Characterizing a Detection Strategy for Transient Faults in HPC

DIEGO MONTEZANTI[1,4], DOLORES REXACHS[2], ENZO RUCCI[1,3], EMILIO LUQUE[2], MARCELO NAIOUF[1] AND ARMANDO DE GIUSTI[1,3]

[1] III-LIDI, Facultad de Informática, UNLP
Calle 50 y 120, 1900 La Plata (Buenos Aires), Argentina
{dmontezanti, erucci, mnaiouf, degiusti}@lidi.info.unlp.edu.ar
[2] Departamento de Arquitectura de Computadoras y Sistemas Operativos, UAB
Campus UAB, Edifici Q, 08193 Bellaterra (Barcelona), Spain
{dolores.rexachs, emilio.luque}@uab.es
[3] Consejo Nacional de Investigaciones Científicas y Técnicas
[4] Instituto de Ingeniería y Agronomía, UNAJ
Av. Calchaquí 6200, 1888 Florencio Varela (Buenos Aires), Argentina

Abstract. *Handling faults is a growing concern in HPC; greater varieties, 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 that they will propagate to generate errors that will range from process crashes to corrupted results, with undetected errors in applications that are still running. In this article, we analyze a methodology for transient fault detection (called SMCV) for MPI applications. The methodology is based on software replication, and it assumes that data corruption is made apparent producing different messages between replicas. SMCV allows obtaining reliable executions with correct results, or, at least, leading the system to a safe stop. This work presents a complete characterization, formally defining the behavior in the presence of faults and experimentally validating it in order to show its efficacy and viability to detect transient faults in HPC systems.*

Keywords: *transient faults, detection, scientific parallel applications, silent data corruption, HPC, fault injection.*

1. Introduction

Processor clock frequency stagnation has resulted in performance improvements being achieved through increasing the number of components. System escalation involves to the problem of a decrease in tension which, together with sub-micron miniaturization challenges, results in great increases in failure rates. Electromagnetic interferences generate current pulses that alter the values that are stored or in combinational logics. The higher variability in manufacturing processes causes inconsistent behaviors, while aging results in permanent errors being more frequent and the likelihood of multiple failures has also increased [1,2]. Because all of this, system reliability has become critical, especially in the area of High-Performance Computing (HPC) with more than hundreds of thousands of cores. Recent studies in modern supercomputers show that Mean Time Between Failures (MTBF) are just a few hours [3], and it is estimated that they could even get to about 30 minutes in large parallel applications in exascale platforms. Consequently, these applications will not be able to progress efficiently without appropriate help [4,5]. The main concern is in relation to silent failures, namely Silent Data Corruption (SDC), with numerous reports and studies on their probabilities and impacts surfacing [2,6,7]. By potentially causing invalid results, SDCs create serious problems in science, which increasingly relies on large-scale simulations. For all these reasons, SDC mitigation is one of the major challenges for current and future resilience.

SDCs appear as bit-flips (change in the value of a bit) that affect the storage or the cores. To detect or correct them, manufacturers add more powerful Error Correcting Codes (ECC) in the memory, protect buses with parity bits, and add redundancy to the circuits of some logical units [8]. However, adding hardware redundancy to the registry and processor arithmetical logic units is too costly [9].

The small supercomputer market, which requires high reliability, can be satisfied with double- and triple-redundancy solutions to achieve detection and correction, respectively. Even though the cost of doing this is high, it is preferable to having corrupt results. SDCs remain latent until the altered data are used, and detection latencies depend on the application.

The standard, most commonly used method to handle errors in current parallel systems (particularly those that run MPI applications), is recording periodical checkpoints. In case of failure, the Checkpoint/Restart (C/R) method re-launches the application from the last checkpoint. Unfortunately, the overhead for using C/R increases with the number of cores. Taking into account the time required for C/R and re-launch, a significant amount of useful computation time could be wasted if the MTBF is very low. The situation gets worse if computation is strongly coupled, since an error in one node could be propagated to the others in micro-seconds [1,10].

The traditional model based on C/R assumes that detection is almost immediate. Additionally, if the stored checkpoint contains undetected failures, recovery will not be possible. The few general detection techniques currently available introduce high overheads in parallel applications [2,11]. Based on all this, detection latency ranges are expected to increase, making the problem even worse due to SDCs. There are no efficient containment mechanisms, either which means that a failure that affects one task can result in the application crash or in incorrect outputs that, in a best-case scenario, are only detected after execution is complete and which are very hard to correct.

Replication at process-level has proven to be a reliable alternative, but in order to make it appealing for HPC, there are some challenges still to be solved, such as minimizing time and resource utilization overheads, ensuring that the inner states of the replicas are equivalent to one another (which is not trivial, since non-deterministic operations could be run), and reducing energy consumption. Traditionally, SDC are detected by replicating executions and comparing the results obtained. RedMPI [2] does this at the level of the processes, but there are other methods that do it at the level of the threads [12]. Other solutions that require less resources and are less accurate have also been explored, such as approximate replication, which implements upper and lower limits for computation results [1].

