Conditional Checkpoint Abort: an Alternative Semantic for Re-synchronization in CCL

Francesco Quaglia, Andrea Santoro and Bruno Ciciani
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, 00198 Roma, Italy

Abstract

Recently, a Checkpointing and Communication Library (CCL) to support optimistic parallel simulation on myrinet based clusters has been presented. Beyond classical low latency message delivery functionalities, this library additionally offers CPU offloaded checkpointing functionalities based on data transfer capabilities provided by a programmable DMA engine on board of myrinet network cards. A re-synchronization functionality is also supported for both logical (i.e. data consistency) and practical (i.e. hardware contention) reasons, which is implemented according to the following semantic: at any re-synchronization point, the simulation application is momentarily frozen until the last activated DMA based checkpoint operation is completed. In case long freezing periods are experienced, the checkpointing functionalities offered by CCL might not be fully effective in reducing the real checkpointing overhead at the simulation application level. To tackle this drawback, we present an alternative semantic for re-synchronization, namely conditional checkpoint abort, leading to application freezing only in case at least a threshold fraction of the state vector currently being checkpointed has already been transferred into the checkpoint buffer. In the opposite case, the checkpoint operation is aborted and the simulation application is immediately allowed to proceed, thus avoiding excessive checkpointing overhead (due to freezing) at the simulation application level. We also report the results of an evaluation, carried out using classical parameterized synthetic benchmarks, which show that the execution speed of the simulation application can be significantly increased by the alternative semantic we propose.

1 Introduction

Recently, a Checkpointing and Communication Library (CCL) to support optimistic parallel discrete event simulation on myrinet based clusters has been presented [9]. This library exploits data transfer potentiality offered by DMA engines on board of myrinet network cards to support not only classical communication functionalities, but also checkpointing functionalities. In particular, CCL offloads checkpoint operations from the CPU, thus allowing the CPU itself to perform other simulation specific operations (e.g. event lists update, event execution) while DMA based checkpointing is in progress.

A first problem associated with this novel approach deals with concurrency in the execution of checkpointing and other application specific operations, which might cause data inconsistency in some circumstances. Specifically, with DMA based, namely asynchronous, checkpointing there is the possibility that an LP modifies its state vector while that same state vector is still being transferred into a checkpoint buffer. As a consequence, the outcoming checkpoint might maintain an incorrect snapshot of the LP state. A second problem is related to hardware contention on the DMA engine, which occurs when a new checkpoint request is issued by the simulation application while the last issued one is not yet completed. Specifically, including software supports for activation and management of concurrent DMA based checkpoint operations, or at least for queuing them, might strongly increase the software complexity (and therefore the overhead) for the DMA based approach itself.

CCL provides a way to avoid both previous problems. Specifically, it offers a re-synchronization functionality whose semantic, namely Freezing (F), is to suspend on demand the simulation application until the last issued DMA based checkpoint operation, if any, is completed. The combined use of DMA based checkpointing and re-synchronization leads to the so called semi-asynchronous execution mode of checkpoint operations supported by CCL [9]. Actually, if no application freezing occurs at all at any invocation of the re-synchronization functionality, CCL effectively removes the checkpointing overhead from the completion time of the simulation application. On the other hand, proper execution dynamics might originate relatively frequent, long freezing intervals, with consequent non-minimal checkpointing overhead experienced at the simulation application level (\(^1\)).

To reduce the overhead due to freezing, we present in this paper an alternative semantic for the re-synchronization functionality, which we call Conditional Checkpoint Abort (CCA). With this semantic, the application is actually frozen, upon re-synchronization occurrence, only in case at least a threshold fraction, “threshold” for short, of the state vector currently being checkpointed through DMA has been already transferred into the checkpoint buffer. In the opposite case, the checkpoint operation is aborted and the application is allowed to proceed. In practice, the semi-
tic we propose permits to skip a checkpoint (due to checkpoint abort) in case, upon re-synchronization occurrence, the residual completion latency of that checkpoint operation would actually produce non-minimal freezing overhead at the application level.

