Coordinated checkpoint versus message log for fault tolerant MPI

Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik, Franck Cappello
LRI, Université de Paris Sud, Orsay, France
E-mail: {bouteill,lemarini,gk,fci}@lri.fr

Abstract

MPI is one of the most adopted programming models for Large Clusters and Grid deployments. However, these systems often suffer from network or node failures. This raises the issue of selecting a fault tolerance approach for MPI. Automatic and transparent ones are based on either coordinated checkpointing or message logging associated with uncoordinated checkpoint. They are many protocols, implementations and optimizations for these approaches but few results about their comparison. Coordinated checkpoint has the advantage of a very low overhead on fault free executions. In contrary a message logging protocol systematically adds a significant message transfer penalty. The drawbacks of coordinated checkpoint come from its synchronization cost at checkpoint and restart times. In this paper we implement, evaluate and compare the two kinds of protocols with a special emphasis on their respective performance according to fault frequency. The main conclusion (under our experimental conditions) is that message logging becomes relevant for a large scale cluster from one fault every hour for applications with large dataset.

Keywords: Fault tolerant MPI, coordinated checkpoint, message log, performance.

1 Introduction

A current trend in high performance computing is the use of large scale clusters gathering hundreds or thousands of processors. The June 2003 Top 500 list of highest performance machines in the world shows a total number of CPUs of more than 239,000. The distributions of the number of CPUs across the entire lists from 1993 to 2003 is given in figure 1. Top 500 lists are plotted with different colors and clusters appear in the graph as ranked in the lists. In this figure, a particular list appears as a dot cloud. Between 1993 and 2003 there is an increase of the number of CPUs of about one order of magnitude. 1/3rd of the installations have 500 CPUs or more. These elements suggest that the high performance computing community is globally building larger clusters and because of this long term trend, it is likely that in the near future a large number of clusters will have 1000 or more CPUs.
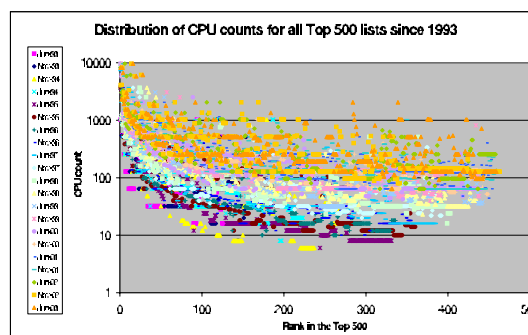

Figure 1. CPU count statistics for the cluster category of the Top 500 lists.

Large clusters are subject to node and network failures [8, 11] requiring the use of fault tolerance mechanisms for long duration applications. Users of these platforms who are familiar with explicit message passing and their applications often use MPI as the message passing library. There are several ways to implement fault tolerance in MPI: a) the programmer of the application may save periodically intermediate results on reliable media during the execution in case of global restart, b) the functions of the MPI implementation may return fault notification information and accept reconfiguration of the communication context and c) the MPI implementation provides a fully automatic fault detection and transparent recovery. The automatic approach suffers either of limited fault tolerance capabilities or high resource cost. Examples of such automatic fault tolerant MPI implementations are following coordinated checkpoint and message log strategies.

While global checkpoint and message log have been deeply studied and many optimizations have been proposed, there is no research result concerning the comparison of the two approaches for large clusters. More specifically, it is an open issue to determine which of the two approaches leads to a better overall performance when faults are occurring during the execution. Coordinated checkpoint has the

advantage of a very low overhead as long as the execution stays fault free. Its drawbacks are the synchronization cost before the checkpoint, the cost of synchronized checkpoint and the restart cost after a fault. The performance of coordinated checkpoint is highly related to the checkpoint storage location. If the checkpoint images are stored remotely on an independent checkpoint server, the global checkpoint and restart implies a huge stress on the checkpoint server which may lead to a high overhead, even if the checkpoint system uses several checkpoint servers. Message log does not suffer from these problems, since processes checkpoint and restart independently of the others. However, the log adds a significant penalty for all message transfers even if no fault occurs during the execution. The differences between the two approaches suggest that the best approach depends on the fault frequency: the higher the fault frequency is, the smaller the checkpoint period should be for coordinated checkpoint. Since every checkpoint and every fault increase the overhead, the execution time of the coordinated checkpoint approach increases rapidly. The purpose of this article is to determine for which fault frequency this overhead becomes higher than the one induced by the message log strategy.

We have implemented two strategies: a pessimistic message log protocol (MPICH-V2) and a global checkpoint strategy based on Chandy-Lamport algorithm (MPICH-CL).

