Replication for send-deterministic MPI HPC applications

Arnaud Lefray, Thomas Ropars, André Schiper

Replication for Send-Deterministic MPI HPC Applications

Arnaud Lefray,* Thomas Ropars and André Schiper
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

ABSTRACT

Replication has recently gained attention in the context of fault tolerance for large scale MPI HPC applications. Existing implementations try to cover all MPI codes and to be independent from the underlying library. In this paper, we evaluate the advantages of adopting a different approach. First, we try to take advantage of a communication property common to many MPI HPC application, namely send-determinism. Second, we choose to implement replication inside the MPI library. The main advantage of our approach is simplicity. While being only a small patch to the Open MPI library, our solution called SDR-MPI supports most main features of the MPI standard including all collectives and group operations. SDR-MPI additionally achieves good performance: Experiments run with HPC benchmarks and applications show that its overhead remains below 5%.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; C.2.4 [**Computer-Communcation Networks**]: Distributed Systems—*Distributed applications*

Keywords

Replication, High Performance Computing, Message-Passing

1. INTRODUCTION

Future exascale systems are expected to experience much more failures (permanent and transient) than existing large scale HPC systems. New fault tolerant solutions are required to deal with this high failure rate. In this context, replication techniques have recently gained attention when they were previously considered too expensive for HPC [9].

*Current Affiliation: University of Lyon - LIP Laboratory, UMR CNRS - ENS de Lyon - INRIA - UCB Lyon 5668, ENSI de Bourges - LIFO, France, email: `arnaud.lefray@ens-lyon.fr`.

Checkpointing techniques and more specifically coordinated checkpointing protocols are today the main solution to provide fault tolerance for MPI HPC applications. However, such protocols have severe scalability issues: i) coordinating all processes before saving their image on a Parallel File System (PFS) can lead to contention [16]; ii) a single process failure requires the rollback of all processes, wasting a large amount of computational resources [7]. A high failure rate exacerbates the scalability issues by reducing the optimal checkpointing period. Prospective studies estimate that if traditional coordinated checkpointing is used in future exascale systems, more than 50% of the execution time would be spent saving checkpoints or recovering from a failure [16]. Thus, one can wonder if duplicating each process to avoid application failures could be beneficial.

The study presented in [9] is the first showing that active replication could outperform coordinated checkpointing at scale[1]. In that work, replication is combined with coordinated checkpointing: If each process is replicated, the probably that the application needs to be restarted from a checkpoint, meaning that all replicas of one process have failed, is dramatically reduced compared to a scenario without replication. Consequently, the checkpoint frequency can be greatly reduced. Another major advantage of replication is that it can be used to detect and correct silent data corruptions by comparing the outputs of replicas [10].

Since the first comparison between checkpointing and replication techniques, several solutions have been proposed to improve the efficiency of checkpointing-based solutions, including solutions to improve checkpoint storage performance [15, 11], and new checkpointing protocols [17]. But, to our knowledge no work have focused on improving replication solutions. Existing replication solutions that can handle crashes [8, 9] target transparency. They try to cover all MPI applications and they are implemented in the profiling MPI layer (PMPI) to be independent from the underlying MPI library. In this paper, we investigate whether transparency can be traded for simplicity and performance.

To this end, we study how to use the *send-determinism* common to most MPI HPC application [5] to design an efficient replication protocol. In a *send-deterministic* application, a process always sends the same sequence of messages in any correct execution for a given set of input parameters: The execution is not impacted by the reception order of concurrent messages. This property has been used in the design of scalable checkpointing protocols [12, 13]. We

[1]Actually, semi-active replication [18] is used to be able to deal with non-determinism

explain how the leader-based approach can be avoided in a replication protocol to deal with non-determinism thanks to send-determinism. We implement the resulting solution called SDR-MPI (Send-Deterministic Replicated MPI) inside the Open MPI library. More precisely, we make the following contributions:

- We present SDR-MPI, a replication protocol for send-deterministic MPI applications. It includes a recovery protocol that works for a replication degree of two (dual replication).
- We describe the implementation of SDR-MPI in the Open MPI library. It is only a small patch to the Open MPI code and can handle most MPI function calls (including collective operations and operations on communicators and groups).
- We evaluate our protocol on a high-performance network (InfiniBand), and show using a set of benchmarks and real applications that *SDR-MPI* induces almost no overhead on the applications' performance.

Section 2 details the context of this study including the related work. Section 3 presents SDR-MPI protocol. Section 4 describes the implementation of SDR-MPI in Open MPI and the results of our experiments. Finally, Section 5 presents our conclusions.

2. CONTEXT

In this paper we consider a crash failure model for the processes. We start by defining *send-determinism* and describing the functioning of MPI libraries. Then we introduce some notations used in the the paper. Finally, we present the related work on MPI replication.

2.1 Send-deterministic Applications

To model a message-passing application, we assume a set $P = \{p_1, p_2, ..., p_n\}$ of n processes, and a set C of channels connecting any ordered pair of processes. Channels are assumed to be FIFO and reliable but no assumption is made on system synchrony.

