Controlador/Precompilador de *Checkpoints* Portables

Gabriel Rodríguez, María J. Martín, Patricia González, Juan Touriño y Ramón Doallo

Resumen— **La técnica de salvado de ficheros de estado (*checkpointing*) es actualmente una de las más populares para proporcionar tolerancia a fallos a aplicaciones tanto secuenciales como paralelas. Sin embargo, aunque en los últimos años las investigaciones en este campo han aumentado, todavía son escasas las herramientas disponibles que ayuden a los programadores a dotar de tolerancia a fallos sus aplicaciones MPI. El propósito de este trabajo es describir la implementación, y también el uso, de una librería que proporciona tolerancia a fallos en códigos MPI, y de un precompilador basado en *SUIF* que facilita al usuario la utilización de esta librería.**

Palabras clave— **tolerancia a fallos, *checkpointing*, MPI, SUIF**

I. Introducción

En los últimos tiempos se ha popularizado la utilización de *clusters* en aplicaciones que requieren un gran tiempo de computación para su terminación. En este ámbito, es común que dicho tiempo de ejecución supere con creces al tiempo medio hasta fallo (*MTTF*), lo que hace necesaria la introducción de técnicas de tolerancia a fallos, bien en el propio *software* o bien en el *hardware*. Una de las soluciones comunmente adoptadas es el *checkpointing* de las aplicaciones. El estado de la computación se almacena periódicamente, de manera que sea posible reiniciarla en caso de fallo. Podemos hablar básicamente de dos aproximaciones para realizar esta tarea: *checkpointing* a nivel de sistema y *checkpointing* a nivel de aplicación. El primero consiste en almacenar todo el estado de la aplicación, incluído el estado *hardware* de la máquina. Esta técnica tiene dos desventajas básicas. En primer lugar, es una aproximación ciega, es decir, almacena siempre todo el estado. Esto provoca una sobrecarga innecesaria en muchos casos, en los que es suficiente con almacenar un pequeño núcleo de información a partir del cual puede recuperarse el resto. El segundo punto en contra de esta aproximación es el hecho de que los ficheros de estado sólo pueden ser utilizados bajo la arquitectura en la que fueron generados. La ventaja inherente es que puede ser implementada de forma eficiente y genérica, de modo que una misma solución puede ser aplicada a diferentes programas de forma totalmente transparente al usuario. En [1] se puede consultar una extensa revisión sobre las diferentes alternativas de implementación de esta técnica. En el otro extremo, en el *checkpointing* a nivel de aplicación, el usuario especifica qué estado es relevante, y sólo éste es almacenado. Esta aproximación elimina las desventajas vistas en el *checkpointing* a nivel de sistema, pero conlleva el que debe analizarse cada aplicación concreta por separado. Es habitual que sea el propio programador el que implemente el *checkpointing* para cada aplicación, aunque también existen librerías y compiladores que permiten automatizar o semi-automatizar el proceso tanto para aplicaciones secuenciales [2], [3], [4] como paralelas [5], [6], [7]. Usualmente estas soluciones imponen condiciones excesivamente fuertes sobre el estado que debe ser salvado, en el que a menudo se incluye información de bajo nivel, como el contador de programa, las direcciones de memoria en las que se ubicaban originalmente los datos o el contenido de la pila. Presentan así una de las desventajas comentadas para el *checkpointing* a nivel de sistema: la baja portabilidad de los ficheros generados.

Grupo de Arquitectura de Computadores, Departamento de Electrónica y Sistemas, Universidad de A Coruña. Correo electrónico: infgra00@ucv.udc.es,mariam@udc.es, pglez@udc.es, juan@udc.es, doallo@udc.es

En este trabajo presentamos una librería y un precompilador basado en SUIF (*Stanford Universal Intermediate Format*) [8] que facilitan al usuario el *checkpointing* de aplicaciones MPI [9]. El usuario tendrá que incluir en el programa directivas que indiquen en qué puntos del programa se quiere salvar el estado y qué variables es necesario almacenar. El precompilador se encarga de traducir estas directivas a llamadas a la librería (librería *CCCP*) e introducir el código adicional necesario para salvar todo el estado relevante de la aplicación. La información almacenada es básicamente información de alto nivel, nunca relacionada con la arquitectura subyacente, lo que permite su portabilidad entre diferentes arquitecturas.