For a fair comparison of the two approaches and independence with the MPI implementation, we have implemented them in the same framework and tested them in the same conditions. The two fault tolerance strategies are implemented at the same level of the software hierarchy, between a MPI high level protocol management layer and the low level network transport layer (TCP). The two strategies also use the same checkpoint service.

The second section of the paper presents the research results previously obtained in fault tolerance for explicit message passing. We discuss the various approaches according to several parameters and especially scalability and performance in practice. Section 3 describes the framework used for developing and comparing fault tolerant protocols. It also discusses the two protocols compared in this study: a coordinated checkpoint protocol directly deriving from the original "Chandy-Lamport" algorithm and a message log protocol combined with an uncoordinated checkpoint strategy. Performance comparison of the two protocols is presented in section 4.

2 Related work

Two principal techniques have been proposed for Rollback-Recovery protocols [10]: global checkpoint or message log. Global checkpoint consists in taking the snapshot of the entire system state regularly so that when a fail-

ure occurs on any process, all the system rolls back to the latest checkpoint image to continue the computation.

In message log protocols, all processes can checkpoint without being coordinated. A process execution is supposed to be piecewise deterministic [20], which means be governed by its message receptions. Thus all communications are logged in a stable media so that only the crashed processes rollback to a precedent local snapshot and execute the same computation as in initial execution, receiving the same messages from the stable storage.

In the next section, we describe the global checkpoint and message log approaches and their use in the different fault tolerant MPI project [12].

2.1 Global checkpoint

There are three classes of global checkpoint protocols [10]: uncoordinated, communication induced and coordinated checkpoint.

In uncoordinated checkpoints protocols without message log, the checkpoints of each process are executed independently of the other processes and no further information is stored on a reliable media leading to the well known domino effect (processes may be forced to rollback up to the execution beginning). Since the cost of a fault is not known and there is a chance for losing the whole execution, these protocols are not used in practice.

Communication Induced Checkpointing (CIC) tries to take advantage of uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, it piggy backs causality dependencies in all messages and detects risk of inconsistent state. When such a risk is detected, some processes are forced to checkpoint. While this approach is very appealing theoretically, relaxing the necessity of global coordination, it turns out to be inefficient in practice. [2] presents a deep analysis of the benefits and drawbacks of this approach. The two main drawbacks in the context of cluster computing is 1) CIC protocols do not scale well (the number of forced checkpoints increases linearly with the number of processes) and 2) the storage requirement and usage frequency are unpredictable and may lead to checkpoint as frequently as coordinated checkpoint.

In coordinated checkpoint protocols, all processes coordinate their checkpoints so that the global system state composed of the set of all process checkpoints, is coherent. If every process restarts from its checkpoint image, there is no process for which a message is considered to be received while the message's sender did not send it.

The first algorithm to coordinate all the checkpoints is presented in [7]. This algorithm supposes all channels as FIFO queues. Any process can decide to start a checkpoint. When a process checkpoints, it sends special messages called markers in the computing channels. When a process receives a marker for the first time, it checkpoints.

After beginning a checkpoint, all messages received from a neighbor is added to the checkpoint image, until the marker reception. In fact most fault tolerant MPI implementations based on global checkpoint [1, 19] use this algorithm for its simplicity. For example, Cocheck [19] is implemented on top of the message passing system to be easily adapted for different system. Starfish [1] modifies the MPI API so that users can integrate some checkpointing policies. Comparatively to Cocheck and Starfish, our implementation of this protocol (MPICH-CL) is implemented as a driver for MPICH, independently of the MPI API, thus at a lower level of the software stack.

2.2 Message Log

There are three classes of message log protocols: optimistic, pessimistic and causal. The properties of the three classes can be found in [3].

Pessimistic log protocols ensure that all messages received by a process are first logged by this process before it causally influences the rest of the system. MPICH-V [5] is based on this type of protocol. It uses reliable processes called channel memories. Every MPI computing node is connected to a channel memory. When a node sends a message, it sends it to the channel memory of the receiver and when it wants to receive a message, it asks its own memory channel for it. Thus all messages are logged and can be retrieved after a crash. In MPICH-V2 [6], an optimization of MPICH-V, messages are logged locally to the sender and only an information, about the reception causal order, is logged on reliable media.

The optimistic log protocols [20] eventually log receptions but do not wait for them before sending new messages. Therefore they are faster in non-faulty executions but do not exclude to rollback some non-crashed processes if a fault occurs before the receptions logging.

Causal log protocols try to conceal the optimistic and the pessimistic approaches. When a process sends a message, it logs it locally and appends information about its past receptions. Thus when a process crashes, it can either retrieve information about its initial execution's receptions, or no process depends on its precedent computation. Manetho [9] is a typical causal log protocol. The main difference between MPICH-V2 and Manetho, separating them in different classes of protocols, is that information -about receptions- is logged on a reliable media rather than appended to the messages.