An MPI application implements an algorithm A. An execution E_A of algorithm A has initial state $\Sigma^0 = \{\sigma_1^0, \sigma_2^0, ..., \sigma_n^0\}$, where σ_i^0 is the initial state of process p_i, and generates a sequence S_E of events e_i^k, where e_i^k is the k^{th} event on process p_i. The state of process p_i after the occurrence of e_i^k is σ_i^k. The sequence of events in S_E is a total order that complies with Lamport's *happened-before* partial order relation [14].

Starting from initial state Σ^0, an algorithm may generate different executions. We define $\mathscr{E}_A$ as the set of executions that can be generated by algorithm A when no crash occurs. The set $\mathscr{S}_A$ includes the sequences of events S_E corresponding to the executions in $\mathscr{E}_A$. The sub-sequence of S_E consisting of events on process p_i is denoted $S_E|p_i$.

Using this model, we can define a send-deterministic algorithm [5]:

Definition 1 (Send-deterministic algorithm). *An algorithm A is send-deterministic if, considering an initial state Σ^0, for each $p \in P$ and $\forall S \in \mathscr{S}_A$, $S|p$ contains the same sub-sequence of* send *events.*

It is usually considered that in MPI HPC applications, only events related to message delivery can be non deterministic. Send-determinism implies that this non-determinism related to the timing or the relative reception order of messages has no impact on the execution of the application. A static analysis of a representative set of HPC benchmarks and applications shows that most MPI HPC applications are send-deterministic [5]. More precisely, all SPMD applications analyzed in that study are send-deterministic. The main class of applications that are not send-deterministic are Master-Workers applications.

In previous works, send-determinism has been leveraged to design new rollback-recovery protocols [12, 13]. It has been shown that thanks to send-determinism, it is possible to design an uncoordinated checkpointing protocol that does not suffer from the domino effect [12]. The idea behind leveraging send-determinism is to propose efficient fault-tolerant solutions that could be applied to a large number of applications, rather than trying to cover all existing applications because it usually results in less efficient solutions. In this paper, we study how send-determinism can be used in replication protocols for MPI applications.

2.2 MPI Applications

The MPI standard defines a set of functions for point-to-point communication and also a set of collective operations. In this paper, we assume that collective operations are implemented on top of the point-to-point functions[2]. Thus, in the following we focus on point-to-point communication.

We describe the main events that can be associated with the sending and the reception of MPI messages. To define these events, we need to consider two layers in the execution of a MPI process, namely the MPI library level and the application level. Figure 1 describes a scenario where a process p_1 sends a message m to a process p_2. It describes the general case where a process uses `MPI_Isend` (resp. `MPI_Irecv`) to post a *send* (resp. *recv*) request and then, uses `MPI_Wait` to wait for the request completion. Furthermore Figure 1 considers a *normal* send operation, *i.e.*, no special protocol such as *rendez-vous* is used for this message.

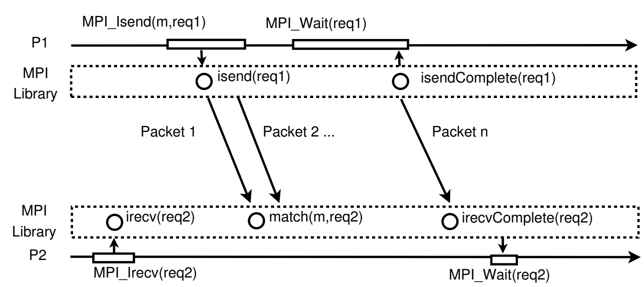


Figure 1: MPI point-to-point communication

To send a message, process p_1 posts a *send* request to the MPI library (event `isend`). The library then sends the message, divided into multiple packets if needed, to the destination. The message sending completes once the last packet has been sent (event `isendComplete`). To know when a message has been sent, the MPI process uses the `MPI_Wait` function. The exact semantic of the `MPI_Wait` function called on a *send* request is that it returns when the payload buffer associated with the *send* request can safely be modified:

[2]This assumption is valid in the MPICH2 and in the Open MPI libraries if collectives are not provided in hardware.

It means either that the message has been completely sent (case illustrated in Figure 1), or that the payload has been internally copied into another buffer.

On the receiver side, the process posts a *recv* request (event `irecv`). In this request, the process specifies the buffer in which it wants to receive the message, and a set of metadata including the identifier of the process it wants to receive from. The matching between an incoming message and a posted reception request is done when the first packet of a message arrives (event `match`). The receiver uses the `MPI_Wait` completion function to know when a message has been fully received. Note that the *recv* request might be completed at the MPI level (event `irecvComplete`), before the application calls `MPI_Wait`.

Non-determinism in MPI applications can have different causes. First, instead of providing a process identifier in a *recv* request, a process may use the wildcard `MPI_ANY_SOURCE` to receive the next message coming from any process. In this case, the output of the completion function that will be used for this request is non-deterministic. Second, the result of completion functions that test the current status of requests (*e.g.*, `MPI_Test`) depends on the progression of the tested requests. Lastly, the result of completion functions, such as `MPI_Waitany`, depends on the relative progress speed of multiple requests. In *send-deterministic* applications, the impact of these non-deterministic calls cannot be observed externally, *i.e.*, on the messages sent.