In this context, the SMCV methodology [13,14] has been proposed in recent years. SMCV is designed to detect transient failures in HPC, specifically for scientific applications that use MPI on multicore clusters. SMCV allows obtaining reliable executions with correct results or, at the very least, report the occurrence of SDCs and taking the system to a safe stop after a limited detection latency, saving significant time, especially in long applications.

The remaining sections of this document are organized as follows: Section 2 reviews some basic concepts, while Section 3 describes related work. Section 4 details the strategy used in SMCV, in which the behavior in case of failure, its Sphere of Replication (SoR), and its vulnerabilities are formally defined. Section 5 describes the experiments carried out through a controlled fault injection, in order to validate the behavior defined and showing the efficacy and viability of SMCV to detect transient failures in HPC systems. Finally, in Section 6, the conclusions and future lines of work are presented.

2. Basic Concepts

Depending on the impact on application execution, transient faults can be classified as follows [13]:

- Latent Error (LE): it affects data that are not used afterwards, so it does not have an impact on results.
- Detected Unrecoverable Error (DUE): it causes an anomaly that the system software can detect and that is unrecoverable; it usually causes the application to end abruptly.
- Time Out Error (TO): the program does not end within a given period of time.
- Silent Data Corruption (SDC): it is not detected by any system software level, and its effects are propagated until the program ends with an incorrect output. In parallel applications with message passing, these can cause: Transmitted Data Corruption (TDC), which affects data

that are part of the contents of the messages to be transmitted (if undetected, it propagates to other processes), or Final Status Corruption (FSC), where the altered data are not transmitted, but they are propagated locally, corrupting the final status of the affected process.

3. Related Work

Current technologies cannot deal with frequent SDCs. Existing algorithmic solutions [15] can only be applied to specific kernels; hence, mechanisms that allow dealing with the errors that are beyond their scope should be assessed. On the other hand, compiler- or runtime software-based detection strategies can be applied to any code, but they are more complex in nature.
Contention aims to avoid the propagation to other nodes of the damage caused by the fault, or to prevent it from corrupting the data stored as a checkpoint, which would make recovery impossible [1]. In [16], the authors propose the use of redundancy in HPC systems, which allows increasing system availability and offers a trade-off between the number of components and their quality. In [17], the authors show that replication is more efficient than C/R in situations where MTBF is low and the time overhead of C/R is high. Software-redundancy solutions are focused on replication at the level of the threads [12], processes [9] and machine status to remove the need for expensive hardware.
MR-MPI [19] is another proposal for transparent redundancy in HPC - it offers partial replication (only some processes are replicated); it can be used in combination with C/R in non-replicated processes [20,21].
rMPI [18] is a protocol for the redundant execution of MPI applications, focused on failures that cause the system to stop; it used the profiling layer to interpose MPI functions. Each node has a replica so, in case of a permanent failure, the redundant node continues without interruptions; the application fails 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 used and quadrupling the number of messages. RedMPI [2] is a MPI library that exploits rMPI's process replication to detect and correct SDC, comparing at the receiver 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 that replicas are run deterministically. Results show that it can protect applications even with high failure rates with time overheads below 30%, so it can potentially be used on large-scale systems. The authors in [2] analyze the propagation of SDCs among nodes through MPI communications, and they show that even a single transient failure can have a deep effect on the application, causing a cascading corruption pattern towards all other processes.

The same as SMCV, by focusing on messages, RedMPI monitors the most critical data for the application; communication correction is necessary to output correction. Since SDC can affect data that are not communicated immediately, the failure is detected upon transmission. However, unlike SMCV, RedMPI performs its validation on the receiver side. This is because, on the side of the issuer, all replicas must communicate with the others to verify their contents internally before sending the message. This results in additional overhead and latency, since the receiver loses all that time before being able to continue. Since SMCV replicates at the level of the threads and not the processes, it does not need to send messages among issuers for validation. By sending just one message after the validation, it does not cause network congestion. The same as SMCV, with RedMPI corruption remains confined to a process, even without correction. It also allows customizing replica mapping on the same physical node as the native processes (or in their neighbors with lower network latency).

4. Characterizing SMCV

In this section, SMCV is characterized.

4.1 Brief Review of SMCV

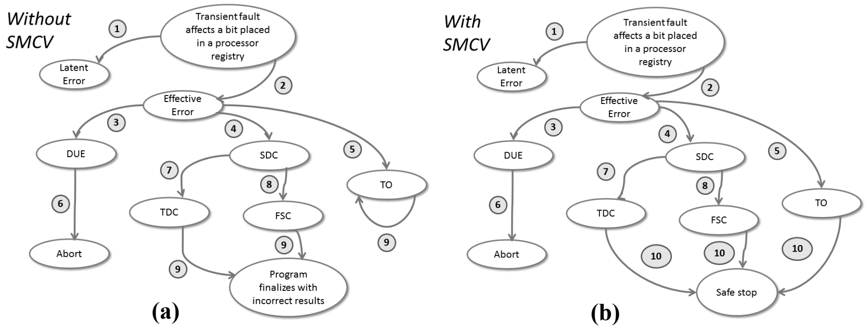
SMCV is a detection strategy that is based on validating the contents of the messages that are going to be sent among processes in deterministic parallel applications. It is designed to detect failures that cause SDCs (both variants) and TO. SMCV duplicates each application process in a thread, and it requires synchronization mechanisms between both concurrent replicas. When a communication is about to be established, the thread stops running and waits for its replica to catch up to it, and all message fields, calculated by both replicas, are compared in search for differences. If they match, only one of the threads sends the message, preventing errors to be propagated to other processes without using additional bandwidth. The receiver is synchronized with its replica, it receives the message and makes a copy for the replica, and then both replicas continue the execution. When they finish, results are verified to detect failures that may have been locally propagated to the end of the application.