EGIDA [18] is a message log toolbox to test and compare different protocols. EGIDA replaces MPICH's upper layer calls of communication functions and runs over the lower layer of P4. The costs of recovery in message log protocols have been studied in [17] using EGIDA. Compared to EGIDA, MPICH-V2 is integrated in the lower level of the MPI library, as a specific driver. This integration level allows us to fully control what happens inside the MPICH device driver at the cost of a higher porting effort.

2.3 Checkpoint optimizations

Checkpointing has a significant overhead increasing the application execution time. There is a tradeoff between the checkpoint cost and the cost of restarts due to faults that leads to the selection of the best checkpoint interval [15, 16, 21]. However, the checkpoint interval should be computed from the application execution time without checkpoint and the cluster mean time between failure (MTBF), thus for each configuration of application, platform performance and number of processes. Knowing all these parameters before the execution is not easy in practice since the execution time of many applications depends on the input parameters. Instead of considering this exact value, we will consider two extremes depending only on pragmatic data: the time to take a checkpoint and the MTBF. The first parameter only depends on the application memory occupation which is not related to the execution time. The two considered extremes are: 1) executing checkpoint as frequently as possible and 2) executing only two checkpoints during the MTBF (less checkpoints would lead to use a checkpoint frequency too close to the fault frequency potentially avoiding the application to progress in its execution).

Usually, the technique of global checkpointing is associated with a remote storage of the process checkpoint images. This approach leads to stress the network between the nodes and the checkpoint repository proportionally to the number of processes checkpointing concurrently. An obvious optimization would be to store the checkpoint image of the processes on the nodes themselves. This technique has two main drawbacks: a) restarting the system involves repairing the faulty nodes in order to retrieve the checkpoint image of the dead process. The MTTR (mean time to repair) becomes a part of the restarting overhead, which is not the case if the process can be restarted from a remotely stored checkpoint image and if there are enough spare computing nodes available. b) Depending on the fault type, it might be impossible to restart the execution. This is the case if a single checkpoint image on a disk becomes not accessible anymore. These two drawbacks make this approach falls into the category of non fault tolerance protocols (it might still be used for migration).

3 Fault tolerance framework

The fault tolerance framework gathers several components: the adapted MPICH layered communication library, a dispatcher and a checkpoint server. Additionally, some specific components may be added like a checkpoint scheduler for the specific purpose of one fault tolerance protocol.
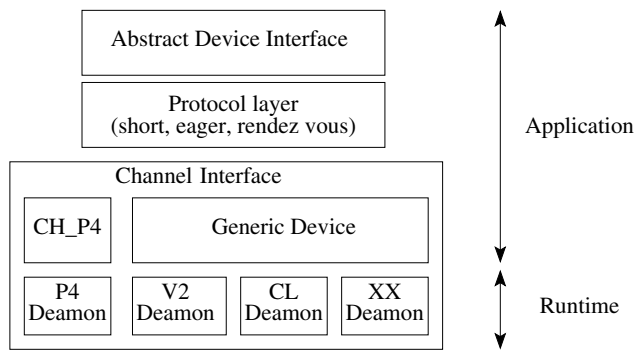
Figure 2. General architecture of MPICH and the generic runtime of our framework

3.1 Generic architecture

MPICH is a layered implementation of MPI described in [13](figure 2). The MPI API is implemented by high-level primitives of the Abstract Device Interface (ADI). The ADI is implemented over the protocol layer which implements the short, eager and rendez-vous protocols. These protocols are implemented over the basic primitives of the channel interface .

MPICH-V2 and MPICH-CL are implemented on top of MPICH 1.2.5 [6] within the channel interface of MPICH using two components: a generic device and a specific communication daemon. The generic device (identical for MPICH-V2 and MPICH-CL) implements a set of six primitives used by the protocol layer. The device includes two communication functions PIbrecv and PIbsend which are blocking operations for receiving or sending a block of data. It includes four other functions: PInprobe to check if a message is pending; PIfrom to get the identifier of the last message sender; PIiInit to initialize the channel and PIiFinish to finish the execution.

As in the P4 channel, the reference implementation for TCP/IP, the MPI process does not connect directly to all the other computing nodes. This is the job of a communication daemon which is connected to the MPI process and handles the asynchrony of the network. This daemon is specific for all fault tolerant protocols and implements the fault tolerance strategy. At runtime, it establishes a UNIX socket and then spawns the MPI process. Two kinds of messages are exchanged using this socket: control messages (for init, finish and probe) and protocol messages (bsend, breceive).

The device is generic in the sense that an application compiled with it can run unchanged with the MPICH-V2 daemon or the MPICH-CL one.

