Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Soft errors detection and automatic recovery based on replication combined with different levels of checkpointing

Diego Montezanti [a,*], Enzo Rucci [a], Armando De Giusti [a], Marcelo Naiouf [a], Dolores Rexachs [b], Emilio Luque [b]

[a] III-LIDI - Instituto de Investigación en Informática LIDI, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Buenos Aires, Argentina
[b] CAOS - Computer Architecture and Operating Systems, Universidad Autónoma de Barcelona, Barcelona, Spain

ARTICLE INFO

Article history:
Received 6 April 2019
Received in revised form 14 May 2020
Accepted 3 July 2020
Available online 7 July 2020

Keywords:
Soft error detection
Automatic recovery
System-level checkpoint
User-level checkpoint

ABSTRACT

Handling faults is a growing concern in HPC. In future exascale systems, it is projected that silent undetected errors will occur several times a day, increasing the occurrence of corrupted results. In this article, we propose SEDAR, which is a methodology that improves system reliability against transient faults when running parallel message-passing applications. Our approach, based on process replication for detection, combined with different levels of checkpointing for automatic recovery, has the goal of helping users of scientific applications to obtain executions with correct results. SEDAR is structured in three levels: (1) only detection and safe-stop with notification; (2) recovery based on multiple system-level checkpoints; and (3) recovery based on a single valid user-level checkpoint. As each of these variants supplies a particular coverage but involves limitations and implementation costs, SEDAR can be adapted to the needs of the system. In this work, a description of the methodology is presented and the temporal behavior of employing each SEDAR strategy is mathematically described, both in the absence and presence of faults. A model that considers all the fault scenarios on a test application is introduced to show the validity of the detection and recovery mechanisms. An overhead evaluation of each variant is performed with applications involving different communication patterns; this is also used to extract guidelines about when it is beneficial to employ each SEDAR protection level. As a result, we show its efficacy and viability to tolerate transient faults in target HPC environments.

1. Introduction

In the area of High-Performance Computing (HPC), parallel systems continue increasing the number of components to improve their performance and, as a consequence, ensuring their reliability has become a critical issue. Nowadays, fault rates involve just a few hours on modern platforms [1] but it is forecasted that large parallel applications will have to manage fault rates of barely some minutes in exascale supercomputers [2]. In that sense, these applications require some help to progress efficiently.

Currently, there are different types of transient errors affecting parallel programs; Silent Data Corruption (SDC) is the most dangerous of these as several recent reports have stated [3]. When SDC occurs, the application seems to run correctly but, at the end, the results are incorrect. Science is one of the areas strongly affected by SDC, since historically it has relied on large-scale simulations. Therefore, the treatment of silent errors is one of the greatest challenges in current and future resilience.

Without a fault tolerance mechanism, a whole application could misbehave due to a failure affecting just one task. Even worse, the program could output invalid results that, in the best-case scenario, will be noticed when the execution is concluded; as a consequence, silent errors require a detection mechanism [4]. In addition, a single SDC can cause deep effects, propagating them across all processes that communicate in message-passing applications [5]. One way to tolerate transient faults is to rely on hardware redundancy (registers and processor arithmetical logic units). Nevertheless, this approach is highly expensive and difficult to implement [6]. Given the high cost of re-running the application from the start if a fault is detected, specific software strategies are required to reach a suitable cost–benefit trade-off.

As opposed to the silent errors, the fail-stop failures cause a process to crash, making their detection almost immediate. A common, well-studied technique to reduce their impact consists of Checkpoint-based Rollback recovery (C/R) [7]. When coordinated checkpointing is used, the entire state of an application is saved in a periodic manner. So, if a failure takes place, all

* Corresponding author.
E-mail addresses: dmontezanti@lidi.info.unlp.edu.ar (D. Montezanti), erucci@lidi.info.unlp.edu.ar (E. Rucci), degiusti@lidi.info.unlp.edu.ar (A. De Giusti), mnaiouf@lidi.info.unlp.edu.ar (M. Naiouf), dolores.rexachs@uab.es (D. Rexachs), emilio.luque@uab.es (E. Luque).

https://doi.org/10.1016/j.future.2020.07.003
0167-739X/© 2020 Elsevier B.V. All rights reserved.

the processes can restart from their saved checkpoints. Instead, if uncoordinated checkpointing is used, only the state of the process being checkpointed is dumped. Unfortunately, C/R can be time-consuming and the overhead increases as the number of cores grow. Nonetheless, despite being effective when dealing with fail-stop errors, C/R shows weakness when facing silent errors. Since a stored checkpoint could contain undetected corruption, C/R cannot guarantee a correct recovery. This situation becomes aggravated in the case of strongly coupled computation, since an error in one node could propagate to the others in microseconds [8].

Performing redundant software execution is a common way to provide resilience. Following the state machine replication approach, a process is duplicated and both copies proceed with the same execution sequence. As a result, for deterministic applications, they produce the same output for the same input [9]; these two outputs can be compared to provide error detection, despite not being enough for recovery. In the HPC context, using multicore architectures represents a viable solution for detecting SDCs as a result of their intrinsic natural redundancy [1].

Considering these circumstances of non-reliable results and their expensive verification, this paper presents SEDAR, which is a methodology designed to provide transient fault tolerance for scientific message-passing parallel applications that execute in multicore clusters. SEDAR seeks to help programmers and users of parallel scientific applications to accomplish reliability in their executions. It works as a static library that is compiled with the application. Even though this changes the model of execution, it is still almost transparent to the algorithm, as opposed to specific detectors that force modifying it and do not cover all faults [4,10]. Following this methodology, each process of the parallel application gets replicated and both copies execute on different cores of the same socket, taking advantage of the multicore's intrinsic hardware redundancy. SEDAR can detect and recover from all transient faults that cause SDC and TOE (Time Out Errors). Three different ways are provided by SEDAR so it can achieve full silent error coverage: (1) only detection with notification; (2) recovery based on multiple system-level checkpoints; and (3) recovery utilizing a single safe application-level checkpoint. Each of these alternatives has particular features and provides a different cost-performance trade-off.

A preliminary, more conceptual version of this work is available in [11]. This article fully describes and validates the methodology, extending the insights already offered in the previous version, with the following new contributions:

- The introduction of an analytical model to verify the efficacy both of the detection strategy and of the recovery mechanism based on multiple, system-level coordinated checkpoints. The model is based on the predictability of the data affected in a well-known test application (considering its computation and communication stages), so the consequences of each fault occurrence can be anticipated.
- The design of a complete workfault (i.e. a set of representative fault cases that emulate real faults experienced by the system, which we use as workload for testing purposes) that permits a grouping of all possible faulty situations in *scenarios*. This set includes the effects of the faults, their latency of detection and recovery point.
- The implementation of the multiple-checkpoint-based recovery algorithm, using the DMTCP library. To verify its operation, SEDAR has been incorporated into the test application. The empirical validation has been achieved through controlled fault injection.
- The evaluation of the overhead of each alternative SEDAR strategy. For this purpose, we have attached SEDAR to three parallel benchmarks with different communication patterns and workload demands, thus measuring or estimating the execution parameters for each of them.
- The introduction of a function that describes the average execution time of using each SEDAR strategy.
- A qualitative evaluation of the incidence of the communication pattern's influence on temporal behavior.
- A discussion about the convenience of saving multiple checkpoints for recovery, compared to just employing the detection mechanism. In addition, a brief analysis of how to determine the best moment to start protection has been conducted.

The remainder of the paper is organized as follows: Section 2 reviews some basic concepts and related work. Section 3 describes the proposed methodology from a functional point of view, separating it in the three aforementioned strategies. Section 4 presents the evaluation of the recovery strategy, in addition to a discussion about the convenience of utilization and adaptation, considering the temporal behavior of the possible variants. Finally, in Section 5, the conclusions and future lines of work are detailed.

2. Background and related work

Transient faults can be classified according to their consequences on the program execution [12]. A Latent Error (LE) is a non-harmful fault since it does not affect the final results. This kind of error alters data that are not used anymore. On the contrary, when a Detected Unrecoverable Error (DUE) occurs, a program ends suddenly; the system software can be aware of DUEs but cannot recover from them. In the case of a Time Out Error (TOE), the application does not finish within a stipulated time range. Finally, as mentioned before, when SDC takes part, the application seems to execute correctly, although invalid results are produced. Particularly, in message-passing parallel programs, SDC can be sub-classified into two different types of errors according to [13]: (1) Transmitted Data Corruption (TDC) alters data to be sent by a process, which will propagate to others if it is not detected; (2) Final Status Corruption (FSC) has an effect on non-communicated data, spreading the error in a local manner and invalidating the final results.

Nowadays, there is no *silver bullet* to manage frequent SDC. Some available algorithmic solutions only apply to specific kernels [14], decreasing the cost of error detection; this kind of solution can be used in HPC environments [10]. Among them, some well-known methods like ABFT [10] can detect up to a maximum number of errors in linear-algebra problems. However, each kernel requires an *ad-hoc* implementation, which represents a lot of work for large HPC software. Moreover, algorithm-based solutions are more intrusive as they modify the algorithms [4,14,15]. On the contrary, compiler or runtime software-based detection proposals are more general since they can be employed to any code, but at the cost of a significant increase in complexity. In addition, containment strategies seek to reduce the fault consequences, either stopping its propagation to the other nodes or to the data stored in checkpoints [7]. In [16], the authors increase availability and offer a trade-off between the number and the quality of components through redundancy in HPC systems. In the same way, [17] showed that replication strategy is more efficient than C/R under circumstances of high error rate and large values of C/R overhead.

