Capítulo 4

PROCESAMIENTO PARALELO

4.1. Guión-Esquema

Los contenidos que se tratan a lo largo del tema se resumen en los siguientes puntos:

- Organización y principales características de las diferentes plataformas de computación paralela.
- Paradigmas de programación paralela, detallando dos casos: cliente/servidor y SPMD (Single Program Multiple Data).
- Descripción de las arquitecturas basadas en comunicaciones mediante espacio de memoria compartido.
- Descripción del sistema de comunicación mediante paso de mensajes.
- Tipos de redes de intercomunicación en sistemas paralelos: estáticas y dinámicas.
- Análisis del problema de coherencia de caché en sistemas multiprocesador y sus posibles soluciones.
- Estudio de los *clusters* como principales representantes de los sistemas de memoria distribuida.
- Análisis del rendimiento y costes de computación en sistemas paralelos.

4.2. Introducción

Hasta llegar a este punto, todo lo que se ha estudiado en los tres temas previos estaba orientado a la mejora de las prestaciones de computadores construidos con un único procesador. Sin embargo, hoy en día se sabe que los sistemas basados en un único procesador presentan límites en el rendimiento que se pueden alcanzar, por lo que la tendencia que se ha establecido es la de construir máquinas dotadas de varios procesadores, pudiendo oscilar el número de procesadores desde unos pocos, como sucede en los computadores orientados al ámbito doméstico o empresarial, hasta alcanzar los cientos de miles en sistemas orientados a la computación de alto rendimiento. Evidentemente, esta tendencia, basada en el incremento de rendimiento en base a la suma de procesadores, va unida a toda la investigación que se continúa realizando para mejorar las características de los procesadores a nivel individual (consumo de energía, disipación de calor, tamaño, velocidad, etc.), ya se trate de procesadores superescalares, vectoriales o VLIWs.

En este capítulo se presentan las arquitecturas más relevantes para construir computadores dotados de varios procesadores, es decir, sistemas de procesamiento paralelo. El procesamiento paralelo es el método de organización de las operaciones en un sistema de computación donde más de una operación es realizada de manera simultánea (o concurrente). La finalidad inicial del procesamiento paralelo es obtener una mayor capacidad de cómputo. Para ello se intenta dividir el problema en múltiples tareas que se ejecutarán de manera paralela. De esta manera, para poder realizar varias tareas de manera simultánea es necesario disponer de múltiples procesadores.

La evolución de los sistemas paralelos ha sido constante desde sus inicios. Los primeros multiprocesadores datan de mediados del siglo XX, como por ejemplo el 704 de IBM que fue el primer sistema en incluir un sistema de cálculo en coma flotante o el sistema D825 de la Burroughs Corporation que incluía hasta 4 procesadores y 16 módulos de memoria, conectados por un switch crossbar. También cabe mencionar los sistemas Multics desarrollados por Honeywell a partir de 1969, el C.mmp desarrollado por la Universidad Carnegie Mellon en 1970 y el Synapse N+1 desarrollado en 1984, que fue el primer computador basado en bus que incluía un sistema snoopy para solucionar el problema de coherencia de cachés. Aún así, hoy en día la investigación en el campo de la computación paralela sigue siendo un campo muy activo.

Aunque no existe un consenso sobre la definición de un multiprocesador, las siguientes propiedades corresponden a una de las primeras caracterizaciones de sistemas multiprocesador, que fue realizada por P.H. Enslow en 1977:

- Debe estar compuesto por dos o más procesadores.
- Los procesadores deben compartir el acceso a una memoria común.
- Los procesadores deben compartir acceso a canales de E/S, unidades de control y dispositivos.
- El sistema es controlado por un único sistema operativo.

A continuación, se describen las características, estructura, organización y diseño de las diferentes plataformas de computación paralela que existen en la actualidad.

4.3. Tipos de plataformas de computación paralela

La estructura de los tipos de plataformas de computación paralela depende de su organización lógica y física:

- La *organización lógica* se refiere a la visión que el programador tiene de la plataforma, es decir, las capacidades para expresar tareas paralelas (la *estructura de control*) y los métodos de comunicación entre dichas tareas (el *modelo de comunicación*).
- La *organización física* se refiere a la estructura del hardware que compone la plataforma. Como ya se ha comentado, una de las características principales de un sistema multiprocesador es el acceso compartido a una región de memoria común a todos los procesadores. Existen dos modelos para implementar este espacio de direcciones común:
  - Sistemas de *memoria compartida*, en los que un único sistema de memoria física es compartido por todos los procesadores.
  - Sistemas de *memoria distribuida*, en los que cada procesador tiene su propia memoria física, a la que el resto de procesadores no tiene acceso directo.

En esta sección se describe la organización lógica de las plataformas de computación paralela, relativa a la estructura de control y al modelo de comunicación. En sucesivas secciones se detallará la organización física, describiendo las plataformas de memoria compartida y distribuida.

4.3.1. Organización basada en la estructura de control

Las aplicaciones paralelas se pueden clasificar en algunos paradigmas de programación claramente establecidos. Basta con unos pocos paradigmas para, utilizándolos repetidamente, poder desarrollar la mayoría de los programas paralelos. En este contexto, por paradigma se entiende una clase de algoritmos que tienen la misma estructura de control.

La elección del paradigma a utilizar depende de la disponibilidad de los recursos computacionales paralelos que se tengan y del tipo de paralelismo inherente al problema. Los recursos computacionales definen el grado de acoplamiento o nivel de granularidad que puede soportar el sistema de forma eficiente, por ejemplo, un programa paralelo con granularidad gruesa se corresponde con un programa con muchas instrucciones secuenciales, lo que equivale a un grado de acoplamiento bajo. El tipo de paralelismo refleja la estructura de la aplicación y/o de los datos. El paralelismo debido a la estructura de la aplicación se denomina *paralelismo funcional*. En este caso, las diferentes partes de un programa pueden realizar distintas tareas de una manera concurrente y cooperativa. Pero el paralelismo se puede encontrar también en la estructura de los datos; esta clase de paralelismo permitirá la ejecución de procesos paralelos con

idéntico modo de operación pero sobre distintas partes de los datos. A este segundo tipo de paralelismo se le denomina *paralelismo estructural* o *paralelismo a nivel de datos*.

En el área de la computación paralela diferentes autores presentan distintas clasificaciones de paradigmas. La mayoría de estas clasificaciones se basan en los siguientes criterios:

- propiedades del proceso (estructura, topología y ejecución).
- propiedades de interacción.
- propiedades de los datos (división y localización).

Aunque no todos los autores proponen exactamente la misma clasificación, realizando la unión de todas ellas se puede crear un amplio conjunto de los paradigmas que se utilizan en las aplicaciones paralelas. Los más comunes son:

- *Descomposición iterativa*: Algunas aplicaciones están basadas en la ejecución de un lazo donde cada iteración se puede realizar de forma independiente. Esta técnica se implementa a través de una cola central de tareas ejecutables, y se corresponde con el paradigma de descomposición en grupos de tareas.
- *Paralelismo algorítmico*: el cual se centra en paralelizar el flujo de los datos de entrada.
- *Descomposición geométrica*: El dominio del problema se divide en pequeños subdominios y cada procesador ejecuta el algoritmo en la parte del subdominio que le corresponde.
- *Descomposición especulativa*: Se intentan N técnicas de solución simultáneamente, y $(N-1)$ de ellas se eliminan tan pronto como una de ellas devuelve una respuesta correcta.
- *Descomposición funcional*: La aplicación se divide en distintas fases, y cada fase ejecuta una parte diferente del algoritmo para resolver el problema. Tambien puede denominarse *segmentación de datos*.
- *Maestro/Esclavo*: El proceso maestro es el responsable de descomponer el problema entre sus procesos esclavos y de, posteriormente, recoger los resultados que le envían los esclavos para ordenarlos y obtener el resultado final.
- *SPMD (Single Program Multiple Data)*: En este paradigma cada procesador ejecuta el mismo código pero sobre distintas partes de los datos.
- *Descomposición recursiva o divide y vencerás*: El problema se divide en subproblemas que se resuelven de forma independiente para, posteriormente, combinar sus resultados parciales y obtener el resultado final.

A continuación se comentan con mayor detalle únicamente los dos paradigmas más extendidos: Maestro/Esclavo y SPMD.

4.3.1.1. Paradigma Maestro/Esclavo

El paradigma Maestro/Esclavo es el más utilizado en las aplicaciones paralelas y en la mayoría de los casos, como el propio nombre indica, consta de dos tipos de entidades: un maestro y varios esclavos. El maestro es el responsable de la descomposición del problema en pequeñas tareas, de distribuir estas tareas entre el conjunto de procesadores esclavos y de recoger los resultados parciales obtenidos de cada procesador esclavo para ordenarlos y obtener el resultado final del problema. Los procesadores esclavos reciben un mensaje con la tarea a realizar, realizan la tarea y envían el resultado al maestro. Normalmente, la comunicación tiene lugar únicamente entre el maestro y los esclavos.

El paradigma Maestro/Esclavo puede utilizar un *balance de carga estático* o un *balance de carga dinámico* (que serán descritos en la Sección 4.6.1.7). En el primer caso, la distribución de las tareas se realiza al comienzo de la computación, lo cual permite al maestro participar en la computación una vez que haya asignado una fracción del trabajo a cada esclavo. La asignación de tareas se puede realizar de una sola vez o de manera cíclica. La Figura 4.1 muestra una representación esquemática de una estructura Maestro/Esclavo estática.

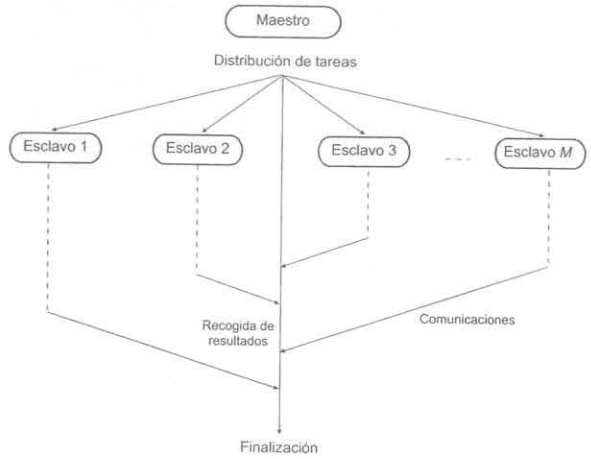


Figura 4.1: Estructura Maestro/Esclavo estática.

La otra forma es utilizar el paradigma Maestro/Esclavo con balance de carga dinámico, el cual es muy útil cuando el número de tareas es mayor que el número de procesadores disponibles o cuando el número de tareas es desconocido al comienzo de la aplicación. Una característica importante del balance de carga dinámico es la capacidad que tiene la aplicación de adaptarse a los posibles cambios del sistema, no solo

a la carga de los procesadores sino también a posibles reconfiguraciones de los recursos del sistema. Debido a esta característica, este paradigma puede responder bastante bien cuando se produce el fallo de algún procesador, lo cual simplifica la creación de aplicaciones tolerantes a fallos que sean capaces de sobrevivir cuando se pierde algún esclavo o incluso el maestro.

Este paradigma puede alcanzar elevadas aceleraciones computacionales y un interesante grado de escalabilidad[1]. Sin embargo, para un gran número de procesadores el control centralizado del proceso maestro puede llegar a ser el cuello de botella de las aplicaciones. Pero a pesar de ello, es posible mejorar la escalabilidad del paradigma pasando de un procesador maestro a un conjunto de procesadores maestros, controlando a su vez cada uno de ellos a un conjunto de procesadores esclavos.

4.3.1.2. Paradigma SPMD (Single Program Multiple Data)

En el paradigma SPMD cada procesador ejecuta básicamente el mismo código pero sobre distintas partes de los datos. Esto supone dividir los datos de la aplicación entre los procesadores disponibles. A diferencia del paradigma Maestro/Esclavo, en este modelo la comunicación se establece entre esclavos. A este tipo de paralelismo también se le denomina paralelismo geométrico, estructural o paralelismo a nivel de datos. La Figura 4.2 muestra un esquema de esta clase de paradigma.

Figura 4.2: Estructura básica de un programa SPMD.

En un caso extremo, este paradigma también engloba aquellas aplicaciones donde un algoritmo secuencial se ejecuta simultáneamente en diferentes procesadores pero con diferentes datos de entrada. En estas aplicaciones no hay dependencias entre las tareas de los distintos procesadores esclavos por lo que no se necesita la comunicación ni la coordinación entre ellos.

[1]La escalabilidad puede entenderse como la capacidad del sistema para mejorar la potencia de cálculo cuando el número de componentes del sistema aumenta.

Muchos problemas físicos tienen una estructura geométrica regular, homogeneidad que permite distribuir los datos uniformemente entre los procesadores. Los procesadores se comunican entre sí, donde la carga de comunicación será proporcional al tamaño de la información enviada y la carga computacional será proporcional al volumen de los elementos. Este paradigma se puede utilizar para realizar alguna sincronización global periódica entre todos los procesadores. Normalmente, los esquemas de comunicación son muy estructurados y extremadamente predecibles. Los datos iniciales pueden ser generados por cada procesador o pueden ser leídos de memoria secundaria o distribuidos por uno de los procesadores. En este último caso se puede presuponer la existencia de un procesador maestro en la fase inicial.

Las aplicaciones SPMD pueden ser muy eficientes si los datos están bien distribuidos entre los procesadores y el sistema es homogéneo. Si los procesadores presentan distintas cargas de trabajo o capacidades, el paradigma necesitará algún mecanismo de planificación de balance de carga centralizado, completamente distribuido o parcialmente distribuido para distribuir los datos durante el tiempo de ejecución.

Sin embargo, este paradigma es muy sensible a la pérdida de algún procesador. Normalmente, el fallo de un único procesador es suficiente para bloquear la aplicación, debido a que entonces ningún procesador podrá avanzar más allá del punto de sincronización global.

4.3.2. Organización basada en el modelo de comunicación

Existen principalmente dos modelos de comunicación de información entre tareas paralelas:

- El espacio de direcciones único y compartido (memoria compartida).
- El paso de mensajes.

4.3.2.1. Espacio de direcciones compartido

Los procesadores de los sistemas con memoria compartida se caracterizan por compartir físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito en memoria por un procesador puede ser leído directamente por cualquier otro. En principio, en esta arquitectura la memoria es igualmente accesible por todos los procesadores a través de la red de interconexión. La Figura 4.3 muestra el esquema básico de la arquitectura de memoria compartida.

En este contexto, en el que la red de interconexión es determinante para la eficacia del sistema, son fundamentales dos parámetros que caracterizan la velocidad de transferencia entre elementos del sistema paralelo:

- *Latencia de red*, es el tiempo que se tarda en enviar un mensaje a través de la red de interconexión del sistema paralelo.
- *Ancho de banda*, definido como el número de bits que se pueden enviar por unidad de tiempo.

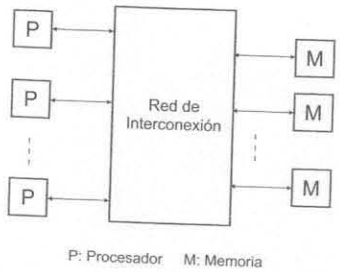


Figura 4.3: Esquema de la arquitectura de memoria compartida.

Para garantizar la eficacia de esta arquitectura es fundamental que el ancho de banda sea elevado, ya que en cada ciclo de instrucción cada procesador puede necesitar acceder a la memoria a través de la red de interconexión. Este acceso a memoria puede ser lento debido a que las solicitudes de lectura o de escritura pueden tener que pasar por varias etapas en la red. En la arquitectura de memoria compartida se satisface que la latencia de red es baja y el ancho de banda alto, solo si no se encuentran varios procesadores tratando de acceder al medio utilizado para la transmisión de datos *simultáneamente*. Por este hecho y otros problemas relacionados con el mismo, el número de procesadores en este tipo de sistemas suele ser pequeño y la arquitectura más común para la comunicación es la de bus, en la cual todos los procesadores y módulos de memoria se conectan a un único bus, tal como muestra la Figura 4.4. En esta arquitectura el tiempo de acceso a memoria es el mismo para cualquier palabra accedida por un procesador, por lo que se denomina arquitectura de acceso uniforme a memoria (UMA - *Uniform Memory Access*).

La mayoría de los sistemas de memoria compartida incorporan una memoria caché local en cada procesador y, del mismo modo que en los computadores secuenciales, sirve para aumentar el ancho de banda entre el procesador y la memoria local. Análogamente, puede existir una memoria caché para

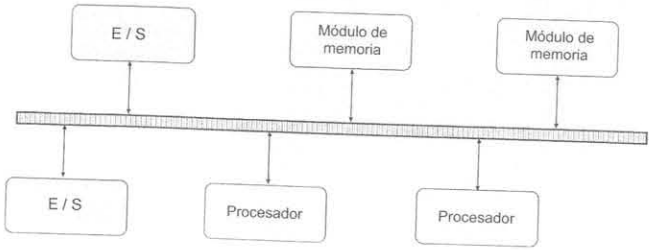


Figura 4.4: Organización multiprocesador de bus común.

la memoria global. Cuando se utilizan memorias caché es fundamental asegurar la coherencia de la información en la memoria caché, de modo que cuando un procesador modifique el valor de una variable compartida los demás procesadores no consideren su valor anterior, ya que es incorrecto desde ese momento. Sin embargo, existe otro factor fundamental en sistema paralelos que limita a la arquitectura UMA: la *escalabilidad*. De forma genérica puede entenderse como la capacidad del sistema para mejorar la potencia de cálculo cuando el número de componentes del sistema aumenta. En el caso ideal sería lineal, de manera que al tener dos procesadores se dispondría del doble de potencia de cálculo, con cuatro el cuádruple, y con cien tendríamos cien veces más. Sin embargo, debido a factores como los tiempos de comunicación, la escalabilidad no suele ser lineal y, además, llega a saturarse. La saturación indica que por muchos componentes más que se incorporen, no se consigue ninguna mejora. Actualmente, los sistemas UMA se utilizan para la construcción de arquitecturas multiprocesador de bus compartido con un reducido número de procesadores, ya que con un mayor número de procesadores los problemas de acceso a la memoria remota se incrementan. Incluso incorporando mecanismos de memoria caché, los sistemas UMA presentan una escalabilidad limitada. Normalmente, en la actualidad su número máximo de procesadores está entre 16 y 32.

Sin embargo, algunas mejoras pueden realizarse dotando a cada procesador de una memoria local, en la que se almacena el código que se está ejecutando en el procesador y aquellos datos que no tengan que ser compartidos por los otros procesadores y, por tanto, son locales a ese procesador. Mediante esta estrategia se evitan accesos a memoria a través de la red de interconexión relacionados con búsquedas

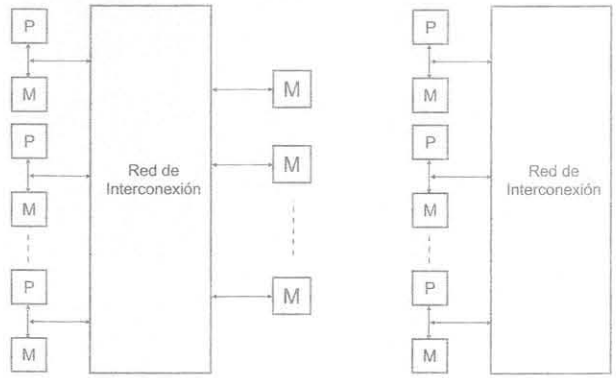


Figura 4.5: a) Arquitectura de memoria compartida con acceso no uniforme a memoria, b) memoria compartida con acceso no uniforme a memoria solo con memoria local.

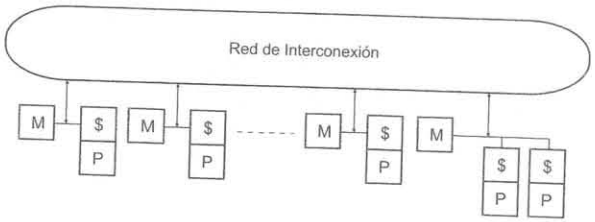
de código y datos locales, lo que mejora la eficacia del sistema. Esta modificación de la arquitectura es conocida como memoria compartida con acceso no uniforme a memoria (NUMA - *Non Uniform Memory Access*) y se ilustra en la Figura 4.5.a. El tiempo de acceso a memoria es no uniforme dependiendo de si el acceso se realiza a la memoria local del procesador o a alguna otra memoria remota.