Note that checkpoint skipping, namely sparse state saving, is a classical solution for reducing the checkpointing overhead in CPU charged checkpointing techniques [1, 3, 5, 11, 13, 14]. However, in those techniques, skipping is achieved by completely avoiding the activation of the checkpoint operation at specific points of the simulation execution, which is rather different from aborting a checkpoint operation after the operation itself has already been activated. More precisely, the CCA semantic we propose for the re-synchronization functionality in CCL actually introduces a kind of a posteriori determination of the state vectors that must be recorded as checkpoints (here “a posteriori” means “after” the activation of the checkpoint operation itself), which is different from the classical a priori determination performed by existing checkpointing techniques. In other words, the two approaches are orthogonal to each other.

We report the results of an empirical study on classical parameterized benchmarks, which show that the CCA semantic can accelerate the simulation execution speed up to 11% as compared to the original semantic, and up to 13% as compared to CPU charged checkpointing techniques based on sparse state saving.

The remainder of this paper is structured as follows. In Section 2 we provide a short, technical description of CCL. In Section 3 we describe the implementation of CCA. The performance results are reported in Section 4.

2 Description of CCL

CCL has been designed for the M2M-PCI32C myrinet card, based on the LANai 4 chip [6]. This chip, whose high level structure is schematized in Figure 1, is a programmable communication device consisting of: (A) An internal bus, namely LBUS (Local BUS), clocked at twice the chip-clock speed. (B) A programmable processor connected to the LBUS, which we will refer to as LANai processor. (C) A RAM bank of 1 Mbyte (LANai internal memory), connected to the LBUS, which can be mapped into the memory address space of the host. (D) A packet interface between the myrinet switch and the LANai chip, accessible by the LANai processor. (E) Three DMA engines used respectively for (i) packet-interface/internal-memory transfer (Receive DMA), (ii) internal-memory/packet-interface transfer (Send DMA), (iii) internal-memory/host-memory transfer or vice-versa (EBUS DMA, namely External Bus DMA).

Host access to the LANai internal memory takes place through a PCI bridge, which is also used for EBUS DMA data transfer from the host memory to the LANai internal memory and vice versa.

In CCL messages incoming from the network are temporarily buffered into the LANai internal memory (data transfer between the packet interface and the internal memory takes place through the Receive DMA) and then transferred into the receive queue, located onto host memory, through the EBUS DMA (see the directed dashed line in Figure 1). Also, CCL adopts a classical optimization called “block-DMA” that allows incoming messages stored in contiguous message slots of the LANai internal memory to be transferred into the receive queue using a single EBUS DMA operation.

Following a common design choice [7], any message send issued by the application involves copying the message content directly into the LANai internal memory, with no intermediate buffering. This is also referred to as “zero-copy” send. Then the message is transferred onto the network through the Send DMA.

In CCL the EBUS DMA is used not only to transfer messages from the LANai internal memory to the receive queue, but also to perform data transfer associated with checkpointing. Specifically, a checkpoint operation involves data transfer from the LP current state buffer (located onto host memory) to the stack of the checkpointed states of the LP (also located onto host memory). As shown by the directed dotted lines in Figure 1, the transfer operation is charged to the EBUS DMA that uses the LANai internal memory as a temporary buffer.

Note that the responsibility to program the three DMA engines anytime there is the need for supporting a given data transfer operation pertains to a control program run by the LANai processor. Therefore, issuing a checkpoint request at the application level actually means requesting the LANai processor to program the EBUS DMA for the data transfer operation. This takes place through the function semi_asynchkpt(int LP_id, time_type simulation_clock), included in the API provided by CCL.

