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

Diego Montezanti
Directores por UNLP: Armando De Giusti y Marcelo Naiouf
Directores por UAB: Dolores Rexachs del Rosario y Emilio Luque Fadón
Fecha de exposición: 18/03/2020
Facultad de Informática, Universidad Nacional de La Plata, Argentina
dmontezanti@lidi.info.unlp.edu.ar

1. Introducción

En primer lugar, se presenta la motivación de esta tesis. Luego se enuncian los objetivos y finalmente, se describen las contribuciones y limitaciones de esta investigación.

1.1 Motivación

La confiabilidad y, como consecuencia, la Tolerancia a Fallos (TF) se han vuelto aspectos críticos de relevancia creciente, en especial en el ámbito del Cómputo de Altas Prestaciones (HPC, por sus siglas en inglés), debido al considerable incremento en la probabilidad de que ocurran fallos de diferentes clases en estos sistemas [1]. Este incremento es consecuencia, fundamentalmente, de dos factores:

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

b) El rápido aumento del tamaño de los sistemas paralelos, 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.

En relación con estos aspectos de complejidad y tamaño, actualmente la escena de las supercomputadoras es dominada por los *clusters* de multiprocesadores, interconectados por redes de comunicaciones de alta velocidad, habiendo 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 [2].

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 [3]). 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.

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. En el ámbito de HPC, el riesgo de fallos transitorios aumenta con la cantidad de procesadores que trabajan juntos [4]. 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. Estudios recientes muestran que, a medida de que los sistemas de HPC continúan creciendo e incluyendo más componentes de hardware de distintos tipos, el Tiempo Medio Entre Averías (MTBF, por sus siglas en inglés) para una aplicación determinada disminuye, resultando en una tasa de fallos más alta en general [5]. Por lo tanto, parece claro que los sistemas futuros de la Exa-escala presentarán altas tasas de errores: está previsto que las grandes aplicaciones paralelas tengan que lidiar con fallos que ocurran cada pocos minutos, por lo que requerirán ayuda externa para progresar eficientemente [6]. Algunos reportes recientes han señalado a la Corrupción Silenciosa de Datos (SDC, por sus siglas en inglés) como el tipo de fallo más peligroso que se puede presentar durante la ejecución, 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 [21]. 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 [20].

Dado que las aplicaciones científicas, intensivas en cómputo, 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 se utilizan para explotar el poder de cómputo proporcionado por supercomputadoras carecen de soporte de tolerancia a fallos [2].

1.2 Objetivos

En este contexto, que conjuga altas tasas de fallos, resultados no fiables y altos costos de verificación, 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 a sus resultados, dentro de un tiempo predeciblemente acotado.

Con este objetivo, en los últimos años se ha diseñado y desarrollado la metodología **SEDAR** (*Soft Error Detection and Automatic Recovery* [7,8], a partir de una estrategia de detección llamada SMCV [9,10,11]), cuyo objetivo es proveer tolerancia a fallos transitorios en 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 [12,22] y monitorización de los envíos de mensajes y el cómputo local, que aprovecha la redundancia de hardware que es intrínseca a los *multicores*.

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 mecanismo de recuperación automática, basado en un único *checkpoint* seguro de capa de aplicación. El objetivo principal de esta 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 realiza una implementación práctica de un prototipo funcional (utilizando herramientas existentes y conocidas), y, a partir de las pruebas realizadas sobre ella, se caracteriza también el comportamiento temporal, es decir, la *performance* y el *overhead* introducido por cada una de las tres alternativas. Además se busca 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, 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. A diferencia de las estrategias específicas, que proporcionan resiliencia para ciertas aplicaciones, a costa de modificarlas, y que no cubren todos los fallos [6,13], **SEDAR** es esencialmente transparente y agnóstico respecto del algoritmo al cual protege.

1.3 Contribuciones y limitaciones

Durante el desarrollo de la tesis hemos arribado a una metodología completa y funcionalmente válida, que contempla todos los escenarios posibles de fallos, y a un prototipo de un sistema automático capaz de recuperar sin intervención del usuario, garantizando así la fiabilidad de los resultados dentro de un tiempo factible de ser acotado.

