Caracterización de una estrategia de detección de fallos transitorios en HPC

Diego Montezanti[1,4], Dolores Rexachs[2], Enzo Rucci[1,3]
Emilio Luque[2], Marcelo Naiouf[1] y 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 Tecnológicas
[4] Instituto de Ingeniería y Agronomía, UNAJ
Av. Calchaquí 6200, 1888 Florencio Varela (Buenos Aires), Argentina

Resumen. El manejo de fallos es una preocupación creciente en HPC; en el futuro, se esperan mayores variedades y tasas de errores, intervalos de detección más largos y fallos silenciosos. Se proyecta que, en sistemas de exa-escala, los errores ocurran varias veces al día y se propaguen para generar desde caídas de procesos hasta corrupciones de resultados, con fallos no detectados en aplicaciones que siguen operando. En este trabajo se estudia una metodología de detección de fallos transitorios (SMCV) en aplicaciones MPI basada en replicación de software, asumiendo que la corrupción en los datos se manifiesta produciendo mensajes diferentes entre réplicas. SMCV permite obtener ejecuciones fiables con resultados correctos o, en su defecto, conducir al sistema a una parada segura. Se presenta una caracterización completa, definiendo formalmente el comportamiento frente a fallos y validándolo experimentalmente para mostrar la eficacia y viabilidad para detectar fallos transitorios en sistemas de HPC.

Palabras clave: fallos transitorios, detección, aplicaciones paralelas científicas, corrupción silenciosa de datos, HPC, inyección de fallos.

1 Introducción

El estancamiento en las frecuencias de reloj de los procesadores ha llevado a que la mejora de rendimiento se logre mediante el aumento en la cantidad de componentes. El hecho de escalar los sistemas conlleva el problema de la disminución de la tensión que, junto con los desafíos de la miniaturización submicrónica generan grandes incrementos en las tasas de fallos. Las interferencias electromagnéticas generan pulsos de corriente que alteran valores almacenados o en lógica combinatoria. Las variabilidades mayores en los procesos de fabricación ocasionan comportamientos inconsistentes, mientras que el envejecimiento vuelve más frecuentes los errores permanentes, y también los fallos múltiples han aumentado su probabilidad [1,2]. Debido a todo esto la fiabilidad de los sistemas se ha vuelto crítica, especialmente en

el área de de HPC con más de cientos de miles de *cores*. Estudios recientes en supercomputadoras modernas muestran que los tiempos medios entre fallos (MTBF-*Mean Time Between Failures*) son de pocas horas [3] y se estima que podrían llegar a unos 30 minutos en grandes aplicaciones paralelas en plataformas de exa-escala. En consecuencia, estas aplicaciones no podrán progresar eficientemente sin la ayuda adecuada [4,5]. La preocupación principal recae sobre los fallos silenciosos (SDC-*Silent Data Corruption*), habiendo aparecido numerosos reportes y estudios sobre sus probabilidades e impactos [2,6,7]. Al tener la capacidad de invalidar los resultados, los SDC crean graves problemas para la ciencia, que confía cada vez más en simulaciones de gran escala. Por todo esto, la mitigación de SDC es uno de los grandes retos de la resiliencia presente y futura.

Los SDC se generan en la forma de *bit-flips* (cambio en el valor de un bit) que afectan al almacenamiento o a los *cores*. Para detectarlos o corregirlos, los fabricantes colocan ECCs (*Error Correcting Code*) más potentes en la memoria, protegen los buses con bits de paridad y agregan redundancia a los circuitos de algunas unidades lógicas [8]. Sin embargo, resulta demasiado costoso incorporar redundancia hardware en los registros y en las ALUs del procesador [9].

El pequeño mercado de los supercomputadores, que requieren de alta fiabilidad, se puede satisfacer con soluciones de redundancia doble y triple para lograr detección y corrección, respectivamente. Aunque esto conlleva un alto costo, es preferible a tener resultados corruptos. Los SDC permanecen latentes hasta que los datos alterados son utilizados, siendo las latencias de detección dependientes de las aplicaciones.

