

Transparent Message-Passing Parallel Applications Checkpointing in KERRIGHED

Matthieu Fertré
PARIS project team
IRISA/INRIA, France
Email: matthieu.fertre@irisa.fr

Christine Morin
PARIS project team
IRISA/INRIA, France
Email: christine.morin@irisa.fr

Abstract—Nowadays, clusters are widely used to execute scientific applications. These applications are often message-passing parallel applications with long execution time. Since the number of nodes in clusters is growing, the probability of a node failure during the execution of an application increases and the application execution time may be greater than the cluster mean time between failures (MTBF). To avoid restarting application from the beginning, some fault tolerant mechanisms such as checkpoint/restart are needed. Currently, checkpoint/restart mechanisms are either implemented directly in the application source code by applications programmers or are integrated in communication environments such as MPI or PVM. We propose in this paper a new approach in which checkpoint/restart mechanisms for parallel applications are implemented in a cluster single system image operating system. While this kernel level approach is more complex to implement than other approaches, it is more general because it does not require any modification, compilation or relinking of the applications whatever the communication environment they rely on. Our approach has been implemented in KERRIGHED single system image operating system based on LINUX. Performance results are presented in this paper.

Keywords: single system image, checkpointing, parallel application, global coordination.

I. INTRODUCTION

Scientific applications are often message-passing parallel applications with long execution time. To guarantee their reliable execution, fault tolerance mechanisms are needed since application execution time may be greater than the mean time before failure (MTBF) of the cluster.

A traditional approach is checkpoint and restart mechanisms. The state of the program is periodically saved on stable storage during execution; when a failure occurs, the computation is shut down and the program is restarted from the last checkpoint.

In the area of scientific computing, programmers usually implement application specific checkpoint/restart mechanisms which are integrated in the application source code. To relieve developers from the burden to manage fault tolerant issues, an approach is to implement checkpoint/restart strategies in the communication environments such as MPI and PVM on which message-passing based applications are based. However, re-compiling or relinking the application is often needed.

We propose a new approach which consists in implementing checkpoint/restart mechanisms for parallel applications as part of a cluster Single System Image (SSI) Operating System

(OS). A SSI operating system gives the illusion that a cluster is a single multiprocessor machine. It provides the same interface as a traditional operating system for an SMP machine with some additional functionalities such as process migration. KERRIGHED [1] and OpenSSI [2] are two examples of such systems based on LINUX. Our contribution is an extension of the interface of an SSI operating system for clusters with checkpoint/restart system calls for parallel applications. While the implementation of checkpoint/restart mechanisms is more complex at kernel level than at user level, this approach provides full transparency for applications and message-passing communication environments they rely on. With our approach, legacy applications, whatever the communication library they use, could be checkpointed without changing the binary.

In this paper, we propose new system calls for transparent message-passing parallel applications checkpointing in the KERRIGHED Single System Image cluster Operating System [3]. The proposed interface allows system initiated checkpointing. Moreover, from the identifier of *one* process of a parallel application, all the objects (processes, data streams) needed to be checkpointed for the whole parallel application are discovered.

The remaining of this paper is organized as follows: Section II presents a background on mechanisms for checkpointing parallel applications and for checkpointing processes in LINUX operating system. Section III describes the design of mechanisms we have integrated in KERRIGHED in order to implement a coordinated checkpointing strategy for message-passing based parallel applications. Section IV presents preliminary performance results obtained on the prototype we have implemented. Finally, Section V concludes.

II. BACKGROUND

There are several strategies, like coordinated or uncoordinated checkpointing protocols, log-based protocols, to allow rollback/recovery of message-passing parallel applications. They differ on the way they manage dependencies incurred by communications between the application processes to guarantee that a global consistent state can be restored [4].

Recently a lot of work has been done to introduce fault tolerance mechanisms in parallel programming environments, such as MPI and PVM, used in the domain of high performance computing.

The global coordinated checkpointing strategy has been implemented in Failsafe PVM [5], CoCheck [6], Starfish [7], MPICH-VCL [8], FTC-Charm++ [9] and LAM/MPI [10]. The Paris Sud university has implemented others scheme such as pessimistic [11] and causal [12] logging. However, comparing these different strategies they have shown in [12] that the coordinated checkpoint scheme offers the best complexity performance trade-off in clusters. Using these communication libraries, rebuilding or relinking the parallel application is often needed.

