Simulación Paralela de Sistemas de Sucesos Discretos

José Miguel Alonso
Dept. Arquitectura y Tecnología de Computadores
Universidad del País Vasco
Apdo. 649, 20080 San Sebastián
acpmialj@si.ehu.es

Ramón Beivide Palacio *
Departamento de Electrónica
Universidad de Cantabria
Av. de los Castros s/n, 39005 Santander

Resumen: La simulación por computador es una herramienta esencial para la evaluación de nuevas propuestas de diseño en arquitectura de computadores. Revisamos dos formas básicas de simular sistemas: simuladores dirigidos por el tiempo y simuladores dirigidos por los sucesos. Se justifica la necesidad de paralelizar estos algoritmos, si se quieren acelerar las simulaciones. Se presentan los dos principales métodos de simulación paralela dirigida por los sucesos: conservadores y optimistas, analizando sus ventajas y sus inconvenientes. Se concluye que los métodos de simulación paralela pueden suponer mejoras considerables en la velocidad de los simuladores, pero sólo si el modelo simulado posee determinadas características.

Abstract: Computer simulation is an essential tool for the evaluation of new proposals in the computer architecture field. We give an overview of the two basic ways of simulating systems: time-driven and event-driven. If higher speeds are needed, it is necessary to parallelize these algorithms. We introduce the two main methods of parallel discrete event-driven simulation: conservative and optimistic, and analise their pros and cons. It is concluded that PDES can offer good levels of speed, but only if the model being simulated obeys some constraints.

* Este trabajo ha contado con la ayuda económica de la CICYT, bajo contrato TIC93-0243

1 Introducción

En los últimos años, los avances que se han producido en las tecnologías de diseño hardware han sido superiores a los que han tenido lugar en otras áreas relacionadas el tratamiento automático de la información. Los programas que se ejecutan en los computadores actuales son más rápidos y más complejos que antes, fundamentalmente gracias a que los circuitos integrados que subyacen en ellos son también más rápidos y más complejos. Aunque todavía se pueden esperar nuevos avances en lo que se refiere al diseño de procesadores, se prevé que las mejoras que se obtengan, aunque importantes, no serán suficientes para dar soporte a aplicaciones con demanda creciente de cálculo. Parece comunmente aceptado que, en un futuro cercano, el desarrollo de aplicaciones que requieren gran cantidad de cálculo, como son aquellas relacionadas con el tratamiento de sonido o imágenes en tiempo real, la predicción del tiempo o de movimientos sísmicos, genoma humano y otra serie de aplicaciones desafiantes sólo resolubles al alcanzar la barrera del TeraFLOPS, será posible con la introducción de sistemas masivamente paralelos (MPPs), construidos en base a centenares o miles de elementos de proceso sencillos [46].

Hoy en día ya es posible adquirir, a precios razonables, sistemas paralelos con memoria distribuida, que pueden servir de punto de partida para familiarizarse con las nuevas tecnologías que incorporarán los futuros supercomputadores. Uno de los problemas que plantean estas máquinas paralelas de memoria distribuida es la falta de sistemas de desarrollo de software que permitan extraer su potencial de forma efectiva y amistosa. En la actual situación, el desarrollo de aplicaciones paralelas es exageradamente costoso, y el aprovechamiento que se hace de las máquinas disponibles no está a la altura de lo esperado. En este artículo nos vamos a ocupar de una clase importante de aplicaciones susceptibles de ser ejecutadas en procesadores masivamente paralelos de memoria distribuida: las simulaciones por computador.

Como es bien sabido, la práctica totalidad de los avances que se producen en el ámbito de la arquitectura de computadores exige el desarrollo de simuladores que permitan predecir, con cierto grado de precisión, el comportamiento de un nuevo sistema antes de entrar en las fases finales de diseño y fabricación. En nuestro grupo de investigación sobre paralelismo se ha utilizado ampliamente la simulación, como una herramienta imprescindible para evaluar nuestras propuestas de diseño de mecanismos de interconexión para MPPs [3, 1, 19]. Tradicionalmente, hemos desarrollado diferentes herramientas de simulación sobre arquitecturas monoprocesador. A medida que los sistemas a simular se hacen más complejos, o queremos elevar el grado de detalle del modelo elegido, el tiempo de CPU utilizado por los simuladores se hace insostenible [2]. Esto nos ha llevado a prestar atención a las técnicas para

realizar simulaciones distribuidas, de tal manera que se pueda explotar el paralelismo existente en las nuevas arquitecturas y así acelerar sustancialmente nuestras herramientas de análisis y evaluación.

Desafortunadamente, los algoritmos de simulación utilizados habitualmente en sistemas monoprocesador resultan difíciles de paralelizar [27]. Si se quiere obtener un buen rendimiento de los multicomputadores presentes y futuros, deben desarrollarse nuevos algoritmos específicamente adaptados a esos entornos, es decir, capaces de aprovechar al máximo el paralelismo de las máquinas empleadas. Por este motivo, nuestro grupo de investigación se ha planteado como objetivo proponer técnicas de simulación paralela, orientadas a resolver ciertos problemas que nos interesan: análisis y evaluación de los mecanismos de interconexión que intervienen en las arquitecturas masivamente paralelas. Este artículo representa un primer paso en este camino, ya que vamos a considerar aquí la revisión de los trabajos más significativos dentro del área de la simulación paralela.

2 Modelado y Simulación

Como punto de partida vamos a empezar por describir el problema que queremos resolver, esto es, la simulación por computador de sistemas reales [45, 29, 4]. Esta actividad consta de, al menos, estas fases:

a) Estudio del sistema real, para entender sus características.

b) Modelado del mismo, fase en la que se decide qué características del sistema son relevantes para la simulación, y en la que se eliminan las irrelevantes.

c) Simulación del modelo usando computadores.

d) Análisis de los resultados de la simulación, para así entender y predecir el comportamiento del sistema real.

En este trabajo nos centramos en el problema de la simulación por computador, dejando a un lado los procesos de estudio de sistemas reales y el modelado, ya que dependen de los experimentos que se pretenda realizar. Tampoco nos preocupamos, por el mismo motivo, de los procesos de análisis de los resultados obtenidos.

Los sistemas a simular se suelen dividir en dos categorías: discretos y continuos. Un *sistema discreto* es aquel en el que el *estado* del mismo cambia únicamente en un conjunto discreto de instantes en el tiempo. En contraposición, un *sistema continuo* es aquel en el que el estado del sistema varía continuamente en el tiempo.

Los modelos de simulación que se pueden realizar a partir de un sistema suelen clasificarse en estáticos o dinámicos, deterministas o estocásticos, y discretos o continuos. Un modelo es

estático cuando trata de representar el estado de un sistema en un punto particular del tiempo, y es *dinámico* cuando trata de representar la evolución del sistema a lo largo de un determinado intervalo de tiempo. Un modelo es *determinista* cuando tiene un conjunto conocido de valores de entrada y produce un único conjunto de valores de salida, mientras que es *estocástico* cuando se usan como entrada una o más variables aleatorias. Entradas aleatorias producen salidas aleatorias, luego estas salidas sólo pueden ser consideradas como una estimación del comportamiento real del modelo. Respecto a la división de los modelos en *discretos* y *continuos*, tomamos como referencia los ya definidos conceptos de sistema continuo y sistema discreto. Es importante destacar que no tiene necesariamente que modelarse un sistema de una clase mediante un modelo de la misma clase: es posible, por ejemplo, usar un modelo discreto para simular un sistema continuo. El tipo de modelo que se elija depende tanto de la naturaleza del sistema como de los resultados que se deseen obtener.

Dado el tipo de sistemas que a nosotros nos interesa, vamos a centrarnos en los modelos de simulación discretos, dinámicos y estocásticos, y sobre ellos presentaremos diferentes formas de realizar una simulación por computador. A continuación vamos a caracterizar los modelos que van a ser centro de nuestro estudio, con el objetivo de definir la terminología utilizada, y poder comparar las diferentes formas de simular estos modelos.

Se considera que un sistema real o *sistema físico* se modela como un conjunto de uno o varios *procesos físicos* (*PF*s) que evolucionan en el tiempo. Se supone que todos los *PF*s tienen la misma visión del tiempo, esto es, que de alguna forma existe un *reloj global* que sirve de referencia al sistema completo. Cada proceso físico opera de forma autónoma, excepto para interactuar con otros procesos físicos en el sistema. Estas interacciones, denominadas *sucesos*, se producen en instantes de tiempo discretos, y se modelan a través de un intercambio de *mensajes* entre parejas de *PF*s. La información que contienen los mensajes es la siguiente: el suceso que representan y el instante previsto en el que debería acontecer ese suceso (su marca de tiempo o *timestamp*).

Se está asumiendo en todo momento que los *PF*s tienen cierta capacidad de predicción de lo que va a acontecer en un futuro cercano. Cuando un *PF* prevé que, en un instante *t* del futuro, va a interactuar con otro, y que la interacción va a ser del tipo *e*, *planifica* la ocurrencia de un suceso en el otro *PF*. Esta planificación se modela mediante el envío del mensaje <*e*, *t*> (ver Figura 1).

El contenido de un mensaje depende de las características del *PF* emisor (estado inicial, reglas de comportamiento) y de los mensajes que ese *PF* haya recibido previamente. Un mensaje <*e*, *t*> será *consumido* en el *PF* receptor cuando el reloj alcance el valor *t*. Como resultado, el estado del *PF* cambiará según corresponda a la realización de la interacción modelada por *e*. Este cambio de estado puede, a su vez, provocar la planificación de nuevos sucesos para el futuro y, por lo tanto, el envío de nuevos mensajes.

Cabe la posibilidad de que un suceso planificado para el futuro deba *cancelarse* antes de que ocurra (antes de que el reloj alcance el valor del timestamp del suceso), de ahí que se diga que el timestamp indica el instante *previsto* de ocurrencia de un suceso. Un suceso planificado (un mensaje enviado) puede cancelarse mediante el envío de mensajes especiales de cancelación.

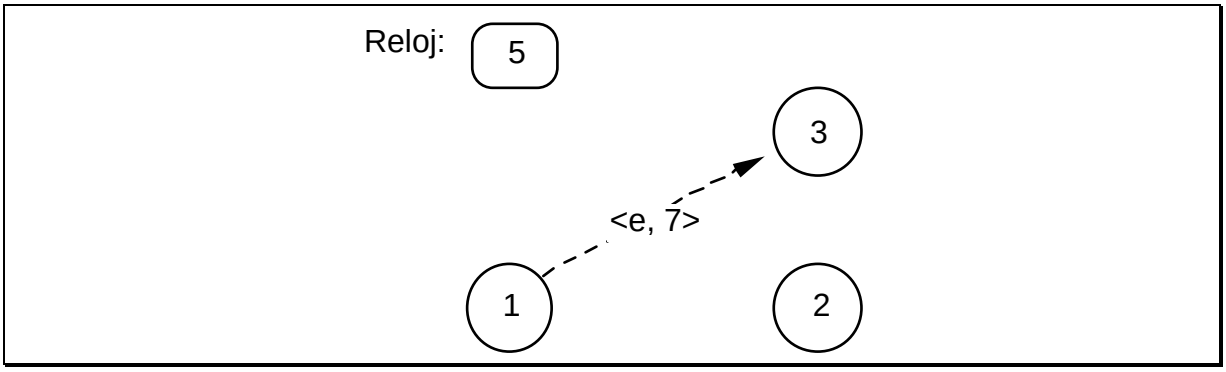

Figura 1: Una colección de 3 procesos físicos que forman parte de un sistema físico. Un reloj, común para los tres *PF*s, indica que la hora es las 5. El *PF* 1 planifica un suceso *e* para que ocurra a las 7 en el *PF* 3. Para ello envía el mensaje <*e*, 7>.

Una vez que tenemos un sistema modelado de la forma descrita, puede procederse a su simulación por computador. Podemos hacer una simulación *dirigida por el tiempo*, o bien una simulación *dirigida por los sucesos*. A continuación describimos un algoritmo secuencial para realizar una simulación dirigida por el tiempo. Los elementos (estructuras de datos) empleados son:

- Una variable *reloj*, que representa el avance del tiempo en el sistema físico. En cada paso del algoritmo, este reloj se incrementa en una unidad de tiempo. El valor del reloj indica hasta qué instante de tiempo se ha simulado la evolución del sistema.
- Un conjunto de variables de *estado*, que representan el estado de los diferentes procesos físicos del sistema. Entre la información de estado asociada a un *PF* se encuentran también, de forma explícita o implícita, los mensajes recibidos pendientes de ser consumidos, que representan sucesos planificados para su futuro.

Además se utilizan un conjunto de variables para la recogida de diferentes estadísticas que describen las características más importantes del sistema simulado.

Tal como se ha dicho, en cada paso del algoritmo se incrementa en una unidad de tiempo el valor del reloj. Tras ello se examinan las variables de estado para comprobar qué sucesos están pendientes de ocurrir (qué mensajes se tienen almacenados), y se consumen los mensajes cuyo timestamp es igual al valor del reloj. Esto equivale a decir que se simula la ocurrencia de todos los sucesos que estan planificados para ese preciso instante de tiempo. Consumir un mensaje en un *PF* puede producir los siguientes efectos:

- Un cambio en el estado del *PF*, que se reflejará modificando las variables de estado correspondientes.
- La planificación de nuevos sucesos para el futuro, esto es, el envío de mensajes a otros *PF*s.
- La cancelación de sucesos que se habían planificado con anterioridad.

Los pasos descritos se repiten hasta que se termina la simulación, normalmente cuando el reloj alcanza un valor prefijado o cuando que se llega a un determinado estado en el sistema.

Tal como se ha visto, es el avance en el reloj el que marca el avance en la simulación, y en cada paso de simulación se avanza en *una* unidad de tiempo. En muchos sistemas, sin embargo, los sucesos ocurren con bastante diferencia (temporal) entre ellos, de tal manera que, en la mayoría de los pasos del algoritmo, no hay acción ninguna que realizar, o bien hay muy pocas. Esta circunstancia lleva a la utilización de otras estrategias de simulación en las que el reloj que simula el transcurso del tiempo puede avanzar con más rapidez. Se trata de las técnicas de simulación dirigidas por los sucesos.

Los elementos más importantes en una simulación dirigida por los sucesos son un reloj y un conjunto de variables de estado, como en el caso anterior, a los que se añade una nueva estructura de datos: una *lista de sucesos* planificados para el futuro, que almacena todos los mensajes <suceso, timestamp> que se intercambian los *PF*s, junto con información sobre su destinatario. Cada uno de estos mensajes se entregará en su destino (será consumido) cuando el reloj alcance el timestamp del mensaje, a no ser que el suceso representado por ese mensaje sea cancelado antes de que se produzca. Esta lista de sucesos futuros se mantiene ordenada en orden creciente de timestamp.

En cada paso del algoritmo se elimina de la lista de sucesos el primer mensaje (el de timestamp mínimo), se avanza el reloj hasta alcanzar el valor del timestamp del mensaje y se simula su entrega en el sistema físico, donde es inmediatamente consumido con los efectos descritos anteriormente (cambio de estado del *PF* receptor, posible planificación de nuevos sucesos y posible cancelación de sucesos planificados con anterioridad). Este proceso se muestra en la Figura 2. La forma de avanzar el reloj es precisamente lo que marca la diferencia fundamental entre la simulación dirigida por tiempo y la simulación dirigida por sucesos. En la simulación por sucesos el reloj avanza, tras simular un suceso, hasta alcanzar el valor del timestamp del siguiente suceso de la lista, en saltos que *no* corresponden necesariamente a una unidad de tiempo.