Figure 3 presents typical settings for MPICH-V2 and MPICH-CL. MPICH-V2 implements a pessimistic sender-based protocol. It uses a dispatcher, a checkpoint scheduler, event loggers, checkpoint servers, computing nodes and their communication daemons. In figure 3 the dispatcher, the event logger and the checkpoint scheduler are located on the same computer. MPICH-CL implements a global checkpoint protocol using the algorithm presented in [7]. It uses a large part of the MPICH-V2 architecture as shown in figure 3. The only architectural difference is the disappearance of the event loggers.

The implementations of the dispatcher and the checkpoint servers are much the same in MPICH-V2 and MPICH-CL and are presented in the next section. The implementations of the communication daemons of the computing nodes and the checkpoint scheduler are different in the two protocols and are presented in specific sections.

3.1.1 Checkpoint architecture

The checkpoint server is a reliable repository storing the checkpoint images of the MPI processes and of the communications daemons. The checkpoint of the MPI process uses the Checkpoint Condor stand-alone library. The checkpoint of the communication daemon is handled by a user level method.

When the MPI process receives a checkpoint order from the communication daemon, the process forks in two parts. The first part sends its process image generated by the Condor library [14] to the communication daemon. Simultaneously, the second forked part continues the MPI application execution. During the transfer of the MPI process checkpoint image and its own checkpoint, the communication daemon does not suspend the communications with the others computing nodes. These two properties ensure that the checkpointing can be overlapped with MPI process computations.

User level checkpointing of the communication daemon is quite different between MPICH-V2 and MPICH-CL, as outlined in 3.2.1.

The checkpoint server is a multiprocess server. A process is dedicated to each concurrent checkpointing request.

3.1.2 Dispatcher

The run preparation consists in a shell script (which may inter-operate with a batch scheduler) creating a 'program file' from a list of available machines for a run and the MPICH-V2/MPICH-CL specific commands (executable, number of processors to be used). The program file is equivalent to the 'P4PGFILE' of the original MPICH-P4. It describes for each machine 1) its role inside the system (Computing Node, Event Logger, Checkpoint Server, Checkpoint Scheduler) and 2) the list of options for that role. The user can specify these options through the use of an extended MPICH-like machines file, or with general defined attributes by using special option flags, or with default configurations.
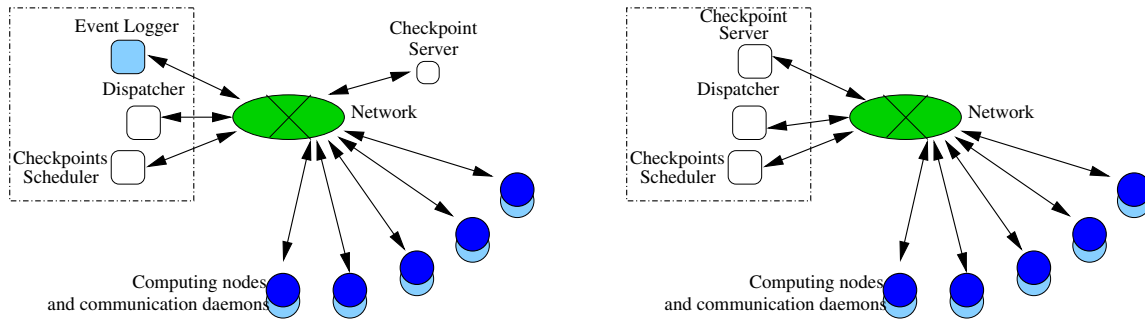
Figure 3. Typical setup of a MPICH-V2 and MPICH-CL

The execution monitor first launches the execution (by rsh or ssh) of the different programs, and then monitors the execution potentially re-launching the crashed programs. To detect faulty nodes, we assume that the whole execution runs on a synchronous (e.g. a Cluster) or controlled area network (e.g. a Grid). In such networks, a socket disconnection is considered as a trusty fault detector. Basically, at the beginning of the execution, one socket is opened for every computing node. The monitor pulls these sockets for detecting disconnections. The number of open sockets might be quite large. However, this has a negligible influence on the overall network usage, since only very few messages are sent during the whole execution. At the end of the program, the monitor receives a finalize message from every computing node. It cleans the execution pool by stopping the different auxiliary programs.

3.2 Specific components

Both implementations use two common components: the checkpoint scheduler and the communication daemon. However these components have specific implementations corresponding to the protocol features.

3.2.1 MPICH-V2

Checkpoint scheduler

The checkpoint scheduler is not really required by the fault tolerance protocol of MPICH-V2. All processes checkpoints do not have to be coordinated. Nonetheless, in message log protocols, the message payload is saved and checkpointing reduces the storage occupation of the logged messages.