Traditionally, SDC detection has been achieved through execution replication combined with partial or total result comparison during the execution. Software-redundancy solutions remove the need for expensive hardware performing replication at the level of threads [18], processes [6] or machine status. On the other hand, other proposals require fewer resources but at the cost of reducing their accuracy. One of them is approximate replication, which implements upper and lower limits for computation results [7]. MR-MPI [16] employs a transparent redundancy approach for HPC. It proposes a partial process replication and can be used together with C/R in non-replicated processes [19]. rMPI [17] takes on failures by redundant execution of MPI applications. Using this protocol, the program fails only if two corresponding replicas fail, because each node is duplicated. The probability of simultaneous failure of a node and its replica decreases when the number of nodes increases, so redundancy scales. However, this benefit requires duplicating the number of resources and quadrupling the number of messages. Faults in shared-memory systems are explored in [9], where a scheme based on multi-threaded processes is proposed, including non-determinism management due to memory accesses. A protocol for hybrid task-parallel MPI programs is described in [1], which carries out recovery based on uncoordinated checkpoints and message logging. Only the task that presented the error is restarted and all the MPI calls are handled inside it. RedMPI [5] is an MPI library that exploits rMPI's per-process replication to detect and correct SDC, comparing the messages sent by replicated issuers at the receiver side. It avoids sending all messages and comparing their entire contents through a hashing-based optimization. In addition, it does not require source code modification and guarantees determinism among replicated processes. As it offers protection even when high failure rates occur, RedMPI is a potential alternative to be used on large-scale systems. It is shown that even a single transient error can produce deep effects on the program, causing a corruption pattern that cascades toward all other processes through MPI messages. In a similar way to SEDAR, RedMPI also enables replica mapping on the same physical node as the native processes, or in neighbors with lower network latency. Like our proposal, it monitors communications as a strategy which attempts to provide the correct output. Detection is delayed upon transmission, but, as opposed to SEDAR, validation is carried out on the receiver side. This produces an additional overhead, as well as latency and network congestion that are not present in our solution. Fault tolerant protocols for other parallel programming models, such as PGAS [20] have been also explored. The combination of checkpointing the output of tasks and replicating for application-specific detection is explored in [2] for a linear workflow context, in the presence of both fail-stop and silent faults. Finally, in a recent study, the authors of [21] explore the combination of replication with checkpointing for fail-stop errors, and compute the optimal checkpoint interval for this approach.

3. Description of the methodology

The following subsections are dedicated to the description of the basics of the different proposed options to accomplish transient fault tolerance. In addition, an evaluation of the temporal behavior of implementing each specific feature is also included. In that sense, a simple model has been developed which considers the factors that have influence over the total execution time, both in the absence of faults as when a single silent error occurs during the execution. It is important to remark that SEDAR can functionally manage multiple fault occurrence [22]. However, the proposed performance model contemplates a single fault for the sake of clarity.

To evaluate the different strategies, a baseline is used, which consists of a manual method for ensuring reliable results. This method involves launching two simultaneous instances of the application and comparing the final results of both executions in a semi-automatic manner. In this way, the same computing resources that are consumed in our proposal are assigned to each individual instance (i.e. half of the total cores of the system), which is the fairest way to compare. In the absence of faults, the final results will match. However, if a transient fault occurs, a third re-execution (maintaining the same mapping) and a new comparison are required to pick the outputs of the runs that form a majority (using a voting mechanism) as the correct ones.

Table 1
Name and meaning of the parameters involved in temporal characterization of each alternative strategy.

Parameter	Meaning
T_{prog}	The execution time of two instances of the original application in parallel
T_{comp}	Time of semi-automatic comparison of results; may include calculating a hash
T_{rest}	Time of manually restarting the application. An automatic restart may take shorter. In simplified model, it is considered the same
f_d	The factor of overhead due to detection mechanism. It is application dependent and can be experimentally determined. $0 < fd < 1$
X	Instant of fault detection, expressed as a fraction of the application progress. Random $0 < X < 1$
n	Number of checkpoints made during the whole execution, given a checkpoint interval
t_{cs}	Time involved in storing a system checkpoint
t_i	Checkpoint interval. It can be adjusted to minimize overhead
k	A number of additional checkpoints that the application needs to rollback to find a non-corrupted one. It depends on the application and the detection latency
t_{ca}	The time involved in storing an application checkpoint. It should be shorter than t_{cs}
T_{compA}	The time for validating an application checkpoint; it may include calculating a hash

The time elapsed by this manual method in the absence of faults (fault absence, T_{FA}) is given by Eq. (1). It consists of the time spent by the two independent instances to run simultaneously (T_{prog}) plus the time of comparing the results of the two executions. On the other hand, Eq. (2) is the time when a fault occurs (fault presence, T_{FP}), which is the time of a new re-execution and a new comparison for voting, besides a restart time (T_{rest}) to relaunch the third run (which takes the same time as the previous one because it uses the same mapping) after the two original ones. In Table 1, the parameters involved in all the equations and their meanings are summarized.

$$T_{FA} = T_{prog} + T_{comp} \tag{1}$$

$$T_{FP} = 2(T_{prog} + T_{comp}) + T_{rest} \tag{2}$$

3.1. Error detection with notification

In order to accomplish detection when running deterministic parallel applications, which is the first SEDAR feature, the messages between processes are validated before being sent. Thus, the error that affects any process can be isolated, preventing it from propagating to the others.

The detection strategy consists of duplicating each application process in a thread, which requires a synchronization mechanism between both replicas. Every time a communication is to be performed, the leading thread stops running and then waits for its replica to reach the same point. Once there, the detection mechanism compares the entire contents of the messages computed by both redundant threads and, if there is a coincidence, only one of them sends the message. Such a mechanism does not require additional network bandwidth. When the receiver process reaches the receive operation, it gets the message and in turn waits for its replica to synchronize. Next, it makes a copy of the received contents before resuming execution. Additionally, because a failure may have locally propagated until the end of the execution, a comparison of the final results of the application is performed, thus allowing to detect such failures.

This detection strategy is capable of detecting failures that cause SDC (both TDC and FSC variants, at the cost of sacrificing latency in the latter case) and TOE. As regards TOE, they can be detected under the assumption that the execution time of two redundant threads is similar in a homogeneous, dedicated system [22]. Hence, if an appreciable delay is noticed between the two replicas, it is considered that a silent error has caused the separation of their flows. As a single time-out interval is not optimal, it should be configured taking into account the application's needs: if it is too long, the detection latency enlarges, but being too short may cause false positive detection. Anyway, a TOE is definitely detected if a process enters an infinite loop.

A scheme of the proposed detection strategy is shown in Fig. 1. Each replica runs in a core that shares a cache level with the core in which the original process executes. Thus, the comparisons are solved with no need to access main memory, taking advantage of the memory hierarchy. It should be clear that the proposed detection mechanism is based on launching a single instance of the application, with each process internally replicated in a thread. This is different from the baseline, in which two independent instances of the application are launched in parallel. Nevertheless, both cases make the same use of half of the available cores from the application performance point of view.

As this methodology is based on process-replication, it is *a priori* capable of performing only detection (triple redundancy needs to be used to achieve correction). The ways to accomplish recovery without the need of triplicating are described in the two next subsections.

Implementing any resilience strategy involves unavoidable costs, both in execution time and in resource utilization. In particular, a duplication-based mechanism aims to achieve reliability, at the cost of assigning half of the system cores to protect the executions. In this context, it is important to note that SEDAR provides fault tolerance without introducing any additional cost regarding resource utilization; it takes advantage of the available cores (the intrinsic redundancy of the system), without any change or need for specific hardware. As a consequence of the problems related to strong scalability, parallel applications may not always make an efficient use of all the available cores, both in terms of time reduction and energy consumption. Therefore, using all the available cores may not necessarily be better (as expected) than using half of them (some examples of this behavior can be found in [23,24]). In addition, as performance can depend on the mapping, providing resilience is a useful way to take advantage of the available resources.

Another remarkable aspect is that SEDAR does not modify the algorithm (as algorithm-based detectors do) and it is almost transparent to the application, which makes it generally usable for message-passing applications.

The steps involved in the detection, such as the replication of processes, synchronization between redundant threads, comparison before sending, copying messages upon reception and final verification of the results, are the cause of the overhead introduced in the overall execution time.

A trade-off is reached in relation to the validation interval. The overhead involved in detection is minimized if results are compared only at the end, but because the detection latency increases, a lot of computation could be useless. On the other hand, if the frequency of partial results validation is high, the introduced overhead enlarges but the fault can be quickly detected, and hence more computation is profitable. There is evidence that, depending on the computation-to-communication ratio of the particular application, as well as on the size of the workload, the overhead can differ in a significant manner [13]. As a consequence, the combination of the validation of the messages with the final comparison of results aims to detect all faults and obtain a reliable system by introducing a reasonable overhead. SEDAR's detection mechanism could also be adapted to partial replication, similar to [16]. However, extra work should be done to identify the application's critical parts that need to be replicated, which is a non-general procedure. Consequently, SEDAR could be disabled in the non-critical parts. Moreover, the partial replication would not result in a benefit regarding resource utilization, since the cores were previously assigned to replicas (whether they run or not). Nevertheless, it could be potentially profitable from the standpoint of energy consumption.

In the absence of a recovery strategy, the occurrence of an SDC or a TOE causes the detection-only strategy to notify the user and lead the system to a safe stop, preventing it from delivering defective results. As validating the messages has the effect of limiting the detection latency, the implementation of such a strategy permits relaunching the execution as soon as the error is detected, thus avoiding the needless and expensive wait for the termination with corrupted results. The execution time of the detection strategy in the absence of faults is given by Eq. (3). The time is the same as the one of the baseline (Eq. (1)), but, in this case, T_{prog} is negatively affected (increased) by a factor f_d, which represents the overhead of the detection mechanism. If a fault occurs, the execution time is accounted in Eq. (4). The first term comprises the time executed until the detection instant (X) plus the whole re-execution after the stop caused by the error. Once it is detected, a restart is required, and, in the re-execution, the final comparison is needed.

