Capítulo 7

Multiprocesadores Escalables

7.1 Introducción

En este capítulo estudiaremos el diseño de máquinas que pueden escalarse de manera práctica hasta centenares o incluso millares de procesadores. La escalabilidad tiene profundas implicaciones a todos los niveles de diseño. Para empezar, debe de ser físicamente posible y técnicamente fácil la ampliación de una configuración dada. Añadir procesadores incrementa la capacidad de computación potencial de un sistema, pero para obtener este potencial, el ancho de banda de la memoria debe aumentar con el número de procesadores. Una solución natural es distribuir la memoria con los procesadores de tal manera que cada procesador tiene acceso directo a la memoria local. Sin embargo, la red de interconexión que conecte estos nodos debe proporcionar un ancho de banda escalable a una latencia razonable. Además, los protocolos usados para transferir datos dentro del sistema deben escalar, al igual que las técnicas usadas para la sincronización. Con protocolos escalables sobre una red escalable, un gran número de transacciones pueden tener lugar en el sistema de forma simultánea no pudiendo basarnos en una información global para establecer el orden o arbitrar el acceso a los recursos. De este modo, para obtener un rendimiento escalable de las aplicaciones, la escalabilidad debe manejarse como un "problema vertical" que incide en todos los niveles del diseño del sistema.

Consideremos dos modelos de diseño familiares que nos permitan concretar algunas de estas cuestiones. Las máquinas de memoria compartida de pequeña escala que se describieron en el capítulo anterior pueden verse como un punto extremo. Un bus compartido tiene típicamente una longitud máxima de 80 cm, un número fijo de slots, y un ancho de banda máximo fijo, por lo que su escalabilidad está limitada. El interface al medio de comunicación es una extensión del interface de la memoria, con líneas de control y estados del controlador adicionales para implementar el protocolo de consistencia. Se establece un orden total mediante el arbitraje del bus, limitándose el número de transacciones que pueden realizarse en un momento dado. La protección se consigue haciendo que todas las operaciones de comunicación pasen a través del mecanismo de traducción estándar de direcciones virtuales a físicas. Existe una confianza total entre los procesadores del sistema, que es visto como un sistema bajo el control de un único sistema operativo que se ejecuta en todos los procesadores, con estructuras de datos comunes. Normalmente, cuando un procesador falla todo el sistema debe de reinicializarse. Existe poca o nula intervención del software entre el modelo de programación y las primitivas hardware. Las máquinas escalables están menos acopladas que los multiprocesadores de memoria compartida basados en bus y nos fuerzan a pensar de nuevo como interactúan los procesadores entre ellos y con la memoria.

En el extremo opuesto podemos considerar las estaciones de trabajo convencionales sobre una red de área local o incluso de área extensa. En este caso no hay un límite claro del escalamiento físico y muy poca confianza entre los procesadores que conviven en el sistema. El interface con el medio de comunicación es el interface usual de un periférico a nivel hardware, con el sistema operativo entre las primitivas a nivel de usuario y la red para asegurar el control de acceso y la protección. Cada elemento de proceso está controlado por su propio sistema operativo, que trata al resto de ordenadores como no fiables. No existe un orden global de las operaciones y el consenso es difícil de alcanzar. El medio de comunicación es externo a los nodos individuales y potencialmente inseguro. Las estaciones de trabajo pueden fallar y reinicializarse de forma independiente, excepto cuando uno está proporcionando servicios a otro. Existe una capa software sustancial entre las primitivas hardware y las operaciones de comunicación a nivel de usuario, independientemente del modelo de programación, por lo que la latencia de las comunicaciones tiende a ser bastante alta y el ancho de banda de las comunicaciones suele ser bajo. Dado que las operaciones de comunicación se manejan mediante software no hay un límite claro en el número de transacciones que se están ejecutando en un momento dado o incluso el significado de las transacciones. Por lo tanto, incluso cuando varios procesadores están trabajando juntos sobre un único problema, es difícil obtener un buen rendimiento del sistema.

Entre estos extremos existe un espectro de alternativas de diseño interesantes y razonables, varias de las cuales pueden observarse en los actuales computadores paralelos comerciales. Muchos de los *sistemas de procesamiento masivamente paralelo* (MPPs) emplean un empaquetamiento sofisticado y redes propietarias de alto rendimiento, con lo que es posible emplazar un alto número de procesadores en un espacio con comunicaciones de baja latencia y con un alto ancho de banda. Otras máquinas escalables usan computadores convencionales como nodos sobre redes rápidas. En cualquier caso, existe un alto grado de seguridad física y la opción de un alto grado de fiabilidad o una autonomía sustancial.

Una máquina escalable que soporte memoria compartida puede elegir excluir o incluir coherencia de caché. El esquema más simple a nivel hardware es excluir la coherencia de caché, centrándose en su lugar en un sistema de memoria escalable. Varias compañias han construido máquinas de este estilo; el Cray T3D es un ejemplo muy conocido. En estas máquinas la memoria se encuentra distribuida entre los nodos y todos los nodos están interconectados a través de una red. Los accesos pueden ser a memoria local o remota -un controlador dentro de cada nodo decide, en base a la dirección, si los datos residen en la memoria local o en una memoria remota. En el último caso se envía un mensaje al controlador de la memoria remota para acceder a los datos.

Estos sistemas tienen cachés, pero no solucionan los problemas de coherencia, los datos compartidos se marcan como no cacheables y únicamente los datos privados se mantienen en las cachés. Por supuesto, el software todavía puede almacenar explícitamente los datos compartidos copiandolos de la porción compartida del espacio de direcciones a la porción local privada del espacio de direcciones. La coherencia se controla por software. La ventaja de este mecanismo es que el soporte hardware necesario es pequeño.

Existen varias desventajas a esta aproximación. En primer lugar, los mecanismos de compilación para una coherencia de caché transparente por software son muy limitados. Las técnicas que existen actualmente se aplican principalmente a programas con bucles paralelos bien estructurados, y estas técnicas presentan una sobrecarga significante por la copia explícita de datos. Para problemas irregulares o problemas que incluyen estructuras de datos dinámicas y punteros (incluyendo los sistemas operativos, por ejemplo), la coherencia de caché basada en compiladores es impracticable en la actualidad. La dificultad básica es que los algoritmos software de coherencia de cachés deben de ser conservadores: cada bloque que pueda ser compartido debe de tratarse como si lo fuera. El resultado es un exceso de sobrecarga debido a la coherencia, ya que el compilador no puede predecir la compartición real de manera suficientemente precisa.

En segundo lugar, sin coherencia de cachés, la máquina pierde la ventaja de captar y usar varias palabras en un único bloque de caché por cerca del coste de captar una única palabra. Los beneficios de la localidad espacial en los datos compartidos no pueden utilizarse cuando en cada referencia a memoria remota sólo se capta una palabra.

En tercer lugar, los mecanismos para tolerar la latencia como la preadquisición son más útiles cuando pueden captar multiples palabras, tales como un bloque de caché, y en donde las palabras captadas permanecen coherentes.

Estas desventajas se magnifican por la alta latencia de los accesos remotos. Por ejemplo, en el Cray T3D un acceso a la caché local tiene una latencia de dos ciclos de relojs y está segmentada, mientras que un acceso remoto tarda unos 150 ciclos.

Por estas razones, la coherencia de caché es un requisito aceptado para multiprocesadores de pequeña escala. Para arquitecturas de gran escala, existen nuevos desafíos a resolver para extender el modelo de memoria compartida con coherencia de cachés. Aunque el bus puede reemplazarse por una red de interconexión más escalable, y podemos distribuir la memoria de tal manera que el ancho de banda de la memoria también aumente al aumentar el número de procesadores, es necesario resolver el problema de falta de escalabilidad de un esquema de coherencia basado en *snooping*. Un protocolo *snooping* necesita una comunicación con todas las cachés en cada fallo de caché, incluyendo las escrituras sobre datos potencialmente compartidos. La ausencia de una estructura de datos centralizada que lleve el estado de las cachés es a la vez la ventaja fundamental de estos esquemas, ya que permiten que no sean costosos, y su talón de Aquiles a la hora de su escalabilidad. Por ejemplo, con únicamente 16 procesadores, un tamaño de bloque de 64 bytes, y un tamaño de caché de 64 KB, la demanda de ancho de banda (ignorando ciclos de parada) para los cuatro programas paralelos que hemos utilizado para evaluar el rendimiento varía de casi 500 MB/sec (para Barnes) a más de 9400 MB/sec (para Ocean), suponiendo un procesador que realiza una referencia a sus datos cada $5ns$. En comparación, el bus de la Silicon Graphics Challenge, el multiprocesador basado en bus con mayor ancho de banda en 1995, proporciona un ancho de banda de 1200 MB.

Alternativamente, podemos construir arquitecturas de memoria compartida escalables que incluyan coherencia de caché. La clave es encontrar un protocolo de coherencia alternativo a los protocolos snooping. Una alternativa es la utilización de protocolos basados en directorios. En este tipo de protocolo existe un directorio que mantiene el estado de cada bloque que puede almacenarse en la caché. La información del directorio incluye las cachés que tienen copias del bloque, si ha sido modificado, etc.