El método estándar para manejar errores más utilizado en los sistemas paralelos actuales (en especial los que ejecutan aplicaciones MPI), es la realización de *checkpoints* periódicos. En caso de falla, el método de *Checkpoint/Restart* (C/R) relanza la aplicación desde el último *checkpoint*. Desafortunadamente, el *overhead* de C/R aumenta con el número de *cores*. Tomando en cuenta el tiempo requerido en C/R y re-ejecución, se podría desperdiciar gran cantidad de tiempo de cómputo útil si el MTBF es muy bajo. La situación empeora si el cómputo es fuertemente acoplado, porque un error en un nodo puede propagarse a los demás en microsegundos [1,10].

El modelo tradicional basado en C/R asume que la detección se produce casi de inmediato. Además, si el *checkpoint* almacenado contiene fallos no detectados, no se podrá efectuar la recuperación. Las pocas técnicas generales de detección introducen altos *overheads* en aplicaciones paralelas [2,11]. En base a lo anterior, se espera que aumenten los rangos de latencia de detección, incrementando el problema debido a los SDC. Tampoco hay mecanismos eficientes de contención, por lo que un fallo que afecta a una tarea puede resultar en la caída de la aplicación o en salidas incorrectas que, en el mejor caso, sólo se detectan cuando ha finalizado la ejecución y que son muy difíciles de corregir.

La replicación a nivel de procesos se ha mostrado como una alternativa fiable, pero para que sea atractiva en HPC aún debe resolver desafíos como minimizar el *overhead* temporal y de utilización de recursos, garantizar que los estados internos de las réplicas sean equivalentes (lo cual no es trivial, ya que se podrían ejecutar operaciones no determinísticas) y reducir el consumo energético. La forma clásica de detectar los SDC consiste en replicar ejecuciones y comparar sus resultados. RedMPI [2] lo hace a nivel de procesos aunque existen otros que lo hacen a nivel de *threads* [12]. También se han explorado soluciones que requieren menos recursos y relajan la

precisión, como la replicación aproximada, que proporciona límites superior e inferior para el resultado del cómputo [1].

En este contexto, en los últimos años se ha propuesto la metodología SMCV [13,14], diseñada para detectar fallos transitorios en HPC, específicamente para aplicaciones científicas que utilizan MPI en *clusters* de *multicores*. SMCV permite obtener ejecuciones fiables con resultados correctos o, al menos, reportar la ocurrencia de SDC y conducir al sistema a una parada segura luego de una latencia de detección acotada, ahorrando así un tiempo significativo, especialmente en aplicaciones largas.

El resto del documento se organiza de la siguiente manera. La Sección 2 repasa algunos conceptos básicos mientras que la Sección 3 describe trabajos relacionados. La Sección 4 presenta una caracterización de la estrategia SMCV, en la cual se define formalmente su comportamiento frente a fallos, su esfera de replicación (SoR-*Sphere of Replication*) y sus vulnerabilidades. La Sección 5 describe los experimentos realizados de inyección de fallos controlada de forma de validar el comportamiento definido y mostrar la eficacia y viabilidad de SMCV para detectar fallos transitorios en sistemas de HPC. Por último, la Sección 6 presenta las conclusiones y las líneas de trabajo futuras.

2 Conceptos básicos

Dependiendo de sus efectos sobre la ejecución de las aplicaciones, los fallos transitorios pueden clasificarse de la siguiente forma [13]:

- Error Latente (LE-*Latent Error*): afecta datos que no son utilizados posteriormente, por lo que no tiene impacto en los resultados.
- Error Detectado Irrecuperable (DUE-*Detected Unrecoverable Error*): causa una anomalía detectable para el software del sistema, sin posibilidad de recuperación; suele producir que la aplicación finalice de forma abrupta.
- Error por *Time-Out* (TO): el programa no finaliza dentro de un lapso de tiempo determinado.
- Corrupción Silenciosa de Datos (SDC): no es detectada por ningún nivel de software del sistema y sus efectos se propagan para producir la finalización con salida incorrecta. En aplicaciones paralelas con paso de mensajes, pueden causar: Corrupción de Datos Transmitidos (TDC-*Transmitted Data Corruption*), que afecta a datos que forman parte del contenido de mensajes a transmitir (si no se detecta se propaga a otros procesos); o Corrupción de Estado Final (FSC-*Final Status Corruption*), donde los datos alterados no se transmiten, pero se propagan localmente, corrompiendo el estado final del proceso afectado.

