Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI

Camille Coti camille.coti@lri.fr
Thomas Herault thomas.herault@lri.fr
Pierre Lemarinier pierre.lemarinier@lri.fr
Laurence Pilard laurence.pilard@lri.fr
Ala Rezmerita ala.rezmerita@lri.fr
Eric Rodriguez eric.rodriguez@lri.fr
Franck Cappello franck.cappello@lri.fr

INRIA-Futurs / Grand-Large
Laboratoire de Recherche en Informatique
Université Paris-XI
France

Abstract

A long-term trend in high-performance computing is the increasing number of nodes in parallel computing platforms, which entails a higher failure probability. Fault tolerant programming environments should be used to guarantee the safe execution of critical applications. Research in fault tolerant MPI has led to the development of several fault tolerant MPI environments. Different approaches are being proposed using a variety of fault tolerant message passing protocols based on coordinated checkpointing or message logging. The most popular approach is with coordinated checkpointing. In the literature, two different concepts of coordinated checkpointing have been proposed: blocking and non-blocking. However they have never been compared quantitatively and their respective scalability remains unknown. The contribution of this paper is to provide the first comparison between these two approaches and a study of their scalability. We have implemented the two approaches within the MPICH environments and evaluate their performance using the NAS parallel benchmarks.

1 Introduction

A long-trend in high-performance computing systems is the increase of the number of nodes. This is illustrated by the composition of the Top500 supercomputer list. The average number of processors per machine in the top 500 supercomputers is currently greater than 1000. Moreover, more than three quarter of these supercomputers have between 257 and 1024 processors, and the three most powerful systems have more than 10,000 processors. As the number of processors increases, the probability of failure of a single component also increases. So fault-tolerance becomes a key property for parallel applications running on these systems.

The concept of grids has emerged recently, consisting of gathering resources of different parallel computers (clusters or constellations), often increasing the system size to thousands of processors (TeraGrid, EGEE, Grid'5000, DEISA, NAREGI, etc.). These Grids span multiple domains, often administrated whith different active policies. Because of the system and administrative complexities, it becomes cumbersome for the users to manage failures occurring during application execution. Thus, it is essential to provide a certain level of automation to allow application to run until completion, when failures occur during execution.

The Message Passing Interface (MPI) is currently the programming paradigm and communication library most commonly used on supercomputers. Thanks to its high availability on parallel machines from low cost clusters to clusters of vector multiprocessors, it allows the same code to run on different kind of architectures. Moreover, it also allows the same code to run on different generations of machines, ensuring a long lifetime for the code. MPI conforms to popular high-performance, message passing programming styles. Even if many applications follow the SPMD programming paradigm, MPI is also used for Master-Worker execution, where MPI nodes play different roles. For these reasons, MPI is the preferred programming environment for many high-performance applications. MPI in its specification [Snir et al. 1996] and most deployed implementations (MPICH [Gropp et al. 1996]) follows the *fail stop* seman-

SC2006 November 2006, Tampa, Florida, USA

0-7695-2700-0/06 $20.00

tic (specification and implementations do not provide mechanisms for fault detection and recovery). Thus, MPI applications may be stopped at any time during their execution due to an unpredictable failure.

In order to avoid complete restarts of an MPI application because of only one failure, a fault tolerant MPI implementation is essential. The typical fault tolerant technique implemented in a MPI library is coordinated checkpointing [Bosilca et al. 2002] [Burns et al. 1994]. This technique consists of regularly taking a global state of the system and, if a failure occurs, restarting this application from this global state. There are two main ways to implement this technique. The first one, called the blocking coordinated checkpointing, consists of stopping the MPI computation to take the global state. This permits better control on the state of the different processes and their communication channels. The second one, called non-blocking coordinated checkpointing, does not provide this kind of control, but does not require the interruption of the MPI computation.

The blocking solution is simple to implement in a high-performance driver because it requires few modifications in the low-level communication layer. The non-blocking solution, even if it does not stop the computation, can require modifications that introduce overheads in the driver. As the number of processes regularly increases, it is important to evaluate the impact of these kinds of fault tolerant protocols on large-scale MPI computations. In this paper, we compare these two protocols, the blocking and the non-blocking ones, and evaluate their impact on large-scale applications. We detail the implementation of the blocking protocol inside MPICH2, compare it with our previous non-blocking implementation MPICH-Vcl [Lemarinier et al. 2004] and evaluate its impact on overall performance.

The paper is organized as follows. Section 2 presents the related works highlighting the originality of this work. Section 3 presents the common principle of the global checkpointing protocols, then the blocking and non-blocking solutions. Section 4 presents the two implementations used to compare these two fault tolerant MPI protocols in a fair way. Section 5 presents the experimental results in terms of application performance and fault tolerance using the NAS benchmarks. Section 6 sums up what we learned from these experiments.

2 Related Works

MPI is a standard for message-passing systems widely used for parallel applications. Several implementations of this standard are available, among them are two main open-source projects: MPICH [Gropp et al. 1996] and OpenMPI [Gabriel et al. 2004].

Fault tolerance in MPI applications can be implemented following three strategies: explicit (managed by the programmer), semi-automatic (guided by the programmer) and automatic (transparent for the programmer/user). In this paper we focus on the last strategy, that achieves fault tolerance without any intervention from the programmer.

Several techniques are used to implement fault tolerance in high-performance computing. Simple replication is not relevant for such systems, since if the system is designed to tolerate n faults, every component must be replicated n times and the computation resources are thus divided by n. An important part of the ressources are then used for something that does not contribute directly to the computation. The two main techniques used are message-logging and coordinated checkpoints. A review of the different techniques can be found in [Elnozahy et al. 2002].

Message-logging consists of saving the messages sent between the compute nodes, and replaying them if a failure occurs. It is based on the *piecewise deterministic assumption*: the execution of a process is a sequence of deterministic events separated by non-deterministic ones (generally the reception events) [Strom and Yemini 1985]. With this assumption, replaying the same sequence of non-deterministic events at the same moment makes possible the recovery of the state preceding a failure. Thus, for every process, these protocols consist of saving all its non-deterministic events in a reliable manner and to checkpoint regularly. When a failure occurs, only the crashed process is restarted from its last checkpoint, and it recovers its last state after having replayed all saved events. There is no need to coordinate the checkpoints of the different processes. No orphaned processes (*i.e.*,processes whose state depends on a non-deterministic event that cannot be reproduced during recovery) are created. The recover mechanism is more complex than with coordinated checkpoints as a process must obtain its past events and be able to replay them. Moreover the overhead induced during failure-free execution decreases the performance in reliable environments, such as clusters [Lemarinier et al. 2004]. Furthermore, it can lead to the domino effect [Randell 1975]: a process that rolls back and that needs a message to be replayed, asks another process to rollback. This process does, and asks another one to do so, etc. The whole execution can be restarted from the beginning because of cascading rollbacks and so the benefits of fault tolerance are lost.

