CPPC: una herramienta portable para el checkpointing de aplicaciones paralelas

CPPC: A Portable Tool for Checkpointing Parallel Applications

G. Rodríguez, M. J. Martín, P. González et al.

Resumen

CPPC (Controller/Precompiler for Portable Checkpointing) es una herramienta de tolerancia a fallos basada en checkpointing, para la ejecución de aplicaciones paralelas sobre clusters heterogéneos o el Grid. Además de utilizar código, protocolos y formatos de almacenamiento portables, es necesario realizar análisis en tiempo de compilación de las aplicaciones que se pretenden tratar, de forma que se eviten inconsistencias en tiempo de reinicio sin caer en soluciones tradicionales, como la coordinación de procesos o el registro de mensajes, que redundan en una sobrecarga no escalable. Un preprocesador realiza transformaciones fuente a fuente, transformando una aplicación paralela en una versión tolerante a fallos mediante llamadas a la librería CPPC.

Palabras clave: Tolerancia a fallos, *checkpointing,* programación paralela, paso de mensajes, MPI.

Summary

CPPC (Controller/Precompiler for Portable Checkpointing) is a checkpoint-based fault tolerance tool for the execution of parallel applications over heterogeneous clusters or Grids. In addition to using portable code, protocols and data storage formats, compile-time analysis of the application to be checkpointed is needed in order to avoid restart inconsistencies, without falling back to traditional approaches like process coordination or message-logging, which typically introduce unscalable overheads. These solutions are implemented into the CPPC library, used by the CPPC source-to-source preprocessor to automatically transform a parallel application into a fault tolerant counterpart. Experimental results to assess the checkpointing tool are also provided.

Keywords: Fault tolerance, checkpointing, parallel programming, message-passing, MPI.

1. Introducción

La evolución de la computación paralela hacia infraestructuras *cluster* y Grid ha creado nuevas necesidades de tolerancia a fallos. A medida que las máquinas paralelas ven incrementado su número de procesadores, también aumenta la tasa de fallos del sistema global. Esto no supone un problema mientras el tiempo medio que una aplicación tarda en ejecutarse permanezca por debajo del tiempo medio hasta el fallo del sistema subyacente. Pero esta condición no siempre se cumple. En aplicaciones con ejecuciones largas, los usuarios y programadores necesitan disponer de un mecanismo que garantice que no toda la computación realizada se perderá en caso de fallo. El *checkpointing* se ha convertido en una técnica ampliamente utilizada para la obtención de dicha garantía. Proporciona tolerancia a fallos almacenando periódicamente el estado de la computación en almacenamiento estable, de modo que dicho estado pueda ser recuperado en caso de fallo. Se ha propuesto un número de soluciones y técnicas [1], cada una con sus propias ventajas e inconvenientes.

CPPC es una herramienta de tolerancia a fallos basada en *checkpointing,* enfocada en la ejecución de aplicaciones paralelas sobre *clusters* heterogéneos o Grid

La propiedad más importante de las técnicas de *checkpointing* es la *granularidad*. El *checkpointing* puede aplicarse desde dos niveles de granularidad diferentes: *nivel de segmento de datos* y *de variable*. Desde el nivel de segmento de datos, el estado de la aplicación se salva completamente. Esta aproximación tiene una ventaja indiscutible: es independiente de la aplicación considerada, puesto que ésta se trata como una caja negra. Pero cuanto mayor sea el estado almacenado, menor eficiencia presentará la técnica de *checkpointing*. Además, almacenar el estado completo de la aplicación provoca falta de portabilidad, debido a que se almacenarán estructuras no portables junto con los datos de la aplicación (por ejemplo, la pila de la misma).

Almacenar el estado completo de la aplicación provoca falta de portabilidad

CPPC (*Controller/Precompiler for Portable Checkpointing*) es una infraestructura de *checkpointing* que resuelve estos problemas, implementando una solución escalable, eficiente y portable utilizando una técnica a nivel de variable, no coordinada y sin registro de mensajes.

