CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **25**:572–585
Published online 12 July 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2859

SPECIAL ISSUE PAPER

Correlated set coordination in fault tolerant message logging protocols for many-core clusters

Aurelien Bouteiller*,[†], Thomas Herault, George Bosilca and Jack J. Dongarra

Innovative Computing Laboratory, The University of Tennessee, USA

SUMMARY

With our current expectation for the exascale systems, composed of hundred of thousands of many-core nodes, the mean time between failures will become small, even under the most optimistic assumptions. One of the most scalable checkpoint restart techniques, the message logging approach, is the most challenged when the number of cores per node increases because of the high overhead of saving the message payload. Fortunately, for two processes on the same node, the failure probability is correlated, meaning that coordinated recovery is free. In this paper, we propose an intermediate approach that uses coordination between correlated processes but retains the scalability advantage of message logging between independent ones. The algorithm still belongs to the family of event logging protocols but eliminates the need for costly payload logging between coordinated processes. Copyright © 2012 John Wiley & Sons, Ltd.

Received 9 December 2011; Revised 27 April 2012; Accepted 29 April 2012

KEY WORDS: fault tolerance; multicore clusters; checkpoint/restart

1. INTRODUCTION

High-performance computing, as observed by the Top 500 ranking[‡], has exhibited a constant progression of the computing power by a factor of two every 18 months for the last 15 years. Following this trend, the Exaflops milestone should be reached as soon as 2019. The International Exascale Software Project [1] proposes an outline of the characteristics of an Exascale machine on the basis of the foreseeable limits of the hardware and maintenance costs. A machine in this performance range is expected to be built from gigahertz processing cores, with thousands of cores per computing node (up to 10^{12} flops per node), thus requiring millions of computing nodes to reach the Exascale. Software will face the challenges of complex hierarchies and unprecedented levels of parallelism.

One of the major concerns is reliability. If we consider that failures of computing nodes are independent, the reliability probability of the whole system (i.e., the probability that all components will be up and running during the next time unit) is the product of the reliability probability of each of the components. A conservative assumption of a ten-year mean time to failure translates into a probability of 0.99998 that a node will still be running in the next hour. If the system consists of a million of nodes, the probability that at least one unit will be subject to a failure during the next hour jumps to $1-0.99998^{10^6} > 0.99998$. This probability being disruptively close to 1, one can conclude that many computing nodes will inevitably fail during the execution of an Exascale application.

*Correspondence to: Aurelien Bouteiller, Innovative Computing Laboratory, 1122 Volunteer Blvd., 37996 Knoxville, TN, USA.

[†] E-mail: bouteill@eecs.utk.edu

[‡] http://www.top500.org/

Automatic fault tolerant algorithms, which can be provided either by the operating system or the middleware, remove some of the complexity in the development of applications by masking failures and the ensuing recovery process. The most common approaches to automatic fault tolerance are replication, which consumes a high number of computing resources, and rollback recovery. Rollback recovery stores system-level checkpoints of the processes, enabling rollback to a saved state when failures happen. Consistent sets of checkpoints must be computed, using either coordinated checkpointing or some variant of uncoordinated checkpointing with message logging (for brevity, in this article, we use indifferently message logging or uncoordinated checkpointing). Coordinated checkpointing minimizes the overhead of failure-free operations, at the expense of a costly recovery procedure involving the rollback of all processes. Conversely, message logging requires every communication to be tracked to ensure consistency, but its uncoordinated recovery procedure demonstrates unparalleled efficiency in failure prone environments.

Although the low mean time to failure of Exascale machines calls for preferring an uncoordinated checkpoint approach, the overhead on communication of message logging is bound to increase with the advent of many-core nodes. Uncoordinated checkpointing has been designed with the idea that failures are mostly independent, which is not the case in many-core systems where multiple cores crash when the whole node is struck by a failure. Not only do simultaneous failures negate the advantage of uncoordinated recovery, but also the logging of messages between cores is also a major performance issue. All interactions between two uncoordinated processes have to be logged, and a copy of the transaction must be kept for future replay. Because making a copy has the same cost as doing the transaction itself (as the processes are on the same node, we consider the cost of communications equal to the cost of memory copies), the overhead is unacceptable. It is disconcerting that the most resilient fault tolerant method is also the most bound to suffer, in terms of performance, on expected future systems.

In this paper, which is an extension of the distinguished work presented in [2], we consider the case of *correlated failures*: we say that two processes are correlated or codependent if they are likely to be subject to a simultaneous failure. We propose a hybrid approach between coordinated and noncoordinated checkpointing, which prevents the overhead of keeping message copies for communications between correlated processes but retains the more scalable uncoordinated recovery of message logging for processes whose failure probability is independent. The coordination protocol we present is a split protocol, which takes into account the fragmentation of messages, to avoid long waiting cycles, while still implementing a transactional semantic for whole messages. Additionally, we demonstrate that application's communication pattern is likely to adopt a topology that is beneficial to the correlated set approach we propose and leads to a drastic reduction of log volume.

The rest of the paper is organized as follows: in Section 2, we present in more details the execution model, and we recall the behavior of coordinated and uncoordinated checkpointing. In Section 3, we present our protocol in details, which we evaluate using microbenchmarks and application codes on many-core platforms in Section 4. Related works are presented in Section 5. Then, we discuss in section 6 the implications of grouping processes according to hardware locality rather than communication pattern, and we conclude in Section 7.

2. ROLLBACK RECOVERY BACKGROUND

In the following section, we define our execution model. We consider a distributed execution, with explicit message passing. Any process may be subject to permanent (fail-stop) failures. After a failure, a process will be replaced and rejoin the distributed execution by loading a checkpoint image saved by the failed processes prior to the failure.