3 Trabajo relacionado

Las tecnologías actuales no puedan lidiar con SDC frecuentes. Las soluciones algorítmicas existentes [15] sólo se pueden aplicar a *kernels* específicos, por lo que hay que evaluar mecanismos que permitan tratar con los errores que escapan de su alcance. En tanto, las estrategias de detección que se basan en el compilador o en

software de tiempo de ejecución se pueden aplicar a cualquier código aunque resultan más complejas.

La contención busca evitar que el daño causado por el fallo se propague a otros nodos o que corrompa datos de un *checkpoint*, volviendo imposible la recuperación [1]. En [16] se propone redundancia en sistemas de HPC, lo que permite incrementar la disponibilidad del sistema y ofrece un compromiso entre cantidad y calidad de componentes. En [17] se muestra que la replicación es más eficiente que C/R en situaciones donde MTBF es bajo y el *overhead* temporal de C/R es alto. Soluciones de redundancia por software se enfocan en replicación a nivel de *threads* [12], de procesos [9] y de estado de máquina para eliminar la necesidad de hardware costoso.

MR-MPI [19] es otra propuesta para redundancia transparente en HPC, que ofrece replicación parcial (sólo se replican algunos procesos); se puede complementar con C/R en los procesos no replicados [20,21].

rMPI [18] es un protocolo para ejecución redundante de aplicaciones MPI, enfocado en fallos que causan la parada del sistema, y que utiliza la capa de *profiling* para realizar interposición de funciones MPI. Cada nodo tiene una réplica, de modo que ante un fallo permanente, el nodo redundante continúa sin interrupciones; la aplicación falla si fallan dos réplicas correspondientes. La redundancia escala, es decir, la probabilidad de fallo simultánea entre un nodo y su réplica decrece cuando aumenta la cantidad de nodos, a costa de duplicar la cantidad de recursos utilizados y cuadruplicar el número de mensajes. En tanto, RedMPI [2] es una bliblioteca MPI que aprovecha la replicación de procesos de rMPI para detectar y corregir SDC, comparando en el receptor mensajes enviados por emisores replicados. Implementa una optimización basada en *hashing* para evitar enviar todos los mensajes y comparar sus contenidos completos. No requiere modificaciones al código de la aplicación y asegura que las réplicas se ejecutan de manera determinística. Los resultados muestran que puede proteger las aplicaciones incluso con tasas de fallos altas con *overheads* temporales menores a 30%, por lo que potencialmente puede ser utilizado en sistemas de gran escala. Una contribución de [2] es el análisis de la propagación de los SDC entre nodos a través de las comunicaciones MPI, mostrando que incluso un único fallo transitorio puede tener un profundo efecto sobre la aplicación, causando un patrón de corrupción en cascada hacia todos los demás procesos.

Al igual que SMCV, al enfocarse en los mensajes, RedMPI monitorea los datos más críticos para la aplicación; la corrección en las comunicaciones es necesaria para la corrección de la salida. Como el SDC puede afectar a datos que no se comunican inmediatamente, el fallo es detectado al momento de la trasmisión. Sin embargo, a diferencia de SMCV, RedMPI realiza la validación del lado del receptor. Esto se debe a que, en el emisor, todas las réplicas se deben comunicar con las demás para verificar internamente sus contenidos antes de enviar el mensaje. Esto incurre en *overhead* y latencia adicionales, ya que el receptor pierde todo ese tiempo antes de poder continuar. Como SMCV replica a nivel de *threads* y no de procesos, no necesita hacer circular mensajes entre los emisores para validar. Al enviar sólo un mensaje luego de la validación, no congestiona la red. Al igual que SMCV, RedMPI permite que, aún sin corrección, la corrupción quede confinada en un proceso. También permite personalizar el *mapping* de las réplicas en el mismo nodo físico que los procesos nativos (o en sus vecinos con menor latencia de red).