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 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 un 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 de **SEDAR**, mostrando los beneficios que ofrecen las distintas variantes, 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 tiempo obtenida junto 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. Otra

limitación es que, actualmente, **SEDAR** funciona únicamente para aplicaciones paralelas determinísticas, ya que 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 se comparan para detectar los fallos. El modelo actual de caracterización temporal no permite predecir la respuesta cuando suceden dos o más errores distintos. El mecanismo de recuperación basado en múltiples *checkpoints*, en su implementación actual, no soporta de manera óptima la ocurrencia de varios fallos no relacionados, por lo que requiere ser refinado. Finalmente, el desarrollo y la implementación actual de **SEDAR** soportan únicamente *checkpoints* coordinados de nivel de sistema o no-coordinados (por proceso, en capa de aplicación); en este sentido, requiere ser extendida de forma de poder soportar diferentes tecnologías, como el *checkpointing* diferencial o incremental.

2. Metodología SEDAR

SEDAR consiste en tres estrategias alternativas y complementarias para alcanzar cobertura completa frente a errores silenciosos: **(1)** sólo detección con notificación al usuario y relanzamiento automático; **(2)** recuperación basada en múltiples *checkpoints* de nivel de sistema; y **(3)** recuperación utilizando un único *checkpoint* seguro de capa de aplicación. A continuación se revisan los principales aspectos de la metodología, los cuales han sido publicados en congresos internacionales [7,8], revistas [9,10] y capítulos de libros [11].

2.1 Detección y Relanzamiento Automático

Para detectar fallos transitorios durante la ejecución de aplicaciones paralelas determinísticas, **SEDAR** valida los contenidos de los mensajes que se van a enviar, acotando así la latencia de detección, aislando el fallo y evitando que se propague a los demás procesos. Para ello, duplica en un *thread* cada proceso de la aplicación. Al alcanzar el instante de envío de un mensaje, las réplicas utilizan un mecanismo de sincronización para garantizar que se encuentran en el mismo punto, y allí se comparan los contenidos de los mensajes computados por ambos hilos; si hay coincidencia, sólo una réplica envía el mensaje (bajo la premisa de que no hay fallos en las comunicaciones), no congestionando la red. El proceso receptor espera a que su réplica alcance el mismo punto y le copia los mensajes que ha recibido para luego reanudar la ejecución. Por otro lado, los fallos cuyos efectos no se transmiten a otros procesos, pero que se propagan localmente hasta el final, son detectados en una operación de validación de resultados finales. En [9] se estudia compromiso entre la latencia de detección y la sobrecarga computacional involucrada en realizar validaciones frecuentes.

Este mecanismo posibilita detectar todos los fallos que causan SDC, en sus dos variantes: Corrupciones en Datos Transmitidos (TDC, por sus siglas en inglés) y Corrupciones en el Estado Final (FSC, por sus siglas en inglés), asegurando la fiabilidad de la ejecución. Además, bajo la premisa de que, en un sistema homogéneo dedicado, los tiempos de ejecución de dos réplicas que realizan el mismo cómputo deben ser similares, es posible detectar los Errores de Temporización (TOE, por sus siglas en inglés) al notar una diferencia considerable entre los tiempos de procesamiento de ambas réplicas [11], a causa posiblemente de un fallo silencioso. Por lo tanto, **SEDAR** permite la configuración de retardos programables de sincronización entre réplicas, para evitar que las aplicaciones ingresen en esperas infinitas. La Figura 1 esquematiza el funcionamiento del mecanismo de detección frente a la ocurrencia de un fallo.

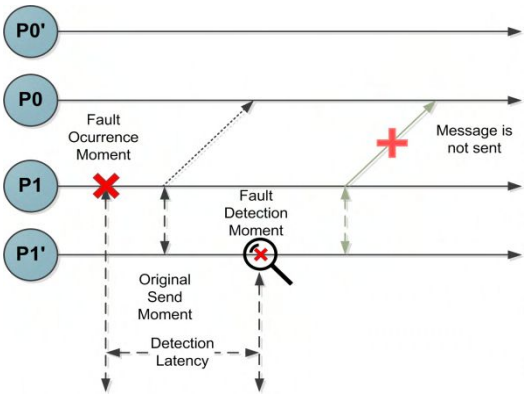

Fig. 1. Operación del mecanismo de detección frente a la ocurrencia de un fallo

El mecanismo de detección propuesto tiene situaciones de vulnerabilidad asociadas, que han sido descriptas previamente en [11]; sin embargo, las situaciones que lo afectan son extremadamente improbables, de modo que pueden ignorarse sin riesgos considerables.