$$T_{FA} = T_{prog}(1 + f_d) + T_{comp} \tag{3}$$

$$T_{FP} = T_{prog}(1 + f_d)(X + 1) + T_{rest} + T_{comp} \tag{4}$$

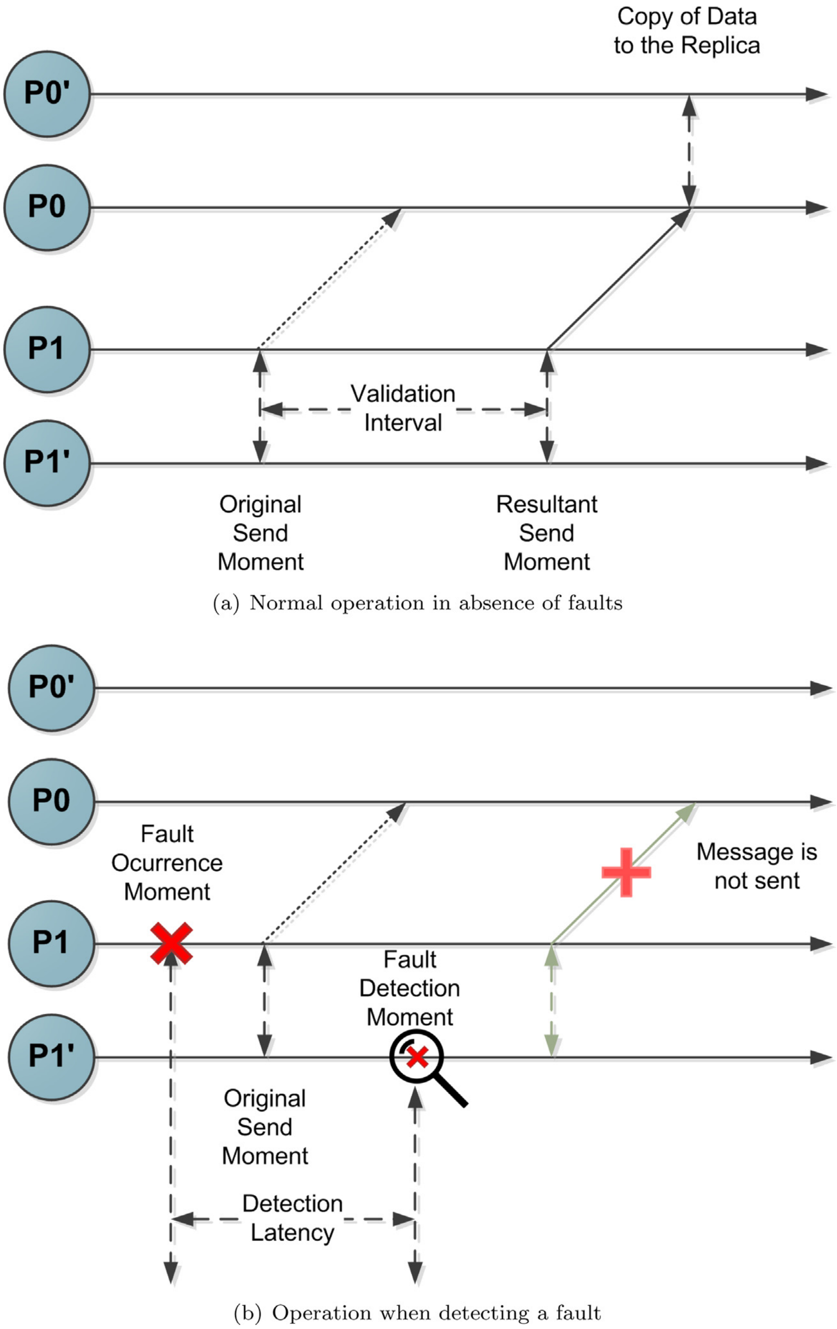
It is important to point out that the parameter X represents the instant of the fault detection and not the moment of the fault occurrence (which cannot be exactly known). The value of X is related to the latency of detection and depends on how the data (and then the messages) are affected by the fault, so it varies according to the communication pattern of the target application.

A remarkable fact is that the proposed detection strategy is equally effective when multiple errors occur during the execution. The first difference, caused by an error, which is observable in the contents of a message or in the final results will lead to the system stopping safely. The vulnerability of this mechanism is reduced to extremely unlikely cases, which are detailed in [22,25]. Consequently, despite the fact that we have limited the analysis of the temporal behavior to the cases of fault absence or single error occurrence, the detection strategy can handle multiple non-related errors. As regards the implementation, the required procedure for adding the detection functionality to the parallel application (that could be automated) is detailed in [13].

3.2. Recovery based on multiple system-level checkpoints

The next step in the search for transient fault tolerance consists of adding a recovery mechanism. In SEDAR, it is proposed to store a chain of distributed coordinated checkpoints, built with a system-level checkpointing library.

It is not possible to ensure that any particular checkpoint holds a safe state for recovery because a silent error can spoil



(a) Normal operation in absence of faults

(b) Operation when detecting a fault

Fig. 1. Outline of the operation of the detection mechanism.

the internal state of one of the replicas that is going to be checkpointed. Therefore, recovering from the last stored checkpoint is not always feasible and using an older one may be required. Thus, multiple checkpoints have to be saved to guarantee recovery [8]. As the transient faults are fleeting, it is important to note that the restart can be attempted from the same node where the corruption took place. There are two possible cases:

1. *The transient fault occurs and is detected inside the boundaries of a checkpoint interval.* In this situation, the last checkpoint can be used to resume the execution. As a particular case, if the detection occurs previously to the first checkpoint, the application must be relaunched from the beginning. This situation is outlined in Fig. 2(a).
2. *The detection latency transposes the limits of the checkpoint interval*. This circumstance arises when the fault occurs before storing a checkpoint but the detection takes place after that. In this situation, the last checkpoint is invalid, so the corresponding restart causes the same error to manifest. Consequently, the previous checkpoint must be used to attempt to recover. Generalizing, the fault can traverse any number of checkpoints (depending on the detection latency), requiring several tries to make rollback recovery possible. In turn, this situation is outlined in Fig. 2(b).

Controlled fault injection experiments are needed to verify the operation of the recovery strategy when the two aforementioned cases occur. The data corruption becomes evident as an observable difference between the memory state of the replicas. Hence, in order to simulate a bit-flip in a processor register, the value of a variable is changed in only one of the replicated threads, in a single iteration of the computation. Such an injection is made from inside the code of the application. The details of the injection method are described in Section 4.2. As regards the checkpoints, the best moments to take them are when the communications have just been validated. This strategy reduces the windows of vulnerability [25] (we have previously studied this issue in [22]), as the probability of an error corrupting the state of a checkpoint is smaller in that situation. However, the mechanism even works if the checkpoints are taken somewhere, as in a more realistic scenario, with periodic checkpointing made externally to the application. However, in general, a considerable overhead would be involved if a checkpoint is taken after each communication.

As mentioned before, the two possible behaviors of the recovery strategy are outlined in Fig. 2, while Algorithm 1 describes the pseudo-code of the proposed method. For the sake of clarity, the injection mechanism is not included in the algorithm, nor is the recording of system checkpoints; in general, they can be taken anytime.

Algorithm 1 Recovery algorithm with multiple system-level checkpoints

```
/* extern_counter is an external counter that controls the number of rollbacks (not included in checkpoint) */
int extern_counter = 0;
/* fault_detected is a boolean variable that reports if a failure was detected in the last execution */
boolean fault_detected = FALSE;
/* run parallel application app under SEDAR monitoring */
SEDAR_run(app);
/* if fault_detected is TRUE then a fault was detected in the last execution */
while fault_detected == TRUE do
    /* extern_counter is increased by 1 */
    extern_counter++;
    /* get the number of checkpoints done */
    ckpt_count = get_ckpt_count();
    /* calculate the number of the checkpoint for restart */
    ckpt_no = ckpt_count - extern_counter;
    /* reset detection flag for restart */
    fault_detected = FALSE;
    /* restart app from checkpoint ckpt_no */
    SEDAR_restart(app, ckpt_no);
end while
```

Automating this method would increase its usability. This can be accomplished if an outsider process is allowed to read the *extern_counter* and, based on its value, find the correct restart script to try recovery; besides that, the wrong-restart checkpoint has to be erased (and stored again in re-execution). In turn, the code of fault injection is executed only once, and the target application modifies the value of *extern_counter* every time a fault is detected.

It is worth noting that this mechanism is also able to detect multiple faults (if they are independent of each other) [22]. If a different fault is detected during a re-execution, the algorithm will recover but at a sub-optimal computational cost. In the current state of the proposal, the algorithm is optimized to deal with a single error. So, when any error is detected in a re-execution, the algorithm assumes that it is the same as was previously detected (the detection latency has exceeded the checkpoint interval, as mentioned in Section 3.2). Therefore, the detection of a different error during re-execution will generate an unnecessary rollback attempt (i.e. the algorithm assumes that the last checkpoint is corrupted, although this is not the case). In the worst-case situation, the recovery mechanism goes back to the beginning. While predicting the recovery time becomes difficult in the case of multiple faults, a reliable conclusion is still ensured. The described inefficiency can be fixed by adding a more sophisticated mechanism, which is briefly described in Section 4.2.

In the absence of faults, the execution time of this mechanism is given by Eq. (5). Compared with the only detection case (Eq. (3)), the extra term accounts for the time involved in saving n system-level checkpoints. When a fault occurs, the execution time is the one shown in Eq. (6). The parameter k is the number of extra checkpoints that need to be reversed if the restart from the last one does not succeed. Therefore, the third term represents the time spent in checkpointing, taking into account that various checkpoints (k) might be recorded again if there is any corruption that prevents successful recovery. The fourth term is an estimate of the re-execution time, considering that, on average, the fault may be detected midway through the checkpoint interval. This lapse will need to be re-executed in the best case, (i.e. when recovery is possible from the last stored checkpoint ($k = 0$)). If $k > 0$, the same portion plus a number of checkpoint intervals (which depend on the value of k) will require being re-executed. Finally, the last term represents the number of needed restarts.

