Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 24 (2008) 73–84
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols

Darius Buntinas[d], Camille Coti[b], Thomas Herault[a,c,b,*], Pierre Lemarinier[b,a,c,1], Laurence Pilard[a,c,b], Ala Rezmerita[b,a,c], Eric Rodriguez[b,a,c], Franck Cappello[b,a,c]

[a] *Univ Paris-Sud, UMR8623, LRI, Orsay, F-91405, France*
[b] *INRIA Futurs, Orsay, F-91893, France*
[c] *CNRS, Orsay, F-91405, France*
[d] *Mathematics and Computer Science Division, Argonne National Laboratory, United States*

Received 1 February 2007; accepted 8 February 2007
Available online 16 February 2007

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 MPIs 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 scalabilities remain unknown. The contribution of this paper is to provide the first comparison between these two approaches and a study of their scalabilities. We have implemented the two approaches within the MPICH environments and evaluate their performance using the NAS parallel benchmarks.

Keywords: Fault tolerant MPI; Performance evaluation; Coordinated checkpointing; Rollback/Recovery; Large-scale

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 inside the whole system also increases [1]. 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, which are often administrated with 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 applications 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 kinds 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

* Corresponding author.
E-mail address: lemarini@lri.fr (P. Lemarinier).
[1] Tel.: +33 169154222.

0167-739X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2007.02.002

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 [2] and most deployed implementations, (MPICH [3]) follows the *fail stop* semantic (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 due to only one failure, a fault tolerant MPI implementation is essential. The typical fault tolerant technique implemented in an MPI library is coordinated checkpointing [4,5]. 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 *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, blocking and non-blocking, and evaluate their respective impacts on large-scale applications. We detail the implementation of the blocking protocol inside MPICH2, and compare it with our previous non-blocking implementation MPICH-Vcl [6] 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, and then the blocking and non-blocking solutions. Section 4 presents the 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 two main open-source projects: MPICH [3] and OpenMPI [7].

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 to the programmer/user). In this paper we focus on the last strategy, one 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.

Message-logging is based on the *Piecewise Deterministic Assumption*, according to which execution of a process is a sequence of deterministic events separated by non-deterministic ones (generally the reception events) [8]. As a consequence, replaying the same sequence of non-deterministic events at the same moment makes possible the recovery of the state preceding a failure. Therefore, there is no need to coordinate the checkpoints of the different processes, since each one can checkpoint its state independently from the other ones. The recovery 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 [6].

Coordinated checkpointing has been introduced by Chandy and Lamport [9]. This technique requires that at least one process send a marker to notify the other ones to take a snapshot of their local states 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 processes, nor domino effects, but all the compute nodes must rollback to a previous state in case of any failure. The recovery process is straightforward, and 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 remote), 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 [10], blocking checkpoints cause significant latency, and non-blocking checkpoints are more efficient.

Non-blocking coordinated checkpoints with distributed snapshots consist 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.

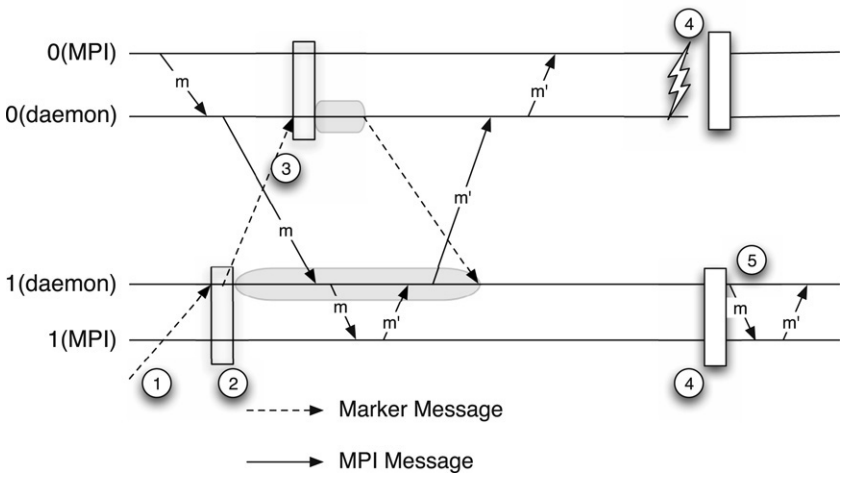


Fig. 1. A Vcl execution.