En ausencia de un mecanismo de recuperación, la ocurrencia de un fallo de los tipos mencionados causa que la aplicación detenga su ejecución al momento de la detección, evitando así la costosa e innecesaria espera a que finalice con resultados incorrectos, y permitiendo su relanzamiento inmediato y automático desde el comienzo [7].

2.2 Recuperación Basada en Múltiples *Checkpoints* de Nivel de Sistema

El mecanismo de recuperación de **SEDAR** se basa en el almacenamiento de una cadena de múltiples *checkpoints* coordinados y distribuidos que se realizan por medio de una librería de *checkpointing* de capa de sistema. El hecho de guardar varios *checkpoints* es necesario debido a que, en el *checkpoint* de nivel de sistema, el usuario no tiene control sobre la información que se guarda. En consecuencia, frente a la perspectiva de los errores silenciosos, no hay garantía de que un *checkpoint* determinado sea un punto seguro de recuperación, debido a que sus datos pueden haber sido alterados por la ocurrencia de un fallo transitorio antes de ser almacenados. El *checkpoint* puede contener errores no detectados que permanecían latentes al momento de su almacenamiento. Por lo tanto, debe almacenarse una cadena de *checkpoints*, ya que previos al último no pueden ser descartados. En el caso de que se detecte un fallo, se debe determinar la factibilidad de recuperar la ejecución desde el último *checkpoint* (en el mejor caso) o desde alguno anterior [14]. Como casos particulares, si un fallo es detectado cuando aún no se ha almacenado ningún *checkpoint*, la aplicación debe ser relanzada desde el comienzo. En tanto, si en caso de detectar un fallo, se regresa a un *checkpoint* corrupto, en el intento de re-ejecución se vuelve a detectar el mismo fallo y es necesario retroceder hasta uno previo; esta situación puede ocurrir repetidas veces si la latencia de detección es muy alta [7]. En las Figuras 2 (a) y (b) se esquematizan estos comportamientos. El algoritmo de recuperación, en pseudocódigo, se muestra como Algoritmo 1.

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

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 e*xtern_counter* y, de acuerdo a él, buscar el *script* de reinicio correspondiente (entre todos los generados por la librería de c*heckpointing*). Además, el *checkpoint* inválido, que ha causado el *restart* erróneo, debe ser eliminado, sobreescribiéndolo y volviendo a almacenarlo nuevamente durante la re-ejecución).