La estructura de este artículo es la siguiente. La sección II describe la librería *CCCP*, que proporciona las funciones necesarias para la realización del *checkpoint* y su implementación. La sección III describe las directivas soportadas por el precompilador propuesto y las tareas que realiza asociadas a esas directivas. En la sección IV se indican algunos detalles importantes en el diseño de la propuesta, relacionados con la comunicación entre procesos. Por último, la sección V resume las conclusiones que se extraen de este trabajo y la sección VI su proyección futura.

II. Librería *CCCP*

La librería *CCCP* proporciona un reducido núcleo de funciones orientadas al *checkpointing* de cualquier tipo de programa. En concreto, la librería permite realizar las siguientes tareas: arranque y finalización del controlador, registro de variables, volcado del estado a disco, reinicio de la aplicación a partir de los ficheros de estado. Las siguientes subsecciones describen en detalle cada una de estas tareas.

A. *Arranque y finalización del controlador*

Al iniciar cada proceso paralelo de la aplicación, es necesario cargar en memoria un controlador que resuelva las peticiones relacionadas con el *checkpointing*. Este controlador debe ser cargado después de la librería *MPI*, es decir, después de la ejecución de `MPI_Init()`, pues podría ser necesaria la comunicación con el resto de controladores involucrados en la computación. En concreto, si debe reiniciarse la aplicación desde un *checkpoint*, los controladores deben llegar a un acuerdo sobre el punto exacto de reinicio. Por otra parte, el sistema debería arrancarse antes del comienzo del programa principal, de manera que pueda absorber los parámetros de línea de comandos correspondientes a la librería para que no trasciendan al código de la aplicación.

El sistema podría ser apagado en cualquier momento, siempre antes de la terminación del sistema paralelo, es decir, antes de la ejecución de `MPI_Finalize()`.

Las funciones de la librería que permiten el arranque y finalización del controlador son, respectivamente, `CCCP_Init()` y `CCCP_Shutdown()`.

B. *Registro de variables*

Cuando una variable es relevante para el estado del programa debe ser registrada, utilizando para ello la función `CCCP_Register()`. Para simplificar el proceso, todas las variables se tratan como espacios de memoria. El controlador registra únicamente la dirección que debe ser almacenada, así como el tamaño. De este modo, realizar un *checkpoint* se reduce básicamente a leer estas direcciones de memoria y volcarlas al disco, con un formato que permita recuperarlas en caso de reinicio.

Aparte del registro de regiones de memoria, se contempla el borrado de dichos registros (`CCCP_Unregister()`), para permitir la desactivación de variables salvadas a medida que se va avanzando en el código de la aplicación.

C. *Volcado del* checkpoint

Una de las principales cuestiones a considerar a la hora de hacer el volcado del *checkpoint* en cada uno de los procesos de un programa paralelo es el punto en el que se va a llevar a cabo este volcado. En nuestra propuesta, las llamadas a la función `CCCP_Do_checkpoint()` han de ser introducidas en puntos seguros. Entendemos por punto seguro aquel en el que no hay mensajes viajando de un proceso a otro, es decir, mensajes que hayan sido enviados pero no recibidos. Si se realizase un *checkpoint* en ese momento, al reiniciar la aplicación dichos mensajes no serían enviados nuevamente, pero el proceso destino sí esperaría recibirlos.

Al alcanzar un punto de volcado, y si se supera la frecuencia de almacenamiento, los datos registrados son grabados a disco. La frecuencia de almacenamiento es un parámetro opcional de línea de comandos que indica la cadencia con la que se realiza el volcado a disco del *checkpoint* para los puntos de volcado que se encuentran en el interior de un lazo. La frecuencia especificada indica cada cuantas iteraciones se ha de realizar el volcado. El valor por defecto es `1`.