2.1 Re-synchronization

Two issues arise when charging the checkpoint operation to the EBUS DMA. The first issue is related to the fact that the operation is executed asynchronously so that the simulation application is allowed to proceed, which might cause

\footnote{Temporary buffering is needed since the EBUS DMA does not support host-memory to host-memory data transfer directly. It only supports host-memory to internal-memory transfer or vice versa.}
problems if the LP whose state vector is currently being
checkpointed is scheduled for event execution. More pre-
cisely, checkpoint inconsistency might arise due to updates
of the state vector issued by the LP before the completion
of the checkpoint operation.

The second issue is related to the management of an
EBUS DMA based checkpoint request coming up while the
last accepted one, if any, is not yet completed. In case
the new request is accepted, the control program run by
the LANai processor must be able either to handle time-
sharing of multiple checkpoint requests, or at least to queue
the new request. However, non-minimal time requirements
for handling time-sharing or queuing might reduce respon-
siveness of the control program as respect to the activation
of DMA functionalities for supporting message transfer.
This might increase the delivery delay of simulation mes-
sages/antimessages at the application level which might, in
turn, induce rollback thrashing [2].

CCL provides a re-synchronization functionality that, if
carefully managed at the application level (i) prevents the
checkpoint inconsistency problem and (ii) avoids the ac-
tivation of a new checkpoint request until the last issued
one is completed (this removes the need for time-sharing or
queuing of checkpoint requests at the level of the control
program run by the LANai processor).

Actually, re-synchronization takes place through the
function ckpt_wait(), included in the API provided by
CCL, which implements the previously mentioned freez-
ing semantic, namely F. ckpt_wait() returns immediately
if there is no pending checkpoint operation, other-
wise it yields application suspension until the completion
of the on-going operation. Suspension is achieved by spin-
locking the simulation application around a flag, namely
cckpt_completed, implemented as a word (4 bytes) lo-
cated in the LANai internal memory (\(\text{\text{\text{\text{\text{(}}}\}}\)). This flag is set
to zero by the function semi_asynch_ckpt(), upon is-
suing a checkpoint request. It is eventually set to one by
the control program run by the LANai processor as soon as
the data transfer associated with the checkpoint request has
been completed.

3 The CCA Semantic

In this section we present the CCA semantic for the re-
synchronization functionality. As already pointed out, CCA
is different from F since it leads to application suspension
only in case at least a threshold fraction of the state vector
currently being checkpointed has already been transferred
into the checkpoint buffer (instead F leads to unconditional
application suspension). In the negative case, the check-
point operation is aborted, thus yielding no suspension at
all.

As compared to F, the implementation of CCA poses an
additional problem since not only the completion but also
the advancement status of any checkpoint operation must
be tracked. Moreover, such a tracking should be done in a
non-intrusive way to avoid software inefficiency caused by
the implementation of CCA. With the term “non-intrusive”

---

\(^3\)We have used a word instead of a single byte for the flag
cckpt_completed since the internal circuitry of the LANai 4 chip is op-
timized for aligned words access. Also, the host CPU accesses an aligned
word using a single bus cycle, as it would be for the case of a single byte.

---

\(^4\)Recall that access to the EBUS DMA, the PCI bridge and the LBUS is
required for block-DMA operations which transfer messages incoming
from the network into the receive queue. Also, access to the PCI bridge
and the LBUS is required for zero-copy sends issued by the application.
structural modifications of the control program since the counter increment requires a single additional instruction to be executed as soon as the termination of EBUS DMA data transfer associated with checkpointing is detected by the control program. This matches the previous requirement in point (A). (ii) The overhead for handling the counter is negligible in practice since the counter increment instruction is not time consuming. This matches the previous requirement in point (B).

Beyond the absence of structural modifications, we have also mentioned parameter tuning as an important feature to preserve for achieving non-intrusiveness of the tracking mechanism. Actually, the work in [10] has shown there is a single parameter, namely the burst length, whose value is critical to the performance of both communication and checkpointing functionalities offered by CCL. That work provides also a methodology to select at compile time the best suited value for the burst length, i.e. the value that, for the specific cluster environment, is expected to produce the best performance of the checkpointing functionality while maintaining in practice communication performance identical to the one that would have been achieved in case of no activation at all of the checkpointing functionality.