Coordinated checkpointing has been introduced by Chandy and Lamport [1985]. This technique requires that at least one process sends a marker to notify the other ones to take a snapshot of their local state and then form a global checkpoint. The global state obtained from a coordinated checkpoint is coherent, allowing the system to recover from the last full completed checkpoint wave. It does not generate any orphan pro-

cesses nor domino effect, but all the compute nodes must rollback to a previous state in case of any failure. The recover process is straightforward, and a simple garbage collection reduces the size needed to store the checkpoints.

In blocking coordinated checkpointing protocols, the processes stop their execution to perform the checkpoint, save it on a reliable storage support (that can be distant), send an acknowledgment to the checkpoint initiator and wait for its commit. They continue the execution only when they have received this commit. The initiator sends the commit only when it has received all the acknowledgments from all the computing nodes to make sure that the global state that has been saved is fully completed. As reported in [Elnozahy et al. 1992], blocking checkpoints cause significant latency, and non-blocking checkpoints are more efficient.

Non-blocking coordinated checkpoints with distributed snapshots consists of taking checkpoints when a marker is received. This marker can be received from a centralized entity, that initiates the checkpoint wave, or from another compute node which has itself received the marker and transmits the checkpoint signal to the other nodes. This algorithm assumes that all the communication channels comply with the FIFO property. Therefore the computational processes do not have to wait for the other ones to finish their checkpoint, and then the delay induced by the checkpoint corresponds only to the local checkpointing.

Checkpointing can be performed at two abstraction levels: system-level or application-level. System-level checkpoints at remote storage cause large amounts of data to be sent through the network, but application-level checkpoints require modifications of the application code, and as such are not completely transparent to the programmer, in the sense that a code written for a non-fault-tolerant implementation of MPI requires some modifications to be executed on a fault-tolerant implementation of MPI using application-level checkpoints [Schulz et al. 2004] [Bronevetsky et al. 2003].

Communication-induced checkpoint protocols (CIC) perform uncoordinated checkpoints but avoid the domino effect [Hélary et al. 1999]. Unlike coordinated checkpoints, they do not require additional messages for a process to know when it has to perform a local checkpoint. The information about when a local checkpoint must be performed are piggybacked in the messages exchanged between the processes. Two kinds of checkpoints are defined: local and forced. Local checkpoints are decided by the local process, and forced ones are decided by the process according to the information piggybacked in the messages. The forced ones avoid the domino effect and ensure the progress of the recovery line, *i.e.*, the set of checkpoints of all the processes involved in a coherent global state. When a failure occurs, all the processes rollback to their most recently stored local checkpoint and then to the last recovery line. CIC is an interesting theoretical solution but it has been shown in [Alvisi et al. 1999], using NPB 2.3 benchmark suite [center 1997], that it is not relevant for typical cluster applications.

Several MPI libraries are fault tolerant [Gropp and Lusk 2002]. Coordinated checkpointing has been implemented in several MPI implementations at different levels of the application.

LAM/MPI [Burns et al. 1994] [Sankaran et al. 2003] implements the Chandy-Lamport algorithm for a system-initiated global checkpointing. When a checkpoint must be performed, the *mpirun* process receives a checkpoint request from a user or from the batch scheduler. It propagates the checkpoint request to each MPI process to initiate a checkpoint wave. Each MPI process then coordinates itself with all the others, flushing every communication channel, in order to reach a consistent global state. We used this method in our blocking Chandy-Lamport implementation. If a failure occurs, *mpirun* restarts all the processes from their most recently stored state. Finally, processes rebuild their communication channels with the other ones and resume their execution.

3 Protocols

In this paper, we compare two global checkpointing protocols. These are rollback recovery protocols. To perform this rollback recovery, they regularly take a snapshot of the local state of every process (of the system), such that when a failure occurs, all processes are rolled back to their most recently stored state. In order to ensure the global checkpoint coherence resulting from the collection of the different local states, these two protocols rely on the Chandy and Lamport algorithm [1985]. In this algorithm, one or more processes can initiate a checkpoint wave. When a process starts a checkpoint, it records its local state and sends a marker to all its neighbors. When a process receives a marker, if it has not started its checkpoint wave yet, it starts it. Every message a process receives after it has started its checkpoint wave and before having received the marker of the sender is recorded in the receiver image as the channel's state.

The first protocol we consider in this paper, called *Vcl*, is a direct implementation of the Chandy and Lamport algorithm for MPI computations. A MPI process consists in two Unix processes: a computation process (MPI) and a communication process (daemon). The communication process is used to store in-transit messages and to replay these messages when a restart is performed. Moreover, we added a process, the *checkpoint scheduler*, which is the only one that can initi-

ate a checkpoint wave. Furthermore, specific processes, called *checkpoint servers* are used to store the local images of all processes. Finally, we define a *dispatcher* for launching the different processes in the system, detecting failures and restarting the failed application.

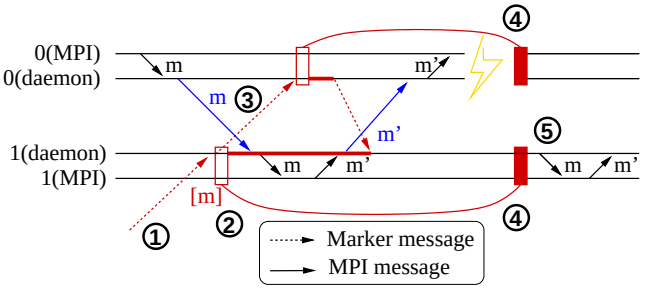


Figure 1: A Vcl Execution

The protocol works as shown in the Figure 1. The MPI process 1 initially receives the marker from the *checkpoint scheduler* (1), stores its local state (2) and sends a marker to every process (2). From this point, every message, like m in the figure, received after the local checkpoint and before having received the marker of the sender, is stored by the daemon process. When the MPI process 0 receives the marker, it starts its local checkpoint and sends a marker to every other process (3). The reception of this marker by 1 concludes the local checkpoint of 1. If a failure occurs, all processes restart from their last stored checkpoint (4) and the daemon process replays the delivery of the stored messages (5). Note that the message m' may be not sent again in the new execution.

The second protocol, which we call *Pcl*, is used in other implementations [Sankaran et al. 2003]. This protocol consists in synchronizing the different processes for emptying the communication layer. Thus, during the checkpoint wave, no message is still being exchanged, and there is no need to store the channel state in any way. When the system is restarted after a failure, every process reloads its last local image and reinitializes the communication layer for establishing a connection; then, the computation can continue.