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
```

Este método tiene como principal ventaja el ser transparente a la aplicación,

en el sentido de que no es necesario conocer su estructura interna, ni disponer de *checkpointing* a medida de la aplicación; además, el mismo mecanismo de C/R es el que ya se usa para los fallos permanentes [23]. Por lo tanto, la utilización de un mecanismo de detección basado en duplicación, y uno de recuperación basado en C/R, posibilita detectar y corregir fallos silenciosos sin la necesidad de recurrir a la triplicación con mecanismo de votación [15]. En tanto, sus limitaciones consisten, por un lado, en la cantidad de almacenamiento requerido, al no poder eliminar los *checkpoints* previos; por otro, en el gasto de tiempo involucrado en los múltiples intentos de reinicio posibles [14]; y, por último, en la baja escalabilidad de los *checkpoints* coordinados al crecer la cantidad de procesos, además del hecho de que almacenan gran cantidad de información relacionada al sistema [1].

(a) Baja latencia de detección: la recuperación se realiza desde el último *checkpoint*

(b) Alta latencia de detección: la recuperación se realiza desde el *checkpoint* anterior

Fig. 2. Posibles casos de recuperación utilizando múltiples *checkpoints* de nivel de sistema, dependiendo de la latencia de detección

2.3 Recuperación Basada en un Único *Checkpoint* de Capa de Aplicación

A pesar de requerir un conocimiento detallado de la estructura interna de la aplicación, los *checkpoints* de capa de aplicación constituyen una opción más adecuada, debido a 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 también un mecanismo de recuperación basado en un único *checkpoint* de capa de aplicación, en conjunto con una estrategia para garantizar la validez del último *checkpoint* guardado. Esto disminuye el tiempo y el trabajo asociados a la recuperación, a costa de requerir una validación de los datos contenidos en los *checkpoints* [7].

La solución propuesta se basa en almacenar *checkpoints* de aplicación (no coordinados, para mejorar la escalabilidad) a nivel de *thread* (cada *thread* guarda un *checkpoint* propio), aprovechando el mismo mecanismo de sincronización entre réplicas que se desarrolló en la fase de detección. Cuando uno de los *threads* llega a la instancia de almacenar un *checkpoint*, espera 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 ambos *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. En este escenario, el *checkpoint* previo puede descartarse sin peligro, reduciendo así el espacio de almacenamiento, ya que el *checkpoint* actual constituye un estado consistente y seguro para la recuperación de un error.

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 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, debiendo borrarse el *checkpoint* corrompido, y retomar la ejecución desde el anterior (que fue verificado). Como consecuencia, existe un único *checkpoint* válido almacenado en un momento determinado (excepto durante el intervalo de validación), independientemente del resultado de la comparación.

El mecanismo de recuperación propuesto se muestra como Algoritmo 2 (en pseudocódigo).

Cada una de estas alternativas tiene características particulares y provee un *trade-off* diferente entre costo y performance. La posibilidad de elegir entre estas alternativas brinda flexibilidad para adaptarse a la relación costo-beneficio requerida para el sistema particular.

3. Herramienta SEDAR y verificación funcional

En su estado actual, **SEDAR** permite 3 modos de utilización. El modo de **Sólo Detección y Relanzamiento Automático** permite evitar el coste asociado al almacenamiento de los *checkpoints*, posibilitando que la aplicación pueda ser relanzada desde el principio ni bien transcurre la latencia de detección. El modo **Recuperación Basada en *Checkpoints* de Nivel de Sistema Disparados por Eventos** es de utilidad cuando se tiene un conocimiento detallado del comportamiento de la aplicación a proteger. En este caso, **SEDAR** puede sintonizarse con la aplicación: 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 corruptos y de propagarlos. Por otra parte, seleccionar cuántos *checkpoints* se realizan (y en qué instantes) posibilita optimizar la relación costo/beneficio del mecanismo de protección, y mantener acotado el *overhead*. Este modo fue el utilizado principalmente en la etapa de validación funcional del mecanismo. Como desventaja, este mecanismo no es transparente (ni automático), en el sentido de que el código de los *checkpoints* debe insertarse dentro de la aplicación. Por último, el modo **Recuperación Basada en *Checkpoints* Periódicos de Nivel de Sistema** permite proteger a la aplicación "desde el exterior". 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 conocer la aplicación en profundidad, a costa de no tener control sobre el instante preciso en que se realiza el *checkpoint*, ni de desde dónde podrá recuperarse la ejecución. El *overhead* tendrá un comportamiento asociado al costo de almacenar un *checkpoint* y a la cantidad de ellos que se almacenen dado el intervalo de periodicidad. Como ventaja, esta protección pude realizarse de forma completamente automática.