As respect to the latter point, the tracking mechanism based on the completed_transfers counter does not impose modifications of the well suited compile time selected value for the burst length, which contributes to non-intrusiveness. On the other hand, the granularity of the tracking mechanism is obviously dependent on the selected value for the burst length. Issues related to this dependency will be discussed in Section 3.3.

3.2 The Conditional Decision and the Abort Flag

We have decided to support CCA through a proper function provided within the API of CCL. The function prototype is ckpt_cond_abort(float threshold), where the parameter threshold indicates the completion percentage of the last activated checkpoint operation, if any, under which the checkpoint operation must be aborted.

To decide whether to abort the checkpoint operation or not, ckpt_cond_abort() needs information on both the current value of the completed_transfers counter and the total number of EBUS DMA data transfers required for the operation. To simplify the API, we have decided not to provide the latter information as a parameter of the function. Instead, we maintain this information into an additional variable located onto host memory, namely total_transfers, which is visible to the function ckpt_cond_abort(), as well as to the function semi_sync_ckpt(). In other words, the library handles this information in a totally transparent way to the application programmer.

As soon as the checkpoint operation is issued by the application, the function semi_sync_ckpt() computes the total number of EBUS DMA transfers (from/to host memory) to complete the operation according to the following expression:

\[
total\ trans\ fers = 2 \times \left\lfloor \frac{\text{state vector size}}{\text{burst length}} \right\rfloor
\]

where the multiplier factor 2 takes into account the fact that each burst requires two EBUS DMA transfers. The variable total_transfers stores the obtained result, making it available to the function ckpt_cond_abort(). When executed, the function ckpt_cond_abort() takes the decision on whether to abort or not the last activated checkpoint operation on the basis of the condition

\[
\frac{\text{completed_transfers}}{\text{total_transfers}} < \text{threshold}
\]

The operation is aborted if the condition is verified.

Checkpoint abort requires (i) notification of the abort decision to the LANai control program and also (ii) a kind of handshake between the software executed by the host CPU and the LANai control program since the application should not be allowed to issue any further checkpoint operation while abort handling of the last issued one is not yet completed. Limitation in point (ii) derives from the fact that, due to the software structure, issuing a new checkpoint request while the last issued one has not yet been completely handled might cause data structures inconsistency that, in turn, might yield unpredictable software behavior.

Issues in points (i) and (ii) have been tackled by the use of an additional flag, namely ckpt_abort, implemented as a word located into the LANai internal memory, which is managed as follows. The flag is initialized to zero. It is set to one by the function ckpt_cond_abort() in case the checkpoint abort decision is taken. It is eventually reset by the control program after it has interrupted EBUS DMA data transfer associated with checkpointing. Actually the function semi_sync_ckpt() spin-locks around this flag as soon as it is activated so that checkpoint request notification to the control program takes place only after the last checkpoint abort, if any, has been completely handled.

Given that the software at the host side and the control program execute asynchronously, and given that the length of the time interval between the evaluation of the abort condition and the setting of the ckpt_abort flag performed by the function ckpt_cond_abort() is unpredictable (for example due to time-sharing of multiple applications at the host), there exists the possibility that ckpt_abort is set to one while the last issued checkpoint operation has already been completed. To avoid live-lock of the application due to spin-locking of the function semi_sync_ckpt() around the ckpt_abort flag, the control program run by the LANai processor takes care of the reset of ckpt_abort within its main loop independently of the fact that the last checkpoint operation has been really interrupted or not.

Actually, interruption of an on-going checkpoint operation cannot take place as a preemption on an in-progress EBUS DMA data transfer since, according to hardware specifications [6], it might cause problems to the PCI protocol. Therefore, interruption is implemented by simply avoiding the activation of additional EBUS DMA transfers related to that checkpoint operation. Note that this type of implementation does not require any structural modification to the control program. More precisely, in the original version [9, 10] the control program manages the checkpoint operation as follows: it checks whether the last activated EBUS DMA transfer has been completed and, in the positive case, activates a new transfer. In the current version, the program must activate the new EBUS DMA transfer only in case the ckpt_abort flag has not been set to one. As a
consequence, abort implementation results non-intrusive.