Thus a checkpoint scheduler is important for efficiency reasons. The checkpoint scheduler developed for MPICH-V2 is a simple state machine. It can ask for information about the amount of messages logged and can order a node to checkpoint. Different scheduling policies, fair or not, can be easily plugged and experimented. We have developed two checkpointing policies: a round robin that is used in this paper, and an adaptive one.

Communication daemon

The daemon is basically a select loop: it handles one socket for every computing node and one socket for every server (event logger in MPICH-V2, checkpoint server and checkpoint scheduler). These sockets are TCP streams and every send or receive operation is asynchronous. Thus, a communication is not blocked by a slower one. On the other hand, the communication across the UNIX socket to the MPI process is synchronous and its granularity is the whole protocol message.

Thus a daemon is composed of a list of pending messages and one list of messages to send per computing node. MPICH-V2 is based on a pessimistic log protocol, so when a message is delivered from the daemon to its local MPI process, the daemon sends asynchronously to the Event Logger some information about this reception. When the daemon has to send a message to another process, the communication daemon logs this message locally and has to wait if necessary the acknowledgment from the Event Logger about all the preceding receptions.

Event logger

An Event Logger is a reliable medium. It stores causal informations about message receptions of computing nodes. When a node crashes and restarts, its daemon asks the Event Logger for information about all receptions it has made in precedent execution. The daemon can then enforce the same receptions in re-execution and thus the same re-execution.

3.2.2 MPICH-CL

Checkpoint scheduler

In the original Chandy-Lamport algorithm, some processes decide a global snapshot and send markers to all neighbors. A process takes a local snapshot on its own decision or when it receives a marker. The checkpoint scheduler developed for MPICH-CL can be viewed as a simple process that is connected to all computation nodes. Moreover it is the only one that can decide of taking a snapshot. The checkpoint frequency is a parameter of the runtime. A number n is given as a parameter so that it launches the snapshot procedure every n seconds.

When a computation node has performed its checkpointing, it informs the checkpoint scheduler, so that the checkpoint scheduler can manage global acknowledgment between all checkpoint servers.

Communication daemon

As in MPICH-V2, the communication daemon is connected to the computing nodes, checkpoint server and checkpoint scheduler. It uses a local Unix socket to communicate synchronously with its associated MPI computing node. All communications between two communication daemons are made asynchronously so the communication daemon is composed of one list of pending messages for all messages received and a list of messages to send per neighbor. When it receives a marker from a neighbor or the checkpoint scheduler, it changes its state to the checkpointing state, piggybacks a signal to a MPI computing node query and adds marker messages to all messages to be send. All messages received before markers and not delivered to the MPI process at the time of the beginning of the local checkpoint are logged to be checkpointed. When all markers from computing nodes and the checkpoint scheduler have been received, the communication daemon sends a message to the checkpoint scheduler and sets its state to "not checkpointing".

4 Experiments

The purpose of this paper is to evaluate the fault frequency (if any, in practice) from which the coordinated checkpoint strategy becomes more time consuming than the message logging strategy. We evaluate this threshold base on an experimental study on real implementations. For the communication performance, we consider MPICH-P4 as the reference implementation.

4.1 Experimental Conditions

We present a set of experiments in order to evaluate the different components of the system.

All experiments are run on a 32-node cluster, connected by a fast ethernet switch. Each node is equipped with an AthlonXP processor, running at 1.5GHz, 1GB of main memory (DDR SDRAM), and a 70GB IDE ATA100 hard drive. The operating system is Linux 2.4.18, the compilers used are GCC and PGI Fortran77 compiler. All tests are run in dedicated mode.

The first experiments are synthetic benchmarks analyzing the individual performance of the subcomponents. The second set of experiments is run with a real application, the BT program. BT is one of the mini-applications of the NAS Parallel Benchmark suite [4], written by the NASA NAS research center to test high performance parallel machines.

For all experiments, we consider a single checkpoint server connected to the rest of the system by a 100 Mb/s switched Ethernet network. While other architectures have
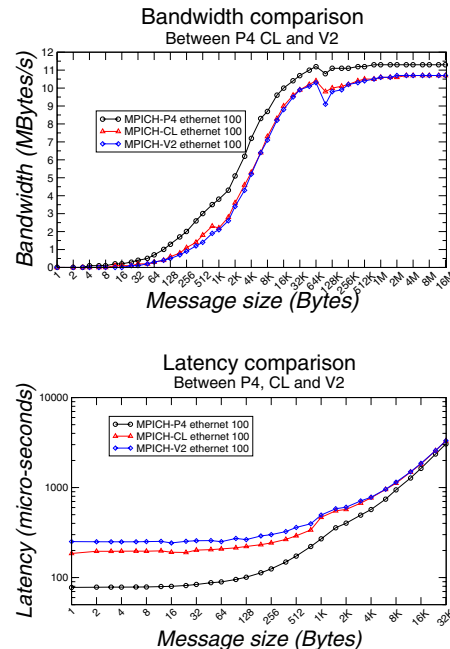
Figure 4. Bandwidth and latency comparison for a ping-pong test using the three different MPI libraries

been studied for checkpoint servers (distributed file systems, parallel file systems), we consider that this system impacts the performance of checkpointing similarly for MPICH-V2 and MPICH-P4.

