SEDAR: Detección y Recuperación Automática de Fallos Transitorios en Sistemas de Cómputo de Altas Prestaciones

DIEGO MIGUEL MONTEZANTI

FACULTAD DE INFORMÁTICA
UNIVERSIDAD NACIONAL DE LA PLATA

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas

Fecha: Febrero de 2020

Director y Co-director (UNLP):
Ing. Armando De Giusti y Dr. Marcelo Naiouf

Director y Co-director (UAB):
Dra. Dolores Rexachs y Dr. Emilio Luque

Dirección conjunta por convenio de colaboración entre Universidad Nacional de La Plata y la Universidad Autónoma de Barcelona

Agradecimientos

Una de las cosas de las que estoy más firmemente convencido en la vida es de la importancia de ser agradecido. Durante este largo camino recorrido, muchas han sido las personas que, de una u otra forma, han ayudado a hacer posible que hoy pueda ver el final.

Quiero agradecer en primer lugar a mis queridos directores de la UAB, Dolores Rexachs y Emilio Luque. Es difícil dimensionar la paciencia, perseverancia y ayuda que he recibido de ellos durante todos estos años, en tantas reuniones de seguimiento y de discusión sobre los avances, y su infinita generosidad cada vez que compartí con ellos una estancia de investigación en el Campus de Bellaterra. Me han aportado muchas ideas y devoluciones con rigor científico, pero sobre todo me han demostrado su enorme calidad humana. Han sido realmente una guía imprescindible para mi trabajo en todo este tiempo. A ellos, todo mi agradecimiento.

También a mis directores en Argentina, Armando de Giusti y Marcelo Naiouf. Durante todos estos años me han acompañado y apoyado con paciencia, escuchando mis dificultades, corrigiendo mi trabajo y poniendo a mi alcance todos los medios y recursos para desarrollar mi tarea en el III - LIDI y para poder viajar a realizar provechosas estancias de investigación en la UAB.

Quiero agradecer enormemente a mi querido amigo y admirado investigador Enzo Rucci, por su colaboración permanente y desinteresada, en especial durante toda la última parte de este trabajo. Ha puesto a disposición todo su conocimiento, ayudándome con el diseño y la implementación del trabajo experimental, y leyendo, aconsejando y corrigiendo gran parte de las publicaciones que constituyen el cuerpo de esta Tesis. Parece una frase hecha, pero no lo es: sin su ayuda, la concreción de este trabajo no hubiese sido posible.

A mi amigo Emmanuel Frati, que me dio un impulso fundamental durante la etapa inicial de mi investigación, ayudándome con los primeros diseños experimentales y discutiendo mis resultados. A Jorge Villamayor, a quien conocí en la UAB y con quien hemos discutido ideas

y desarrollos que resultaron en avances significativos para la etapa final.

A mis amigos y compañeros del III - LIDI, en especial a Diego Encinas y a Pablo Thomas, con quienes he mantenido numerosas charlas, y quienes siempre se han preocupado por mi y por el estado de mi trabajo.

A todas aquellas personas que me ayudaron con mi tarea y que hicieron más agradables y fructíferas mis estadías en la UAB, compartiendo momentos y experiencias de investigación. En especial a Joao Gramacho, Mónica Denham, Álvaro Wong, Remo Suppi, Sandra Méndez y Marcela Castro. También a Marina Morán, que en la última etapa se interesó por mi progreso, aportándome su apoyo y sus conocimientos.

A mi madre, que me ayudó cuidando a mi hijo muchas veces en las que tuve cursos y reuniones en la Facultad, y a mi padre, que siempre me motivó a seguir y a no desanimarme hasta cumplir el objetivo.

A mis suegros por su apoyo y compañía y por todas las veces que se hicieron presentes cuando estaba sobrepasado, encargándose de mi hijo y de muchas otras cosas.

A tantos amigos que estuvieron junto a mi todo este tiempo, y muchas personas que pasaron por mi vida, y de una u otra forma me ayudaron a llegar hasta aquí. Son muchos y no podría nombrarlos a todos sin ser injusto.

Finalmente, quiero agradecerle a mi familia: a mi esposa Débora, que durante todos estos años fue como una roca firme a mi lado, sosteniéndome, animándome y no dejándome caer, confiando en que podría alcanzar el final de este camino aún cuando yo pensaba en que no lo lograría. A mi hijo Juan Diego, que me enseñó y me enseña tantas cosas cada día, de esas que no se aprenden investigando según métodos científicos; a nuestro pequeño hijo en camino y a la familia que pudimos formar.

Y en último, pero primer lugar, agradezco a Dios, mi Salvador, quien siempre estuvo a mi lado, sosteniéndome y poniendo las personas correctas en mi camino; ninguna meta en mi vida podría haber alcanzado si no fuera gracias a Él.

Resumen

El manejo de fallos es una preocupación creciente en el contexto del HPC; en el futuro, se esperan mayores variedades y tasas de errores, intervalos de detección más largos y fallos silenciosos. Se proyecta que, en los próximos sistemas de exa-escala, los errores ocurran incluso varias veces al día y se propaguen en grandes aplicaciones paralelas, generando desde caídas de procesos hasta corrupciones de resultados debidas a fallos no detectados.

En este trabajo se propone SEDAR, una metodología que mejora la fiabilidad, frente a los fallos transitorios, de un sistema que ejecuta aplicaciones paralelas de paso de mensajes. La solución diseñada, basada en replicación de procesos para la detección, combinada con diferentes niveles de *checkpointing* (*checkpoints* de nivel de sistema o de nivel de aplicación) para recuperar automáticamente, tiene el objetivo de ayudar a los usuarios de aplicaciones científicas a obtener ejecuciones confiables con resultados correctos. La detección se logra replicando internamente cada proceso de la aplicación en *threads* y monitorizando los contenidos de los mensajes entre los *threads* antes de enviar a otro proceso; además, los resultados finales se validan para prevenir la corrupción del cómputo local. Esta estrategia permite relanzar la ejecución desde el comienzo ni bien se produce la detección, sin esperar innecesariamente hasta la conclusión incorrecta. Para la recuperación, se utilizan *checkpoints* de nivel de sistema, pero debido a que no existe garantía de que un *checkpoint* particular no contenga errores silenciosos latentes, se requiere el almacenamiento y mantenimiento de múltiples *checkpoints*, y se implementa un mecanismo para reintentar recuperaciones sucesivas desde *checkpoints* previos si el mismo error se detecta nuevamente. La última opción es utilizar un único *checkpoint* de capa de aplicación, que puede ser verificado para asegurar su validez como punto de recuperación seguro. En consecuencia, SEDAR se estructura en tres niveles: **(1)** sólo detección y parada segura con notificación al usuario; **(2)** recuperación basada en una cadena de *checkpoints* de nivel de sistema; y **(3)** recuperación basada en un único *checkpoint* válido de capa de aplicación. Cada una de estas variantes brinda una

cobertura particular, pero tiene limitaciones inherentes y costos propios de implementación; la posibilidad de elegir entre ellos provee flexibilidad para adaptar la relación costo-beneficio a las necesidades de un sistema particular. Se presenta una descripción completa de la metodología, su comportamiento en presencia de fallos y los *overheads* temporales de emplear cada una de las alternativas. Se describe un modelo que considera varios escenarios de fallos y sus efectos predecibles sobre una aplicación de prueba para realizar una verificación funcional. Además, se lleva a cabo una validación experimental sobre una implementación real de la herramienta SEDAR, utilizando diferentes *benchmarks* con patrones de comunicación disímiles. El comportamiento en presencia de fallos, inyectados controladamente en distintos momentos de la ejecución, permite evaluar el desempeño y caracterizar el *overhead* asociado a su utilización. Tomando en cuenta esto, también se establecen las condiciones bajo las cuales vale la pena comenzar con la protección y almacenar varios *checkpoints* para recuperar, en lugar de simplemente detectar, detener la ejecución y relanzar. Las posibilidades de configurar el modo de uso, adaptándolo a los requerimientos de cobertura y máximo *overhead* permitido de un sistema particular, muestran que SEDAR es una metodología eficaz y viable para la tolerancia a fallos transitorios en entornos de HPC.

Palabras clave

Fallos transitorios, *soft errors*, detección, replicación de procesos, recuperación automática, corrupción silenciosa de datos, aplicaciones de HPC, *clusters* de multicores, inyección de fallos, *checkpoint* de nivel de sistema, *checkpoint* de capa de aplicación.

Abstract

Handling faults is a growing concern in the context of HPC; higher error rates and varieties, larger detection intervals, and silent faults are expected in the future. It is projected that, in upcoming exascale systems, errors will occur even several times a day and propagate across large parallel applications, increasing the occurrence of problems that will range from process crashes to corrupted results because of undetected errors.

In this work, we propose SEDAR, a methodology that improves system reliability against transient faults, when running parallel message-passing applications. The designed solution, based on process replication for detection, combined with different levels of checkpointing (i.e. system-level or user-level checkpoints) for automatic recovery, has the goal of helping users of scientific applications to achieve reliable executions with correct results. Detection is achieved by internally replicating each process of the application in threads and monitoring the contents of messages between threads before sending to another process; additionally, the final results are validated to prevent the corruption of the local computation. This strategy allows relaunching execution from the beginning upon detection, without unnecessarily waiting to an incorrect conclusion. To accomplish recovery, system-level checkpoints are used, but because there is no guarantee that a particular checkpoint does not contain silent latent corruption, multiple checkpoints need to be stored and maintained, and a mechanism is implemented for successively retry recovery from previous checkpoints if the same error is detected again. The last option is to use a single, custom user-level checkpoint, which can be verified to ensure its validity as a safe recovery point. Consequently, SEDAR is structured in three levels: **(1)** only detection and safe-stop with notification to the user; **(2)** recovery based on a chain of system-level checkpoints; and **(3)** recovery based on a single valid user-level checkpoint. Each of these variants supplies a particular coverage, but has inherent limitations and its own implementation costs; the possibility of choosing between them provides flexibility to adapt the cost-benefit relation to the needs of a particular sys-

tem. This work includes a full description of the methodology, its behavior in the presence of faults and the temporal overheads of employing each option. A model is described, which considers various scenarios of faults and their predictable effects over a test application in order to perform a functional verification. Additionally, an experimental validation is made across a real implementation of SEDAR tool, using different benchmarks involving unalike communication patterns. The behavior in the presence of faults, injected in a controlled manner in different moments during the execution, allows evaluating and characterizing the introduced overhead. Considering this, the conditions under which it is worth starting protection and storing several checkpoints for recovery, instead of simply detecting, stopping and relaunching, are stated. The possibility of configuring the mode of use, adapting to the coverage and maximum overhead requirements of a particular system, show the efficacy and viability of SEDAR to tolerate transient faults in target HPC environments.

Keywords

Transient faults, soft errors, detection, process replication, automatic recovery, silent data corruption, HPC applications, multicore clusters, fault injection, system-level checkpoint, user-level checkpoint.

Índice general

Prefacio

SEDAR: Apaciguar, sosegar, calmar, particularmente si se efectúa mediante la administración de algún fármaco.

En el área del Cómputo de Altas Prestaciones (*HPC - High-Performance Computing*), el número de componentes de los sistemas paralelos continúa en aumento, en la búsqueda de mejorar la performance, y, como consecuencia, la confiabilidad se ha vuelto un aspecto crítico. En las plataformas actuales, las tasas de fallos son de pocas horas, y está previsto que en los próximos sistemas de la exa-escala, las grandes aplicaciones paralelas tengan que lidiar con fallos que ocurran cada pocos minutos, por lo que requerirán ayuda externa para progresar eficientemente. Algunos reportes recientes han señalado a la Corrupción Silenciosa de Datos (*SDC*) como el tipo de fallo más peligroso que se puede presentar durante la ejecución de aplicaciones paralelas, ya que corrompen bits de la caché, de la memoria principal o de los registros de la CPU, produciendo que el programa finalice con resultados incorrectos, aunque en apariencia se ejecuta correctamente. Las aplicaciones científicas y las simulaciones a gran escala son las áreas más afectadas de la computación; en consecuencia, el tratamiento de los errores silenciosos es uno de los mayores desafíos en el camino hacia la resiliencia en los sistemas de HPC. En aplicaciones de paso de mensajes, y sin mecanismos de tolerancia apropiados, un fallo silencioso (que afecte a una única tarea) puede producir profundos efectos de corrupción de datos, causando un patrón de propagación en cascada, a través de los mensajes, hacia todos los procesos que se comunican; en el peor escenario, los resultados finales erróneos no podrán ser detectados al finalizar la ejecución y serán tomados como correctos. Si bien una forma posible de tolerar estos fallos es incorporar hardware redundante o especializado a los procesadores, esta es una solución costosa y difícil de implementar. Por lo tanto, se requieren soluciones de software específicas, con relaciones costo/beneficio adecuadas, para evitar la necesidad de realizar nuevamente la ejecución completa en caso

de detectar un fallo. Entre ellas, se destacan las estrategias basadas en replicación, en las cuales un proceso es duplicado, y ambas réplicas realizan la misma secuencia de ejecución; en aplicaciones determinísticas, ambas copias producen la misma salida para la misma entrada. En el contexto de HPC, la utilización de arquitecturas multicore con este fin representa una solución viable para detectar los fallos que producen *SDC*, debido a su redundancia natural. Las técnicas de *Checkpoint & Restart (C/R)* constituyen una alternativa bien conocida para restaurar las ejecuciones de aplicaciones paralelas cuando existen fallos permanentes en los sistemas, que causan caídas de procesos o de nodos de cómputo completos (*fail-stop*). En el *checkpointing* coordinado, por ejemplo, se almacena periódicamente el estado completo de una aplicación, de forma de que todos los procesos pueden recomenzar desde el último *checkpoint* almacenado si ocurre un fallo. Lamentablemente, las técnicas de C/R introducen un alto *overhead* temporal, que crece a medida que aumenta el tamaño del sistema. Sin embargo, su mayor limitación consiste en que no hay garantía, frente a la perspectiva de los errores silenciosos, de que un *checkpoint* determinado sea un punto seguro de recuperación, debido a que en su estado pueden haberse almacenado errores no detectados que permanecían latentes al momento de su almacenamiento. En este escenario que conjuga resultados no fiables y altos costos de verificación, en los últimos años se ha diseñado la metodología SEDAR (*Soft Error Detection and Automatic Recovery*), que proporciona tolerancia a fallos transitorios en sistemas formados por aplicaciones paralelas de paso de mensajes que se ejecutan en *clusters* de multicores. SEDAR es una solución basada en replicación de procesos y monitorización de los envíos de mensajes y el cómputo local, que aprovecha la redundancia de hardware de los multicores, en la búsqueda de brindar ayuda a programadores y usuarios de aplicaciones científicas a obtener ejecuciones confiables. SEDAR proporciona tres variantes para lograr la tolerancia a fallos: un mecanismo de detección y relanzamiento automático desde el comienzo de la aplicación; un mecanismo de recuperación automática, basada en el almacenamiento de múltiples *checkpoints* de nivel de sistema (ya sean periódicos o disparados por eventos); y un mecanimo de recuperación automática, basado en un único

checkpoint seguro de capa de aplicación. Esta tesis describe la metodología y sus pautas de diseño, y se enfoca en validar su eficacia para detectar los fallos transitorios y recuperar automáticamente las ejecuciones, mediante un modelo analítico de verificación. Si bien esta validación es esencialmente funcional, también se realiza una implementación práctica de un prototipo y se caracteriza también el comportamiento temporal, es decir, la *performance* y el *overhead* introducido por cada una de las tres alternativas. Además se intenta mostrar que existe la posibilidad de optar (incluso dinámicamente) por la alternativa que resulte más conveniente para adaptarse a los requerimientos de un sistema (por ejemplo, en cuanto a máximo *overhead* permitido o máximo tiempo de finalización), convirtiendo a SEDAR en una metodología eficaz, viable y flexible para la tolerancia a fallos transitorios en sistemas de HPC.

Lista de publicaciones

Los resultados de este trabajo de investigación han sido publicados en las siguientes revistas y congresos:

- D. Montezanti, E. Frati, D. Rexachs, E. Luque, M. Naiouf, y A. De Giusti, "*SMCV*: a methodology for detecting transient faults in multicore clusters", CLEI Electronic Journal, vol. 15, no. 3, paper 5, 2012.
- D. Montezanti, E. Rucci, D. Rexachs, E. Luque, M. Naiouf, y 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, pp. 32-38.
- D. Montezanti, D. Rexachs, E. Rucci, E. Luque, M. Naiouf, y A. De Giusti, "Characterizing a detection strategy for transient faults in HPC", in Computer Science Technology Series. XXI Argentine Congress of Computer Science. Selected papers. Editorial de la Universidad Nacional de La Plata (EDULP), 2016, pp. 77-90.

- D. Montezanti, A. De Giusti, M. Naiouf, J. Villamayor, D. Rexachs, y E. Luque, "A methodology for soft errors detection and automatic recovery", 2017 International Conference on High Performance Computing & Simulation (*HPCS*). IEEE, 2017, pp. 434-441.

- D. Montezanti, E. Rucci, D. Rexachs, E. Luque, M. Naiouf, y A. De Giusti, "SEDAR: Detectando y Recuperando Fallos Transitorios en Aplicaciones de HPC", XXV Argentine Congress of Computer Science - 2019. En prensa

- D. Montezanti, E. Rucci, D. Rexachs, E. Luque, M. Naiouf, y A. De Giusti, "Soft Errors Detection and Automatic Recovery based on Replication combined with different Levels of Checkpointing", *Future Generation Computer Systems* - En evaluación

Estructura de la Tesis

Este trabajo de tesis se organiza de la siguiente forma:

- El Capítulo 1 se enfoca en los fallos transitorios, sus causas de ocurrencia y sus efectos sobre la ejecución de los programas, especialmente en aplicaciones paralelas. Se describen los objetivos del presente trabajo y se presenta brevemente SEDAR, la propuesta desarrollada, y las principales contribuciones y limitaciones de la tesis.

- El Capítulo 2 revisa el trabajo relacionado con la tesis: las estrategias basadas en replicación para detección y recuperación de *soft errors*; las estrategias que incorporan hardware adicional y las puramente basadas en software; las técnicas de *checkpoint/restart* para fallos permanentes; las propuestas para algoritmos específicos y los intentos para obtener aplicaciones resilientes de paso de mensajes.

- El Capítulo 3 describe detalladamente *SMCV*, la metodología de detección de SEDAR, que proporciona detección de fallos para aplicaciones paralelas de paso de mensajes

mediante la replicación de procesos, validación de contenidos de mensajes y comparación de resultados finales. *SMCV* evita la propagación de los fallos al resto de la aplicación y restringe la latencia de detección y notificación de la ocurrencia del error, aprovechando además la redundancia intrínseca de hardware propia de las arquitecturas multicores, y brindando cobertura frente a fallos a costa de un *overhead* reducido.

- El Capítulo 4 describe las dos maneras alternativas que propone la metodología SEDAR para proporcionar un estado seguro, desde el cual pueda recuperarse una ejecución errónea y un mecanismo automático para recuperar la aplicación, permitiéndole concluir con resultados finales válidos. La primera de ellas se basa en la utilización de una cadena de múltiples *checkpoints* de capa de sistema, y la segunda se basa en la utilización de un único *checkpoint* seguro de capa de aplicación.

- El Capítulo 5 proporciona los detalles respecto de implementación de las funciones que posibilitan la detección y la recuperación automática, de forma de integrarlas en una librería SEDAR, constituyendo el prototipo de una herramienta de tolerancia a fallos transitorios que puede ser utilizada en conjunto con aplicaciones de HPC que utilizan paso de mensajes. Se describen las diferentes formas de utilización de la herramienta. Por otra parte, se incluye un modelo que contempla todos los posibles escenarios de fallo, y se describen las pruebas de inyección de fallos controlada, que verifican la correcta operación funcional de los mecanismos de detección y de recuperación automática basada en múltiples *checkpoints* de nivel de sistema.

- El Capítulo 6 se enfoca en los aspectos prestacionales, presentando una caracterización y evaluación del comportamiento temporal de las distintas estrategias propuestas. Para esto, SEDAR se utiliza en conjunto con tres *benchmarks* paralelos con diferentes patrones de comunicación y tamaños de *workloads*, obteniendo los parámetros de ejecución. Se evalúa cualitativamente la incidencia del patrón de comunicaciones, del intervalo de *checkpointing* y de la latencia de detección en el comportamiento

temporal. Se proveen elementos de análisis que permiten determinar el beneficio de emplear cada estrategia y de decidir cuándo es el instante más conveniente para comenzar la protección. Por último, se realizan mediciones del *overhead* de ejecución en escenarios realistas, tanto en ausencia de fallos como en presencia de fallos, aportando información para alcanzar la configuración óptima de la herramienta.

- El Capítulo 7 finaliza el trabajo, resumiendo las características más salientes de la metodología desarrollada y enumerando algunas líneas que quedan abiertas, de modo de plantear el trabajo que dará continuidad a la tesis.

Capítulo 1

Fallos Transitorios

Resumen

Este primer capítulo se enfoca en la descripción del fenómeno que ha motivado esta investigación: los fallos transitorios, sus causas de ocurrencia y sus posibles efectos sobre la ejecución de los programas, puntualizando lo que ocurre específicamente con las aplicaciones paralelas. Se describen, como consecuencia, los objetivos del presente trabajo. Se presenta brevemente la propuesta desarrollada para cumplir con el objetivo planteado, y las principales contribuciones y limitaciones de la tesis.

1.1. Introducción

La Tolerancia a Fallos (TF) se ha convertido en un requerimiento de relevancia creciente, en especial en el ámbito del Cómputo de Altas Prestaciones (HPC), debido al considerable incremento en la probabilidad de que ocurran fallos de diferentes clases en estos sistemas. Este incremento es consecuencia, fundamentalmente, de dos factores:

1. La creciente complejidad de los procesadores, en la búsqueda de mejorar la potencia computacional de los sistemas de cómputo. Esta complejidad está dada por el aumento en la escala de integración de sus componentes. Estos componentes, trabajando cerca de sus límites tecnológicos, son cada vez más propensos a la ocurrencia de fallos.

2. El rápido aumento del tamaño del sistema, entendido como la cantidad de nodos de procesamiento que lo componen, y la cantidad de cores por *socket*, de forma obtener mejores prestaciones en la resolución de problemas demandantes de potencia de cómputo. Como un dato representativo de esto, a mediados de 2018 la lista del TOP500 mostraba que el 95 % de las supercomputadoras listadas tenían entre 6 y 24 cores por *socket*.

En relación con estos aspectos de complejidad y tamaño, actualmente la escena de las supercomputadoras es dominada por los *clusters* de multiprocesadores *commodities*, interconectados por redes de comunicaciones de alta velocidad. Además, desde 2006 ha crecido en popularidad la utilización de dispositivos aceleradores, que proporcionan mejoras notables tanto en los tiempos de ejecución como en el consumo energético, si se los compara con los sistemas basados únicamente en las CPUs de propósito general [82].

A medida que las aplicaciones demandan mayores tiempos de cómputo ininterrumpido, la influencia de los fallos se vuelve más significativa, debido al costo que requiere relanzar una ejecución que fue abortada por la ocurrencia de un fallo, o, peor aún, que finaliza con resultados erróneos. Si las aplicaciones (y sus resultados) son relevantes, o incluso críticos, es necesario ejecutarlas sobre sistemas tanto de alta disponibilidad (es decir, que se mantienen funcionando un alto porcentaje de tiempo) como de alta fiabilidad (es decir, cuyo comportamiento no se aparta del especificado, proporcionando, por lo tanto, resultados correctos [120]). Las estrategias de tolerancia a fallos, capaces de proveer detección, protección y recuperación frente a fallos, se emplean en la búsqueda de obtener sistemas paralelos altamente disponibles y fiables.

La tolerancia a fallos se puede implementar a dos niveles diferentes: a nivel de hardware (o arquitectural) o a nivel software (o de aplicación). La técnica particular que se describe en este trabajo, y otras que se analizan y comparan, operan a nivel de software.

Por otra parte, la detección y la protección y recuperación de los fallos pueden verse como problemas relativamente independientes, ya que una vez que el fallo se detecta, la recupera-

ción puede realizarse de una variedad de formas diferentes; la estrategia de recuperación se ejecuta sólo en caso de la ocurrencia del fallo, en tanto que la detección se encuentra activa continuamente.

Dependiendo de su duración, los fallos pueden clasificarse en permanentes (que permanecen hasta que son reparados), intermitentes (que aparecen y desaparecen, bajo una combinación específica de circunstancias en el sistema; siguen apareciendo hasta que se tome alguna acción correctiva) o transitorios (que aparecen aleatoriamente y desaparecen solos al cabo de un tiempo [62]). En el resto de este trabajo, el foco estará puesto principalmente en los fallos transitorios.

El resto del capítulo tiene por objetivo definir y analizar los fallos transitorios, las métricas utilizadas en su estudio y sus consecuencias, en especial sobre sistemas paralelos en los que se ejecutan aplicaciones científicas intensivas en cómputo.

1.2. Concepto. Causas de ocurrencia

En general, se define un fallo como una imperfección o defecto físico que afecta a cualquier componente del sistema. A diferencia de los fallos llamados permanentes, los fallos transitorios no se reflejan en una disfuncionalidad permanente del sistema, ni ocurren de una manera consistente (como los fallos de diseño o fabricación). Un fallo permanente es capaz de causar errores o comportamientos inesperados cada vez que el componente afectado es utilizado, y permanece en el sistema hasta que dicho componente es reparado o reemplazado. En cambio, un fallo transitorio ocurre sólo una vez y no vuelve a repetirse de la misma manera durante el resto de la vida útil del componente, afectando temporariamente al hardware del sistema de cómputo. Comúnmente son consecuencia de alguna forma de interferencia, ya sea interna al sistema o externa (proveniente del entorno). Entre las fuentes externas se pueden mencionar la radiación cósmica o las partículas de alta energía que pueden causar pulsos de tensión en los circuitos digitales [36], mientras que las influencias internas están relacionadas con las altas temperaturas de operación y las variaciones

(ruidos) en la tensión de la fuente de alimentación [145]. Los fallos transitorios pueden afectar a los procesadores, a la memoria, a los buses internos y a los dispositivos, resultando habitualmente en la inversión del estado de un bit (*single bit flip*) en la ubicación afectada [11].

Por definición, los fallos transitorios son de corta duración, por lo que no son capaces de afectar la operación regular del sistema ni causar daño físico permanente a ningún componente. Sin embargo, dependiendo de su ubicación específica y su momento de ocurrencia, pueden corromper el cómputo, produciendo que el comportamiento de una aplicación sea diferente al esperado (por ejemplo, que intente escribir en una posición de memoria inválida o ejecutar una instrucción inválida). Este tipo de conductas causa que la aplicación sea abruptamente interrumpida por el mecanismo de $fail - stop$ del sistema operativo. Sin embargo, la situación más riesgosa ocurre cuando el $bit - flip$ resulta en modificaciones en el flujo de un programa o causa corrupción de datos que no puede ser detectada (y que, para empeorar las cosas, pueden propagarse), produciendo que la ejecución de una aplicación finalice con resultados incorrectos, que podrían nunca ser notados [96, 112, 119]. De hecho, han causado averías costosas en sistemas de altas prestaciones en los últimos años [11, 87].

La principal causa externa de fallos transitorios es la radiación electromagnética. ésta ocurre cuando partículas de alta energía inciden sobre zonas sensibles de los circuitos digitales, causando pulsos de tensión que interfieren con el funcionamiento normal. Un ejemplo son los neutrones de la atmósfera o las partículas alfa, que al incidir sobre dispositivos semiconductores, generan pares electrón-hueco. Las fuentes de los transistores y los nodos de difusión pueden almacenar esas cargas, acumulando la cantidad suficiente para invertir el estado de un dispositivo lógico, inyectando de esa forma un fallo en la operación del circuito (por ejemplo, invirtiendo un bit de una posición de memoria o de un registro del procesador) [99]. Esta situación se esquematiza en la Figura 1.1

Los fallos transitorios comenzaron siendo un problema para los diseñadores de sistemas de alta disponibilidad y computadoras que funcionan en ambientes electrónicamente hostiles

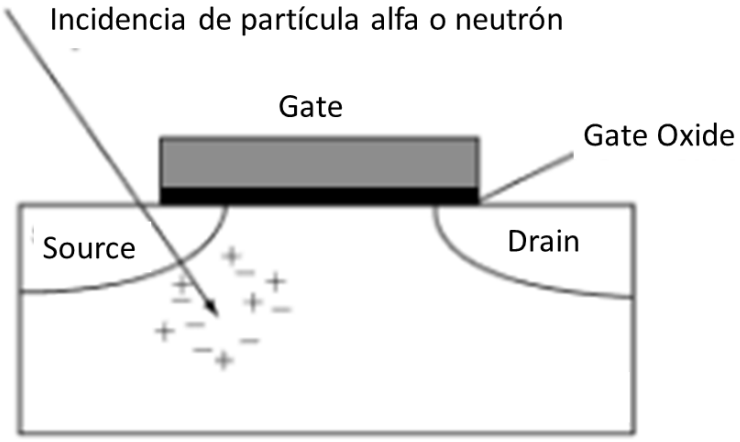

Figura 1.1: *Incidencia de partículas cargadas sobre dispositivos semiconductores*

como el espacio exterior [145], surgiendo en ese contexto las investigaciones acerca de sus causas y posibles consecuencias [42]. Sin embargo, la situación fue cambiando, y los reportes oficiales sobre los efectos de los fallos transitorios en grandes computadoras comenzaron a publicarse a partir del año 2000. Esos reportes evidenciaban los riesgos de fallos transitorios en HPC a causa de la gran cantidad de componentes funcionando en conjunto. Debido al peligro de que estos fallos afecten a los resultados del cómputo, los investigadores necesitaron forzar la ocurrencia de estos fallos para estudiar sus efectos y así poder evaluar su trabajo.

La evolución del proceso de miniaturización de los componentes de los procesadores para lograr una mejor $performance$, han ido llevando a aumentos en la cantidad de transistores, con mayores densidades y operando a tensiones más bajas. Todo estos factores han incidido en que los procesadores se hayan ido volviendo menos robustos frente a las influencias internas y externas que causan los fallos transitorios [145]. Con el crecimiento de la incidencia de la radiación terrestre, muchos sistemas comenzaron a implementar detección extensiva y/o corrección de errores, principalmente para las memorias $on-chip$. Sin embargo, proteger sólo la memoria resulta insuficiente para escalas de miniaturización menores a las tecnologías de 65nm.

La necesidad de protección frente a los efectos de los fallos transitorios motivaron nuevos

mecanismos $on-chip$ para proteger los $latches$ y los $flip-flops$. Eventualmente, incluso se requiere protección de la lógica combinatoria en los procesadores, a medida que siguen usando crecientes cantidades de transistores en el futuro [88].

Con la llegada de las computadoras con $multicores$ y $manycores$, la cantidad de transistores y buses en un procesador es tan grande que la industria de los $chips$ espera que los sistemas operativos, los $frameworks$ para cómputo paralelo e incluso las aplicaciones) deban lidiar frecuentemente con fallos transitorios [30].

1.3. Terminología asociada a los fallos transitorios

La terminología que se utiliza en este trabajo está basada en la taxonomía planteada en [9] y en otros trabajos relacionados [30, 62, 82]. Un fallo consiste en un defecto físico en el hardware. Un fallo determinado es capaz de producir uno o más errores latentes. Un error es la manifestación de un fallo en el sistema, es decir, un estado interno incorrecto. Un error latente se hace efectivo cuando el recurso en el que ocurre el error es utilizado para realizar algún cómputo. Por eso, el problema es que la detección de un error latente no es immediata, ya que el error es identificado sólo cuando los datos corrompidos son activados. El error efectivo puede propagarse de un componente a otro, causando nuevos errores. Dicho de otra forma, el fallo es una imperfección a nivel físico, mientras que un error es un estado interno inconsistente del sistema, a nivel de información [99].

Así como un error ocurre como consecuencia de un fallo, puede a su vez ser causa de una avería. Un error que se propaga conduce a una avería cuando causa una incorrecta funcionalidad o prestación del sistema, que se puede percibir externamente. Por lo tanto, una avería es la manifestación de un error en el servicio provisto por el sistema, observable a nivel externo. La avería ocurre cuando el sistema se comporta de un modo diferente del especificado. En general, la resiliencia se define como la colección de técnicas que se utilizan para mantener la ejecución de las aplicaciones hasta llegar a resultados correctos de manera oportuna y eficiente a pesar de los fallos subyacentes del sistema. Cualquier estrategia de

tolerancia a fallos tiene por objetivo, no impedir completamente los fallos (esto es imposible), sino evitar que éstos lleguen a transformarse en averías.

A continuación, se presenta un ejemplo que ilustra estos conceptos, que se encuentra esquematizado en la Figura 1.2:

1. Si una partícula cargada de energía incide sobre una celda de una memoria DRAM puede producir un fallo.

2. Una vez que este fallo altera el estado de la celda de memoria se genera un error latente.

3. El error permanece latente hasta que la posición de memoria afectada es leída por algún proceso, transformándose en un error efectivo.

4. La avería ocurre si la lectura de la posición de memoria alterada por el error afecta la operación del sistema, modificando su comportamiento.

Figura 1.2: *Propagación de un fallo que resulta en una avería*

En resumen, cuando el dato afectado por un fallo es activado, el fallo se convierte en un error efectivo; mientras el fallo permanezca inactivo, el error causado por él es latente. En tanto, cuando el error se propaga hasta ser apreciable desde el exterior del sistema, se convierte en una avería.

Los errores producidos por los fallos transitorios son llamados *soft errors*. El hecho de observar un *softerror* no implica que el sistema sea menos fiable que antes de su observación.

Los *soft errors* alteran datos pero no modifican físicamente al circuito afectado. Si los datos vuelven a escribirse, el circuito podrá funcionar perfectamente otra vez.

1.4. Métricas utilizadas

En los trabajos sobre fallos transitorios y *soft errors* se utilizan algunas métricas comúnmente usadas en tolerancia a fallos, y se incorporan algunas otras que facilitan la estimación de la probabilidad de fallos de un sistema. El tiempo medio hasta la ocurrencia de una avería ($MTTF$ – $Mean\ Time\ To\ Failure$) expresa, en promedio, el tiempo transcurrido entre el último arranque (o reinicio) del sistema hasta el próximo error del componente. El $MTTF$ de un componente se expresa normalmente en años y se obtiene en base a un promedio estimativo de predicción de fallos realizado por el fabricante del componente. El $MTTF$ de un sistema completo (conjunto de componentes) se puede obtener combinando los $MTTF$ de todos sus componentes [99]. En la Ecuación 1.1 se muestra el $MTTF$ del sistema completo.

$$MTTF_{system} = \frac{1}{\sum_{i=0}^{n} \frac{1}{MTTF_i}} \tag{1.1}$$

La utilización del término averías en el tiempo (FIT – $Failures\ in\ Time$) es más útil, a causa de su propiedad aditiva. Un FIT representa un error en un billón de horas. Para evaluar el FIT de un sistema, sólo es necesario sumar los FIT de todos sus componentes [99]. La estimación del valor FIT debido a los fallos transitorios recibe la denominación SER ($Soft\ Error\ Rate$). En la Ecuación 1.2 se muestra la tasa FIT de un sistema determinado.

$$FITrate = \sum_{i=0}^{n} FITrate_i \tag{1.2}$$

Bajo ciertas condiciones, las tasa FIT y el $MTTF$ de un componente son inversamente proporcionales. En la Ecuación 1.3 se muestra la relación entre FIT y $MTTF$ de un cierto sistema.

$$MTTF(years) = \frac{10^9}{FITrate * 24hs * 365days} \tag{1.3}$$

En el área de tolerancia a fallos existen dos métricas adicionales que se utilizan frecuentemente: el tiempo medio de reparación ($MTTR$ – *Mean Time To Repair*) y el tiempo medio entre averías ($MTBF$ – *Mean Time Between Failures*). El $MTTR$ es el tiempo requerido para reparar un error una vez detectado. En tanto, el $MTBF$ representa el tiempo promedio entre las ocurrencias de dos averías. El $MTBF$ se puede expresar como $MTBF = MTTF + MTTR$. En la Figura 1.3, adaptada de [99], se muestra la relación entre las métricas de tolerancia a fallos.

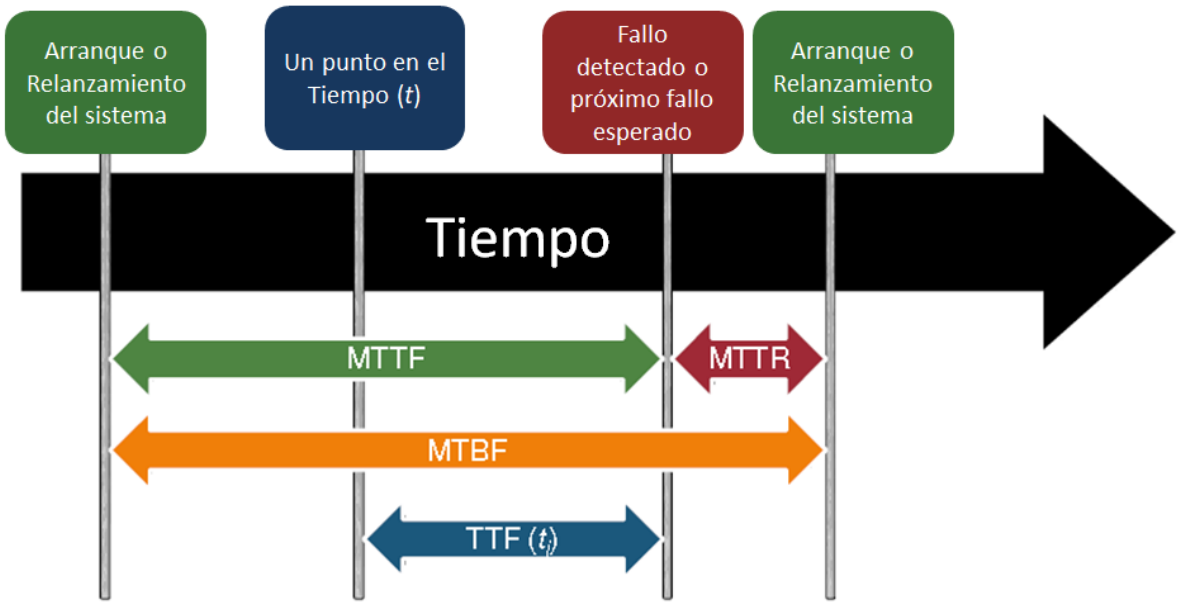

Figura 1.3: *Relación entre las métricas de tolerancia a fallos*

El factor de vulnerabilidad arquitectural (AVF - *Architectural Vulnerability Factor*) es una métrica para evaluar vulnerabilidad desde la perspectiva de la arquitectura, como la probabilidad de que ocurra un error en la salida del programa en caso de un fallo en una estructura del hardware [98]. Dado que no todos los fallos transitorios alteran la salida del programa, debido a que muchos permanecen latentes, el programa continúa su ejecución

sin verse afectado por el error. Los bits que influyen en la salida son llamados bits *ACE* (*Architecturally Correct Execution*), y se utilizan en el cómputo de la métrica *AVF*, que se estima como el número promedio de bits *ACE* sobre el número total de que existen esa estructura de hardware en un ciclo; cuanto más alto el valor de *AVF*, significa que es más probable tener un *soft error* e la correspondiente estructura de hardware.

Finalmente, una métrica que se utiliza específicamente en el contexto de los $soft\ errors$, y en particular, de los errores silenciosos, es el tiempo medio entre errores ($MTBE$ - $Mean\ Time\ Between\ Errors$). Esta métrica es la misma que $MTBF$, pero enfatiza que se trata de errores debidos a fallos transitorios, y no averías. En el Capítulo 6 se utiliza particularmente esta métrica.

1.5. Algunos casos reales

Como se mencionó anteriormente en la Sección 1.2, los fallos transitorios comenzaron siendo un problema para los diseñadores de sistemas críticos o que operan en ambientes hostiles desde el punto de vista de la radiación electromagnética (cuya intensidad se incrementa mucho en la altura), donde las consecuencias de un fallo pueden ser considerables. A modo de ejemplo, el satélite meteorológico chino FengYun 1(B), lanzado en 1990, quedó fuera de servicio antes de lo previsto, a causa de que el sistema de control de altitud perdió el control debido a fallos transitorios [81]. Hacia finales de los años 90, Boeing diseñó su aeronave 777 (por supuesto, un sistema crítico) con tres procesadores y buses de datos diferentes, utilizando un esquema de votación por mayoría para lograr tanto detección como recuperación de fallos [149]. Más recientemente, en marzo del 2013, el rover marciano Curiosity sufrió la incidencia de la rayos cósmicos en su sistema de memoria y debió activar su computador de respaldo para continuar su operación en modo seguro (en el cual sólo se realizan actividades mínimas de mantenimiento y control, pero permanece interrumpida la operación de los instrumentos científicos de a bordo), retrasando por lo tanto la misión. Sin embargo, hoy en día, los fallos transitorios constituyen una considerable fuente de errores

no sólo para circuitos que se desempeñan en el espacio exterior, sino también para equipamiento que opera dentro de la atmósfera a altitudes elevadas e incluso a nivel del suelo. En general, no hay demasiadas publicaciones que reporten la ocurrencia de *soft errors*. Las primeras evidencias provienen de la contaminación de la producción de circuitos integrados a fines de los años 70 y en los 80. A partir del año 2000, los reportes de *soft errors* en supercomputadoras o *server farms* se han vuelto más frecuentes y han causado averías costosas [11, 87]. Esto ocurre porque la cantidad de componentes en estas instalaciones es muy grande (con decenas de miles de procesadores *multicore* y de *terabytes* de memoria), lo que los hace potencialmente más vulnerables a los fallos transitorios (Sección 1.2). La susceptibilidad de un sistema a los fallos transitorios es normalmente impredecible durante el proceso de diseño y fabricación. Por ejemplo, durante la puesta en producción del supercomputador ASC Q (2003), los científicos del Laboratorio Nacional de Los Alamos documentaron una alta proporción de averías causadas por fallos transitorios [87, 133]. En el 2000, Sun Microsystems admitió que rayos cósmicos habían interferido con la operación de memorias caché, causando serios inconvenientes en servidores de sitios de sus principales clientes [11]. En la Figura 1.4 se esquematiza la evidencia de *soft errors* en sistemas reales.

Desafortunadamente, las tendencias actuales de diseño del hardware sugieren que las tasas de fallos irán en aumento. A esto contribuyen las altas frecuencias de operación, densidades de transistores en los *chips* y temperaturas internas, así como las pequeñas tensiones de alimentación y sus consecuentes bajos umbrales de ruido [62, 112]. Debido a una combinación de estos factores, la tasa de *soft errors* SER se viene incrementando aproximadamente un 8 % en cada generación de procesadores modernos [23]. En particular, los *latches*, que son circuitos potencialmente vulnerables a los fallos transitorios, son utilizados en varias estructuras lógicas y de datos internas, constituyendo una fracción significativa del área del procesador [27]. Los sistemas de alta disponibilidad requieren mucha más redundancia de hardware que la que proveen los *ECC*s (*Error Correcting Codes*) y los bits de paridad. Por ejemplo, históricamente, IBM ha añadido 20-30 % de lógica adicional en sus procesa-

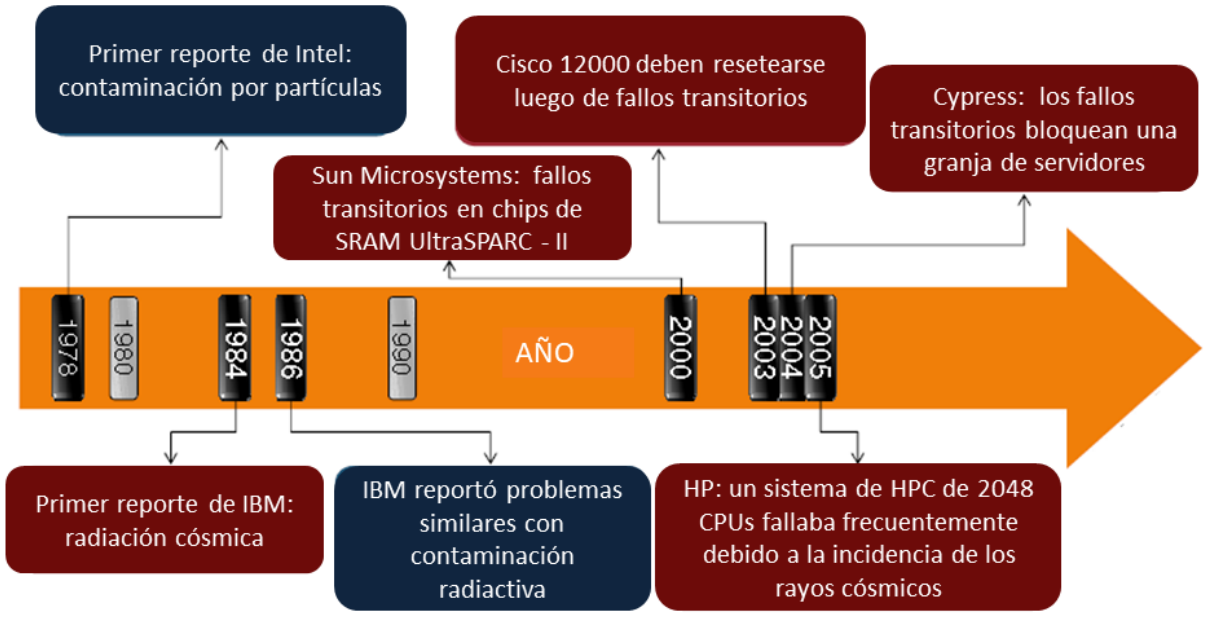

Figura 1.4: *Evidencia de soft errors en sistemas reales*

dores para *mainframes*. En el diseño del S/390 G5, IBM incorporó aún más redundancia, replicando completamente las unidades de ejecución del procesador para evitar varios problemas que tenía su enfoque anterior de tolerancia a fallos. Para minimizar el efecto de los fallos transitorios, en 2003 Fujitsu lanzó la quinta generación de procesadores SPARC64 con el 80 % de sus 200.000 *latches* con alguna forma de protección, incluyendo la generación de paridad en la ALU y una verificación de residuos en las operaciones de multiplicación y división [119]. Además, los *soft errors* son críticos en la operación de sistemas de gran escala, con cientos de miles de procesadores trabajan juntos para resolver un problema. En el año 2008, el supercomputador BlueGene/L contaba con 128.000 nodos, y experimentaba un *soft error* en su caché L1 cada 4-6 horas, debido a la desintegración radiactiva en las soldaduras de plomo. El supercomputador ASCI Q tenía una tasa de fallos de CPU inducidos por radiación de 26.1 por semana, mientras que para un computador de un tamaño similar, el Cray XD1, se estimaban 109 fallos semanales entre CPUs, memorias y $FPGA$s [27]. Más recientemente, una serie de estudios [128] han permitido llegar a la conclusión de que el

$MTBF$ depende principalmente de la cantidad de procesadores, resultando inversamente proporcional al tamaño del sistema. Por lo tanto, desde el punto de vista de la resiliencia, la escala es el gran enemigo [14]. Se proyecta de los sistemas de exa-escala contengan del orden de decenas o centenares de millones de cores dentro de la década actual; de hecho, el supercomputador que ocupa actualmente el tercer lugar de la lista del Top500 (es decir, en noviembre de 2019 - https://www.top500.org/list/2019/11/) tiene 10.649.600 cores (*Sunway TaihuLight*). En estas circunstancias, se espera que el $MTBF$ de los principales supercomputadores llegue a caer por debajo de los 10 minutos en los próximos años [72].

En particular, en el ámbito del HPC, donde estos sistemas ejecutan aplicaciones paralelas intensivas en cómputo y de alta duración, el impacto de relanzar la ejecución a causa de haber obtenido resultados incorrectos como consecuencia de los fallos, justifica la necesidad de adoptar estrategias de tolerancia a fallos para mejorar la robustez de estos sistemas.

Los errores silenciosos han producido que se reconsidere la replicación en el marco de las aplicaciones paralelas científicas que se ejecutan en plataformas de HPC de gran escala. Debido a que la replicación se da actualmente a nivel de procesos, la escala se vuelve un problema aún más grave [14]. Con millones de procesadores (y billones de *threads*), la probabilidad de errores durante las ejecuciones puede llegar a ser significativa, dependiendo de si los fabricantes de circuitos incrementen o no significativamente la protección sobre la lógica, los *latches*, los $flip-flops$ y los arreglos estáticos en los procesadores. En una publicación reciente, los autores consideran que, con un nivel significativo de protección (es decir, más hardware y consumo energético), la tasa FIT para los errores no detectados en un procesador podría mantenerse alrededor de 20. Pero, sin protección adicional (como en la situación actual) la tasa FIT para los errores no detectados podría ser cercano a 5.000 (es decir, 1 error cada 200.000 horas). Por lo tanto, para dar una idea de la dimensión del problema, la combinación de esta tasa FIT con más de 1.000.000 de cores, resultaría en 1 error cada 72 segundos [137].

1.6. Consecuencias de los fallos transitorios

La Figura 1.5 (adaptada de [96]) describe las todas las posibles consecuencias de la incidencia de una partícula energética en el procesador o la memoria de una computadora. Esta situación puede causar un fallo en el hardware, es decir, un fallo físico (ya sea permanente o transitorio), que puede resultar en los efectos que se describen a continuación.

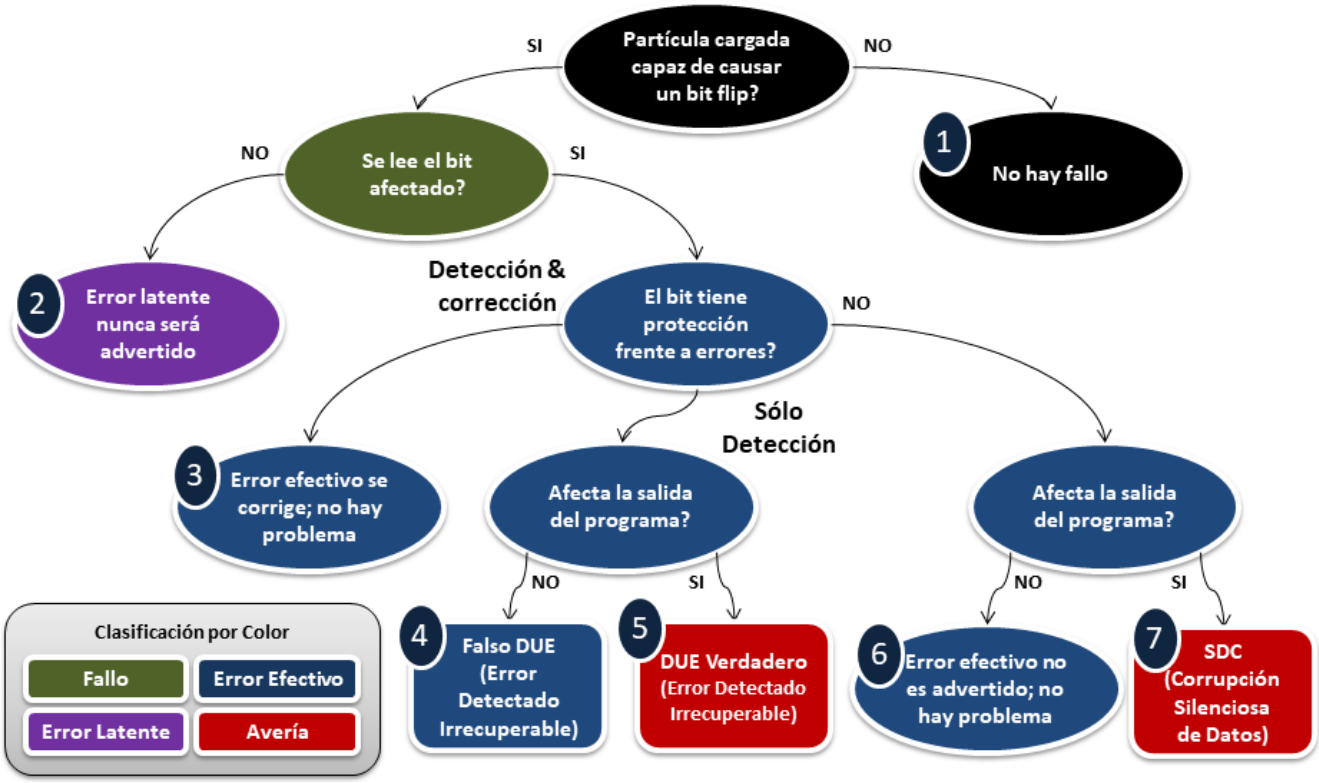

Figura 1.5: *Posibles consecuencias de los fallos transitorios*

La salida 1 de la Figura 1.5 indica que la energía de la partícula no es suficiente para generar un fallo. El error latente (*LE - Latent Error*, salida 2 de la Figura 1.5) ocurre cuando la energía de la partícula tiene energía suficiente para invertir un bit en la memoria, en un registro del procesador o en algún *latch*; si el bit alterado no es leído, o se sobreescribe en algún momento antes de ser utilizado, el error permanece latente, es decir, sin consecuencias perceptibles. En tanto, si el bit alterado es leído por algún componente del sistema, el *soft*

error se hace efectivo. Este $soft\ error$ efectivo puede pasar desapercibido para las capas superiores del componente que efectúa la lectura, en caso de que el bit alterado esté protegido con detección y corrección (salida 3 de la Figura 1.5). A este efecto se lo conoce también como "error detectado corregible" (DCE - $Detectable\ Correctable\ Error$) [82]. Los DCEs son manejados por medio de mecanismos de hardware, como es el caso, por ejemplo, de las memorias que utilizan ECCs, comprobaciones de paridad, o $Chipkill - Correct\ ECC$s, y resultan inadvertidos para las aplicaciones. Un tipo común de dispositivo de memoria que usa ECCs para mejorar su fiabilidad son las memorias dinámicas de acceso aleatorio ($DRAM$), que resultan más vulnerables a los fallos transitorios debido a su simplicidad estructural. Por lo tanto, cuentan con bits extra que pueden ser utilizados por los controladores de memoria para alamcenar la paridad de segmentos de bits. Sin embargo, si el bit afectado cuenta sólo con un mecanismo de detección de errores, conduce a un estado llamado "error detectado irrecuperable" (DUE - $Detected\ Unrecoverable\ Error$), en el que la prioridad es evitar que se generen salidas incorrectas. En el caso de un DUE, el componente que efectúa la lectura del bit alterado es advertido de la existencia del error pero no cuenta con un mecanismo capaz de corregirlo. La salida 4 de la Figura 1.5 representa el caso en el que el $soft\ error$ no afecta el resultado generado por el programa; en ese caso, es llamado falso DUE (en esta circunstancia, sería preferible evitar el mecanismo de detección para mejorar la $performance$ del sistema, ya que éste agrega *overhead* cuando el error detectado no afecta realmente la salida del programa).