7.2 Coherencia de cachés escalable

Antes de discutir protocolos específicos, esta sección describirá las alternativas de organización principales para proporcionar una replicación coherente en una máquina con memoria físicamente distribuida. En el resto del capítulo supondremos que la replicación de datos sólo se realiza en las cachés, y que la coherencia se mantiene mediante hardware con una granularidad de bloque de caché al igual que en el caso de máquinas basadas en bus. Dado que la memoria principal se encuentra físicamente distribuida y tiene un coste de acceso no uniforme para un procesador, a las arquitecturas de este tipo se les suelen denominar arquitecturas CC-NUMA (*cache-coherent, nonuniform memory access*) o de forma más general DSM (*distributed shared memory*).

Cualquier aproximación a la coherencia, incluyendo la coherencia *snoopy*, debe proporcionar ciertos mecanismos críticos. En primer lugar, un bloque puede estar en cada caché en uno de los posibles estados, potencialmente en diferentes estados para diferentes cachés. El protocolo debe proporcionar estos *estados de la caché*, el *diagrama de transición de estados* acorde con las posibles transiciones que el estado de un bloque puede sufrir en cada caché, y el conjunto de *acciones* asociadas con el diagrama de transición de estados. Los protocolos basados en directorio tendrán también un *estado de directorio* para cada bloque, que será el estado del bloque en el directorio. El protocolo puede basarse en la invalidación, la actualización, o ser un híbrido, y los estados estables de la caché son a menudo los mismos (MESI, MSI), sin importar que el sistema esté basado en directorios o en *snoopy*. Las ventajas e inconvenientes de elegir los posibles estados son similares a las discutidas en el capítulo 6. Conceptualmente, el estado de caché de un bloque de memoria es un vector conteniendo su estado en cada una de las cachés del sistema (incluso cuando no está presente en la caché, que puede asimilarse al caso de un bloque presente en estado inválido). El mismo diagrama de transición se utiliza para manejar las copias en las diferentes cachés, aunque el estado actual del bloque en cualquier instante puede ser diferente en diferentes cachés.

Dado un protocolo a nivel de transición entre estados en la caché, un sistema coherente debe proporcionar los mecanismos para manejar el protocolo. En primer lugar, se necesita un mecanismo para determinar cuando invocar el protocolo. Esto se hace de la misma manera en la mayoría de los sistemas: a través de un mecanismo de detección de fallos de cachés. El protocolo se invoca si el procesador realiza un acceso que no puede satisfacer la caché; por ejemplo, un acceso a un bloque que no está en la caché, o un acceso de escritura a un bloque que está presente en un estado compartido. Sin embargo, incluso cuando se usa el mismo conjunto de estados de la caché, las diferentes alternativas usadas para mantener la coherencia de la caché difieren sustancialmente unos de otros en los mecanismos que proporcionan para las siguientes tres funciones de importancia que pueden ser necesarias ejecutar cuando ocurre un fallo en un acceso:

1. Obtener suficiente información acerca del estado del bloque en las otras cachés para determinar la acción a llevar a cabo,
2. Localizar esas copias, si es necesario, por ejemplo para invalidarlas, y
3. Comunicarse con las otras copias, es decir, obtener datos de las mismas o invalidarlas/actualizarlas.

En los protocolos *snooping*, todas estas funciones se realizaban mediante el mecanismo de *broadcast* y *snooping*. El procesador pone una petición de búsqueda en el bus, conteniendo la dirección del bloque, y el controlador de otra caché recibe la petición y responde en consecuencia. El método de broadcast también puede utilizarse en el caso de máquinas distribuidas; el asistente del nodo que produce el fallo puede enviar un mensaje a todos los demás nodos, y sus asistentes pueden examinar las peticiones y responder de forma adecuada.

Sin embargo, los broadcast no son escalables, dado que generan una gran cantidad de tráfico (p transacciones en la red para cada fallo en una máquina con p nodos). Las aproximaciones escalables incluyen el snooping jerárquico y las aproximaciones basadas en directorio (ver figura 7.1).

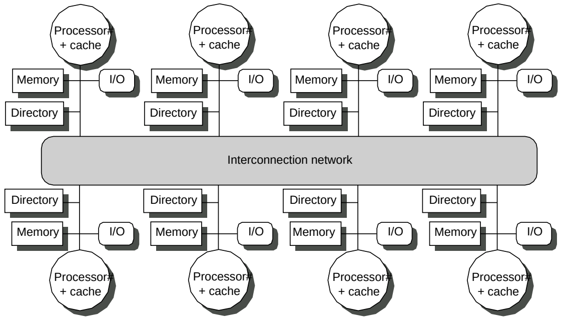


Figura 7.1: Multiprocesador de memoria distribuida con coherencia basada en directorios.

En la aproximación de snooping jerárquico, la red de interconexión no es un único bus (o anillo) sino un árbol de buses. Los procesadores se encuentran en las hojas de ese árbol. Los buses padres se conectan a los hijos mediante un interfaz que rastrea los buses de ambos lados y propagan las transacciones relevantes hacia arriba o hacia abajo en la jerarquía. La memoria principal puede estar centralizada en la raíz o distribuida entre las hojas. En este caso, todas las funciones comentadas arriba se realizan utilizando extensiones jerárquicas del mecanismo de snooping y broadcast: Un procesador realiza una petición de búsqueda en su bus como antes, y esta se propaga por la jerarquía basándose en los resultados de la monitorización de la red.

En la aproximación de directorios, la información acerca del estado de los bloques en las demás cachés se encuentra buscando en el directorio a través de transacciones de la red. La localización de las copias también se encuentra en el directorio, y las copias se comunican mediante transacciones de red punto a punto en una red de interconexión arbitraria, como se muestra en la figura 7.2, sin el uso de broadcast.

En un fallo de lectura, el directorio indica de que nodo podemos obtener los datos, como muestra la figura 7.2(a). En un fallo de escritura, el directorio identifica las copias del bloque, pudiéndose utilizar transacciones de red de invalidación o actualización a dichas copias (figura 7.2(b)). (Recordar que una escritura a un bloque en estado compartido también se considera un fallo de escritura). Dado que las invalidaciones o actualizaciones se envían a varias copias a través de caminos potencialmente diferentes de la red, determinar la finalización de una escritura requiere ahora que todas las copias respondan a las invalidaciones con transacciones de asentimiento explícitas: no podemos suponer la finalización cuando la petición de acceso en modo exclusivo o la actualización ha ganado el acceso a la red de interconexión al no estar en un bus compartido. Las peticiones, respuestas, invalidaciones, actualizaciones y asentimientos entre los nodos son todas transacciones de red con la diferencia de que en este caso el punto final de procesamiento en el destino de la transacción (invalidaciones de datos, obtención y envío de datos) se realiza en el asistente de comunicación en lugar de en el procesador principal. Dado que el esquema de directorios descansa en transacciones punto a punto, pueden utilizarse con cualquier tipo de red de interconexión. En la siguiente sección se verá una visión general de las organizaciones del directorio y protocolos existentes.

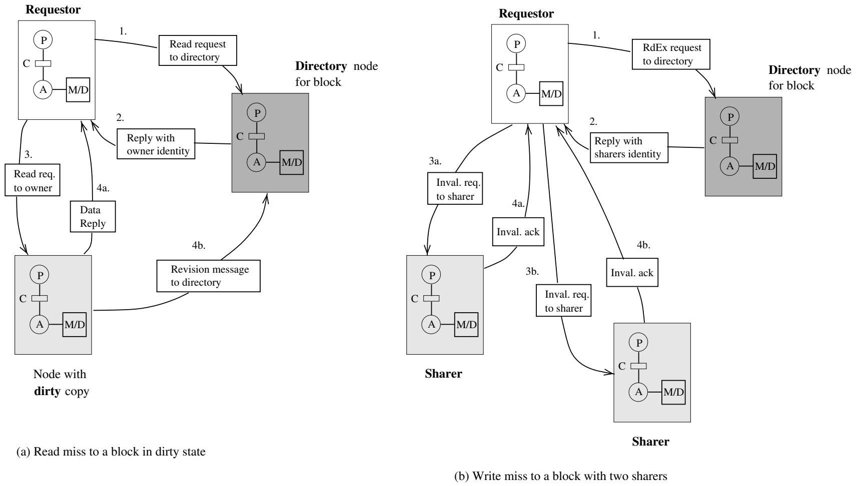


Figura 7.2: Operaciones básicas en un directorio.

7.3 Conceptos básicos de los protocolos basados en directorio

En esta sección describiremos un esquema de directorio simple y como podría operar usando los estados de las cachés, estados del directorio, y transacciones de la red. Después discutiremos las consecuencias al escalar los directorios a un gran número de nodos, proporcionando una clasificación de las organizaciones de directorios escalables, y discutiremos los protocolos básicos asociados con esas organizaciones.

Las siguientes definiciones serán útiles a lo largo de nuestra discusión de los protocolos de directorio. Para una caché o bloque de memoria:

El nodo *origen* (*home*) es el nodo en cuya memoria principal se encuentra el bloque.

El nodo *sucio* (*dirty*) es el un nodo que tiene una copia del bloque es su caché en estado modificado (*dirty*). Obsérvese que el nodo origen y el nodo sucio para un bloque puede ser el mismo.

El nodo *exclusivo* (*exclusive*) es el nodo que tiene una copia del bloque en su caché en un estado exclusivo, ya sea en estado modificado o exclusivo (recuérdese que el estado exclusivo significa que es la única copia de caché válida y que el bloque está actualizado en la memoria principal). Así, el nodo sucio es también el nodo exclusivo.

El nodo *local*, o nodo *peticionario*, es el nodo que contiene el procesador que emite una petición para el bloque.

El nodo *propietario* es el nodo que mantiene en la actualidad la copia válida de un bloque y debe suplir los datos cuando sean necesarios; éste es o bien el nodo origen, cuando el bloque no ha sido modificado, o el nodo sucio.