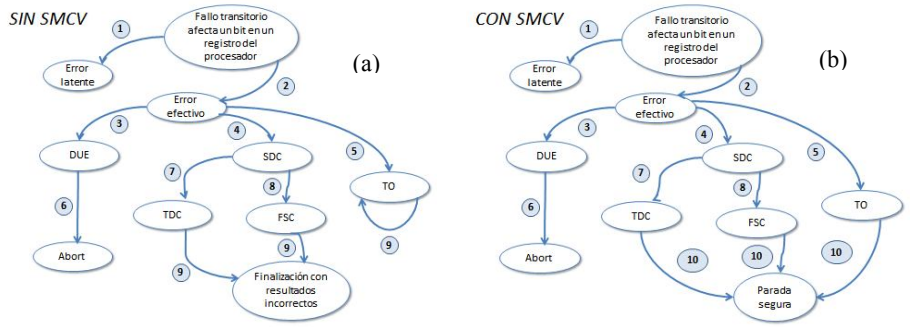
4 Caracterización de la metodología SMCV

4.1 Breve revisión de la metodología SMCV

SMCV es una estrategia de detección basada en la validación de contenidos de mensajes que se van a enviar entre procesos de aplicaciones paralelas determinísticas. Está diseñado para detectar los fallos que causan SDC (en sus dos formas) y TO. SMCV duplica en un *thread* cada proceso de la aplicación, requiriendo mecanismos de sincronización entre ambas réplicas concurrentes. Cuando se va a realizar una comunicación, el *thread* detiene su ejecución a la espera de que su réplica alcance el mismo punto, y todos los campos del mensaje, calculados por ambas replicas, se comparan en la búsqueda de diferencias. En caso de coincidencia, sólo uno de los hilos envía el mensaje, evitando que un error se propague a otros procesos sin consumir ancho de banda adicional. El receptor se sincroniza con su réplica, recibe el mensaje y realiza una copia para ella, continuando ambas con su ejecución. Al concluir, se verifican los resultados para detectar fallos que se puedan haber propagado localmente hasta el final de la aplicación.

4.2 Comportamiento frente a fallos

En esta sección se define el comportamiento de la metodología de detección. En la Fig. 1(a) se muestra un diagrama de los estados posibles de una ejecución cuando no hay ninguna estrategia implementada, mientras que en la Fig. 1(b) se ve el mismo diagrama cuando se aplica SMCV. Las elipses representan estados y los arcos representan eventos que producen transiciones de un estado a otro. Las transiciones aparecen numeradas, por lo que se describe cada una de ellas.



1. El bit afectado no se utiliza.
2. El bit afectado es utilizado por la aplicación.
3. El bit alterado afecta datos controlados por el sistema operativo.
4. El bit alterado afecta datos de la aplicación del usuario.
5. El bit alterado produce que la aplicación no responda en un tiempo límite.
6. El sistema operativo detecta el fallo y aborta la aplicación.
7. El dato afectado se transmite a otro proceso de la aplicación paralela.
8. El dato afectado sólo es utilizado por el proceso local.
9. Transcurre tiempo de ejecución.
10. SMCV detecta el fallo al cabo de un tiempo y conduce a una parada segura.

Fig. 1(a) Diagrama de estados de la ejecución sin estrategia de detección de fallos.
(b) Ejecución aplicando la estrategia de detección SMCV.

