A Methodology for Soft Errors Detection and Automatic Recovery

Diego Montezanti
and A. De Giusti
and M. Naiouf
Instituto de Investigacion en Informatica LIDI (III-LIDI)
Universidad Nacional de La Plata
La Plata (1900), Buenos Aires, Argentina
{dmontezanti,degiusti,mnaiouf}@lidi.info.unlp.edu.ar

Jorge Villamayor
and Dolores Rexachs
and Emilio Luque
CAOS - Computer Architecture and Operating Systems
Universidad Autónoma de Barcelona, Barcelona, Spain
{jorgeluis.villamayor,dolores.rexachs,emilio.luque}@uab.es

***Index Terms*—soft error detection, automatic recovery, system-level checkpoint, user-level checkpoint**

***Abstract*—Handling faults is a growing concern in HPC; higher error rates, larger detection intervals and silent faults are expected in the future. It is projected that, in exascale systems, errors will occur several times a day, and they will propagate to generate errors that will range from process crashes to corrupted results because of undetected errors. In this article, we propose a methodology that improves system reliability against transient faults, when running parallel message-passing applications. The proposed solution, based on process replication, has the goal of helping programmers and users of parallel scientific applications to achieve reliable executions with correct results. This work presents a characterization of the strategy, defining its behavior in the presence of faults and modeling the temporal costs of employing it. As a result, we show its efficacy and viability to tolerate transient faults in HPC systems.**

I. Introduction

System reliability has become critical, especially in the area of High-Performance Computing (HPC), as systems performance and number of cores continue growing. Applications running on modern supercomputers must deal with fault rates of just few hours [1], and it is estimated that they could even get to about a few minutes in large parallel applications in exascale platforms. Consequently, these applications will not be able to progress efficiently without appropriate help [2]. The main concern is in relation to silent errors, namely Silent Data Corruption (SDC), with numerous recent reports and studies on their probabilities and impacts [3]. SDC is the most dangerous type of fault that can occur, because the program appears to be running correctly but, upon conclusion, invalid results are produced. Consequently SDC create serious problems in science, which relies on large-scale simulations, so its mitigation is one of the major challenges for current and future resilience. Given the difficulty and cost of adding hardware redundancy to the registry and processor arithmetical logic units [4], and the high cost of re-running HPC applications from the beginning, specific software strategies are needed for the target systems. Without efficient containment mechanisms, a failure that affects one task can result in the entire application crash or in incorrect outputs that, in a best-case scenario, are only detected after execution is complete, and very hard to correct. A single SDC causes deep effects on all processes that communicate in a message-passing application [5].

A common approach for providing fault tolerance is to perform redundant software execution, using the state machine replication approach, which implies that the replicas of a process follow the same execution sequence and produce the same output if given the same input [6]. Multicore architectures are convenient for redundant execution, which is a viable solution for detecting SDC in the context of HPC [1].

Checkpoint-based rollback recovery (C/R) is a common, well-studied technique for mitigate the losses of useful work caused by fail-stop failures, in which a process crashes [2]. Checkpointing involves periodically saving the application state; if a failure occurs, all processes restart from their recent checkpoints. Unfortunately, the overhead for using C/R increases with the number of cores. Taking into account the time required for C/R, a significant amount of computation time could be wasted if the fault rate is high. However, using C/R for transient fault mitigation is not so effective, because the stored checkpoint could contains undetected failures, making recovery impossible. The situation gets worse if computation is strongly coupled, since an error in one node could be propagated to the others in micro-seconds [7].

In this context of results not being reliable and costly to verify, this paper presents a methodology which is designed to provide transient fault tolerance for scientific message-passing parallel applications that execute in multicore clusters. The methodology has the goal of helping programmers and users of parallel scientific applications to achieve reliable executions. It is based in duplicating the execution of each process of the parallel application in a core of the same socket of the original process, leveraging the multicores hardware redundancy. The proposed solution achieves silent error coverage by addressing the problem in three different ways, each of them with particular features and performance: only detection with notification, and two different forms of recovery: the first one based in multiple system-level checkpoints, and the second utilizing a single safe application-level checkpoint. A functional description and temporal characterization are presented.