Los bloques cuyo origen es local al procesador que ha emitido la petición se denominan bloques *localmente asignados* (o simplemente *locales*), mientras que a los otros se les denominan *asignados remotamente* (o *remotos*)

Empecemos con las operaciones básicas de los protocolos basados en directorio usando una organización muy simple.

7.3.1 Operaciones para un esquema de directorio simple

Como se discutió anteriormente, un forma natural de organizar un directorio es mantener la información del directorio para un bloque junto con el bloque en memoria principal, es decir, en el nodo origen del bloque. Una organización simple de la información de directorio para un bloque es un vector de bits. El vector contiene *p bits de presencia* y uno o más bits de estado para una máquina de *p* nodos. Cada bit de presencia representa a un nodo particular (uniprocesador o multiprocesador), y especifica si ese nodo tiene una copia en su caché del bloque. Supongamos por simplicidad que sólo existe un bit de estado, llamado *dirty* bit, que indica si el bloque ha sido modificado en una de las cachés. Por supuesto, si el bit está activado, entonces únicamente un nodo (el nodo *dirty*) puede tener en caché ese bloque y únicamente estará activado el bit de presencia de ese nodo. Con esta estructura, un fallo de lectura puede determinar fácilmente, consultando el directorio, qué nodo, si existe, tiene una copia modificada del bloque o si el bloque es válido en la memoria principal, y un fallo de escritura puede determinar qué nodos están compartiendo el bloque y que, por tanto, tienen que invalidar sus copias.

Junto con este estado en el directorio, también se le asocia un estado a cada bloque que se encuentra en la caché al igual que en el caso de los protocolos snoopy. De hecho, la información del directorio para un bloque es simplemente el punto de vista de la memoria principal del estado de ese bloque en diferentes cachés. El directorio no necesita, obligatoriamente saber el estado concreto (por ejemplo, MESI) en cada caché, sino la información suficiente para determinar las acciones a realizar. De hecho, dado que el directorio y las cachés se comunican a través de una interconexión distribuida, habrá periodos de tiempo en donde el conocimiento del directorio sobre el estado de una caché es incorrecto ya que el estado de la caché se habrá modificado pero la notificación de esa modificación no habrá alcanzado el directorio. Durante ese tiempo el directorio puede enviar un mensaje a la caché basándose en el estado antiguo (que ya no es válido).

Para ver cómo un fallo de lectura o escritura puede interactuar con los vectores de bits que forman una organización en directorio, consideraremos, por simplicidad, un protocolo con tres estados estables (MSI), un único nivel de caché por procesador , y un único procesador por nodo. En un fallo de lectura o escritura en el nodo *i* (incluyendo una actualización en un estado compartido), el asistente de las comunicaciones o controlador del nodo *i*, buscará la dirección del bloque de memoria para determinar si es local o remoto. Si es remoto, se generará una transacción de red al nodo origen de ese bloque. Allí, se consultará la entrada del directorio, y el controlador del nodo origen tratará el fallo como sigue, usando transacciones de red como las que se muestran en la figura 7.2:

- Si el bit *dirty* está a OFF, el controlador obtiene el bloque de la memoria principal, lo envía al nodo peticionario en una transacción de red de respuesta, y poniendo a ON el i-ésimo bit de presencia (*presencia[i]*).
- Si el bit *dirty* está a ON, el controlador responde al peticionario con la identidad del nodo cuyo bit de presencia está a ON, es decir, el propietario o el nodo sucio. El peticionario después envía una transacción de red de petición al nodo propietario. En el propietario, la caché cambia su estado a compartido, y suministra el bloque al nodo peticionario, que guarda el bloque en su caché en estado compartido, y a la memoria principal del nodo origen. En la memoria, el nodo *dirty* se pone a OFF, y *presencia[i]* se pone a ON.

Ante un fallo de escritura por parte de un procesador *i* se maneja como sigue:

- Si el bit *dirty* está a OFF, la memoria principal tiene una copia actualizada de los datos. Se envían transacciones de invalidación a todos los nodos *j* para los cuales *presencia[j]* está a ON. El controlador del directorio espera la respuesta de esos procesadores - indicando que la escritura se ha completado respecto a ellos - y suministra el bloque al nodo *i* donde se pone en la caché en estado modificado. Las entradas del directorio se ponen a OFF, dejando únicamente *presencia[i]* y el bit *dirty* a ON. Si la petición era de una actualización en vez de una lectura exclusiva, se devuelve al peticionario un acuse de recibo en lugar de los datos.
- Si el bit *dirty* está a ON, el bloque se obtiene desde el nodo sucio usando transacciones de red. Esa caché cambia su estado a inválido, y después el bloque es suministrado al procesador peticionario que pone el bloque en la caché en estado modificado. Las entradas del directorio se ponen a OFF, dejando únicamente *presencia[i]* y el bit *dirty* a ON.

En una post-escritura de un bloque modificado por parte del nodo *i*, los datos modificados se escriben en la memoria principal y se actualiza el directorio poniendo a OFF el bit *dirty* y *presencia[i]*. Finalmente, si se reemplaza un bloque que está en estado modificado, se puede enviar un mensaje al directorio para poner a OFF el bit correspondiente del tal manera que no se envíe una invalidación a este nodo la siguiente vez que se escriba el bloque. Este mensaje, se denomina *indicación de reemplazamiento* (*replacement hint*); el que se envíe o no dicho mensaje no afecta a la corrección del protocolo o de la ejecución.

7.3.2 Escalamiento

El principal objetivo de utilizar protocolos de directorio es permitir el escalamiento de la coherencia de la caché más allá del número de procesadores que pueden compartir un bus. Los aspectos más importantes para los protocolos de directorio incluyen tanto el rendimiento como la sobrecarga de almacenar la información del directorio. Dado un sistema con memoria distribuida e interconectado, los factores que más influyen en el rendimiento son cómo escalan la latencia y el ancho de banda demandado que proporciona el sistema al escalar el protocolo mediante el aumento del número de nodos que lo usan. La demanda de ancho de banda viene dado por el número de transacciones generadas por un fallo (multiplicado por la frecuencia de fallo), y la latencia por el número de transacciones que se encuentran en el camino crítico de un fallo (cuantas transacciones tiene que esperar). Estos aspectos del rendimiento se ven afectados, como veremos posteriormente, tanto por la organización del directorio como por lo bien que esté optimizado el flujo de transacciones de red en el protocolo (para una organización dada). El almacenamiento, sin embargo, sólo se ve afectado por la organización de la información del directorio. Para una organización mediante un vector de bits, el número de bits de presencia necesarios aumenta de forma lineal con el número de procesadores (*p* bits por bloque de memoria) y la cantidad de memoria principal (un vector por bloque de memoria), dando lugar a una sobrecarga potencialmente alta para el directorio. Con un tamaño de bloque de 64-bytes y 64 procesadores, la sobrecarga del directorio es

$$\frac{vector\ size}{block\ size} = \frac{64\ bits}{64\ bytes} = \frac{1}{8} = 0.125$$

ó 12.5%. Con 256 procesadores y el mismo tamaño de bloque, la sobrecarga es ya del 50%, y para 1024 procesadores es del 200%. La sobrecarga del directorio no escala bien, pero puede ser aceptable si el número de nodos visibles por el directorio en una máquina dada no es muy alto.

Afortunadamente, hay otras maneras de organizar la información del directorio para mejorar su escalabilidad. Estas organizaciones dan lugar a protocolos de alto nivel con diferentes características de rendimiento. El resto de esta sección la dedicaremos a exponer las diferentes organizaciones de directorios existentes y a describir brevemente como los fallos de lectura y escritura pueden implementarse de forma directa mediante un protocolo del tipo petición-respuesta. Al igual que en el caso anterior, consideraremos que no existe otro fallo de caché al mismo tiempo, no existiendo condiciones competitivas, y que por lo tanto el directorio y las cachés siempre se encuentran en estados estables.

7.3.3 Alternativas en la organización de los directorios

A alto nivel, los esquemas de directorio se diferencian por los mecanismos que proporcionan para las tres funciones de coherencia discutidas en la sección 7.2. Dado que las comunicaciones con las copias de caché se realizan siempre a través de transacciones de red, la diferencia real está en las dos primeras funciones: encontrar la información en el directorio al ocurrir un fallo, y determinar la localización de las copias relevantes.

Existen dos alternativas principales para encontrar la información de un bloque:

- *Esquemas de directorios planos*, y
- *Esquemas de directorios jerárquicos*.

El esquema que hemos descrito anteriormente es un esquema plano. Los esquema planos (*flat schemes*) se denominan así porque la información del directorio para un bloque se encuentra en un lugar fijo, normalmente en el nodo origen determinado por la dirección del bloque, y ante un fallo se envía una transacción de red directamente al nodo fuente para buscar en el directorio. En los esquemas jerárquicos, la memoria está también distribuida entre los procesadores, pero la información del directorio está lógicamente organizada en una estructura de datos jerárquica (un árbol). Los nodos de procesamiento, cada uno con su porción de memoria, son las hojas del árbol. Los nodos internos del árbol son simplemente información de directorio mantenida jerárquicamente para el bloque: Un nodo mantiene un seguimiento de si alguno de sus hijos tiene una copia de un bloque. Al producirse un fallo, la información de directorio para ese bloque se encuentra subiendo en la jerarquía nivel a nivel mediante transacciones de red hasta que se encuentra un nodo de directorio cuyo subárbol tenga el bloque en el estado apropiado. Así, un procesador que falla envía simplemente un mensaje de búsqueda a su padre, y así sucesivamente, más que dirigirse al nodo origen de ese bloque con una única transacción de red. El árbol de directorios para un bloque es lógico, no necesariamente físico, y puede estar embebido en una red de interconexión general. De hecho, en la práctica cada nodo de procesamiento del sistema

no es únicamente un nodo hoja para los bloques que contienen sino que también guarda información del directorio como un nodo interno del árbol para otros bloques.