En cambio, si el $soft\ error$ afecta los resultados del programa en ejecución, el sistema debe informar a la capa superior (por ejemplo, el sistema operativo) de que existe un error efectivo, evitando que el programa continúe bajo esa condición (salida 5 de la Figura 1.5). En ese caso, el error recibe el nombre de verdadero DUE ($True\ DUE$). Normalmente, bajo esta circunstancia, el sistema operativo produce la interrupción de la ejecución por comportamiento anormal (es decir, aborta la aplicación), pero el resto del sistema y las demás aplicaciones continúan funcionando normalmente ($process - kill$). En el caso de que

el bit alterado sea utilizado por el sistema operativo, se provoca una situación, en la que la parte afectada del sistema sólo puede recuperarse por medio de un reinicio del sistema, para lo cual deben interrumpirse todas las aplicaciones en ejecución ($system - kill$).

Sin embargo, el mayor inconveniente se produce cuando un sistema es afectado por un $soft\ error$ que ocurre en un componente que no cuenta con ningún nivel de protección. El bit alterado es leído y potencialmente utilizado por la aplicación. La salida 6 de la Figura 1.5 representa la situación en la que el $soft\ error$ no detectado no afecta el resultado generado por el programa en ejecución.

En cambio, si el sistema utiliza en su operación el bit afectado sin apercibirse de su alteración, el sistema experimenta una corrupción silenciosa de datos (SDC - $Silent\ Data\ Corruption$), también conocida como $Silent\ Errors$. Los errores que producen SDC son los más peligrosos desde el punto de vista de la fiabilidad del sistema, ya que generan escenarios en los cuales las aplicaciones aparentan ejecutarse y finalizar correctamente pero, silenciosamente, producen resultados incorrectos, por lo que el usuario podría nunca darse cuenta. El SDC (salida 7 de la Figura 1.5) es procesado por la aplicación o por el sistema operativo y puede causar efectos impredecibles sobre los resultados de las aplicaciones. Debido a esto, resulta particularmente importante el desarrollo de estrategias de tolerancia a fallos que sean capaces de interceptar los errores que derivan en SDC. Actualmente, la industria especifica la tasa SER de los componentes en términos de las cantidades de SDC y DUE, es decir $SER = DUE + SDC$.

1.7. Posibles efectos de los fallos transitorios

Los $soft\ errors$ (causados por los fallos transitorios) tienen cuatro posibles consecuencias para el software que se ejecuta sobre el sistema, generadas a partir de los DUEs y los SDCs explicados en la Sección 1.6: **(a)**una excepción por instrucción inválida, **(b)** un error de paridad durante un ciclo de lectura, **(c)** una violación en un acceso a memoria [80] y **(d)** un cambio en un valor producido por algún componente o por algún cálculo hecho por el

programa [131]. Estas situaciones se muestran esquemáticamente en la Figura 1.6.

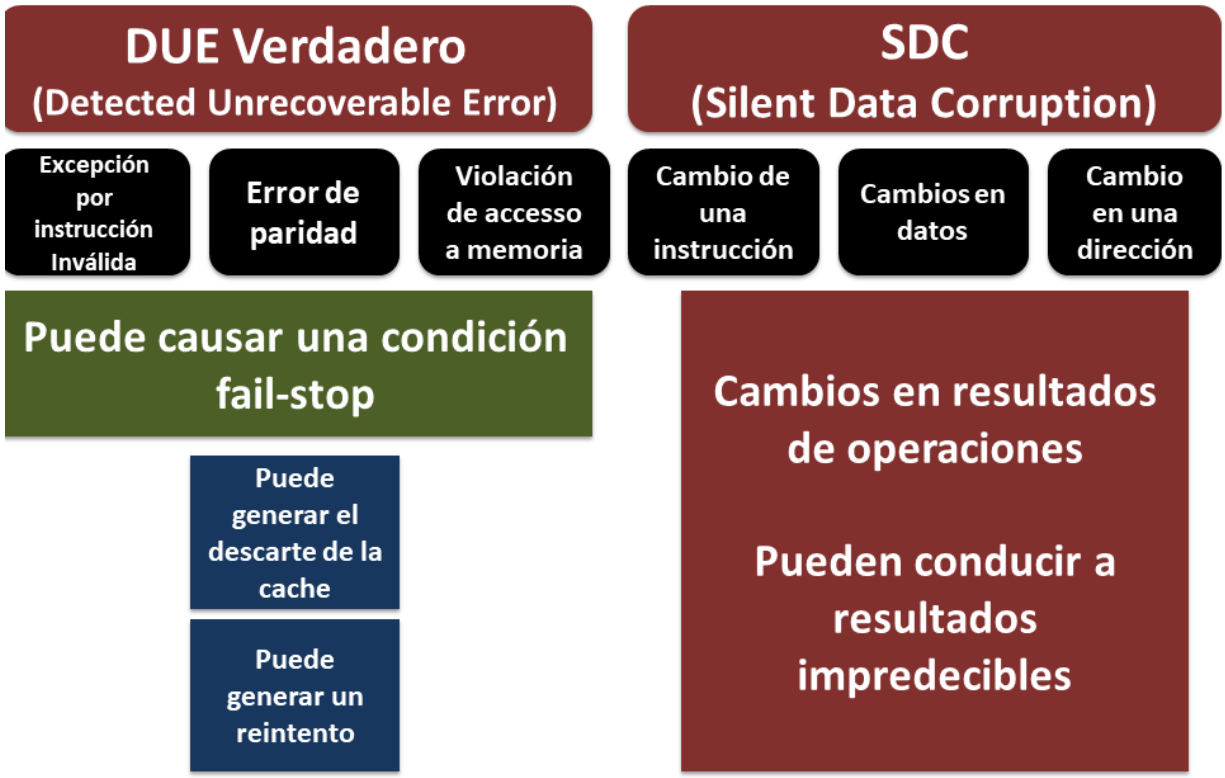

Figura 1.6: *Posibles consecuencias de los soft errors*

1.7.1. Excepción por instrucción inválida

En casos poco frecuentes, un fallo transitorio podría corromper un bit del código de una instrucción del programa, causando una combinación que la arquitectura del procesador no pueda reconocer. De esa forma, se puede generar una excepción por instrucción inválida (DUE) si el procesador es incapaz de decodificar y ejecutar la instrucción resultante.

Otra forma de excepción por instrucción inválida puede ocurrir si el procesador no puede operar con los datos de una determinada instrucción. Por ejemplo, en una división de dos valores numéricos, el denominador no puede ser igual a cero. Esa situación conduce a una excepción por División por Cero, que usualmente lleva a abortar la ejecución. En este ejemplo, el denominador no ha sido originalmente igual a cero, pero un fallo transitorio

puede haber corrompido el dato, cambiándolo por un cero y resultando en la excepción. En otros casos, si la instrucción afectada se transforma en otra instrucción válida, es posible que los parámetros de la instrucción original no se ajusten a los que requiere la instrucción modificada, por lo que ese caso resultará, de todos modos, en una excepción por instrucción inválida (DUE).

Sin embargo, puede darse también la situación en la que la alteración de la instrucción resulte en otra instrucción válida en la cual los parámetros se ajusten, por lo cual el $soft$ *error* no sea detectado por el procesador, que probablemente siga adelante con la ejecución luego de haber procesado la instrucción indeseada. En este caso se produce entonces un SDC. Algunas arquitecturas pueden ser más vulnerables que otras a los cambios en las instrucciones. Por ejemplo, en la arquitectura x86, la diferencia entre un salto condicional en el que se testea si un valor es mayor a otro y un salto condicional en el que se testea si un valor es menor o igual que otro (condiciones inversas) es de un sólo bit.

1.7.2. Error de paridad durante un ciclo de lectura

Un error de paridad generado por un fallo transitorio puede ocurrir en dispositivos de memoria (principal o caché) o en los buses.

Es común que los buses que implementan control de paridad en las transmisiones también tengan implementada la posibilidad de retransmisión de los datos afectados. Esto tiene por efecto un retardo en la transmisión, pero el sistema continúa operando normalmente.

Cuando un $soft$ *error* modifica una posición de memoria, y el componente afectado usa paridad para detectar esa alteración, las consecuencias del error dependen de la ubicación de la posición afectada en la jerarquía de memoria.

Si el error de paridad está en una posición de la memoria principal, es posible recuperarlo si el sistema operativo usa los recursos de memoria virtual del procesador y tiene una copia de la porción afectada de la memoria almacenada en el archivo de *swap*. Si la porción afectada no ha sido modificada previamente por el cómputo normal, el sistema operativo

puede restaurar la página afectada a su estado previo desde el archivo de *swap*.

La recuperación también es posible si la porción afectada de la memoria es un segmento de código de una aplicación y existen binarios de la aplicación en otro dispositivo. En este caso, el sistema operativo puede leer nuevamente esos binarios y restaurar la porción afectada de la memoria.

En cambio, si la memoria ha sido modificada y no hay copia actualizada, el sistema operativo puede abortar la aplicación, o enviar el código del error a la aplicación, permitiendo que ésta intente recuperarse.

En el caso de que el error de paridad ocurra en una memoria caché, la situación es muy similar a la explicada anteriormente: si la porción afectada no ha sido modificada, el controlador de memoria puede buscar una nueva copia en la memoria principal (o el nivel superior de caché) y restaurar el estado conocido anterior; si ha habido modificación, el sistema operativo puede interrumpir la aplicación o permitirle tratar con el error.

1.7.3. Violación en acceso a memoria

Un $soft\ error$ que afecte a un puntero a memoria puede producir que la aplicación intente leer o escribir datos, o producir un salto, a direcciones que se encuentran en un espacio de memoria que se encuentra fuera del alcance de la aplicación ($segmentation\ fault$). Los procesadores modernos cuentan con mecanismos de protección ante esta situación, y envían los errores por accesos ilegales al sistema operativo. Una vez notificado de que una aplicación está intentando acceder a una dirección inexistente, o que pertenece al espacio de otro proceso, el sistema operativo detiene la ejecución, evitando que el error se propague a otras partes del sistema.

1.7.4. Cambio de un valor

Un único bit que se haya alterado debido a un fallo transitorio es suficiente para generar un $soft\ error$ que afecte la operación de un componente, cambiando un resultado esperado por uno inesperado. Este es el caso de los errores que afectan a los componentes internos del

procesador. Los registros, el *pipeline*, la ALU, las unidades de punto flotante y casi todos los elementos de un procesador moderno tienen algún tipo de memoria, para almacenar resultados intermedios de las operaciones, y alguna forma de bus, para comunicarse con los demás componentes del procesador. Todas estas memorias auxiliares y buses internos son blancos posibles de los fallos transitorios.

Un *soft error* en un componente interno del procesador puede pasar desapercibido si el componente no cuenta con ningún mecanismo de protección, y si el resultado no viola el espacio de direcciones de la aplicación ni produce ninguna operación inválida.

Entre estos posibles efectos de los *soft errors*, la *SDC* es el más frecuente. La ejecución no se detiene, y el usuario sólo puede apercibirse de su ocurrencia si los resultados generados por la aplicación difieren significativamente de los usuales.

Debido al *overhead* introducido por cualquier técnica de tolerancia a fallos, los autores de [119] introdujeron una métrica diferente, el trabajo promedio entre averías (*MWTF* - *Mean Work To Failure*), que considera un balance entre el beneficio obtenido sobre la robustez del programa por el agregado de la tolerancia a fallos, y el *overhead* introducido por el funcionamiento del mecanismo que produce ese beneficio.

1.8. Fallos transitorios en sistemas de HPC

El HPC y el uso de supercomputadoras se han convertido en factores clave para el avance de muchas ramas de la ciencia. La gran capacidad de cálculo de estas máquinas, hoy en día del orden de 10^{15} operaciones de punto flotante por segundo (*FLOPs* - *Floating Point Operations per Second*), permiten la resolución de problemas científicos, de ingeniería o analíticos (actualmente, la supercomputadora que lidera el *ranking* Top500, SUMMIT, alcanza los 148.6 *petaFLOPs*). Sin embargo, la demanda computacional de la ciencia continúa creciendo, principalmente debido a dos motivos: la aparición de nuevos problemas en los que el tiempo de resolución es crítico (por ejemplo, el diseño de fármacos personalizados, donde los pacientes no pueden esperar años por la molécula específica que necesitan), y

el crecimiento exponencial del volumen de datos que deben ser procesados (por ejemplo, aquellos originados por grandes telescopios, aceleradores de partículas, redes sociales, redes de sensores en *smart cities*, etc.).

El intento para satisfacer este rápido incremento de las demandas computacionales por parte de la ciencia ha llevado al aumento de la *performance* ofrecida por las supercomputadoras. Se construyen *clusters* cada vez más grandes, en los que una red de comunicaciones interconecta un gran número de nodos, cada uno de ellos con uno o varios procesadores multinúcleo de propósito general y que, en muchos casos, incluyen también procesadores especializados o aceleradores tales como *GPU*s o *Xeon Phis*. Esta tendencia se muestra en la Figura 1.7, que grafica el crecimiento exponencial de la potencia de supercómputo registrada en la lista del TOP500, medida en *FLOP*s hasta el año 2019. La *performance* de estos sistemas se mide utilizando los *benchmarks LINPACK* [45].

En los próximos años está previsto alcanzar la era de la exa-escala, en la que se construyan supercomputadoras formadas por millones de núcleos de procesamiento, y capaces de realizar en el orden de 10^{18} cálculos por segundo (1 *exaFLOP*). Esto constituye una gran ventana de oportunidad para las aplicaciones de HPC, aunque también se incrementa el peligro de que no completen sus ejecuciones. Como se mencionó en la Sección 1.5, en el ámbito de HPC, el riesgo de fallos transitorios aumenta con la cantidad de procesadores que trabajan juntos [105]. Por lo tanto, el peligro de obtener datos y resultados corrompidos silenciosamente es directamente proporcional a la potencia de cómputo que se logra agregando más procesadores al sistema [27].

Estudios recientes muestran que, a medida de que los sistemas de HPC continúan creciendo e incluyendo más componentesde hardware de distintos tipos, el *MTBF* para una aplicación determinada también disminuye, resultando en una tasa de fallos más alta en general. Aunque un nodo de cómputo presente una avería cada 100 años, una máquina con 1.000.000 de esos nodos experimentaría una avería cada 53 minutos en promedio. Los autores de [40] estudiaron la supercomputadora *Blue Waters* de *Cray* durante 261 días, reportando

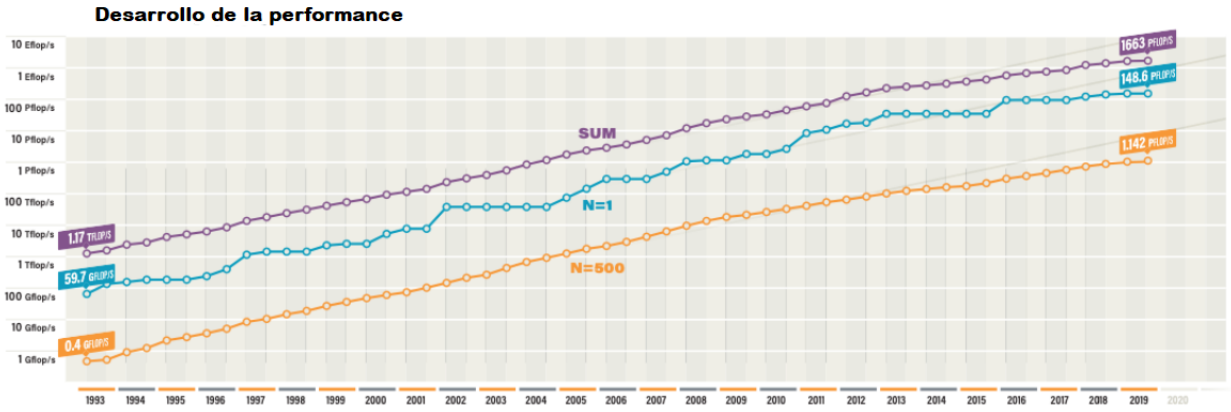

Figura 1.7: *Evolución de la performance a través del tiempo: performance agregada de todos los sistemas, del primero (#1) y del último (#500) del ranking TOP500*

que 1.53 % de las aplicaciones que se ejecutaban fallaban por cuestiones relacionadas con el sistema. Más aún, las aplicaciones que usaban nodos heterogéneos mostraron un porcentaje mayor de averías debidas a errores del sistema, que también aumentaban con la escala. En total, las aplicaciones que fallaron se ejecutaron por un tiempo que representó aproximadamente el 9 % de las horas de producción del nodo. El costo en energía eléctrica de no haber utilizado ningún mecanismo de tolerancia a fallos en esas aplicaciones fue estimado en aproximadamente medio millón de dólares durante el período de tiempo estudiado.

Por lo tanto, parece claro que los sistemas futuros de la exa-escala presentarán tasas de averías más altas, debido a su tamaño y complejidad. En consecuencia, como se mencionó en la Sección 1.1, las aplicaciones paralelas intensivas en cómputo y de larga duración requerirán de la implementación de técnicas de tolerancia a fallos, no sólo para mejorar la robustez garantizando que las ejecuciones finalicen con resultados correctos, sino también para ahorrar energía, dado que si no se utiliza ninguna estrategia, una ejecución fallida debería volver a lanzarse desde el comienzo.

Sin embargo, los modelos más populares de programación paralela, que se usan en aplicaciones de HPC para explotar la potencia de las supercomputadoras, no cuentan con soporte para tolerancia a fallos. Por ello, en las últimas décadas muchos trabajos se han centrado

en el estudio de técnicas y herramientas que permitan dotar de tolerancia a fallos a estas aplicaciones. Una de las técnicas más populares es el *checkpoint/restart* [7, 30, 44, 49]. Esta técnica consiste en salvar periódicamente el estado de la aplicación a almacenamiento estable en archivos de *checkpoint*, que permiten reiniciar la ejecución desde estados intermedios en caso de fallo. Debido a la gran popularidad de los sistemas de memoria distribuida (típicamente *clusters* de nodos, cada uno con su memoria privada, e interconectados por una red de comunicaciones), la mayor parte de la investigación en este campo se ha centrado en aplicaciones de memoria distribuida. En concreto, la mayoría de las propuestas se centran en aplicaciones paralelas de paso de mensajes, ya que es el modelo de programación más usado en sistemas de memoria distribuida, siendo MPI ($Message\ Passing\ Interface$) [43, 136] el estándar $de\ facto$ para este modelo de programación. Además, la inmensa mayoría de estas técnicas se corresponden con estrategias $stop - and - restart$. Es decir, tras un fallo que aborta la ejecución de la aplicación, esta se recupera a un estado salvado previamente a partir del cual puede continuar la computación. Sin embargo, con la popularización de otras arquitecturas hardware, otros paradigmas de programación paralela han cobrado relevancia en los últimos años. Tal es el caso de los modelos de programación híbridos, que combinan paso de mensajes con modelos de programación de memoria compartida, además de modelos de programación heterogéneos, en los que se hace uso de procesadores especializados (por ejemplo $GPUs$ o $Xeon\ Phis$) para cómputo de propósito general. Además, cuando se produce un fallo, frecuentemente está limitado a un subconjunto de los nodos en los que la aplicación está ejecutándose. En este contexto, abortar la aplicación y volver a enviarla a ejecución introduce sobrecargas innecesarias, por lo que se necesitan explorar soluciones más eficientes que permitan la implementación de aplicaciones resilientes [82].

1.8.1. Características de aplicaciones paralelas científicas de paso de mensajes

Los usuarios de entornos de HPC aprovechan la potencia computacional proporcionada por las supercomputadoras a través de modelos de programación paralela. En la actualidad,

los modelos de hardware predominantes en el mundo del HPC son los sistemas de memoria distribuida, los sistemas de memoria compartida y los sistemas heterogéneos que explotan el uso de aceleradores.

Los sistemas de memoria distribuida, en los que nos centraremos en este trabajo, contienen múltiples procesadores, cada uno de los cuales con su propio espacio de memoria, e interconectados por una red de comunicaciones. Como se mencionó en la Sección 1.8, MPI es el estándar *de facto* para la programación de aplicaciones paralelas en las arquitecturas de memoria distribuida. MPI está diseñado para obtener comunicaciones de alta *performance* en aplicaciones científicas paralelas; provee topología virtual esencial, sincronización y funcionalidades de comunicación entre conjuntos de procesos mapeados a nodos. Cada proceso tiene su propio espacio de, direcciones, y los procesos se comunican y sincronizan intercambiando datos por la red [82].

En las Secciones 1.1 y 1.8 se señalaba que la incidencia de los fallos transitorios se vuelve más significativa en el ámbito del HPC, especialmente en el caso de las aplicaciones de cómputo intensivo y de gran duración, debido al alto costo que implica volver a lanzar la ejecución desde el comienzo. La metodología de tolerancia a fallos que constituye el foco principal de este trabajo está diseñada para aplicaciones paralelas científicas que utilizan paso de mensajes. Por lo tanto, en esta sección se revisan las brevemente las características principales de este tipo de aplicaciones.

La computación científica estudia la construcción de modelos matemáticos y técnicas numéricas para resolver problemas científicos, de ciencias sociales y de ingeniería. Se desarrollan aplicaciones informáticas que responden a modelos matemáticos de los sistemas en estudio, de forma de poder ejecutarlas con diferentes conjuntos de datos de entrada. Generalmente, las aplicaciones científicas modelan situaciones del mundo real o cambios en las condiciones de diversos fenómenos naturales. Algunas aplicaciones relevantes son, por ejemplo, en el campo de estudio de fenómenos naturales, como la predicción meteorológica, la simulación para estudio del cambio climático, la reconstrucción de desastres naturales

(terremotos, tsunamis, etc.) o el comportamiento de partículas subatómicas; en el campo de la biología molecular y la bioinformática, como el modelado de estructuras de ADN, el alineamiento de secuencias de ADN, el modelado de mecanismos enzimáticos, el reconocimiento de ácidos nucleicos en proteínas o la comparación de secuencias moleculares; en el campo de la medicina, como el diseño de fármacos, la búsqueda de genomas completos o los vínculos entre anticuerpos y antígenos; en el campo de la astronomía, el movimiento de cuerpos celestes por el espacio; o en el campo de la ingeniería, la dinámica de fluidos, el transporte aéreo y turbulencias, las simulaciones de comportamiento de vehículos, los estudios de ciencia de los materiales y el diseño de semiconductores. El aumento de la potencia computacional ha beneficiado enormemente al cómputo científico, facilitando el manejo de grandes cantidades de datos de forma más rápida y eficiente (muchas veces con restricciones temporales críticas), y consiguiendo resolver problemas que responden a modelos de cada vez mayor complejidad. Teniendo en cuenta todo esto, se puede afirmar que el cómputo científico es el principal beneficiario del HPC [124].

En una aplicación paralela, el conjunto de procesos trabajan simultáneamente, cooperando para realizar una tarea. Cada uno de los procesos realiza localmente una cantidad de cómputo, y se comunica con los demás procesos de la aplicación para intercambiar resultados. En el modelo de programación de pasaje de mensajes, cada proceso tiene su propio espacio de direcciones, por lo que los datos son vistos como asociados a un proceso en particular. La comunicación y sincronización entre los procesos se da a través del envío y la recepción de mensajes, de forma que el acceso a un dato remoto requiere de una comunicación explícita. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización. El modelo de pasaje de mensajes tiene la ventaja de que los programas son altamente portables, pudiendo ejecutarse sobre prácticamente cualquier computadora o sistema paralelo. Otra ventaja consiste en el programador tiene control explícito sobre la ubicación de los datos en la memoria. El acceso y el manejo de la memoria inciden directamente sobre el rendimiento de la aplicación, por lo que este control le permite

al programador la posibilidad de obtener un alto desempeño en ejecución. Sin embargo, esto mismo constituye la mayor desventaja de este modelo, ya que el programador no puede desligarse de los detalles (como la ubicación de los datos en la memoria) y debe ocuparse del orden de las sentencias de comunicación.

1.8.2. Consecuencias de fallos transitorios en sistemas paralelos

Como se explicó en la Sección 1.3, los fallos transitorios pueden afectar a diferentes componentes del hardware de una computadora (en particular, de una computadora paralela), mientras que los *soft errors* son las manifestaciones externas que resultan de ese estado interno inconsistente, observables a nivel del comportamiento o de los resultados de la ejecución (bajo la presunción de que los programas son determinísticos). La cantidad total de *soft errors* que ocurren durante un ejecución puede clasificarse según sus efectos, y, por lo tanto, la tasa total de *soft errors* puede descomponerse de la siguiente manera [96]:

$$SER = LE + DUE + TO + SDC \tag{1.4}$$

La Ecuación 1.4 muestra las diferentes contribuciones a la tasa total de *soft errors* desde el punto de vista de sus efectos sobre la aplicación. Dentro de ellas, un LE (error latente, también llamado fallo benigno) es un fallo que causa la alteración de datos que no volverán a ser utilizados por la aplicación. Si bien el fallo efectivamente ocurre, no se propaga ni tiene impacto en los resultados de la ejecución.

En tanto, de manera similar a la explicada en la Sección 1.6, un DUE es un error detectado sin posibilidad de recuperación. Son las consecuencia de fallos que causan condiciones anormales, detectables a nivel del sistema operativo o de la librería de comunicaciones. Normalmente, causan la finalización abrupta de la aplicación. Ejemplos de esto son los intentos de acceso a direcciones ilegales de memoria (*segmentation faults*), Sección 1.7.3) o ejecución de instrucciones no permitidas, como una división por cero (Sección 1.7.1.

Un error TOE (*Time Out Error*) se produce cuando, debido a la ocurrencia de un fallo,

el programa no logra finalizar dentro de un determinado lapso de tiempo. Un ejemplo de esto es la alteración del valor límite de un bucle, que produce que el programa entre en un lazo infinito.

Finalmente, como se mencionó en la Sección 1.6, se produce una SDC cuando el dato alterado por el fallo no causa ninguna condición anormal, por lo que no es detectado por el software del sistema. El error se propaga silenciosamente, llegando a afectar el resultado final. Desde la perspectiva del hardware, la SDC es el resultado de la inversión de uno o más bits de un registro del procesador que es utilizado por la aplicación.

En un procesador sobre el que se ejecutan procesos de una aplicación paralela de paso de mensajes, existen dos situaciones en las cuales un fallo transitorio resulta en una SDC [92]. Las contribuciones a la tasa total de SDC en aplicaciones paralelas se muestra en la Ecuación 1.5:

$$SDC_{paralelo} = TDC + FSC \tag{1.5}$$

El término TDC ($Transmitted\ Data\ Corruption$) representa la situación en la que el dato afectado por el fallo forma parte del contenido de un mensaje que debe ser enviado. Si el error no es detectado antes del evío, la corrupción se propaga a otros procesos de la aplicación paralela.

En cambio, en el caso de FSC ($Final\ Status\ Corruption$), el fallo afecta un dato que no debe ser enviado a otros procesos, pero que es utilizado por el proceso de manera local, propagando el error a lo largo de su ejecución y corrompiendo su estado final. En este caso, el error se comporta de la misma manera que en el caso de un programa secuencial, afectando sólo el resultado del proceso sobre el cual influye. Esta clasificación posibilita el desarrollo de estrategias de detección diferentes para cada caso.

Uniendo las Ecuaciones 1.4 y 1.5, se obtiene la Ecuación 1.6.

$$SER = LE + DUE + TOE + TDC + FSC \tag{1.6}$$

Como se comentó en la Sección 1.8.1, una aplicación paralela consiste en un conjunto de procesos que cooperan para realizar una tarea. Por lo tanto, su éxito depende fuertemente de la comunicación de los resultados del cómputo local de un proceso a los demás. Debido a esto, todos los fallos que conducen a TDC tienen un alto impacto sobre los resultados finales: este es el principio sobre el cual se sustenta el mecanismo de detección de SEDAR. Por otra parte, los fallos que provocan FSC están relacionados con la fracción centralizada del cómputo, por lo que pueden ser detectados mediante una validación de los resultados finales de la aplicación. Extendiendo este razonamiento, si el número de procesos aumenta, en general, circularán más mensajes, por lo que la tendencia es al crecimiento de la proporción de TDC.

1.9. Objetivos

Dado que las aplicaciones científicas presentan tiempos de ejecución prolongados, usualmente del orden de horas o días, resulta imprescindible encontrar estrategias de tolerancia a fallos que permitan que las aplicaciones se completen de forma satisfactoria, alcanzando una solución correcta, en un tiempo finito y de una forma eficiente a pesar de los fallos subyacentes del sistema. La utilización de estas estrategias tendrá además por efecto evitar que se dispare el consumo de energía, ya que de no utilizar ningún mecanismo de tolerancia a fallos, la ejecución tendría que volver a comenzar desde el principio. A pesar de esto, los modelos de programación paralela más populares que las aplicaciones HPC utilizan para explotar el poder de cómputo proporcionado por supercomputadoras carecen de soporte de tolerancia a fallos [82].

En este contexto de altas tasas de fallos y resultados poco fiables y costosos de verificar, el objetivo de esta tesis es ayudar a los programadores y a los científicos que ejecutan aplicaciones críticas, o al menos relevantes, a proporcionar fiabilidad sus resultados, dentro de un tiempo predeciblemente acotado. Con este objetivo, en los últimos años se ha desarrollado la metodología SEDAR (*Soft Error Detection and Automatic Recovery*) [90, 93], diseñada a

partir de una estrategia de detección llamada *SMCV* [91, 92, 94]), cuyo objetivo es proveer tolerancia a fallos transitorios a sistemas formados por aplicaciones paralelas que utilizan paso de mensajes y se ejecutan en *clusters* de multicores. SEDAR es una solución basada en replicación de procesos [17] y monitorización de los envíos de mensajes y el cómputo local, que aprovecha la redundancia de hardware de los multicores, en la búsqueda de brindar ayuda a programadores y usuarios de aplicaciones científicas a obtener ejecuciones confiables. A diferencia de las estrategias específicas, que proporcionan tolerancia a fallos para determinadas aplicaciones a costa de modificarlas, y que no cubren todos los fallos [14, 24], SEDAR es esencialmente transparente y agnóstico respecto del algoritmo al cual protege.

SEDAR consiste en tres estrategias alternativas y complementarias para alcanzar cobertura completa frente a errores silenciosos: **(1)** sólo detección con notificación al usuario; **(2)** recuperación basada en múltiples *checkpoints* de nivel de sistema; y **(3)** recuperación utilizando un único *checkpoint* seguro de capa de aplicación. Cada una de estas alternativas tiene características particulares y provee un *trade-off* diferente entre costo y $performance$. SEDAR es capaz de detectar y recuperar una ejecución que haya sido afectada por cualquier fallo transitorio que cause una *SDC* o un *TOE*.

El objetivo principal de la tesis es la descripción de la metodología y de sus pautas de diseño, y la validación de su eficacia para detectar los fallos transitorios y recuperar automáticamente las ejecuciones, mediante un modelo analítico de verificación. Si bien esta validación es esencialmente funcional, también se ha planteado como objetivo realizar la implementación práctica de un prototipo funcional (utilizando herramientas existentes y conocidas), y, a partir de las pruebas realizadas sobre ella, caracterizar también el comportamiento temporal, es decir, la *performance* así como también el *overhead* introducido por cada una de las tres variantes. Además de la eficacia de SEDAR, intentamos demostrar que existe la posibilidad de optar (incluso dinámicamente) por la alternativa que resulte más conveniente para adaptarse a los requerimientos de un sistema (por ejemplo, en cuanto a máximo *overhead* permitido o máximo tiempo de finalización), convirtiendo a SEDAR en

una metodología flexible.

Por lo tanto, nuestra premisa es arribar a una metodología completa y funcionalmente válida, que contemple de por sí todos los escenarios posibles de fallos, y a un prototipo de un sistema automático que sea capaz de recuperar sin intervención del usuario, garantizando así la fiabilidad de los resultados dentro de un tiempo factible de ser acotado.

1.10. Contribuciones y limitaciones

La metodología SEDAR es capaz de interceptar todos los fallos que producen *SDC* y *TOE* en una aplicación paralela, proponiendo un esquema de detección basado en el control de los contenidos de los mensajes antes de enviarlos (detección de *TDC*) e incorporando una comparación final de resultados (detección de *FSC*) que asegura la fiabilidad del sistema. Además, permite la configuración de retardos programables de sincronización entre réplicas para evitar que las aplicaciones ingresen en esperas infinitas (detección de *TOE*). En tanto, para proporcionar recuperación, SEDAR propone una estrategia basada en múltiples *checkpoints* de capa de sistema (que toma en cuenta la latencia de detección y la incertidumbre sobre la validez de los *checkpoints*, y otra basada en un único *checkpoint* de capa de aplicación (que permite tener certeza sobre la validez del *checkpoint*). Por lo tanto, hemos planteado una solución por niveles. El objetivo es que las ejecuciones lleguen al final con resultados fiables.

1. Sin SEDAR - sin protección. No hay garantías sobre la finalización. Si la aplicación finaliza, no hay garantías sobre la fiabilidad de los resultados. El usuario podría nunca darse cuenta de los errores.

2. Sólo detección. Si la ejecución finaliza, hay garantía de que los resultados son fiables, pero la limitación consiste en que si ocurre un fallo, no finaliza y se conduce a una parada segura. Introduce el mínimo *overhead*.

3. Recuperación en capa de sistema. La aplicación finaliza correctamente y con resultados

fiables. La diferencia, si ocurre un fallo, estará dada por el tiempo total de ejecución. La limitación consiste en que se deben almacenar múltiples *checkpoints*, y no existe seguridad de que un determinado *checkpoint* no esté corrompido por un fallo que permanece latente al momento del almacenamiento.

4. Recuperación en capa de aplicación (tendencia actual). Si se puede validar el contenido de un determinado *checkpoint*, sólo hace falta uno único, y se puede garantizar que no contiene fallos.

Las principales contribuciones de este trabajo son:

- El desarrollo completo de una metodología de tolerancia a fallos, que integra la duplicación (efectiva para detectar los fallos transitorios), con el *checkpoint/restart* que se utiliza normalmente para garantizar disponibilidad frente a los fallos permanentes, obteniendo, a partir de esta combinación, una estrategia que asegura tanto la finalización como la fiabilidad de los resultados.

- La descripción y verificación del comportamiento funcional en presencia de fallos, demostrando la eficacia de la estrategia de detección y la validez del mecanismo de recuperación basado en múltiples *checkpoints* coordinados de nivel de sistema.

- La comprobación empírica de las predicciones del modelo, por medio de inyección controlada de fallos, que evidencia la fiabilidad provista por las estrategias de SEDAR.

- La herramienta SEDAR, que implementa el mecanismo de detección y el algoritmo de recuperación basado en múltiples *checkpoints*, y el trabajo experimental realizado para incorporar SEDAR a aplicaciones paralelas.

- La caracterización temporal y la evaluación de los *overheads* introducidos para cada una de las tres estrategias alternativas, dimensionando la influencia del patrón de comunicación de la aplicación. De esta forma se muestran los beneficios que ofrecen las

distintas variantes de SEDAR, tanto en el tiempo de ejecución como en confiabilidad de los resultados.

- La evidencia de la flexibilidad de SEDAR para adaptarse de forma de alcanzar un determinado compromiso entre costo y desempeño obtenido.

- La determinación de la relación entre la cantidad de recursos requeridos para implementar la estrategia, y la ganancia de tiempos obtenida en conjunto con la fiabilidad de la ejecución, mostrando la viabilidad y la eficacia de SEDAR para tolerar los fallos transitorios en sistemas de HPC.

Entre las principales limitaciones de SEDAR, su aplicabilidad está restringida a los sistemas de memoria distribuida, o, mejor dicho, a los sistemas que ejecutan aplicaciones de paso de mensajes, a causa de que la validación de los mensajes es el fundamento del mecanismo de detección; no se aplica a entornos de memoria compartida, ni heterogéneos donde las aplicaciones no se comuniquen mediante mensajes.

Otra limitación considerable es que, actualmente, SEDAR funciona únicamente para aplicaciones paralelas determinísticas. Este es un requerimiento motivado porque sólo en esas aplicaciones, se puede garantizar que el cómputo, sobre los mismos datos de entrada, por parte de dos réplicas, arroja los mismos resultados, que son los que deben compararse para detectar los fallos.

En el estado actual, no existe aún una implementación de una estrategia de adaptación automática del mecanismo de recuperación, es decir, que la protección de la ejecución en curso mediante *checkpoints* comience dinámicamente, basándose en la determinación del momento a partir del cual vale la pena comenzar a proteger dicha ejecución. Por otra parte, el modelo de caracterización temporal actual no permite predecir la respuesta cuando suceden dos o más errores distintos. El mecanismo de recuperación basada en múltiples *checkpoints* no soporta de manera óptima la ocurrencia de varios fallos, de momento, por lo que requiere ser refinado.

Finalmente, en su estado actual, la metodología y la implementación de SEDAR soportan únicamente los *checkpoints* coordinados de nivel de sistema ($DMTCP$) o no-coordinados (por proceso, en capa de aplicación); en este sentido, requiere ser extendida de forma de poder soportar diferentes tecnologías de *checkpointing*, como los *checkpoints* semicoordinados o el *checkpointing* diferencial o incremental, no tenidas en cuenta hasta el momento.

Capítulo 2

Trabajo relacionado

Resumen

En este capítulo se revisa el trabajo relacionado con esta tesis: las estrategias basadas en replicación para detección y recuperación de *soft errors*; las estrategias que incorporan hardware adicional y las puramente basadas en software; las técnicas de *checkpoint/restart* para fallos permanentes; las propuestas para algoritmos específicos y los intentos para obtener aplicaciones resilientes de paso de mensajes.

2.1. Objetivos de la detección

La estrategia general del cualquier programa que incorpora detección de fallos transitorios se basa en la replicación, que consiste en mantener dos hilos de cómputo redundantes e independientes. Esto se fundamenta en el hecho de que dos instancias de ejecución redundante deben escribir valores idénticos en ubicaciones de almacenamiento idénticas y deben emitir órdenes para transferir el control a direcciones idénticas. La ejecución de uno de estos hilos (*leading thread*) va levemente por delante de la del otro (*trailing thread*), pero existe una gran flexibilidad respecto cómo pueden intercalarse las instrucciones para cada paso del cómputo. Previo a la escritura de un valor en un dispositivo mapeado en memoria, los resultados de cada cómputo se comparan para ver si hay equivalencia. Si los resultados parciales son diferentes, el sistema debe ser notificado de la ocurrencia de un fallo. También deben

verificarse los argumentos de todas las transferencias de flujo de control. Esta metodología se ha mostrado en la literatura como una forma efectiva de implementar la detección [33]. Para un paso de la ejecución de un programa, se define que, si existen dos cómputos, uno con un fallo y otro sin él, entonces el cómputo con fallo debe dar un paso indistinguible del que no tiene fallo, o el cómputo defectuoso debe alcanzar el estado de fallo [112]. Todas las soluciones que existen para detectar fallos involucran el agregado de algún tipo de redundancia, pero los detalles varían significativamente.

Tradicionalmente, las propuestas se han dividido en las que atacan el problema de la detección desde la perspectiva del hardware y las que lo hacen desde la óptica del software. Existen propuestas de solución basadas únicamente en agregados al hardware específico, como los *ECC*s (redundancia de información, agrega bits extra para la detección o corrección de errores) o los co-procesadores *watchdog* [84] e hilos redundantes de hardware; y técnicas de software puro, diseñadas tanto para monoprocesadores como para multicores [33, 117]. En términos generales, las soluciones por hardware son más eficientes para la aplicación de una política de confiabilidad, pero las soluciones de software puro son más flexibles (se pueden aplicar y configurar de acuerdo a las necesidades del entorno concreto) y menos costosas en términos de hardware [112, 119]. Un aspecto importante, vinculado a la detección, es el intervalo de comprobación. Si los resultados parciales son validados frecuentemente, se introduce una alta sobrecarga, que redunda en una gran cantidad de tiempo adicional de ejecución a causa del mecanismo de detección. Como ventaja, se acota la latencia de detección del fallo. Una vez detectado, el mecanismo de recuperación podrá partir de un estado consistente reciente (ya que la mayor parte del cómputo realizado antes de la detección estará validado). En el otro extremo, si sólo se comparan los resultados finales de las aplicaciones, el *overhead* introducido por la detección es mínimo, pero la latencia de detección y el costo de la recuperación son altos (ya que hay que volver al comienzo de la ejecución). Por lo tanto, debe balancearse la precisión (es decir, minimizar la detección de falsos positivos), la velocidad (minimizar la latencia entre la ocurrencia del fallo y el instante de su detección)

y su costo asociado (la introducción de un bajo *overhead* en ausencia de fallos), mediante un compromiso entre el intervalo de comparación y la sobrecarga introducida [91].

Una vez que se ha detectado un fallo transitorio, se pueden utilizar distintas estrategias de recuperación. Gracias a la facilidad de desacoplar la detección y la recuperación en esta forma, usualmente es posible estudiarlas por separado. Inclusive, debido a la relativamente baja frecuencia de los fallos, el mecanismo de recuperación se ejecuta mucho menos que el de detección, que debe estar activo continuamente [119]. Más aún, el beneficio de ejecutar un mecanismo de recuperación depende del momento de ocurrencia del fallo respecto del ciclo de vida de la aplicación. Una aplicación puede decidir no realizar la recuperación si llevarla a cabo implica volver tan atrás que el costo es el mismo que el de relanzar desde el principio, por lo que no se obtendría ningún beneficio apreciable [10]. Debido a esto, una estrategia de detección diseñada de forma compatible con la mayoría de los mecanismos de reporte y recuperación puede ser extendida con relativa sencillez para proveer tolerancia a fallos completa. En el resto de este capítulo se revisan las principales propuestas que se han realizado en la literatura en el campo de la detección de fallos transitorios.

2.2. Propuestas basadas en redundancia

Para lidiar con problema de la fiabilidad en las arquitecturas modernas, los investigadores han intentado proveer soluciones eficientes para detectar o corregir *soft errors* explotando la redundancia inherente de los sistemas modernos y utilizando recursos redundantes como los cores o los *threads* extra. En las técnicas tradicionales de redundancia modular doble (*DMR - Dual Modular Redundancy*) o triple (*TMR - Triple Modular Redundancy*), que son transparentes a los programadores, se utilizan componentes de hardware especializados para replicar las instrucciones, y la detección o recuperación se logran comparando las salidas de réplicas. Varias aproximaciones se mencionan en [119], como el procesador *watchdog* para comparar los valores frente a los del procesador principal. Esta redundancia espacial a nivel de hardware implica costo adicional en componentes y *overheads* en consumo energético.

Son embargo, son las más utilizadas en ámbitos críticos reales, como los sistemas de vuelo o los servidores de alta disponibilidad. Por ejemplo, una aeronave del tipo del Boeing 777 ha utilizado tres procesadores y buses de datos redundantes diferentes, con un esquema de votación de mayoría para lograr tanto detección como recuperación en tiempo real [81].

Las soluciones basadas en replicación de software son más atractivas, ya que, al no requerir hardware especializado, son mucho menos costosas. Por supuesto, también tienen ciertas desventajas: normalmente incurren en un costo significativo en cuanto al uso de los recursos (cores, memoria), y un *overhead* temporal no despreciable, además de no ser transparentes para el programador; normalmente, requieren del agregado de pequeñas estructuras auxiliares de hardware específicas para el funcionamiento de la ejecución replicada. Por este motivo, son soluciones basadas en software, pero no de software puro (las cuales se tratan aparte en la Sección 2.3. Las técnicas de redundancia de software se pueden clasificar en dos niveles: nivel de procesos o nivel de *threads*. La replicación de procesos crea un clon completo de los procesos de la aplicación (duplicando el $footprint$ de memoria pero manteniendo y protegiendo los datos de cada proceso en espacios separados). Los valores se comparten entre los procesos para comparar sus resultados y detectar cualquier disidencia que pueda indicar *SDC* [133]. En tanto, a nivel de *threads*, los datos no se replican completamente, ya que pueden compartirse entre hilos "hermanos"; además, la sincronización y la comunicación entre hilos es más sencilla que entre procesos. A su vez, los mecanismos de replicación a nivel de *thread* pueden clasificarse en redundancia a nivel de instrucciones (*ILR - Instruction Level Redundancy*) y multi-hilos redundantes (*RMT - Redundant Multi-Threading*) [111].

2.2.1. Redundancia a nivel de instrucciones

En esta categoría, toda la duplicación y la detección de errores es realizada por el hilo principal. Las instrucciones se replican, creando un flujo de datos separado del original, y se agregan verificaciones de integridad para detectar errores. El segundo flujo trabaja sobre registros diferentes, permitiendo comparaciones seguras entre los valores de ambos.

Debido a que no hay dependencia entre ambos flujos, es potencialmente posible ejecutarlos en paralelo, aprovechando el paralelimo a nivel de instrucción existente en los procesadores modernos [111]. En la Figura 2.1 se muestra una transformación de código típica de *ILR*, donde se realiza una simple suma dos veces, y se agregan operaciones extra operations para las verificaciones de integridad.

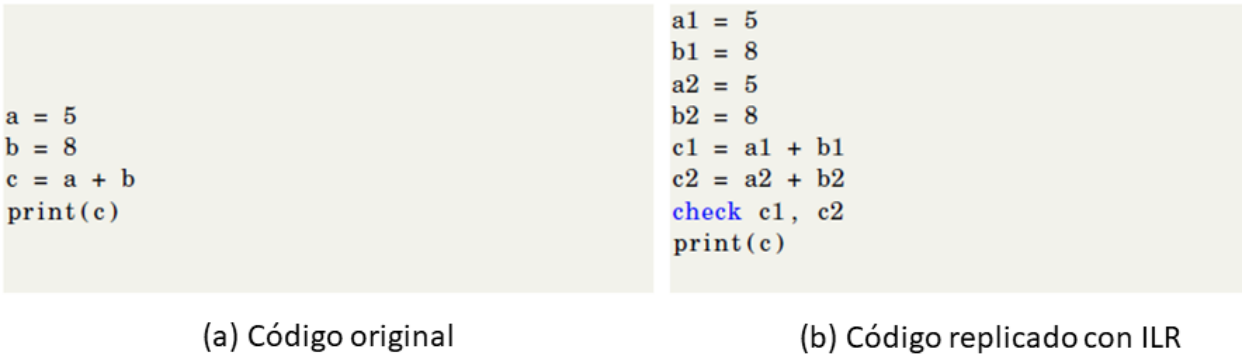

Figura 2.1: *Transformación de código con ILR - extraído de [111]*

Una técnica conocida de *ILR* es *SWIFT* (*Software Implemented Fault Tolerance*) [119]. *SWIFT* utiliza el compilador para realizar una transformación, durante la generación del código, en la que duplica todas las instrucciones del programa dentro de un solo hilo e inserta comprobaciones de validación apropiadas en puntos estratégicos, verificando además explícitamente el flujo de control durante la ejecución. De esta forma provee un alto grado de cobertura frente a fallos que producen errores tanto en los datos como en el flujo de control, asegurando la corrección de aplicaciones de un único hilo. En ejecución, los valores se computan dos veces y se comparan por equivalencia antes de que alguna diferencia debida a un fallo transitorio pueda afectar la salida del programa.

El código resultante requiere del doble de registros que el original, lo que potencialmente puede causar que se utilice la RAM cuando no haya más espacio disponible en el procesador. Se aprovecha la existencia de *ECC*s en el subsistema de memoria para no duplicar su utilización. Una característica fundamental es que es una estrategia de software puro (no requiere ningún hardware adicional).

Una posible solución para evitar la duplicación de cargas de datos desde la memoria consiste en que el compilador realice una única lectura y luego duplique el valor leído para ser utilizado por las versiones original y redundante del programa. Esta alternativa es de implementación sencilla, pero tiene el inconveniente de quitar la redundancia a las instrucciones de carga, transformándolas en puntos de falla centralizados.

Los resultados obtenidos muestran que la eficiencia del procesador para planificar la ejecución de instrucciones, asignar recursos (que no son utilizados en la ejecución de la versión original) y explotar el paralelismo a nivel de instrucción para gestionar la detección, produce un overhead en el tiempo de ejecución de sólo un 41 % en promedio respecto de la versión sin detección de fallos, mucho menor de lo esperable (más del doble, debido a la duplicación de instrucciones equivalente a ejecutar el programa dos veces, y el código adicional para comparar y validar las salidas). De esta manera, se puede integrar una técnica de fiabilidad por software al compilador para proveer detección utilizando el hardware comercialmente disponible.

SWIFT es una propuesta basada únicamente en software; provee únicamente detección de fallos, pudiéndose integrar con una técnica común de *checkpointing* para protección completa. Como desventaja, todas las técnicas basadas en el compilador impactan negativamente en la $performance$. El tamaño del código crece significativamente, y las verificaciones se agregan al camino crítico del programa, introduciendo *overhead*. Además, requieren recompilación de las aplicaciones; por otra parte, es indispensable contar con el código fuente, lo que no se cumple en todos los casos de aplicaciones de interés [133].

En general, las soluciones de detección por software basadas en el compilador son más generales, ya que se pueden aplicar a cualquier código, a costa de un incremento considerable en la complejidad.

2.2.2. Redundancia a nivel de *threads*

Los mecanismos de *multithreading* redundante (*RMT - Redundant Multithreading*) emplean redundancia temporal a nivel de *threads*, y son atractivos por su bajo costo comparados con la redundancia de hardware (espacial). En la mayoría de los casos, requieren algunos componentes adicionales de hardware en el procesador para su implementación ($buffers$ utilizados como colas). RMT proporciona detección de errores mediante la ejecución de copias idénticas del programa como *threads* separados en unidades de ejecución paralelas, operando sobre entradas idénticas y comparando las salidas. Los *threads* redundantes pueden ejecutarse en el mismo procesador multi-hilado (*SMT - Simultaneous Multithreading)*, en diferentes cores de un *CMP* (*Chip Multi-Processor*) o en unidades de ejecución paralela en $GPGPUs$ (*General Purpose Graphics Processing Unit*. Sus principales limitaciones están relacionadas con el uso de memoria adicional o la degradación de *performance* causada por la re-ejecución, más que costos de hardware. Por lo tanto, las soluciones basadas en RMT brindan un entorno de ejecución flexible, pero intentando proveer una cobertura frente a fallos similar a la redundancia espacial con costos de hardware mínimos [106].