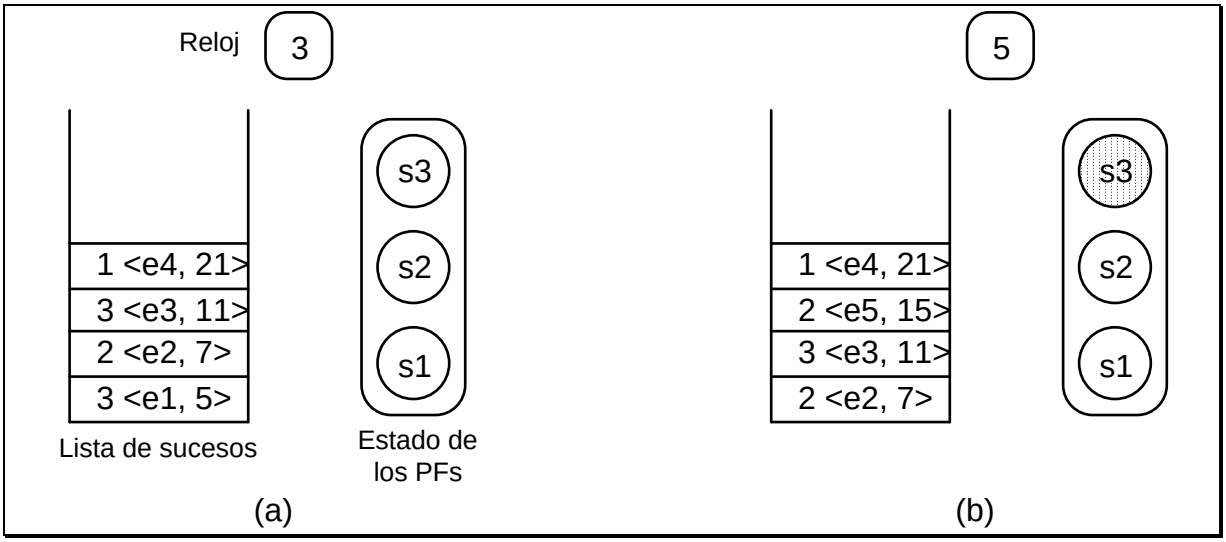

Figura 2: (a) Estado del simulador dirigido por sucesos antes de simular el suceso *e1*, planificado para el *PF* 3 y el instante 5. (b) Estado tras simular *e1*: el reloj ha avanzado hasta alcanzar el instante 5, el estado del *PF* 3 se ha modificado, y se ha planificado el suceso *e5* para el *PF* 2 y el instante 15. Nótese cómo el suceso *e5* se ha insertado en la lista de sucesos en la posición que le corresponde por su timestamp.

Uno de los aspectos que más se ha estudiado para el método de simulación dirigida por sucesos ha sido el relacionado con la gestión de la lista de sucesos futuros. Esta estructura de datos es el núcleo de la simulación, y una mala implementación de la lista puede resultar en programas con una complejidad inaceptable. Para evitarlo se han desarrollado múltiples algoritmos de gestión de listas que hacen que los procesos de inserción y eliminación de sucesos se realicen de forma muy optimizada.

3 Necesidad de la simulación paralela

Con la disponibilidad cada vez mayor de computadores paralelos, la necesidad de adaptar los lentos y costosos programas secuenciales de simulación a las nuevas arquitecturas se ha hecho imperiosa. En la literatura se pueden encontrar un buen número de alternativas para aprovechar el parelelismo ofrecido por estas arquitecturas, algunas de las cuales recogemos aquí.

Una forma inmediata de aprovechar el paralelismo ofrecido por un sistema con múltiples procesadores es lanzar, en paralelo, tantas simulaciones como procesadores tengamos [16]. Esta decisión no es tan absurda como puede parecer, porque normalmente el estudio de un sistema necesita de múltiples ejecuciones de un simulador, para poder así analizar el comportamiento del sistema con diferentes conjuntos de valores de los parámetros de entrada. Sin embargo, en este proceso de análisis, los resultados de una simulación suelen servir precisamente para afinar más los parámetros de la siguiente, exigiéndose de alguna forma una secuencialidad en las simulaciones. Haciendo muchas a la vez, es muy probable que algunas

(o casi todas) no sean válidas. Esta alternativa tiene el problema adicional de que cada elemento de proceso debe ser lo suficientemente capaz (tener la suficiente memoria) como para contener el simulador entero. En [40] se puede encontrar una propuesta de simulación paralela con este esquema, específicamente desarrollada para su implementación en arquitecturas SIMD como la Connection Machine CM-2 [37].

Parece claro, después de lo dicho, que lo que conviene es utilizar varios procesadores cooperando para acelerar la ejecución de *una* simulación, no para hacer más simulaciones a la vez. Surge entonces la idea de paralelizar los algoritmos de simulación utilizados habitualmente. Konas y Yew describen un método de simulación paralela esencialmente síncrono, aunque con ciertas características propias de la simulación por sucesos, especialmente diseñado para su implementación en multiprocesadores con memoria compartida [22]. Se trata de mantener una lista central de sucesos, formada por una colección de sub-listas, cada una de las cuales mantiene todos los sucesos planificados para un instante de tiempo dado. El simulador avanza cada vez en una unidad de tiempo, y entonces se simulan, en paralelo, todos los sucesos planificados para ese instante. Si cada una de esas sublistas tiene un tamaño significativo, es evidente que se consigue un buen nivel de paralelismo; en caso contrario, el algoritmo exhibirá un comportamiento muy similar a su equivalente secuencial.

También se puede intentar paralelizar el algoritmo de simulación dirigido por sucesos. Una propuesta consiste en usar una distribución de tareas entre diferentes unidades funcionales o elementos de proceso especializados [33]. Como ejemplos de tareas se pueden citar: generación de variables aleatorias, gestión de la lista de sucesos, recolección de estadísticas, entrada/salida, manipulación de ficheros, generación de gráficos, supervisión, etc. El grado de mejora en velocidad que se puede conseguir con esta técnica es bastante limitado, porque el número de unidades funcionales que se pueden identificar y poner a trabajar en paralelo es muy reducido, lo que no es apropiado para su implementación en sistemas masivamente paralelos. También se ha propuesto paralelizar la ejecución de cada suceso, dividiéndolo en sub-sucesos que se pueden ejecutar en paralelo. Esta alternativa sólo es útil cuando la simulación de las acciones asociadas a la ocurrencia de un suceso se realiza mediante un algoritmo altamente paralelo, lo que no se puede decir que ocurra habitualmente.

En general, paralelizar la simulación dirigida por sucesos es difícil porque el algoritmo es secuencial por naturaleza. En cada ciclo del simulador se toma un suceso planificado para el instante de tiempo indicado por el reloj y se simula su ocurrencia en el sistema, lo cual puede llevar a la planificación de sucesos para el futuro. Existen entre los sucesos fuertes dependencias causa-efecto que impiden la simulación de varios en paralelo. A modo de ejemplo, supongamos que los dos primeros mensajes de la lista de sucesos futuros son $m_1 = \langle e_1, t_1 \rangle$ y $m_2 = \langle e_2, t_2 \rangle$, donde los m_i son mensajes, los e_i son sucesos y los t_i son

timestamps, con $t_1 < t_2$. El algoritmo secuencial elimina de la lista en primer lugar el mensaje m_1. La simulación de e_1, aparte de tener otros efectos en las variables de estado, genera en el sistema un nuevo mensaje $m_3 = <e_3, t_3>$, donde $t_3 < t_2$. El siguiente mensaje que se selecciona es m_3, y al simular e_3 resulta que se cancela el suceso e_2, por lo que se elimina de la lista y nunca se simulará. Si en el ejemplo anterior se intentase simular en paralelo los sucesos e_1 y e_2, resultaría que, tras ellos, se intentaría simular e_3, que intenta evitar la simulación de un suceso, e_2, ya simulado. Obviamente, esta situación no puede permitirse.

Con lo visto hasta ahora, se deduce que el objetivo de aprovechar de forma óptima sistemas paralelos para la ejecución de programas de simulación sólo se puede conseguir si se parte de aproximaciones diferentes a las seguidas con los simuladores secuenciales. Deben desarrollarse técnicas nuevas, especialmente diseñadas para acelerar la ejecución de simulaciones mediante el uso masivo del paralelismo. Además, debe procurarse que el *speedup* conseguido al utilizar varios procesadores (aumento de velocidad en relación al número de procesadores utilizado, tomando como base la velocidad de ejecución en un único procesador) sea lineal con el número de procesadores, de tal forma que las simulaciones puedan acelerarse añadiendo más potencia (procesadores) a las máquinas que ejecuten los programas. El objetivo de conseguir un speedup lineal está dentro de lo que podríamos considerar como utópico, pero nos servirá como punto de referencia para evaluar las mejoras obtenidas al probar diferentes simuladores paralelos.

En esta línea se han propuesto varias técnicas de simulación parelela, entre las que destacan las de Chandy y Misra [6], similares a las propuestas por Bryant [5], conocidas como *conservadoras*, y la de Jefferson, conocida como *optimista* o también *Time Warp* [20]. Estas técnicas tienen una serie de puntos en común, ya que se trata de técnicas de simulación paralela dirigida por sucesos, y se diferencian principalmente en la forma de sincronización entre los diferentes procesos que se ejecutan en paralelo. En las siguientes secciones se describen cada una de estas técnicas, así como diferentes variaciones o evoluciones de las mismas, fruto del trabajo de otros investigadores.

4 Simulación paralela dirigida por sucesos

Ya se ha dicho que una de las mayores dificultades que plantea la paralelización de la simulación dirigida por sucesos es garantizar que se respetan las dependencias entre los sucesos en la simulación. No puede permitirse que la simulación de un suceso afecte a la de otros sucesos simulados con anterioridad, lo que sería considerado una incorrección en la simulación. En este sentido, se puede demostrar que el algoritmo de simulación secuencial es correcto: los sucesos se simulan en el orden correcto, ya que en cada paso del algoritmo se selecciona el suceso de menor timestamp y esto garantiza que las dependencias entre sucesos

se respetan [29]. Afortunadamente, no es ésta la única forma de garantizar la corrección de una simulación.

4.1 Dependencias entre sucesos

En esta sección vamos a definir más formalmente qué clase de dependencias entre sucesos hay que respetar en cualquier simulación dirigida por sucesos, sea ésta secuencial o paralela.

Definición 1: Se dice que el suceso e_i *afecta* al suceso e_j si se verifica al menos una de estas condiciones:

- La simulación de e_i hace que se planifique e_j.
- La simulación de e_i hace que se cancele e_j, planificado con anterioridad.
- La simulación de e_j exige leer o escribir información de estado que fue creada o alterada durante la simulación de e_i.

En cualquier caso, se asume que el timestamp de e_i es estrictamente menor que el de e_j, puesto que se supone que un suceso no puede tener influencia sobre sucesos que ya han ocurrido.

Definición 2: Se dice que un suceso *b depende causalmente* de un suceso *a* si existe una secuencia de sucesos $a=e_0, e_1, \dots, e_n=b$ tal que para cada par (e_i, e_{i+1}) ocurre que e_i afecta a e_{i+1}.

Definición 3: Para cualquier par de sucesos, *a* y *b*, puede darse el caso que ni *a* dependa causalmente de *b* ni al revés; son entonces *sucesos causalmente independientes.*

La relación de dependencia causal define un orden parcial en los sucesos de una simulación. Nótese que dos sucesos con el *mismo* timestamp son siempre causalmente independientes.

En un simulador secuencial los sucesos se simulan en orden no decreciente de timestamp. El orden de simulación de los sucesos no tiene por qué ser estrictamente creciente, ya puede haber sucesos planificados para un mismo instante de tiempo que pueden simularse en cualquier orden, o incluso a la vez. Ésto puede indicarnos que es posible realizar ciertas acciones en paralelo en un simulador. Sin embargo, el paralelismo extraído es, en general, escaso. Sin embargo, es posible pensar en extraer más paralelismo si se simulan a la vez sucesos que *no tienen el mismo timestamp*. Para ello relajamos la necesidad de ejecutar los sucesos en orden temporal, limitándonos al orden causal. Teniendo en cuenta las definiciones anteriores, si se realiza una simulación paralela en la que todos los pares de sucesos causalmente dependientes se simulan en el orden causal correcto, queda garantizado que:

(a) Se simulan exactamente los mismos sucesos en el simulador paralelo que en uno secuencial.

(b) Cuando se simula un determinado suceso, la parte del estado del sistema que afecta a la simulación de ese suceso es exactamente la misma en el simulador paralelo y en el secuencial.

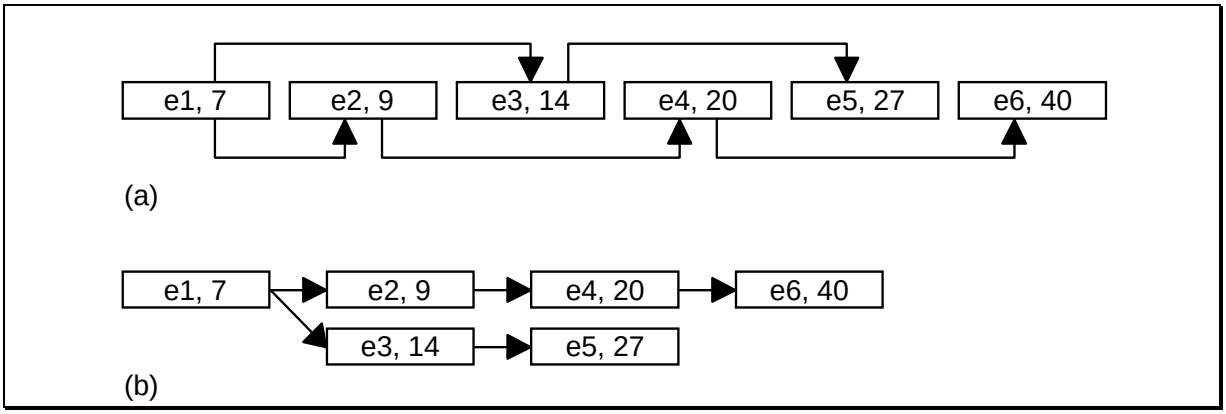

Figura 3: (a) Secuencia de sucesos planificados para el futuro, ordenados temporalmente. Las flechas indican dependencias causales entre sucesos. (b) Ordenando los sucesos de acuerdo a sus dependencias causales, *e2* se puede simular en paralelo con *e3*, y *e4* en paralelo con *e5*.

Por lo tanto, una simulación en la cual los sucesos se simulan en cualquier orden consistente con el orden parcial definido por la relación de dependencia causal es indistinguible de una simulación secuencial. Es evidente que la ordenación temporal de los sucesos planificados para el futuro que realiza un simulador secuencial es consistente con la ordenación causal, y que lo contrario no es siempre cierto. Para que la simulación sea correcta basta con garantizar una ordenación causal en la simulación de los sucesos, coincida ésta o no con la ordenación temporal. Los métodos de simulación paralela más importantes tratan precisamente de permitir la simulación en paralelo de sucesos causalmente independientes con timestamps potencialmente distintos.

La Figura 3 refleja, para un caso particular, las restricciones impuestas por las dependencias causales sobre una posible simulación paralela de un conjunto de sucesos.

4.2 Paralelización de la simulación por sucesos

A continuación vamos a dar una serie de características comunes a las dos familias más importantes de algoritmos de simulación paralela dirigida por sucesos, que reciben los nombres de conservadores y optimistas. Se considera, tal como se describió al principio de este artículo, que el modelo de sistema físico a simular está constituido por un conjunto de procesos físicos, que interactúan sólo en instantes discretos, mediante el envío de mensajes. Recuérdese que un mensaje contiene dos campos de información: el suceso que representa y el timestamp, que es el instante determinado de tiempo en el que debe acontecer ese suceso.