4.2 Behavior in Case of Failure

In this section, the behavior of the detection methodology is described. Figure 1(a) shows a diagram with the possible status of an execution when there are no strategies implemented, while Figure 1(b) shows the same diagram when SMCV is applied. Ellipses represent statuses and arrows represent events that cause the transition from one status to another. Transitions are numbered, and each of them is described.

4.3 Sphere of Replication

Sphere of Replication [9] is a commonly accepted concept to describe the logical redundant execution domain of a given technique and specify the limits for failure detection. All data that enter the SoR are replicated, execution within its scope is redundant in some form, and output data are compared to ensure data correction before they leave it. Any execution outside the SoR is not covered for failures and should be protected by other means. The original concept of SoR was used to define reliability limits for redundant hardware designs, placing it around specific units. However, its application is not suitable for proposals implemented in software, despite which there are some solutions that use the compiler to insert redundant instructions that have tried imitating a SoR centered on hardware [22]. On the other hand, the failure detection paradigm centered on software places the SoR around software layers [9]. This shows that, even though failures affect hardware, only those that affect application accuracy are relevant, while it is safe to ignore those that remain latent. The disadvantage of this approach, however, is that detection is delayed until the error is confirmed through invalid data leaving the SoR, which means that a failure can remain latent indeterminately.


Without SMCV
Transient fault affects a bit placed in a processor registry
Latent Error
Effective Error
DUE
SDC
TO
TDC
FSC
Abort
Program finalizes with incorrect results
1
2
3
4
5
6
7
8
9
(a)
With SMCV
Safe stop
10
(b)

1. The affected bit is not used.
2. The affected bit is used by the application.
3. The altered bit affects data controlled by the operating system.
4. The altered bit affects user application data.
5. The altered bit causes the application to become unresponsive within a time limit.
6. The operating system detects the failure and aborts the application.
7. The affected data are transmitted to other process in the parallel application.
8. The affected data are only used by the local process.
9. Runtime.
10. SMCV detects the failure after some time and leads to a safe stop.

Fig. 1. *Diagram of execution statuses. (a) No failure detection strategy. (b) SMCV as detection strategy.*

SMCV is a software technique and, as such, adopts a SoR centered on software. Its objective is detecting failures that affect data that are handled inside processor registers, which are the most vulnerable part of the computer due to the difficulties involved with the implementation of hardware protection. As already explained, SMCV replicates in a thread the computations carried out by each process of the parallel application. Each thread operates on a local copy of the input data that is generated so that computation can be independent from that done on the replica. Therefore, the SoR is placed around the user application and its data, and it does not include the operating system or the communications library. Even though the memory is outside the SoR of SMCV, the use of global variables is not recommended, since they are centralized points of failure. If a failure that alters global variable occurs, both redundant threads would use the wrong value and, if no other failure occurred, SMCV would detect no errors.

4.4 Multiple Failures and Vulnerabilities

Most existing proposals can detect failures if it is assumed that a single bit-flip occurs during execution, but they are not as effective for failures that affect multiple bits. Fortunately, there are only two situations in which multiple failures can be combined to cause issues. The first of these situations is when the same bit is altered in both replicas, which results in a correct comparison and the failure is not detected. The second situation is when the failure affects one of the replicas, and the result of the verification is also altered, masking the original failure. However, the likelihood that any of these combinations occurs is very low, so they can be ignored without any serious risks. All other combinations of multiple failures are detected as simple failures as soon as the first difference is detected during verification [22]. SMCV can detect any simple transient failure that causes SDC or TO, but it does not support related multiple failures.
All failure tolerance techniques have vulnerabilities, i.e., circumstances under which they cannot detect the failures that effectively affect execution. The design characteristics of a strategy and the tests to which it is subjected (usually through failure injection) must allow making those vulnerabilities explicit.
Vulnerabilities are typically associated to failures that affect the detection mechanism itself [9], and SMCV is no exception. SMCV minimizes the delay between the time data are checked and the time when the validated values are used because verification is done when the data in a message are about to be used. This reduces the likelihood of failure in the time between both events (as in [22]); once the data are in the output buffer, they are outside of the SoR. On the other hand, checking the values to be sent is a centralized point of failure. If any error is detected when checking the data after a correct execution, a false positive has occurred and a safe stop is generated when the problem was in fact introduced by the detector itself. This vulnerability can be improved by a double comparison; however, even though it is not entirely reliable, partial redundancy in general is enough to meet user requirements [9]. Similarly, if validation is correct after a faulty execution, it means that the failure remained

hidden due to a second failure occurring. As already mentioned, SMCV cannot deal with this situation, but the likelihood of this happening is extremely low [22]. Additionally, the fact that SMCV can detect as TOs other failures that would be vulnerabilities if no such mechanism were available should also be considered. For instance, if an operation code is modified in such a manner that the resulting instruction is sending a message, or if a failure occurs while the tool is running, both replicas separate their execution flows. When one of them sends a message, synchronization is not successful, and the failure is detected after a period of time longer than the one established.