Con la llegada de los procesadores multi-hilo y multicore, y sus capacidades de ejecutar múltiples *threads* independientes simultáneamente, RMT comenzó a ser una técnica de tolerancia a fallos ampliamente utilizada. RMT permite que las ejecuciones de dos copias del programa como hilos separados en múltiples unidades de ejecución (procesadores SMT [117], $CMPs$ [97] o $GPUs$ [73]) puedan comparar sus resultados para detectar fallos. Los dos hilos redundantes son denominados "hilo productor" (o *leading thread*) e "hilo consumidor" (o *trailing thread*). RMT replica el código de la aplicación original en ambos hilos (en lugar de duplicar su tamaño como en ILR). El *leading thread* produce un valor que el *trailing thread* consume, verificándolo con el que él mismo ha calculado para detectar *soft errors*; de esta forma, RMT quita las validaciones del camino crítico de la aplicación (otro problema de ILR). En la Figura 2.2 se muestra cómo el código original se replica en hilos RMT; las operaciones en azul son las agregadas para detectar los fallos.

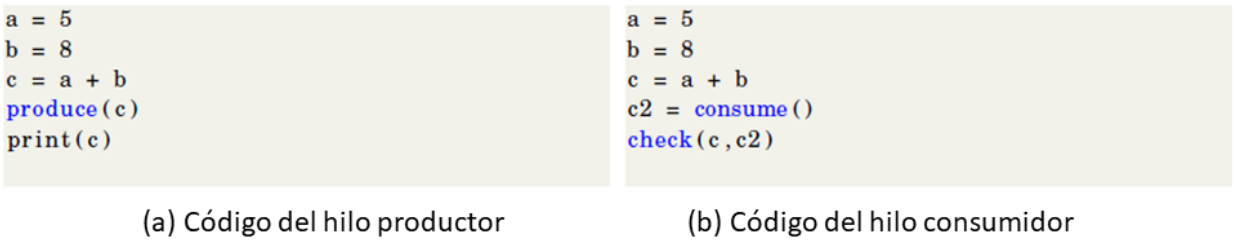

Figura 2.2: *Transformación de código con RMT - extraído de [111]*

La clave para *RMT* son los componentes que se incluyen en la ejecución redundante, las entradas que deben ser replicadas y las salidas que deben compararse luego de la ejecución redundante. Dentro de estos conceptos, el procesador *SRT* (*Simultaneous and Redundantly Threaded*) [117] introduce la esfera de replicación (*SoR - Sphere of Replication*) como el límite lógico de la ejecución redundante. Los componentes dentro de la *SoR* se incluyen en la ejecución redundante, mientras que los que están afuera de la *SoR* deben ser protegidos por medio de otros mecanismos. Los valores que ingresan a la *SoR* (entradas) deben replicarse, y los que salen de la *SoR* deben ser validados (esta idea se retoma en la Sección 3.6). Este es el motivo detrás de la práctica usual de no replicar las llamadas a función de librería y las instrucciones *load* y *store* [89, 152]. El código de las funciones de librería no está disponible, y el resultado de dos llamadas al mismo procedimiento podría ser diferente (como en el caso de una función que genera números aleatorios), por lo que el valor de retorno se comparte desde el *leading thread* al *trailing thread*. Las lecturas y escrituras a memoria tampoco se replican, aunque las direcciones y valores son verificados por el *trailing thread* para asegurar que esas operaciones están libres de errores. Esto se debe a que las técnicas de detección de *soft errors* se enfocan en los fallos que ocurren dentro del procesador, y no en la memoria: la RAM cuenta con los *ECC*s y otros mecanismos de protección (es decir, está afuera de la *SoR*, pero los registros son vulnerables aún [63]. Una vez que un valor se ha leído desde la memoria, el *leading thread* lo comparte con el *trailing thread*. En tanto, sólo una de las réplicas realiza las escrituras en memoria, luego de la validación.

Las soluciones basadas en *RMT* han dado origen a una gran cantidad de trabajo de

investigación. Todas ellas requieren de la incorporación al procesador de alguna estructura de hardware (es decir, no son técnicas puramente de software), ya que intentan proteger al procesador de los fallos (es decir, la *SoR* está alrededor del procesador).

Los procesadores multi-hilo (*SMT*) permiten lanzamiento de múltiples instrucciones desde múltiples *threads* independientes explotando un procesador superescalar con hardware de *multithreading*. El paralelismo a nivel de instrucción mejora el rendimiento y presenta oportunidades para la tolerancia a fallos por redundancia temporal. El cuello de botella siempre se encuentra en la comunicación entre los dos hilos, habiéndose explorado diferentes variantes [111]. Entre las principales técnicas que se aplican a *SMT*s (y que influyen más sobre el resto de la literatura), se encuentran: *AR-SMT (Active-stream/Redundant-stream Simultaneous Multithreading)* [123], *SRT (Simultaneous and Redundant Threaded)* [117] y *SRTR (SRT with Recovery)* [141]. Entre ellas, existen diferencias leves respecto de la *SoR*; todas incorporan áreas de *buffer* adicionales para almacenar los resultados redundantes.

Mientras que las soluciones mencionadas para *SMT*s intentan cubrir todos los fallos, existen también una serie de propuestas basadas en replicación parcial, que realizan diferentes opciones de diseño para disminuir el *overhead* y mejorar la *performance* a costa de tolerar sólo algunos errores. Una propuesta pionera en este sentido es *Slipstream Processor* [139], donde se propone la obtención de una versión reducida de la aplicación, quitando cómputo ineficaz y flujo de control predecible. La aplicación original y su versión reducida se ejecutan en *threads* separados, proveyendo redundancia parcial y adelantando resultados parciales que aceleran la ejecución. Otras propuestas relevantes son *Opportunistic transient-fault detection* [60] y *Slice-based Locality Exploitation* [108]. Relacionado con esto, existen las técnicas de replicación selectiva: los autores exploran la posibilidad de disminuir el *overhead* protegiendo sólo las partes de la aplicación que resultarán corrompidas si ocurre un *soft error*, dado que no todos los *bit flips* conducen necesariamente a una avería (es decir, la determinación de los bits *ACE* y el cómputo del *AVF*). *IPAS* [79] intenta proteger sólo el código que necesita cobertura. Utiliza un proceso de *machine learning* para identificar

las instrucciones que requieren duplicación. El argumento es que se desperdician muchos recursos en proteger la aplicación completa, debido a la baja probabilidad de que un *soft error* modifique datos realmente importantes. Incluso en esos casos, las propiedades de los algoritmos en aplicaciones de HPC producen un enmascaramiento del error, en el sentido de que el resultado puede ser aceptable incluso en su presencia.

Con el surgimiento de las arquitecturas multicore, el empleo de *RMT* en conjunto con *CMP*s se volvió muy popular. En estas técnicas, los hilos redundantes se ejecutan en diferentes cores del procesador para balancear la carga y mejorar la utilización de recursos. Por otra parte, el costo de comunicación entre los *threads* redundantes es el principal cuello de botella de estas soluciones. Los estudios más destacados son *Chip-level Redundant Multithreading* [97] y *Chip-level Redundantly Threaded Multiprocessor with Recovery* [59], que proveen detección y recuperación de fallos transitorios respectivamente. Este último utiliza el mecanismo mejorado de detección de *CRT* para multicores e incorpora la recuperación propuesta en *SRTR*. Otras contribuciones significativas en la literatura se encuentran en *Reunion* [135] (*Complexity-Effective Multicore Redundancy*) y *DCC* (*Dynamic Core Coupling*) [76]. Todas estas propuestas difieren en la *SoR*, en la estrategia que adoptan para reducir la comunicación entre procesadores y en cómo manejan la sincronización de los recursos compartidos. Las ventajas principales de la utilización de técnicas que utilizan *RMT* en entornos de *CMP*s son el balance de carga entre los procesadores, para alcanzar una utilización eficiente de los recursos, y proporcionar también cobertura frente a fallos permanentes, debido a que el *leading thread* y el *trailing threads* de una aplicación no comparten un único procesador. Sin embargo, como hemos mencionado, la desventaja principal radica en la comunicación intensiva entre procesadores, con alta demanda del ancho de banda.

En [58] se proponen mejoras a la técnica *DDMR* (*Dynamic Double Modular Redundancy*), permitiendo configurar el sistema para operar en modo redundante o utilizando los cores por separado para procesamiento. En la misma línea, en [146] se propone *MMM* (*Mixed Mode Multicore*), un modo de proporcionar soporte para que las aplicaciones que

requieren fiabilidad (incluyendo el software del sistema) se ejecuten en modo redundante y, simultáneamente, las aplicaciones que necesitan alto rendimiento puedan evitar la penalización. Se abordan las dificultades propias de los cambios de modo mediante técnicas de virtualización, y se permiten opciones de configuración para proveer flexibilidad.

Más recientemente, se han estudiado técnicas basadas en *RMT* para utilizar la cantidad masiva de *threads* paralelos en *GPGPU*s ([41, 73, 142]).

Si bien estas propuestas están diseñadas para funcionar en sistemas que utilizan procesadores de altas prestaciones, no son suficientes en ámbitos críticos. El campo de la replicación, específicamente del *multitheading* para tolerancia a fallos transitorios es amplio, y difícil de cubrir en forma exhaustiva. Un excelente y completo estudio clasificatorio y comparativo puede encontrarse en [106].

2.3. Propuestas basadas puramente en software

Las técnicas que requieren agregados de hardware son un tanto ineficientes en computadoras de propósito general. El diseño y verificación de elementos de hardware agregados *ad-hoc* (y equipados con mecanismos de detección de errores que verifican la operación correcta en tiempo de ejecución) son procedimientos costosos. Además, las condiciones ambientales en las que se desempeñan las computadoras, así como el envejecimiento de los componentes, son causas de fallo que no pueden ser predichas durante la etapa de desarrollo [132]. Por otro lado, la evolución arquitectural hacia los multicores ha producido un gran interés en la adaptación de los recursos paralelos que proporcionan para utilizarlos, tal y como se presentan, con el objeto de lograr fiabilidad frente a los fallos transitorios [92]. Mientras que la lógica combinatoria interna al procesador se puede proteger por medio de duplicación, en la práctica esta técnica es imposible de aplicar a elementos como *latches* o unidades aritméticas [112]; como consecuencia, la tarea de proteger la lógica interna resulta compleja, siendo los registros la preocupación principal por su propensión a los fallos [63]. En todos los procesadores modernos (y no sólo los que se usan en aplicaciones de alto

desempeño o disponibilidad) se contempla la vulnerabilidad a los fallos transitorios como un problema a tratar en un diseño agresivo [119]. Por otra parte, en muchas aplicaciones (como el video bajo demanda), las consecuencias de los fallos no son tan severas, por lo que no es crítico incorporar mecanismos exhaustivos de tolerancia a fallos. Bajo estas circunstancias, en las que los costos de desarrollo son elevados, y aún en aplicaciones de seguridad crítica, los diseñadores tienden a adoptar el hardware disponible en el mercado. En este contexto, se han propuesto soluciones basadas en software puro [112, 119], que resultan atractivas ya que permiten implementar sistemas fiables frente a los fallos transitorios sin requerir ningún hardware adicional, siendo esta su principal ventaja que las hace las apropiadas para los sistemas de cómputo de propósito general [81].

Las técnicas basadas en software puro representan un compromiso entre el nivel de cobertura frente a fallos que pueden lograr, el bajo costo de diseño (en términos de hardware) y la flexibilidad en el despliegue, ya que pueden ser configuradas de diferentes formas para adaptarse a las necesidades específicas de las aplicaciones). A pesar de no ser capaces de lograr el mismo desempeño de las técnicas de hardware (ya que deben ejecutar instrucciones adicionales y no pueden examinar el estado de la microarquitectura), se muestran como alternativas aceptables, ya que logran niveles significativos de fiabilidad introduciendo un *overhead* razonable a la ejecución [33]. La idea básica que hay detrás de las propuestas basadas puramente en software para la detección de fallos, es la duplicación del cómputo de la aplicación. Las dos réplicas deben operar sobre copias de los mismos datos de entrada y deben comparar sus salidas periódicamente; si los resultados difieren, el sistema ha experimentado un fallo transitorio [119]. Debido a que los errores que afectan a registros (o posiciones de memoria) impactan sobre los datos y valores de variables utilizados por la aplicación, algunas aproximaciones existentes almacenan copias redundantes de la misma información e introducen verificaciones de consistencia entre las réplicas [81].

Otra clase de técnicas de detección de fallos transitorios son las que utilizan replicación de procesos. En ellas, se crea un conjunto redundante de procesos por cada proceso de la

aplicación original y se comparan sus salidas para garantizar una correcta ejecución. Entre estas técnicas, se destaca la propuesta PLR (*Process-Level Redundancy*) [132]. En ella, se permite al sistema operativo planificar libremente los procesos redundantes, por lo que se aprovecha el paralelismo de recursos de hardware disponibles. En ese sentido, PLR escala con la tendencia arquitectural hacia las grandes máquinas multicore y manycore de propósito general, logrando mejorar la cobertura y el rendimiento sin agregar hardware ni modificar el sistema. Frente al problema de la utilización ineficiente de un gran número de cores, por parte de aplicaciones que no pueden explotar todos ellos para obtener mayores prestaciones (ver Sección 3.2.4, PLR provee una alternativa de aprovechamiento en beneficio del sistema. Además, se presenta un prototipo real que se ejecuta en conjunto con programas serie de forma transparente a la aplicación, sin realizarle ningún tipo de modificación, incorporando la detección de fallos a costa de introducir un 16.9 % de *overhead* en el tiempo de ejecución.

Uno de los aportes significativos de esta propuesta es que se centra en garantizar la ejecución correcta de las aplicaciones a nivel del software, y no en garantizar la ejecución sin errores que afecten al hardware. En concordancia con [119], se muestra que una alta proporción de los fallos que ocurren (y que serían detectados por técnicas basadas en hardware) resultan en errores latentes, por lo que pueden ser ignorados sin riesgo alguno. Por lo tanto, la detección está basada en los efectos de los fallos durante la ejecución. Por el contrario, [119] detecta fallos que producen salidas correctas, de manera inútil. Sin embargo, en algunas circunstancias particulares, PLR detecta algunos falsos positivos. En particular, el trabajo muestra la manera en que los fallos que afectan a los registros de CPU se propagan desde la perspectiva de la aplicación. Mientras que muchos de ellos resultan en errores latentes, otros se propagan a través de cientos o miles de instrucciones. Como se mencionó anteriormente, una de las características salientes de PLR es la transparencia al usuario y a la aplicación (no para el sistema, que es consciente de la existencia de las réplicas). Los procesos redundantes interactúan con el sistema como si sólo se ejecutara el proceso original, por lo que la incorporación de PLR pasa desapercibida para la aplicación, la cual no tiene que ser

modificada ni recompilada. PLR se ejecuta en espacio de usuario, lo que lo hace flexible, de modo que las aplicaciones con requerimientos de fiabilidad pueden utilizarlo y otras no, de manera similar a lo propuesto en [146]. Los procesos redundantes emulan las *system calls* y las operaciones de E/S (sólo el proceso original las realiza efectivamente); previamente, se verifica que todas las réplicas deben realizar la misma *system call*, de forma de detectar posibles errores en el flujo de control. Todas las entradas del proceso original se replican y comunican a los procesos redundantes por memoria compartida; de la misma forma, la comparación de valores de salida se realiza colocándolos previamente en un segmento de memoria compartida. Como limitación, PLR está diseñado únicamente para aplicaciones serie, no soportando las características de las de HPC.

Los autores realizan también un análisis sobre la latencia de detección, realizando mediciones de la cantidad de instrucciones que se ejecutan entre la inyección del fallo y su detección. La tendencia muestra que los fallos que se detectan, que hubieran producido SDC tienen latencias elevadas, en tanto que los que se detectan con un manejador de señales (que hubiera conducido a DUEs) tienen mayor probabilidad de ser detectados rápidamente. De todas formas, la perspectiva de detección basada en la ejecución correcta a nivel de software produce que los fallos permanezcan en estado latente un lapso de tiempo no acotado.

Algunas propuestas similares, pero específicas para aplicaciones de HPC, se comentan en la próxima sección.

Existe una categoría de soluciones basadas en RMT, pero implementadas únicamente en software. Estas estrategias lidian con los fallos transitorios a nivel del compilador, del sistema operativo o de la aplicación, considerando el efecto de los errores en el software que se ejecuta.

Las técnicas de *redundant multithreading* a nivel de compilador (también conocidas como software RMT en la literatura) transforman el código de la aplicación en una ejecución redundante con dos *threads* que se comunican. El *leading thread* ejecuta todas las instrucciones del código original y código de comunicación adicional, mientras que el *trailing thread*

replica el cómputo y compara los resultados con los que reciben del *leading thread*. Bajo la presunción de que la memoria está protegida por *ECC*s y la *SoR* sólo incluye el cómputo, las operaciones de *load/store* en memoria sólo son realizadas por el *leading thread*, mientras que las operaciones de ALU son replicadas en el *trailing thread*. En la Figura 2.3 se muestra un ejemplo de la transformación realizada para un fragmento de código sencillo.

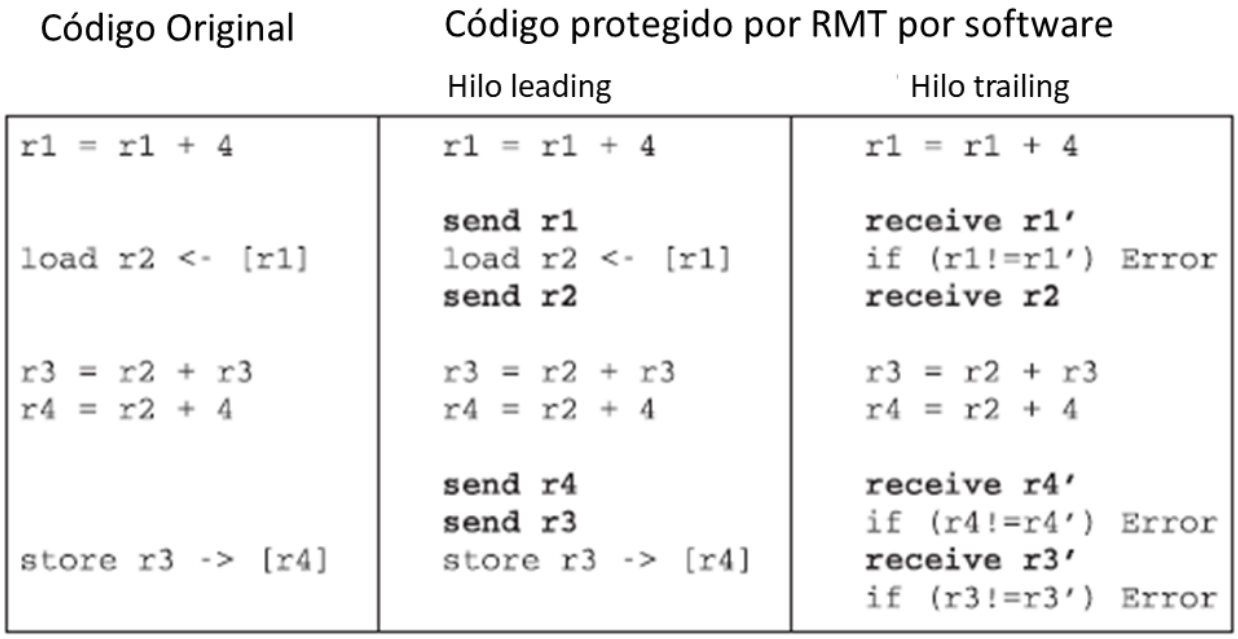

Figura 2.3: *Transformación de código con SRMT - extraído de [106]*

Debido a que las técnicas de *RMT* por software se basan en la duplicación del código, y el *overhead* de la comunicación entre *threads* es muy alto, como consecuencia de los requerimientos de sincronización los trabajos relacionados en esta área proponen optimizaciones para mejorar la degradación de la *performance*. Entre los más relevantes se encuentran *SRMT* (*Software Redundant Multithreading*) [144], *DAFT* (*Decoupled Acyclic Fault Tolerance*) [152], *COMET*, (*Communication-Optimized Multithreaded Error detection Technique*) [89] y *EXPERT* (*Effective and Flexible Error Protection by Redundant Multithreading*) [138].

Algunos pocos estudios proponen *RMT* a nivel de la aplicación, brindando *multithrea-*

ding como una interfaz configurable para el programador. El más destacado es *RedThreads (Interface for Application-Level Fault Detection/Correction Through Adaptive Redundant Multithreading)* [70]. En tanto, otros presentan un servicio del sistema operativo basado en *RMT* implementada en software para detección y recuperación de errores; el estudio más relevante es *Romain (Operating System Support for Redundant Multithreading)* [57].

En un intento de combinar las mejores características de las propuestas que utilizan componentes adicionales de hardware y las basadas en software puro, algunos autores han propuesto soluciones híbridas, que consisten en mecanismos robustos de tolerancia implementados en el hardware, pero controlados por el software que se ejecuta, buscando la protección completa del procesador [118].

2.4. Propuestas híbridas

En un enfoque diferente, los autores de [112] proponen *TALFT* (*Fault-Tolerant Typed Assembly Language*), un lenguaje ensamblador diseñado para ser tolerante a fallos. El trabajo se enfoca en proveer un marco teórico y probar formal y rigurosamente las propiedades que garantizan la corrección y fiabilidad del código máquina. La solución propuesta utiliza una combinación de software y hardware para lograr la protección para el flujo de control, similar a los procesadores *watchdog*, pero que hace explícitas ambas versiones de la protección, como en las estrategias basadas únicamente en software [33]. El objetivo principal de un sistema de tipo *TALFT* es asegurar que los programas escritos correctamente (siguiendo las reglas propuestas en el trabajo) mantienen un comportamiento seguro en presencia de fallos transitorios. En otras palabras, los programas escritos con un lenguaje ensamblador tolerante a fallos deben garantizar que los dispositivos de Entrada/Salida mapeados en memoria nunca pueden leer un valor erróneo y hacerlo visible para el usuario. De hecho, los autores hacen foco en que, independientemente del tipo de implementación (software puro, hardware o híbrida), ninguna otra solución comprueba, mediante un marco teórico formal, la robustez del código máquina para garantizar la fiabilidad y tolerancia a fallos, la cual es

la principal diferencia con el restante trabajo relacionado.

2.5. Tolerancia a fallos transitorios en cómputo paralelo

En el contexto de la popularidad alcanzada por los *clusters* y las aplicaciones paralelas que utilizan MPI, la fiabilidad de las aplicaciones y las capas de middleware se ha vuelto un aspecto relevante. La disponibilidad de los *clusters* ha aumentado inherentemente a nivel del hardware mediante el agregado de componentes redundantes. Las máquinas han incrementado su robustez, mediante mejoras en las redes, los sistemas operativos y los sistemas de archivos. Sin embargo, esa disponibilidad y fiabilidad no se transfieren automáticamente a las capas superiores.

A pesar de que el estándar MPI es el modelo de programación paralela más popular en los sistemas de memoria distribuida, no cuenta con soporte para tolerancia a fallos, y su especificación no permite explotar estas cualidades de las arquitecturas. La especificación asume que la infraestructura subyacente es confiable, y por lo tanto no trata con los errores que ocurren en tiempo de ejecución. Por lo tanto, las implementaciones no proveen a las aplicaciones mecanismos para tratar con fallas en los nodos o mensajes perdidos.

Dado que agregar características que mejoren la confiabilidad en las comunicaciones aumenta el *overhead*, el uso de MPI está limitado a entornos en los que la ocurrencia de fallos externos es abordada desde las aplicaciones, o en los cuales el hardware y los servicios de bajo nivel enmascaran los fallos. La detección de fallos no está definida en el estándar: sólo provee la detección de algunos errores internos locales, como el reporte de argumentos incorrectos de funciones o errores de recursos, como el intento de exceder el espacio de un buffer. En lugar de esto, MPI plantea un modelo estático que requiere que todos los procesos terminen exitosamente. Frente a una avería en un proceso, el estado de MPI resulta indefinido, y no existen garantías de que el programa pueda seguir correctamente con su ejecución. Por ende, el comportamiento por defecto es abortar la ejecución completa, lo cual implica que una falla que afecta a un único proceso resulta en la caída de toda la aplicación (de hecho,

este es el comportamiento que exige el estándar). Por ejemplo, si un único proceso MPI deja de responder (entra en un interbloqueo o en un lazo infinito) o se aborta prematuramente, se requiere la finalización completa de la aplicación (ya sea de forma "elegante" o anormal). Como consecuencia, la aplicación no puede lanzar un nuevo proceso para reemplazar al que ha fallado, ni ignorarlo para continuar la ejecución en un modo degradado. Estas limitaciones del estándar y de las implementaciones dificultan la detección y recuperación automática basada en software [10, 52]. Sin embargo, cuando ocurre un error, en muchas ocasiones tiene un impacto limitado y afecta sólo a un conjunto de los cores o nodos de cómputo en los cuales se está ejecutando la aplicación. En consecuencia, varios nodos pueden permanecer operativos, con lo que abortar la aplicación MPI y volver a lanzarla introduce *overheads* de recuperación innecesarios [82].

MPI proporciona al usuario dos posibles maneras para lidiar con los fallos. El modo por defecto es abortar inmediatamente la aplicación, como se mencionó. La segunda posibilidad, algo más flexible, consiste en devolver el control a la aplicación de usuario, sin garantizar el éxito de ninguna comunicación más allá del límite actual. Esta segunda opción busca darle a la aplicación la chance de realizar operaciones locales antes de finalizar, como por ejemplo cerrar todos los archivos o almacenar un estado consistente para la recuperación posterior. La notificación de los errores se lleva a cabo mediante la invocación de una función de manejo del error especificada por la aplicación. Si el manejador de error no detiene la aplicación, la llamada MPI retorna un código de error. El estándar no especifica todos los tipos de errores que se detectan ni los códigos de error [10].

La popularidad de MPI ha conducido a una cantidad de esfuerzos de investigación para robustecer la especificación, mediante la incorporación de características de fiabilidad. La mayoría de las soluciones propuestas (y los middlewares que resultan de ellas) se basan en un modelo de "caja negra", según el cual proveen abstracciones estáticas para la tolerancia a fallos, y un conjunto fijo de servicios (*checkpointing* transparente, *logging* de mensajes y recuperación por *rollback*) para todas las clases de aplicaciones paralelas [104]. Sin embargo,

la rigidez de estas soluciones resulta en *overheads* innecesariamente altos que se contraponen con el objetivo de MPI de lograr alto rendimiento, además de limitar la flexibilidad alcanzable.

Existen algunas aproximaciones que extienden MPI para implementar réplicas de los procesos de la aplicación para tolerar fallos permanentes [50, 53, 85]. Para limitar el *overhead*, sólo brindan cobertura a los fallos externos que producen la caída de un proceso. Como consecuencia, los fallos transitorios que provocan resultados incorrectos pero permiten que la aplicación MPI continúe su ejecución son ignorados. La detección de esos fallos debe ser provista por la aplicación por medio del uso explícito de técnicas tradicionales.

En los últimos años se han explordo nuevas soluciones para la tolerancia a fallos, y han surgido métodos para evitar las averías, que preventivamente migran procesos de procesadores propensos a fallar [143]. Lamentablemente, estas soluciones no pueden lidiar con los errores que ya han ocurrido.

MPI/FT [10] es un middleware basado en MPI que proporciona servicios adicionales de detección de fallos y recuperación de procesos MPI fallidos. Estos servicios están adaptados para diferentes modelos específicos de ejecución de aplicaciones, con el objetivo de optimizar el rendimiento y minimizar el *overhead* introducido. Esto resulta en una implementación del middleware diferente para cada modelo, por lo que la elección de servicios de tolerancia a fallos depende de la aplicación en particular. *MPI/FT* proporciona tolerancia a fallos introduciendo un coordinador central y/o replicando procesos MPI. De esta forma puede detectar mensajes erróneos, utilizando un algoritmo de votación entre réplicas, y es capaz de sobrevivir a la caída de los procesos. El inconveniente que presenta es la necesidad de recursos y la degradación parcial de rendimiento.

El modelo de fallo de MPI/FT asume que las aplicaciones no contienen errores internos (es decir, defectos en el software). Toda la detección se realiza por *time-out*. Para limitar el *overhead* introducido (cuya causa principal es el mecanismo de detección), MPI/FT sólo ofrece cobertura frente a los fallos externos que llevan a una caída del proceso que se

manifiesta como una falta de respuesta por parte del proceso involucrado. En cambio, no se trata con los fallos transitorios que producen SDC. La detección y cobertura frente a estos fallos debe ser provista por la aplicación.

Una conclusión generalizable útil es que el beneficio de invocar al mecanismo de recuperación depende claramente del momento en el que ocurre el fallo dentro del ciclo de vida de la aplicación. Si una aplicación está cercana a su finalización, y el proceso de recuperación no produce una mejora en el tiempo total de ejecución, o extiende ese tiempo, la aplicación puede decidir no realizar la recuperación y continuar, en modo degradado, sin el proceso que ha fallado.

FT-MPI (Fault-Tolerant MPI) [52] consiste en una extensión de MPI para proporcionar tolerancia frente a fallos de los procesos. Especifica la semántica para una versión de MPI tolerante a fallos, la arquitectura de la librería de comunicaciones y provee los detalles de una implementación de la especificación, que proporciona funcionalidad extendida para un modelo de recuperación frente a la ocurrencia de fallos. En *FT-MPI*, si un proceso termina prematuramente, los comunicadores que contienen a ese proceso quedan en un estado inválido. *FT-MPI* permite que la aplicación continúe, utilizando el comunicador que contiene al proceso fallido (pero excluyendo explícitamente las comunicaciones con ese proceso), o bien reduce el comunicador excluyendo a dicho proceso fallido, o bien genera un nuevo proceso para reemplazar el que ha fallado. Por lo tanto, *FT-MPI* no es un sistema automático de *checkpoint/restart*, pero proporciona a la aplicación la posibilidad de continuar a pesar de la ocurrencia de fallos en los nodos o en los canales de comunicación. Los comunicadores se reorganizan y la aplicación retoma su ejecución desde un punto bien definido a nivel de usuario. Sin embargo, definir e implementar un estado consistente en la aplicación es responsabilidad del desarrollador.

La capacidad del middleware de reparar un comunicador inválido produce que existan opciones de recuperación basadas en la aplicación, distintas de re-ejecutar desde el último *checkpoint*, lo que brinda flexibilidad para construir aplicaciones con mecanismos de recu-

peración. Una ventaja de esto es que no se requiere detener la aplicación, lo que ocurre en la mayoría de los sistemas de *checkpoint* a nivel de procesos. Sin embargo, *FT-MPI* no elabora estrategias de detección de fallos; las aplicaciones deben ser diseñadas específicamente para beneficiarse de sus características de tolerancia a fallos.

La especificación define dos modos para tratar con los mensajes cuando ocurre un error. En el primero, todos los mensajes en tránsito son cancelados por el sistema. Esto es particularmente útil para las aplicaciones que ante un error regresan al último estado consistente. El segundo modo completa la transferencia de todos los mensajes luego de la operación de recuperación, exceptuando los mensajes desde y hacia el proceso fallido. Este modo requiere que la aplicación mantenga información detallada del estado de cada proceso. Existen modos similares para las comunicaciones colectivas.

Las aplicaciones pueden detectar y manejar los eventos asociados con los fallos de dos maneras posibles: verificando los códigos de retorno de cada función de MPI, o utilizando los manejadores de error de MPI. Este segundo modo brinda a los usuarios la posibilidad de incluir una función, en la librería MPI, que es llamada en caso de que ocurra un error. De esta forma, no se debe modificar el código fuente existente para introducir funcionalidad detallada de verificación de errores para cada función MPI que se utilice. La utilización de manejadores de error de la especificación MPI mejora significativamente la legibilidad y mantenibilidad de las aplicaciones tolerantes a fallos.

Por otra parte, las estrategias de contención buscan mitigar las consecuencias de los fallos, previniendo su propagación hacia otros nodos o hacia los datos almacenados en *checkpoints*, que son necesarios para la recuperación [30]. En ausencia de mecanismos eficientes de contención, 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 son muy difíciles de corregir.

Por último, se debe destacar que también se han desarrollado protocolos tolerantes a fallos para otros modelos de programación paralela, como *PGAS* (*Partitioned Global Address*

Space) [5].

2.5.1. Aplicaciones MPI resilientes: *ULFM*

Como se mencionó en la Sección 2.5, un fallo en uno de los procesos de una aplicación paralela causa la finalización de ese proceso y provoca un estado indefinido en MPI, en el que no existen garantías de que la ejecución pueda continuar con éxito. Por lo tanto, el comportamiento por defecto es abortar la aplicación completa. Por este motivo, tradicionalmente, las averías en aplicaciones MPI (causadas por fallos de hardware o software que interrumpen la ejecución de la aplicación) han sido tratados mediante soluciones de *stop-and-restart* basadas en *checkpointing*, en las que se relanza la aplicación, recuperando su estado completo. Sin embargo, dados los efectos limitados que frecuentemente presentan las averías (a pesar de las cuales la mayoría de los nodos de cómputo suelen permanecer operativos), esta estrategia puede significar, a gran escala, abortar la ejecución de miles de procesos cuando únicamente uno de ellos falla, lo cual resulta sumamente ineficiente [82]. A raíz de esto se han realizado diferentes esfuerzos para incluir capacidades de tolerancia a fallos en el estándar MPI [52, 71]. En este contexto, hace algunos años, el *Fault Tolerance Working Group* (dentro del MPI Forum) ha propuesto la interfaz *ULFM* (*User Level Failure Mitigation*) [20, 21] como el último esfuerzo en esa dirección, para integrarlo en el futuro MPI 4.0. *ULFM* propone extender MPI para adecuarlo a entornos propensos a fallos, como los futuros sistemas de exa-escala, proporcionando funcionalidades que permiten la implementación de aplicaciones MPI resilientes, capaces de detectar y reaccionar a las averías, recuperándose por sí mismas sin interrumpir toda la ejecución, y evitando, por lo tanto, volver a lanzar la aplicación completa.

ULFM incluye nuevas semánticas para detectar averías que afectan a procesos (relacionados con comunicaciones), revocar comunicadores (es decir, invalidarlos para futuras comunicaciones), y reconfigurar comunicadores (lo que puede hacerse eliminando procesos fallidos del comunicador y rebalanceando la carga; o bien generando nuevos procesos y re-

construyendo el comunicador; a estas opciones se las denomina *shrinking* y *non-shrinking* respectivamente). Así, se pueden volver a lanzar los procesos fallidos, y el estado de la aplicación puede recuperarse con un *rollback* global. Sin embargo, *ULFM* no incluye ningún mecanismo especializado para recuperar el estado de los procesos fallidos. Esto permite flexibilidad a los desarrolladores del aplicaciones, para que puedan implementar la metodología de *checkpointing* óptima en cada caso, tomando en cuenta las propiedades de la aplicación particular.

Sin embargo, la incorporación de las funcionalidades de *ULFM* a los códigos de aplicaciones existentes es una tarea compleja y costosa en términos de tiempo. Existen diferentes propuestas en la literatura para implementar aplicaciones resilientes utilizando *ULFM*, la mayoría de las cuales son específicas para una aplicación o un conjunto de ellas. Todas estas propuestas toman en cuenta las características particulares de la aplicación para simplificar el proceso de recuperación. Los trabajos [109] y [22] se enfocan en los métodos Monte Carlo; en tanto, en [77] se evalúa *ULFM* sobre un código de dinámica molecular escalar masivo, mientras que en [4] y [121] se aplica a códigos de ecuaciones diferenciales parciales (*PDE*s - *Partial Differential Equations*).

Las soluciones a medida permiten reducir el *overhead* de recuperación cuando ocurre una avería, simplificando la detección mediante la verificación del estado de la ejecución en puntos específicos; evitan la necesidad de volver a generar los procesos fallidos cuando la aplicación permite "contraer" (*shrinking*) la cantidad de procesos MPI; y permiten recuperar los datos de la aplicación basándose en sus propiedades, como una alternativa al *checkpointing*. Sin embargo, no son soluciones generales, y no se pueden aplicar a cualquier aplicación *SPMD*.

Otras alternativas a *ULFM*, para implementar aplicaciones MPI resilientes, son *NR-MPI* [140], *FMI* [127] y *Reinit* [78]. A diferencia de *ULFM*, que propone una API de bajo nivel que soporta una variedad de modelos para tolerancia a fallos, estas alternativas proponen una interfaz simplificada, reparando el estado interno de MPI frente a la avería y regenerando los procesos fallidos. *Reinit* [78] propone un prototipo de interfaz de tolerancia a fallos

para recuperación global. *FMI* [127] es un prototipo de un modelo de programación, con una semántica similar a MPI, incluyendo el almacenamiento en *checkpoints* del estado de la aplicación, el relanzamiento de los procesos fallidos y la asignación de nodos adicionales cuando hace falta. Finalmente, *NR-MPI* [140] es una implementación resiliente de la semántica de *FT-MPI* [52] sobre MPICH. Estas propuestas ocultan la complejidad de reparar el estado de MPI; sin embargo, confían en los programadores para instrumentar y modificar el código de la aplicación para obtener soporte para tolerancia a fallos, incluyendo la responsibilidad de identificar qué datos de la aplicación se deben almacenar y en qué puntos de la ejecución.

2.6. *Checkpoint-Restart*

Las técnicas de *Checkpoint-Restart* (C/R) [44, 49] son las más populares dentro de aquellas que intentan tolerar los fallos limitando el impacto de las averías que causa la interrupción de la ejecución (*fail-stop*). Cada proceso de una aplicación almacena periódicamente su estado actual y sus resultados parciales en archivos de *checkpoint* en almacenamiento estable, lo que les permite utilizar esos datos para restaurar el programa a alguno de los estados intermedios para recuperar la ejecución. En la situación de que una falla externa afecte un proceso MPI, causando la caída de toda la aplicación, el usuario debe volver a lanzar la ejecución.

Las diferencias entre las formas de hacer *checkpointing* están relacionadas con el nivel del software en el que son creados los *checkpoints*, el modo en que son generados y almacenados. En el nivel más bajo, las herramientas de *checkpointing* a nivel de sistema, como las extensiones al sistema operativo, proporcionan *checkpointing* transparente de los procesos. Para ello, intentan salvar el estado sin tener ningún conocimiento previo, tratando a la aplicación como una "caja negra", por lo que no requieren ninún esfuerzo del usuario o del programador para obtener soporte para la tolerancia a fallos. Sin embargo, almacenan el estado completo de la aplicación, incluyendo registros de la CPU, la pila, etc. Esto atenta contra la portabilidad, debido a que el *restart* debe llevarse a cabo en la misma plataforma

de hardware y software [7]. El *overhead* de los *checkpoints* de nivel de sistema está vinculado con la cantidad de memoria utilizada por los procesos. Se han desarrollado varias herramientas para *checkpointing* transparente, de las cuales un ejemplo es *BLCR* (*Berkeley Lab Checkpoint Restart*) [66]. *BLCR* se basa en el *kernel* para suspender la aplicación y alamcenar los *checkpoints*. Una de sus ventajas es que evita parte del *overhead* ya que no encapsula por completo todas las llamadas al sistema. Además, como forma parte del *kernel* de Linux puede relanzar las aplicaciones en exactamente el mismo entorno UNIX (por ejemplo, conservando el *PID*).

Las soluciones a nivel de *kernel* requieren de la intervención del administrador para la instalación. Además, si no se toman en cuenta los recursos por fuera del sistema operativo local, no son capaces de almacenar y restaurar, por ejemplo, los *sockets* que se utilizan para la comunicación por la red. Como consecuencia, *BLCR* no puede utilizarse directamente en contextos MPI, por lo que las aplicaciones paralelas deben integrar soporte explícito para lidiar con esas limitaciones y permitir su uso en ambientes distribuidos [1].

Otra posibilidad consiste en proporcionar *checkpointing* transparente del espacio de usuario, encapsulando todas las llamadas al sistema, para hacer un *tracking* constante del estado de la aplicación. De hecho, como las herramientas son agnósticas al comportamiento de la aplicación, todas las llamadas potenciales que involucran recursos externos al proceso (como la red o el almacenamiento) deben capturarse. La herramienta *DMTCP* (*Distributed Multi-threaded Checkpointing*) [7] puede realizar *checkpointing* a nivel del espacio del usuario, inyectando una librería precargada compartida al inicio de la aplicación para encapsular las llamadas al sistema. Esto tiene la ventaja de no requerir acceso al *kernel* ni privilegios de administrador para la instalación, ni recompilar la aplicación para habilitar o actualizar el soporte. Desde esta óptica, la colaboración entre varios nodos es más sencilla, y el *checkpointing* de aplicaciones distribuidas no requiere necesariamente de implementaciones concientes de MPI. Por supuesto, la intercepción de todas las llamadas al sistema (y toda la contabilidad asociada) causan un *overhead* significativo. Más aún, se debe mantener un

log de los mensajes, para poder reproducirlos en caso de avería, lo que introduce un costo no despreciable para la aplicación.

En cambio, en las soluciones en capa de aplicación, sólo se realiza *checkpointing* de los datos que son relevantes o críticos para la aplicación, delegando el soporte en los desarrolladores; es decir, estas técnicas se basan en el conocimiento específico sobre la aplicación, ya sea provisto por el usuario o el programador [13] o por compiladores que analizan el código [28]. Una ventaja es que el *checkpointing* de capa de aplicación reduce el tamaño de los *checkpoints* (y por lo tanto el *overhead*), además de que generan archivos portables que permiten el *restart* en diferentes plataformas. Además, el usuario podría potencialmente modificar el comportamiento de la aplicación durante el *restart* y evitar la repetición de cierto cómputo (por ejemplo, ejecutar varias simulaciones modificando parámetros pero sin repetir la inicialización).

El empleo de esta técnica en aplicaciones paralelas supone que un programas es capaz de restaurar su estado (datos) y su tiempo actual (control) para retomar el cómputo desde donde había sido salvado. La forma más básica de lograr esto es que el mismo programa almacene manualmente los datos asociados a un cierto instante de tiempo y volverlos a cargar para relanzar. Este es un método portable, que tiene la ventaja de no requerir ninguna herramienta externa. La aplicación define qué datos deben ser salvados y el archivo de *checkpoint* resultante contiene exactamente lo que se requiere para el *restart*. Mientras el tiempo en que el programa permanece interrumpido sea bajo, el *overhead* del tiempo total de ejecución resulta pequeño. Un paso más allá consiste en considerar el almacenamiento de los archivos de *checkpoint* de forma redundante y compartida. Sin embargo, esto presenta problemas cuando se escala a miles de nodos o procesos. En este sentido, *SCR* (*Scalable Checkpoint and Restart*) [95] y *ACR* (*Automatic Checkpoint and Restart*) [102] solucionan esto almacenando los archivos de *checkpoint* en puntos locales rápidos, replicándolos para garantizar la redundancia. *(FTI)* (*Fault Tolerance library*) [13] también busca resolver estos problemas.

Lamentablemente, esta aproximación básica tiene ciertas limitaciones. En primer lugar, requiere que el conjunto de datos completo sea sencillo de serializar, se preserven y se mantengan en condiciones de ser restaurados, lo que puede resultar una tarea compleja. Además, requiere que cada aplicación implemente su propio formato de *checkpoint* y dediquen esfuerzos de desarrollo para proporcionar características similares.

Respecto a la generación y el almacenamiento de los *checkpoints*, existen diferentes enfoques en la literatura que buscan mejorar tanto la *performance* como la fiabilidad. Por ejemplo, la generación asincrónica de *checkpoints* reduce el *overhead* volcando los archivos de estado en el almacenamiento estable en *background*. Con la aparición de nuevos dispositivos de almacenamiento, se ha propuesto el *checkpointing* multinivel [13, 65], para explotar los distintos niveles de la jerarquía de memoria, y que incluyen características adicionales, como transferencias asíncronas al sistema de archivos paralelo. Los algoritmos semi-bloqueantes buscan almacenar los *checkpoints* sin detener la ejecución de la aplicación [103].

En cuanto a la implementación del mecanismo de *checkpointing* de capa de aplicación, existen diferentes técnicas de optimización que se enfocan en reducir el tamaño de los datos almacenados para reducir el costo. El *checkpointing* incremental [2, 101] propone reducir la cantidad de datos que se escriben en *checkpoints* consecutivos, pero no siempre se obtienen beneficios considerables. Por lo tanto, el *checkpointing* sin disco (*diskless*) [115] intenta resolver este problema. Otras estrategias son la compresión de datos en los archivos [114] y la exclusión de memoria [113].

Las aplicaciones paralelas requieren complejidad adicional en el protocolo de *checkpointing*. En las aplicaciones de memoria distribuida, las dependencias entre los procesos imposibilita la recuperación independiente de los procesos individuales. En cambio, una recuperación exitosa requiere que la aplicación sea restaurada desde un estado global consistente. Para esto, todos los procesos deben identicar el conjunto más reciente de *checkpoints* que corresponden a ese estado global consistente (llamado línea de recuperación válida). Los *checkpoints* coordinados [37] garantizan que la línea de recuperación es consistente, me-

diante la sincronización de todos los procesos al momento de hacer el *checkpointing*. Estos protocolos simplifican la recuperación pero requieren coordinación global entre todos los procesos, forzando el *rollback* todos ellos frente a una avería. Existen propuestas que se enfocan en mejorar la escalabilidad del C/R coordinado, de nivel de sistema, en entornos de HPC [29].

En cambio, los *checkpoints* no-coordinados permiten que los procesos almacenen sus estados independendientemente. Esto permite que los *checkpoints* se realicen cuando resulte más conveniente (por ejemplo cuando se requiere el almacenamiento de menor cantidad de datos). Sin embargo, el conjunto de archivos de *checkpoint* más recientes, generados por cada proceso pueden no representar un estado global consistente. En las aplicaciones MPI, las inconsistencias debidas a comunicaciones que cruzan una posible línea de recuperación, como mensajes huérfanos o en tránsito (ilustrados en la Figura 2.4 pueden forzar a un proceso a retroceder a un *checkpoint* previo, lo cual a su vez puede provocar que otros procesos hagan lo mismo. Incluso, las dependencias entre procesos pueden llevar a una situación en la cual todos los procesos necesiten ser relanzados desde el comienzo de la ejecución, llamada "efecto dominó", que implica un costo de recuperación inaceptable. La combinación entre *checkpointing* no-coordinado y protocolos de *logging* de mensajes [25, 86, 122] logra evitar este problema. El *logging* de mensajes provee flexibilidad, ya que potencialmente un proceso puede ser relanzado sin forzar el *rollback* de otros procesos. Sin embargo, los requerimientos de memoria y el *overhead* introducidos por el *logging* pueden ser factores limitantes.

En general, las técnicas basadas en C/R introducen un *overhead* asociado a la frecuencia del *checkpointing*: si se almacenan demasiado seguido, aumenta innecesariamente el *overhead* en ausencia de fallos, en términos de *performance*, espacio de almacenamiento y ancho de banda; sin embargo, si no se realizan con la suficiente frecuencia, la ocurrencia de fallos puede producir que se deba repetir una gran cantidad de trabajo, aumentando el *overhead* frente a las averías. Ambas condiciones evitan o retardan el avance del cómputo. Además, si se consideran máquinas con decenas de miles de procesadores, las técnicas de C/R tienen

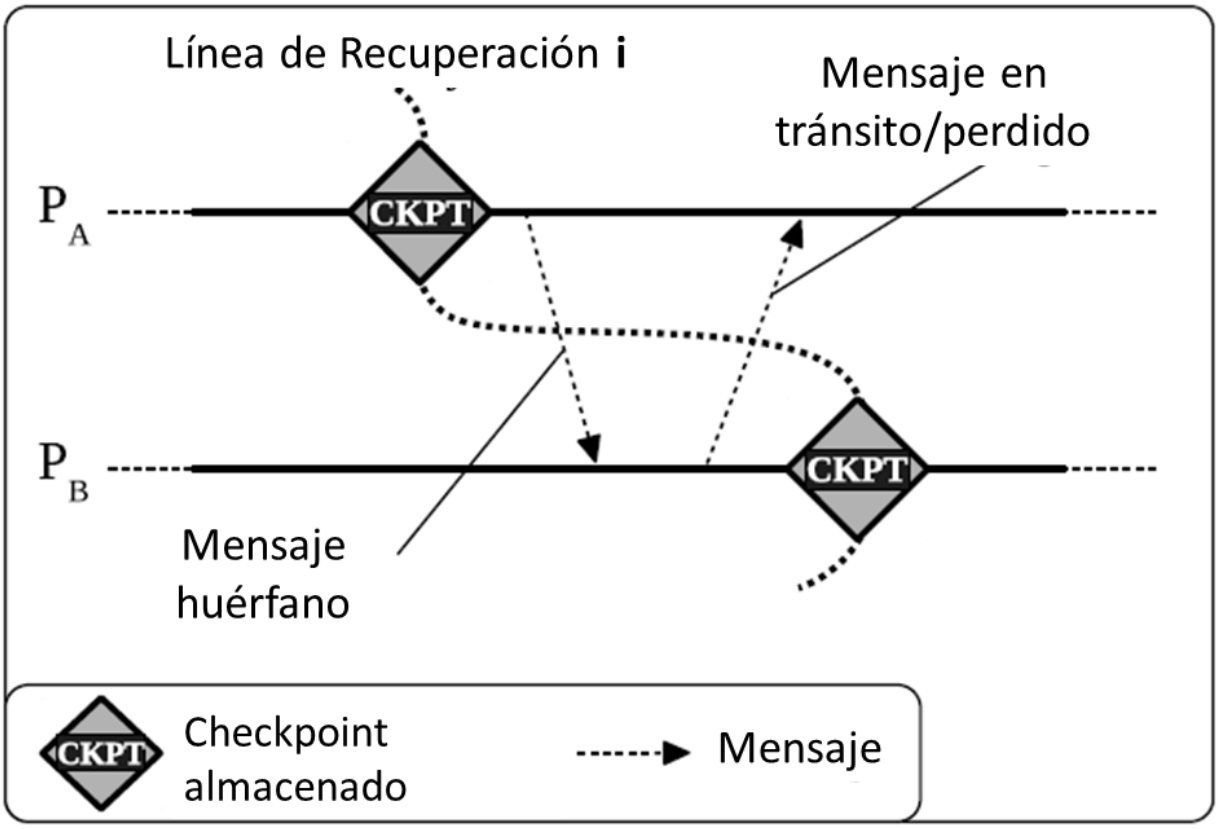

Figura 2.4: *Inconsistencias causadas por comunicaciones que cruzan la línea de recuperación - extraído de [82]*

limitaciones de rendimiento respecto de la escalabilidad [52]. Sin embargo, respecto del marco de esta tesis, las técnicas de C/R encuentran sus la mayores limitaciones cuando se enfrentan a fallos transitorios, ya que, en el modelo tradicional, asume que la detección se produce casi de inmediato. Por lo tanto, si los *checkpoints* almacenados contienen fallos no detectados, no se podrá efectuar la recuperación. En el escenario más desfavorable, si un fallo transitorio corrompe el resultado de una aplicación pero sin causar su finalización abrupta, el usuario debe esperar el término de la ejecución para hallar que los resultados son incorrectos o, peor aún, recolectarlos y utilizarlos como si fueran correctos [10, 16]. En tanto, las pocas técnicas generales de detección introducen altos *overheads* en aplicaciones

paralelas [55, 67], por lo que se espera que aumenten los rangos de latencia de detección, incrementando el problema debido a los *SDC*.

2.7. Soluciones específicas

Existen soluciones para tolerancia a fallos que aprovechan la información referente a las características de aplicaciones o algoritmos particulares para implementar técnicas de detección y recuperación *ad - hoc*, que pueden resultar en mejoras considerables de *performance* en el proceso de recuperación. Algunas estrategias sólo son aplicables a *kernels* algorítmicos específicos [34]; pueden utilizarse para reducir el costo de la detección en entornos de HPC [14]. Entre estos métodos, existen algunos bien conocidos como *ABFT* (*Algorithm-Based Fault Tolerance*) [24], que se basan en la utilización de *checksums* y son capaces de detectar hasta un número máximo de errores en problemas de álgebra lineal. Sin embargo, *ABFT* sólo puede proteger datos en este tipo de aplicaciones, y cada *kernel* requiere una implementación a medida, lo que representa un trabajo significativo para las grandes aplicaciones de HPC; por otra parte, las soluciones específicas son más intrusivas, debido a que modifican los algoritmos [34][130], a diferencia de propuestas como [14] o nuestra metodología SEDAR, que son agnósticas a los algoritmos.