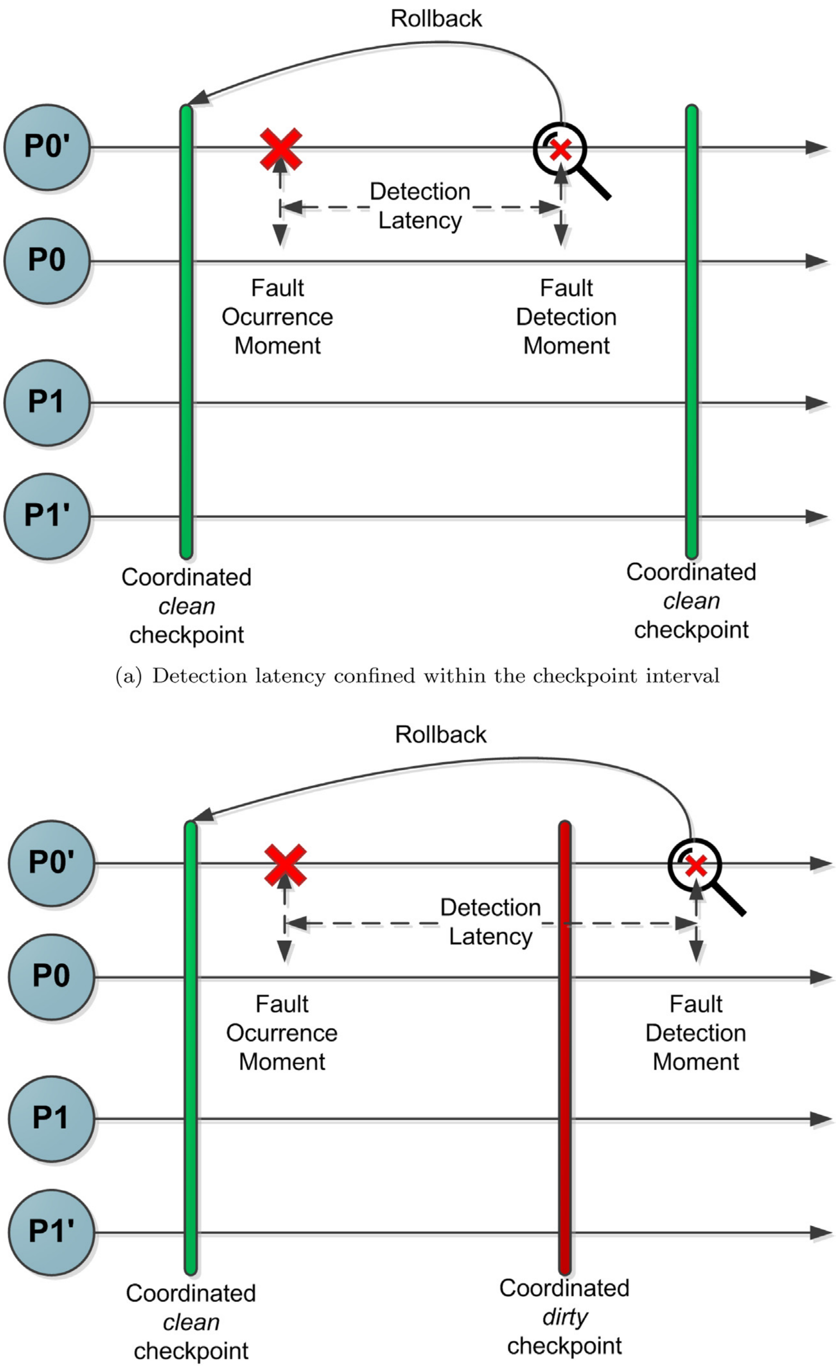
$$T_{FA} = T_{prog}(1 + f_d) + T_{comp} + nt_{cs} \tag{5}$$

$$\begin{aligned} T_{FP} &= T_{prog}(1 + f_d) + T_{comp} + (n + k)t_{cs} + \\ &+ (\sum_{m=0}^{k}(k - m + 1/2))t_i + (k + 1)T_{rest} \end{aligned} \tag{6}$$

When system-level checkpoints are the only available option, this strategy becomes appropriate, despite having two significant limitations. The first one refers to the amount of required storage. None of the checkpoints can be erased, given the uncertainty about the validity of the data recorded in them: if a consistent checkpoint cannot be found, a significant part of the application will need to be re-executed; in an extreme case, the whole execution will have to be relaunched from the beginning [8]. In any case, the negative impact of multiple checkpoints over the storage can be reduced by solutions based on multi-level checkpointing [7]. The second important drawback is related to scalability: in upcoming exascale systems, and despite some considerable efforts [26], coordinated-system-level checkpoints would not be the most suitable solution, because they keep a large amount of information related to the system. In an unrefined version, our method is an expensive approach, because it needs to keep an undetermined number of active checkpoints and may require several restart attempts. Instead, user-level checkpoints are becoming more usual, especially due to their lower costs and portability options [1].

3.3. Recovery based on a single safe application-level checkpoint

This third alternative included in SEDAR is designed to overcome the limitations caused by the utilization of system-level checkpoints. In this context, and despite requiring detailed knowledge of the application's internal organization (computing and communication), user-level checkpoints are a more appropriate option, given the fact that they only save the application-related information [4]. Besides that, they are smaller, more portable and scale better than the system-level versions. As a consequence, the utilization of a single user-level checkpoint for recovery is proposed, in conjunction with a strategy to ensure the validity of the last recorded checkpoint. Therefore, the prior checkpoints can be removed, thus decreasing storage usage and reducing the relaunching latency.



(a) Detection latency confined within the checkpoint interval

(b) Detection latency transposing the checkpoint interval

Fig. 2. Possible cases of recovery using multiple system-level checkpoints, depending on the detection latency.

The proposed solution is based on recording per-thread user-level checkpoints, taking advantage of the synchronization mechanism between replicas. Such checkpoints just save the set of variables that are significant to the application at that specific moment. As both thread checkpoints are stored, a hash on each one is calculated. To collate the two hashes, the mechanism used for validating message contents in the detection phase is employed again. Hence, the checkpoint is considered *valid* only if the comparison proves to be true. In this situation, the previous checkpoint can be safely discarded to save storage, as the current one constitutes a consistent state for recovery. On the other hand, if a difference is detected on the verification phase, it is necessarily due to a fault that has occurred within the last checkpoint interval. As this checkpoint is considered *corrupted*, it is not possible to use it for recovery and it should be erased. Then, the execution has to be resumed from the previous checkpoint. As a consequence, there is a single valid checkpoint at any given time (except for the validation interval), which is independent from the comparison result. Algorithm 2 describes the pseudo-code of the proposed mechanism.

In the absence of faults, the execution time of this mechanism is given by Eq. (7). The time is equal to the detection-only strategy, but incorporates an additional last term which represents the time employed for n user-level checkpoints to be recorded (t_{ca}) after being validated (t_{compA}). Instead, Eq. (8) accounts the time when a fault occurs. The fourth added term shows that, on

Algorithm 2 Recovery algorithm with application-level checkpoints

```
function USR_CKPT(n)                         // usr_ckpt function definition
    for (tid=0; tid $<$ 2; tid++) do         // for both replicas
        /* record its custom checkpoint */
        store_all_significant_variables(tid);
        hash_array[tid]=compute_hash(tid);
    end for
    synch_threads();                         // wait for each other
    /* only one of the replicas compares hashes */
    if tid==0 then
        if hash_array[0]==hash_array[1] then // they match
            /* delete own checkpoint */
            remove_all_significant_variables(tid);
            /* this is a valid checkpoint so the previous can be discarded */
            return TRUE;
        else
            return FALSE                     // this is a corrupted checkpoint
        end if
    end if
end function

(...)                                        // application code
/* $n$ represents the current checkpoint */
if usr_ckpt(n)== TRUE then
    /* delete previous checkpoint since the current is valid */
    remove_usr_ckpt(n-1);
else
    /* remove current corrupted checkpoint */
    remove_usr_ckpt(n);
    /* restart from previous checkpoint */
    restart_from_usr_checkpoint(n-1);
end if
```

average, just half of the checkpoint interval has to be re-executed, as barely a single rollback is required. For the sake of clarity: as each checkpoint is validated, the latency of detection is confined within the checkpoint interval. In the worst-case scenario, the re-execution time will be t_i, if the error is detected just before taking a new checkpoint; while, in the best-case scenario, in which the error is detected as soon as a checkpoint has been taken, it will be near to 0. As the probability of an error is equally distributed along with the checkpoint interval, we state that, on average, the re-execution time is (1/2) t_i. Ultimately, the last term represents the only restart time that is required, as the algorithm performs a single rollback at most. It is important to notice that T_{comp} represents the time required for the validation of application results, while T_{compA} comprises the time required to validate an application-level checkpoint.

$$T_{FA} = T_{prog}(1 + f_d) + T_{comp} + n(t_{ca} + T_{compA}) \tag{7}$$

$$T_{FP} = T_{prog}(1 + f_d) + T_{comp} + n(t_{ca} + T_{compA}) + (1/2)t_i + T_{rest} \tag{8}$$

3.4. Average execution time

As previously mentioned, Eqs. (3) to (8) describe the time required by each strategy in two cases: (1) in the absence of faults (T_{FA}); and (2) in the presence of a single silent fault (T_{FP}). As a fault has an associated occurrence probability, we introduce a general formulation that predicts the Average Execution Time considering it and, as a consequence, the *Mean Time Between Errors* (MTBE) parameter. This function allows us to estimate the average overhead introduced by each SEDAR strategy. Let α be the probability of a fault occurrence. Then, the Average Execution Time function is given by:

$$AET = T_{FP}(\alpha) + T_{FA}(1 - \alpha) \tag{9}$$

The *MTBE* of a system with N processors decreases linearly with N, that is $MTBE = MTBE_{ind}/N$, where $MTBE_{ind}$ is the *MTBE* of an individual processor [4]. If $\lambda = 1/MTBE_{ind}$ is the silent error rate of an individual processor, then the silent error rate of the whole system is λN. Assuming that errors occur according to an exponential distribution, the probability of a silent error affecting a computation that lasts T_{prog} and executes on a system with N processors is:

$$P(N, T_{prog}) = 1 - e^{-\lambda N T_{prog}} = 1 - e^{-NT_{prog}/MTBE_{ind}} = 1 - e^{-T_{prog}/MTBE} \tag{10}$$

The latter expression is the probability of a silent fault occurrence, that is, α. By considering this in Eq. (9), we can obtain the Average Execution Time as a function of the *MTBE* of the system and the baseline program execution time T_{prog}.

$$AET = T_{FP}(1 - e^{-T_{prog}/MTBE}) + T_{FA}(e^{-T_{prog}/MTBE}) \tag{11}$$

Eq. (11) is useful for estimating the average overhead of using each SEDAR strategy, considering both the cases of fault absence and fault presence. If Error Detection with Notification strategy is used, T_{FA} and T_{FP} of Eq. (11) are obtained from Eqs. (3) and (4); when using Recovery based on Multiple System-Level Checkpoints, they are obtained from Eqs. (5) and (6); and when Recovery based on a Single Safe Application-Level Checkpointing is the chosen strategy, the times are obtained from Eqs. (7) and (8).

