Simulación bajo memoria compartida de un sistema distribuido que simula memoria compartida

Juan E. Navarro
Departamento de Ciencia de la Computación
P. Universidad Católica de Chile
Casilla 306 - Santiago 22 - CHILE
jnavarro@ing.puc.cl

Resumen

En este artículo se describe un simulador de un sistema distribuido que se construyó para realizar una evaluación preliminar de un nuevo protocolo para memoria compartida distribuida. La simulación es dirigida por ejecución y se efectúa en un monoprocesador, que divide su tiempo entre los múltiples procesos simulados. Un aspecto novedoso es el uso de la sobrecarga de operadores de C++ para entregar el control al simulador en los puntos en que los procesos efectúan accesos a la memoria compartida.

1. Introducción

En un sistema distribuido el paso de mensajes es el único medio de comunicación entre procesadores. Sin embargo, la programación de aplicaciones paralelas usando paso de mensajes es considerablemente compleja, lo que ha motivado la proposición de abstracciones para facilitar esta tarea. La memoria compartida distribuida o MCD [LH89] provee un espacio de direccionamiento global compartido por todos los procesadores de un sistema distribuido, ofreciendo al programador una abstracción que lo aísla de las complejidades inherentes al paso de mensajes, y le permite desarrollar aplicaciones distribuidas empleando el familiar y confortable paradigma de la memoria compartida.

Las implementaciones de MCD utilizan técnicas de replicación de datos para reducir la latencia de los accesos a la memoria compartida, pero la replicación introduce el clásico problema de la consistencia entre múltiples copias de un mismo objeto. En la medida que se impongan menos restricciones a las réplicas se pueden obtener implementaciones más eficientes, pero a costa de hacer más difícil (menos intuitivo) el desarrollo de aplicaciones. La consistencia causal [HA90] es un modelo de consistencia que ofrece un conveniente equilibrio: permite implementaciones eficientes sin complicar mayormente la programación.

En [NC95] se propuso un protocolo de consistencia causal para MCD. En [Nav96] se realizó una evaluación preliminar del protocolo, comparándolo con el protocolo de consistencia causal más ampliamente difundido, a saber, el propuesto por John y Ahamad en [JA93]. Este artículo se centra no en los protocolos ni en los resultados de la comparación, sino en el simulador que se construyó para realizar la comparación.

En la sección 2 se describe el modelo de sistema distribuido en el que se basa el protocolo. El contexto que determinó los objetivos del simulador se delinea en la sección 3. La sección 4 explica en detalle el simulador, y la sección 5 reseña otros simuladores en la literatura. Finalmente, las conclusiones se entregan en la sección 6.

2. Modelo del sistema

Un sistema distribuido consiste en una colección de nodos interconectados por una red confiable de comunicaciones (Figura 1).

Cada nodo se compone de un proceso, memoria privada del proceso, memoria local utilizada como caché de la MCD, y un administrador de la consistencia del caché. Los administradores se comunican entre sí exclusivamente mediante mensajes a través de la red de comunicaciones. El conjunto de procesos conforma una aplicación distribuida; cada uno de ellos interactúa con la MCD únicamente a través de operaciones de lectura y escritura sobre el caché local, con la supervisión del administrador de la consistencia. Si alguna de estas operaciones pudiera violar el modelo de consistencia, entonces el administrador la suspende para reanudarla una vez que haya efectuado en el caché las actualizaciones del caso.

Esta *supervisión* del administrador se suele implementar eficientemente dividiendo la memoria en páginas y utilizando técnicas de memoria virtual. El administrador protege aquellas páginas del caché que el proceso no pueda accesar sin violar la consistencia. Sólo si el proceso intenta una operación no permitida sobre una de estas páginas, se gatilla una interrupción (una falta de página) que entrega el control al administrador.