2. La infraestructura CPPC

CPPC [2] se compone de una herramienta de compilación y una librería de funciones que simplifican la inserción de tolerancia a fallos en una aplicación paralela. El usuario inserta directivas de compilación en el código de la aplicación original, y ejecuta el precompilador que realiza una transformación fuente a fuente. El resultado será un código tolerante a fallos, que realiza llamadas a la librería CPPC. El usuario puede introducir las siguientes directivas:

- *cppc init:* inicializa la librería, creando las estructuras necesarias.
- *cppc shutdown:* marca el punto de finalización de CPPC y garantiza el correcto apagado de la librería, asegurándose de que todos los ficheros de estado quedan almacenados consistentemente.
- *cppc register (var1[size1], ...):* identifica las variables cuyos valores serán almacenados en el fichero de estado. Sus valores serán recuperados al reiniciar la aplicación.
- *cppc unregister (var1, var2, ...):* elimina de las variables marcadas para ser almacenadas a aquellas que han dejado de ser necesarias.
- *cppc execute/end execute:* marca bloques de código que deben ser ejecutados obligatoriamente al reiniciar la aplicación, dado que recuperan estado no portable, como por ejemplo comunicadores MPI. Si estas estructuras no portables fueran almacenadas en disco, se perdería la portabilidad.
- *cppc checkpoint:* puntos donde se almacena el estado. En ellos, las variables marcadas se almacenan en disco.

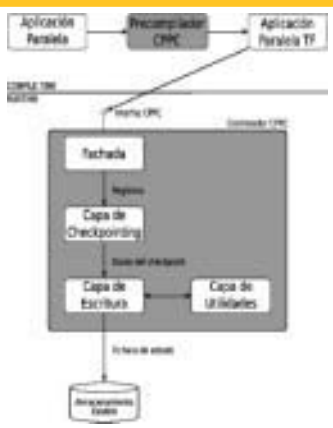
CPPC se compone de una herramienta de compilación y una librería de funciones que simplifican la inserción de tolerancia a fallos en una aplicación paralela

El precompilador de CPPC, construido sobre la infraestructura de compilación Cetus [3], realiza de forma automática la inserción de las cuatro primeras directivas.

Una vista general del diseño de CPPC puede observarse en la figura 1. La librería posee una única implementación en C++, que será invocada a través de interfaces para diferentes lenguajes de programación (actualmente soporta C y Fortran 77). Estas interfaces enmascaran características dependientes del lenguaje de programación. Redirigen las peticiones a la fachada, responsable a su vez de gestionar el estado a ser volcado cuando se alcance un *checkpoint*. De este modo, las peticiones que afectan el estado a ser volcado son rápidas, al ser tratadas por la capa superior de la librería. Cuando la aplicación solicita la creación de un fichero de estado, la fachada envía la información de variables registradas a la capa de *checkpointing,* que da formato a los datos de modo que la capa de escritura sea capaz de procesarlos. Existen diferentes implementaciones de la capa de escritura. La elegida determinará el formato del fichero de estado y sus propiedades. La capa de escritura podría interactuar con la capa de utilidades, donde se encuentran implementaciones de algoritmos comúnmente utilizados (como compresión de datos o generación de códigos de detección de errores).

CPPC soporta códigos escritos en C y Fortran 77

FIG.1: DISEÑO DE CPPC



3. Ejemplo de uso

A modo ilustrativo, se presenta un ejemplo de la inserción de tolerancia a fallos mediante directivas CPPC en un código que simula el Juego de la Vida de Conway. La figura 2(a) muestra el código original, anotado. Hay básicamente tres secciones en dicho código. La primera inicializa variables

locales a cada proceso paralelo. La segunda crea estructuras MPI, como comunicadores y tipos de datos para modelar las comunicaciones entre procesos. La tercera parte es el bucle computacional, que calcula el tablero para la siguiente "generación" del juego.

En primer lugar, se anotará el bloque que crea las estructuras y tipos MPI. Debe ser re-ejecutado, dado que el estado MPI es no portable. De este modo, CPPC garantiza que todas las estructuras MPI serán recuperadas de un modo portable: ejecutando las mismas llamadas a función que fueron usadas en la ejecución original. Además, se sitúa un *checkpoint* al comienzo del bucle computacional.

