Propuestas para Integrar la Arquitectura RADIC de Forma Transparente

Hugo Meyer[1], Marcela Castro León [1], Dolores Rexachs[1] y Emilio Luque[1]

[1] Departamento de Arquitectura de Computadoras y Sistemas Operativos, Universitat Autònoma de Barcelona,
08193 Bellaterra (Cerdanyola del Vallès), España
{hugo.meyer, marcela.castro}@caos.uab.es, {dolores.rexachs, emilio.luque}@.uab.es

Resumen. El aumento en tamaño de los clústeres de computadores trae consigo un incremento en la tasa de fallos. En este trabajo se presentan dos propuestas de integración de la arquitectura RADIC, una a nivel de librería de comunicaciones y otra a nivel de protocolo de red, específicamente a nivel de *sockets*. Dado que MPI es un estándar que frente a fallos propone realizar una parada (*fail-stop)*, RADIC se encarga de enmascarar los fallos para que la aplicación termine, para ello utiliza un controlador distribuido que protege el cómputo utilizando estrategias de *rollback-recovery*. Resultados iniciales demuestran RADIC puede integrarse en diferentes capas del sistema para que actúe de forma transparente, que la arquitectura propuesta escala correctamente con la aplicación y que los *overheads* dependen de la configuración del sistema y del comportamiento de la aplicación. Además incluyendo nodos *spare* para recuperar procesos fallados evita la sobrecarga en nodos de cómputo y mantiene las prestaciones similares a las iniciales.

Palabras clave: RADIC, MPI, tolerancia a fallos, descentralización, nodos *spare*.

1 Introducción

En los computadores se han ido añadiendo y agrupando cada vez más y más componentes en pequeños espacios lo cual ha traído consigo problemas de refrigeración o de disipación. Estos problemas tienen como consecuencia un incremento en el número de fallos de hardware que llevan a una parada segura de las aplicaciones paralelas en ejecución. Este incremento en la tasa de fallos afecta de manera considerable el rendimiento de las aplicaciones de *High Performance Computing* cuando estas se ejecutan durante varios días, dado que un fallo lleva a la re-ejecución de las mismas. Servicios que funcionan 24/7, no puede permitirse paradas en la ejecución de trabajos, por lo cual se hace importante gestionar los fallos que ocurren.

Los mecanismos de tolerancia a fallos se hacen cada vez más necesarios, si analizamos estadísticas de centros de cómputo podemos encontrar que en los supercomputadores actuales ocurren fallos en promedio dos veces al día [1]. Es deseable la utilización de arquitecturas de tolerancia a fallos que introduzcan el menor

This research has been supported by the MICINN Spain, under contract TIN2007-64974

overhead posible al dar protección a las aplicaciones, y además maximizar el tiempo medio de interrupción en el sistema.

Se han desarrollado muchas técnicas que aumentan la fiabilidad y disponibilidad a los sistemas paralelos y distribuidos. Las técnicas más utilizadas incluyen la utilización de transacciones, replicación y técnicas basadas en *rollback-recovery* [2]. Cada una de estas técnicas implica diferentes relaciones de compromiso y enfocan la tolerancia a los fallos de diferentes maneras.

En este trabajo se parte de una arquitectura tolerante a fallos basada en protocolos de *rollback-recovery* como base. RADIC (*Redundant Array of Distributed and Independant fault tolerance Controllers*) [3] es una arquitectura que provee tolerancia a fallos a sistemas que utilizan paso de mensajes. La arquitectura RADIC actúa como una capa que aísla la aplicación paralela de los fallos en los nodos del sistema, enmascarando y tolerando fallos sin necesitar recursos extras para cumplir con sus objetivos.