2.1. Execution model

Events and states. Each computational or communication step of a process is an event. An execution is an alternate sequence of events and process states, with the effect of an event on the preceding state leading the process to the new state. As the system is basically asynchronous, there is no

DOI: 10.1002/cpe

direct time relationship between events occurring on different processes. However, Lamport defined a causal partial ordering between events with the *happened-before* relationship [3].

Events can be classified into two categories. An event is *deterministic* if, in a given state, no other event can apply. On the contrary, if in a given state multiple events can apply and lead to different outcome states, these events are considered nondeterministic. The arrival of a network packet is a notorious example of a nondeterministic event: the ordering of packet arrival depends on network jitter between independent channels, resulting in an uncertain matching between packets and posted receptions (see [4] for a classification of MPI reception events).

Recovery line. Rollback recovery addresses mostly fail-stop errors: a failure is the loss of the complete state and actions of a process. A checkpoint is a copy of a past state of a particular process stored on some persistent memory (remote node, disk, etc.) and used to restore the process in case of failure. The recovery line is the configuration of the entire application after some processes have been reloaded from checkpoints. If the checkpoints can happen at arbitrary dates, some messages can cross the recovery line [5]. Consider the example execution of Figure 1. When process P_1 fails, it rolls back to checkpoint C_1^1. If no other process rolls back, messages m_3, m_4, m_5 are crossing the recovery line. A recovery set is the union of the saved states (checkpoint, messages, events) and a recovery line.

In-transit messages. Messages m_3 and m_4 are crossing the recovery line from the past, and they are called *in-transit* messages. The *in-transit* messages are necessary for the progression of the recovered processes, but they are not available anymore as the corresponding send operation is in the past of the recovery line. For a recovery line to form a complete recovery set, every *in-transit* message must be added to the recovery line.

Orphan messages. Message m_5 is crossing the recovery line from the future to the past; such messages are referred to as *orphan* messages. By following the happened-before relationship, the current state of P_0 depends on the reception of m_5; by transitivity, it also depends on events e_3, e_4, e_5 that occurred on P_1 because C_1^1. Because the channels are asynchronous, the reception of m_3 and m_4, from different senders, can occur in any order during reexecution, leading to a recovered state of P_1 that can diverge from the initial execution. As a result, the current state of P_0 depends on a state that P_1 might never reach after recovery. Checkpoints leading to such inconsistent states are useless and must be discarded; in the worst case, a domino effect can force all checkpoints to be discarded.

2.2. *Building a consistent recovery set*

Two different strategies can be used to create consistent recovery sets. The first one is to create checkpoints at a moment in the history of the application where no *orphan* messages exist, usually through coordination of checkpoints. The second approach avoids coordination but instead saves all *in-transit* messages to keep them available without sender rollback, and keep track of nondeterministic events, so that *orphan* messages can be regenerated identically. We focus our work on this second approach, deemed more scalable.

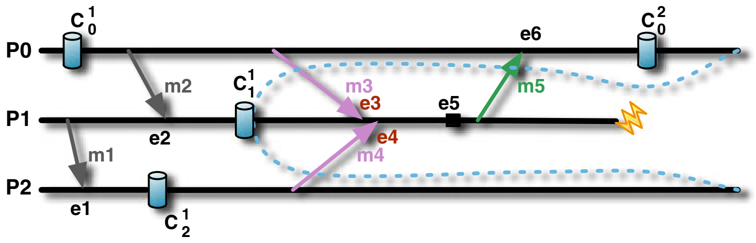


Figure 1. Recovery line based on rollback recovery of a failed process.

DOI: 10.1002/cpe

Coordinated checkpoint. Checkpoint coordination aims at eliminating *in-transit* and *orphan* messages from the recovery set. Several algorithms have been proposed to coordinate checkpoints, the most usual being the Chandy–Lamport algorithm [6] and the blocking coordinated checkpointing, [7, 8], which silences the network. In these algorithms, waves of tokens are exchanged to form a recovery line that eliminates *orphan* messages and detects *in-transit* messages. Coordinated algorithms have the advantage of having almost no overhead outside of checkpointing periods but require that every process, even if unaffected by failures, rolls back to its last checkpoint as this is the only recovery line that is guaranteed to be consistent.

Message logging. Message logging is a family of algorithms that attempt to provide a consistent recovery set from checkpoints taken at independent dates. As the recovery line is arbitrary, every message is potentially *in-transit* or *orphan*. Event logging is the mechanism used to correct the inconsistencies induced by *orphan* messages and nondeterministic events, whereas payload copy is the mechanism used to keep the history of *in-transit* messages. While introducing some overhead on every exchanged message, this scheme can sustain a much more adverse failure pattern, which translates to better efficiency on systems where failures are frequent [9].

Event logging. In event logging, processes are considered *piecewise deterministic*: only sparse nondeterministic events occur, separating large parts of deterministic computation. Event logging suppresses future nondeterministic events by adding the outcome of nondeterministic events to the recovery set so that it can be forced to a deterministic outcome (identical to the initial execution) during recovery. In message logging, the network, more precisely the order of reception, is considered the unique source of nondeterminism. The relative ordering of messages from different senders (e_3, e_4 in Figure 1) is the only information necessary to be logged. For a recovery set to be consistent, no unlogged nondeterministic event can precede an *orphan* message.

Payload copy. When a process is recovering, it needs to replay any reception that happened between the last checkpoint and the failure. Consequently, it requires the payload of *in-transit* messages (m_3, m_4 in Figure 1). Several approaches have been investigated for payload copy; the most efficient one being the sender-based copy [10]. During normal operation, every outgoing message is saved in the sender's volatile memory. The surviving processes can serve past messages to recovering processes on demand, without rolling back. Unlike events, sender-based data do not require stable or synchronous storage (although this data is also part of the checkpoint). Should a process holding useful sender-based data crash, the recovery procedure of this process replays every outgoing send and thus rebuilds the missing messages.