El sistema global se simula mediante una colección de *procesos lógicos* (*PL*s), uno por cada *PF*, que también se comunican enviándo y recibiendo mensajes de la forma <suceso,

timestamp>. No existe información compartida entre dos o más *PL*s, por lo que no se permite otro medio de sincronización e intercambio de información que no sea el ofrecido por el mecanismo de paso de mensajes. Cada uno de los *PL*s tiene su propio *reloj local*, que indica hasta qué punto ha progresado el *PF* correspondiente, en la simulación.

La interacción entre dos *PF*s, modelada planificando un suceso para un cierto instante *t*, se simula mediante el envío de un mensaje con timestamp *t* entre los *PL*s correspondientes. El timestamp de cualquier mensaje enviado por un *PL* debe ser estrictamente mayor que el valor del reloj local de ese *PL*, como requisito para mantener las relaciones causales en el sistema.

Como forma de tener almacenados los sucesos que están planificados para que ocurran en el futuro, cada *PL* dispone de una o varias *colas de entrada*. Si tiene una, todos los mensajes recibidos de otros *PL*s se almacenan conjuntamente. Alternativamente, se puede mantener una cola de entrada reservada para cada posible emisor, por lo que los mensajes que vienen de diferentes *PL*s no se mezclan. Los mensajes guardados en las colas de entrada se encuentran a la espera de que llegue el momento en el que deben ser consumidos. Recuérdese que consumir un mensaje supone avanzar el reloj hasta el valor del timestamp de ese mensaje y simular la ocurrencia del suceso asociado al mensaje, y que ésto puede suponer modificaciones en el estado del *PF* (por tanto, del *PL*) y el envío de nuevos mensajes.

Es muy importante recalcar que este conjunto de *PL*s no comparten ningún tipo de información global. En concreto, no existe un reloj centralizado que indique hasta qué punto ha progresado la simulación de un modo global, sino un conjunto de relojes, uno por *PL*, que avanzan por separado y que en un momento dado señalarán instantes diferentes del tiempo simulado. Tampoco existe una lista central de sucesos futuros, sino un conjunto de colas de mensajes: los sucesos futuros se encuentran distribuidos entre las colas de entrada de los *PL*s. La Figura 4 representa cómo un sistema físico compuesto por 3 *PF*s se simula por medio de tres *PL*s. En este caso, cada *PL* dispone de una cola de entrada por cada posible emisor de mensajes, por lo que mensajes enviados por diferentes emisores no se mezclan.

Conviene aquí señalar que, para que se pueda implementar un simulador con las características descritas, se necesita que los *PL*s dispongan de un método de intercambio de información. Éste puede ser un sistema de paso de mensajes, pero también sería posible un intercambio de información mediante memoria común, en el que se diseñen funciones que emulen el sistema de paso de mensajes. La topología de interconexión de los procesadores sobre los que se van a ejecutar los PLs debe ser capaz de ofrecer la conectividad requerida por el modelo. De aquí en adelante supondremos que se dispone de un sistema de paso de mensajes fiable (los mensajes no se pierden ni se corrompen) y con entrega ordenada (los mensajes llegan a su destino en el mismo orden en el que son enviados por el emisor).

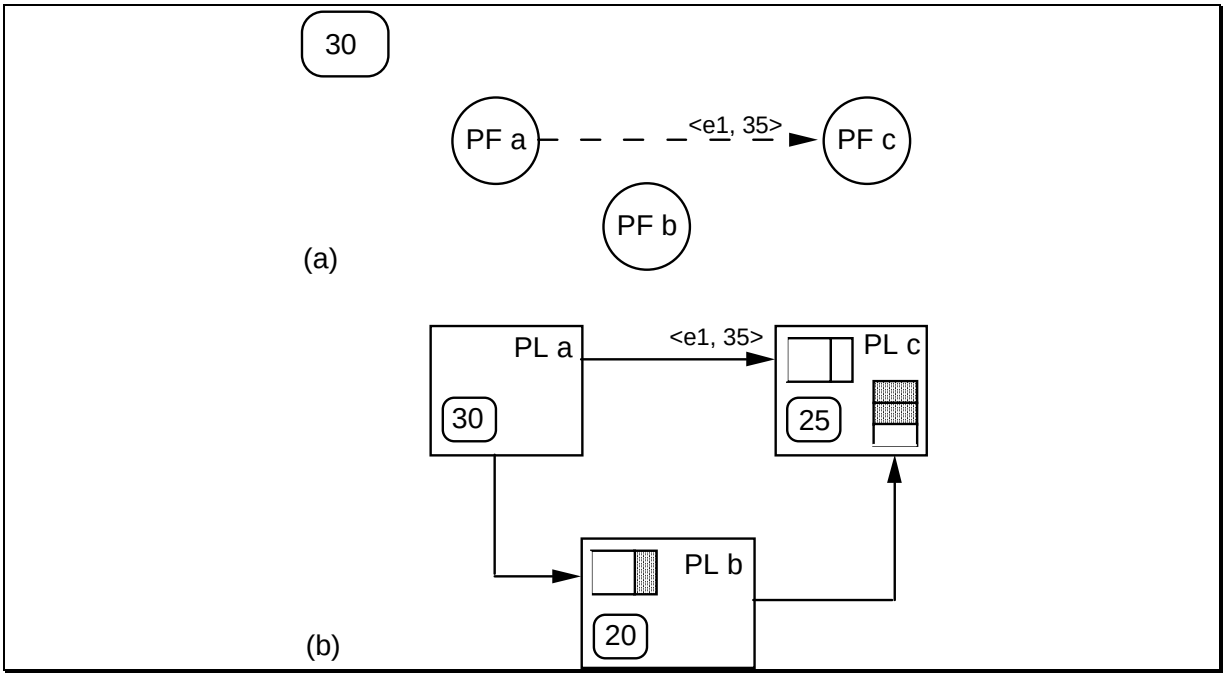

Figura 4: (a) En el sistema físico, PF_a planifica, cuando el reloj marca 30, el suceso *e1*, que ocurrirá a las 35 en PF_c. (b) Simulación de la situación anterior: el PL_a, cuando su reloj local marca las 30, envía el mensaje <*e1*, 35> al PL_C, que lo almacena en una cola de entrada reservada para los mensajes que vienen del PL_a, y allí queda a la espera de que sea consumido.

Cuando se usa un simulador secuencial y existe una única lista de sucesos, es sencillo determinar cuál es la acción a realizar en cada paso de la simulación: consumir el primer mensaje de la lista, es decir, el de timestamp mínimo. Sin embargo, cuando la lista de sucesos no es única, la decisión de "qué hacer ahora" no es tan fácil. Por supuesto, si se selecciona el mensaje con timestamp mínimo de entre todas las colas de entrada del conjunto de *PL*s, y se simula ese mensaje en su *PL* mientras todos los demás *PL*s se quedan inactivos, las relaciones temporales (y, por tanto, las relaciones causales) se mantienen correctamente, pero no se consigue paralelismo alguno. Afortunadamente, el criterio de seleccionar y ejecutar únicamente el suceso de menor timestamp es innecesariamente restrictivo: para que la simulación sea correcta es suficiente respetar la relación de dependencia causal en la ejecución de sucesos. Misra y Wagner [29, 42] demuestran que:

Si cada PL *consume en orden no decreciente de timestamp los mensajes que recibe, entonces la ejecución de la simulación es correcta.*

Recordemos que el término "correcta" aplicado a una simulación significa que garantiza la no existencia errores de causalidad en la simulación de los sucesos.

Es posible, cumpliendo esta restricción, que un conjunto de *PL*s, simulen varios sucesos en paralelo. Los diferentes métodos de simulación paralela dirigida por sucesos difieren en la estrategia elegida para garantizar que se cumple el requisito anterior. Los *conservadores* garantizan que esta condición se verifica siempre. Los *optimistas* permiten que a veces no se

cumpla, pero detectan tal eventualidad y, tras un proceso de *vuelta atrás*, retoman una situación en la cual todos los mensajes se han consumido en el orden correcto.

5 Métodos conservadores

Los trabajos de Bryant [5] y Chandy y Misra [6] son pioneros en el campo de la simulación paralela dirigida por sucesos. En ellos se describe un método de simulación paralela denominado *conservador*, en el cual se respetan las relaciones de causalidad entre sucesos forzando a los *PL*s a no consumir un mensaje hasta que no exista la *certeza* de que es imposible que llegue otro mensaje con timestamp menor. Para conseguirlo, se imponen ciertas restricciones a los *PL*s:

- Cada PL_j mantiene una cola de entrada para cada posible PL_i que puede enviarle mensajes. Por lo tanto, la topología de la red de interconexión entre los *PL*s debe estar definida estáticamente.
- Los *PL*s deben enviar los mensajes por cada uno de sus canales de salida en orden no decreciente de timestamp.

La primera restricción implica que todos los *PL*s que forman parte de la simulación deben existir desde el comienzo de la misma, y que todos los canales de comunicación entre *PL*s y sus colas de entrada correspondientes también deben estar definidos desde el principio. No se pueden crear *PL*s de forma dinámica, y un *PL* no se puede comunicar con otro si esa circunstancia no estaba prevista.

La segunda restricción conlleva que un PL_i no puede enviar un mensaje *m* por el canal de salida que le comunica con PL_j hasta estar *completamente seguro* de que no va a enviar por ese canal ningún otro mensaje con timestamp menor que el de *m*. Una consecuencia inmediata es que no se pueden enviar mensajes de cancelación, ya que requerirían un timestamp menor que el del mensaje a cancelar. Esta restricción es muy importante para garantizar que cada *PL* mantiene sus colas de entrada perfectamente ordenadas. Se está asumiendo, como ya se ha mencionado, que el mecanismo que los *PL*s utilizan para el intercambio de mensajes está libre de fallos y asegura que el orden de recepción de mensajes es el mismo que el de emisión.

Una vez que se tienen definidos todos los procesos lógicos de la simulación y los canales de comunicación entre ellos, se puede poner en funcionamiento el simulador. Todos y cada uno de los *PL*s ejecutan, de forma independiente, el mismo algoritmo. La Figura 5 esquematiza el núcleo del algoritmo conservador de simulación paralela dirigida por sucesos para un PL_j tal como aparece en [42].

Las variables que se utilizan son:

C_j: Reloj local de PL_j. Indica hasta qué punto ha avanzado PL_j en la simulación. Da una cota inferior de los timestamps de los mensajes que PL_j puede generar: sólo se pueden planificar sucesos futuros.

t_{ij}: Vector de relojes asociados a los canales de comunicaciones entre los diferentes PL_i y el PL_j. Cada elemento de este vector contiene el valor de timestamp del último mensaje que se recibió por el canal correspondiente.

m_j: Variable auxiliar destinada a contener el mensaje a consumir en cada paso del algoritmo. Es una estructura con dos campos: el contenido o *suceso* asociado al mensaje y el *timestamp* o instante para el cual está planificado el suceso.

H_j: Horizonte de aceptación de mensajes. Da una cota superior del valor de timestamp permitido a un mensaje recibido para que sea consumido. Dicho de otra forma, no serán admitidos para su consumo mensajes con timestamp mayor que H_j, ya que se considera que suponen un riesgo. Se calcula como $\mathbf{min}_i\{t_{ij}\}$, es decir, como el valor mínimo de los relojes de los canales de comunicaciones que tienen como extremo receptor al PL_j.

```
Cj := 0;
para cada i hacer tij:=0;
mientras no terminado hacer
    mientras no haya mensajes en las colas de entrada hacer
        esperar(llegada de mensaje);
    fin mientras
    mj:=mensaje con timestamp minimo de todos los encolados;
    Hj:=min_i{tij};
    mientras mj.timestamp > Hj hacer
        esperar(llegada de mensaje)
        mj:=mensaje con timestamp mínimo;
        Hj:=min_i{tij};
    fin mientras
    eliminar(mj);
    Cj:=max(Cj, mj.timestamp);
    simular(mj.suceso);
    /* Puede cambiar Cj y/o enviar mensajes a otros PLs */
fin mientras
```

Figura 5: Algoritmo conservador de simulación paralela dirigida por sucesos, para un PL_j.

Aunque no aparezca de forma explícita en el algoritmo, se asume que cada *PL* se encarga también de gestionar sus colas de entrada. De forma asíncrona, el *PL* va recibiendo mensajes por sus canales de entrada. Cuando se recibe un mensaje por un canal, se actualiza el valor del reloj de ese canal, y se almacena el mensaje en la cola de entrada correspondiente, en función de qué *PL* sea el emisor.

Cada *PL* inicializa al comenzar la simulación el reloj local y los relojes asociados a cada uno de los canales de entrada. Tras ello, se queda bloqueado a la espera de tener algún mensaje que consumir: no puede avanzar si no tiene nada que hacer. Una vez que se dispone de algún mensaje, se selecciona el de menor timestamp y se calcula el horizonte de aceptación de mensajes. Si el mensaje seleccionado está dentro del horizonte de aceptación, entonces se consume. Si no es así, el *PL* se bloquea a la espera de nuevos mensajes.

Esta técnica de simulación paralela se denomina conservadora porque no acepta riesgos en cuanto a la ejecución de sucesos. Supongamos que $H_j = t_{ij}$, es decir, que el reloj del canal de comunicación que une PL_i con PL_j es el que impone el valor del horizonte de aceptación. Si t_{ij} es menor que el valor del mínimo timestamp de todos los mensajes almacenados en las colas de entrada (llamemos a este timestamp t_{min} y al mensaje correspondiente m_{min}), PL_j se bloquea. Se dice entonces que el canal *ij* es un *canal bloqueante*. Esta situación se representa en la Figura 6. El motivo del bloqueo es sencillo: puesto que t_{ij} es menor que t_{min}, cabe la posibilidad de que el PL_i envíe por ese canal un mensaje con timestamp mayor o igual que t_{ij}, y menor que t_{min}. Si se consumiese el mensaje m_{min}, se estaría corriendo el riesgo de incumplir las condiciones que garantizan la corrección causal. En estas condiciones, la única forma de que PL_j progrese es recibir un mensaje por el canal bloqueante, mensaje que, o será consumido, o hará aumentar el horizonte de aceptación y permitirá consumir otros.

Nótese que es la llegada de mensajes, con su timestamp correspondiente, el motor que hace que la simulación avance. Permite, por una parte, que el *PL* tenga trabajo que realizar, pero también supone incrementos en los relojes de los canales por los que se reciben, lo que conlleva la posibilidad de aumentar el horizonte de aceptación y, en consecuencia, permite que algunos de los mensajes encolados sean candidatos a ser consumidos. Un canal por el que se envíen mensajes con poca frecuencia se convierte en un cuello de botella, ya que será casi de continuo un canal bloqueante y forzará al *PL* a pasar la mayor parte del tiempo bloqueado.

