper is organized as follows. Section 2 presents background about C&R and related works. Section 3 describes the coordination protocol for global checkpointing of PVM applications. Section 4 presents solutions to enable checkpointing PVM applications even when applications spawn new tasks during execution. Section 5 deals with problems raised when a task in PVM exits sooner than other tasks in the program. Section 6 describes the recovery process after failure. Section 7 gives more details about problems related to implementation of the protocol. Finally, Section 8 offers conclusions.

2 C&R for Message Passing Applications

2.1 C&R

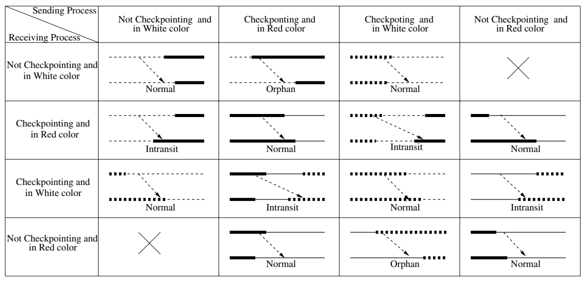
C&R techniques allow to save intermediate states of parallel programs to a stable storage. On restart, a program will be started from the last available state. These techniques help to reduce the amount of computing work lost when a failure occurs. C&R not only are used for fault-tolerance but also for migration, debugging, etc.

A *global state* of a message-passing program is a collection of the individual states of participating processes and of the states of communication channels. The line, that connects all checkpoints, divides the execution of the program into two parts. The left part is called the past, the right part is called the future. *In-transit messages* are messages sent in the past and received in the future. In the other hand, *orphan messages* are the messages sent in the future and received in the past. To preserve the soundness, program must be recovered from a *consistent global state*. Consistent global state is a global state in which if a process's state reflects a message receipt, then the state of the corresponding sender reflects sending that message [5] and, therefore, consistent global state contains no orphan message.

Assume the system does not initiate the creation of a global checkpoint before all previous global checkpoints have been created and committed to global storage. The execution of an application process can therefore be divided into successive *epochs*, where an epoch is the period between two successive local checkpoints.

Checkpointing protocols can be classified by the technique used to coordinate parallel processes when checkpoints need to be taken. The first one is *uncoordinated checkpointing*, in which each process saves its state independently. The other one is *coordinated checkpointing*, processes must coordinate during checkpoint to determine a consistent global state. Uncoordinated checkpointing is simple but it has several disadvantages, such as high memory cost, heavy recovery, complicated garbage collection and high susceptibility to domino effect. Therefore, coordinated checkpointing are mostly used.

Coordinated checkpointing can be divided into *blocking* and *non-blocking* checkpointing. In the former type, processes have to stop while checkpointing protocol executes. This approach is straightforward but it can result in large overhead. Therefore, non-blocking schemes are preferable.



Sending Process
Receiving Process
Not Checkpointing and in White color
Checkponting and in Red color
Checkpoting and in White color
Not Checkpointing and in Red color
Not Checkpointing and in White color
Normal
Orphan
Normal
Checkpointing and in Red color
Intransit
Normal
Intransit
Normal
Checkpointing and in White color
Normal
Intransit
Normal
Intransit
Not Checkpointing and in Red color
Normal
Orphan
Normal

Figure 1. Classification of Program Messages

2.2 PVM (Parallel Virtual Machine)

PVM is a library that provides an unified platform for computing over a network of heterogeneous parallel and serial computers. A typical PVM system consists of a cluster of interconnected computers. The system hosts a Global Resource Manager (GRM), several PVM daemons and a runtime library called *pvmlib* linked into each application process running under it.

Every node on the cluster runs a daemon, called *pvmd*, which is responsible for maintaining communication, authentication and control protocols with the virtual machine. A PVM *task* is a process linked with *pvmlib*. Both pvmd and task are assigned a globally unique descriptor *TID*. These *TIDs* is used to designate source and destination tasks for messages.

2.3 Related works

C&R has been studied intensively in the last two decades. Surveys in [5, 13] present most of the works related to C&R techniques for MPI. However, the underlying assumption in MPI is the fixed number of processes of an application. Therefore, the same techniques can apply for PVM applications with fixed number of processes.

