MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes

George Bosilca, Aurelien Bouteiller, Franck Cappello,
Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault,
Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette,
Vincent Neri, Anton Selikhov
LRI, Université de Paris Sud, Orsay, France

Abstract

Global Computing platforms, large scale clusters and future TeraGRID systems gather thousands of nodes for computing parallel scientific applications. At this scale, node failures or disconnections are frequent events. This Volatility reduces the MTBF of the whole system in the range of hours or minutes.

We present MPICH-V, an automatic Volatility tolerant MPI environment based on uncoordinated checkpoint/ rollback and distributed message logging. MPICH-V architecture relies on Channel Memories, Checkpoint servers and theoretically proven protocols to execute existing or new, SPMD and Master-Worker MPI applications on volatile nodes.

To evaluate its capabilities, we run MPICH-V within a framework for which the number of nodes, Channels Memories and Checkpoint Servers can be completely configured as well as the node Volatility. We present a detailed performance evaluation of every component of MPICH-V and its global performance for non-trivial parallel applications. Experimental results demonstrate good scalability and high tolerance to node volatility.

1 Introduction

A current trend in Technical Computing is development of Large Scale Parallel and Distributed Systems (LSPDS) gathering thousands of processors. Such platforms are the result of construction of a single machine or the clustering of loosely coupled computers that may belong to geographically distributed computing sites. TeraScale computers like the ASCI machines in US, the Tera machine in France, large scale PC clusters, large scale LANs of PC used as clusters, future GRID infrastructures (such as the US and Japan TeraGRID), large scale virtual PC farms built by clustering PCs of several sites, large scale distributed systems like Global and Peer-to-Peer computing systems are examples of this trend.

For systems gathering thousands of nodes, node failures or disconnections are not rare, but frequent events. For Large Scale Machine like the ASCI-Q machine, the MTBF (Mean Time Between Failure) for the full system is estimated to few hours. The Google Cluster using about 8000 nodes experiences a node failure rate of 2-3% per year [6]. This can be translated to a node failure every 36 hours. A recent study of the availability of desktop machines within a large industry network ($>$ 64,000 machines) [5], which is a typical Large Scale Virtual PC Farms targeted for Global Computing platforms in industry and university, demonstrates that from 5% to 10% of the machines become unreachable in a 24 hour period. Moreover a life time evaluation states that 1/3 of the machines disappeared (the connection characteristics of the machine changed) in a 100 days time period. The situation is even worse for Internet Global and Peer-to-Peer Computing platforms relying on cycle stealing for resource exploitation. In such environment, nodes are expected to stay connected (reachable) less than 1 hour per connection.

A large portion of the applications executed on large scale clusters, TeraScale machines and envisioned for TeraGRID systems are parallel applications using MPI as message passing environment. This is not yet the case for Large Scale Distributed Systems. Their current application scope considers only bag of tasks, master-worker applications, or document exchanging (instant messaging, music and movie files). However many academic and industry users would like to execute parallel applications with communication between tasks on Global and Peer-to-Peer platforms. The current lack of message passing libraries allowing the execution of parallel applications is one limitation to a wider distribution of these technologies.

For parallel and distributed systems, the two main sources of failure/disconnection are the nodes and the network. Human factors (machine/application shutdown, network disconnection), hardware or software faults may also

be at the origin of failures/disconnections. For a sake of simplicity, we consider the failures/disconnections as node volatility: the node is no more reachable and the eventual results computed by this node after the disconnection will not be considered in the case of a later reconnection. The last statement is reasonable for message passing parallel applications since a volatile node will not be able to contribute any more (or for a period of time) to the application and, further more, it may stall (or slowdown) the other nodes exchanging messages with it. Volatility can affect individual or group of nodes when a network partition failure occurs within a single parallel computer, a large scale virtual PC farm or an Internet Global Computing platform.

The cornerstone for executing a large set of existing parallel applications on Large Scale Parallel and Distributed Systems is a scalable fault tolerant MPI environment for Volatile nodes. Building such message passing environment means providing solution for several issues:

Volatility tolerance It relies on redundancy and/or checkpoint/restart concepts. Redundancy implies classical techniques such as task cloning, voting, and consensus. Checkpoint/restart should consider moving task contexts and restarting them on available nodes (i.e., task migrations) since lost nodes may not come back before a long time.

Highly distributed & asynchronous checkpoint and restart protocol Designing a volatility tolerant message passing environment and considering thousands of nodes, it is necessary to build a distributed architecture. Moreover, a checkpoint/restart based volatility tolerant system should not rely on global synchronization because 1) it would considerably increase the overhead and 2) some nodes may leave the system during synchronization.

Inter-administration domain communications Harnessing computing resources belonging to different administration domains implies dealing with fire-walls. When the system gathers a relatively small number of sites like for most of the currently envisioned GRID deployments, security tool sets like GSI [13] can be used to allow communications between different administration domains. Global and Peer-to-Peer Computing systems generally use a more dynamic approach because they gather a very large number of resources for which the security issue cannot be discussed individually. Thus, many P2P systems overcomes fire-walls by using their asymmetric protection set-up. Usually, fire-walls are configured to stop incoming requests and accept incoming replies to outgoing requests. When client and server nodes are both fire-walled, a non protected resource implements a relay between them which works as a post office where communicating nodes deposit and collect messages.

Non named receptions The last difficulty is in message receptions with no sender identity. Some MPI low level control messages as well as the user level API may allow such receptions. For checkpoint/restart fault tolerance approaches, the difficulty comes from two points: node volatility and scalability. For the execution correctness, internal task events and task communications of restarted tasks should be replayed in a consistent way according to previous executions on lost nodes. The scalability issue comes from the unknown sender identity in non named receptions. A mechanism should be designed to prevent the need for the receiver to contact every other nodes of the system.

In this paper we present MPICH-V, a distributed, asynchronous automatic fault tolerant MPI implementation designed for large scale clusters, Global and Peer-to-Peer Computing platforms. MPICH-V solves the above four issues using an original design based on uncoordinated checkpoint and distributed message logging. The design of MPICH-V considers additional requirements related to standards and ease of use: 1) it should be designed from a widely distributed MPI standard implementation to ensure wide acceptance, large distribution, portability and take benefit of the implementation improvements and 2) a typical user should be able to execute an existing MPI application on top of volunteer personal computers connected to Internet. This set of requirements involves several constraints: a) running an MPI application without modification, b) ensuring transparent fault tolerance for users, c) keeping the hosting MPI implementation unmodified, d) tolerating N simultaneous faults (N being the number of MPI processes involved in the application), e) bypassing fire-walls, f) featuring scalable infrastructure and g) involving only user level libraries.

The second section of the paper presents a survey of fault-tolerance message passing environments for parallel computing and show how our work differs. Section 3 presents the architecture, and overviews every component. Section 4 evaluates the performance of MPICH-V and its components.

2 Related Work

A message passing environment for Global and Peer-to-Peer computing involves techniques related to Grid computing (enabling nodes belonging to different administration domains to collaborate for parallel jobs) and fault tolerance (enabling the parallel execution to continue despite the node volatility).

Many efforts have been conducted to provide MPI environments for running MPI applications across multiple parallel computers: MPICH-G* [12], IMPI, MetaMPI, MPI-connect [11], PACX-MPI [14], StaMPI. They essentially

consist in layering MPI on top of vendor MPI, redirecting MPI calls to the appropriate library and using different techniques (routing processes, tunnels, PVM) to provide high level management and inter operability between libraries. Compared to GRID enabled MPI, MPICH-V provides fault tolerance.

Several research studies propose fault tolerance mechanisms for message passing environments. The different solutions can be classified according to two criteria : 1) the level in the software stack where the fault-tolerance mechanism lies and 2) the technique used to tolerate faults.

We can distinguish between three main levels in the context of MPI:

- at the upper level, an user environment can ensure fault-tolerance for a MPI application by re-launching the application from a previous coherent snapshot,

- at a lower level, MPI can be enriched to inform applications that a fault occurred and let the applications take corrective actions,

- at the lowest level, the communication layer on which MPI is build can be fault-tolerant, inducing a transparent fault-tolerance property for MPI applications.

Only the upper and the lowest levels approaches provide automatic fault tolerance for MPI applications, transparent for the user. Others approaches require the user to manage fault tolerance by adapting its application.

As discussed in [8, 21], many techniques of fault tolerance have been proposed in previous works, lying in various level of the software stacks : one solution is based on global coherent checkpoints, the MPI application being restarted to the last coherent checkpoint in case of faults (even in case of a single crash). Other solutions use a log technique that consists in saving the history of events occurring to processes, in order to re-execute the part of the execution located between the last checkpoint of the faulty processes and the failure moment. This assumes that every non-deterministic event of the application can be logged. This is generally the case for MPI applications. Three main techniques of logging are defined :

- optimistic log (in which we can include the sender-based techniques) assumes that messages are logged, but that part of the log can be lost when a fault occurs. Either this technique uses a global coherent checkpoint to rollback the entire application when too much information has been lost, or they assume a small number of faults (mostly one) at one time in the system;

- causal log is an optimistic log, checking and building an event dependence graph to ensure that potential incoherence in the checkpoint will not appear;

- pessimistic log is a transaction logging ensuring that no incoherent state can be reached starting from a local checkpoint of processes, even with an unbounded number of faults.