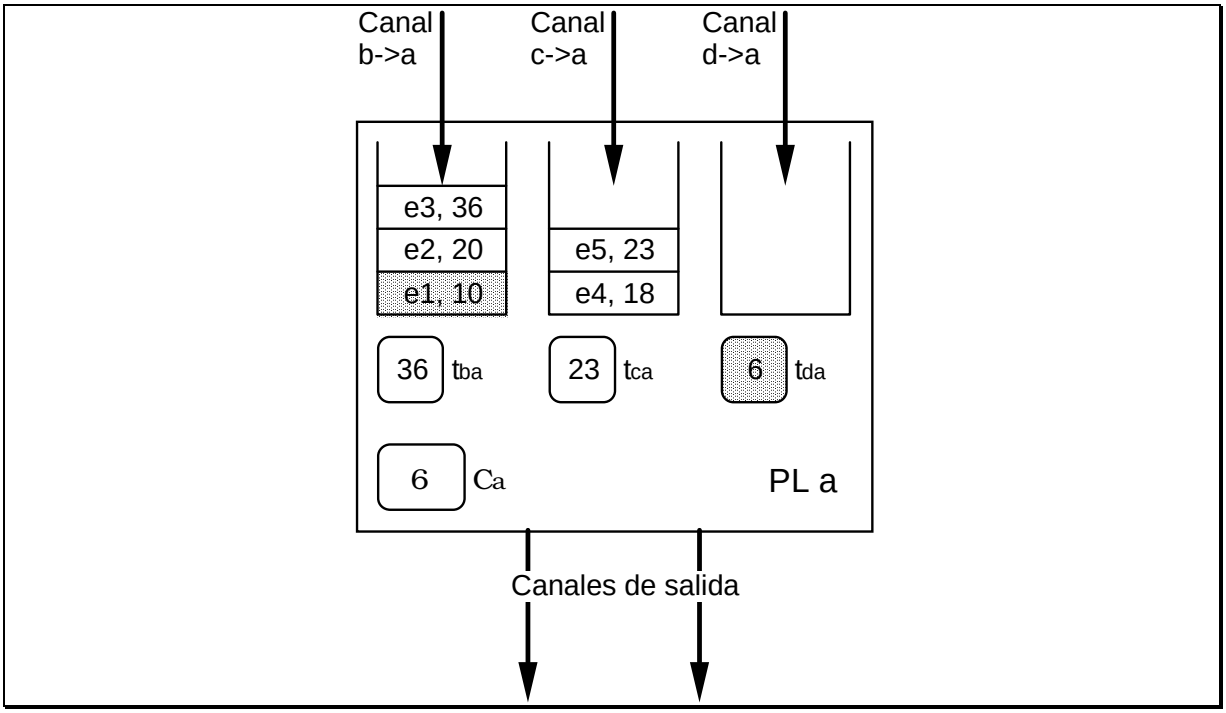

Figura 6: El PL_a puede recibir mensajes de PL_b, PL_c y PL_d, por lo que tiene una cola de entrada asociada a cada uno de ellos. El horizonte de aceptación es 6, el timestamp del último mensaje recibido desde PL_d. El canal bloqueante es, por tanto, el *da*. El mensaje candidato a ser consumido es <*e1*, 10>, pero debe esperar, porque cabe la posibilidad de que PL_d envie un mensaje con timestamp mayor o igual que 6, pero menor que 10.

5.1 El problema del interbloqueo

Uno de los problemas que presenta el algoritmo anterior es que puede llevar a un estado de *interbloqueo* entre los *PL*s. Con el objetivo de caracterizar esta situación, Wagner divide en cinco categorías los posibles estados de los *PL*s [42]:

- Un *PL* está *parado* si su estado satisface las condiciones de terminación de la simulación. Por lo tanto, no generará más mensajes.
- Un *PL* que no está parado está *inactivo* si no tiene ningún mensaje almacenado en sus colas de entrada.
- Un *PL* que no está ni parado ni inactivo está *bloqueado* si su horizonte de aceptación es menor que el valor del mínimo timestamp de todos los mensajes almacenados en sus colas de entrada.
- Un *PL* está *interbloqueado* si (a) está inactivo y todos sus posibles emisores de mensajes están interbloqueados o parados, o (b) está bloqueado y todos los *PL*s emisores correspondientes a los canales bloqueantes están interbloqueados o parados.
- Un *PL* está *activo* si no está ni parado, ni inactivo, ni bloqueado ni interbloqueado.

Se dice que una simulación paralela conservadora está *interbloqueada* si al menos un *PL* está interbloqueado y el resto están parados o interbloqueados.

Nos encontramos con dos fuentes básicas de interbloqueo. El caso más sencillo de ver es aquel en el que un *PL* no puede avanzar porque se encuentra a la espera de mensajes procedentes de otro *PL* que ya ha terminado su simulación (está parado) y que, por tanto, no generará nuevos mensajes. Es algo más complejo el segundo caso: la aparición de interbloqueos debidos la formación de *ciclos* de *PL*s en los cuales cada uno de ellos está bloqueado esperando la recepción de mensajes desde otros *PL*s pertenecientes al ciclo. La Figura 7 muestra un ciclo de *PL*s interbloqueados.

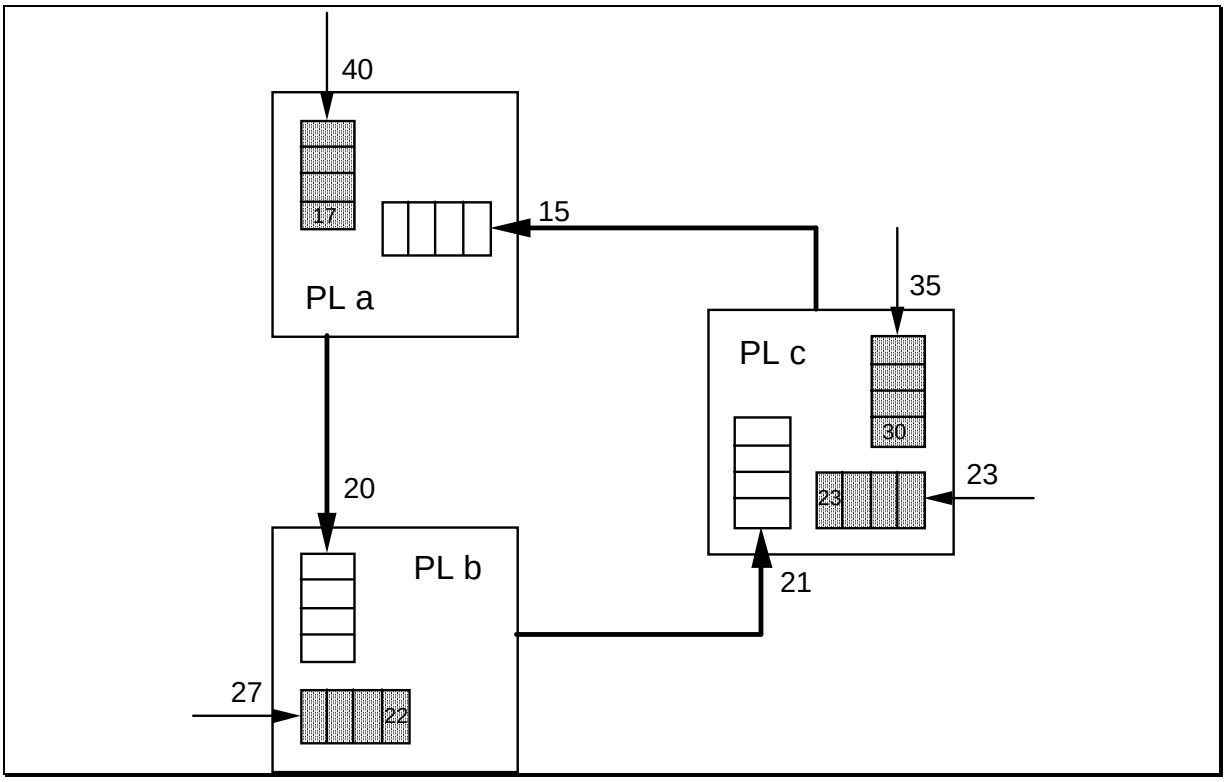

Figura 7: Una situación de interbloqueo. Los 3 *PL*s de la figura están interbloqueados. Si nos fijamos en PL_C, los relojes de sus tres canales son 35, 23 y 21, luego H_C = 21. El timestamp mínimo de entre los mensajes almacenados en las colas de entrada es 23, luego no puede consumirse ningún mensaje y el canal *bc* es un canal bloqueante. Situación análoga ocurre en PL_a y PL_b. Nótese el ciclo de canales bloqueantes, y la presencia de colas vacías en todos los *PL*s del ciclo.

Evidentemente, la posibilidad de interbloqueo es una característica inaceptable del simulador, ya que impide que la simulación avance. Se necesitan mecanismos para tratar los interbloqueos que garanticen el progreso de la simulación y que, a ser posible, introduzcan el mínimo esfuerzo de proceso adicional. La primera posibilidad de falta de progreso de un *PL* es fácil de evitar si cada *PL* envía por todos sus canales de salida, cuando termina su trabajo, un mensaje especial de fin de simulación con timestamp infinito. Así, ese canal nunca más será un canal bloqueante. Sin embargo, la resolución de interbloqueos surgidos por ciclos de *PL*s es más compleja. Se han propuesto dos alternativas: la primera es evitar por todos los

medios que aparezcan interbloqueos (*deadlock avoidance)*; la segunda es permitir que ocurran, detectarlos y volver a poner en marcha el simulador tras deshacer el interbloqueo (*deadlock detection and recovery*).

5.1.1 Técnicas para evitar el interbloqueo

El artículo original de Chandy y Misra propone un sistema que evita la aparición de interbloqueos [6]. Para ello los *PL*s se envían, además de mensajes asociados a sucesos, mensajes especiales de control denominados *mensajes nulos*. Estos mensajes nulos no representan interacciones entre los *PF*s simulados por los *PL*s (no tienen ningún efecto en lo que se refiere a la modificación de variables de estado, ni en la planificación de sucesos), pero permiten a los *PL*s receptores, al llevar asociado un timestamp, el avance de los relojes de los canales por los que se reciben y, por tanto, el avance de los horizontes de aceptación.

A modo de ejemplo del funcionamiento de esta técnica, veamos cómo se puede deshacer la situación de interbloqueo reflejada en la Figura 7. En PL_c el valor de t_{bc} es 21. Esto significa que por ese canal se recibió un mensaje con timestamp 21, que ya ha sido consumido; se asume, por tanto, que el reloj de PL_c es, al menos, 21. Para ese instante de tiempo no hay ningún suceso que simular pero, aunque llegase uno, dadas las características del proceso físico simulado por PL_c, se sabe que no se podría planificar ningún suceso anterior al instante 22. Por lo tanto, PL_c puede enviar a PL_a el mensaje <NULL, 22>. La recepción de ese mensaje hace que en PL_a el reloj de canal t_{ca} pase a valer 22, luego el canal de comunicaciones que una PL_c con PL_a ya no es un canal bloqueante y PL_a puede progresar, avanzando su reloj hasta el valor 17 y consumiendo el mensaje correspondiente.

Para garantizar progreso en la simulación, es suficiente que cada *PL* envíe a sus vecinos mensajes nulos cuando detecta que va a bloquearse [29]. Esta circunstancia se detecta de modo trivial: si un *PL* no tiene ningún mensaje para consumir, o tiene uno pero su timestamp es menor que el valor del horizonte de aceptación, entonces es que el *PL* va a bloquearse. Para el PL receptor de un mensaje nulo, ese mensaje se interpreta como un compromiso del emisor, que le asegura que no va a mandar mensajes “válidos” con timestamp menor al del mensaje nulo. Así, aunque el emisor se bloquee, se permite que el receptor pueda seguir avanzando en la medida de lo posible. Evidentemente, la magnitud de este avance depende del timestamp que lleve el mensaje nulo.

Este método de evitar la existencia de interbloqueos presenta dos problemas serios. El primero de ellos es que, la mayoría de los experimentos realizados con implementaciones del mismo, muestran que el número de mensajes nulos que circulan por la red de *PL*s es exageradamente elevado, superando ampliamente en muchos casos al número de mensajes válidos. Se llega a situaciones en las que el simulador pasa la mayor parte del tiempo enviando y recibiendo mensajes nulos, sin realizar ningún trabajo útil. Por otra parte, en

determinados casos, el método de evitación de interbloqueos puede no funcionar. Supóngase una situación en la que, como respuesta al consumo de un mensaje <NULL, t>, un *PL* no puede hacer otra cosa que avanzar su reloj local y enviar otro mensaje <NULL, t>. Nos encontramos con que ese *PL* no tiene *ninguna capacidad de predicción* de lo que puede ocurrir en un futuro próximo. Si todos los los miembros de un ciclo de *PL*s interbloqueados adolecen de esta falta de visión de futuro, es fácil imaginarse cómo puede llegarse a una situación en la que los *PL*s del ciclo se intercambian continuamente mensajes nulos con el mismo timestamp, con lo que, aunque se consumen recursos del simulador, la simulación no progresa. En la siguiente sección se describirá el concepto de *lookahead*, que mide precisamente la capacidad de predicción de un *PL*, y se relacionará este concepto con la eficiencia del método de evitación de interbloqueos mediante mensajes nulos.

5.1.2 Técnicas de detección y recuperación de interbloqueos

Dados los problemas que presenta el método de los mensajes nulos para evitar los interbloqueos, Chandy y Misra proponen un método alternativo que consiste en dejar que el sistema se interbloquee y, tras *detectar* esta situación, ejecutar un proceso de *recuperación* que permita a la simulación seguir avanzando [7]. El simulador progresa de forma normal, sin ponerle ningún medio para evitar la aparición de interbloqueos; eventualmente, se producirá un interbloqueo, con lo que esta *fase paralela* llegará a su término. Un proceso especial, denominado *controlador* detecta el interbloqueo, bien sea de forma centralizada o de forma distribuida [8], y entonces lo resuelve. La forma más sencilla de resolver un interbloqueo es buscar en todas las colas de entrada de todos los *PL*s el mensaje con *menor* timestamp, y permitir que sea consumido.

Durante la fase de resolución del interbloqueo no se produce avance en la simulación, por lo que esta fase se puede considerar como una sobrecarga añadida al simulador. Esta sobrecarga es precisamente objeto de las críticas que se hacen al método de detección-recuperación de interbloqueos. Si se producen interbloqueos con frecuencia, y resolverlos es costoso en comparación con el avance que se hace en cada fase paralela, entonces el método es muy poco eficaz.

5.1.3 Evitación perezosa del interbloqueo

Wagner y Lazowska proponen, como alternativa a los dos métodos de tratamiento de interbloqueos considerados hasta ahora, un método adicional que aprovecha la capacidad de memoria compartida del multiprocesador en el que realizan sus experimentos: un sistema Sequent Symmetry con 20 procesadores que comparten 32 Mb de memoria [10]. Consiste en resolver los interbloqueos utilizando un micro-kernel que se encarga de la planificación de procesos en la máquina. Cuando se detecta que en un procesador todos los procesos están

bloqueados, se analiza el estado de sus relojes y colas de mensajes. Si se encuentra un PL_j bloqueado a la espera de un mensaje desde PL_i, y el reloj de PL_i tiene un valor superior a t_{ij} (el reloj del canal que comunica PL_i con PL_j), entonces PL_j puede hacer avanzar el valor de t_{ij} por lo menos hasta alcanzar el valor del reloj de PL_i, ya que tiene garantías de que no va a llegar ningún mensaje de timestamp menor. Con esto se consigue que PL_j avance su horizonte de aceptación y pueda desbloquearse [41, 42].

Este método no es directamente aplicable en arquitecturas de memoria distribuida, ya que el micro-kernel descrito tiene una visión global del espacio de memoria de todos los procesos que intervienen en la simulación. Dado que nuestro grupo de investigación trabaja habitualmente en arquitecturas paralelas de memoria distribuida, con comunicación mediante paso de mensajes, no podemos aplicar esta optimización en nuestros simuladores.

5.2 El concepto de lookahead

Cuando se trata de acelerar una simulación utilizando paralelismo, una característica muy importante del modelo a simular es su *lookahead*. Este concepto está relacionado con la capacidad que tiene un *PL* de precedir su comportamiento futuro [12].