2.3 Notations

To replicate a MPI application, several replicas (*physical processes*) of each MPI rank (*logical processes*) are created. In the paper, we use *process* or *replica* to refer to *physical processes*. *Logical processes* are referred to as *MPI ranks*. We use the notation p_i^k to name the k-th replica of the logical MPI rank i.

2.4 Related Work

A replication protocol has to ensure (a) that all replicas of a MPI rank receive the same set of messages despite failures (and that this set is the same as in a non-replicated failure-free execution), and (b) that the output of all MPI calls that could be non-deterministic is the same on all replicas. To ensure (a), two kinds of replication protocols have been defined [3]: mirror and parallel protocols.

To explain these two kinds of protocols, we consider the case of rank A sending a message m to rank B. In a mirror protocol, all replicas of A send m to all replicas of B. Thus, as long as one replica of A is non-faulty, all replicas of B receive m. The drawback is the communication cost. If r is the replication degree and q is the number of application messages in a non-replicated execution, the number of application messages with replication is $O(q * r^2)$. In a parallel protocol, replica i of rank A sends its message only to replica i of rank B. Replicas of A additionally need to synchronize to ensure that they all managed to send m to the replicas of B. Thus, the complexity in terms of application messages is only $O(q * r)$, but additional acknowledgements at the protocol level have to be sent.

Solving (b) requires all replicas of a rank to agree on the output of non-deterministic MPI functions. In existing replication protocols [9, 8, 10], this problem is solved by electing one replica as a leader. These solutions assume that an external service provides a consistent view of the failures in the system to all replicas [9]. When a non-deterministic function is called, the leader decides and informs the other replicas of the output. SDR-MPI leverages send-determinism to avoid such a leader-based approach.

These replication protocols can be considered as semi-active protocols as defined in the context of distributed systems replication [18]. However, contrary to semi-active protocols, atomic broadcast is not required because most of the time MPI processes can rely on the MPI semantic to decide locally on the delivery order of messages (*i.e.*, when a process posts a *recv* request where the source is specified, a single message can be delivered by this request). As a consequence, mirror and parallel protocols only provide a service similar to *reliable broadcast*.

Replication solutions for MPI applications in HPC systems that targets crash failures include rMPI [9] and MR-MPI [8]. Both are implemented using the profiling interface of MPI (PMPI). It allows them to provide a transparent and non-intrusive replication solution that can be used with any MPI library. However, this transparency comes at the cost of a high complexity in the implementation. Handling replication requires re-implementing some complex functions, such as collective operations and operations on communicators or groups at the PMPI level. Consequently, performance can also be impacted in case not all optimized algorithms for collectives are re-implemented.

MR-MPI is based on a mirror protocol and can provide partial replication (*i.e.*, only a subset of the MPI ranks might be replicated). It implements all collective operations as well as all operations on communicators and groups. However, the overhead on performance induced by MR-MPI is high (up to 160%). On the other hand, rMPI does not implement all operations on groups but provides much better performance. Note that rMPI includes a slight modification of the underlying MPI library, MPICH2 in their case. Both with a mirror protocol and with a parallel protocol, their performance overhead remains below 20%. Additionally, their results show that choosing the best protocol depends on the characteristics of the application. In the implementation of SDR-MPI, we decided to trade transparency for performance and simplicity. SDR-MPI is a parallel protocol implemented inside the Open MPI library. It implements all collective and communicator operations while being only a small patch to the Open MPI library.

Finally, redMPI [10] aims at detecting and correcting silent faults by comparing the messages sent by the replicas of a MPI rank. Each replica sends a message to one receiver plus a hash to all other replicas to do the comparison. Since redMPI does not deal with crashes, it can avoid synchronization between replicas to ensure that all replicas of a rank receive the same set of messages. However, redMPI also adopts a leader-based approach to deal with non-determinism. As a consequence, the overhead induced by redMPI is low when executing *deterministic* applications (less than 6.8%), but this overhead increases when the application includes non-deterministic function calls (up to 29%). The solutions we propose could also be used by redMPI.

3. A REPLICATION PROTOCOL BASED ON SEND-DETERMINISM

In this section, we present our parallel-based replication protocol for send-deterministic MPI applications, called SDR-

MPI. We first explain how send-determinism can be used in such replication protocol. Then we provide a complete description of the protocol including failure recovery.

3.1 Using Send-determinism

As explained in Section 2, MPI applications may include non-deterministic events. Existing protocols adopt a *semi-active* approach to deal with these non-deterministic events where a leader decides on the non-deterministic outputs and broadcasts the decision to all replicas. On the other hand, send-determinism states that such non-deterministic events have no impact on the externally observable behavior of the MPI processes. Obviously, with send-deterministic applications, there is no need to ensure that the output of non-deterministic function calls is the same on all replicas of an MPI rank. Replicas execution can diverge temporarily and this divergence will not be observed from the outside. The leader-based approach can be avoided.