RADIC, que es una arquitectura transparente, descentralizada, escalable y flexible, se ha diseñado e implementado para proveer tolerancia a fallos a aplicaciones que utilizan el estándar MPI [4]. Según este estándar, cuando ocurre un fallo, la aplicación finaliza, sin embargo RADIC busca continuar con la ejecución de la aplicación inclusive en presencia de fallos evitando la pérdida del cómputo y mejorando el MTTI (tiempo medio hasta la interrupción).

Figura 1. Capas de RADIC que proveen tolerancia a fallos de manera transparente a las aplicaciones

En este trabajo se propone integrar la arquitectura RADIC de un modo transparente en un sistema paralelo, permitiendo seleccionar en que capa integrar la tolerancia a fallos, el sistema de paso de mensajes o el sistema operativo. Considerando las capas de un sistema paralelo (Figura 1), se puede integrar RADIC:

- En la librería de comunicación, en este trabajo se ha diseñado específicamente para Open MPI. Una arquitectura de tolerancia a fallos a nivel de librerías da transparencia a las aplicaciones que se utilizan con dicha librería, y evita lidiar con los protocolos de comunicación de capas inferiores (Ethernet, Infiniband, etc.), pero para proveer estas ventajas se debe utilizar una librería específica.
- A nivel de sistema operativo, específicamente a nivel de *sockets* y de esta manera tener una implementación que puede ser utilizada con cualquier librería de comunicaciones MPI, manteniendo las cualidades iniciales, pero funcionando con sistemas operativos que utilicen el estándar POSIX de comunicaciones.

En la siguiente sección se presenta la arquitectura RADIC y en la sección 3 se ubica y compara a la misma con otros trabajos. En la sección 4 se trata la inclusión de RADIC en Open MPI y en la sección 5 los resultados experimentales de esta implementación, por último en la sección 6 se presentan las conclusiones.

2 RADIC

La arquitectura RADIC [3] está basada en un protocolo de *checkpoint* no coordinado combinado con log pesimista basado en el receptor **[2]**. Los datos críticos, como *checkpoints* y logs de un proceso de la aplicación son guardados en otro nodo diferente de aquel en el que se ejecuta el proceso. RADIC por medio de sus políticas provee: Transparencia, Descentralización, Escalabilidad, Flexibilidad.

Dos entidades conforman la arquitectura RADIC, a continuación se describen:

- Observador: encargado de monitorear la aplicación y enmascarar errores. Realiza log de los mensajes recibidos de manera pesimista y realiza *checkpoints* periódicamente y los envía a los protectores.
- Protector: cada nodo sólo ejecuta un solo protector, y se encarga de almacenar los *checkpoints* y logs de los procesos (datos críticos). Cuando ocurre un fallo es el encargado de recuperar a los procesos fallados junto con su observador.

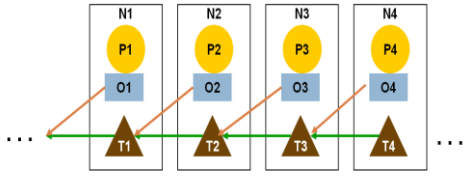


Figura 2. Relación entre los nodos de una aplicación MPI utilizando RADIC.

En la Figura 2 puede observarse como se combinan los elementos funcionales de RADIC. Los elementos Tx representan a los protectores, los elementos Ox representan a los observadores y los elementos Px son los procesos de la aplicación. Las líneas horizontales entre los protectores representan el mecanismo de *heartbeat/watchdog* y las diagonales entre observadores y protectores representa el envío de log y *checkpoints* de un proceso a su protector. Además en la Figura 2 se observa como los observadores van ligados a los procesos de aplicación y que existe un protector por nodo del computador paralelo.

Además de enmascarar los fallos de comunicación que ocurren, RADIC se encarga de recuperar a los procesos que han fallado cuando confirma los fallos, los procesos son recuperados en el protector en el mismo nodo en el cual reside.

En RADIC también existe la posibilidad de utilizar nodos *spare* [5] como elementos redundantes que son utilizados cuando ocurren fallos y permiten la restauración de procesos en ellos en lugar de sobrecargar el protector y de esta forma mantener la configuración y prestaciones.