En una simulación conservadora, cuando un proceso lógico PL_i planifica un suceso *e* para que ocurra en PL_j en el instante *t*, prepara el mensaje $m = <e, t>$. Ese mensaje *debe ser retenido* en PL_i hasta que se tenga la certeza absoluta de que no se va a enviar ningún otro mensaje a PL_j con un timestamp menor que *t*. Esto se expresó anteriormente diciendo que los *PL*s deben enviar los mensajes por cada uno de sus canales de salida en orden no decreciente de timestamp. Por lo tanto, pasará un tiempo desde que PL_i haya planificado un suceso hasta que pueda enviar el mensaje correspondiente con garantías de cumplir las restricciones causales. Esta cantidad de tiempo depende de la capacidad que tenga el proceso de *predecir* su futuro (*look ahead*). Cuanto mayor sea la capacidad de predicción de un *PL*, antes podrá enviar un mensaje. En general el valor del lookahead es una función compleja que varía con el tiempo y que es altamente dependiente de los detalles de la simulación que se esté realizando.

La importancia del lookahead se hace especialmente patente cuando se usan mensajes nulos para evitar los interbloqueos. Cuando un proceso lógico PL_j está a punto de bloquearse, envía a todos sus vecinos mensajes nulos, con un timestamp calculado como $t = C_j + L_j$ con $L_j>0$, es decir, el valor del reloj local más un cierto incremento L_j. Este incremento es precisamente el lookahead de PL_j. Como ya se ha dicho, un mensaje nulo puede interpretarse como una promesa de no enviar mensajes válidos con timestamp menor que *t*. El proceso lógico PL_j puede realizar esta promesa porque tiene un conocimiento de su comportamiento, y puede determinar qué acciones va a realizar en un futuro cercano, lo que implica que *sabe* que no va a planificar ningún suceso con timestamp menor que *t*. Obviamente, cuanto mayor sea el

valor de L_j mayor será el valor de t y, por lo tanto, mayor podrá ser el avance que un proceso receptor haga en el reloj del canal correspondiente (y, consecuentemente, en su horizonte de aceptación). En otras palabras, cuando la simulación avanza *artificialmente* por medio de mensajes nulos, el avance que se consigue es proporcional al lookahead. Si se consigue aumentar la capacidad de lookahead de un *PL*, el número total de mensajes nulos enviados también se reduce, y la simulación se acelera al tener menos sobrecarga.

Fujimoto destaca también que el lookahead no es importante sólo en la simulación paralela, sino que lo es en cualquier simulación dirigida por sucesos: se progresa porque, cuando se simula la ocurrencia de un suceso en un proceso físico, el simulador es capaz de predecir el comportamiento futuro de ese proceso físico, y planifica nuevos sucesos que ocurrirán con posterioridad.

5.3 Evaluaciones de los métodos conservadores

Algunos de los estudios más interesantes realizados para evaluar los algoritmos conservadores corresponden a Reed y sus colaboradores [32] y a Fujimoto [13]. Ambos estudios cubren tanto el caso de evitación del interbloqueo como el de detección y recuperación.

Los estudios de Reed fueron realizados con un multiprocesador Sequent Balance 21000 con 20 procesadores [36]. Esta máquina dispone de memoria compartida, característica de la que se hace uso para intentar optimizar los algoritmos. En concreto, el envío de mensajes se implementa mediante escrituras en variables compartidas, y el método de detección de interbloqueos utilizado no es distribuido (como proponen Chandy-Misra-Haas en [8]), sino centralizado. Como modelos a simular se tomaron varias redes de colas FCFS (*First Come First Serve*), y con cada una de ellas se realizaron un conjunto de experimientos variando por un lado parámetros del modelo tales como las probabilidades de encaminamiento, las tasas de llegada, los tiempos de servicio, etc., y por otro ciertas características del simulador como son las políticas de asignación de *PL*s a elementos de proceso, las técnicas de tratamiento de interbloqueos, el número de procesadores empleados, etc.

Los resultados obtenidos son un tanto desalentadores en cuanto a la aceleración que se puede obtener con una simulación paralela. De hecho, para algunos casos de simulación en los que los *PF*s simulados forman ciclos, se consiguen speedups inferiores a uno. Dicho de otra forma, paralelizar la simulación no sólo no supone una ganancia en velocidad de ejecución, sino una pérdida. Esto es debido a la enorme sobrecarga de trabajo que supone la generación y consumo de mensajes nulos, en el caso de que se emplee la técnica de evitación de interbloqueos, o, si no se utiliza esta técnica, el sobreesfuerzo necesario para la detección y recuperación de interbloqueos.

Cabe destacar que en los simuladores realizados no se ha intentado en ningún caso aprovechar, con el objetivo de acelerar la simulación, las características propias del modelo simulado. En concreto, no se hace esfuerzo alguno por explotar el lookahead que los modelos pueden ofrecer. Por este motivo, hay que relativizar los resultados que se presentan. Estudios posteriores demuestran que se pueden aprovechar características de los modelos para obtener simuladores con escasa sobrecarga [23], y que también se pueden obtener valores altos de lookahead en muchos modelos, en concreto en las redes de colas FCFS [30].

Un aspecto de especial interés en el trabajo de Reed es la importancia que se da a la asignación de *PL*s a los diferentes elementos de proceso. Dada la arquitectura disponible, resulta relativamente sencillo asignar procesos a procesadores de forma dinámica, aunque también pueden escogerse alternativas de asignación estática. En general la asignación dinámica obtiene mejores resultados, al distribuir mejor la carga de trabajo entre los diferentes procesadores.

El estudio de Fujimoto recogido en [13] es más amplio que el de Reed, y sí intenta aprovechar el lookahead correspondiente a los modelos simulados; sus resultados, por tanto, no son tan desalentadores. Las pruebas fueron realizadas sobre un multiprocesador BBN Butterfly con 17 elementos memoria-procesador, interconectados mediante una red Omega [9]. A pesar de que la memoria está distribuida entre los 17 elementos, el modelo de programación ofrecido hace que se vea toda ella como memoria común, aunque por supuesto los accesos a la memoria local resultan más rápidos que los accesos a memoria de otros elementos. Como sistema a simular, Fujimoto eligió redes cuadradas toroidales, en las que circulan una cantidad fija de *trabajos* que van pasando de nodo a nodo. Cada *nodo* dispone de un centro de servicio y de una cola FCFS asociada. Un centro de servicio de un nodo procesa un trabajo durante un cierto tiempo (*tiempo de servicio*) y, cuando termina con él, se lo envía a otro nodo. Si mientras un centro de servicio está ocupado se recibe un trabajo, se encola a la espera de que el centro de servicio quede libre. Cada nodo del modelo se implementa mediante un *PL*, y el envío de un trabajo de un nodo a otro se simula mediante el intercambio de un mensaje entre los *PL*s correspondientes.

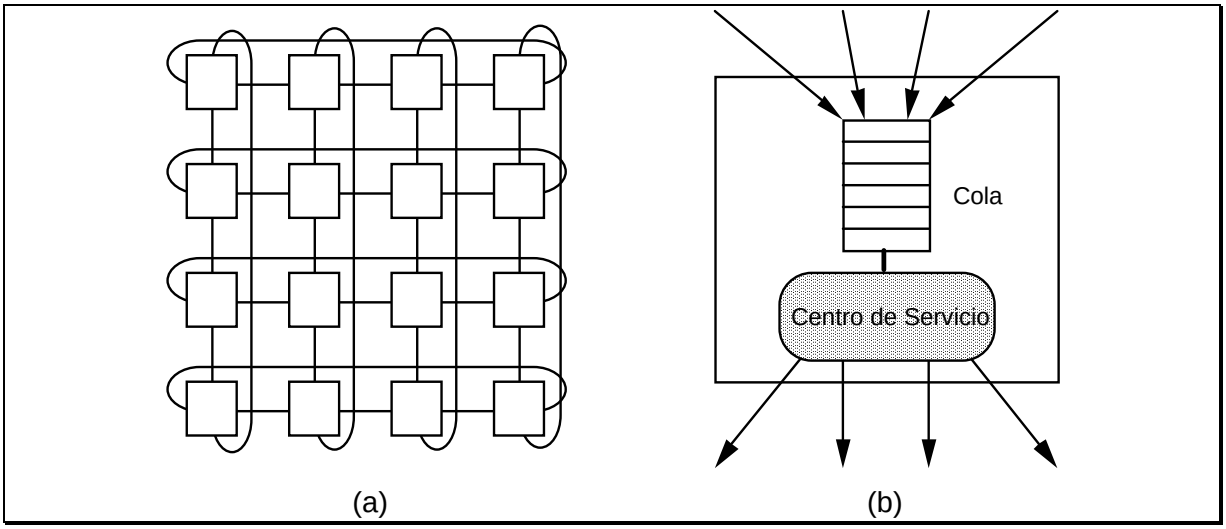

Figura 8: (a) Red toroidal de 4x4 nodos, como las simuladas en [13]. Los canales de comunicación que unen dos nodos son bidireccionales. (b) Detalle de cada nodo.

En este modelo se pueden variar parámetros como: distribución del tiempo de servicio, número total de trabajos en el toro, tamaño del toro, función de encaminamiento de los trabajos y distribución del cómputo en cada nodo. Este último parámetro representa una carga sintética que se añade al simulador, carga que intenta reflejar el esfuerzo de cómputo que supondría la simulación de cada uno de los trabajos en un simulador "real", no en uno construido únicamente para efectuar una evaluación de los métodos de simulación paralela.

Se eligen, a la hora de hacer los experimentos, este tipo de modelos porque suponen una prueba de estrés de los métodos de simulación paralela: aparece un número considerable de *LP*s, con un alto nivel de comunicación entre ellos; el mayor esfuerzo del simulador no es la simulación de cada trabajo (de hecho, se puede añadir al simulador una esfuerzo sintético, si así se desea), sino la sincronización entre los *LP*s para que la simulación avance. Si un método da buenos resultados con un modelo de esta naturaleza, premeditadamente "malo", se puede inferir que dará buenos resultados en general.

Uno de los parámetros del modelo que se considera más interesente es la función de distribución del tiempo de servicio de los centros de servicio, dada su estrecha relación con el lookahead. Un trabajo que entre en el centro de servicio de un nodo en el instante t saldrá del nodo cuando el reloj alcance el valor $t' = t + t_S$, donde t_S es un tiempo de servicio calculado (normalmente) de forma aleatoria. En el instante t, el *PL* puede no saber exactamente cuál será el valor exacto de t_S, pero sí puede conocer su valor mínimo t_{min}. Esta información puede ser útil para que el *PL* notifique a sus vecinos (por ejemplo, por medio de un mensaje nulo) que ningún trabajo va a salir de ese nodo hasta un instante de tiempo posterior a $t + t_{min}$. Por lo tanto, el *PL* está *prediciendo* lo que pasará en su futuro cercano, es decir, está explotando su lookahead. En este caso, el lookahead es igual al valor mínimo de la distribución del tiempo de servicio.

Fujimoto experimienta con diferentes funciones de distribución y, por lo tanto, con diferentes lookaheads. La Tabla 1 recoge las características de estas distribuciones. Se define además el concepto de *lookahead ratio* (LAR) como la media de la distribución dividida entre el lookahead. En todas las funciones de distribución empleadas se garantiza que el lookahead es mayor que cero, porque esta restricción es imprescindible si se quiere utilizar una política de evitación de interbloqueos mediante envío de mensajes nulos. Los valores del LAR varían entre 1.0 para el caso de la distribución determinista y 11.0 para el caso de la exponencial, con algunos valores intermedios como 5.5 para la distribución uniforme.

Distribución	**Expresión de cómputo**	**Lookahead**	**LAR**
Determinista	1.0	1.0	1.0
Polarizada 0.9-1.0	0.9 + 0.2 rand	0.9	1.11
Exponencial	0.1 - *ln*(rand)	0.1	11.0
Uniforme	0.1 + rand	0.1	5.5
Bimodal	0.95238 rand + (if rand<0.1 then 9.5238 else 0)	0.1	10.0

Tabla 1: Distribuciones de probabilidad para el tiempo de servicio utilizadas por Fujimoto. “rand” es una función que devuelve un número aleatorio uniformemente distribuido entre 0 y 1. “ln” es la función *logaritmo neperiano*.

A modo de ejemplo de los resultados que obtiene, simulando con 16 procesadores un toro de 8x8 nodos, con una carga inicial de 4 trabajos por nodo, se consigue un speedup de aproximadamente 11 cuando la distribución del tiempo de servicio es polarizada (LAR = 1.11), mientras que no llega a 5 cuando es exponencial (LAR = 11). Con la distribución uniforme (LAR = 5.5), el speedup es de aproximadamente 7.

Tras una gran colección de experimentos, Fujimoto detecta y razona los siguientes aspectos relacionados con la simulación conservadora:

1. La capacidad de predicción del futuro en un PL (su lookahead) juega un papel fundamental a la hora de conseguir simulaciones paralelas eficientes, ya que los procesos pasan menos tiempo esperando hasta tener la certeza de que un mensaje puede ser enviado con seguridad. No es tan importante el valor absoluto del lookahead como la relación que existe entre éste valor y los incrementos medios de timestamp entre mensajes (el LAR). A menor LAR, mayor eficiencia. Diferentes simulaciones con un valor semejante de LAR exhiben un comportamiento (speedup) muy similar.

2. Los valores de speedup obtenidos mejoran cuando se aumenta el valor de la distribución de cómputo (la carga sintética asociada al simulador). Esto quiere decir que, en el caso de una simulación real en la que el consumo de un mensaje

supusiese un elevado esfuerzo de proceso, un simulador conservador es capaz de distribuir de forma eficaz ese esfuerzo entre los diferentes procesadores. Dicho de otra forma, conforme aumenta la *granularidad* de la simulación, se consigue una mejora en las prestaciones, porque se equilibra la relación entre cómputo (consumo de mensajes) y sincronización (envío y recepción de mensajes, método de tratar el interbloqueo).

3. Si se usa para resolver los interbloqueos un método de detección y recuperación, la eficiencia de las simulaciones es pobre, a no ser que la carga del sistema (en los ejemplos, el número de trabajos) sea elevada. Se detecta un umbral de carga por debajo del cual la eficiencia es pobre y por encima del cual se consiguen muy buenos resultados (speedups prácticamente lineales con el número de procesadores). Este umbral varía en función de parámetros como el LAR. La explicación que da Fujimoto, confirmada por nuestros experimentos, es que un simulador conservador da buenos resultados cuando la carga de trabajo es elevada. En estas circunstancias, el número de mensajes que circulan por la red de *PL*s es alto y la probabilidad de que un *PL* se quede bloqueado, bien sea a la espera de tener algún mensaje para consumir, o a la espera de que se amplie su horizonte de aceptación, se reduce [26].

4. Basta con que existan unos pocos *PL*s con baja capacidad de lookahead para que la eficiencia de la simulación disminuya. Este efecto es especialmente notorio en el caso de que se emplee la técnica de detección y recuperación de interbloqueos. Los *PL*s con poco lookahead se comportan como verdaderos cuellos de botella, que producen la aparición continua de interbloqueos que hay que deshacer. Un simulador con evitación de interbloqueos es menos sensible a estas asimetrías en el lookahead, ya que el esfuerzo de evitación de interbloqueos se concentra entre los *PL*s con poco lookahead y sus vecinos.

5. Los experimentos de Reed sobre redes de colas con ciclos eran exageradamente pesimistas. Se pueden conseguir simulaciones eficientes si se explota el lookahead inherente al sistema simulado.

6. En general, el método de evitación de interbloqueos da mejores resultados que el de detección y recuperación. Recordemos no obstante que no siempre puede ser utilizado.

7. Los métodos conservadores pueden resultar apropiados para la simulación de sistemas con un nivel de paralelismo entre medio y alto. Sin embargo, que el modelo exhiba paralelismo no garantiza una simulación eficiente, ya que puede ocurrir que el riesgo de intebloqueo en el simulador sea alto y los medios puestos para evitar sus efectos añadan una sobrecarga excesiva.