ABFT fue originalmente introducido en [69], con el objetivo de detectar y corregir errores permanentes y transitorios en operaciones de matrices. El método se basa en la codificación de los datos en un alto nivel y en el diseño de algoritmos para operar sobre los datos codificados. *ABFT* ha sido usado en combinación con *diskless checkpointing* [35], y se ha implementado sobre algoritmos como los *benchmarks HPL* (*High Performance Linpack*) [39], la factorización Cholesky [64], algoritmos que utilizan matrices ralas y vectores densos [110], y aplicaciones basadas en tareas [148]. Un aspecto clave del proceso de recuperación es la localidad, es decir, la cantidad de procesos que no resultan afectados por el error pero que requieren participar en la recuperación. El hecho de restringir las acciones de recuperación sólo a aquellos procesos afectados, o a un subconjunto de los procesos (en los escenarios

en los que los procesos fallidos no pueden recuperarse por sí mismos y requieren de la participación de sus procesos vecinos) también contribuye a la efciencia de la solución de tolerancia a fallos.

Otras técnicas que se han explorado son las que buscan detectar errores en análisis numéricos de ecuaciones diferenciales ordinarias y parciales [18]. En [126] se proponen detección y corrección de errores en el método de gradiente conjugado. En [68] se utiliza replicación selectiva para tolerar *soft errors* en ecuaciones lineales, mientras que en [46] se diseña un *GMRES* (*Generalized Minimal Residual Method*) capaz de converger a pasar de errores silenciosos. En [27] se proporciona un estudio comparativo de costos de detección para métodos iterativos.

Las estrategias de recuperación hacia adelante (*forward recovery*) intentan construir un nuevo estado de la aplicación a partir del cual se pueda retomar la ejecución, sin retroceder hasta un estado almacenado previamente ni repetir el cómputo ya realizado, reduciendo significativamente el *overhead* de recuperación. Este es el caso de la re-computación parcial [134], que se enfoca en limitar el alcance de lo que vuelve a computarse luego de una avería.

Recientemente, se han propuesto detectores de errores silenciosos basados en el análisis de datos. Estos detectores utilizan técnicas de interpolación como la predicción de series temporales [19] e interpolación espacial multivariada [12, 61]. Estas técnicas ofrecen alta cobertura, introduciendo un *overhead* despreciable; sin embargo, no ofrecen cobertura completa, sino que pueden detectar un determinado porcentaje de corrupciones. A pesar de esto, la relación beneficio/costo de estos detectores es muy alta, lo que los convierte en opciones viables para grandes escalas.

2.8. Replicación de procesos en HPC

La replicación de procesos es una técnica bien conocida para tolerar fallos en sistemas que se enfocan en la obtención de una alta disponibilidad, manejando dinámicamente los procesos para mantener la "buena salud" de los comunicadores. En este tipo de soluciones,

el estado de los procesos se replica, de modo que si el proceso original falla, su réplica está disponible (o puede ser generada) para tomar el lugar del proceso original sin afectar a los demás procesos de la aplicación.

La replicación de procesos puede ser costosa en términos espaciales (si las réplicas utilizan recursos dedicados) o temporales (si las réplicas se alojan junto con los procesos primarios). Sin embargo, la replicación de procesos es capaz de incrementar significativamente el tiempo promedio hasta la ocurrencia de una interrupción en la ejecución (*MTTI - Mean Time To Interrupt*) de una aplicación. Algunas variantes de esta técnica pueden utilizarse para detectar fallos que producen *SDC* [31].

Principalmente debido a sus costos asociados, la replicación de procesos no ha sido demasiado aplicada, en sus comienzos, en el ámbito de HPC, sino que, en general, se ha confiado en los mecanismos de *checkpoint* coordinado y *rollback recovery* a un estado consistente previo a la ocurrencia del fallo. Sin embargo, en los últimos años, algunos estudios han concluido que las altas tasas de fallos en sistemas de exa-escala, en conjunto con los niveles de *overhead* que implican los mecanismos de C/R son factores que harán inviable la aplicación de soluciones basadas puramente en ellos. Por ejemplo, estudios independientes han llegado a la conclusión de que un sistema de exa-escala podría pasar potencialmente un porcentaje significativo de su tiempo leyendo y escribiendo *checkpoints* [48, 129]. Estos inconvenientes han sido la motivación para diferentes investigaciones que apuntan a mejorar la escalabilidad de los mecanismos de C/R [74, 95]. La mayoría de las propuestas involucran recursos adicionales, como almacenamiento local, enlaces de comunicaciones entre nodos o capacidad de memoria. Sin embargo, a la luz de los requerimientos de fiabilidad que se esperan para los sistemas de exa-escala, las investigaciones también se han volcado a examinar la aplicabilidad de la replicación de procesos para estos sistemas [51, 153].

En [54] se examina la viabilidad de la replicación de procesos como mecanismo primario de tolerancia a fallos, manteniendo el C/R para proporcionar un mecanismo secundario en caso de ser necesario, buscando las ventajas y limitaciones de ese esquema para aplicaciones

MPI de gran escala. Las réplicas de los procesos permiten la conmutación entre ellas en caso de error, lo que implica que se puedan atravesar las ocurrencias de fallos de manera transparente sin necesidad de retroceder. El esquema es mejorado mediante C/R, que se utiliza en los casos en los que la replicación es insuficiente (por ejemplo, si todas las réplicas de un proceso fallan simultáneamente o se vuelven inconsistentes debido a corrupciones del estado del sistema). Los autores realizan un estudio sobre cómputo redundante para aplicaciones de exa-escala, en donde analizan los problemas de escalabilidad de las técnicas de C/R (que a gran escala tienen requerimientos de hardware comparables a los de la replicación) y su incapacidad para tolerar fallos de hardware no detectados y errores que no provocan la interrupción de la ejecución. En particular, la utilización del C/R resulta problemática frente a los fallos transitorios, debido a que los éstos pueden corromper el estado almacenado en los *checkpoints*, de modo de que no podrían utilizarse en la recuperación. Frente a estos errores, las posibles soluciones serían re-ejecutar las aplicaciones desde el comienzo o analizar los contenidos de los *checkpoints* (ver Sección 2.9), en la búsqueda de uno que haya sido almacenado previamente al momento de la ocurrencia del fallo que corrompió el estado de la aplicación. En consecuencia, tanto en [54] como en otra serie de trabajos más recientes [14, 17] se evalúa la replicación del cómputo a nivel de procesos como una alternativa viable (en el contexto de errores *fail-stop*), ya que, para sistemas con escalas menores, conlleva un *overhead* relativamente reducido, resultando más eficiente que que el C/R tradicional en contextos de altas tasas de fallo y altos *overheads* de *checkpointing*. Sin embargo, 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 cuando se ejecutan operaciones no determinísticas) y reducir el consumo energético.

2.8.1. Replicación de procesos para aplicaciones de HPC con paso de mensajes

La replicación de procesos es conceptualmente directa en aplicaciones de HPC con paso de mensajes. Para cada proceso de la aplicación original se genera una réplica que se ejecuta en un hardware independiente. No siempre es necesario replicar todos los *ranks* (por ejemplo, en una aplicación con un patrón de comunicaciones *Master/Worker*, en la que el *Master* es capaz de recuperar la caída de un *Worker*, sólo se requiere replicar el proceso *Master*). El sistema de replicación garantiza que cada réplica recibe los mismos mensajes en el mismo orden y que una copia de cada mensaje de un *rank* se envía a cada réplica en el *rank* de destino. Por otra parte, el sistema de replicación debe detectar errores en las réplicas, reparar nodos caídos cuando sea posible y relanzar réplicas que han fallado desde nodos activos. Además, debe verificar periódicamente que los *ranks* replicados tengan el mismo estado. En tanto, el mecanismo de C/R aún resulta necesario para lograr la recuperación ante los fallos que afectan a todas las réplicas de un *rank* particular, así como para recuperar situaciones en las que el estado se vuelve inconsistente, como por ejemplo debido a fallos silenciosos no detectados.

Las soluciones basadas en replicación requieren de una gran cantidad de recursos adicionales: para la mayoría de las aplicaciones de HPC, se requieren $2N$ nodos de cómputo para replicar completamente un programa que de otro modo se ejecutaría en N nodos. Si es posible replicar sólo una parte de la aplicación (replicación parcial), estos requerimientos pueden reducirse significativamente. En cualquier caso, se introduce un *overhead* en tiempo de ejecución para mantener la consistencia entre las réplicas.

Sin embargo, estos costos traen aparejadas ciertas ventajas significativas:

- El $MTTI$ aumenta considerablemente, debido a que la replicación reduce la cantidad de errores visibles. Específicamente, la aplicación sólo es consciente de los errores que hacen fallar todas las réplicas de un *rank* particular. El $MTTI$ es el tiempo medio entre dos averías de la aplicación, tomando en cuenta la replicación. Si un proceso

experimenta una avería, la ejecución es capaz de continuar hasta tanto la réplica del mismo proceso también experimente una avería. Más precisamente, los procesadores se "aparean", es decir, cada procesador aloja una réplica, y la aplicación falla sólo si los dos procesadores del mismo par sufren averías. Por lo tanto, con la replicación, se considera el *MTTI* en lugar del *MTBF*, debido a que la aplicación puede "sobrevivir" a varias averías antes de caer.

- Los requerimientos de Entrada/Salida se reducen significativamente, debido a que el incremento del *MTTI* reduce la frecuencia a la que se deben almacenar los *checkpoints*, permitiendo que las aplicaciones utilicen más eficientemente el sistema.

- La comparación de los estados de las múltiples réplicas (por ejemplo, utilizando *checksums* de la memoria), previo a la escritura de un *checkpoint*, permite detectar si el estado de la aplicación ha sido corrompido (es decir, detectar un *soft error*) y relanzar la aplicación desde un checkpoint anterior.

- Los nodos adicionales que se utilizan como redundancia pueden utilizarse como potencia de cómputo extra cuando el sistema ejecuta muchas tareas menores, en las que la tolerancia a fallos no fuera un problema significativo.

Para estudiar el *overhead* en tiempo de ejecución introducido por la replicación de procesos, los autores de [53] han diseñado e implementado *rMPI* (*redundant MPI*), una librería completa MPI de nivel de usuario portable, que proporciona ejecución redundante para aplicaciones MPI determinísticas de manera transparente. *rMPI* está implementada sobre una versión de MPI existente, utilizando capa de *profiling*, y se ha evaluado en un sistema de gran escala (Cray XT-3/4), obteniendo como resultado que los *overheads* introducidos por la replicación son muy bajos para aplicaciones reales.

La idea básica de *rMPI* es replicar los procesos de la aplicación a nivel de *ranks*, y permitir a las réplicas que continúen en caso de que un *rank* original falle. Para garantizar que las réplicas mantienen estados consistentes, *rMPI* implementa protocolos que aseguran que los

mensajes están ordenados idénticamente entre las réplicas. A diferencia de otros protocolos de replicación más generales [31], estos protocolos son específicos para las necesidades de MPI, con el objetivo de reducir el *overhead* en tiempo de ejecución. *rMPI* se ocupa de fallos que causan la caída de procesos: el programa falla sólo si lo hacen dos nodos replicados. Esta forma de redundancia escala, en el sentido de la probabilidad de que un nodo y su réplica fallen simultáneamente disminuye al aumentar el número de nodos; sin embargo, este beneficio tiene el costo de duplicar la cantidad de recursos y cuadruplicar el número de mensajes. Una descripción completa de *rMPI*, que incluye los protocolos de bajo nivel y detalles de la implementación se encuentra en [26, 53].

Los autores de [54] utilizan una combinación de modelado, experimentación y simulación para evaluar las ventajas y desventajas de la replicación de procesos para un rango de parámetros del sistema, incluyendo los costos de hardware y software para aplicaciones MPI, diferentes distribuciones de fallos, anchos de banda de E/S y rangos de *MTTI*. Sus resultados muestran que, en un amplio rango de diseño de sistemas de exa-escala (aunque no para todos), la replicación de procesos mejora a las soluciones basadas en C/R. En particular, la replicación es viable en los casos en los que la cantidad de procesadores es alta pero el ancho de banda de E/S es limitado. Sin embargo, se requiere cuantificar el costo de software que implica la detección de errores silenciosos, debido a que las pruebas realizadas en [54] sólo buscan medir el costo de utilizar la replicación para proteger al sistema de errores que provocan la caída de la operación. En cambio, para las tecnologías actuales es muy complejo lidiar con *SDC* frecuentes.

La replicación de procesos puede ser parcial, es decir, sólo de un subconjunto de los procesos de la aplicación. En [50] se propone *MR-MPI* (*Modular Redundant MPI*), una solución que busca incrementar la disponibilidad en sistemas de HPC, a través de redundancia, ofreciendo un compromiso entre la cantidad y calidad de los componentes. *MR-MPI* propone replicación parcial, que puede combinarse con C/R en los procesos que no se replican, para proporcionar una solución transparente para el ámbito de HPC [47, 102].

Así como la replicación puede ser parcial, también pueden serlo las comparaciones. En este sentido, existen también soluciones que involucran menos recursos, a costa de sacrificar precisión en la detección. Una de ellas es la replicación aproximada [30], que establece límites superior e inferior para los resultados del cómputo. En [100] se abordan los fallos en sistemas de memoria compartida, proponiendo un esquema basado en múltiples hilos por proceso, que incluye el manejo del no-determinismo entre las réplicas debido a los accesos a memoria. En tanto, en [85] se describe un protocolo para programas paralelos híbridos que utilizan tareas, basados en MPI, que permite la recuperación basada en *checkpoints* no coordinados y *logging* de mensajes: únicamente la tarea afectada por el error se reinicia, y todas las llamadas a MPI son manejadas dentro de ella; sin embargo, es capaz de interceptar los *DUE*s, pero no funcional para errores silenciosos. También existen soluciones de replicación de software que operan a nivel de *threads*, en lugar de procesos [150]; en [147], esto se aplica a programas paralelos que utilizan memoria transaccional, siendo capaz de detectar tanto fallos permanentes como transitorios.

La mayoría de las propuestas de replicación de procesos para MPI se enfocan en los fallos que conducen a errores *fail-stop*.

RedMPI [55] es una librería MPI que utiliza la replicación por proceso implementada en *rMPI* para detectar y corregir *SDC*, comparando los mensajes enviados por emisores replicados en el lado del receptor. Utiliza una optimización basada en el cálculo de un *hashing* para evitar enviar y comparar los contenidos completos de los mensajes. Por otra parte, no requiere cambios al código fuente, garantizando además determinismo entre las réplicas. Los autores afirman que *RedMPI* es potencialmente utilizable en sistemas de gran escala, ya que provee protección aún en entornos con altas tasas de fallos introduciendo un *overhead*. Una importante contribución del trabajo es que se realiza un análisis de la propagación de los *SDC* entre los nodos del sistema 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.

2.9. Propuestas basadas en la combinación de Replicación y C/R

Debido a las altas tasas de fallos en los sistemas de gran escala, la replicación de procesos se combina con C/R periódico en platformas de HPC [51, 153]. Por lo tanto, cuando la aplicación falla, es posible recuperar la ejecución desde el último *checkpoint* válido, de la misma forma en la que ocurre cuando no hay replicación. Sin embargo, debido a que deben ocurrir varias averías para interrumpir la aplicación, el intervalo de *checkpointing* puede ser mucho más largo que sin replicación. En el reciente trabajo [17], los autores exploran la combinación de la replicación y el *checkpointing* para errores *fail-stop*, y computan el intervalo óptimo de *checkpoint* para este escenario. Además, proponen la estrategia *restart*, en la cual, luego de cada *checkpoint* se relanzan todos los procesos que hayan fallado hasta el momento, en lugar de sólo relanzar cuando la aplicación sufra una avería fatal. En tanto, en [15] se explora la combinación de realizar *checkpointing* de los resultados de las tareas, y replicación para detección específica para la aplicación, en un contexto de flujos de trabajo lineales, tanto en presencia de errores *fail-stop* como silenciosos.

Así como el C/R es la técnica de recuperación estándar para los errores *fail-stop*, no hay una estrategia de propósito general ampliamente adoptada para lidiar con los errores silenciosos. El problema principal es la latencia de detección, ya que, contrariamente a los errores *fail-stop* no se detectan immediatamente, sino sólo cuando los datos corrompidos conducen a comportamientos inusuales de la aplicación, mientras que la recuperación basada en C/R asume que la detección es instantánea. La dificultad que deviene de esto es que si el fallo incide antes del almacenamiento del último *checkpoint*, pero se detecta después, el *checkpoint* está corrompido y no puede utilizarse para recuperar (esto se retoma en la Sección 4.2).

Para solucionar esto, una posibilidad es mantener varios *checkpoints* en la memoria, restaurando la aplicación desde el último *checkpoint* válido, retrocediendo al último estado correcto [83]; una de las estrategias de recuperación de SEDAR se basa en implementar una

idea similar a esta. Esta solución basada en múltiples *checkpoint* tiene tres inconvenientes. En primer lugar, es altamente demandante en términos de almacenamiento, debido a que cada *checkpoint* típicamente representa una copia completa del espacio de memoria de la aplicación, que podría consistir en varios terabytes. El segundo problema es la posibilidad de averías fatales. Aún manteniendo k *checkpoints*, es factible que el error que se detecta haya ocurrido previo al almacenamiento del primer *checkpoint*; por lo tanto, todos los *checkpoints* activos estarían corrompidos, y la única opción sería re-ejecutar la aplicación completa desde el comienzo. La probabilidad de que esto ocurra se evalúa en [8] para varias distribuciones posibles de errores y valores de k. El tercer problema es el más crítico, aún si no existieran limitaciones de memoria y se pudieran mantener infinitos *checkpoints* en el almacenamiento, y consiste en la determinación de cuál es el último *checkpoint* válido, la cual resulta una información necesaria para lograr una recuperación segura a partir de ese punto. Excepto que se implemente un mecanismo de verificación de los *checkpoints* (justo antes o después de almacenarlos), es imposible identificar el último *checkpoint* válido, debido a que la latencia de detección no permite conocer el momento exacto de ocurrencia del error silencioso. En [16] se evalúa el costo de ese mecanismo de verificación, independientemente de la naturaleza del mismo (*checksum*, ECC, tests de coherencia, etc.). La evaluación es de propósito general; sin embargo, si hay disponible información específica de la aplicación, puede utilizarse para reducir el costo del mecanismo de verificación. En este contexto, proponen almacenar sólo *checkpoints* que han sido verificados (los denominan VCs - *Verified Checkpoints*), realizando la verificación justo antes del almacenamiento. Si la verificación es exitosa, el *checkpoint* se almacena con seguridad, en tanto que si falla, significa que el error silencioso ha ocurrido después del *checkpoint* previo, que había sido verificado en su momento, por lo que retomar la ejecución desde ese *checkpoint* es posible. Por supuesto, si ocurre un error *fail-stop*, también es posible recuperar desde el último *checkpoint*, como en el método tradicional de C/R. Sin embargo, el análisis que conduce a la estrategia óptima es más complejo, por cuanto se deben tomar en cuenta dos tipos de errores, de los cuales uno se detecta de immediato

y el otro sólo durante la operación de verificación. Los autores de [16] plantean que, si los errores silenciosos son suficientemente frecuentes, incluso valdría la pena validar los datos entre medio de dos *checkpoints* (verificados), para reducir la latencia de detección, y por lo tanto re-ejecutar menos trabajo.

Siguiendo esta línea, los autores de [15] proponen la verificación a nivel de tareas, en aplicaciones divisibles en tareas periódicas, pasando a la tarea siguiente si la verificación es exitosa, o regresando al último *checkpoint* almacenado si falla. Debido que (a diferencia de los errores *fail-stop*), los errores silenciosos no causan la corrupción de la memoria completa del procesador afectado, los autores utilizan un esquema de dos niveles de *checkpointing*: el archivo se almacena en la memoria principal del procesador antes de ser transferido a algún medio de almacenamiento resiliente a los errores *fail-stop* (como el disco), de modo de que es posible recuperarse más rápido de un error silencioso que de un error *fail-stop*. Esta combinación de verificación y *checkpointing* garantiza que no hay errores irrecuperables que produzcan la caída de la aplicación: el último *checkpoint* de cada tarea periódica siempre es correcto, ya que siempre se realiza una verificación justo antes de su almacenamiento. En tanto, si la verificación revela un error, se retrocede hasta el último punto de verificación correcta, que a lo sumo es al final de la tarea previa, pero no más de allí.

En [83] se introduce un modelo basado en múltiples *checkpoints* y se computa el intervalo óptimo de *checkpointing* y su relación con la latencia de detección. Se realiza un conjunto de simulaciones para evaluar el *overhead* introducido al mantener sólo los últimos k *checkpoints* por limitaciones de almacenamiento. Sin embargo, el modelo introducido en [83] no considera la forma de determinar cuál es el último *checkpoint* válido, al no tomar en cuenta la latencia de detección. En [8] se introduce un modelo que combina verificación y *checkpointing*, y determina analíticamente el mejor balance entre los *checkpoints* y las verificaciones de modo de optimizar la productividad de la plataforma. La dificultad principal es que cuando la latencia de detección es demasiado alta, no es posible recuperar desde un *checkpoint* válido, y se debe relanzar la ejecución desde el comienzo. Los autores analizan

el modo de mantener ese riesgo bajo un cierto umbral. En tanto, si se introduce un mecanismo de verificación, y de forma agnóstica a su naturaleza y al mecanismo de detección subyacente, resuelven numéricamente cuál es la estrategia óptima para minimizar el gasto debido a dicho mecanismo de verificación, como función del *MTBE*, considerando los costos de *checkpointing*, recuperación, verificación y tiempo fuera de servicio como parámetros de entrada al modelo.

2.10. Diferencias de SEDAR con las propuestas existentes

A diferencia la mayoría de las propuestas descriptas en este capítulo para aplicaciones de HPC, que se enfocan en el soporte para fallos que provocan la finalización abrupta de los procesos o aplicaciones, SEDAR es una estrategia diseñada específicamente para detectar y recuperar las ejecuciones frente a fallos transitorios que afectan silenciosamente los resultados de aplicaciones paralelas científicas de paso de mensajes.

De manera similar a SEDAR, *RedMPI* [55] también permite que las réplicas sean mapeadas en los mismos nodos físicos que los procesos originales, o en sus vecinos con menor latencia de comunicación por red (ver Sección 3.2.4). Sin embargo, la replicación de SEDAR es a nivel de *threads* (posibilitando el aprovechamiento de niveles compartidos de caché o memoria), mientras que *RedMPI* realiza la replicación a nivel de procesos.

Al igual que SEDAR, *RedMPI* también se enfoca en los contenidos de los mensajes, como forma de monitorear los datos más críticos para la aplicación, bajo la misma premisa de que la correctitud en las comunicaciones es necesaria para la correctitud de la salida. La detección se difiere hasta el momento de la trasmisión, ya que el *SDC* puede afectar a datos que no se comunican inmediatamente. Sin embargo, a diferencia de SEDAR, *RedMPI* realiza la validación del lado del receptor, ya que, en el emisor, todas las réplicas deberían comunicarse 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. En cambio, como SEDAR realiza la replicación 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, SEDAR no congestiona la red con tráfico adicional. Al igual que SEDAR, *RedMPI* permite que, aún sin recuperación, la corrupción quede confinada en un proceso sin propagarse a los demás. 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).

Como se mencionó en la Sección 2.9 anterior, los autores de [16] proponen el almacenamiento de *checkpoints* verificados, realizando la verificación justo antes de almacenar el archivo. En el caso de SEDAR, el mecanismo de recuperación basada en *checkpoints* de capa de aplicación (ver Sección 4.3) propone una estrategia similar. Sin embargo, en SEDAR se almacena un *checkpoint* por cada *thread*, además de que *checkpoints* se validan ni bien acaban de ser almacenados.

Por otra parte, los autores de [16] afirman que el almacenamiento de *checkpoints* sin ninguna verificación de sus datos no es una buena idea, debido al costo de memoria y al riesgo de almacenar datos corrompidos. En el trabajo, se evalúa el costo del mecanismo de verificación, pero no realizan ninguna implementación, debido a que el análisis es independiente del mecanismo de verificación utilizado. Por otra parte, cada *checkpoint* almacenado debe ser verificado. Si se detecta un error, debe volver a verificarse en la re-ejecución; además, se proponen verificaciones adicionales. En cambio, en el caso de SEDAR (como se describe en la Sección 4.2) se ha encontrado la dificultad de implementar el mecanismo de verificación para *checkpointing* transparente de nivel de sistema. En consecuencia, en SEDAR se ha implementado un mecanismo de recuperación basado en múltiples *checkpoints* sin verificación. Esta estrategia es más optimista en ausencia de fallos, ya que no consume el tiempo en verificar cada *checkpoint*, pero tiene la desventaja de no poder determinar el último *checkpoint* válido, y de, en caso de latencias altas, requerir acumular intentos de *restart*. En tanto, dada su factibilidad, en SEDAR se ha diseñado un mecanismo de verificación para los *checkpoints*

de capa de aplicación.

Capítulo 3

Detección de fallos transitorios en sistemas de HPC

Resumen

En este capítulo se describe detalladamente la metodología de detección de SEDAR, llamada *SMCV* (*Sent Message Content Validation*) [91, 92, 94], que es una técnica de software puro, completamente distribuida. *SMCV* proporciona detección de fallos para aplicaciones paralelas de paso de mensajes mediante la replicación de procesos, la validación de los contenidos de los mensajes que se van a enviar y la comparación de resultados finales. De esta forma, evita que los fallos que se producen en el ámbito de un proceso se propaguen al resto de la aplicación y restringe la latencia de detección y notificación de la ocurrencia del error. Por otra parte, aprovecha la redundancia intrínseca de hardware que es propia de la arquitectura de *cluster* de multicores. La técnica *SMCV* es capaz de brindar cobertura frente a todos los fallos *SDC* y *TOE*, a costa de un *overhead* reducido. Además, logra un compromiso entre la latencia de detección moderada y la sobrecarga de operación.

3.1. Modelo de fallo

En el resto del trabajo, se asumirá el modelo de fallo *SEU* (*Single Event Upset*), en el que exactamente un bit resulta invertido a lo largo de la ejecución. La técnica *SMCV* propuesta tiene por objetivo detectar este tipo de fallos, pero presenta algunas vulnerabilidades en

casos donde se presenten fallos múltiples relacionados. Sin embargo, como se justifica en la literatura [119], los casos en los que múltiples fallos pueden combinarse para escapar de los mecanismos de detección basados en replicación son extremadamente poco probables, de manera que la detección de fallos *SEU* es la meta principal. Sin embargo, debe diferenciarse entre fallos relacionados y fallos independientes (o no relacionados). Más adelante en este capítulo se darán mayores detalles sobre este punto (Secciones 3.7 y 3.8).

Por otra parte, también se asume que el subsistema de memoria, incluyendo los diferentes niveles de caché, están adecuadamente protegidos mediante la utilización de técnicas como los bits de paridad o los Códigos de Corrección de errores (*ECC*s). Esta afirmación se cumple para la gran mayoría de los sistemas de cómputo modernos [119].

Como se vio en el capítulo 1, los fallos transitorios representan un problema sólo cuando modifican resultados de cómputo que son observables externamente, como resultados totales o parciales del programa. El único evento observable desde el exterior de la máquina es la secuencia de escrituras sobre la memoria los dispositivos de salida. Por lo tanto, independientemente de la operación interna del sistema, la única forma de que el efecto de un fallo se haga visible es que el programa escriba valores alterados en memoria, donde un dispositivo pueda leerlos y procesarlos [112]. En consecuencia, aunque el comportamiento del procesador se aparte del esperado, se considera que el programa se ejecuta correctamente si la secuencia de valores escritos a memoria resulta inalterada. Un fallo transitorio puede afectar a un bit cualquiera dentro del procesador. Este bit puede pertenecer a una instrucción, un dato o una dirección. En la Sección 1.7.1 se explicó el caso de los fallos que afectan a los códigos de instrucción, que normalmente se resuelven en excepciones por instrucción inválida.

Si un fallo afecta a una dirección, es decir, altera el flujo, en última instancia se podrá observar una modificación en la secuencia de valores que el programa almacena en memoria. En general, el hardware provee mecanismos para detectar fallos en las direcciones que sirven como destinos de saltos. Un fallo que genere una dirección inválida, en la práctica producirá una excepción de hardware (como un $segmentation fault$)

En el caso de que un fallo altere un dato, las instrucciones de carga de datos (*load*) no son peligrosas, en el sentido de que no pueden afectar la fiabilidad de la ejecución, aunque resulten en la carga de un valor arbitrario. Si el dato corrompido es leído luego, la consecuencia será observable al momento de escribir en memoria un resultado calculado en base a dicho valor erróneo. Debido a esto, el fallo será, en definitiva, detectado (aún a costa de cierta latencia adicional) si se verifican los valores que se escriben en memoria.

En general, en los modelos no se contemplan los fallos que ocurren **durante** la ejecución de las instrucciones. Esto se debe a que estos fallos se pueden equiparar con una ejecución correcta seguida inmediatamente de un fallo. Por ejemplo, en una sencilla instrucción de suma aritmética entre valores alojados en registros, una falla en el hardware del sumador durante la ejecución es equivalente a la ejecución correcta de la suma, seguida de un fallo que afecte al registro de destino. De esta misma manera, la mayor parte de los fallos intra-instrucción se pueden modelar modificando el archivo de registros antes o después de la instrucción. Debido a esta equivalencia, las instrucciones aritmético-lógicas básicas no se consideran operaciones peligrosas desde el punto de vista de la fiabilidad. En conclusión, las instrucciones de almacenamiento de datos en memoria (*store*) son las únicas que influyen sobre la fiabilidad de la ejecución, ya que hacen visibles los resultados hacia el exterior. Sin un hardware especial es imposible garantizar que las instrucciones de almacenamiento acceden correctamente a memoria. Una verificación realizada por software justo antes de una instrucción *store*, por sofisticada que sea, resultará inútil si el fallo ocurre justo entre la comprobación y la escritura en memoria [112]. Retomaremos este aspecto al tratar sobre las vulnerabilidades del método, en la Sección 3.7

3.2. Metodología *SMCV* para detección de fallos transitorios

3.2.1. Fundamentación

SMCV, la metodología de detección de SEDAR, es una propuesta diseñada específicamente para detectar fallos transitorios en aplicaciones científicas paralelas determinísticas que utilizan paso de mensajes, y que se ejecutan sobre un *cluster* de multicores. *SMCV* es una técnica basada puramente en replicación de software, que busca utilizar la redundancia de hardware propia de los multicores en beneficio del sistema, ejecutando las réplicas de los procesos de la aplicación paralela en cores del mismo procesador. La detección se realiza mediante la validación de los contenidos de los mensajes que se van a enviar a otros procesos de la aplicación (lo que implica un intervalo de validación moderado), introduciendo una reducida sobrecarga de trabajo y un bajo *overhead* respecto del tiempo de ejecución.

SMCV es una estrategia distribuida, que mejora la fiabilidad del sistema compuesto por el *cluster* y la aplicación paralela, confinando el efecto de los fallos que ocurren en el contexto de un proceso y evitando que se propague hacia los demás. El objetivo principal es que las aplicaciones que logren finalizar lo hagan con resultados correctos. En particular, *SMCV* es capaz de detectar todos los fallos que producen *TOE* y *SDC* (tanto *TDC* como *FSC*). A diferencia de otras propuestas basadas puramente en software, descriptas en el Capítulo 2 (como por ejemplo *SWIFT* [119] o *PLR* [133]), que están diseñadas para programas secuenciales, *SMCV* es específica para aplicaciones paralelas determinísticas de paso de mensajes. Por otra parte, en comparación a las propuestas que se utilizan en entornos MPI, como por ejemplo *MPI/FT* [10] o *FT-MPI* [52], que soportan fallos que producen la caída de los procesos, *SMCV* provee un mecanismo para detectar fallos transitorios silenciosos.

Otro aspecto destacable es que *SMCV* no realiza modificaciones en los algoritmos hacen los detectores específicos basados en el algoritmo como ABFT y otros específicos para *kernels* particulares ([24] [34] [130]), y es casi transparente para la aplicación. Esto hace que, en general, cualquier aplicación de paso de mensajes pueda utilizarlo. En el Capítulo 5

se profundizará en este aspecto.

En las siguientes subsecciones se describen los fundamentos sobre los que se basa *SMCV*, la hipótesis de partida y la solución propuesta. En primer lugar, se explica la utilidad de validar los contenidos de los mensajes y de comparar los resultados finales, y luego, se describe el aprovechamiento de recursos redundantes de hardware para mejorar la fiabilidad del sistema.

3.2.2. Validación de contenidos de mensajes antes de enviar

La metodología de detección propuesta se basa esencialmente en la hipótesis de que, en un sistema formado por un *cluster* de multicores sobre el que se ejecuta una aplicación paralela de paso de mensajes, la mayor parte del cómputo significativo (entendido como el que incide sobre los resultados de la aplicación) forma parte del contenido de un mensaje que se envía a otro proceso de la aplicación, en algún momento durante la ejecución.

Como se explicó en el Capítulo 1, los fallos pueden corromper datos, direcciones, códigos de operación o flags. Sin embargo, si el valor alterado es relevante para el resultado de la aplicación, la situación eventualmente se reflejará en un mensaje incorrecto. Por lo tanto, de todos los fallos que pueden producir *SDC*, la mayor parte entra en la categoría de *TDC* (ver Sección 1.8.2). En consecuencia, la monitorización del contenido de los mensajes es un método viable para detectar errores que afectan datos importantes.

Tomando esto en cuenta, *SMCV* se basa en validar los contenidos de los mensajes que se van a enviar. Para ello, cada proceso de la aplicación se duplica, y ambas réplicas comparan todos los campos que conforman el mensaje previo al envío; el mensaje se envía sólo si la comparación resulta exitosa. De esta manera, la utilización de *SMCV* permite detectar todos los fallos que causan *TDC*; desde el punto de vista de la aplicación paralela, garantiza que la ocurrencia de un fallo que afecta el estado de un proceso no se propaga a ningún otro proceso de la aplicación, confinando su efecto al proceso local. En ausencia de una estrategia de recuperación, la detección de un *SDC* o *TOE* produce que el mecanismo *SMCV* notifique

al usuario y conduzca al sistema a una parada segura, evitando que se comuniquen resultados defectuosos. En el Capítulo 4 se verá la integración de esta estrategia con algoritmos de recuperación automática.

Como se mencionó, los contenidos de los mensajes son validados previamente al envío. Por ende, sólo una de las réplicas necesita efectivamente enviar el mensaje, lo que implica que no se consume ancho de banda de red adicional, como ocurre en otras propuestas [55]. Si se considera que las redes actuales utilizan protocolos que garantizan comunicaciones confiables, no es necesario validar los contenidos de los mensajes del lado del receptor, lo cual requeriría tener que transmitir dos copias del mensaje.

3.2.3. Comparación de resultados finales

En la Sección 1.8.2 se mencionó que, en una aplicación paralela, una fracción de los fallos que ocurren alteran datos que no se comunican a otros procesos, sino que se propagan localmente produciendo resultados incorrectos que sólo modifican los resultados propios del proceso afectado. La consecuencia de estos fallos es la *FSC*, es decir, se comportan de la misma forma que los fallos que ocurren durante la ejecución de aplicaciones secuenciales.

La forma de detectar estos fallos que afectan a la fase serie de la aplicación (es decir, donde no se producen comunicaciones), es sencillamente agregar una instancia de validación de los resultados finales del programa. *SMCV* incorpora esta verificación final para garantizar la fiabilidad del sistema (por supuesto, la latencia de detección es elevada en este caso); por lo tanto, los resultados de todas las aplicaciones que consiguen llegar al final de su ejecución (sin que se produzca parada segura) son los correctos. Dicho de otra forma, ninguna aplicación paralela determinística, cuya ejecución sea monitorizada por *SMCV*, finalizará su ejecución con resultados erróneos.

3.2.4. Aprovechamiento de recursos redundantes del sistema

Actualmente, se tiende a incorporar cada un mayor número de cores a los procesadores. Sin embargo, no siempre las aplicaciones son capaces de sacar provecho a los recursos de

cómputo en forma eficiente. Por otra parte, el incremento de la cantidad de fallos transitorios va de la mano con el crecimiento del número de núcleos de procesamiento; dicho de otra forma, el $MTBE$ de un sistema con N procesadores decrece linealmente con N, es decir $MTBE = MTBE_{ind}/N$, donde $MTBE_{ind}$ es the $MTBE$ de un procesador individual [14]. Como consecuencia, el foco que se ha hecho tradicionalmente en la performance del procesador se ha vuelto más amplio, y factores como la fiabilidad y la disponibilidad se han vuelto más relevantes. Por lo tanto, el uso de cores de procesamiento para realizar tareas relacionadas con la tolerancia a fallos tiene dos grandes ventajas: mejora la utilización eficiente de los recursos e incorpora una característica beneficiosa para el sistema. En este contexto, *SMCV* aprovecha la redundancia de hardware que existe intrínsecamente en los multicores, utilizando cores del CMP para alojar a las réplicas de los procesos de la aplicación. De esta forma, la mitad de los recursos de cómputo se utilizan como redundancia. El aspecto crítico, al momento de elegir los cores que se van a utilizar para detectar los fallos que ocurren en los demás, es el acceso a memoria principal. *SMCV* busca explotar la jerarquía de memoria del CMP, de forma que la replicación del cómputo que se lleva a cabo en un determinado core se asigna a otro core con el que comparte algún nivel de caché. Por lo tanto, muchas operaciones de comparación entre réplicas se resuelven en el último nivel de caché, minimizando el acceso a la memoria principal. En la Figura 3.1 se esquematiza esta forma de utilizar el hardware para la detección de fallos.

Como cualquier mecanismo basado en duplicación, *SMCV* incrementa la fiabilidad, a costa de asignar la mitad de los cores del sistema para proteger las ejecuciones. Sin embargo, no introduce ningún costo adicional: aprovecha los cores disponibles, es decir, la redundancia intrínseca del sistema, sin modificaciones ni necesidad de hardware específico. Debido al hecho de que no todas las aplicaciones son capaces de hacer un uso eficiente de los recursos disponibles, tanto en términos de aceleración del tiempo de ejecución (*speedup*) como de consumo energético (es decir, no poseen escalabilidad fuerte) [107, 116], utilizarlos para resiliencia es una manera útil de sacar ventaja de ellos.

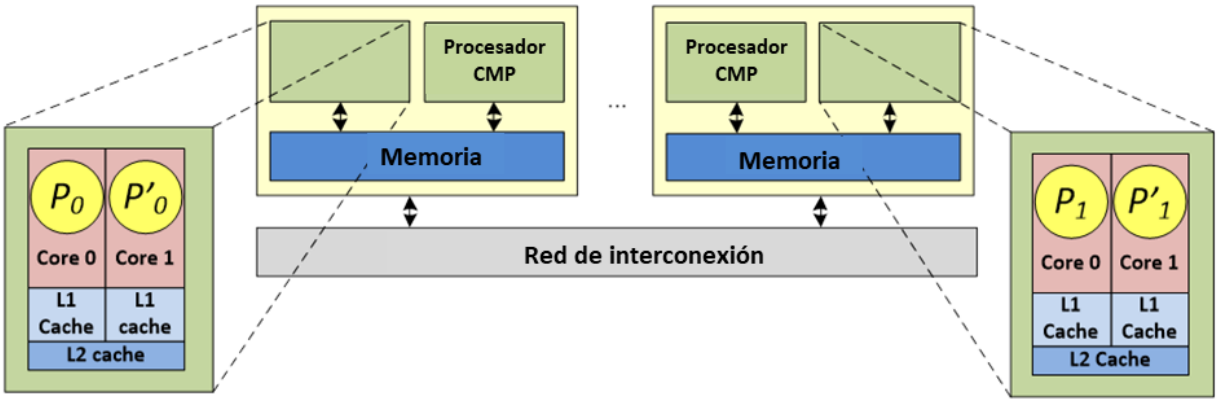

Figura 3.1: *Ubicación de las réplicas de los procesos de la aplicación para aprovechar el hardware redundante*

3.3. Descripción de la operación

Como se mencionó en la sección 3.2.1, *SMCV* es una estrategia basada únicamente en software, capaz de detectar fallos transitorios que ocurren en los *clusters* de multicores sobre los que se ejecutan aplicaciones paralelas científicas de paso de mensajes. Cuando se detecta un fallo, se reporta al usuario y se detiene la aplicación, mejorando la fiabilidad del sistema. La Figura 3.2 muestra un esquema de la metodología de detección propuesta, mientras que en la 3.3 se observa el comportamiento frente a la ocurrencia de un fallo.

Cada proceso de la aplicación paralela se ejecuta sobre un core del CMP, y el cómputo que realiza es replicado en un *thread*, que a su vez se ejecuta en un core que comparte algún nivel de caché con el core original. Este esquema minimiza el acceso a la memoria principal, aprovechándose la jerarquía de memoria para resolver las comparaciones. Cada proceso se ejecuta concurrentemente con su réplica, lo que requiere un mecanismo de sincronización entre ellas. Cada vez que se va a realizar una comunicación, ya sea punto a punto o colectiva, el proceso se detiene temporariamente y espera a que su réplica alcance el mismo punto de su ejecución. Una vez allí, se verifican todos los campos que componen el mensaje MPI que se va a enviar, byte a byte, para validar que los contenidos calculados por ambas réplicas son los mismos. Sólo si se cumple esta condición, una de las réplicas envía el mensaje, asegurando

de esta forma que ningún dato corrompido se propaga a los demás procesos.

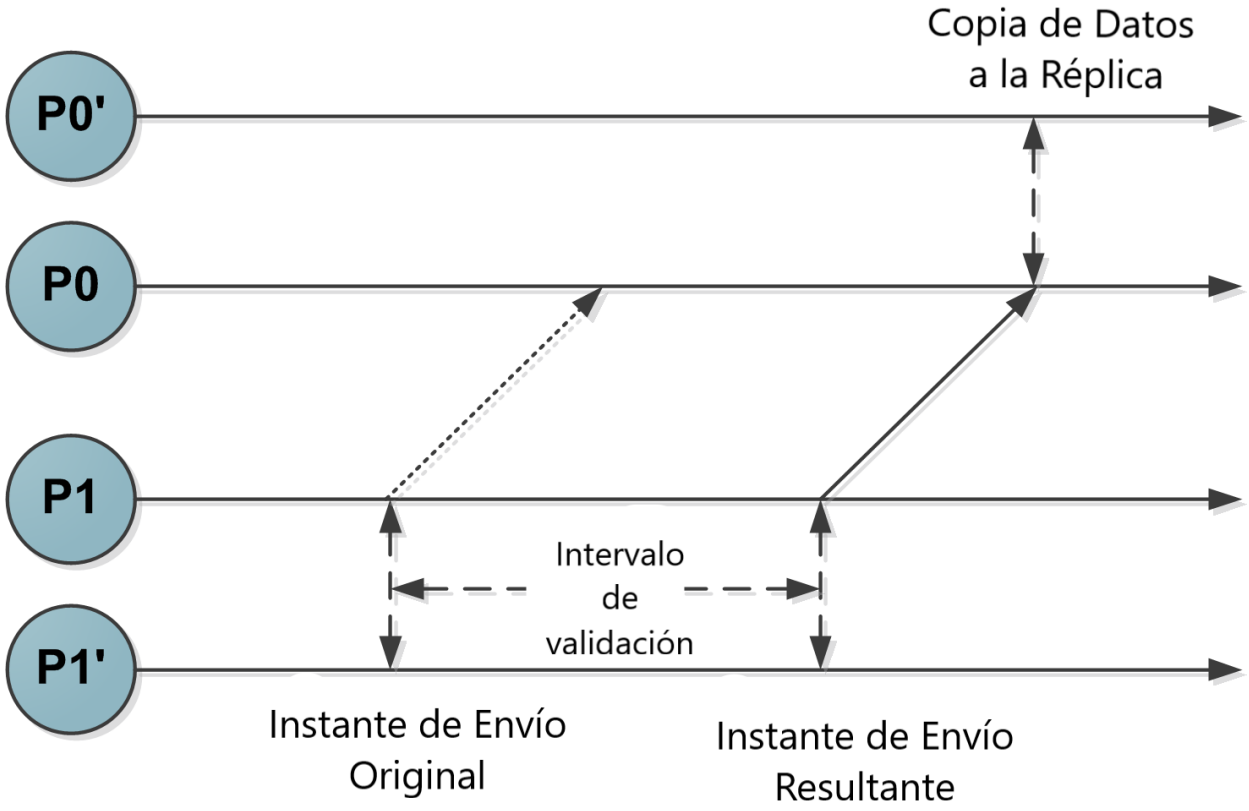

Figura 3.2: *Esquema de detección de SMCV*

El proceso receptor se detiene en el instante de la recepción, obtiene el mensaje y se mantiene en espera hasta que su réplica se sincronice en el mismo punto. Una vez allí, copia el contenido a la réplica (aprovechando también de la jerarquía de memoria) y ambas réplicas continúan con su cómputo. Si se asume que los errores de comunicación son detectados y corregidos por el software de la capa de red, los mensajes validados llegan a destino sin alteraciones. Al verificar los mensajes antes de enviarlos, es posible realizar sólo un envío, mientras que si fueran validados del lado del receptor, se requeriría el envío de dos copias, lo cual es perjudicial para el ancho de banda de la red.

Por último, sobre la finalización de la ejecución, los resultados obtenidos son verificados para detectar fallos que puedan haber ocurrido durante la fracción serie de la aplicación

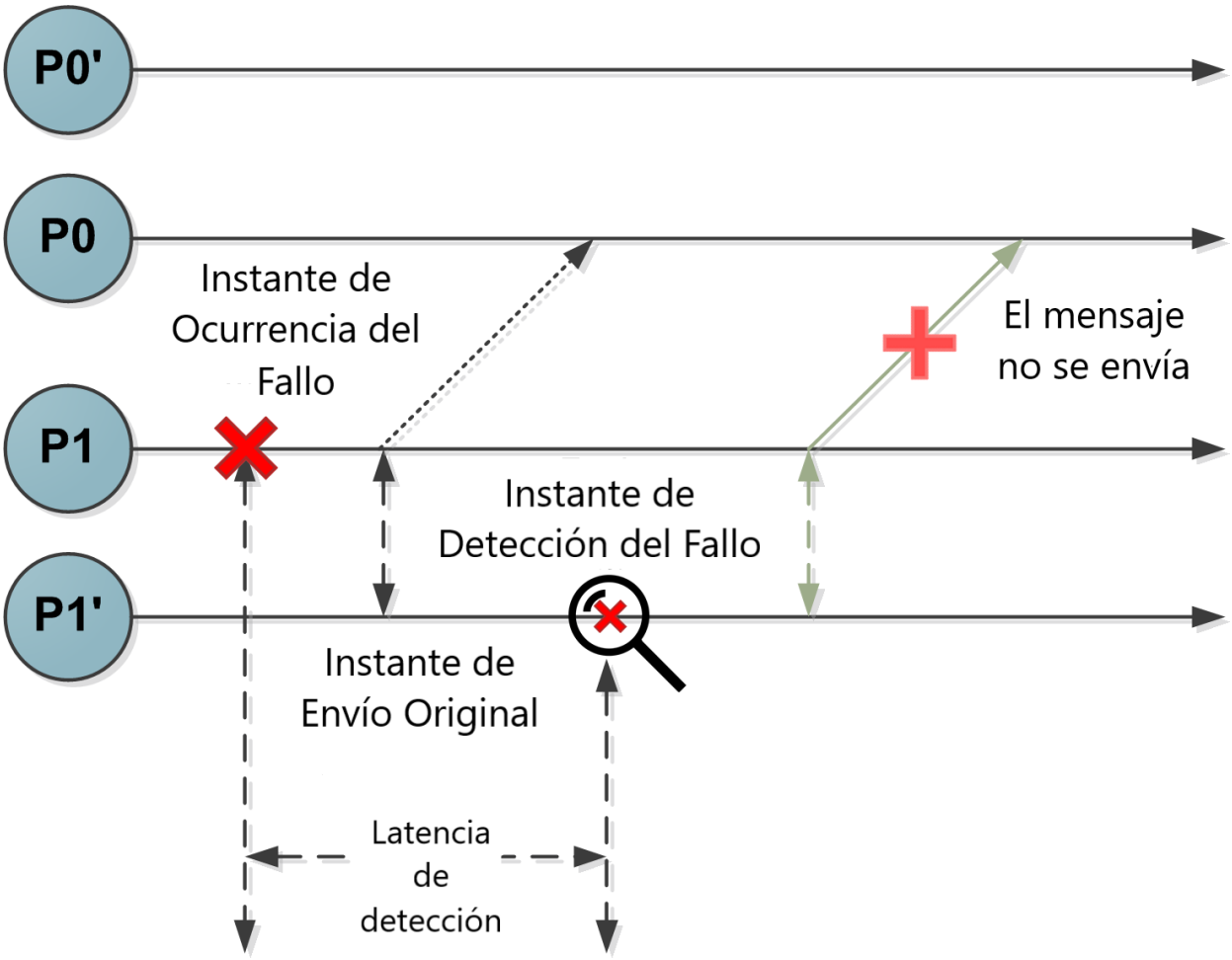

Figura 3.3: *Comportamiento de SMCV en presencia de un fallo transitorio*

(es decir, luego de la finalización de todas las comunicaciones) o que se hayan propagado localmente hasta el final.

Respecto de los *TOE*, la idea de su detección parte de la premisa de que, si una aplicación paralela se ejecuta sobre un multicore dedicado, los tiempos de ejecución de dos réplicas que realizan el mismo cómputo deben ser similares [94]. Por lo tanto, un retardo apreciable entre los tiempos de procesamiento de ambas réplicas puede considerarse como una separación entre los flujos de ejecución de ambas réplicas, posiblemente a causa de un fallo silencioso. Por lo tanto, el mecanismo de detección de *TOE* consiste en asignar un lapso máximo para que las réplicas se sincronicen; si esto no ocurre al cabo de ese lapso, *SMCV* reporta un

time − out y conduce a una parada segura. Ideas similares a esta han sido adoptadas con posterioridad en otros trabajos. Por ejemplo, en [151], se plantea que, como en la ejecución de aplicaciones MPI suele haber operaciones frecuentes de intercambio de mensajes, esos mensajes son tratados como "latidos" (*heartbeats*), de modo de que si no hay operaciones de paso de mensajes en un proceso específico por un lapso de tiempo considerable, se sospecha de la ocurrencia de un error.

Un aspecto importante a destacar es que el lapso pasado el cual se asume la detección de un fallo es configurable. No existe un valor óptimo, sino que debe ser determinado dependiendo de la necesidad de la aplicación particular. La configuración de este lapso debe ser cuidadosa, tomando en cuenta lo normalmente esperable de la aplicación. Si el valor es demasiado alto, transcurrirá mucho tiempo antes de la detección, aumentando innecesariamente la latencia. En tanto, si es demasiado bajo, una pequeña asimetría en los tiempos de cómputo resultaría en la detección de un falso positivo. El mecanismo de *SMCV* está pensado, básicamente, para detectar un error en el caso de que uno de los procesos ingrese en un bucle infinito.

Los pasos requeridos para el funcionamiento del mecanismo de detección, es decir, la replicación de procesos, la sincronización entre los *threads* redundantes, la comparación de contenidos de mensajes antes del envío, la copia de los mensajes en la recepción y la verificación final de los resultados, son la causa del *overhead* que afecta al tiempo total de ejecución.

SMCV alcanza un compromiso respecto del intervalo de validación. El *overhead* implicado por el mecanismo de detección se minimizaría si los resultados se comparan sólo al final, pero una gran cantidad de cómputo resultaría inútil a causa de la elevada latencia de detección. En el otro extremo, si la frecuencia de validación de resultados parciales es alta, el *overhead* introducido aumenta pero el fallo es detectado rápidamente, produciendo que mayor cantidad de cómputo sea aprovechable. Existe evidencia de que, dependiendo de la relación de cómputo a comunicación de una aplicación particular y del tamaño del *workload*,

el *overhead* puede variar significativamente [92]. Como consecuencia, la combinación entre la validación de los mensajes con la comparación final de resultados apunta a detectar todos los fallos y obtener un sistema confiable introduciendo un *overhead* razonable. Incluso, el mecanismo de detección de SEDAR podría ser adaptado a un modelo de replicación parcial, de forma similar a [50]. Sin embargo, se debe realizar trabajo extra (basado en el conocimiento de la aplicación, debido a que no es un procedimiento general) para identificar las partes críticas que requieren ser replicadas. Por lo tanto, SEDAR podría desactivarse en zonas no-críticas. Incluso, la replicación parcial no se aprovecharía desde el punto de vista de la utilización de recursos, debido a que los cores ya han sido previamente asignados a las réplicas (se estén utilizando o no), aunque podría ser beneficioso respecto del consumo energético.