4.3 Esfera de replicación

La esfera de replicación [9] es un concepto comúnmente aceptado para describir el dominio lógico de ejecución redundante de una técnica particular y especificar los límites para la detección de fallos. Todos los datos que ingresan en la SoR son replicados, la ejecución dentro de su ámbito es redundante de alguna forma y los datos de salida son comparados para asegurar su corrección antes de abandonarla. Toda la ejecución por fuera de la SoR no está cubierta frente a fallos y debe ser protegida de otra manera. El concepto original de SoR se utilizó para definir los límites de la fiabilidad en diseños de hardware redundante, ubicándola alrededor de unidades específicas. Sin embargo, su aplicación resulta inadecuada para las propuestas implementadas en software, a pesar de lo cual algunas soluciones que utilizan el compilador para insertar instrucciones redundantes han intentado imitar una SoR centrada en hardware [22]. En tanto, el paradigma de detección de fallos centrado en software ubica la SoR alrededor de capas de software [9]; esto pone de manifiesto que, aunque los fallos afectan al hardware, sólo son relevantes los que influyen sobre la corrección de la aplicación, mientras que los que permanecen latentes pueden ignorarse sin riesgo. Sin embargo, esto tiene la desventaja de retardar la detección hasta que se produce certeza del error por datos inválidos que salen de la SoR, lo que implica que un fallo puede permanecer latente por tiempo indeterminado.

SMCV es una técnica de software y, en consecuencia, adopta una SoR centrada en software. Su objetivo es detectar fallos que afectan datos que se manipulan dentro de los registros del procesador, que constituyen la parte más vulnerable del computador, debido a la dificultad de implementar protección por hardware. Como se explicó, SMCV replica en un *thread* el cómputo que realiza cada proceso de la aplicación paralela. Cada *thread* opera sobre una copia local de los datos de entrada, que se genera para que pueda computar de forma independiente de su réplica. Por lo tanto, la SoR se encuentra alrededor de la aplicación del usuario y sus datos, sin incluir al sistema operativo ni a la biblioteca de comunicaciones. Aunque la memoria se halla por fuera de la SoR de SMCV, es aconsejable no utilizar variables globales, ya que constituyen puntos de falla centralizados. Si ocurre un fallo que altera su valor, ambos *threads* redundantes utilizarán el valor erróneo y, si no ocurre otro fallo, SMCV no detectará ningún error.

4.4 Fallos múltiples y vulnerabilidades

La mayoría de las propuestas existentes son capaces de detectar fallos si se asume que ocurre un único *bit-flip* a lo largo de la ejecución, pero no son tan efectivas para fallos que afectan a múltiples bits. Afortunadamente, existen sólo dos situaciones en las que fallos múltiples se pueden combinar para causar inconvenientes. La primera es aquella en la que el mismo bit es alterado en ambas réplicas, por lo que la comparación resulta correcta y el fallo no se detecta. La segunda se da cuando un fallo afecta a una de las réplicas, y el resultado de la verificación también es alterado, enmascarando el fallo original. Sin embargo, estas combinaciones tienen una probabilidad de ocurrencia muy baja, por lo que pueden ser ignoradas sin riesgos serios. Todas las demás combinaciones de fallos múltiples son detectadas como fallos simples ni bien se registra la primera diferencia en una comprobación [22]. SMCV es

capaz de detectar cualquier fallo transitorio simple que cause SDC o TO, pero no soporta fallos múltiples relacionados.

Todas las técnicas de tolerancia a fallos tienen vulnerabilidades, es decir, circunstancias en las cuales no son capaces de detectar los fallos que influyen efectivamente sobre la ejecución. Las características de diseño de una estrategia y las pruebas a las que es sometida (usualmente mediante inyección de fallos) deben permitir explicitar esas vulnerabilidades.