Nuestro grupo de trabajo ha realizado experimentos similares a los de Fujimoto y Reed, implementando un simulador paralelo conservador con evitación de interbloqueos sobre un Supernode SN-100 de la firma PARSYS. Es éste un sistema paralelo con 34 elementos de proceso Transputer T-805, cada uno de ellos con una cantidad de memoria local que varía entre 2 y 8 Mb [18, 38]. Una característica fundamental de esta máquina es que se trata de una arquitectura paralela con memoria distribuida, circunstancia que supone una novedad con respecto a trabajos anteriores de otros autores, en su mayoría realizados sobre arquitecturas de memoria compartida.

A partir de un modelo similar al de Fujimoto (red toroidal de servidores FCFS) se ha evaluado el efecto de la carga (número de trabajos en el sistema), el lookahead, el número de procesadores utilizados por el simulador y la distribución de cómputo. Tras realizar una buena colección de experimentos para evaluar nuestro simulador, las conclusiones a las que llegamos coinciden en su mayor parte con las de Fujimoto. Sin embargo, nuestros resultados nos llevan a relativizar el efecto del lookahead en el speedup que se puede obtener con un simulador paralelo conservador. La carga del simulador (número total de trabajos que se mueven por el toro) es mucho más importante para conseguir speedups aceptables: los sistemas muy cargados exigen un alto grado de comunicación entre los *PL*s, lo que posibilita que estos avancen en la simulación a la vez que procesan trabajos útiles. El lookahead juega un papel importante en sistemas poco cargados, cuando los *PL*s tienen que sincronizarse de manera artificial usando mensajes nulos. De nuestros experimentos se deduce también que los speedups obtenidos escalan muy bien con el número de procesadores empleados, por lo que merece la pena paralelizar al máximo para conseguir buenos resultados. De lo que no queda ninguna duda, a la luz de nuestros experimentos, es de que el parámetro que más influye a la hora de conseguir un buen speedup es la granularidad de la simulación (parámetro *peso*). Se puede encontrar una descripción más detallada de nuestros experimentos en [26] y [27], y un resumen de los resultados más importantes en la Tabla 2 [28].

Nº procesadores	A	B	C	D	E
16	0,29	2,52	3,39	5,32	9,56
9	0,20	1,33	2,57	-	-
4	0,09	0,73	1,02	-	-
1	1	1	1	1	1

Tabla 2: Speedups conseguidos con un simulador paralelo conservador. Simulación de una red toroidal del 12x12 nodos con disciplina FCFS. (A) Carga baja, lookahead bajo, peso nulo. (B) Carga media, lookahead medio, peso nulo. (C) Carga alta, lookahead alto, peso nulo. (D) Carga baja, lookahead bajo, peso elevado. (E) Carga media, lookahead bajo, peso elevado. Los datos para 1 procesador han sido medidos con un simulador secuencial.

5.4 Incremento del lookahead

Con el objetivo de conseguir mayores grados de speedup, Nicol propone en [30] realizar modificaciones a los algoritmos de Fujimoto para incrementar la capacidad de lookahead de los *PL*s, aprovechando características propias de los sistemas que se simulan: redes de colas FCFS. Para Fujimoto, cuando un trabajo entra en un nodo lo único que se puede determinar es que no saldrá de él, y por tanto no se generará ningún mensaje, por lo menos durante el tiempo mínimo de servicio. A partir de esta consideración se efectúa el cálculo del lookahead. Esta decisión es excesivamente pesimista, porque es muy probable que el trabajo esté en el nodo un tiempo mayor al mínimo.

Con el objetivo de incrementar la capacidad de lookahead de los *PL*s, Nicol propone calcular el tiempo de servicio de un trabajo en el momento en el que éste se recibe. Al fin y al cabo, en el simulador esto supone poco más que hacer un cálculo de un número aleatorio. Ese trabajo entra en la cola correspondiente al nodo, junto con el resto de los trabajos pendientes de ser procesados y cuyo tiempo de proceso también se conoce. En estas circunstancias, el *PL* puede calcular exactamente el tiempo que va a tardar el trabajo en salir del nodo: el correspondiente a la suma de los tiempos de servicio de los trabajos en cola más el tiempo de servicio que le resta al trabajo que en ese momento está siendo servido. Así, cada *PL* tiene un amplio conocimiento de los mensajes que va a producir en un futuro más o menos cercano.

En este caso, el lookahead no es una cantidad fija, sino que está continuamente siendo recalculado. Se aprecia claramente que su valor es, en promedio, mucho mayor que el calculado por Fujimoto. Si se utiliza un sistema de evitación de interbloqueos, este método hace que el número de mensajes nulos que circulan por la red sea más reducido que el necesario sin esta optimización, ya que estos mensajes se generan cada más tiempo e informan de incrementos de timestamp mayores, lo que permite que la simulación progrese más rápidamente.

Wagner y Lazowska profundizan en el trabajo iniciado por Nicol, mostrando cómo técnicas similares a las del cálculo del lookahead en redes de colas FCFS pueden ser aplicadas para otras disciplinas de colas: MCPP (*Multiple Class Preemptive Priority*), LCFS (*Last Come First Serve*) y PS (*Processor Sharing*) [41]. La disciplina PS es una idealización de Round Robin. Con RR, cada trabajo recibe un *quantum* de servicio antes de ceder el turno al siguiente trabajo de la cola, reincorporándose al final de la misma. PS es el caso límite de RR, cuando el *quantum* tiende a cero. El centro de servicio atiende simultáneamente a varios trabajos, pero cada uno de los n trabajos sólo recibe $1/n$ de su atención. Se puede encontrar una descripción de estas disciplinas de colas en [24]. Por su parte, Nicol también ha continuado investigando en esta línea, desarrollando métodos eficientes de simular, de forma conservadora, redes de colas con prioridad [31].

6 Métodos optimistas

El segundo gran grupo de técnicas para realizar simulaciones paralelas dirigidas por sucesos lo constituyen los métodos optimistas, basados en el mecanismo *Time Warp* propuesto por Jefferson [20]. Los métodos optimistas surgen con posterioridad a los conservadores, en parte debido a los siguientes puntos criticables en los simuladores conservadores:

- La topología de interconexión de los *PL*s debe, en un simulador conservador, estar definida estáticamente. Esta restricción es, en algunos casos, excesiva: en determinadas simulaciones los *PL*s deberían crearse dinámicamente y la topología de interconexión debería poder variar con el tiempo, si así lo requiriese el sistema físico simulado. Una simulación conservadora puede emular este comportamiento creando desde el principio todos los *PL*s que se prevén necesarios, así como todas las interconexiones que se estima serán precisas, pero esto supone una seria sobrecarga en la simulación: se ocupan permanentemente recursos (*PL*s, canales, colas de entrada) que sólo se usan en determinados momentos de la simulación.
- Se ha comprobado que existen modelos que, pese a disponer de un alto grado de paralelismo, no se adecúan a una simulación conservadora, exhibiendo los simuladores speedups sorprendentemente bajos. Los métodos conservadores eliminan el problema de los errores causales, pero a cambio introducen el riesgo de interbloqueo en el simulador, y evitar o resolver los interbloqueos es, en algunos casos, prohibitivamente costoso.

Los métodos optimistas se diferencian de los conservadores en la forma de tratar los errores de causalidad. Mientras los métodos conservadores los evitan por construcción, los métodos optimistas llegan a *permitir* que estos errores ocurran. Cada *PL* del simulador puede ir consumiendo mensajes tan rápidamente como le sea posible, sin tener en cuenta las restricciones causales. Puede ocurrir en un momento dado que se reciba un mensaje con un timestamp anterior al del último mensaje consumido en el *PL*. En estas circunstancias, no se cumple el principio básico de que cada *PL* debe consumir los mensajes que recibe en orden de timestamp. Por lo tanto, no tenemos garantías de que la simulación se esté llevando a cabo correctamente. Denominamos *mensaje extraviado* a aquel que ha permitido detectar la situación de riesgo.

Una vez recibido el mensaje extraviado, el *PL* debe retomar su trabajo en un punto en el que tenga garantías de que no hay errores causales. Para ello lo que hace es emplear un mecanismo de *vuelta atrás* o *rollback*, que deshace parte de la simulación realizada. El *PL* retoma la simulación en un instante de tiempo igual o anterior al del timestamp del mensaje extraviado, y después consume el mensaje extraviado. Así, si se cumple la restricción de procesar los mensajes recibidos en orden de timestamp.

Durante el tiempo en el que no se ha respetado el orden temporal de los mensajes, el *PL* habrá modificado, con toda seguridad, sus variables de estado y enviado mensajes a otros *PL*s. El proceso de vuelta atrás supone retomar un estado anterior, lo que exige, por un lado, recuperar un estado libre de errores y, por otro, deshacer los envíos de mensajes realizados durante el periodo de cómputo erróneo. Para que sea posible recuperar un estado pasado, debe realizarse periódicamente una salvaguarda del estado del *PL*. Para deshacer los envíos de mensajes se envían *antimensajes* que tienen la propiedad de *aniquilar* mensajes normales.

Un antimensaje es una copia *negativa* de un mensaje normal (*positivo*) generado anteriormente. Cuando un *PL* realiza una vuelta atrás envía un antimensaje por cada mensaje positivo enviado durante el periodo de simulación errónea. Al recibir un antimensaje, un *PL* debe comprobar si la copia positiva correspondiente ha sido ya consumida o no. Si no lo ha sido (está aún en una cola de entrada), el *PL* se limita a eliminar las dos copias, positiva y negativa. Si hubiese sido procesado, el *PL* se enfrenta a un riesgo causal (ha consumido un mensaje que no debería haber sido generado), por lo que debe realizar una vuelta atrás para deshacer los efectos originados por el consumo del mensaje positivo. La vuelta atrás puede forzar a su vez el envío de nuevos antimensajes. Así, de forma recursiva, es posible anular los efectos de un cómputo erróneo. A continuación vamos a describir los pasos fundamentales del algoritmo de simulación optimista que sigue cada *PL*, así como las estructuras de datos necesarias.

En lo que se refiere a la imposición de una topología de interconexión estática en los métodos conservadores, los optimistas eliminan esta deficiencia manteniendo en cada *PL* una única cola de entrada. No existe una cola asociada a cada canal de entrada, ni tampoco se utilizan relojes de canal. Todos los mensajes que un *PL* recibe se almacenan, de forma ordenada por timestamp, en una cola única, sin distinguir su origen. El mecanismo de paso de mensajes que dé soporte al simulador debe permitir comunicación entre dos *PL*s cualesquiera, pero no es necesaria una reserva previa de recursos para cada par de *PL*s comunicantes.

6.1 Algoritmo de simulación optimista

Para implementar el algoritmo de simulación descrito por Jefferson, cada *PL* debe mantener las siguientes estructuras de datos:

- Una *cola de entrada*, en la que se almacenan los mensajes recibidos, en orden no decreciente de timestamp. Deben almacenarse en ella *todos* los mensajes, tanto los que están pendientes de consumo como los ya consumidos. Asociado a la cola de entrada se tendrá un apuntador al primer mensaje sin procesar, al que denominamos *siguiente_suceso.*

- Una *cola de estados*, que contiene las copias del estado del *PL* salvadas cada cierto tiempo. Aunque existen formas más optimizadas de mantener esta cola, se supone que el estado de un *PL* se salva justo inmediatamente antes de consumir cada mensaje.
- Una *cola de salida*, en la que se guardan copias negativas de todos los mensajes generados por el *PL*.
- El *reloj* local, que indica el instante hasta el cual ha progresado el *PL*. Se dice que este reloj indica la *hora virtual* del *PL*.

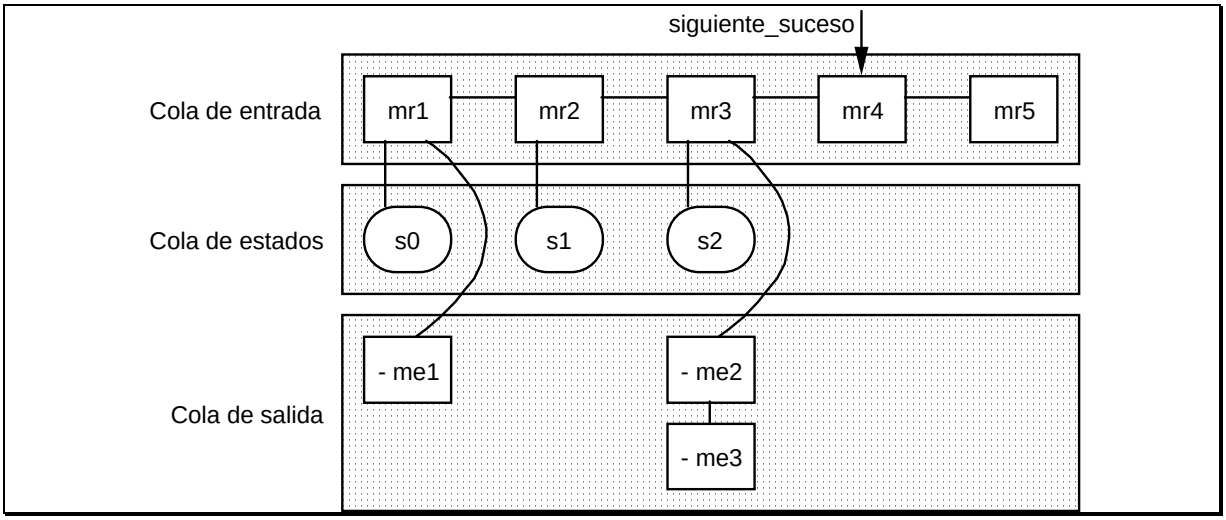

Figura 9: Estructuras de datos mantenidas por cada *PL* en una simulación que usa Time Warp. Antes de consumir el mensaje *mr3* se realizó una copia del estado del *PL*, almacenada en la cola de estados como *s2*. Como consecuencia de simular el suceso asociado a *mr3*, se enviaron los mensajes *me2* y *me3*, por lo que se han almacenado dos antimensajes en la cola de salida. En estos momentos, el mensaje candidato a ser consumido es *mr4*.

La Figura 9 representa de forma gráfica las mencionadas estructuras de datos, así como la relación entre ellas. La secuencia de acciones que realiza cada *PL* viene dada por el siguiente algoritmo:

1. Si no hay mensajes sin procesar en la *cola de entrada*, esperar. Una vez que se tiene un mensaje, avanzar al siguiente paso.
2. Guardar una copia del estado del *PL* en la *cola de estados*.
3. Consumir el mensaje, esto es, avanzar el *reloj* y ejecutar el código de simulación asociado al suceso contenido en el mensaje.
4. Añadir a la *cola de salida*, por cada mensaje enviado en el paso anterior, el antimensaje correspondiente.
5. Una vez consumido el mensaje, avanzar el apuntador *siguiente_suceso* y volver al paso 1.

El algoritmo puede verse interrumpido cuando se recibe un nuevo mensaje. Ese mensaje puede ser positivo o negativo, y puede tener un timestamp menor (pasado) o mayor (futuro) que el del suceso que se está procesando, o del último suceso que se procesó si no hay ninguno en ejecución. Se pueden considerar cuatro casos:

1. Mensaje positivo en el futuro. Se trata de un mensaje normal, con información sobre un suceso planificado para el futuro. Se encola en la *cola de entrada*.

2. Antimensaje en el futuro. Se localiza el mensaje positivo correspondiente, que debe encontrarse en la *cola de entrada*, pendiente de ser consumido. Se aniquilan ambos.