Several MPI libraries are fault tolerant [11]. Coordinated checkpointing has been implemented in several MPI implementations at different levels of the application.

LAM/MPI [5,12] implements the Chandy–Lamport algorithm for a system-initiated global checkpointing. The synchronization is performed at the network level by the communication protocol.

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 states 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 [9]. 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 neighbours. 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. An MPI process consists of 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 initiate 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.

The protocol works as shown in Fig. 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 (3). 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 [12]. This protocol consists in synchronizing the different processes for emptying the communication layer. Thus, during the checkpoint wave, no messages are being exchanged, so 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.

Fig. 2 illustrates this protocol. 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 changes its state to `checkpointing`, then 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 (segment 3). Such messages, still in the process memory, are automatically stored in the checkpoint. Similarly, after having 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 notify it about the end

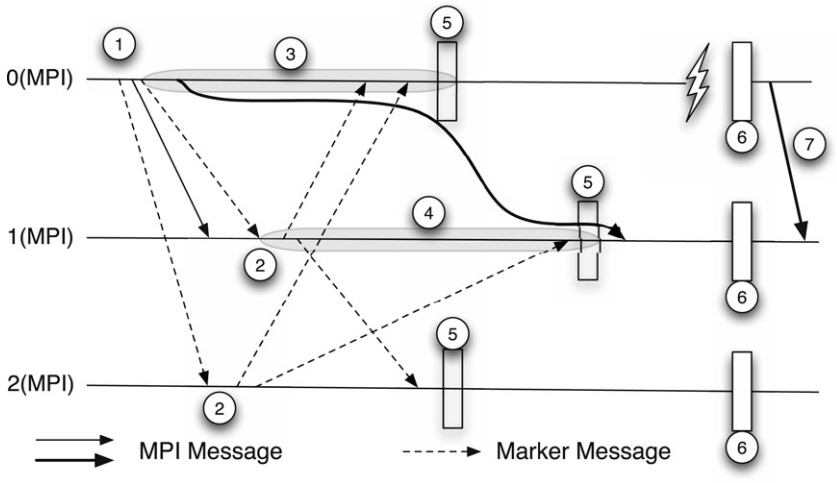


Fig. 2. A Pcl execution.

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 *device*. Such a device implements the basic communication routines for specific hardware or for new communication protocols. We developed a generic framework, called MPICH-V [4,6], to compare different fault tolerance protocols for MPI applications. This framework implements a device for the MPICH 1.2.7 library, based on the ch_p4 default device.

MPICH-V (see Fig. 3) is composed of a set of runtime components and a device called ch_v. This device 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 (cf. Section 3).

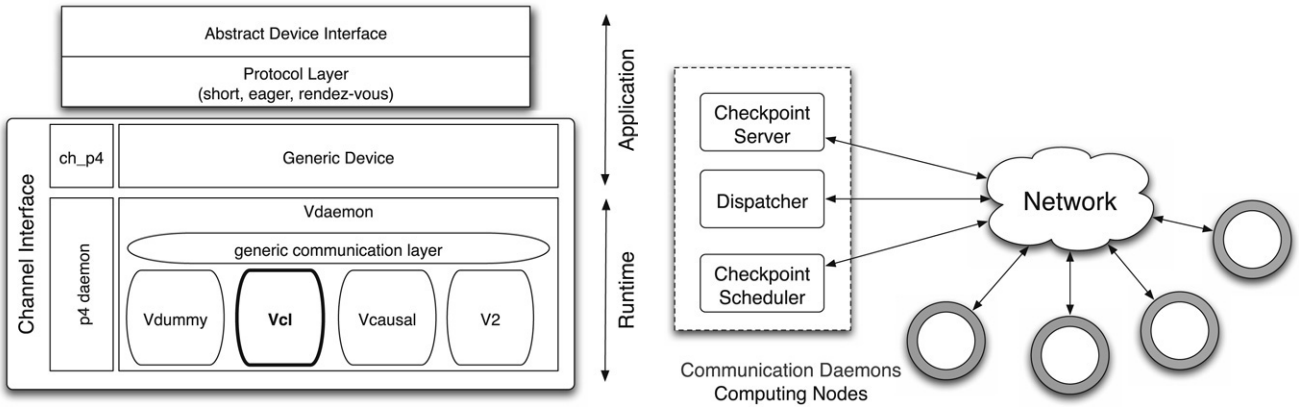
Daemon. A daemon manages the communication between nodes, namely sending, receiving, reordering and establishing connections. It opens one TCP socket per MPI process and one per server type (a dispatcher and a checkpoint server for the Vcl implementation). It is implemented as a single-threaded process that multiplexes communications through *select* calls. 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. Firstly it starts the servers, then the MPI processes using *ssh* command. The dispatcher is also responsible for detecting failures and restarting nodes. A failure is assumed after any unexpected socket closure.