In Figure 1, we present a classification of well known MPI implementations providing fault-tolerance according to the technique used and the level in the software stack. MPICH-V uses pessimistic log to tolerate an unbounded (but finite) number of faults in a distributed way (there is no centralized server to log messages for all nodes).

2.1 Checkpoint based methods

Cocheck [22] is an independent application making a MPI parallel application fault tolerant. It is implemented at the runtime level (but its implementation on top of tuMPI required some modification of the tuMPI code), on top of a message passing library and a portable process checkpoint mechanism. Cocheck coordinates the application processes checkpoints and flushes the communication channels of the target applications using a Chandy-Lamport's algorithm. The checkpoint and rollback procedures are managed by a centralized coordinator.

Starfish [1] is close to Cocheck providing failure detection and recovery at the runtime level for dynamic and static MPI-2 programs. Compared to Cocheck, it provides a user level API allowing the user to control checkpoint and recovery. User can choose between coordinated and uncoordinated (for trivial parallel applications) checkpoints strategies. Coordinated checkpoint relies on the Chandy-Lamport's algorithm. For an uncoordinated checkpoint, the environment sends to all surviving processes a notification of the failure. The application may take decision and corrective operations to continue execution (i.e. adapts the data sets repartition and work distribution).

Clip [7] is a user level coordinated checkpoint library dedicated to IntelParagon systems. This library can be linked to MPI codes to provide semi-transparent checkpoint. The user add checkpoint calls in his code but does not need to manage the program state on restart.

2.2 Optimistic log

A theoretical protocol [23] presents the basic aspect of optimistic recovery. It was first designed for clusters, partitioned into a fixed number of *recovery units (RU)*, each one considered as a computation node. Each *RU* has a message log vector storing all messages received form the other *RU*. Asynchronously, an *RU* saves its log vector to a stable storage. It can checkpoint its state too. When a failure occurs, it tries to replay input messages stored in its reliable storage from its last checkpoint; messages sent since last check point are lost. If the *RU* fails to recover a coherent state,
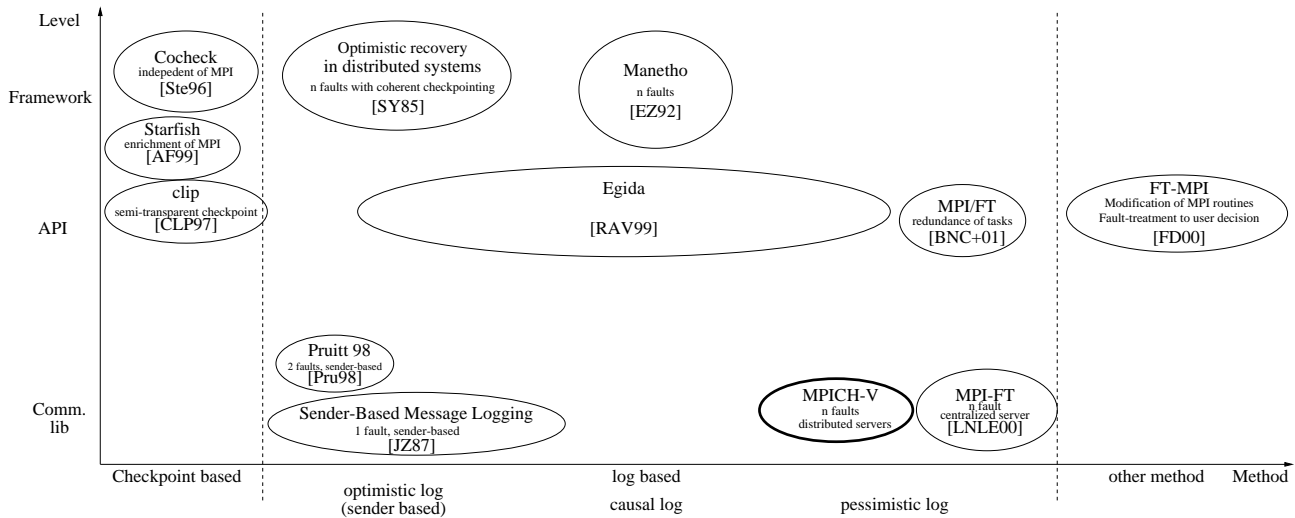
Figure 1. Classification by techniques and level in the software stack of fault-tolerance message passing systems

other *RU* concerned by lost messages should be rolled back too until the global system reaches a coherent state.

Sender based message logging [17] is an optimistic algorithm tolerating one failure. Compared to [23], this algorithm consists in logging each message in the volatile memory of the sender. Every communication requires 3-steps: send; acknowledge + *receipt count*; acknowledge of the acknowledge. When a failure occurs, the failed process rolls back to its last checkpoint. Then it broadcasts requests for retrieving initial execution messages and replays them in the order of the *receipt count*.

[20] presents an asynchronous checkpoint and rollback facility for distributed computations. It is an implantation of the sender based protocol proposed by Juang and Venkatesan [18]. This implementation is built from MPICH.

2.3 Causal Log

The basic idea of Manetho [9] is to enhance the sender based protocol with an Antecedence Graph (AG) recording the 'happen before' Lamport's relationship. All processes maintained an AG in volatile memory, in addition to the logging of sent messages. AGs are asynchronously sent to a stable storage with checkpoints and message logs. Each time a communication occurs, an AG is piggybacked, so the receiver can construct its own AG. When failures occur, all processes are rolled back to their last checkpoints, retrieving saved AG and message logs. The different AGs are used by receivers to retrieve the next lost communications and to wait for the sender processes to re-execute their sending.

2.4 Pessimistic log

MPIFT [4] implements failure detection at the MPI level and recovery at the runtime level re-executing the whole MPI run (possibility from a checkpoint). Several sensors are used for fault detection at the application, MPI, network and operating system levels. A central coordinator monitors the application progress, logs the messages, restarts the failed processes from checkpoints and manages control messages for self-checking tasks. MPIFT considers coordinator as well as task redundancy providing fault tolerance but requires to implement voting techniques to detect failures. A drawback of this approach is the central coordinator which, according to the authors, scale only up to 10 processors.

MPI-FT [19] is the closest to MPICH-V but built from LAM-MPI. It uses a monitoring process (called the observer) for providing MPI level process failure detection and recovery mechanism (based on message logging). To detect failures, the observer assisted by a Unix script periodically checks the existence of all the processes. Two message logging strategies are proposed: an optimistic one, decentralized, requiring every MPI process to store its message traffic, or a pessimistic one, centralized, logging all messages in the observer. For recovery, the observer spawns new processes which involves a significant modifications of the MPI communicator and collective operations. MPI-FT restarts failed MPI processes from the beginning which has two main consequences: 1) a very high cost for failure recovery (the whole MPI process should be replayed) and 2) the requirement of a huge storage capacity to store the mes-

sage traffic.

2.5 Other methods

FT-MPI [10] handles failure at the MPI communicator level and lets the application to manage the recovery. When a fault is detected, all the processes of a communicator are informed about the fault (message or node). This information is transmitted to the application by the returning value of MPI calls. The application can take decisions and execute corrective operations (shrinking, rebuilding, aborting the communicator) according to various fault states. The main advantage of FT-MPI is its performance since it does not checkpoint neither MPI processes nor log MPI messages. Its main drawback is the lack of transparency for the programmer.

Egida [21] is an object oriented toolkit supporting transparent log-based rollback-recovery. By integrating Egida with MPICH, existing MPI applications can take advantage of fault-tolerance transparently without any modification. Egida implements failure detection and recovery at the low-level layer. The Egida object modules replaces P4 for the implementation of MPICH send/receive messages, but Egida modules still use P4 as the actual communication layer. A dedicated language allows to express different protocols from events themselves (failure detection, checkpoint, etc.) to responses to these events.

3 MPICH-V architecture

MPICH-V environment encompasses a communication library based on MPICH [16] and a runtime environment. The MPICH-V library can be linked with any existing MPI program as usual MPI libraries.

The library implements all communication subroutines provided by MPICH. Its design is a layered architecture: the peculiarities of the underlying communication facilities are encapsulated in a software layer called a *device*, from which all the MPI functions are automatically built by the MPICH compilation system. The MPICH-V library is build on top of a dedicated device ensuring a full-fledged MPICH v.1.2.3., implementing the Chameleon-level communication functions. The underlying communication layer relies on TCP for ensuring message integrity.