Los sistemas NUMA son también conocidos como sistemas de memoria compartida distribuida (DSM - *Distributed Shared Memory*). La idea de distribuir la memoria compartida puede llevarse al límite eliminando completamente el bloque de memoria compartida común, la global, tal y como se muestra en la Figura 4.5.b, siempre y cuando los tiempos de acceso a las memorias locales sean muy inferiores al tiempo de acceso por la red de interconexión. En este diseño los accesos de un procesador a las memorias locales de los otros se realizan mediante el control de un hardware específico.

En general, el esquema de la arquitectura NUMA no requiere del empleo de mecanismos especiales que aseguren que todos los nodos tienen un conocimiento coherente del espacio de direcciones global. Es posible manejar un espacio común de direcciones y que sea el programador el encargado de manipular el contenido de esas direcciones de memoria evitando cualquier falta de coherencia.

Por otro lado, el mantenimiento de la coherencia de las cachés de los diferentes procesadores de manera transparente al programador supone una gran ventaja. Las arquitecturas NUMA que incluyen mecanismos hardware dedicados a mantener la coherencia de caché son: ccNUMA (*caché-coherent NUMA*) y COMA (*Caché-Only Memory Access*).

En los sistemas ccNUMA cada nodo contiene una porción de la memoria total del sistema (ver Figura 4.6). Las variables compartidas se reparten directamente por el programador o por la máquina haciendo uso del sistema operativo, de manera que solo existe una única copia de cada variable en la memoria principal. Cada nodo puede constar de uno o varios procesadores con sus cachés y una memoria principal. Los diferentes procesadores pueden almacenar múltiples copias de los mismos datos en sus cachés locales, por lo que es necesario mantener la coherencia de los datos a través de todo el sistema. La coherencia se mantiene normalmente mediante el escaneo de las operaciones que realizan los procesadores y se transmiten por la red (protocolos snoopy) o manteniendo un registro de la localización

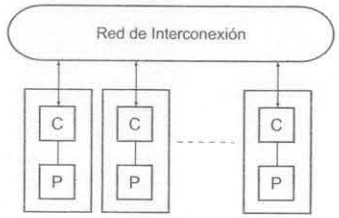


$: Protocolo de Coherencia de Cache

Figura 4.6: Arquitectura ccNUMA.

de las variables en cada procesador (directorios). Ambas técnicas se detallarán en la Sección 4.4.2.

En el caso de la arquitectura COMA, los procesadores que componen cada nodo no incluyen memoria local sino que solo poseen caché. La idea básica es emplear la memoria local de cada nodo del multiprocesador como si fuese una caché del resto del sistema, transfiriendo los datos de cada nodo en función de las necesidades del código en ejecución. Esta idea es análoga a la de caché de un sistema monoprocesador, de modo que la memoria local de un nodo actúa como caché y el conjunto de la memoria local de los otros nodos sería el equivalente a la memoria principal. Si un procesador tiene que acceder repetidamente a una posición de memoria que se encuentra en un nodo remoto se puede copiar la posición de memoria remota en la memoria local, y en los siguientes accesos no habrá que realizar un acceso remoto para acceder al dato. El esquema de esta arquitectura se muestra en la Figura 4.7. Como ya se ha comentado, los esquemas de coherencia de caché más utilizados se detallarán en la Sección 4.4.2.



C: Memoria caché

Figura 4.7: Arquitectura COMA.

La ventaja fundamental de los sistemas COMA es que son capaces de tratar los fallos de acceso remoto distribuyendo los datos que está utilizando la aplicación automáticamente por el sistema. La principal desventaja es la complejidad para mantener la coherencia de las copias de las variables a lo largo de todo el sistema, ya que no existe una única copia segura, como sucede en el caso ccNUMA.

4.3.2.2. Paso de mensajes

El paso de mensajes consiste en el intercambio de información, en forma de mensajes, entre los diferentes procesadores que componen el sistema. Cada uno de estos procesadores puede estar compuesto por un único procesador, o un sistema multiprocesador con emoria compartida. Esta última opción está ganando fuerza en las plataformas modernas de computación paralela.

Los elementos necesarios para describir el sistema de comunicación por medio de mensajes son los siguientes: *emisor*, *receptor*, *canal de comunicación* y el *mensaje*. El emisor y el receptor están representados por el procesador que inicia la comunicación y el que la recibe, respectivamente. El canal de comunicación lo compone la red de interconexión existente entre los diferentes procesadores. Las diferentes tipologías de red que se pueden encontrar son descritas más adelante en este capítulo.

El mensaje está compuesto por la información enviada, ya sean datos, trabajos o información de sincronización de tareas.

Existen cuatro operaciones básicas necesarias para establecer un mecanismo de paso de mensajes:

- `Envío`: Usada para enviar el mensaje.
- `Recepción`: Usada para recibir el mensaje.
- `Identificación`: Usada para identificar cada uno de los procesadores del sistema, mediante un identificador único. Este identificador debe acompañar a las operaciones de envío y recepción para poder identificar al emisor y el receptor.
- `Número de participantes`: Usada para identificar el número de procesadores que participan en la comunicación.

Mediante estas cuatro operaciones es posible escribir cualquier tipo de programa por medio de paso de mensajes. Diferentes implementaciones de mecanismos de paso de mensajes, tales como *Parallel Virtual Machine* (PVM) y *Message Passing Interface* (MPI), incluyen estas cuatro operaciones así como otras operaciones de más alto nivel.

En un sistema con el espacio de direcciones compartido es relativamente sencillo emular un sistema de comunicación por paso de mensajes. Simplemente dividiendo el espacio de direcciones en fracciones iguales, cada una asignada a un procesador, las acciones de envío y recepción de mensajes estarían representadas por la escritura y lectura de información en dichas fracciones de memoria. Por ejemplo, un procesador P_1 podrá enviar un mensaje al procesador P_2 escribiendo la información del mensaje en la parte del espacio de direcciones asignada a éste. Sin embargo, en un sistema de paso de mensajes se hace muy costoso emular un sistema con espacio de direcciones único. Esto es porque los accesos a la memoria correspondiente a otros procesadores requieren del envío y recepción de múltiples mensajes.

4.4. Sistemas de memoria compartida

Cuando se quiere realizar el diseño de un sistema de memoria compartida se deben tener en cuenta tres aspectos básicos:

- La organización de la memoria principal.
- El diseño de la red de interconexión.
- El diseño del protocolo de coherencia de la caché.

El primer aspecto sobre la organización de la memoria principal ha sido descrito previamente en la Sección 4.3.2.1 por lo que no se tendrá en cuenta a lo largo de esta sección. A continuación se introducirán y se aclararán los conceptos relacionados con la red de interconexión y los mecanismos de coherencia de caché.

4.4.1. Redes de interconexión

Independientemente del mecanismo usado para comunicar los procesadores, las operaciones deben ser transmitidas por medio de una red de interconexión. Esta red puede utilizarse para acceder a memoria remota o para transportar mensajes entre los diferentes procesadores.

En esta sección se analizan las diferentes topologías de redes que se pueden encontrar en computadores paralelos, y principalmente en sistemas de memoria compartida. Estas se dividen en dos clases principales: las redes *estáticas*, definidas durante la construcción de la máquina y las redes *dinámicas*, que pueden adaptarse a los requisitos de comunicación de los programas que se ejecuten.

4.4.1.1. Redes estáticas

Una red estática, también denominada red directa, es una red cuya topología queda definida de manera definitiva y estable durante la construcción de la máquina paralela. Como se puede apreciar, el papel de la red de interconexión es tanto más importante cuanto mayor sea el número de elementos físicos que se deben unir y el flujo de información que se desee intercambiar.

En general, las redes estáticas pueden presentar distintas topologías en función de las conexiones punto a punto que se establezcan entre sus procesadores. Se pueden dividir en cuatro tipos básicos: redes unidimensionales, bidimensionales, tridimensionales e hipercubos.

Las *redes unidimensionales* son las más sencillas de realizar. La idea más inmediata es conectar cada procesador a dos procesadores vecinos, uno a la derecha y otro a la izquierda. La Figura 4.8.a muestra esta disposición, conocida como *red lineal*, en la que todos los procesadores salvo los extremos están enlazados a otros dos procesadores.

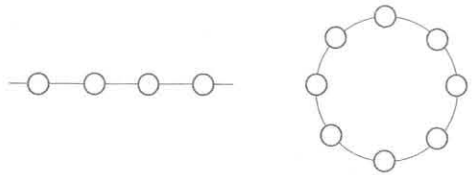


Figura 4.8: Topologías de red unidimensionales: a) lineal, b) anillo.

La diferencia fundamental de las redes lineales con el *bus* radica en el hecho de que en un momento dado puede realizarse más de una transferencia simultáneamente siempre que sea a través de enlaces diferentes. Por ejemplo, un procesador puede enviar un mensaje simultáneamente a un procesador situado a su izquierda y a otro a su derecha. Esta topología es muy simple pero presenta problemas de comunicación cuando el número de procesadores es elevado.

Una pequeña modificación de esta estructura, en la que se enlazan los extremos finales, permite notables mejoras en la comunicación ya que cada procesador enviará los mensajes por su izquierda o por su derecha, dependiendo de cual sea el camino más corto. Esta red, también unidimensional, es denominada *anillo*, y se muestra en la Figura 4.8.b.

Una primera estrategia bidimensional puede obtenerse a partir de un anillo incrementando el número de enlaces por nodo, con lo cual disminuyen los tiempos de transferencia entre los nodos de la red. Esta topología denominada de *anillo cordal*, se ilustra en la Figura 4.9.a. Otros esquemas intuitivos son la *malla* (véase la Figura 4.9.b) y la *red sistólica* o *array sistólico* (Figura 4.9.c). Obsérvese que la red sistólica es una malla con conexión en una diagonal entre los procesadores de un cuadrado.

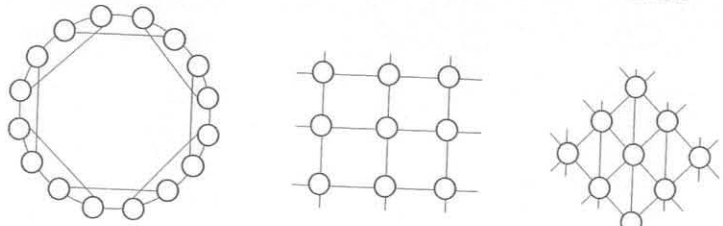


Figura 4.9: Topologías de red bidimensionales: a) anillo cordal, b) malla, c) red sistólica.

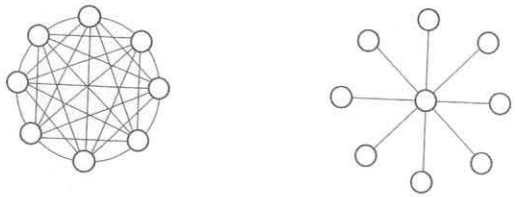


Figura 4.10: Topologías de red bidimensionales: a) red completamente conectada, b) red en estrella.

El esquema bidimensional ideal es la *red completamente conectada*, en la que cada procesador se comunica directamente con cualquier otro, de manera que los mensajes se envían en una única etapa o paso. La Figura 4.10.a muestra la estructura de esta red.

Obsérvese que una red completamente conectada no es más que un anillo cordal llevado a su máxima expresión. Sin embargo, al incrementarse el número de enlaces, se incrementa el coste de la red. Estas

redes son la equivalencia estática a las redes dinámicas *crossbar*, que se describirán más adelante. Ambas son redes no-bloqueantes, ya que una conexión dada entre dos elementos de la red no puede ser bloqueada por otra conexión previa entre otros dos elementos diferentes de la red.

Una estrategia que alivia el coste de las conexiones al aumentar el número de procesadores consiste en disponer de un procesador que actúe como procesador central, de manera que para cada uno de los demás procesadores se establece un camino de comunicación con él. La Figura 4.10.b muestra esta topología, denominada en *estrella*. Como puede observarse, esta estrategia es también similar a la del bus común, ya que las comunicaciones entre dos procesadores se establecen a través del procesador central. En estas redes dicho procesador es fundamental.

Como aplicación directa de las estructuras de datos tipo árbol surgen las *redes árbol*. En ellas hay un procesador en cada nodo del árbol y solo un camino de conexión entre cualquier par de procesadores. En la Figura 4.11 se muestran dos redes de árboles binarios. En la Figura 4.11.a todos los nodos son procesadores mientras que en la Figura 4.11.b solo las hojas del árbol son procesadores siendo el resto elementos de conmutación (que no procesan información, solo la reenvían).

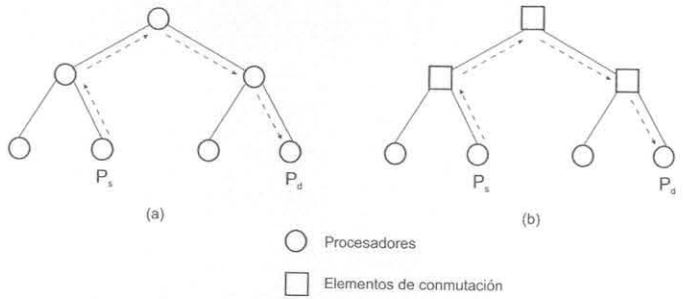


Figura 4.11: Topología de red bidimensional: red de árboles binarios.

Las redes *lineales* y las redes en *estrella* son casos particulares de la topología en árbol. El camino de comunicación se realiza de la siguiente forma: cuando un procesador envía un mensaje lo transmite hacia arriba en el árbol hasta que encuentra el procesador destino o llega al nodo raíz del menor subárbol que contiene tanto al procesador origen como al destino. En este caso una vez alcanzado este nodo raíz, el mensaje desciende por el árbol hasta encontrar el destino.

Las redes de árboles tienen la desventaja de que las comunicaciones pueden verse comprometidas en un nodo cuando el número de procesadores es grande y se realizan comunicaciones entre procesadores situados en los niveles superiores. Así por ejemplo, si muchos procesadores del subárbol izquierdo requieren comunicarse con muchos procesadores del derecho, entonces el nodo raíz debe manejar todos los mensajes, con lo que la eficacia del sistema disminuye significativamente ya que aumenta el tiempo empleado para las comunicaciones.

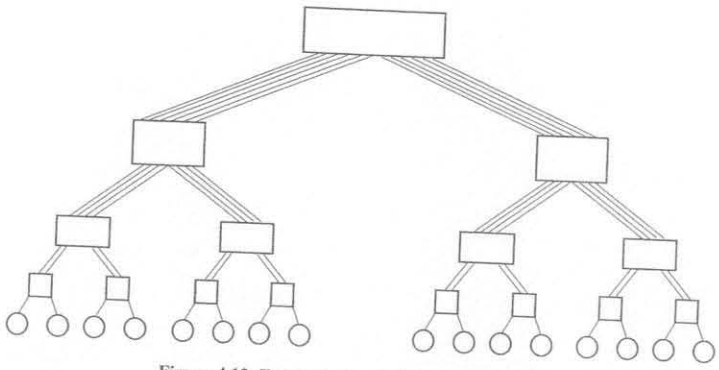


Figura 4.12: Topología de red bidimensional: fat tree.

Una estrategia comúnmente usada para aliviar esta desventaja consiste en aumentar el número de conexiones de comunicación entre los procesadores de menor nivel, es decir, los cercanos al nodo raíz. Esta red, que se muestra en la Figura 4.12, es conocida como *red de árbol grueso (fat tree)*.

Las redes en árbol tienen una equivalencia dinámica, realizada de manera que los nodos intermedios son elementos de conmutación mientras que las hojas son procesadores.

Por último, las redes bidimensionales tipo *mesh* surgen como una extensión de las redes lineales. En las redes *mesh bidimensionales* cada procesador se conecta directamente con otros cuatro procesadores salvo en los extremos, tal y como se muestra en la Figura 4.13.a. Cuando los procesadores forman una estructura cuadrada con igual número de procesadores en cada dimensión se denomina *mesh cuadrado*, y si el número de procesadores es diferente en las dos dimensiones se denomina *mesh rectangular*. Como es natural los procesadores extremos pueden conectarse entre ellos, como se hace con las redes lineales para formar un anillo, tal y como se muestra en la Figura 4.13.b. A esta topología se le denomina *mesh cerrada* o *toro*. En estas topologías mesh la ruta de comunicación puede realizarse a través de una dimensión y seguidamente a través de la otra.

Una extensión inmediata de los mesh bidimensionales es la topología *mesh tridimensional*, que de nuevo puede establecerse con los procesadores periféricos conectados o no conectados. La Figura 4.14 muestra esta topología.

Más allá de las tres dimensiones aparecen las *redes hipercubo*. Estas redes son mesh multidimensionales con dos procesadores en cada dimensión, de manera que un hipercubo de dimensión d está constituido por $p = 2^d$ procesadores.

Los hipercubos pueden construirse de forma recursiva teniendo en cuenta que un hipercubo de dimensión cero consta de un único procesador, uno de dimensión uno se forma conectando dos hipercubos de dimensión cero, y así sucesivamente, de manera que un hipercubo de dimensión l se forma

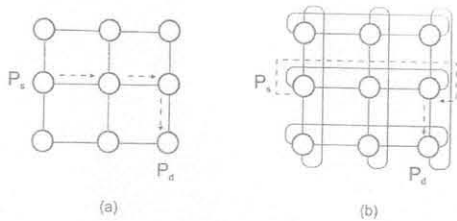


Figura 4.13: Topologías de red bidimensionales: a) mesh cuadrada, b) mesh cerrada o toro.

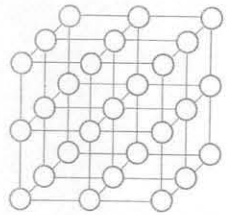
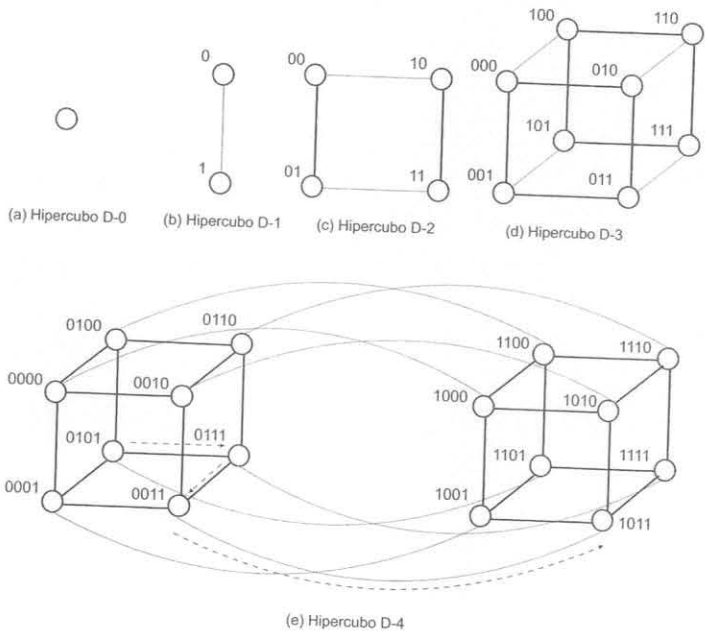


Figura 4.14: Topología de red tridimensional: mesh tridimensional.

conectando los procesadores correspondientes a dos hipercubos de dimensión $l - 1$. En la Figura 4.15 se muestran los hipercubos de dimensión cero, uno, dos, tres y cuatro.

Los hipercubos presentan propiedades especialmente interesantes, entre las que destacan las siguientes:

- Dos procesadores se conectan entre sí, si y solo si sus etiquetas, en binario, tienen exactamente un bit distinto en una posición determinada, tal y como se muestra en la Figura 4.15.
- Un procesador de un hipercubo de dimensión d se conecta directamente a d procesadores.
- Todo hipercubo de dimensión d puede dividirse en dos de dimensión $d - 1$. Para ello se selecciona la posición de un bit y se agrupan todos los procesadores que tengan un cero en esa posición. Todos ellos forman una partición y el resto forma la segunda partición. La Figura 4.16 muestra las tres particiones de un hipercubo tridimensional. Las líneas gruesas conectan los procesadores que pertenecen a una misma partición. Las etiquetas que identifican un procesador en un hipercubo de dimensión d constan de d bits. Para cualquier grupo de k bits fijo, los procesadores que difieren en los demás $d - k$ bits forman un subcubo de dimensión $d - k$ formado por 2^{d-k} procesadores. Dado que con k bits se pueden obtener 2^k combinaciones diferentes, se tendrán 2^k subcubos distintos. En la Figura 4.17 se ilustra este hecho, para $k = 2$ y $d = 4$. En la Figura 4.17.a se muestran los



(a) Hipercubo D-0 (b) Hipercubo D-1 (c) Hipercubo D-2 (d) Hipercubo D-3

(e) Hipercubo D-4

Figura 4.15: Hipercubos de dimensión: a) cero, b) uno, c) dos, d) tres, e) cuatro.