2.1 RADIC independiente de la librería de comunicación

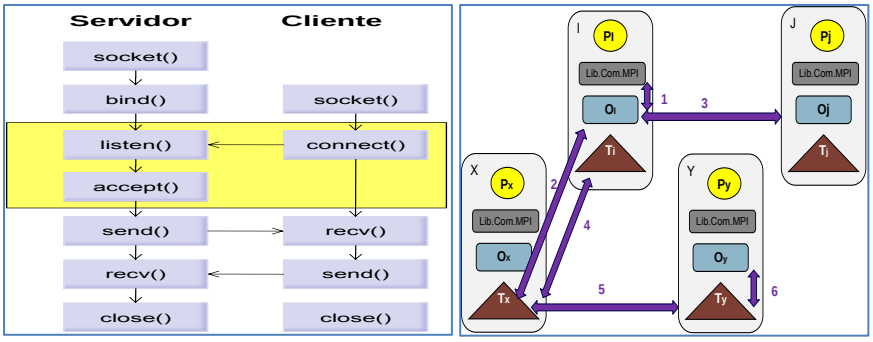
Este modelo permite tolerar los fallos que ocurren en el computador paralelo independiente a la librería de paso de mensaje que utilice la aplicación. Para ello, se integra RADIC en la capa del sistema operativo, más en concreto, a nivel de *sockets*, interponiendo las funciones que utilizan las redes Ehernet.

La mayoría de las implementaciones de MPI utilizan este protocolo estándar como interfaz con los sistemas operativos y por eso se elige esta capa ya que es un punto

común donde se consigue la independencia buscada. Esto permite que las aplicaciones puedan ejecutarse con la librería de MPI que prefieran (Open Mpi, Mpich, Lam-Mpi, etc.), y aún así, disponer del servicio de tolerancia a fallos de RADIC, que actúa en forma distribuida, y es escalable y flexible.

Este nivel permite observar todas las comunicaciones de la aplicación y hacer el log, aislar a la aplicación de los fallos en los nodos, enmascarando los fallos en las comunicaciones con los procesos en dichos nodos, recuperándolos en otros y redirigiendo las comunicaciones.

En Figura 3 (a) se muestran las principales funciones de socket que se interceptan. La aplicación gestiona en 3 tipos de sockets que provienen de la capa superior: los que actúan como clientes, que denominaremos **"*conectores*"** porque realizan un "connect", los que esperan peticiones **("*listeners*")** y los que tienen una comunicación activa luego de aceptar una conexión **("*accepters*").** En el momento de realizar un checkpoint, se cierran las comunicaciones activas, y al finalizar, los **"*conectores*"** se restablecen con sus **"*accepters*"** ubicados en otros nodos, cuyos observadores, reiniciarán sus **"*accepters*"** al detectar el cierre. En manera inversa, se restablecen los **"*accepters*"** que vuelven a conectarse con los nodos remotos. Sin embargo, cuando se reinicia una aplicación con un restart en otro nodo, el observador, antes de rehacer los **"*accepters*",** ejecuta los sockets del tipo **"*listeners*"** que describen las conexiones del tipo servidor. Estas acciones se realizan en forma transparente a la capa superior, que en este caso es la librería.



(a) (b)

Figura 3 (a) Funciones de socket que se interceptan. (b) Conexiones entre componentes.

La Figura 3 (b) muestra las diferentes comunicaciones entre los componentes que establece RADIC para cumplir con su objetivo. Se representan 4 nodos del computador paralelo, con un proceso paralelo en ejecución en cada uno, una librería de comunicaciones, un observador y un protector. Para mostrar una descripción funcional vamos a considerar al proceso en el nodo **I,** como se comunica y como se recupera en otro nodo tras un fallo.