3. GROUP-COORDINATED MESSAGE LOGGING

In this section, we present our approach, designed to reduce the performance penalty because of message logging, suffered by distributed applications on many-core systems. On such systems, the communication subsystem moving data between processes on the same physical node is usually implemented on top of a shared memory substrate. Taking advantage of this geographical proximity of processes on a many-core system, our message logging protocol significantly reduces the amount of payload to be logged by emphasizing characteristics linked to the processes location.

3.1. Shared memory and message logging

3.1.1. Problem statement. In uncoordinated checkpoint schemes, the ordering between checkpoint and message events is arbitrary. As a consequence, every message is potentially *in-transit* and must be copied. Although the cost of the sender-based mechanism involved to perform this necessary copy is not negligible, the cost of a memory copy is often one order of magnitude lower than the cost of the network transfer. Furthermore, the copy and the network operation can overlap. As a result, proper optimization greatly mitigates the performance penalty suffered by network communications

DOI: 10.1002/cpe

(typically to less than 10%, [4, 11]). One can hope that future engineering advances will further reduce this overhead.

Unlike a network communication, a shared memory communication is a strongly memory-bound operation. In the worst case, memory copy induced by message logging doubles the volume of memory transfers. Because it competes for the same scarce resource—memory bandwidth—the cost of this extra copy cannot be overlapped; hence, the time to send a message is irremediably doubled.

A message is *in-transit* (and needs to be copied) if it crosses the recovery line from the past to the future. The emission and reception dates of messages are beyond the control of the fault tolerant algorithm: one could delay the emission or reception dates to match some arbitrary ordering with checkpoint events, but these delays would obviously defeat the goal of improving communication performance. The only events that the fault tolerant algorithm can alter, to enforce an ordering between message events and checkpoint events, are checkpoint dates. Said otherwise, the only way to suppress *in-transit* messages is to synchronize checkpoints.

3.1.2. Correlated failures. Fortunately, although many-core machines put a strain on message logging performance, a new opportunity opens, thanks to the side effect that failures do not have an independent probability on such an environment. All the processes hosted by a single many-core node are prone to fail simultaneously: they are located on the same piece of silicon, share the same memory bus, network interface, cooling fans, power supplies, and operating system, and are subject to the same physical interferences (rays, heat, vibrations, etc.). One of the motivating properties of message logging is that it tolerates a large number of independent failures very well. If failures are correlated, the fault tolerant algorithm can be more synchronous without decreasing its effective efficiency.

The leading idea of our approach is to propose a partially coordinated fault tolerant algorithm that retains message logging between sets of processes experiencing independent failure probability but synchronize the checkpoints of processes that have a strong probability of simultaneous failures, what we call a *correlated set*. It leverages the correlated failure property to avoid message copies that have a high chance of being useless.

3.2. Correlated set coordinated message logging

Whenever a process of a correlated set needs to take a checkpoint, it forces a synchronization with all other processes of the set. If a failure hits a process, all processes of that set have to roll back to their last checkpoint (see the recovery line in example execution depicted in Figure 2). Considering a particular correlated set (as an example S_1), every message can be categorized as either *ingoing* (m_1, m_2), *outgoing* (m_5), or *internal* (m_3, m_4). Between sets, no coordination is enforced. A process failing in another correlated set does not trigger a rollback, but messages between sets have no guaranteed properties with respect to the recovery line and can still be *orphan* or *in-transit*. Therefore, regular message logging, including payload copy and event logging, must continue for outgoing and ingoing messages.

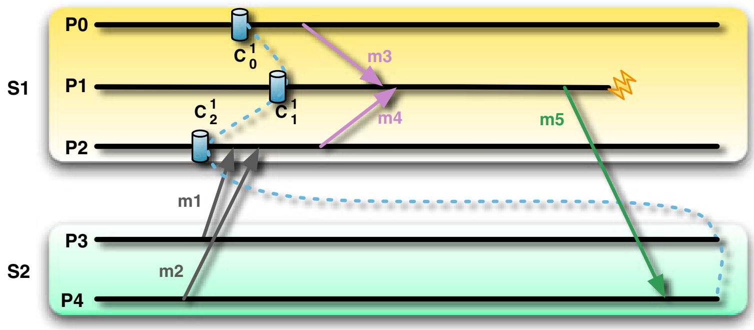


Figure 2. An execution of the correlated set coordinated message logging algorithm.

As checkpoints are coordinated, all *orphan* and *in-transit* messages are eliminated between processes of the correlated set. However, as the total recovery set does contain *in-transit* and *orphan* messages, the consistency proof of coordinated checkpoint does not hold for the recovery set formed by the union of the coordinated sets. In an uncoordinated protocol, a recovery set is consistent if all *in-transit* messages are available and no *orphan* message depends on the outcome of a nondeterministic event. In the next paragraphs, we demonstrate that payload copy can be disabled for internal messages, but that event logging must apply to all types of messages.

3.2.1. Intraset payload copy

Theorem 1
There is no recovery set containing an *in-transit* message between two processes of the same correlated set.

If two processes are part of the same correlated set, they roll back together to the recovery line containing their last checkpoint. By the direct application of the coordination algorithm, no message is *in-transit* between any pair of synchronized processes at the time of checkpoint (in the case of the Chandy/Lamport algorithm, occasional *in-transit* messages are integrated inside the receiver's checkpoint; hence, they are considered as already delivered).

Because an internal message cannot be *in-transit*, it is never sent before the recovery line and received after. Therefore, the payload copy mechanism, used to recover past sent messages during the recovery phase, is unnecessary for internal messages.

3.2.2. Intraset event logging

Theorem 2
In a fault tolerant protocol creating recovery sets with at least two distinct correlated sets, if the nondeterministic outcome of any internal messages preceding an outgoing message is omitted from the recovery set, there exists an execution that reaches an inconsistent state.