The dispatcher is responsible for starting the MPI application. It starts the different processes and servers first, then the 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. Then, the dispatcher signals all the other processes to exit, removes the non-responding machines from the list of available processors and launches the restarting application in the rest of the processors. This may lead to overloading of some processors, or the running of more than one process on a single processor. In order to address this issue, one has to overbook processors to have available spare nodes.

Failure detection relies on the OS TCP keep-alive parameters. 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.



(a) MPICH-V architecture.

(b) Typical deployment of MPICH-Vcl environment.

Fig. 3. MPICH-V architecture and typical deployment.

This mechanism provides a unified API to address three system-level task checkpointing libraries, namely Condor Standalone Checkpointing Library [13], libckpt [14] and the Berkeley Linux Checkpoint/Restart [15,12]. 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 technique used (system-level checkpointing) saves the whole process image, i.e. its memory map, kernel state and process registers. So the size of the checkpoint images is directly proportional to the memory allocated, and few optimizations can be used to reduce this size.

The checkpoint servers are responsible for collecting local checkpoints of all MPI processes. When an 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 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.

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 goal of checkpoint scheduler is to manage the checkpoint waves. It regularly (parameter defined by the user) sends markers to every MPI process. Before asserting the end of the global checkpoint to the checkpoint servers, the scheduler waits for an acknowledgment of the end of the checkpoint from every MPI process.

4.2. Blocking checkpointing implementation inside MPICH2

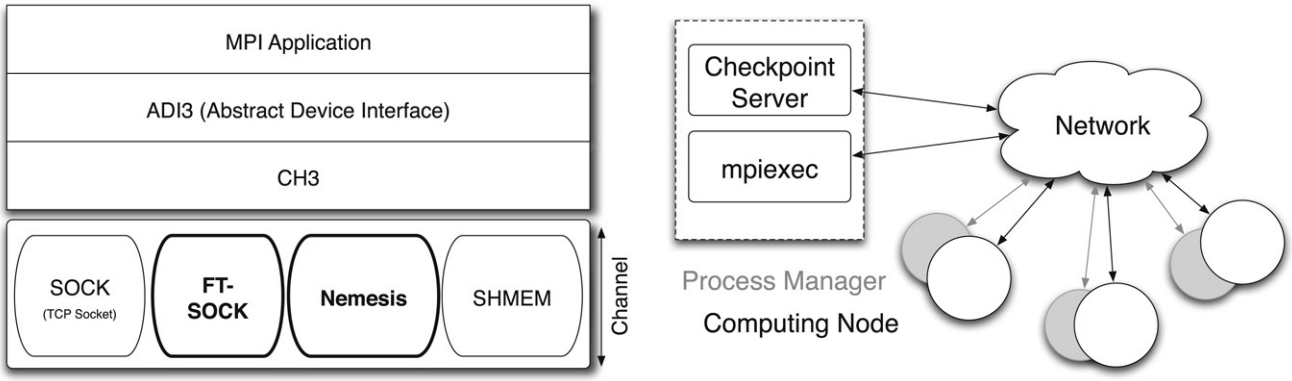
MPICH2 is a new implementation of the MPI standard, which extends results obtained in MPICH and addresses the issue of the MPI2's new functionalities. MPICH2 is structured in three layers: (1) the abstract device interface (ADI3) which links the MPI standard to an extended set of high-level communication routines, (2) the CH3 device which abstracts the ADI3 routines to an API composed of a few (ten to twenty) communication routines, and (3) a channel 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 Fig. 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. We also implemented the blocking checkpointing mechanism in the Nemesis channel [16] to allow us to use high-performance networks.

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 CH3 interface's communication system is based on sequences of request-to-sends and request-to-receives 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 (cf. 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.



(a) MPICH2-Pcl architecture.

(b) Typical deployment of MPICH2-Pcl environment.

Fig. 4. MPICH2-Pcl architecture and typical deployment.