La conexión **1** representa la interposición del observador con la librería MPI. La conexión **2** es un canal que cada observador **(O_I)** establece con su protector **(T_X)** para enviar la información redundante generada para proteger el cómputo de la aplicación. (logs y *checkpoints*). La conexión **3** se establece entre los observadores (**O_I-O_J**) que intercambian mensajes de la aplicación y que requieren la implementación del log de

mensajes pesimista en el receptor. Esto implica que por cada mensaje que reciba **OI**, lo envía a **T**x (2) y luego confirma a **OJ** (**3**). La conexión **4** se establece entre los protectores protector/protegido **(TI-TX),** por la que se envía la señal de heartbeat y confirmación. La conexión **5** es la que utiliza un protector para comunicarse con el protector de un proceso que no responde, para diagnosticar un posible fallo. Si esto ocurre en el nodo **I**, el protector **TX** envía a **TY** la nueva localización del proceso **PI,** que se recupera en el mismo nodo **X** o en nodo *Spare*, si hubiera uno disponible. Se utiliza la conexión **6** para que **TY** comunique a los observadores que residen en su mismo nodo, el cambio de localización del proceso **PI.**

Todas estas conexiones se realizan utilizando protocolo TCP. En todos los casos, al inicio, los procesos se intercambian el identificador del proceso **(pid)** que cada uno tiene en su nodo y la dirección (ip address) del nodo protector. En caso de caída de la conexión, la parte contraria, ya sea un protector u observador, contacta con el protector para diagnosticar el fallo. Si este se confirma, el protector envía la nueva dirección dónde se ha recuperado el proceso, y con este dato se restablece la conexión perdida. De esta misma forma se procede con los sockets de la aplicación. Debido a que el sistema operativo puede cambiar el identificador de socket al rehacerse la operación, el observador lleva una tabla de referencia, que relaciona el identificador original (*socket* virtual) con el real que mantiene efectivamente la conexión con el otro proceso (*socket* real).

3 Estado del Arte

Los protocolos de *rollback-recovery* son de los más utilizados para proveer tolerancia a fallos en los sistemas paralelos. En estos protocolos se utilizan mecanismos de *checkpoint-restart*, donde el estado de una aplicación paralela es salvado para una posible restauración futura.

Existen varias librerías de *checkpoint-restart* entre las cuales se pueden citar: *libckpt* [6], *Condor checkpoint library* [7], *BLCR* (*Berkeley Lab's Checkpoint/Restart*) [8] y una librería que trabaja a en espacio de usuario llamada *DMTCP* (*Distributed MultiThreaded Checkpointing*) [9]. Estos procedimientos difieren en como implementan el mecanismo de *checkpoint*, el porcentaje del estado del proceso que es preservado, como es almacenado y el nivel en que trabajan. La mayoría de las estrategias de *checkpoint-restart* distribuidas requieren la coordinación entre los procesos individuales para crear estados consistentes de la aplicación paralela. La mayoría de las técnicas de *checkpoint* caen en una de las tres siguientes categorías: coordinados, no coordinados o inducidos por mensajes.

En la Figura 4, adaptada de [10] vemos que existen varias alternativas implementadas en la capa de la librería de comunicación. Al igual que RADIC existen otras alternativas automáticas basadas en *checkpoints* y logs, pero utilizan elementos coordinadores en su mayoría. A continuación se describen las implementaciones de tolerancia a fallos más relevantes realizadas en la librería de paso de mensajes denominada MPICH.

En **MPICH-V** [11] se utiliza un protocolo de *checkpoint/restart* no coordinado en conjunto con log de mensajes para preservar el estado del proceso y recuperar un

proceso fallado automáticamente. Esta implementación se ha diseñado para computación a gran escala utilizando redes heterogéneas. **MPICH-V1** [11] está diseñado para grids y computación global dado que pueden soportar una tasa alta de fallos, pero requiere un alto ancho de banda para lograr buenas prestaciones.