4. An evaluation of the recovery method

4.1. Analytical model

To describe and validate the functional behavior of the detection and automatic recovery strategies in the presence of faults, we have built a model based on combining a well-known test application with a complete, controlled workfault. The analytical model contemplates all the possible faults that can occur, based on the deep knowledge of the behavior of the application. Each fault has a predictable effect, a moment in which it is certainly detected and a determinable point for recovery. Obviously, there are infinite physical possibilities of fault occurrences, but all of them are represented in the listed scenarios; a scenario represents a class of errors, so it includes a set of cases that behave in the same way. For each experiment, a single fault is injected.

The test application is synthetic, built-up over an MPI Master/Worker matrix multiplication ($C = A \times B$). The modifications consist of replicating processes in threads for detection, in addition to final validation of the resulting matrix. Every time the application performs a communication, a system-level checkpoint is carried out as messages are sent only if the involved data are safe. The matrix-multiplication has been selected because it is a regular, computationally-intensive, representative parallel application, with a well-known communication pattern. The deep knowledge about its behavior allows the clear identification of the moments of communication between processes and of the data involved in each communication. As a consequence, the precise effect of each injected fault can be predicted, as well as the state of each recorded checkpoint (*clean* or *dirty*), and, therefore, which checkpoint makes the recovery possible.

The pseudo-code for the test application is shown in Algorithm 3.

Algorithm 3 Pseudo-code for the test application

```
SEDAR_Init()
SEDAR_Ckpt()                 // Checkpoint #0 (CK0)
SEDAR_Scatter(A)             // Master scatters matrix A (SCATTER)
SEDAR_Ckpt()                 // Checkpoint #1 (CK1)
SEDAR_Bcast(B)               // Master broadcasts matrix B (BCAST)
SEDAR_Ckpt()                 // Checkpoint #2 (CK2)
matmul(A,B,C)                // Each process computes its block (MATMUL)
SEDAR_Gather(C)              // Master gathers matrix C (GATHER)
SEDAR_Ckpt()                 // Checkpoint #3 (CK3)
if rank == MASTER then
    /* Master validates final result (VALIDATE) */
    SEDAR_Validate(C)
end if
```

The possible scenarios for injection experiments are organized according to the following criteria:

- P_{inj}: the execution instant where the injection is carried out, taking as reference the structure of the application (e.g. between the SCATTER and CK1).
- *Process*: if the injection is made in the code executed by the Master or any Worker.
- *Data*: if the injection is made either on an element of the matrix *A*, *B* or *C*, or on an index variable. Also, if the injected value is used by the Master or a Worker for its computation (e.g. A(M), C(W), i(M), etc.).
- *Effect*: TDC, FSC, LE or TOE.
- P_{det}: the execution moment where the fault is detected. It could be at communication or at the final validation.
- P_{rec}: the nearest checkpoint from which it is possible to recover.
- N_{roll}: the number of attempts required for correct recovery. The possible values are: 0, if the injected fault causes a LE; 1, if recovery is possible from the last recorded checkpoint; 2, if it is necessary to rewind to the last but one checkpoint; and so on.

Based on the combinations of these factors, we have designed a set of 64 injection experiments that cover all the situations that can occur in the target test application. The 64 scenarios have been designed according to the following criteria: the faults are injected both in the code executed by the Master and by the Workers, in elements of each of the three matrices. The injections in the Master code are made both in data that it transmits and in others that are kept for local use. The injections in the Workers' code are made in data that will be transmitted, as the Workers do not locally retain results. In both the Master and the Workers, there are faults injected after each checkpoint, i.e. the checkpoint is *clean*, so recovery is possible. Nevertheless, other faults are injected after a communication operation but before the subsequent checkpoint (i.e. between them, into already validated data), making that checkpoint *dirty* and forcing more than one rollback to recover. On the other hand, injections in index variables are made, both in the Master and the Workers codes, during the actual matrix-multiplication operation, in order to make the processing time of both redundant threads asymmetric. As previously mentioned, every fault affecting a certain subset of data, and occurring (at any moment) during the lapse of the execution comprised in a particular scenario, is detected at the same time, so this it can be recovered from the same checkpoint. In other words, any case of error has a similar effect to one of the 64 provided scenarios; these 64 scenarios are derived from the study of the application behavior.

Table 2
Selected representative injection scenarios: effects and predicted points of detection and recovery.

Scenario	P_{inj}	Process	Data	Effect	P_{det}	P_{rec}	N_{roll}
2	CK0 - SCATTER	Master	A(W)	TDC	SCATTER	CK0	1
29	BCAST - CK2	Worker	C(W)	LE	–	–	0
50	GATHER - CK3	Master	C(M)	FSC	VALIDATE	CK2	2
59	MATMUL	Worker	i(W)	TOE	GATHER	CK2	1

To illustrate the method followed, only a few representative scenarios are detailed in Table 2. These scenarios were selected mainly to make evident the four possible effects of a fault (TDC, FSC, LE or TOE), but also to display injections made in the codes of both the Master and the Workers, showing different moments of detection and reflecting various possible situations of recovery (i.e. no need to rollback, rollback to the last checkpoint or multiple rollback attempts).

From Table 2 we can observe that:

- In Scenario 2, the injection is carried out modifying matrix *A* from the Master process between CK0 and SCATTER stages. The injected element of *A* is going to be transmitted to a Worker, so this injection produces a TDC error, which will be detected at SCATTER moment. The recovery is possible from the last checkpoint carried out (CK0, a *clean* checkpoint).
- An example of LE error is described in Scenario 29. The injection happens between BCAST and CK2 affecting matrix *C* from a Worker. As this matrix has not been computed yet and will be overwritten afterwards, the error does not modify the final result.
- Scenario 50 shows the details for an injection experiment that causes a FSC error. The injection is made in an element of matrix *C* that has already been calculated and received by the Master (GATHER), but before making the checkpoint CK3. The error will be detected at the VALIDATE stage, and, because CK3 is *dirty* (the fault occurred before recording it), an additional rollback is required to recover.
- A possible TOE error is detailed in Scenario 59. This injection takes place at MATMUL stage and affects an index variable used by a Worker, causing one of the replicas to restart its computation after it has already done part of its task. This causes a delay in the affected thread, which is detected as a TOE; only the other replica reaches the GATHER operation within a configurable lapse. The recovery is built from the CK2 point (a single rollback is enough).

A conclusion of our analysis is that any random fault that can occur along the execution resembles some of the modeled scenarios. A method of analysis like the one described, based on the knowledge of the target application and the moments of checkpointing, in combination with controlled and systematic fault injection, allows us to predict the behavior of both the detection and the automatic recovery mechanisms. Thus, the efficiency of the strategy is shown when running in a real environment with random faults.

It is to be noted that the aim of the performed functional analysis is the evaluation of the efficacy of SEDAR's detection and recovery mechanisms. As the correct operation of both mechanisms is checked for all the errors that can occur in the particular selected test application, and other types of errors do not exist, the carried-out validation verifies the functional suitability of SEDAR.

4.2. Results over a real implementation

All the experiments that are described in this section have been carried out utilizing standard tools. The implementation of the fault tolerance strategy consists of a library of modified MPI functions and data types with extended functionality for fault detection. This includes abuffers comparison before sending, message copies upon reception, and synchronization between replicated threads.

The coordinated system-level checkpoints are built with the DMTCP library [27], which generates distributed-per-process checkpoint files, and a single restart script for each checkpoint. All processes of the application call SEDAR_Ckpt(), but a single process (e.g. the Master) is in charge of checkpointing the whole application. Both redundant threads of this process synchronize with each other (in the same way that they do when a message is to be sent), and only one of them calls DMTCP_Ckpt( ) from inside of SEDAR.

The fault injection is made from inside the test application. An *ad-hoc* function is included in the library, which contains the 64 scenarios, and a conditional compilation is used to make a single injection in each experiment. Depending on the number of the particular injection scenario, the function is invoked in a different place during the execution. This function works in conjunction with a file that is used to control if an injection has been already made (named *injected.txt*). The content of this file is evaluated in each function call. In the first instance, the file contains the value *0* and when the injection is made, its content is *incremented* (it is changed to *1*). In the recovery process, the code is re-executed calling the injection routine again. As the file content is re-evaluated, the function returns without making a new injection since it has changed. This flag needs to be external to the application so that its content is independent of the checkpointing storage (i.e. when a checkpoint is made, the value in the file is not affected).

The recovery mechanism, based on keeping a chain of checkpoints, uses another file to control how many times the same fault has been detected (named *failures.txt*). This file is initialized with *0* and its content is incremented each time that a fault is detected. Assuming that a single error occurs during execution, then the file contains *1* for the first detection, and a rollback to last restart script is tried. If an error is detected during re-execution, it is assumed that it is the same error as the previous one and that the checkpoint is *dirty* due to the detection latency. In this case, the file's content becomes *2* and a rollback to the prior checkpoint is attempted. Therefore, the content of the file is used to choose the number of the restart script that has to be executed. Once again, the file needs to be external to the application, so that its content is independent of the checkpoint storage.

To recover from multiple different faults, the mechanism should be slightly modified to achieve a better performance. Some additional data, related to the current fault, might also need to be stored in the file, in order to be able to distinguish between a repetition of the previous fault and a new fault. In the latter case, the file size would not increase and the mechanism would behave as in the "first" detection scenario.

The 64 injection experiments, each one being representative of a particular scenario, were performed over the test application, using two nodes of a Blade cluster with eight nodes. Each node contains two quad-core Intel Xeon e5405 2.0 GHz processors (6 MB L2 cache, shared between pairs of cores), 10 GB RAM memory (shared between both processors) and 250 GB local disk storage. The operating system is GNU/Linux Debian 6.0.7 (64 bits, kernel version 2.6.32), the message-passing library is MPICH (version 3.3.1), and the checkpointing library is DMTCP (version 2.4.4).

Fig. 3 shows the output file of one of the injection experiments, namely Injection Scenario 50, in order to demonstrate the followed methodology and the obtained behavior.

In order to provide more detail, in this stage of SEDAR's functional validation, our implementation of fault-tolerant MPI functions is based on point-to-point communications, which makes it more general and allows us to show up the FSC scenarios, at the expense of sacrificing performance. However, we have also developed versions of optimized collective communications, which are used in the stage of the temporal behavior evaluation. It is important to note that in collective communications, the sender process also participates in these, either sending or receiving. If our test application just uses collective operations, the corrupted data gets transmitted and hence it is validated. In this way, only TDC scenarios remain and FSC scenarios should not be present any longer. It is clear that this idea could be extended to many other applications.