The synchronization is performed by marker exchanges to flush all channels. In *Pcl*, a global checkpoint is made by following this sequence of actions. The MPI process of rank 0 regularly starts a new checkpoint wave, and it changes its state to `checkpointing` and sends markers to every other process (1). When a process receives the first marker, it changes its state to `checkpointing` and sends markers to every other process (2). After having sent a marker, a process does not send any other message through the same channel until it takes its checkpoint (blue segment 3). Such messages, still in the process memory, are automatically stored in the checkpoint. Similarly, after having

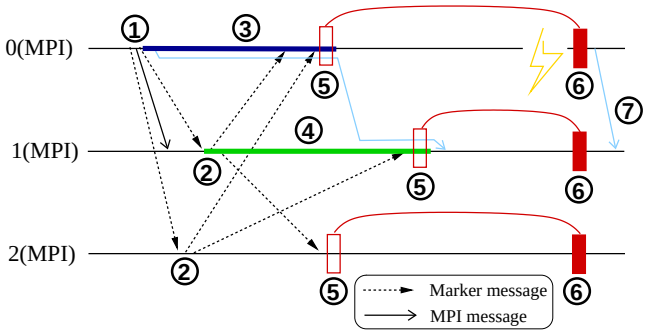


Figure 2: A Pcl Execution

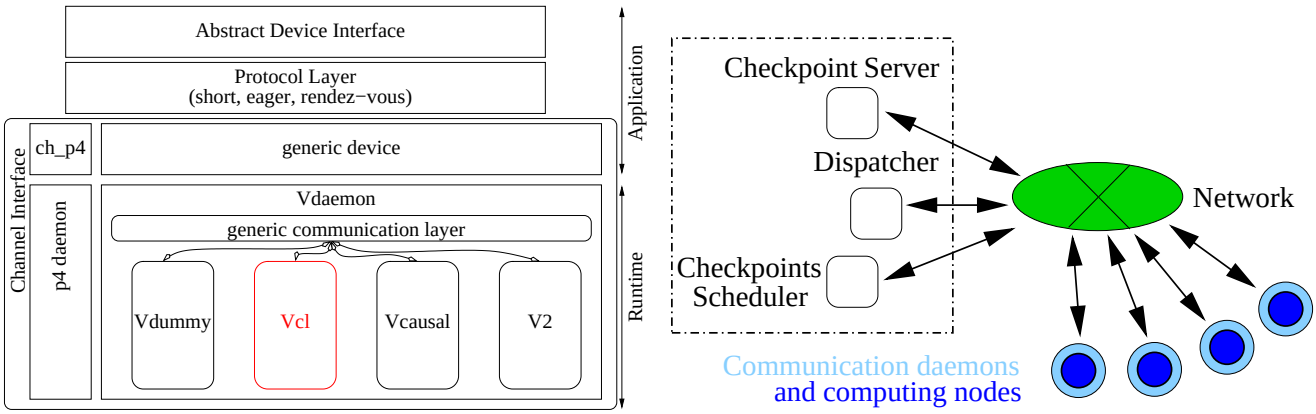
received a marker, a process does not receive any other message from the same channel: message receptions are delayed up until the end of the checkpoint of the process (green segment 4). When a process has received the marker of every other process, it checkpoints and sends the resulting image to the checkpoint server (5). After having taken its checkpoint, a process can send and receive any messages. When the images are completely stored, the process sends a message to rank 0 to warn it about the end of its checkpoint and continues its execution. Finally the rank 0 MPI process acknowledges the different checkpoint servers about the coherence of the wave after having received every confirmation of the end of the checkpoint from every process. If a failure occurs, all processes restart from their last stored checkpoint (6) and every message delayed in emission will be sent again after the restart (7).

4 Implementation Details

MPICH is a prominent project developed at the Argonne National Laboratory. It aims at providing a high-performance MPI library implementation. The first major revision, called MPICH, addresses the MPI-1 specification. The latest major revision, called MPICH2, extends the performance of the first one and addresses the new specifications of MPI-2. In this section we present the details of the integration of the global checkpointing mechanism inside these two major versions.

4.1 Non-Blocking Checkpointing Implementation Inside MPICH

A fundamental abstraction used by MPICH to implement the MPI standard is the notion of a *channel*. Such a channel implements the basic communication routines for a specific hardware or for new communication protocols. We developed a generic framework, called MPICH-V [Bosilca et al. 2002] [Lemarinier et al. 2004], to compare different fault tolerance protocols for MPI applications. This framework implements a channel for the



(a) MPICH-V Architecture

(b) Typical Deployment of MPICH-Vcl Environment

Figure 3: MPICH-V Architecture and Typical Deployment

MPICH 1.2.7 library, based on the ch_p4 default channel.

MPICH-V (see Figure 3) is composed of a set of runtime components and a channel called ch_v. This channel relies on a separation between the MPI application and the actual communication system. Communication daemons (Vdaemon) provide all communication routines between the different components involved in MPICH-V. The fault tolerance is performed by implementing hooks in relevant communication routines. This set of hooks is called a V-protocol. The two main V-protocols of interest in this paper are Vcl and Vdummy. Vdummy is a minimalist implementation of a non-fault-tolerant protocol using the MPICH-V architecture. Vdummy is used to measure the performance of the ch_v device and its communication daemon. Vcl implements the Chandy and Lamport algorithm (c.f. Section 3).

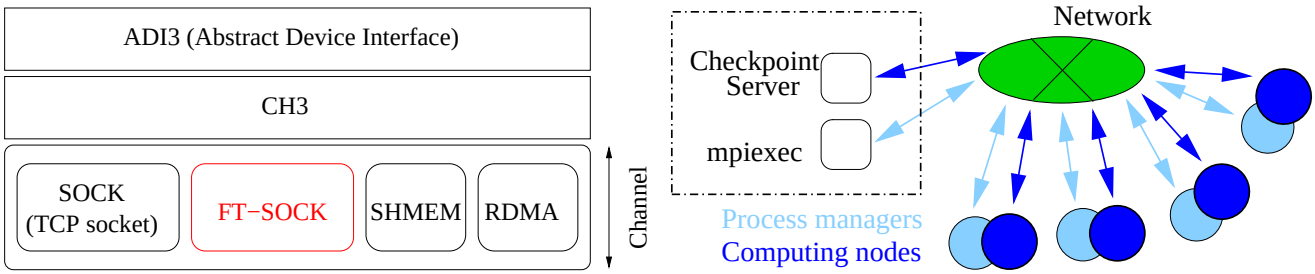
Daemon A daemon manages communication between nodes, namely sending, receiving, reordering and establishing connections. It opens one TCP socket per MPI process and one per server type (the dispatcher and a checkpoint server for the Vcl implementation). It is implemented as a single-threaded process that multiplexes communications through *select* calls. Moreover, to limit the number of system calls, all communications are packed using *iovec* techniques. The communication with the local MPI process is done using blocking send and receive on a Unix socket.