En el caso jerárquico, la información acerca de las localizaciones de las copias también se mantiene a través de la propia jerarquía; las copias se encuentran y se comunican entre ellas atravesando la jerarquía guiado por la información del directorio.

En los esquemas planos, el cómo se almacena la información sobre las copias varía considerablemente. A nivel superior, los esquemas planos pueden dividirse en dos clases:

- Esquemas *basados en memoria*, en donde toda la información del directorio acerca de todas las copias de caché están almacenadas en el nodo origen del bloque. El esquema de vector de bits descrito anteriormente es de este tipo. Las localizaciones de todas las copias se descubren en el nodo origen, y pueden ser comunicadas directamente a través de mensajes punto a punto.
- Esquemas *basados en caché*, en donde la información de las copias de caché no está totalmente almacenada en el nodo origen sino distribuido entre las propias copias. El nodo origen simplemente contiene un puntero a la copia del bloque. Cada copia almacenada en la caché contiene un puntero al (o un identificador del) nodo que tiene la siguiente copia almacenada del bloque, en una organización de lista enlazada distribuida. Las localizaciones de las copias se determinan atravesando esta lista mediante transacciones de la red.

La figura 7.3 resume la taxonomía. Los directorios jerárquicos tienen algunas ventajas potenciales. Por ejemplo, un fallo de lectura a un bloque cuyo origen se encuentra lejano en la topología de la red de interconexión puede satisfacerse antes enviando al procesador otra copia más cercana en la topología, en lugar de tener que recorrer todo el camino hasta el nodo origen. Además, las peticiones de varios nodos pueden ser combinadas en un antecesor común en la jerarquía, enviándose una única petición a partir de ese punto. Estas ventajas van a depender de lo bien que se amolde la jerarquía lógica a la topología física subyacente. Sin embargo, en lugar de unas pocas transacciones punto a punto para satisfacer un fallo en muchos de los sistemas planos, el *número* de transacciones de red necesarias para atravesar la jerarquía arriba y abajo puede ser mucho mayor, y cada transacción necesita consultar la información del directorio para determinar su nodo destino. Como resultado, los requisitos de latencia y ancho de banda para los esquemas de directorio jerárquicos tienden a ser mucho mayores, estas organizaciones no son populares entre los sistemas modernos. El resto de esta sección la dedicaremos a examinar los esquemas de directorios planos, tanto basados en memoria como en caché, observando las organizaciones del directorio, la sobrecarga en el almacenamiento, la estructura de los protocolos, y el impacto en las características que miden el rendimiento.

7.3.3.1 Esquemas de directorio plano basado en memoria

La organización de vector de bit vista anteriormente, denomina organización *full bit vector*, es la manera más directa de almacenar la información del directorio en un esquema de directorio basado en memoria. Consideremos sus características de rendimiento en las escrituras. Dado que este protocolo preserva la información de los nodos que comparten un bloque de forma precisa y en el nodo origen, el número de transacciones por invalidación crece únicamente con el número de nodos que comparten el bloque. Dado que la identidad de todos ellos está disponible en el nodo origen, los mensajes de invalidación se pueden solapar e incluso enviarse en paralelo; en este caso, el número transacciones

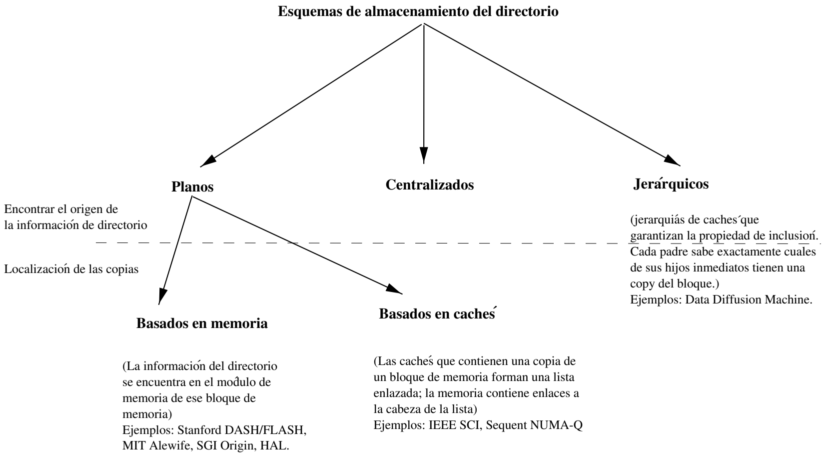


Figura 7.3: Alternativas para guardar la información del directorio.

en el camino crítico deja de ser proporcional al número de nodos que comparten el bloque, reduciéndose la latencia. La principal desventaja, como se discutió anteriormente, es la sobrecarga de almacenamiento.

Existen dos formas de reducir la sobrecarga en los esquemas *full bit vector*. La primera es incrementar el tamaño de los bloques en la caché. La segunda es poner varios procesadores en un nodo; es decir, usar un protocolo de dos niveles. La máquina DASH de Stanford usa un esquema de vector de bit completo, siendo cada uno de sus nodos un multiprocesador basado en bus formado por 4 procesadores. Estos dos métodos hacen de los directorios basados en vectores de bit una alternativa muy atractiva incluso para máquinas bastante grandes. Por ejemplo, para una máquina de 256 procesadores usando nodos de 4 procesadores y tamaño de bloque de caché de 128 bytes, la sobrecarga del directorio es únicamente del 6.25%.

Sin embargo, estos métodos reducen la sobrecarga únicamente por un factor constante. La sobrecarga total de almacenamiento es todavía proporcional a $P \times M$, donde *P* es el número de nodos de proceso y M es el número total de bloques de memoria en la máquina ($M = P * m$, donde m es el número de bloques por memoria local), convirtiéndose en intolerable para máquinas muy grandes. La sobrecarga del directorio se puede reducir más si nos centramos en los factores de la expresión $P \times M$. Podemos reducir la *anchura* de cada entrada del directorio no dejando que esta crezca de forma proporcional a *P*. O podemos reducir el número total de entradas en el directorio, o la *altura* del directorio, no teniendo una entrada por bloque de memoria.

La anchura del directorio se puede reducir usando directorios con *apuntadores limitados*, que están motivados por la observación de que la mayor parte del tiempo sólo unas pocas cachés tienen una copia de un bloque cuando se escribe en dicho bloque. Los esquemas de apuntadores limitados no guardan un si o un no por cada nodo, sino que mantienen un número fijo de punteros, digamos *i*, cada uno apuntando a un nodo que tiene una copia del bloque. Cada puntero necesita $log\ P$ bits para

guardar la información de *P* nodos, pero el número de punteros usados es pequeño. Por ejemplo, para una máquina con 1024 nodos, cada puntero necesita 10 bits, con lo que incluso teniendo 100 punteros se usa menos almacenamiento que en un esquema de vector de bits completo. En la practica, cinco o menos punteros parecen suficientes. Por supuesto, estos esquemas necesitan alguna clase de cobertura o estrategia de desbordamiento para la situación en donde existen más de *i* copias. Una estrategia puede ser la utilización de *broadcast* para invalidar los bloques en el caso de que existan más de *i* copias. Se pueden desarrollar muchas otras estrategias para evitar el broadcast incluso en estos casos. La diferencia entre los diferentes esquemas de este tipo estriba principalmente en sus estrategias en el caso de desbordamiento y el número de punteros usados.

La altura del directorio se puede reducir organizando el directorio como una caché, aprovechándonos del hecho de que la cantidad total de la caché en la máquina es mucho menor que la memoria total, sólo una fracción muy pequeña de los bloques de la memoria estarán replicados en las cachés en un momento dado, así la mayor parte de las entradas del directorio permanecerán sin usar.

A pesar de estas optimizaciones, las operaciones básicas para encontrar las copias y comunicarse con ellas son las mismas para los diferentes esquemas basados en memoria. La identidad de los nodos que comparten un bloque se mantiene en el nodo origen y, al menos en el caso en el que no hay desbordamiento, la comunicación con las copias se realiza mediante transacciones punto a punto.

7.3.3.2 Esquemas de directorio plano basados en caché

En los esquemas de directorio plano basados en caché existe todavía una memoria principal origen de cada bloque; sin embargo, la entrada del directorio en el nodo origen no contiene la identidad de todos los nodos que comparten el bloque, sino que sólo tiene un nodo al primero de los nodos más unos pocos bits de estado. A este puntero se le denomina *puntero cabecera* del bloque. El resto de los nodos que tienen ese bloque se encuentran unidos mediante una *lista doblemente enlazada distribuida*, usando punteros adicionales que están asociados con cada línea de caché en un nodo. Es decir, si una caché contiene una copia de un bloque, también contiene los punteros a la siguiente y anterior caché que tienen una copia.