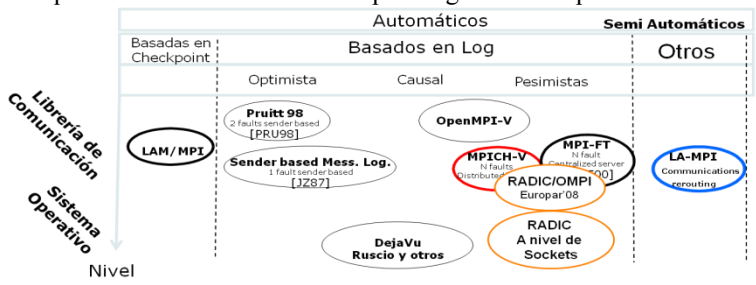


Figura 4. Implementaciones de sistemas de tolerancia a fallos.

Dentro de Open MPI [12] se han realizado diversas implementaciones de mecanismos de tolerancia a fallos, a continuación se describen brevemente algunos.

En **MPI-FT** [13] se busca ofrecer una API que permita al usuario final gestionar la librería de comunicación y extender alguna semántica de MPI para dar a la aplicación la posibilidad de recuperar un proceso fallado. Dentro del código estable de Open MPI existen ciertos mecanismos que proveen tolerancia a fallos de manera no transparente, y utiliza una estrategia de *checkpoint* coordinado [14].

DejaVu [15] ofrece un sistema transparente y automático de tolerancia a fallos para sistemas distribuidos y paralelos, en la misma capa de sistema operativo que proponemos, virtualizando los canales de comunicación. Utiliza un mecanismo de checkpoint incremental con log de mensajes, pero es un sistema coordinado, lo cual no permite que escale a un número importante de procesos. En cambio, puede utilizarse en otros tipos de redes como Infiniband.

4 Integración de RADIC en Open MPI

Una opción es integrar RADIC dentro de la librería Open MPI dado que es una librería bastante utilizada en el mundo científico. Un estudio profundo de la inclusión de RADIC en Open MPI fue realizado en [16]. Dicha versión no integra la gestión de nodos *spare*, ni la gestión de reparación e integración. Esta versión se denomina RADIC-OMPI.

Open MPI utiliza una arquitectura modular denominada MCA (*Modular Component Architecture*) que cuenta con tres capas funcionales: (a)*Root*: el cual provee básicamente todas las funciones MCA. (b)*Frameworks*: que especifica las interfaces funcionales a los servicios específicos (Ej. Transporte de mensajes). (c)*Components*: que son las implementaciones de cada *framework* en particular (Ej. TCP).

Los *frameworks* se encuentran en tres sub-capas que se complementan y son: *Open MPI (OMPI)* que provee la API para escritura de aplicaciones, *Open Run-Time Environment(ORTE)* que provee el ambiente para la ejecución de aplicaciones

paralelas y *Open Portable Layer (OPAL)* que provee abstracción a funciones del sistema operativo.

Los mecanismos de tolerancia a fallos que utiliza RADIC y su implementación en Open MPI son destacados a continuación:

- *Checkpoints no coordinados*: para la realización de cada *checkpoint* los canales de comunicación que utiliza MPI deben ser cerrados para garantizar que no existen mensajes en tránsito. El *checkpoint* es iniciado a través de una función *callback* que es disparada por un temporizador (intervalo de *checkpoint*), esta función se encarga de verificar que no existen comunicaciones. Una vez que se ha creado el *checkpoint* este es transferido utilizando el *framework* ***FileM*** y cuando termina esta transmisión el observador se encarga permitir de vuelta la transmisión de mensajes, y quien recibe el checkpoint se encarga de limpiar los logs almacenados.
- *Log de mensajes*: dado que el observador es el encargado de realizar las comunicaciones en lugar de la propia aplicación, todas las comunicaciones pasan por él y así son almacenadas en el log y trasladadas al protector.
- *Detección y gestión de fallos*: se detectan fallos cuando intentos de comunicación fallan, esto supone modificar las capas más bajas de Open MPI y evitar que los fallos lleguen a las capas superiores, además pueden detectarse fallos mediante el *heartbeat/watchdog* establecido entre protectores. Dado que Open MPI es *fail-stop* se debe añadir al demonio *ORTE* funciones de recuperación a partir de un *checkpoint* cuando es confirmado un fallo.
- *Recuperación y Reconfiguración*: en primer lugar el protector reinicia el proceso fallado desde el *checkpoint*, luego re-ejecuta a partir de ese punto y procesa el log. Cuando el observador es restaurado escoge su nuevo protector y se restablecen el sistema de protección. Reconfigurar las comunicaciones no es una tarea trivial ya que deben modificarse los *endpoints* para que se redirijan las comunicaciones a la nueva ubicación del proceso fallado, y deben actualizarse las tablas internas de RADIC o transferir información bajo demanda por medio de *tokens* [17].
- *Integración de nodos Spare*: deben ser insertados nodos que no pertenecen a la aplicación MPI, esto debe ser hecho desde el arranque de la ejecución o insertarlos en caliente y permitir que se anuncien a los demás por medio de *tokens*.

En la Tabla 1 se comparan los atributos de la implementación de tolerancia a fallos de Open MPI con los atributos que son añadidos por RADIC en dicha librería.

Tabla 1. Comparación de características de Open MPI y RADIC-OMPI.

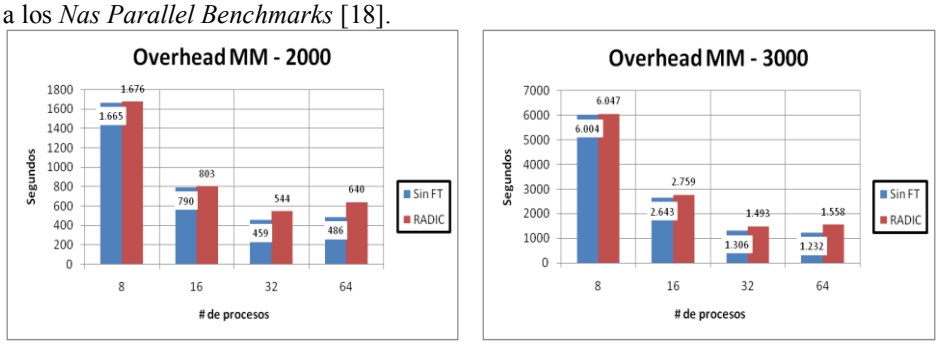
Librería Open MPI	RADIC-OMPI
Checkpoint coordinado. • No transparente y no escalable.	Checkpoint no coordinado más Log de mensajes. • Transparente y escalable.
Gestión dinámica de procesos a través elemento central.	Recuperación automática. • Heartbeat/Watchdog.
No da soporte para integrar nodos spare.	Gestiona nodos *spare*.
Migración en frio.	Soporte a la migración dinámica.

5 Resultados experimentales

En esta sección se presentan resultados experimentales que muestran el comportamiento de la implementación de RADIC en Open MPI en los cuales se han analizado los *overheads* introducidos y el comportamiento en caso de fallos cuando se cuenta o no con nodos *spare*.

El ambiente experimental utilizado es un Cluster Dell *PowerEdge* M600 de 8 nodos, con dos procesadores Intel® Xeon® CPU E5430@2.66GHz quad-core 6MB L2 por nodo y 16GB de memoria RAM de 667 MHz y una tarjeta de red doble integrada Broadcom® NetXtreme IITM 5708 Gigabit Ethernet. RADIC ha sido integrado en la versión 1.7 de Open MPI.

Se han utilizado dos aplicaciones, una de ellas es una multiplicación de matrices estática que sigue un modelo *master/worker* y otra es el *benchmark* LU que pertenece a los *Nas Parallel Benchmarks* [18].