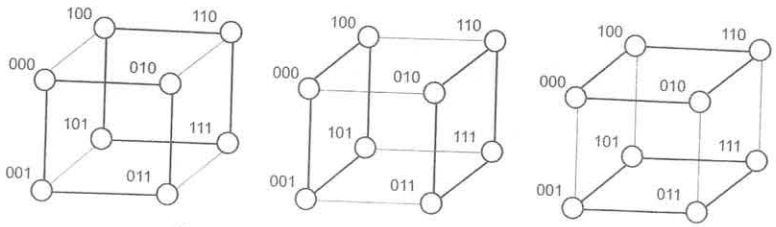
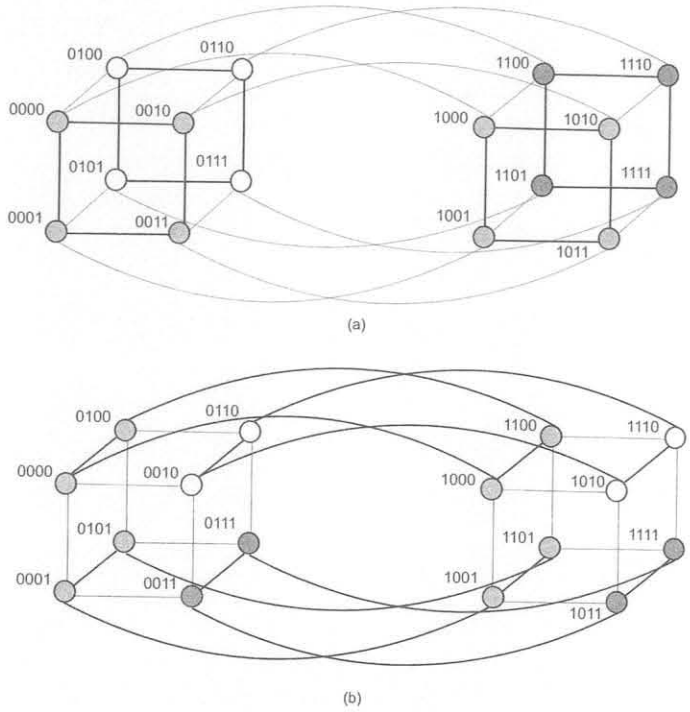


Figura 4.16: Particiones de un hipercubo tridimensional.

cuatro subcubos, cada uno de ellos formado por cuatro procesadores, cuando los bits fijados son los dos más significativos, y en la Figura 4.17.b los obtenidos cuando se fijan los dos bits menos significativos.



(a)

(b)

Figura 4.17: Divisiones de la red hipercubo.

- Un parámetro de especial interés es la denominada *distancia de Hamming*, que se define como el número total de posiciones de bits para los que las etiquetas de dos procesadores son diferentes. Por ejemplo, la distancia de Hamming para los procesadores 011 y 101 es dos, y entre 111 y 000 es tres. Con todo ello, la distancia de Hamming entre dos procesadores de etiquetas a y b es el número de bits a 1 que hay tras el resultado de la operación $a \oplus b$, donde $\oplus$ es la or-exclusiva. El número de enlaces por el camino más corto entre dos procesadores viene dado por la distancia

de Hamming. La Figura 4.15.e muestra la ruta de un mensaje desde el procesador $a = 0101$ al procesador $b = 1011$. La distancia de Hamming en este caso es tres, ya que $a \oplus b = 1110$. El mensaje se transmite por las dimensiones para las que la posición de $a \oplus b$ vale 1, comenzando por el bit menos significativo. En el nodo origen nos fijamos en el segundo bit menos significativo (0101), correspondiente al primer 1 de la distancia Hamming situado en el segundo bit menos significativo, y nos movemos al nodo adyacente que en esa posición tiene un bit distinto del bit del nodo origen (es decir, al nodo 0111). Repetimos la operación fijándonos en el tercer bit menos significativo del nodo actual (0111), correspondiente al siguiente 1 en la distancia Hamming, y nos movemos al nodo adyacente que tenga un bit distinto en esa posición (es decir, al nodo 0011). Desde el nodo actual repetimos la operación, fijándonos en el cuarto bit menos significativo (0011), correspondiente al último 1 de la distancia Hamming, y nos movemos al nodo adyacente que tenga un bit distinto en esa posición (el nodo 1011). Obsérvese que debido a que $a \oplus b$ tiene como máximo d bits que valen 1, el camino más corto entre dos procesadores de un hipercubo consta como máximo de d enlaces.

Generalizando la nomenclatura de las topologías se definen las redes d-cubo k-arias. En ellas, d es la dimensión de la red y k es el número de procesadores en cada dimensión. A este factor, k, se le denomina *radio*. Así por ejemplo, un hipercubo de dimensión d, que no es sino un mesh d-dimensional con dos procesadores en cada dimensión, sería un d-cubo 2-ario. Por otro lado, un anillo sería un 1-cubo p-ario. Ambas topologías serían los extremos de la topología general d-cubo k-arias. Finalmente, obsérvese que en las redes d-cubo k-arias el número total de procesadores será k^d.

4.4.1.2. Caracterización de redes estáticas

En general son cuatro los parámetros que caracterizan una red estática: el diámetro, la conectividad, el ancho de banda de bisección y el coste. La Tabla 4.1 muestra el valor de los parámetros para diferentes redes estáticas.

El *diámetro* de la red se define como la máxima distancia entre dos procesadores cualesquiera, entendiéndose por distancia el mínimo camino entre ellos, es decir, el camino que los une con el menor número de enlaces. Cuanto menor sea la distancia, más rápidas serán las comunicaciones ya que a mayor distancia se necesita más tiempo de comunicación. En este sentido, el diámetro determina el peor de los casos.

La *conectividad* de una red es una medida de la multiplicidad de caminos entre dos procesadores. Cuanto mayor sea mejores prestaciones se obtienen ya que es menor la contención en las comunicaciones. Como medida de la conectividad se suele tomar la *conectividad de arco*, que es el menor número de arcos que deben eliminarse para obtener dos redes disjuntas.

Los parámetros más inmediatos para establecer la rapidez de las comunicaciones son: el *ancho de canal*, la *velocidad de canal* y el *ancho de banda*. Se define el ancho de canal como el número de bits que pueden transmitirse simultáneamente por el canal que comunica dos procesadores, que es igual al número de cables físicos del enlace. La velocidad máxima con que se puede emitir por cada cable físico

Tabla 4.1: Características de las redes estáticas.

Red	Diámetro	Conectividad de arco	Ancho de bisección	Coste = número de enlaces
Completamente conectada	1	$p-1$	$\frac{p^2}{4}$	$\frac{p(p-1)}{2}$
Estrella	2	1	1	$p-1$
Árbol binario	$2\log\frac{p+1}{2}$	1	1	$p-1$
Array lineal	$p-1$	1	1	$p-1$
Anillo	$\lfloor\frac{p}{2}\rfloor$	2	2	p
Mesh bidimensional	$2(\sqrt{p}-1)$	2	$\sqrt{p}$	$2(p-\sqrt{p})$
Mesh bidimensional cerrado	$2\lfloor\frac{\sqrt{p}}{2}\rfloor$	4	$2\sqrt{p}$	$2p$
Hipercubo	$\log p$	$\log p$	$\frac{p}{2}$	$\frac{p\log p}{2}$
k-aria d-cubo cerrada	$d\lfloor\frac{k}{2}\rfloor$	$2d$	dp	dp

se conoce como *velocidad del canal*.

Con todo ello se define el *ancho de banda del canal* como la velocidad máxima con la que los datos pueden enviarse entre dos enlaces de comunicación y es, evidentemente, el producto de la velocidad del canal y el ancho de canal. Sin embargo, con este parámetro no se caracteriza toda la red. Para ello se define el *ancho de bisección* como el mínimo número de enlaces de comunicación que deben eliminarse para que la red quede dividida en dos partes iguales. Con ello, se define el *ancho de banda de bisección* como el menor volumen de comunicaciones permitidas entre dos mitades cualesquiera de la red con igual número de procesadores. Así, el ancho de banda de bisección es el producto del ancho de bisección y del ancho de banda del canal.

Por último, el *coste* puede medirse de muy diversas formas. La más general consiste en contar el número de enlaces de comunicación o la cantidad de cableado necesario en la red. Por otro lado, el ancho de banda de bisección también suele utilizarse como medida de coste ya que establece el menor número de comunicaciones.

4.4.1.3. Redes dinámicas

Como hemos visto, las redes de interconexión estáticas pueden proporcionar soluciones a problemas específicos en el diseño de computadores paralelos. Sin embargo, cuando se quiere diseñar un sistema paralelo de propósito general la opción más recomendable es utilizar una red de interconexión dinámica. Este tipo de redes de interconexión pueden adaptarse para las necesidades de comunicación demandadas por los procesadores del sistema en diferentes ámbitos. Las redes dinámicas pueden clasificarse en los siguientes tipos:

- Redes basadas en bus.
- Redes crossbar (o matriciales).

- Redes multietapa.

A continuación, se detallan las características de cada uno de éstos tipos de red y su método de funcionamiento.

Redes basadas en bus

Las redes basadas en bus son quizá la topología de red más sencilla. En ella todos los nodos comparten un único medio de comunicación, el *bus*. El bus está compuesto por un conjunto de lineas de conexión y conectores a cada procesador. La Figura 4.18 muestra la arquitectura de un sistema multiprocesador conectado por medio de un bus.

En un cierto instante de tiempo solo un único procesador puede transmitir información por el bus. La colisión de peticiones de acceso al bus se soluciona usando una lógica de arbitraje o módulo de arbitraje. Este módulo se encarga de asignar el acceso al bus a los diferentes procesadores que lo soliciten siguiendo una política de asignación previamente determinada. Entre las políticas más usadas podemos encontrar, entre otras, la prioridad fija, FIFO (*First In First Out - Primero en Entrar Primero en Salir*), *Round Robin* y LRU (*Least Recently Used - Menos Usando Recientemente*).

El ancho de banda del bus es limitado, resultando del producto de su frecuencia de reloj por el número de lineas de comunicación existentes. Este ancho de banda debe ser adecuado para atender las necesidades de los procesadores que se conectan a él. Dado que el ancho de banda es limitado,

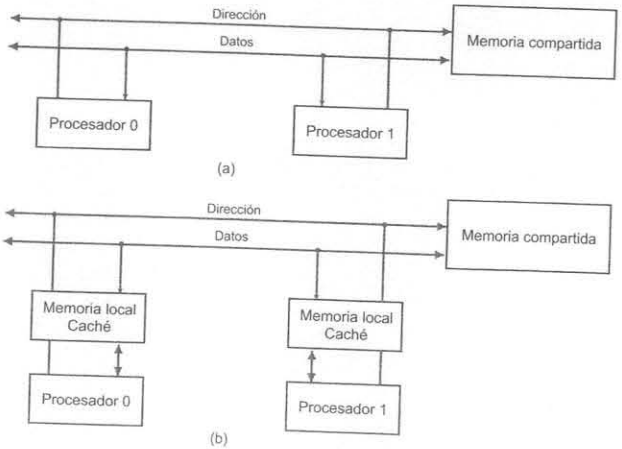


Figura 4.18: Sistema multiprocesador con conexión mediante bus.

el rendimiento de la comunicación por medio de un bus se ve afectado a medida que el número de procesadores conectados a él se incrementa. El rendimiento se puede mejorar incluyendo una caché en cada procesador (tal como se muestra en la Figura 4.18.b), ya que la mayoría de accesos de un procesador se realizan a su memoria local. De esta manera, solo se transmitirían por el bus los datos remotos necesarios para el procesador, usando la caché para datos locales.

Redes crossbar

Otra manera sencilla de conectar múltiples procesadores y elementos de memoria es utilizar una red tipo crossbar. Una red crossbar permite conectar p procesadores con q elementos de memoria utilizando una matriz de *conmutadores*, tal y como se muestra en la Figura 4.19. De manera similar también se puede utilizar una red crossbar para conectar procesadores entre si.

El número de conmutadores necesarios para realizar una red crossbar es $p \times q$, asumiendo que q es al menos igual que p (es decir, hay al menos tantos elementos de memoria como procesadores). De lo contrario, podría darse la situación de que algún procesador no pueda utilizar ninguno de los elementos de memoria. La complejidad y el coste de la red aumentan en un orden aproximado igual o mayor que $O(p^2)$, es decir, el número de conmutadores necesarios para una red con 4 procesadores serían 16 (4^2), con 8 procesadores serían 64 (8^2), con 16 procesadores serían 256 (16^2), etc. El número de conmutadores

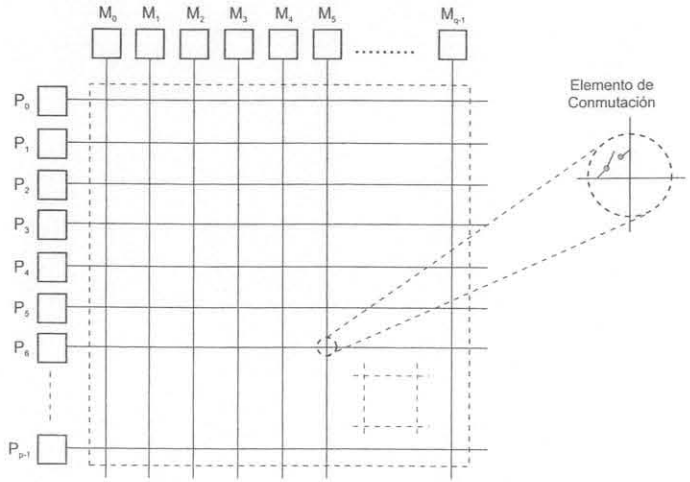


Figura 4.19: Red crossbar que conecta p procesadores con q elementos de memoria.

y sus conexiones crecen a un ritmo cuadrático según aumenta el número de procesadores, lo que supone una clara desventaja para este tipo de redes y reduce su escalabilidad.

Por otro lado, las redes crossbar son de tipo no-bloqueantes, ya que el acceso de un procesador a un elemento de memoria no interfiere en la conexión de otro procesador con otro elemento de memoria. La latencia de comunicación entre elementos en una red crossbar es constante, ya que la comunicación entre elementos se puede considerar como un bus punto-a-punto.

Redes multietapa

Las redes basadas en bus son escalables en coste pero no escalables en rendimiento. Las redes crossbar son escalables en rendimiento pero no escalables en coste. Las redes multietapa suponen un compromiso intermedio entre las dos opciones anteriores.

La estructura general de una red multietapa de p procesadores y q elementos de memoria se muestra en la Figura 4.20. Básicamente, se componen de una serie de etapas (G_i) compuestas de conmutadores conectados a las etapas adyacentes mediante conexiones estáticas (C_j). El número de etapas y el tipo de

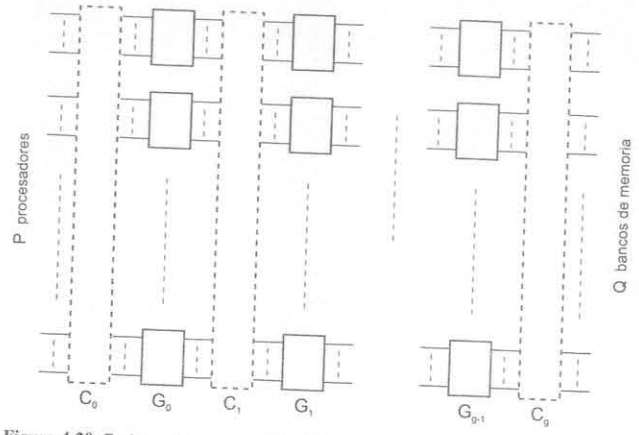


Figura 4.20: Red multietapa generalizada formada por g etapas y $g + 1$ conexiones.

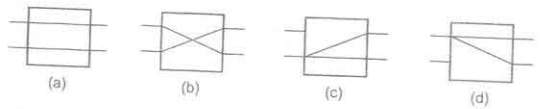


Figura 4.21: Modos de un conmutador binario: a) directo, b) cruzado, c) difusión superior, d) difusión inferior.

conexiones depende de la implementación concreta de la arquitectura.

Un conmutador $a \times b$ es un dispositivo con a entradas y b salidas (normalmente tanto a como b suele ser una potencia de 2). Los conmutadores con igual número de entradas y salidas se dice que tienen un *orden* $= a = b$. Por ejemplo, un conmutador binario sería un conmutador 2×2, con orden $= 2$ y sus posibles modos de conmutación se muestran en la Figura 4.21. Como podemos observar en la figura, se permite conectar una entrada con varias salidas pero no la operación contraria ya que daría como resultado una colisión en la comunicación.

La flexibilidad de las redes multietapa viene de la posibilidad de reconfigurar dinámicamente los modos de conmutación de sus diferentes etapas. La principal diferencia entre las distintas redes multietapa recae en el tipo de conmutadores utilizados y el patrón de conexión entre etapas, G y C respectivamente en la Figura 4.20.

Una propiedad importante de las redes multietapa es que son *redes bloqueantes*. Esto significa que ciertas permutaciones, o conexiones a través de la red, pueden a su vez bloquear otras conexiones. En la Figura 4.23 se muestra un ejemplo de esta situación. La conexión entre el procesador dos (010) y el elemento de memoria siete (111) bloquea la conexión entre el procesador seis (110) y el elemento de memoria cuatro (100).

Algunos ejemplos de las redes multietapa más comunes son:

- *Red Omega*, se basa en la utilización de una permutación por barajamiento perfecto entre sus etapas. Estas etapas están compuestas por conmutadores binarios.

  La permutación por barajamiento perfecto (*perfect suffle*), σ^k, cuyo dominio es el conjunto de enteros $[0, n-1]$, se define como

$$\sigma^k(x_{m-1}x_{m-2}\ldots x_1x_0) = x_{m-2}\ldots x_1x_0x_{m-1}$$

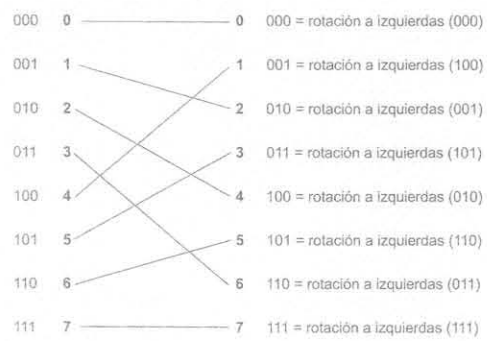


Figura 4.22: Barajamiento perfecto en una conexión de ocho entradas y ocho salidas.

siendo k el orden del conmutador utilizado y $x_{m-1}x_{m-2}\ldots x_1x_0$ la representación binaria del entero x. Esta permutación supone un desplazamiento a la izquierda de los bits que representan el número x, tal y como se muestra en la Figura 4.22. También existe un barajamiento perfecto inverso, en el que se efectúa la misma operación pero con un desplazamiento a la derecha.

La red omega está compuesta de p nodos de entrada, normalmente procesadores, y p nodos de salida, normalmente elementos de memoria, habiendo un total de $\log_2 p$ etapas entre ambos. Las conexiones de barajamiento perfecto entre etapas proporcionan una única conexión entre cada par de componentes. La Figura 4.23 muestra un ejemplo de una red omega que conecta ocho procesadores con ocho elementos de memoria.

Dadas las características de la red omega, el número de conmutadores necesarios para construir la red es $\frac{p}{2}\log_2 p$, siendo el coste de la red de orden $O(p\log_2 p)$. Para una red omega con 4 procesadores el número de conmutadores necesarios será 4 ($\frac{4}{2}\log_2 4 = 2\times 2$), para 8 procesadores será 12 ($\frac{8}{2}\log_2 8 = 4\times 3$), para 16 procesadores será 32 ($\frac{16}{2}\log_2 16 = 8\times 4$), etc. Este ratio de crecimiento, y por tanto su coste, es menor que el de una red crossbar, que como vimos anteriormente crece con orden $O(p^2)$.