Fail-Safe PVM [11] implements C&R inside PVM. For checkpointing a distributed application, it stops all tasks with a barrier so that the system is quiescent; in-transit messages must be pushed from the communication medium. In [3], checkpointing is not controlled by PVM and does not involve pvmds. In case of a machine failure, only failed and orphan tasks, and failed pvmds must rollback. CUMULVS [8] uses an approach that user must modify the code to coordinate checkpoint activity. FTOP [1] implements a non-blocking coordinated C&R algorithm. Other approaches to integrate C&R protocol with PVM system are discussed in [2].

Migration systems also use checkpointing technique. DynamicPVM [4] is a migration facility for PVM processes which has been integrated into the PVM daemons. The integration into the PVM system requires to integrate the checkpointing code into every new PVM version. CoCheck [14] allows the migration of PVM processes by using coordinate blocking algorithms. In UPVM [9], PVM tasks are mapped onto lightweight UPVM processes, which can be migrated within the cluster. Current improvements in migrating systems like given in [7] and tmPVM [15] are using user-level checkpoint.

This brief overview of checkpoint literature for PVM shows that, it still lacks a checkpoint protocol at user-level that supports checkpointing dynamic PVM programs in a non-blocking manner.

3 Coordination Protocol for Global Checkpointing

3.1 Terminology

In PVM, a *task* is a unit of computation. It is often a process, but not necessarily. For sake of simplicity, this study only considers applications in which each task is a single process. In the protocol of this paper, the daemons are not checkpointed, only tasks are checkpointed. Therefore, the words *process* and *task* can be used interchangeably. And there is one *task* (not *pvmd*) in the protocol, called *Master* *process*, which takes the role as coordinator for the checkpoint process as well as the recovery process after failure.

PVM supplies routines that enable an user process to become a PVM task and to become a normal process again. In this study, if a process leaves the PVM tasks pool to become a normal process, it will be considered as it has terminated, and will not be included to the checkpoint.

In the protocol, each epoch is labeled similar to [10]. In this scheme, counting the epoch from zero, the odd epoch is called *red color*, and the even epoch is called *white color*. One-bit data *Process_Color* is used to represent color of process. Another one-bit data, called *Recording*, is used to represent states of processes within each epoch. This bit is *true* if process is checkpointing, and is *false* if process is before the checkpoint.

Because no message goes across two epochs in the protocol, application messages can be classified depending on the color of epochs and the state of process in the epoch as shown in Figure1. In the protocol, two bits *Process_Color* and *Recording* are piggybacked on every application message to reflect status of sending process. The receiver uses this piggybacked information to determine if the message is an in-transit, intra-epoch (normal), or orphan message.

In Figure1 and other figures in this report, dotted line is used for presenting the process which is in white color, continuous line for red color, thin line for state before checkpoint (*Recording = false*), and bold line for state during checkpoint process (*Recording = true*).

Each process holds an integer *Sent_Message_Count*. This variable is initialized to 0 at the beginning of each epoch, and is incremented whenever the process sends a message. This value is used by *Master process* to calculate the number of in-transit messages. Each process maintains a table called *In-transit_Table* to save in-transit messages.

When an application is recovered, the *TID* of its task may be changed. So these tasks could not communicate as before the failure. To solve this problem, each process maintains a table called *PVM_Table*, to map between old *TID* and the new allocated *TID*. Each row in the table has a three values:

- *Real_ID* is the *TID* in the first running time.
- *Virtual_ID* is the *TID* PVM assigns for a task in the most recent running time.
- *Parent_ID* is *Virtual_ID* of process has spawned this process.

There are some special messages using for checkpoint protocol called control messages. The details about the functions of these control messages are described as they are used or can be inferred from the context.