Nemesis channel. The Nemesis is an existing channel that uses shared-memory for intranode communication and high-performance networks for inter-node communication. The fact that Nemesis has a single queue for send requests makes it relatively simple to block sends once the marker messages have been sent. This is done by enqueuing a special *stopper* request on the single send queue, which prevents subsequent send requests from being sent. In order to resume sending messages, the stopper request is dequeued from the send queue and discarded, allowing subsequent messages to be sent.

In the Pcl protocol, packets from a particular process must be blocked once a marker message has been received from that process until the checkpoint has been taken. Because Nemesis has a single receive queue, which may contain packets from any process, it is more difficult to block the reception of packets from a particular process. When a packet is received from a blocked process, the packet is copied into a *delayed receive* queue. Then, after the checkpoint has been taken and messages can be received from any process, the packets in the delayed receive queue are handled before we receive new packets from the receive queue. Upon restarting from a checkpoint, the process discards any packets in the delayed receive queue.

When the process forks before taking a checkpoint image, the lower-level communication mechanism is shut down. In this shutdown, the shared-memory regions are unmapped and any network interfaces are closed. User-level networks typically map a region of memory from the network interface controller directly into the process's address space, through which the process communicates with the NIC. When the process forks, both the child and parent processes have mappings to the same region. Therefore, care must be taken when shutting down the network interface in the child so that the network interface is not also closed for the parent. For the GM communication library using Myrinet, we had to access GM's internal structures to free memory associated with the interface and unmap the memory region to the NIC, without sending close commands that would normally be sent to the NIC when closing the interface.

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 its 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 MPDs 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 PM 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. The current MPD implementation does not allow their direct use. Rather than modifying the MPD environment, we implement a simpler environment, which we call a fault tolerant process manager (FTPM). This environment does not contain MPD daemons and it is used to start, to manage, to detect failures, and to restart applications. The FTPM is composed of an *mpiexec* program and of a modified version of PMs. 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* launches the checkpoint servers, and then the MPI processes through the PMs. Process spawning is done using a *ssh* command. To improve the execution time, these spawns are done in parallel, and 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.

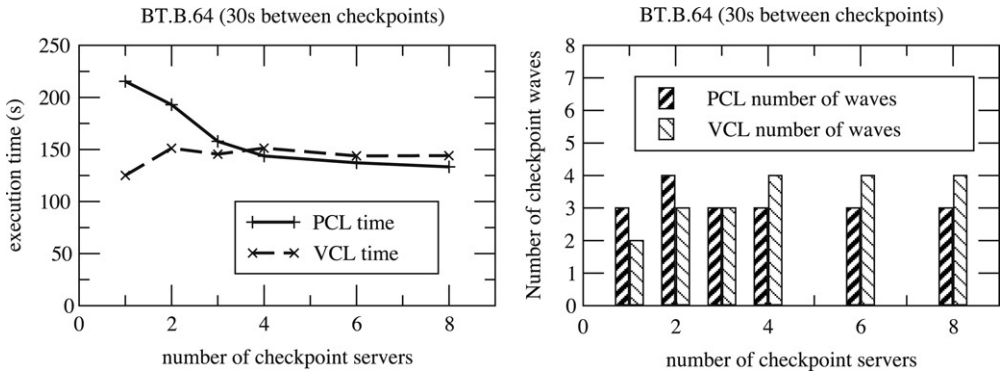


Fig. 5. Impact of the number of checkpoint servers on BT class B for 64 processes with a given period of time between checkpoints.

Each MPI process publishes its location to the others by associating in the distributed database its rank to a *business card* (composed of the process IP address, hostname and port to connect). The database is also used to store the last successful checkpoint wave number, and to locate which checkpoint server holds which local checkpoint. Because 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 a Gigabit-Ethernet 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 [17] is a physical platform featuring 13 clusters, each with between 20 and 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 this article, it consists of 964 computers featuring four architectures (Itanium, Xeon, G5 and Opteron), organized 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 2 GHz AMD Opteron 248 dual-processors. 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 controllers (NICs).

One major feature of the Grid5000 project is the ability of the user to boot her own environment (including the operating system, distribution, libraries, etc.) on all the computing nodes booked for her job. We used this feature to run all our measurements in a 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 times.