As a last point, if the abort condition (i.e. \(\text{completed\_transfers} < \text{threshold}\)) is not verified, the function \(\text{ckpt\_cond\_abort}\) behaves as follows. It returns immediately in case the checkpoint operation has already been completed. Otherwise it spin-locks around the flag \(\text{ckpt\_completed}\) essentially used for the F semantic.

3.3 Back to the Tracking Mechanism

Since the granularity of the checkpoint advancement tracking mechanism presented in Section 3.1 is equal to a whole EBUS DMA data transfer (from/to the host memory), the Minimum Trackable Advancement Percentage (MTAP) of a checkpoint operation depends on the total number of EBUS DMA data transfers required to complete the operation itself. More precisely, MTAP can be computed as:

\[
\text{MTAP} = \frac{1}{\text{total\_transfers}} \quad (2)
\]

Plugging (1) in (2), we get:

\[
\text{MTAP} = \frac{1}{2 \times \left(\frac{\text{state\_vector\_size}}{\text{burst\_length}}\right)} \quad (3)
\]

Expression (3) shows the dependency of MTAP on the size of the state vector to be checkpointed and on the burst length. To provide the reader with insights on the effects of such a dependency, we show in Figure 2 three different plots for MTAP obtained for three different burst lengths (i.e. 512 bytes, 1 Kbyte and 2 Kbytes)\(^5\), while varying the size of the state vector between 1 Kbyte and 10 Kbytes (with step of 1 Kbyte).

![Figure 2. MTAP vs the State Vector Size for Different Values of the Burst Length.](image)

The plots show that MTAP is in the order of 0.25 or less except for the case of small state vector size (i.e. about 1 or 2 Kbytes). However, the reduced precision of the tracking mechanism for the case of small size state vectors should not be a problem in practice. Specifically, for small size state vectors, the likelihood for the semi-asynchronous checkpoint operation to be not yet completed when the re-synchronization functionality is activated is expected to be minimal (this is confirmed by results reported in [9, 10] and also by the results we report in Section 4), which leads CCA to be actually insensitive to the precision of the tracking mechanism.

4 Performance Data

In this section we report the results of an empirical study on the effectiveness of the CCA semantic. The experiments were all performed on a cluster of 4 Pentium II 300 MHz (128 Mbytes RAM - 512 Kbytes second level cache). All the PCs of the cluster run LINUX (kernel version 2.0.32) and are equipped with M2M-PCI132C myrinet cards. On the basis of the methodology presented in [10], burst length of 1 Kbyte has been selected at compile time of CCL for this cluster environment. Also, the optimistic simulation engine used in [9, 10] has been adopted, with the only modification that the function \(\text{ckpt\_wait}\) has been substituted with the function \(\text{ckpt\_cond\_abort}\). This still allows us to consider the F semantic in the study since passing a threshold set to zero to the function \(\text{ckpt\_cond\_abort}\) actually emulates the execution of the \(\text{ckpt\_wait}\) function, i.e. no checkpoint abort at all is ever originated with that parameter value. For the sake of reader’s convenience, the high level structure of the engine is reported below:

```plaintext
1. pending\_LP = no\_LP;
2. while(not end) {
3. \quad \langle\text{inclusive\_messages}\rangle;
4. \quad \text{LP} = \text{schedule\_next}();
5. \quad \text{ckpt\_cond\_abort(threshold)};
6. \quad \text{if (LP = pending\_LP)}
7. \quad \quad \text{pending\_LP} = \text{no\_LP};
8. \quad \text{if (rollback required for LP)} \text{rollback(LP)};
9. \quad \text{next\_event\_execution(LP)};
10. \quad \text{if (checkpoint required for LP)}
11. \quad \quad \text{ckpt\_cond\_abort(threshold)};
12. \quad \text{pending\_LP} = \text{LP};
13. \quad \text{semi\_asynch_ckpt(LP, simulation\_clock(LP))};
14. \quad \langle\text{send\_messages}\rangle;
```