3.2 High-level description of protocol

Master process is responsible for deciding when the checkpointing process should begin. A timer daemon periodically sends a signal to the *Master process*. If the *Master process* is ready for the checkpoint, a new checkpoint process begins. The protocol is divided into phases as follow:

- *Master process* broadcasts a control message called *START_CKPT* to the other processes and
  - takes its local checkpoint,
  - sets *Recording = true* and changes *Process Color*.
- If a process *P* receives a *START_CKPT* message or an *orphan message* then *P* will
  - send the number of messages that *P* has sent, *Sent_Message_Count* (MSG_COUNT), to *Master process*,
  - set *Recording = true* and change *Process Color*,
  - take the local checkpoint,
  - notify *Master process* that the local checkpoint has been completed by sending a control message, called *FIN_LOC_CKPT*, to *Master process*,
  - then continue executing.
- *Master process* receives *Sent_Message_Count* from other processes to calculate the number of *in-transit messages* in the program.
- During checkpointing process, whenever process *P* receives an *in-transit message*, it must save that message in *In-transit Table* and send an acknowledgment message *ACK* to *Master process*.
- Whenever *Master process* receives an *ACK*, it calculates the number of *in-transit messages* in the system. If *Master process* knows that there is no more *in-transit message* and *Master process* has received *FIN_LOC_CKPT* messages from all processes, then, *Master process*
  - sets *Recording = false*,
  - commits the checkpoint that was just created as the one to be used for recovery, and
  - sends a control message called *FINISH_CKPT* to each another processes.
- When a process receives a *FINISH_CKPT* message from *Master process*, it commits the checkpoint and changes *Process Color*. The global checkpoint is then completed.

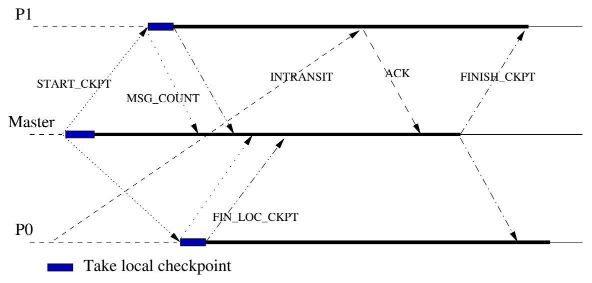


Figure 2. General Algorithm

The protocol saves all in-transit messages and guarantees that there is no orphan message exits, therefore, the global state saved is a consistent global state. A typical checkpoint protocol for program with three processes is shown in Figure2.

4 Dynamic Process Creation

In PVM, applications can spawn new processes during execution. When a process *P* wants to spawn a new process, it calls the fuction *pvm_spawn*. However, in our case, this is not the original *pvm_spawn* funtion of PVM. The underlying mechanism of *pvm_spawn* in this paper is that *P* will send a control message *SPAWN* to *Master process*. When *Master process* receives a *SPAWN* message, it spawns a new process and returns the result *SPAWN_RESULT* to *P*. With this mechanism, *Master process* controls the spawn activities of all processes. Depending on how *SPAWN* is delivered, there are three cases as analyzed below.

4.1 SPAWN message is both sent and received before Checkpoint

In this case, the *Master process* knows that the number of processes in the program changes. At the next checkpoint, the new process will be checkpointed like other processes as shown in Figure3. There is no modification to the main protocol.

4.2 SPAWN message is sent before Checkpoint and received within Checkpoint Process

A process *P0* sends *SPAWN* message and receives *SPAWN_RESULT* message before *START_CKPT* message coming from *Master process* as shown in Figure4. In this case, process *P0* starts its checkpoint process immediately when receiving *SPAWN_RESULT*.

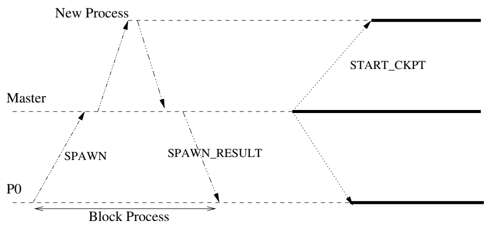


Figure 3. SPAWN Message is sent and received before Checkpoint

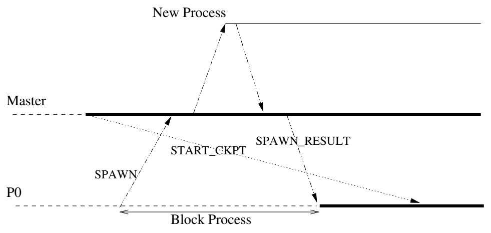


Figure 4. SPAWN Message is sent before Checkpoint and received within Checkpoint Process

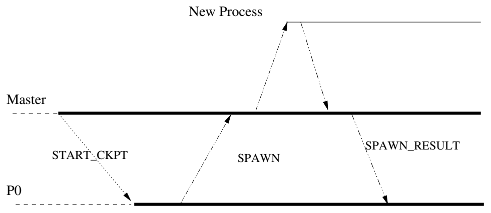


Figure 5. SPAWN Message is sent and received within Checkpoint Process

4.3 SPAWN message is both sent and received within Checkpoint Process

This is a simple case, all control messages related to spawn are normal messages. The state of newly created process, also not be saved in current checkpoint as shown in Figure5.