El mecanismo de detección se implementa en la forma de una librería que modifica las funciones y los tipos de datos definidos en MPI (sintácticamente, sólo cambia el prefijo MPI por **SEDAR**), y agrega las funciones **SEDAR_Call** y **SEDAR_Validate**. Por otro lado, utiliza ***Pthreads*** para la replicación y sincronización de hilos. Para incorporar la detección, el código de la aplicación debe ser levemente modificado y recompilado, en un procedimiento automatizable.

La recuperación automática se implementa mediante la utilización de la librería DMTCP [16] (versión 2.4.4) que provee *checkpointing* coordinado y distribuido de nivel de sistema. Durante la instalación, puede configurarse su comportamiento para almacenar varios *checkpoints* (con números correlativos) y sus correspondientes *scripts* de reinicio, posibilidad en la que se basa el mecanismo de recuperación de **SEDAR**. DMTCP genera un proceso coordinador, que monitorea la aplicación objetivo y es capaz de realizar los *checkpoints* periódicos desde el exterior, lo cual produce que el mecanismo de recuperación sea transparente para la aplicación [7].

Para 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. Obviamente, existen innumerables posibilidades de 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 fue 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, se agrega el almacenamiento de un *checkpoint* de nivel de sistema cada vez que aplicación acaba de realizar una comunicación, debido a que los mensajes circulan sólo cuando los datos involucrados han sido validados (maximizando la probabilidad de datos confiables). El conocimiento detallado sobre el comportamiento de la aplicación de prueba permite identificar 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. 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, obteniendo para cada experimento el comportamiento esperado, y verificando así el funcionamiento correcto de los mecanismos implementados en **SEDAR**.

4. Caracterización Temporal

Una vez realizada la validación funcional, el foco se puso en los aspectos prestacionales. Para ello, se realizó una evaluación cuantitativa del comportamiento temporal de las tres estrategias que componen la metodología. Por un lado, se desarrolló un modelo matemático que toma en cuenta los parámetros que influyen sobre el tiempo de ejecución de cada una de las alternativas. El modelo se materializa en la forma de ecuaciones que pueden utilizarse para estimar el tiempo de ejecución para cada caso, tanto en ausencia de fallos como cuando ocurre un único fallo durante la ejecución, si se pueden medir u obtener valores aproximados para los parámetros mencionados.

Para poner de manifiesto la forma en la cual el modelo presentado se puede utilizar para evaluar el comportamiento temporal de cada estrategia alternativa, se montaron experimentos en los cuales **SEDAR** fue utilizado en conjunto con tres *benchmarks* paralelos con diferentes patrones de comunicación y demandas de *workload*: la multiplicación de matrices; el método de Jacobi para la

solución de la ecuación de Laplace [17]; y el alineamiento de secuencias de ADN con el algoritmo de Smith-Waterman [18]. Estas aplicaciones fueron elegidas porque son bien conocidas, computacionalmente demandantes y representativas del cómputo científico. Además, presentan diferentes patrones de comunicación: *Master/Worker*, SPMD y *Pipeline*, respectivamente [10]. 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, permitió obtener, en base a medidas y estimaciones, los parámetros que caracterizan la ejecución de cada aplicación. A su vez, la obtención de estos parámetros posibilitó evaluar cualitativamente la incidencia del patrón de comunicaciones, el intervalo de *checkpoint* y la latencia de detección sobre la *performance*.

Como resultado, se pudo 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 (que afecta al *overhead* de detección) y de la latencia de detección (cuanto más avanzada la ejecución al momento de la detección, más costoso es relanzar desde el comienzo sino hay ningún *checkpoint* válido cercano disponible). 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 es una ejecución, más útil es la estrategia de tolerancia a fallos, porque la ocurrencia de los fallos se vuelve más probable, en tanto que el *overhead* de guardar *checkpoints* es demasiado alto para aplicaciones breves.

A pesar de que en un sistema que almacena una cadena de *checkpoints*, la recuperación es siempre posible luego de uno o más intentos de *rollback*, existen escenarios posibles en los cuales el tiempo que se gasta en esos intentos puede ser mayor al de simplemente detener la ejecución al momento de la detección y relanzar desde el comienzo, considerando el *overhead* introducido al hacer *checkpointing* y *rollback*. Por lo tanto, se evaluaron las situaciones en las que es conveniente almacenar múltiples *checkpoints* en lugar de activar sólo el mecanismo de detección, extrayendo información útil del modelo de comportamiento temporal desarrollado: si para un sistema particular existen estadísticas disponibles sobre la frecuencia y comportamiento típico de los fallos, es posible optar por la estrategia de protección más apropiada, basándose en el conocimiento de los parámetros del sistema. Si el fallo se detecta cerca de la finalización, incluso hacer varios reintentos de *rollback* puede representar una mejora, comparado con parar y relanzar. El conocimiento de la cantidad de *checkpoints* que conviene 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 avanza la ejecución, manteniendo constante el espacio de almacenamiento requerido. El *overhead* asociado con retroceder y volver a ejecutar es mucho más significativo que el costo temporal de almacenar *checkpoints*, por lo que el beneficio volver atrás un lapso acotado y rehacer sólo una parte del trabajo excede al *overhead* de guardar *checkpoints* más frecuentemente.