Dispatcher The dispatcher is responsible for starting the MPI application. It starts the different processes and servers first, then MPI processes, using *ssh* to launch remote processes. The dispatcher is also responsible for detecting failures and restarting nodes. A failure is assumed after any unexpected socket closure.

Failure detection relies on the OS TCP keep-alive parameters. Typical Linux configurations define a failure detection a 9 consecutive losses of keep-alive probes, where keep-alive probes are expected every 75 seconds. These settings can be changed through the `tcp_keepalive_probes` and `tcp_keepalive_intvl` system parameters to provide more reactivity to hard system crashes. In this work, we emulated failures by killing the task, not the operating system, so failure detection was immediate, and the TCP connection was broken as soon as the task was killed by the operating system.

Checkpoint Server and Checkpoint Mechanism The two implementations use the same abstract checkpointing mechanism. This mechanism provides a unified API to address three system-level task checkpointing libraries, namely Condor Standalone Checkpointing Library [Litzkow et al. 1997], libckpt [Zandy 2005] and the Berkeley Linux Checkpoint/Restart [J. Duell 2003; Sankaran et al. 2003]. All these libraries allow the user to take a Unix process image in order to store it on a disk and to restart this process on the same architecture. By default, BLCR, which is the most up-to-date library, is used.

The checkpoint servers are responsible for collecting local checkpoints of all MPI processes. When a MPI process starts a checkpoint, it duplicates its state by calling the *fork* system call. The forked process calls the checkpoint library to create the checkpoint file while the initial MPI process can continue the computation. The daemon associated with the MPI process connects to the checkpoint server that first creates a new process responsible for managing the checkpoint of this MPI process. Then three new connections are established (data, messages and control) between the daemon and the server. The clone of the MPI process writes its local



(a) MPICH2-Pcl Architecture

(b) Typical Deployment of MPICH2-Pcl Environment

Figure 4: MPICH2-Pcl Architecture and Typical Deployment

checkpoint to a file, and the daemon pipelines the reading and the sending of this file to the checkpoint server using the data connection. When the checkpoint file has been completely sent, the clone of the MPI process terminates and the daemon closes the data connection; then it sends the total file size using the control connection. Every message to be logged according to the Chandy and Lamport algorithm is temporarily stored in the volatile memory of the daemon in order to be sent to the checkpoint server in the same way using the message connection. Using this technique, the whole computation is never interrupted during a checkpoint phase.

When a global checkpoint is complete it is not necessary to still store the past global checkpoints. Thus, checkpoint servers only store one complete global checkpoint at a time using two files alternatively to store the current global checkpoint and the last complete one.

If a failure occurs, all MPI processes restart from the local checkpoint stored on the disk if it exists; otherwise they obtain it from the checkpoint server.

Checkpoint Scheduler The checkpoint scheduler manages the different checkpoint waves. It regularly sends markers to every MPI process. The checkpoint frequency is a parameter defined by the user. It then waits for an acknowledgment of the end of the checkpoint from every MPI process before asserting the end of the global checkpoint to the checkpoint servers. The checkpoint scheduler starts a new checkpoint wave only after the end of the previous one.

4.2 Blocking Checkpointing Implementation Inside MPICH2

MPICH2 is a new implementation of the MPI standard which extends results obtained in MPICH and addresses the issue of the MPI2's new functionalities. MPICH2 is structured in three layers: 1) the abstract device interface (ADI3) which links the MPI standard to an extended set of high-level communication routines, 2) chameleon 3 (CH3) which abstracts the ADI3 routines to an API composed of a few (ten to twenty) communication routines and 3) channels which implements this CH3 API depending on the specific network hardware or communication protocol.

We introduce in this paper a new implementation of a blocking checkpointing mechanism for fault tolerance inside MPICH2 called MPICH2-Pcl (see Figure 4). This implementation consists of a new channel, called *ft-sock*, based on the TCP *sock* channel, and two components, *i.e.*, a checkpoint server and a specific dispatcher.

Ft-sock Channel The ft-sock channel is a derivation of the existing *sock* implementation. It consists of a basic set of communication routines using a *poll* mechanism to multiplex I/O and *iovec* to reduce the number of system calls. The core of the communication system is based on sequences of request to send and request to receive for each MPI peer. Sending or receiving messages consists of posting such requests to the sock channel.

To implement the blocking checkpointing mechanism, the main modifications involve adding a hook in the request posting function for verifying and delaying these posts if a checkpoint wave is currently active. The exchange of markers used in the protocol (c.f. Section 3) is done by using the communication primitive defined in *sock* and adding a new type of packet.

By contrast to MPICH-Vcl, there is not a specific checkpoint scheduler server to start checkpoint waves. This role is now dedicated to the MPI process of rank 0.

Checkpoint Implementation Details The same checkpoint server as in MPICH-V is used to store MPICH2-Pcl checkpoint images. As explained in Section 3, a process starts taking its image only after it receives and sends all markers. At this time, the process forks to create its checkpoint file in the same way as in MPICH-Vcl while the main process releases the delayed requests and continues the MPI computation. When the clone ends the checkpoint, the SIGCHLD signal is delivered to the main process that sends a message to the MPI

process of rank 0 such that a new checkpoint wave can be scheduled.

Runtime: MPD and FTPM MPICH2 introduces a new process management environment called MPD, which runs a persistent daemon on every node of the system for launching MPI jobs. All these daemons are connected in a ring topology. This avoids the use of sequential *ssh* commands to start a job. When a job is launched on n nodes, the n MPD fork to create process managers (PMs). Then the process managers fork to execute n MPI processes. The different MPI processes are not connected together at the start of the execution. Two MPI processes connect themselves only from the first communication request between them. The role of process manager is to provide information about the different nodes' locations. In the current MPICH2 implementation, the MPD is known to be fault tolerant but the process manager is not. When a failure occurs, all the PMs and the MPI processes of the job are killed.

Our implementations of fault tolerant protocols include checkpoint servers. When a failure occurs, all the non-failed processes must be killed but not the checkpoint servers. This could be implemented by the MPD architecture using two daemons: one for the computing nodes and one for the checkpoint servers. However, computing nodes need to locate the checkpoint servers. The MPD implementation does not provide a means of getting information from other jobs. We propose to add a new concept in the MPD architecture; namely a process group. A job would consist of one or more process groups, each process being managed by a process manager.