5 Process Ending

In PVM, one process can exit without waiting for other processes finish. When a process *P* wants to exit, *P* just calls the routine *pvm_exit* to indicate the local pvmd that it is leaving PVM. However, in our case, the underlying mechanism of *pvm_exit* function is not just a call to original PVM function *pvm_exit*. The mechanism used is similar to *pvm_spawn*; process *P* must send a control message *EXIT* to

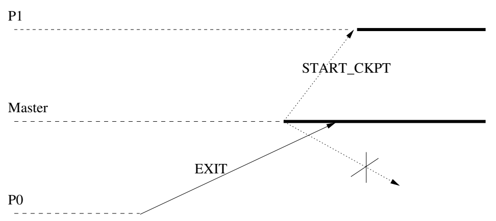


Figure 6. EXIT Message is Sent before Checkpoint and Received within Checkpoint Process

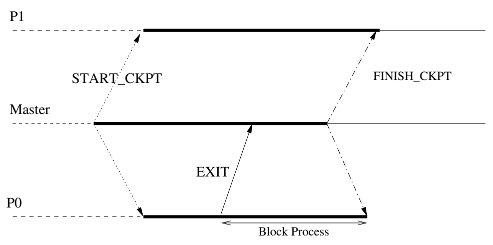


Figure 7. EXIT Message is Sent and Received within Checkpoint Process

Master process notifying its exit. There are three scenarios depending on how the *EXIT* control message is delivered.

5.1 EXIT message is sent and received before Checkpoint

In this case, the *Master process* knows that the number of processes in the program changes. At the next checkpoint, the exited processes will not be checkpointed. There is no modification to the main protocol.

5.2 EXIT message is sent before Checkpoint and received within Checkpoint Process

A process *P0* sends the *EXIT* message to *Master process* before receiving the *START_CKPT* message as in Figure6. When *Master process* receives an *EXIT* message, it just updates the number of processes involved in the current checkpoint process. This means that process *P0* does not appear if the program is recovered at the current checkpoint.

5.3 EXIT Message is sent and received within Checkpoint Process

If a process *P0* sends the *EXIT* message after receiving the *START_CKPT* message as in Figure7, *P0* has to involve in the checkpoint process because messages could be sent and/or received during the checkpoint process. So *P0* must wait for the *FINISH_CKPT* message from *Master process*.

6 Recovery

In coordinated C&R, most of the work is done at the checkpoint time. The recovery process is quite simple: a *Master process* is created first, and this process acts as coordinator for the recovery process as shown in Figure8. The recovery protocol includes the following steps:

- *Master process* spawns a number of processes equal to the number of processes before the fault.

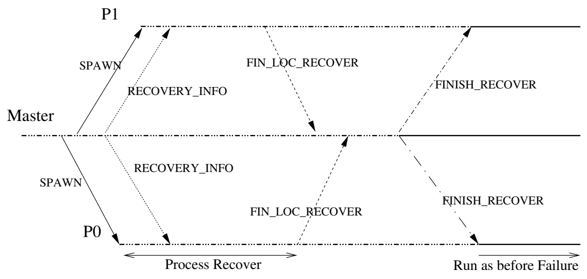


Figure 8. Recovery

- After spawning all processes, *Master process* sends corresponding *TID* to each process. After that, *Master process* waits for the control message *FIN_LOC_RECOVER* from the other processes.

- Other processes, after receiving *TID* from *Master process*, take the local recovery process by:

  - updating *TID* in *Pvm_Table* with the new allocated *TID*,
  - recovering *in-transit messages* saved in *In-transit_Table*, and
  - sending *FIN_LOC_RECOVER* to *Master process*.

- When the *Master process* receives *FIN_LOC_RECOVER* from all other processes, it knows that all local recovery processes have finished. *Master process* will broadcast *FINISH_RECOVER* to notify the other processes. Then, *Master process* continues to work as before the fault.