The remainder of the paper is organized as follows: Section

978-1-5386-3250-5/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCS.2017.71

2 reviews some basic concepts and related work. Section 3 functionally details the proposed methodology separating it in the three aforementioned stages. Section 4 evaluates the temporal behavior of the possible strategies. Finally, in Section 5, the conclusions and future lines of work are presented.

II. Background and Related Work

Depending on the impact on the application execution, different types of transient faults can be found [8][9]. Latent Errors (LE) affect data that are not used afterwards, so do not have an impact on results. Detected Unrecoverable Errors (DUE) cause abnormal conditions, from which the system software can be notified but can not be recovered, resulting in abrupt termination of the application. Time Out Errors (TOE) cause the program to not end within a stipulated time period. Finally, Silent Data Corruption (SDC) have effects that propagate until the program, which appears to execute correctly, ends with incorrect output. In message-passing parallel applications, these can cause: Transmitted Data Corruption (TDC), which affect data that are part of the contents of messages to be transmitted (i.e. if undetected, they propagate to other processes), or Final Status Corruption (FSC), where the altered data are not transmitted, but the error propagates locally, corrupting the final results of the affected process.

Current technologies cannot deal with frequent SDC. Existing algorithmic solutions [10] can only be applied to specific kernels. On the other hand, compiler or runtime software-based detection strategies can be applied to any code, but they are complex in nature. Containment aims to avoid the propagation of the damage to other nodes, or to prevent it from corrupting the data stored in checkpoints, which would make recovery impossible [2]. In [11], the authors propose the use of redundancy in HPC systems, which allows increasing availability and offers a trade-off between the number and the quality of components. In [12], it is shown that replication is more efficient than C/R in situations where both error rate and overhead of C/R are high.

Traditionally, SDC are detected by replicating executions and comparing the obtained results between replicas. Software-redundancy solutions are focused on replication at the level of threads [13], processes [4] and machine status, removing the need for expensive hardware; other solutions that are less accurate but require less resources have been explored, such as approximate replication, which implements upper and lower limits for computation results [2]. MR-MPI [11] proposes transparent redundancy in HPC and partial process replication; it can be used in combination with C/R in non-replicated processes [14]. rMPI [12] is a protocol for redundant execution of MPI applications, focused on failures that cause the system to stop; it uses the profiling layer to interpose MPI functions. Each node is duplicated, so the application fails only if two corresponding replicas fail. Redundancy scales, i.e., the probability of simultaneous failure of a node and its replica decreases when the number of nodes increases, at the cost of duplicating the amount of resources and quadrupling the number of messages. In [6] a scheme based in multithreaded processes for shared memory systems is proposed, handling non determinism due to memory accesses.The authors of [1] propose a protocol for hybrid task-parallel MPI applications, which implements recovery based in uncoordinated checkpoints and message logging, restarting only the task that experienced the error and handling the MPI calls inside the task. RedMPI [5] is a MPI library that exploits rMPI's per-process replication to detect and correct SDC, comparing at the receiver side the messages sent by replicated issuers. It implements an optimization based on hashing to avoid sending all messages and comparing their entire contents. It does not require application code modifications and it ensures determinism between replicated processes. Results show it can protect applications even with high failure rates so it can potentially be used on large-scale systems. The authors show that even a single transient error can have a deep effect on the application, causing a cascading corruption pattern towards all other processes through MPI messages. Like our proposal, RedMPI also allows customizing the mapping of the replicas on the same physical node as the native processes, or in neighbors with lower network latency. Like us, they monitor communications, as their accuracy is necessary to output correction. Detection is delayed upon transmission, but validation is performed on the receiver side. This results in additional overhead, latency and network congestion in which our strategy does not incur. Fault tolerant protocols for other parallel programming models, such as PGAS [15] have been also proposed.