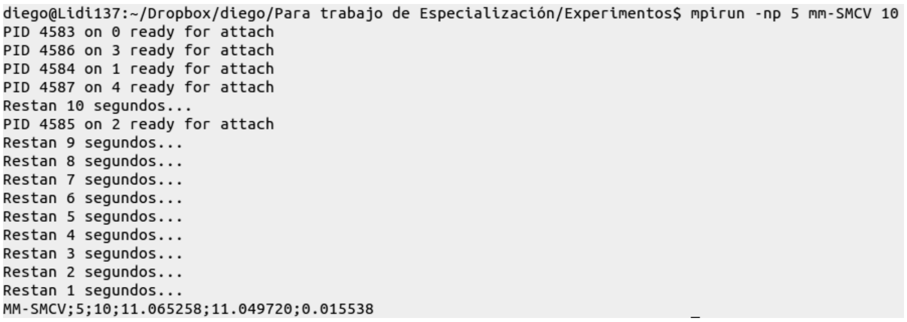
5. Validating Detection Efficacy

A number of tests were carried out to validate SMCV's detection efficacy. The application used was a parallel matrix multiplication MPI application (`C=A×B`) under the Master/Worker paradigm, where the Master participates in result computation [13]. The application operates as follows:

- The Master process divides matrix A among all Worker nodes and, using the function `MPI_Scatter`, sends a piece of the matrix to each one of them, keeping a piece for itself to calculate its portion of the resulting matrix.
- The Master sends a complete copy of matrix B to each Worker using the function `MPI_Broadcast`.
- All processes compute their respective pieces of matrix C, and then send their results to the Master process using function `MPI_Gather`.
- The Master builds matrix C using the pieces sent back by the Workers and its own results.

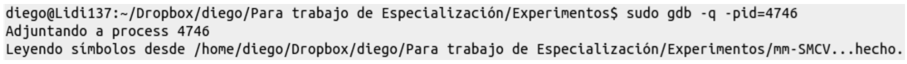
For the validation step, the application was adapted for integration with the functionality offered by SMCV as described in [14]. To do this, the source code of the application has to be modified, with the subsequent recompilation. The experiment consisted in injecting faults in a controlled manner at several points of the application using the GDB debugging tool[1]. To do this, a breakpoint is inserted on one of the running processes, the value of a variable is modified, and execution is resumed. Thus, a bit-flip is simulated in a processor register, since data corruption manifests itself if there is an observable difference between replica memory statuses. Even though transient faults can occur at any place and time during the execution, significant points were selected for this controlled injection process, both in relation to the computation done by the Master and that done by the Workers.

[1] GDB is available at www.gnu.org/software/gdb/



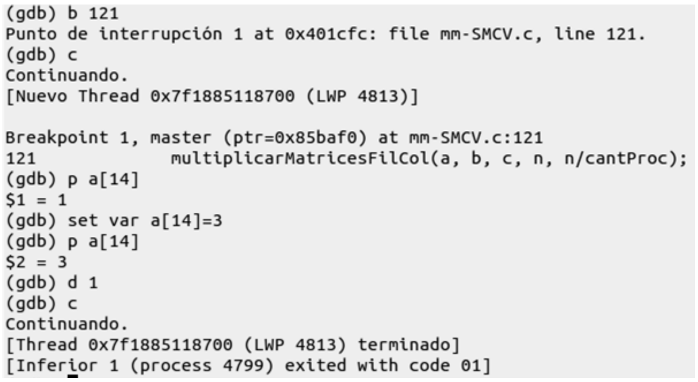
```
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$ mpirun -np 5 mm-SMCV 10
PID 4583 on 0 ready for attach
PID 4586 on 3 ready for attach
PID 4584 on 1 ready for attach
PID 4587 on 4 ready for attach
Restan 10 segundos...
PID 4585 on 2 ready for attach
Restan 9 segundos...
Restan 8 segundos...
Restan 7 segundos...
Restan 6 segundos...
Restan 5 segundos...
Restan 4 segundos...
Restan 3 segundos...
Restan 2 segundos...
Restan 1 segundos...
MM-SMCV;5;10;11.065258;11.049720;0.015538
```

Fig. 2. *Output of a run with no faults. The time to attach the debugger is shown.*