Outgoing messages are crossing a noncoordinated portion of the recovery line; hence, the execution follows an arbitrary ordering between checkpoint events and message events. Therefore, for any outgoing message, there is an execution in which it is *orphan*. Consider the case of the execution depicted in Figure 2. In this execution, the message m_5, between the sets S_1 and S_2, is *orphan* in the recovery line produced by a rollback of the processes of S_1.

Let us suppose that event logging of internal messages is unnecessary for building a consistent recovery set. The order between the internal receptions and any other reception of the same process on another channel is nondeterministic. By transitivity of the Lamport relationship, this nondeterminism is propagated to the dependent outgoing message. Because an execution in which this outgoing message is *orphan* exists, the recovery line in this execution is inconsistent. The receptions of messages m_3, m_4 are an example: the nondeterministic outcome created by the unknown ordering of messages in asynchronous channels is propagated to P_4 through m_5. The state of the correlated set S_2 depends on future nondeterministic events of the correlated set S_1; therefore, the recovery set is inconsistent. One can also remark that the same proof holds for ingoing messages (as illustrated by m_1 and m_2).

As a consequence of this theorem, it is necessary to log all message reception events, even if the emitter is located in the same correlated set as the receiver. Only the payload of this message can be spared.

3.3. Implementation

We have implemented the correlated set coordinated message logging algorithm inside the Open MPI library. Open MPI [12] is one of the leading Message Passing Interface standard implementations [13]. In Open MPI, the PML-V framework enables researchers to express their

DOI: 10.1002/cpe

fault tolerant policies. The Vprotocol Pessimist is such an implementation of a pessimistic message logging protocol [4]. To evaluate the performance of our new approach, we have extended this fault tolerant component with the capabilities listed in the following text.

3.3.1. Construction of the correlated set on the basis of hardware proximity. Open MPI enables the end user to select a very precise mapping of his application on the physical resources, up to pinning a particular MPI rank to a particular core. As a consequence, the Open MPI's runtime instantiates a process map detailing node hierarchies and rank allocations. The detection of correlated sets parses this map and extracts the groups of processes hosted on the same node.

3.3.2. Internal messages detection. In Open MPI, the couple formed by the rank and the communicator is translated into a list of endpoints; each one representing a channel to the destination (eth0, ib0, shared memory, etc.). During the construction of the correlated set, all endpoints pertaining to a correlated process are marked so that set membership can be resolved directly. When the fault tolerant protocol considers making a sender-based copy, the endpoint's mark is simply checked to determine if the message payload has to be copied.

3.3.3. Checkpoint coordination in a correlated set. The general idea of a network silence-based coordination is simple: processes send a marker in their communication channels to notify other processes that no other message will be sent before the end of the phase. When all output channels and input channels have been notified, the network is silenced, and the processes can start communicating again. However, MPI communications do not exactly match the theoretical model, which assumes message emissions or receptions are atomic events. In practice, an MPI message is split into several distinct events. The most important includes the emission of the first fragment (also called eager fragment), the matching of an incoming fragment with a receive request, and the delivery of the last fragment. Most of those events are unordered; in particular, a fragment can overtake another fragment, even from the same message (especially with channel bonding). Fortunately, because the MPI matching has to be FIFO, in Open MPI, eager fragments are FIFO, an advantageous property that our algorithm leverages. Our coordination algorithm has three phases: it silences eager fragments so that all posted sends are matched, it completes any matched receives, and it checkpoints processes in the correlated set.

Eager silence. When a process enters the checkpoint synchronization, it sends a token to all correlated opened endpoints. Any send targeting a correlated endpoint, if posted afterwards, is stalled upon completion of the algorithm. When a process not yet synchronizing receives a token, it enters the synchronization immediately. The eager silence phase is complete for a process when it has received a token from every opened endpoint. Because no new message can inject an eager fragment after the token and eager fragments are FIFO, at the end of this phase, all posted sends of processes in the correlated set have been matched.

Rendezvous silence. Unlike eager fragments, the remainder fragments of a message can come in any order. Instead of a complex non-FIFO token algorithm, the property that any fragment left in the channel belongs to an already matched message can be leveraged to drain remaining fragments. In the rendezvous silence phase, every receive request is considered in turn. If a request has matched an eager fragment from a process of the correlated set, the progress engine of Open MPI is called repeatedly until it is detected that this particular request completed. When all such requests have completed, all fragments of internal messages to this process have been drained.

Checkpoint phase. When a process has locally silenced its internal inbound channels, it enters a local barrier. After the barrier, all channels are guaranteed to be empty. Each process then takes a checkpoint. The second barrier denotes that all processes finished checkpointing and that subsequent sends can be resumed.

4. EXPERIMENTAL EVALUATION

In this section, we assess the performance benefit of the correlated set coordination approach on a variety of platforms. First, we investigate the behavior of coordinated message logging on large multicore nodes. Second application performance on a cluster of multicore nodes is presented. Last, we measure the log volume for several widely used collective communication patterns.

4.1. Experimental conditions

The Pluto platform features 48 cores and is our main testbed for large shared memory performance evaluations. Pluto is based on four 12-core AMD opteron 6172 processors with 128 GB of memory. The operating system is Red Hat 4.1.2 with the Linux 2.6.35.7 kernel. Despite the NUMA hierarchies, in this machine, the bandwidth is almost equal between all pairs of cores. The Dancer cluster is an eight-node cluster, where each node has two quad-core Intel Xeon E5520 CPUs, with 4 GB of memory. The operating system is Caos NSA with the 2.6.32.6 Linux kernel. Nodes are connected through an Infiniband 20G network.