(a) (b)

Figura 5. (a) *Overheads* de la Multiplicación de matrices (2000)(Int. Checkpt.=30 seg). (b) *Overheads* de la Multiplicación de matrices (3000). (Int. Checkpt.=30 seg)

En la Figura 5 (a) y (b) se muestran los *overheads* introducidos por RADIC en la multiplicación de matrices estática, en el primer caso el máximo *overhead* introducido es del 31% y en el segundo caso es del 26%.

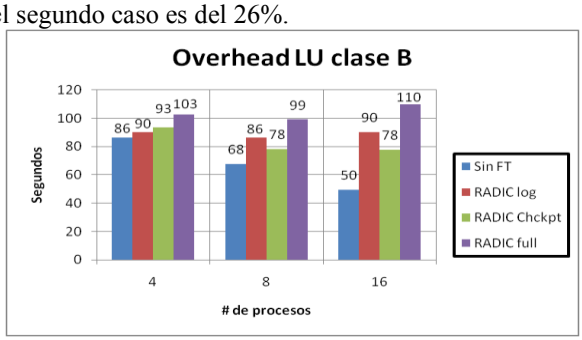


Figura 6. *Overheads* segmentados del *Benchmark* LU clase B.

En la Figura 6 puede verse como cuando a medida que aumenta la cantidad de procesos de la aplicación paralela, el log pesimista es el causante del mayor *overhead* y como RADIC escala correctamente si la aplicación lo hace, pero cuando la aplicación deja de escalar, RADIC colabora con la desintonización de la misma.

En la Figura 7 se muestra el comportamiento de la multiplicación de matrices cuando ocurren fallos y se cuentan con nodos *spare*. Puede apreciarse la degradación en las prestaciones cuando se restauran procesos que han fallado en el protector, dado que esto sobrecarga a dicho nodo disminuyendo considerablemente las prestaciones de la aplicación ya que la misma irá al ritmo del más lento.

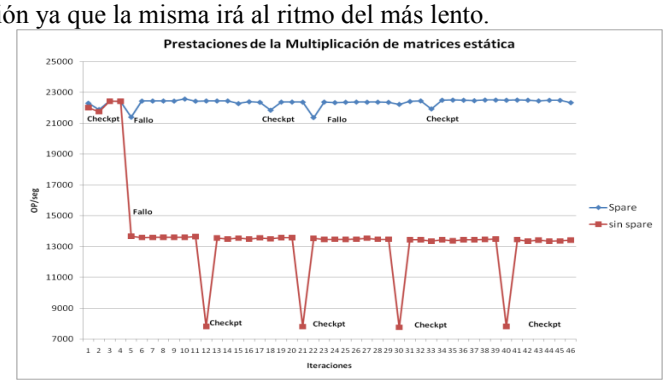


Figura 7. Prestaciones de la multiplicación de matrices (Tam: 3000 x 3000, 32 procesos, intervalo de checkpoint de 30 seg., 2 *spares*.).

6 Conclusiones

Se han presentado dos propuestas de integración de RADIC en aplicaciones paralelas de paso de mensajes. La inclusión de tolerancia a fallos en la librería de comunicaciones permite abstraer la solución de los detalles de implementación a nivel de sistema operativo, como pueden ser los protocolos de comunicación. Sin embargo, en los últimos años se han desarrollado varias implementaciones de librerías MPI, que presentan características diferentes, aun siguiendo el estándar. La tolerancia a fallos se convierte en un requisito obligatorio en los centros de cómputo, por lo que la propuesta de independencia de la librería ofrece una solución a aplicaciones que no puedan abordar el costo de adaptación a una librería específica para tolerar fallos.