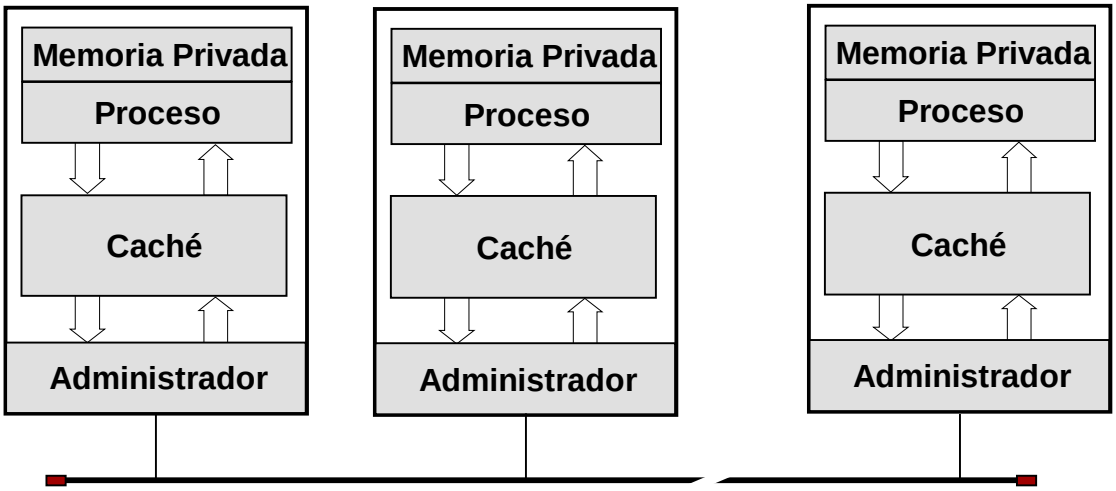

Figura 1: Modelo de sistema distribuido.

Gracias a este mecanismo los accesos legales se efectúan sin incurrir en ningún sobrecosto, y el proceso accesa transparentemente la MCD a través del caché.

3. Evaluación de un protocolo para MCD

El comportamiento de cualquier protocolo para MCD —y en general, el de arquitecturas paralelas— depende en gran medida de los patrones de sincronización y de acceso a la memoria de las aplicaciones que se ejecuten. Dichos patrones varían ampliamente de aplicación en aplicación; incluso, para una misma aplicación pueden depender de los datos de entrada. Por eso los protocolos se suelen evaluar en forma empírica, observando su rendimiento al ejecutar un conjunto de aplicaciones de prueba.

En este caso, se escogieron tres aplicaciones paralelas que, aunque simples, son representativas de patrones de sincronización y acceso diferentes: multiplicación de matrices, ordenamiento de enteros mediante recuento [CLR90], y resolución de una ecuación diferencial mediante el método de Jacobi de diferencias finitas [And91].

Sin duda, la medida más importante para determinar la calidad de un protocolo o arquitectura es el tiempo total de ejecución de las aplicaciones. Una alternativa para efectuar esa medición consistía en implementar un prototipo. Esta opción se descartó en esta primera etapa en favor de la simulación, por el alto costo derivado no sólo de la implementación misma, sino de la experimentación cuando hay varias máquinas involucradas. Por otra parte, la MCD requiere máquinas homogéneas, y en el mejor de los casos se habría contado con una docena.

La evaluación mediante simulación, en cambio, prometía escenarios con bastantes más nodos simulados que los que había físicamente disponibles. Se puede explotar una potencial ventaja adicional de la simulación, si se hace de manera tal que los experimentos sean repetibles. El problema era que no estaba claro cómo realizar simulaciones que modelaran adecuadamente (y con un bajo costo de desarrollo) la red de comunicaciones para poder estimar tiempos de ejecución. En realidad, el

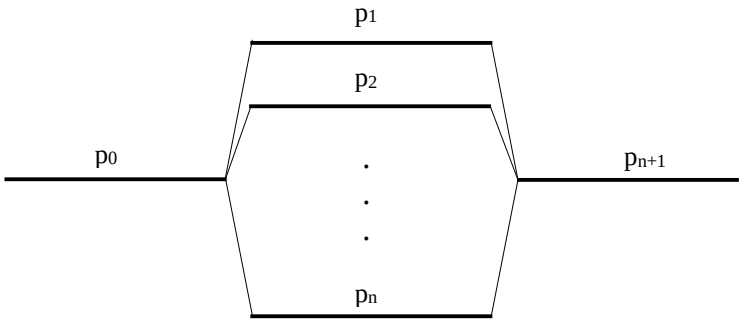

Figura 2: Ejecución paralela.

problema no era tal, puesto que para cumplir los objetivos de la evaluación no era indispensable medir tiempos de ejecución.