III. Description of the Methodology

In the following subsections, the fundamentals of the different proposed alternatives to achieve fault tolerance will be found, as well as an evaluation of the temporal effects of implementing each particular feature. A simplified model is included, which takes into account the factors that affect the total running time, both in the absence of faults as with a single silent error occurring during the execution. The baseline for evaluations is the time involved in a manual strategy for ensuring reliable results, that consists in launching two instances of the application in parallel, and comparing final results in a non-automatic fashion. Such a strategy makes the same utilization of the computing resources that our proposal, i.e. half of the total available cores for each individual instance. Without any faults, there may be a coincidence of final results; however, if a transient fault occurs, a third re-execution and a new comparison are required to pick the outputs of the runs that form a majority as the correct ones (voting mechanism). In table I we summarize the parameters involved and their meanings.

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

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

Equation 1 shows the execution time of the manual strategy in absence of faults (fault absence, T_{FA}), although equation 2 is the time when a fault occurs (fault presence, T_{FP}), when,

Parameter	Meaning
T_{prog}	Execution time of two instances of the original application in parallel.
T_{comp}	Time of manual comparison of results. Automatic comparison may take shorter. In simplified model is considered the same; may include calculating a hash.
T_{rest}	Time of manually restarting the application. Automatic restart may take shorter. In simplified model is considered the same.
f_d	Fraction of overhead due to detection mechanism. Dependent on the application. 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. Can be adjusted to minimize overhead.
k	Number of extra checkpoints the application needs to rollback to find a non-corrupted one. Dependent on the application and detection latency. Can be experimentally determined.
t_{ca}	Time involved in storing an application checkpoint. Shorter than t_{cs}.

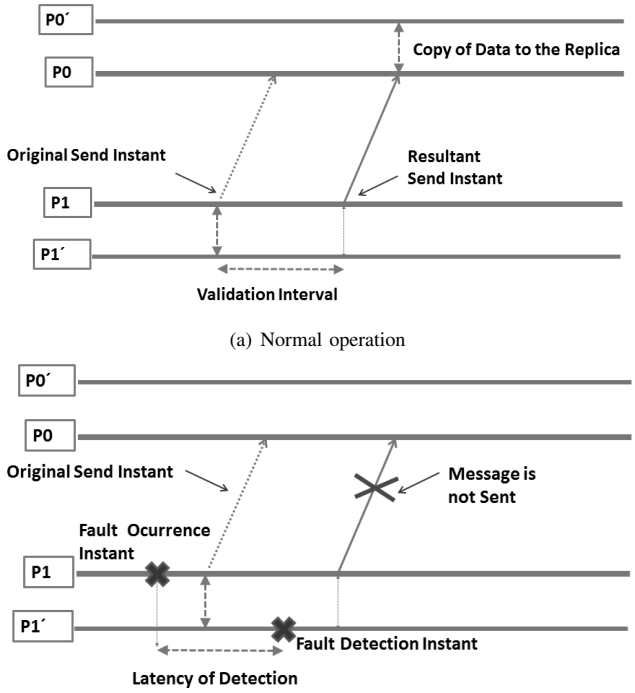
TABLE I
PARAMETERS INVOLVED IN TEMPORAL CHARACTERIZATION

besides the re-execution time, there exists a restart time and (at least) a new comparison for voting.

A. Error detection with notification

The first feature, detection, is achieved by validating the messages between processes in deterministic parallel applications, before being sent. This allows isolating the error that affect a process by preventing it from propagating to the others. The detection strategy is designed to detect failures that cause SDC (both variants) and TOE. It consists in duplicating each application process in a thread, requiring a synchronization mechanism between both redundant threads. When a communication is to be performed, the leading thread stops running and waits for its replica to reach the same point, and all message contents, calculated by both replicated threads, are compared. If they match, only one of the threads sends the message, therefore no consuming additional network bandwidth. When the receptor gets the message, it is synchronized with its replica, makes a copy of the received contents for it, and then both redundant threads continue the execution. When the application processes finish, their results are compared with the ones of their replicated threads to detect failures that may have locally propagated until the end. Additionally, detection of TOEs is based on the premise that, in dedicated homogeneous system, the execution times of two replicated threads should be similar [16]. Therefore, a notorious difference can be assumed as both replicas have separated their flows due to a silent error. Thus, time-out interval should be configured accordingly on the application behavior: if too high, detection latency increases; if too low, a small difference in processing times could result in the detection of a false positive. Anyway, if one of the processes goes into an infinite loop, an error is effectively detected.