MPICH-V runtime is a complex environment involving several entities: Dispatcher, Channel memories, Checkpoint servers, and computing/communicating nodes (Figure 2).

MPICH-V relies on the concept of Channel Memory (CM) to ensure fault tolerance and to allow the firewall bypass. CMs are dedicated nodes providing a service of message tunneling and repository. The architecture does not assume central control nor global snapshot. Fault tolerance is implemented in a highly decentralized and asynchronous way, saving the computations and communication contexts independently. For each node, the execution context is saved (periodically or upon a signal reception) on remote checkpoint servers. A communication context is stored during execution by saving all in-transit messages in CMs. Thus, the whole context of a parallel execution is saved and stored in a distributed way. The MPICH-V runtime assigns CM to nodes by associating each CM to different sets of receivers (Figure 3). Following this rule, a given receiver always receives its messages from the same CM, called its *home CM*. When a node sends a message, it actually puts the message in the receiver *home CM*. MPICH-V runtime provides all nodes with a map of the CM-receiver association during the initialization.

This association scheme is effective for 1) keeping the protocol simple and provable and 2) limiting the number of messages for implementing non-named receptions (receive from any). In one extreme a), there is a single CM for all nodes; numbering all communications managed by the CM provides a total order over the communication events. In such a case, it is easy to prove that the restart protocol is deadlock free. In the other extreme b), each node has it own CM. In that case, numbering all communications for a dedicated receiver provides a total order over the communication events for that receiver.

The difference between situations a) and b) is the lack of a total order over the communications events of different receivers in the situation b). Since there is no ordering dependency between the receptions made by different receivers, it is also easy to prove that the restart protocol is deadlock free. In addition, to implement non-named receptions, the receiver only needs to contact its associated CM.

3.1 Dispatcher

The Dispatcher manages tasks that are instances of traditional MPI processes. During the initialization, the dispatcher launches a set of tasks on participating nodes, each task instantiating a MPI process.

A key task of the dispatcher is the coordination of the resources needed to execute a parallel program. This includes 1) providing nodes for executing services (Channel Memories and Checkpoint Servers) and MPI processes and 2) connecting these running services together. All these services are scheduled as regular tasks, and therefore assigned dynamically to nodes when they are ready to participate. When scheduled, a service is registered in a Stable Service Registry, managed in a centralized way. It records, for each service, the set of hosts providing this service and the protocol information used to bind a service requester to a service host. When a node executes an MPI application, it first contacts the Service Registry to resolve the address of a Checkpoint Server and its related Channel Memory according to
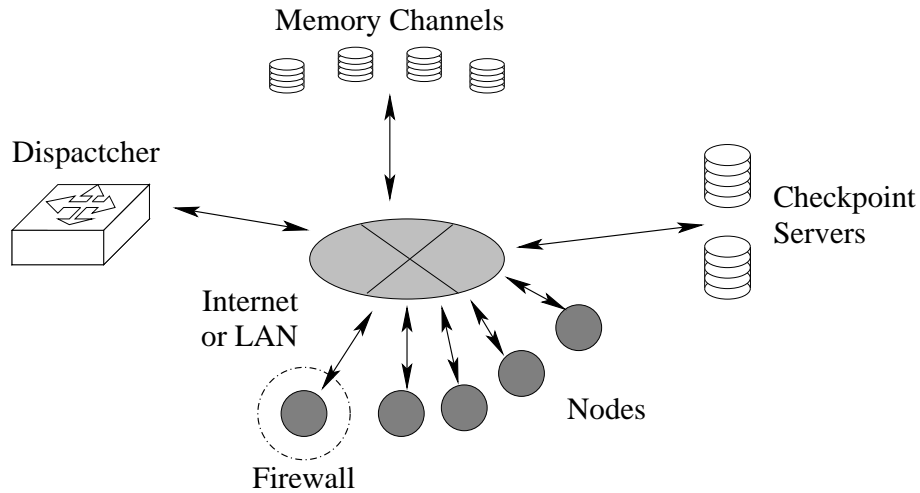
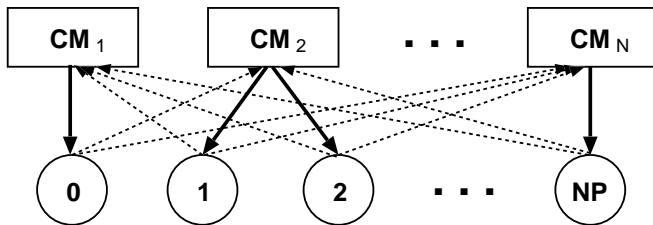**Figure 2. Global architecture of MPICH-V**

Figure 3. Structure of connections between CMs and nodes. Bold pointers correspond to connections with home CM servers to receive incoming messages, whereas dashed pointers correspond to connections with other CMs to send messages

the node rank in the MPI communication group. Checkpoint servers and Channel memories are globally assigned to the nodes in a round robin fashion at the creation time of a parallel execution.

During execution, every participating node sends periodically an "alive" signal to the dispatcher. The "alive" signal only contains the MPI rank number. Its frequency, currently set to 1 minute, is adjusted considering the tread-off between dispatcher reactiveness and communication congestion. The dispatcher monitors the participating nodes considering the "alive" signal; it detects a potential failure when an "alive" signal is timed out, whatever is the reason: either a node or a network failure (the dispatcher makes no difference between those two error origins). It then launches another task (a new instance of the same MPI process). The task restarts the execution, reaches the point of failure and continues the execution from this point. The other nodes are not aware about the failure. If the faulty node reconnect the system, two instances of the same MPI process (two clones) could run at the same time in the system. The Channel Memory manages this situation by allowing only a single connection per MPI rank. When a node leave the system (disconnection or failure), it stops its permanent connection to the Channel Memories. When a node tries to connect a Channel Memory, the Channel Memory checks the MPI rank and returns an error code to the node if the rank is already associated with an open connection. When a node receives an error code for a connection tentative, it kills the current job and re-contacts the dispatcher for fetching a new job. So only the first node among the clones will be able to open a connection with the Channel Memory.

3.2 Channel Memory Architecture

The Channel Memory (CM) is a stable repository for communicating nodes messages. Its main purpose is to save the communications during execution in order to tolerate tasks volatility. Every node contacts all CM servers associated with the MPI application according to the Service Registry provided by the Dispatcher. To send (receive) a message to (from) another node, communicating nodes deposit or retrieve MPI messages via CM transactions. Every communication between the nodes and CMs are initiated by the nodes.

The CM is a multi-threaded server with a fixed pool of threads, which both listen to the connections and handle the requests from the nodes, the checkpoint servers and the dispatcher. To ensure a total order of the messages to be received, the CM manages all the messages as a set of FIFO queues.

In the CM, each receiver is associated with dedicated queues. To be compliant with higher-level MPICH func-

tions, the CM manages two types of queues: a common queue for all control messages, and one queue per sender for data messages. Such architecture of the data storage allows to minimize the time to retrieve any message.

All the message queues are stored in virtual memory area with out-of-core features. When the queues size exceeds the physical RAM, part of the queues are swapped to the disc following a LRU policy.

When a node leaves the system, the Dispatcher orders another node to restart the task from the beginning or from the last checkpoint. Because some communications may be performed from the beginning or from the last checkpoint and before the failure, some communications may be replayed by the restarting task. The CM handles communications of a restarting task by: a) sending the number of messages sent by the failed execution and b) re-sending the message being already sent to the failed task. More detailed description of both Channel Memory and the *ch_cm* device (outlined in the following section) architecture design and functionality, is described in [2]

3.3 Channel Memory Device

The MPICH-V channel memory device implements MPI library based on MPICH for a participating node. The device, called *ch_cm*, is embedded in the standard MPICH distribution and is implemented on the top of TCP protocol.

Functionality of the device supports initialization, communication and finalizing functions of the higher levels according to the features of the MPICH-V system. At the initialization time, when a parallel application has been dispatched, the device provides connection of the application with CM servers and Checkpointing servers. The connections established with CM servers during this phase are TCP sockets opened until the application finishes all its communications.

The implementation of all communication functions includes sending and receiving of special system messages, used for preparation of the main communication part and for identifying the type of communication both for node and for CM server part.

Application finalization includes disconnection notifications to all contacted CMs.

In addition to the *ch_cm* device properties, described above, its important difference from the usual *ch_p4* device is the absence of local message queues. All the queues are served by Channel Memory as it was described above and the device retrieves the required message directly, according to the information about the source/destination and an internal MPI tag.

3.4 Checkpoint System

Checkpoint Servers (CS) stores and provides task images on-demand upon node requests. MPICH-V assumes that: 1) CS are stable and not protected by fire-walls, 2) communication between nodes and CS are transactions, 3) every communication between the nodes and CSs are at the initiative of the nodes.