These environments rely on process checkpointing mechanisms to save the state of each process of a parallel application. To checkpoint a process at operating system level and without modification of the source code, there is a wide range of available libraries such as Condor [13] or Epckpt [14]. In order to save and restore the state of a program, a wide range of system calls are intercepted in order to keep track of the program state (such as memory mapped regions and open file handles). Some systems calls can be forbidden. In addition, user-level checkpoint/restart mechanisms cannot restore for example process session identifier.

BLCR [15] is a project of kernel-level process checkpointing for the LINUX kernel. BLCR can checkpoint both regular and multithreaded (pthreads) programs however it does not support checkpointing and/or restoring of process using open TCP/IP or UNIX domain sockets. Nevertheless, BLCR supports adding 'callbacks' to user-level code, which are called when a checkpoint is about to be performed, and when it is restarted (or carries on after the checkpoint). This is how MPI communication can be handled by the fault tolerant version of LAM/MPI.

Single System Image operating systems dedicated to clusters have received a lot of interest in the cluster community. These operating systems globally manage all operating system resources such as processes, memory segments, data streams, files, thus easing cluster usage and programming. OpenSSI [2] and KERRIGHED [1] are two examples of such systems based on LINUX. Legacy MPI applications designed for multiprocessor machines can be executed on clusters without any modification or recompilation. Our goal is to integrate parallel applications checkpoint/restart mechanisms in such an operating system for full transparency and independence from applications and communication libraries.

III. OVERVIEW

Our work aims at making checkpointing and restarting message-passing applications easy. In contrast to existing projects, the fault tolerance mechanisms are entirely implemented in the OS for applications and libraries independence.

We have defined and implemented a command-line user interface, a way to retrieve all processes of a parallel application from *one* of them, global coordinated checkpointing and restarting protocols. Finally, the restoration of open streams has been realized.

Id.	Vers	Date	Description
37143	1	Sep 9 09:48	No description
48657	1	Sep 8 15:49	Biologic Comp.
48657	2	Sep 8 17:03	Biologic Comp.
96318	1	Sep 5 17:02	Bench 2

 Which application to restart : 48657
 version : 1

Fig. 1. Restart user-interface example

A. Design of a System Checkpoint/Restart Interface

The checkpoint/restart interface must be simple and easy to use. Thus we have chosen to use the same interface for parallel applications and sequential processes.

To checkpoint an application, we suggest the following command-line interface `checkpoint [options] pid`, `pid` being the process identifier of *one* of the parallel application processes. Option `-process-only` (or `-p`) allows to checkpoint only a sequential process. Without the option, the parallel application frontier is calculated as described in Section III-B. Other options enable to choose the storage media (`-media=(disk|memory)`), to add a checkpoint description (`-d "description"`) and to create a checkpoint which is exportable to another cluster (`-exportable`).

A way for periodically checkpointing an application is to launch it with this interface: `reliable_run [options] -periodicity=P command`, `P` being a pattern of periodicity and `command` being the unmodified command to launch the application. It allows *system initiated checkpoint*.

The user can choose the checkpoint from which to restore the application in a list showing its saving date and description (see Figure 1) after executing the `restart` command.

B. How to Retrieve All Processes of an Application

The OS needs to detect all processes of an application from the `pid` of *one* of them by looking at filiation links and communication streams in order to checkpoint a parallel application.

In a SSI cluster OS such as KERRIGHED, processes and streams can be migrated [16] from one node to another. So the different processes of a parallel application can be spread on many nodes, even if there are filiation links between the processes. A process *B*, created by the `fork` system call from process *A*, is a *son* of process *A*. If *A* and *B* are not running on the same node, *B* is a *remote* son of process *A* else *B* is a *local* son of process *A*. If a process *C* is a son of *B*, *C* is a *descendant* of processes *B* and *A*.

To avoid checkpointing the whole system, a solution is to check the CHECKPOINTABLE capability. A process capability [17] is a process attribute that can be set, unset, and inherited. By default, a process does not have the CHECKPOINTABLE capability which can be enabled by a command line for each process of applications launched later from a shell. If a process

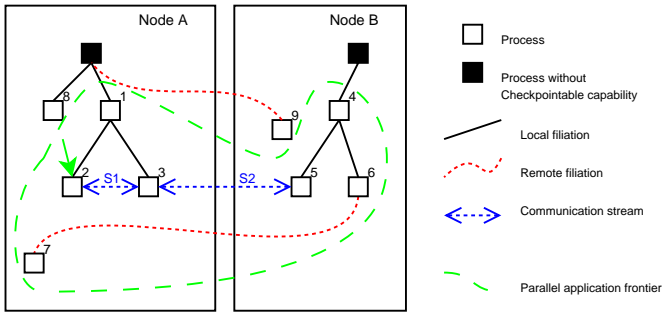


Fig. 2. A parallel application running on KERRIGHED

does not have the CHECKPOINTABLE capability it will not be checkpointed.