Figure 1 shows an outline of the proposed detection mechanism. Each application process is mapped to a core, and each



(a) Normal operation

(b) Operation when detecting a fault

Fig. 1. Detection mechanism

redundant thread runs in a core that shares some cache level with the original; thus, there is no need to access to main memory, harnessing the hierarchy to solve comparisons. As a consequence of this replication scheme, half of computing resources are usable from the standpoint of the parallel application performance. It should be clear that the method consists in launching only one instance of the application, with each of the processes internally replicated in a thread, instead of the baseline case, in which two independent instances of the application are launched in parallel; however, both the baseline case and our strategy make the same use of the computational resources.

As this methodology makes use of process duplication, it is capable of performing only detection (triplication should be used for achieving correction by voting). The two following subsections show how recovery can be made without need of triplication. Because all the described strategies make use of the computational resources available in the system, it is also remarkable that there is not additional cost owing to the incorporation of fault tolerance mechanisms.

The overhead introduced in execution time is a consequence of process duplication, synchronization between replicas, comparison before sending, copy of the received messages and final verification of the results; it is measured for some real HPC applications. As regards as the validation interval, if results are compared only at the end, detection introduces little overhead, but the fault would remain latent for a long time, making much computation unusable; however, if partial

results are validated too frequently (e.g., at any communication) higher overhead is introduced but the fault can be detected faster, thus not computing uselessly. Therefore, a compromise must be reached between detection interval and introduced overhead. Our previous studies [16] show that, depending on the computation-to-communication ratio of the target application and the size of the workflow, the overhead can be widely variable between extreme values.

When a SDC or TOE occurs, the only-detection strategy notifies the user and leads the system to a safe stop, in the absence of a recovery strategy. Having such a mechanism allows immediately re-launching the application upon detection, avoiding unnecessary and costly wait for the conclusion with incorrect results, so validating messages narrows error latency.

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

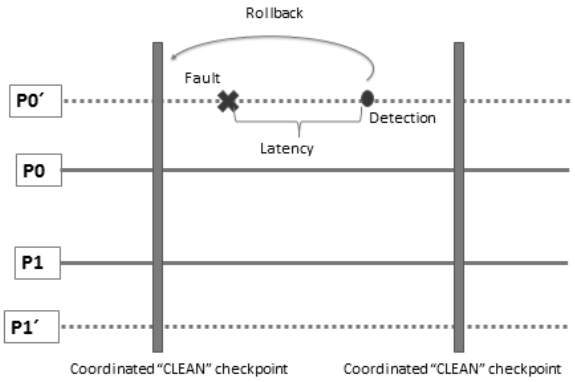
$$T_{FP} = T_{prog}(1 + f_d)(1 + X) + T_{comp} + T_{rest} \quad (4)$$

Equation 3 shows the execution time of the detection strategy in absence of faults. The original time is incremented by a factor f_d, the overhead of detection mechanism, and the final comparison that is performed for correctness. Equation 4 is the time when a fault occurs. The first term is the time executed until the instant of detection, and the whole re-execution. Upon detection a restart is required, and the final comparison in the re-execution. It should be clear that such a verification method will be equally effective when more than a single error occurs during the execution. The first error that causes a discrepance between contents of messages or final results will lead the system to a safe stop. Only two extremely unlikely cases make this mechanism vulnerable; more details on them can be found in [17]. The detection strategy can address multiple non-related errors, but we limit the analysis of the temporal behavior to first-order terms, i.e. with no error or single error ocurring.