Si nos encontramos en un *cluster* homogéneo es posible almacenar directamente datos binarios, consiguiendo así un aumento de eficiencia en la operación del controlador. En otro caso será necesario el almacenamiento en un formato portable, como por ejemplo *UDF* [10], *XDR* [11] o *HDF-5*[1] [12], que permita la interpretación de los datos en cualquier arquitectura. En este caso, es necesario conocer qué tipo de datos estamos almacenando, para poder dar un formato adecuado a los mismos. Para garantizar la total compatibilidad del sistema con las librerías *MPI*, se utilizan los propios *MPI_Datatypes* para marcar cada registro, de modo que se pueda realizar una correcta representación de cada bloque de datos. Una consideración a tener en cuenta es el hecho de que, en realidad, un *MPI_Datatype* no deja de ser un tipo entero. Ya que el objetivo es lograr la portabilidad de los *checkpoints*, es necesario garantizar que el reconocimiento de los tipos no es dependiente de la librería *MPI* utilizada. Por ello, al almacenar los datos en disco se etiquetarán con un formato interno (*CCCP_Datatype*), que garantiza la homogeneidad de la asignación de enteros al mismo.

Además de los datos de alto nivel que la aplicación pide volcar y el tipo de datos correspondiente se añade cierta información de control que estructure los mismos, así como un código de redundancia (hemos utilizado *CRC-32*) para asegurar la integridad de los datos. Esto es importante dado que se trabaja con un entorno potencialmente inestable, y existe la posibilidad de que los ficheros almacenados se vean afectados por un fallo, así como de que el sistema se caiga en el instante en que un *checkpoint* estaba siendo almacenado (quedando éste incompleto).

En cuanto a la estructura de los ficheros en disco, sigue un esquema jerárquico que permite un reinicio incremental y, por tanto, más potente y eficiente. Los datos se van recuperando poco a poco, no se cargan en memoria en un único momento, sino que lo van haciendo a medida que cobran relevancia. Esta característica permite que se efectúen registros de tamaño dependiente de variables del sistema, que deben ser registradas previamente. Los registros se estructuran en memoria en secciones. Se agrupan en una misma sección aquellos registros que se efectúan entre dos puntos de volcado (de este modo los puntos de *checkpoint* dividen el código en secciones). La estructuración en disco es similar, de modo que el formato de cada fichero es el mostrado en la figura 1.

Al estructurar las variables en secciones, es posible determinar cuándo debe ser recuperada cada una, de modo que no es necesario realizar la recuperación en un sólo paso consistente en la carga de todos los datos

[1] *CCCP* se ha programado de forma que sea sencillo extender los formatos de escritura de la librería (es suficiente con implementar una clase que realice una interfaz y registrarla en el controlador)

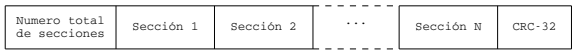

Fig. 1. Estructura global del fichero de *checkpoint*

a memoria y posicionado del flujo de ejecución en el punto adecuado. Más aún, permite registrar variables con el mismo nombre en diferentes secciones. La estructura en disco de cada sección individual puede observarse en la figura 2.

Fig. 2. Estructura de cada sección del fichero de *checkpoint*

Descendiendo un nivel más, encontramos los registros, que se corresponden con un bloque de memoria del programa principal. Esta entidad ya no es una estructuración lógica de los datos, sino que se corresponde con verdaderos datos de la aplicación, junto con información añadida necesaria para terminar de contextualizarlos. Estas partes del fichero son especiales en comparación con las mostradas anteriormente, dado que no existe un formato definido para las mismas. Dependerá de la estrategia de escritura que se haya utilizado para su creación, y sólo debe cumplir un requisito: a partir del mismo debe ser posible recuperar un registro en memoria. La estructura de uno de ellos se especifica en la figura 3. Son la unidad mínima de información utilizada por la librería, y pueden verse como la traducción de una variable a *CCCP*.

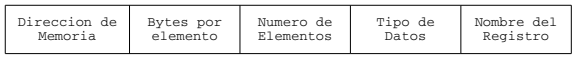

Fig. 3. Estructura de un registro *CCCP*

C.1 Optimizaciones

Para optimizar el tiempo de volcado gran parte del mismo se realiza de forma concurrente con la ejecución del programa. Para conseguir este fin, es suficiente garantizar que los datos que deben ser salvados no son modificados antes del volcado a disco. Para ello, se realiza simplemente una réplica de los mismos en memoria. Tras esto, se lanza un nuevo hilo de ejecución, encargado del almacenamiento físico de la información, mientras que se permite al hilo principal continuar la ejecución del programa [13]. En caso de que las necesidades de memoria sean críticas, será conveniente desactivar esta característica de forma que no se consuman recursos innecesarios.