4.3. Evaluation of the temporal behavior

To show how our descriptive model can be used to evaluate the temporal behavior of each alternative, a set of simple examples is presented, which includes real measured values for the parameters in Table 1, taken from carried-out tests.

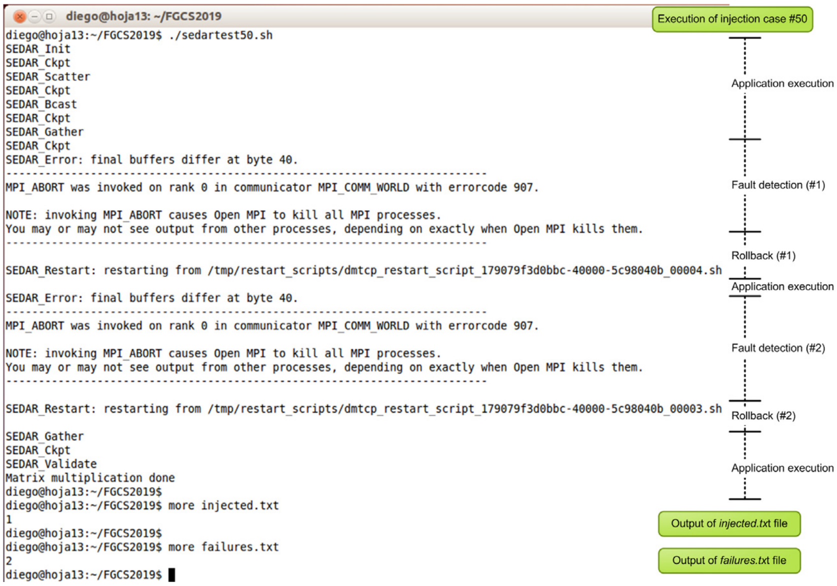
To enrich the analysis, we have used three parallel benchmarks for the testing stage: matrix multiplication; Jacobi's method for Laplace's equation [28]; and DNA sequence alignment with Smith–Waterman algorithm [29]. These applications have been selected because they are well-known, computationally intensive, and representative of scientific computing. In addition, they allow us to study the effects of having different communication patterns: Master–Worker, Single-Program-Multiple-Data (SPMD) and Pipeline, respectively [13]. The comparison between execution times of the three target applications, between raw MPI versions and our MPI-based implementation of SEDAR, has the aim of measuring the execution parameters with each application and pointing out the differences according to the distinct communication patterns. The SEDAR implementation includes mechanisms for duplicating processes, synchronizing replicas, comparing and copying the messages' contents, and validating the final results. Each experiment has been repeated five times, and average values have been taken.

The parameter f_d, the detection overhead, was obtained by comparing the execution time of the SEDAR detection mechanism, in the absence of faults, with the time of executing the manual detection strategy (baseline), for each target application. Explicitly:

$$f_d = \frac{T_{SEDAR\_det\_FA} - (T_{prog} + T_{comp})}{T_{prog} + T_{comp}} \qquad (12)$$

where $T_{SEDAR\_det\_FA}$ is the time of Eq. (3) and $(T_{prog} + T_{comp})$ is the time of Eq. (1).

As regards the parameter T_{comp}, it has been measured for a manually-launched program that compares the contents of two binary files. It is similar to the one required by the function SEDAR_Validate(). An average value of the parameter t_{cs} was obtained by measuring with tools provided by DMTCP library. As regards T_{rest}, it has been indirectly measured (by fault injection experiments which demand recovery), but the obtained values



```
diego@hoja13:~/FGCS2019$ ./sedartest50.sh
SEDAR_Init
SEDAR_Ckpt
SEDAR_Scatter
SEDAR_Ckpt
SEDAR_Bcast
SEDAR_Ckpt
SEDAR_Gather
SEDAR_Ckpt
SEDAR_Error: final buffers differ at byte 40.
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD with errorcode 907.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on exactly when Open MPI kills them.
--------------------------------------------------------------------------

SEDAR_Restart: restarting from /tmp/restart_scripts/dmtcp_restart_script_179079f3d0bbc-40000-5c98040b_00004.sh

SEDAR_Error: final buffers differ at byte 40.
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD with errorcode 907.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on exactly when Open MPI kills them.
--------------------------------------------------------------------------

SEDAR_Restart: restarting from /tmp/restart_scripts/dmtcp_restart_script_179079f3d0bbc-40000-5c98040b_00003.sh

SEDAR_Gather
SEDAR_Ckpt
SEDAR_Validate
Matrix multiplication done
diego@hoja13:~/FGCS2019$
diego@hoja13:~/FGCS2019$ more injected.txt
1
diego@hoja13:~/FGCS2019$
diego@hoja13:~/FGCS2019$ more failures.txt
2
diego@hoja13:~/FGCS2019$ █
```

Fig. 3. Output of the execution of the test application when the fault of Scenario 50 is injected.

Table 3
Values of the parameters utilized in the temporal evaluation of each alternative strategy.

Parameter	MATMUL	JACOBI	SW
T_{prog} [hs]	10.21	8.92	11.15
T_{comp} [s]	42	1	<1
f_d [%]	<0.01	0.6	0.05
X_1; X_2; X_3 [%]	30; 50; 80		
t_i [hs]	1		
n	10	8	11
W [MB]	6016	1920	152
t_{cs} [s]	14.10	9.62	2.55
T_{rest} [s]	14.10	9.62	2.55
t_{ca} [s]	10.58	9.11	1.92
t_{compA} [s]	42	1	<1

were consistent to the assumption that the time needed to perform a checkpoint can be considered equal to the time needed to load a checkpoint from storage [30].

To achieve long-lasting executions for the three applications (around 10 hours for the baseline case), we have adjusted the execution parameters. The matrix product has been repeated 100 times using N $=$ 8192. For the Jacobi algorithm, the size of the workload was N $=$ 8192 and the number of iterations I $=$ 300.000. Finally, for DNA sequence alignment, the length of the sequences was set to N $= 2^{22} =$ 4.194.304. In this context, the parameter t_i has been fixed to 1 hour for all the experiments. Regarding n, this is the number of checkpoints that have been recorded with such an interval value, and it is obtained by dividing the time of the only detection strategy (Eq. (3)) by the checkpoint interval t_i. It is worth noting that (instead of being arbitrarily assigned) the checkpoint interval can be determined, for example, with Daly's formula [31], which takes into account the *MTBE*. The checkpoint interval is intended to be a trade-off between maintaining a low overhead, due to checkpointing, and a reasonable time of rework if a fault occurs.

The parameter X, which depends on the detection latency, has been manually assigned for three cases. On its behalf, the parameter t_{ca} was estimated assuming that an application-level checkpoint is more lightweight for storing than its system-level counterpart. Last, T_{compA} (the validation time of an application-level checkpoint) has been estimated to be equal to the time of validating the results (i.e. parameter T_{comp}), as a way to simplify the model.

Table 3 contains the list of all values used for the temporal evaluation, obtained as described, whereas Table 4 shows the resulting times in each case.

It may be recalled that in the manual strategy (execution of two simultaneous MPI independent instances), a total of eight MPI processes have been launched, with a maximum of four processes mapped in each node, which means that just four cores have been used in that node. The same mapping was assigned in SEDAR implementation, but in this case, all the cores of each node have been used, as the redundant threads run on free cores.

An analysis of the data contained in Table 3 reveals some remarkable facts. In all the cases, the detection overhead f_d is very low. However, the biggest value is for Jacobi's method, which is the application with the most frequent communications. This is the expected behavior since f_d is tightly associated with messages. On the other hand, the matrix product, which is computation bounded, presents negligible detection overhead.

On the other hand, t_{cs} is directly related with the size of the workload W for each application, and therefore with the amount of memory spent, as can be seen in Table 3. The matrix product is the most memory-consuming application. The Master process handles the three entire matrices, whereas each Worker handles the entire B matrix, plus its corresponding chunks of A and C. As regards Jacobi's method, one of the processes handles the two entire matrices, whereas the other processes handle their corresponding chunks of them. Finally, in Smith–Waterman algorithm, all the processes require local buffers; one of them handles two entire sequences, whereas the others handle one entire sequence plus their corresponding chunk of the other. It may be recalled that for the three benchmarks, all the processes are replicated.

Finally, T_{comp} is associated with the size of the results that have to be validated. In the matrix product, the entire matrix C is validated for each instance, so the obtained value is significant compared to the other applications. On the other hand, in the DNA sequence alignment, only the similarity score has to be validated, which involves negligible time. As a middle ground, in Jacobi's method, a single matrix needs validation.

The information shown in Table 4 allows us to survey interesting aspects of SEDAR's behavior. As a remarkable fact, when

Table 4
Execution times [hs] of all SEDAR alternative strategies, both in the absence and in the presence of faults, compared with the baseline.

#	Situation	MATMUL	JACOBI	SW
1	Baseline, without fault (Eq. (1))	10.22	8.92	11.15
2	Baseline, with fault (Eq. (2))	20.45	17.85	22.35
3	Only detection, without fault (Eq. (3))	10.23	8.97	11.16
4	Only detection, with fault (Eq. (4), X = 30%)	13.29	11.67	14.50
5	Only detection, with fault (Eq. (4), X = 50%)	15.33	13.46	16.73
6	Only detection, with fault (Eq. (4), X = 80%)	18.39	16.16	20.08
7	Multiple checkpoints, without fault (Eq. (5))	10.26	9.00	11.17
8	Multiple checkpoints, with fault (Eq. (6), k = 0)	10.77	9.50	11.66
9	Multiple checkpoints, with fault (Eq. (6), k = 1)	12.27	11.01	13.17
10	Multiple checkpoints, with fault (Eq. (6), k = 4)	22.79	21.53	23.67
11	Single checkpoint, without fault (Eq. (7))	10.37	8.99	11.16
12	Single checkpoint, with fault (Eq. (8))	10.87	9.50	11.66

a fault occurs, the detection mechanism (rows 4, 5 and 6) performs better than the baseline (row 2) for all the applications, regardless of the time of detection, due to the low temporal overhead implied. The sooner the error is detected, the better the mechanism behaves, as expected, because less work must be remade after stopping. It is rather obvious that adding the multiple-checkpoint-based recovery strategy (row 7) involves a larger overhead if compared to the detection-only strategy, in absence of faults; the time spent in checkpointing is worth more in the case of these long-lasting executions, but may have a not-negligible impact in shorter programs.