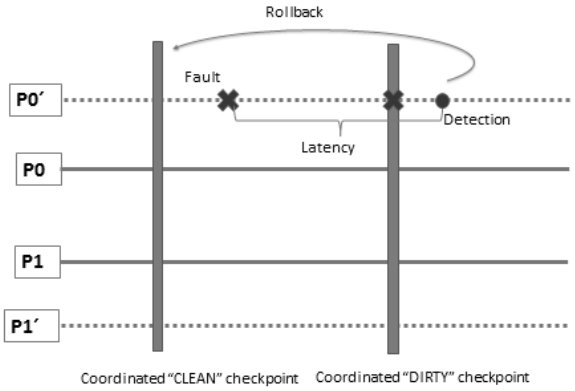
B. Recovery based in multiple system-level checkpoints

The second alternative consists in incorporating a recovery strategy, in the way to tolerate transient faults. This is accomplished by the storage of a number of coordinated distributed checkpoints, built with a system-level tool. Given the possibility of a silent error corrupting the internal state of one of the replicated processes, there is no guarantee of a checkpoint to be consistent. It is necessary to determine if it is possible to restart from the last recorded checkpoint; otherwise, a previous one has to be used. Thus, a chain of checkpoints has to be stored in order to ensure recovery [7]. It is remarkable that, given the short-lived nature of transient faults, the restart can be attempted from the corrupted node itself. Two cases are possible:

1) The transient fault and its detection occur within a checkpoint interval. Application is restarted from the last checkpoint. Particularly, if the fault is detected previously to the first checkpoint, the application is relaunched from the beginning.
2) Transient fault occurs before recording a checkpoint, but is detected after it, i.e. the detection latency transposes the checkpoint interval. In this case, last checkpoint is not valid, and restarting from it should cause the repetition of the previous error. Thus, restart from the prior checkpoint must be attempted. This situation can be generalized: depending on the latency, the fault can go through any number of checkpoints, and several attempts may be done until rollback recovery is possible.



(a) Detection latency bounded in checkpoint interval



(b) Detection latency transposing checkpoint interval

Fig. 2. Use of multiple system-level checkpoints for recovery

To evaluate this recovery scheme, controlled fault injection has to be made, corresponding to the two above mentioned cases. The injection is made from inside the application code, by changing the value of a variable in only one of the redundant threads of a process in a single iteration of the computation; thus, a bit-flip in a processor register is simulated, since data corruption manifests as an observable difference between the memory state of the replicas. Figure 2 outlines the recovery mechanism. The proposed algorithm,

in pseudocode, is described as Algorithm 1. For simplicity, injection mechanism is not shown.

```
1  int extern_counter=0;   /* controls the number
       of rollbacks; external, has not to be
       checkpointed */
2  boolean fault_detected= FALSE;
3
4  (...)              // application execution time
5  sys_ckpt(n-1);     // save a system checkpoint
6
7  (...)              // application execution time
8  sys_ckpt(n);       // save a system checkpoint
9
10 if (fault_detected== TRUE) {   /* fault detected
       when communicating */
11   extern_counter++;       /* extern_counter=1;
       restart from checkpoint n will be tried */
12   fault_detected= FALSE;  /* reset detected
       faults in restarting */
13   restart_from_sys_ckpt(n); /* from last
       checkpoint (Fig 2a) */
14 }
15
16 (...)      // application execution time
       /* If a fault is never detected again,
       checkpoint(n) was clean and execution
       finishes normally */
17 if (fault_detected== TRUE) {   /* fault detected
       when communicating */
18 /* checkpoint(n) was dirty because the fault is
       detected again */
19   extern_counter++;        /* extern_counter=2;
       restart from checkpoint n-1 will be tried */
20   fault_detected= FALSE;  /* reset detected
       faults in restarting */
21   delete_sys_ckpt(n);           // the dirty one
22   restart_from_sys_ckpt(n-1);  /* from last but
       one checkpoint (Fig 2b) */
23 }
```