4.2 Communication performance

As we are comparing two fault tolerant protocols, it is an important issue to validate our implementation compared to a reference one. We have presented a complete performance evaluation of MPICH-V2 in [6]. Figure 4 compares raw performances of MPICH-V2, MPICH-CL and the reference implementation MPICH-P4. As expected, bandwidth is very close for the three protocols. Latency of MPICH-CL is almost half of MPICH-V2 one, but is still not equal to MPICH-P4. Our implementation has more memory copies than the P4 one. As seen in synthetic benchmark presented in figure 5 this has no real impact on our test application performances where MPICH-CL reaches better performances than the reference implementation.

4.3 Checkpoint performances

For this experiment, a dummy MPI application has been written. The checkpointed process only allocates the requested memory size, filling it with random values (as returned by the libc function rand(3)). Then it enters in an infinite probing loop, waiting for a checkpoint order. This
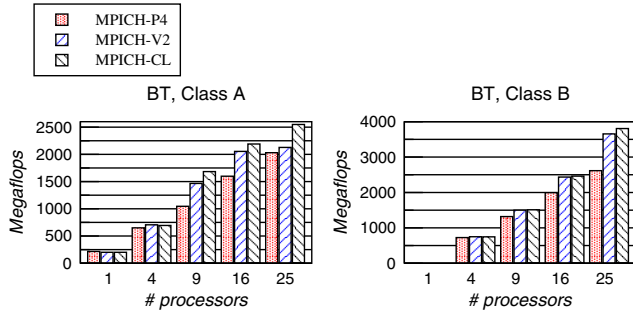
Figure 5. Performance comparison without checkpointing of the P4, V2 and CL implementations for the NPB 2.3 BT Class A and B Benchmark

Figure 6. Time to perform single remote checkpoint for various process image sizes

Figure 7. Time to perform simultaneous checkpoints on the same checkpoint server

application is linked with the MPICH-V2 library, using our remote checkpoint protocol.

Figure 6 outlines the time required to perform a remote checkpoint for various process sizes. The checkpoint time is proportional to the memory size of the process. The last measurement shows the impact of memory swapping on the checkpoint performance. With enough memory, checkpointing a 1 GB process would take about 200 seconds.

Figure 7 presents results of simultaneous checkpoints on the same checkpoint server. Processes are not communicating with each others and have 128MB of allocated memory. Checkpoint time is the same for 1 or 2 simultaneous checkpoints, as the checkpoint of the second process fills unused bandwidth left by the first one. When more than 2 processes are checkpointing simultaneously, checkpoint time increases linearly with the number of concurrent checkpoints.

However the time to perform a global checkpoint using a single checkpoint server does not increase with the number of processes for BT Class A, as shown in figure 8, because the size of each individual process decreases (figure 8). The time to perform uncoordinated checkpoint is related to per process image size and thus for BT Class A decreases with the number of processes. Nonetheless checkpointing every node in MPICH-V2 requires to transfer the same total amount of checkpoint data as in coordinated checkpointing, thus the time to checkpoint all processes is similar to the global checkpoint time of MPICH-CL.

4.4 Impact of faults on performances

Figure 8 presents the time to restart BT Class A after a failure with a single checkpoint server. This experiment does not include re-execution time to reach the failure point, but only the time to re-spawn the MPI computation. For message log it is required to restart only the failed nodes, while for coordinated checkpoint all nodes have to be rolled
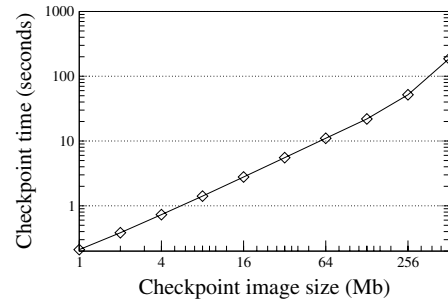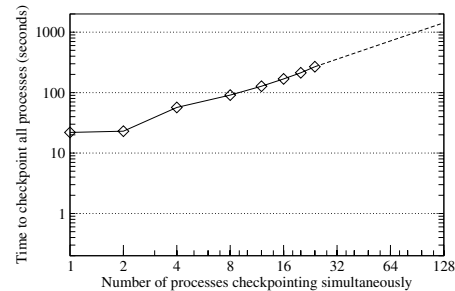
back. As a consequence, restart time for the BT Class A benchmark for coordinated checkpoint is almost constant (the total data to transfer is constant as the number of nodes increases) while the restart time for message log decreases when the number of process increases. This property allows better scalability for fault recovery in message log protocols than in coordinated checkpoint protocols.