The analysis of the values in rows 8, 9 and 10 reveals that, when an error takes place, even rolling back several times is advantageous in respect to the baseline; as long as the number of rollbacks is greater than 4, the time spent in reworking is longer than the baseline strategy.

The values shown in rows 11 and 12 are similar to the ones of rows 7 and 8. As expected, the time of recovery from the last valid application-level checkpoint (Eq. (8)) is almost equal to the time of recovery from the last system-level checkpoint, when this is possible (Eq. (6) with $k = 0$).

In conclusion, it can be seen that the temporal behavior of each SEDAR strategy is dependent on the communication pattern, the computation-to-communication ratio and the detection latency. The examples described, with the three benchmarks, demonstrate how the model can be applied for temporal evaluation if the involved parameters are available (or they can be measured).

Another remarkable fact that can be observed from Table 4 is that the different alternatives of SEDAR offer considerable gains both in time and reliability, when facing the occurrence of a silent error. This item becomes particularly important in executions that can last many hours. Moreover, the longer the execution time (T_{prog}), the more useful the fault-tolerance strategy is, because the failures are more likely to happen. As previously stated, the protection mechanism should be used used for long programs: if the execution is too short, checkpoints become worthless. Despite the fact that these examples cannot be taken as general conclusions, they are illustrative of the potential of SEDAR in helping users of scientific applications to reach reliable executions, as they are representative of the scientific parallel applications.

4.4. Convenience of saving multiple checkpoints for recovery

As mentioned before, in a system that saves a chain of checkpoints for rollback, the recovery is then possible after one or more attempts. However, due to the checkpointing and rolling-back overhead, there are possible scenarios in which the time spent in those attempts could be longer than simply stopping upon detection and relaunching from the beginning. Therefore, it is useful to evaluate the convenience of saving multiple checkpoints. Moreover, the benefits of using the checkpoint-based protection should be considered.

This study is suggestive about how to use the developed model. If statistics about the frequency and typical behavior of the faults are available for a particular system that runs an application (i.e. when the faults are more likely to appear), the strategy of protection can be properly tuned between only detection and checkpointing for recovery.

For different values or parameter X, some quantitative knowledge can be extracted from the evaluation of the execution times in Eq. (4) and in Eq. (6). It can be shown that the fourth term in Eq. (6) is equivalent to:

$$(\sum_{m=0}^{k}(k-m+1/2))t_i = \frac{(k+1)^2}{2}t_i \tag{13}$$

so that Eq. (6) can be rewritten as:

$$T_{FP} = T_{prog}(1+f_d)+T_{comp}+(n+k)t_{cs}+\frac{(k+1)^2}{2}t_i+(k+1)T_{rest} \tag{14}$$

To illustrate this idea, we have selected Jacobi's method as the test case (although the procedure can easily be applied to the other benchmarks). For this, we have evaluated Eq. (4) with three different values of X: if the fault is detected near the beginning ($X = 30\%$), in the middle ($X = 50\%$) or toward the end ($X = 80\%$) of the execution. The times obtained are compared with the ones derived from Eq. (14), taking into account the following considerations.

The reference time for this analysis is the one from Eq. (3): the duration of an execution using the detection mechanism, without faults. In such a total execution time (8.97 hs, see Table 4), $X = 30\%$ means that the fault is detected at $t = 2.69$ hs. At this time, with $t_i = 1$ h, only the two first checkpoints (CK0 and CK1) have been stored. Therefore, recovery must be possible, either from CK1 (i.e. $k = 0$) or from CK0 (i.e. $k = 1$), so both values are admissible in Eq. (14). The same reasoning has been followed for the other values of X.

Table 5 summarizes this data. Based on the value of parameter X, the second column shows the times obtained with detection, safe-stop and relaunching from the start (i.e. from Eq. (4)), the third column shows the times obtained by rewinding to the last (by the moment) stored checkpoint (i.e. from Eq. (14) with $k = 0$), the fourth column shows the times obtained by rewinding to the last but one checkpoint (i.e. from Eq. (14) with $k = 1$), and so on. The value *NA* means that the current value of k is

Table 5
Execution time with the fault detected at X, with only detection and with different number of rollback attempts ($k+1$).

X [%]	Only detection [hs]	$k+1$ rollback attempts [hs]				
		k = 0	k = 1	k = 2	k = 3	k = 4
30	11.66	9.5	11.01	*NA* (Not Admissible)		
50	13.46			13.52	17.02	*NA*
80	16.16					21.53

not admissible in Eq. (14) for the current value of X, because the corresponding checkpoint has not been stored yet by that moment of the progress of the execution.

For the analyzed values of X, the obtained results suggest that rolling back to the last stored checkpoint ($k = 0$), if possible, is always advisable, faced with the case of stopping, notifying the error and relaunching from the beginning. Even to restart from the last but one checkpoint ($k = 1$) is still convenient (obviously, if that checkpoint is not corrupted). However, if the fault is detected around the middle of execution, and two (or more) rollbacks have to made, it would have been preferable to stop and relaunch. In other words, if recovery from the last two checkpoints is not possible, trying from the previous ones is still more expensive than simply halting and getting started again. This is caused by the large overheads involved in not only re-executing the same computation several times, but also re-storing checkpoints and making various restart attempts. This trend also holds as the application progresses: if the fault is detected close to the end, even trying more rollbacks to recorded checkpoints could represent an improvement compared to stop and relaunch.

Of course, there is no way of knowing which checkpoint would enable recovery. However, following this line of reasoning, if we force the time of Eq. (4) to be minor or equal to the one of Eq. (14) with $k = 0$, we obtain $X \leq 5.88\%$. This means that, before that level of progress (which represents about 32 min with this set of parameters), it is not convenient to record any checkpoint; it is less expensive simply to stop and relaunch. On the other hand, if we force the time of Eq. (4) to be greater or equal to the one of Eq. (14) with $k = 1$, we obtain $X \geq 22.67\%$, which represents about 2 h. This means that only when that percentage of execution time has elapsed, rolling back to the last but one checkpoint is preferable than stopping and relaunching. Before that, it is worth storing only a single checkpoint. This result reinforces the idea that this strategy is not so useful if the overall execution time is too short: the time required for storing the checkpoint could be non-negligible if compared with the required checkpoint interval. As a final example, when $X \geq 50.61\%$, rolling back up two checkpoints is beneficial when compared to using the detection-only mechanism.

Despite the fact that this cannot be taken as a general conclusion, the analysis above shows that (with these parameters), the overhead associated with rolling back and re-executing is much more significant than the low cost of saving checkpoints. This suggests that decreasing the checkpoint interval t_i can be convenient, as the advantage of rolling back a shorter span exceeds the checkpointing overhead.

Once again, although these are simple examples, they are illustrative of how useful conclusions can be drawn from the model of temporal behavior, which allow us to adapt the protection strategy based on the knowledge of the system parameters.

5. Conclusions and future work

Exascale computing presents several challenges to future generation computer systems and guaranteeing reliability is one of them. The protection of the MPI applications at message level is a feasible and effective method for detecting, secluding and avoiding the propagation of data corruption, taking into account the deep effects that a single transient fault can cause on all processes that communicate. In this article, SEDAR has been presented as a methodology for detecting and recovering from all silent errors, in an agnostic manner to the algorithms. SEDAR consists of three complementary alternatives for only detection, recovery based on multiple system-level checkpoints and recovery based on a single user-level checkpoint. The most remarkable conclusions are:

- The functional behavior in the presence of faults can be analytically described. We have built a model that considers all the fault scenarios on a well-known test application and SEDAR's response facing each scenario, thus showing the validity of the detection and the recovery mechanisms.
- The predictions of the model can be empirically verified. Through controlled fault injection experiments, the reliability provided by SEDAR strategies has been demonstrated.
- The temporal behavior of each SEDAR strategy can be characterized. When obtaining the execution parametersfor applications with different communication patterns and computation-to-communication ratios, it has been shown that the different variants of SEDAR offer benefits both in execution time and reliability. This becomes particularly profitable in long-lasting programs.
- SEDAR can be adapted to a determined cost-performance trade-off. As each SEDAR strategy supplies a particular coverage but also has limitations and implementation costs, choosing between them allows us to adjust to the needs of a particular system.
- The temporal characterization can be used to extract useful protection guidelines. To illustrate this, it has been shown when it is beneficial to employ each SEDAR strategy.
- Both the viability and efficacy to tolerate transient faults in expected HPC exascale systems have been shown.

As ongoing and future lines of work, we can enumerate:

- Emulating non-deterministic calls, which are required to extend the scope of applications that can be protected with SEDAR.
- Performing experimental validation with customized, non-coordinated user-level checkpoints, calculating the optimal checkpoint interval to minimize execution overhead, and measuring the relationship between the latency of detection and the communication pattern.
- Refining the multiple checkpoint-based recovery mechanism to optimally support various faults, and analytically modeling the temporal response in the presence of multiple non-related faults.
- Implementing an automatic adaptation of the recovery strategy, i.e. dynamically starting protection depending on the progress of the execution (based on the reasoning stated in Section 4.4).

As a final aim, integration with scalable architectures that use C/R for permanent fault tolerance [32] should be attempted. As SEDAR also provides scalable options for detection and recovery, fault-tolerance for both types of errors could be achieved for projected exascale systems. It is important to clarify that a production version of SEDAR is being developed, whereas the current implementation remains as a prototype.

CRediT authorship contribution statement

Diego Montezanti: Conceptualization, Methodology, Software, Formal analysis, Writing - original draft, Visualization. **Enzo Rucci:**

Software, Validation, Formal analysis, Data curation, Writing - review & editing, Visualization. **Armando De Giusti:** Resources, Supervision, Project administration, Funding acquisition. **Marcelo Naiouf:** Resources, Supervision. **Dolores Rexachs:** Conceptualization, Methodology, Writing - review & editing, Project administration. **Emilio Luque:** Conceptualization, Methodology, Writing - review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research has been supported by the Agencia Estatal de Investigación (AEI), Spain and the Fondo Europeo de Desarrollo Regional (FEDER) UE, under contract TIN2017-84875-P and partially funded by a research collaboration agreement with the Fundacion Escuelas Universitarias Gimbernat (EUG). In addition, this research has been supported by the Universidad Nacional de La Plata, Argentina, through the Programas de Incentivos. We also would like to thank the reviewers and the editors for their constructive feedback and significant contributions to this work.