Most of the experiments were done using NAS parallel benchmarks (NPB-2.3) [18] 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. Gigabit-Ethernet clusters

Fig. 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 s. Thus, according to the implementation, after having fully transferred a checkpoint image to a checkpoint server, the system waits for 30 s 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 servers and number of computing nodes: from 1 server per 64 compute nodes to 1 server per 8 compute 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

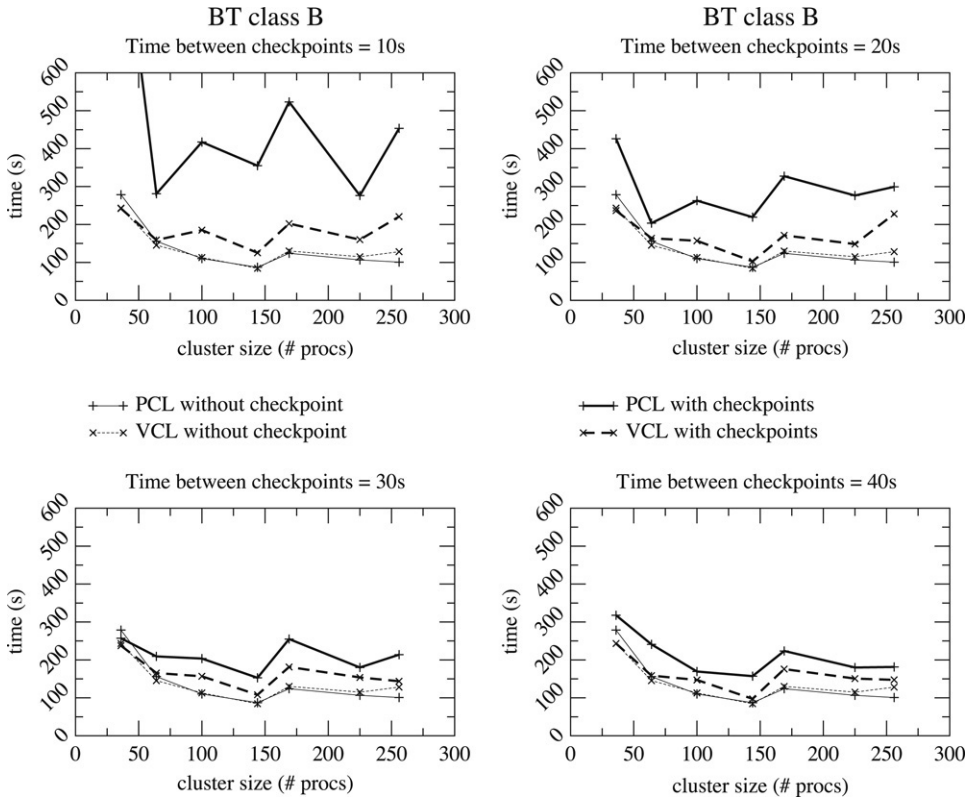


Fig. 6. Execution time as function of the number of processes for four checkpoints frequencies.

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 compete with the checkpoint transfer 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 Fig. 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 same number of checkpoint servers (9).

Without checkpoints, the two implementations behave similarly for all the sizes. The MPICH2 implementation is slightly more efficient for 256 processors. We can observe that the BT class B scales up to 144 nodes. Then, for all implementations, there is an observable slowdown at 169 processors and performances improve again afterwards. 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 controller between the two processors.

Ignoring the graph showing the results for 10 s between checkpoints, where the communications are heavily perturbed by the blocking protocol, one can see that increasing the number of nodes has no measurable impact on the overhead of checkpointing for either of the protocols. The blocking protocol is subject to large performance degradation when the checkpointing frequency is high. It spends most of the time synchronizing to make a global checkpoint. Because MPI communications happen in bursts, as the checkpoint frequency increases, there is a greater probability that the checkpointing operation will interfere with MPI communication. The Vcl implementation does not introduce the same synchronizations,

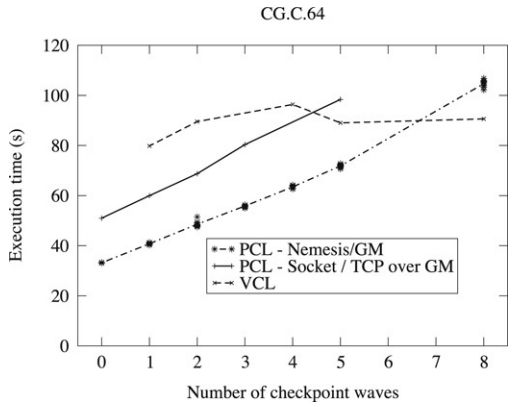


Fig. 7. Impact of the number of checkpoint waves over a high speed network.