3.4. Comportamiento frente a fallos

En esta sección se formaliza completamente el comportamiento de la metodología de detección *SMCV*. En la Figura 3.4 se muestra un diagrama de los estados posibles de la ejecución de una aplicación cuando no se implementa ninguna estrategia de detección, mientras que en la Figura 3.5 se observa el mismo diagrama pero cuando se aplica la metodología *SMCV* para la detección. Las elipses representan estados posibles, mientras que los arcos representan eventos que producen transiciones de un estado a otro. Por claridad, las transiciones aparecen numeradas en las figuras. En consecuencia, se muestra una descripción de cada evento posible que produce un cambio de estado.

1. El bit afectado no se utiliza
2. El bit afectado es utilizado por la aplicación
3. El bit alterado afecta datos monitoreados 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 al cabo de un tiempo máximo
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. Transcurso de tiempo
10. *SMCV* detecta el fallo al cabo de un tiempo y conduce a una parada segura

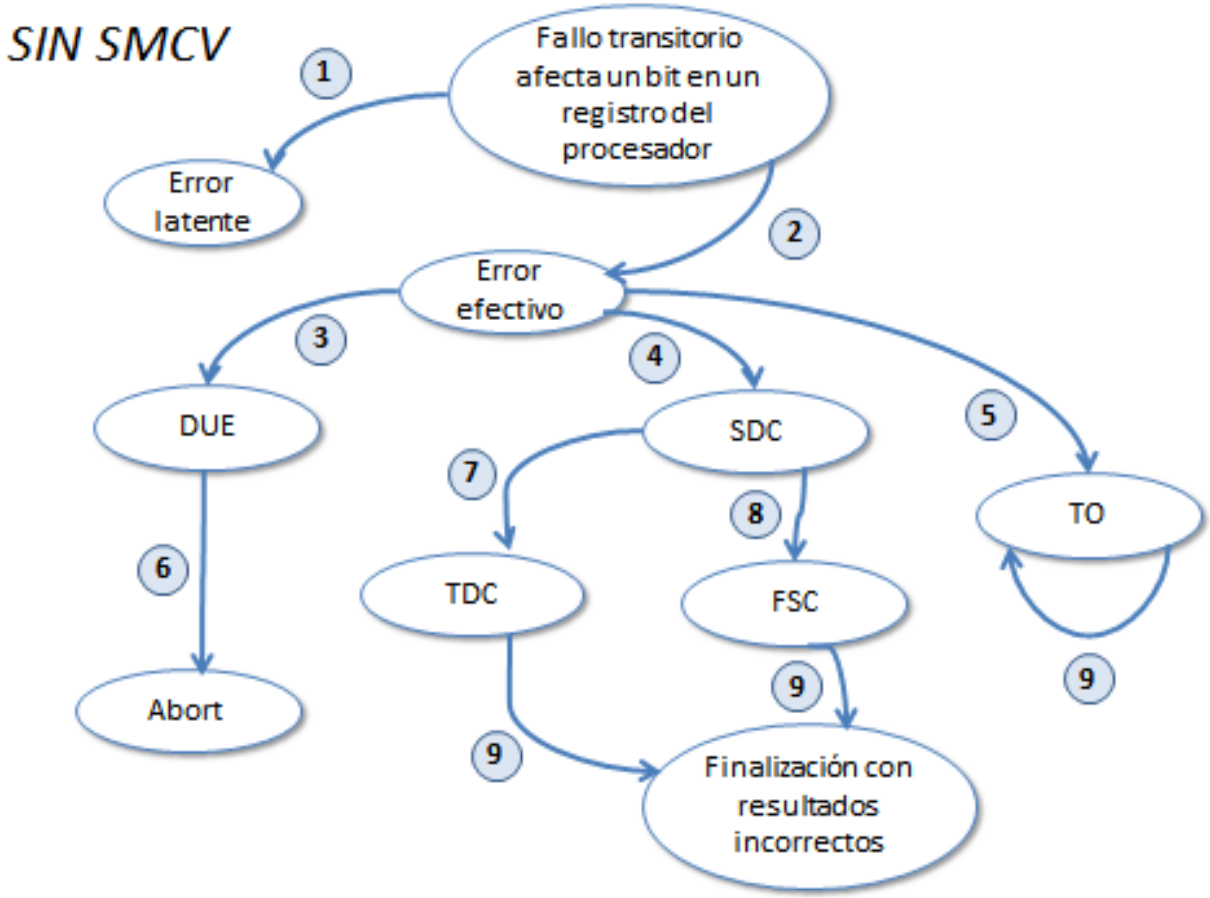

Figura 3.4: *Diagrama de estados de la ejecución sin estrategia de detección de fallos*

Un hecho que se debe resaltar es que la estrategia de detección propuesta es igualmente efectiva cuando ocurren múltiples fallos no relacionados durante la ejecución. La primera diferencia que se puede observar en los contenidos de un mensaje o en los resultados finales, causada por un error, conduce al sistema a una parada segura.

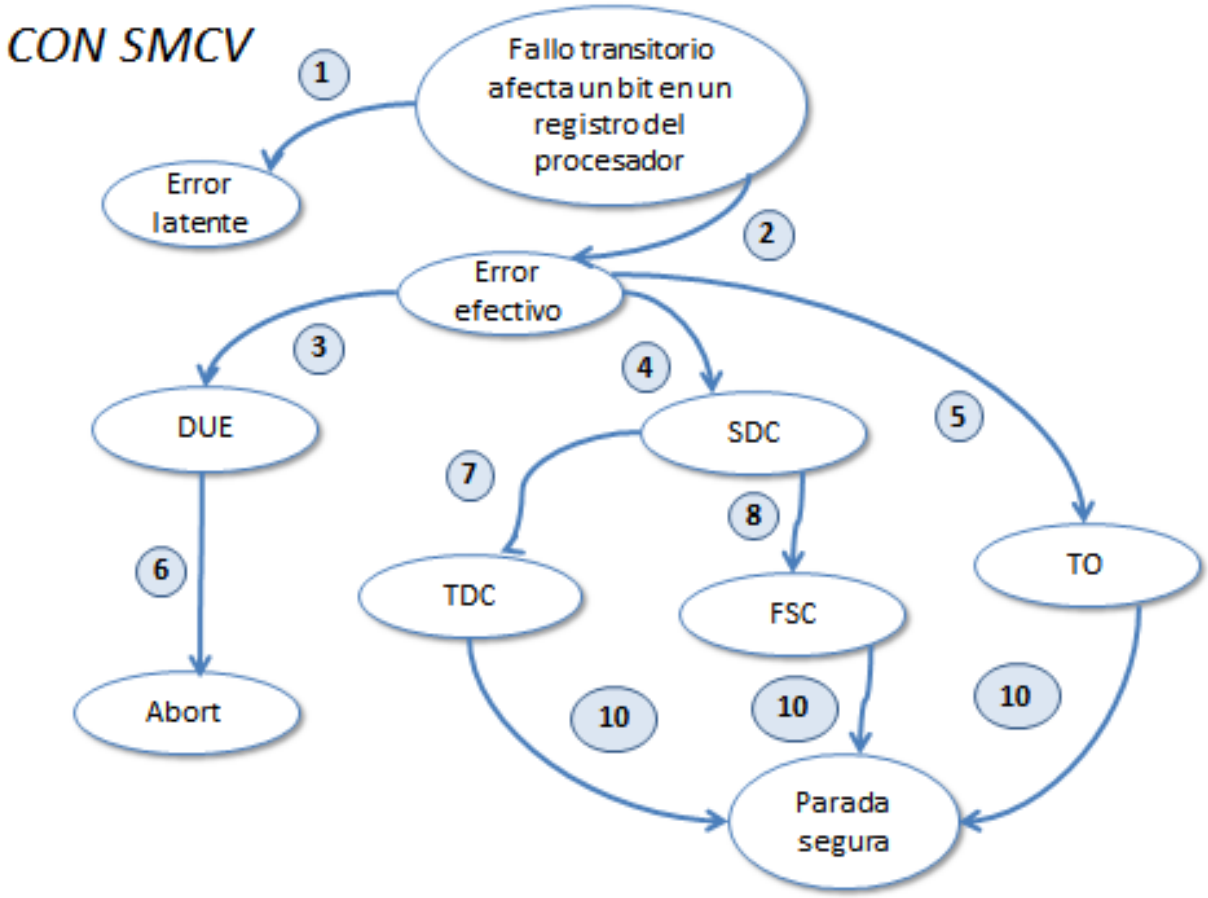

Figura 3.5: *Diagrama de estados de la ejecución aplicando la estrategia de detección SMCV*

3.5. Sobrecarga de operación

La carga de trabajo adicional está relacionada con la cantidad de cómputo agregada por la inclusión del mecanismo de detección de fallos. Esta métrica es útil para comparar la metodología *SMCV* con otras alternativas. Para tener una aproximación, se analiza una estrategia conservadora [112], en la que los procesos de la aplicación paralela también son duplicados en *threads* de la manera descripta, pero se verifican los resultados de todas las operaciones de escritura en memoria (en lugar de sólo los contenidos de los mensajes). Esta estrategia es capaz de detectar todos los fallos, pero a costa de un incremento significativo en la cantidad de cómputo. La sobrecarga de operación W_{WV} introducida por la técnica de

validación de escrituras está dada por:

$$W_{WV} = (S + Mk)(C_{sync} + C_{comp}) \tag{3.1}$$

En la Ecuación 3.1, S representa la cantidad de operaciones de escritura realizadas por la aplicación, sin contar aquellas que corresponden a los mensajes que envía. Se asume que la aplicación envía M mensajes de k elementos (en promedio) cada uno. C_{sync} y C_{comp} representan los costos de una operación de sincronización y de una operación de comparación respectivamente. Por lo tanto, el primer factor de la Ecuación 3.1 es la cantidad total de operaciones de escritura que realiza la aplicación. Si se verifican todas las escrituras, cada una de ellas conlleva una operación de sincronización y una de comparación.

En tanto, la sobrecarga de trabajo W_{MV} introducida por la validación de mensajes está dada por:

$$W_{MV} = M(C_{sync} + kC_{comp}) \tag{3.2}$$

En la Ecuación 3.2, para cada mensaje hay una única operación de sincronización y k operaciones de comparación (una para cada elemento del mensaje).

Por lo tanto, la relación entre la sobrecarga introducida por $SMCV$ y la estrategia que valida todas las operaciones de escritura viene dada por:

$$\frac{W_{MV}}{W_{WV}} = \frac{M(C_{sync} + kC_{comp})}{(S + Mk)(C_{sync} + C_{comp})} \tag{3.3}$$

El cociente de la Ecuación 3.3 es siempre un número menor que 1, lo que significa que la técnica de validación de mensajes introduce una cantidad menor de cómputo adicional que la de validación de todas las escrituras. El análisis anterior es válido para uno de los procesos que comunican sus resultados. En el caso de un proceso que realiza cómputo secuencial, se debe agregar la sobrecarga debida a la comparación de los resultados finales; sin embargo, ésta es la misma para ambas técnicas. Por lo tanto, el análisis es suficientemente general.

A partir de esto, se puede apreciar que *SMCV* es una técnica más liviana, que agrega una sobrecarga de operación reducida frente a estrategias más conservadoras que detectan fallos que no tienen impacto en los resultados de la aplicación.

3.6. Esfera de Replicación (*SoR*)

La esfera de replicación [117] (*SoR*, *Sphere of Replication*) 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. El concepto de *SoR* abstrae la redundancia física de un sistema con *lockstepping* y la redundancia lógica de un procesador con hilos redundantes en el hardware. Todos los datos que ingresan en la *SoR* son replicados y toda la ejecución dentro del ámbito de la *SoR* es redundante de alguna forma. Antes de abandonar la *SoR*, todos los datos de salida son comparados para asegurar su corrección. Toda la ejecución por fuera de la *SoR* no está cubierta por la técnica particular de detección y debe ser protegida de otra manera (por ejemplo la utilización de ECCs). Así, los fallos quedan confinados dentro de la frontera de la *SoR* y son detectados en los datos que dejan la *SoR*. En la Figura 3.6 se muestra gráficamente el concepto de *SoR*. La correcta identificación de la *SoR* es útil para determinar el conjunto de mecanismos de replicación y comparación necesarios y el alcance de la cobertura frente a fallos.

El concepto original de *SoR* se utilizó para definir los límites de la fiabilidad en diseños de hardware redundante, es decir, para modelos de detección centrados en el hardware. Estos modelos ven al sistema como una colección de componentes que deben ser protegidos de los fallos transitorios. La *SoR* está situada alrededor de las unidades de hardware especificadas. Todos los componentes dentro de la SoR están cubiertos por la ejecución redundante. Los valores que ingresan y salen de la *SoR* requieren de replicación y comparación, respectivamente. Sin embargo, resulta incómoda la aplicación de una *SoR* centrada en el hardware para las propuestas de detección implementadas puramente en software. A pesar de esto, algunas soluciones que utilizan el compilador para insertar instrucciones redundantes han

intentado imitar una *SoR* centrada en el hardware. Por ejemplo, *SWIFT* [119] coloca su *SoR* alrededor del procesador, dejando la memoria afuera de ella, ya que cuenta con esquemas de protección por hardware, como los bits de paridad y los *ECC*s. Esto tiene la ventaja de utilizar la mitad de la memoria y de reducir a la mitad la cantidad de escrituras.

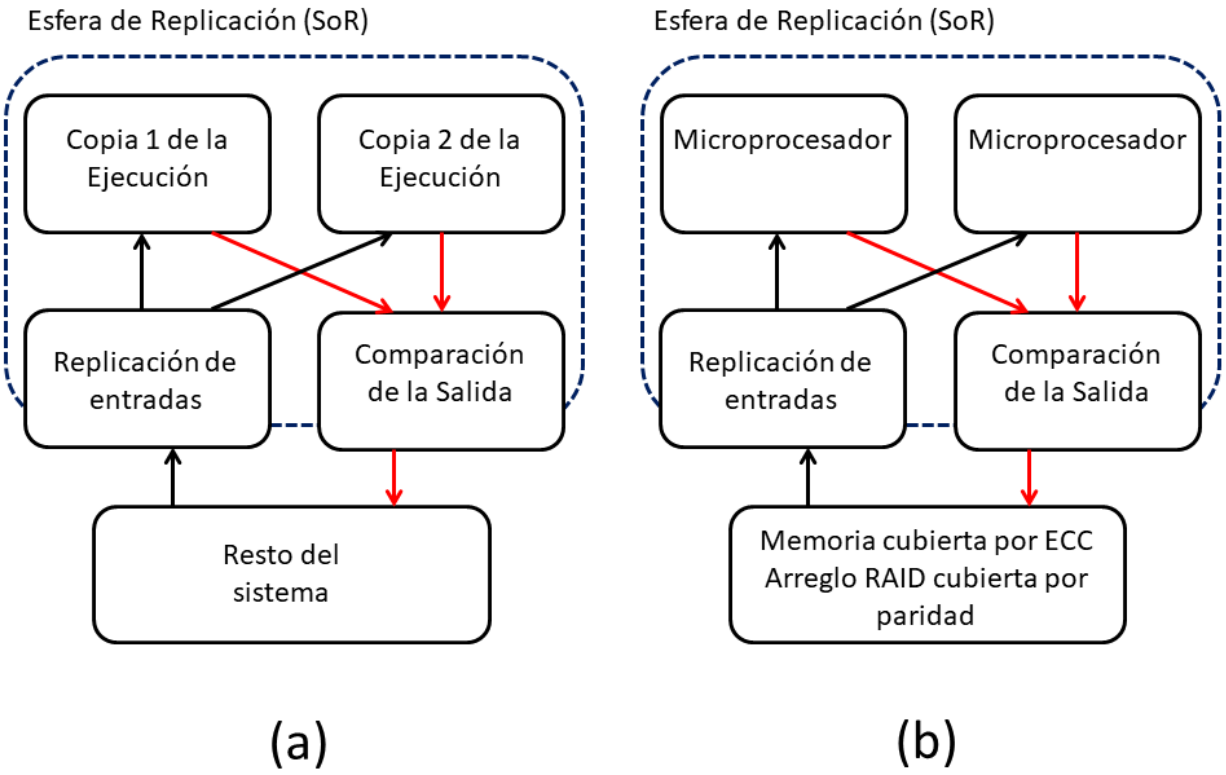

Figura 3.6: *Ejemplos de Esferas de Replicación (a) General (b) Del sistema Compaq Non-Stop Himalaya*

Sin embargo, SWIFT no puede controlar la duplicación a nivel del hardware, sino que sólo duplica las instrucciones a ejecutar. Cada lectura desde la memoria se realiza dos veces para replicar las entradas y todo el cómputo se realiza dos veces sobre las entradas replicadas. Se comparan los datos de salida antes de ejecutar cada instrucción de escritura en memoria para verificar su corrección. Esta solución funciona porque se puede emular la redundancia del procesador con redundancia de instrucciones. Sin embargo, otras *SoR* centradas en hardware (por ejemplo, una implementada alrededor de las cachés) no se pueden emular en

software. En general, este inconveniente se produce en las estrategias de detección que se implementan por software pero cuyo enfoque está centrado en la protección del hardware frente a los fallos.

El paradigma de detección de fallos centrado en el software ve al sistema como un conjunto de capas de software que deben ejecutarse correctamente, por lo que debe utilizar una *SoR* ubicada alrededor de estas capas de software. El software permite analizar los fallos desde una perspectiva más amplia, determinando sus efectos sobre la ejecución de las aplicaciones. PLR [133], como técnica implementada en software, define su *SoR* en términos de software para proveer límites más naturales para la detección. Una *SoR* centrada en el software pone de manifiesto el hecho de que, aunque los fallos afectan al hardware, los únicos que tienen importancia son los que inciden sobre la corrección del programa. De esta forma, los fallos que permanecen latentes se pueden ignorar sin correr ningún riesgo. Por lo tanto, un sistema de detección (sin recuperación) con una *SoR* centrada en software se protege de los fallos que se propagarían como salidas incorrectas de las aplicaciones (DUEs). En tanto, un sistema centrado en software con detección y recuperación no invocará al mecanismo de recuperación para los fallos que no afecten la corrección del software. Una *SoR* centrada en software actúa de la misma forma que una centrada en hardware, es decir: todas las entradas son replicadas, la ejecución dentro de los límites de la *SoR* es redundante, y todos los datos que salen de ella son verificados. La *SoR* utilizada por PLR está ubicada alrededor del espacio de direcciones de las aplicaciones del usuario y las bibliotecas compartidas asociadas a ellas, proveyendo redundancia a nivel de procesos. Toda la ejecución que se realiza en espacio de usuario es duplicada, y los fallos se detectan sólo si resultan en datos incorrectos saliendo del espacio de usuario, lo que permite ignorar la gran mayoría de los fallos benignos. PLR replica el código de la aplicación, de las bibliotecas, las variables globales, la *heap*, la pila, los descriptores de archivos y toda la información asociada al proceso. Todo lo que está por fuera de los límites de la *SoR* debe ser protegido por otros medios. Sin embargo, el modelo de detección centrado en el software tiene la desventaja de retardar la detección

de los fallos hasta que se produce certeza del error por ejecución incorrecta del programa o por datos inválidos que salen de la *SoR*. La detección retardada implica que un fallo puede permanecer latente durante la ejecución durante un período indeterminado de tiempo. Una caracterización de la propagación de los fallos permitiría explorar opciones para limitar los lapsos en los que los fallos permanecen sin ser detectados.

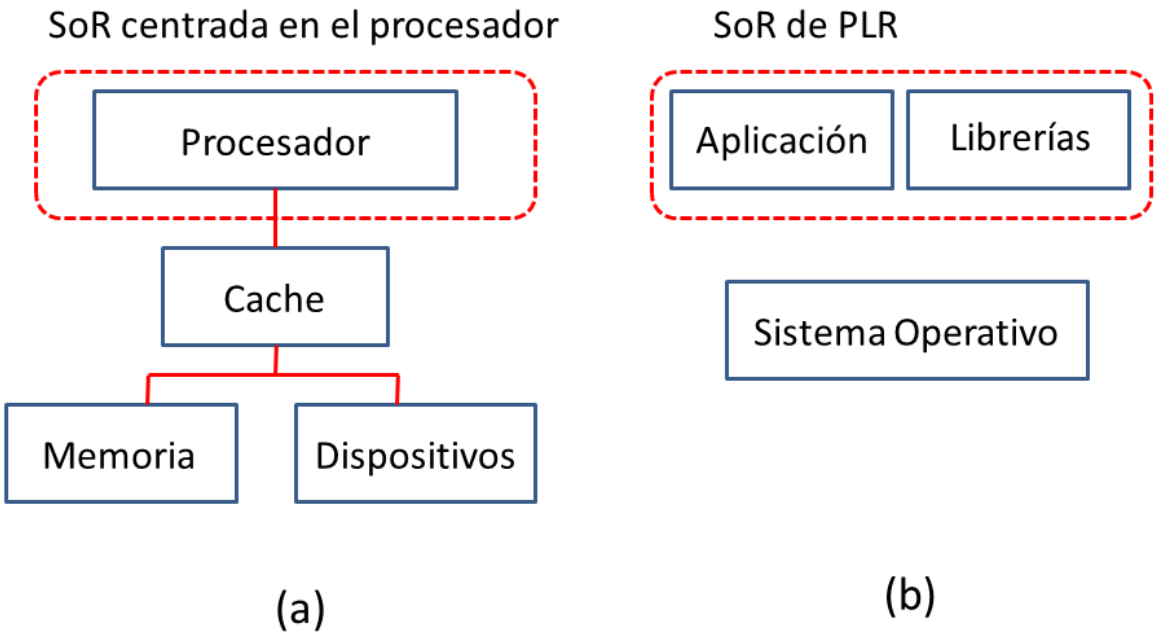

Figura 3.7: *(a) Esfera de replicación de SWIFT (centrada en el hardware) (b): Esfera de replicación de PLR (centrada en el software)*

En la Figura 3.7 se observan las *SoR* de SWIFT y PLR, respectivamente.

SMCV es una técnica de software puro, y por lo tanto se adopta una *SoR* centrada en software, de manera similar a *PLR* ([133]), de modo que se enfoca sólo en los fallos que repercuten sobre la ejecución de la aplicación. El objetivo de *SMCV* es detectar fallos que ocurren durante el cómputo sobre los datos que se manipulan dentro del procesador, específicamente en los registros, que como se mencionó en las en el Capítulo 1, constituyen la parte más vulnerable del sistema de cómputo, debido a la dificultad de implementar

mecanismos de protección por hardware para ellos. Esto implica que la memoria se encuentra por fuera de la *SoR* de *SMCV*. Como se explicó en la Sección 3.3, *SMCV* replica en un *thread* todo el cómputo realizado por cada proceso de la aplicación paralela. Cada *thread* opera sobre una copia local de los datos, es decir, se genera una copia del subconjunto de los datos de entrada para que el *thread* redundante realice el cómputo de forma independiente del proceso al cual replica. Por lo tanto, la *SoR* se encuentra alrededor de la aplicación del usuario y sus datos. No incluye al sistema operativo ni a la biblioteca de comunicaciones.

3.7. Vulnerabilidades

Todas las técnicas de tolerancia a fallos poseen vulnerabilidades, es decir, circunstancias en las cuales los fallos que influyen efectivamente sobre la ejecución pasan desapercibidos. Las características de diseño de una estrategia de detección, sumadas a las pruebas a las que son sometidos los sistemas (usualmente mediante inyección de fallos) deben conducir a la descripción explícita de esas vulnerabilidades, es decir, los casos en que la técnica no será capaz de detectar los fallos. Las vulnerabilidades están normalmente asociadas con fallos que afectan al mecanismo de detección propiamente dicho. Dado que la detección se realiza mediante comparaciones, los sitios en los que se realizan estas comparaciones constituyen puntos de falla centralizados. Sin embargo, a pesar de no ser completamente fiable, la redundancia parcial [60, 139] en general es suficiente para mejorar significativamente la fiabilidad y cubrir los requerimientos. A modo de ejemplo, la técnica *SWIFT* tiene dos vulnerabilidades principales. Debido a que la redundancia se obtiene únicamente por medio de duplicar instrucciones, puede existir un retardo entre la validación y el momento de utilización de los valores de los registros validados. Por lo tanto, cualquier interferencia que ocurra durante ese lapso puede corromper silenciosamente el estado. Los *bit-flips* que ocurren en las direcciones de escritura y en registros de datos no se detectan. Esto puede causar ejecución incorrecta de los programas debido a que implica que escrituras incorrectas salgan de la *SoR*, ya sea por valores de escritura incorrectos o por direcciones de escritura

incorrectas. El segundo punto de falla ocurre si el código de operación de una instrucción es cambiado al de una escritura a causa de un fallo transitorio. Estas escrituras no están protegidas debido a que el compilador no ve dichas instrucciones. La escritura resultante se ejecutará libremente y el valor almacenado por ella abandonará la *SoR*. Debido a que los llamados a funciones afectan la salida de los programas, en SWIFT los llamados a funciones son puntos de sincronización, ya que valores inválidos de parámetros pueden resultar en ejecuciones incorrectas. Antes de cada llamado, todos los operandos de entrada se verifican con sus copias redundantes de la forma habitual. Si no hay diferencia, las versiones originales se pasan como parámetros a la función. Al comienzo de la función, los parámetros deben ser duplicados nuevamente, debido a que la función también se computa por duplicado; esto es similar a las copia de contenidos recibidos de mensajes que realiza *SMCV*. Al terminar, sólo se devuelve una versión de los valores de retorno. Estos valores deben duplicarse para el código redundante restante por fuera de la función. Por supuesto, todo esto produce overhead adicional, pero también introduce puntos de vulnerabilidad. Como sólo una versión de los parámetros se le pasa a la función, un fallo que afecta a un parámetro luego de la verificación realizada por el programa que llama a la función, pero antes de la duplicación que se realiza en la función llamada no será detectado.

En cambio, PLR tiene asociadas sus vulnerabilidades a los fallos que ocurren durante la ejecución de la instrumentación introducida por PLR. Además, si un fallo causa un salto erróneo al código de PLR, puede obtenerse una conducta indefinida. Finalmente, al igual que *SMCV*, PLR no protege al sistema operativo, por lo que cualquier fallo que afecte la ejecución del sistema operativo resultará en un error.

Respecto de las vulnerabilidades de *SMCV*, la primera de ellas está relacionada con los fallos múltiples que ocurren en ciertas combinaciones muy particulares: si ocurren dos fallos que afectan al mismo bit del mismo dato en ambas réplicas, ese fallo no será detectado en la comparación. El mismo efecto se obtiene si ocurre un fallo durante una verificación que oculta o anula el efecto de un fallo anterior.

Como se mencionó anteriormente, debido a que la memoria se asume protegida eficientemente por mecanismos como los ECCs, se encuentra por fuera de la SoR. Por lo tanto, un fallo que afecta a una variable compartida por varios procesos (es decir, un fallo de memoria) no es detectable por $SMCV$. En este caso, el proceso y su réplica leen y utilizan el mismo valor erróneo para sus cálculos, por lo que, si la ejecución subsiguiente es correcta y libre de fallos, pasa desapercibido para $SMCV$, que no detecta ningún error.

De manera similar al caso de PLR [133], otra vulnerabilidad de $SMCV$ está asociada a la ejecución del mecanismo de detección. $SMCV$ minimiza los retardos entre la comprobación y la utilización de los valores validados, debido a que la verificación se realiza justo antes de utilizar los datos para enviar un mensaje. Una vez que los datos que se van a enviar están en el buffer de salida ya se encuentran por fuera de la SoR.

En tanto, el procedimiento mismo de comprobación de contenidos a enviar resulta un punto centralizado de falla. Un análisis de este aspecto arroja como resultado que las posibles combinaciones de salida de la ejecucion y salida de la comprobacion que se pueden producir son las siguientes:

i) Ejecución correcta, comprobación correcta: en este caso no se ha producido hasta el momento ningún fallo durante la ejecución (salvo quizás algún fallo que aún permanezca latente). Al no detectarse un error, la ejecución continúa normalmente.

ii) Ejecución correcta, comprobación incorrecta: en este caso ha ocurrido una falla durante la comparación, por lo que se detecta un falso positivo. La aplicación finaliza en una parada segura cuando en realidad no ha ocurrido ningún fallo en ejecución, sino que ha sido introducido por el propio mecanismo de detección. Esta es una vulnerabilidad clara, que puede mitigarse si se duplica la comparación. La técnica $SMCV$, en su estado actual, no contempla esta duplicación.

iii) Ejecución incorrecta, comprobación incorrecta: este caso corresponde a la operación normal, en la cual el fallo ocurrido durante la ejecución es detectado en el momento de la comparación.

iv) Ejecución incorrecta, comprobación correcta: en esta situación, un fallo ocurrido durante la ejecución resulta oculto tras la comprobación, debido a un segundo fallo que "compensa" o "anula" al primero. Como se mencionó en esta misma sección, *SMCV* no puede lidiar con un doble fallo de este tipo, sino que sólo soporta fallos simples no relacionados. De todas formas, la probabilidad de que ocurran dos fallos que se combinen de esa forma tan particular es extremadamente baja, como se comenta en la Sección 3.8.

Es importante tomar en consideración el hecho de que *SMCV* es capaz de detectar como *TOE*s otros fallos que representarían vulnerabilidades si no se contara con dicho mecanismo. Si el código de una instrucción fuera modificado por un fallo, de modo que la instrucción resultante fuera el envío de un mensaje, o si ocurre un fallo durante la ejecución del código de la herramienta de detección, o si un fallo produjera que se ejecute código de la herramienta en un momento erróneo, ambas réplicas separarán sus flujos de ejecución y, ante el envío de un mensaje por parte de una de ellas, no se sincronizarán adecuadamente, por lo que el fallo se detectará al transcurrir un lapso mayor al determinado.

3.8. Fallos múltiples

En la Sección 3.1 se mencionó el hecho de que la técnica *SMCV* es capaz de detectar fallos si se asume el modelo *SEU*, en el que ocurre un único $bit - flip$ a lo largo de la ejecución. Sin embargo, no son tan efectivas si ocurren fallos que afectan a múltiples bits. En particular, se han mencionado dos casos que resultan en vulnerabilidades. Los autores de [119] han realizdo un análisis sobre los fallos *multibit*, en el cual se describen las dos únicas situaciones en las que los fallos múltiples se pueden combinar para causar inconvenientes. La primera de ellas es aquella en la que el mismo bit resulta alterado tanto en el cómputo original como en el redundante, por lo que la comparación resulta correcta y el fallo no se detecta. La segunda se da cuando ocurre un fallo que afecta a uno de ambos hilos de cómputo redundante, y el resultado de la operación de verificación también es alterado, por lo que no se ejecuta el código de error. Sin embargo, estas combinaciones tienen una probabilidad

de ocurrencia sumamente baja, por lo que pueden ser ignorados sin riesgos serios. Todas las demás combinaciones de fallos múltiples son detectadas como fallos simples, ni bien se realice la primera comprobación que contenga al menos una diferencia. En [119] se realiza una estimación de la probabilidad de que ocurran las combinaciones de fallos *multibit* que escaparían de los mecanismos de detección. Si se utiliza un modelo en el que pueden ocurrir dos fallos (uno en el cómputo original y otro en el redundante), y se asume que el mismo fallo debe ocurrir en el mismo bit de la misma instrucción de ambas ejecuciones redundantes para no ser detectado, la probabilidad viene dada por la Ecuación 3.4:

$$P(error_{redundante}|error_{original}) = 1/64 * 1/(N_{instr}) \tag{3.4}$$

Esta es simplemente la probabilidad de que resulte elegida la misma instrucción (en una inyección de dos fallos distribuidos de forma uniforme y aleatoria) combinada con la probabilidad de que resulte elegido el mismo bit en ambas ejecuciones (asumiendo registros de 64 bits). En el caso de prueba del trabajo, en el que se usa la *suite* de *benchmarks* SPEC, la cantidad promedio de instrucciones dinámicas está en el orden de 10^9 a 10^11, por lo que la probabilidad de que ocurra este caso particular de fallo doble es cercana a uno en un trillón. El otro caso que resultaría en una vulnerabilidad es que un bit resulte alterado, afectando sólo a uno de los hilos de cómputo redundante, y otro bit resulte invertido durante la operación de comprobación, de forma de anular la consecuencia del primero. Si se asume que se realiza una única comparación para cada fallo posible, la probabilidad de que este fallo doble, que no puede ser detectado, viene dada por la Ecuación 3.5:

$$P(error_{redundante}|error_{original}) = 1/(N_{instr}) \tag{3.5}$$

Para los *benchmarks* SPEC, esta probabilidad es cercana a uno en 10 billones en promedio. Esta es una sobreestimación grosera, debido a que se asume que hay una única verificación para cada fallo, cuando en la práctica pueden existir varias comprobaciones sobre un valor erróneo que sea utilizado en el cálculo de distintas cantidades a almacenar o direcciones de saltos.

Claramente, *SMCV* no es afectado de esta misma forma. Las situaciones análogas, desde el punto de vista de nuestra propuesta, consisten en que los contenidos de mensajes, calculados por ambas réplicas, experimenten el mismo fallo; o que una falla durante la comparación "compense" a un fallo en uno de los valores a comprobar. Como se ha dicho, es posible ignorar estas combinaciones sin corres riesgos considerables.

3.9. Memoria compartida

Cuando múltiples procesos se comunican utilizando memoria compartida, el compilador no puede forzar un ordenamiento de las lecturas y las escrituras entre los procesos. Por lo tanto, las dos lecturas de una lectura duplicada no devuelven necesariamente el mismo valor, debido a que siempre existe la posibilidad de que se intercalen escrituras de otros procesos. La situación es similar cuando ocurre una interrupción o una excepción entre el par de lecturas de una duplicación, y el manejador de interrupciones o excepciones modifica el contenido en la dirección de la lectura. Estas circunstancias no afectan la cobertura frente a fallos de ningún sistema, pero son capaces de incrementar la cantidad de fallos detectados que no hubieran causado ninguna avería. Para solucionar esto, se puede recurrir a alguna técnica (basada en hardware) de duplicación de valores leídos, como las que se utilizan en máquinas *RMT* [99], y adaptarla a los esquemas basados en software. Por supuesto, esto conlleva las desventajas y los costos asociados a la introducción estructuras de hardware redundantes.

3.10. Resumen de las características de la metodología

A modo de sumario, se listan las principales características que proporciona la metodología *SMCV*:

- Cada proceso y su réplica son validados localmente, por lo que el mecanismo está totalmente distribuido entre todos los procesos de la aplicación.

- Evita la propagación de errores hacia los demás procesos. Además, detecta los errores en la fracción serie mediante la verificación de los resultados finales.

- Comparada con una estrategia de detección conservadora (diseñada para programas secuenciales), que verifica el valor de cada operación de escritura antes de realizarla para preservar la salida del programa, *SMCV* introduce una sobrecarga de operación reducida.

- Relacionado, con el ítem anterior, *SMCV* introduce un bajo *overhead* respecto del tiempo de ejecución, debido a que sólo se agrega una operación de comparación para cada byte de cada comunicación saliente y del resultado final (el costo de una operación de comparación es menor que el de una comunicación). Este aspecto se tratará con mayor detalle en el Capítulo 6

- El *overhead* total introducido es causado por la replicación de procesos, la sincronización entre los *threads* redundantes, la comparación de contenidos de mensajes antes del envío, la copia de los mensajes en la recepción y la verificación final de los resultados.

- Ni bien se detecta un fallo, se detiene la aplicación, permitiendo relanzar su ejecución. No es necesaria la costosa espera a que la aplicación finalice con resultado incorrecto para re-ejecutar, por lo que *SMCV* reduce el tiempo de recuperación. Esto, además de mejorar la fiabilidad, conlleva una ganancia en términos de tiempo, lo cual se vuelve particularmente significativo cuando se trata de aplicaciones científicas que pueden ejecutarse durante varios días.

- *SMCV* incrementa la fiabilidad del sistema, entendida como la cantidad de veces que la aplicación finaliza correctamente, debido a su capacidad de detectar los fallos silenciosos.

- Logra un compromiso entre latencia de detección, la sobrecarga de operación y los recursos involucrados. *SMCV* no necesariamente mejora la latencia de detección, debido a que, en general, no realiza ninguna verificación en el instante en que el valor corrompido se utiliza por primera vez. Esto pospone la detección hasta el momento en que el dato alterado forma parte del contenido de un mensaje que se va a enviar. Sin embargo, esto conlleva una carga de trabajo menor que la de validar todas las escrituras (baja latencia pero alta sobrecarga) y hace una mejor utilización de los recursos que una única verificación al final (es decir, duplicar todo el cómputo para detectar sólo al final, lo que implica baja sobrecarga pero elevada latencia). Cuanto más frecuente es la comunicación entre los procesos, la latencia es menor y la sobrecarga es mayor.

- *SMCV* es igualmente efectiva cuando ocurren múltiples fallos no relacionados durante la ejecución.

Debido a que *SMCV* es una metodología basada en replicación de procesos, *a priori* es capaz de proporcionar sólo detección de fallos: se requeriría de redundancia triple para poder corregir. En el siguiente capítulo se describen las posibles estrategias para lograr recuperación sin necesidad de triplicación.

Capítulo 4

Recuperación Automática

Resumen

En el capítulo anterior se examinó *SMCV*, la estrategia de detección de SEDAR. Para completar una metodología capaz de tolerar fallos transitorios, se requiere un estado seguro, desde el cual pueda recuperarse una ejecución errónea (es decir, una forma de proteger esa ejecución), y un mecanismo automático para recuperar la aplicación, permitiéndole concluir con resultados finales válidos. La metodología SEDAR propone dos maneras alternativas para lograr esto: la primera de ellas se basa en la utilización de una cadena de múltiples *checkpoints* de capa de sistema, y la segunda se basa en la utilización de un único *checkpoint* seguro de capa de aplicación. En este capítulo se presentan estas dos estrategias, desarrolladas para recuperar automáticamente una ejecución de los fallos transitorios que puedan ocurrir durante ella, en un sistema de HPC.

4.1. Introducción

En general, la tolerancia a fallos incluye las fases de detección, protección y recuperación. La fase de protección consiste en proporcionar un estado, libre de fallo, desde el que se pueda recuperar la ejecución luego de un error.

Como se mencionó en la Sección 2.2, en el contexto de los fallos transitorios que conducen a errores silenciosos, la duplicación es suficiente para detectar los fallos, pero para lograr

la recuperación se requiere de redundancia triple, que, en combinación con un mecanismo de votación, consigue recuperar correctamente la ejecución. Esta técnica de *TMR* (*Triple Modular Redundancy*) [149] es la solución estándar para sistemas críticos que requieren alta fiabilidad, como los dispositivos embebidos en sistemas aeronaúticos. El cómputo es ejecutado tres veces, y se vota la mayoría para seleccionar el resultado correcto entre los tres disponibles: si dos o más de ellos coinciden, se asumen como correctos, debido a que la probabilidad de que dos o más errores conduzcan al mismo resultado incorrecto es tan baja que puede ser ignorada. A pesar de que vale la pena usarla en entornos en los cuales la consecuencia de un error puede ser catastrófica, la triplicación es sumamente costosa en términos de utilización de recursos [14].

La estrategia más ampliamente utilizada para tratar con errores que causan la detención del proceso, o incluso de todo el sistema ($fail-stop$) es la de *checkpointing* y *log* de eventos, mientras que para la recuperación ($restart$) se combina con la técnica de $rollback-recovery$, es decir, el retroceso hasta el estado seguro para retomar desde allí la ejecución (C/R) [48, 49]. Un *checkpoint* consiste en el almacenamiento del estado actual de la ejecución de una aplicación paralela determinada. El *checkpoint* es simplemente un archivo que incluye todos los resultados intermedios y los datos asociados, que se guardan en un medio de almacenamiento que no es afectado por errores; puede ser tanto la memoria de otro procesador, un disco local o remoto. Este archivo puede ser recuperado si un proceso experimenta un error posterior durante su ejecución, para restaurar el estado del programa a ese punto. En ese caso, la aplicación debe retroceder hasta el último estado que haya sido almacenado (es decir, al último *checkpoint*), o relanzar la ejecución desde el comienzo, si no se ha guardado ningún *checkpoint* previo. Por lo tanto, el archivo es leído desde el medio de almacenamiento en la fase de recuperación, y la ejecución se retoma desde ese punto, que, en el caso de los errores $fail - stop$, se asume como un estado seguro (es decir, como un error causa una caída, si el sistema sigue ejecutando, se asumen en este contexto que no hubo error, y por lo tanto el *checkpoint* representa un estado sin error). Un aspecto relevante es el intervalo

de *checkpoint*: si el *checkpoint* fue almacenado mucho antes de la aparición del error, es necesaria una gran cantidad de re-ejecución; en ese sentido, es conveniente una frecuencia mayor. Sin embargo, el proceso de *checkpointing* incurre en un *overhead* significativo, que conlleva un gasto de recursos innecesario si no ocurre ningún error. Por lo tanto, debe hallarse un $trade-off$, e incluso seleccionar cuidadosamente las tareas cuyo estado se va a almacenar [49].

A pesar de que la técnica de C/R constituye un estándar $de-facto$ para los errores $fail-stop$ (ver Sección 2.6, no hay una estrategia suficientemente general y que sea ampliamente utilizada para tratar con los errores silenciosos. Con éstos, el problema principal lo constituye la latencia de detección: a diferencia de los errores $fail-stop$, cuya detección es inmediata, un error silencioso es identificado sólo cuando los datos corrompidos son activados y/o conducen a un comportamiento inusual de la aplicación. La técnica de C/R asume que la detección del error es inmediata, y esto genera una dificultad para aplicarla a los errores silenciosos: si el fallo ocurre antes del último *checkpoint*, pero el error que provoca se detecta luego del *checkpoint*, ese *checkpoint* está corrompido y no puede utilizarse para restaurar el estado.

Básicamente, existen dos maneras de implementar la técnica de C/R:

- De forma explícita, lo cual requiere modificaciones al código de la aplicación paralela. En general, si el *checkpoint* es realizado de esta forma, entra en la categoría de *checkpointing* de capa de aplicación.
- De forma transparente, en el sentido de que el *checkpoint* puede ser almacenado de forma externa a la aplicación, independientemente del código mismo. En general, si el *checkpoint* es realizado de esta forma, entra en la categoría de *checkpointing* de capa de sistema.

En general, los requerimientos del sistema conducen a la decisión de cuál es la estrategia que se adapta mejor a ellos.

Dentro de la metodología SEDAR, nuestra propuesta consiste en integrar la estrategia *SMCV* de detección de fallos transitorios con una estrategia de C/R, que se utiliza para brindar protección y recuperación en el contexto de los fallos permanentes. Esto implica que no se requiere de TMR con mecanismo de votación para detectar y recuperar un entorno de un error silencioso. El mecanismo propuesto tiene la ventaja adicional (que no siempre es posible en el caso de los errores *fail-stop*) de no requerir reconfiguración del sistema ni migración de procesos a otros nodos de cómputo: como los fallos transitorios son, por definición, de corta duración, la re-ejecución necesaria para recuperar la aplicación puede lanzarse en el mismo core que ha experimentado el fallo original.

Para lograr la recuperación de la ejecución sin necesidad de triplicación, SEDAR incorpora dos mecanismos, uno de los cuales utiliza *checkpoints* de capa de sistema, mientras que el otro utiliza *checkpoints* de capa de aplicación. Estos mecanismos se describen en las dos secciones subsiguientes. Como siempre, el objetivo final es que las aplicaciones consigan llegar al final de su ejecución con resultados correctos.

4.2. Recuperación basada en múltiples *checkpoints* de capa de sistema

La primera opción que SEDAR propone para agregar un mecanismo de recuperación automática para fallos transitorios consiste en la construcción de una cadena de múltiples *checkpoints* coordinados y distribuidos que se almacenan por medio de una librería de *checkpointing* de capa de sistema. El requisito de guardar varios *checkpoints* es necesario debido a que, al guardar un *checkpoint* de capa de sistema, no sólo se almacena información sobre el estado de la aplicación, sino también información relativa al estado actual del sistema. Dado que el usuario no tiene control sobre la información que se guarda, estos *checkpoints* no pueden ser validados. Las pruebas preliminares que hemos realizado, relativas a este aspecto, arrojaron como resultado que dos *checkpoints* (almacenados por dos réplicas de un mismo proceso), exactamente en el mismo punto de la ejecución, y en ausencia de

fallos, generaron archivos con diferentes contenidos. Por lo tanto, una diferencia entre estos contenidos no está necesariamente vinculada a la ocurrencia de un error. En consecuencia, es imposible garantizar que un determinado *checkpoint* constituye un punto seguro para la recuperación, ya que no puede determinarse si realmente su contenido está libre de fallos (en ese momento) latentes, y que luego se efectivizarán como errores.

Como se mencionó en la Sección 4.1, los datos contenidos en un determinado *checkpoint* pueden haber sido alterados por la ocurrencia de un fallo transitorio anterior a su almacenamiento, que haya afectado a una de las réplicas cuyo estado fue salvado. Frente a la posterior detección de un error (y la imposibilidad de comprobar la fiabilidad del último *checkpoint* disponible), no es posible *a priori* determinar la factibilidad de la recuperación correcta desde ese punto. Por este motivo, los *checkpoints* previos al último no pueden descartarse, para que estén disponibles en caso de que, eventualmente, se requiera recuperar la ejecución desde alguno de ellos. En conclusión, múltiples *checkpoints* deben permanecer almacenados para garantizar la correcta recuperación [83].

En resumen, existen dos casos respecto de la recuperación:

1. El fallo ocurre, y el error que produce es detectado dentro del límite de un intervalo de *checkpoint*. En esta situación, se puede utilizar el último *checkpoint* disponible para restaurar la ejecución. Como caso particular, si un error es detectado cuando aún no se ha almacenado ningún *checkpoint*, la aplicación debe ser relanzada desde el comienzo.

2. La latencia de detección traspone el límite del intervalo de *checkpoint*. Como se explicó, esta situación tiene lugar cuando el fallo ocurre antes de almacenar un *checkpoint* pero la detección del error se produce después. En este escenario, el último *checkpoint* es inválido, y por lo tanto, el correspondiente intento de regresar a ese punto causa nuevamente la manifestación del mismo error. En consecuencia, se debe intentar recuperar desde el *checkpoint* previo. Extrapolando esta situación, el fallo podría atravesar una cantidad arbitraria de *checkpoints*, si la latencia de detección es muy alta, llegando eventualmente a requerir varias repeticiones de *rollback* para posibilitar la

recuperación [90].

Con esta estrategia, basada en múltiples *checkpoints*, es posible llegar al final de la ejecución, a costa de que a veces incluso llegue a hacer falta regresar hasta el principio. Principalmente se intenta garantizar la seguridad de los datos del usuario.

Estos comportamientos se esquematizan en las Figuras 4.1 y 4.2. El pseudocódigo del mecanismo de recuperación propuesto se muestra como Algoritmo 1.

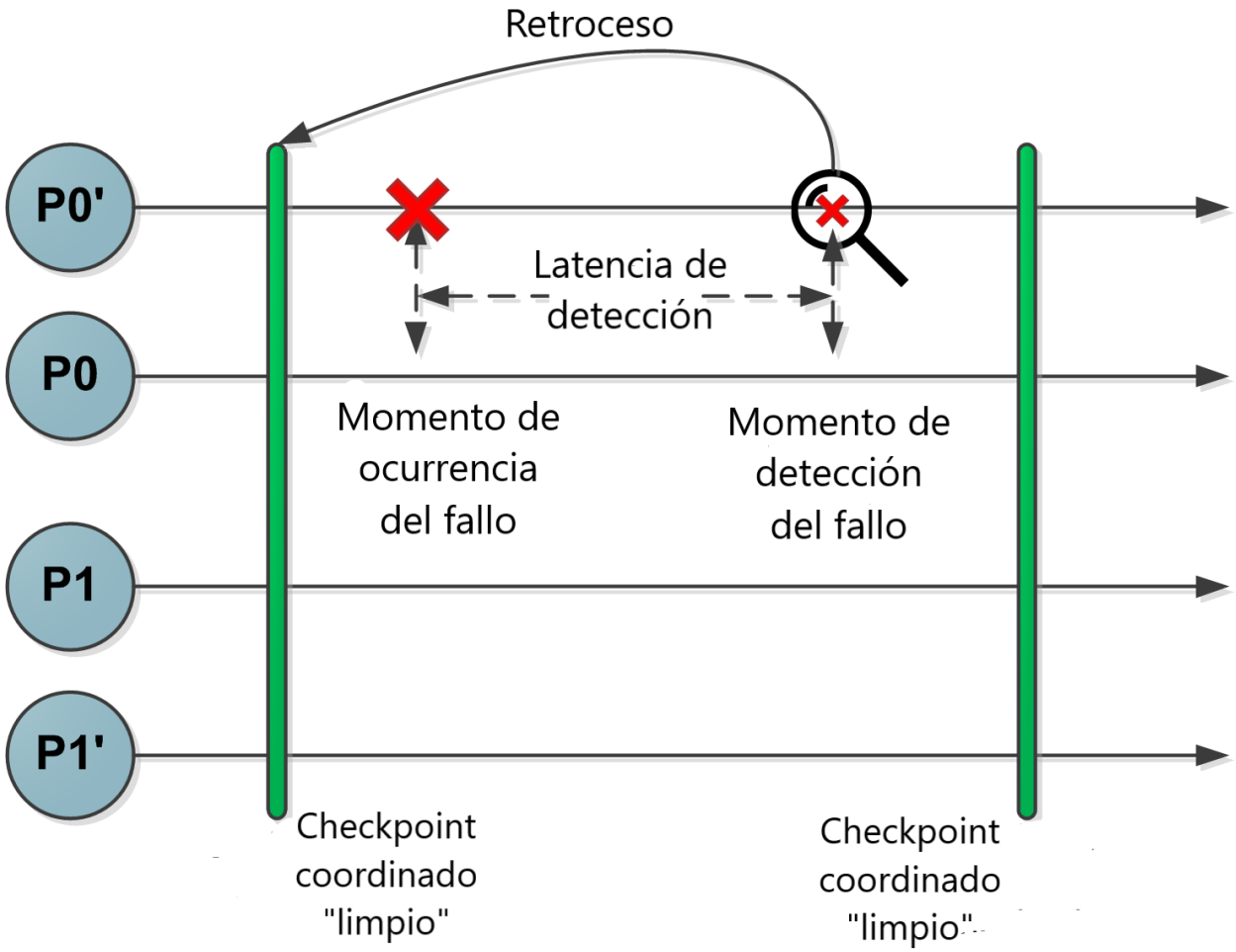

Figura 4.1: *Latencia de detección confinada dentro del intervalo de checkpoint*

El funcionamiento del algoritmo es el siguiente: en simultáneo con la ejecución de los procesos de la aplicación paralela determinística *app* (que se ejecuta monitorizada desde SEDAR), se mantiene una variable *extern_counter* que debe ser externa a la aplicación, de

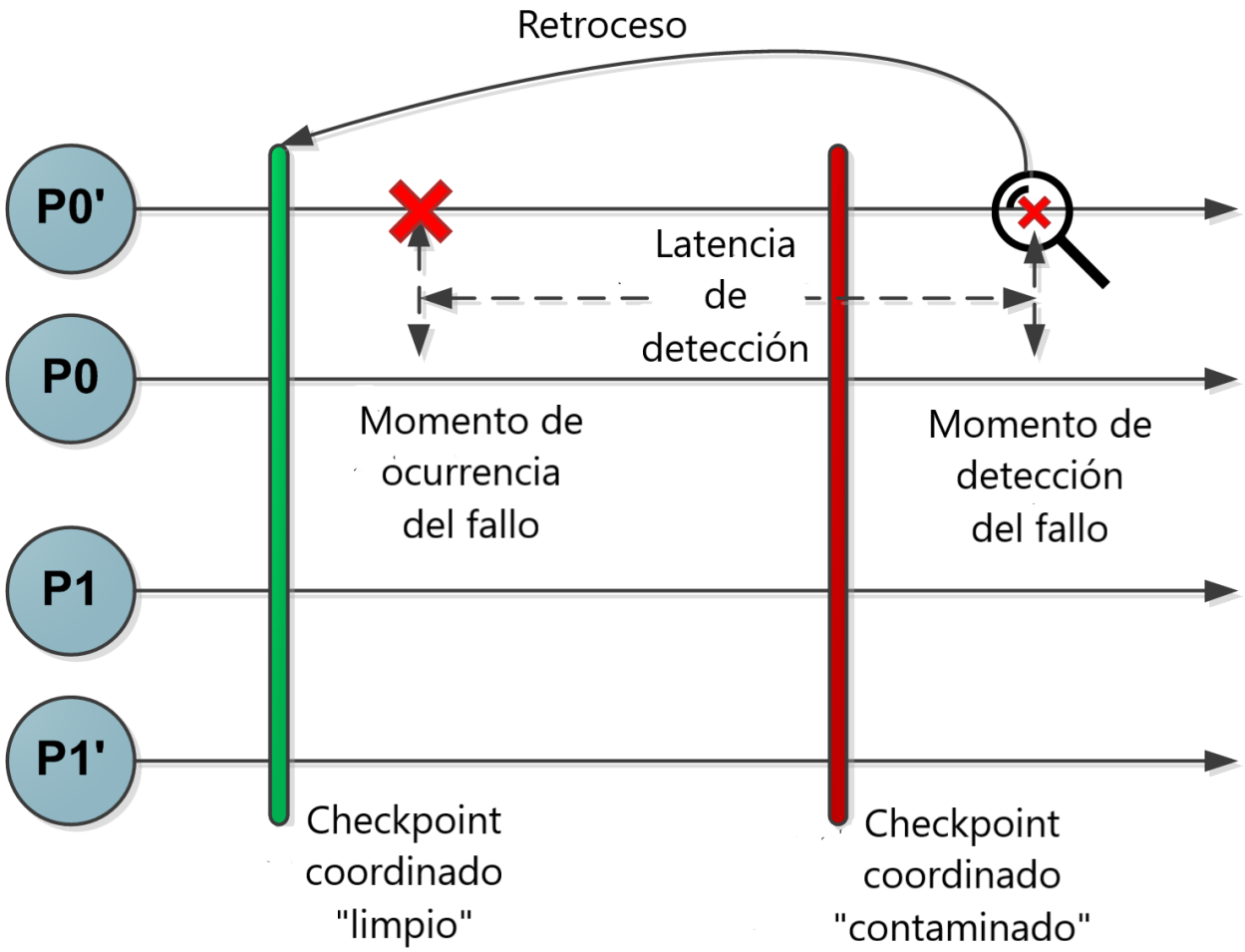

Figura 4.2: *Latencia de detección que traspone el intervalo de checkpoint*

forma de que el almacenamiento de un *checkpoint* no incluya su valor. Esta variable funciona como contador de la cantidad de veces que se detectan fallos, y se inicializa con 0; al ser externa, el historial de fallos detectados permanece registrado aún cuando se realicen diversos intentos de retroceso en el cómputo. Además, la variable booleana *detected* indica si se ha detectado un fallo en la ejecución actual; en una re-ejecución, esta variable es reseteada, de modo de quedar lista para registrar un nuevo error. Cabe aclarar que, por simplicidad, en el código del algoritmo no se incluye el almacenamiento de los *checkpoints* (que puede ser realizado en cualquier momento de la ejecución). La primera vez que se detecta un fallo, se incrementa el valor de la variable *extern_counter*, mientras que *detected* se pone en 1; a

Algoritmo 1 Algoritmo de recuperación con múltiples *checkpoints* de capa de sistema

```
1: int extern_counter = 0;    ▷ un contador externo controla la cantidad de rollbacks (no incluido en el checkpoint)
2: boolean fault_detected = FALSE;    ▷ una variable boolean para reportar si se detectó un error en la última ejecución
3: SEDAR_run(app);    ▷ ejecuta la aplicación paralela app bajo el monitoreo de SEDAR
4: while fault_detected == TRUE do ▷ si la condición se cumple, se detectó un error en la última ejecución
5:     extern_counter++;                    ▷ extern_counter se incrementa en 1
6:     ckpt_count = get_ckpt_count();    ▷ obtiene el número de checkpoints guardados
7:     ckpt_no = ckpt_count - extern_counter + 1;    ▷ calcula el número del checkpoint para el restart
8:     fault_detected = FALSE;                ▷ resetea el flag de detección en el restart
9:     SEDAR_restart(app, ckpt_no);           ▷ relanza app desde el checkpoint ckpt_no
10: end while
```

continuación, el algoritmo obtiene el número del último *checkpoint* almacenado, y luego selecciona el correspondiente *script* de *restart* (se almacena uno junto con cada *checkpoint*) a partir del valor del contador, de la siguiente manera: si la cantidad de *checkpoints* guardados es *ckpt_count*, el *restart* se hace del *script* con el número *ckpt_count* – *extern_counter* + 1. Si *extern_counter* = 1, se intenta recuperar desde el último, que es el comportamiento esperado. Como se mencionó, la variable *detected* se resetea.