También se ofrece la posibilidad de aplicar compresión a los datos. Si bien esto aumenta el tiempo de almacenamiento, mejora la transferencia de los ficheros por la red, así como el espacio utilizado en disco por los mismos. El usuario deberá sopesar qué variables son críticas en su aplicación, y configurar *CCCP* en consecuencia.

D. *Reinicio a partir de un* checkpoint

Se puede intuir que el proceso de reinicio será el inverso al descrito para el almacenamiento. Se lee el fichero y se recupera cada uno de los registros, organizándolos en secciones. Lo que queda por definir es el protocolo en sí, los pasos que deben seguirse para conseguir un reinicio correcto y sincronizado a partir de estos datos.

En primer lugar, todos los procesos participantes deben ponerse de acuerdo sobre el punto desde el que reiniciar. Para ello, cada proceso lee los ficheros de *checkpoint* de los que dispone y determina cuál es el más avanzado. Una vez que cada uno ha calculado este número, se realiza una operación de reducción global sobre el mínimo del mismo. Una vez determinado el fichero desde el que se debe arrancar, continúa el reinicio de forma local. En este punto cada proceso debe leer los contenidos del fichero local que se corresponde con el acuerdo global alcanzado, y convertirlos en registros en memoria. Una vez se ha realizado esta operación, el flujo continúa sobre el código de la aplicación principal.

El esquema de recuperación de variables se define de forma que la librería obliga al programa a ejecutar ciertos bloques de código necesarios para el reinicio, mientras que otros superfluos se descartan. Es el programador el que identifica estas porciones de código a través de directivas *pragma*, en un proceso que se describe en la sección III. Se utiliza la función de librería `CCCP_Jump_next()`, que indica simplemente si se está reiniciando la aplicación (y por tanto deben realizarse saltos entre bloques de código) o no (situación en la que el código debe ejecutarse de forma lineal).

El código se divide en bloques a través de etiquetas de salto, y se dispone de un *array* que indexa estas direcciones. El flujo de programa se controla a través de *arrays* locales de direcciones de memoria y saltos entre dichas direcciones con *gotos*. El proceso global consiste en ir saltando de una etiqueta a otra, hasta que la librería determine que el reinicio ha finalizado, y que es posible continuar una ejecución secuencial del código de la aplicación original. En un código genérico se identifican los siguientes bloques de instrucciones relevantes:

- Inicio del controlador *CCCP*: En el mismo se determina si se debe llevar a cabo el reinicio o no. Tras la ejecución de esta función debe realizarse un salto condicional al siguiente bloque de código relevante (comienzo del reinicio).
- Bloque de registros de variables: Este tipo de bloques se componen de llamadas a las funciones `CCCP_Register()` y `CCCP_Unregister()`. Se ha mostrado como en una ejecución normal éstas se encargan de crear y eliminar en memoria los registros de variables pertinentes. En el momento del reinicio, sin embargo, `CCCP_Register()` debe recuperar los valores de las variables, de modo que tras la llamada a una función de registro el contenido de la memoria involucrada

en el mismo habrá sido recuperado directamente de los valores escritos en el fichero de *checkpoint*. Tras un bloque de registros deberá llevarse siempre a cabo un salto condicional hacia la siguiente porción relevante de código.

- Bloques de ejecución obligatoria: Hay ciertos conjuntos de instrucciones que deben ser ejecutados al reiniciar la aplicación, a pesar de no estar relacionados con la librería *CCCP*. Si bien el programador puede decidir qué conjuntos de código son clasificados en esta categoría, un ejemplo claro son las llamadas a funciones *MPI* que involucran creación de estructuras como comunicadores y topologías[2], o la apertura de ficheros. Tras este tipo de bloques es obligatorio también el salto condicional hacia el siguiente conjunto relevante.
- Puntos de *checkpoint*: Al llegar a uno de estos puntos, el controlador debe comprobar dos cosas. La primera es si todos los valores almacenados en el fichero han sido ya recuperados. En caso contrario no es necesario realizar más comprobaciones: el proceso de reinicio debe continuar. Si se cumple esta primera condición, es preciso verificar si el punto de volcado se corresponde con aquél en el que se creó el *checkpoint* desde el que estamos reiniciando. Si ambas condiciones resultan ser ciertas, el proceso de reinicio ha terminado, y la ejecución debe continuar de forma secuencial desde este punto. Por tanto, a partir de este momento, la función `CCCP_Jump_next()` devolverá `0`. Es necesario también incluír un salto condicional tras estas llamadas.