References

[1] T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, J. Labarta, Fault-tolerant protocol for hybrid task-parallel message-passing applications, in: Cluster Computing (CLUSTER), 2015 IEEE International Conference on, IEEE, 2015, pp. 563–570.

[2] A. Benoit, A. Cavelan, F.M. Ciorba, V. Le Fèvre, Y. Robert, Combining checkpointing and replication for reliable execution of linear workflows with fail-stop and silent errors, International Journal of Networking and Computing 9 (1) (2019) 2–27.

[3] J. Elliott, M. Hoemmen, F. Mueller, Evaluating the impact of SDC on the GMRES iterative solver, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014, pp. 1193–1202.

[4] A. Benoit, A. Cavelan, F. Cappello, P. Raghavan, Y. Robert, H. Sun, Coping with silent and fail-stop errors at scale by combining replication and checkpointing, Journal of Parallel and Distributed Computing 122 (2018) 209–225.

[5] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, R. Brightwell, Detection and correction of silent data corruption for large-scale high-performance computing, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, p. 78.

[6] A. Shye, J. Blomstedt, T. Moseley, V.J. Reddi, D.A. Connors, PLR: A software approach to transient fault tolerance for multicore architectures, IEEE Transactions on Dependable and Secure Computing 6 (2) (2009) 135–148.

[7] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomputing frontiers and innovations 1 (1) (2014) 5–28.

[8] G. Lu, Z. Zheng, A.A. Chien, When is multi-version checkpointing needed? in: Proceedings of the 3rd Workshop on Fault-Tolerance for HPC At Extreme Scale, ACM, 2013, pp. 49–56.

[9] H. Mushtaq, Z. Al-Ars, K. Bertels, Efficient software-based fault tolerance approach on multicore platforms, in: Proceedings of the Conference on Design, Automation and Test in Europe, EDA Consortium, 2013, pp. 921–926.

[10] G. Bosilca, R. Delmas, J. Dongarra, J. Langou, Algorithm-based fault tolerance applied to high performance computing, Journal of Parallel and Distributed Computing 69 (4) (2009) 410–416.

[11] D. Montezanti, A. De Giusti, M. Naiouf, J. Villamayor, D. Rexachs, E. Luque, A methodology for soft errors detection and automatic recovery, in: 2017 International Conference on High Performance Computing & Simulation, HPCS, IEEE, 2017, pp. 434–441.

[12] S.S. Mukherjee, J. Emer, S.K. Reinhardt, The soft error problem: An architectural perspective, in: High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, IEEE, 2005, pp. 243–247.

[13] D. Montezanti, E. Rucci, D. Rexachs, E. Luque, M. Naiouf, A. De Giusti, A tool for detecting transient faults in execution of parallel scientific applications on multicore clusters, Journal of Computer Science & Technology. 14 (2014) 32–38.

[14] Z. Chen, Algorithm-based recovery for iterative methods without checkpointing, in: Proceedings of the 20th International Symposium on High Performance Distributed Computing, ACM, 2011, pp. 73–84.

[15] M. Shantharam, S. Srinivasmurthy, P. Raghavan, Fault tolerant preconditioned conjugate gradient for sparse linear system solution, in: Proceedings of the 26th ACM International Conference on Supercomputing, ACM, 2012, pp. 69–78.

[16] C. Engelmann, S. Böhm, Redundant execution of HPC applications with MR-MPI, in: Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks, PDCN, 2011, pp. 15–17.

[17] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, T. Brightwell, rMPI: Increasing Fault Resiliency in a Message-Passing Environment, Tech. Rep. SAND2011-2488, Sandia National Laboratories, Albuquerque, NM, 2011.

[18] G. Yalcin, O.S. Unsal, A. Cristal, Fault tolerance for multi-threaded applications by leveraging hardware transactional memory, in: Proceedings of the ACM International Conference on Computing Frontiers, ACM, 2013, p. 4.

[19] X. Ni, E. Meneses, N. Jain, L.V. Kalé, ACR: Automatic checkpoint/restart for soft and hard error protection, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 7.

[20] N. Ali, S. Krishnamoorthy, N. Govind, B. Palmer, A redundant communication approach to scalable fault tolerance in PGAS programming models, in: Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on, IEEE, 2011, pp. 24–31.

[21] A. Benoit, T. Hérault, V.L. Fèvre, Y. Robert, Replication is more efficient than you think, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2019, p. 89.

[22] D.M. Montezanti, D. Rexachs del Rosario, E. Rucci, E. Luque Fadón, M. Naiouf, A.E. De Giusti, Characterizing a detection strategy for transient faults in HPC, in: Computer Science & Technology Series. XXI Argentine Congress of Computer Science. Selected Papers, Editorial de la Universidad Nacional de La Plata (EDULP), 2016, pp. 77–90.

[23] J. Panadero, A. Wong, D. Rexachs, E. Luque, P3S: A methodology to analyze and predict application scalability, IEEE Transactions on Parallel and Distributed Systems 29 (3) (2017) 642–658.

[24] V. Puzyrev, J.M. Cela, A review of block krylov subspace methods for multisource electromagnetic modelling, Geophysical Journal International, 202 (2) (2015) 1241–1252.

[25] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, D.I. August, SWIFT: Software implemented fault tolerance, in: Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, 2005, pp. 243–254.

[26] J. Cao, K. Arya, R. Garg, S. Matott, D.K. Panda, H. Subramoni, J. Vienne, G. Cooperman, System-level scalable checkpoint-restart for petascale computing, in: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems, ICPADS, IEEE, 2016, pp. 932–941.

[27] J. Ansel, K. Arya, G. Cooperman, DMTCP: Transparent checkpointing for cluster computations and the desktop, in: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE, 2009, pp. 1–12.

[28] G. Andrews, Scientific computing, in: Foundations of Multithreaded, Parallel and Distributed Computing, Addison-Wesley, 2000, pp. 527–585, ch. 11.

[29] E. Rucci, A. De Giusti, F. Chichizola, Parallel smith-waterman algorithm for DNA sequences comparison on different cluster architectures, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Vol. 1, PDPTA'11, WorldComp, 2011, pp. 666–672, [Online]. Available: http://worldcomp-proceedings.com/proc/p2011/PDP5014.pdf.

[30] L. Fialho, D. Rexachs, E. Luque, What is missing in current checkpoint interval models? in: 2011 31st International Conference on Distributed Computing Systems, IEEE, 2011, pp. 322–332.

[31] J.T. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, Future Generation Computer Systems 22 (3) (2006) 303–312.

[32] M. Castro-León, H. Meyer, D. Rexachs, E. Luque, Fault tolerance at system level based on RADIC architecture, Journal of Parallel and Distributed Computing 86 (2015) 98–111.

Diego Montezanti received his Ph.D. in Computer Science from the National University of La Plata (UNLP, Argentina), where he is currently an Associate Professor at the School of Computer Science. His research subjects are fault tolerance, high performance computing and computer architecture. He has co-authored several technical papers in journals and conference proceedings.

dmontezanti@lidi.info.unlp.edu.ar (Corresponding autor)

III-LIDI, Facultad de Informática, UNLP 50 y120 s/n, La Plata (1900), Bs As, Argentina.

Enzo Rucci received his Ph.D. in Computer Science from the National University of La Plata (UNLP, Argentina), where he is currently an Associate Professor at the School of Computer Science. His research interests include high performance computing, bioinformatics and health informatics. He has co-authored several technical papers in journals and conference proceedings.

erucci@lidi.info.unlp.edu.ar

III-LIDI, Facultad de Informática, UNLP 50 y120 s/n, La Plata (1900), Bs As, Argentina.

Armando De Giusti is a Senior Professor at the School of Computer Science of the National University of La Plata (UNLP, Argentina), and also a Researcher from CONICET. His research interests include parallel and distributed programming, real-time systems and technology applied to education. He has supervised 12 Ph.D. thesis and co-authored more than 100 technical papers in journals and conference proceedings.

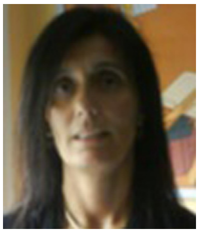
degiusti@lidi.info.unlp.edu.ar

III-LIDI, Facultad de Informática, UNLP 50 y120 s/n, La Plata (1900), Bs As, Argentina.

Marcelo Naiouf is a Chair Professor at the School of Computer Science of the National University of La Plata (UNLP, Argentina). He is the Head of the Ph.D. Program at Informatics Faculty, supervisor of 8 Ph.D. thesis, invited lecturer in Universities of Argentina, and leader in several research projects. His research interests include High Performance Computing and parallel algorithms. He has co-authored 3 books and more than 120 full-reviewed technical papers in journals and conference proceedings.

mnaiouf@lidi.info.unlp.edu.ar

III-LIDI, Facultad de Informática, UNLP 50 y120 s/n, La Plata (1900), Bs As, Argentina.



Dolores Rexachs is an Associate Professor at the Computer Architecture and Operating System Department at University Autonoma of Barcelona (UAB), Spain. She has been the supervisor of 12 Ph.D. thesis and has been invited lecturer in Universities of Argentina, Brazil, Chile and Paraguay. The research interests include parallel computer architecture, parallel I/O subsystem, fault tolerance in parallel computers, tools to evaluate, predict, and improve the performance in parallel computers. She has co-authored more than 80 full reviewed technical papers in journals and conference proceedings.

Dolores.Rexachs@uab.cat

CAOS, Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona.

Department Room: QC/3024 Building Q - Bellaterra – 08193 – Barcelona SPAIN 35.

Emilio Luque is an Emeritus Professor at the Computer Architecture and Operating System Department at University Autonoma of Barcelona, Spain. Invited lecturer at universities in the USA, South America, Europe and Asia, key note speaker in several conferences and leader in several research projects funded by the European Union (EU), Spanish government and different industries. His major research areas are: parallel and distributed simulation, performance prediction and efficient management of multicluster–multicore systems and fault tolerance in parallel computers. He has supervised 19 Ph.D. thesis and co-authored more than 230 technical papers in journals and conference proceedings.

Emilio.Luque@uab.cat

CAOS, Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona.

Department Room: QC/3024 Building Q - Bellaterra - 08193 – Barcelona SPAIN 56.