All protocols are implemented in Open MPI devel r20284. Vanilla Open MPI means that no fault tolerant protocol is enabled, regular message logging means that the pessimistic algorithm is used, and correlated set message logging denotes that the pessimistic algorithm is used, but cores of the same node undergo coordinated checkpoint. The evaluation includes synthetic benchmarks, such as NetPIPE 3.7 and IMB 3.3, and application benchmarks, such as the NAS 3.3 and HPL (with MKL BLAS10.2). The different benchmarks of the NAS suite accept a constrained number of processes (some expect a square number of processes, others a power of two). In all cases, we ran the largest possible experiment, for a given benchmark and a given parallel machine.

4.2. Shared memory performance

4.2.1. Coordination cost. The cost of coordinating a growing number of cores is presented in Figure 3. The first token exchange is a complete all-to-all, which cannot rely on a spanning tree algorithm. Although all other synchronizations are simple barriers, the token exchange dominates the execution time, which grows quadratically with the number of processes. Note, however, that this synchronization happens only during a checkpoint and that its average cost is comparable with sending a 10 KB message. Clearly, the cost of transmitting a checkpoint to the I/O nodes overshadows the cost of this synchronization.

4.2.2. Ping pong. Figure 4 presents the results of the NetPIPE benchmark on shared memory with a logarithmic scale. Processes are pinned to two cores sharing an L2 cache, a worst case scenario for

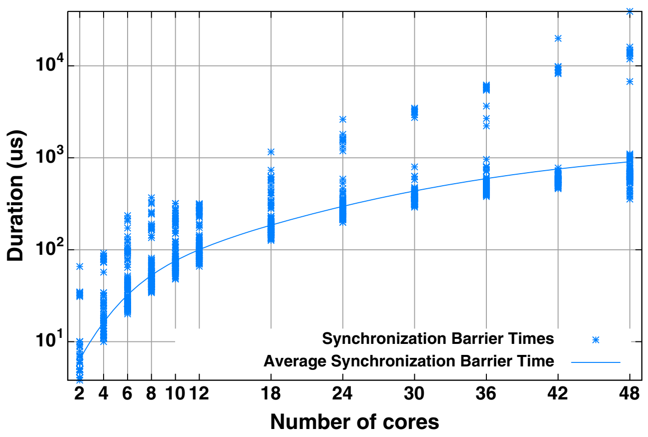


Figure 3. Time to synchronize a correlated set (Pluto platform, log/log scale).

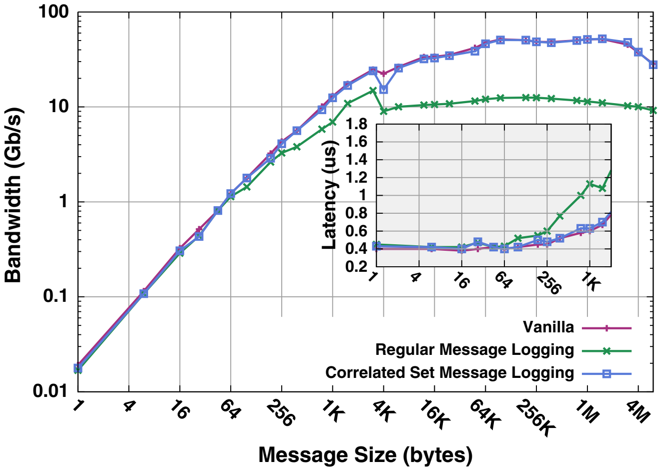


Figure 4. Ping pong performance (Dancer node, shared memory, log/log scale).

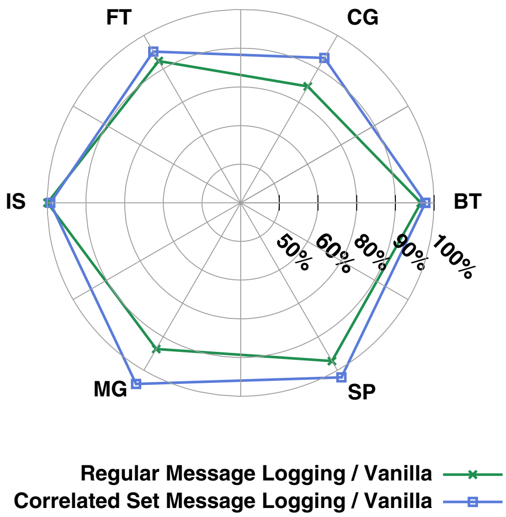


Figure 5. NAS performance (Pluto platform, shared memory, 32/36 cores).

regular message logging. The maximum bandwidth reaches 53 GB/s because communication cost is mostly related to accessing the L2 cache. The sender-based algorithm decreases the bandwidth to 11 GB/s because it copies data to a buffer that is never in the cache. When the communication happens between processes of the same correlated set, the the sender-based mechanism is inactive, and only event logging remains, which enables correlated set message logging to obtain the same bandwidth as the nonfault tolerant execution.

4.2.3. NAS benchmarks. Figure 5 presents the performance of the NAS benchmarks on the shared memory Pluto platform. BT and SP run on 36 cores; all others run on 32. The results presented are the best run out of the ten for each benchmark–protocol combination. One can see that avoiding payload copy enables the correlated set message logging algorithm to experience at most a 7% slowdown and often no overhead, whereas the regular message logging suffers from up to 17% slowdown.

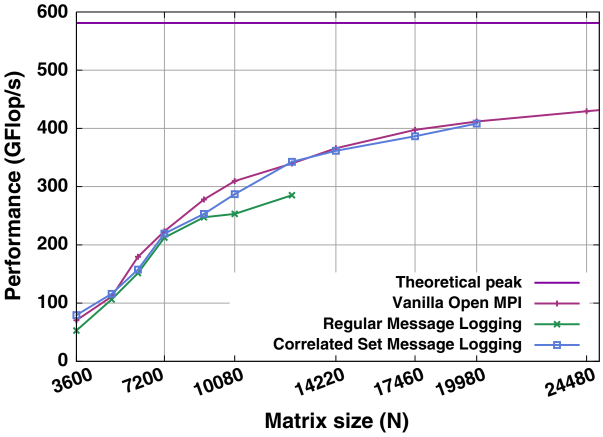


Figure 6. HPL cluster performance (Dancer cluster, IB20G, 8 nodes, 64 cores).