La Figura 2 bosqueja una ejecución típica de un programa paralelo (o de un fragmento de programa). Primero ejecuta un solo proceso en forma secuencial, p_0 (que típicamente realiza la inicialización de los datos), seguida luego de la ejecución en paralelo de n procesos, p_1 a p_n para terminar en la ejecución secuencial de otro proceso (que, por ejemplo, recopila o informa los resultados), p_{n+1}. En ese caso, el tiempo total de ejecución es $t_0+\max(t_1+...+t_n)+t_{n+1}$, en que t_i es el tiempo de ejecución del proceso p_i.

Durante la ejecución, los procesos gatillan el envío de mensajes para solicitar páginas de memoria a otros nodos. Un proceso permanece bloqueado desde que se comienza a enviar una solicitud, hasta que la respuesta se recibe y se procesa. Asimismo, el proceso no puede realizar trabajo útil mientras el procesador esté atendiendo una solicitud de otro nodo (suponiendo que hay un sólo procesador por nodo). En definitiva, una estimación del tiempo total que los procesos permanecen bloqueados es un buen indicio para comparar protocolos.

Evidentemente, situaciones anómalas en que bajo un protocolo A el tiempo total de bloqueo sea superior que con otro protocolo B, pero el tiempo de ejecución sea menor, son factibles. En efecto, en el tiempo total de ejecución no influye precisamente el tiempo total de bloqueo, sino el tiempo de bloqueo para el proceso paralelo que termina último. No obstante, no habría razones para pensar en que los tiempos de bloqueos no van a repartirse más o menos uniformemente entre todos los procesos. De hecho, en las aplicaciones de prueba escogidas, los procesos paralelos son siempre simétricos, realizan la misma cantidad de trabajo, y los protocolos a comparar no tienden a favorecer ni a perjudicar a ningún nodo en particular.

Ahora bien, los factores que inciden en el tiempo total de bloqueo incluyen el número total de mensajes, la longitud de esos mensajes, y el *overhead* o sobrecosto de procesamiento, es decir, el tiempo que los procesadores invierten en preparar solicitudes y procesar las respuestas, y en atender solicitudes de otros nodos. Medir esos tres factores por separado hace más manejable el problema, con el beneficio adicional de contar con información desagregada.

4. El simulador

Habiendo decidido qué convenía medir, el problema era cómo hacerlo. Tal como se mencionó en la introducción, el propósito de la simulación era obtener una evaluación preliminar (a corto plazo y bajo costo), lo que impuso ciertas restricciones. En primer lugar, la simulación debía hacerse en una sola máquina; distribuir la simulación era echarse problemas encima. Sin embargo, el inconveniente es que hay que repartir la memoria disponible entre todos los nodos simulados, lo que limita el tamaño de los problemas y la cantidad de nodos. En consecuencia, el ahorro de memoria es otro punto a tener en cuenta.

La repetibilidad de los experimentos también se consideró primordial, no sólo para poder evaluar adecuadamente algunas variantes del protocolo propuesto, sino también porque la repetibilidad facilita enormemente la depuración del protocolo (y también del simulador). Por último, aún cuando las aplicaciones de prueba son bastante simples, era atractivo que éstas pudieran ser construidas y depuradas como una aplicación paralela de memoria compartida convencional, y finalmente convertidas, con un mínimo de modificaciones, para ejecutar bajo el simulador. La eficiencia de la simulación se consideró un aspecto secundario.

4.1 Qué hay que simular

El administrador de la consistencia es una entidad reactiva que tiene una interfaz muy precisa con el resto de las componentes del sistema. Un administrador debe

- Responder a dos clases de eventos: faltas de página y solicitudes recibidas de otros administradores.
- Enviar solicitudes a otros administradores y recibir una respuesta.
- Escribir páginas del caché para actualizarlo, y leer páginas solicitadas por otros administradores.
- Manejar la protección de las páginas.

El protocolo de consistencia no es otra cosa que las rutinas del administrador. Por eso, estas rutinas no se consideran parte del simulador. Por otra parte, para cambiar de protocolo sólo hay que cambiar estas rutinas, lo que se facilita debido a la reducida interfaz con el resto del sistema. Gracias a eso, el mismo administrador que se implementó para la simulación se va a utilizar casi sin modificaciones, para implementar un prototipo.