Algorithm 1: Recovery algorithm with system-level checkpoints

The implementation of the methodology consists in a library of modified MPI functions and data types with extended functionality for fault detection by comparison before sending, message copies upon reception, and synchronization between replicated threads. The coordinated system-level checkpoints are built with DMTCP library [18], which generates distributed-per-process checkpoint files, and a single restart script for each checkpoint. Master/Worker matrix-multiplication is the test application. Even though transient faults can occur at any place and time during the execution, significant moments and variables were selected for controlled fault-injection experiments. The expected behavior was obtained in both cases of possible silent errors. To make this recovery method more usable, it needs to be automated; this is achieved by letting a process, external to the application, to read the $extern\_counter$ and executing the correspondent restart script respect to its value; also, it has to delete a checkpoint that has caused a failed restart. The target application only needs to read $extern\_counter$ to learn if an injection has to be made (which occurs only once), and then write it upon detection.

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

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

Equation 5 shows the execution time of this strategy in absence of faults. The time is the same as with only-detection but the aggregated term is the time involved in storing n system-level checkpoints. Equation 6 is the time when a fault occurs. Parameter k appears, which represents the number of checkpoints that need to be rewinded if the restart from the last one fails. The third term is the time consumed in checkpointing, considering that one or more checkpoints should be recorded again if found corrupted during the recovery process. Fourth term is an approach of the time of re-execution, taking into account that the fault, in average, may be detected at the half of checkpoint interval, and, in the best case ($k = 0$) this lapse will need to be re-executed; if $k > 0$, that fraction plus an integer number of checkpoint intervals will require several re-executions. The last term is the number of required restarts.

This method is suitable when only system-level-checkpoints are available. However, it has two important drawbacks. The first one is related to the required storage amount. The uncertainty about the validity of the recorded checkpoints avoids deleting previous ones: if so, there is a risk of having to re-execute a significant part of the application, or even relaunching it from the beginning [7]. Anyway, solutions about multilevel checkpointing can be used to mitigate the negative impact of multiple checkpoints over the storage [2]. The second relevant drawback is scalability: coordinated-system-level checkpoints are not expected to be the best solution in upcoming exascale systems, because the large amount of related-to-the-system information they store. Instead, user-level checkpoints are becoming more frequent, due to their lower costs and portability [1]. Clearly, this is a costly method, because of the need of keeping an undetermined amount of active checkpoints and the number or retries that can be required.

C. Recovery based in a single safe application-level checkpoint

The third alternative in the way to achieve transient fault tolerance tries to overcome the problems of using system-level checkpoints. When available, user-level checkpoints are more suitable, because they only store the application-related information. Despite of needing a deep knowledge of the application internal structure (computing and communication), they are smaller, more portable and scalable than their system-level counterparts. Because of this, our methodology proposes the utilization of a single checkpoint for recovery, plus a mechanism to ensure the reliability of the last stored checkpoint. This allows deleting the previous one, saving storage space and guaranteeing that no too far relaunches are