En este trabajo se ha analizado la complejidad, ventajas y desventajas de la adaptación de RADIC en dos capas distintas y además se ha realizado un diseño e implementación dentro de la versión actual de Open MPI y un diseño que permite integrar las tareas de tolerancia a fallos interponiendo los *sockets* de comunicación.

Resultados iniciales de la implementación de RADIC en Open MPI demuestran que la misma escala correctamente junto a la aplicación, añadiendo un *overhead* aceptable. Si se utilizan nodos *spare,* en los cuales se restauran los procesos que han fallado, la aplicación paralela recupera una configuración similar a la inicial sin que se vean afectadas de manera considerable las prestaciones. Si se consideran los resultados mostrados en la Figura 7, puede verse que si un proceso fallado es restaurado en el nodo en el cual reside su protector, las prestaciones caen aproximadamente en un 40%, debido a que ese nodo queda sobrecargado, sin embargo si se restaura en un *spare* las prestaciones iniciales se mantienen.

Como trabajo futuro aún queda la implementación de RADIC a nivel de protocolo de comunicación y la consideración de nodos *multi-core* al momento de trasladar

procesos que han fallado. Además es necesaria una adaptación de RADIC a aplicaciones con eventos de entrada-salida, como son los sistemas transaccionales.

Referencias

1. Schroeder B., Gibson G. A.: Understanding failures in petascale computers. In: Journal of Physics: Conference Series, vol. 78, 2007.
2. Elnozahy E., Alvisi L., Wang Y., and Johnson D.: A survey of rollback-recovery protocols in message-passing systems. In: ACM Comput. Surv., vol. 34, no. 3, pp. 375--408, 2002.
3. Duarte A.: RADIC: a powerful fault-tolerant architecture. Bellaterra, España: Universitat Autònoma de Barcelona, 2007.
4. Forum MPI: MPI: A Message-Passing Interface Standard Version 2.2., 2009.
5. Santos G., Duarte A.: Increasing the Performability of Computer Clusters Using RADIC II. In: Int. Conference on Availability, Reliability and Security., pp. 653--658, 2008.
6. Plank J. S., Beck M., Kingsley G., Li K.: Libckpt: Transparent Checkpointing under Unix. In: Usenix Winter Technical Conference, pp. 213--223, 1995.
7. Litzkow M., Tannenbaum T., Basney J., Livny M.: Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Techreport, 1997.
8. Duell J.:The design and implementation of Berkeley Labs linux Checkpoint/Restart, 2003.
9. Ansel J., Arya K., Cooperman G.: DMTCP: Transparent checkpointing for cluster computations and the desktop. pp. 1--12, 2009.
10. Cappello F.: Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. Int. J. High Perform. Comput. Appl., vol. 23, no. 3, pp. 212--226, 2009.
11. Bosilca G.: MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In: Supercomputing, ACM/IEEE 2002 Conference, pp. 29 – 29, 2002.
12. Indiana University and Partners. (2011) Open MPI: Open Source High Performance Computing. [Online]. http://www.open-mpi.org/
13. Fagg G., Dongarra J.J.: FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol. 1908, p. 346-353, 2000.
14. Hursey J., Squyres J. M., Mattox T., Lumsdaine A.: The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: Workshop on Dependable Parallel, Distributed and Network-Centric Systems(DPDNS), in conjunction with IPDPS, p. 1-8, 2007.
15. Ruscio J. F., Heffner M. A., and Varadarajan S.: DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems. In: Parallel and Distributed Processing Symposium, International, vol. 0, pp. 119, 2007.
16. Fialho L.: Applying RADIC in Open MPI. Universitat Autònoma de Barcelona, Ed. Bellaterra, España, 2008.
17. Jalote P.: Reliable, Atomic and Causal Broadcast. In: Fault Tolerance in Distributed Systems. Estados Unidos de América: P T R Prentice Hall, pp. 142, 1994.
18. Bailey D.: The Nas Parallel Benchmarks. In: International Journal of High Performance Computing Applications, 1994.