El algoritmo de encaminamiento usado para comunicar nodos en una red omega resulta bastante sencillo. Cada paquete transmitido va precedido de la dirección del nodo destino en binario. Los conmutadores de cada etapa deciden el camino por el que transmitir el paquete dependiendo del

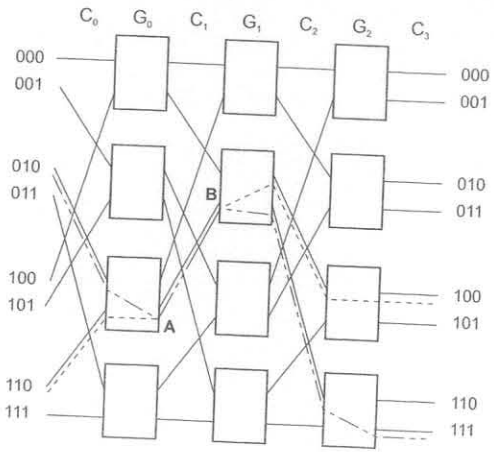


Figura 4.23: Red omega de 8 entradas y 8 salidas.

valor del bit de la dirección destino correspondiente a la etapa actual. Si el bit es 0, se encamina por la salida superior, y si es 1, se utiliza la salida inferior. Por ejemplo, en la red de la Figura 4.23, supongamos que el procesador 110 quiere enviar un paquete al elemento de memoria 100. El conmutador de la primera etapa, marcado con una A en la figura, ve que el bit que le corresponde de la dirección destino es un 1 (**1**00), por lo que encamina el paquete por la salida inferior hacia el conmutador marcado con una B en la figura. El conmutador B observa que el bit que le corresponde es un 0 (1**0**0), por lo que encamina el paquete por su salida superior. Finalmente, el último conmutador también observa su bit correspondiente es 0 (10**0**) y encamina el paquete por su salida superior, llegando éste a su destino correcto.

- *Red Baseline*, se construye recursivamente por bloques, conectando los conmutadores de la etapa i con ambos sub-bloques de la etapa $i+1$. Así, la primera etapa se construye con un bloque que tendrá un tamaño $n\times n$, siendo n el número de entradas. La segunda etapa se construye con dos sub-bloques de tamaño $(\frac{n}{2})\times(\frac{n}{2})$, y así recursivamente hasta llegar en la última etapa a $\frac{n}{2}$ sub-bloques de tamaño 2×2. La Figura 4.24 muestra un ejemplo de red baseline. Observe los bloques

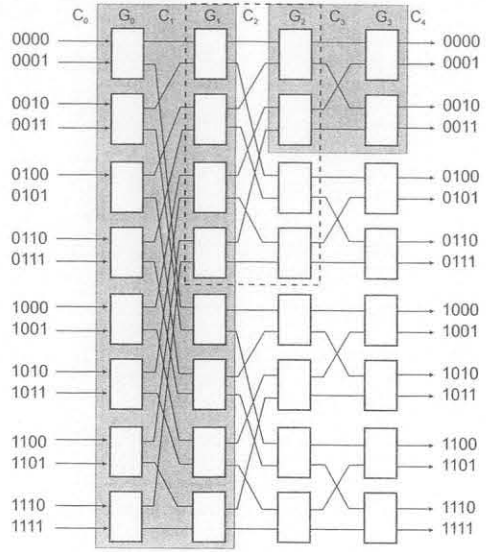


Figura 4.24: Red baseline de 16 entradas y 16 salidas.

y sub-bloques destacados en la figura.

El encaminamiento de paquetes en una red baseline es análogo al de la red omega.

- *Red Butterfly*, se construye usando el modelo de permutación básico mostrado en la Figura 4.25.a. Este modelo se puede extender a una red más amplia aplicando sistemáticamente los cruces de entradas y salidas de sus conmutadores (tal y como se detalla en la Figura 4.25.b). De manera general, en la construcción de una red butterfly, las salidas de un conmutador j en la etapa i (identificado como $[i, j]$) se conectarán a los conmutadores $[i + 1, j]$ e $[i + 1, j \oplus 2^i]$ (es decir, difieren en el i-ésimo bit). La Figura 4.25.b muestra un ejemplo de red butterfly con 16 entradas y 16 salidas.

  El encaminamiento de paquetes en una red butterfly se realiza de la siguiente manera. Siendo A el conmutador conectado al nodo de origen y B el conmutador conectado al nodo destino, ambos representados en binario, se calcula la ruta $R = A \oplus B$. La ruta entre A y B se define eligiendo el camino directo en el conmutador de la etapa i si el bit $R_i = 0$, o el cruzado si el bit $R_i = 1$ (siendo R_0 el bit menos significativo). Por ejemplo, los conmutadores de la red de la Figura 4.25.b se pueden representar y nombrar tal y como muestra la Figura 4.25.c. Queremos enviar un paquete desde el nodo 0100 al nodo 1000. El conmutador conectado al nodo 0100 es el 010 y el conmutador conectado al nodo de llegada 1000 es el 100. Se calcula la ruta entre conmutadores $R = 010 \oplus 100 = 110$. Con esto se tiene que desde el conmutador 010 se debe tomar la conexión directa al conmutador A. Desde A se toma la conexión cruzada al conmutador B. Finalmente, desde B se toma también la conexión cruzada para llegar al conmutador 100, que conecta con el destino final.

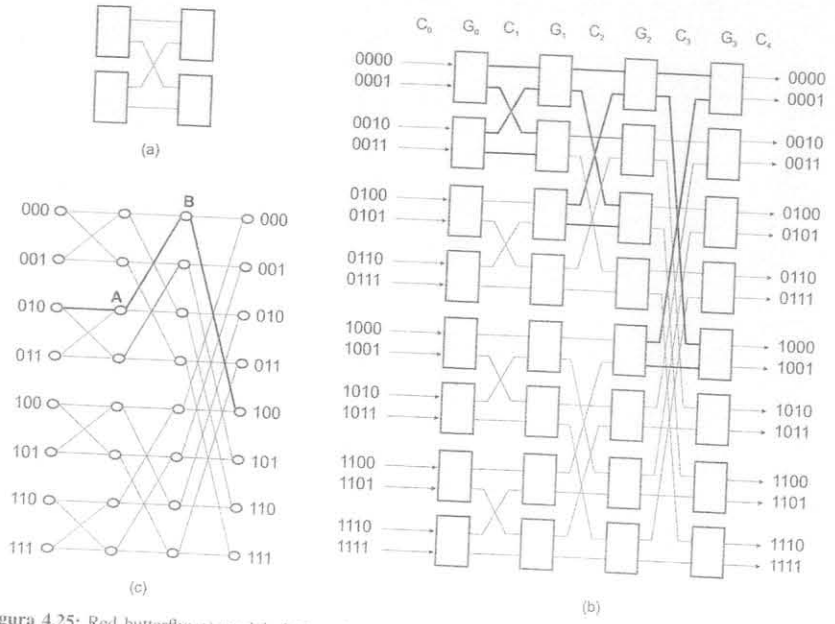


Figura 4.25: Red butterfly: a) modelo básico de permutación, b) red butterfly de 16 entradas y 16 salidas, c) diagrama de conmutadores de la red butterfly.

4.4.1.4. Comparación del rendimiento de redes dinámicas

A pesar de ser redes bloqueantes, las redes multietapa son una solución aceptable teniendo en cuenta el coste y las prestaciones de las arquitecturas bus y crossbar. En la Figura 4.26 se muestra el coste y las prestaciones en función del número de procesadores para las tres arquitecturas.

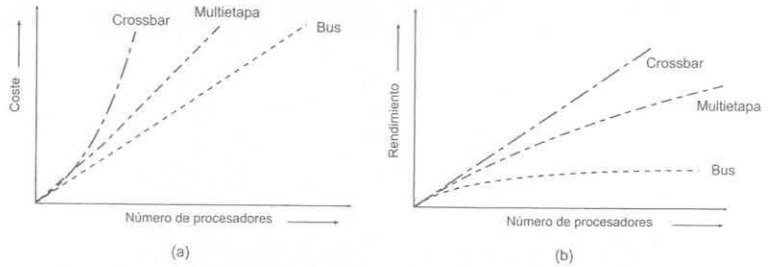


Figura 4.26: Comparativa de las arquitecturas crossbar, bus y multietapa: a) Coste, b) Prestaciones.

Como puede observarse, la arquitectura multietapa se encuentra en un lugar intermedio en cuanto al coste, y lo que es más importante, crece casi linealmente en vez de cuadráticamente como en la red crossbar. Análogamente, en relación con las prestaciones también se encuentra en una situación intermedia, pero con la ventaja de que no se produce la drástica saturación del caso del bus.

4.4.2. Protocolos de coherencia de caché

Cuando se trata con sistemas multiprocesador se puede comprobar que el uso de las memorias cachés para mejorar el rendimiento del sistema es generalizado. Con la utilización de las cachés se pretende conseguir, en definitiva, reducir el tiempo de latencia de la memoria.

Evidentemente, en los sistemas multiprocesador, donde cada procesador puede necesitar una copia del mismo bloque de memoria, surgen problemas nuevos, en concreto el problema de consistencia y

coherencia entre las distintas copias utilizadas. Es decir, varios procesadores pueden guardar en sus respectivas cachés locales una copia de un mismo bloque de memoria. Este bloque es modificado por cada procesador, por lo que es necesario que los cambios realizados se comuniquen al resto de procesadores, actualizando las diferentes copias de las cachés locales.

Este problema se denomina *problema de coherencia de caché*. Un sistema de memoria es coherente si el valor devuelto por una operación de lectura sobre una dirección de memoria es siempre el mismo valor que el almacenado por la última operación de escritura realizada sobre esa misma dirección, independientemente de qué procesador realice las operaciones.

La solución más sencilla para el problema de coherencia de caché en sistemas de memoria compartida es, simplemente, no incorporar un protocolo de coherencia de caché y centrarse en la escalabilidad del sistema. De esta manera, solo se permiten dos tipos de accesos a memoria:

- *Local*, en el que el procesador accede a datos privados y puede utilizar una caché local.
- *Remoto*, en el que el procesador accede a datos externos que no se almacenan en caché. En este caso dichos datos pueden ser accedidos utilizando, por ejemplo, la técnica de paso de mensajes.

Los problemas de coherencia de caché están provocados en la práctica básicamente por tres factores:

- Por *modificar datos compartidos*. En un sistema multiprocesador donde cada procesador tiene una caché local puede darse la situación de que dos o más procesadores estén accediendo simultáneamente a la misma estructura de datos. En esta situación los datos se encontrarán en la memoria principal compartida y en las cachés de cada procesador.

  Si un procesador modifica su caché local, el contenido de la memoria principal y las copias de las cachés no serán coherentes. Si el sistema utiliza una política de escritura directa, la memoria principal se actualizará pero no así las copias de caché del resto de procesadores. Si el sistema utiliza una política de post-escritura, la memoria principal no se actualizará hasta que se reemplace el bloque de caché modificado. Ambas situaciones llevan a un problema de incoherencia de memoria.

- Por *migración de procesos*. En algunos sistemas multiprocesador se permite la migración de procesos, es decir, un proceso puede ser planificado en diferentes procesadores durante su tiempo de vida. Durante el tiempo de ejecución en un procesador dicho proceso realiza modificaciones en los datos que usa. Estas modificaciones son almacenadas en la caché local del procesador. Si por alguna razón, el proceso es intercambiado a otro procesador antes de que las modificaciones realizadas se actualicen en la memoria principal, los datos que cargue el proceso en la caché del nuevo procesador serán incoherentes con las modificaciones realizadas en el anterior procesador. Por ejemplo, si el proceso A se está ejecutando en el procesador P_1 y asigna la variable $V = 10$ (suponiendo que su valor anterior fuese 0) y por alguna razón es intercambiado a otro procesador P_2 antes de que la memoria principal se actualice, el valor de V que obtiene A será 0 ya que el valor 10 solo está almacenado en la caché de P_1.

- Por el *uso de Entrada/Salida mediante Acceso Directo a Memoria*. Mediante el uso de la técnica de Acceso Directo a Memoria (*DMA - Direct Memory Access*) el procesador de E/S a través del controlador de DMA transporta datos desde los periféricos a la memoria principal del sistema.

  En el caso de la entrada de datos, los datos escritos por el procesador de E/S en la memoria principal pueden ser inconsistentes con las copias existentes en las cachés locales de los procesadores del sistema.

  En el caso de la salida de datos, si se utiliza una política de post-escritura se puede generar inconsistencia ya que el procesador de E/S estaría leyendo datos que puede que todavía no hayan sido actualizados desde las cachés locales.

  Una posible solución a este problema sería configurar los procesadores de E/S para que actuasen directamente sobre las cachés locales de los procesadores, en lugar de sobre la memoria principal. De esta manera las operaciones de E/S no generarían inconsistencias, aunque la coherencia de las cachés se debería seguir garantizando a través del bus del sistema. El inconveniente de esta solución es la pobre localidad de los datos de E/S en las cachés, lo que incrementa la tasa de fallos de las mismas.

Existen principalmente dos opciones para solucionar los problemas de incoherencia de caché en sistemas multiprocesador:

- *Invalidar*, que consiste en invalidar las copias en las cachés del dato modificado (ver Figura 4.27.a).

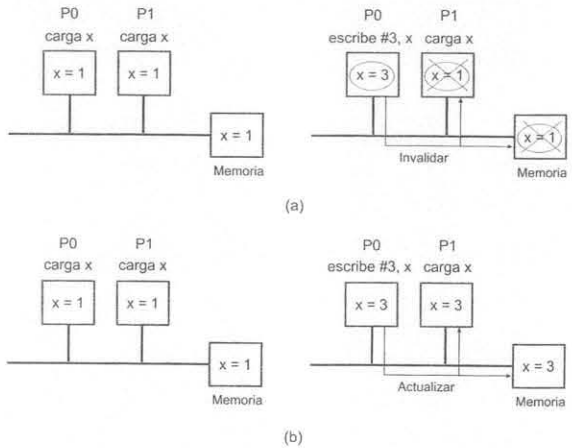


Figura 4.27: Protocolos de coherencia de caché en sistemas multiprocesador: a) invalidación, b) actualización.

- *Actualizar*, que consiste en actualizar todas las copias en las cachés del dato que se acaba de modificar (ver Figura 4.27.b).

Las principales desventajas de estas dos técnicas son que la primera requiere la espera de acceso al dato causada por la carga del valor correcto de la variable invalidada (que se realiza desde la caché que contiene el dato correcto) y la segunda provoca un mayor tráfico de datos entre los procesadores debido a las operaciones de actualización de las cachés. En la actualidad la mayoría de los protocolos y políticas de coherencia de caché utilizan un esquema basado en invalidar datos.

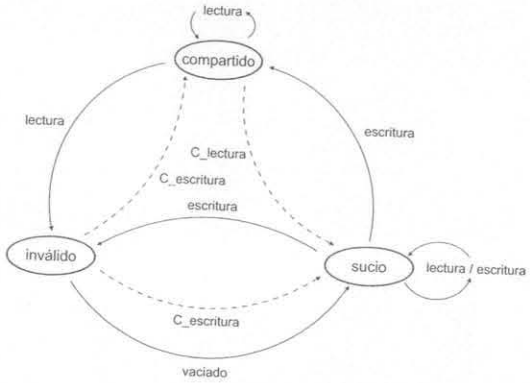


Figura 4.28: Diagrama de estados de un protocolo simple de coherencia de caché basado en invalidación.

Un posible método para mantener la coherencia de las diferentes copias de caché podría ser la monitorización del número de copias existentes y el estado de cada copia. La Figura 4.28 muestra un posible conjunto de estados y transiciones para los datos almacenados en caché. Las líneas sólidas representan acciones de los procesadores y las discontinuas representan acciones para asegurar la coherencia de caché. El estado *compartido* corresponde a una variable que ha sido cargada en las cachés de varios procesadores (por ejemplo, cuando se carga la variable al inicio de un programa). Cuando un procesador modifica una variable *compartida* ésta pasa al estado *sucio*, y todas las copias de esa variable en las cachés de otros procesadores pasan al estado *inválido*. Para garantizar la coherencia, todos los accesos sobre una variable *inválida* deberán ser servidos por el procesador que tiene dicha variable en el estado *sucio*, en lugar de ser servidos por la memoria principal. A su vez, las lecturas y escrituras realizadas por los procesadores generan acciones de coherencia (*C_lecturas* y *C_escrituras*). Si un procesador realiza una lectura sobre una variable *inválida* se propaga una acción *C_lectura* que actualiza el resto de copias, además de la memoria principal, y las devuelve al estado *compartido*. Si un procesador realiza una escritura sobre una variable *inválida* se propaga una acción *C_escritura*

para invalidar el resto de copias y posteriormente pasar dicha variable al estado *sucio*. De la misma manera, como ya se ha mencionado, si un procesador realiza una escritura sobre un bloque *compartido* se generará una acción *C_escritura* para invalidar el resto de copias en las cachés, pasando la variable modificada al estado *sucio*. Finalmente, cuando un procesador vacía su caché todos los bloques pasan al estado *inválido*.

Por ejemplo, siguiendo el sistema mostrado en la Figura 4.27.a, cuando P_1 acceda a la variable x tras haber sido modificada por P_0, deberá ser P_0 el que actualice el valor en la caché de P_1 y en la memoria principal. A su vez, la variable x en P_1 pasaría del estado *inválido* al *compartido* ya que su valor actual es coherente.

Diferentes mecanismos hardware implementan el protocolo de coherencia de caché basado en invalidación descrito anteriormente:

- Sistemas *snoopy* o de vigilancia del bus. Este mecanismo es muy común en sistemas multiprocesador que usan una red de difusión, tales como un bus o un anillo. Su funcionamiento se basa en que cada procesador monitoriza el tráfico de la red en busca de transacciones, para poder actualizar el estado de sus bloques de caché. La Figura 4.29 muestra un esquema simple para un sistema snoopy. La caché de cada procesador tiene asociadas unas etiquetas que usa para determinar el estado de sus bloques de caché. Por ejemplo, si el procesador detecta una escritura en un bloque del cual mantiene una copia, éste invalida su copia siguiendo el protocolo descrito. Si, por el contrario, detecta una lectura en un bloque que tiene marcado como sucio, el procesador solicita el control del bus y envía el bloque para que el valor pueda ser leído por el otro procesador.

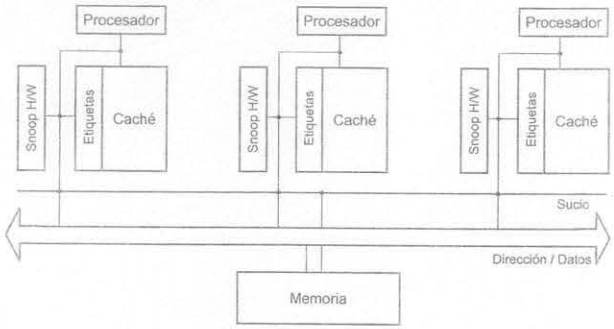


Figura 4.29: Sistema snoopy.

La principal limitación de los sistemas snoopy es la limitación de ancho de banda del bus. Si varios procesadores realizan operaciones sobre los mismos datos, las acciones de coherencia necesarias se distribuirán por el bus a todos los procesadores del sistema. Dado que el ancho de banda del bus

es finito, solo un número constante de acciones de coherencia puede ser transmitido por unidad de tiempo. Una posible solución sería propagar acciones de coherencia solo a los procesadores afectados (que tengan una copia de la variable accedida). Esta solución la proporcionan los sistemas basados en directorios.