Por último, para cuantificar el efecto de aplicar las distintas alternativas de **SEDAR**, y su relación con las características de las aplicaciones, se realizaron mediciones del *overhead* de ejecución en escenarios realistas. Para ello, se han comparado los modos **Sólo Detección y Relanzamiento Automático** y **Recuperación Basada en Checkpoints Periódicos de Nivel de Sistema**, tanto en ausencia de fallos como con fallos inyectados en distintos puntos durante la ejecución, y con distintos intervalos de *checkpoint*, buscando ofrecer distintas alternativas de compromiso entre la protección brindada y el *overhead* introducido, frente a distintos valores posibles del Tiempo Medio Entre Errores (MTBE, por sus siglas en inglés, el equivalente a MTBF pero para errores silenciosos). De esta forma, se busca aportar información que permita alcanzar la configuración óptima de la herramienta según las características particulares del sistema, cuando se la utiliza, integrada de forma transparente, en entornos y problemas de mayor escala. Este análisis se realizó con el producto de matrices, que

demanda gran cantidad de cómputo local y requiere pocas comunicaciones al inicio y al final de cada tarea, y con la aplicación de Jacobi, que realiza una gran cantidad de comunicaciones regulares y permanentes pero con menor volumen de datos. Como conclusiones, para aplicaciones limitadas por cómputo la latencia de detección tiende a ser elevada, por lo que pueden requerirse varios intentos de recuperación; en esa situación, la estrategia de sólo detección y relanzamiento automático puede ser beneficiosa. En tanto, 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; en ese escenario resulta conveniente el almacenamiento de múltiples *checkpoints*. Por lo tanto, se puede destacar la flexibilidad de **SEDAR** para ajustarse a las necesidades de un sistema particular, proveyendo diferentes posibilidades de cobertura 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 las características de la aplicación a proteger.

5. Conclusiones y trabajos futuros

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. Con el objetivo de ayudar a programadores y usuarios de aplicaciones científicas, hemos arribado a un prototipo de un sistema automático que brinda una calidad de servicio, siendo capaz de almacenar los *checkpoints* y de recuperar sin intervención del usuario, garantizando así la finalización de las ejecuciones con resultados correctos dentro de un tiempo factible de ser acotado.

Las principales líneas abiertas son:

- Ampliar la validación experimental, utilizando el algoritmo de recuperación basado en *checkpoints* no-coordinados con aplicaciones que incluyen sus propios *checkpoints* a medida, que será la tendencia más marcada en el futuro.
- Calcular el intervalo óptimo de *checkpoint*, de modo de minimizar el *overhead* de ejecución pero también la cantidad de trabajo que debe volverse a hacer en la re-ejecución, cuantificando la relación entre la latencia de detección y el patrón de comunicaciones.
- 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 no relacionados.

- Implementar una estrategia de adaptación dinámica del mecanismo de recuperación, y herramientas auxiliares que le brinden al usuario reportes de los fallos detectados, si han podido ser recuperados y cómo, de modo de servir como información estadística para realizar análisis posteriores.
- Integrar **SEDAR** con arquitecturas que utilizan C/R para tolerancia a fallos permanentes [19]. Debido a que **SEDAR** proporciona detección y la recuperación, se podrían soportar ambos tipos de fallos con una única herramienta funcional para las plataformas que se proyectan en la Exa-escala, y tomar en cuenta el impacto del consumo energético sobre la resiliencia.

Referencias

1. T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and J. Labarta, “Fault-tolerant protocol for hybrid task-parallel message-passing applications," in 2015 IEEE International Conference on Cluster Computing , pp. 563-570.
2. Nuria Losada. Application-level fault tolerance and resilience in HPC applications. 2018.
3. Dolores Rexachs and Emilio Luque. High availability for parallel computers. Journal of Computer Science & Technology (JCS&T), 10(3), 2010.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. G. Lu, Z. Zheng, and A. A. Chien, “When is multi-version checkpointing needed?" in Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale. ACM, 2013, pp. 49-56.
15. F. Mathur and A. Avizienis, “Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair”, in Proc. Spring Joint Computer Conference (AFIPS '70), New York, USA, May. 1970, pp. 375 – 383.
16. Ansel, J., Arya, K., & Cooperman, G. (2009, May). DMTCP: Transparent checkpointing for cluster computations and the desktop. In 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-12). IEEE.
17. Gregory Andrews. Scientific computing. In Foundations of Multithreaded, Parallel and Distributed Computing, chapter 11, pages 527–585. Addison-Wesley, 2000.
18. 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.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.