Rather than modifying the process manager, we implement a simpler environment, which we call a fault tolerant process manager (FTPM), to start, manage, detect failures and restart applications. The FTPM is composed of an *mpiexec* and PMs. In FTPM, there is no MPD daemon, and we use a modified version of PMs. In particular, checkpoint servers are now launched through them. We also modify the *machinefile* format in order to add the specification of the mapping between machines used as computing nodes and machines used as checkpoint servers.

At run time, *mpiexec* determines which computing nodes are used as MPI processes and launches the corresponding checkpoint servers and then the processes through the PMs. Process and checkpoint server spawning is done using a *ssh* command. To improve the execution time, these spawns are done in parallel. To avoid throttling a node running *mpiexec*, the number of concurrent ssh connections is bounded by a parameter. For the remainder of the execution, *mpiexec* has to monitor the MPI processes and to maintain a distributed database. Node monitoring is done in the same way as in MPICH.

Each MPI process publishes its location to the others by associating in the distributed database its rank to a *business card* (composed as in MPD of the process IP address, hostname and port to connect). The database is also used to store the greatest successful checkpoint wave number and to locate which checkpoint server holds which local checkpoint. Since at restart time MPI processes may be assigned to spare nodes, their last local checkpoint may be not located on the local disk or on the local server associated with the running machine.

5 Performance Measurements

In this section we present the performance measurements of the two implementations introduced in this paper. We conducted the experiments on three classical platforms of high-performance computing; namely clusters of workstations with GigaEthernet network, clusters connected with high-performance communication networks and computational grids. We conducted all the experiments on the experimental Grid5000 platform or some of its components.

5.1 Grid5000

Grid5000 [F. Cappello *et al.* 2005] is a physical platform featuring 13 clusters, each with 20 to 216 PCs, connected by the Renater French Education and Research Network. Grid5000 is a computer science project dedicated to the study of grids, and is funded by the French government through the ACI Grid initiative.

At the time of writing article, it consisted of 964 computers featuring four architectures (Itanium, Xeon, G5 and Opteron) organised as 13 clusters over 9 cities in France.

For the three platforms previously mentioned (cluster, high speed network and grid), we used only homogeneous clusters with AMD Opteron 248 (2.2 GHz/1MB L2 cache) dual-processors running at 2GHz. This included 6 of the 13 clusters of Grid5000: the 48-node cluster at Bordeaux, the 53-node cluster at Lille, the 216-node cluster at Orsay, a 64-node cluster at Rennes, the 105-node cluster at Sophia and the 58-node cluster at Toulouse. Moreover, each node featured 20GB of swap and SATA hard drives. All the cluster experiments were run on the 216-node cluster at Orsay. Nodes were interconnected by a Gigabit Ethernet switch. Myrinet experiments were run on the 48-node cluster at Bordeaux. Each node was similar to the nodes at Orsay, interconnected by a Myrinet2000 M3-E64 with 48 ports and PCIXD (Lanai XP) network interface cards.

One major feature of the Grid5000 project is the ability of the user to boot her own environment (includ-

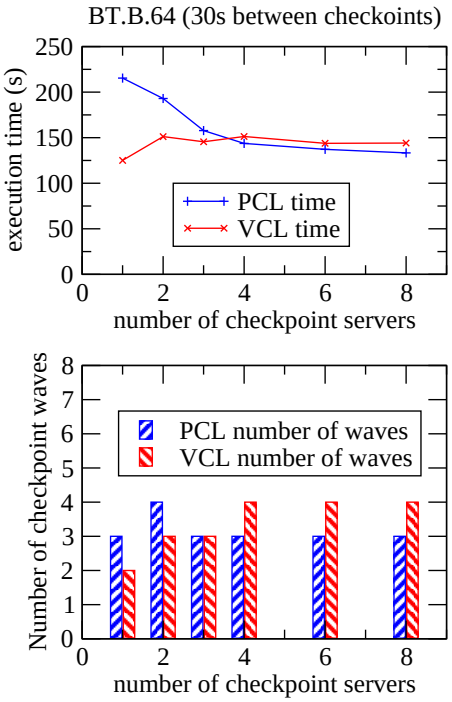


Figure 5: Impact of the Number of Checkpoint Servers on BT Class B for 64 Processes with a Given Period of Time Between Checkpoints

ing the operating system, distribution, libraries...) on all the computing nodes booked for her job. We used this feature to run all our measurements in an homogeneous environment including the Berkeley Linux Checkpoint/Restart library. All the nodes were booted under Linux 2.6.13.5. The tests and benchmarks are compiled with GCC-4.0.3 (with flag -03). All tests were run in dedicated mode, and each measurement was repeated 5 times, and we present the mean time.

Most of the experiments were done using NAS parallel benchmarks (NPB-2.3) [Bailey et al. 1995] written by the NASA NAS research center to test high-performance parallel machines. These benchmarks exhibit classical communication patterns, which are significant for the performance evaluation of fault tolerant implementations. Checkpoints were triggered by timeouts. In the following experiments, we used very small values for these timeouts (tens of seconds) in order to emphasize the impact of checkpoint frequency while maintaining reasonable experimental times.

5.2 GigaEthernet Clusters

Figure 5 presents first a study on the scalability of checkpoint servers. We executed the BT benchmark of class B with 64 processors (over 32 dual-processor nodes), and set a period of time between checkpoints of 30 seconds. Thus, according to the implementation, after having fully transferred a checkpoint image to a checkpoint server, the system waits for 30 seconds before beginning a new checkpoint wave. The figure consists of two parts. In the upper part, we measure the execution time for various ratios of the number of checkpoint server and number of computing nodes: from 1 server for the 64 computing nodes to 1 server for 8 computing nodes. In the lower part, we present the number of checkpoint waves completed by the system during the corresponding executions. We ran this experiment for the two implementations (Pcl, the blocking implementation in MPICH2 and Vcl, the non-blocking implementation in MPICH-1.2).

The completion time of Vcl remains almost constant whereas the completion time of Pcl decreases when the number of checkpoint servers increases. When the number of checkpoint servers increases, the duration of checkpoint image transfer decreases. For Pcl, this is seen clearly in the first part of the curve as completion time decreases. In Pcl, before a process takes a checkpoint image, it has blocked all its communication. When it starts its checkpoint image transfer, it simultaneously continues these communications. So, these communications and the checkpoint transfer compete for the network bandwidth. When the bandwidth contention decreases (e.g. when the number of checkpoint servers increases), overall performance increases. The timeout for the next checkpoint wave is set as soon as every process has transferred its image. So, increasing the number of checkpoint servers decreases the time between two checkpoint waves. However, as seen in the bottom half of the figure, the overall completion time decreases enough to prevent triggering an additional checkpoint wave.