During execution, every node performs checkpoints and sends them to a CS. When a node restarts the execution of a task, the Dispatcher informs it about: 1) the task to be restarted and 2) which CS to contact to get the last task checkpoint. The node issues a request to the relevant CS using the MPI rank and application id. The CS responds to the request by sending back the last checkpoint corresponding to the MPI rank. Thus, this feature provides a task migration facility between nodes.

Checkpoint can be performed periodically or upon reception of an external signal. Currently, the decision of initiating a checkpoint is taken periodically on local node, without any central coordination.

We currently use the Condor Stand-alone Checkpointing Library (CSCL), which checkpoints the memory part of a process and provides optional compression of the process image. CSCL does not provide any mechanism to handle communications between nodes. In particular, sockets that were used by a checkpointed process are not re-opened after a restart. In order to workaround this limitation, when a process is restarted, MPICH-V restores the connections with CMs before resuming execution.

To permit overlap between checkpoint calculation and network transfer of the process image, the checkpoint image is never stored on the local node disk. Generation and compression of the checkpoint image, and transmission to the checkpoint server are done in two separate processes. Size of the image is not precomputed before sending data, so we begin transmission as soon as CSCL library begins to generate checkpoint data. The transmission protocol relies on the underlying network implementation (TCP/IP) to be able to detect errors.

When restarting a process from its checkpoint image, messages are not replayed from the beginning. So the CS and CM associated with a task must be coordinated to ensure coherent restarts. The last message before the beginning of a checkpoint is a special message describing from which message the replay should start. In order to ensure transactional behavior of the checkpointing mechanism, this special message is acknowledged by the CS when the transfer of the checkpoint image is completed.

The process of checkpointing a program requires to stop its execution. However, generating, compressing and sending a checkpoint image could take a long time. To reduce the stall time, MPICH-V actually checkpoints a clone of

the running task. The process clone is created using standard *fork* unix process creation system call. This system call duplicates all memory structures and process flags; file descriptor's states are inherited as well as opened sockets. Since this process has the same data structures and is in the same state as the original one, we use it to perform a checkpoint while the original process can continue its execution.

3.5 Theoretical foundation of the MPICH-V Protocol

The MPICH-V protocol designed relies on uncoordinated checkpoint/rollback and distributed messages logging. According to our knowledge, such fault tolerant protocol has not been proved yet. To ensure the correctness of our design, we have formally proved the protocol. The following paragraphs sketch the theoretical verification (the complete proof is out of scope of this paper; it will be published separately later).

Model We use well known concepts of distributed system: *processes* and *channels*, defined as deterministic finite-state machines, and *a distributed system* as processes collection linked together with channels. Channels are FIFO, unbounded and lossless. An *action* is either a communication routine or an internal state modification. A *configuration* is a vector of the states of every components (processes and channels) of the system. An *atomic step* is the achievement of an action on one process of the system. Finally, an *execution* is an alternate sequence of configurations and atomic steps, each configuration being obtained by applying the action of atomic step to the preceding configuration.

We define a *fault* as a specific atomic step where one process changes its state back to a known previous one. This defines the crash, recovery and rollback to a checkpointed state. Any previous state is reachable, so if no checkpoint is performed, we simply reach back the initial state (recovery). This model also states that the checkpoints are not necessary synchronized, since for two different faults, the two targeted processes may rollback to unrelated previous states.

A *canonical execution* is an execution without faults. We define an *equivalence relationship* between executions, such that if two executions lead to the same configuration, they are equivalent.

Definition 1 (fault-tolerance to crash with recovery and rollback to non-synchronized checkpoints) *Let $\mathcal{E}$ be a canonical execution, for any configuration C of $\mathcal{E}$, consider the sequence $C, A_0, C_0, \ldots, C_f$ where A_i are atomic steps of the system or faults. A protocol is fault-tolerant if the execution starting from C_f is equivalent to a canonical execution.*

Definition 2 (pessimistic-log protocols) *Let S be a distributed system of processes $\mathcal{P}$, and p be a process. Let $\mathcal{E}$ be an execution of S and C and C' be two configurations of $\mathcal{E}$ (C before C'). Let f_p be a fault action of p, and C_f a configuration such that f_p leads to C_f from C' and state of p in C_f is equal to its state in C. Let $\mathcal{E}_f$ be an execution starting from C_f without faults for p. A protocol is a pessimistic-log protocol if and only if there is a configuration C'_f such that*

logging *projection on p of $\mathcal{E}$ between C and C' is equal to projection on p of $\mathcal{E}_f$ between C_f and C'_f;*

silent re-execution *projection on $\mathcal{P}\backslash p$ of $\mathcal{E}_f$ between C_f and C'_f does not hold actions related to p.*

Theorem 1 *Any protocol verifying the two properties (logging and silent re-execution) of pessimistic-log protocols is fault-tolerant to crash with recovery and rollback to not-synchronized checkpoints.*

Key points of the proof: In this paper we only present the key-points of the proof by constructing a canonical execution starting from C_f. We consider first every event non-related to crashed processes of $\mathcal{E}_f$ between C_f and C'_f, and we prove that this event is a possible event of a canonical execution. Then, we consider every events related to crashed processes that are not re-execution events (therefore after recover), and we state that second property of pessimistic log protocols ensure that these events can happen, but will happen after the crashed process reaches its state in C'_f. Lastly, first property of pessimistic-log protocols induces that crashed processes reaches this state. We conclude that the system reaches a configuration which is member of a canonical execution, thus that $\mathcal{E}_f$ is equivalent to a canonical execution. ∎

Therefore, we have proven that the two properties of pessimistic-log protocols were sufficient to tolerate faults for any asynchronous distributed system. In order to replay the events, we log them on reliable media. Since we are designing a scalable pessimistic-log protocol, the reliable media has to be distributed. We present now the algorithm used in MPICH-V with the topology we introduced in preceding section, and we have proven in the preceding model that this protocol verify the two pessimistic-log protocol properties.

Theorem 2 *The MPICH-V protocol of figure 4 is a pessimistic-log protocol.*

Key points of the proof: We proove that MPICH-V protocol verifies the two properties of pessimistic-log protocols

a) MPICH-V has the logging property Logging property states that events between the recovery point and the crash are logged and that the order of events is also logged. Non-deterministic events are receptions and probes. They are all logged by one single CM and the order is kept in the

Algorithm of Worker
On Recv/Probe $\longrightarrow$
    send Recv/Probe request to CM
    recv answer of CM
    deliver answer
On Send m to W' $\longrightarrow$
    if(nbmessage > limit)
        send m to Channel Memory of W'
    nbmessage $\leftarrow$ nbmessage+1
On Begin Checkpoint $\longrightarrow$
    Send Begin Checkpoint to CM
    checkpoint and send checkpoint to CS
On Restart $\longrightarrow$
    send Restart to CS
    recv Checkpoint and recover
    For each Channel Memory C,
        send Restart to C
        receive nbmessage
        limit $\leftarrow$ nbmessage + limit

Algorithm of Checkpoint Server of Worker W
On Receive checkpoint from W $\longrightarrow$
    save checkpoint
    send End Checkpoint to CM
On Receive restart $\longrightarrow$
    send last successful checkpoint to W

Algorithm of Channel Memory of Worker W
On Receive non-deterministic event from W $\longrightarrow$
    if(no more events in past events)
        replay $\leftarrow$ False
    if(replay)
        lookup next event in past events
        send next event to W
    else
        choose event in possible events
        add event in queue of past events
        send event to W
On Receive a message m for W from W' $\longrightarrow$
    add m to possible events
    Inc(received messages [W'])
On Receive a Begin checkpoint from W $\longrightarrow$
    mark last event of past events as
        checkpoint
On Receive a End checkpoint from CS $\longrightarrow$
    free every events of past events
        until checkpoint marked event
On Receive a restart from W' $\neq$ W $\longrightarrow$
    send received messages [W'] to W'
On Receive a restart from W $\longrightarrow$
    send received messages [W] to W
    replay $\leftarrow$ true

Figure 4. MPICH-V Protocol

past events list. Moreover, the events are kept if they are after the last successful checkpoint, thus this protocol has the logging property

b) MPICH-V has the silent re-execution property. Events that are related to two processes are solely emissions: receptions and probing are related to the process and its memory, whereas emissions are related to the process and other memories of the system. On restart, the process contacts every channel memory of the system, and recover the number of messages already delivered to this memory. Thus, it knows the number of messages to ignore and emissions has no effects on non-crashed processes. Thus, MPICH-V is silently re-executing. ∎

4 Performance evaluation

The purpose of this section is to provide reference measurements for MPICH-V. Performance evaluation for a larger application set and for different experimental platforms will be published later.