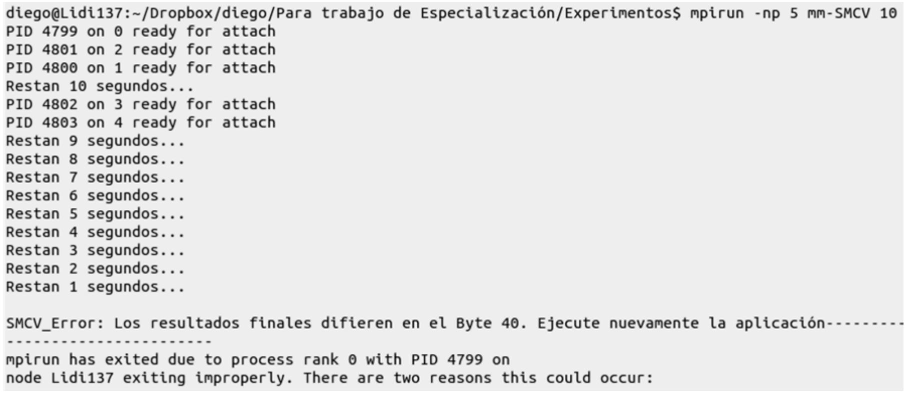
```
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$ sudo gdb -q -pid=4746
Adjuntando a process 4746
Leyendo símbolos desde /home/diego/Dropbox/diego/Para trabajo de Especialización/Experimentos/mm-SMCV...hecho.
```

Fig. 3. *Example showing how to attach the debugger to inject faults.*



```
(gdb) b 121
Punto de interrupción 1 at 0x401cfc: file mm-SMCV.c, line 121.
(gdb) c
Continuando.
[Nuevo Thread 0x7f1885118700 (LWP 4813)]

Breakpoint 1, master (ptr=0x85baf0) at mm-SMCV.c:121
121             multiplicarMatricesFilCol(a, b, c, n, n/cantProc);
(gdb) p a[14]
$1 = 1
(gdb) set var a[14]=3
(gdb) p a[14]
$2 = 3
(gdb) d 1
(gdb) c
Continuando.
[Thread 0x7f1885118700 (LWP 4813) terminado]
[Inferior 1 (process 4799) exited with code 01]
```

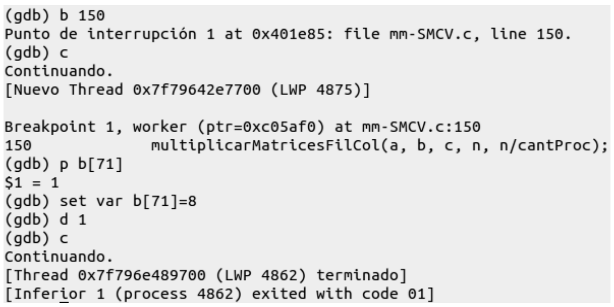
Fig. 4. *Injection of a fault that causes FSC.*



```
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$ mpirun -np 5 mm-SMCV 10
PID 4799 on 0 ready for attach
PID 4801 on 2 ready for attach
PID 4800 on 1 ready for attach
Restan 10 segundos...
PID 4802 on 3 ready for attach
PID 4803 on 4 ready for attach
Restan 9 segundos...
Restan 8 segundos...
Restan 7 segundos...
Restan 6 segundos...
Restan 5 segundos...
Restan 4 segundos...
Restan 3 segundos...
Restan 2 segundos...
Restan 1 segundos...

SMCV_Error: Los resultados finales difieren en el Byte 40. Ejecute nuevamente la aplicación---------
----------------------
mpirun has exited due to process rank 0 with PID 4799 on
node Lidi137 exiting improperly. There are two reasons this could occur:
```

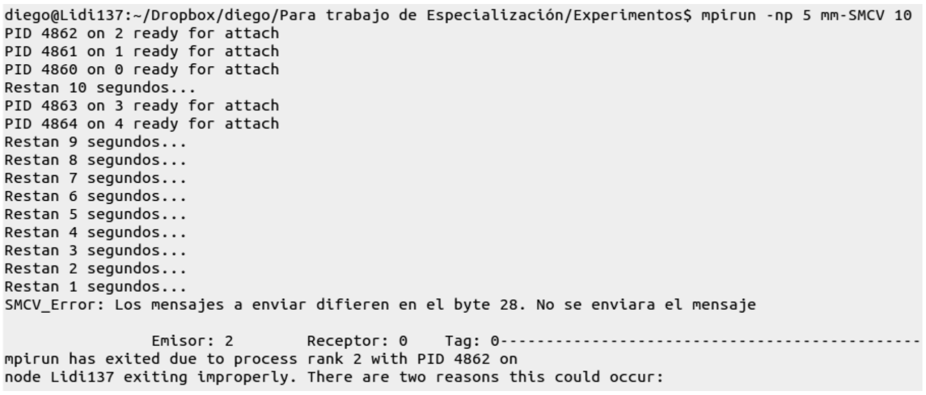
Fig. 5. *Output when a FSC occurred using SMCV as detection strategy.*