- Sistemas *basados en directorios*. Los sistemas basados en directorios usan un mapa de bits (o bits de presencia) para almacenar la localización de las copias de caché que tiene cada procesador en su caché. Este mapa de bits puede incluirse en la memoria principal (con un sistema de *directorio centralizado* tal como muestra la Figura 4.30.a) o en el caso de sistemas escalables de memoria distribuida puede incluirse en las memorias locales de cada procesador (obteniendo un sistema de *directorio distribuido* como se muestra en la Figura 4.30.b). Este tipo de sistemas usa el contenido del mapa de bits para propagar las acciones de coherencia del protocolo únicamente a los procesadores que mantienen una copia del bloque afectado. Otras acciones pueden afectar únicamente a los datos locales de un procesador y generar cambios de estado en el mapa de bits. Por ejemplo, si dos procesadores acceden al mismo bloque de datos, su estado pasa a compartido y su bit de presencia es activado para ambos procesadores (indicando que los dos procesadores guardan

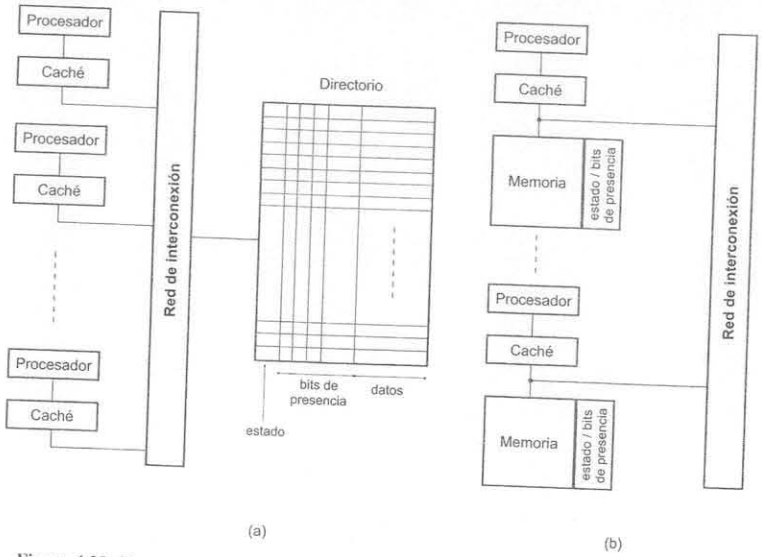


Figura 4.30: Sistema basado en directorios: a) directorio centralizado, b) directorio distribuido.

una copia del bloque). Si uno de los procesadores modifica el bloque, su estado pasa a sucio y el bit de presencia correspondiente al otro procesador se desactiva. De esta manera, el sistema mantiene el bit de presencia para el procesador que modificó el bloque. Posteriores modificaciones del bloque por parte del mismo procesador se resuelven localmente. Cualquier acceso posterior a ese bloque por parte de otro procesador, dado que está en estado sucio, deberá ser servido por el procesador que tiene el bit de presencia activo.

En los sistemas de directorio centralizado el principal cuello de botella está en la memoria, ya que todas las acciones de coherencia representan accesos a la misma. La memoria del sistema solo puede servir un número finito de peticiones por unidad de tiempo, pudiendo reducir considerablemente el rendimiento del sistema. También hay que considerar en este caso el coste del mapa de bits en memoria, ya que crece con orden $O(mp)$, siendo m el tamaño de la memoria y p el número de procesadores.

Los sistemas de directorio distribuido permiten p acciones de coherencia simultáneas. Sin embargo, dado que los accesos a memorias remotas se realizan a través de la red de interconexión, la latencia y el ancho de banda de dicha red representan partes fundamentales del rendimiento general del sistema.

4.5. Sistemas de memoria distribuida

En los sistemas de memoria distribuida cada procesador dispone de su propia memoria, denominada local o privada, independiente del resto y accesible solo por su procesador. La comunicación se realiza por paso de mensajes, es decir, para que un dato que reside en la memoria de un procesador pase a la de otro, el primero debe construir un mensaje por software, enviarlo a través de una red de interconexión y el segundo debe recibirlo. Como puede observarse es un mecanismo más complejo que con memoria compartida. La Figura 4.31.a muestra el esquema básico de la arquitectura distribuida. Esta arquitectura es también conocida como *arquitectura de memoria privada* o *arquitectura de paso de mensajes*.

La misión de la red de interconexión es facilitar el paso de mensajes entre los procesadores nodo. La Figura 4.31.b muestra un ejemplo real, en el que los nodos son procesadores IBM SP-2, y la conexión entre ellos se realiza mediante un *switch* (conmutador) de ocho puertos.

El concepto de paso de mensajes parece ser la estrategia dominante en los sistemas con gran número de procesadores (mayor que 100), y es especialmente útil en entornos donde la ejecución de los programas puede dividirse en pequeños subprogramas independientes. La latencia de red puede ser alta, pero para analizar el ancho de banda es necesario atender a otro factor de eficiencia de los sistemas paralelos conocido como granularidad del computador paralelo. En general, se entiende por granularidad del computador paralelo el cociente entre el tiempo requerido para realizar una operación básica de comunicación y el tiempo requerido para realizar una operación básica de cálculo de los procesos. Para un tiempo dado de operación básica de cálculo ofrece una medida del número y del tamaño de los paquetes de información utilizados en las comunicaciones. En los sistemas de paso de mensajes para

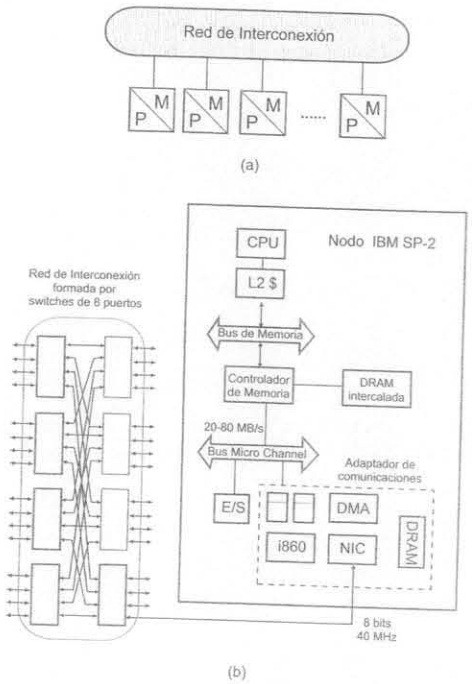


Figura 4.31: a) Arquitectura de memoria distribuida, b) IBM SP-2.

lograr un uso adecuado del ancho de banda es necesario realizar un cuidadoso reparto de los datos sobre los procesadores con el fin de disminuir la granularidad de las comunicaciones. Por disminuir la granularidad de las comunicaciones se entiende minimizar el número de mensajes y maximizar su tamaño. Evidentemente, aunque el concepto genérico puede aplicarse a sistemas SIMD, también es aplicable a sistemas MIMD. A la combinación de sistema MIMD con arquitectura de paso de mensajes se le conoce como *multicomputadores*.

Los sistemas con memoria distribuida o multicomputadores pueden, a su vez, ser un único computador con múltiples CPUs comunicadas por un bus de datos o bien múltiples computadores, cada uno con su propio procesador, enlazados por una red de interconexión más o menos rápida. En el primer caso se habla de *procesadores masivamente paralelos* (MPPs - *Massively Parallel Processors*), como

los Fujitsu VPP, IBM SP2 o SGI T3E; y en el segundo se conocen de forma genérica como *cluster*. Un cluster, a nivel básico, es una colección de estaciones de trabajo o PCs interconectados mediante algún sistema de red de comunicaciones. En la literatura existente de arquitecturas paralelas se utilizan numerosos nombres para referirse a un cluster, entre los que destacan:

- Redes de estaciones de trabajo (*NOWs - Network of Workstations*).
- Redes de PCs (*NOPCs - Network of PCs*).
- Cluster de estaciones de trabajo (*COWs - Cluster of Workstations*).
- Cluster de PCs (*COPCs - Cluster of PCs*).

Realizando una clasificación estricta, los clusters pueden ser de dos tipos dependiendo de si cada computador del cluster está o no exclusivamente dedicado a él. Si es así, se habla de un cluster de clase *Beowulf*. En cualquier otro caso se suele definir al cluster como *NOW*. En muchas ocasiones los términos cluster y Beowulf se confunden y se utilizan indistintamente. El término Beowulf lo utilizó la NASA para dar nombre a un proyecto que pretendía construir un ordenador capaz de alcanzar el gigaflops a partir de componentes asequibles. Y lo consiguieron, el Beowulf original de 1994, con 16 procesadores 486 DX4 ejecutando Linux, fue capaz de alcanzar 1,25 Gigaflops. El término no se debe a razones técnicas. Beowulf era un guerrero escandinavo del siglo VI cuyas aventuras se relatan en el primer texto conocido en lengua inglesa (similar al Cantar del Mío Cid en la lengua española). Las características más relevantes de los sistemas Beowulf son las siguientes:

- Un sistema Beowulf es un conjunto de nodos minimalistas[2] conectados por un medio de comunicación barato, en el que la topología de la red se ha diseñado para resolver un tipo de problema específico.
- Cada nodo de un Beowulf se dedica exclusivamente a procesos del supercomputador.
- En una red de estaciones de trabajo (NOWs) suele existir un switch central para realizar las comunicaciones, mientras que en un Beowulf el mecanismo es más rudimentario: conexiones placa a placa por cable RJ-45 cruzado.
- La programación de un Beowulf es fuertemente dependiente de la arquitectura y siempre se realiza por paso de mensajes.
- Para programar un Beowulf en primer lugar se diseña el modelo de paralelismo, se observan cómo son las comunicaciones entre los nodos y se implementan físicamente. De esta forma se evita el hardware innecesario a cambio de una fuerte dependencia entre el software y la topología de la red. En caso de que cambie el problema hay que recablear el sistema.

[2] Los nodos minimalistas, habitualmente, constan solamente de una placa madre, una CPU, las memorias y algún dispositivo de comunicaciones.

4.5.1. Consideraciones generales sobre los clusters

Tal y como se ha comentado, un cluster a nivel básico es un conjunto de estaciones de trabajo o PCs interconectados mediante algún sistema de red de comunicaciones. Teniendo presentes las diferencias anteriormente comentadas, para los objetivos que se pretenden resulta irrelevante la distinción entre las dos clases de sistemas: cluster y Beowulf, por ello se ha optado por utilizar el término cluster para hacer referencia a ambos.

Las arquitecturas paralelas en general, y los clusters en particular, pueden presentar distintas topologías en función de las conexiones punto a punto que se establezcan entre sus procesadores. Las topologías pueden ser muy sencillas, como las redes lineales, de anillo o en estrella, pero también pueden realizarse redes más complejas como los hipercubos.

La implementación más natural y habitual de los clusters es como parte de una red de área local, en la que las máquinas que forman el cluster se conectan a una red de alta velocidad y la máquina que actúa como servidor se conecta además a la red exterior.

En este contexto debe tenerse en cuenta un factor adicional que afecta significativamente al rendimiento final del cluster: el dispositivo de conexión a la red o adaptador. Como adaptador se puede utilizar un hub o un switch, siendo este último el más habitual.

Con un switch la transmisión de datos entre dos procesadores solo genera tráfico en el segmento correspondiente. El ancho de banda no es compartido entre todos los procesadores conectados al switch por lo que cada procesador dispone del 100 % del ancho de banda. El switch permite establecer cualquier número de interconexiones simultáneas entre sus puertos siempre y cuando no coincida el receptor. Con un hub la transmisión de datos entre dos procesadores genera tráfico por toda la red, consumiendo el ancho de banda compartido entre todos los demás, lo que tiene como consecuencia una saturación rápida y unas comunicaciones lentas.

4.5.2. ¿Por qué clusters?

Las exigencias de cálculo de las aplicaciones comerciales y científicas crecen día a día. Para hacer frente a esta demanda es preciso aumentar la capacidad de la unidad de procesamiento, o bien hacer que varias unidades colaboren para la resolución conjunta de una tarea. La primera alternativa es fácil de llevar a la práctica, basta con aumentar la memoria o sustituir el procesador por otro de tecnología superior y/o de mayor velocidad de reloj. Pero esto a menudo es insuficiente: ¿qué CPU se tendría que utilizar si la necesidad de potencia de cálculo se multiplicase por cien? La segunda opción, el procesamiento en paralelo, no es de implantación tan inmediata, pero escala mucho mejor con las necesidades del problema.

Recuérdese que el procesamiento paralelo consiste en acelerar la ejecución de un programa mediante su descomposición en fragmentos que puedan ejecutarse de forma paralela, cada uno en su propia unidad de proceso. La mayoría de los procesadores modernos incorporan en su diseño unidades que trabajan de forma paralela, como los que emplean la tecnología del pipelining (operar a la vez sobre las distintas fases que requiere la ejecución de una instrucción del procesador, como en una cadena de

montaje), o el procesamiento superescalar (la ejecución en paralelo de instrucciones independientes). Los denominados procesadores vectoriales que forman parte de los supercomputadores tradicionales, como el Cray C90, son capaces de operar simultáneamente sobre varios elementos de un vector. Más recientemente toma cuerpo el concepto de paralelismo sobre un registro, como el que implementan las extensiones multimedia de Intel MMX.

No obstante, desde comienzos de la década de los 90 ha existido una tendencia creciente a alejarse de los supercomputadores especializados paralelos (supercomputadores vectoriales y procesadores masivamente paralelos, MPPs), debido a sus elevados costes en hardware, mantenimiento y programación. En este contexto los clusters constituyen una alternativa de menor coste ampliamente utilizada y consolidada.

Entre los motivos que han hecho posible este hecho cabe destacar el gran progreso en la disponibilidad de componentes de un alto rendimiento para PCs/estaciones de trabajo y redes de interconexión. Gracias a estos avances, se ha logrado que un cluster sea hoy día un sistema muy atractivo en cuanto a su relación coste/rendimiento para el procesamiento paralelo. Se puede decir que los clusters de computadores se han convertido en la opción más sencilla y extendida en computación paralela. Ofrecen a cualquier institución académica una atractiva oportunidad para utilizar y enseñar *computación de altas prestaciones* (HPC - *High Performance Computing*)[3] sin requerir el acceso a equipamientos costosos.

Las principales características de estos sistemas, que les hacen tener algunas ventajas sobre otros tipos de arquitecturas paralelas, son las siguientes:

- Se pueden construir con un esfuerzo relativamente moderado.
- Son sistemas de bajo coste.
- Utilizan hardware convencional y accesible (el disponible en el mercado de consumo).
- Utilizan un sistema de comunicación basado en una red de área local rápida como Myrinet o Fast Ethernet.
- Utilizan un software de libre distribución, como Linux, y algún entorno de programación paralelo como pueden ser PVM (*Parallel Virtual Machine*) o MPI (*Message Passing Interface*).
- Son sistemas escalables, es decir, se pueden ajustar a las necesidades computacionales y permitir una ejecución eficiente teniendo en cuenta las demandas de las aplicaciones secuenciales y paralelas.

[3]Se aplica este calificativo a las aplicaciones de cómputo que requieren mayores prestaciones que las normalmente disponibles con una estación de trabajo o computador personal. Por extensión, a equipos que posibilitan la realización de dichas aplicaciones: estaciones de trabajo científicas, supercomputadores, redes de altas prestaciones, etc. Las aplicaciones típicamente HPC se suelen denominar "Grand Challenges" (Grandes Retos). Ejemplos de ellos son las predicciones meteorológicas, la superconductividad, la biología estructural, la voz y visión por computador, etc. Al avanzar la tecnología, algunas aplicaciones dejan de ser consideradas como grandes retos.

- Cada máquina de un cluster puede ser un sistema completo utilizable o aprovechable para otros propósitos.

- Reemplazar un computador defectuoso de un cluster es trivial, incluso es posible diseñar el cluster de tal forma que si un nodo falla, el resto continúe trabajando.

- El rendimiento y los recursos del cluster pueden crecer con el tiempo beneficiándose de las últimas tecnologías computacionales y de redes.

- Además, debido a la estandarización se tiene la garantía de que los programas escritos para un cluster funcionarán en cualquier otro con independencia del tipo de procesador de cada nodo sin más que recompilar el código para esa arquitectura.

Pero a pesar de estas ventajas los clusters también presentan algunas desventajas:

- Las redes ordinarias no están diseñadas para el procesamiento paralelo. La latencia de red es alta y el ancho de banda relativamente bajo si se comparan con los de un sistema SMP[4] (*Symmetric MultiProcessors*). Además, si el cluster no está aislado del resto de la red de la institución, la situación es aún peor.

- En los sistemas operativos monoprocesador existe muy poco software para tratar un cluster como un único sistema. Por ejemplo, el comando ps solo lista los procesos de un sistema Linux, no los de todo el cluster.

Indudablemente existen y han existido plataformas más potentes, más costosas y más difícilmente accesibles que los clusters de computadores. El desarrollo de software para estas plataformas es costoso, dado que debe ser programado o comprado por un reducido número de usuarios. Su vida útil suele acabar cuando aparece un nuevo tipo de supercomputador con arquitectura y procesadores superiores, lo cual obliga a iniciar de nuevo el costoso proceso de adquirir el hardware y el software.

En comparación, los clusters pueden ir incorporando computadores cada vez más potentes conforme vayan estando disponibles comercialmente, equilibrando la carga de trabajo según las prestaciones de cada computador, retirando eventualmente los computadores más antiguos si fuera conveniente, y utilizando siempre no solo el software previamente existente sino también el nuevo software que previsiblemente irá apareciendo, desarrollado por una base de usuarios cada vez más extensa (atraídos por el bajo costo del hardware y del software), con menor costo, menor esfuerzo y mayor utilización.

Por tanto, debido a la flexibilidad que ofrece este tipo de arquitectura, cada día aparecen sistemas más potentes y con mayor número de procesadores.

[4]Recuérdese que los sistemas SMP suelen tener de 2 a 64 procesadores, y se les conoce como la arquitectura que comparte todo, es decir, los procesadores utilizan conjuntamente la totalidad de los recursos que tenga disponible el sistema (bus, memoria, módulos de E/S, etc.). Además, en este tipo de sistemas únicamente se ejecuta una copia del sistema operativo.

4.5.3. ¿Cuándo y cómo utilizar un cluster?

La razón de ser del procesamiento paralelo es acelerar la resolución de un problema. La aceleración (*speedup*) que puede alcanzarse depende tanto del problema en sí como de la arquitectura del computador paralelo. Las aplicaciones que se benefician de una aceleración más significativa son aquellas que describen procesos intrínsecamente paralelos. En cualquier caso debe tenerse en cuenta el hardware de la máquina ya que es preciso maximizar la relación entre el tiempo de cálculo útil y el "perdido" en el paso de mensajes, parámetros que dependen de la capacidad de proceso de las CPUs y de la velocidad de la red de comunicaciones respectivamente. La clave consiste en descomponer el problema de tal forma que cada procesador pueda operar el mayor tiempo posible sobre su fragmento de los datos, sin tener que recurrir a los de los demás procesadores. Por ejemplo, el cálculo del área bajo una curva por integración numérica es un ejemplo de aplicación completamente paralelizable. Basta con dividir el intervalo de integración entre todos los procesadores disponibles y que cada uno resuelva su fragmento sin preocuparse de qué hacen los demás. Al final, los resultados parciales se recolectan y se suman convenientemente.

Con n procesadores es posible resolver el problema n veces más rápido que haciendo uso de uno solo (salvo por el mínimo retraso que supone el reparto de trabajo inicial y la recolección de datos final), consiguiendo una aceleración lineal con el número de procesadores. La Figura 4.32 muestra la curva de aceleración frente al número de procesadores para distintos casos de aplicaciones paralelas. Si las condiciones son muy favorables es incluso posible alcanzar la aceleración superlineal, donde el programa se ejecuta aún más rápido que en régimen lineal. La aparente paradoja se entiende recordando que cada procesador cuenta con su propia memoria principal y su memoria caché, que pueden ser utilizadas de forma más eficiente con un subconjunto de los datos. De hecho, es posible que el problema no se pueda resolver en un único procesador pero sí sobre un cluster, simplemente por cuestión de tamaño de los datos. En el ejemplo de la Figura 4.32 se ha conseguido que un cluster de seis nodos haga el trabajo de siete computadores independientes. En el extremo opuesto se encuentran los problemas que se paralelizan muy mal, es decir, necesitan estar continuamente compartiendo datos entre procesadores para obtener el resultado final. Las comunicaciones determinan el avance del programa y es posible encontrar que su ejecución en un cluster sea más lenta que en un único computador. Este caso corresponde a la curva etiquetada como "Deceleración" en la Figura 4.32.