We use XtremWeb [15] as a P2P platform for the performance evaluation. XtremWeb is a research framework for global computing and P2P experiments. XtremWeb executes parameter sweep, bag of tasks and embarrassingly parallel applications on top of a resource pool connected by Internet or within a LAN. XtremWeb architecture contains three entities: dispatcher, clients and workers. For our tests, we use a single client requesting the execution of a parallel experiment. XW-Workers execute MPICH-V computing nodes, CMs and CSs. XW-Dispatcher runs the MPICH-V Dispatcher. XtremWeb framework allows to control several parameters for every experiment: the task distribution among the XW-Workers (the number of CMs, CSs and computing nodes), the number of times each experiment should be executed.

In order to reproduce experimental conditions at will, we choose to execute our tests on a stable cluster and we simulate a volatile environment by enforcing process crashes. The platform used for the performance evaluation (ID IMAG 'Icluster') is a cluster of 216 HP e-vectra nodes (pentium III 733 MHZ, memory of 256 Mo, disc of 15 Go, with Linux 2.2.15-4 mdk or Linux 2.4.2) interconnected in 5 subsystems by HP procurve 4000 switches (1 GB). In each subsystem, 32 nodes are connected through a 100BaseT switch. Only about 130 nodes were available for our tests. All tests were compiled with the PGI Fortran compiler or GCC.

Several measurements are based on the NAS BT benchmark [3]. This code (Block Tridiagonal matrix solver) is a simulated Computational Fluid Dynamics application. We have chosen this code because 1) it is widely accepted as a parallel benchmark, 2) it features significant amount of

communications, 3) it requires large memory for the computing nodes.

We first present the performance of basic components and then the performance of the whole system.

4.1 Ping-Pong RTT

First we compare MPICH-V with MPICH over TCP (P4). Figure 5 presents the basic round-trip time measurement (mean time over 100 measurements) for each version according to the message size. We measure the communication time for blocking communications.

As expected the MPICH-V memory channel induces a factor 2 of slowdown compared to MPICH-P4 on the communication performances with its fastest version. Other fault tolerant MPI may provide better communication performance but, as it was explained in the related work section, they have some limitations in the number of tolerated faults or they rely on coordinated checkpoint. The factor 2 slowdown is mostly due to the current store and forward protocol used by the Channel Memory. One of the future work will consider a faster protocol based on pipelining the receive and send operations performed by the CM for relaying the message.

The in-core version stores messages until no more memory is available. The out-of-core version presented on this figure stores all exchanged messages using the virtual memory mechanism. This system shows slower communication times (worst) when several virtual memory page faults occur. When all virtual memory pages are in-place, the performance is similar to the in-core performance.

The next experiment investigates the impact on the communication performance when several nodes stress a CM simultaneously. The benchmark consists of an asynchronous MPI token ring ran by 8 and 12 nodes using one CM. We use from 1 to 8 threads inside the CM to manage incoming and outgoing communications. We measure the communication performance 100 times and compute the mean for each node. Figure 6 shows the improvement of the total communication time according to the number of threads on the CM.

The figure 6 shows that increasing the number of threads improves the response time whatever the number of nodes involved in the token ring. The overlap of the message management for one thread and the communication delay experienced by other threads explains this performance.

Figure 7 shows the impact of the number of nodes supported simultaneously by the CM on the communication time. We measure the communication performance 100 time, compute the mean for each node and the standard deviation across all the nodes. We only plot the mean value for each node to improve readability. For every curve we select the number of threads within the CM giving the best

performance.

Figure 7 demonstrates that the response time increases linearly with the number of node per CM. We have measured a standard deviation lower than 3% across the nodes. This relatively low standard deviation shows that the protocol is fair.

4.2 Performances of re-execution

Figure 8 shows the replay performances. The benchmark consists of an asynchronous MPI token ring ran by 8 nodes using one CM.

The origin curve (0 restart) shows the performance of the first complete run of the benchmark. Next, the benchmark is stopped before being finalized and restarted from the beginning. The curves "i restart" show the time for replaying this application simultaneously on i nodes.

Because the nodes start to replay the execution when all the messages are stored within the CM, the replay only consists in performing the receive and probe actions to the CM. The send actions are simply discarded by the senders. Thus as demonstrated on figure 8, the re-execution is faster than the original execution even when all nodes of the application restart.

As a consequence, when a node restart the execution of a failed task, the part re-executed is played quicker than the original one. This feature combined with the uncoordinated restart provides a major difference compared to the traditional Chandy Lamport algorithm where nodes replays exactly the first execution at the same speed.

4.3 Checkpoint performance

Figure 9 presents the characteristics of the checkpoint servers from the round trip time (RTT) benchmark measuring the time between the checkpoint signal reception and the actual restart of the process. This time includes a fork of the running process, the Condor checkpoint, the image compression, the network transfer toward the CS, the way back, the image decompression, and the process restart. The top figure presents the cost of remote checkpointing compared to a local one (on the node disc) for different process sizes. The considered process image sizes correspond to the image of one MPI process for different configurations/classes of the BT benchmark. The bottom figure presents RTT as experienced by every MPI process (BT.A.1) when several other processes are checkpointing simultaneously on a single CS. For this measurement, we synchronized the checkpoint/restart transactions initiated by the nodes in order to evaluate the CS capability to handle simultaneous transactions. The figure presents the mean value for up to 7 nodes and the standard deviation.
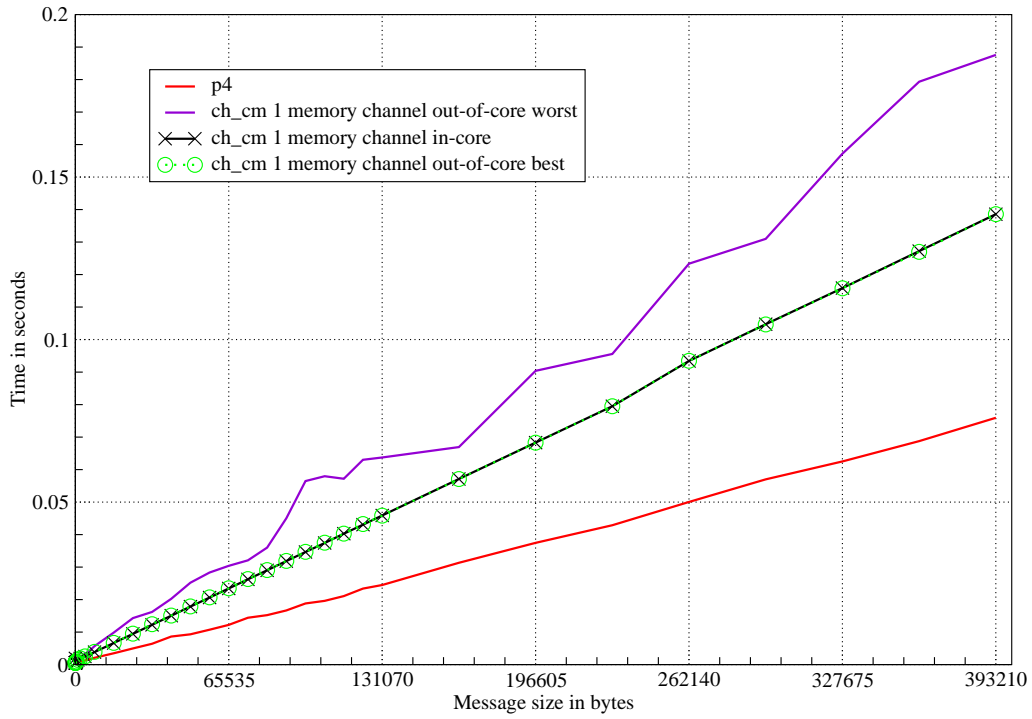
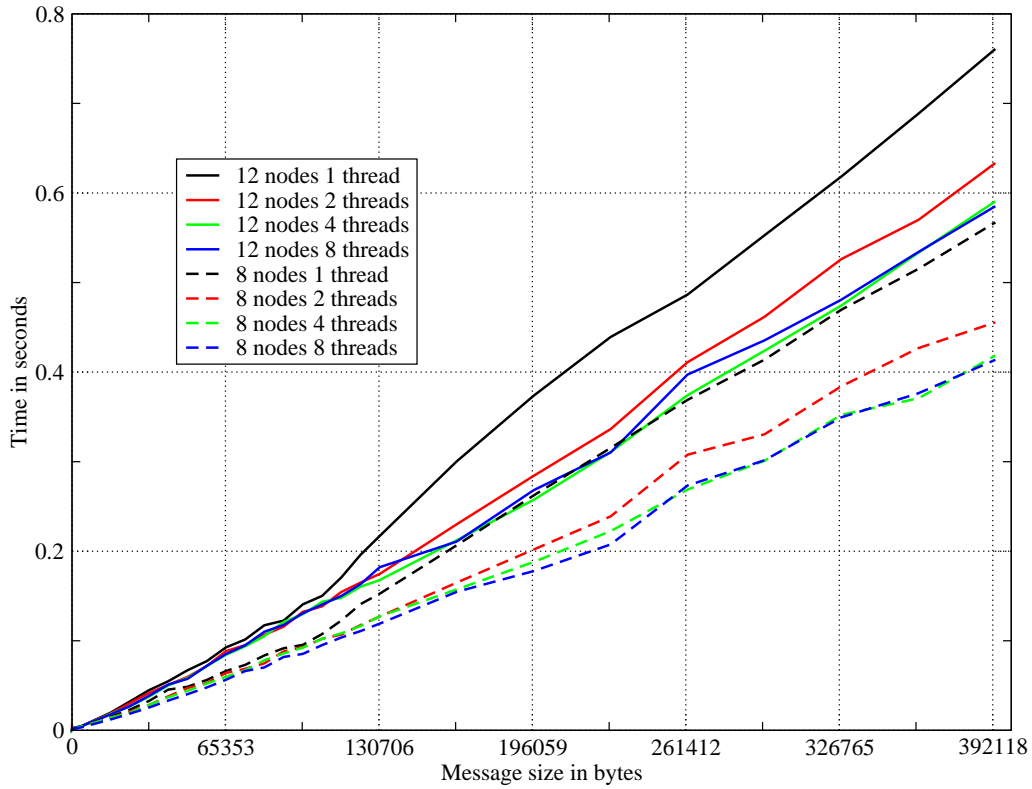**Figure 5. Round trip time of MPICH-V compared to MPICH-P4**