Figure 2 illustrates the performance gain that can be obtained due to send-determinism. It shows an application running with two ranks and dual replication. Physical processes p_0^0, p_0^1 are replicas of rank 0 and p_1^0, p_1^1 are replicas of rank 1. A message is sent from rank 1 and rank 0 tries to receive it using an *anonymous* reception request. On the left side, without send-determinism, the leader-replica p_0^0 receives the message from rank 1 and then imposes reception from rank 1 to p_0^1 (message $ANY\_SOURCE = p_1$). On the right side, with send-determinism, p_0^0 and p_0^1 can decide locally on the message to deliver. This figure illustrates how send-determinism can simplify replication protocols. Performance can also be improved since the extra synchronization between the leader and other replicas is removed from the critical path.

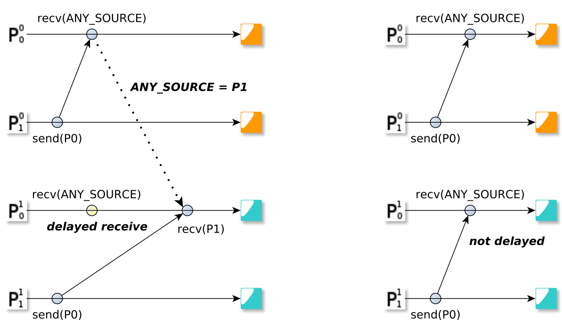


Figure 2: Handling an anonymous reception without send-determinism (left) and with send-determinism (right)

In addition to the synchronization cost, the leader-based approach increases the probability of *unexpected messages* (messages that arrive before the receive request has been posted) when replicas have to delay posting their request until they receive the information on the message to receive from the leader. Unexpected messages may increase execution time because they imply an extra copy of the message from the unexpected queue to the application buffer.

3.2 SDR-MPI in a Nutshell

SDR-MPI is a parallel protocol for send-deterministic applications. Thanks to send-determinism, SDR-MPI only has to ensure that when an application message is sent to a MPI rank, all replicas of this rank receive the message. When there is no failure, replica k of rank i (p_i^k) only sends application messages to replica k of the other ranks. However, if p_i^k fails, the protocol has to ensure that replica k of the other ranks will continue receiving the messages from rank i. To do so, when an application message is sent, replicas of the sender and of the receiver need to synchronize to make sure all replicas received the message before continuing.

In SDR-MPI, when replica p_i^k sends a message m to replica p_j^k, it has to wait for an acknowledgment (ack) from all other replicas of rank j before deleting m. We assume that failures are detected by an external service provided in the system. If replica p_i^k crashes, another replica of the same rank (*e.g.*, p_i^{k+1}) should emit application messages on behalf of the failed one. If p_i^{k+1} did not receive an ack from p_j^k for the message m, it will not have deleted it and will be able to send it to p_j^k.

The functioning of SDR-MPI is illustrated in Figure 3. It considers the dual-replication of two ranks. Rank 1 sends a message to rank 0 ($send(p_0)$) then rank 0 emits a message to rank 1 ($send(p_1)$). This pattern is repeated twice.

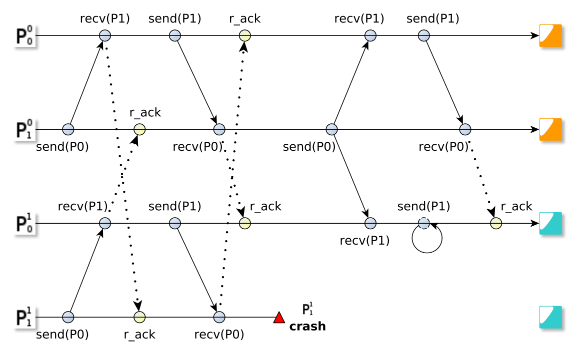


Figure 3: Scenario with a fault on the replica P_1^1

As we use a parallel protocol, each application message is sent once. On message reception, an ack message is sent to all other sender replicas. For example, when p_0^0 finishes $recv(p_1)$, an ack is sent to p_1^1. A sender replica can continue its execution only once it has collected all acks (*i.e*, $r-1$ acks for r replicas). When p_1^1 crashes, the underlying system notifies every processes. Then, p_1^0 starts sending on behalf of p_1^1. The acks on messages ensure that p_1^0 will be able to send all messages that p_1^1 did not send because of the crash. At this point, p_0^1 does not send messages to p_1^1 anymore but just waits for the ack from p_1^0.

Having acks sent by receivers rather than by senders allow us to implement SDR-MPI with minimum modifications to the underlying library. Recall that MPI specifies that a *send* request completion means that the payload buffer associated with the *send* request can safely be modified: It can mean that the payload has been internally copied into another buffer. Thus, knowing on the sender side when a message has been successfully transmitted, would require to modify the low-level layers of the MPI library. Using received-based acknowledgments allows us to modify only the high-level layers of the MPI library.

In our solution, we wait until all acks have been collected before completing a *send* request. (Once the *send* request is completed, the message buffer can then be modified by the application). This solution introduces a small delay but experiments presented in section 4 show that this delay usually has only little impact in practice.

3.3 Replication Algorithm

We provide a detailed description of SDR-MPI and how it handles failures in Algorithm 1. Recovery is discussed in the next section.