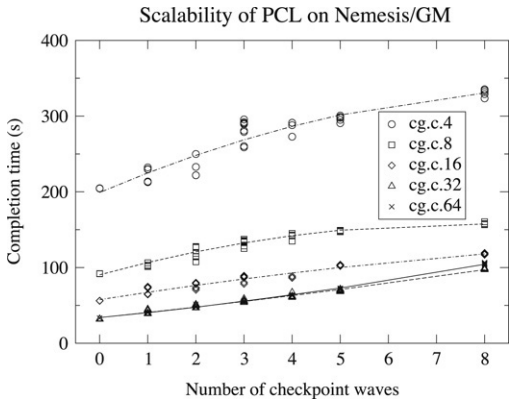


Fig. 8. Impact of the size of the system for varying number of checkpoint waves over high speed network.

and is therefore 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

Fig. 7 shows the results of the CG class C benchmark on 64 processors run over a 32-node cluster interconnected by a Myri2000 network. Two nodes were used as checkpoint servers, and the computing nodes were distributed equally among the checkpoint servers. For the two TCP implementations (PCL – Socket and VCL), the experiments were conducted with the MX-2G 1.1.1 driver from Myricom, enabling Ethernet over Myri2000. The PCL – Nemesis/GM line presents the performance of the other PCL implementation using the Nemesis channel of MPICH2 over GM version 2.1.26. The figure shows the completion time of the benchmark as a function of the number of checkpoint waves during the execution. In order to evaluate the impact of the number of checkpoint waves, we ran the benchmark varying the timeout values between checkpoints, and obtained the number of checkpoint waves successfully completed from the traces.

The execution times for both Pcl implementations are proportional to the number of checkpoint waves. 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 is 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, resulting in unnecessary copies and a high latency overhead. This is why Pcl performs much better than Vcl for this benchmark.

The performance hit is even more obvious when we compare the differences between the VCL implementation and the Nemesis implementation of PCL. In this case, the VCL implementation behaves better only with a checkpoint wave every 15 s or less. This is easily explained, since VCL uses TCP over Myrinet and adds unnecessary copies by its design through a communication daemon. However, one has to consider that any implementation of the non-blocking strategy will have to copy messages, and to introduce new message queues in the driver. Those queues will have to be polled at restart time and take precedence over the normal message queue. So, for high speed networks, when a performance bottleneck is introduced in the critical path of message reception, a blocking implementation will always perform better for low frequencies of checkpoint waves.

Another issue of the blocking implementation is its scalability. In the Fig. 8, we present the completion time of many CG.C benchmarks with between 4 and 64 processes running on the same cluster as a function of the number of checkpoint waves. We used only the PCL implementation over Nemesis, since it is the implementation presenting the best performance for this cluster.

The 64- and 32-process deployments present approximately the same performance. This is due to the fact that the CG.C benchmark is I/O bound. Since for these deployments, one process is running on each of the two processors in every node, two processes must share the single NIC. We therefore see a smaller improvement than for the other experiments, which ran with one NIC per process.

One can see that all the curves demonstrate a slowdown proportionate to the number of checkpoint waves. All the benchmarks show approximately the same slope, indicating that, up to the number of processes we measured, the impact of taking checkpoints is not particularly sensitive to the number of processes. While further experimentation with even larger systems is necessary, this suggests that the PCL algorithm scales well over high-performance networks.

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 [19] utility. This is a ping-pong test for several message sizes and small perturbations of these sizes. This test shows that the network is