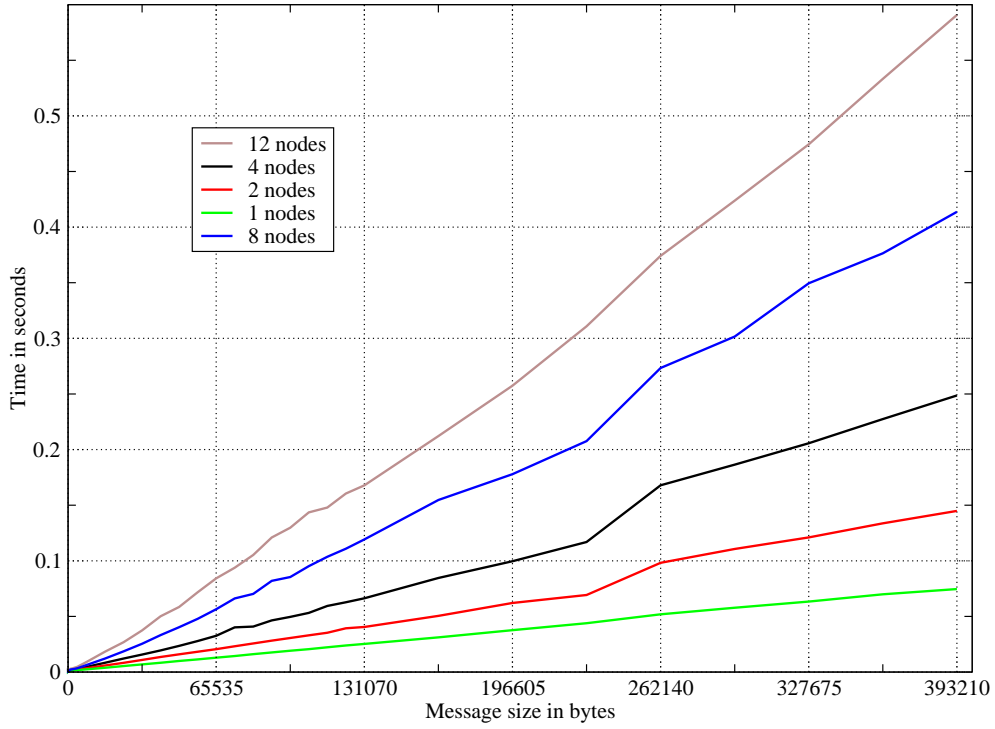**Figure 6. Impact of the number of threads running within the CM on the communication time.**

Figure 7. Impact of the number of nodes managed by the CM on the communication time.
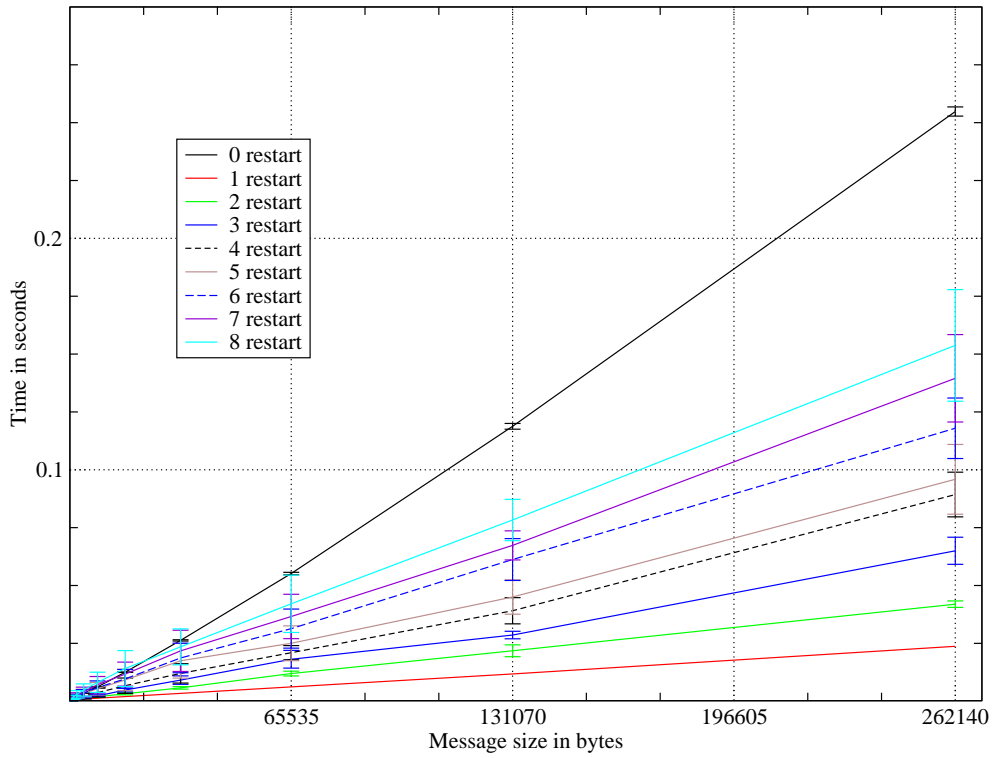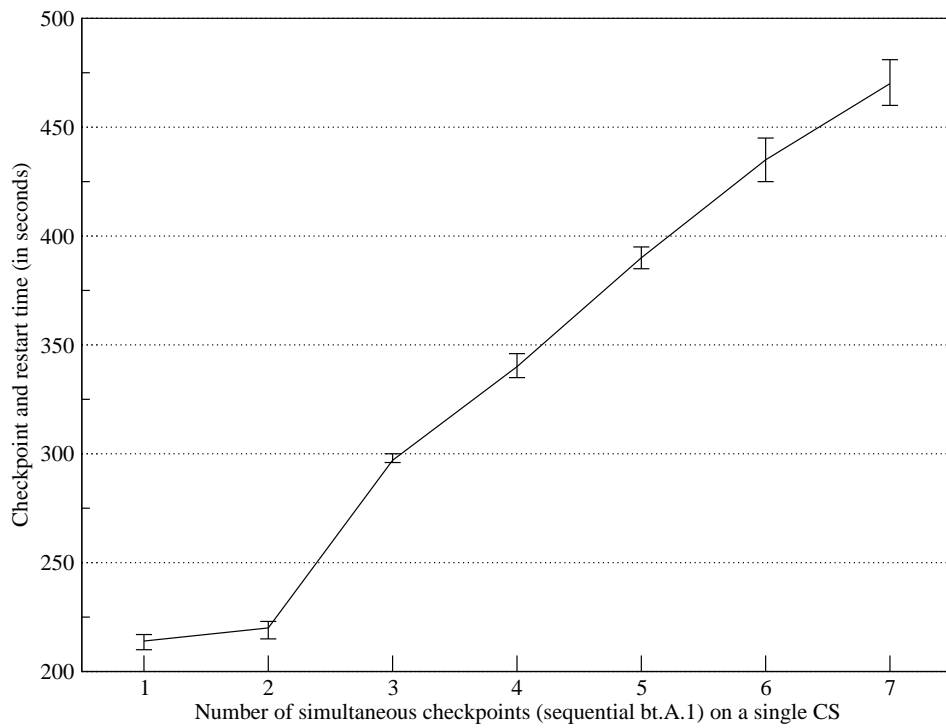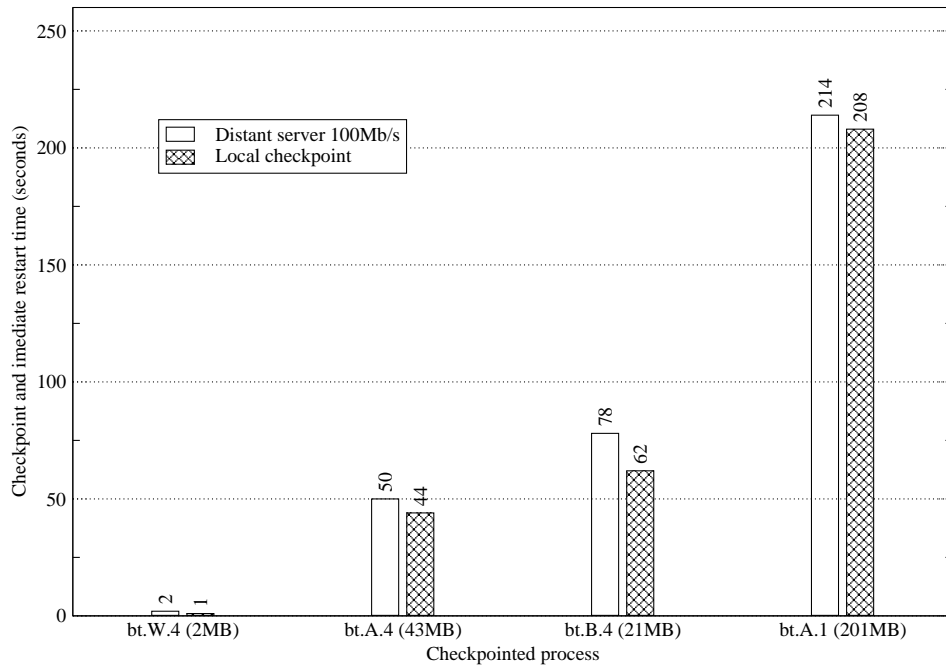