La simulación debe emular N nodos, cada uno con su memoria local y su caché para la MCD, su proceso, y su administrador. Hay que simular el envío y recepción de mensajes, el hardware de paginación, y la ejecución paralela de los N procesos.

Se escogió C++ para implementar el simulador, principalmente porque la sobrecarga de operadores facilita considerablemente la simulación del hardware de paginación. Se optó por simular todos los elementos dentro de un mismo espacio de direccionamiento, representando cada proceso mediante una hebra o *thread* [Doe87] o proceso liviano. Según se ve a continuación, el espacio compartido hace que la simulación sea más eficiente, permite ahorrar memoria, y facilita la simulación del envío y recepción de mensajes. Lo anecdótico es que se simula mediante memoria compartida un sistema distribuido que simula memoria compartida.

4.2 Nodos

Cada nodo Q es una instancia de la clase Node, que contiene las estructuras de datos usadas por el administrador, y un arreglo pageTable en el que se almacena información de las páginas que conforman el caché de la MCD, incluyendo un puntero al contenido de la página. O sea:

```
struct Page {
   char *data;
   int access;   // INVALID, READONLY, READWRITE
     otros datos usados por el administrador
};

class Node {
   Page pageTable[CACHESIZE];
     otros datos usados por el administrador
public:
   void ReadFault (int pageid);
   void WriteFault (int pageid);
   void Service (int from, char* req, char *answer);
}

node Node[NODES];
```

Los métodos de la clase Node corresponden a los manejadores de eventos del administrador; por eso hay funciones para atender faltas de lectura, faltas de escritura, y solicitudes de otros nodos. En el arreglo global node se declara una instancia de la clase Node por cada nodo simulado.

4.3 Hardware de paginación

En la MCD basada en páginas, el espacio de direccionamiento se divide en páginas virtuales, y cada variable compartida tiene una dirección virtual, global, que corresponde a alguna página virtual. En cada instante, un subconjunto de esas páginas se encuentra en el caché de cada nodo. Si un proceso intenta leer o escribir una variable cuya dirección virtual corresponde a la página virtual p, el hardware de paginación busca en la tabla de páginas la entrada correspondiente a p. Si la operación es incompatible con la protección de la página se gatilla una interrupción que es atendida por el administrador de la consistencia. Más en detalle, una lectura sobre una página inválida gatilla una *falta de lectura*, y una escritura sobre una página inválida o sólo para lectura gatilla una *falta de escritura*. El administrador realiza las acciones que correspondan (que dependen del protocolo y eventualmente incluyen la invalidación de otras páginas), cambia el nivel de protección de p, y reanuda el proceso, quien ahora puede completar la operación. Cuando la operación es compatible con el nivel de protección de p, el hardware de paginación sólo transforma la dirección virtual de la variable en su dirección física en el caché.

Para emular ese comportamiento, el simulador debe interceptar cada acceso a las variables compartidas, determinar si corresponde gatillar una falta de lectura o una falta de escritura, y transformar el acceso en una operación sobre el caché. Eso se logra con la sobrecarga de operadores de C++.

Por ahora, el simulador sólo soporta variables y arreglos compartidos de tipos numéricos, los que deben declararse anteponiendo el prefijo Shared_. Por ejemplo,

```
#include "shared.h"
Shared_int i,j;
Shared_double matrix[N][M];
```

En shared.h se define una clase Shared_xx para cada tipo numérico xx. Cada instancia de Shared_xx no contiene el valor de la variable, sino su dirección virtual en la MCD. Las direcciones las asigna el constructor de la clase en el orden en que se van construyendo las instancias, cuidando que cada variable quede enteramente contenida en una sola página. Por ejemplo, el constructor de Shared_int es:

```
Shared_int::Shared_int()
{
   if (currentOffSet + sizeof(int) > PAGESIZE) {
      currentPage++;
      currentOffSet=0;
   }
   address = currentPage*PAGESIZE+currentOffSet;
   currentOffSet += sizeof(int);
}
```

donde currentPage y currentOffSet son variables globales inicializadas en 0, y address es el único miembro no funcional de Shared_int.