![Figure 1. Details on a Priori Checkpoint Skipping](image)

The re-synchronization point in line 6 prevents the checkpoint inconsistency problem which might arise in case the LP scheduled for execution coincides with the last LP that issued a semi-asynchronous checkpoint request, namely \(\text{pending\_LP}\). The re-synchronization point in line 11 avoids concurrent activation of multiple checkpoint requests that, at present, cannot be managed by CCL. As a final observation on the engine structure, we note that in line 10 there is a check on whether the state vector of the LP must be checkpointed after the event execution. In our test experiments on CCA we have decided to impose always positive result to the check, thus achieving a situation in which activation of a checkpoint request occurs at each event execution. In this way we are able to test the effectiveness of the a posteriori checkpoint skipping technique associated with the CCA semantic with no interference due to additional use of a priori checkpoint skipping (we recall that a priori skipping is proper of CPU charged checkpointing techniques for sparse state saving and means avoiding the activation of checkpoint operations at specific execution points).

In the analysis we have treated \(\text{threshold}\) as the independent parameter. Therefore, we have tested CCA assuming a set of different values for \(\text{threshold}\), manually moving it from one value to another while performing the experiments (\(^6\)).

We have used a classical parameterized benchmark, namely the PHOLD model [4], with size fixed at 32 LPs evenly distributed among the 4 machines in the cluster. Two different message populations have been used, namely 1

---

\(^5\)Results presented in [10] have shown that well suited burst lengths are expected to be around 1 Kbyte.

\(^6\)Hints on automatic tuning of this parameter can be found in [12].
and 10 messages per LP. As compared to 1 message per LP, 10 messages per LP produce longer rollbacks (occurring less frequently), which allow us to test CCA in case of non-minimal antimessage communication bursts. In both configurations, messages are equally likely to be forwarded to any LP, with timestamp increments following an exponential distribution with mean 10 simulation time units.

In the experiments we have fixed the event execution time at about 150 microseconds (obtained by structuring the event routine as a simple CPU busy loop), selected as an intermediate event granularity value for parallel discrete event simulation applications. As respect to the LP state vector size, we have varied it between 100 bytes and 8 Kbytes (passing through 2 and 4 Kbytes), so as to cover a relatively wide range of values for the state granularity.

We report measures related to the event rate, that is the number of committed events per time unit, which is representative of the simulation execution speed, and also to the following parameters. (i) The frequencies F1 and F2 of checkpoint abort upon the invocation of the re-synchronization functionality. F1 is related to re-synchronization in line 6 of the engine, while F2 is related to re-synchronization in line 11. (ii) The average and the maximum distances (number of events) between two successive committed checkpoints of the same LP. The average distance is important to determine the “density of committed checkpoints” (as respect to the amount of executed events) at the point where the event rate, namely the performance, is maximized. The maximum distance is important to show whether the a posteriori determination of the positions of check points performed by CCL produces on an LP very long sequences of event executions with no committed checkpoint available for recovery purposes. As commonly recognized, very long sequences of uncheckpointed events (in the order of several tens of events) are undesirable due to possible effects on rollback increase and to possible interactions with the memory recovery procedures [3, 8, 13].

We have already mentioned that using a threshold value set to zero allows us to observe the execution behavior for the case of CCA emulating the F semantic. However, we do not limit our analysis to the case of a pure comparison between F and CCA since we also report the event rate achievable using a classical Periodic State Saving (PSS) strategy based on CPU charged checkpointing, which has been selected as a representative sparse state saving strategy relying on a priori determination of the positions of checkpoints. For this strategy, we report both the peak event rate and the corresponding value of the checkpoint interval $\chi$.