4.3. *Cluster of multicore performance*

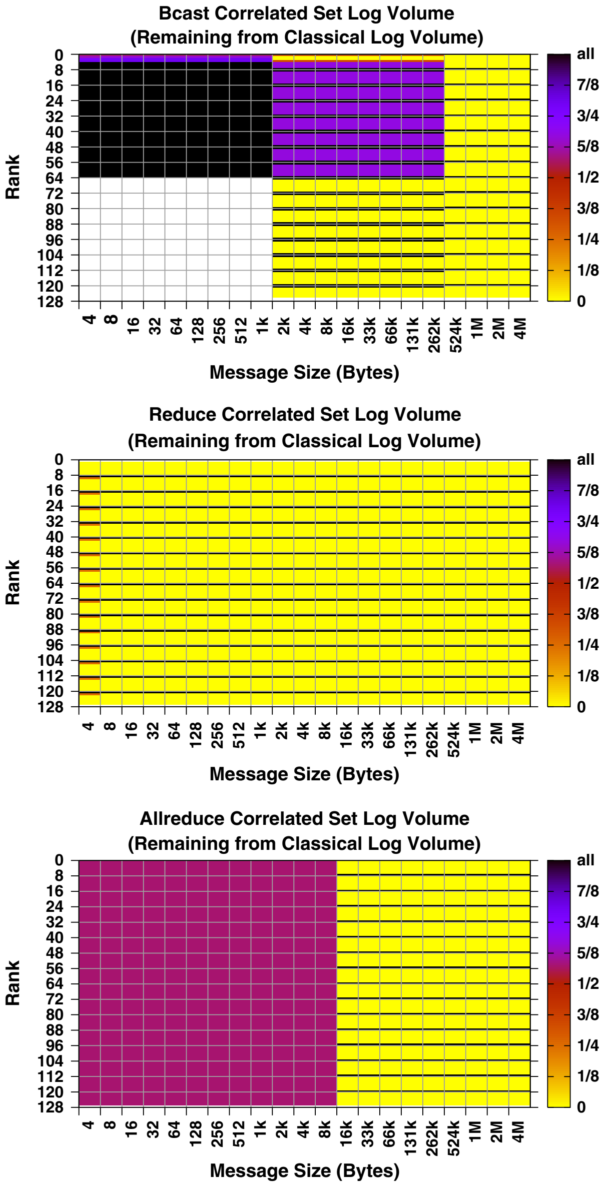
Figure 6 presents the performance of the HPL benchmark on the Dancer cluster, with a one process per core deployment. For small matrix sizes, the behavior is similar between the three MPI versions. However, for slightly larger matrix sizes, the performance of regular message logging suffers. Conversely, the correlated set message logging algorithm performs better and only slightly slower than the nonfault tolerant MPI, regardless of the problem size.

On the Dancer cluster, the available 500 MB of memory per core is a strong limitation. In this memory envelope, the maximum computable problem size on this cluster is $N = 28260$. The extra memory consumed by payload copy limits the maximum problem size to only $N = 12420$ for regular message logging, whereas the reduction on the amount of logged messages enables the correlated set message logging approach to compute problems as large as $N = 19980$. Not only does partial coordination of the message logging algorithm increase communication performance, but it also decreases memory consumption.

4.4. *Collective communications and log volume*

In the experiments presented in Figure 7, the benefit of the correlated set message logging approach is compared with the legacy sender-based approach in terms of logged message volume, for a variety of collective operations. We consider the hierarchical collective communication implementations provided by Open MPI's *Hierarch* module. These operations have the particularity of being implemented in a topology-aware, hierarchical, manner; this is interesting to demonstrate the symbiotic relationship between the desired property of the application—reducing internode communication volume—and the desired property of the message logging scheme—reducing log volume. For a given collective, the colormap presents the ratio between the remaining log volume of the correlated set approach over the total log volume incurred by legacy message logging. The brighter the color, the less log is incurred by correlated set message logging, compared with legacy message logging. Each line represents the volume ratio on a particular rank (i.e., a core); the horizontal grid boundaries, every 8 cores, denote ranks allocated on the same node. The columns, as separated by the grid, pertain to a particular message size.

The first figure presents the log volume of the hierarchical broadcast algorithm. For small messages, between 4 B to 1 KB, half of the processes are sending only messages crossing node boundaries that consequently incur logging in any cases. The other half of the processes are leaves in the topology and therefore do not send any message (denoted by the white area). This illustrates the typical behavior of a nontopology-aware algorithm; for small messages, it is more beneficial to favor operation parallelism over locality. The choice of this latency optimizing binary tree algorithm has dire consequences on message logging as it incurs the logging of the complete volume,


Bcast Correlated Set Log Volume
(Remaining from Classical Log Volume)

Reduce Correlated Set Log Volume
(Remaining from Classical Log Volume)

Allreduce Correlated Set Log Volume
(Remaining from Classical Log Volume)

Figure 7. Remaining log volume of correlated set message logging (as a ratio over regular sender-based message logging, lower is better).

even with correlated sets. Furthermore, it generates log volume imbalance across nodes. However, one should notice that latency avoiding strategies are, by definition, beneficial only for small messages, where logging has no performance penalty on remote link bandwidth and generates small absolute log volume. In contrast, for large messages, which are obviously generating a much larger communication volume, the broadcast algorithm favors a hierarchical, topology-aware pipeline algorithm. This algorithm features the optimal volume of cross-node communications and maximizes bandwidth (the operation extract parallelism between progression of fragments in the pipeline, hence, does not require to extract node parallelism with an aggressive dissemination strategy, as is the case for short messages). As a consequence, on large messages, the correlated set approach reaches the optimal log volume per node: only one core per node logs messages; all other message payloads, having local destinations, are ignored. The overall log volume of the operation is hence divided by the number of cores per node. For intermediate messages, both bandwidth and latency are important; hence, the broadcast algorithm undergoes a hybrid approach that starts by parallelizing the operation as much as possible, first sending fragments along a binary tree and then finishes the broadcast by transmitting large fragments through a pipeline chain. The resulting log volume for the correlated set approach, in this case, reflects the dual nature of the algorithm, which is still imbalanced but does not require to log all messages.