The last experiment presents the impact of fault frequency on the total execution time of BT Class B using 25 processes. The number of iterations has been increased to allow longer execution and a lower fault frequency. Average time to perform a global checkpoint of BT Class B on 25 nodes is 68.7 seconds, while time to perform a complete asynchronous checkpoint (all processes have been checkpointed at least one time) is 73.9 seconds. The time to restart after a failure is 65.8 seconds for MPICH-CL while it is 5.3 seconds for MPICH-V2. Each process generates a 32MB checkpoint image. We consider a non checkpointed execution over MPICH-CL as reference time. Even when fault probability is zero, MPICH-V2 requires checkpointing, due to the necessary garbage collection of useless logged messages.

For a fault time interval Tf, we have chosen a checkpoint interval $T = 2/3(Tf - 2*Tc)$, where Tc is the time to checkpoint. This checkpoint interval should be chosen ensuring that the computation progresses despite faults. For the ex-
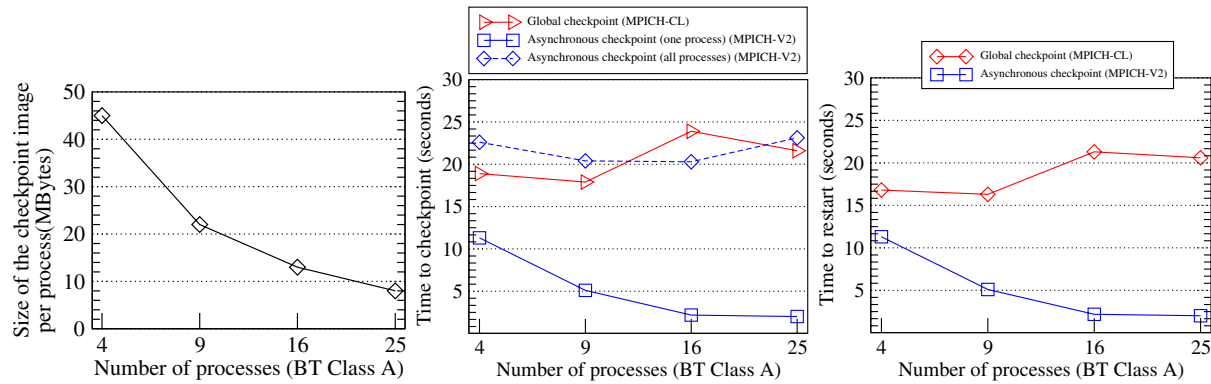
Figure 8. Checkpoint parameter and performance comparison between CL and V2 for BT class A

perimentation, we have considered an average fault delay equals to the half of the checkpoint interval.
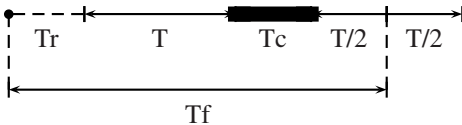

Tr : restart time ($\simeq$ Tc)

Tc : Checkpoint time

T : checkpoint interval

Tf : fault interval

Results are presented in figure 9. As expected the overhead of message logging decreases by 40% the overall performance of MPICH-V2 when no fault occurs compared with a non checkpointed execution of MPICH-CL. Note that in practice, a minimum number of checkpoints is also required for MPICH-CL.

For low fault frequencies, MPICH-CL has better performances than MPICH-V2, due to a lesser runtime overhead. As the fault frequency increases, the overhead due to long restart time of all processes in global checkpointing scheme becomes predominant. Even with such a small dataset, message logging shows better performances than periodic global checkpointing when fault frequency reaches 1 fault every 10 minutes. As the overhead due to recover from a fault is a linear function of the size of the checkpointed processes, with larger datasets it is expected that message log would perform better than global checkpointing for lower fault frequencies. For example, if we consider a 1 GB memory occupation by all processes of the MPI execution, then the checkpoint time (according to linear extrapolation of figure 6 and a multiplicative factor of 10 related to simultaneous checkpointing - figure 7) for MPICH-CL would be approximatively 2000 seconds for 25 nodes. In that case the minimum fault interval ensuring that the application still progresses is 4000 seconds or about 1h. At this point the application execution is progressing very slowly.
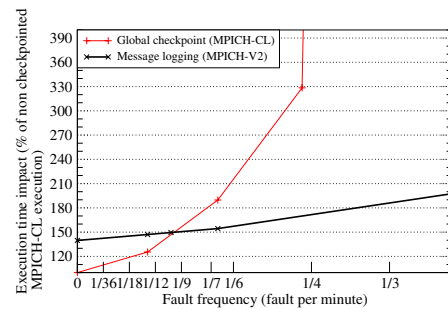
Figure 9. Impact of fault frequency on execution time for BT Class B

For the same conditions with MPICH-V2 and in the worst case (all fault occurring on the same node), the minimum fault interval ensuring that the application still progresses is 400 seconds. Altogether, these figures show that MPICH-V2 allows the application to progress significantly even in the presence of a fault every hour, frequency from which MPICH-CL becomes useless.