Figure 8. Performance of re-execution.

Figure 9. Performance characteristics of the checkpoint servers. The top figure presents the round trip time for local or remote checkpoints of the BT benchmark for several dataset sizes and processor numbers. The bottom figure presents the round trip time for remote checkpoint when several other processes perform simultaneously a checkpoint (of BT.A.1) on the same CS.

Figure 9 demonstrates that the cost of remote and local checkpointing are very close. This is mostly due to the overlap of the checkpoint transfer by the overhead of data compression made by the node. Network transfer time on a 100BaseT LAN is marginal. When several applications are checkpointing their processes (BT.A.1) simultaneously, the RTT increases linerly after the network saturation is reached. The standard deviations demonstrates that the CS provides a fair service for multiple nodes.

Figure 10 presents the slowdown of an application (BT class A, 4 processes), compared to an application without checkpoint, according to the number of consecutive checkpoints performed during the execution. Checkpoints are performed at random times for each process using the same CS (at most 2 checkpoints are overlapped in time).

When four checkpoints are performed per process (16 checkpoints in total), the performance is about 94% of the performance without checkpoint. This result demonstrates that 1) several MPI processes can use the same checkpoint server and 2) several consecutive checkpoints can be made during the execution with a tiny performance degradation.

4.4 Performance of the whole system

To evaluate the performance of the whole system, we conducted three experiments using the NAS BT benchmark and the MPI-PovRay applications. The three tests were intended to measure the scalability of MPICH-V, the performance degradation when faults occur and the individual contribution of every MPICH-V component to the execution time. For all tests, we compare MPICH-V to P4.

4.4.1 Scalability for the PovRay application

This experiment evaluates the scalability of MPICH-V. The benchmark is a parallelized version of the ray-tracer program Povray called MPI-PovRay. MPI-PovRay is a master worker application which exhibits a low communication/computation ratio and therefore is highly scalable. For this experiment, we used a ratio of 1 CM per 8 computing nodes. Table 11 compares the execution times to render a complex image at resolution 450x350 with MPI-Povray built on top of MPICH-P4 and MPICH-V.

# nodes	16	32	64	128
MPICH-P4	744 sec.	372 sec.	191 sec.	105 sec.
MPICH-V	757 sec.	382 sec.	190 sec.	107 sec.

Figure 11. Execution time for MPI-PovRay for MPICH-V compared to P4.

The results show an almost linear speedup up to 128 pro-

cessors for both MPICH-V and MPICH-P4. Note that the lower value for 128 nodes is due to the irregular distribution of the load to the workers. The performance of MPICH-P4 and MPICH-V are similar for this benchmark. The communication to computation ratio for this test is about 10% for 16 nodes. It explains the good performance of MPICH-V even when few CMs are used. For an application with a higher ratio, more CMs are needed per computing nodes as shown in section 4.4.3.

4.4.2 Performance with Volatile nodes

The experiment of figure 12 was intended to measure application performance when many faults occur during execution. We run the NAS BT benchmark with 9 processors and Class A. For this test, MPICH-V configuration uses three Channel Memory Servers (three nodes per CM), two Checkpoint Servers (four nodes on one CS and five nodes on the other).

During execution, every node performs and sends a checkpoint image with a 130s period. We generate faults randomly according to the following rules: 1) faults occur between two consecutive checkpoints and not during the checkpoint procedure, 2) faults occur on different nodes and no more that two faults occur during the same checkpoint phase.

The overhead of the MPICH-V for BT.A.9 is about 23% when no fault occurs during the execution due to the checkpoint cost. When faults occur, the execution time smoothly increases up to 180% of the execution time without fault.

This result demonstrates that the Volatility tolerant feature of MPICH-V based on uncoordinated checkpoint and distributed message logging is actually running and that we can execute a real parallel application efficiently on volatile nodes: MPICH-V successfully executes a MPI application on a platform where one fault occurs every 110 seconds with a performance degradation less than a factor two.

4.4.3 Performance for NAS NPB 2.3 BT

Figure 13 presents the execution time break down for the BT benchmark when using from 9 to 25 nodes. For each number of computing nodes, we compare P4 with several configurations of MPICH-V: base (using on CMs as relays for messages), w/o checkpoint (full fledged MPICH-V but checkpoint) and whole (checkpointing all nodes every 120s). For this experiment there is an equal number of CM and computing nodes for MPICH-V. There is 1 CS per set of 4 nodes.

Figure 13 demonstrates that MPICH-V compares favorably to P4 for these benchmark and platform. The execution time break down shows that the main variation comes from the communication time. The difference between MPICH-V and P4 communications times is due to the way asyn-
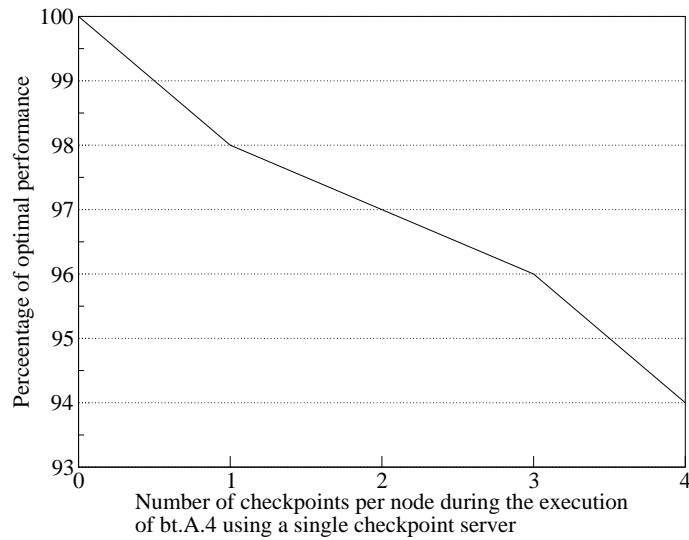
Figure 10. Slowdown (in percentage of the full performance) of an application according to the number of consecutive checkpoints. Note that for each X value, 4 processes are checkpointed (BT.A.4).
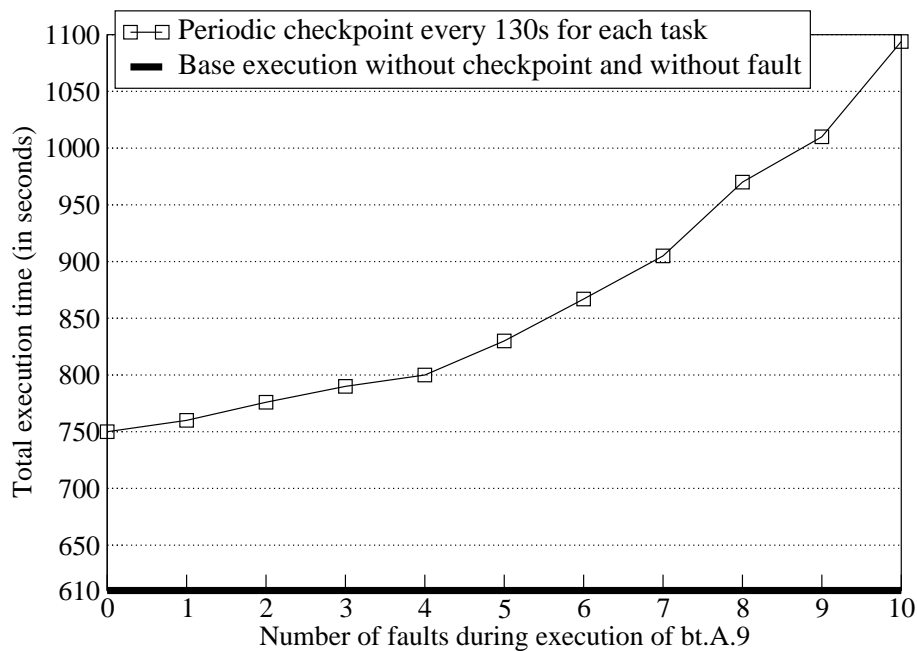