The Reduce algorithm is completely hierarchical, no matter the size of the message. Thanks to the topology-aware nature of that algorithm, it can divide the log volume per node by the optimal factor: the number of cores per nodes.

The Allreduce operation presents the behavior of a many-to-many algorithm. For small and intermediate messages, up to 8 KB, the collective operation is implemented by an algorithm that enables large instant parallelism, at the expense of internode communication volume. Still, because the many-to-many nature of the algorithm incurs a significant stress on the network cross-section bandwidth, the implementors have taken great care of balancing the communication volume per node. They also favor to some extent intranode communications, as is illustrated by the overall 3/8 log volume reduction in correlated set message logging. For large messages, similarly to the one-to-all communications, the main concern is to maximize cross-section bandwidth, which results in favoring the hierarchical pipeline chain algorithm, which is also optimal for correlated set message logging.

Overall, as soon as the collective operation incurs a significant message volume, the implementation of the collective systematically favors an algorithm that (i) minimizes the internode communication volume as this is the crucial performance impacting factor and (ii) balances communication volume per node. As a consequence, the correlated set message logging approach spares the optimal amount of log volume ratio for operations involving the largest absolute communication volume. We advocate that inherently, scalable collective operation implementations aim at reducing the internode communication volume. Hence, they induce a symbiotic relation between the communication pattern and the correlated set message logging approach.

5. RELATED WORKS

Recent advances in message logging have decreased the cost of event logging [4]. As a consequence, more than the logging scheme adopted (a thorough survey of possible approaches is given in [14]), the prominent source of overhead in message logging is the copy of message payload caused by *in-transit* messages [15]. Although attempts at decreasing the cost of payload copy have been successful to some extent [11], these optimizations are hopeless at improving shared memory communication speed. Our approach circumvents this limitation by completely eliminating the need for copies inside many-core processors.

Communication-induced checkpoint (CIC) [16] is another approach that aims at constructing a consistent recovery set without coordination. The CIC algorithm maintains the dependency graph of events and checkpoints to compute *Z-paths* as the execution progresses. Forced checkpoints are taken whenever a Z-path would become a consistency breaking *Z-cycle*. This approach has several drawbacks: it adds piggyback to messages and is notably not scalable because the number of forced checkpoints grows uncontrollably [17].

Group-coordinated checkpoint have been proposed in MVAPICH2 [18] to solve I/O storming issues in coordinated checkpointing. In this paper, the group coordination refers to a particular scheduling of the checkpoint traffic, intended to avoid overwhelming the I/O network. Unlike our approach, which is partially uncoordinated, this algorithm builds a completely coordinated recovery set.

In [19], Ho, Wang, and Lau proposed a group-based approach that combines coordinated and uncoordinated checkpointing, similar to the technique we use in this paper, to reduce the cost of message logging in uncoordinated checkpointing. Their work, however, focuses on communication patterns of the application to reduce the amount of message logging. Similarly, in the context of Charm++ [20] and AMPI [21], Meneses, Mendes, and Kalé have proposed in [22] a team-based approach to reduce the overhead of message logging. The Charm++ model advocates a high level of oversubscription, with a ratio of *Chares* threads per hardware thread much larger than one. In their work, teams are of fixed, predetermined sizes. The paper does not explicitly explain how teams

DOI: 10.1002/cpe

are built, but an emphasis on communication patterns seems preferred. In contrast, our work takes advantage of hardware properties of the computing resources, proposing to build correlated groups on the basis of likeliness of failures and relative efficiency of the communication medium.

6. DISCUSSION ON PROCESS GROUPING

In this section, we compare the interest of dynamically discovering the groups of processes on the basis of their communication patterns, as is often proposed in the literature, versus defining the groups from the physical hierarchy of the machine, as proposed in this work. We proved in Theorem 2 that all nondeterministic events must be logged to maintain the recovery line consistency, independently of the shape or size of the process groups. Therefore, establishing checkpoint synchronization groups is only beneficial by reducing the payload logging mechanism.

Logging the payload of intranode communications introduces an overhead that is of the same order of magnitude as the communications themselves. On the opposite, logging the payload of internode communications is orders of magnitude faster than the communication themselves. Hence, as long as storage memory is available, the logging of internode communications does not introduce a significant slowdown of the application. As a result, grouping coordinated sets according exclusively to communication patterns, without taking into account the relative overhead of the logging operation, can lead to suboptimal performance.

For applications using a large amount of internode communications, memory consumption might become a dominating problem. When the storage space for message payload is exhausted, forced checkpoints must be regularly taken or payload logging must be transferred to a larger, and usually slower, storage. In that respect, grouping according to discovered communication patterns could yield better results than according to hardware process mapping as it is specifically designed to decrease internode communication volume. However, we argue that on practical applications, the difference in volume of logged communications between the two approaches tends to be minimal. Indeed, as seen in the performance evaluation (Section 4.4), collective communications in MPI tend to be implemented in a hierarchical way, naturally grouping processes per levels of hierarchy in the underlying system. As a result, processes physically located on the same machine log only external communications, automatically realizing the same gain as a dynamic group discovery algorithm. Even for applications that rely on point-to-point to express their communication patterns, an application that would intensely communicate between nodes would present scalability and performance issues on a machine with a deep hierarchy. Hence, application programmers and users have a strong incentive to map the communication pattern of the application according to the hardware topology. This results in a symbiotic mapping between the communication patterns and the correlated set message logging, enabling, in practice, close to optimal message logging volume reductions.