A process which needs to be checkpointed but whose father process is not checkpointable is a *head root* (for example, processes 1 and 4 in Figure 2 which shows a parallel application to be checkpointed). A process whose father is remote and needs to be checkpointed is a *local root* (for instance, process 7 in Figure 2. We called *sub-application* a local root process and all its local descendants.

From *one* of the application processes, traversing local and remote filiation tree and checking the CHECKPOINTABLE capability, we find *one* of the parallel application *head root*. Then, we add all the local and remote descendants of this process to the list of processes to be checkpointed.

When a process of this list is attached to a stream S we check if the other processes attached to S are already in the list. Else, if processes have the CHECKPOINTABLE capability, we go up in the local and remote filiation tree looking for another head root process. If it is the same head root, stream S can be ignored in the parallel application frontier calculation because it is redundant with filiation. In Figure 2, stream $S1$ is ignored in frontier calculation in contrast with stream $S2$.

C. Hierarchical Coordinated Checkpoint and Restart

1) *Coordinated checkpoint*: The checkpointing protocol is divided in three steps : (1) to retrieve all processes of an application and make them sleeping in order to stop communications, (2) to checkpoint them, (3) to commit the checkpoint if successful.

No process has a complete knowledge of processes involved in the checkpoint. From a root process, we care about a whole sub-application and remote sons of the sub-application processes but we do not know if remote sons have descendant or not. The information is distributed.

First of all, the parallel application frontier is calculated, then all the application processes have been stopped receiving a signal. Then, for the sake of checkpointing performance, a process and its local descendants are checkpointed in parallel with remote descendants (and their relative line of descent).

When receiving a sub-application checkpoint request for a local root process P , non blocking process checkpoint requests are sent for all its local descendants and non blocking sub-application checkpoint requests for all the remote sons of these

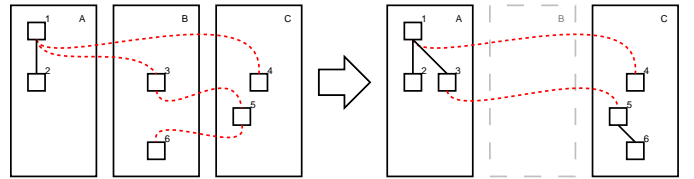


Fig. 3. Parallel application before and after a restoration with fewer nodes (Node B is out of order)

processes. When positive replies from all the descendants have been received, a positive reply is sent from P to its solicitor (local checkpoint commit). When all head root processes have locally committed their checkpoint, the checkpoint is fully committed.

As local root processes are coordinators for their respective subapplication, the coordination is hierarchical.

2) *Restart*: For each restarted process P , we make non blocking requests to restore all local sons of P and send non blocking requests to restore the sub-applications of which the local root is a remote son of P . The following step consists in waiting replies from the descendants to commit the restoration. Restarting begins with the application head roots processes.

Before checkpointing, processes have been deployed by the KERRIGHED global task scheduler or by the programmer. That is why processes are restarted on their original execution node, if possible, to respect the original deployment. However, at the time of restoration, processes, which were previously running on a failed node, are restarted on the node hosting their parent process. Figure 3 shows an example of an application previously running on 3 nodes which is restarted on only 2 nodes.

D. Open Streams Restoration

KERRIGHED provides data streams with extremities which can be moved from one node to another and are identified by a unique number [18]. This identifier may be already in use at the time of restoration. We use a shared hashtable on which the key is the old stream identifier and data is the new stream identifier. When restarting, if a process needs to be binded to a stream, it looks at the old stream identifier in the hashtable and if it does not exist, a new stream is created. At the end of restoration the hashtable can be deleted. Methods used to bind and detach a process to a stream in case of migration are the same for checkpoint/restart. A restored communication stream does not imply any overhead on computation since there is no message forwarding.

IV. EVALUATION

A prototype has been implemented in KERRIGHED SSI OS [3]. This prototype includes a checkpoint/restart user-interface (see Section III-A), parallel application frontier calculation (see Section III-B) which does not yet take care of streams, open streams restoration (see Section III-D) for INET socket streams, global hierarchical coordination checkpointing and restoration (see Section III-C).

This section presents experiments realized to evaluate our algorithms, to measure the storage impact on performance, to validate application restoration without node failures and to verify stream restoration.