For the experiments, five processes were used (one Master and four Workers) and 10x10 square matrixes, so each of the five processes calculates two rows of matrix C. Even though this size does not really require parallel execution, it is used solely to show the consequences of failure injection and SMCV's detection capabilities. The experimental platform is an Intel Core i5-2310 2.9Ghz CPU with 6MB L3 cache memory and 8GB RAM, and the operating system is GNU/Linux Ubuntu 14.04.
Figure 2 shows a normal run of the application, with no fault injection. The initial count corresponds to the time used to attach the debugger to one of the processes, in order to simulate a fault that affects data used by that process. Figure 3 shows how the debugger is attached to perform the injection experiments.
Figure 4 shows the procedure carried out to inject a fault during the execution of the Master process in one of the first 20 elements in matrix A (those kept for local computation), after executing function `MPI_Scatter` but before the multiplication operation. This situation simulates the occurrence of a failure that corrupts a datum that is used for computing the result, but is never transmitted to other process in the application, causing FSC. Figure 5 shows the output of the application, with error detection and safe stop.
Figure 6 shows the injection of a fault during the operation of a Worker process in an element of matrix B after the execution of `MPI_Broadcast` but before the multiplication operation. This allows simulating the corruption of a datum that is part of the calculation carried out by that Worker. The results of these calculations are transmitted to the Master in the subsequent `MPI_Gather`, so the incorrect result (calculated using the altered value) is detected as TDC. Figure 7 shows the output of the application, with error detection and safe stop. Since the fault caused TDC, the output message is different from that of the previous case.



```
(gdb) b 150
Punto de interrupción 1 at 0x401e85: file mm-SMCV.c, line 150.
(gdb) c
Continuando.
[Nuevo Thread 0x7f79642e7700 (LWP 4875)]

Breakpoint 1, worker (ptr=0xc05af0) at mm-SMCV.c:150
150           multiplicarMatricesFilCol(a, b, c, n, n/cantProc);
(gdb) p b[71]
$1 = 1
(gdb) set var b[71]=8
(gdb) d 1
(gdb) c
Continuando.
[Thread 0x7f796e489700 (LWP 4862) terminado]
[Inferior 1 (process 4862) exited with code 01]
```

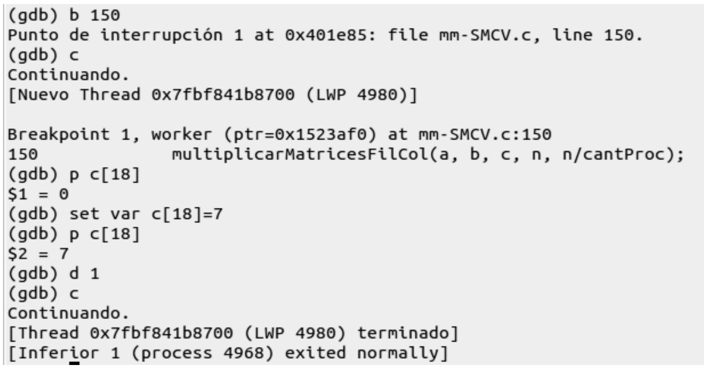
Fig. 6. *Injection of a fault that causes TDC.*



```
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$ mpirun -np 5 mm-SMCV 10
PID 4862 on 2 ready for attach
PID 4861 on 1 ready for attach
PID 4860 on 0 ready for attach
Restan 10 segundos...
PID 4863 on 3 ready for attach
PID 4864 on 4 ready for attach
Restan 9 segundos...
Restan 8 segundos...
Restan 7 segundos...
Restan 6 segundos...
Restan 5 segundos...
Restan 4 segundos...
Restan 3 segundos...
Restan 2 segundos...
Restan 1 segundos...
SMCV_Error: Los mensajes a enviar difieren en el byte 28. No se enviara el mensaje

                Emisor: 2       Receptor: 0     Tag: 0---------------------------------------------
mpirun has exited due to process rank 2 with PID 4862 on
node Lidi137 exiting improperly. There are two reasons this could occur:
```

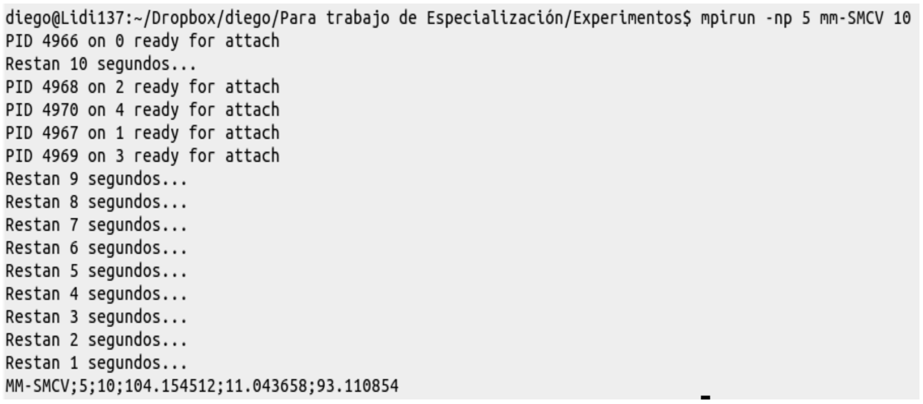
Fig. 7. *Output when TDC occurred using SMCV as detection strategy.*