3. Mensaje positivo en el pasado (extraviado). Se ha producido un error de causalidad, por lo que se inicia una vuelta atrás. Todos los efectos de simular los sucesos posteriores al extraviado deben deshacerse, para luego volver a simularlos. Para cada suceso anulado, se elimina de la *cola de estados* el estado asociado, y se envían los antimensajes necesarios. El apuntador *siguiente_suceso* debe pasar a señalar al mensaje extraviado. Tras todo esto, se continúa normalmente.

4. Antimensaje en el pasado. Se localiza su mensaje positivo, ya consumido, y se aniquilan los dos. Además, debe volverse atrás hasta el estado asociado al mensaje positivo aniquilado, anulando todos los mensajes enviados con posterioridad, tal como se describe en el punto anterior. El apuntador *siguiente_suceso* debe señalar al mensaje positivo inmediatamente posterior al aniquilado. Tras esto, se continúa normalmente. La Figura 10 representa esta situación.

Conviene repetir que, una vez que un *PL* recibe un mensaje extraviado, sus efectos raramente se limitan al *PL* receptor, sino que se produce una "reacción en cadena": el *PL* realiza un rollback, lo que supone el envío de cierto número de antimensajes a otros *PL*s, tal como se describe en el caso 3. Estos antimensajes pueden forzar a su vez la realización de rollbacks, de la forma descrita en el caso 4, con lo que la simulación incorrecta se va corrigiendo progresivamente en todos los *PL*s afectados.

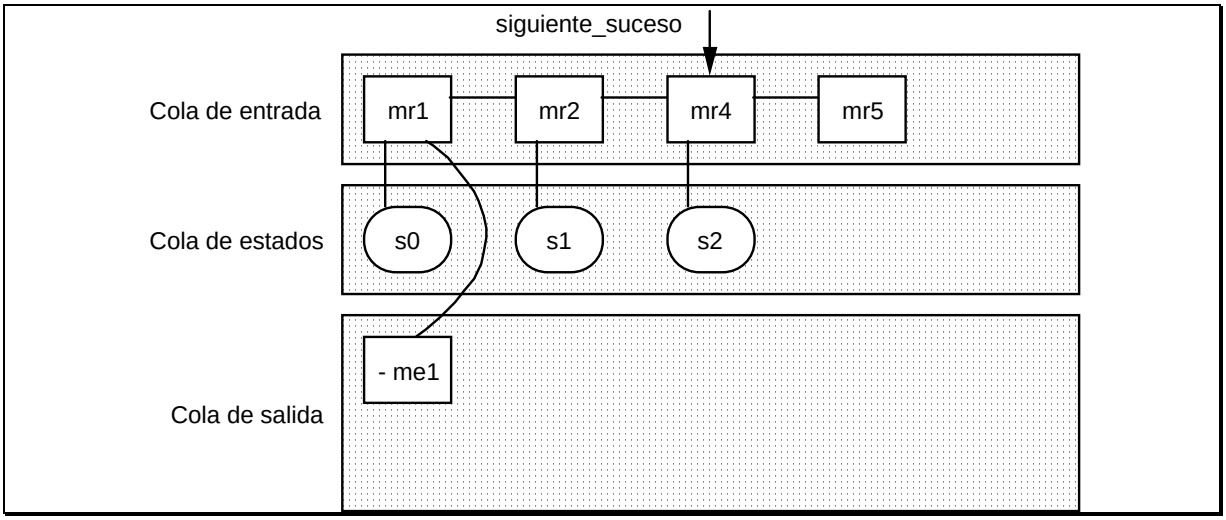

Figura 10: Efecto que tendría, en el *PL* de la Figura 9, la recepción del antimensaje *-mr3* antes de consumir *mr4*. Se ha recuperado el estado s2, inmediatamente anterior al consumo de *mr3*. Se ha aniquilado *mr3*, y se han enviado los antimensajes *-me2* y *-me3*. Una vez hecho esto, el proceso de rollback ha terminado. Se puede continuar y consumir *mr4*.

6.2 Control Global

Aunque en Time Warp la mayoría de las operaciones se pueden hacer de forma local, tal como se ha expuesto en el apartado anterior, para que el sistema sea realmente operativo es necesaria la realización de una serie de operaciones globales que, entre otras cosas, satisfagan los siguientes requerimientos:

- Garantizar que la simulación progresa a pesar de las vueltas atrás. En cada *PL* ocurre que el reloj local no indica de forma fiable hasta qué punto ha avanzado la simulación: puede ocurrir que llegue un mensaje extraviado y se produzca un salto atrás en el tiempo. Se necesita algún método para fijar un punto seguro en el tiempo, de tal forma que se pueda certificar que nunca se va a retroceder hasta antes de ese punto.

- Detectar el final de la simulación. El hecho de que el reloj de un *PL* alcance el valor que marca el final de la simulación no garantiza que ésta haya terminado de forma efectiva. De nuevo existe la posibilidad de dar marcha atrás en el tiempo y tener que rehacer parte de la simulación.

- Realizar operaciones de entrada/salida (normalmente, lecturas y escrituras en terminales y/o ficheros) sin riesgo. Si un *PL* tiene que realizar una operación de E/S en un instante de tiempo dado, debe estar seguro de que esa operación no va a tener que ser cancelada en una vuelta atrás. Las operaciones de E/S tienen, normalmente, un carácter irrevocable, por lo que se consideran no cancelables.

- Gestionar la memoria. Este es uno de los problemas más serios del Time Warp: de las descripciones que se han hecho sobre las estructuras de datos que manejan los *PL*s, se deduce que dichas estructuras crecen sin medida conforme avanza la simulación. Se guarda información sobre el pasado como precaución ante una posible vuelta atrás, pero hay que poner algún límite porque, de lo contrario, se puede agotar la capacidad de memoria asignada a cada *PL*.

Para resolver todos estos problemas, una simulación optimista cuenta, además de con la colección de *PL*s, con un mecanismo de *control global*. Este control global debe realizar periódicamente el cálculo del la *hora global virtual* (*HGV*), que es un indicador de hasta qué punto se ha completado la simulación, tomándola de forma global. Esta HGV se calcula como el mínimo de entre los timestamps de todos los mensajes sin consumir en el simulador (ver Figura 11).

Partiendo del principio de que consumir un mensaje nunca puede tener efectos en el pasado, se puede asegurar que no es posible que se produzca una vuelta atrás hasta un instante de tiempo anterior a la HGV. Por lo tanto, todo el espacio de memoria que ocupan los mensajes que ya han sido consumidos y cuyos timestamps son anteriores a esa hora puede ser liberado. Lo mismo se puede decir para todas las copias de estados pasados de los *PL*s y para los antimensajes anteriores a la HGV. Además, operaciones irrevocables, como las de entrada/salida, pueden realizarse con total garantía cuando la HGV sobrepasa el instante en el que esas operaciones deben ser realizadas. Se conoce como *recolección de fósiles* al proceso de liberar memoria ocupada por sucesos antiguos y realizar operaciones irrevocables (normalmente, las de E/S). Además, la HGV también sirve para indicarle al simulador de forma fiable cuándo debe parar: en el momento en que la HGV alcance el valor predeterminado para el fin de la simulación.

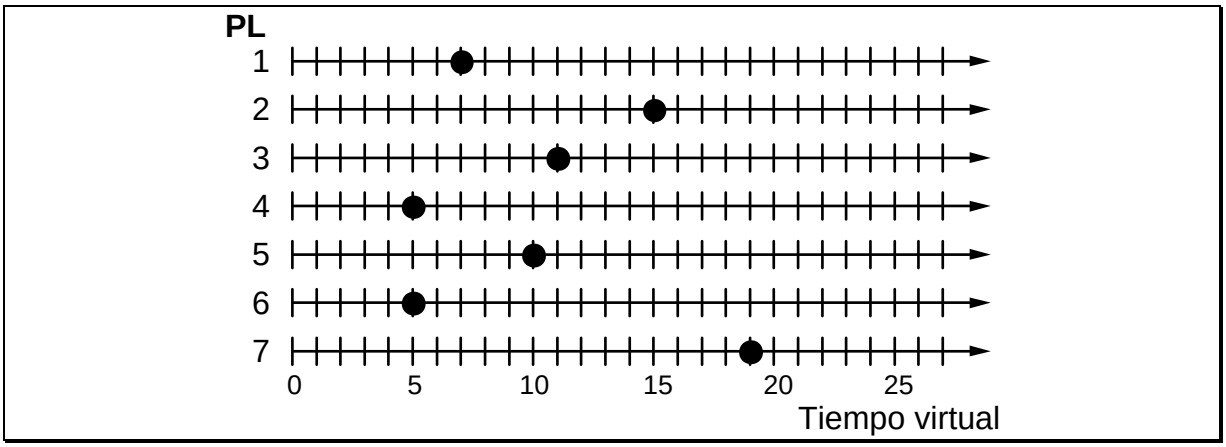

Figura 11: Instantánea de los valores de los relojes de los *PL*s en una simulación Time Warp. La HGV es 5, y todo cómputo realizado con timestamp menor está consolidado. El resto del cómputo es *especulativo*, porque puede que haya que deshacerlo en un rollback [11].

La complejidad que supone la gestión de la memoria asignada a los *PL*s y la necesidad de este control global es lo que hace que no sea sencillo utilizar el Time Warp para realizar una simulación distribuida, si se compara con los requerimientos de los métodos conservadores. Sin embargo, todo este soporte puede ser realizado independientemente del modelo que se esté simulando, por lo que, una vez hecho, puede ser reutilizado para simulaciones de diferentes sistemas físicos, mientras que los simuladores conservadores están más "hechos a medida". Se ha creado así, por parte del equipo de Jefferson, el Time Warp Operating System, que da soporte a todas sus simulaciones [21].

6.3 Variaciones sobre el algoritmo básico

Fujimoto caracteriza en [14] cuatro fuentes de overhead que aparecen utilizando Time Warp para una simulación paralela, en comparación con una simulación secuencial equivalente:

- Mantenimiento de la historia de los *PL*s, esto es, de la *cola de estados*, de la *cola de entrada* y de la *cola de salida*.
- Paso de mensajes. Esto es inherente al tipo de simulaciones paralelas que estamos describiendo. Los *PL*s deben emplear parte de su tiempo preparando los mensajes, enviándolos, recibiéndolos y extrayendo de ellos información. Además, mover un mensaje de un *PL* a otro puede ser costoso, dependiendo de la máquina y el mecanismo de comunicación empleado.
- Cancelaciones y rollbacks. Hacer un rollback en un *PL* no supondrá demasiados problemas, pero un rollback produce normalmente una avalancha de rollbacks en otros *PL*s, y el movimiento de un número importante de antimensajes.
- Computaciones erróneas. Todo el tiempo que dedica un *PL* a simular sucesos cuyos efectos van a tener que ser deshechos es un esfuerzo inútil.

Caracterizados estos aspectos, se pueden encontrar en la literatura mecanismos que intentan mejorar el algoritmo básico corrigiendo alguna de sus deficiencias [34, 25, 17].

En el algoritmo de simulación optimista descrito, que podemos denominar básico, cada vez que se realiza un rollback se envía, de forma inmediata, un antimensaje por cada mensaje enviado durante el cómputo erróneo. A esta política de gestión de los mensajes se la denomina denomina *cancelación agresiva*. De la misma forma, la política de eliminar copias pasadas del estado de un *PL* en cada rollback se denomina *re-evaluación agresiva*. Se han desarrollado técnicas alternativas de *cancelación perezosa* y *re-evaluación perezosa*, con el objetivo de minimizar algunos de los efectos negativos de los rollbacks [15, 16].

La cancelación perezosa trata de minimizar la sobrecarga de trabajo que supone enviar antimensajes en una vuelta atrás y de reducir las avalanchas de rollbacks. Con esta técnica,

cuando se produce un rollback en un *PL*, no se procede a enviar inmediatamente los antimensajes correspondientes a mensajes positivos erróneos. En vez de ello, el *PL* continúa progresando en la simulación, a la vez que comprueba si en ese progreso se generan *exactamente los mismos* mensajes que durante la fase errónea. Si así ocurriese, podrían evitarse los procesos de aniquilación y de reevaluación de los sucesos asociados en otros *PL*s. Las ventajas de esta modificación al Time Warp original no son simpre evidentes. En determinados campos de aplicación puede ocurrir que la cancelación perezosa deje avanzar excesivamente una simulación errónea, provocando posteriores vueltas atrás y limitando el avance de la simulación.

La re-evaluación perezosa consiste en no eliminar inmediatamente las copias del estado del *PL* posteriores a las del mensaje extraviado en una vuelta atrás. Si, tras evaluar el mensaje extraviado, resulta que el estado es igual al que había *antes* de procesar dicho mensaje, entonces es evidente que, a no ser que lleguen nuevos mensajes extraviados, la re-evaluación de los sucesos será idéntica a la evaluación original. Por lo tanto, no se necesita re-simular los sucesos afectados por la vuelta atrás. Aunque las ventajas de implementar este mecanismo parecen obvias, lo cierto es que suponen una incremento considerable en la complejidad de los programas, lo que no siempre resulta conveniente.

6.4 Evaluaciones de los métodos optimistas

Los estudios más importantes sobre los métodos optimistas han sido realizados por el equipo de Jefferson y Wieland, descritos en [43, 44], y por Fujimoto, recogidos en [14].

Los experimentos de Fujimoto aprovechan la circunstancia de utilizar en sus experimentos un sistema con memoria compartida para implementar una optimización denominada *cancelación directa*, que permite enviar mensajes mediante escrituras en variables compartidas y elimina la necesidad de utilizar antimensajes. Se simulan redes de colas, con diferentes disciplinas (FCFS, prioridades sin expulsión, prioridades con expulsión) y con topología hipercubo, usando un multiprocesador BBN Butterfly-1. Se comparan los resultados con simulaciones conservadoras, con evitación de interbloqueo y con recuperación de interbloqueo. Entre las principales conclusiones destaca que, en general, las prestaciones de Time Warp son mucho mejores que las de los métodos conservadores, especialmente en aquellos casos en los que resulta difícil para los métodos conservadores extraer lookahead del dominio simulado, como puede ser el caso de una red de colas con prioridad. Con un modelo de estas características y usando para los simuladores 8 procesadores, los speedups obtenidos con métodos conservadores, independientemente del método de tratamiento de interbloqueos utilizado, ni siquiera llegan a 1, mientras que se acerca a 7 cuando se utiliza Time Warp.

En lo que se refiere a métricas sobre el propio mecanismo de simulación, se muestra cómo el tamaño de la información de estado que hay que salvaguardar periódicamente afecta de forma

drástica al speedup de la simulación. Si la información de estado es muy voluminosa, la eficacia de estos métodos es muy reducida. También se muestra cómo el speedup escala bastante bien con el número de procesadores, al menos para los modelos elegidos. Por ejemplo, simulando una red de 256 nodos consigue unos valores de speedup de aproximadamente 7 con 8 procesadores, 15 con 16 procesadores, 30 con 32 procesadores y 57 con 64 procesadores.

Los estudios de Jefferson, Wieland y su equipo consisten en aplicar el Time Warp a diferentes dominios de simulación (combates, redes de comunicaciones, sistemas biológicos, etc.). Los resultados de estas simulaciones no se comparan con los obtenidos con algoritmos conservadores, aunque sí se dan cifras de speedups. Por ejemplo, en [44] se describe la simulación, utilizando Time Warp, de modelos que representan combates entre ejércitos. Utilizan dos computadores paralelos diferentes: un Caltech/JPL Mark III Hypercube [39] y un BBN Butterfly, y consiguen un speedup de 28.6 con 60 procesadores en el Mark III y 36.8 con 100 procesadores en el Butterfly.