Los programas que se ejecutan sobre arquitecturas paralelas consisten, a menudo, en un algoritmo básico que se repite muchas veces y donde entre paso y paso los procesadores intercambian información con sus vecinos. Por ejemplo, en el clásico juego de la vida en dos dimensiones cada celda sobrevive en la próxima generación si en la actual tiene exactamente dos vecinos vivos. Si se programa el juego en un cluster y se decide que cada procesador va a encargarse de una única celda, cada computador en cada paso precisa conocer el estado de sus ocho celdas vecinas, lo que supone un intercambio de mensajes con los ocho computadores correspondientes. Otros problemas requieren que cada procesador conozca, tras cada paso, el estado del sistema completo, con lo que todos los datos deben propagarse a todos los computadores. Es el caso de las simulaciones que involucran interacciones de largo alcance, como la gravitatoria o la culombiana. Todas estas aplicaciones ofrecen un grado de paralelismo intermedio entre

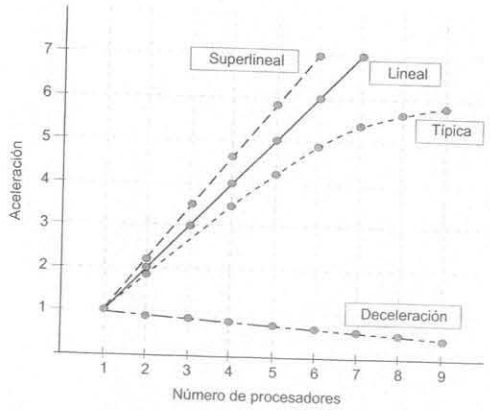


Figura 4.32: Curva de aceleración frente al número de procesadores para distintos casos de aplicaciones paralelas.

el régimen lineal y el de deceleración. La curva de aceleración resultante se asemeja a la etiquetada como "Típica" en la Figura 4.32. La aceleración es prácticamente lineal cuando el número de computadores en el cluster es pequeño, pero al ir añadiendo nodos el tiempo de distribución de los datos comienza a ser significativo y la ejecución ya no se acelera tanto. En el ejemplo de la Figura 4.32 es preciso utilizar nueve nodos para hacer que el programa se ejecute aproximadamente seis veces más rápido que en un sistema monoprocesador; esta es una cifra habitual en clusters reales.

Por ello, es interesante utilizar un cluster si la aplicación es suficientemente paralela y/o ha sido ya paralelizada (reescrita para hacer uso de procesamiento paralelo) o se está en disposición de hacerlo. Cada vez es posible encontrar más paquetes de software que han sido adaptados para su ejecución en un cluster. Red Hat y NASA han organizado una distribución de Linux orientada al aprovechamiento de clusters llamada Extreme Linux que contiene un conjunto de tales aplicaciones paralelizadas, como el generador de imágenes POV (*Persistence Of Vision*). Pero a menudo se está interesado en adaptar una aplicación propia de cálculo intensivo para sacar provecho de un cluster, y para ello no hay más remedio que recurrir a su paralelización. Existen dos opciones para hacer que la aplicación sea capaz de usar los computadores de un cluster: programar explícitamente el paso de mensajes o utilizar herramientas que lo hagan. El segundo caso abarca las herramientas de paralelización automática y los lenguajes de programación paralelos.

Desde luego, lo más sencillo es proporcionar el código fuente a un programa paralelizador y dejar que este produzca código paralelo. El problema es que estos paralelizadores suelen ser muy caros y, lo que es peor, funcionan realmente mal. Aunque en la actualidad muchos grupos de investigación están dedicando un gran esfuerzo al desarrollo y mejora de herramientas de paralelización automática, lo cierto

es que por el momento no son muy útiles.

En los años 80 se diseñaron un gran número de lenguajes explícitamente paralelos, como por ejemplo Orca3 o Parallaxis4, pero últimamente los desarrolladores se han dado cuenta de que los usuarios no van a usar su nuevo lenguaje, pese a lo bueno que sea, si no es compatible con Fortran o C. Por ello, hoy en día los esfuerzos se dirigen al desarrollo de extensiones paralelas para estos lenguajes. El ejemplo más notable es HPF (*High Performance Fortran*). HPF es fundamentalmente Fortran 90 estándar al que se le han añadido una serie de directivas, que se introducen como comentarios en el código fuente para informar al compilador, por ejemplo, sobre cómo distribuir los datos entre los procesadores o qué bucles se pueden ejecutar en paralelo sin preocuparse por las dependencias de las expresiones.

Sin embargo, habitualmente se obtiene un mayor rendimiento programando explícitamente el paso de mensajes entre procesadores. Es posible optimizar hasta el último microsegundo utilizando la red a bajo nivel, usando sockets o el dispositivo de red directamente, pero al precio de una programación compleja, susceptible de errores y, lo que es peor, no transportable. El mejor compromiso entre eficiencia, facilidad de uso y portabilidad lo dan las bibliotecas de paso de mensajes, en las que el programador construye explícitamente el mensaje entre dos procesadores, o las de función colectiva, modelo intrínsecamente paralelo en el que un grupo de procesadores se comunica a la vez. Entre las primeras destacan PVM (*Parallel Virtual Machine*) y MPI (*Message Passing Interface*) y el ejemplo más significativo de las segundas es AFAPI (*Aggregate Function API*). Todas estas bibliotecas, que se encuentran disponibles en Linux, se pueden utilizar en cualquier arquitectura paralela a la que se hayan transportado, no solamente en clusters de computadores.

4.5.4. Programación de clusters

Desde hace varios años se puede observar que ha habido un incremento sostenido en la aceptación y utilización del procesamiento paralelo. Este crecimiento se debe principalmente al diseño y utilización de los clusters, tal y como ya se ha detallado en las secciones anteriores.

En cualquier aplicación que implique procesamiento paralelo, los datos se tienen que intercambiar entre las tareas que cooperan para resolver el problema planteado. Para llevar a cabo esta tarea, como ya se ha indicado, existen dos paradigmas de programación: la *memoria compartida* y el *paso de mensajes*.

Conviene tener presente las siguientes consideraciones sobre ambos paradigmas:

- El modelo de memoria compartida es similar al de un tablón de anuncios con acceso restringido, donde el que tiene permiso pone una noticia con lo que le interesa, y el que tiene permiso lee el tablón. A pesar de que este mecanismo parece sencillo, y de hecho es el más intuitivo que se puede emplear para compartir información, presenta un problema: el control de acceso a la información. Dicho de otro modo, hay que tener la seguridad de que la información que un procesador lee es semánticamente válida, y no son datos inválidos, debido a que uno de los procesadores que está escribiendo los mismos datos con los que otro procesador está realizando una operación de lectura. Este problema tiene unos mecanismos de resolución elaborados (semáforos, monitores, regiones críticas, etc.). Por estas razones, hoy en día la memoria compartida solo se emplea si el

grado de acoplamiento[5] es tan alto que un mecanismo de paso de mensajes puede llegar a ser muy poco eficaz.

- El modelo de paso de mensajes es un mecanismo menos intuitivo pero más cómodo para transmitir información. Un proceso envía a otro proceso un mensaje que contiene la información que debe conocer. Por ello, no existe el problema que se señaló anteriormente del control de acceso a la información, ya que si a un proceso le llega un mensaje, este mensaje ya es correcto salvo errores de transmisión. Por lo tanto, la programación y verificación de un programa paralelo con comunicación por paso de mensajes es más sencilla que empleando memoria compartida. Sin embargo, si el grado de acoplamiento es alto, el trasiego de mensajes es desmesurado y este sistema queda totalmente descartado.

En la actualidad el modelo de paso de mensajes es el más aceptado, desde el punto de vista del número y variedad de procesadores que lo soportan, así como por los lenguajes y sistemas software que lo utilizan.

4.6. Rendimiento y costes en sistemas paralelos

En esta sección se describen en primer lugar los principales factores que influyen en la velocidad computacional de un sistema paralelo. En segundo lugar, dado que uno de los factores más importantes para calcular el rendimiento de un sistema paralelo está en la eficiencia del sistema de comunicación, se evalúan los costes de la comunicación mediante el uso de paso de mensajes y memoria compartida.

4.6.1. Factores que influyen en la velocidad computacional

Son muchos los factores que están directamente ligados con la velocidad computacional de un sistema paralelo. A continuación, se estudian los más destacables.

4.6.1.1. Granularidad de los procesos

Independientemente de la arquitectura paralela de que se disponga, para lograr una mejora en la velocidad haciendo uso del paralelismo es necesario dividir el cálculo en tareas o procesos que se puedan ejecutar de forma simultánea. El tamaño de un proceso se puede describir por su granularidad. Se dice que un proceso tiene granularidad gruesa cuando está compuesto por un gran número de instrucciones secuenciales, es decir, instrucciones que no necesitan de la comunicación con otros procesos para ser ejecutadas. Y se dice que el proceso tiene granularidad fina cuando dispone de pocas instrucciones secuenciales. Normalmente, es deseable incrementar la granularidad para reducir los costes de la creación

[5]El grado de acoplamiento es sinónimo de granuralidad. Es uno de los factores críticos a la hora de diseñar aplicaciones paralelas. Un grado de acoplamiento alto significa que los distingos procesos comparten mucha información. Desde el punto de vista práctico, mayor acoplamiento significa mayor trasiego de información entre procesadores.

y comunicación de procesos, pero probablemente esto reducirá el número de procesos concurrentes y la cantidad de paralelismo. Por tanto, es necesario encontrar un compromiso.

Para el modelo de paso de mensajes es particularmente deseable reducir la sobrecarga en la comunicación, sobre todo en los clusters donde la latencia de comunicación puede llegar a ser importante. A medida que se divide el problema en partes paralelas se llegará a un punto en el que el tiempo de comunicación domine sobre el tiempo total de ejecución. Se puede utilizar la razón:

$$\frac{\text{tiempo de computación}}{\text{tiempo de comunicación}} = \frac{t_{comp}}{t_{com}}$$

como medida de la granularidad. Es muy importante tratar de maximizar esa razón, siempre y cuando se mantenga el paralelismo suficiente. También hay que decir que la granularidad está relacionada con el número de procesadores que se utilicen. Por ejemplo, a la hora de repartir los datos, el tamaño de un bloque de datos utilizado por un procesador se podría aumentar, incrementando de esta forma la granularidad. Sin embargo, si el problema a resolver tiene un tamaño fijo de datos, para incrementar la granularidad habría que disminuir el número de procesadores. En general es conveniente diseñar programas paralelos en los que se pueda variar la granularidad, es decir, conseguir un diseño de programa escalable.

4.6.1.2. Factor de aceleración (*speedup*)

Una medida del rendimiento relativo entre un sistema multiprocesador y un sistema con un único procesador es el factor de aceleración denominado *speedup*, $S(M)$, que se define como:

$$S(M) = \frac{t_s}{t_p}$$

donde t_s es el tiempo de ejecución de un programa en un único procesador y t_p es el tiempo de ejecución en un sistema paralelo con M procesadores. El parámetro $S(M)$ da el incremento de velocidad que se obtiene cuando se utiliza un sistema con M procesadores en lugar de un único procesador.

Para comparar una solución paralela con una solución secuencial, se utilizará el algoritmo secuencial más rápido para ejecutarlo en un único procesador. En un análisis teórico el factor de aceleración se puede dar también en términos de *pasos o etapas computacionales*. Recuérdese que se entiende por paso o etapa computacional a las operaciones que en su conjunto permiten obtener un resultado parcial completo de la tarea a realizar. Normalmente se componen de tareas repetitivas independientes entre sí. Por ejemplo, una misma operación que debe realizarse sobre un gran número de datos diferentes se podría realizar secuencialmente con una iteración sobre los datos. Cada vez que se calcula un resultado parcial completo se realiza una etapa computacional. En un sistema paralelo, se pueden obtener varios resultados al realizar los cálculos paralelamente sobre diferentes datos, obteniéndose así varios resultados en una etapa. Así, el *speedup* será:

$$S(M) = \frac{\text{Número de pasos computacionales utilizando un procesador}}{\text{Número de pasos computacionales paralelos con } M \text{ procesadores}}$$

es decir, el cociente entre la suma de los pasos computacionales que se necesitan para ejecutar el algoritmo en un único procesador y la suma de los pasos computacionales que son necesarios para ejecutar ese mismo algoritmo en un sistema con M procesadores. Recuérdese que todos aquellos pasos computacionales que se realizan en paralelo contabilizan solo como uno. La máxima aceleración que se puede lograr con M procesadores es M (aceleración lineal). La aceleración máxima absoluta de M se debería alcanzar cuando: la computación se puede dividir en procesos de igual duración, cada proceso se localiza en un procesador y no hay sobrecarga (*overhead*), es decir:

$$S(M) = \frac{t_s}{\frac{t_s}{M}} = M$$

Una aceleración superlineal, donde $S(M) > M$, se puede encontrar excepcionalmente en alguna ocasión, pero debe recordarse que normalmente se debe a una utilización subóptima del algoritmo secuencial o a alguna característica de la arquitectura que favorece la programación paralela del problema.

Existen varios factores que aparecen como sobrecarga en los programas paralelos y que limitan la aceleración. Muy especialmente deben tenerse en cuenta los siguientes:

- Los periodos en los que no todos los procesadores están realizando un trabajo útil, es decir, cuando en el programa existen partes secuenciales que no se pueden dividir y las tiene que ejecutar un único procesador.

- Los cálculos adicionales que aparecen en el programa paralelo y que no aparecen en el secuencial.

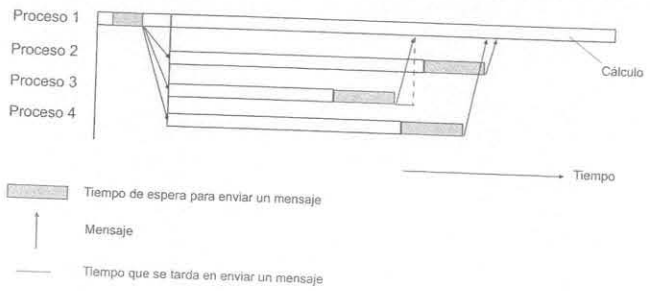


Figura 4.33: Diagrama espacio-temporal de un programa de paso de mensajes.

- El tiempo de comunicación para enviar mensajes.

En la Figura 4.33 se pueden observar las sobrecargas y retrasos que se producen en la ejecución de un programa paralelo por paso de mensajes. Parece también razonable, como se observa en la Figura 4.33, esperar que exista alguna parte del programa que no se pueda dividir en procesos paralelos y se tengan que ejecutar de forma secuencial. Por ejemplo, en el periodo inicial, antes del arranque de los procesos paralelos, únicamente un procesador está haciendo un trabajo útil, mientras que durante el resto de la computación los procesadores restantes estarán ejecutando procesos.

4.6.1.3. Ley de Amdahl

De los factores limitadores de la aceleración, la cantidad de parte secuencial existente en un programa paralelo fue considerada por Amdahl como el factor más determinante en la aceleración, es decir, la parte secuencial de un programa determina una cota inferior para el tiempo de ejecución, aún cuando se utilicen al máximo las técnicas de paralelismo.

Si se llama f a la fracción del programa que no se puede dividir en tareas paralelas, $0 \leq f \leq 1$, y se considera que no hay sobrecarga cuando el programa se divide en tareas paralelas, el tiempo de computación necesario para ejecutar el programa en M procesadores ($t_p(M)$) vendrá dado por la siguiente expresión:

$$t_p(M) = f \cdot t_s + \frac{(1-f) \cdot t_s}{M}$$

En la Figura 4.34 se muestra gráficamente la paralelización en M procesadores de un programa

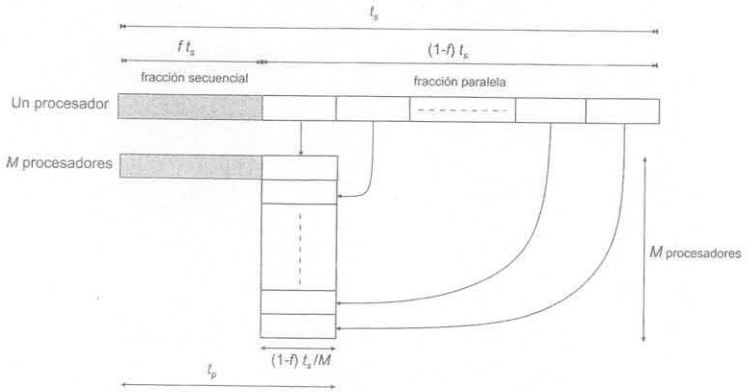


Figura 4.34: Programa secuencial paralelizado. Ley de Amdahl.

secuencial. Se observa que la parte secuencial está localizada al comienzo del programa, pero hay que hacer notar que podría estar distribuida a lo largo de toda la aplicación paralela.

Por lo tanto, la ley de Amdahl expresa que el factor de aceleración viene dado por:

$$S(M) = \frac{t_s}{f \cdot t_s + (1-f) \cdot \frac{t_s}{M}} = \frac{M}{1 + (M-1) \cdot f}$$

En la ecuación anterior se observa que el factor de aceleración nunca puede ser mayor que $\frac{1}{f}$, es decir, no importa lo grande que sea M (número de procesadores), la máxima aceleración que se puede conseguir para un determinado problema está acotada.

Ejemplo: Suponer que se quiere lograr una aceleración lineal con 100 procesadores. ¿Qué fracción del cálculo original puede ser secuencial?

La ley de Amdahl es:

$$S(M) = \frac{1}{f + \frac{(1-f)}{M}}$$

Sustituyendo el objetivo de aceleración lineal con 100 procesadores se obtiene:

$$100 = \frac{1}{f + \frac{(1-f)}{100}}$$

despejando el porcentaje mejorado, se obtiene $f = 0$. Por tanto, para conseguir una aceleración lineal con 100 procesadores, ningún cálculo original puede ser secuencial. Dicho de otra forma, para obtener una aceleración de 99 a partir de 100 procesadores significa que la fracción secuencial del programa original (f) debe de ser aproximadamente 0,0001.

La ley de Amdahl expresa que el tiempo de ejecución de la parte paralelizable de un programa es independiente del número de procesadores. Aunque esto es verdad cuando se ejecuta un problema de tamaño fijo en diferentes números de procesadores, no lo es tanto cuando el tamaño del problema se escala con el número de procesadores disponibles. En la práctica, el número de procesadores se suele ajustar al tamaño del problema para mantener el tiempo total de computación en un valor deseado. Por tanto, según Amdahl la razón para aumentar el número de procesadores debe ser para resolver problemas de tamaño mayor, y no para resolver más rápidamente un problema de tamaño fijo.

En la Figura 4.35 se puede observar la variación del factor de aceleración en función del número de procesadores y de la fracción secuencial del programa. Por ejemplo, siendo el 5 % de la computación total la parte secuencial, la máxima aceleración que se puede conseguir es de 20, independientemente del número de procesadores. Amdahl utilizó este argumento para promover en los años 60 el empleo de los sistemas monoprocesadores. Esta afirmación hoy en día no es totalmente cierta ya que en algunas situaciones un factor de aceleración de 20 puede resultar impresionante.

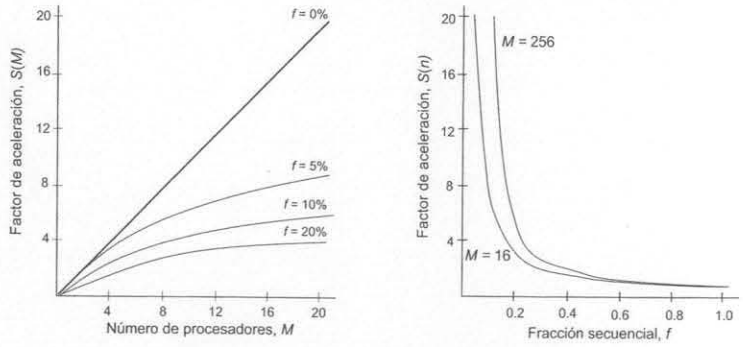


Figura 4.35: Gráficas del factor de aceleración, $S(M)$.

Del estudio de la ley de Amdahl, se podría definir el límite de Amdahl ($\frac{1}{f}$) como el mayor factor de aceleración posible cuando el número de procesadores disponibles tiende a infinito. Por ejemplo, si un programa contiene un 10 % de código no paralelizable ($f = 0{,}1$), el límite de Amdahl sería 10 ($\frac{1}{0{,}1}$), independientemente del número de procesadores del sistema.

Conviene recordar que en los resultados de Amdahl se supone que no hay sobrecarga, es decir, el tiempo de comunicación no se tiene en cuenta; en la práctica no se puede obviar.

4.6.1.4. Eficiencia

Si se normaliza la aceleración de un programa paralelo dividiéndola entre el número de procesadores se obtiene la eficiencia E de un sistema:

$$E = \frac{\text{Tiempo de ejecución utilizando un único procesador}}{\text{Tiempo de ejecución utilizando un multiprocesador} \times \text{número de procesadores}} = \frac{t_s}{t_p \times M}$$

Si se tiene en cuenta la ecuación anterior se puede expresar la eficiencia E en función del factor de aceleración $S(M)$ mediante la siguiente expresión:

$$E = \frac{S(M)}{M} \times 100\,\% = \frac{1}{1 + (M-1) \cdot f} \times 100\,\%$$

donde E se expresa en porcentaje. La eficiencia da la fracción de tiempo que se utilizan los procesadores durante la computación, es decir, da una medida de lo adecuadamente que han sido utilizados los procesadores. Así por ejemplo, si la eficiencia E es del 50 % se puede afirmar que, en promedio, los procesadores se utilizan la mitad del tiempo de lo que dura la computación. Por tanto, para un mismo

programa, un incremento en el número de procesadores del sistema supondrá un decremento en la eficiencia, ya que el trabajo realizado por cada uno se reparte y el tiempo de uso de cada procesador disminuye.