El hardware de paginación se simula sobrecargando todos los operadores sobre los tipos básicos, para que operen también sobre tipos compartidos. Evidentemente, la semántica de cada operador se preserva, pero el acceso a la variable se redirige al caché, gatillando una falta de página, si corresponde. Por ejemplo, la comparación entre un int y un Shared_int que se ejecuta en el nodo Q se sobrecarga de la siguiente manera:

```
int operator< (int i, Shared_int J)
{
   int page   = J.address/PAGESIZE;
   int offset = J.address%PAGESIZE;
   if (node[Q].pageTable[page].access == INVALID)
      node[Q].ReadFault(page);
   int *p = node[Q].pageTable[page].data + offset;
   return ( i < *p);
}
```

En castellano, si la página page=J.address/PAGESIZE donde J reside no está presente en el caché de Q, se gatilla una falta de lectura, la que se simula llamando directamente al manejador de esa clase de eventos del nodo. Se supone que el manejador remedia la situación, y trae una copia de esa página al caché, la que queda apuntada por pageTable[Q].data. Finalmente, se usa el valor J en el caché para hacer la comparación. La asignación de un Shared_int a otro se hace de la siguiente forma:

```
int operator= (Shared_int I, Shared_int J)
{
   int pageI   = I.address/PAGESIZE;
   int offsetI = I.address%PAGESIZE;
   int pageJ   = J.address/PAGESIZE;
   int offsetJ = J.address%PAGESIZE;

   if (node[Q].pageTable[pageI].access != READWRITE)
      node[Q].WriteFault(pageI);
   if (node[Q].pageTable[pageJ].access == INVALID)
      node[Q].ReadFault(pageJ);

   memcpy (node[Q].pageTable[pageI].data + offsetI,
           node[Q].pageTable[pageJ].data + offsetJ,
           sizeof(int));
}
```

4.4 Mensajes

Para simular el paso de mensajes se saca partido de que todas las interacciones entre los administradores son del tipo solicitud-respuesta. La solicitud es un mensaje (una secuencia de caracteres) en que se indica el tipo de solicitud y otros datos específicos, según los cuales el receptor prepara la respuesta y la envía al solicitante, como otra secuencia de caracteres. Esta interacción se emula simplemente con una invocación a una función. Un nodo S envía una solicitud s al nodo R, simplemente haciendo

```
char r[MAXREPLYSIZE];
node[R]->Service (S, s, r);
```

La función Node::Service prepara la respuesta según el tipo de solicitud y otros argumentos empaquetados en s, y la escribe en r. En algunos casos R debe delegar a otro nodo, Q, la responsabilidad de responder a S. En una implementación real, R reenvía el mensaje a Q, indicando que el remitente es S, para que Q le responda directamente. En el simulador, se hace así:

```
void Node::Service (int from, char* req, char *answer)
{
   if (Hay que reenviar a Q)
      node[Q]->Service (from, req, answer);
   else {
      Procesar el mensaje
      Escribir respuesta en r
   }
}
```

4.5 Procesos y paralelismo

Cada proceso se simula mediante una hebra. Entonces, para escribir una aplicación paralela para el simulador, se puede escribir y depurar primero usando hebras, pero sin el simulador, y después, con mínimos cambios, integrar la simulación. Una aplicación multihebra sin el simulador tiene más o menos la siguiente estructura:

```
/* variables compartidas */
int i,j,...
double x,y,...

Task()
{
   variables privadas
   código
}

main()
{
   /* La hebra principal inicializa variables compartidas */
   i=0; x=0; ...

   /* Ejecución paralela: se crean NODES-1 hebras... */
   for (h=1; h<NODES; h++) newthread(h,Task);
   /* ...y la hebra principal también hace su parte */
   Task();

   /* Imprimir resultados cuando todas las hebras terminen */
   printf (resultados);
}
```

donde `newthread(h, Task)` crea un nuevo proceso liviano con identificador `h` que ejecuta la función `Task`. La hebra principal tiene identificador 0. Para ejecutar este programa bajo el simulador, sólo hay que cambiar la declaración de las variables compartidas a:

```
#include "protocols.h"
#include "shared.h"
Shared_int i,j,...
Shared_double x,y,...
```

y agregar en el `main()` una llamada a una función que inicializa los administradores.