Algorithm 1 SDR-MPI with failure management for replica $p = p_i^k$ (k-th replica of MPI rank i) – application size n, replication degree r

Variables:

1: $\forall rank \in 0..n-1, physicalDests_p[rank] \leftarrow p^k_{rank}$
2: $\forall rank \in 0..n-1, physicalSrc_p[rank] \leftarrow p^k_{rank}$
3: $\forall rep \in 0..r-1, substitute_p[rep] \leftarrow p_i^{rep}$

4: **function** MPI_Isend(*msg*, *rank*, *sendReq*)
5:   **for all** $rep \in 0..r-1$ **do**
6:     **if** $p^{rep}_{rank} \in physicalDests_p[rank]$ **then**
7:       $sendReq.reqs[p^{rep}_{rank}] \leftarrow$isend$(msg, p^{rep}_{rank})$
8:     **else if** p^{rep}_{rank} is alive **then**
9:       $sendReq.acks[p^{rep}_{rank}] \leftarrow$irecv$(ack, p^{rep}_{rank})$

10: **function** MPI_Irecv(*msg*, *rank*, *recvReq*)
11:   $recvReq \leftarrow$irecv$(msg, physicalSrc_p[rank])$

12: **function** MPI_Wait (*sendReq*)
13:   waitall(*sendReq.reqs*)
14:   waitall(*sendReq.acks*)

15: **upon** Event `irecvComplete`(*recvReq*) from p^{rep}_{rank} **do**
16:   **for all** $l \in [0..r-1]$, $rep \neq l$, p^l_{rank} is alive **do**
17:     isend(ack, p^l_{rank})

18: **upon** failure of p^{rep}_{rank} **do**
19:   $sub \leftarrow electSubstitute(rep)$
20:   **if** $rank = i$ **then**
21:     **if** $sub = k$ **then**
22:       **for all** $l \in [0..r-1]$ such that $substitute_p[l] = rep$, and $\forall j \in [0..n-1]$ such that P^l_j is alive **do**
23:         add P^l_j to $physicalDests_p[j]$
24:         **for all** *sendReq* to j **such that** $\nexists sendReq.acks[p^l_j]$ **do**
25:           $sendReq.reqs[p^l_j] \leftarrow$ isend$(sendReq.msg, p^l_j)$
26:     **for all** $l \in [0..r-1]$, $substitute_p[l] = p^{rep}_{rank}$ **do**
27:       $substitute_p[l] \leftarrow p^{sub}_{rank}$
28:   **else**
29:     **if** $physicalSrc_p[rank] = p^{rep}_{rank}$ **then**
30:       $physicalSrc_p[rank] \leftarrow p^{sub}_{rank}$
31:     **for all** *sendReq*, $sendReq.dest = rank$ **do**
32:       cancel $sendReq.reqs[p^{rep}_{rank}]$
33:       cancel $sendReq.acks[p^{rep}_{rank}]$
34:     **for all** *recvReq*, $recvReq.src = p^{rep}_{rank}$ **do**
35:       $recvReq.src \leftarrow p^{sub}_{rank}$

Algorithm 1 is executed by every physical process. The algorithm is presented for physical process $p = p^k_i$. We only present the MPI calls `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`. The modifications to apply to other MPI functions related to communication are the same as the one applied to these three functions. The algorithm additionally considers two events: `failure` and `irecvComplete`. The event `irecvComplete` is triggered when a message has been fully received at the MPI library level. The corresponding *recv* request does not have to be completed at the application level.

In the algorithm, *rank* refers to a logical MPI rank, *rep* refers to the *id* of one replica in the set of replicas of one MPI rank. Three data structures are used: $physicalDests_p[rank]$ specifies the set of replicas of *rank* to which physical process p should send a message, when it sends a message to *rank*; $physicalSrc_p[rank]$ defines the replica to receive from when p tries to receive a message from *rank*; $substitute_p[rep]$ defines, inside a set of replicas, the replica that is in charge of sending application messages on behalf of replica *rep*. When there is no failure, there is no need for substitution: the substitute of a replica is the replica itself.

When a process p^{rep}_{rank} fails, one alive replica of *rank* is deterministically elected to send on its behalf (line 19). If process p is also a replica of *rank*, it has to update $substitute_p$ for all replicas where p^{rep}_{rank} was the previous substitute(line 27). Furthermore, if p is elected has the substitute of p^{rep}_{rank}, it has to update its set $physicalDests_p$ with the physical processes p^{rep}_{rank} was previously sending to (line 23). It also has to send the missing messages if any (line 25). Other processes cancel their send/ack requests to/from the failed process p^{rep}_{rank} and replace their receive requests from p^{rep}_{rank} with the new $physicalSrc_p[rank]$ (lines 31- 35).