Tras anotar el código, el precompilador realiza análisis para, como paso intermedio, insertar de forma automática las directivas *init, shutdown* y *register*. En este caso no se insertarán directivas *unregister,* dado que no se ha insertado más que un *checkpoint*.

A continuación, comienza la traducción de las directivas. El precompilador inserta variables para controlar el flujo de ejecución en reinicio, garantizando que sólo se visitan los bloques de código relevantes y coloca saltos condicionales y etiquetas de salto al final y principio, respectivamente, de cada bloque a ser ejecutado. El código resultante se muestra en la figura 2(b).

Utilizando *checkpointing* a nivel de variable se reduce de forma importante el tamaño de los ficheros de estado

En términos de flujo de ejecución, al reiniciar la aplicación, se inicializan las variables y después se ejecutan las funciones de inicialización de MPI y CPPC. Después, se ejecuta el salto condicional, transfiriendo el flujo de ejecución a la directiva *execute*. Tras recuperar el estado MPI, el flujo de control se mueve a la directiva *register*, restaurando los valores de sus variables asociadas. Finalmente, se alcanza el checkpoint. CPPC determina que todo el estado ha sido recuperado, y la ejecución continúa de forma normal.

4. Integración en un Grid

CPPC puede configurarse para utilizar formatos de salida portables para los ficheros de estado, de modo que las aplicaciones se pueden reiniciar sobre plataformas diferentes a aquellas que albergaron la ejecución original. Además, utilizando *checkpointing* a nivel de variable se reduce de forma importante el tamaño de los ficheros de estado. Añadiendo a esto la opción de aplicar compresión a los ficheros generados, se puede observar que CPPC es una infraestructura compatible con los entornos Grid: es portable y está diseñada para minimizar el impacto de la transferencia de ficheros a través de la red.

Para alcanzar una funcionalidad completa en un entorno Grid, la infraestructura requiere de un servidor de ficheros seguro, capaz de almacenar los ficheros de estado. Además, son necesarias herramientas de gestión y monitorización de recursos que sean capaces de detectar fallos en la ejecución y controlar los recursos necesarios para el reinicio, así como realizar la migración de los mismos y relanzar la ejecución.

Para alcanzar una funcionalidad completa en un entorno Grid, la infraestructura requiere de un servidor de ficheros seguro, capaz de almacenar los ficheros de estado

5. Trabajo futuro

Actualmente estamos trabajando en técnicas para extraer información que permita la gestión automática de las directivas execute y *checkpoint*. Esto convertiría a CPPC en una herramienta totalmente transparente, conservando sus características de portabilidad, escalabilidad y eficiencia.

Puede obtenerse una versión de prueba de CPPC en: http://*cppc.des.udc.es/*.

FIGURA 2: EJEMPLO DE INSERCIÓN DE TOLERANCIA A FALLOS MEDIANTE CPPC

```
int main( int argsc, char ** argsv ) {
  /* Variable definitions here */
  MPI_Init( &argsc, &argsv );

  MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
  MPI_Comm_size( MPI_COMM_WORLD,
    &worldsize );

  /* Check and obtain parameters */
  ...
  /* Initialize game boards */
  if( mytid == 0 ) {
    global_board = (char *)malloc( ... );
  }
  local_board = (char *)malloc(
    local_size * sizeof( char ) );
  #pragma cppc execute
  /* Initialize communications and types */
  MPI_Cart_create( ..., &gridComm );
  MPI_Comm_rank( gridComm, &mytid_grid );
  MPI_Cart_coords( gridComm, ...,
    my_coords );
  /* Initialize here datatypes for comms
    such as board row */
  ...
  /* Find TIDs for adjacent processes in
    the cartesian grid */
  MPI_Cart_shift( gridComm, ..., &tid_up );
  ...
  #pragma cppc end execute

  while( 1 ) {
    #pragma cppc checkpoint
    /* Send local board ends to neighbors
      */
    MPI_Send( local_board, 1, gameRow,
      tid_up, 0, gridComm );
    ...
    /* Receive from adjacent processes */
    MPI_Recv( local_board, 1, gameRow,
      tid_up, 2, gridComm );
    ...
    /* Compute next local iteration */
    ...
    /* Gather all boards in a global one
      for displaying it */
    MPI_Gatherv( local_board, ..., ...,
      global_board, ..., 0, gridComm );
    if( mytid_grid == 0 ) {
      showBoard( globalBoard );
      /* Code prompting user for next
        iteration */
      ...
      option = getchar();
    }
    /* Broadcast user option */
    MPI_Bcast( &option, 1, MPI_CHAR, 0,
      CommGrid );
    if( option == 'q' ) {
      MPI_Finalize();
      exit( 0 );
    }
  }
}
```