En un fallo de lectura, el nodo peticionario envía una transacción de red a la memoria origen de ese bloque para determinar la identidad del nodo cabecera en la lista enlazada, si existe, para ese bloque. Si el puntero cabecera está a *null* (no se está compartiendo) la memoria origen devuelve los datos. Si el puntero cabecera no es nulo, el nodo peticionario debe añadirse a la lista de nodos que comparten ese bloque. El nodo origen responde al peticionario con el puntero cabecera. El peticionario envía entonces un mensaje al nodo cabecera, pidiéndole que le inserte a la cabeza de la lista y, por tanto, convertirse el nuevo nodo cabecera. Los datos de ese bloque son proporcionados por el nodo origen si tiene la última copia o, en caso contrario, por el nodo cabecera que siempre tiene la última copia (es el nodo propietario).

En un fallo de escritura, el nodo obtiene la identidad del nodo cabecera, si existe, del nodo origen. Después se inserta en la lista como nodo cabecera al igual que en el caso anterior (si el nodo ya se encontraba en la lista como nodo que compartía el bloque, lo que se está realizando ahora es una actualización, éste se elimina de su posición actual en la lista y se inserta como nueva cabecera). Después de esto, el resto de la lista distribuida se atraviesa nodo a nodo vía transacciones de red para encontrar e invalidar sucesivas copias del bloque. Si un bloque que ha de ser escrito está compartido por tres nodos A, B y C, el nodo origen sólo conoce al nodo A de tal manera que el nodo que modifica el bloque le envía un mensaje; la identidad del siguiente nodo sólo se sabe una vez que se accede a

A, y así sucesivamente. Los acuses de recibo de estas invalidaciones se envían al nodo que realiza la escritura. De nuevo, si se necesita el bloque, este es proporcionado por el nodo origen o por el nodo cabecera. El número de mensajes por invalidación -el ancho de banda demandado- es proporcional al número de nodos que comparten el bloque como en los esquemas basados en memoria, pero ahora también lo es el número de mensajes en el camino crítico, es decir, la latencia. Cada uno de estos mensajes serializados invocan al asistente de comunicación e incrementan aún más la latencia. De hecho, incluso en un fallo de lectura a un bloque sin modificar se ven envueltos los controladores de múltiples nodos para insertar el nodo en la lista enlazada.

Las actualizaciones de la memoria principal (*writeback*) o en el reemplazo de la caché también es necesario que el nodo se elimine de la lista compartida, lo que significa comunicarse con los nodos que se encuentran antes y después en la lista. Esto es necesario ya que el nuevo bloque que reemplaza al antiguo necesitará los punteros al nodo anterior y al nodo siguiente en la entrada de la caché para su propia lista compartida.

Para contrarestar la desventaja de la latencia, los esquemas basados en caché tienen algunas importantes ventajas sobre los esquemas basados en memoria. En primer lugar, la sobrecarga del directorio no es tan grande. Cada bloque de la memoria principal tiene un único puntero cabecera. El número de punteros siguientes y anteriores es proporcional al numero de bloques de caché en la máquina, los cuales son muchos menos que el números de bloques de memoria. La segunda ventaja es que una lista enlazada guarda el orden en el cual los accesos a memoria se han producido para un bloque, haciendo más fácil proporcionar una equidad entre los accesos y evitar el problema del *live-lock* en un protocolo. En tercer lugar, el trabajo a realizar por los controladores al enviar invalidaciones no está centralizado en el nodo origen sino distribuido entre los nodos que comparten el bloque, diseminando la ocupación de los controladores y reduciendo la necesidad de ancho de banda en el nodo origen.

La inserción y eliminación de nodos en las listas enlazadas puede dar lugar a implementaciones complejas del protocolo. Por ejemplo, la eliminación de un nodo de una lista compartida requiere una coordinación cuidadosa y una exclusión mutua con el procesador anterior y posterior en la lista enlazada, dado que estos procesadores pueden estar intentando reemplazar el mismo bloque de forma concurrente. Estos asuntos complejos se han visto reducidos enormemente con la formalización y publicación de un estándar para la organización del directorio basado en cachés así como del protocolo para su manejo: el estándar de IEEE 1596-1992 *Scalable Coherence Interface* (SCI). El estándar incluye una especificación completa y el código C para el protocolo. Varias máquinas comerciales han empezado a utilizar este protocolo (por ejemplo, Sequent NUMA-Q, Convex Exemplar y Data General), y se han explorado diferentes variantes que utilizan representaciones alternativas de la lista (listas simplemente enlazadas en lugar de doblemente enlazadas).

7.3.4 Rendimiento de los protocolos basados en directorios

El rendimiento de una máquina basada en directorio depende de muchos de los factores que influían en el rendimiento de las máquinas basadas en bus (tamaño de la caché, número de procesadores y tamaño del bloque), como de la distribución de los fallos en la jerarquía de la memoria. La posición de los datos demandados depende tanto de la localización inicial como de los patrones de compartición. Empezaremos examinando el rendimiento básico de la caché usando la carga de programas paralelos que utilizamos en la sección 6.6, para después determinar los efectos de los diferentes tipos de fallos.

Dado que las máquinas son más grandes y las latencias mayores que en el caso de multiprocesadores basados en snooping, empezaremos con una caché mayor (128 KB) y un bloque de 64 bytes. En las

arquitecturas de memoria distribuida, la distribución de las peticiones entre locales y remotas es el factor clave para el rendimiento, ya que ésta afecta tanto al consumo del ancho de banda global como a la latencia vista por el nodo que realiza la petición. Por tanto, separaremos en las figuras de esta sección los fallos de caché en locales y remotos. Al mirar estas figuras hay que tener en cuenta que, para estas aplicaciones, la mayoría de los fallos remotos se producen debido a fallos de coherencia, aunque algunos fallos de capacidad también puede ser remotos, y en algunas aplicaciones con una distribución de datos pobre, estos fallos pueden ser significativos.

Como muestra la figura 7.4, la tasa de fallos con estas cachés no se ve afectada por el cambio en el número de procesadores, con la excepción de Ocean, donde la tasa de fallos aumenta con 64 procesadores. Este aumento se debe al aumento de conflictos en el mapeo de la caché que ocurren cuando se reduce el tamaño de la malla, dando lugar a un aumento de los fallos locales, y al aumento de los fallos de coherencia, que son todos remotos.

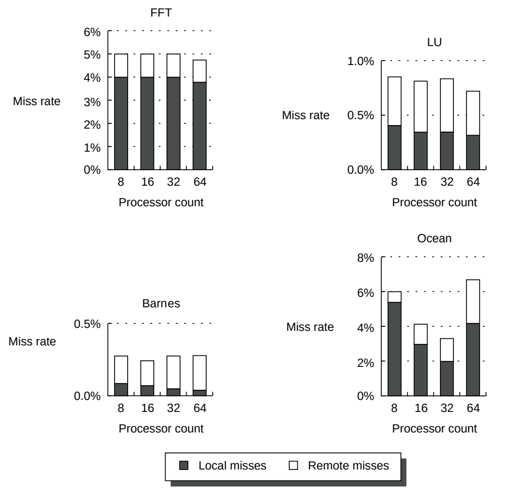


Figura 7.4: Tasa de fallos en función del número de procesadores.

La figura 7.5 muestra como cambia la tasa de fallos al incrementarse el tamaño de la caché, suponiendo una ejecución con 64 procesadores y bloques de 64 bytes. Estas tasas de fallos descienden tal como cabría esperar, aunque el efecto amortiguador causado por la casi nula reducción de los fallos de coherencia dan lugar a un menor decrecimiento de los fallos remotos que de los fallos locales. Para una caché de 512 KB, la tasa de fallos remota es igual a o mayor que la tasa de fallos local. Tamaños de caché mayores sólo darán lugar a amplificar este fenómeno.

Finalmente, la figura 7.6 muestra el efecto de cambiar el tamaño del bloque. Dado que estas aplicaciones tienen buena localidad espacial, el incrementar el tamaño del bloque reduce la latencia de fallo, incluso para bloques grandes, aunque los beneficios en el rendimiento de utilizar los bloques de mayor tamaño son pequeños. Además, la mayor parte de la mejora en la tasa de fallos se debe a los

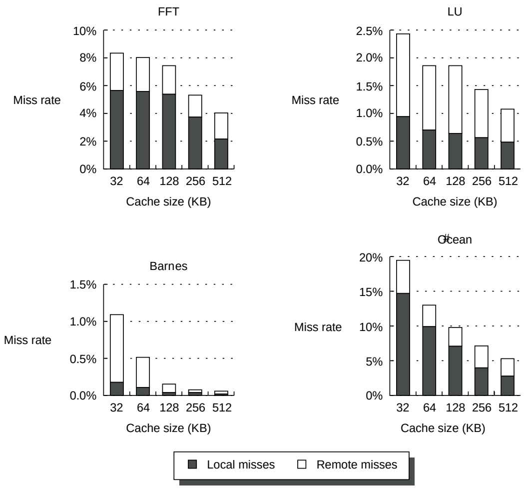


Figura 7.5: Tasa de fallos en función del tamaño de la caché.

fallos locales.

Más que mostrar el tráfico de la memoria, la figura 7.7 muestra el número de bytes necesarios por referencia de datos en función del tamaño del bloque, dividiendo los requisitos de ancho de banda en local y global. En el caso de un bus, podemos unir ambos factores para determinar la demanda total para el bus y el ancho de banda de la memoria. Para una red de interconexión escalable, podemos usar los datos de la figura 7.7 para determinar el ancho de banda global por nodo y estimar el ancho de banda de la bisección, como muestra el siguiente ejemplo.