Las vulnerabilidades están normalmente asociadas con fallos que afectan al propio mecanismo de detección [9], y SMCV no es la excepción. SMCV minimiza el retardo entre la comprobación y la utilización de los valores validados, ya que realiza la verificación al momento de utilizar los datos en un mensaje; de esta manera reduce la probabilidad de fallo durante el lapso que transcurre entre ambos eventos (como ocurre en [22]); una vez que los datos están en el *buffer* de salida se encuentran fuera de la SoR. En tanto, la comprobación de valores a enviar resulta un punto centralizado de falla. Si la comprobación resulta incorrecta luego de una ejecución correcta, se ha producido un falso positivo y se genera una parada segura cuando en realidad el problema fue introducido por el mismo detector. Esta vulnerabilidad se puede mejorar duplicando la comparación; sin embargo, a pesar de no ser completamente fiable, la redundancia parcial en general es suficiente para cubrir los requerimientos de los usuarios [9]. En tanto, si la comprobación resulta correcta tras una ejecución incorrecta, el fallo ha quedado oculto tras la comprobación, debido a un segundo fallo. Como se mencionó, SMCV no puede lidiar con esta situación, aunque la probabilidad de que ocurra es extremadamente baja [22]. Además, resulta importante considerar el hecho de que SMCV es capaz de detectar como TO otros fallos que constituirían vulnerabilidades si no se contara con dicho mecanismo. Por ejemplo, si un código de operación es modificado, de modo que la instrucción resultante es el envío de un mensaje, o si ocurre un fallo durante la ejecución de la herramienta, ambas réplicas separan sus flujos de ejecución. Ante el envío de un mensaje por parte de una de ellas, no se sincronizan adecuadamente, por lo que el fallo se detecta al transcurrir un lapso mayor al determinado.

5 Validación de la eficacia de detección

Se realizaron una serie de pruebas para validar la eficacia de detección de SMCV. La aplicación utilizada fue una multiplicación de matrices paralela MPI (`C=A×B`) bajo el paradigma *Master/Worker*, en la que el *Master* toma parte en el cómputo del resultado [13]. La aplicación opera de siguiente forma:

- El proceso *Master* divide la matriz `A` entre todos los *Workers*, y, mediante la función `MPI_Scatter`, envía un trozo a cada uno, manteniendo él mismo uno para calcular su parte de la matriz resultado.
- El *Master* envía a cada *Worker* una copia de la matriz B completa mediante la función `MPI_Broadcast`.
- Todos los procesos computan su trozo correspondiente de la matriz `C`, enviando luego lo que han calculado al proceso *Master* mediante la función `MPI_Gather`.

- El *Master* construye la matriz `C` a partir de lo que los *Workers* le han enviado y lo que él mismo ha calculado.

Para la validación, se adaptó la aplicación para integrarla con la funcionalidad de la herramienta SMCV de la forma descripta en [14]. Para esto se requiere modificar el código fuente de la aplicación y la posterior recompilación. El experimento consistió en inyectar fallos, controladamente, en varios puntos de la aplicación, por medio de la herramienta de depuración GDB[1]. Para esto, se inserta un *breakpoint* en la ejecución de uno de los procesos, se modifica el valor de una variable y se retoma la ejecución; de esta manera se simula un *bit-flip* en un registro del procesador, ya que la corrupción se manifiesta si se puede observar una diferencia entre los estados de memoria de las réplicas. Pese a que un fallo transitorio puede ocurrir aleatoriamente en cualquier lugar y momento de la ejecución, para la inyección controlada se seleccionaron puntos significativos, tanto en el cómputo realizado por el *Master* como por los *Workers*.

Para los experimentos se utilizaron cinco procesos (un *Master* y cuatro *Workers*) y matrices cuadradas de 10 x 10, por lo que cada uno de los cinco procesos calcula dos filas de la matriz `C`. Aunque este tamaño no requiere realmente de una ejecución paralela, se utiliza con la única finalidad de mostrar las consecuencias de la inyección de fallos y la capacidad de detección de SMCV. La plataforma experimental se compone de una CPU Intel Core i5-2310 2.90Ghz con 6MB de memoria caché L3 y 8GB RAM, y el sistema operativo es GNU/Linux Ubuntu 14.04.

En la Fig. 2(a) se muestra una ejecución normal de la aplicación, sin inyección de fallos. El conteo inicial corresponde al lapso utilizado para adjuntar el depurador a alguno de los procesos, de manera de simular un fallo que afecta un dato utilizado por dicho proceso. En la Fig. 2(b) se muestra la forma en la que se adjunta el depurador para realizar los experimentos de inyección.