Si en la re-ejecución que se realiza para recuperar se detecta nuevamente un fallo, al utilizar el modelo *SEU*, se asume que es el mismo fallo que en la ejecución original; puede ser un nuevo fallo distinto, pero al ocurrir en el mismo lapso que el anteriormente detectado, se asume que el último *checkpoint* está corrompido. En ese caso, se incrementa el contador de fallos (*extern_counter* = 2), con lo cual el restart se intentará desde *ckpt_count* – 1, que es el anteúltimo; si la latencia de detección es muy alta, este procedimiento se repite cada vez que se detecta el error, generando sucesivos intentos de *rollback*, retrocediendo cada vez al *checkpoint* anterior al del último reintento. En el caso extremo de una latencia que atraviese todos los *checkpoints* almacenados, corrompiéndolos, la ejecución recomienza desde el principio, tras acumular intentos fallidos. Además, se debe aclarar que, cada vez que se regresa un *checkpoint* hacia atrás, hay que rehacer los *checkpoints* que ya se habían

hecho y resultaron inválidos.

Para verificar la operación del mecanismo de recuperación propuesto, para los dos casos mencionados, se requiere de experimentos de inyección controlada de fallos. El método de inyección de fallos y su implementación se verán con mayor detalle en el Capítulo 5 (el pseudocódigo de este mecanismo tampoco se muestra en el Algoritmo 1).

Respecto de los *checkpoints*, los momentos más adecuados para guardarlos son luego de las comunicaciones, ya que los datos acaban de ser validados: de esta forma se reducen las ventanas de vulnerabilidad [94], ya que, como hemos visto, la probabilidad de que un error afecte el estado de un *checkpoint* se minimiza de esa forma. De todas maneras, el mecanismo funciona aún cuando los *checkpoints* sean tomados en cualquier instante, como ocurriría en un escenario más realista, donde se realiza *checkpointing* periódico desde el exterior de la aplicación. Además, en general, se introduciría un *overhead* considerable si se almacenara un *checkpoint* luego de cada comunicación.

La usabilidad de este método puede incrementarse si se automatiza el mecanismo. Esto se logra permitiendo que un proceso externo a la aplicación pueda acceder al valor de *extern_counter* y, de acuerdo a él, buscar el *script* de reinicio correspondiente (entre todos los generados por la librería de *checkpointing*). Además, el *checkpoint* inválido, que ha causado el *restart* erróneo debe ser eliminado, sobreescribiéndolo y almacenado nuevamente durante la re-ejecución).

4.2.1. Comportamiento con múltiples fallos

Se ha establecido que SEDAR está diseñado en base a un modelo que contempla la ocurrencia de un único error durante la operación. Sin embargo es importante destacar que, como se explicó en la Sección 3.4, la estrategia de detección *SMCV* es capaz de detectar correctamente múltiples fallos, siempre que sean independientes entre ellos.

En el estado actual de desarrollo de SEDAR, el algoritmo está preparado para lidiar con un único error. Por lo tanto, como se explicó en la Sección 4.2, la situación de detectar

un fallo diferente durante una re-ejecución es equiparada con haber vuelto a detectar el mismo fallo de la ejecución previa, con una latencia detección que traspone el intervalo de *checkpoint*.

En este escenario, el algoritmo es capaz de recuperar la ejecución, aunque no se comporta de manera óptima desde el punto de vista de la *performance* temporal. Al asumir que el último *checkpoint* está corrompido, cuando en realidad no lo está, la detección de un error diferente genera un intento innecesario de *rollback*. En el caso extremo de que en cada intento de re-ejecución se detectara un nuevo error, el mecanismo de recuperación va retrocediendo sucesivamente hasta llegar al comienzo, lo que, a pesar de asegurar la finalización con resultados correctos, produce que el tiempo de recuperación sea muy difícil de predecir en el caso de múltiples fallos.

Esta ineficiencia puede mejorarse si modifica el algoritmo, agregándole un mecanismo algo más sofisticado. Además de guardar la cantidad de fallos que se han detectado en *extern_counter*, deberían almacenarse en un archivo algunos datos relativos al último fallo detectado (por ejemplo, el *tag* de MPI, el *rank* del proceso emisor y del receptor, el byte, dentro del mensaje, en el cual se detectó el fallo, etc), es decir, información necesaria para su identificación. En ese caso, si en la re-ejecución se detecta un error, la verificación de esta información permitiría determinar si este error es una nueva detección del anterior, o bien es un error diferente. En este último caso, el contenido de *extern_counter* no se incrementaría, y el mecanismo buscaría nuevamente la recuperación desde el mismo *checkpoint* que en el intento anterior. La implementación de este mecanismo más refinado se plantea como trabajo futuro.

4.2.2. Ventajas y limitaciones

Una de las principales ventajas del método de recuperación basado en múltiples *checkpoints* de capa de sistema está relacionada con la transparencia para la aplicación. Esto significa que no es necesario conocer su estructura interna (pudiendo utilizarse un modelo de "caja

negra", y, por lo tanto, no se requiere de disponer de *checkpoints* construidos a la medida de la aplicación. En muchas ocasiones, las librerías de *checkpointing* a nivel de sistema son las únicas herramientas disponibles; bajo esas circunstancias, la utilización de esta estrategia resulta aceptable para proporcionar fiabilidad a las ejecuciones. Además, el mecanismo de C/R es el mismo que ya se utiliza para los fallos permanentes. Por lo tanto, la utilización de una estrategia de detección basada en duplicación, combinada con una de recuperación basada en C/R posibilita detectar y corregir las consecuencias de los fallos silenciosos sin la necesidad de recurrir a la triplicación con votación.

Sin embargo, esta metodología presenta también tres limitaciones significativas. La primera de ellas está relacionada con la cantidad de espacio de almacenamiento requerido: debido a la incertidumbre acerca de la validez de los datos contenidos en los *checkpoints*, en principio ninguno de ellos puede ser eliminado hasta no saber si será necesario. En cualquier caso, el impacto negativo sobre el almacenamiento de tener múltiples *checkpoints* puede reducirse por medio de soluciones basadas en $multi-level$ *checkpointing* [30].

La segunda limitación es respecto del gasto de tiempo involucrado en los múltiples intentos de reinicio posibles: si no se encuentra un *checkpoint* consistente, una parte considerable de la aplicación requerirá ser re-ejecutada, pudiendo llegar incluso al extremo de tener que relanzar la ejecución completa desde el principio [16, 83].

La última limitación está relacionada con la baja escalabilidad de los *checkpoints* coordinados al crecer la cantidad de procesos. En la próxima era de la computación con sistemas de exa-escala, y a pesar de algunos esfuerzos considerables [29], los *checkpoints* coordinados de capa de sistema no serían la solución más adecuada, debido a que almacenan una gran cantidad de información relacionada con la plataforma y el sistema operativo. En esta primera versión básica, nuestro método es costoso, debido a que requiere mantener una cantidad indeterminada de *checkpoints* activos y puede necesitar de varios intentos de *restart*.

4.3. Recuperación basada en un único *checkpoint* de capa de aplicación

Como consecuencia de las desventajas mencionadas respecto de los *checkpoints* de nivel de sistema, la utilización de *checkpoints* de capa de aplicación se ha vuelto cada vez más frecuente, especialmente debido a sus menores costos y a sus posibilidades de portabilidad [85]. En esta línea, la tercera alternativa propuesta como parte de la metodología SEDAR busca sobreponerse a estas limitaciones.

A pesar de requerir un conocimiento detallado de la estructura interna de la aplicación (tanto cómputo como comunicaciones), los *checkpoints* de capa de aplicación constituyen una opción más adecuada, debido al hecho de que sólo guardan información relacionada con la aplicación. Además, son más pequeños (ocupando menor espacio de almacenamiento), más portables y escalan mejor que sus contrapartes de nivel de sistema. Por este motivo, en SEDAR se propone la utilización de un único *checkpoint* de nivel de aplicación para la recuperación, en conjunto con una estrategia para garantizar la validez del último *checkpoint* guardado (lo que no se podía hacer con los de nivel de sistema). Por lo tanto, los *checkpoints* previos pueden ser eliminados, reduciendo así el espacio de almacenamiento y decrementando la latencia requerida para re-ejecutar.

La solución propuesta se basa en almacenar *checkpoints* de aplicación (en principio, no-coordinados, para mejorar los problemas de escalabilidad) a nivel de *thread* (es decir, que cada *thread* guarde un *checkpoint* propio), aprovechando el mismo mecanismo de sincronización entre réplicas que se desarrolló en la fase de detección. Entonces, cuando uno de los *threads* llega a la instancia de almacenar un *checkpoint*, se queda esperando allí hasta que su réplica también haya guardado el suyo correspondiente. Estos *checkpoints* consisten en salvaguardar sólo el conjunto de variables que son relevantes para la aplicación en ese momento específico. Como se almacenan *checkpoints* de ambos hilos, es posible calcular un *hash* sobre cada uno de ellos y aplicar el mismo mecanismo utilizado para validar contenidos de mensajes, para comparar en este caso los dos *hashes*. Por lo tanto, un *checkpoint* se

considera "válido" sólo si la verificación resulta exitosa, y en consecuencia sólo permanece almacenado el *checkpoint* cuyo contenido ha probado ser válido, de una manera similar a la que se propone en [16] (aunque en ese trabajo el contenido se verifica antes del almacenamiento). En este escenario, el *checkpoint* previo puede descartarse sin peligro para ahorrar espacio de almacenamiento, ya que el *checkpoint* actual constituye un estado consistente y seguro, y la detección de un error logrará recuperarse correctamente desde él.

En tanto, si se detecta una diferencia en la comprobación, ésta necesariamente se debe a un fallo ocurrido dentro del último intervalo de *checkpoint*; por ende, este último *checkpoint* no puede ser utilizado como punto seguro para la recuperación. Por lo tanto, se debe borrar el *checkpoint* corrompido, y la ejecución debe retomarse desde el anterior (que sí fue verificado). Como consecuencia de esto, existe un único *checkpoint* válido y, en general, un único *checkpoint* almacenado en un momento determinado (excepto durante el intervalo de validación), independentemente del resultado de la comparación.

El pseudo-código del mecanismo de recuperación propuesto se muestra en el Algoritmo 2.

El código del algoritmo refleja el comportamiento descripto anteriormente: la función $usr\_ckpt(n)$ consiste en que, para el *checkpoint* n, cada hilo replicado (identificado por su tid, $threadID$) almacene sus variables de trabajo relevantes y compute el $hash$ sobre ellas. Luego ambos hilos se sincronizan, y sólo uno de ellos compara los $hashes$; si son iguales, borra el *checkpoint* almacenado por una de las réplicas (no son necesarias dos copias iguales) y retorna con éxito; sino, retorna el error en la comparación.

Mientras se ejecutan los procesos de la aplicación paralela determinística *app*, se llama a la función que almacena un *checkpoint*, en un momento determinado (elegido a conveniencia de acuerdo a la estructura de la aplicación particular). Si el llamado retorna correctamente, se elimina el *checkpoint* anterior, ya que no tiene sentido conservarlo. En tanto, si se retorna incorrectamente, se borra el *checkpoint* actual y se retrocede hasta el anterior para volver a lanzar. Con este mecanismo, la latencia de detección siempre queda confinada al intervalo de *checkpoint*.

Algoritmo 2 Algoritmo de recuperación con *checkpoints* de capa de aplicación

```
1: function USR_CKPT(n)                          ▷ definición de la función usr_ckpt
2:     for (tid=0; tid < 2; tid++) do                          ▷ cada una de las réplicas
3:         store_all_significant_variables(tid);        ▷ almacenan su propio checkpoint
4:         hash_array[tid]=compute_hash(tid);
5:     end for
6:     synch_threads();                                          ▷ se esperan mutuamente
7:     if tid==0 then                  ▷ sólo una de las réplicas compara los hashes
8:         if hash_array[0]==hash_array[1] then                          ▷ si coinciden
9:             remove_all_significant_variables(tid);      ▷ elimina su propio checkpoint
10:            return TRUE;          ▷ este es un checkpoint válido, por lo que se puede
    descartar el anterior
11:        else
12:            return FALSE                          ▷ este es un checkpoint inválido
13:        end if
14:    end if
15: end function
16:
17: SEDAR_run(app)    ▷ ejecuta la aplicación paralela app bajo el monitoreo de SEDAR
18: if usr_ckpt(n)== TRUE then                          ▷ n representa el checkpoint actual
19:    remove_usr_ckpt(n-1); ▷ elimina el checkpoint previo, dado que el actual es válido
20: else
21:    remove_usr_ckpt(n);                                ▷ elimina el checkpoint actual
22:    restart_from_usr_checkpoint(n-1);          ▷ restart desde el checkpoint anterior
23: end if
```

En conclusión, SEDAR es capaz de proporcionar protección a las ejecuciones de las aplicaciones de HPC con paso de mensajes, a tres diferentes niveles: (1) sólo detección, notificación al usuario y parada segura; (2) recuperación basada en una cadena de *checkpoints* de capa de sistema; y (3) recuperación basada en un único *checkpoint* válido de capa de aplicación. Cada una de estas variantes provee una cobertura particular pero tiene limitaciones inherentes y costos de implementación; la posibilidad de elegir entre estas alternativas brinda flexibilidad para adaptarse a la relación costo-beneficio requerida para el sistema particular.

Capítulo 5

Implementación y Validación Funcional

Resumen

El objetivo de este capítulo es proporcionar cierto nivel de detalle respecto de cómo se han implementado las funciones que posibilitan tanto la detección como la recuperación automática, de forma de integrarlas en una librería SEDAR, que constituye el prototipo de una herramienta de tolerancia a fallos transitorios que puede ser utilizada en conjunto con aplicaciones de HPC que utilizan paso de mensajes. Se describen las adaptaciones que se deben realizar para integrar SEDAR con el código de las aplicaciones y se describen las diferentes formas de utilización que brinda la herramienta. Por otra parte, se incluye un modelo, basado en el conocimiento de la estructura interna de una aplicación de prueba, que contempla todos los posibles escenarios de fallo que pueden ocurrir. A partir de él, se describen todas las pruebas que se han realizado, mediante inyección de fallos controlada, de manera de verificar la correcta operación, a nivel funcional, de los mecanismos de detección y de recuperación automática basada en múltiples *checkpoints* de nivel de sistema.

5.1. SEDAR como herramienta

En su estado actual de desarrollo e implementación, SEDAR admite 3 modos diferentes de utilización. Como se mencionó en la Sección 4.3, la posibilidad de elegir entre estas alternativas brinda flexibilidad para adaptarse a la relación costo-beneficio requerida para

el sistema particular. Los 3 modos de utilización son:

(a) Sólo detección y relanzamiento automático

(b) Recuperación basada en *checkpoints* de nivel de sistema disparados por eventos

(c) Recuperación basada en *checkpoints* periódicos de nivel de sistema

En cuanto a la estrategia de recuperación automática basada en un *checkpoint* único de capa de aplicación, actualmente está diseñada y desarrollada a nivel de modelo. El trabajo futuro pendiente está relacionado con su implementación y su validación experimental. A los efectos de este trabajo, nos hemos enfocado en las opciones que son más generales y que proporcionan mayor transparencia, dado que no son dependientes de una aplicación específica.

El modo **(a)** permite evitar el costo de los *checkpoints*, tanto en tiempo como en espacio de almacenamiento. Posibilita que la aplicación pueda ser relanzada desde el principio ni bien transcurre la latencia de detección del error. Este modo es útil cuando el *overhead* asociado a los *checkpoints* es muy alto, o cuando los sucesivos intentos fallidos de *rollback* y re-ejecución representan un gasto de tiempo superior al de simplemente relanzar desde el comienzo. En la Sección 6.1.3 se verá una evaluación de estos costos para un caso de estudio.

El modo **(b)** es de utilidad cuando, en la búsqueda de obtener ejecuciones confiables, se tiene un conocimiento detallado del comportamiento de la aplicación a proteger, por ejemplo, en cuanto a la relación cómputo/comunicaciones. En este caso, SEDAR puede sintonizarse con la aplicación, de forma de reducir el *overhead* introducido, ya que los *checkpoints* pueden sincronizarse con eventos significativos de la ejecución, como pueden ser determinadas fases de comunicación. Tomando en cuenta que los datos a comunicar son validados previamente, almacenar *checkpoints* en los instantes de comunicación permite minimizar los riesgos de almacenar valores corrompidos y de propagarlos (es decir, la ventana de vulnerabilidad, ver Secciones 3.7 y 4.2); esto lo hace un mecanismo muy viable para ser utilizado en aplicaciones SPMD, que suelen intercalar secciones de cómputo con comunicaciones frecuentes. Por otra

parte, la flexibilidad de poder seleccionar cuántos *checkpoints* se realizan (y en qué instantes) posibilita optimizar la relación costo/beneficio alcanzada con el mecanismo de protección, y mantener acotado el *overhead*. Este modo fue el utilizado principalmente en la etapa de validación funcional del mecanismo de recuperación automática basada en múltiples *checkpoints* de capa de sistema: basándose en el conocimiento sobre la estructura interna de una aplicación de prueba, y mediante experimentos de inyección controlada de fallos y *checkpoints* sincronizados con las comunicaciones, pudo determinarse, en cada caso, qué dato resultaba afectado, en qué instancia se detectaba ese fallo y desde qué punto podía recuperarse la ejecución. Esto se trata en detalle en la Sección 5.3.1. Como desventaja, esta alternativa no es transparente, en el sentido de que los *checkpoints* deben insertarse dentro del código de la aplicación.

A diferencia del anterior, el modo **(c)** permite proteger a la aplicación "desde el exterior", es decir, de manera transparente. Lanzando la ejecución bajo la órbita de un proceso coordinador de la librería de *checkpointing*, se establece un intervalo de almacenamiento periódico. En este caso, no es necesario interactuar con el código de la aplicación, ni conocer la misma en profundidad, a costa de no tener control sobre el instante preciso en que se realiza el *checkpoint* (respecto del cómputo y la comunicación), ni de desde dónde podrá efectivamente recuperarse la ejecución. El *overhead* tendrá un comportamiento determinado por el costo de almacenar un *checkpoint* y por la cantidad de ellos que se almacenen durante la duración de la ejecución, dado un cierto intervalo de periodicidad. Como ventaja, esta protección pude realizarse de forma completamente automática. Este modo fue el utilizado principalmente en la etapa de caracterización del comportamiento temporal del mecanismo de recuperación automática basada en múltiples *checkpoints* de capa de sistema, el cual se retoma en la Sección 6.1.4. Además, una comparación de los comportamientos temporales y los *overheads* introducidos, entre los modos **(a)** y **(c)**, tanto en presencia como en ausencia de fallos, en distintos escenarios, se desarrolla en la Sección 6.4

5.2. La herramienta de detección *SMCV*

En esta sección se describen la herramienta *SMCV* (el módulo de detección de SEDAR) y su forma de utilización. Como se ha mencionado, *SMCV* intenta ayudar al programador y al usuario de las aplicaciones científicas paralelas determinísticas a obtener resultados fiables o, al menos, a conducir al sistema a una parada segura, reportando la ocurrencia de los fallos silenciosos. Esto permite evitar la innecesaria y costosa espera hasta la finalización de la ejecución, posibilitando el relanzamiento de la ejecución luego del retardo debido a la latencia de detección. Esta es una característica importante, a causa de que este tipo de aplicaciones suelen tener ejecuciones muy prolongados.

La implementación de *SMCV* consiste en un conjunto de funciones y tipos de datos modificados sobre la base de los proporcionados por el estándar MPI. La funcionalidad se extiende para detectar los fallos (mediante comparación de contenidos de $buffers$) al momento del envío de mensajes, duplicar (copiar) los contenidos de los mensajes del lado del receptor y controlar la concurrencia entre las réplicas.

En su versión actual, el módulo *SMCV* de la librería SEDAR puede ser utilizado con aplicaciones MPI desarrolladas en lenguaje C. Redefine una parte de las funciones y tipos de datos de la biblioteca MPI, realizando un único cambio sintáctico: el prefijo MPI se reemplaza por el prefijo SEDAR. Además, agrega dos funciones propias: $SEDAR\_Call$ y $SEDAR\_Validate$. Se utilizan funciones de la librería *Pthreads* para la creación de los hilos replicados y para la sincronización entre ellos, que se realiza por medio de los semáforos provistos por la misma librería. La redefinición de las funciones MPI es necesaria para proporcionar la funcionalidad de detección de fallos de una manera lo más transparente posible para los programadores de las aplicaciones. Este hecho implica la necesidad de modificar levemente el código fuente de la aplicación (por medio de un procedimiento automatizable, ver Sección 5.2.2), y su posterior recompilación.

5.2.1. Funciones básicas

El estándar MPI provee seis funciones básicas [43]. El núcleo del módulo *SMCV* consiste en las redefiniciones de estas seis funciones básicas, y el agregado de dos funciones adicionales. Las funciones básicas de *SMCV* (dentro de la librería SEDAR) se describen a continuación:

- $SEDAR\_INIT$: Inicializa el entorno SEDAR. Esta función debe utilizarse antes de llamar a cualquier otra función de la librería
- $SEDAR\_FINALIZE$: Finaliza la ejecución de todos los procesos que forman una sesión dentro de SEDAR
- $SEDAR\_COMM\_SIZE$: Devuelve el número de procesos dentro del entorno SEDAR actual (asociados a un comunicador)
- $SEDAR\_COMM\_RANK$: Permite determinar el identificador (*rank*) del proceso que invoca a la función
- $SEDAR\_CALL$: Función propia de SEDAR. Crea un nuevo *thread* que ejecuta el código que debe ser validado
- $SEDAR\_SEND$: Sincroniza un proceso con su réplica. El que llega antes al punto de sincronización espera a la réplica. El que llega después compara todos los campos del mensaje a enviar (byte a byte). Si todos coinciden, el primer *thread* envía el mensaje. Si algún campo difiere, se notifica del error y se produce una parada segura. Además, implementa un lapso de tiempo (configurable) para que el segundo *thread* llegue al punto de sincronización, luego del cual se reporta un *TOE*.
- $SEDAR\_RECV$: Sincroniza un proceso con su réplica. El que llega antes al punto de sincronización recibe el mensaje y espera a la réplica. Cuando llega el segundo *thread*, obtiene una copia del contenido del mensaje recibido. Además, implementa un lapso

de tiempo (configurable) para que el segundo *thread* llegue al punto de sincronización, luego del cual se reporta un *TOE*.

- $SEDAR\_VALIDATE$: Sincroniza un proceso con su réplica. El que llega antes al punto de sincronización espera a la réplica. El que llega después compara los resultados finales de ambos *threads* (byte a byte; también podrían calcularse *hashes* de los resultados de cada *thread* y comparar esos *hashes*, para ahorrar tiempo). Si coinciden, ambos *threads* continúan su ejecución. Si difieren, se notifica del error y se produce una parada segura. Además, implementa un lapso de tiempo (configurable) para que el segundo *thread* llegue al punto de sincronización, luego del cual se reporta un *TOE*.

En el caso de $SEDAR\_SEND$, una vez enviado el mensajes, ambos *threads* replicados continúan su ejecución. En tanto, en el caso de $SEDAR\_RECV$, una vez que la réplica obtiene una copia del mensajes, ambos *threads* continúan su ejecución.

Estas 8 funciones son suficientes para desarrollar un amplio espectro de aplicaciones paralelas con capacidad incorporada de detección de fallos transitorios.

5.2.2. Forma de utilización

Para integrar la funcionalidad de detección de *SMCV* al código de una aplicación paralela con MPI, se lleva a cabo el siguiente procedimiento:

1. Se reemplaza el encabezado de MPI por el encabezado de SEDAR.
2. La parte del código que requiere validación (datos e instrucciones) debe ser encapsulada dentro de una función $void*$
3. Se realiza una llamada a la función $SEDAR\_Call$, a la cual se le pasa como argumento la función definida en el paso anterior
4. Se reemplaza el prefijo MPI por SEDAR en todas las funciones y tipos de datos MPI

5. Se realiza una llamada a la función *SEDAR_Validate* para validar el resultado final de la aplicación

En los Algoritmos 3 y 4 se muestra un ejemplo de adaptación de una aplicación MPI para incorporar la funcionalidad de detección con SEDAR (*SMCV*). En el algoritmo 3 se ve el código fuente (simplificado) de la aplicación MPI, mientras que en el algoritmo 4 se muestra el código fuente de la aplicación MPI adaptado para funcionar con SEDAR.

Algoritmo 3 Código de la aplicación MPI original

```
    #include <mpi.h>
int main (int argc, char **argv)
{
MPI_Init();
/* Datos de los procesos, instrucciones y funciones MPI */
MPI_Finalize();
return 0;
}
```

Algoritmo 4 Código de la aplicación MPI adaptado para detección con SEDAR (*SMCV*)

```
    #include <sedar.h>
int main (int argc, char **argv)
{
SEDAR_Init();
SEDAR_Call(&sedar_process);
SEDAR_Finalize();
return 0;
}

void * sedar_process()
{
/* Datos de los threads, instrucciones y funciones SEDAR */
SEDAR_Validate();
}
```

5.2.3. Verificación funcional de la eficacia de detección

La metodología *SMCV* ha sido validada experimentalmente para determinar su eficacia en la detección de fallos transitorios que ocurren en los entornos para los cuales ha sido diseñada. Los fallos que pueden detectarse son los que causan *TDC*, *FSC* y *TOE*. Por otra parte, se ha evaluado también el *overhead* que introduce su utilización respecto del tiempo de ejecución de la aplicación. Esta evaluación temporal se detalla en el Capítulo 6. En esta sección se describen las pruebas realizadas y los resultados obtenidos.

Todo el trabajo experimental fue llevado a cabo en un *cluster* de multicores Blade con ocho nodos (hojas). Cada hoja tiene dos procesadores Quad-Core Intel Xeon e5405 2.0GHz, con 64KB de memoria caché L1 privada, 6MB de memoria caché L2 (compartida entre pares de cores), 10GB de memoria RAM (compartida entre los dos procesadores que componen la hoja) y 250GB de disco local. El sistema operativo es GNU/Linux Debian 6.0.7 de 64 bits, con versión de kernel 2.6.32. En esta etapa, se utilizó la librería de comunicaciones OpenMPI 1.7.5.

Para verificar funcionalmente la eficacia de detección de *SMCV*, la aplicación utilizada fue un pequeño sintético de una multiplicación de matrices paralela MPI ($C = A \times B$) programada bajo el paradigma $Master/Worker$ con 5 procesos (el $Master$ y 4 $Workers$). El proceso $Master$ es responsable de la distribución del trabajo y la recuperación de los resultados finales, y también toma parte en el cómputo de la matriz resultado C [92]. Todas las comunicaciones MPI utilizadas en esta versión son bloqueantes. La operación de la aplicación es la siguiente:

- El proceso $Master$ divide la matriz A en bloques de filas, y envía a cada uno de los $Workers$ su trozo correspondiente de ella. También el $Master$ se envía a sí mismo, manteniendo un trozo para participar también en el cálculo de la matriz resultado. Esto se realiza mediante la operación de comunicación colectiva $MPI\_Scatter$.

- El $Master$ envía a cada $Worker$ una copia de la matriz B completa (y se queda él

mismo con una copia). Esto se realiza mediante la operación de comunicación colectiva $MPI\_Broadcast$.

- Todos los procesos $Workers$ son responsable del cálculo de un bloque de filas de la matriz C; cada uno de ellos computa su trozo correspondiente de forma local, y, cuando finalizan, envían la parte que han calculado al proceso $Master$. Esto se realiza mediante la operación de comunicación colectiva MPI_Gather.

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

Hemos seleccionado la multiplicación de matrices debido a que es una aplicación de estructura regular, intensiva en cómputo, y representativa de las aplicaciones de HPC, con un patrón de communicaciones bien conocido. Esta aplicación demanda gran cantidad de cómputo local, pero requiere pocas comunicaciones, que están acotadas al inicio y al final de cada tarea.

Para poder integrar la funcionalidad de la metodología *SMCV*, debió adaptarse la aplicación de forma de replicar cada uno de sus procesos en un *thread*, para lo que se requiere modificar el código fuente de la aplicación, siguiendo los pasos descriptos en la Sección 5.2.2, y la posterior recompilación.

Para esta primera etapa, los experimentos consistieron en inyectar fallos, de manera controlada, en varios puntos de la aplicación, por medio de la herramienta de *debugging* GDB (el depurador de Linux). La metodología seguida para esto fue: insertar un *breakpoint* en algún punto seleccionado dentro de la ejecución de uno de los procesos, modificar el valor de una determinada variable, y retomar la ejecución con el valor alterado, de forma de poder analizar las consecuencias de esta modificación a lo largo del resto de la ejecución. De esta manera se simula un fallo transitorio simple en un registro interno del procesador, debido a que la corrupción de un dato se manifiesta sólo si puede observarse como una diferencia entre los estados de las réplicas en memoria. Cabe aclarar que este método de

inyección, extremadamente manual o "artesanal" fue utilizado sólo en esta etapa inicial, y automatizado más adelante, como se explica en la Sección 5.3.2.

Si bien los fallos transitorios pueden ocurrir aleatoriamente en cualquier lugar y momento durante la ejecución, la inyección controlada consiste en seleccionar puntos significativos del procesamiento, tanto en el cómputo realizado por el *Master* como en el de los *Workers*.

Como el objetivo de estos experimentos fue realizar una validación simplemente a nivel funcional, se utilizó la aplicación de multiplicación de matrices descripta, en la con tamaño de las matrices cuadradas de 10 x 10 (con todos sus elementos inicializados con el valor 1). Por lo tanto, cada uno de los 5 Workers debe procesar 2 filas (20 elementos) de la matriz A, y producir como resultado 2 filas (20 elementos) de la matriz C.

En la Figura 5.1 se muestra una ejecución normal de la aplicación, sin inyectar ningún fallo, y por lo tanto con salida correcta. 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 tanto, en la Figura 5.2 se muestra la forma en la que se adjunta el depurador GDB para realizar los experimentos de inyección de fallos.

```
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
diego@Lidi137:~/Dropbox/diego/Para trabajo de Especialización/Experimentos$
```

Figura 5.1: *Salida de una ejecución sin fallos*

En la Figura 5.3 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

```
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.
```

Figura 5.2: *Ejemplo de cómo adjuntar el depurador para inyectar fallos*

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 Figura 5.4 se ve la salida de la aplicación, con detección del error y parada segura.

```
(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]
```

Figura 5.3: *Ejemplo de inyección de un fallo que causa 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:
```

Figura 5.4: *Salida de una ejecución en la que se inyectó un fallo que causa FSC*

En la Figura 5.5 se muestra la inyección de un fallo durante la operación de un Worker ($rank = 0$) 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 $Worker$ (y que fue recibido correctamente). 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 Figura 5.6 nuevamente muestra la salida de la aplicación, con detección del error y parada segura. Como el fallo ha causado *TDC*, el mensaje de error es diferente al del caso anterior.

```
(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]
```

Figura 5.5: *Ejemplo de inyección de un fallo que causa 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:
```

Figura 5.6: *Salida de una ejecución en la que se inyectó un fallo que causa TDC*

En la Figura 5.7 se muestra la inyección de un fallo en un elemento cualquiera de la matriz C, en uno de los procesos *Workers*. En este caso, la multiplicación posterior sobreescribe el valor alterado, por lo que el fallo produce un *LE*, que justamente se caracteriza por afectar un dato que no es utilizado en el cómputo posterior. En consecuencia, la salida de la ejecución es normal y correcta, exactamente como la de la Figura 5.1.

```
(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]
```

Figura 5.7: *Ejemplo de inyección de un fallo que causa LE*

Finalmente, en la Figura 5.8 se observa la salida de la aplicación cuando ha ocurrido un fallo que produce un *TOE*; nuevamente, se puede ver la notificación del error y la parada segura. En este caso, el fallo se inyectó en una variable que actúa como índice, produciendo que una de las réplicas del *Worker* recomience desde el principio con 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 $time-out$. El caso ideal de un fallo que produzca *TOE* es que su consecuencia sea que el proceso ingrese a un bucle infinito, pero en la aplicación seleccionada no se puede provocar este comportamiento mediante un fallo simple.

A partir de las pruebas realizadas, se concluye que la incorporación de la estrategia de detección *SMCV* de SEDAR es capaz de detectar los fallos que afectan contenidos de mensajes (*TDC*), notificando al usuario y produciendo una parada segura de la aplicación, de forma que la corrupción no pueda propagarse. De esta forma, todo el cómputo que realizan

```
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:
```

Figura 5.8: *Salida de una ejecución en la que se inyectó un fallo que causa un TOE*

los $Workers$ se encuentra protegido. Por otra parte, los fallos que afectan a datos que el $Master$ mantiene para su cómputo local, y los que se producen luego de que recoge los resultados parciales de todos los $Workers$ en la última etapa de la ejecución (correspondientes a la fracción FSC) son detectados durante la comparación de resultados finales. En tanto, los fallos que producen asimetrías considerables en los tiempos de cómputo de las réplicas (TOE) son detectadas por medio del mecanismo de $time - out$.

5.3. La herramienta SEDAR de recuperación automática

Además de la implementación y verificación funcional de la herramienta de detección $SMCV$, hemos también implementado y validado experimentalmente la funcionalidad de la herramienta de recuperación automática basada en múltiples *checkpoints* de nivel de sistema. En las siguientes subsecciones se detallará el metodo seguido para la comprobación de su funcionamiento, así como las particularidades de la implementación. Como se mencionó en la Sección 5.1, la implementación de la recuperación automática basada en *checkpoints* a nivel de aplicación es un trabajo pendiente aún.

5.3.1. Modelo para la verificación funcional

Para poder validar el funcionamiento de la estrategia de recuperación automática, se ha desarrollado un modelo sobre un caso, basado en una aplicación de prueba bien conocida, combinada con una carga de fallos ($workfault$) completa y totalmente controlada. El caso modelado contempla todos los escenarios de fallos posibles que pueden ocurrir, a partir del profundo conocimiento sobre el comportamiento de la aplicación de prueba, por lo que el $workfault$ está diseñado para para cubrir esos escenarios.

Cada fallo que se inyecta tiene un efecto predecible, un instante en el que se puede prever que será detectado, y un punto desde el cual podrá la recuperarse la ejecución, que puede determinarse con certeza. Como es obvio, existen innumerables posibilidades, a nivel físico del procesador, de que ocurra un fallo (cualquier bit, de cualquier registro o unidad funcional, en cualquier momento de la ejecución); sin embargo, todas ellas están representadas en los escenarios previstos. Un escenario representa una clase de errores, por lo que engloba un amplio conjunto de casos que darán origen a un mismo comportamiento. Para cada experimento que se realizó en esta etapa, se inyectó un único fallo.

La aplicación sintética de prueba, nuevamente, está construida sobre la base de una multiplicación de matrices $Master/Worker$ en MPI. Además de las modificaciones relativas a la replicación de procesos en *threads* para detección y validación final de la matriz resultante (descriptas en la Sección 5.2.3), se agrega el almacenamiento de un *checkpoint* de nivel de sistema cada vez que aplicación acaba de realizar una operación de comunicación. Estos instantes son elegidos debido que los mensajes circulan sólo cuando los datos involucrados han sido validados (es decir, maximiza la probabilidad de datos confiables). El conocimiento detallado sobre el comportamiento de la aplicación de prueba permite identificar claramente en qué instantes ocurren las comunicaciones entre procesos, y los datos que forman parte de cada comunicación. Como consecuencia, se puede predecir el efecto preciso de cada fallo inyectado, como también el estado de cada *checkpoint* almacenado ("seguro" o "corrompido"), y, por ende, qué *checkpoint* posibilita la correcta recuperación.

En el Algoritmo 5 se muestra el pseudocódigo de la aplicación sintética de prueba.

Algoritmo 5 Pseudocódigo de la aplicación de prueba

```
1:  SEDAR_Init()
2:  SEDAR_Ckpt()                          ▷ Checkpoint #0 (CK0)
3:  SEDAR_Scatter(A)                      ▷ El Master reparte la matriz A (SCATTER)
4:  SEDAR_Ckpt()                          ▷ Checkpoint #1 (CK1)
5:  SEDAR_Bcast(B)                        ▷ El Master envía copias de la matriz B (BCAST)
6:  SEDAR_Ckpt()                          ▷ Checkpoint #2 (CK2)
7:  matmul(A,B,C)                         ▷ Cada proceso computa su bloque (MATMUL)
8:  SEDAR_Gather(C)                       ▷ El Master recolecta la matriz C (GATHER)
9:  SEDAR_Ckpt()                          ▷ Checkpoint #3 (CK3)
10: if rank == MASTER then
11:     SEDAR_Validate(C)                 ▷ El Master valida el resultado final (VALIDATE)
12: end if
```

Los experimentos de inyección de fallos, que dan origen a los escenarios posibles, están organizados de acuerdo a los siguientes parámetros:

- P_{inj}: el instante, durante la ejecución, el el cual se realiza la inyección, tomando como referencia la estructura de la aplicación (ejemplo: entre $SCATTER$ y $CK1$)
- *Proceso*: el proceso en cuyo código se realiza la inyección (el que ejecuta el $Master$ o el de alguno de los $Workers$)
- *Dato*: el dato en el cual se inyecta; puede ser un elemento de la matriz A, B o C, o en alguna variable índice. También, si el valor afectado por la inyección es utilizado por el $Master$ o por un $Worker$ para su cómputo (ejemplo: A(M), C(W), i(M), etc.)
- $Efecto$: Consecuencia del fallo: *TDC*, *FSC*, *LE* o *TOE*
- P_{det}: el momento de la ejecución en que se detecta el error. Puede ser al hacer una comunicación o en la validación final
- P_{rec}: el punto más cercano desde el cual es posible recuperar la ejecución

- N_{roll}: la cantidad de intentos requeridos para la correcta recuperación. Los valores posibles son: 0, si el fallo inyectado causa un LE; 1, si es posible recuperar desde el último *checkpoint* almacenado; 2, si es necesario retroceder hasta el anteúltimo checkpoint; y así sucesivamente

Todos estos factores fueron combinados para dar origen a un conjunto de 64 experimentos de inyección de fallos que contemplan todas las situaciones que pueden ocurrir para la aplicación de prueba particular. Los 64 escenarios han sido diseñados de acuerdo a los siguientes criterios: los fallos se inyectan tanto en el código ejecutado por el $Master$ como por los $Workers$; en elementos de cada una de las tres matrices. Las inyecciones en el código del $Master$ se realizan tanto sobre datos que se transmiten como sobre otros que se conservan para cómputo local; en tanto, las inyecciones en el código de los $Workers$ se realizan sobre datos que se transmiten, ya que los $Workers$ no conservan resultados locales. Tanto en el $Master$ como en los $Workers$, se inyectan fallos luego de cada *checkpoint* (es decir, el *checkpoint* es "seguro" o está "limpio"), por lo que es posible la recuperación; en cambio, otros fallos se inyectan justo después de una operación de comunicación pero antes del *checkpoint* subsiguiente (es decir, entre ambos, en datos que acaban de validarse, "corrompiendo" o "ensuciando" el *checkpoint*), forzando que se requiera más de un *rollback* para la recuperación. Por otra parte, se realizan inyecciones en variables índices (tanto en el $Master$ como en los $Workers$), durante el desarrollo de la operación de multiplicación de matrices, de modo de provocar una asimetría en el procesamiento de ambos *threads* redundantes. Es importante remarcar que, como se ha explicado, cualquier fallo que afecte un dato dentro de un cierto subconjunto y que ocurra en cualquier momento dentro del lapso de la ejecución comprendido dentro de un escenario particular (por ejemplo, entre entre $SCATTER$ y $CK1$) se detecta en la misma instancia y permite recuperación desde el mismo *checkpoint*. Dicho de otra forma, cualquier error posible tiene un efecto similar al de alguno de los 64 escenarios previstos; estos 64 escenarios se derivan del estudio del comportamiento de la aplicación.

Caso	P_{Inj}	Proceso	Dato	Efecto	P_{det}	P_{rec}	N_{roll}
1	CK0 - SCATTER	Master	A (M)	FSC	VALIDATE	CK0	4
2	CK0 - SCATTER	Master	A (W)	TDC	SCATTER	CK0	1
3	CK0 - SCATTER	Master	B (M or W)	TDC	BCAST	CK0	2
4	CK0 - SCATTER	Master	C (M)	LE	-	-	0
5	CK0 - SCATTER	Worker	A (W)	LE	-	-	0
6	CK0 - SCATTER	Worker	B (W)	LE	-	-	0
7	CK0 - SCATTER	Worker	C (W)	LE	-	-	0
8	SCATTER - CK1	Master	A (M)	FSC	VALIDATE	CK0	4
9	SCATTER - CK1	Master	A (W)	LE	-	-	0
10	SCATTER - CK1	Master	B (M or W)	TDC	BCAST	CK0	2
11	SCATTER - CK1	Master	C (M)	LE	-	-	0
12	SCATTER - CK1	Worker	A (W)	TDC	GATHER	CK0	3
13	SCATTER - CK1	Worker	B (W)	LE	-	-	0
14	SCATTER - CK1	Worker	C (W)	LE	-	-	0
15	CK1 - BCAST	Master	A (M)	FSC	VALIDATE	CK1	3
16	CK1 - BCAST	Master	A (W)	LE	-	-	0
17	CK1 - BCAST	Master	B (M or W)	TDC	BCAST	CK1	1
18	CK1 - BCAST	Master	C (M)	LE	-	-	0
19	CK1 - BCAST	Worker	A (W)	TDC	GATHER	CK1	2
20	CK1 - BCAST	Worker	B (W)	LE	-	-	0
21	CK1 - BCAST	Worker	C (W)	LE	-	-	0
22	BCAST - CK2	Master	A (M)	FSC	VALIDATE	CK1	3
23	BCAST - CK2	Master	A (W)	LE	-	-	0
24	BCAST - CK2	Master	B (M)	FSC	VALIDATE	CK1	3
25	BCAST - CK2	Master	B (W)	LE	-	-	0
26	BCAST - CK2	Master	C (M)	LE	-	-	0
27	BCAST - CK2	Worker	A (W)	TDC	GATHER	CK1	2
28	BCAST - CK2	Worker	B (W)	TDC	GATHER	CK1	2
29	BCAST - CK2	Worker	C (W)	LE	-	-	0
30	CK2 - MATMUL	Master	A (M)	FSC	VALIDATE	CK2	2
31	CK2 - MATMUL	Master	A (W)	LE	-	-	0
32	CK2 - MATMUL	Master	B (M)	FSC	VALIDATE	CK2	2

Figura 5.9: *Primeros 32 escenarios de fallos en la aplicación de prueba*

En las Figuras 5.9 y 5.10 se muestra la tabla completa de los 64 escenarios posibles de ocurrencias de fallos y sus consecuencias.

Para ilustrar el método aplicado en la verificación funcional, hemos seleccionado 4 casos representativos: el Caso 2, el Caso 29, el Caso 50 y el Caso 59. Estos casos fueron elegidos para poner de manifiesto los 4 posibles efectos de los fallos: *TDC*, *FSC*, *LE* o *TOE*, pero también para mostrar inyecciones realizadas en los códigos tanto del *Master* como de los

Caso	P_{Inj}	Proceso	Dato	Efecto	P_{det}	P_{rec}	N_{roll}
33	CK2 - MATMUL	Master	B (W)	LE	-	-	0
34	CK2 - MATMUL	Master	C (M)	LE	-	-	0
35	CK2 - MATMUL	Worker	A (W)	TDC	GATHER	CK2	1
36	CK2 - MATMUL	Worker	B (W)	TDC	GATHER	CK2	1
37	CK2 - MATMUL	Worker	C (W)	TDC	GATHER	CK2	1
38	MATMUL - GATHER	Master	A (M)	LE	-	-	0
39	MATMUL - GATHER	Master	A (W)	LE	-	-	0
40	MATMUL - GATHER	Master	B (M)	LE	-	-	0
41	MATMUL - GATHER	Master	B (W)	LE	-	-	0
42	MATMUL - GATHER	Master	C (M)	FSC	VALIDATE	CK2	2
43	MATMUL - GATHER	Worker	A (W)	LE	-	-	0
44	MATMUL - GATHER	Worker	B (W)	LE	-	-	0
45	MATMUL - GATHER	Worker	C (W)	TDC	GATHER	CK2	1
46	GATHER - CK3	Master	A (M)	LE	-	-	0
47	GATHER - CK3	Master	A (W)	LE	-	-	0
48	GATHER - CK3	Master	B (M)	LE	-	-	0
49	GATHER - CK3	Master	B (W)	LE	-	-	0
50	GATHER - CK3	Master	C (M)	FSC	VALIDATE	CK2	2
51	GATHER - CK3	Worker	A (W)	LE	-	-	0
52	GATHER - CK3	Worker	B (W)	LE	-	-	0
53	GATHER - CK3	Worker	C (W)	LE	-	-	0
54	CK3 - VALIDATE	Master	A (M)	LE	-	-	0
55	CK3 - VALIDATE	Master	A (W)	LE	-	-	0
56	CK3 - VALIDATE	Master	B (M)	LE	-	-	0
57	CK3 - VALIDATE	Master	B (W)	LE	-	-	0
58	CK3 - VALIDATE	Master	C (M)	FSC	VALIDATE	CK3	1
59	MATMUL	Master	i	TOE	GATHER	CK2	1
60	MATMUL	Master	j	TOE	GATHER	CK2	1
61	MATMUL	Master	k	TOE	GATHER	CK2	1
62	MATMUL	Worker	i	TOE	GATHER	CK2	1
63	MATMUL	Worker	j	TOE	GATHER	CK2	1
64	MATMUL	Worker	k	TOE	GATHER	CK2	1

Figura 5.10: *Últimos 32 escenarios de fallos en la aplicación de prueba*

Workers, evidenciar diferentes instantes de detección y reflejando las distintas posibles situaciones de recuperación: que no se requiera *rollback*, que se requiera hasta el último *checkpoint* o que se requieran varios intentos.

De las Figuras 5.9 y 5.10, se puede observar que:

- En el Caso 2, la inyección modifica el contenido de la matriz A del proceso *Master*, entre el *checkpoint* inicial $CK0$ y la comunicación $SCATTER$. El elemento de A en

el que se inyecta se va a transmitir a un $Worker$; por lo tanto, esta inyección produce un error del tipo TDC, que es detectado al momento del $SCATTER$. En este caso, es posible recuperar la ejecución desde el último *checkpoint* almacenado $CK0$, que está "limpio" (no ha sido afectado).

- El Caso 29 describe un ejemplo de un error latente. La inyección se realiza entre la operación $BCAST$ y el *checkpoint* $CK2$, y afecta al espacio reservado para la matriz C de uno de los $Workers$. En esta instancia, la matriz resultado no ha sido computada aún, por lo que la ubicación en donde se realizó la inyección va a ser sobreecrita más adelante. Por lo tanto, este fallo no modifica el resultado final.

- El Caso 50 consiste en un fallo que causa un error del tipo FSC. La inyección se realiza en un elemento de la matriz C que ya ha sido calculado y recibido por el $Master$ en la operación de comunicación $GATHER$ (por lo tanto, ya validado), pero antes del almacenamiento del *checkpoint* $CK3$ (por lo tanto, se corrompe el *checkpoint*). El error se va a detectar en la operación $VALIDATE$, y, debido a que $CK3$ está "sucio", se requiere un intento de *rollback* adicional para lograr la recuperación.

- En el Caso 59 se describe una inyección que causa un TOE. Esta inyección se realiza durante la operación MATMUL y afecta una variable índice que es utilizada por uno de los procesos $Workers$, provocando que una de las réplicas recomience con su cómputo luego de haber realizado parte de su tarea. Esto produce un retardo en el *thread* afectado, que se detecta como un TOE; sólo la réplica donde no se ha inyectado alcanza la operación de $GATHER$ dentro de un lapso configurado previamente. La recuperación se realiza desde el $CK2$ (es decir, un único *rollback* es suficiente).

Del análisis realizado se puede extraer como conclusión es que cualquier fallo aleatorio que pueda ocurrir durante la ejecución es semejante a alguno de los escenarios modelados. Un método de análisis como el descripto, basado en el conocimiento sobre la aplicación y el control sobre los momentos en los que se almacenan los *checkpoints*, en combinación

con la inyección de fallos controlada y sistemática, permite predecir el comportamiento tanto del mecanismo de detección como del de recuperación automática. A partir de eso, la eficiencia de la estrategia puede mostrarse con una aplicación que se ejecute en un entorno real, con fallos aleatorios. Esta evaluación de los mecanismos de SEDAR es el objetivo de la verificación funcional realizada. Debido a que la operación de ambos mecanismos se prueba para todos los errores que pueden ocurrir en la aplicación de prueba particular seleccionada, y a que no existen otras clases de errores, la comprobación realizada verifica la validez funcional de SEDAR.