Figure 12. Execution time degradation of NAS BT with 9 processors and Class A running under MPICH-V when random faults (from 1 to 10) occur.

chronous communications are handled by the two environments. The CM and its out-of-core message storage mechanism do not increase the communication time on a real applications. Conversely, the remote checkpointing system consumes a significant part of the available network bandwidth. If we reduce the number of CMs per computing node, the performance smoothly decreases: for 49 computing nodes, the performance degradation when using 25 CMs
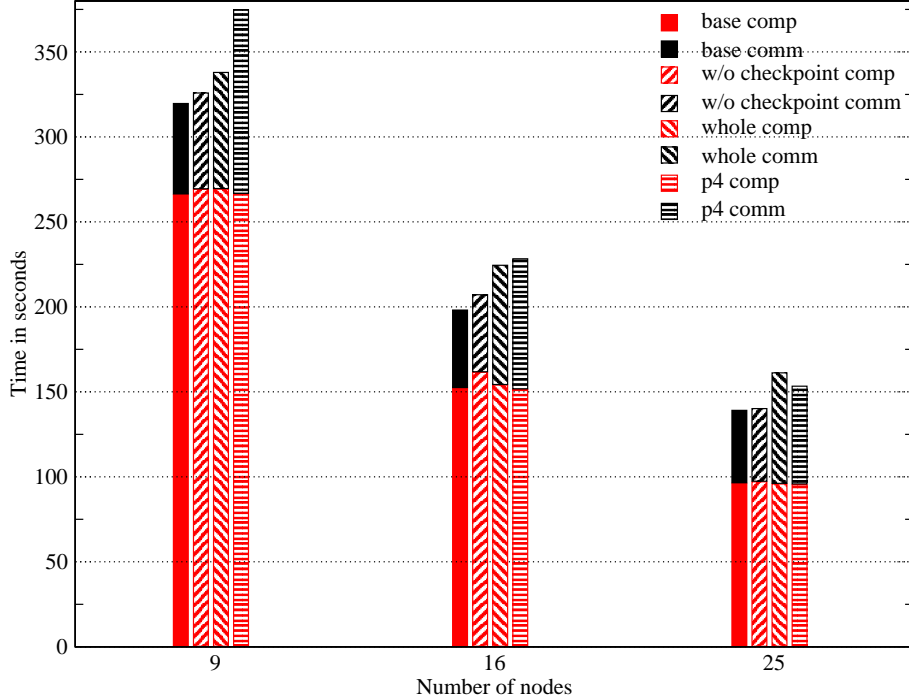
15

Figure 13. Execution time break down of MPICH-V for NAS BT compared to MPICH-P4.

instead of 49 is about 6% and about 20% when using 13 CMs instead of 49.

5 Conclusion and future work

We have presented MPICH-V, a MPI fault tolerant implementation based on uncoordinated checkpointing and distributed pessimistic message logging. MPICH-V relies on the MPICH-V runtime to provide automatic and transparent fault tolerance for parallel applications on large scale parallel and distributed systems with volatiles nodes. By reusing a standard MPI implementation and keeping it unchanged, MPICH-V allows to execute any existing MPI application and only requires to re-link them with the MPICH-V library. MPICH-V architecture gathers several concepts: Channel Memory for implementing message relay and repository, Checkpoint Servers for storing remotely the context of MPI processes.

One of the first issue when designing a fault tolerant distributed system is to prove its protocols. We have sketched the theoretical foundation of the protocol underlying MPICH-V and proven its correctness with respect to the expected properties.

We have used a Global Computing / Peer-to-Peer system (XtremWeb) as a framework for performance evaluation. The performance of the basics components: Communication Channel and Checkpoint Server have been pre-

sented and discussed. MPICH-V reaches approximatively half the performance of MPICH-P4 for the round trip time test. Stressing the CM by increasing number of nodes leads to a smooth degradation of the communication performance and a fair distribution of the bandwidth among the nodes. Checkpoint servers implement a tuned algorithm to overlap process image compression and network transfer. The result is a very low overhead for remote checkpoint relatively to local checkpoint.

We have tested the performance of the whole system using the NAS NPB2.3 BT benchmark and the MPI-PovRay parallel ray tracing program. The test using MPI-PovRay demonstrates a scalability of MPICH-V similar to the one of P4. For testing the volatility tolerance we ran BT on a platform where one fault occurs every 110 seconds. MPICH-V successively executed the application on 9 nodes with a performance degradation less than a factor 2. This result demonstrates the effectiveness of the uncoordinated checkpoint and distributed message logging and that MPICH-V can execute a non-trivial parallel application efficiently on volatile nodes. The execution time break down of the BT benchmark demonstrates that MPICH-V compares favorably to P4 and that the main performance degradation on real application is due to the checkpoint mechanism when the number of CM equals the number of nodes.

Future works will consist in increasing the bandwidth and reducing the overhead of MPICH-V by implementing a pipeline inside the CM and adding the management of

redundant executions.

6 Acknowledgments

References

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

[2] A.Selikhov, G.Bosilca, S.Germain, G.Fedak, and F.Cappello. MPICH-CM: a communication library design for P2P MPI implementation. In *To appear in Proceedings of 9-th EuroPVM/MPI conference, Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York, 2002*.

[3] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Mail Stop T 27 A-1, Moffett Field, CA 94035-1000, USA, December 1995.

[4] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte. MPI/FT^{TM}: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *In Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid held in Melbourne, Australia. *, 2001.

[5] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the ACM international conference on Measurement and modeling of computer systems, SIGMETRICS*, pages 34–43, 2000.

[6] A. Brown and D. A. Patterson. Embracing failure: A case for recovery-oriented computing (roc). In *High Performance Transaction Processing Symposium, Asilomar, CA*, October 2001.

[7] Yuqun Chen, Kai Li, and James S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings the IEEE Supercomputing '97 Conference (SC97)*, november 1997.

[8] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. In *Technical Report CMU-CS-96-181, Carnegie Mellon University, October*, 1996.

[9] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed processes in manetho. In *FTCS-22: 22nd International Symposium on Fault Tolerant Computing*, pages 18–27, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[10] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting 2000 ,Springer-Verlag, Berlin, Germany, pp.346-353.*, 2000.

[11] G. Fagg and K. London. MPI interconnection and control. In *Technical Report TR98-42, Major Shared Resource Center, U.S. Army Corps of Engineers Waterways Experiment Station, Vicksburg, Missippissi*, 1998.

[12] I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *In Proceedings of SC 98. IEEE, Nov.*, 1999.

[13] Ian T. Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[14] Edgar Gabriel, Michael Resch, and Roland Rhle. Implementing MPI with optimized algorithms for meta-computing. In Yoginder S. Dandass Anthony Skjellum, Purushotham V. Bangalore, editor, *Proceedings of the Third MPI Developer's and User's Conference*. MPI Software Technology Press, 1999.

[15] Cecile Germain, Vincent Neri, Gille Fedak, and Franck Cappello. XtremWeb: Building an experimental platform for global computing. In *The 1st IEEE/ACM International Workshop on Grid Compuing, Springer Verlag, LNCS 1971*, pages 91–101, 2000.

[16] 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.

[17] D B. Johnson and W Zwaenepoel. Sender-based message logging. Technical report, Department of Computer Science, Rice University, Houston, Texas, 1987.

[18] T. T-Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *11th Int. Conf on Distributed Computing Systems, ICDCS-11*, pages 454–461, MAY 1991.

[19] Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evripidou. MPI-FT: Portable fault tolerance scheme for MPI. In *Parallel Processing Letters, Vol. 10, No. 4, 371-382 , World Scientific Publishing Company.*, 2000.

[20] P. N. Pruitt. *An Asynchronous Checkpoint and Rollback Facility for Distributed Computations*. PhD thesis, College of William and Mary in Virginia, May 1998.

[21] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.

[22] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

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