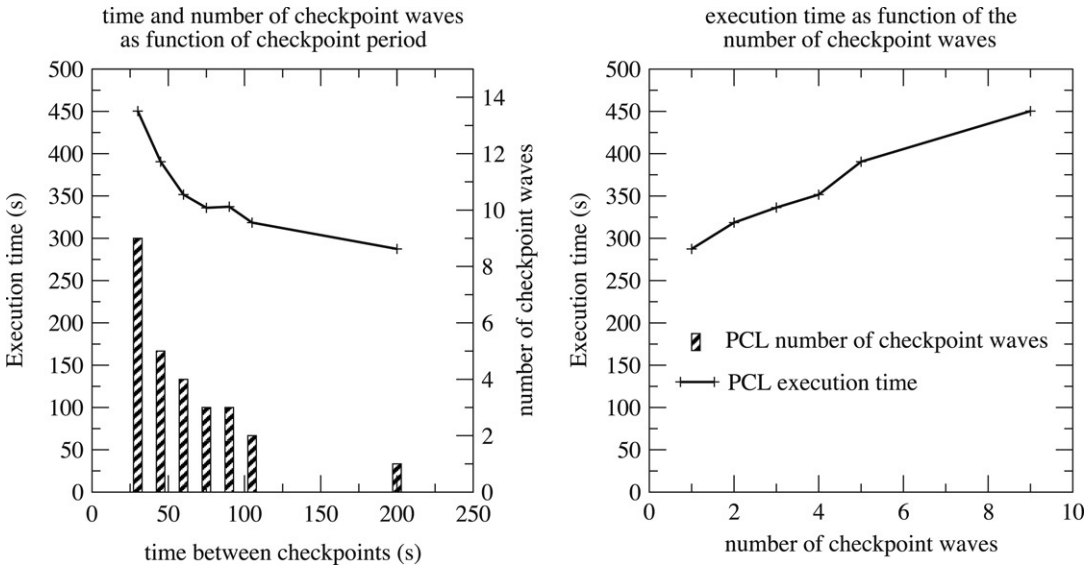


Fig. 9. Impact of checkpoint frequency on blocking checkpointing at large scale (400 processes).

up to 20 times faster between two nodes of the same cluster than between two nodes of two distinct 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 B). 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.

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.

Fig. 10 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 60 s, 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 60 s.

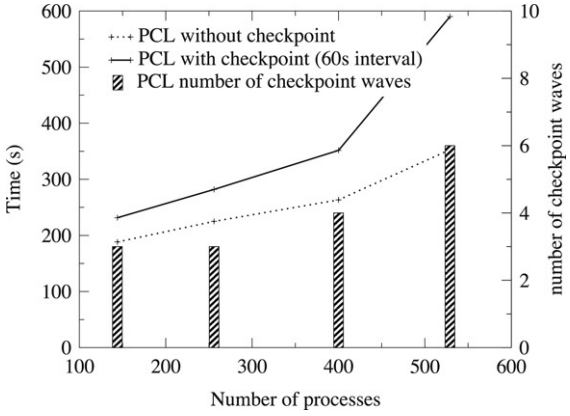


Fig. 10. Impact of large scale on blocking checkpointing.

This is confirmed by the Fig. 9, which presents on its left side the completion time and number of checkpoint 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 checkpoints, 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 its performance to that of 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 coherence 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 progress engine and queue mechanisms.

The experimental study demonstrated that for high-speed networks, the blocking implementation gives the best performance for sensible checkpoint frequencies. 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 a 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. Evaluating the MTTF (mean time to failure) of the system can significantly improve performances, since the best value for the checkpoint wave frequency is close to the MTTF, trying to make a checkpoint just before every failure. Components detecting an increasing failure probability (e.g. through their CPU temperature probe) should also trigger a checkpoint wave.

The non-blocking protocol seems to provide good performances for large scales, but suffers from implementation issues. We plan to integrate this protocol in the MPICH2-Nemesis framework in order to improve its performances and evaluate it on high speed networks.

Acknowledgements

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.

The first author's work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, US Dept. of Energy, under Contract DE-AC02-06CH11357.

References

[1] D.A. Reed, C. da Lu, C.L. Mendes, Reliability challenges in large systems, Future Generation Computer Systems 22 (3) (2006) 293–302.

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

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

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

[5] G. Burns, R. Daoud, J. Vaigl, LAM: An open cluster environment for MPI, in: Proceedings of Supercomputing Symposium, 1994, pp. 379–386.

[6] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, F. Cappello, Improved message logging versus improved coordinated checkpointing for fault tolerant MPI, in: IEEE International Conference on Cluster Computing, Cluster 2004, IEEE CS Press, 2004.

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

[8] E. Strom, S. Yemini, Optimistic recovery in distributed systems, Transactions on Computer Systems 3 (3) (1985) 204–226.

[9] K.M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, Transactions on Computer Systems 3 (1) (1985) 63–75.

[10] E.N. Elnozahy, D.B. Johnson, W. Zwaenepoel, The performance of consistent checkpointing, in: Symposium on Reliable Distributed Systems, 1992, pp. 39–47.