Moreover MPICH-CL shows that it cannot tolerate a high rate of faults. When the fault interval reaches twice the time to checkpoint, the execution may never have the time to checkpoint before a fault occurs.

5 Conclusion

Large scale cluster and Grid systems raise the issue of tolerance to frequent and numerous faults. Since these systems are mostly programmed using MPI, the use of a fault tolerant MPI implementation will become unavoidable. Among the automatic/transparent fault tolerant approaches, two main classes can be considered using either coordinated checkpoint or message logging. The purpose of this paper was to compare the overhead imposed by these two techniques according to fault frequency, in the potential perspective of finding a crosspoint from which one technique becomes better than the other.

We have implemented the two fault tolerance approaches from the MPICH 1.2.5 and compared the two resulting implementations (MPICH-V2 and MPICH-CL) after having validated their performance from the MPICH-P4 reference implementation.

We have demonstrated (considering our experimental conditions) that the checkpoint overhead is similar for the two protocols, unlike the recovery one which is significantly higher for coordinated checkpoint, due to a higher stress of the checkpoint server.

The recovery overhead is a main factor differentiating the performance of the two techniques when faults occur during the execution. Even with a small dataset of 30MB per nodes and a rather small cluster (25 nodes), when the fault frequency increases over 1 per 10 minutes, message logging performs better than global checkpointing and offers a better level of fault tolerance. If we simply consider applications with a larger data set (1GB per node), then the coordinated checkpointing protocols becomes useless from 1 fault every hour. For that fault frequency, the application execution still progresses significantly with the message logging protocol. Measuring the performance crosspoint of the two protocols for that data set size belong to a set of planned experiments a) to understand in practice (i.e. with real software) the impact of checkpoint server architecture and high performance networks (Myrinet, Infiniband) on the performance of both fault tolerance protocols and b) to measure the performance of the two checkpoint strategies on large scale clusters.

6 Acknowledgments

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *In 8th International Symposium on High Performance Distributed Computing (HPDC-8 '99)*. IEEE CS Press, August 1999.

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

[3] L. Alvisi and K. Marzullo. Message logging : Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*, pages 229–236. IEEE CS Press, May-June 1995.

[4] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, 1995.

[5] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fédak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *SC2002: High Performance Networking and Computing (SC2002)*, Baltimore USA, November 2002. IEEE/ACM.

[6] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *to appear in High Performance Networking and Computing (SC2003)*. Phoenix USA, IEEE/ACM, November 2003.

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

[8] S.-E. Choi and S. J. Deitz. Compiler support for automatic checkpointing. In *16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS 2002)*, page 213, Canada, June 2002. IEEE.

[9] Elnozahy, Elmootazbellah, and Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers*, 41(5), May 1992.

[10] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[11] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desaia nd Ronald G. Minnich, Craig E Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *International Conference on Supercomputing(ICS'02)*, pages 77–83, New York City, NY, USA, June 2002. ACM.

[12] William Gropp and Ewing Lusk. Fault tolerance in MPI programs. *special issue of the Journal High Performance Computing Applications (IJHPCA)*, 2002.

[13] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

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

[15] James S. Planck and Michael G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and distributed Computing*, 2001.

[16] James S. Plank and Wael R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *28th Symposium on Fault-Tolerant Computing (FTCS'98)*, pages 48–57. IEEE CS Press, June 1998.

[17] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. The cost of recovery in message logging protocols. In *17th Symposium on Reliable Distributed Systems (SRDS)*, pages 10–18. IEEE CS Press, October 1998.

[18] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*, pages 48–55. IEEE CS Press, 1999.

[19] Georg Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, April 1996. IEEE CS Press.

[20] E. Strom and S. Yemini. Optimistic recovery in distributed systems. In *Transactions on Computer Systems*, volume 3(3), pages 204–226. ACM, August 1985.

[21] Kenneth F. Wong and Mark A. Franklin. Distributed computing systems and checkpointing. In *2nd Int. Symp. on High Performance Distributed Computing (HPDC'93)*, pages 224–233. IEEE CS Press, July 1993.