On the contrary, for Vcl, most of the time saved for transferring the checkpoint image is used to increase the number of checkpoint waves. Vcl does not block the communications for the checkpoint, and less communications compete with the checkpoint transfers. So, it has a lesser impact on the MPI communication and decreasing the time to take the checkpoint still decreases the period of time between two checkpoint waves. This introduces more checkpoint waves without altering the near-optimal completion time. The small difference between Pcl and Vcl for 8 checkpoint servers illustrates the better performance of MPICH2 as compared to MPICH-Vcl.

The four graphs of Figure 6 present the scalability of fault tolerance with respect to the number of processes for given times between checkpoints. The BT class B benchmark is run at varying sizes, for different values of time between checkpoints, and the completion time is measured for the two implementations and compared to a checkpoint-free execution. All executions use the

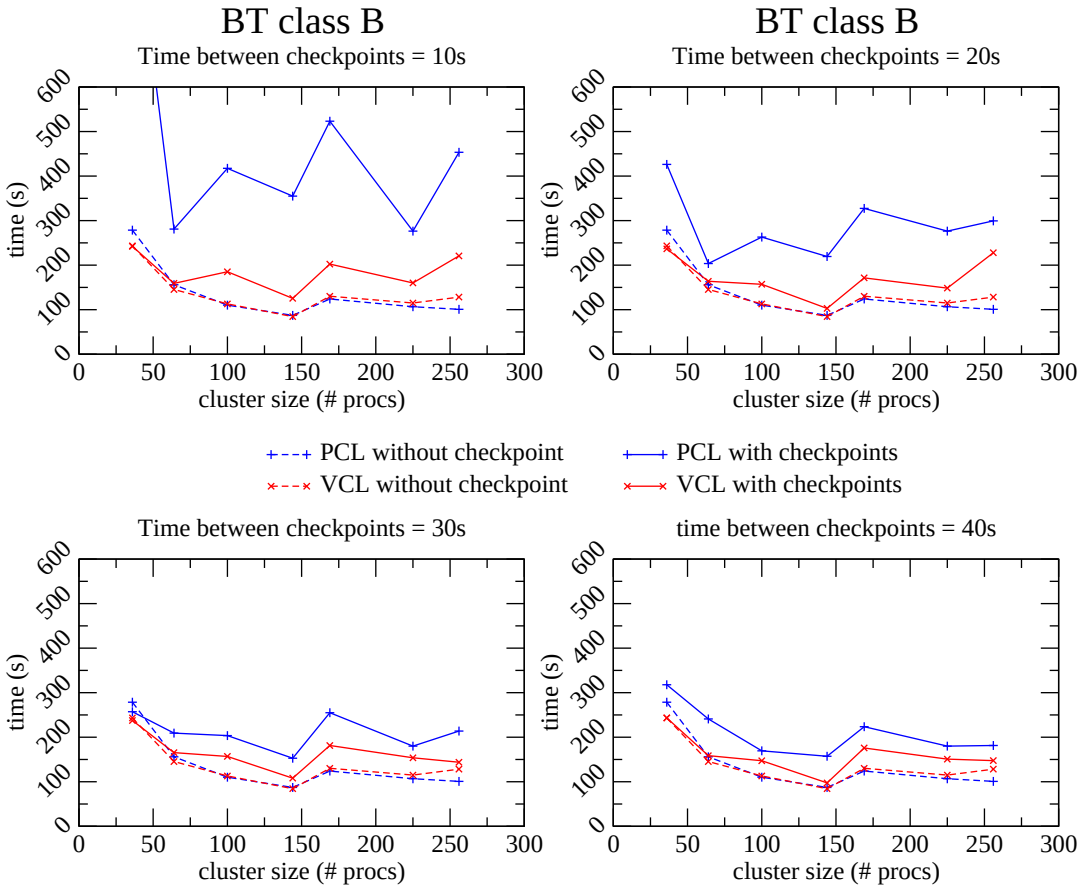


Figure 6: Execution Time as Function of the Number of Processes for Four Checkpoints' Frequency

same number of checkpoint servers (9).

Without checkpoint, the two implementations behave similarly at all the sizes. The MPICH2 implementation is slightly more efficient for 256 processors. For all implementations, there is an observable slowdown at 169 processors. Only 150 computers were available for this test, and we used single process deployments for up to 144 computing nodes, and bi-processor deployments (limiting the number of computers to 128) for experiments with more than 160 computing nodes. The gap is due to sharing the network interface card between the two processors.

Without considering the measurement at 10s between checkpoints, where the communications are heavily perturbed by the blocking protocol, one can see that the number of nodes has no measurable impact on the overhead of checkpointing, whatever the protocol used. The blocking protocol has a high-performance degradation when subject to high checkpointing frequencies. It spends most of the time synchronizing to make a global checkpoint, and MPI communications happens in bursts. The Vcl implementation does not introduce the same synchronizations and is always closer to the executions without checkpointing.

When the time between checkpoints increases, this gap reduces to a constant overhead for the two checkpointing implementations.

5.3 High-Performance Communication Clusters

The first row of graphs in Figure 7 shows the results of the BT class B benchmark on 64 processors, and in the second row the CG class C benchmark on 64 processors; all benchmarks ran over a 36-node cluster interconnected by a myri2000 network. Four nodes were used as checkpoint servers, and the computing nodes were distributed equally among the checkpoint servers. The experiments were conducted with the MX-2G 1.1.1 driver from myricom, enabling Ethernet over myri2000.

The left column of graphs presents the completion time as a function of time between two checkpoints, and the number of checkpoint waves for each run, while the right column presents the completion time of the same experiments as a function of the number of checkpoint waves.