Figure 8 shows the injection of a fault on an element of matrix C for one of the Workers. The subsequent multiplication operation overwrites the altered value, so the failure results in a LE. Consequently, Figure 9 shows that the output is normal and correct.
Finally, Figure 10 shows the application output when a fault that causes TO has occurred; both detection and the safe stop can be seen. In this case, the failure is injected on a variable that acts as index, making one of the Worker replicas to restart its computation after it has already done part of its task. This causes a difference in time between the progress of both redundant threads, which is detected as a TO error. The ideal consequence of a failure that causes TO is that the process enters an infinite loop, but this behavior cannot be forced in the selected application with a simple failure.
It should be noted that the time at which the failure is assumed to have occurred is configurable. There is no optimal value; it depends on each particular application. To clarify this, detection through TO is based on the premise that, in an application that is run on a dedicated homogeneous system, the execution times of two replicas that carry out the same computation should be similar [14]. Therefore, a notorious difference in processing times assumes that both replicas have separated their flows due to a silent fault. Thus, TO time should be configured based on what is to be expected for the application: if this value is too high, detection latency will increase; if it is too low, a small difference in computation times will result in the detection of a false positive. In the previous test, the injected failure only causes an abnormal delay in synchronization. A short time was deliberately configured to show that the mechanism can react to this event. However, if one of the processes went into an infinite loop, SMCV would be effective in detecting an error.



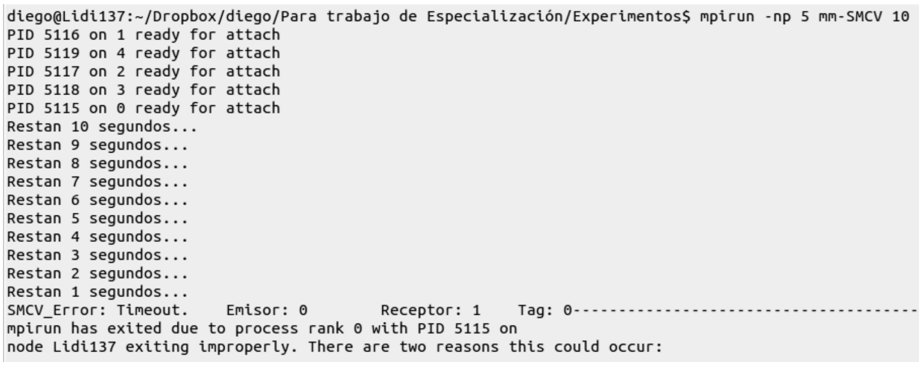
```
(gdb) b 150
Punto de interrupción 1 at 0x401e85: file mm-SMCV.c, line 150.
(gdb) c
Continuando.
[Nuevo Thread 0x7fbf841b8700 (LWP 4980)]

Breakpoint 1, worker (ptr=0x1523af0) at mm-SMCV.c:150
150             multiplicarMatricesFilCol(a, b, c, n, n/cantProc);
(gdb) p c[18]
$1 = 0
(gdb) set var c[18]=7
(gdb) p c[18]
$2 = 7
(gdb) d 1
(gdb) c
Continuando.
[Thread 0x7fbf841b8700 (LWP 4980) terminado]
[Inferior 1 (process 4968) exited normally]
```

Fig. 8. *Injection of a fault that causes LE.*



```
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$ mpirun -np 5 mm-SMCV 10
PID 4966 on 0 ready for attach
Restan 10 segundos...
PID 4968 on 2 ready for attach
PID 4970 on 4 ready for attach
PID 4967 on 1 ready for attach
PID 4969 on 3 ready for attach
Restan 9 segundos...
Restan 8 segundos...
Restan 7 segundos...
Restan 6 segundos...
Restan 5 segundos...
Restan 4 segundos...
Restan 3 segundos...
Restan 2 segundos...
Restan 1 segundos...
MM-SMCV;5;10;104.154512;11.043658;93.110854
```

Fig. 9. *Output of the execution when LE occured.*



```
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$ mpirun -np 5 mm-SMCV 10
PID 5116 on 1 ready for attach
PID 5119 on 4 ready for attach
PID 5117 on 2 ready for attach
PID 5118 on 3 ready for attach
PID 5115 on 0 ready for attach
Restan 10 segundos...
Restan 9 segundos...
Restan 8 segundos...
Restan 7 segundos...
Restan 6 segundos...
Restan 5 segundos...
Restan 4 segundos...
Restan 3 segundos...
Restan 2 segundos...
Restan 1 segundos...
SMCV_Error: Timeout.    Emisor: 0       Receptor: 1    Tag: 0-----------------------------------
mpirun has exited due to process rank 0 with PID 5115 on
node Lidi137 exiting improperly. There are two reasons this could occur:
```