4.6.1.5. Coste

El coste C (o trabajo) de una computación, se define como:

$$C = \text{Tiempo de ejecución} \times \text{Número de procesadores utilizados}$$

El coste de una computación secuencial es simplemente su tiempo de ejecución t_s, y el de una computación paralela es $t_p \times M$. El tiempo de ejecución paralelo, t_p, viene dado por $\frac{t_s}{S(M)}$. Por lo tanto, el coste de una computación paralela se puede expresar como:

$$C = \frac{t_s \cdot M}{S(M)} = \frac{t_s}{E}$$

Un coste óptimo de un algoritmo paralelo es aquel que es proporcional al coste (tiempo de ejecución) que tiene en un sistema con un único procesador.

4.6.1.6. Escalabilidad

La *escalabilidad* se suele utilizar para indicar un diseño hardware que permite ampliar su tamaño para obtener una mejora en el rendimiento. A este tipo de escalabilidad se le denomina *escalabilidad hardware*. También se utiliza el término escalabilidad para indicar que un algoritmo paralelo puede soportar un incremento grande de datos con un incremento bajo y acotado de pasos computacionales. A este otro tipo de escalabilidad se le denomina *escalabilidad algorítmica*. La definición más simple de escalabilidad es que un sistema es escalable si el rendimiento del mismo se incrementa linealmente con relación al número de procesadores usados para una cierta aplicación. El rendimiento de un sistema depende de un gran número de factores que influyen en la escalabilidad de la arquitectura del sistema y en el programa de aplicación que se ejecute.

El análisis de la escalabilidad de un sistema debe realizarse para una cierta aplicación y bajo diferentes restricciones en el crecimiento del tamaño del problema (carga de trabajo) y el tamaño del sistema (número de procesadores).

Los estudios de escalabilidad determinan el grado de afinidad entre una arquitectura determinada y una aplicación. Para diferentes pares (arquitectura, algoritmo), el análisis puede obtener diferentes resultados ya que una máquina puede ser muy escalable para un algoritmo y muy poco para otro.

Los principales parámetros que afectan a la escalabilidad de un sistema para una determinada aplicación son:

- *Tamaño del sistema*: El número de procesadores utilizados en el sistema. Un tamaño grande implica más recursos y más potencia de procesamiento.

- *Frecuencia de reloj*: La frecuencia de reloj determina el ciclo de máquina básico. Es deseable un sistema cuyos componentes (procesadores, memorias, buses, etc.) estén controlados por un reloj cuya frecuencia pueda incrementarse cuando la tecnología mejore.
- *Tamaño del problema*: La cantidad de trabajo computacional necesaria para resolver un determinado problema.
- *Tiempo de CPU*: El tiempo de CPU real usado en la ejecución de un determinado programa en un sistema con M procesadores.
- *Capacidad de memoria*: La cantidad de memoria principal utilizada para la ejecución del programa. Hay que tener en cuenta que la demanda de memoria viene afectada por el tamaño del problema, el tamaño del programa, los algoritmos y las estructuras de datos usadas. Puesto que la demanda de memoria puede variar en tiempo de ejecución, este parámetro se refiere a la máxima cantidad de memoria solicitada durante la ejecución del programa.
- *Pérdidas (overhead) de comunicación*: La cantidad de tiempo gastada en la comunicación entre procesadores, sincronización, etc.
- *Coste del sistema*: El coste total económico de los recursos hardware y software necesarios para llevar a cabo la ejecución de un programa.

En función de los objetivos establecidos y las restricciones de recursos impuestas, se pueden fijar algunos de los parámetros anteriores y optimizar los restantes para conseguir el mayor rendimiento con el coste menor.

La noción de escalabilidad está ligada a las nociones de incremento de velocidad y eficiencia que se han discutido en esta sección. Una buena expresión de la escalabilidad debe ser capaz de incluir los efectos de la red de interconexión de la arquitectura, de los patrones de comunicación inherentes a los algoritmos empleados, de las restricciones físicas impuestas por la tecnología y de la eficiencia del sistema.

4.6.1.7. Balance de carga

El *balance de carga* consiste en distribuir de una forma equitativa la carga computacional entre todos los procesadores disponibles y con ello conseguir la máxima velocidad de ejecución.

Se considera que el problema a resolver se divide en un número fijo de procesos que pueden ejecutarse en paralelo. Cada proceso realiza una cantidad conocida de trabajo. Además, se supone que los procesos se distribuyen entre las máquinas disponibles sin tener en cuenta el tipo de procesador y su velocidad.

Sin embargo, puede ocurrir que algunos procesadores finalicen sus tareas antes que el resto y queden libres debido a que el trabajo no se haya repartido de una forma equitativa o porque algunos procesadores sean más rápidos que otros, o por ambas situaciones. La situación ideal es que todos los procesadores

trabajen de una forma continuada sobre las tareas disponibles para conseguir el mínimo tiempo de ejecución. A la consecución de este objetivo dividiendo las tareas equitativamente entre los procesadores es a lo que se le denomina *balance de carga*.

La Figura 4.36 muestra como el balance de carga produce el mínimo tiempo de ejecución. En la Figura 4.36.a, el procesador P_1 está calculando durante un periodo de tiempo mayor que el resto y el P_4 completa su trabajo en menos tiempo que los demás. El tiempo de ejecución final viene dado por el tiempo del procesador P_1. Lo ideal sería que parte del trabajo del procesador P_1 lo hiciera P_4 para igualar las cargas de trabajo. En la Figura 4.36.b, todos los procesadores están calculando durante todo el tiempo, t segundos, consiguiéndose un balance de carga perfecto, con lo que se disminuye el tiempo de ejecución. Otra forma de ver este problema es la siguiente: para resolver la aplicación en un único procesador se requieren k ciclos de reloj con p procesadores. Sin considerar sobrecargas adicionales (tiempo de comunicación) para la implementación paralela, el tiempo de ejecución se reduciría a $\frac{k}{p}$ ciclos de reloj.

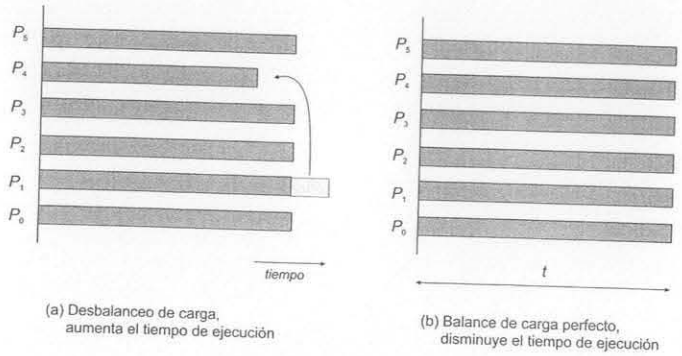


Figura 4.36: Balance de carga.

El balance de carga se puede tratar de forma estática, antes de la ejecución de cualquier proceso, y de forma dinámica, durante la ejecución de procesos. Al *balance de carga estático* se le suele denominar mapeado del problema o planificación del problema.

El balance de carga estático tiene serios inconvenientes que lo sitúan en desventaja sobre el balance de carga dinámico. Entre ellos caben destacar los siguientes:

- Es muy difícil estimar de forma precisa el tiempo de ejecución de todas las partes en las que se divide un programa sin ejecutarlas.
- Algunos sistemas pueden tener retardos en las comunicaciones que pueden variar bajo diferentes circunstancias, lo que dificulta incorporar la variable retardo de comunicación en el balance de

carga estático.

- A veces los problemas necesitan un número indeterminado de pasos computacionales para alcanzar la solución. Por ejemplo, los algoritmos de búsqueda normalmente atraviesan un grafo buscando la solución y, a priori, no se sabe cuántos caminos hay que probar, independientemente de que la programación sea secuencial o paralela.

Con el *balance de carga dinámico* todos los inconvenientes que presenta el balance de carga estático se tienen en cuenta. Esto es posible porque la división de la carga computacional depende de las tareas que se están ejecutando y no de la estimación del tiempo que pueden tardar en ejecutarse. Aunque el balance de carga dinámico lleva consigo una cierta sobrecarga durante la ejecución del programa, resulta una alternativa mucho más eficiente que el balance de carga estático.

En el balance de carga dinámico, las tareas se reparten entre los procesadores durante la ejecución del programa. Dependiendo de dónde y cómo se almacenen y repartan las tareas el balance de carga dinámico se divide en:

- Balance de carga *dinámico centralizado*. Se corresponde con la estructura típica de Maestro/Esclavo. El proceso maestro es el que tiene la colección completa de tareas a realizar. Las tareas son enviadas a los procesos esclavos. Cuando un proceso esclavo finaliza una tarea, solicita una nueva al maestro. Esta técnica también se denomina *programación por demanda* o *bolsa de trabajo*, y no solo es aplicable a problemas que tengan tareas de un mismo tamaño.

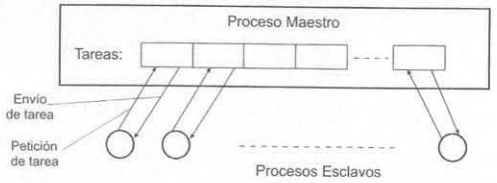


Figura 4.37: Balance de carga dinámico centralizado.

En problemas con tareas de distintos tamaños es mejor repartir primero aquellas que tengan una mayor carga computacional. Si la tarea más compleja se dejase para el final, las tareas más pequeñas serían realizadas por esclavos que después estarían esperando sin hacer nada hasta que alguno completara la tarea más compleja. También se puede utilizar esta técnica para problemas donde el número de tareas pueda variar durante la ejecución.

En algunas aplicaciones, especialmente en algoritmos de búsqueda, la ejecución de una tarea puede generar nuevas tareas, aunque al final el número de tareas se debe de reducir a cero para alcanzar la finalización del programa. En este contexto se puede utilizar una cola para mantener las tareas pendientes tal y como muestra la Figura 4.37. Si todas las tareas son del mismo tamaño y de la

misma importancia o prioridad, una cola FIFO (*First In First Out - Primero en Entrar Primero en Salir*) puede ser más que suficiente, en otro caso debe analizarse la estructura más adecuada.

- Balance de carga *dinámico distribuido* o descentralizado. Se utilizan varios maestros y cada uno controla a un grupo de esclavos. Una gran desventaja del balance de carga dinámico centralizado es que el proceso maestro únicamente puede repartir una tarea cada vez, y después de que haya enviado las tareas iniciales solo podrá responder a nuevas peticiones de una en una. Por tanto, se pueden producir colisiones si varios esclavos solicitan peticiones de tareas de manera simultánea.

  La estructura centralizada únicamente será recomendable si no hay muchos esclavos y las tareas son intensivas computacionalmente. Para tareas de grano fino (tareas con muy poca carga computacional) y muchos esclavos es apropiado distribuir las tareas en más de un sitio.

  La Figura 4.38 muestra el esquema de un balance de carga dinámico distribuido. El maestro divide su trabajo inicial en varias partes y envía una a cada máquina que actúa como mini-maestro (de M_0 a M_{n-1}). Cada mini-maestro controla un conjunto de máquinas esclavas.

  Así por ejemplo, para un problema de optimización, las máquinas que actúan como mini-maestros podrían encontrar un óptimo local y enviárselo al maestro para que seleccione la mejor solución.

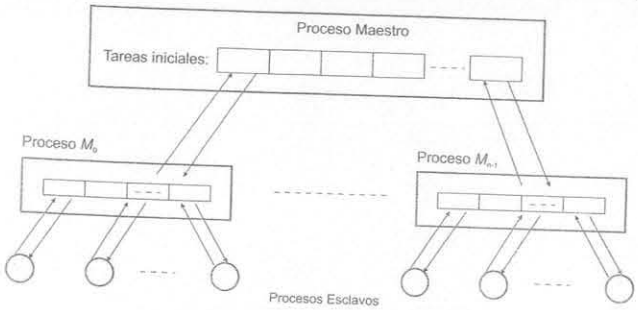


Figura 4.38: Balance de carga dinámico distribuido.

4.6.2. Costes de la comunicación mediante paso de mensajes

El tiempo empleado en transmitir un mensaje entre dos componentes de un sistema paralelo es la suma entre el tiempo necesario para preparar el mensaje y el tiempo que éste emplea en atravesar la red de comunicación hasta su destino. Este tiempo se denomina *latencia de comunicación* y sus principales parámetros son los siguientes:

- *Tiempo de inicialización* (t_s): Este tiempo incluye el tiempo en preparar el mensaje, el tiempo de

ejecución del algoritmo de enrutamiento y el tiempo de conexión entre el emisor y el enrutador. Este tiempo se contabiliza una vez por cada mensaje transmitido.

- *Tiempo de salto* (t_h): Es el tiempo que tarda la cabecera de un mensaje en viajar entre dos nodos directamente conectados en la red (también se suele denominar *latencia de nodo*). Este tiempo está ligado con el tiempo que tarda un enrutador en decidir el siguiente salto del mensaje.

- *Tiempo de transferencia por palabra* (t_w): Si el ancho de banda de un canal es r palabras por segundo, su tiempo de transferencia por palabra es $\frac{1}{r}$. Es el tiempo que tarda una palabra en ser transmitida por el canal.

El coste de la comunicación varía dependiendo del tiempo del algoritmo de enrutamiento utilizado. A continuación, se describen dos algoritmos de enrutamiento ampliamente utilizados en sistemas paralelos.

- *Almacenamiento y reenvío (store-and-forward)*

  En este algoritmo de enrutamiento cada nodo intermedio entre el emisor y el receptor reenvía el mensaje únicamente cuando lo ha recibido y almacenado completamente. Se supone un mensaje de tamaño m que se transmite a través de l enlaces intermedios. Donde en cada enlace el mensaje tarda t_h para la cabecera y $t_w m$ para el resto del mensaje. El tiempo total para el mensaje sería:

  $$t_s + (mt_w + t_h)l$$

  Teniendo en cuenta que el tiempo de salto en los sistemas paralelos actuales es mucho menor que el tiempo de transferencia del mensaje (mt_w), incluso para un número reducido de palabras, éste puede ser ignorado. Por tanto tendríamos que:

  $$t_{com} = t_s + mlt_w$$

  La Figura 4.39.a muestra un ejemplo de transferencia de un mensaje a través de una red que usa el enrutamiento mediante almacenamiento y reenvío. Observe cómo cada uno de los nodos ($P0$, $P1$, $P2$ y $P3$) espera a recibir todo el mensaje antes de reenviarlo al próximo nodo.

- *Corte y continuación (cut-through)*

  Dado que el anterior algoritmo hace un uso pobre de los recursos de comunicación del sistema, el algoritmo cut-through se basa en los sistemas de enrutamiento de paquetes (normalmente utilizados en sistemas grandes y muy distribuidos, como Internet) para mejorar los costes de comunicación en sistemas paralelos.

  El algoritmo cut-through divide cada mensaje en un número fijo de unidades llamados *dígitos de control de flujo (flow control digits o flits)*. Los flits son más pequeños que los paquetes porque no incluyen información de enrutamiento y la información para la corrección de errores es más

sencilla (aprovechando que las redes de comunicación en sistemas paralelos son poco propensas a la transmisión con errores).

Antes de enviar el primer flit el emisor establece un camino hasta el receptor mediante el envío de un paquete especial llamado *tracer*. Una vez se ha establecido la conexión se envían los flits uno tras otro, siguiendo todos la misma ruta. Los nodos intermedios no esperan a recibir todo el mensaje, sino que reenvían los flits según los van recibiendo. De esta manera se reducen, o casi se eliminan, los buffers de almacenamiento en cada nodo, El algoritmo cut-through requiere menos memoria y es más rápido que el algoritmo de almacenamiento y reenvío.

Se supone un mensaje de m palabras que viaja a través de una red con l enlaces. Si t_h es el tiempo de salto, la cabecera del mensaje tardará un tiempo lt_h en llegar al receptor. El contenido del mensaje tardará un tiempo mt_w en ser transmitido. Por tanto se tiene:

$$t_{com} = t_s + lt_h + mt_w$$

que supone una mejora respecto del algoritmo de almacenamiento y reenvío, ya que los términos correspondientes al número de saltos y el tamaño del mensaje son sumados en lugar de

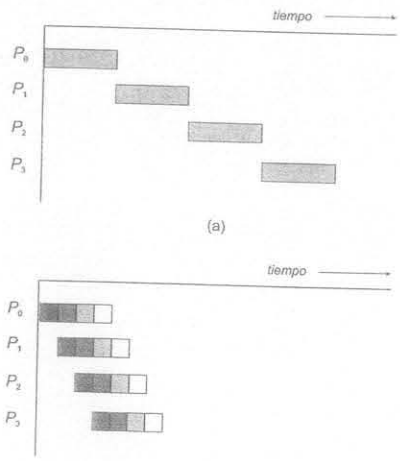


Figura 4.39: Transferencia de un mensaje: a) enrutamiento mediante almacenamiento y reenvío (*store-and-forward*), b) enrutamiento mediante *cut-through*.

multiplicarse. Ambos algoritmos se comportan de manera aproximada en los casos en los que la comunicación sea entre nodos cercanos ($l = 1$) o con tamaños de mensaje pequeños ($m = 1$).

La Figura 4.39.b muestra un ejemplo de transferencia de un mensaje, dividido en 4 flits, a través de una red que usa el enrutamiento mediante cut-through. En este caso, el tiempo de transferencia se reduce, en comparación con la versión que usa almacenamiento y reenvío, ya que cada uno de los nodos transmite las partes del mensaje tan rápido como puede sin esperar a recibir la siguiente parte o el mensaje completo.

El tamaño del flit es un parámetro básico en el diseño de un sistema de enrutamiento basado en el algoritmo cut-through. Si el flit es demasiado pequeño, la lógica de control de los enrutadores deberá trabajar demasiado deprisa para poder cubrir la tasa de envío de flits. Por el contrario, si el flit es demasiado grande, el tamaño de los buffers de los enrutadores se debe incrementar reduciendo así la latencia de transferencia de mensajes. En la actualidad el tamaño más usado suele estar entre 4 y 32 bytes.

4.6.3. Costes de la comunicación mediante memoria compartida

Encontrar una unidad de medida para el coste de la comunicación de un programa paralelo mediante paso de mensajes es mucho más sencillo que mediante el uso de memoria compartida. Las siguientes razones lo explican:

- La disposición de los datos en memoria está determinada por el sistema. El programador tiene un control reducido sobre la localización de los objetos en memoria. Esto hace muy difícil la distinción entre el acceso a datos locales y remotos, dificultando en gran medida el cálculo del tiempo de acceso a dichos datos.
- La limitación en el tamaño de las caché puede provocar un problema de hiperpaginación (*thrashing*). Si el tamaño de los datos que necesita un nodo para realizar su trabajo es mayor que el tamaño disponible de la caché local, parte de dichos datos deberá sobreescribir otra porción de los mismos en la caché para que puedan ser utilizados. La sobreescritura de los datos provoca múltiples accesos a caché y una degradación en el rendimiento de la misma conforme el tamaño del problema se incrementa. Este problema se agudiza en multiprocesadores ya que cada fallo de la caché puede provocar operaciones de coherencia de caché y comunicaciones entre procesadores.
- La sobrecarga (*overhead*) incluida por las operaciones de coherencia de caché (invalidar y actualizar) es difícil de cuantificar. Dependiendo de la configuración de los datos en las caché y el número de copias existentes, los diferentes accesos pueden provocar múltiples acciones de coherencia. Por ejemplo, un procesador que esté trabajando con un dato en su caché podrá realizar algunas operaciones directamente con la caché, pero sin embargo otras operaciones generarán un fallo. Estos fallos generan la transferencia del dato desde el procesador que mantiene la copia modificada hasta el procesador que necesita el dato. El número de copias de los datos en las diferentes cachés y la secuencia de accesos a los mismos están fuera del alcance del programador.

- La localidad espacial es un fenómeno difícil de modelar. La latencia de acceso a las diferentes palabras de una caché puede variar, incluso para los primeros accesos. Si se accede a una palabra "vecina" de un acceso previo, la lectura resulta muy rápida solo en el caso que la caché no se haya tenido que sobreescribir.