Es importante aclarar que este procedimiento puede ser adaptado para verificar la eficacia de SEDAR junto con cualquier aplicación de prueba. Los instantes y los datos sobre los que se inyecta pueden variar ampliamente de una aplicación a otra, pero el método está diseñado para cubrir todos los posibles escenarios de error.

5.3.2. Implementación y validación experimental

La herramienta de recuperación automática de SEDAR se implementa integrando la herramienta de detección *SMCV* (que cuenta con la funcionalidad descripta en la Sección 5.2) con la librería de *checkpointing DMTCP* [7]. *DMTCP* proporciona *checkpointing* coordinado y distribuido de nivel de sistema, generando archivos de *checkpoint* distribuidos en cada proceso, y un *script* de *restart* único para cada *checkpoint*.

Por defecto, la librería almacena un único *checkpoint*, sobreescribiendo el archivo generado anteriormente cada vez que se guarda un *checkpoint* nuevo; sin embargo durante la instalación, puede modificarse este comportamiento mediante la configuración del **flag** *enable_unique_checkpoint* = **no**, de modo que se pueden almacenar varios *checkpoints* (con números correlativos) y sus correspondientes *scripts* de reinicio, sin perder ninguno de los estados previos; justamente, el mecanismo de recuperación de SEDAR está basado en esta posibilidad. *DMTCP* genera un proceso coordinador, que es el encargado de monitorear la aplicación objetivo y realizar los *checkpoints*.

A partir de la integración de estas herramientas (cabe remarcar que todas ellas son estándar), se ha implementado la aplicación sintética de prueba detallada en el Algoritmo 5 y los experimentos de inyección correspondientes a los 64 escenarios. Para la realización de los *checkpoints*, todos los procesos de la aplicación invocan a la función $SEDAR\_Ckpt$, pero sólo uno de los procesos (en este caso, el $Master$) es el encargado de almacenar el estado de toda la aplicación. Cuando se realiza el llamado a la función, ambos *threads* redundantes de cada proceso se sincronizan, de la misma forma en que lo hacen antes de enviar un mensaje para validar sus datos, y sólo uno de ellos llama efectivamente a $DMTCP\_Ckpt$ desde el interior de SEDAR (de esta forma se genera un solo *checkpoint* de cada proceso, y no dos a nivel de hilos).

La mecánica de inyección de fallos es esencialmente la misma que se describió en la Sección 5.2.3, es decir, simular un $bit - flip$ en un registro del procesador, modificando el valor de una variable en sólo una de las réplicas, en una única iteración del cómputo. Sin embargo, como allí se mencionó, el mecanismo de inyección fue automatizado para esta segunda etapa de verificación funcional de la estrategia de recuperación automática, de forma de que la inyección sea realizada desde el interior del código de la aplicación. Para esto, se ha incluido una función *ad-hoc* en la librería, llamada $SEDAR\_Inject$, que contiene el código de los 64 casos, cada uno identificado con un número. Se utiliza compilación condicional para seleccionar qué caso de inyección se realiza en cada experimento: en la línea de compilación se incluye un $flag$ (-D) que produce que se compile sólo el código correspondiente al número de caso seleccionado (es decir, que compile el código para un cierto valor de una constante definida en tiempo de compilación), excluyendo el resto de los casos. De esta forma, se parametriza el código fuente, produciendo que sea el mismo para todos los experimentos, pero, dependiendo de la inyección particular, la función $SEDAR\_Inject$ es invocada en un sitio diferente durante la ejecución.

Esta función trabaja en conjunto con un archivo, llamado *injected.txt*, que se utiliza para controlar si ya se ha realizado una inyección, de modo de hacer sólo una en cada experimento.

Cada vez que se llama a la función *SEDAR_Inject*, se lee y evalúa el contenido de este archivo. La primera vez, el archivo contiene el valor *0*, por lo cual se realiza la inyección, y una vez que eso ocurre, su contenido se "incrementa" (se cambia a *1*). Durante el proceso de recuperación (una vez que se detectó el error), se re-ejecuta parte del código de la aplicación, incluyendo la llamada a la rutina que realiza la inyección. Como se vuelve a leer y evaluar el contenido del archivo (que ahora es distinto de *0*), la función retorna sin realizar una nueva inyección. Como consecuencia, el código de inyección de fallos se ejecuta sólo una vez. Por supuesto, el archivo *injected.txt* debe ser externo a la aplicación, de forma de que su contenido "sobreviva" (no resulte afectado) por el almacenamiento de un *checkpoint*.

La función *SEDAR_Inject*, en pseudocódigo, se muestra en el Algoritmo 6. El valor de la constante *current_inj*, en una ejecución particular, es el que se define en la línea de compilación condicional. En tanto, el Algoritmo 7 muestra el pseudocódigo de la aplicación de prueba, incluyendo (a modo de ejemplo) algunos casos de inyección.

El mecanismo de recuperación, basado en mantener una cadena de *checkpoints*, utiliza otro archivo para controlar el número de veces que se ha detectado un fallo, llamado *failures.txt*. Este archivo es la implementación de la variable *extern_counter* descripta en la Sección 4.2. El archivo se inicializa con *0*, y su contenido se incrementa cada vez que se detecta un fallo. Si se asume que, durante una ejecución, ocurre un único error, el archivo contiene *1* para la primera detección. Una vez que *SMCV* detecta el fallo, lee el contenido del archivo, y, según la mecánica descripta en el Algoritmo 1, intenta el *rollback* al último *restart script* almacenado. Si en la re-ejecución que ocurre al recuperar se detecta un error, se asume que es el mismo que en la ejecución original, y que por lo tanto el último *checkpoint* estaba "corrompido", habiendo sido atravesado por la latencia de detección. En este caso, el contenido del archivo se incrementa hasta *2*, y se intenta un *rollback* al *checkpoint* anterior. Por lo tanto, el contenido del archivo se utiliza para seleccionar el número de *restart script* que se debe ejecutar. Como se explicó, el archivo es externo a la aplicación, de modo de que su contenido sea independiente del almacenamiento de los *checkpoints*. En tanto, si no

Algoritmo 6 Pseudocódigo de la función *ad − hoc* para inyección de fallos

```
1: procedure SEDAR_Inject(data)
2:     sync_threads()                                  ▷ Se sincronizan las réplicas
3:
4:     /*Solo uno de los hilos ejecuta el código siguiente*/
5:
6:     leer (injected.txt)          ▷ Abre el archivo injected.txt para recuperar su valor
7:
8:     if injected == 0 then                          ▷ Si aun no se realizó la inyección
9:
10:        #if current_inj == 1
11:        /*Código que realiza la inyección 1*/
12:        #endif
13:
14:        #if current_inj == 2
15:        /*Código que realiza la inyección 2*/
16:        #endif
17:
18:        ...
19:
20:        escribir (injected.txt)          ▷ Abre el archivo injected.txt y guarda "1"
21:    end if
22:    sync_threads()                                  ▷ Se sincronizan las réplicas
23: end procedure
```

es posible asumir que ocurre un único error durante la ejecución, el comportamiento del mecanismo es el detallado en la Sección 4.2.1.

Como aclaración, una variante simplificada de este mecanismo es la que posibilita pasar de la estrategia de sólo detección del error con notificación al usuario y parada segura (Secciones 3.2.2 y 3.2.3) a la de sólo detección de error con relanzamiento desde el comienzo (variante **(a)** descripta en la Sección 5.1). En este caso, cuando *SMCV* detecta un fallo, se lee el archivo *failures.txt* y, en lugar de buscar un *script* para recuperación, simplemente ejecuta la línea de lanzamiento de la aplicación.

Se realizaron los 64 experimentos de inyección sobre la aplicación de prueba, utilizando dos nodos (hojas) de la plataforma que para las pruebas detalladas en la Sección 5.2.3. La única diferencia es que, en el caso actual, la librería de paso de mensajes utilizada fue

Algoritmo 7 Pseudocódigo de la aplicación de prueba (incluyendo inyecciones según Figuras 5.9 y 5.10)

```
1:  SEDAR_Init()
2:
3:  SEDAR_Ckpt()                                   ▷ Checkpoint #0 (CK0)
4:
5:  #if current_inj == 1                           ▷ Inyección #0
6:   SEDAR_Inject(A)
7:  #endif
8:  ...                                            ▷ Resto de las inyecciones en esta sección
9:  #if current_inj == 7                           ▷ Inyección #7
10:  SEDAR_Inject(C)
11: #endif
12:
13:  SEDAR_Scatter(A)                              ▷ El Master reparte la matriz A (SCATTER)
14:
15: #if current_inj == 8                           ▷ Inyección #8
16:  SEDAR_Inject(A)
17: #endif
18: ...                                            ▷ Resto de las inyecciones en esta sección
19: #if current_inj == 14                          ▷ Inyección #14
20:  SEDAR_Inject(C)
21: #endif
22:
23:  ...                                           ▷ Resto del código
```

MPICH (versión 3.3.1), además de la librería de *checkpointing DMTCP* (version 2.4.4).

La Figura 5.11 muestra, a modo de ejemplo, la salida generada por SEDAR para uno de los experimentos de inyección, en este caso, el Escenario 50, de forma de ilustrar la metodología adoptada y el comportamiento obtenido. Se puede observar que, frente a la primera detección del error tipo *FSC*, que ocurre en la operación $VALIDATE$, SEDAR intenta recuperar desde el último *restart script* generado por DMTCP; sin embargo, el error vuelve a detectarse en la misma instancia, y SEDAR intenta la recuperación desde el *restart script* anterior (observar los número de los *scripts*, se aclara que SEDAR comienza la numeración en 1). El cómputo vuelve atrás hasta el anteúltimo *checkpoint* almacenado, llegando esta vez al final. Se pueden ver también los contenidos de los archivos asociados

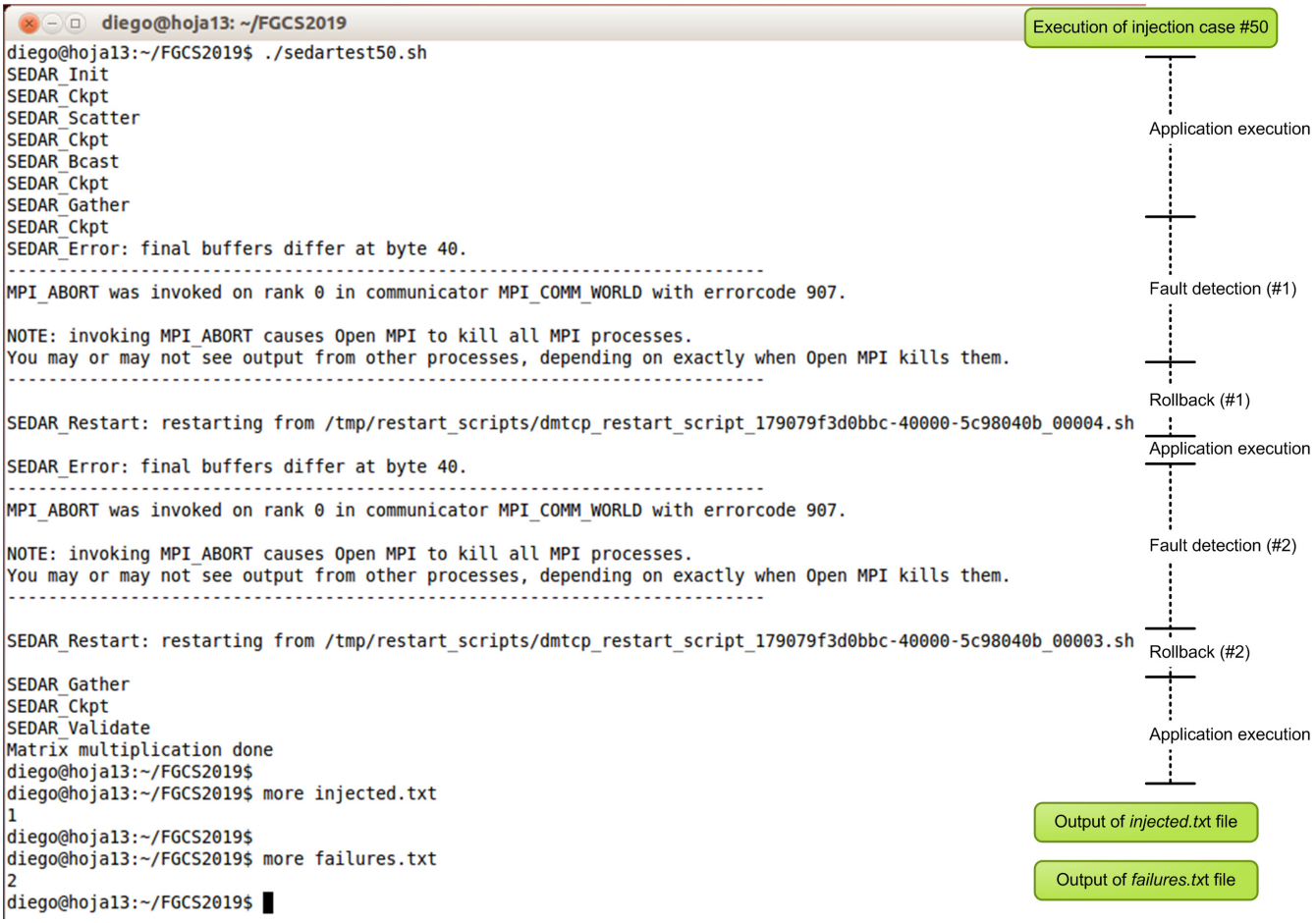

Figura 5.11: *Salida de la ejecución de la aplicación de prueba cuando se inyecta el fallo del Caso 50*

al mecanismo: *injected.txt* contiene *1*, que significa que se realizó la inyección (y por lo tanto, en la re-ejecución no se volvió inyectar), mientras que *failures.txt* contiene *2*, lo que significa que el error se detectó *2* veces, como se esperaba.

Como un detalle, para esta etapa de verificación funcional, la implementación tolerante a fallos de las funciones de MPI se basó en comunicaciones de punto a punto, lo que las vuelve más generales, y además permiten mostrar los casos de *FSC*, a expensas de sacrificar *performance*. Sin embargo, también hemos desarrollado versiones optimizadas de comunicaciones colectivas. En este tipo de comunicaciones colectivas, el proceso que envía también participa en la operación, tanto enviando como recibiendo (es decir, se envía mensajes a

sí mismo). Por lo tanto, si una aplicación de prueba sólo utiliza comunicaciones colectivas, todos los datos se trasmiten (incluso los que el proceso emisor conserva para sí mismo), por lo que la corrupción se detecta al momento de validar contenidos de mensajes. De esta forma, sólo quedaría los fallos que causan *TDC*, mientras que los *FSC* no deberían estar presentes. Esta idea podría extenderse a muchas otras aplicaciones.

Capítulo 6

Caracterización Temporal y Resultados Experimentales

Resumen

En el capítulo anterior nos hemos centrado en los aspectos funcionales de SEDAR, y en la verificación de la eficacia de su operación. En este capítulo, el foco está puesto en los aspectos prestacionales. Se presenta una caracterización y evaluación del comportamiento temporal de las distintas estrategias que conforman la metodología. Para esto, SEDAR se utiliza en conjunto con tres *benchmarks* paralelos con diferentes patrones de comunicación y tamaños de carga de trabajo (*workloads*), obteniendo, en base a medidas y estimaciones, los parámetros de ejecución. Como consecuencia, se evalúa cualitativamente la incidencia del patrón de comunicaciones, del intervalo de *checkpointing* y de la latencia de detección en el comportamiento temporal. Se propone una pequeña discusión sobre la conveniencia de almacenar múltiples *checkpoints* en lugar de activar sólo el mecanismo de detección, considerando el *overhead* introducido. Esto representa una primera aproximación para proveer elementos de análisis que permitan determinar cuándo es beneficioso emplear cada estrategia. También se estudia brevemente la forma de decidir cuándo es el instante más conveniente para comenzar la protección. Por último, se realizan mediciones del *overhead* de ejecución en escenarios realistas, tanto en ausencia de fallos como con fallos inyectados en distintos puntos durante la ejecución, en la búsqueda de aportar información que permita

alcanzar la configuración óptima de la herramienta cuando se la utiliza, integrada de forma transparente, en entornos y problemas de mayor escala.

6.1. Caracterización temporal de SEDAR

En esta sección se realiza una evaluación cuantitativa del comportamiento temporal de las tres estrategias que componen SEDAR. Se presenta un modelo que tiene en cuenta los parámetros de ejecución de cada una de las alternativas, tanto en ausencia de fallos como cuando ocurre un único fallo durante la ejecución. El modelo se concreta en la forma de ecuaciones, que pueden utilizarse para estimar el tiempo de ejecución para cada caso, si se pueden medir u obtener valores aproximados para los parámetros mencionados.

6.1.1. Caso base (*baseline*)

Con el objetivo de evaluar las diferentes estrategias que componen SEDAR, se utiliza una estrategia como punto de comparación, o *baseline*. Esta estrategia consiste en un método manual que garantiza resultados confiables. El método plantea ejecutar dos instancias simultáneas de la aplicación y comparar los resultados finales de ambas ejecuciones de forma semi-automática. De esta manera se logra un esquema de detección manual (que requiere acciones por parte del usuario), en el que todos los recursos de cómputo del *cluster* se utilizan para tener ejecuciones redundantes, con posibilidad de comparar los resultados finales (como procesamiento $offline$), posteriormente a la finalización de la ejecución. En ausencia de fallos, los resultados finales coinciden. En tanto, en caso de diferencia, es posible lanzar una tercera instancia (manteniendo el mismo *mapping* de la ejecución original), de forma de poder votar el resultado correcto (deberían ocurrir al menos dos fallos en ejecuciones distintas para que no hubiera ningún resultado coincidente).

Los autores de [14] mencionan una idea similar a esta cuando se refieren a "replicación grupal" (*group replication*), que es una técnica que puede utilizarse cuando la replicación de procesos no está disponible (debido, por ejemplo, a la gran cantidad de trabajo que demanda

implementarla explícitamente dentro del código de la aplicación). Como la replicación grupal ve a la aplicación como una "caja negra", es agnóstica al modelo de programación paralela, requieriendo sólo que la aplicación pueda ser lanzada desde un *checkpoint* almacenado. La replicación grupal consiste en ejecutar múltiples instancias de la aplicación de forma concurrente. Por ejemplo, dos instancias separadas de la aplicación, cada una de P procesos, pueden ejecutarse en una plataforma de $2P$ procesadores. Con esta aproximación, pareciera que la mitad de los recursos se "desperdician" (desde el punto de vista de la $performance$) en efectuar el cómputo redundante. Por supuesto, es un precio que se debe pagar para obtener resultados confiables, o al menos detección de errores. Sin embargo, los autores establecen que se puede obtener una mejora en la $performance$. Además de los aspectos relacionados con la escalabilidad y la eficiencia en la utilización de los recursos que se han comentado en la Sección 3.2.4 [107, 116], la posición es justificada con el hecho de que cada instancia de la aplicación se ejecuta en una plataforma de una escala menor (la mitad de la original). Debido a que la tasa de fallos crece con el número de procesadores (o, dicho de otra forma, el $MTBE$ disminuye al aumentar el tamaño de la plataforma, como también se mencionó en la Sección 3.2.4), reducir la escala impacta positivamente; además, como consecuencia, cada instancia individual puede utilizar una menor frecuencia de *checkpointing*, por lo que la ejecución puede tener una mejor eficiencia paralela en presencia de fallos, si se la compara con una única instancia que se ejecuta a escala completa. Por último, el gasto de potencia de procesamiento que se realiza para el cómputo redundante se compensa con el beneficio de un mayor intervalo de *checkpointing*.

Teniendo en cuenta esta idea, se observa que una estrategia manual como la propuesta consume los mismos recursos de cómputo que SEDAR; es decir, la mitad de todos los cores existentes en el sistema se asignan a cada instancia individual. Este hecho hace que la comparación entre el *baseline* y SEDAR sea mas justa que comparar con una ejecución que utiliza el *cluster* completo pero no proporciona ninguna forma de protección.

En ausencia de fallos, el tiempo de ejecución de una aplicación protegida por esta estra-

tegia manual viene dado por la Ecuación 6.1 ($fault\ absence$, T_{FA}). Consiste en el tiempo utilizado por dos instancias independientes para ejecutarse simultáneamente (T_{prog}), sumado al tiempo necesario para comparar los resultados de ambas ejecuciones. En cambio, la Ecuación 6.2 da cuenta del tiempo consumido cuando ocurre un fallo ($fault\ presence$, T_{FP}), que es el tiempo de la nueva re-ejecución y de una nueva comparación para votar, sumado al tiempo requerido para relanzar esta tercera ejecución ($restart\ time$, T_{rest}), que demanda el mismo tiempo que las dos originales anteriores debido a que utiliza el mismo *mapping*.

$$T_{FA} = T_{prog} + T_{comp} \tag{6.1}$$

$$T_{FP} = 2(T_{prog} + T_{comp}) + T_{rest} \tag{6.2}$$

6.1.2. Parámetros de la caracterización temporal

En la Tabla 6.1 se listan y resumen los significados de todos los parámetros involucrados en las ecuaciones que describen el comportamiento temporal de las distintas alternativas incluidas en SEDAR.

6.1.3. Caracterización temporal de la estrategia de detección *SMCV*

SMCV comparte ciertas características con la estrategia *baseline*, como se comentó en la Sección 6.1.1. Como cualquier método basado en duplicación, apunta a lograr fiabilidad, a costa de asignar la mitad de los cores del sistema para proteger las ejecuciones. Sin embargo, debería quedar claro que el mecanismo de detección de *SMCV* se basa en lanzar una única instancia de la aplicación, con cada uno de sus procesos replicado internamente en un *thread*. Esto es diferente al *baseline*, donde dos instancias independientes de la aplicación se lanzan en paralelo; de todas formas, ambas estrategias hacen el mismo uso de la mitad de los cores disponibles, desde el punto de vista de la $performance$ de la aplicación.

El tiempo de ejecución de la estrategia de detección *SMCV*, en ausencia de fallos, está

Parámetro	Significado
T_{prog}	Tiempo de ejecución de dos instancias de la aplicación original en paralelo
T_{comp}	Tiempo que toma realizar una comparación semi-automática de los resultados; puede incluir el cálculo de un *hash*
T_{rest}	Tiempo necesario para realizar un *restart* manual de la aplicación. Un *restart* automático podría requerir menos; como simplificación, se consideran iguales
f_d	El factor de *overhead* debido al mecanismo de detección. Es dependiente de la aplicación y puede determinarse experimentalmente $0 < fd < 1$
X	Instante de detección de una fallo, expresado como una fracción del progreso de la aplicación. Random. $0 < X < 1$
n	Número de *checkpoints* almacenados durante toda la ejecución, dado un intervalo de *checkpoint*
t_{cs}	Tiempo requerido para almacenar un *checkpoint* de sistema
t_i	Intervalo de *checkpoint*. Puede ajustarse para minimizar el *overhead*
k	La cantidad de *checkpoints* adicionales que la aplicación debe retroceder hasta encontrar uno no corrompido para recuperar. Depende de la aplicación particular y de la latencia de detección
t_{ca}	Tiempo requerido para almacenar un *checkpoint* de nivel de aplicación. Debería ser menor a t_{cs}
T_{compA}	Tiempo requerido para validar un *checkpoint* de aplicación; puede incluir el cálculo de un *hash*

Tabla 6.1: *Nombre y significado de los parámetros involucrados en la caracterización temporal de las estrategias alternativas de SEDAR*

dada por la Ecuación 6.3. El tiempo es el mismo que para el caso del *baseline* (Ecuación 6.1), pero, en este caso, T_{prog} es afectado negativamente (es decir, incrementado) por el factor

f_d, que representa el *overhead* del mecanismo de detección. En tanto, si ocurre un fallo, el tiempo de ejecución resultante es el de la Ecuación 6.4. El primer término comprende lo que se ha ejecutado hasta el instante de la detección (X), sumado a la re-ejecución completa, luego de la parada causada por el error. Una vez detectado, se requiere un tiempo para el *restart*, y una comparación final en la re-ejecución.

$$T_{FA} = T_{prog}(1 + f_d) + T_{comp} \tag{6.3}$$

$$T_{FP} = T_{prog}(1 + f_d)(X + 1) + T_{rest} + T_{comp} \tag{6.4}$$

Es importante señalar que el parámetro X representa el instante de la detección del fallo, y no el momento de la ocurrencia (que no puede conocerse con exactitud). El valor de X se relaciona con la latencia de detección y depende de cómo los datos (y por lo tanto los mensajes) resultan afectados por el fallo; por ende, varía de acuerdo con el patrón de comunicación de la aplicación particular.

6.1.4. Caracterización temporal de la estrategia de recuperación basada en múltiples *checkpoints* de nivel de sistema

En ausencia de fallos, el tiempo de ejecución de este mecanismo de recuperación de SEDAR viene dado por la Ecuación 6.5. A comparación del caso de sólo-detección (Ecuación 6.3), el término extra da cuenta del tiempo necesario para almacenar n *checkpoints* de nivel de sistema. En tanto, cuando ocurre un fallo, el tiempo de ejecución es el de la Ecuación 6.6. El parámetro k es la cantidad extra de *checkpoints* que se deben "rebobinar" o retroceder si el *restart* desde el último fracasa. El tercer término representa el tiempo utilizado en hacer *checkpointing*, tomando en cuenta que, en las re-ejecuciones necesarias, se requerirá almacenar varios *checkpoints* nuevamente si el patrón de corrupción no permite el éxito de la recuperación. El cuarto término es una estimación del tiempo de re-ejecución con-

siderando que, en promedio, el error va a detectarse a la mitad del intervalo de *checkpoint*. En el mejor caso, este lapso requerirá ser re-ejecutado; es decir, cuando es posible recuperar desde el último *checkpoint* almacenado ($k = 0$). En tanto, si $k > 0$, esa misma fracción de tiempo, sumada a un número entero de intervalos de *checkpoint* (que depende del valor de k) requerirá de re-ejecución. Finalmente, el último término representa el tiempo consumido en la cantidad necesaria de *restarts*.

$$T_{FA} = T_{prog}(1 + f_d) + T_{comp} + nt_{cs} \tag{6.5}$$

$$\begin{aligned} T_{FP} = T_{prog}(1 + f_d) + T_{comp} + (n + k)t_{cs} + \\ +(\sum_{m=0}^{k}(k - m + 1/2))t_i + (k + 1)T_{rest} \end{aligned} \tag{6.6}$$

Se puede demostrar que el cuarto término de la Ecuación 6.6 es equivalente a:

$$(\sum_{m=0}^{k}(k - m + 1/2))t_i = \frac{(k + 1)^2}{2}t_i \tag{6.7}$$

Por lo tanto, la Ecuación 6.6 se puede reescribir de una forma más compacta como:

$$T_{FP} = T_{prog}(1 + f_d) + T_{comp} + (n + k)t_{cs} + \frac{(k + 1)^2}{2}t_i + (k + 1)T_{rest} \tag{6.8}$$

Un hecho destacable es que, como se explicó en la Sección 4.2.1, la estrategia de detección es igualmente efectiva cuando ocurren múltiples errores durante la ejecución, debido a que la primera diferencia observable, causada por un error, conduce al sistema a detenerse y recomenzar. Sin embargo, el mecanismo basado en cadena de *checkpoints* no se comporta de manera óptima a la hora de la recuperación. Desde el punto de vista de la respuesta temporal, el modelo se vuelve muy complejo si se quieren incluir los casos en que ocurran dos o más fallos, debido a que la cantidad de escenarios posibles se expande rápidamente, dificultando extremadamente el análisis y la predicción de los tiempos de ejecución resultantes. El trabajo requerido, no sólo para contemplar todas las posibles combinaciones, sino también para

plantear un mecanismo óptimo de recuperación, nos ha llevado, como consecuencia, a limitar el análisis del comportamiento temporal a los casos de ausencia de fallos u ocurrencia de un único fallo. Debido a que el mecanismo de detección es capaz de manejar múltiples errores no relacionados, el impacto se genera sobre la *performance*, pero no afecta la funcionalidad.

6.1.5. Caracterización temporal de la estrategia de recuperación basada en único *checkpoints* seguro de capa de aplicación

En ausencia de fallos, el tiempo de ejecución de este mecanismo de recuperación de SEDAR viene dado por la Ecuación 6.9. El tiempo es el mismo que el de la estrategia de sólo detección, pero incorpora un último término adicional, que representa el tiempo empleado para almacenar n *checkpoints* de capa de aplicación (t_{ca}) luego de validar cada uno (t_{compA}). En cambio, la Ecuación 6.10 cuantifica el tiempo de ejecución cuando ocurre un fallo. El cuarto término agregado muestra que, en promedio, se debe re-ejecutar la mitad del intervalo de *checkpoint*, ya que sólo se requiere un único *rollback*. Para ser claros: como cada *checkpoint* se valida, la latencia de detección está confinada dentro del intervalo de *checkpoint*. En el peor caso, el tiempo de re-ejecución sería t_i, si el error se detecta justo antes de almacenar un nuevo *checkpoint*; en tanto, en la mejor situación, en la que el error se detecta ni bien se ha guardado el *checkpoint*, el tiempo de re-ejecución sería cercano a 0. Debido a que la probabilidad del error está uniformemente distribuida en el intervalo de *checkpoint*, se establece que, en promedio, el tiempo de re-ejecución $(1/2)\ t_i$. Finalmente, el último término representa el tiempo consumido en el único intento de *restart* que se requiere, dado que el algoritmo realiza a lo sumo un único *rollback*. Es importante notar que T_{comp} representa el tiempo necesario para validar los resultados de la aplicación, mientras que T_{compA} comprende el lapso requerido para validar un *checkpoint* de capa de aplicación.

$$T_{FA} = T_{prog}(1 + f_d) + T_{comp} + n(t_{ca} + T_{compA}) \tag{6.9}$$

$$T_{FP} = T_{prog}(1 + f_d) + T_{comp} + n(t_{ca} + T_{compA}) + \\ +(1/2)t_i + T_{rest} \tag{6.10}$$

6.1.6. Tiempo promedio de ejecución

Como se ha discutido, las Ecuaciones 6.3 a 6.10 (excepto la 6.7) describen los tiempos utilizados por cada estrategia, tanto en ausencia de fallos como cuando ocurre un único fallo silencioso durante la ejecución. Como existen probabilidades de ocurrencias de los fallos, se introduce una formulación general que busca predecir el Tiempo Promedio de Ejecución (AET, *Average Execution Time*), tomando en cuenta esta probabilidad y, en consecuencia, el $MTBE$. Esta función permite estimar el *overhead* promedio (es decir, que tiene en cuenta tanto la ausencia como la presencia de los fallos) introducido por cada alternativa de SEDAR. Si α es la probabilidad de ocurrencia de un fallo, la función AET está dada por:

$$AET = T_{FP}(\alpha) + T_{FA}(1 - \alpha) \tag{6.11}$$

El $MTBE$ de un sistema con N procesadores decrece linealmente con N, es decir $MTBE = MTBE_{ind}/N$, donde $MTBE_{ind}$ es el $MTBE$ de un procesador individual [14]. Si $\lambda = 1/MTBE_{ind}$ es la tasa de fallos silenciosos de un procesador individual, la tasa de fallos silenciosos del sistema completo es λN. Si se asume que la tasa de arribos de los fallos está distribuida exponencialmente, la probabilidad de que un fallo silencioso afecte a un cómputo que dura T_{prog} y se ejecuta en un sistema con N procesadores es:

$$P(N, T_{prog}) = 1 - e^{-\lambda N T_{prog}} = 1 - e^{-N T_{prog}/MTBE_{ind}} = 1 - e^{-T_{prog}/MTBE} \tag{6.12}$$

Esta expresión representa la probabilidad de ocurrencia de un fallo silencioso en las condiciones mencionadas, es decir, α. Considerando esto en la Ecuación 6.11, es posible obtener el Tiempo Promedio de Ejecución como una función del $MTBE$ del sistema y del tiempo base de ejecución del programa T_{prog}.

$$AET = T_{FP}(1 - e^{-T_{prog}/MTBE}) + T_{FA}(e^{-T_{prog}/MTBE}) \quad (6.13)$$

La Ecuación 6.13 es útil para estimar el *overhead* promedio de utilizar cada estrategia de SEDAR, considerando tanto los casos de ausencia y presencia de fallos. Si se opta por la estrategia de detección de errores con notificación, los valores de T_{FA} y T_{FP} de la Ecuación 6.13 se obtienen de las Ecuaciones 6.3 y 6.4; cuando la recuperación basada en múltiples *checkpoints* de nivel de sistema es la estrategia adoptada, los valores se obtienen de las Ecuaciones 6.5 y 6.6 (o su equivalente 6.8); y cuando se elige utilizar recuperación basada en un único *checkpoint* seguro de capa de aplicación, los valores se obtienen de las Ecuaciones 6.9 y 6.10.

6.2. Evaluación del comportamiento temporal

Para poner de manifiesto la forma en la cual el modelo presentado se puede utilizar para evaluar el comportamiento temporal de cada estrategia alternativa, hemos desarrollado un conjunto de ejemplos sencillos, que incluyen valores reales para los parámetros que se listan en la Tabla 6.1, obtenidos a través de mediciones tomadas en las pruebas realizadas.

Para enriquecer el análisis, y poner en juego más variantes, se han utilizado tres *benchmarks* paralelos en la etapa de prueba: la multiplicación de matrices; el método de Jacobi para la ecuación de Laplace [6]; y el alineamiento de secuencias de ADN con el algoritmo de Smith-Waterman [125]. Estas aplicaciones fueron elegidas porque son bien conocidas, computacionalmente demandantes y representativas del cómputo científico. Además de esto, permiten estudiar los efectos de tener diferentes patrones de comunicación: $Master - Worker$, $Single - Program - Multiple - Data$ $(SPMD)$ y $Pipeline$, respectivamente [92]. La comparación de los tiempos de ejecución de las tres aplicaciones de prueba seleccionadas, entre las versiones MPI puras y nuestra implementación de SEDAR basada en MPI, tiene como objetivo medir los parámetros de ejecución que caracterizan a cada aplicación y puntualizar las diferencias debidas a los distintos patrones de comunicación. La implementación

de SEDAR incluye mecanismos para duplicar los procesos, sincronizar réplicas, comparar y copiar contenidos de mensajes y validar resultados finales. Cada experimento realizado fue repetido cinco veces, para obtener valores promedio.

El parámetro f_d, el factor de *overhead* introducido por el mecanismo de detección, se obtiene comparando el tiempo de ejecución del mecanismo de detección de SEDAR (en ausencia de fallos) con el tiempo de ejecutar la estrategia de detección manual (*baseline*), para cada aplicación particular. Explícitamente:

$$f_d = \frac{T_{SEDAR\_det\_FA} - (T_{prog} + T_{comp})}{T_{prog} + T_{comp}} \tag{6.14}$$

donde $T_{SEDAR\_det\_FA}$ es el tiempo de la Ecuación 6.3 y $(T_{prog} + T_{comp})$ es es el tiempo de la Ecuación 6.1.

Respecto del parámetro T_{comp}, se ha medido para un programa, lanzado manualmente, que compara los contenidos de dos archivos binarios; este lapso es similar al que transcurre para ejecutar la función $SEDAR\_Validate()$.

Un valor promedio del parámetro t_{cs} se obtuvo a través de medidas realizadas con herramientas provistas por *DMTCP*, que son específicas para tomar tiempos de funciones y eventos propios de la librería. En cuanto a T_{rest}, se midió indirectamente, realizando experimentos de inyección de fallos que demandaban recuperación; sin embargo, los valores obtenidos son consistentes con la presunción habitual de que el tiempo requerido para almacenar un *checkpoint* se puede considerar similar al tiempo requerido para cargar un archivo de *checkpoint* desde el medio de almacenamiento [56].

En la búsqueda de obtener ejecuciones de larga duración (del orden de las 10 horas para el *baseline*), de modo de ver claramente la incidencia de cada estrategia de SEDAR, se han ajustado ciertos parámetros de ejecución; fundamentalmente, la cantidad de iteraciones realizadas por la aplicación, y el tamaño del *workload*. La multiplicación de matrices se repitió $I = 100$ veces usando un tamaño de matriz $N = 8192$. Para la aplicación de Jacobi, el tamaño del *workload* fue $N = 8192$ and the number of iterations $I = 300.000$. Finalmente,

para el alineamiento de secuencias de ADN, la longitud de las secuencias fue configurado a $N = 4.194.304 = 2^{22}$.

Dentro de este contexto de ejecuciones largas, el parámetro t_i, el intervalo de *checkpoint*, se fijó a 1 hora para todos los experimentos. En cuanto a n, es el número de *checkpoints* que se han almacenado con ese valor de intervalo; se obtiene como el cociente entre el tiempo de la estrategia de detección pura (Ecuación 6.3) y el intervalo de *checkpoint* t_i. Es interesante notar que, en lugar de ser asignado arbitrariamente, el intervalo de *checkpoint* puede determinarse, por ejemplo, con la fórmula de Daly [38], que toma en cuenta el $MTBE$. Se intenta que el intervalo de *checkpoint* sea un $trade - off$ entre introducir un bajo *overhead* debido al *checkpointing*, y una cantidad razonable de trabajo que debe volver a realizarse si ocurre un fallo (un intervalo mayor tiende a alargar la latencia de detección).

El parámetro X, que depende de la latencia de detección, fue asignado manualmente para tres escenarios diferentes: cercano al comienzo, en la mitad y cercano al final de la ejecución. Por su parte, el parámetro t_{ca} se estima asumiendo que es más liviano (demanda menos tiempo) almacenar un *checkpoint* de capa de aplicación que su contraparte de nivel de sistema; esto es consistente con el hecho de que almacena menor cantidad de información. Por último, T_{compA} (el tiempo requerido para validar un *checkpoint* de capa de aplicación) ha sido estimado como similar al tiempo necesario para validar los resultados (es decir, T_{comp}), como una forma de simplificar el modelo.

La Tabla 6.2 contiene todos los valores utilizados para la evaluación temporal, obtenidos de la manera descripta, en tanto que en la Tabla 6.3 se muestran los tiempos resultantes en cada caso.

Para los experimentos se utilizaron dos nodos (hojas) de la plataforma descripta en la Sección 5.2.3. Se debe recalcar que, en el caso de la estrategia *baseline* (ejecución de dos instancias simultáneas independientes MPI), se lanzaron un total de 8 procesos MPI para cada instancia, con un máximo de 4 procesos asignados en cada nodo, lo que significa que

una instancia sólo utilizó 4 cores de cada uno de ambos nodos (y lo mismo la otra instancia). El mismo *mapping* fue asignado para la implementación de SEDAR (8 procesos MPI, con 4 en cada nodo) pero, debido a que los *threads* redundantes se ejecutan en los cores libres, SEDAR utilizó los 16 cores que en total están disponibles entre ambas hojas.

Un análisis de los datos contenidos en la Tabla 6.2 revela algunos datos significativos. En todos los casos, el factor de *overhead* de detección f_d es muy bajo; sin embargo, el mayor valor se da para el método de Jacobi, que es la aplicación con comunicaciones más frecuentes. Este comportamiento es esperable, ya que f_d está fuertemente relacionado con los mensajes (a mayor cantidad de mensajes, más sincronizaciones, validaciones y copias). En cambio, el producto de matrices, que es una aplicación limitada por cómputo, presenta un *overhead* de detección prácticamente despreciable.

Por otro lado, t_{cs} está directamente asociado con el tamaño del *workload* W que maneja cada aplicación, ya que el tiempo utilizado para almacenar los *checkpoints* está relacionado con la porción de información que se debe guardar, y por lo tanto con la cantidad de memoria consumida, como se puede ver en la Tabla 6.2. El producto de matrices es la aplicación que consume más memoria: el proceso $Master$ maneja las tres matrices completas, mientras que cada $Worker$ requiere de la matriz B completa y de sus correspondientes trozos de A y C. En cuanto a la aplicación de Jacobi, uno de los procesos maneja las dos matrices completas, mientras que los demás procesos manejan sus trozos correspondientes de ellas. Finalmente, en el algoritmo de Smith-Waterman, todos los procesos requieren $buffers$ locales; además, uno de los procesos maneja dos secuencias completas, mientras que los demás manejan una secuencia completa y sus trozos correspondientes de la otra. Por supuesto, para los tres *benchmarks*, todos los procesos están replicados. Los valores de t_{cs} son promedios de los tiempos que duraron cada unos de los *checkpoints* almacenados durante la ejecución; de todas formas, es un parámetro que ha mostrado una baja variabilidad, siendo los tamaños de los *checkpoints* y sus tiempos de almacenamiento aproximadamente constantes.

Por último, T_{comp} está asociado con el tamaño de los resultados que deben ser validados.

En la multiplicación de matrices, se valida la matriz C completa para cada iteración, por lo que el valor obtenido es considerable si se lo compara con el de las demás aplicaciones. En el otro extremo, en el alineamiento de secuencias de ADN, sólo se valida el puntaje (*score*) de similaridad, lo que conlleva un tiempo despreciable. A medio camino, en el método de Jacobi, sólo la matriz de resultado final requiere de validación.

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

Tabla 6.2: *Valores de los parámetros utilizados en la evaluación temporal de cada estrategia*

#	Situación	MATMUL	JACOBI	SW
1	*Baseline*, sin fallo (Ec. 6.1)	10.22	8.92	11.15
2	*Baseline*, con fallo (Ec. 6.2)	20.45	17.85	22.35
3	Sólo detección, sin fallo (Ec. 6.3)	10.23	8.97	11.16
4	Sólo detección, con fallo (Ec. 6.4, X = 30 %)	13.29	11.67	14.50
5	Sólo detección, con fallo (Ec. 6.4, X = 50 %)	15.33	13.46	16.73
6	Sólo detección, con fallo (Ec. 6.4, X = 80 %)	18.39	16.16	20.08
7	Múltiples *checkpoints*, sin fallo (Ec. 6.5)	10.26	9.00	11.17
8	Múltiples *checkpoints*, con fallo (Ec. 6.8, k = 0)	10.77	9.50	11.66
9	Múltiples *checkpoints*, con fallo (Ec. 6.8, k = 1)	12.27	11.01	13.17
10	Múltiples *checkpoints*, con fallo (Ec. 6.8, k = 4)	22.79	21.53	23.67
11	Único *checkpoint*, sin fallo (Ec. 6.9)	10.37	8.99	11.16
12	Único *checkpoint*, con fallo (Ec. 6.10)	10.87	9.50	11.66

Tabla 6.3: *Tiempos de ejecución [hs] para todas las estrategias de SEDAR, tanto en ausencia como en presencia de fallos, comparado con el baseline*

La información contenida en la Tabla 6.3 permite revisar algunos aspectos interesantes

del comportamiento de SEDAR. Como un hecho destacable, cuando ocurre un fallo, el mecanismo de detección (filas **4**, **5** y **6**) logra una mejor *performance* que el *baseline* (fila **2**) para todas las aplicaciones, independientemente del momento de la detección, debido al bajo *overhead* temporal de la estrategia. De cualquier manera, el mecanismo se comporta mejor cuanto antes se detecta el error, como es esperable, debido a que se debe rehacer menos trabajo una vez que se detiene la ejecución y se vuelve a comenzar.

En tanto, resulta bastante obvio que utilizar la estrategia de recuperación basada en múltiples *checkpoints* (fila **7**) implica un *overhead* mayor si se compara con la de sólo detección, en ausencia de fallos; el tiempo gastado en almacenar *checkpoints* vale más la pena en el caso de estas ejecuciones de larga duración, pero su impacto puede ser considerable en programas cortos.

El análisis de los valores de las filas **8**, **9** y **10** revela que, cuando ocurre un error, incluso el hecho de retroceder varias veces es ventajoso respecto al *baseline*; recién cuando el número de *rollbacks* es mayor a 4, el tiempo consumido en volver a realizar el trabajo se vuelve más prolongado que en la estrategia manual.

Los valores que se muestran en las filas **11** y **12** son similares a los de las filas **7** y **8**; como es esperable, el tiempo de recuperación desde el último (y válido) *checkpoint* de capa de aplicación (Ecuación 6.10) es prácticamente igual al tiempo de recuperación desde el último *checkpoint* de nivel de sistema, cuando esto es posible (Ecuación 6.8 con $k = 0$).

Como conclusión, se puede observar que el comportamiento temporal de cada estrategia de SEDAR es dependiente del patrón de comunicaciones, el *ratio* de cómputo a comunicaciones, la cantidad y frecuencia de datos que se transmiten y la latencia de detección. Los ejemplos descriptos, con los tres *benchmarks*, sirven como muestra de la forma en la se puede aplicar el modelo para la evaluación temporal si se dispone de los parámetros involucrados (o si pueden ser medidos).

Otro hecho remarcable que se puede observar de la Tabla 6.3 es que las diferentes alternativas de SEDAR pueden alcanzar mejoras considerables cuando se enfrentan a la ocurrencia

de un error silencioso, tanto en fiabilidad como en tiempo de ejecución, lo que resulta particularmente valioso en ejecuciones que pueden durar muchas horas. Más aún, cuanto más larga sea una ejecución (es decir, T_{prog}), más útil es la estrategia de tolerancia a fallos, porque la ocurrencia de los fallos se vuelve más probable (dado un $MTBE$ determinado). Como se ha mencionado, el mecanismo de protección debería ser utilizado para pogramas largos, ya que el *overhead* de guardar *checkpoints* es demasiado alto para aplicaciones breves.

Más allá de que estos ejemplos no se pueden tomar como conclusiones generales, son ilustrativos respecto del potencial de SEDAR para ayudar a los usuarios de aplicaciones científicas a lograr ejecuciones confiables, ya que son representativos de las aplicaciones de HPC.

6.3. Conveniencia de almacenar múltiples *checkpoints* para la recuperación

Como se ha explicado, en un sistema que almacena una cadena de *checkpoints*, la recuperación es siempre posible luego de uno o más intentos de *rollback*. Sin embargo, debido al *overhead* involucrado en hacer *checkpointing* y *rollback*, existen escenarios posibles en los cuales el tiempo que se gasta en esos intentos podría ser mayor al de simplemente detener la ejecución al momento de la detección y relanzar desde el comienzo. Por lo tanto, es útil evaluar en qué situaciones es conveniente almacenar múltiples *checkpoints* o, más aún, cuándo una protección basada en *checkpointing* resulta beneficiosa.

Esta pequeña discusión también sugiere cómo utilizar el modelo desarrollado, comparando el comportamiento temporal de la estrategia de sólo detección con la alternativa de recuperación basada en múltiples *checkpoints*. Si, para un sistema particular en el que se ejecuta una aplicación, existen estadísticas disponibles sobre la frecuencia y comportamiento típico de los fallos (es decir, cuándo es más probable que ocurran), se puede adaptar la estrategia de protección más apropiada entre sólo detección y *checkpoints* para recuperación.

Para diferentes valores del parámetro X, se puede extraer cierto conocimiento a partir

de la evaluación de los tiempos de ejecución dados por la Ecuación 6.4 y por la Ecuación 6.8. Para ilustrar esta idea, hemos elegido el método de Jacobi como aplicación de prueba; por supuesto, el procedimiento puede aplicarse fácilmente a los otros *benchmarks*; para este ejemplo, hemos evaluado la Ecuación 6.4 con tres valores diferentes de X: si el fallo se detecta cerca del comienzo ($X = 30\,\%$), por la mitad ($X = 50\,\%$) o cerca de la finalización de la ejecución ($X = 80\,\%$). Los tiempos obtenidos se comparan con los derivados de la Ecuación 6.8, tomando en cuenta las siguientes consideraciones.

El tiempo de referencia para este análisis es el de la Ecuación 6.3, que es la duración de una ejecución utilizando el mecanismo de detección, sin fallos. En este tiempo total (8.97 horas, como se ve en la Tabla 6.3), $X = 30\,\%$ significa que el fallo se detecta a $t = 2.69$ horas. Para ese momento, con $t_i = 1$ hora, sólo se han almacenado los dos primeros *checkpoints* ($CK0$ y $CK1$). Por lo tanto, es posible recuperar la ejecución, ya sea desde $CK1$ (es decir, $k = 0$) o desde $CK0$ (es decir, $k = 1$); de esta forma, esos dos valores de k son admisibles en la Ecuación 6.8. Este mismo razonamiento se aplicó para el resto de los valores of X.

La Tabla 6.4 resume estos datos. Basándose en el valor del parámetro X, la segunda columna muestra los tiempos obtenidos con la estrategia de sólo detección, parada segura y relanzamiento desde el comienzo (es decir, los de la Ecuación 6.4); la tercera columna muestra los tiempos obtenidos al retroceder al último *checkpoint* almacenado hasta el momento, (es decir, los de la Ecuación 6.8 con $k = 0$), la cuarta columna muestra los tiempos obtenidos al retroceder al anteúltimo *checkpoint* (es decir, los de la Ecuación 6.8 con $k = 1$), y así sucesivamente. El valor NA significa que el valor correspondiente de k no es admisible en la Ecuación 6.8 para el valor actual de X, debido a que el *checkpoint* correspondiente aún no ha sido almacenado para ese momento del progreso de la ejecución.

Para los valores analizados de X, los resultados obtenidos sugieren que retroceder al último *checkpoint* almacenado ($k = 0$), si es posible, siempre es conveniente frente a la opción de parar, notificar sobre el error y relanzar desde el comienzo. Incluso, recuperar desde el anteúltimo *checkpoint* ($k = 1$) sigue siendo conveniente (obviamente, si ese *checkpoint* no

X [%]	Sólo detección [hs]	$k+1$ intentos de *rollback* [hs]				
		k=0	k=1	k=2	k=3	k=4
30	11.66	9.5	11.01	NA (*NotAdmissible*)		
50	13.46			13.52	17.02	NA
80	16.16					21.53

Tabla 6.4: *Tiempos de ejecución con el fallo detectado en X, con sólo detección y con distinta cantidad de intentos de rollback (k+1)*

está corrompido). Sin embargo, si el fallo se detecta hacia la mitad de la ejecución, y se requieren dos o más *rollbacks*, hubiese sido preferible parar y relanzar. En otras palabras, si no es posible recuperar desde los dos últimos *checkpoints*, intentar retroceder más resulta más costoso que simplemente detenerse y volver a comenzar. Esto ocurre así debido al gran *overhead* que implica, no sólo re-ejecutar el mismo cómputo varias veces, sino también volver a guardar *checkpoints* durante las re-ejecuciones y hacer varios intentos de *restart*. Esta tendencia se mantiene también a medida que la aplicación progresa: si el fallo se detecta cerca de la finalización, incluso hacer varios reintentos de *rollback* a *checkpoints* previos puede representar una mejora, comparado con parar y relanzar.

Por supuesto, no hay forma de saber desde qué *checkpoint* resultará posible la recuperación. Sin embargo, continuando con esta línea de razonamiento, si se fuerza a que el tiempo dado por la Ecuación 6.4 sea menor o igual al de la Ecuación 6.8, con $k = 0$, se obtiene $X \leq$ 5,88 %. Esto significa que, antes de ese nivel de progreso, (que con los parámetros de la Tabla 6.1 representa unos 32 minutos) no es conveniente almacenar ningún *checkpoint*; resulta menos costoso parar y relanzar, simplemente. En cambio, si se fuerza al tiempo dado por la Ecuación 6.4 a ser mayor o igual al de la Ecuación 6.8, con $k = 1$, se obtiene $X \geq$ 22,67 %, (que representa aproximadamente 2 horas); esto significa que, recién cuando ha transcurrido ese porcentaje de tiempo de la ejecución, es preferible retroceder incluso hasta el anteúltimo *checkpoint* antes de detenerse y volver al comienzo. Antes de ese lapso, conviene almacenar sólo un único *checkpoint*. Este resultado refuerza la idea de que esta estrategia no es muy útil si el tiempo total de ejecución es demasiado corto: el tiempo que se requiere para almacenar el *checkpoint* podría no ser despreciable si se compara con el

intervalo de *checkpoint* requerido. Como ejemplo final, cuando $X \geq$ 50,61 %, volver atrás incluso hasta dos *checkpoints* representa una ventaja comparado con utilizar el mecanismo de sólo detección.

El conocimiento de la cantidad de *checkpoints* que vale la pena tener almacenados en un momento determinado permite también ahorrar espacio, ya que se pueden borrar los *checkpoints* a los que ya no valdrá la pena regresar a medida que se avanza en la ejecución. De esta forma el espacio de almacenamiento requerido tiende a mantenerse "constante".