Lo que queda por resolver es cómo simular adecuadamente la ejecución paralela de los procesos. El paquete de hebras usa *round-robin*, asignando un quantum de tiempo a cada hebra. Esta política no es apropiada, primero, porque cuando se envía una solicitud, el proceso no puede efectuar trabajo útil mientras no llegue la respuesta; *round-robin* no toma en cuenta este factor. Y segundo, porque las ejecuciones no son repetibles; por diversas razones, las hebras no ejecutan siempre exactamente el mismo número de instrucciones dentro de su quantum y, por consiguiente, distintas ejecuciones de una misma aplicación siguen distintos caminos.

Para resolver el segundo punto, es necesario que cada hebra ceda el procesador en puntos bien específicos de la ejecución. El candidato obvio es el acceso a variables compartidas, pues es en esos puntos donde el simulador tiene el control de la situación. En consecuencia, se modificó el paquete de hebras para que no expropiara el procesador; cada hebra lo cede explícitamente (mediante una llamada a `yield()`), cada vez que usa un operador que involucre variables compartidas.

El primer punto se resolvió haciendo que cada hebra ceda una cantidad de turnos, sin hacer trabajo útil, por cada mensaje que deba esperar. El resultado es que la hebra se retrasa respecto de las demás, que es precisamente el efecto que se requiere.

Para ejemplificar, el operador `<` y la función `Node::Service` quedan, en definitiva, así (los cambios están en negrita):

```
int operator< (int i, Shared_int J)
{
   int page   = J.address/PAGESIZE;
   int offset = J.address%PAGESIZE;
   if (node[TID].pageTable[page].access == INVALID)
      node[TID].ReadFault(page);
   int *p = node[TID].pageTable[page].data + offset;
   yield();
   return ( i < *p);
}

void Node::Service (int from, char* req, char *answer)
{
   if (Hay que reenviar a Q)
       for (int i=0; i<TURNSPERMESSAGE; i++) yield();
      node[Q]->Service (from, req, answer);
   else {
       for (int i=0; i<TURNSPERMESSAGE; i++) yield();
      Procesar el mensaje
      Escribir respuesta en r
       for (int i=0; i<TURNSPERMESSAGE; i++) yield();
   }
}
```

TID es una macro que se expande al identificador de la hebra que está ejecutando. Si la hebra gatilla el envío de mensajes, entonces se *castiga*, obligándola a ceder TURNSPERMESSAGE turnos por la solicitud, TURNSPERMESSAGE turnos por la respuesta, y otros tantos por cada reenvío.

En el fondo, la simulación de la red se concentra en el valor de TURNSPERMESSAGE que resume la relación entre la velocidad de los procesadores, los retardos de la red, y la cantidad de trabajo que se alcanza a realizar entre accesos a la memoria compartida. Aunque es una forma tosca de integrar los efectos de la red, lo cierto es que la ejecución simulada es una ejecución que efectivamente puede ocurrir en la realidad, y no hay ninguna razón para pensar que esta decisión pueda favorecer artificialmente a ninguno de los protocolos.

4.6 Cómo economizar memoria

Si dos o más nodos tienen una copia de la misma página virtual, p, es usual que el contenido de todas las copias sea idéntico (bajo consistencia estricta, las copias deben ser iguales, pero con modelos más relajados no siempre es así). Aprovechando que la simulación se está haciendo bajo memoria compartida, se puede economizar memoria si se usa una única copia cuando los contenidos son idénticos.

No obstante, si uno de los nodos quiere escribir en esa página, entonces, debe *separarse* una copia para él. Para llevar a la práctica esta idea, es necesario saber, para cada página física, cuántos nodos tienen un puntero a ella. Cuando un nodo Q envía una página p a otro nodo R, se debe hacer

```
node[R].pageTable[p].data = node[Q].pageTable[p].data
```

e incrementar el contador de punteros hacia esa página. Si un nodo quiere escribir en una página cuyo contador es mayor que 1, entonces previamente debe decrementar el contador, y crear una nueva página física inicialmente idéntica a la original (y con su contador en 1). Por último, al invalidar una página, un nodo debe decrementar el

contador; en caso que llegue a 0, debe además liberar la memoria asignada a la página.