A. Experimental Conditions

Experimentations have been done using clusters of 2 to 4 nodes communicating through Ethernet 100 Mbit/s. The nodes are personal computers with Pentium II CPU and 512 MB of memory.

The KERRIGHED global task scheduler is disabled during the experiments. So there is no preemptive process migration and processes are spread following this rule : a son process runs on node $n + 1$ modulo N , N being the total number of nodes and n the node which hosts its parent process.

Checkpointing times result from the average of 5 consecutive measurements. Restarting time has been measured after rebooting each nodes of the cluster.

During experiments, many storage approaches have been used. The NFS centralized server used to store checkpoint files is external to the cluster. When the chosen storage approach is the use of the nodes local hard drives formatted using the *Ext2* file-system, the checkpoint files must be manually copied on all nodes of the storage before the restoration can take place.

B. Results

In a first experiment, we have compared two checkpointing algorithms: the first one is traversing the tree of processes from the root and checkpoints each process one after the other, the second one is traversing the tree of processes, sending non blocking process checkpoint requests and then waiting for acknowledgments. As application processes are dispatched on the different nodes of the cluster, sending non blocking checkpoint requests offers of course better results than waiting sequentially for each process checkpoint. Results for an application composed of 16 processes are shown in Figure 4. To study storage system impact, we have used a NFS centralized server or the local disk of each cluster node. Results demonstrate the interest of the parallel algorithm and show that it is usable using a centralized file server. Of course, the storage system has an impact, so it should be interesting to test it on a distributed file system.

A second experiment has validated our restart mechanism on a fewer number of nodes. An application constituted of 16 processes running on 4 nodes has been checkpointed. Then the application has been restarted on 2, 3 or 4 nodes from the same checkpoint, using the local disk of each cluster node or the external NFS server for the storage. Figure 5 shows the results. During experiments, each process of the parallel application runs on the node $n + 1$ with n the node hosting its parent process. It is the worst case for the restarting algorithm since a process cannot be restored until its parent process is restored. That is why there is a weak difference using NFS server or the local disks of each cluster node.

A third experiment has been done to validate the stream restoration. Process waiting in a system call cannot be check-

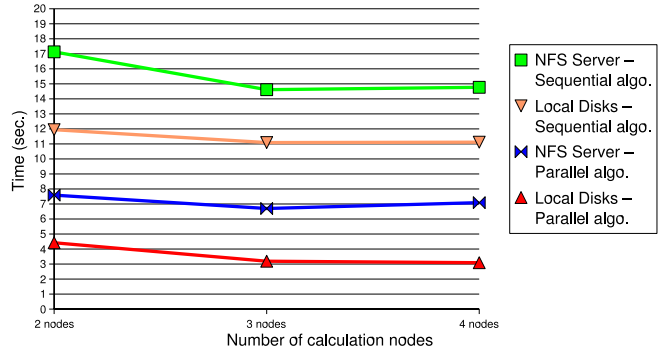


Fig. 4. Cost of parallel application checkpointing using sequential or parallel algorithms and different storage systems. Checkpoint size is 64 Mo.

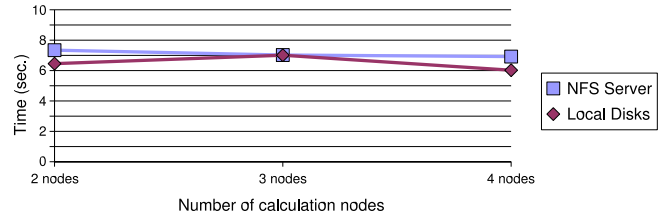


Fig. 5. Cost of application restarting on 2, 3 or 4 nodes from the same checkpoint. Checkpoint size is 64 Mo.

pointed by the current implementation of the process state extraction mechanism provided by KERRIGHED. Thus it makes not currently possible to checkpoint real world applications. As soon as this limitation will be solved, we should be able to checkpoint industrial applications. For now, we have tested our work about open streams restoration on an especially written application. This application is composed of two processes (a parent process and its son). First a `INET socket` stream is opened, then the child process sends data to its parent by the stream, the parent prints the received data. Later, the communication is stopped (but the stream remains open) to make a quite long computation during which a checkpoint is saved. At the end of the computation, the child process sends some data to its parent to ensure that the stream is still open. At last, the stream is closed and the application terminates. This application has been successfully checkpointed and restarted, rebooting or not all nodes of the cluster. This experiment validates the stream restoration.