Claramente, los resultados de esta breve discusión no pueden tomarse como conclusiones generales. Sin embargo, el análisis realizado muestra que (con estos parámetros), el *overhead* asociado con retroceder y volver a ejecutar es mucho más significativo que el bajo costo temporal de almacenar *checkpoints*. Esto sugiere que reducir el intervalo de *checkpoint* t_i puede resultar una opción conveniente, ya que el beneficio volver atrás un lapso acotado y rehacer poco trabajo excede al *overhead* de guardar *checkpoints* más frecuentemente.

Una vez más, si bien estos son ejemplos simples, también son ilustrativos respecto de cómo se puede extraer información de utilidad si se emplea el modelo de comportamiento temporal desarrollado; estas conclusiones permiten adaptar la estrategia de protección, basándose en el conocimiento de los parámetros del sistema.

6.4. Mediciones de *overhead*

Para cuantificar la incidencia de las distintas alternativas de SEDAR y su relación con las características de las aplicaciones, se han llevado a cabo algunas pruebas a pequeña escala, destinadas a describir los comportamientos temporales y los *overheads* introducidos. En este sentido, hemos comparado los modos de operación **(a)** y **(c)** mencionados en la Sección 5.1, tanto en ausencia como en presencia de fallos, en distintos escenarios de intervalos de *checkpointing* y momentos de inyección. De esta forma, se aporta información en la búsqueda de criterios que permitan alcanzar la configuración óptima de la herramienta según las características particulares del sistema.

6.4.1. Diseño de la experimentación

Los experimentos diseñados están destinados a determinar la incidencia de la incorporación de SEDAR a la ejecución de las aplicaciones paralelas MPI, mostrando además la eficacia de los mecanismos de detección y de recuperación automática, y sus tiempos de respuesta, en distintos escenarios en un entorno pequeño, de forma de realizar una prueba de concepto. Con este fin, adjuntamos SEDAR al código de las aplicaciones para integrar su funcionalidad, siguiendo el esquema descripto en la Sección 5.2.2; SEDAR se utilizó tanto en las variantes de sólo detección y relanzamiento automático (**SEDAR-det** en las Tablas 6.5 y 6.6) y de recuperación basada en *checkpoints* periódicos de nivel de sistema (**SEDAR-rec** en las Tablas 6.5 y 6.6). Una vez adaptado y compilado SEDAR junto a las aplicaciones, el funcionamiento de la detección es transparente para el usuario. En tanto, como los *checkpoints* periódicos son tomados desde el exterior, el mecanismo de recuperación también resulta transparente para el usuario. En situaciones reales, donde los fallos son aleatorios (y no inyectados desde el interior del código de la aplicación), los fallos se detectan y recuperan, aunque no se puede predecir exactamente el momento de la detección ni el punto de recuperación posible, por lo que la funcionalidad es correcta, pero los tiempos ya no son controlados ni predecibles. Estos dos son los modos de operación de SEDAR que se utilizarían en problemas reales de mayor escala.

Las aplicaciones de prueba seleccionadas son la multiplicación de matrices paralela $Master - Worker$ y el método de Jacobi para la Ecuación de Laplace ($SPMD$). Como se ha explicado, estas aplicaciones son elegidas a causa de representar diferentes paradigmas de programación paralela, pero también, fundamentalmente, por estar en situaciones prácticamente opuestas respecto del patrón de comunicaciones y las relaciones de cómputo a comunicación: mientras que el producto de matrices demanda gran cantidad de cómputo local y requiere poca cantidad de comunicaciones al inicio y al final de cada tarea, la aplicación de Jacobi realiza una gran cantidad de comunicaciones regulares y permanentes de menor volumen de datos.

En ambas aplicaciones, para los experimentos, se utilizan matrices cuadradas de 8192 x 8192 elementos. Además, en el caso del método de Jacobi, para alcanzar una duración del orden deseado (entre 5 y 10 minutos para la aplicación MPI pura, sin SEDAR), se realizan 5000 iteraciones del algoritmo.

Los experimentos consisten en medir los tiempos de ejecución de ambas variantes mencionadas de SEDAR, en ausencia de fallos en una primera instancia. En el caso de la estrategia de detección y relanzamiento automático, el objetivo es determinar el *overhead* introducido respecto de: (1) una ejecución sin ninguna protección y (2) una ejecución protegida manualmente por medio de ejecución simultánea de dos instancias independientes, es decir, el *baseline* descripto en la Sección 6.1.1. En tanto, en el caso del mecanismo de recuperación automática (y siempre en ausencia de fallos), se buscó establecer la relación entre el *overhead* y el intervalo de *checkpointing* t_i, midiéndolo para tres valores distintos: t_i = 60s, t_i = 90s, t_i = 120s.

En el caso de la multiplicación de matrices, para las duraciones de ejecución planteadas, la cantidad de *checkpoints* almacenados fue $N_{ckpt} = 6$, $N_{ckpt} = 4$ y $N_{ckpt} = 3$, respectivamente; en tanto, para la aplicación de Jacobi la cantidad de *checkpoints* almacenados fue $N_{ckpt} = 9$, $N_{ckpt} = 6$ y $N_{ckpt} = 4$, respectivamente. Estos intervalos fueron definidos, tomando en cuenta la breve duración total de estas ejecuciones para prueba de conceptos, buscando ofrecer distintas alternativas de compromiso entre la protección brindada y el *overhead* introducido, frente a distintos valores posibles del $MTBE$.

Por otra parte, con la variante de detección de SEDAR, se han realizado experimentos en presencia de fallos, contemplando tres casos de inyección: uno al comienzo de la ejecución, uno aproximadamente a la mitad, y uno hacia el final. Para la estrategia de recuperación, los experimentos para cada de estos tres casos de inyección se repitieron con cada uno de los intervalos de *checkpoint*. El objetivo de estas pruebas fue examinar el comportamiento de SEDAR en presencia de fallos, y poder obtener conclusiones respecto de la conveniencia de utilizar cada variante, en función de los probables momentos de ocurrencia de fallos y

del intervalo de *checkpoint* configurado. La información obtenida debe proporcionarle al usuario elementos que le permitan configurar SEDAR para funcionar de manera óptima en conjunto con la aplicación monitorizada.

6.4.2. Resultados experimentales

Nuevamente, los experimentos de este pequeño conjunto se llevaron a cabo utilizando dos nodos del *cluster* descripto en la Sección 5.2.3. Los resultados de las pruebas realizadas se resumen en las Tablas 6.5 y 6.6. En todos los casos se utilizaron 8 procesos, y los tiempos obtenidos son el promedio de 5 repeticiones de cada experimento particular. En ambas tablas, el tiempo de la fila **1** es el de la ejecución de la aplicación MPI pura, con la particularidad de que el *mapping* se realizó con 4 procesos en cada nodo, aun quedando cores libres, de forma de forzar la utilización de la red, para que la comparación con SEDAR (que replica los procesos y utiliza todos los cores) fuese más justa. En la fila **2** de ambas tablas, se muestra el tiempo de ejecutar el esquema de detección manual consistente en dos instancias independientes simultáneas de la aplicación, con el mismo *mapping* del caso anterior (ver Sección 6.2). La diferencia de tiempos con la ejecución original se debe a la competencia por los recursos entre los procesos de las dos instancias. Si ocurre un fallo en la ejecución de dos instancias simultáneas (por supuesto, para la estrategia de la fila **1** pasará desapercibido), se requiere una tercera ejecución. Para realizarla, sin modificar el *mapping*, se ejecutaría una sola instancia de desempate, igual a la de la primera fila. Por lo tanto, el tiempo de ejecución total sería la suma del *baseline* (fila **2**) con el valor de la fila **1** (es decir, sin competencia por los recursos). Se debe aclarar que esto sólo tiene en cuenta los tiempos de ejecución puros, sin contemplar el tiempo de comparación $offline$ de los resultados, ni el de volver a lanzar la ejecución. Además, debe notarse que el tiempo total es independiente del instante de ocurrencia del fallo, debido a que, en cualquier caso, la ejecución debe concluir para poder verificar la validez de los resultados.

En la fila **3** de ambas tablas se muestra el tiempo cuando se incorpora el mecanismo de

	Estrategia	Tiempo [s] según caso de inyección			
		Sin fallo	Al comienzo	Intermedio	Al final
1	**Sin SEDAR – 1 instancia**	329,57	-		
2	**Sin SEDAR – 2 instancias simultáneas**	373,16	702,73		
3	**SEDAR – det**	374,19	745,06	748,82	747,62
4	**SEDAR – rec (t_i = 60s); N_{ckpt} = 6**	440,76	2066,23	1000,12	477,35
5	**SEDAR – rec (t_i = 90s); N_{ckpt} = 4**	418,91	1569,85	822,93	459,63
6	**SEDAR – rec (t_i = 120s); N_{ckpt} = 3**	407,06	1303,28	895,41	454,22

Tabla 6.5: *Comparación de los tiempos de las distintas estrategias de SEDAR, con diferentes casos de inyección e intervalos de checkpoint, para la multiplicación de matrices*

	Estrategia	Tiempo [s] según caso de inyección			
		Sin fallo	Al comienzo	Intermedio	Al final
1	**Sin SEDAR – 1 instancia**	495,7	-		
2	**Sin SEDAR – 2 instancias simultáneas**	560,4	1056,1		
3	**SEDAR – det**	562,5	619,9	844,6	1069,7
4	**SEDAR – rec (t_i = 60s); N_{ckpt} = 9**	594,7	662,6	637,7	622,9
5	**SEDAR – rec (t_i = 90s); N_{ckpt} = 6**	583,2	640,1	597,5	656,6
6	**SEDAR – rec (t_i = 120s); N_{ckpt} = 4**	576,2	633,2	620,5	605,3

Tabla 6.6: *Comparación de los tiempos de las distintas estrategias de SEDAR, con diferentes casos de inyección e intervalos de checkpoint, para la aplicación de Jacobi*

detección de SEDAR. Se puede observar que el *overhead* introducido (debido a la duplicación de procesos, sincronización entre réplicas, comparación y copia de los contenidos de los mensajes y validación final de los resultados) es extremadamente bajo (menor al 0.3 % para la multiplicación de matrices y menor al 0.4 % para el método de Jacobi), aunque, en consonancia con los valores de la Tabla 6.1, es algo mayor en el caso de Jacobi. En tanto, en caso de fallo, el tiempo total, que proviene de ejecutar hasta el momento de la detección, parar y relanzar desde el comienzo, es aproximadamente el doble del de la ejecución sin fallos, para la multiplicación de matrices, y representa un *overhead* aproximado menor al 7 %, respecto de realizar una tercera ejecución (compárense los valores de las filas **2** y **3** de la Tabla 6.5, con fallos). Sin embargo, para cualquier aplicación, **SEDAR-det** ofrece la ventaja de incluir los tiempos de comparación de resultados y de relanzamiento automático, que en el caso del *baseline* deberían ser llevados a cabo por el usuario una vez terminado el procesamiento ($offline$). Por su parte, con el método de Jacobi, el comportamiento es muy diferente, ya que **SEDAR-det** tiende a funcionar mejor que el *baseline*, salvo cuando

el fallo se detecta muy cerca del final, manteniendo un *overhead* menor al 2 % (compárense los valores de las filas **2** y **3** de la Tabla 6.6, con fallos). Debido a que los comportamientos para las dos aplicaciones son muy distintos, se estudian por separado.

En primer lugar, se analiza lo que ocurre con la multiplicación de matrices. Si se observa la fila **3** del Tabla 6.5, se ve que el tiempo total de ejecución es prácticamente insensible al momento de inyección del fallos. Desde el punto de vista del funcionamiento de **SEDAR-det**, este es el peor escenario posible, ya que todos los fallos inyectados son detectados casi sobre el final de la ejecución. Esto se debe al comportamiento particular de la aplicación de prueba, y su interacción con el mecanismo de detección. Como se ha explicado, la multiplicación de matrices es fuertemente limitada por cómputo, y las fases de comunicación son breves y están ubicadas al comienzo y al final. Esto produce que, para los tres instantes de inyección que hemos seleccionado, la detección se produzca siempre durante la última comunicación colectiva ($SEDAR\_Gather$). Como consecuencia, cuando el fallo se inyecta sobre el final (al finalizar el cómputo y antes de dicha comunicación), la latencia de detección es baja; pero cuando el fallo se inyecta antes de comenzar la fase de cómputo, o durante ella, permanece latente por mucho tiempo antes de ser detectado.

En cambio, para el método de Jacobi, la observación de los valores de la la fila **3** de la Tabla 6.6 evidencia un comportamiento más lineal y acorde a lo esperable. La aplicación realiza comunicaciones frecuentes, por lo que, en promedio, la latencia de detección es menor. Por lo tanto, consistentemente con lo observado en la Tabla 6.1, el tiempo total de ejecución va creciendo a medida que el fallo tarda más en ser detectado, ya que se debe sumar el tiempo de una ejecución completa (fijo) a la fracción que haya llegado a realizarse de la ejecución original (variable). Como conclusión, cuanto más tempranamente puedan ser detectados los fallos (menor latencia), mejor será el desempeño de la estrategia de detección y relanzamiento; éste se ve favorecido por las aplicaciones cuyos procesos se comunican más frecuentemente.

En la primera columna de las filas **4** a **6** de ambas Tablas 6.5 y 6.6, se pueden obser-

var los tiempos correspondientes al agregado del mecanismo de recuperación **SEDAR-rec**, en ausencia de fallos. Se aprecia que el *overhead* es inversamente proporcional al intervalo de *checkpointing*. Este comportamiento es lógico, debido a que, si el intervalo es menor, aumenta el número de *checkpoints*, y por lo tanto se consume más tiempo en su almacenamiento.

En tanto, en el resto de las columnas de ambas tablas se muestran los tiempos de recuperación en presencia de los diversos momentos de inyección de fallos. Como característica común para ambas aplicaciones, puede notarse que el mecanismo permite recuperar correctamente y finalizar la ejecución en todas las ocasiones. Sin embargo, según la latencia de detección, la cantidad de reintentos luego de la cual la recuperación será exitosa es variable, y nuevamente, da origen a comportamientos distintos para cada aplicación.

En la multiplicación de matrices, sólo en el caso del fallo inyectado sobre el final, el tiempo de recuperación requerido es considerablemente menor que en la detección y relanzamiento automático desde el comienzo; pero en todos los demás, el almacenamiento de múltiples *checkpoints* para recuperar no representa una alternativa conveniente frente a simplemente detectar, parar y relanzar. Una vez más, este hecho está relacionado con la latencia de detección. Cuando la latencia es alta, los *checkpoints* que se almacenan mientras el fallo permanece latente están corruptos, y por lo tanto no son válidos para la recuperación. El algoritmo intenta recuperar sucesivamente hacia atrás desde cada uno de ellos, acumulando los tiempos involucrados en cada intento hasta encontrar un *checkpoint* válido. Como ejemplo, en la inyección intermedia (durante la fase de cómputo local) con intervalo de *checkpointing* t_i = 60s, al momento de la detección se han realizado 6 *checkpoints*; la recuperación se intenta sucesivamente, sin éxito, desde el 6, luego desde el 5 y por último desde el 4, rehaciendo cada vez todo el cómputo hasta que la detección vuelve a producirse (y rehaciendo también los *checkpoints* que ya se habían almacenado); la recuperación recién es exitosa desde el *checkpoint* 3. En cambio, en el caso de los fallos inyectados sobre el final, que son rápidamente detectados, la recuperación desde el último *checkpoint* (que se ha

realizado recientemente) es exitosa, mejorando notablemente el tiempo total. Por lo tanto, en una aplicación tan limitada por cómputo, la latencia de detección tiende a ser alta, y el mecanismo **SEDAR-det** tiende a resultar más conveniente que el **SEDAR-rec**.

En el caso del algoritmo de Jacobi, nuevamente, se observa un comportamiento diferente, causado por su relación de cómputo a comunicaciones. El tiempo total de ejecución, luego de la detección de un fallos, tiene una baja variabilidad, y es (en apariencia) casi independiente del momento de la inyección. Esta conducta sugiere que, como circulan mensajes permanentemente, los errores tienen una alta probabilidad de ser detectados rápidamente (es decir, la latencia de detección tiende a ser baja). Consecuentemente, también aumenta la probabilidad de encontrar *checkpoints* válidos cercanos, produciendo que la recuperación tienda a ser exitosa desde el último *checkpoint* almacenado; al bastar un solo intento (en general), el mecanismo **SEDAR-rec** tiende a resultar más conveniente que el **SEDAR-det**. Generalizando, para las aplicaciones que realizan comunicaciones frecuentes, la latencia de detección tiende a ser baja y, por lo tanto, también la cantidad de intentos de recuperación.

Como conclusión, se puede destacar la flexibilidad de SEDAR para ajustarse a las necesidades de un sistema particular, proveyendo diferentes posibilidades de cobertura (detección y relanzamiento automático o recuperación basada en múltiples *checkpoints*), que le permiten adaptarse para obtener un compromiso en la relación costo/beneficio. El intervalo de *checkpoint* óptimo y la frecuencia de validación de las comunicaciones pueden ser determinados a partir de la caracterización de la aplicación a proteger.

Capítulo 7

Conclusiones y trabajos futuros

Resumen

Este capítulo finaliza el presente trabajo, resumiendo, a modo de conclusión, las características más salientes de la metodología desarrollada, que fueron detalladas en los capítulos anteriores. Además, se enumeran algunas líneas que quedan abiertas, de modo de plantear el trabajo que dará continuidad a esta tesis.

7.1. Conclusiones

El camino hacia los sistemas de cómputo en la exa-escala presenta varios desafíos para las próximas generaciones. La obtención de garantías respecto de la confiabilidad de las ejecuciones y el manejo eficaz de los fallos son algunos de los principales, constituyendo una de las preocupaciones crecientes en el ámbito del HPC. En el futuro, se esperan mayores variedades y tasas de errores, intervalos o latencias de detección más prolongados y errores silenciosos. Se proyecta que, en los sistemas de exa-escala, los errores ocurran varias veces al día, y se propaguen de forma de generar desde caídas de procesos hasta corrupciones de resultados debidas a fallos no detectados. En este contexto, si se toman en cuenta los profundos efectos que un único fallo transitorio puede causar en todos los procesos que se comunican, la protección de las aplicaciones MPI a nivel de las comunicaciones es un método factible y efectivo para detectar y aislar la corrupción de datos, evitando su propagación.

En este trabajo se propone, diseña e implementa SEDAR, una metodología que permite detectar los fallos transitorios y recuperar automáticamente las ejecuciones, aumentando la fiabilidad y la robustez en sistemas en los que se ejecutan aplicaciones paralelas determinísticas de paso de mensajes, de una manera agnóstica a los algoritmos a los que protege. La metodología desarrollada está basada en la replicación de procesos para la detección, combinada con distintos niveles de *checkpointing* (de capa de sistema o de capa de aplicación) para la recuperación automática. El objetivo de SEDAR es ayudar a programadores y usuarios de aplicaciones científicas a obtener ejecuciones fiables, posibilitando su finalización con resultados correctos.

La detección se logra replicando internamente cada proceso de la aplicación en *threads* y comparando los contenidos de los mensajes entre los *threads* antes de enviar a otro proceso (monitorización de las comunicaciones); además, los resultados finales son validados (monitorización del cómputo local). Esta estrategia permite relanzar la ejecución desde el comienzo ni bien se produce la detección, sin necesidad de esperar hasta la conclusión incorrecta. Para conseguir recuperación, se usan *checkpoints* de nivel de sistema, pero, debido a que no se puede garantizar que un *checkpoint* particular no está corrompido por un fallo silencioso, se requiere el almacenamiento de múltiples *checkpoints*, y se implementa un mecanismo para volver a intentar la recuperación exitosa desde un *checkpoint* previo si el mismo error se detecta nuevamente. La última opción consiste en utilizar un único *checkpoint* de capa de aplicación (a medida del programa particular), que puede ser verificado para asegurar su validez como punto seguro de recuperación. Como consecuencia, SEDAR plantea tres alternativas complementarias: (1) sólo detección y parada segura con notificación del error; (2) recuperación basada en una cadena de *checkpoints* de nivel de sistema y (3) recuperación basada en un único *checkpoint* válido de capa de aplicación. Cada una de estas variantes alcanza una cobertura particular pero también tiene limitaciones inherentes y costos de implementación; la posibilidad de elegir entre ellas brinda flexibilidad para adaptar la relación costo/beneficio a los requerimientos de un sistema particular.

Este trabajo presenta una descripción de la metodología y los tiempos implicados en la utilización de cada variante. El estudio de su comportamiento temporal en ausencia y en presencia de fallos, inyectados en distintos momentos durante la ejecución, permite evaluar su desempeño y caracterizar el *overhead* asociado a su utilización. Se presenta un modelo que considera varios escenarios de fallos sobre una aplicación de prueba y sus efectos predecibles, y se realiza validación experimental sobre una implementación real, utilizando diferentes aplicaciones con patrones de comunicación disímiles. Considerando el *overhead* introducido, el modelo también establece las condiciones bajo las cuales vale la pena comenzar con la protección y almacenar varios *checkpoints* para la recuperación, en lugar de simplemente detectar, detenerse y relanzar.

También se introduce y describe el funcionamiento de la herramienta SEDAR, que proporciona tres modos de utilización: sólo detección y relanzamiento automático, y recuperación basada en múltiples *checkpoints* de capa de sistema, ya sean periódicos o disparados por eventos. Las posibilidades de seleccionar entre las estrategias de detección y de recuperación, configurando el modo de uso y el intervalo de *checkpoint* para adaptarse a los requerimientos de cobertura y máximo *overhead* permitido de un sistema particular, hacen que SEDAR resulte una metodología factible, viable y eficaz para la tolerancia a fallos transitorios en los entornos de HPC a los que se apunta como objetivo.

Entre las contribuciones más significativas, se pueden mencionar:

- La integración de una estrategia como la duplicación, que es efectiva para detectar los fallos, con las técnicas de C/R, cuya utilización normal se da en el campo de los fallos permanentes para brindar disponibilidad (es decir, garantizar que la aplicación finalice). Sin embargo, a partir de esta combinación de estrategias diferentes, se ha logrado asegurar tanto la finalización como la fiabilidad de los resultados.

- La descripción del comportamiento funcional en presencia de fallos, por medio de un modelo analítico que muestra la eficacia de la estrategia de detección y la validez del mecanismo de recuperación basado en múltiples *checkpoints* coordinados de nivel de

sistema. El modelo se basa en la predictibilidad de los datos que resultan afectados en cada etapa del cómputo y las comunicaciones, y por lo tanto, de las consecuencias de cada posible fallo.

- La verificación empírica de las predicciones del modelo, por medio de experimentos controlados de inyección de fallos, lo que permite evidenciar la fiabilidad provista por las estrategias de SEDAR. Para realizar esta verificación sobre un caso de prueba bien conocido, se diseñó un *workfault* completo que permite clasificar cuidadosamente todos los fallos posibles en escenarios, de acuerdo a sus efectos, latencia de detección y punto de recuperación.

- La implementación del mecanismo de detección y del algoritmo de recuperación basado en múltiples *checkpoints*, a partir de la librería *DMTCP*. Se detalla el trabajo experimental realizado para incorporar SEDAR a las aplicaciones de prueba. Una vez integrado, es posible lanzar los programas de usuario como tareas ordinarias, que se ejecutan en espacio de usuario y son transparentes al sistema de manejo del entorno de HPC.

- La caracterización temporal y la evaluación de los *overheads* introducidos para cada una de las tres estrategias alternativas. Esto implica que SEDAR fue añadido a la ejecución de tres *benchmarks* paralelos con diferentes patrones de comunicación, *ratios* de cómputo a comunicaciones y tamaños de *workload*. Para cada uno de ellos, se obtuvieron los parámetros de ejecución a partir de mediciones y estimaciones. Se pone de manifiesto que las distintas variantes de SEDAR ofrecen beneficios tanto en el tiempo total de ejecución como en la confiabilidad de los resultados, que representa un aspecto particularmente relevante en programas que pueden ejecutar por muchas horas.

- La introducción de una función que describe el tiempo de ejecución promedio, que es útil para estimar el *overhead* promedio de usar cada estrategia de SEDAR, conside-

rando tanto los casos de ausencia y presencia de fallos.

- Una evaluación cualitativa de la incidencia del patrón de comunicaciones sobre los parámetros de ejecución, y por lo tanto sobre el comportamiento temporal.

- La evidencia de la flexibilidad de SEDAR para adaptarse de forma de alcanzar un determinado compromiso entre costo y desempeño obtenido. Debido a que cada estrategia de SEDAR proporciona una cierta cobertura pero tiene limitaciones y costos de implementación, la posibilidad de elegir entre ellas permite ajustarse a los requerimientos de un sistema particular.

- Una discusión sobre los escenarios en los cuales es útil almacenar múltiples *checkpoints* para recuperación automática, en lugar de mantener en funcionamiento sólo el mecanismo de detección. Además, un breve análisis sobre la determinación del momento más conveniente para comenzar la protección; esto es ilustrativo sobre cómo utilizar el modelo de comportamiento temporal para extraer pautas respecto de cuándo es beneficioso emplear cada estrategia de SEDAR, brindando información útil para el usuario.

- La determinación de la relación costo/beneficio, es decir, la correlación entre la cantidad de recursos que se gastan en implementar la estrategia, y la ganancia de tiempos que se obtiene en el logro de la fiabilidad de la ejecución. Como conclusión, se pone de manifiesto la viabilidad y la eficacia de SEDAR para tolerar los fallos transitorios que se esperan en el futuro en los sistemas de HPC de la exa-escala.

Si bien existen varias líneas abiertas que dan lugar a posibles trabajos futuros, creemos haber cumplido el objetivo que nos planteamos al comienzo de nuestro trabajo, que fue, en principio, desarrollar una idea que se materializó en una metodología y en un prototipo que contiene la funcionalidad requerida desde el diseño, pero lejano aún de una herramienta real que funcione en entornos de producción. Nuestro foco principal ha sido validar esta

metodología, centrándonos especialmente en la perspectiva funcional, aunque sin dejar de lado los aspectos prestacionales, mediante un nivel razonable (pero no exhaustivo) de verificación. Hemos conseguido establecer un modelo analítico que sirve para validar la estrategia de tolerancia a fallos, y un procedimiento para implementarla en la práctica. Por lo tanto, hemos arribado a una metodología completa, que contempla de por sí todos los escenarios posibles de fallos, mientras que los experimentos nos han mostrado que se pueden integrar herramientas de software estándar (es decir, tecnologías existentes y conocidas) para llevar a cabo una implementación de nuestra propuesta. Hemos arribado a un prototipo de un sistema automático que brinda una cierta calidad de servicio, siendo capaz de almacenar los *checkpoints* y de recuperar sin intervención del usuario, garantizando así la fiabilidad de los resultados dentro de un tiempo factible de ser acotado.

7.2. Trabajos futuros

Creemos que hemos sentado las bases y dado los primeros pasos fundamentales en la definición de la metodología. Sin embargo, la resiliencia en el camino hacia la exa-escala es un tema muy amplio y con mucho campo para la investigación y el desarrollo. En ese contexto, tanto SEDAR como otras propuestas que se abren en la temática tienen potencial para continuar avanzando y obteniendo resultados. Las principales líneas abiertas que permiten delinear el trabajo que queda por delante, son:

- Ampliar la validación experimental, utilizando el algoritmo de recuperación basado en *checkpoints* no-coordinados de capa de aplicación en conjunción con aplicaciones que incluyen sus propios *checkpoints* no-coordinados a medida, que será la tendencia más marcada en el futuro.

  Los problemas asociados a los *checkpoints* de capa de sistema (básicamente, la falta de transparencia sobre sus contenidos internos), plantean dos opciones posibles a seguir: o se usan aplicaciones que ya traen sus *checkpoints*, o se diseñan *checkpoints* propios, que basados en el conocimiento de la aplicación, almacenan sólo lo que es relevante, y,

por lo tanto, se pueden validar; en cualquier caso, hay un análisis de la aplicación que no existe en las librerías de *checkpointing*. En las aplicaciones que tienen incorporados sus *checkpoints*, éstos forman parte de la aplicación, por lo que el tiempo de hacerlos es parte del tiempo de ejecución, y no cuenta como *overhead*.

- Si los *checkpoints* se hacen por tiempo, es decir, periódicamente, se debe modelar y definir la forma de calcular el intervalo óptimo de *checkpoint*, de modo de minimizar el *overhead* de ejecución pero también la cantidad de trabajo que se debe rehacer en la re-ejecución. En tanto, si los *checkpoints* se manejan sincronizadamente con eventos, se debe establecer el criterio según el cual se almacenan automáticamente. Todo esto implica cuantificar la relación entre la latencia de detección y el patrón de comunicaciones. Por lo tanto, también se requiere ampliar el trabajo experimental hacia aplicaciones con diferentes patrones de comunicación

- Refinar el mecanismo de recuperación basada en múltiples *checkpoints*, de forma de soportar de manera óptima la ocurrencia de varios fallos, y predecir la respuesta temporal cuando suceden dos o más errores distintos. Someter también al sistema a campañas de inyección de fallos aleatorios, para complementar la experimentación realizada por medio de inyección controlada.

- Implementar una estrategia de adaptación automática del mecanismo de recuperación, es decir, que la protección de la ejecución en curso mediante *checkpoints* comience dinámicamente, basándose en la determinación del momento a partir del cual vale la pena comenzar a proteger dicha ejecución, a partir de un razonamiento similar al expuesto en la Sección 6.3 (antes de ese instante, es más efectivo simplemente detectar y relanzar desde el comienzo).

- Formalizar un modelo matemático, basado en ecuaciones donde participan los parámetros de ejecución, que permita establecer un criterio para determinar en qué casos vale la pena utilizar la estrategia, tomando en cuenta el *overhead* en tiempo y el costo en

almacenamiento de hacer los *checkpoints* (cuando no ocurren fallos) en contraposición con el costo de detectar y recuperar cuando no se pone ninguna estrategia y ocurre el fallo.

- Implementar herramientas auxiliares que contribuyan con el usuario de las aplicaciones, brindándole, al terminar la ejecución, reportes de los fallos detectados, si han podido ser recuperados y cómo. En el futuro, esto puede servir como información estadística para realizar análisis.

- Extender la metodología y la implementación de forma de poder soportar diferentes tecnologías de *checkpointing*. Hasta ahora, las soluciones que hemos aportado en este trabajo se basan en la utilización de *checkpoints* coordinados de nivel de sistema (*DMTCP*) o no-coordinados (por proceso, en capa de aplicación). Sin embargo, existen también otras variantes, como los *checkpoints* semicoordinados o el *checkpointing* diferencial (un método que reduce la carga de Entrada/Salida de realizar *checkpoints* consecutivos actualizando sólo aquellos bloques de datos que han cambiado desde el almacenamiento del último *checkpoint* [75]), que no hemos tenido en cuenta hasta el momento.

- Implementar un mecanismo para la emulación de llamadas no determinísticas, requeridas para extender el horizonte de aplicaciones que se pueden proteger con SEDAR.

- Como objetivo final, buscar la integración con arquitecturas que utilizan C/R para tolerancia a fallos permanentes [32]. Debido a que SEDAR proporciona alternativas escalables para la detección y la recuperación, se podría alcanzar el soporte de ambos tipos de fallos con una única herramienta funcional para las plataformas que se proyectan en la exa-escala.

Un aspecto importante a tener en cuenta es el impacto de la resiliencia en el consumo energético. La reducción de la frecuencia de operación, una de las estrategias que se utiliza

para disminuir el consumo energético, redunda en tiempos de ejecución más prolongados, y en consecuencia, dado un valor de *MTBE*, se espera una mayor cantidad de errores, introduciendo un mayor *overhead*. Incluso, usualmente la reducción en la velocidad de operación se obtiene disminuyendo la tensión, lo cual empeora el escenario del *overhead* ya que se es más probable el aumento de las tasas de fallos [16].

Como consecuencia, los modelos de resiliencia deben tener en cuenta el problema del consumo energético. Una de las ideas, aplicables a sistemas que utilizan replicación (similar a SEDAR). Por ejemplo, en [3] se presenta un modelo computacional, denominado replicación por imitación (*Mimic Replication*), que proporciona tolerancia a errores *SDC* a partir de re-ejecución dinámica de los procesos que son vulnerables o propensos a tener sus datos contaminados. En el caso del artículo mencionado, la estrategia se aplica a la propagación de *SDC* a través de de aplicaciones de cálculos de plantilla (*stencil computation*), un tipo de *kernel* iterativo con patrón de comunicación estructurado que se encuentra en muchos problemas científicos y de ingeniería. Sin embargo, en general, los modelos actuales deben tener en cuenta el compromiso entre utilización de recursos, consumo energético y resiliencia.

Bibliografía

[1] Julien Adam, Maxime Kermarquer, Jean-Baptiste Besnard, Leonardo Bautista-Gomez, Marc Perache, Patrick Carribault, Julien Jaeger, Allen D Malony, and Sameer Shende. Checkpoint/restart approaches for a thread-based mpi runtime. *Parallel Computing*, 85:204–219, 2019.

[2] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, 2004.

[3] Anis Alazzawe and Krishna Kant. Mimic: Fast recovery from data corruption errors in stencil computations. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2019.

[4] Md Mohsin Ali, Peter E Strazdins, Brendan Harding, and Markus Hegland. Complex scientific applications made fault-tolerant with the sparse grid combination technique. *The International Journal of High Performance Computing Applications*, 30(3):335–359, 2016.

[5] Nawab Ali, Sriram Krishnamoorthy, Niranjan Govind, and Bruce 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*, pages 24–31. IEEE, 2011.

[6] Gregory Andrews. Scientific computing. In *Foundations of Multithreaded, Parallel and Distributed Computing*, chapter 11, pages 527–585. Addison-Wesley, 2000.

[7] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing

for cluster computations and the desktop. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[8] Guillaume Aupy, Anne Benoit, Thomas Hérault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. On the combination of silent error detection and checkpointing. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 11–20. IEEE, 2013.

[9] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[10] Rajanikanth Batchu, Yoginder S Dandass, Anthony Skjellum, and Murali Beddhu. Mpi/ft: a model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.

[11] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

[12] Leonardo Bautista-Gomez and Franck Cappello. Exploiting spatial smoothness in hpc applications to detect silent data corruption. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 128–133. IEEE, 2015.

[13] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pages 1–32, 2011.

[14] Anne Benoit, Aurélien Cavelan, Franck Cappello, Padma Raghavan, Yves Robert, and Hongyang Sun. Coping with silent and fail-stop errors at scale by combining replication and checkpointing. *Journal of Parallel and Distributed Computing*, 122:209–225, 2018.

[15] Anne Benoit, Aurélien Cavelan, Florina M Ciorba, Valentin Le Fèvre, and Yves Robert. Combining checkpointing and replication for reliable execution of linear workflows with fail-stop and silent errors. *International Journal of Networking and Computing*, 9(1):2–27, 2019.

[16] Anne Benoit, Aurélien Cavelan, Yves Robert, and Hongyang Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. *ACM Transactions on Parallel Computing (TOPC)*, 3(2):1–36, 2016.

[17] Anne Benoit, Thomas Hérault, Valentin Le Fèvre, and Yves Robert. Replication is more efficient than you think. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 89. ACM, 2019.

[18] Austin R Benson, Sven Schmit, and Robert Schreiber. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications*, 29(4):403–421, 2015.

[19] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 275–278, 2015.

[20] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[21] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. An evaluation of user-level failure mitigation support in mpi. In *European MPI Users' Group Meeting*, pages 193–203. Springer, 2012.

[22] Wesley Bland, Kenneth Raffenetti, and Pavan Balaji. Simplifying the recovery model of user-level failure mitigation. In *2014 Workshop on Exascale MPI at Supercomputing Conference*, pages 20–25. IEEE, 2014.

[23] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Ieee Micro*, 25(6):10–16, 2005.

[24] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[25] Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack J Dongarra. Correlated set coordination in fault tolerant message logging protocols. In *European Conference on Parallel Processing*, pages 51–64. Springer, 2011.

[26] Ron Brightwell, Kurt Ferreira, and Rolf Riesen. Transparent redundant computing with mpi. In *European MPI Users' Group Meeting*, pages 208–218. Springer, 2010.

[27] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 155–164, 2008.

[28] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. C 3: A system for automating application-level checkpointing of mpi programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 357–373. Springer, 2003.

[29] Jiajun Cao, Kapil Arya, Rohan Garg, Shawn Matott, Dhabaleswar K Panda, Hari Subramoni, Jérôme Vienne, and Gene Cooperman. System-level scalable checkpoint-restart for petascale computing. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 932–941. IEEE, 2016.

[30] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[31] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[32] Marcela Castro-León, Hugo Meyer, Dolores Rexachs, and Emilio Luque. Fault tolerance at system level based on RADIC architecture. *Journal of Parallel and Distributed Computing*, 86:98–111, 2015.

[33] Jonathan Chang, George A Reis, and David I August. Automatic instruction-level software-only recovery. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 83–92. IEEE, 2006.

[34] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 73–84. ACM, 2011.

[35] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[36] Cristian Constantinescu. Dependability benchmarking using environmental test tools. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pages 567–571. IEEE, 2005.

[37] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguezb, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 18–18. IEEE, 2006.

[38] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems*, 22(3):303–312, 2006.

[39] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing*, pages 162–171, 2011.

[40] Catello Di Martino, Zbigniew Kalbarczyk, and Ravishankar Iyer. Measuring the resiliency of extreme-scale computing environments. In *Principles of Performance and Reliability Modeling and Evaluation*, pages 609–655. Springer, 2016.

[41] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 94–104, 2009.

[42] Paul E Dodd and Lloyd W Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on nuclear Science*, 50(3):583–602, 2003.

[43] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of parallel computing*, volume 3003. Morgan Kaufmann Publishers San Francisco^ eCA CA, 2003.

[44] Jack Dongarra, Thomas Herault, and Yves Robert. Fault tolerance techniques for high-performance computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer, 2015.

[45] Jack J Dongarra. *Performance of various computers using standard linear equations software*. University of Tennessee, Computer Science Department, 1993.

[46] James Elliott, Mark Hoemmen, and Frank Mueller. Evaluating the impact of sdc on the gmres iterative solver. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1193–1202. IEEE, 2014.

[47] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626. IEEE, 2012.

[48] Elmootazbellah N Elnozahy and James S Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[49] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[50] Christian Engelmann and Swen 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)*, pages 15–17, 2011.

[51] Christian Engelmann, Hong H Ong, and Stephen L Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (PDCN)*, pages 189–194, 2009.

[52] Graham E Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J Dongarra. Process fault tolerance: Semantics, design

and applications for high performance computing. *The International Journal of High Performance Computing Applications*, 19(4):465–477, 2005.

[53] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and T Brightwell. rMPI: increasing fault resiliency in a message-passing environment. *Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488*, 2011.

[54] Kurt Ferreira, Jon Stearley, James H Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[55] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron 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*, page 78. IEEE Computer Society Press, 2012.

[56] Leonardo Fialho, Dolores Rexachs, and Emilio Luque. What is missing in current checkpoint interval models? In *2011 31st International Conference on Distributed Computing Systems*, pages 322–332. IEEE, 2011.

[57] Jian Fu, Qiang Yang, Raphael Poss, Chris R Jesshope, and Chunyuan Zhang. On-demand thread-level fault detection in a concurrent programming environment. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 255–262. IEEE, 2013.

[58] Amit Golander, Shlomo Weiss, and Ronny Ronen. Synchronizing redundant cores in a dynamic dmr multicore architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(6):474–478, 2009.

[59] Mohamed Gomaa, Chad Scarbrough, TN Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 98–109. IEEE, 2003.

[60] Mohamed A Gomaa and TN Vijaykumar. Opportunistic transient-fault detection. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 172–183. IEEE, 2005.

[61] Leonardo Arturo Bautista Gomez and Franck Cappello. Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *2015 IEEE International Conference on Cluster Computing*, pages 595–602. IEEE, 2015.

[62] Joao Gramacho, Dolores Rexachs, and Emilio Luque. A methodology to calculate a program s robustness against transient faults. In *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 645–651, 2011.

[63] Florian Haas, Sebastian Weis, Theo Ungerer, Gilles Pokam, and Youfeng Wu. Fault-tolerant execution on cots multi-core processors with hardware transactional memory support. In *International Conference on Architecture of Computing Systems*, pages 16–30. Springer, 2017.

[64] Doug Hakkarinen and Zizhong Chen. Algorithmic cholesky factorization fault recovery. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[65] Doug Hakkarinen and Zizhong Chen. Multilevel diskless checkpointing. *IEEE Transactions on Computers*, 62(4):772–783, 2012.

[66] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[67] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[68] Mark Frederick Hoemmen and Michael Allen Heroux. Fault-tolerant iterative methods. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2011.

[69] Kuang-Hua Huang and Jacob A Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[70] Saurabh Hukerikar, Keita Teranishi, Pedro C Diniz, and Robert F Lucas. Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *International Journal of Parallel Programming*, 46(2):225–251, 2018.

[71] Joshua Hursey, Richard L Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G Solt. Run-through stabilization: An mpi proposal for process fault tolerance. In *European MPI Users' Group Meeting*, pages 329–332. Springer, 2011.

[72] Dewan Ibtesham, Dorian Arnold, Patrick G Bridges, Kurt B Ferreira, and Ron Brightwell. On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. In *2012 41st international conference on parallel processing*, pages 148–157. IEEE, 2012.

[73] Hyeran Jeon and Murali Annavaram. Warped-dmr: Light-weight error detection for gpgpu. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 37–47. IEEE, 2012.

[74] Qiangfeng Jiang and D Manivannan. An optimistic checkpointing and selective message logging approach for consistent global checkpoint collection in distributed systems.

In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

[75] Kai Keller and Leonardo Bautista Gomez. Application-level differential checkpointing for hpc applications with dynamic datasets. *arXiv preprint arXiv:1906.05038*, 2019.

[76] Christopher LaFrieda, Engin Ipek, Jose F Martinez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 317–326. IEEE, 2007.

[77] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, and Bronis R de Supinski. Evaluating user-level fault tolerance for mpi applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 57–62, 2014.

[78] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in mpi applications. *The International Journal of High Performance Computing Applications*, 30(3):305–319, 2016.

[79] Ignacio Laguna, Martin Schulz, David F Richards, Jon Calhoun, and Luke Olson. Ipas: Intelligent protection against silent output corruption in scientific applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 227–238. IEEE, 2016.

[80] A Lesiak, Piotr Gawkowski, and Janusz Sosnowski. Error recovery problems. In *2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'07)*, pages 270–277. IEEE, 2007.

[81] Aiguo Li and Bingrong Hong. Software implemented transient fault detection in space computer. *Aerospace science and technology*, 11(2-3):245–252, 2007.

[82] Nuria Losada. Application-level fault tolerance and resilience in hpc applications. 2018.

[83] Guoming Lu, Ziming Zheng, and Andrew A Chien. When is multi-version checkpointing needed? In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pages 49–56. ACM, 2013.

[84] Aamer Mahmood and Edward J McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[85] Tatiana Martsinkevich, Omer Subasi, Osman Unsal, Franck Cappello, and Jesus Labarta. Fault-tolerant protocol for hybrid task-parallel message-passing applications. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 563–570. IEEE, 2015.

[86] Esteban Meneses and Laxmikant V Kalé. Camel: collective-aware message logging. *The Journal of Supercomputing*, 71(7):2516–2538, 2015.

[87] Sarah E Michalak, Kevin W Harris, Nicolas W Hengartner, Bruce E Takala, and Stephen A Wender. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, 2005.

[88] Subhasish Mitra, Ming Zhang, Norbert Seifert, TM Mak, and Kee Sup Kim. Soft error resilient system design through error correction. In *VLSI-SoC: Research Trends in VLSI and Systems on Chip*, pages 143–156. Springer, 2008.

[89] Konstantina Mitropoulou, Vasileios Porpodas, and Timothy M Jones. Comet: Communication-optimised multi-threaded error-detection technique. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10. IEEE, 2016.

[90] Diego Montezanti, A De Giusti, Marcelo Naiouf, Jorge Villamayor, Dolores Rexachs, and Emilio Luque. A methodology for soft errors detection and automatic recovery. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 434–441. IEEE, 2017.

[91] Diego Montezanti, Emmanuel Frati, Dolores Rexachs, Emilio Luque, Marcelo Naiouf, and Armando De Giusti. SMCV: a methodology for detecting transient faults in multicore clusters. *CLEI Electronic Journal*, 15(3):1–11, 2012.

[92] Diego Montezanti, Enzo Rucci, Dolores Rexachs, Emilio Luque, Marcelo Naiouf, and Armando De Giusti. A tool for detecting transient faults in execution of parallel scientific applications on multicore clusters. *Journal of Computer Science & Technology*, 14:32–38, 2014.

[93] Diego Miguel Montezanti, Dolores Rexachs del Rosario, Enzo Rucci, Emilio Luque, Marcelo Naiouf, and Armando Eduardo De Giusti. Sedar: Detectando y recuperando fallos transitorios en hpc. In *XXV Congreso Argentino de Ciencias de la Computación (Río Cuarto, 2019)*, 2019. En prensa.

[94] Diego Miguel Montezanti, Dolores Rexachs del Rosario, Enzo Rucci, Emilio Luque Fadón, Marcelo Naiouf, and Armando Eduardo De Giusti. Characterizing a detection strategy for transient faults in hpc. In *Computer Science & Technology Series. XXI Argentine Congress of Computer Science. Selected papers*, pages 77–90. Editorial de la Universidad Nacional de La Plata (EDULP), 2016.

[95] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[96] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. The soft error problem: An architectural perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243–247. IEEE, 2005.

[97] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th annual international symposium on computer architecture*, pages 99–110. IEEE, 2002.

[98] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 29–40. IEEE, 2003.

[99] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[100] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Efficient software-based fault tolerance approach on multicore platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 921–926. EDA Consortium, 2013.

[101] Nichamon Naksinehaboon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L Scott. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 783–788. IEEE, 2008.

[102] Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant 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*, page 7. ACM, 2013.

[103] Xiang Ni, Esteban Meneses, and Laxmikant V Kalé. Hiding checkpoint overhead in hpc

applications with a semi-blocking algorithm. In *2012 IEEE International Conference on Cluster Computing*, pages 364–372. IEEE, 2012.

[104] Ron A Oldfield, Sarala Arunagiri, Patricia J Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C Roth. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 30–46. IEEE, 2007.

[105] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 575–584. IEEE, 2007.

[106] Isil Oz and Sanem Arslan. A survey on multithreading alternatives for soft error fault tolerance. *ACM Computing Surveys (CSUR)*, 52(2):1–38, 2019.

[107] Javier Panadero, Alvaro Wong, Dolores Rexachs, and Emilio Luque. P3s: A methodology to analyze and predict application scalability. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):642–658, 2017.

[108] Angshuman Parashar, Anand Sivasubramaniam, and Sudhanva Gurumurthi. Slick: slice-based locality exploitation for efficient redundant multithreading. *ACM SIGOPS Operating Systems Review*, 40(5):95–105, 2006.

[109] Stefan Pauli, Manuel Kohler, and Peter Arbenz. A fault tolerant implementation of multi-level monte carlo methods. *Parallel computing: Accelerating computational science and engineering (CSE)*, 25:471–480, 2014.

[110] Grzegorz Pawelczak, Simon McIntosh-Smith, James Price, and Matt Martineau. Application-based fault tolerance techniques for fully protecting sparse matrix solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 733–740. IEEE, 2017.

[111] Diego Simón Pérez-Arroyo. Improving redundant multithreading performance for soft-error detection in hpc applications. 2018.

[112] Frances Perry, Lester Mackey, George A Reis, Jay Ligatti, David I August, and David Walker. Fault-tolerant typed assembly language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 42–53, 2007.

[113] James S Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[114] James S Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology: Systems & Applications*, 2(2):62–67, 1994.

[115] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. *IEEE Transactions on parallel and Distributed Systems*, 9(10):972–986, 1998.

[116] Vladimir Puzyrev and José María Cela. A review of block krylov subspace methods for multisource electromagnetic modelling. *Geophysical Journal International*, 202(2):1241–1252, 2015.

[117] Steven K Reinhardt and Shubhendu S Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*, pages 25–36. IEEE, 2000.

[118] George A Reis, Jonathan Chang, Neil Vachharajani, Shubhendu S Mukherjee, Ram Rangan, and David I August. Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 148–159. IEEE, 2005.

[119] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[120] Dolores Rexachs and Emilio Luque. High availability for parallel computers. *Journal of Computer Science & Technology (JCS&T)*, 10(3), 2010.

[121] Francesco Rizzi, Karla Morris, Khachik Sargsyan, Paul Mycek, Cosmin Safta, Bert Debusschere, Olivier LeMaitre, and Omar Knio. Ulfm-mpi implementation of a resilient task-based partial differential equations preconditioner. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, pages 19–26, 2016.

[122] Thomas Ropars, Tatiana V Martsinkevich, Amina Guermouche, André Schiper, and Franck Cappello. Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[123] Eric Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*, pages 84–91. IEEE, 1999.

[124] Enzo Rucci. *Evaluación de rendimiento y eficiencia energética de sistemas heterogéneos para bioinformática*. PhD thesis, Universidad Nacional de La Plata, 2016.

[125] Enzo Rucci, Armando De Giusti, and Franco Chichizola. Parallel smith-waterman algorithm for dna sequences comparison on different cluster architectures. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, volume 1, pages 666–672. WorldComp, 2011.

[126] Piyush Sao and Richard Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the*

Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, pages 1–8, 2013.

[127] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R de Supinski, Naoya Maruyama, and Satoshi Matsuoka. Fmi: Fault tolerant messaging interface for fast and transparent recovery. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1225–1234. IEEE, 2014.

[128] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009.

[129] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[130] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 69–78. ACM, 2012.

[131] Premkishore Shivakumar, Michael Kistler, Stephen W Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings International Conference on Dependable Systems and Networks*, pages 389–398. IEEE, 2002.

[132] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2008.

[133] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A Con-

nors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009.

[134] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[135] Jared C Smolens, Brian T Gold, Babak Falsafi, and James C Hoe. Reunion: Complexity-effective multicore redundancy. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 223–234. IEEE, 2006.

[136] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[137] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[138] Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. Expert: Effective and flexible error protection by redundant multithreading. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 533–538. IEEE, 2018.

[139] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000.

[140] Guang Suo, Yutong Lu, Xiangke Liao, Min Xie, and Hongjia Cao. Nr-mpi: a non-stop

and fault resilient mpi. In *2013 International Conference on Parallel and Distributed Systems*, pages 190–199. IEEE, 2013.

[141] TN Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 87–98. IEEE, 2002.

[142] Jack Wadden, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. Real-world design and evaluation of compiler-managed gpu redundant multithreading. *ACM SIGARCH Computer Architecture News*, 42(3):73–84, 2014.

[143] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. Proactive process-level live migration in hpc environments. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE, 2008.

[144] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 244–258. IEEE, 2007.

[145] Nicholas J Wang, Justin Quek, Todd M Rafacz, and Sanjay J Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks, 2004*, pages 61–70. IEEE, 2004.

[146] Philip M Wells, Koushik Chakraborty, and Gurindar S Sohi. Mixed-mode multicore reliability. *ACM SIGARCH Computer Architecture News*, 37(1):169–180, 2009.

[147] Gulay Yalcin, Osman Sabri Unsal, and Adrian Cristal. Fault tolerance for multi-threaded applications by leveraging hardware transactional memory. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 4. ACM, 2013.

[148] Joy Yeh, Grzegorz Pawelczak, James Sewart, James Price, A Avila Ibarra, Simon McIntosh-Smith, Ferad Zyulkyarov, Leonardo Bautista-Gomez, and Osman Unsal. Software-level fault tolerant framework for task-based applications. *Poster session at High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[149] Ying C Yeh. Design considerations in boeing 777 fly-by-wire computers. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, pages 64–72. IEEE, 1998.

[150] Jiaxin Yu, Dong Jian, Zhibo Wu, and Hongwei Liu. Thread-level redundancy fault tolerant cmp based on relaxed input replication. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pages 544–549. IEEE, 2011.

[151] Guozhen Zhang, Yi Liu, Hailong Yang, and Depei Qian. A lightweight and flexible tool for distinguishing between hardware malfunctions and program bugs in debugging large-scale programs. *IEEE Access*, 6:71892–71905, 2018.

[152] Yun Zhang, Jae W Lee, Nick P Johnson, and David I August. Daft: decoupled acyclic fault tolerance. *International Journal of Parallel Programming*, 40(1):118–140, 2012.

[153] Ziming Zheng and Zhiling Lan. Reliability-aware scalability models for high performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9. IEEE, 2009.