[11] W. Gropp, E. Lusk, Fault tolerance in MPI programs, Journal High Performance Computing Applications (2002) (special issue).

[12] S. Sankaran, J.M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, E. Roman, The LAM/MPI checkpoint/restart framework: System-initiated checkpointing, in: Proceedings, LACSI Symposium, Sante Fe, New Mexico, USA, October 2003.

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

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

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

[16] D. Buntinas, G. Mercier, W.D. Gropp, Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem, in: B. Mohr, J.L. Träff, J. Worringen, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, in: Lecture Notes in Computer Science, vol. 4192, Springer, 2006, pp. 86–95.

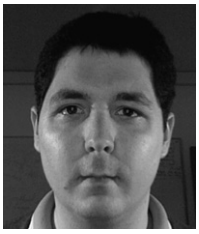
[17] F. Cappello, et al., Grid'5000: A large scale, reconfigurable, controlable and monitorable grid platform, in: Proceedings of IEEE/ACM Grid'2005 Workshop, Seattle, USA, 2005.

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

[19] Q. Snell, A. Mikler, J. Gustafson, Netpipe: A network protocol independent performance evaluator, in: IASTED International Conference on Intelligent Information Management and Systems, June 1996.

Darius Buntinas is an assistant computer scientist at Argonne National Laboratory. His primary research area is communication subsystems, specifically support for MPI and hybrid programming models. He received his B.S. and M.S. degrees from Loyola University Chicago, and his Ph.D. from The Ohio State University.

Camille Coti is a Ph.D. student in the Grand-Large team of INRIA. She obtained a Master of Science in telecommunications at the engineering school "Telecom INT", specializing in parallel and distributed computing. Her research interests include large-scale MPI computing and fault tolerant protocols for MPI computing.



Thomas Herault is associate professor at the Paris South University. He defended his Ph.D. on the mending of transient failure in self-stabilizing systems under the supervision of Joffroy Beauquier. He is a member of the Grand-Large INRIA team and works on fault-tolerant protocols in large scale distributed systems. He leads the MPICH-V project of the Grand-Large team, whose goal is to evaluate and compare fault tolerant protocols for MPI.

Pierre Lemarinier is a postdoc at INRIA Futurs. He defended his Ph.D. on the fault tolerant protocols for message passing systems under the supervision of Joffroy Beauquier and Franck Cappello at the Paris-Sud University. He is a member of the Grand-Large INRIA team and works on fault tolerant protocols for large scale distributed systems. He contributes to the MPICH-V project since its foundation.

Laurence Pilard defended her Ph.D. on December 2005 under the supervision of Joffroy Beauquier at the University of Paris Sud (France). Then she was a teaching and research assistant at the University of Paris Sud for one year. Laurence Pilard worked on self-stabilzing systems and large scale systems in the Laboratoire de Recherche en Informatique (LRI) in the Parallelism Team. Now, Laurence Pilard is a postdoctoral researcher at the University of Iowa (USA) in the Department of Computer Science and she is working with Professor Ted Herman on Sensors Networks.

Ala Rezmerita is a Ph.D. student in the Cluster and Grid group of the LRI laboratory at Paris-South University and is a member of the Grand-Large team of INRIA. She has obtained a Master in computer science in 2005 from the French University of Paris 7 — Denis Diderot. She contributes to the MPICH-V project, a fault tolerant MPI implementation comparing different fault tolerant protocols. Her research interests include parallel and distributed computing, grid middleware and Desktop Grid.

Eric Rodriguez is an associate engineer at INRIA; his research area is fault tolerance in MPI. He received his engineer degree in Mathematics and Computer Science from ISTIL.

Franck Cappello holds a Research Director position at INRIA, after having spent 8 years as a CNRS researcher. He leads the Grand-Large project at INRIA and the Cluster and Grid group at LRI. He has authored more than 50 papers in the domains of High Performance Programming, Desktop Grids and Fault tolerant MPIs. He is editorial board member of the "international Journal on GRID computing" and steering committee member of IEEE/ACM CCGRID. He organizes annually the Global and Peer-to-Peer Computing workshop. He has initiated and heads the XtremWeb (Desktop Grid) and MPICH-V (Fault tolerant MPI) projects. He is currently involved in two new projects: Grid eXplorer (a Grid Emulator) and Grid5000 (a Nation Wide Experimental Grid Testbed).