In our experiments, CPU charged checkpointing is implemented efficiently as a `memcpy()` call having the effect of copying the content of the current state buffer of the LP into the stack of checkpointed states of that LP. As CCL adopts reserved, unwappable main memory pages for both the LPs state vectors and their stacks ($^7$), to ensure fairness in the comparison we have used the same approach for the case of CPU charged checkpointing.

Each reported value results as the average over 10 runs, all done with different random seeds.

$^7$ This is because the EBUS DMA needs main memory addresses to perform data transfer associated with checkpointing.

![Figure 3. Event Rate - 1 Message per LP.](image1)

![Figure 4. Event Rate - 10 Messages per LP.](image2)

The plots for the event rate, reported in Figure 3 and in Figure 4, show that semi-asynchronous checkpointing with the CCA semantic provides significant execution speed improvements over PSS (up to 13% for 8 Kbytes state vector size and 1 message per LP) except for the case of minimal state granularity (i.e. 100 bytes or 2 Kbytes state vector). Such a result is an expected one and is aligned with results reported in [9] since minimal state granularity implies negligible overhead in case of CPU charged checkpointing, which can be comparable with the overhead due to the management of semi-asynchronous checkpointing (e.g. the overhead due to activation of semi-asynchronous checkpoint operations, which involves data exchange between host and LANai memory passing through the PCI bridge).

Another interesting point is that plots for the event rate assume different shapes depending on the size of the state vector. In particular, plots for 100 bytes and 2 Kbytes state vector size are almost flat. Instead, those for 4 Kbytes and 8 Kbytes state vector size exhibit a clear dependency on threshold. Specifically, for 4 Kbytes state vector size and 1 message per LP, the event rate increases from about 12700 to 13300 committed events per sec. while moving threshold from 0.0 to 0.6, and then stabilizes. Similarly, for the case of 10 messages per LP, it increases from 15000 to 15600 and then stabilizes. This points out how the CCA semantic actually allows execution speed improvement of up to 6% as compared to F. Such a gain is even larger for
8 Kbytes state vector size and 1 message per LP, case in which the event rate starts from about 11700 committed events per sec. when threshold is set to 0.0, and then reaches a peak of about 12860 committed events per sec. when threshold is set to 0.4. Notice that after reaching the maximum value, the curve of the event rate does not stay flat as for the case of 4 Kbytes state vector size. In case of 10 messages per LP, that same curve stays almost flat after reaching the peak value of about 15400 for a threshold of 0.4, which shows how CCA allows a gain of up to 11% as compared to F. Overall, CCA exhibits the potential to provide performance improvements over F for medium/large state vectors, cases in which aborting a checkpoint operation instead of freezing the simulation application for non-minimal periods can positively impact performance. To better summarize these improvements, we report in Figure 5 the speedup achievable using F, CCA (best observed result while varying threshold) and PSS (best observed result while varying $\chi$) against a sequential execution of the same simulation model on one of the four machines of the cluster. The plots show that CCA actually allows notably larger speedup values in case of medium/large state vector size, independently of the message population used.

From the values of F1 and F2 reported in Figure 6 and in Figure 7 we note that, with the exception of 8 Kbytes state vector size, almost all the checkpoint aborts are due to re-synchronization in line 6 of the engine, i.e. re-synchronization preventing the checkpoint inconsistency problem. We recall that re-synchronization in line 6 is activated only in case the pending LP (i.e. the LP that was lastly scheduled for execution and that issued the last semi-asynchronous checkpoint request) coincides with the LP currently scheduled for execution. Therefore, uncommitted checkpoints are mainly due to locality of event occurrences at the same LP over a period of simulation time.

Instead, for the case of 8 Kbytes state vector size and threshold set to 1.0, some aborts are due to the fact that sometimes, but not frequently, CCL is unable to complete the last activated semi-asynchronous checkpoint request before a new activation. Given the largeness of the state vector, this was an expected behavior. Anyway F2 remains null, or almost null, for values of threshold up to 0.8, therefore we have a clear indication that, in spite of the largeness of the state vector, almost all the work associated with data transfer due to checkpointing was actually completed before the activation of a new checkpoint request.