Ejemplo Supongamos un multiprocesador de 64 procesadores a 200 MHz con una referencia a memoria por ciclo de reloj. Para un tamaño de bloque de 64 bytes, la tasa de fallos remota es del 0.7%. Determinar el ancho de banda por nodo y el ancho de banda estimado para la bisección para FFT. Suponer que el procesador no se para ante peticiones a memoria remota; esto puede considerarse cierto si, por ejemplo, todos los datos remotos son precargados.

Respuesta El ancho de banda por nodo es simplemente el número de bytes por referencia multiplicado por la tasa de referencias: $0.007 \times 200 \times 64 = 90\ MB/sec$. Esta tasa es aproximadamente la mitad del ancho de banda del más rápido MPP escalable disponible en 1995. La demanda de ancho de banda por nodo excede a las redes de interconexión ATM más rápidas existentes en 1995 por un factor de 5, y excederá ligeramente a la siguiente generación de ATM

FFT realiza comunicaciones del tipo todos a todos, así que el ancho de banda de la bisección será igual a 32 veces el ancho de banda por nodo, o $2880\ MB/sec$. Para una

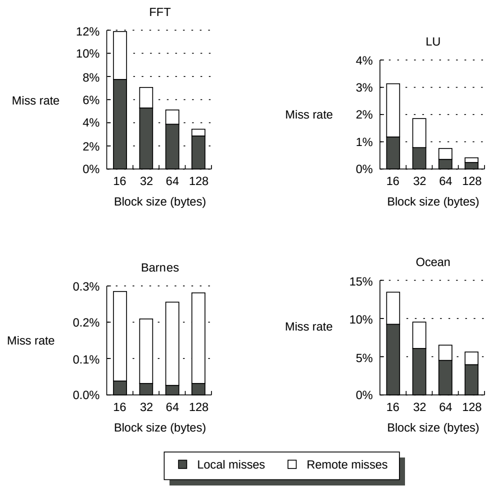


Figura 7.6: Tasa de fallos en función del tamaño del bloque suponiendo una caché de 128 KB y 64 procesadores.

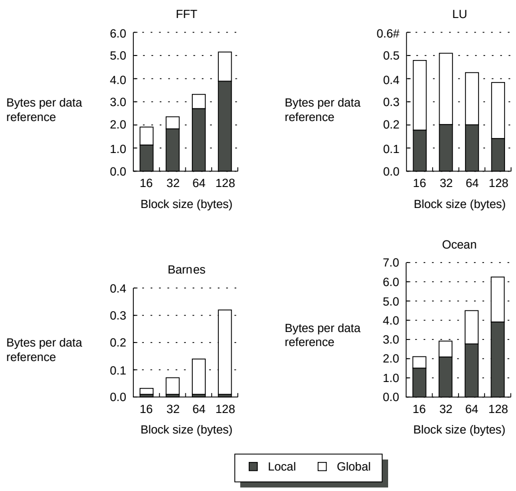


Figura 7.7: Número de bytes por referencia en función del tamaño del bloque.

máquina de 64 procesadores conectada mediante una malla 2D, el ancho de banda de la bisección crece proporcionalmente a la raíz cuadrada del número de procesadores. Así, para 64 procesadores el ancho de banda de la bisección es 8 veces el ancho de banda del nodo. En 1995, las interconexiones del tipo MPP ofrecen cerca de 200 MB/sec por nodo para un total de 1600 MB/sec. Para 64 nodos una conexión del tipo malla 3D dobla este ancho de banda de la bisección (3200 MB/sec), que excede el ancho de banda necesario. La siguiente generación de mallas 2D también conseguirán alcanzar el ancho de banda necesario.

El ejemplo previo muestra la demanda del ancho de banda. Otro factor clave para los programas paralelos es el tiempo de acceso a memoria remota, o latencia. Para examinar este factor usaremos un ejemplo sobre una máquina basada en directorio. La figura 7.8 muestra los parámetros que supondremos para nuestra máquina. Supondremos que el tiempo de acceso para la primera palabra es de 25 ciclos y que la conexión a la memoria local tiene un ancho de 8 bytes, mientras que la red de interconexión tiene una anchura de 2 bytes. Este modelo ignora el efecto de la contención, que probablemente no es muy serio en los programas paralelos examinados, con la posible excepción de FFT, que usa comunicaciones del tipo todos a todos. La contención podría tener un serio impacto en el rendimiento para otro tipo de cargas de trabajo.

Characteristic	**Number of processor clock cycles**
Cache hit	1
Cache miss to local memory	$25 + \frac{\text{block size in bytes}}{8}$
Cache miss to remote home directory	$75 + \frac{\text{block size in bytes}}{2}$
Cache miss to remotely cached data (3-hop miss)	$100 + \frac{\text{block size in bytes}}{2}$

Figura 7.8: Características de la máquina basada en directorio del ejemplo.

La figura 7.9 muestra el coste en ciclos para una referencia a memoria, suponiendo los parametros de la figura 7.8. Sólo se contabilizan las latencias de cada tipo de referencia. Cada barra indica la contribución de los éxitos de caché, fallos locales, fallos remotos y fallos remotos a una distancia de tres saltos. El coste está influenciado por la frecuencia total de los fallos de caché y actualizaciones, al igual que por la distribución de las localizaciones donde se satisface el fallo. El coste de una referencia a memoria remota permanece estable al aumentar el número de procesadores, excepto para Ocean. El incremento en la tasa de fallos en Ocean para 64 procesadores está claro a partir de la figura 7.4. Al aumentar la tasa de fallos, cabría esperar que el tiempo gastado en las referencias a memoria se incremente también.

Aunque la figura 7.9 muestra el coste de acceso a la memoria, que es el coste dominante en estos programas, un modelo completo del rendimiento debería considerar el efecto de la contención en el sistema de memoria, al igual que las pérdidas ocasionadas por los retrasos en la sincronización.

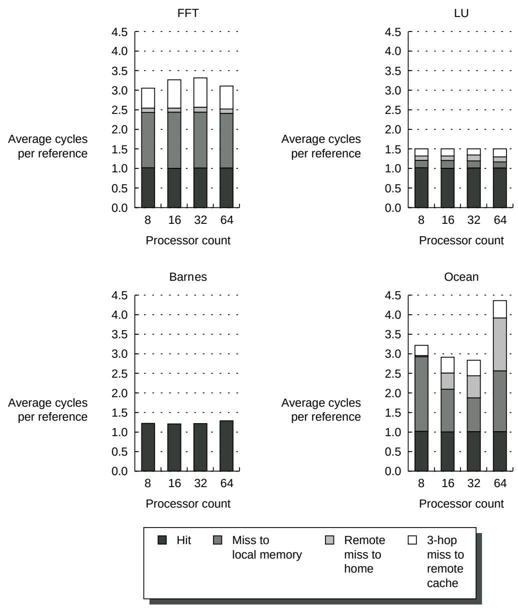


Figura 7.9: La latencia efectiva de las referencias a memoria en una máquina DSM depende de la frecuencia relativa de los fallos de caché y de la localización de la memoria desde donde se producen los accesos.

7.4 Sincronización

En el caso de los eventos de exclusión mutua y de punto a punto, los conceptos son esencialmente los mismos que se discutieron en el capítulo anterior. Al igual que en el caso de máquinas de memoria compartida de pequeña escala, la meta en las máquinas escalables es construir operaciones de sincronización de alto nivel (como los cerrojos y las barreras) en software y basándose en un conjunto de primitivas atómicas. Existen, sin embargo, dos diferencias importantes que pueden afectar a la elección de los algoritmos. En primer lugar, la red de interconexión no está centralizada sino que tiene varios caminos paralelos. Por una parte, esto significa que conjuntos disjuntos de procesadores pueden coordinarse entre ellos en paralelo sobre caminos enteramente disjuntos; por otra parte, este hecho puede complicar la implementación de las primitivas de sincronización. En segundo lugar, la memoria físicamente distribuida hace que sea importante la localización de las variables de sincronización. La importancia de este segundo efecto depende de si las cachés permiten el almacenamiento de datos compartidos no locales, y este efecto es claramente superior en el caso de máquinas que no lo permiten. En esta sección describiremos nuevos algoritmos para cerrojos y barreras que son apropiados para máquinas con memoria físicamente distribuida e interconectada, empezando desde los algoritmos que discutimos en la sección 6.7.

7.4.1 Algoritmos para cerrojos

En la sección 6.7 se discutieron los distintos tipos de cerrojos existentes. Cada uno de los cuales daba un paso más en la reducción del tráfico del bus y de la equidad, pero a menudo a costa de una mayor sobrecarga. Por ejemplo, el *ticket lock* permite que únicamente un procesador pida el cerrojo cuando este se libera, pero la notificación de la liberación llega a todos los procesadores a través de una invalidación y el posterior fallo de lectura. El cerrojo basado en array soluciona este problema haciendo que cada proceso espere sobre diferentes localizaciones, y el procesador que libera el cerrojo únicamente notifica la liberación a un procesador escribiendo en la localización correspondiente.

Sin embargo, los cerrojos basados en array tienen dos problemas potenciales en máquinas escalables con memoria física distribuida. En primer lugar, cada cerrojo requiere un espacio proporcional al número de proesadores. En segundo lugar, y más importante para las máquinas que no permiten tener en caché datos remotos, no hay forma de saber por anticipado sobre que localización iterará un proceso concreto, dado que dicha localización se determina en tiempo de ejecución a través de una operación *fetch&increment*. Esto hace imposible almacenar las variables de sincronización de manera que la variable sobre la cual itera un proceso esté siempre en su memoria local (de hecho, todos los cerrojos del capítulo anterior tienen este problema). En máquinas de memoria distribuida sin coherencia de cachés, como la Cray T3D y T3E, éste es un gran problema ya que los procesos iterarían sobre localizaciones remotas, dando lugar a una gran cantidad de tráfico y contención. Afortunadamente, existe un algoritmo software que reduce la necesidad de espacio al tiempo que asegura que todas las iteraciones se realizarán sobre variables almacenadas localmente.