4.7 Mediciones

Según lo señalado en la sección 3, había tres factores que medir: número de mensajes, cantidad de información transmitida y sobrecosto de procesamiento. Los dos primeros se miden en `Node::Service`, ya que todos los mensajes pasan por ahí. El sobrecosto de procesamiento se determina midiendo el tiempo de procesador consumido por las rutinas de los administradores.

Los experimentos se realizaron en un Digital DECServer 3000/700 equipado con un procesador Alpha A21064A a 225 MHz y 256 MB de memoria RAM, y operando bajo OSF1 V3.0. Aún sin incorporar el mecanismo de ahorro de memoria descrito en la sección anterior, se logró simular hasta 64 nodos, con problemas de tamaño no despreciable (por ejemplo multiplicación de matrices de 450x450 y ordenamiento de tres millones de números). En cuanto a eficiencia, la simulación tarda cerca de 10 veces lo que demora la ejecución directa del programa original; éste es el castigo impuesto por interceptar todos los accesos a la memoria compartida.

5. Trabajos relacionados

La simulación se ha usado extensivamente en los últimos años como un mecanismo para evaluar arquitecturas paralelas, evitando los costosos y largos ciclos que involucra construir un prototipo. Los simuladores varían mucho en flexibilidad, acuciosidad y eficiencia, puesto que son objetivos contrapuestos [DGH91]. Una simulación más precisa requiere considerar más detalles, lo que reduce la eficiencia. Un simulador flexible inevitablemente debe soslayar algunos detalles. Las técnicas de simulación pueden clasificarse en tres categorías [ACLS93]: simulación estadística o dirigida por trazas, simulación funcional, y simulación por ejecución directa o dirigida por ejecución.

En general, se supone que las ejecuciones son *deterministas por intervalos*, es decir, la ejecución de cada proceso se divide en intervalos deterministas de computación local, separados por eventos globales. Los eventos globales corresponden a operaciones de sincronización y de comunicación o acceso a datos compartidos, y son los únicos que pueden afectar la computación local.

La simulación dirigida por trazas consta de dos etapas. En la primera se recopila información acerca de los tiempos en los que cada proceso se ve involucrado en una operación de comunicación o de sincronización; es decir, se generan las trazas del programa. La relevancia de la computación local se reduce al impacto que ésta tiene en las trazas, o sea, simplemente a la duración de los intervalos [DGH91]. En la segunda etapa, se usan la trazas (ya no el programa) para conducir la simulación de la arquitectura o protocolo en estudio. Las trazas se pueden generar ejecutando el programa en una máquina secuencial o paralela. La simulación dirigida por trazas es muy eficiente debido a que en ningún momento se requiere simular la ejecución del programa. Sin embargo, sufre de poca precisión porque el ambiente en que se generan las trazas rara vez coincide con el ambiente objetivo [GH93]; en consecuencia, en el ambiente objetivo la ejecución probablemente seguiría un camino distinto.

Los simuladores funcionales se centran en los detalles de más bajo nivel de una arquitectura, simulando instrucción por instrucción. La simulación es extremadamente detallada y, por ende, precisa, pero varios órdenes de magnitud más lenta que una ejecución real. Ya que los programas de interés suelen ser de largo aliento, es difícil simular ejecuciones completas. La simulación por ejecución directa se sitúa en un punto intermedio entre las anteriores, tanto en cuanto a eficiencia como a precisión. Las simulaciones de esta naturaleza explotan las similitudes entre la arquitectura objetivo y la arquitectura anfitriona (aquella en la que se realiza la simulación), para ejecutar directamente la aplicación, simulando sólo aquellas operaciones que no son soportadas o que tienen características diferentes en la arquitectura anfitriona. En esta categoría caen Tango, Proteus, Wisconsin Wind Tunel (WWT), Maya, y el simulador aquí descrito. Salvo el último, todos los simuladores entregan como resultado el tiempo (simulado) total de ejecución de las aplicaciones. Tango [DGH91] genera un proceso por cada procesador en la arquitectura objetivo, los que ejecutan directamente el código de la aplicación, pero comunicándose con un proceso central de simulación ante la mayoría de los accesos a la memoria. Para que el simulador intercepte estos accesos, las aplicaciones de prueba deben usar la memoria compartida a través de un conjunto de macros. El proceso central coordina el orden de ejecución de los procesos. Proteus [BDCW92] usa hebras en lugar de procesos para reducir los costos de los cambios de contexto. Un preprocesador del código fuente reemplaza los accesos a memoria por llamadas a funciones del simulador. WWT [RHL+93] difiere de los anteriores en dos aspectos: los procesos ejecutan en paralelo en un multiprocesador (un CM-5), y sólo las referencias a elementos de la memoria que no están en el caché son capturadas por el simulador. Ambos factores aceleran notablemente la simulación. Inspirado en WWT y apuntando a la portabilidad, Maya [ACLS93] utiliza un sistema distribuido para paralelizar la simulación. De manera similar a Proteus, un preprocesador traduce los accesos a la memoria compartida a llamadas al simulador.