The plots in Figure 8 and in Figure 9 show that, independently of the value of threshold, the maximum distance between committed checkpoints is less than 10 in case of 1 message per LP, and is less than 13 in case of 10 messages per LP. (The slightly higher value in case of 10 messages per LP is supposed to derive from longer antimessage communication bursts, due to longer rollback length, which keep the myrinet network interface hardware busy for communication for a while. This produces delay in data transfer associated with checkpointing since the control program run by the LANai processor assigns higher priority to communication.) This is an indication of the potential of CCA not to give rise to pathological situations with very long sequences of uncheckpointed event executions. In other words, it indicates that CCA should not increase the amount of rollback and should not negatively interfere with the activation frequency of memory recovery procedures since both these effects are typically expected for larger checkpoint distances [3, 13]. As respect to the amount of rollback, in the experiments we have also measured the efficiency, namely the ratio between committed and executed (committed plus rolled back) events, and we have noted that for any considered value of the state vector size the execution with semi-asynchronous checkpointing produces in practice the same (or slightly better) efficiency as the corresponding PSS execution originating the peak event rates reported in Figure 3 and in Figure 4. Anyway, embedding simple, additional mechanisms in the simulation engine for safety purposes, in...
order to avoid the maximum checkpoint distance overstepping a given bound (as typically happens when using most CPU charged sparse state saving strategies) is actually an easy task and is out of the scope of the CCL development presented in this paper.

An interesting observation is related to the particular shape of the curves for the average distance between committed checkpoints in case of 8 Kbytes state vector size. In particular, for both 1 and 10 messages per LP, these curves show two different steps, one for threshold set to 0.2, the other one for threshold set to 1.0. However, only the first step produces an increase in the event rate due to a decrease in the checkpointing overhead (i.e., the freezing overhead). The reason for this is as follows. The plots related to F1 show that when threshold is moved from 0.0 to at least 0.2, we get a non-minimal amount of aborts due to re-synchronization in line 6. This allows aborting most of the checkpoints associated with calls to the function semi_asynckpt() in line 13 in case the calling LP is re-scheduled for execution, which produces a strong reduction of the checkpointing overhead since committing those checkpoints would be costly due to their recent activation. At the same time, the second step in the average distance between committed checkpoints is exclusively due to the increase in the frequency F2 when threshold is moved from 0.8 to 1.0, which, unlike the previous case, does not produce relevant decrease in the checkpointing overhead since the additional aborts involve checkpoints with short expected completion latency due to their non-recent activation. Combining this with the monotonic increase of coasting forward cost vs threshold (due to the increase in the average distance between committed checkpoints), we get the slight decrease in the event rate noted after the peak in 0.4 in case of 1 message per LP.

As a final observation, the low sensitivity of the execution speed vs threshold for the case of relatively small state vectors is a support to the effectiveness of the tracking mechanism for the checkpoint operation advancement described in Section 3.3. As already said, the mechanism exhibits finer granularity for medium/large state vectors, case in which the selection of threshold, and thus adequate management of the CCA semantic, is a critical factor for the final performance.

5 Summary

In this paper we have proposed a new semantic for the re-synchronization functionality in CCL, namely Conditional Checkpoint Abort (CCA). As compared to the previously employed semantic, namely Freezing (F), CCA allows reduction of the checkpointing overhead at the simulation application level since it gives the possibility to perform the abort of a checkpoint operation in case the residual completion latency of the operation itself at the re-synchronization point is expected to be long. This avoids long application freezing periods. Experiments conducted to test CCA have shown that, in case of medium/large state vector size, CCA allows a clear performance increase (in the order of 10-13%) as compared to both the F semantic and the peak performance of a classical periodic state saving strategy based on CPU charged checkpointing.

References