Software Queuing Lock

Este cerrojo es una implementación software de un cerrojo totalmente hardware propuesto inicialmente en el proyecto Wisconsin Multicube. La idea es tener una lista distribuida o cola de procesos en espera de la liberación del cerrojo. El nodo cabecera de la lista representa el proceso que tiene

el cerrojo. Los restantes nodos son procesos que están a la espera, y están en la memoria local del proceso. Un nodo apunta al proceso (nodo) que ha intentado adquirir el cerrojo justo después de él. También existe un puntero cola que apunta al último nodo de la cola, es decir, el último nodo que ha intentado adquirir el cerrojo. Veamos de una manera gráfica como cambia la cola cuando los procesos adquieren y liberan el cerrojo.

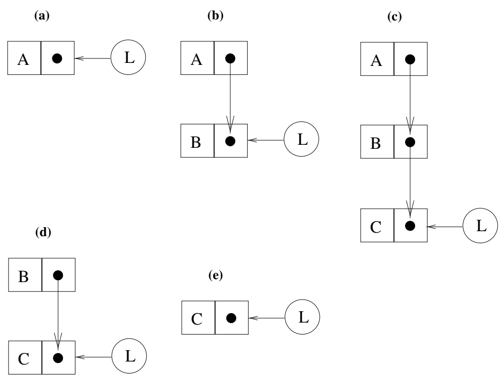


Figura 7.10: Estados de la lista para un cerrojo cuando los procesos intentan adquirirlo y cuando lo liberan.

Supongamos que el cerrojo de la figura 7.10 está inicialmente libre. Cuando un proceso A intenta adquirir el cerrojo, lo obtiene y la cola quedaría como se muestra en la figura 7.10(a). En el paso (b), el proceso B intenta adquirir el cerrojo, por lo que se pone a la cola y el puntero cola ahora apunta a él. El proceso C es tratado de forma similar cuando intenta adquirir el cerrojo en el paso (c). B y C están ahora iterando sobre las variables locales asociadas con sus nodos en la cola mientras que A mantiene el cerrojo. En el paso (d), el proceso A libera el cerrojo. Después “despierta” al siguiente proceso, B, de la cola escribiendo en la variable asociada al nodo B, y dejando la cola. B tiene ahora el cerrojo, y es la cabeza de la cola. El puntero cola no cambia. En el paso (e), B libera el cerrojo de manera similar, pasándoselo a C. Si C libera el cerrojo antes de que otro proceso intente adquirirlo, el puntero al cerrojo será NULL y el cerrojo estará otra vez libre. De esta manera, los procesos tienen garantizado un acceso al cerrojo en una ordenación FIFO con respecto al orden en el que intentaron adquirirlo.

Debe de quedar claro que en el cerrojo por cola software el espacio necesario es proporcional al número de procesos que están esperando o participando en el cerrojo, no al número de procesos en el programa. Este sería el cerrojo que se elegiría para máquinas que soportan un espacio de memoria compartido con memoria distribuida pero sin coherencia de cachés.

7.4.2 Algoritmos para barreras

El problema de eventos globales como las barreras es una preocupación clave en las máquinas escalables. Una cuestión de considerable debate es si es necesario un soporte hardware especial para las operaciones globales o si son suficientes los algoritmos software. El CM-5 representa una parte del espectro, con una red de "control" especial que proporciona barreras, reducciones, broadcasts y otras operaciones globales sobre un subárbol de la máquina. Cray T3D proporciona soporte hardware para las barreras, mientras que T3E no lo hace. Dado que es sencillo construir barreras que iteren sobre variables locales, la mayoría de las máquinas escalables no proporcionan un soporte hardware especial para las barreras sino que se construyen en librerías software usando primitivas de intercambio o LL-SC.

En la barrera centralizada usada en la máquinas basadas en bus, todos los procesadores usan el mismo cerrojo para incrementar el mismo contador cuando indican su llegada a la barrera, y esperan sobre la misma variable de flag hasta que son liberados. En una máquina más grande, la necesidad de que todos los procesadores accedan al mismo cerrojo y lean y escriban de las mismas variables puede dar lugar a mucho tráfico y contención. De nuevo, esto es particularmente cierto en el caso de máquinas sin coherencia de cachés, donde la variable se puede convertir en un punto caliente de la red cuando varios procesadores iteran sobre ella.

Es posible implementar la llegada y la salida de la barrera de una manera más distribuida, en donde no todos los procesadores tienen que acceder a la misma variable o cerrojo. La coordinación de la llegada o liberación puede realizarse en fases o rondas, con subconjuntos de procesos que se coordinan entre ellos, de tal manera que después de algunas fases todos los procesos están sincronizados. La coordinación de diferentes subconjuntos puede realizarse en paralelo ya que no es necesario mantener ninguna serialización entre ellos. En una máquina basada en bus, distribuir las acciones de coordinación no importa mucho ya que el bus serializa todas las acciones que requieren comunicación; sin embargo, puede ser muy importante en máquinas con memoria distribuida e interconectada donde diferentes subconjuntos pueden coordinarse en diferentes partes de la red. Veamos algunos de estos algoritmos distribuidos.

Árboles de combinación software

Una manera sencilla de coordinar la llegada o partida es a través de una estructura de árbol. Un árbol de llegada es una árbol que usan los procesadores para indicar su llegada a la barrera. Se reemplaza un único cerrojo y contador por un árbol de contadores. El árbol puede ser de cualquier tipo, por ejemplo k-ario. En el caso más simple, cada hoja del árbol es un proceso que participa en la barrera. Cuando un proceso llega a la barrera, indica su llegada realizando una operación *fetch&increment* sobre el contador asociado a su padre. Después chequea que el valor devuelto por la operación para comprobar si es el último de los hermanos en llegar. Si no, simplemente espera a la liberación. En caso contrario, el proceso pasa a ser el representante de sus hermanos en el siguiente nivel del árbol y realiza una operación *fetch&increment* sobre el contador de ese nivel. De esta manera, cada nodo del árbol envía un único representante al primer nivel del árbol donde todos los procesos están representados por ese nodo hijo que ha llegado. Para un árbol de grado k, será necesario pasar por $log_k p$ niveles para completar la notificación de llegada de los p procesos. Si los subárboles de procesadores están repartidos en diferentes partes de la red, y si las variables de los nodos del árbol están distribuidas de manera apropiada en la memoria, las operaciones *fetch&increment* sobre nodos que no tienen relación ascendiente-descendiente no necesitan serialización.

Se puede utilizar una estructura de árbol similar para la liberación, de tal manera que los procesadores no realicen una espera activa sobre la misma variable. Es decir, el último proceso en llegar a la barrera pone a uno la variable de liberación asociada con la raíz del árbol, sobre la cual están realizando una espera activa $k-1$ procesos. Cada uno de los k procesos pone a uno la variable de liberación del

siguiente nivel del árbol sobre la que están realizando espera activa $k-1$ procesos, y así bajando en el árbol hasta que todos los procesos se liberan. La longitud del camino crítico de la barrera en términos del número de operaciones dependientes o serializadas (es decir, transacciones de red) es por tanto $O(log_k p)$, frente al $O(p)$ para una la barrera centralizada ó $O(p)$ para cualquier barrera sobre un bus centralizado.

Aunque las barreras en árbol distribuyen el tráfico en la red de interconexión, tienen el mismo problema que un simple cerrojo para máquinas que no cachean los datos remotos compartidos: las variables sobre las que los procesadores iteran no están necesariamente almacenadas en sus memorias locales. Múltiples procesadores iteran sobre la misma variable y qué procesadores alcanzan los niveles superiores del árbol e iteran sobre esas variables depende del orden en el cual los procesadores alcanzn la barrera y realizan sus instrucciones *fetch&increment,* lo que es imposible de predecir. Esto conduce a mucho tráfico en la red mientras se itera.

Barreras en árbol con iteración local

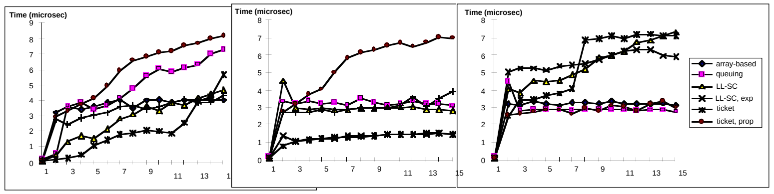
Existen dos formas de asegurarse de que un procesador itere sobre una variable local. Una es predeterminar qué procesador se moverá desde un nodo a su padre en el árbol basándose en el identificador del proceso y el número de procesadores que participan en la barrera. En este caso, un árbol binario facilita la implementación de la iteración local, ya que la variable sobre la que se itera puede almacenarse en la memoria local del proceso en espera en lugar de aquel que pasa al nivel del padre. De hecho, en este caso es posible implementar la barrera sin operaciones atómicas, como *fetch&increment,* usando únicamente operaciones de lectura y escritura como se indica a continuación. En la llegada, un proceso que llega al nodo simplemente itera sobre la variable de llegada asociada a ese nodo. El otro proceso asociado con el nodo simplemente escribe en la variable al llegar. El proceso cuyo papel sea iterar ahora simplemente itera sobre la variable liberada asociada a ese nodo, mientras que el otro proceso procede a ir al nodo padre. A este tipo de barrera de árbol binario se le denomina "barrera por torneo" (*tournament barrier*), dado que uno de los procesos puede pensarse como el perdedor del torneo en cada etapa del árbol de llegada.