6. Conclusiones

En comparación con otros simuladores, el que se ha descrito no destaca ni por eficiencia, ni por precisión ni por flexibilidad, pero sí por brevedad —apenas unas 500 líneas de código (sin contar el código de los protocolos)— y por el uso de la sobrecarga de operadores de C++ para interceptar los accesos a la memoria.

Lo más importante, en todo caso, es que cumplió los objetivos que impulsaron su construcción. De manera rápida y simple, con una solución *ad-hoc*, fue posible depurar el protocolo bajo estudio, y compararlo con el de John y Ahamad.

En cuanto al objetivo principal de esta investigación, la comparación de los protocolos, cabe mencionar que, en todos los experimentos, los valores de todas las mediciones son mejores para el protocolo propuesto que para el de John y Ahamad, y con frecuencia sustancialmente mejores. Los resultados obtenidos hacen pensar que bajo otras circunstancias (por ejemplo, usando otras aplicaciones de prueba, o mediante una simulación más acabada que permita estimar tiempos de ejecución, o evaluando directamente una implementación de los protocolos) las diferencias entre un protocolo y otro podrían cambiar en magnitud, pero no en signo. Estos resultados motivaron la continuación de esta línea de investigación, cuyo próximo paso es la implementación de un prototipo de MCD usando el protocolo propuesto. De todas formas se piensa seguir usando el mismo simulador para conducir pequeños experimentos para evaluar y depurar variantes del protocolo.

Agradecimientos

El autor agradece a Alvaro Campos por impulsar la presentación de este artículo y por sus valiosos comentarios. Este trabajo fue parcialmente financiado por la Dirección de Investigación y Postgrado de la Pontificia Universidad Católica de Chile y por Fondecyt, a través de los proyectos 96/08E y 196-0381, respectivamente.

Referencias

[ACLS93] D. Agrawal, M. Choy, H.-V. Leong, and A. K. Singh. Maya: A simulation platform for distributed shared memories. In Proceedings of the 8th Workshop on Parallel and Distributed Simulation, pages 151-155, 1993.

[And91] G.R. Andrews. *Concurrent Programming*. Benjamin/Cummings, Redwood City, California, 1991.

[BDCW92] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. In *Proceedings of the SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, pages 247-248, New York, NY, USA, June 1992. ACM Press.

[CLR90] Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.

[DGH91] Helen Davis, Stehpen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II-99-II-107, Boca Raton, FL, August 1991. CRC Press.

[Doe87] T.W. Doeppner. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Department of Computer Science, Brown University, June 1987.

[GH93] Stephen R. Goldschmidt and John L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (SIGMETRICS93)*, pages 146-157, 1993.

[HA90] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302-311, May 1990.

[JA93] R. John and M. Ahamad. Causal memory: Implementation, programming support and experiences. Technical Report 93/10, College of Computing, Georgia Institute of Technology, 1993.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321-359, November 1989.

[Nav96] J.E. Navarro. Un protocolo de consistencia causal para memoria compartida distribuida. Master's thesis, Departamento de Ciencia de la Computación, P. Universidad Católica de Chile, 1996.

[NC95] J.E. Navarro and A.E. Campos. Coherent causal consistency in distributed shared memory. In *Proceedings of the XV International Conference of the Chilean Computer Science Society*, pages 328-338, November 1995.

[RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin wind tunnel: Virtual prototyping of parallel computers. In Blaine D. Gaither, editor, *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, volume 21-1 of *Performance Evaluation Review*, pages 48-60, New York, NY, USA, May 1993. ACM Press.