Fig. 10. *Output when TO occurred using SMCV as detection strategy.*

5. Conclusions and Future Work

As HPC systems are scale and the likelihood of node failures and SDC increases, the need to protect the data and obtaining availability at low cost becomes even more critical. Redundancy is a viable solution for detecting SDCs in the context of HPC. 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.
Based on the tests carried out, it is concluded that SMCV is capable of detecting failures that affect message contents, notifying the user and leading the application to a safe step so that the data corruption does not propagate. On the other hand, in the case of failures that affect data that are kept for local computation, and those that occur during the final phase (corresponding to the FSC fraction), are detected when comparing the results. Finally, those failures that result in considerable asymmetries in the computation times of the replicas are detected through a TO mechanism.
Our future work will include completing a transient failure-tolerant methodology that incorporates a recovery mechanism that is based on multiple incremental distributed checkpoints, so that a process can store information about the failure that occurred in another process, and thus determine if the last checkpoint is valid or if a previous one should be used for recovery [10].

References

1. Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., & Snir, M.: Toward exascale resilience: 2014 update. Supercomputing frontiers and innovations, 1(1) (2014).
2. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., & Brightwell, R.: 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 (p. 78). IEEE Computer Society Press (2012).
3. Zheng, Z., Yu, L., Tang, W., Lan, Z., Gupta, R., Desai, N., & Buettner, D.: Co-analysis of RAS log and job log on Blue Gene/P. In Parallel & Distributed Processing Symposium (IPDPS), IEEE International (pp. 840-851) (2011).
4. Borkar, S., & Chien, A.: The future of microprocessors.Communications of the ACM, 54(5), 67-77 (2011).
5. Moody, A., Bronevetsky, G., Mohror, K., & De Supinski, B. R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for (pp. 1-11). IEEE (2010).
6. Elliott, J., Hoemmen, M., & Mueller, F.: Evaluating the impact of SDC on the GMRES iterative solver. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International (pp. 1193-1202). IEEE (2014).
7. Li, D., Vetter, J. S., & Yu, W.: Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In Proceedings of

the International Conference on High Performance Computing, Networking, Storage and Analysis (p. 57). IEEE Computer Society Press (2012).
8. Snir, M., Wisniewski, R. W., Abraham, J. A., Adve, S. V., Bagchi, S., Balaji, P., ... & Van Hensbergen, E.: Addressing failures in exascale computing. International Journal of High Performance Computing Applications (2014).
9. Shye, A., Blomstedt, J., Moseley, T., Reddi, V. J., Connors, D. A.: PLR: A software approach to transient fault tolerance for multicore architectures; IEEE Transactions on Dependable and Secure Computing. 6(2), pp. 135-148 (2009).
10. Lu, G., Zheng, Z., & Chien, A.: When is multi-version checkpointing needed? In Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale (pp. 49-56). ACM (2013).
11. Hari, S. K. S., Adve, S. V., & Naeimi, H.: Low-cost program-level detectors for reducing silent data corruptions. In Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on (pp. 1-12). IEEE (2012).
12. Yalcin, G., Unsal, O. S., & Cristal, A.: Fault tolerance for multi-threaded applications by leveraging hardware transactional memory. InProceedings of the ACM International Conference on Computing Frontiers (p. 4). ACM (2013).
13. Montezanti, D., Frati, F.E., Rexachs, D., Luque, E., Naiouf, M.R., De Giusti, A.: SMCV: a Methodology for Detecting Transient Faults in Multicore Clusters.; CLEI Electron. J. 15(3), pp. 1-11 (2012).
14. Montezanti, D., Rucci, E., Rexachs, D., Luque, E., Naiouf, M.R., De Giusti, A.: A tool for detecting transient faults in execution of parallel scientific applications on multicore clusters; Journal of Computer Science & Technology , 14(1), pp. 32-38 (2014).
15. Chen, Z.: Algorithm-based recovery for iterative methods without checkpointing. In Proceedings of the 20th international symposium on High performance distributed computing (pp. 73-84). ACM (2011).
16. Engelmann, C., Ong, H., & Scott, S. L.: The case for modular redundancy in large-scale high performance computing systems. In Proceedings of the IASTED International Conference (Vol. 641, p. 046) (2009).
17. Ferreira, K., Stearley, J., Laros III, J. H., Oldfield, R., Pedretti, K., Brightwell, R., ... & Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (p. 44). ACM (2011).
18. Ferreira, K., Riesen, R., Oldfield, R., Stearley, J., Laros, J., Pedretti, K., & Brightwell, T.: rMPI: increasing fault resiliency in a message-passing environment. Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488 (2011).
19. Engelmann, C., & Böhm, S.: Redundant execution of HPC applications with MR-MPI. In Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) (pp. 15-17) (2011).
20. Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., & Engelmann, C.: Combining partial redundancy and checkpointing for HPC. InDistributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on (pp. 615-626). IEEE (2012).
21. Ni, X., Meneses, E., Jain, N., & Kalé, L. V.: ACR: automatic checkpoint/restart for soft and hard error protection. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (p. 7). ACM (2013).
22. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I.: SWIFT: Software Implemented Fault Tolerance. In: Proceedings of the International Symposium on Code generation and optimization, pp. 243–254. IEEE Press, Washington DC (2005).