De este modo, debe verse el flujo de ejecución en el proceso de reinicio como una sucesión de saltos a bloques relevantes en tanto éste no termine. Se comienza en un punto claro: tras la inicialización de la librería (llamada a la función `CCCP_Init()`). A partir de ese momento se comenzará a saltar sobre los diferentes bloques de código, restaurando valores de variables, inicializando *handlers MPI*, abriendo ficheros necesarios para la aplicación, etc. Una vez que el proceso de reinicio se completa el flujo de ejecución estará posicionado en el último punto de *checkpoint* que se hubiese llevado a cabo. A partir de ahí la ejecución de la aplicación continuará de forma lineal.

Puede suceder que el punto en el que se ha creado el *checkpoint* en la ejecución original se alcance antes de que el proceso de reinicio se haya completado totalmente. Esto sucede, por ejemplo, cuando el último *checkpoint* se ha llevado a cabo en una iteración intermedia de un bucle. En este caso el controlador ha de terminar el proceso de reinicio, saltanto a bloques relevantes posteriores en el código pero pertenecientes a la iteración anterior del bucle, antes de posicionarse en el punto de volcado para continuar con la ejecución normal de la aplicación.

Un punto que no ha sido clarificado todavía es cómo tratar las llamadas a subrutinas en reinicio. Puede ocurrir que la función a la que se llama no contenga código *CCCP*: ni registros ni *checkpoints*. En este caso no es necesario realizar ninguna acción. De lo contrario, debe haber un punto de salto a la instrucción que realiza la llamada a la función, de modo que se entre en la misma. Internamente, la función se tratará exactamente igual que el procedimiento principal (tendrá etiquetas y saltos condicionales) con la diferencia de que el último salto será a una instrucción *return* genérica (dependiente del tipo de retorno original de la función). En este punto debemos hacer dos consideraciones:

- No se debe saltar directamente al código interno de la función, ya que no se realizaría el cambio de contexto correctamente y el comportamiento sería indefinido. En concreto, al realizar el *return*, la dirección de retorno no estaría en la pila por lo que el destino del salto sería aleatorio.
- El *return* genérico no afecta al resultado del reinicio. Si el valor devuelto por la función se almacena en una variable que es registrada en la librería, este registro deberá hacerse de forma posterior a la llamada a la función. De este modo, el valor correcto será recuperado en el momento de la llamada a `CCCP_Register()`.

En cuanto a las llamadas a `CCCP_Unregister()`, no necesitan un tratamiento especial más allá de eliminar realmente el registro del conjunto de los supervisados por la librería. En realidad, si el registro de una variable había sido eliminado previamente a la realización del *checkpoint* ni siquiera existirá en memoria, por lo que las llamadas a ésta función no provocarán ningún efecto.

III. Precompilador

El procedimiento de reinicio descrito anteriormente, necesario para el correcto funcionamiento de la librería, puede llegar a ser un tanto pesado pues, si bien puede realizarse siguiendo unas reglas deterministas y bien definidas, es repetitivo y tedioso. Sería deseable que el programador dispusiese de un conjunto de directivas *pragma* que pudiese utilizar para indicar a un precompilador *CCCP* qué es lo que desea realizar. Este precompilador se ha implementado utilizando *SUIF* [8], e incluye las siguientes directivas:

- `#pragma cccp init`: Indica que debe iniciarse el sistema *CCCP*. El precompilador podría hacerlo automáticamente tras la llamada a `MPI_Init()`, pero es posible que no se requiera que el controlador esté funcionando durante toda la ejecución de la aplicación, por lo que es preferible mantener un inicio manual. Esta *pragma* se traduce como una llamada a `CCCP_Init()` seguida de un salto condicional.
- `#pragma cccp register(var1[size1], var2[size2],...)`: Indica que deben regis-