required. The proposed solution benefits from the synchronization mechanism between replicas to record per-thread user-level checkpoints, which consist in storing only the variables that are significant to the application at the particular moment. Both-replicas checkpoints are recorded, and then a hash on each one is calculated. The two hashes are compared, in the same way as contents of messages in the detection stage. If the comparison is successful, the checkpoint is valid, i.e. represents a consistent state for recovery, so the previous one can be safely discarded to save storage. The new checkpoint recorded by one of the redundant threads substitutes the old one in stable storage. In the other hand, if an error is detected, the checkpoint has been corrupted and cannot be used for recovery, so it has to be deleted and rollback to prior checkpoint has to be made. Therefore, there is only one valid checkpoint at a time, except for the lapse of validation, but after it there is only one again, independently of the results of the verification. Proposed algorithm, in pseudocode, is described as Algorithm 2.

```
1 /*  usr_ckpt function definition  */
2
3 boolean usr_ckpt(n) {
4   for (thread_id=0, thread_id<2, thread_id++) {
        // for both replicas
5     store_all_significant_variables(thread_id);
6         // makes its custom checkpoint
7     hash_array[thread_id]=compute_hash(thread_id);
8   }
9   synch_threads();     // wait for each other
10  if (thread_id==0) { /* only one of the replicas
      compares hashes */
11    if (hash_array[0]==hash_array[1]) {  /* if
      successful */
12      remove_all_significant_variables(thread_id);
      // deletes its own checkpoint
13      return TRUE;  /* this is a valid
      checkpoint; previous can be discarded */
14    } else return FALSE; // corrupted checkpoint
15  }
16 }                          // end of function
17 /*******************************************/
18
19 (...)          // application execution time
20 if (usr_ckpt(n)== TRUE) // n: current checkpoint
21   remove_usr_ckpt(n 1);  /* delete previous since
       current is valid */
22 else {
23   remove_usr_ckpt(n);  /* remove current corrupted
       checkpoint */
24   restart_from_usr_checkpoint(n 1); /* from
       previous */
25 }
```

Algorithm 2: Recovery algorithm with application-level checkpoints

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

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

Equation 7 shows the execution time in absence of faults. The time is the same as with only-detection but the aggregated term is the time involved in storing n user-level checkpoints and validating each of them. Equation 8 is the time when a fault occurs. As no more than a single rollback is needed, in average only half of the checkpoint interval has to be re-executed (fourth term), and also there is a single restart time (last term).

IV. Evaluation of the Model

In order to show how our model can be used to evaluate the temporal behavior of each alternative, a simple example is presented, which includes real measured overhead values taken from carried-out tests [16] for parameters in table I. The tests were performed using a cluster of Blade multicores with four blades. Each blade has two quad core Intel Xeon e5405 2.0GHz processors with 64Kb private L1 cache and 6Mb L2 cache (shared between pairs of cores), 10Gb RAM memory (shared between both processors) and 250Gb local disk. The operating system is GNU/Linux Debian 6.0.7 (64 bits, kernel version 2.6.32) and the MPI library used is OpenMPI (version 1.6.4).

Three parallel benchmark applications were used for tests: matrix multiplication; solution to Laplaces equation; and DNA sequence alignment. These benchmarks are well-known, representative, computationally intensive scientific applications, and they have three different communication patterns: Master-Worker, Single-Program-Multiple-Data (SPMD) and Pipeline, respectively. Benchmark applications were tested using different number of processes: P = 4, 8, 16. Various problem sizes were used for each application: N=2048, 4096, 8192, 16384 for matrix multiplication; N=4096, 8192, 16384 for solution to Laplaces equation and N=65536, 131072, 262144, 524288 for DNA sequence alignment.

Tests were carried out to compare execution times of the three applications between raw MPI versions and an MPI-based implementation of our strategy, that incorporates mechanisms of process duplication, synchronization between replicas, comparison and copy of the messages contents and final validation of the results. The resulting overheads of adding these features were measured and therefore, average parameters T_{comp} and f_d were obtained. For clarity, in the case of the original MPI executions, at most four application processes were mapped by node, which means that only four cores of each node were used. In the case of our implementation, the same mapping was assigned, but as the redundant threads execute on available cores, all the cores of each node were used. On the other hand, parameters t_{cs} and T_{rest} were obtained from averaging times measured with DMTCP library tools. Last, parameters X, n and t_i are manually assigned for a typical case.

All values used for the temporal evaluation, obtained as described, are listed in table II, and the resulting times in each case are summarized in table III.