(a)

```
diego@Lidl137:~/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
```

(b)

```
diego@Lidl137:~/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. 2(a) Salida de una ejecución sin fallos. Se muestra el tiempo para adjuntar el depurador. **(b)** Ejemplo de cómo adjuntar el depurador para inyectar fallos.

En la Fig. 3(a) se muestra el procedimiento realizado para inyectar un fallo durante la operación del *Master*, en uno de los primeros 20 elementos de la matriz `A` (los que conserva para su cómputo local), después de la ejecución de la función `MPI_Scatter` pero antes de la multiplicación. Esta situación simula la ocurrencia de un fallo que corrompe un dato que interviene en el cómputo del resultado, pero nunca es transmitido a otro proceso de la aplicación, produciendo FSC. En la Fig. 3(b) se ve la salida de la aplicación, con detección del error y parada segura.

[1] GDB se encuentra disponible en www.gnu.org/software/gdb/

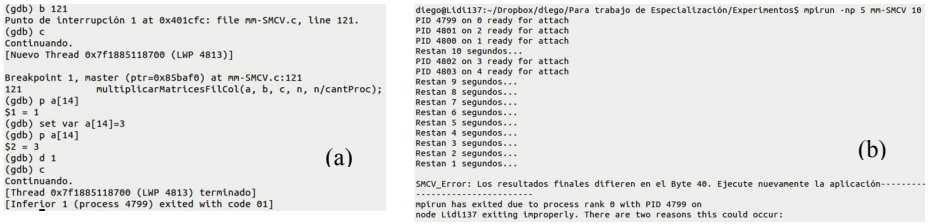


Fig. 3(a) Inyección de un fallo que causa FSC. **(b)** Salida con la detección de SMCV.

En la Fig. 4(a) se muestra la inyección de un fallo durante la operación de un *Worker* en un elemento de la matriz `B`, después de la ejecución de `MPI_Broadcast` pero antes de la multiplicación. De esta forma, se simula la corrupción de un dato que interviene en el cálculo que realiza ese *Worker*. Los resultados de estos cálculos se transmiten al *Master* en el `MPI_Gather` posterior, por lo que el resultado incorrecto (calculado a partir del valor alterado) es detectado como TDC. La Fig. 4(b) nuevamente muestra la salida de la aplicación, con detección del error y parada segura. Como el fallo ha causado TDC, el mensaje es diferente del caso anterior.

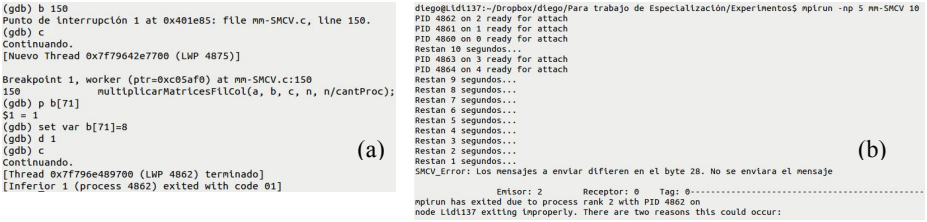


Fig. 4 (a) Inyección de un fallo que causa TDC. **(b)** Salida con la detección de SMCV.

En la Fig. 5(a) se muestra la inyección de un fallo en un elemento de la matriz `C` en uno de los *Workers*. La multiplicación posterior sobrescribe el valor alterado, por lo que el fallo produce un LE. En consecuencia, en la Fig. 5(b) se ve que la salida es normal y correcta.

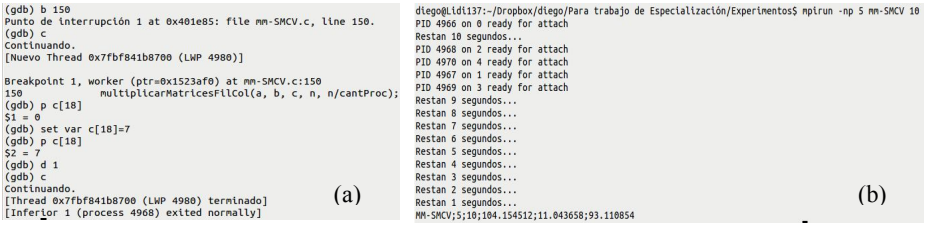


Fig. 5(a) Inyección de un fallo que causa LE. **(b)** Salida de la ejecución.

Finalmente, en la Fig. 6 se observa la salida de la aplicación cuando ha ocurrido un fallo que produce TO; se puede ver la detección y parada segura. En este caso, el fallo se inyecta en una variable que actúa como índice, produciendo que una de las réplicas del *Worker* recomience su cómputo cuando ya ha realizado parte de su tarea. Esto ocasiona un desfase temporal entre los progresos de ambos hilos redundantes, que es

detectado como un error por TO. La consecuencia ideal de un fallo que produzca TO es que el proceso ingrese a un bucle infinito, pero en la aplicación seleccionada no se puede provocar este comportamiento mediante un fallo simple.