We highlight that acknowledging messages on `irecvComplete` (line 15), and not when the messages are completed at the application level (*e.g.*, when `MPI_Wait` returns), is mandatory to avoid extra copies of messages. To illustrate this point, consider two processes sending a message to each other using the sequence of MPI calls `MPI_Irecv-MPI_Send-MPI_Wait`. With our protocol, `MPI_Send` needs to receive acks for the sent message before terminating. If acks were sent during the `MPI_Wait` of the *recv* request, this scenario would result in a deadlock. Here, we assume that the MPI library does not allow asynchronous message-passing progress, *i.e.*, the library can only *progress* when the application makes a MPI call. This is the default behaviour for Open MPI [3] and MPICH2 [4]. One way to avoid the deadlock would be to allow `MPI_Send` to terminate before receiving all acks by making an extra copy of the message in case it is later needed. Acknowledging on *irecvComplete* avoids the problem because the processes will be able the send the acks while they are executing `MPI_Send`: while waiting for acks, the processes will try to make all pending requests complete, and so, eventually the *recv* request will be completed generating the *irecvComplete* event.

3.4 Recovery

Existing replication protocols for MPI applications do not recover failed replicas. Not recovering replicas has two drawbacks. First, if all replicas of a MPI rank fail, the system has to rely on checkpointing to avoid losing all computations. Second, In case of a parallel protocol, a replica that has to send messages on behalf of a failed replica gets additional work to do, and so, may slow-down the whole application. In this section, we explain how replicas can be recovered in SDR-MPI. The proposed solution works only for dual replication, which is the common case to deal with crashes.

We explain recovery using the example of Figure 4 where process p^1_1 is recovered. The substitute of the failed process,

[3] www.open-mpi.org
[4] http://www.mpich.org/

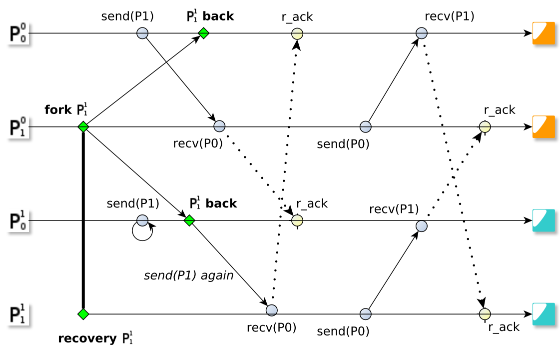


Figure 4: Recovery of process p_1^1

p_1^0 in the figure, is in charge of *forking* the new process. Discussing when to recover a process is outside the scope of this paper. Once the new process has been forked, p_1^0 broadcasts a notification to all alive physical processes. Since channels are FIFO, all messages to rank 1 not acknowledged by p_1^0 at the time the notification arrives, need to be sent to the newly created replica: these messages were not yet received by p_1^0 when the new replica was created. In the figure, p_0^1 sends the missing message to the new p_1^1. Regarding messages sent by rank 1, they need to be acknowledged by p_0^0 to p_1^1 for the messages that are sent after p_1^1 has been recovered. Again, relying on the FIFO channels, p_0^0 knows that it only needs to send an ack for messages received after the notification. Note that for this solution to work, we have to require that p_1^0 does not fail between the fork of p_1^1 and the broadcast of the corresponding notification.

This solution only works for a replication degree of two because if there would be more replicas a single broadcast made by the replica forking the new process would not allow other processes to know whether a message is before of after the creation of the new replica. For instance, if we assume a replication degree of three in Figure 4, *i.e.* we assume two additional replicas p_0^2 and p_1^2, FIFO channels would not help ordering messages exchanged between p_0^2 and p_1^2 with respect to the message broadcast by p_1^0.

4. IMPLEMENTATION AND EVALUATION

Before presenting the evaluation of SDR-MPI, we describe its implementation in Open MPI.

4.1 Implementation in Open MPI

SDR-MPI is integrated into Open MPI[5]. Each replica is a MPI process. Thus, the whole protocol can be implemented using MPI communication functions. Note that checkpointing, recovery are not implemented in the current prototype. Since I/O operations are often used to save intermediate results and implement application-level checkpointing, we plan to integrate application level checkpointing using the solution proposed in [1] to handle IO in a replicated MPI application.

As illustrated in Figure 5, Open MPI is composed of multiple layers. Here we details the layers related to communication. Open MPI implements the PMPI interface to intercept MPI calls before they enter the library. The *OMPI* layer is the internal binding of MPI calls (*e.g*, `MPI_Send` is bound with `OMPI_Send`). Communication calls are transmitted to the *PML* (*Peer-to-peer Managment Layer*). The PML implements the communication protocols. Finally, the *BTL* (*Byte Transfert Layer*) is the interface to the network protocols (e.g Infiniband, TCP).

SDR-MPI is implemented as an additional layer between the OMPI layer and the PML. It is independant of the underlying network. It leverages the *vProtocol* framework [2] to intercept calls to the PML[6]. SDR-MPI does not override the PML functions but just adds some pre-treatment or post-treatment (*e.g*, waiting for acks before returning from a `pml_send`). Since collective operations also rely on the functions of the PML, SDR-MPI supports all collectives without any additional modifications.

Note that SDR-MPI is not fully independant from the underlying PML component. Namely, to capture the events `pml_match` (corresponding to event `match`) and `pml_recv_complete` (corresponding to event `irecvComplete`), we had to patch the PML component. We did it for *ob1*, which is the default implementation of the PML. Nevertheless, the patch being less than a dozen lines of code, it can be applied to any other PML component. In total, SDR-MPI is less than one thousand lines of code.