V. CONCLUSION

We have presented the implementation of parallel application checkpoint/restart mechanisms in a cluster SSI OS. Legacy applications can take advantage of these mechanisms to tolerate node failures in a cluster without requiring any modification or rebuilding.

A convenient command-line interface has been designed to trigger parallel applications checkpointing and restarting. This interface allows system initiated checkpoint. All system objects of a parallel application can be detected from the

identifier of *one* of the application processes. We have implemented a global coordinated checkpointing strategy. Moreover we are able to restore communication streams without any overhead after restarting since direct communication is preserved between communicating processes even if processes are not restored on their original execution node.

Furthermore, we have experimented a prototype in KERRIGHED operating system which demonstrates the efficiency of the parallel checkpointing algorithm against a sequential implementation. A parallel application can be restarted even if some nodes are out of order at restarting time. Finally, the restoration of open streams (*INET socket*) has been validated by experiments.

Currently, the user interface is nearly complete but it does not support yet periodical application checkpoint. The restart algorithm is parallelized and the parallelization of the checkpointing algorithm is as best as we can since we do not want to centralize information about which processes needed to be checkpointed.

Checkpointing and restarting real MPI applications should be possible soon. The main problem is a limitation of the process state extraction mechanism [16] provided by KERRIGHED which is not able to checkpoint a process waiting in a system call.

We plan to experiment on larger clusters to evaluate the scalability of our approach and with real industrial applications. We aim at integrating our results in the public release [1] of KERRIGHED single system image operating system.

Our future work will focus on shared memory application checkpointing in single system image operating systems and on implementing other rollback/recovery strategies such as logging to allow programmers to select the strategy which is the more appropriate to their application.

Acknowledgments: We would like to thank Pascal Gallard and Renaud Lottiaux who have helped to integrate the proposed parallel application checkpoint/restart mechanisms in the KERRIGHED operating system.

REFERENCES

- [1] "Kerrighed website," <http://www.kerrighed.org/>.
- [2] "OpenSSI website," <http://www.openssi.org/>.
- [3] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, D. Margery, J.-Y. Berthou, and I. Scherson, "Kerrighed and data parallelism: Cluster computing on Single System Image operating systems," in *Proc. of Cluster 2004*. IEEE, Sept. 2004.
- [4] E. N. M. 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.
- [5] J. Leon, A. L. Fisher, and P. Steenkiste, "Fail-Safe PVM: A portable package for distributed programming with transparent recovery," Tech. Rep., 1993.
- [6] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*. IEEE Computer Society, 1996, pp. 526–531.
- [7] 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*, Los Alamitos, 1999.
- [8] A. Bouteiller, H.-L. Bouziane, P. Lemarinier, T. Héroult, and F. Cappello, "Hybrid preemptive scheduling for mpi applications on the grids," Presentation in 5th IEEE/ACM International Workshop on Grid Computing, Novembre 2004.

- [9] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *2004 IEEE International Conference on Cluster Computing*, San Diego, CA, September 2004.
- [10] 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," in *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.
- [11] A. Bouteiller, F. Cappello, T. Héroult, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *High Performance Networking and Computing (SC2003)*. Phoenix USA: IEEE/ACM, November 2003.
- [12] A. Bouteiller, P. Lemarinier, T. Héroult, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in *The 2004 IEEE International Conference on Cluster Computing*, San Diego USA, September 2004.
- [13] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience," *Concurrency and Computation: Practice and Experience*, 2004.
- [14] E. Pinheiro, "Truly-transparent checkpointing of parallel applications," citeseer.ist.psu.edu/434768.html.
- [15] J. Duell, P. Hargrove, and E. Roman., "The design and implementation of Berkeley Lab's Linux Checkpoint/Restart," Future Technologies Group white paper, Tech. Rep., 2003.
- [16] G. Vallée, R. Lottiaux, D. Margery, C. Morin, and J.-Y. Berthouand, "Ghost process: a sound basis to implement new mechanisms for global process management in linux clusters," in *ISPDC 2005*, Lille, France, July 2005.
- [17] D. Margery, R. Lottiaux, and C. Morin, "Capabilities for per process tuning of distributed operating systems," INRIA, IRISA, Rennes, France, Rapport de Recherche RR-5411, Dec. 2004. [Online]. Available: <http://www.inria.fr/rrrt/rr-5411.html>
- [18] P. Gallard and C. Morin, "Dynamic streams for efficient communications between migrating processes in a cluster," *Parallel Processing Letters*, vol. 13, no. 4, Dec. 2003.