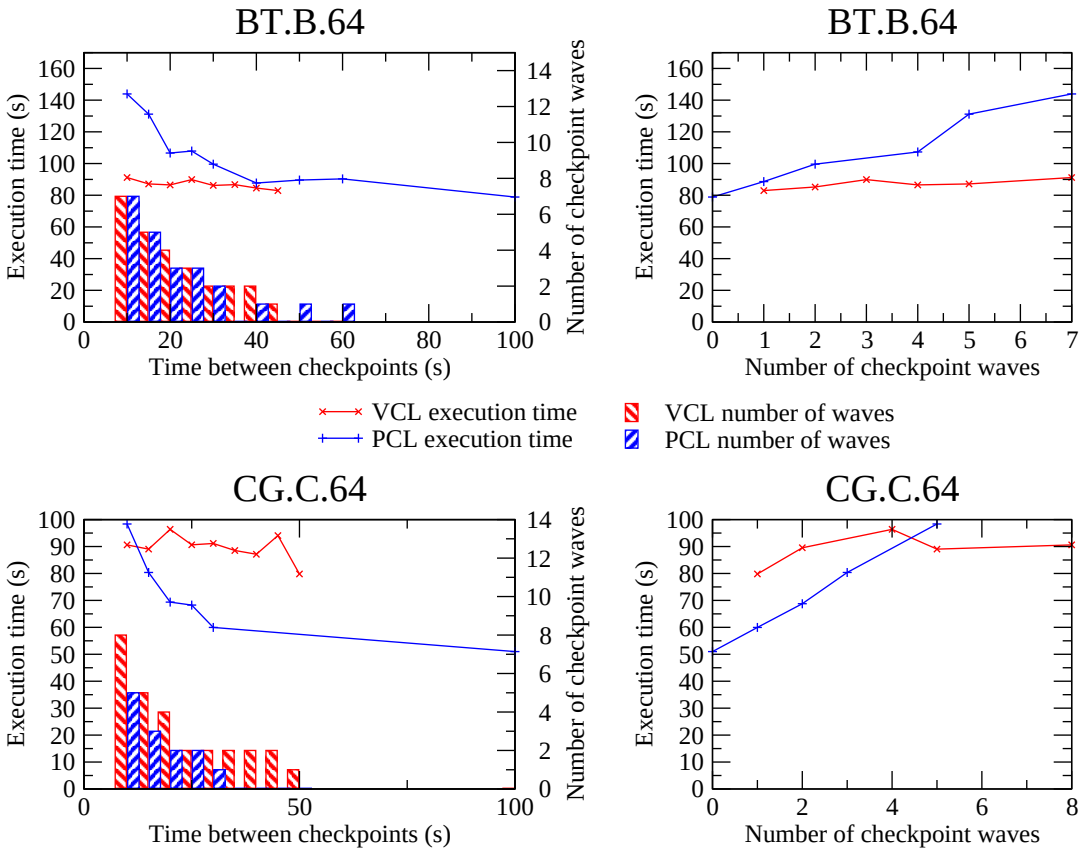


Figure 7: Impact of Checkpoint Frequency on BT.B.64 and CG.C.64 for Myricom Network

The Pcl execution time follows the number of checkpoint waves, and the right column of graphs demonstrate that the completion time is roughly linear with the number of checkpoints. This is easily explained by the synchronizations introduced by the blocking protocol. As explained in the cluster experiments, the number of checkpoint waves does not directly influence the performance of the Vcl implementation.

CG is a benchmark with a lot of small communications, and therefore a latency-bound benchmark. Vcl is implemented with a communication daemon, and each message has to pass through two UNIX sockets and the Ethernet emulation of the myri2000 card, implying unnecessary copies and a high latency overhead. This is why Pcl performs much better than Vcl for this benchmark.

BT is a computation-bound benchmark, with a relatively small number of long communications. So the Vcl implementation does not suffer from the overhead of its messages copies, and the overhead of the synchronization of Pcl results in better performance for Vcl with high checkpoint wave frequency.

5.4 Large Scale Experiments

The large-scale experiments are conducted on Grid5000. Its clusters are interconnected with internet links. In order to evaluate the results of the benchmarks, we first measure the raw performance of this platform using the NetPIPE [Snell et al. 1996] utility. This is a ping pong test for several message sizes and small perturbations of these sizes.

Figure 8 presents the bandwidth and latency measurements between each pair of clusters. The network is up to 20 times faster between two nodes of the same cluster than between two nodes of two clusters. Moreover, the latency is up to two orders of magnitude greater between clusters than between nodes.

We present here results only for the Pcl implementation. The Vcl implementation was not designed for this scale, because it uses the `select` system call to multiplex its communication channels, and this tool is not scalable beyond a thousand sockets (in Linux, a file descriptor set has a size of at most 1024/8 bytes). Each node of the Vcl implementation opens up to 3 sockets with the dispatcher (one for alive messages and availability, two for standard input and output), and this precludes tests with more than 300 processes.

	Bordeaux		Orsay		Rennes		Sophia		Lille		Toulouse	
	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)
Toulouse	190	1.5	59.81	5.51	34.79	9.92	83.7	3.74	26.97	13.04	930.4	0.04
Lille	30.23	11.62	132.55	2.25	56.84	5.83	37.41	9.22	938	0.04		
Sophia	68.1	5.2	42.4	8.6	40.2	9.1	940.5	0.04				
Rennes	110.1	4.0	95.4	4.7	940.4	0.04						
Orsay	108.1	4.1	930.4	0.06								
Bordeaux	940.2	0.04										

Figure 8: Raw Performance Evaluation of the Experimental Grid

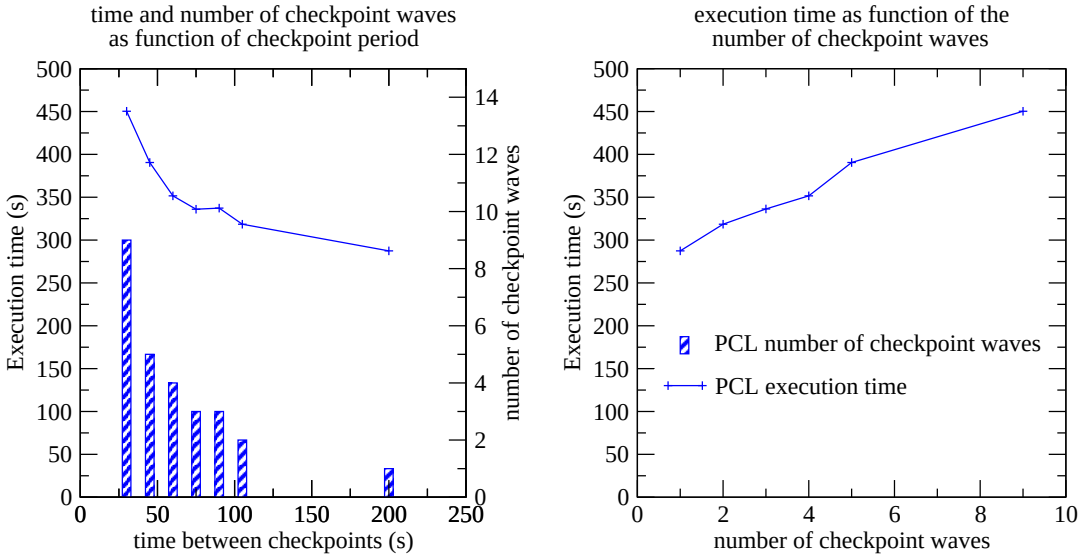


Figure 10: Impact of Checkpoint Frequency on Blocking Checkpointing at Large Scale (400 processes)

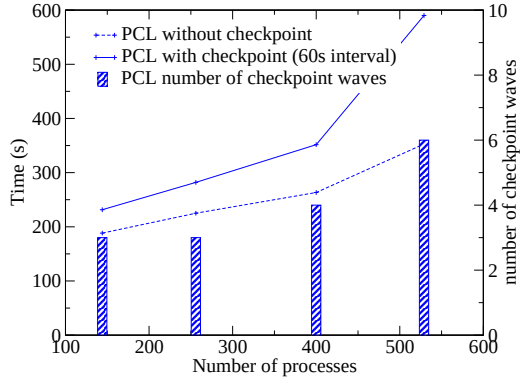


Figure 9: Impact of Large Scale on Blocking Checkpointing

By contrast, Pcl was designed to scale to large platforms, and we conducted experiments with up to 1024 processes. Due to insufficient host availability in Grid5000, we cannot be certain of its scalability at the moment, but we present here results up to 529 processors.