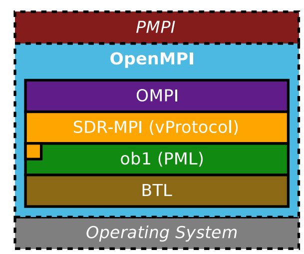


Figure 5: SDR-MPI architecture

To make replication transparent for the application processes, SDR-MPI creates as many `MPI_COMM_WORLD` as the replication degree when the application is created as illustrated in Figure 6. The Open MPI application is launched with $r * n$ processes where r is the replication degree and n the number of logical ranks ($r = 3$ and $n = 2$ on Figure 6). The initial `MPI_COMM_WORLD` is duplicated and kept internal to SDR-MPI to be able to send messages or acks across *worlds*. The duplicated `MPI_COMM_WORLD` is splitted into r *worlds*, that are assigned to processes. When the application refers to `MPI_COMM_WORLD`, it actually manipulates one of the extra *worlds*. Hence, all operations on communicators are transparently handled.

4.2 Experimental Setup

Our experiments are run on 64 nodes of the Grid'5000 cluster located in Nancy. Nodes are equipped with 2 Intel Xeon L5420 (4 cores) processors, 16 GB of memory, and a Mellanox ConnectX IB MHGH29-XTC network adapter (20 Gbps). Operating system is Linux (kernel 2.6.32).

To evaluate the impact of SDR-MPI on the communication latency and throughput, we first run a ping-pong test using NetPipe. Then, we present performance evaluation with five of the NAS Parallel Benchmarks using class D prob-

[5]We use OpenMPI 1.7 unrealeased dev version.

[6]Similar interception mechanisms could be easily implemented in the CH3 channel interface of MPICH2

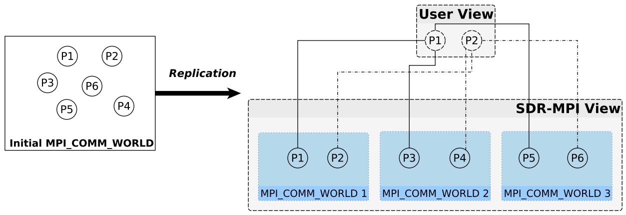


Figure 6: *MPI_COMM_WORLD* separation

	Native (sec)	Replicated (sec)	Overhead (%)
BT	267.24	271.21	1.49
CG	210.37	220.71	4.92
FT	130.61	134.58	3.04
MG	35.14	36.04	2.56
SP	418.62	428.70	2.41

Table 1: Impact of SDR-MPI on the NAS Benchmarks (class=D, nb procs=256, replication degree=2)

lem size. Then we present results with HPCCG, a miniapplication from the Mantveto project implementing a conjugte gradient for a 3D chimney domain, and CM1 [4], an application used to model small-scale atmosphere phenomena such as thunderstorms and tornadoes. HPCCG and CM1 were chosen because they include some receptions with the wildcard *any_source*. HPCCG is run with a problem size of 128x128x64 and CM1 with a problem size of 160x160x160.

All experiments are made with a replication degree of two. For the tests made with NetPipe, each MPI process runs on a different node. In the applications tests, each MPI process is run on a dedicated core, and the two replicas of the same logical rank are run on different nodes. More precisely, the first set of 256 replicas run on the first half of the nodes, and the second set on the other half. Reported executions durations are average values over five executions of each application. Evaluating our protocol with faults is part of the future work.

4.3 Latency and Throughput

Figures 7a and 7b show the latency and the throughput achieved by SDR-MPI on a InfiniBand-20G network, measured using NetPipe. The figures present the performance of SDR-MPI, of the native version of Open MPI, and the performance decrease introduced by SDR-MPI in percent of Open MPI performance (right-hand axis).

The results show that SDR-MPI introduces a noticeable overhead (more than 25%) on the latency only for small messages (less than 100 bytes). Even with such small messages the overhead remains acceptable: For a one-byte message the latency is 2.37 μs with SDR-MPI and 1.67 μs with Open MPI. Similar results are observed for the throughput. This overhead is due to the additional acknowledgement that has to be sent for each message that is received.

4.4 Applications Performance

Table 1 compares the performance of Open MPI and SDR-MPI with a replication degree of two for the NAS benchmarks. Results show that the overhead induced by SDR-MPI is less than 5% in all cases. Of course, in addition to the overhead on the wall-clock time, SDR-MPI doubles the amount of required physical resources (with a replication degree of two). For these applications that do not include *anonymous* receptions, the performance achieved by SDR-MPI is similar to the results reported for the parallel protocol in rMPI [9].

Table 2 presents the results with the two applications that include *anonymous* receptions. It shows that the performance of SDR-MPI does not degrade when *anonymous* receptions are used, contrary to rMPI and RedMPI [10]. These results highlight the benefits of leveraging send-determinism to avoid having to implement a costly protocol to deal with non-determinism.