Otra forma de asegurar la iteración local es usar árboles p-nodo para implementar una barrera entre los p procesos, donde cada nodo del árbol (hoja o interno) se asigna a un único proceso. Los árboles de llegada y la activación pueden ser los mismos, o pueden mantenerse como árboles con diferentes factores de ramaje. Cada nodo interno (proceso) en el árbol mantiene un array de variables de llegada, con una entrada por hijo, almacenadas en la memoria local del nodo. Cuando un proceso llega a la barrera, si su nodo en el árbol no es una hoja entonces comprueba en primer lugar su array de flags de llegada y espera hasta que todos sus hijos hayan indicado su llegada poniendo a uno su correspondiente entrada en el array. A continuación, pone a uno su entrada en el array de llegada de su padre (remoto) y realiza una espera activa sobre el flag de liberación asociado a su nodo en el árbol de liberación. Cuando llega el proceso raíz y todas los flags de llegada de su array están activos, entonces todos los procesos han llegado. La raíz pone a uno los flags (remotos) de liberación de todos sus hijos en el árbol de liberación; estos procesos salen de su espera activa y ponen a uno los flags de sus hijos, y así hasta que todos los procesos se liberan.

En el caso de máquinas con coherencia de caché, se pueden aplicar los mismos algoritmos teniendo en cuenta dos diferencias. Por una parte, las implicaciones de rendimiento de iterar sobre variables remotas son menos significativas, dada la posibilidad de tener una copia de la variable en la caché local. El que cada procesador itere sobre una variable distinta es, por supuesto, útil ya que de esta forma no todos los procesadores deben ir al mismo nodo origen para solicitar el nuevo valor de la variable cuando esta se invalida, reduciendo la contención. Sin embargo, existe una razón (muy poco probable) que puede ser muy importante para el rendimiento en el caso de que la variable sobre la cual itera un procesador esté almacenada localmente: si la caché está unificada, es de asignación directa y las instrucciones del bucle de iteración entran en conflicto con la propia variable, en ese caso los fallos debidos a conflictos se satisfacen localmente.

7.4.3 Rendimiento

Los experimentos realizados para ilustrar el rendimiento de la sincronización son los mismos que se usaron en la sección 6.7 para el SGI Challenge, usando de nuevo LL-SC como primitiva para construir operaciones atómicas. El resultado para los algoritmos que hemos discutido se muestran en la figura 7.11 para ejecuciones con 16 procesadores sobre la máquina SGI Origin 2000. De nuevo se usan tres conjuntos diferentes de valores para los retrasos dentro y después de la sección crítica.



(a) null (c=0, d=0) (b) critical-section (c=3.64 µs, d=0) (c) delay (c=3.64 µs, d=1.29 µs)

Figura 7.11: Rendimiento de los cerrojos son la máquina SGI Origin2000, para tres escenarios diferentes.

Al igual que para el caso de memoria compartida, hasta que no se introducen retrasos entre secciones críticas el comportamiento de los cerrojos no equitativos presentan un mayor rendimiento. El retraso exponencial ayuda al LL-SC cuando existe una sección crítica nula, dado que en este caso es posible reducir la alta contención que se produce. En caso contrario, con secciones criticas nulas el cerrojo basado en ticket escala pobremente, aunque en el caso de usar retraso exponencial escala muy bien. Los cerrojos basados en arrays también escalan muy bien. En el caso de existir coherencia de cachés, el mejor emplazamiento de las variables cerrojo en la memoria principal proporcionada por el cerrojo por colas software no es particularmente útil, y de hecho incurre en contención al realizar las operaciones de comparación e intercambio (implementadas con LL-SC) y escala peor que el cerrojo basado en array. Si forzamos a los cerrojos a que tengan un comportamiento equitativo, su comportamiento es muy parecido al cerrojo basado en ticket sin retraso exponencial.

Si usamos una sección crítica no nula y un retraso entre los accesos al cerrojo (Figura 7.11(c)), todos los cerrojos se comportan equitativamente. Ahora, el cerrojo LL-SC pierde su ventaja, mostrando su peor escalamiento. El cerrojo basado en array, el cerrojo por colas, y el cerrojo basado en ticket con retraso exponencial escalan muy bien. La mejor colocación de los datos del cerrojo por colas no importa, pero su contención no es peor que la del resto. En resumen, podemos concluir que los cerrojos basados en array y en ticket presentan un buen comportamiento y robustez para máquinas escalables con coherencia de caché, al menos al ser implementadas con LL-SC, y el cerrojo LL-SC con retraso exponencial se comporta mejor cuando no existe retraso entre la liberación y la siguiente

adquisición, debido a su repetido comportamiento no equitativo. El cerrojo más sofisticado, el basado en cola, es innecesario pero también se comporta bien cuando existen retrasos entre la liberación y la siguiente adquisición.

7.5 Conclusiones

En este capitulo hemos vistos que la mayoría de las máquinas escalables modernas están construidas a partir de nodos de propósito general con una jerarquía completa de memoria local junto con un asistente de comunicación que interactúa con una red de interconexión escalable. La mayoría de los diseños de máquinas paralelas están basadas en la revolución tecnológica de mediados de los 80 - microprocesador de un único chip - que fue acuñado como “killer micro” cuando las máquinas MPP empezaron a obtener mayor rendimiento que los tradicionales supercomputadores vectoriales. Sin embargo, estas máquinas fueron a su vez pioneras de otra revolución tecnológica - el conmutador de red escalable en un único chip. Como los microprocesadores, esta tecnología ha crecido más allá de su uso inicial habiendo surgido una amplia variedad de sistemas escalables, incluyendo las ethernets conmutadas gigabit. Como resultado, el diseño de las máquinas paralelas con gran número de procesadores se ha dividido en dos ramas. Las máquinas orientadas al concepto de paso de mensajes (multicomputadores) y su acceso explícito a la memoria remota han sido superados por los clusters y redes de estaciones de trabajo (NOW), dado su muy bajo coste, fácil utilización, y posibilidad de estar tecnológicamente al día. La otra rama es la de máquinas que integran profundamente la red en el sistema de memoria para proporcionar coherencia de caché y un espacio de memoria global con replicación automática. Este último tipo de sistemas que soportan un espacio de direcciones coherente se ha convertido en una importante parte del paisaje de los multiprocesadores. El soporte hardware para la coherencia caché se ha hecho crecientemente popular tanto en entornos técnicos como comerciales. La mayoría de estos sistemas usan protocolos basados en directorios, ya sean a nivel de memoria o a nivel de caché. Estos sistemas han mostrado un buen comportamiento, al menos para escalamientos moderados como los que han sido construidos, siendo más fáciles de programar que el paso explícito de mensajes para muchas aplicaciones.

Ejercicios

1. Dado un sistema con 512 procesadores en el que cada nodo visible al directorio tiene 8 procesadores, 1 GB de memoria principal y un tamaño de bloque de cache de 64 bytes, ¿Cuál es el overhead en memoria del directorio para el esquema *full bit vector*?.

2. ¿Porqué es la escritura atómica más difícil de proporcionar con protocolos de actualización que con protocolos basados en invalidación en los sistemas basados en directorios?. ¿Cómo se podría resolver el problema? ¿Existe la misma dificultad en un sistema basado en bus?.

3. Considerar el siguiente segemento de programa ejecutándose en un multiprocesador con coherencia de cache, asumiendo que todos los valores están inicializados a 0.

P_1	P_2	P_3	P_4
`A=1`	`u=A`	`w=A`	`A=2`
	`v=A`	`x=A`	

Hay sólo una variable compartida (A). Supón que el que lo ha escrito sabe donde están las copias de cache y les envía actualizaciones directamente sin consultar el nodo dirERECTORIO. Construir una situación en la que la atomicidad de la escritura pueda ser violada, asumiendo un protocolo basado en actualización.

(a) Muestra la violación de la consistencia secuencial que ocurre en los resultados

(b) ¿Puedes poner algún caso donde también sea violada la coherencia ? ¿Podrías resolver estos problemas?

(c) ¿Puedes construir los mismos problemas para un protocolo basado en invalidación?

(d) ¿Puedes construirlos para protocolos de actualización en un bus?.

4. ¿Cuántas transacciones de bus y cuánto tiempo se necesita si tenemos 20 procesadores haciendo lock y unlock a la misma variable uando *queuing loc*k que actualiza el lock en un fallo?. Asume que cada transacción del bus (fallo de lectura o de escritura) necesita 50 ciclos como mucho. Se puede ignorar el tiempo que la lectura o escritura actual del lock en la cache, así como el tiempo para ejecutar otras operaciones que no sean de sincronización en la barrera. Asume que el bus es totalmente justo de tal manera que cualquier petición pendiente se sirve antes que cualquier nueva petición y que los procesadores son iguales de rápidos.

5. Escribir el código para la barrera *sense-reversing* usando *fetch&increment.* Haciendo las mismas suposiciones que en el ejercicio anterior, y asumiendo que una operación *fetch&increment* necesita 50 ciclos de reloj, determinar el tiempo para que 20 procesadores atraviesen la barrera. ¿Cuántos ciclos de bus se necesitan?.