Figure 9 presents the measurement of the BT class B benchmark with a varying number of processes distributed over the grid. Each node used a local machine (among 4) as its checkpoint server. The figure presents three results: the completion time without checkpointing, the completion time with a checkpoint wave every 60s and the number of checkpoint waves for each run.

Although BT.B is not scalable on such a grid deployment, we consider it a stress test for the fault tolerant protocol, since it introduces complex communication schemes among all nodes.

The execution without checkpointing presents a slowdown for 529 processes due to the heterogeneity of the grid, and the use of remote processors at this scale. This leads to a longer execution time, in which the checkpointing execution has more time to make up to 6 checkpoint waves. Since the completion time is proportional to the number of checkpoint waves, this increases the completion time of the execution with checkpoints every 60s.

This is confirmed by the Figure 10, which presents on its left side the completion time and number of check-

point waves according to the time between checkpoints and on its right side the completion time as function of the number of checkpoint waves for the BT class B with 400 processes benchmark. The benchmark is run in similar conditions as the previous experiment.

Even in grid deployments, the execution time is still linear to the number of checkpoint waves. This number itself is proportional to the frequency of checkpoint, that is the inverse of the time between checkpoints.

6 Conclusion

In this paper, we present Pcl, a new implementation of a blocking, coordinated checkpointing, and fault tolerant protocol inside MPICH2. We evaluate its performance on three typical high performance architectures: clusters of workstations, high speed network clusters, and computational grids. We compare the performance to Vcl, an implementation of a non-blocking, coordinated checkpointing protocol.

A blocking, coordinated checkpointing protocol requires flushing communication channels before taking the state of a process in order to ensure the coherency of the view built. It introduces synchronization in the distributed system while communications are frozen. However, since it does not require copies of incoming or outgoing messages, it is simpler to implement in an existing high-performance communication driver.

A non-blocking, coordinated checkpointing protocol consists of saving the state of the communication channels during the checkpoint without interrupting the computation. It requires logging in-transit messages and replaying them at restart, which implies coordination with the match-making engine and queue mechanisms.

The experimental study demonstrated that for high-speed networks, the blocking implementation gives the best performance for sensible checkpoint frequency. On clusters of workstations and computational grids, the high cost of network synchronization to produce the checkpointing wave of the blocking protocol introduces a high overhead that does not appear with the non-blocking implementation.

An experimental study on a cluster demonstrated that the checkpoint frequency has more significant impact on the performance than the number of nodes involved in a checkpoint synchronization for both non-blocking and blocking protocols. We are conducting a larger study to evaluate this result on computational grids.

Thanks

Many thanks to Darius Buntinas from Argonne National Laboratory for his sage advice on MPICH2, and to the MPICH2 team at Argonne National Laboratory for many fruitful communications.

The authors would like to thank Derrick Kondo for his comments and suggestions.

This work was done thanks to the Grid5000 project founded by the ACI Grid incentive action from the French research ministry.

References

Alvisi, L., Elnozahy, E., Rao, S., Husain, S. A., and Mel, A. D. 1999. An analysis of communication induced checkpointing. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, IEEE CS Press.

Bailey, D., Harris, T., Saphir, W., Wijngaart, R. V. D., Woo, A., and Yarrow, M. 1995. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center.

Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fédak, G., Germain, C., Hérault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Néri, V., and Selikhov, A. 2002. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *High Performance Networking and Computing (SC2002)*, IEEE/ACM, Baltimore USA.

Bronevetsky, G., Marques, D., Pingali, K., and Stodghill, P. 2003. Automated application-level checkpointing of MPI programs. In *PPOPP*, ACM, 84–94.

Burns, G., Daoud, R., and Vaigl, J. 1994. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, 379–386.

center, N. A. R., 1997. Nas parallel benchmarks. http://science.nas.nasa.gov/Software/NPB/.

Chandy, K. M., and Lamport, L. 1985. Distributed snapshots : Determining global states of distributed systems. In *Transactions on Computer Systems*, ACM, vol. 3(1), 63–75.

Elnozahy, E. N., Johnson, D. B., and Zwaenepoel, W. 1992. The performance of consistent checkpointing. In *Symposium on Reliable Distributed Systems*, 39–47.

Elnozahy, M., Alvisi, L., Wang, Y. M., and Johnson, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR) 34*, 3 (september), 375 – 408.

F. Cappello *et al.* 2005. Grid'5000: a large scale, reconfigurable, controlable and monitorable grid platform. In *proceedings of IEEE/ACM Grid'2005 workshop.*

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 97–104.

Gropp, W., and Lusk, E. 2002. Fault tolerance in MPI programs. *special issue of the Journal High Performance Computing Applications (IJHPCA).*

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22*, 6 (September), 789–828.

Hélary, J.-M., Mostefaoui, A., and Raynal, M. 1999. Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems 10*, 9, 865–877.

J. Duell, P. Hargrove, E. R. 2003. The design and implementation of berkeley lab's linux checkpoint/restart. Tech. Rep. publication LBNL-54941, Berkeley Lab.

Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., and Cappello, F. 2004. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *IEEE International Conference on Cluster Computing (Cluster 2004)*, IEEE CS Press.

Litzkow, M., Tannenbaum, T., Basney, J., and Livny, M. 1997. Checkpoint and migration of UNIX processes in the condor distributed processing system. Tech. Rep. 1346, University of Wisconsin-Madison.

Randell, B. 1975. System structure for software fault tolerance. *IEEE Transactions on Software Engineering SE-1*, 2, 220–232.

Sankaran, S., Squyres, J. M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. 2003. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium.*

Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K., and Stodghill, P. 2004. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. IEEE Computer Society, 38.

Snell, Q., Mikler, A., and Gustafson, J. 1996. Netpipe: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems.*

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. 1996. *MPI: The Complete Reference.* The MIT Press.

Strom, E., and Yemini, S. 1985. Optimistic recovery in distributed systems. In *Transactions on Computer Systems*, ACM, vol. 3(3), 204–226.

Zandy, V., 2005. libckpt http://www.cs.wisc.edu/~zandy/ckpt/.