Tallieu y Verboven describen y analizan en [35] una simulación de una red Ethernet sobre un sistema multicomputador basado en Transputers, de características similares al nuestro. Desgraciadamente, no dan cifras de speedup, ni comparan con otras técnicas de simulación paralela. Sí concluyen que, para el sistema concreto que simulan, el comportamiento del simulador no es bueno, porque se llega a una situación en la que el número de rollbacks es prácticamente igual al número de mensajes consumidos por el simulador. El motivo es que se manejan *LP*s con diferentes escalas de tiempo (entre el misilegundo y el nanosegundo), lo que hace que una vuelta atrás de un milisegundo en un *LP* supone, por poner un ejemplo, el envío de un antimensaje, mientras que en otro (que se mueve en una escala de nanosegundos) hay que enviar varios cientos. Esto conlleva, además, serios problemas en lo que se refiere al almacenamiento de antimensajes en las colas de salida.

Nº procesadores	A	B	C	D	E
16	0,40	0,68	0,93	3,78	3,25
9	0,32	0,48	0,74	-	-
4	0,20	0,34	0,49	-	-
1	1	1	1	1	1

Tabla 3: Speedups conseguidos con un simulador paralelo optimista. (A) Carga baja, lookahead bajo, peso nulo. (B) Carga media, lookahead medio, peso nulo. (C) Carga alta, lookahead alto, peso nulo. (D) Carga baja, lookahead bajo, peso elevado. (E) Carga media, lookahead bajo, peso elevado.

Nuestro grupo también ha realizado experimentos similares a los descritos en el apartado 5.3, esta vez utilizando una implementación del mecanismo Time Warp. Nuestras conclusiones no son tan alagüeñas como las de Fujimoto, estando más en línea con las de Tallieu-Verboven,

tal puede verse a partir de los datos representados en la tabla 3. En efecto, la gestión de la memoria (de las estructuras de datos de cada PL) es un punto crítico, hasta el punto de que, en algunos de los casos probados, el overhead que supone esta gestión en un simulador optimista el mucho mayor al overhead que suponen los bloqueos y su resolución en un simulador conservador. Además, si no se pone ningun límite al optimismo de los PLs, éstos pueden avanzar demasiado deprisa y consumir toda la memoria asignada; como efecto añadido, un ulterior rollback genera tal avalancha de antimensajes que desborda la capacidad de nuestro sistema (software) de paso de mensajes. Por último, aunque el algoritmo optimista es tan bueno como el conservador a la hora de distribuir trabajo entre procesadores, si se tiene en cuenta que mucho del trabajo que se realiza anticipadamente tiene más tarde que ser deshecho (cosa que nunca ocurre en un simulador conservador), no está tan claro que los modelos de grano grueso, en los que la simulación de un suceso requiere un alto esfuerzo de computación (en el simulador, alto valor en el parámetro *peso*) sean apropiados para su simulación con Time-Warp.

Debemos aclarar que estas conclusiones son fruto de un conjunto reducido de experimentos. Nuestro trabajo con Time-Warp acaba de empezar: en futuros trabajos iremos afinando estas conclusiones a partir de un conjunto más amplio de experimentos, y de refinamientos en la implementación del simulador.

7 Conclusiones

La simulación por computador es un área de trabajo en la que todo método para acelerar las ejecuciones de los programas es bienvenido, porque así se pueden realizar simulaciones con gran detalle y/o simulaciones de sistemas más complejos en menor tiempo. Los avances en la velocidad de proceso de los sistemas monoprocesador han sido relevantes, pero no resultan suficientes. A corto y medio plazo, los sistemas paralelos son mucho más prometedores, por lo que es necesario paralelizar los programas de simulación si se quiere aprovechar la elevada potencia de cálculo de estas arquitecturas.

Puesto que los métodos de simulación por sucesos utilizados habitualmente son inherentemente secuenciales, estando únicamente adaptados para obtener el máximo rendimiento de un monoprocesador, se impone el desarrollo de nuevos algoritmos específicamente diseñados para sistemas paralelos. Hemos descrito las dos familias de métodos de simulación paralela por sucesos discretos con más presencia en la literatura. Ambas familias paralelizan la simulación de un sistema físico asignando procesos lógicos a cada uno de los procesos físicos que se simulan; esos procesos lógicos interactúan intercambiándose mensajes. La diferencia fundamental entre las dos familias está en la forma de sincronización de los *PL*s, para que todos ellos avancen de una forma ordenada. En un caso nos encontramos con métodos *conservadores*, en los cuales los *PL*s no avanzan hasta

estar completamente seguros de que van coordinados con los demás. En contraposición con los métodos *optimistas* los *PL*s avanzan cada uno a su propio ritmo y, si se detecta un error de sincronización, se realizan mecanismos de *vuelta atrás* para recuperar un estado sin errores.

Las evaluaciones de los métodos de simulación paralela son en algunos casos muy esperanzadoras, mientras que en otros se muestran inútiles. Se comprueba que, en términos generales, ninguno de los métodos es eficiente "per se", sino que el rendimiento depende de parámetros propios del modelo que se está simulando: características como el grado de acoplamiento de los *PL*s, la carga del sistema o el *lookahead* del modelo. Aún así, la mayor parte de los trabajos realizados indican que los métodos optimistas superan en general a los conservadores.

Conviene destacar que, de cara al usuario de los simuladores (la persona que describe un modelo y que quiere estudiar su comportamiento) no debería existir ninguna diferencia entre usar un simulador secuencial o uno paralelo. La descripción del modelo es independiente del "motor" de la simulación. Eso sí, dada la alta dependencia entre el speedup conseguido y el modelo simulado, sería necesario que los simuladores fuesen capaces, a partir de la descripción del modelo a simular, de extraer las características más importantes del mismo y eligiesen el la técnica de simulación más apropiada, teniendo en cuenta que "paralelo" no necesariamente va a significar "más rápido". En este sentido, las técnicas de simulación optimistas parecen más prometedoras, al ser menos dependientes del sistema físico a simular [27]. En la actualidad existen varias herramientas de simulación en las que el "motor" se da implementado, y el usuario se limita a describir el modelo usando un lenguaje específico. Estas herramientas pueden ampliarse suministrando motores alternativos (paralelos), manteniendo el lenguaje de descripción de modelos. Por lo tanto, el esfuerzo añadido que se puede requerir al usuario es el de selección del motor a emplear y facilitar, en el caso del simulador paralelo conservador, información sobre el *lookahead.*

Aunque queda aún mucho trabajo por hacer en este área, dado que a nosotros nos interesa aplicar las técnicas de simulación paralela para un conjunto de modelos muy concretos, relacionados con las redes de interconexión para arquitecturas masivamente paralelas, esperamos que una evaluación de los métodos aquí descritos nos permitan seleccionar el que, en líneas generales, mejores resultados nos ofrezca.

Referencias

[1] A. Arruabarrena, R. Beivide, E. Herrada, J.C. Balcázar, C. Izu, "An Optimal Topology for Multicomputer System Based on a Mesh of Transputers". OUG Newsletter, No. 12, Jan. 1990, pp. 31—39.

[2] A. Arruabarrena, R. Beivide, E. Herrada, “A Comparative Study of Bidimensional Grid Topologies with Oblivious Cut-through Routing”. Technical Report EHU-KAT-IK-93-2, Univ. of the Basque Country, Computer Architecture and Technology Dept., 1993.

[3] R. Beivide, E. Herrada, J.L. Balcázar, A. Arruabarrena, “Optimal Distance Networks of Low Degree for Parallel Computers”. IEEE Trans. Comp., Vol. 40, No. 10, 1991, pp. 1109—1124.

[4] J. Banks and J.S. Carson. “Discrete-Event System Simulation”. Prentice Hall, Inc. 1984. ISBN 0-13-215582-6.

[5] R.E. Bryant, “Simulation of Packet Communications Architecture Computer Systems”. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[6] K.M. Chandy and J. Misra, “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs”. IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, Sept. 1979, pp. 440—452.

[7] K.M. Chandy and J. Misra, “Asynchronous Distributed Simulation via a Sequence of Parallel Computations”. Comm. of the ACM, Vol. 24, No. 11, April 1981, pp. 198—205.

[8] K.M. Chandy, J. Misra and L.M. Haas, “Distributed Deadlock Detection”. ACM Transactions on Computer Systems, Vol. 1, No. 2, May 1983, pp. 144—156.

[9] W. Crowther et. al., “The Butterfly Parallel Processor”. IEEE Comp. Arch. Tech. Comp. Newsletter, sept/dic. 1985, pp. 18—45.

[10] A.L. Decegama, “The Technology of Parallel Processing” Vol. 1. Prentice-Hall, 1989. ISBN 0-13-898438-7.

[11] R.E. Felderman, L. Kleinrock, “Bounds and Approximations for Self-Initiating Distributed Simulation Without Overhead”. ACM Trans. on Modeling and Comp. Simulation, Vol. 1, No. 4, Oct. 1991, pp. 386—406.

[12] R.M. Fujimoto, “Lookahead in Parallel Discrete Event Simulation”. Proc. Int. Conf. Parallel Processing, Aug. 1988, pp 34—41.

[13] R.M. Fujimoto, “Performance Measurements of Distributed Simulation Strategies”. Trans. of The Society for Comp. Simulation, Vol. 6, No. 2, 1989, pp. 89—132.

[14] R.M. Fujimoto, “Time Warp on a Shared Memory Multiprocessor”. Proc. Int. Conf. Parallel Processing, Vol. III, Aug.1989, pp. 242—249.

[15] R.M. Fujimoto, “Optimistic Approaches to Parallel Discrete Event Simulation”. Trans. of The Society for Comp. Simulation, Vol. 7, No. 2, 1990, pp. 153—191.

[16] R.M. Fujimoto, “Parallel Discrete Event Simulation”. Comm. of the ACM, Vol. 33, No. 10, Oct. 1990, pp. 31—53.

[17] D.W. Glazer, C. Tropper, “On Process Migration and Load Balancing in Time Warp”. IEEE Trans . on Parallel and Dist. Systems, Vol. 4 No. 3, March 1993, pp. 318 —327.

[18] “The Transputer Databook” INMOS Databook Series, INMOS Ltd., Bristol, U.K., 1989.

[19] C. Izu, R. Beivide, C.R. Jesshope, A. Arruabarrena, “Experimental Evaluation of Mad Postman Bidimensional Routing Networks”. Proc. 19th Euromicro Conf., Open System Design, 1993.

[20] D.R. Jefferson, “Virtual Time”. ACM Trans. on Programming Languages and Systems, Vol. 7, No. 3, July 1985.

[21] D.R. Jefferson et. al., “Distributed Simulation and the Time Warp Operating System”, Proc. 11th Annual ACM Symp. on Operating System Principles, Austin, Texas, Nov. 1987, pp. 77—93.

[22] P. Konas, P-C. Yew, “Parallel Discrete Event Simulation on Shared Memory Multiprocessors”. Proceedings of the 24th Annual Simulation Symposium, New Orleans, Luisiana, April 1991, pp. 134—148.

[23] D. Kumar, “Systems with Low Distributed Simulation Overhead”. IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 2, March 1992, pp. 155 —165.

[24] E. Lazowska et. al., “Quantitative System Performance: Computer System Analysis Using Queueing Network Models”. Prentice-Hall, 1984.

[25] Y-B. Lin, B.R. Preiss, “Optimal Memory Management for Time Warp Parallel Simulation”. ACM Trans. on Modeling and Comp. Simulation, Vol. 1, No. 4, Oct. 1991, pp. 283—307.

[26] J. Miguel, A. Arruabarrena and R. Beivide, “Conservative Parallel Discrete Event Simulation in a Transputer-Based Multicomputer”. Proc. World Transputer Congress WTC’93, Aachen (Germany), Sept. 1993.

[27] J. Miguel and M. Graña, “Towards the Distributed Implementation of Discrete Event Simulation Languages”. Proc. Int. Conf. on Decentralized and Distributed Systems ICDDS’93, Palma de Mallorca (Spain), Sept. 1993.

[28] J. Miguel, A. Arruabarrena and R. Beivide, “Simulación de Sistemas de Sucesos Discretos en Arquitecturas Masivamente Paralelas”. Actas de las IV Jornadas de Paralelismo, Universidad de Cantabria, Santander, Sept. 1993.

[29] J. Misra, “Distributed Discrete-Event Simulation”. Computer Surveys, Vol. 18, No. 1, March 1989, pp. 39—65.

[30] D.M. Nicol, “Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks”. Proc. ACM SIGPLAN Symp. Parallel Programming Experience in Applications, Languages and Syst., New Haven CT, July 1988, pp. 124—137.

[31] D. Nicol, “Conservative Parallel Simulation of Priority Class Queueing Networks”. IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 3, May 1992, pp. 294 —303.

[32] D.A. Reed, A.D. Malony and B.D. McCredie, “Parallel Discrete Event Simulation Using Shared Memory”. IEEE Transactions on Software Engineering, Vol. SE-14, No. 4, April 1988, pp. 541—553.

[33] R. Righter and J.C. Walrand. “Distributed Simulation of Discrete Event Systems”. Proc. IEEE, Vol 77, No. 1, Jan. 1989, pp. 99 —113.

[34] L.M. Sokol, B.K. Stucky and V.S. Hwang, “MTW: A Control Mechanism for Parallel Discrete Simulation”, Proc. Int. Conf. Parallel Processing, Vol. III, Aug.1989, pp. 250 —254.

[35] F. Tallieu, F. Verboven, “Using Time Warp for Computer Network Simulation on Transputers”. Proceedings of the 24th Annual Simulation Symposium, New Orleans, Luisiana, April 1991, pp. 112—117.

[36] S. Thakkar, P. Grifford and G. Fielland, “Balance: a Shared Memory Multiprocessor System”. Proc. II Int. Conf. on Supercomp. ICS’87, Vol. 1, pp. 93 —101.

[37] “Connection Machine CM-2 technical summary”. Thinking Machines Corp. Tech. Rep., Thinking Machines Corp., Cambridge, MA, April 1987.

[38] A. Trew and G. Wilson (Eds.), “Past, Present, Parallel: a Survey of Available Parallel Computing Systems”. Springer-Verlag, 1991. ISBN 0-387-19664-1.

[39] J. Tuazon, J. Peterson and M. Pniel, “Mark IIIfp Hypercube Concurrent Processor Architecture”. Proc. III Conf. on Hypercube Concurrent Computers and Applications, Vol. 1. Pasadena, CA, 1988, pp. 71—80.

[40] P. Vakili, “Massively parallel and distributed simulation of a class of discrete event systems: a different perspective”. ACM Trans. on Modeling and Comp. Simulation, Vol. 2, No. 3, July 1992, pp. 214—238.

[41] D.B. Wagner and E.D. Lazowska, “Parallel Simulation of Queueing Networks: Limitations and Potentials”. In ACM Performance Evaluation Review, Vol. 17, No. 1, May 1989, pp. 146—155.

[42] D.B. Wagner, “Conservative Parallel Discrete-Event Simulation: Principles and Practice”. Ph.D. dissertation, Univ. of Washington, 1989.

[43] F. Wieland and D. Jefferson. “Case Studies in Serial and Parallel Simulation”. Proc. Int. Conf. Parallel Processing, Vol. III, Aug.1989, pp. 255—258.

[44] F. Wieland et. al., “The Performance of a Distributed Combat Simulation with the Time Warp Operating System”. Concurrency: Practice and Experience, Vol. 1, No. 1, Sep. 1989, pp. 35—50.

[45] B.P. Zeigler, “Theory of Modelling and Simulation”. Wiley-Interscience, 1976. ISBN 0-471-98142-4.

[46] G. Zorpette, “The Power of Parallelism”. IEEE Spectrum, Sept. 1992, pp. 29—33.