Although this is a simple test case with real values, it can be observed that adding degrees of sophistication involves larger overheads in absence of faults, as expected. However, when a silent error occurs, the different alternatives may be

Parameter	Value
T_{prog}	10 hours
T_{comp}	1 hour
T_{rest}	0.3 hours
f_d	0.15
X	0.5
n	4
t_{cs}	0.9 hours
t_i	2.5 hours
t_{ca}	0.5 hours

TABLE II
VALUES UTILIZED FOR TEMPORAL EVALUATION

Situation	Execution Time [hs]
Baseline, without fault (Eq. 1)	11
Baseline, with fault (Eq. 2)	22,3
Only detection, without fault (Eq. 3)	12.5
Only detection, with fault (Eq. 4)	18.5
Multiple checkpoints, without fault (Eq. 5)	16,1
Multiple checkpoints, with fault (Eq. 6, k = 0)	17.7
Multiple checkpoints, with fault (Eq. 6, k = 1)	22.6
Single checkpoint, without fault (Eq. 7)	18.5
Single checkpoint, with fault (Eq. 8)	20

TABLE III
RESULTS OF TEMPORAL EVALUATION

capable to offer a gain both in time and reliability, which becomes particularly significant in applications that can run for many hours. Even, with this set of parameters, the situation of rolling back to the last but one system checkpoint (Equation 6, k=1) temporally behaves worse than the baseline case, suggesting that storing more than a single checkpoint is not always convenient. It should be clear that this is not a general conclusion (the temporal behavior is highly-dependent of the application communication pattern, the volume of transmitted data, and the instant of detection), but it shows how the model can be used for temporal evaluation if the involved parameters can be measured or estimated, as well as the potential of the proposed methodology in aiding users of scientific applications to reach dependable executions.

V. CONCLUSIONS AND FUTURE WORK

Given the fact that a single SDC causes deep effects on all processes that communicate, it can be concluded that protecting the applications at the level of the MPI messages is a feasible and effective method for detecting, isolating and preventing subsequent data corruption. In this paper, a methodology for detecting and recovering from silent errors is presented, which consists in three complementary alternatives. Functional and temporal characterization is made, showing the viability and efficacy to tolerate transient faults in HPC systems.

As a future work, emulation of non-deterministic calls is required in order to enlarge the scope of applications that can be protected. Experimental validation with applications with customized user-level checkpoints has to be extended. Optimal checkpoint interval to minimize expected execution time has to be derived. On the other hand, forcing the time in equation 6 to be smaller than equation 4 allows to obtain the maximum number of worth stored checkpoints for recovery. Because the replicated threads should be mapped on cores that share low level caches for best performance, it would be interesting to study the effects of moving the replicas to another socket in the node because the application has the mapping binded to intra-socket cores.

As a final goal, the integration with architectures that use C/R strategies for tolerating permanent will must be intended, in order to achieve fault-tolerance for both types of errors.It is important to remark that the actual implementation of our proposal is in a prototype stage, but a stable productive version of the library is being developed and characterized.

REFERENCES

[1] T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and 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] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[3] J. Elliott, M. Hoemmen, and 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. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, 2009.

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

[6] H. Mushtaq, Z. Al-Ars, and 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.

[7] G. Lu, Z. Zheng, and 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.

[8] S. S. Mukherjee, J. Emer, and 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.

[9] D. Montezanti, E. Frati, D. Rexachs, E. Luque, M. Naiouf, and A. De Giusti, "Smcv: a methodology for detecting transient faults in multicore clusters," *CLEI Electronic Journal*, vol. 15, no. 3, pp. 5–5, 2012.

[10] 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.

[11] C. Engelmann and 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.

[12] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, and T. Brightwell, "rmpi: increasing fault resiliency in a message-passing environment," *Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488*, 2011.

[13] G. Yalcin, O. S. Unsal, and 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.

[14] X. Ni, E. Meneses, N. Jain, and 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.

[15] N. Ali, S. Krishnamoorthy, N. Govind, and 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.

[16] D. Montezanti, E. Rucci, D. Rexachs, E. Luque, M. Naiouf, and A. De Giusti, "A tool for detecting transient faults in execution of parallel scientific applications on multicore clusters," *Journal of Computer Science & Technology*, vol. 14, 2014.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and 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.

[18] J. Ansel, K. Arya, and 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.