[2]Objetos opacos que no deberían ser preservados a través del fichero de *checkpoint*, pues obligarían a un tratamiento complejo de bajo nivel que no es deseable

trarse las variables indicadas. Si se omite el tamaño entre corchetes se asume que son variables escalares. Si se especifica puede hacerse a través de constantes o de variables. En éste último caso las mismas deben haber sido registradas previamente, pues en el momento del reinicio será necesario conocer el tamaño del *array*. Este registro es realizado por el precompilador de forma automática en caso de que el usuario no lo realice explícitamente. Antes del bloque de registros se introduce una etiqueta de salto, y tras el mismo el precompilador añadirá un salto condicional.

- `#pragma cccp execute on restart`: Va acompañada de `#pragma cccp end execute` e implica que el bloque de código contenido entre ambas directivas debe ser ejecutado en el reinicio una vez. Aquí se incluirán aperturas de ficheros, creación de comunicadores, etc. Antes y después de un bloque de éste tipo el precompilador introduce una etiqueta de salto y un bloque de salto condicional, respectivamente.
- `#pragma cccp checkpoint`: Se traducen como llamadas a `CCCP_Do_checkpoint()`. Deben estar introducidas en puntos "seguros" de la computación. Se insertarán entre una etiqueta y un salto condicional.
- `#pragma cccp unregister (var1,var2,...)`: Es la directiva opuesta a `cccp register`. Se traduce por llamadas a `CCCP_Unregister()`, y se introduce entre una etiqueta y un bloque de salto condicional.
- `#pragma cccp shutdown`: Indica que ya no es necesaria la existencia del controlador, y que *CCCP* puede ser desactivado. Al igual que `#pragma cccp init`, podría ser ubicada automáticamente por el precompilador antes de la llamada a `MPI_Finalize()`, pero es preferible dejarlo a elección del usuario para permitir el ahorro de recursos del sistema.

Además de las *pragmas* introducidas explícitamente por el programador, el precompilador realiza otras tareas de forma automática. Una de ellas es la creación de un *array* estático de direcciones de salto en cada función cliente de *CCCP*, que permita la realización del reinicio. Este tipo de *arrays* es una alternativa elegante a las construcciones *switch* cuando éstas no pueden ser utilizadas (por ejemplo, puede ser necesario realizar un salto directamente al interior de un *if*, lo cual no sería posible utilizando un *switch*). Además, este sistema garantiza la compatibilidad con la gran mayoría de los compiladores *C*, que permiten obtener la dirección de una etiqueta de salto mediante el operador `&&`. Una vez construído este *array* de direcciones, se añade una variable índice que, comenzando en `0`, indique el siguiente salto. De este modo, cada bloque de salto condicional se reduce a ejecutar la función `CCCP_Jump_next()`, que indica si se está llevando a cabo el reinicio, y en caso afirmativo realizar un salto a la siguiente dirección del *array*, aumentando el contador de saltos en un elemento. También es sencillo hacer que una sección de código se ejecute una única vez (por ejemplo, los bloques *execute on restart*), sencillamente decrementando la variable que indica el tamaño del *array* en una unidad y reorganizando las direcciones.

La otra tarea fundamental es la detección de funciones clientes de *CCCP* e inserción de las llamadas a las mismas entre etiquetas y saltos condicionales. De esta forma se consigue hacer un poco más sencilla al programador la utilización de la librería, pues no debe preocuparse de la inserción de código para la realización de esta tarea. El punto negativo es que la precompilación será más lenta, pues debe realizarse una pasada previa al comienzo que identifique qué funciones incluyen directivas y cuáles no, que no puede ser llevada a cabo en un *pipeline* junto con las demás pasadas. En cualquier caso, dado que la precompilación sólo se realiza una vez y que la carga que introduce esta pasada no es excesiva, se considera que el cambio es beneficioso.

IV. Comunicaciones entre procesos

Si bien *CCCP* puede funcionar correctamente sobre programas secuenciales, ha sido diseñado para controlar el *checkpointing* en aplicaciones paralelas *MPI*. Esto, obviamente, requerirá de ciertas comunicaciones entre los procesos participantes, si bien no muy frecuentes.