7. CONCLUDING REMARKS

In this paper, we proposed a novel approach combining the most advantageous features of coordinated and uncoordinated checkpointing. The resulting fault tolerant protocol, belonging to the event logging protocol family, spares the payload logging for messages belonging to a correlated set but retains uncoordinated recovery scalability. We demonstrate formally, on the one hand, that any pessimistic logging protocol must log all nondeterministic event outcomes, regardless of the type of communication generating them, to fulfill the piecewise deterministic assumption. On the other hand, we prove that payload logging of messages within the group can be safely avoided.

The benefit on shared memory point-to-point performance is significant, translating directly into an observable performance improvement for many types of applications. Even though internode communications are not modified by this approach, the shared memory speedup translates into a reduced overhead on cluster of multicore type platforms. Moreover, the memory required to hold message payload is greatly reduced; our algorithm provides a flexible control of the tradeoff between synchronization and memory consumption. Our discussion emphasizes that the hardware-conscious mapping of the correlated sets not only accounts for failure probability but also tends toward minimizing the volume of payload logging per node. Overall, this work greatly improves the

DOI: 10.1002/cpe

applicability of message logging in the context of distributed systems on the basis of a large number of many-core nodes.

ACKNOWLEDGEMENT

This work was supported by the INRIA-Illinois Joint Laboratory for Petascale Computing, ANR RESCUE project.

REFERENCES

1. Dongarra J, Beckman P, *et al.* The. *Intl. Journal of High Performance Computer Applications* 2011; **25**(11). to appear.
2. Bouteiller A, Hérault T, Bosilca G, Dongarra JJ. Correlated set coordination in fault tolerant message logging protocols. In *Euro-par 2011 Parallel Processing - 17th International Conference, Proceedings, Part II, Lecture Notes in Computer Science*, Vol. 6853. Springer, September 2011; 51–64, DOI: http://dx.doi.org/10.1007/978-3-642-23397-56.
3. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565. DOI: http://doi.acm.org/10.1145/359545.359563.
4. Bouteiller A, Bosilca G, Dongarra J. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience* 2010; **22**(16):2196–2211. DOI: http://dx.doi.org/10.1002/cpe.1589.
5. Bronevetsky G. Portable checkpointing for parallel applications. *Ph.D. Thesis*, Cornell University, Department of Computer Science, 2007.
6. Chandy KM, Lamport L. Distributed snapshots : determining global states of distributed systems. *Transactions on computer systems* February 1985; **3**(1):63–75. ACM.
7. Buntinas D, Coti C, Herault T, Lemarinier P, Pilard L, Rezmerita A, Rodriguez E, Cappello F. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Future Generation Computer Systems* 2008; **24**(1):73–84. http://www.sciencedirect.com/science/article/B6V06-4N2KT6H-1/2/00e790651475028977cc3031d9ea3980.
8. Plank JS. Efficient checkpointing on MIMD architectures. *Ph.D. Thesis*, Thesis, Princeton University, June 1993. http://www.cs.utk.edu/~plank/plank/papers/thesis.html.
9. Lemarinier P, Bouteiller A, Herault T, Krawezik G, Cappello F. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *IEEE International Conference on Cluster Computing*. IEEE CS Press, 2004.
10. Rao S, Alvisi L, Vin HM. The cost of recovery in message logging protocols. In *17th Symposium on Reliable Distributed Systems (SRDS)*. IEEE CS Press, October 1998; 10–18.
11. Bosilca G, Bouteiller A, Herault T, Lemarinier P, Dongarra JJ. Dodging the cost of unavoidable memory copies in message logging protocols. In *EuroMPI, Lecture Notes in Computer Science*, Vol. 6305, Keller R, Gabriel E, Resch MM, Dongarra J (eds). Springer, 2010; 189–197.
12. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004; 97–104.
13. The MPI F. MPI: a message passing interface. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. ACM Press: New York, NY, USA, 1993; 878–883, DOI: http://doi.acm.org/10.1145/169627.169855.
14. Elnozahy EN, Alvisi L, Wang YM, Johnson DB. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 2002; **34**(3):375–408.
15. Bouteiller A, Ropars T, Bosilca G, Morin C, Dongarra J. Reasons to be pessimist or optimist for failure recovery in high performance clusters. In *Proceedings of the 2009 IEEE Cluster Conference*, IEEE (ed.), September 2009.
16. Hlary JM, Mostefaoui A, Raynal M. Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems* 1999; **10**(9):865–877. DOI: http://dx.doi.org/10.1109/71.798312.
17. Alvisi L, Elnozahy E, Rao S, Husain SA, Mel AD. An analysis of communication induced checkpointing. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*. IEEE CS Press, June 1999.
18. Gao Q, Huang W, Koop MJ, Panda DK. Group-based coordinated checkpointing for mpi: a case study on infiniband. *Parallel Processing, 2007. ICPP 2007. International Conference on*, 2007.
19. Ho JCY, Wang CL, Lau FCM. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008; 1–12.
20. Kale L, Charm++. *Encyclopedia of Parallel Computing (To Appear)* (Padua D, ed.) Springer Verlag, 2011.
21. Negara S, Pan KC, Zheng G, Negara N, Johnson RE, Kale LV, Ricker PM. Automatic MPI to AMPI program transformation. *Technical Report 10-09*, Parallel Programming Laboratory, March 2010.
22. Meneses E, Mendes CL, Kalé LV. Team-based message logging: preliminary results. *3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010)*, 2010.