(a) Juego de la Vida Conway con directivas CPPC

```
int main( int argsc, char **argsv ) {
  /* Variable definitions here */
  void * cppc_jump_points[3] = {
    &&CPPC_EXECUTE_BLOCK_0,

    &&CPPC_REGISTER_BLOCK_0,
    &&CPPC_RESTART_BLOCK_0 };
  int cppc_next_jump_point = 0;
  MPI_Init( &argsc, &argsv );
  CPPC_Init( &argsc, &argsv );
  /* Conditional jump to next REC: code */
  if( CPPC_Jump_next() ) {
    const int jump_index =
      cppc_next_jump_point;
    cppc_next_jump_point++;
    goto * cppc_jump_points[ jump_index ];
  }

  MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
  MPI_Comm_size( MPI_COMM_WORLD,
    &worldsize );
  /* Check and obtain parameters */
  ... /* Not executed on restart */
  /* Initialize game boards */
  ... /* Not executed on restart */
CPPC_EXECUTE_BLOCK_0:
  /* Initialize communications and types */
  MPI_Cart_create( ..., &gridComm );
  ...
  /* Find TIDs for adjacent processes in
    the cartesian grid */
  ...
  /* Conditional jump to next REC */
  ...

CPPC_REGISTER_BLOCK_0:
  CPPC_Register( 0, local_size, 1,
    CPPC_INT, ..., CPPC_STATIC );
  local_board = CPPC_Register( 0,
    local_board, local_size, CPPC_CHAR,
    ..., CPPC_DYNAMIC );
  CPPC_Register( 0, mytid, 1, CPPC_INT,
    ..., CPPC_STATIC );
  CPPC_Register( 0, worldsize, 1, CPPC_INT,
    ..., CPPC_STATIC );
  /* Conditional jump to next REC */
  ...

  while( 1 ) {
CPPC_RESTART_BLOCK_0:
    CPPC_Do_checkpoint( 0 );
    /* Execution is resumed from this point
      */
    /* Original code for the loop here */
    ...
    if( option == 'q' ) {
      CPPC_Shutdown();
      MPI_Finalize();
      exit( 0 );
    }
  }
}
```

(b) Código tras pasar el precompilador CPPC

Agradecimientos

Esta investigación ha sido financiada por el Ministerio de Educación y Ciencia y fondos FEDER de la Unión Europea (Proyecto TIN-2004-07797-C02 y beca FPU AP-2004-2695) y por la Xunta de Galicia (Proyecto PGIDIT04TIC105004PR).

Gabriel Rodríguez, María J. Martín
(grodriguez@udc.es), (mariam@udc.es)
Patricia González, Juan Touriño
(pglez@udc.es), (juan@udc.es)
Ramón Doallo
(doallo @udc.es)
Grupo de Arquitectura de Computadores
Departamento de Electrónica y Sistemas
Universidade da Coruña

Referencias

[1] E.N. Elnozahy, L. Alvisi, Y-M. Wang, D.B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", *ACM Computing Surveys*, vol. 34, no. 3, pp. 375-408, 2002.

[2] G. Rodríguez, M. J. Martín, P. González, J. Touriño, "Controller/Precompiler for Portable Checkpointing", *IEICE Transactions on Information and Systems*, Vol. E89-D, No. 2, pp. 408-417, Feb. 2006.

[3] S-I. Lee, T.A. Johnson, R. Eigenmann, "Cetus – an Extensible Compiler Infrastructure for Source-to-source Transformation", en *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, College Station, TX, 2003, pp. 539-553, Springer.