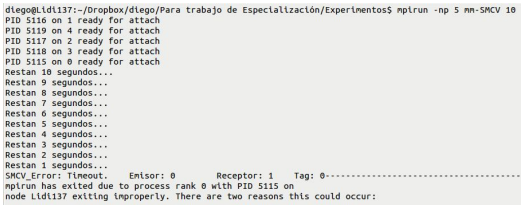


Fig. 6 Salida con la detección de TO por SMCV.

Es importante destacar que el lapso transcurrido, el cual se asume la ocurrencia de un fallo, es configurable. No existe un valor óptimo, sino que depende de la aplicación particular. Para clarificar esto, se debe considerar que la detección por TO parte de la premisa de que, en una aplicación que se ejecuta sobre un sistema homogéneo dedicado, los tiempos de ejecución de dos réplicas que realizan el mismo cómputo deben ser similares [14]. Por lo tanto, una asimetría notoria en los tiempos de procesamiento supone que ambas réplicas han separado sus flujos a causa de un fallo silencioso. Por lo tanto, se debe configurar el lapso de TO de acuerdo a lo esperable para la aplicación: si el lapso resulta muy alto, aumentará la latencia de detección; si es demasiado bajo, una pequeña asimetría en los tiempos de cómputo resultará en la detección de un falso positivo. En el caso de prueba anterior, el fallo inyectado sólo provoca una demora anormal en la sincronización. Se configuró adrede un lapso breve para mostrar que el mecanismo es capaz de reaccionar frente a este evento. Sin embargo, si uno de los procesos ingresara en un bucle infinito, SMCV detectaría efectivamente un error.

6 Conclusiones y trabajo futuro

A medida que los sistemas de HPC escalan y aumenta la probabilidad de fallos de nodo y SDC, se vuelve aún más crítica la necesidad de proteger datos y obtener disponibilidad a bajo costo. La redundancia es una solución viable para la detección de SDC en el ámbito de HPC. El hecho de que un único SDC produzca efectos profundos en todos los procesos que se comunican permite concluir que la protección de aplicaciones a nivel de mensajes de MPI es un método factible y efectivo para detectar, aislar y prevenir corrupción de datos posterior.

A partir de las pruebas realizadas, se concluye que SMCV es capaz de detectar los fallos que afectan contenidos de mensajes (TDC), notificando al usuario y produciendo parada segura para que la corrupción no se pueda propagar. Por otra parte, los fallos que afectan a datos que se mantienen para cómputo local, y los que ocurren en la fase final (correspondientes a la fracción FSC) son detectados durante la comparación de resultados. En tanto, los fallos que producen asimetrías considerables en los tiempos de cómputo de las réplicas son detectadas por medio del mecanismo de TO.

El trabajo futuro consiste en completar una metodología tolerante a fallos transitorios, incorporando a la detección un mecanismo de recuperación basado en múltiples *checkpoints* incrementales distribuidos, de forma de poder almacenar en un proceso la información sobre el fallo ocurrido en otro, y así determinar si el último *checkpoint* es válido o si hay que retroceder a uno anterior para recuperar [10].

Referencias

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).