- When a process receives *FINISH_RECOVER* from *Master process*, it begins working as normal.

At the end of recovery protocol, the whole program runs as it did before the failure.

7 Implementation

The proposed checkpoint protocol is implemented as a user-level library. When programmers want to incorporate checkpointing to their programs, they need to compile their programs with the library. The task of sending control messages, piggy-packing information is done by functions in the library. No modification to PVM library or operating system is needed.

Libckpt [12] is used as uni-process checkpointing tool. Since it provides user-level, transparent checkpointing. Libckpt also supports optimization techniques, such as incremental, copy-on-write and user-directed checkpointing; these help reducing the overheads of individual processes checkpoint.

8 Conclusion

This paper presents a checkpoint protocol used to support fault-tolerance for PVM programs. The protocol provides transparent C&R at user-level. The transparency helps programmers to avoid the burden of including checkpointing into their programs. Because the checkpoint system is at user-level, it does not require any modification to the PVM library or operating system. Instead of using epoch number, the protocol uses only two bits piggy-packed to the messages to determine the state of processes and messages. This technique makes the overhead small.

The main contribution of this paper is the proposed protocol which can checkpoint in a non-blocking way PVM applications that having dynamic number of processes. By solving problems raised by the changing number of processes in PVM applications, the protocol can support fault-tolerance for a large range of PVM applications. Programmers can freely create new processes during the execution of a program, as well as remove a process without waiting other processes to finish. The resulting programs can still be checkpointed by the proposed protocol. Since the main protocol is non-blocking, it reduces the side-effect of checkpoint activities to normal execution of the checkpointed program.

Acknowledgments

We would like to thank our colleagues and students for implementation and discussions.

References

[1] R. Badrinath, R. Gupta, and N. Shrivastava. FTOP: A library for fault tolerance in a cluster. In *Proceedings of IASTED PDCS 2002*, pages 515–520, 2002.

[2] A. Clematis and V. Gianuzzi. Extending PVM with consistent cut capabilities: Application aspects and implementation strategies. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting*, pages 101–108, London, UK, 1999. Springer-Verlag.

[3] D. Conan, P. Taponot, and G. Bernard. Rollback Recovery of PVM Applications. In *Proceedings of Romanian Open Systems Event (ROSE '95)*, Bucharest, Nov. 1995.

[4] L. Dikken, F. van der Linden, J. Vesseur, and P. Sloot. DynamicPVM: Dynamic load balancing on parallel systems. In W. Gentzsch and U. Harms, editors, *Lecture notes in computer science 797, High Performance Computing and Networking*, pages 273–277, Munich, Germany, April 1994. Springer Verlag.

[5] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, Oct. 1996.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[7] P. K. J. Kovacs. Server Based Migration of Parallel Applications. In *Proceedings of DAPSYS 2002*, pages 30–37, Linz, Austria., Jan. 2002.

[8] J. A. Kohl and P. M. Papadopoulas. Efficient and flexible fault tolerance and migration of scientific simulations using CUMULVS. In *Proceedings of the SIGMETRICS symposium on Parallel and Distributed Tools*, pages 60–71, New York, NY, USA, 1998. ACM Press.

[9] R. B. Konuru, S. W. Otto, and J. Walpole. A migratable user-level process package for PVM. *Journal of Parallel and Distributed Computing*, 40(1):81–102, 1997.

[10] T. H. Lai and T. H. Yang. On distributed napshots. *Information Processing Letters*, 25(3):153–158, 1987.

[11] J. Léon, A. L. Fisher, and P. Steenkiste. Fail-safe PVM: A Portable package for Distributed Programming with Transparent Recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, Feb. 1993.

[12] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of USENIX Winter 1995 Technical Conference*, pages 213–224, New Orleans, Louisiana/U.S.A., Jan. 1995.

[13] E. Roman. A survey of checkpoint/restart implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, 2003.

[14] G. Stellner. Consistent Checkpoints of PVM Applications. In *Proceedings of the First European PVM User Group Meeting*, 1994.

[15] C. P. Tan, W. F. Wong, and C. K. Yuen. tmPVM - task migratable PVM. In *Proceedings of IPPS/SPDP '99*, pages 196–202, Washington, DC, USA, 1999. IEEE Computer Society.