	Native (sec)	Replicated (sec)	Overhead (%)
HPCCG	91.13	91.29	0.002
CM1	210.21	216.80	3.14

Table 2: Impact of SDR-MPI on HPCCG and CM1 (nb procs=256, replication degree=2)

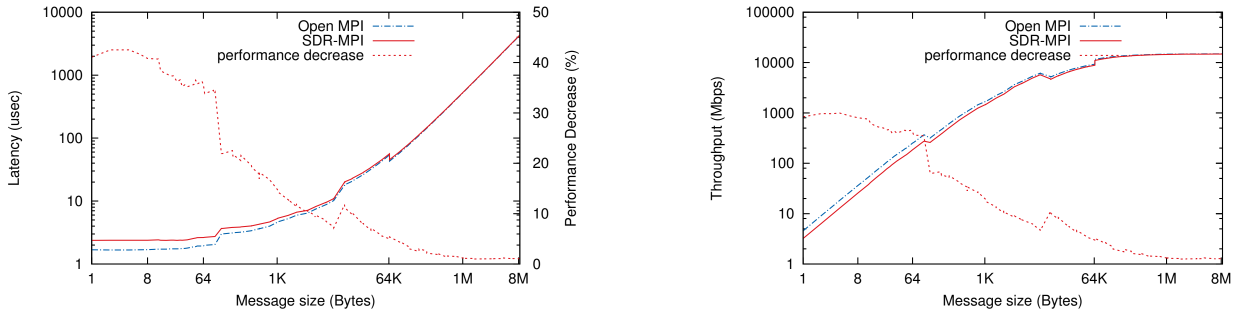
5. CONCLUSION AND PERSPECTIVES

In this paper, we evaluated the benefits of two options that had not yet been considered in the design and implementation of MPI replication: i) leveraging the send-determinism common to many MPI HPC applications and, ii) implementing replication inside the MPI library. We present a full description of the resulting protocol including recovery for a replication degree of two. Our study shows that the main advantage of SDR-MPI is simplicity. While being only a small patch to the Open MPI library, it can handle all MPI collective and group operations. This simplicity is first due to the absence of leader-based protocol to deal with non-determinism. Intercepting communication operations when they enter the point-to-point layer of the MPI library also simplifies the implementation since it allows reusing all complex and optimized algorithms already implemented inside the library, *e.g.* algorithms for collectives. The second advantage of SDR-MPI is performance. On all tested benchmarks and applications, the overhead remains bellow 5%.

With dual replication, SDR-MPI reaches an efficiency close to 50%: two times more resources are used but the wall-clock execution time is very close to a non-replicated one. However,recent advances in checkpointing techniques such as multi-level checkpointing [15, 11] would probably lead to an efficiency higher than 50% for checkpointing at exascale. By definition achieving an efficiency higher than 50% with replication seems impossible. But one can wonder if the entire application and the associated computational workload need to be replicated to avoid application failure. One research direction is to use partial replication [6]. Another approach would be to try to avoid computing everything twice by dividing the computations into multiple tasks and to introduce collaboration between replicas to get these tasks executed.

6. ACKNOWLEDGMENTS

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).



(a) Latency

(b) Throughput

Figure 7: Performance of SDR-MPI on InfiniBand-20G (replication degree=2)

7. REFERENCES

[1] S. Bohm and C. Engelmann. File i/o for mpi applications in redundant execution scenarios. *20th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2012)*, 0:112–119, 2012.

[2] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, Nov. 2010.

[3] R. Brightwell, K. Ferreira, and R. Riesen. Transparent redundant computing with MPI. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 208–218, 2010.

[4] G. H. Bryan and J. M. Fritsch. A Benchmark Simulation for Moist Nonhydrostatic Numerical Models. *Monthly Weather Review*, 2002.

[5] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *19th International Conference on Computer Communications and Networks (ICCCN 2010)*, 2010.

[6] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 615 –626, june 2012.

[7] E. N. Elnozahy et al. System Resilience at Extreme Scale. Technical report, DARPA, 2008.

[8] C. Engelmann and S. Böhm. Redundant execution of hpc applications with mr-mpi. In *International Conference on Parallel and Distributed Computing and Networks (PDCN)*, Feb. 2011.

[9] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *IEEE/ACM SuperComputing 2011*, pages 44:1–44:12, 2011.

[10] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *IEEE/ACM SuperComputing 2012*, pages 78:1–78:12, 2012.

[11] L. B. Gomez, N. Maruyama, D. Komatitsch, S. Tsuboi, F. Cappello, S. Matsuoka, and T. Nakamura. FTI: high performance Fault Tolerance Interface for hybrid systems. In *IEEE/ACM SuperComputing 2011*, Seatle, USA, November 2011.

[12] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications. In *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS2011)*, Anchorage, USA, 2011.

[13] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications. In *26th IEEE International Parallel & Distributed Processing Symposium (IPDPS2012)*, Shanghai, China, 2012.

[14] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[15] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *IEEE/ACM SuperComputing 2012*, SC '10, 2010.

[16] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the Impact of Checkpoints on Next-Generation Systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST '07)*, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[17] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, and P. G. Bridges. Alleviating scalability issues of checkpointing protocols. In *IEEE/ACM SuperComputing 2012*, SC '12, pages 18:1–18:11, 2012.

[18] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, ICDCS '00, pages 464–, 2000.