Así, en un principio la librería sólo requiere de una comunicación global en cada reinicio: aquella que asegura que todos los procesos arrancan desde un mismo punto. A partir de ese momento, el controlador de cada proceso trabaja de forma independiente con respecto al resto. La consistencia global está asegurada pues las llamadas a la función `CCCP_Do_checkpoint()` habrán sido introducidas en puntos seguros.

Esta aproximación es mucho más escalable que la aplicación de protocolos que aseguren la consistencia del *checkpointing* (en [7] se describe uno de estos protocolos). En contrapartida, habrá que realizar análisis de flujo de mensajes para garantizar la seguridad del protocolo. Este análisis puede ser llevado a cabo de forma semiautomática, y en cualquier caso presentará siempre un menor coste que la introducción de mecanismos de seguridad en el propio código de la librería, dado que ésta solución implica *overhead* en tiempo de ejecución, mientras que el análisis se realizará antes de aplicar el precompilador *CCCP*.

V. Conclusiones

En este trabajo hemos presentado una librería y un precompilador que facilitan al usuario realizar el *checkpointing* de un programa MPI a nivel de aplicación.

En la solución propuesta es responsabilidad del usuario encontrar puntos seguros dentro del programa en los cuales realizar el volcado del estado

a disco, es decir, puntos donde no haya mensajes en tránsito, lo cual simplifica la implementación.

Dado que en estos puntos no hay mensajes en tránsito no será necesario construir ningún protocolo que los contemple, ni será necesaria la coordinación ni la sincronización entre los procesos para realizar el volcado de datos a disco. Esto resuelve los problemas de escalabilidad asociados con dichos protocolos, y los problemas de contención de la red asociados con la sincronización.

Además, el usuario especifica exactamente qué desea almacenar, lo cual reduce drásticamente el tamaño de los ficheros de estado y con ello sus requerimientos de memoria y su *overhead*, tanto en tiempo de volcado como en tiempo de reinicio. Adicionalmente, y gracias a la implementación de la directiva `execute on restart` se ofrece al usuario la posibilidad de elegir entre almacenar una variable o recalcularla, lo cual puede reducir todavía más dicho tamaño.

VI. Trabajo futuro

Como trabajo futuro pretendemos trasladar la responsabilidad del programador en la utilización y ubicación de las directivas al precompilador. El objetivo es la extracción automática o semiautomática de toda la información referente a puntos seguros, variables que es necesario volcar, o bloques de código que deben ser ejecutados en el reinicio.

Referencias

[1] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[2] "CKPT library," http://www.cs.wisc.edu/ zandy/ckpt.

[3] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, January 1995, pp. 213–223.

[4] G. Kingsley, M. Beck, and J. S. Plank, "Compiler-assisted checkpoint optimization using SUIF," in *First SUIF Compiler Workshop*, January 1996.

[5] Y. Chen, J. S. Plank, and K. Li, "CLIP: A checkpointing tool for message-passing parallel programs," in *SC97: High Performance Networking and Computing*, San Jose, November 1997.

[6] Sunil Ahn, Jungwhan Kim, and Sangyong Han, "PC/MPI: Design and implementation of a portable MPI checkpointer," *Lecture Notes in Computer Science*, vol. 2840, pp. 302–308, 2003.

[7] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill, "C3: A system for automating application-level checkpointing of mpi programs," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC'03)*, October 2003.

[8] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy, "Suif: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, no. 12, pp. 31–37, 1994.

[9] Message Passing Interface Forum, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, Department of Computer Science, University of Tennessee, Apr. 1994, Tue, 22 May 101 17:44:55 GMT.

[10] B. Ramkumar and V. Strumpen, "Portable checkpointing for heterogenous architectures," in *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, Washington - Brussels - Tokyo, June 1997, pp. 58–67, IEEE.

[11] Sun Microsystems, *XDR: External data representation standard*, Sun Microsystems, June 1987, RFC 1014, 20 pages.

[12] "HDF5: File Format Specification," http://hdf.ncsa.uiuc.edu/HDF5/doc/.

[13] Kai Li, Jeffrey F. Naughton, and James S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, Aug. 1994.