- La lectura-anticipada de palabras juega un papel importante en la reducción de los tiempos de acceso a los datos. Los compiladores pueden realizar este tipo de acciones para reducir los tiempos de acceso. Sin embargo, dado que esto solo depende del compilador y la disponibilidad de recursos, realizar un modelo preciso resulta realmente difícil.

- El fenómeno de *false-sharing* puede incluir una sobrecarga (*overhead*) importante en muchos programas. Se considera false-sharing cuando diferentes procesadores acceden a diferentes datos que están almacenados en el mismo bloque de caché. Esta situación puede provocar acciones de coherencia incluso cuando en realidad ambos procesadores no están compartiendo ningún dato.

- La competición por los recursos del sistema normalmente supone una de las mayores sobrecargas (*overheads*) en sistemas de memoria compartida. Desafortunadamente, dado que la competición por los recursos depende de la ejecución del programa también es un fenómeno muy difícil de modelar.

Cualquier modelo de coste para sistemas de emoria compartida debe tener en cuenta las razones mencionadas anteriormente. De todas maneras, los modelos de dichas características son normalmente muy complejos y poco útiles.

En lineas generales se puede ver que el acceso a datos remotos provoca la lectura de dichos datos para escribirlos en la caché local. Esto implica un tiempo para las acciones de coherencia, la red de interconexión y el acceso a memoria. Por similitud con los modelos de aso de mensajes, se puede denominar a este tiempo como t_s (que supone un acceso inicial a un bloque de datos compartidos de m palabras). Por otro lado, se puede asumir que el acceso a los datos locales es menos costoso que el acceso a datos remotos. El tiempo de acceso a una palabra remota se denomina como t_w. El coste de compartir un bloque de datos de m palabras será $t_s + mt_w$, que es análogo al caso de paso de mensajes (teniendo en cuenta que t_s será mucho más pequeño que t_w en un sistema de memoria compartida). Esta aproximación de coste asume un acceso de solo-escritura sin tener competición por los recursos del sistema. Si múltiples procesadores acceden a los mismos datos, el coste se multiplica por el número de procesadores, de manera análoga al paso de mensajes donde el procesador que tiene los datos envía mensajes a los otros procesadores. Si el acceso es en modo lectura-escritura, el coste se incrementa en los accesos de los procesadores posteriores al que escribe. Este comportamiento también es análogo al paso de mensajes, donde el procesador que modifica los datos tiene que enviar un mensaje para actualizar los datos locales del resto de procesadores.

El modelo descrito proporciona una aproximación para el coste de acceso a datos compartidos. Sin embargo, no se tiene en cuenta la competición por los recursos de la máquina, la posibilidad de que los tamaños de las caché de los procesadores no sea el mismo, la localidad espacial de los datos en caché,

el false-sharing, ni las posibles colisiones en las comunicaciones. Existen otros modelos más complejos que incluyen algunos de estos parámetros, pero como ya se ha comentado su resultado es poco útil como apoyo para el desarrollo de programas paralelos.

4.7. Resumen

El procesamiento paralelo trata de mejorar las capacidades de resolución de problemas de los sistemas actuales mediante la distribución del trabajo entre múltiples procesadores. Dichos procesadores trabajan en paralelo, de manera simultánea, para ejecutar las tareas de cómputo de un problema. Dependiendo del punto de vista desde el que se observe el sistema paralelo, pueden ser organizados en diferentes plataformas.

Así, si se observa desde el punto de vista de un programador, los sistemas paralelos estarán organizados desde su estructura de control o desde el modelo de comunicación utilizado para transmitir información entre procesadores. La estructura de control se refiere al paradigma de programación utilizado para distribuir los trabajos entre los componentes del sistema. Existen multiples paradigmas entre los que cabe destacar el Maestro/Esclavo y el SPMD (*Single Program Multiple Data*). El modelo de comunicación hace referencia al método usado para compartir datos entre los componentes: memoria compartida o paso de mensajes.

Si se toma el punto de vista físico, o del hardware, los sistemas paralelos se organizan en: sistemas de memoria compartida o sistemas de memoria distribuida. En los sistemas de memoria compartida existen tres parámetros fundamentales: la organización de la memoria principal, teniendo en cuenta la distribución de memoria y cachés entre procesadores (ya sean locales o remotas a cada uno), la red de interconexión y sus diferentes topologías (ya sean estáticas o dinámicas), y por último el sistema de coherencia de caché (que asegura una coherencia de datos entre las cachés de los procesadores del sistema). Los sistemas de memoria distribuida están representados principalmente por los clusters, que son sistemas compuestos por varios computadores interconectados que interactúan para resolver problemas de manera más eficiente.

Dado que el principal objetivo del procesamiento paralelo es mejorar el rendimiento de cómputo, se han descrito los principales factores con los que medir dicho rendimiento. Entre ellos se puede mencionar la granularidad de los procesos, el *speedup*, la ley de Amdahl, la eficiencia, el coste, la escalabilidad y el balance de carga. Muchos de estos factores están relacionados entre sí. También cabe tener en cuenta los costes relacionados con la comunicación entre procesadores, ya sea para el caso de sistemas que utilizan paso de mensajes como para los sistemas de memoria compartida. Para el caso del paso de mensajes, el modelo de coste es relativamente factible de obtener, pero para el caso del sistema de memoria compartida obtener un modelo útil que sirva para mejorar la programación del sistema es muy complejo y difícil.

4.8. Referencias

Buyya R. *High Performance Cluster Computing*, vols. 1 y 2, Prentice Hall, NJ, USA, 1999.

Codenotti B., Leoncini M. *Introduction to Parallel Processing*. Addison-Wesley, Reading, MA, USA, 1993.

Culler D.E., Shing J.P., Gupta A. *Parallel Computer Architecture. A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, CA, USA, 1999.

Doallo R., Carneiro V.M., Fraguela B.B., Touriño J. *Multiprocesadores: Estructura y Programación*. Tórculo Ediciones, Universidad de La Coruña, 1995.

Enslow P.H. Multiprocessor Organization - A Survey. *ACM Computing Surveys*, 9(1), pp. 103–129, 1977.

Grama A., Gupta A., Karypis G., Kumar V. *Introduction to Parallel Computing*, 2ª ed., Pearson Education Limited, England, 2003.

Leighton F.T. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, San Maleo, CA, USA, 1992.

Pritchard D. Mathematical Models of Distributed Computation, *Proc. of OUG-7, Parallel Programming on Transputer Based Machine*, IOS Press, pp. 25–36, 1988.

Quinn M.J. *Parallel Computing: Theory and Practice*. McGraw-Hill, New York, NY, USA, 1994.

Roosta S.H. *Parallel Processing and Parallel Algorithms: Theroy and Computation*. Springer-Verlag, New York, NY, USA, 2000.

Tabak D. *Multiprocessors*. Prentice-Hall, Englewood Cliffs, NI, USA, 1990.

Tabak D. *Advanced Multiprocessors*. McGraw-Hill, New York, NY, USA, 1991.

Wilson G. *Parallel Programming for Scientists and Engineers*. MIT Press, Cambridge, MA, USA, 1995.

4.9. Preguntas de autoevaluación

P4.1 ¿Qué se entiende por procesamiento paralelo?

El procesamiento paralelo es el método de organización de las operaciones en un sistema de computación donde más de una operación es realizada de manera simultánea.

P4.2 ¿Cuáles son las características principales de un sistema multiprocesador?

- Debe estar compuesto por dos o más procesadores.
- Los procesadores deben compartir el acceso a una memoria común.
- Los procesadores deben compartir acceso a canales de E/S, unidades de control y dispositivos.
- El sistema es controlado por un único sistema operativo.

P4.3 ¿Qué tipos de plataformas de computación paralela se contemplan teniendo en cuenta su organización física?

Sistemas de memoria compartida y sistemas de memoria distribuida.

P4.4 ¿Que se entiende por paralelismo estructural?

El paralelismo que se puede encontrar en la estructura de los datos. Esta clase de paralelismo permitirá la ejecución de procesos paralelos con idéntico modo de operación pero sobre distintas partes de los datos.

P4.5 ¿En qué consiste la descomposición recursiva?

El problema se divide en subproblemas que se resuelven de forma independiente para, posteriormente, combinar sus resultados parciales y obtener el resultado final.

P4.6 ¿A qué se refiere la segmentación de datos como paradigma de programación paralela?

Está basado en el paralelismo funcional donde las diferentes partes de un programa pueden realizar distintas tareas de una manera concurrente y cooperativa.

P4.7 ¿Qué tareas se atribuyen al maestro en el paradigma Maestro/Esclavo?

El maestro es el responsable de la descomposición del problema en pequeñas tareas, de distribuir estas tareas entre el conjunto de procesadores esclavos y de recoger los resultados parciales obtenidos de cada esclavo para ordenarlos y obtener el resultado final del problema.

P4.8 ¿Cuándo conviene utilizar un balance de carga dinámico en un sistema Maestro/Esclavo?

Cuando el número de tareas es mayor que el número de procesadores disponibles o cuando el número de tareas es desconocido al comienzo de la aplicación.

P4.9 ¿En qué consiste el paradigma SPMD?

Cada procesador ejecuta básicamente el mismo código pero sobre distintas partes de los datos.

P4.10 ¿Qué consecuencias tiene el fallo de un procesador en un sistema SPDM?

Normalmente, el fallo de un único procesador es suficiente para bloquear la aplicación, debido a que entonces ningún procesador podrá avanzar más allá del punto de sincronización global.

P4.11 ¿Cuáles son los principales modelos de comunicación en sistemas paralelos?

El espacio de direcciones único y compartido, y el paso de mensajes.

P4.12 ¿Qué parámetros caracterizan la velocidad de transferencia entre elementos de un sistema paralelo?

La latencia de la red de interconexión y el ancho de banda.

P4.13 ¿Por qué está limitado el número de procesadores en sistemas de memoria compartida?

Para garantizar la eficacia de esta arquitectura es fundamental que el ancho de banda sea elevado, ya que en cada ciclo de instrucción cada procesador puede necesitar acceder a la memoria a través de la red de interconexión. Este hecho solo se satisface si no se encuentran varios procesadores tratando de acceder al medio utilizado para la transmisión de datos simultáneamente, y por tanto se limita el número de los mismos.

P4.14 ¿Qué significado tienen las siglas UMA?

Uniform Memory Access, es una arquitectura donde el tiempo de acceso a memoria es el mismo para cualquier palabra accedida por un procesador.

P4.15 ¿Qué se entiende por escalabilidad?

Es la capacidad del sistema para mejorar la potencia de cálculo cuando el número de componentes del mismo aumenta.

P4.16 ¿Qué propósito busca la incorporación de una memoria caché en un procesador?

Incrementar el ancho de banda entre el procesador y su memoria local.

P4.17 ¿Cuál es la principal diferencia entre las arquitecturas UMA y NUMA?

En la UMA el tiempo de acceso es uniforme, mientras que en la NUMA el tiempo de acceso a memoria es no uniforme dependiendo de si el acceso se realiza a la memoria local del procesador o a alguna memoria remota.

P4.18 ¿Cuáles son las operaciones básicas necesarias para comunicarse mediante paso de mensajes?

Envío, recepción, identificación y número de participantes.

P4.19 ¿Puede emularse un sistema de paso de mensajes en un sistema de memoria compartida?

Sí. Simplemente dividiendo el espacio de direcciones en fracciones iguales, cada una asignada aun procesador, las acciones de envío y recepción de mensajes estarían representadas por la escritura y lectura de información en dichas fracciones de memoria.

P4.20 ¿Qué es una red de interconexión estática?

Es una red cuya topología queda definida de manera definitiva y estable durante la construcción de la máquina paralela.

P4.21 ¿Qué diferencia una red lineal de un bus?

El hecho de que en un momento dado puede realizarse más de una transferencia simultáneamente siempre que sea a través de enlaces diferentes (por ejemplo, uno a la izquierda y otro a la derecha).

P4.22 ¿Cuál es el esquema de interconexión ideal para una red estática?

La red completamente conectada, en la que cada procesador se comunica directamente con cualquier otro, de manera que los mensajes se envían en una única etapa o paso.

P4.23 ¿Cuál es la principal desventaja de las topologías de red en árbol?

Que las comunicaciones pueden verse comprometidas en un nodo cuando el número de procesadores es grande y se realizan comunicaciones entre procesadores situados en los niveles superiores.

P4.24 ¿Qué es la distancia de Hamming?

Es el número total de posiciones de bits para los que las etiquetas de dos procesadores son diferentes.

P4.25 ¿Cómo se define el ancho de banda de bisección?

Es el menor volumen de comunicaciones permitidas entre dos mitades cualesquiera de la red con igual número de procesadores. Así, es el producto del ancho de bisección y el ancho de banda del canal.

P4.26 ¿A qué factor deben las redes multietapa su gran flexibilidad?

La flexibilidad de las redes multietapa viene de la posibilidad de reconfigurar dinámicamente los modos de conmutación de sus diferentes etapas.

P4.27 ¿Qué significa que una red sea bloqueante?

Significa que ciertas permutaciones, o conexiones a través de la red, pueden a su vez bloquear otras conexiones.

P4.28 ¿Qué tipo de permutación se usa en la red omega?

La permutación por barajamiento perfecto.

P4.29 ¿Qué se entiende por sistema de memoria coherente?

Un sistema de memoria es coherente si el valor devuelto por una operación de lectura sobre una dirección de memoria es siempre el mismo valor que el almacenado por la última operación de escritura realizada sobre esa misma dirección, independientemente de qué procesador realice las operaciones.

P4.31 ¿Por qué la migración de procesos puede provocar incoherencia de memoria?

Si por alguna razón el proceso es intercambiado a otro procesador antes de que las modificaciones realizadas se actualicen en la memoria principal, los datos que cargue el proceso en la caché del nuevo procesador serán incoherentes con las modificaciones realizadas en el anterior procesador.

P4.32 ¿En qué se basa un sistema snoopy de coherencia de caché?

Se basa en que cada procesador monitoriza el tráfico de la red en busca de transacciones para poder actualizar el estado de sus bloques de caché.

P4.33 ¿Qué condición debe tener un cluster para que sea denominado Beowulf?

Que cada uno de sus componentes esté exclusivamente dedicado al cluster.

P4.34 ¿Qué ventajas ofrece un switch frente a un hub para conectar los componentes de un cluster?

Con un switch la transmisión de datos entre dos procesadores solo genera tráfico en el segmento correspondiente. El ancho de banda no es compartido entre todos los procesadores conectados al switch por lo que cada procesadores dispone del 100 % del ancho de banda. El switch permite establecer cualquier número de interconexiones simultáneas entre sus puertos siempre y cuando no coincida el receptor.

P4.35 ¿Qué se entiende por deceleración en la ejecución de un programa sobre un cluster?

Cuando el tiempo de ejecución del programa sobre el cluster se incrementa a medida que se aumenta el número de procesadores del mismo. Esto es debido a la mala paralelización del programa, posiblemente por la gran cantidad de datos compartidos entre procesadores.

P4.36 ¿Cuales son las dos principales bibliotecas de paso de mensajes?

PVM (*Parallel Virtual Machine*) y MPI (*Message Passing Interface*).

P4.37 ¿A qué se denomina granuralidad gruesa?

Cuando un proceso está compuesto por un gran número de instrucciones secuenciales, es decir, instrucciones que no necesitan de la comunicación con otros procesos para ser ejecutadas.

P4.38 ¿Qué es el límite de Amdahl?

El límite de Amdahl es $\frac{1}{f}$, siendo f la fracción de código no paralelizable del programa, y se define como el mayor factor de aceleración posible cuando el número de procesadores disponibles tiende a infinito.

P4.39 ¿Qué define la eficiencia de un sistema paralelo?

La fracción de tiempo que se utilizan los procesadores durante la computación, es decir, da una medida de lo adecuadamente que han sido utilizados los procesadores.

P4.40 ¿En qué consiste el balance de carga?

En distribuir de una forma equitativa la carga computacional entre todos los procesadores disponibles y con ello conseguir la máxima velocidad de ejecución.

P4.41 ¿Qué sistema de balance de carga ofrece mejores resultados?

Aunque el balance de carga dinámico lleva consigo una cierta sobrecarga durante la ejecución del programa, resulta una alternativa mucho más eficiente que el balance de carga estático.

P4.42 ¿Cuáles son los principales parámetros de la latencia de comunicación, es decir, del tiempo empleado en transmitir un mensaje entre dos componentes de un sistema paralelo?

El tiempo de inicialización, el tiempo de salto y el tiempo de transferencia por palabra.

P4.43 ¿Cómo afecta al coste de comunicación mediante memoria compartida el fenómeno de hiperpaginación de caché en un sistema paralelo?

Este problema se agudiza en multiprocesadores ya que cada fallo de la caché puede provocar operaciones de coherencia de caché y comunicaciones entre procesadores.

4.10. Actividades

A4.1 Normalmente cuando se evalua el rendimiento evaluamos la diferencia de rendimiento debido a una mejora introducida. De manera más formal:

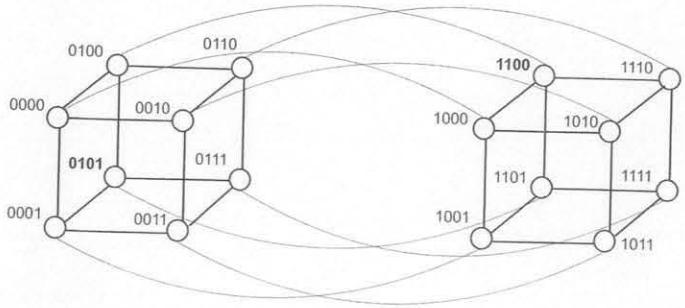
$$\text{Speedup debido a la mejora } E = \frac{Tiempo_{sinE}}{Tiempo_{conE}} = \frac{Rendimiento_{conE}}{Rendimiento_{sinE}}$$

En particular, nos referimos al speedup como una funcion del paralelismo de la máquina. Suponga que tiene un programa que realiza una cantidad fija de trabajo, del que una fracción s debe realizarse de manera secuencial. El resto del trabajo puede ser paralelizado sobre P procesadores. Asumiendo que T_1 es el tiempo que tarda en ejecutar el programa en un único procesador, encontrar una fórmula para describir T_p, como el tiempo que tarda en ejecutar en P procesadores. Usar esto para encontrar una fórmula para el límite superior del speedup potencial sobre P procesadores (esto es una variante de la Ley de Amdahl). Explicar este límite superior.

A4.2 Dibuje una red baseline de 8 entradas y 8 salidas. Explique detalladamente el proceso de construcción de la red.

A4.3 Defina qué entiende por una red omega y dibuje una para 16 procesadores. Explique razonadamente la conmutación de cada conmutador de la red que ha dibujado para enviar un mensaje del procesador 1101 al banco de memoria 1000.

A4.4 A partir de la red estática de la siguiente figura:



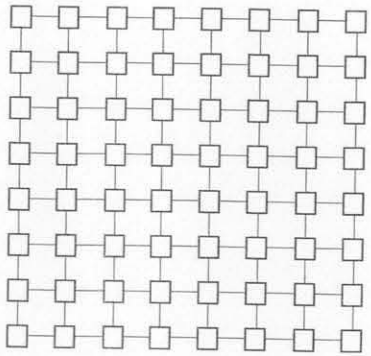
1. ¿Qué tipo de red es?

2. Se desea transmitir un mensaje desde el procesador a = 0101 al procesador b = 1100. Si se comienza a buscar el camino por el bit menos significativo, explique razonadamente cuál es el camino que debe seguir el mensaje.

3. Defina que entiende por conectividad de arco y por ancho de bisección. Calcule la conectividad de arco y el ancho de bisección de la red de la figura.

A4.5 Se dispone de un procesador que tarda 6 segundos en ejecutar un determinado programa. Sea $f = 0{,}3$, donde f representa la fracción del programa que no puede dividirse en tareas paralelas. Si se considera que no hay sobrecarga cuando el programa se divide en tareas paralelas, ¿cuál es el tiempo de computación necesario para ejecutar el programa en 12 procesadores de las mismas prestaciones que el original?

A4.6 Para transmitir un mensaje de n bytes a través de H enlaces de una red sin saturación, que utiliza el protocolo de *almacenamiento y reenvío*, el enrutamiento tarda $H\frac{n}{W} + (H-1)R$, donde W es el ancho de banda del enlace y R es el tiempo de salto. En una red que utilice el algoritmo *cut-through* el tiempo sería $\frac{n}{W} + (H-1)R$. Si consideramos una red de tipo mesh cuadradada de 8×8 (ver figura adjunta), cada procesador con un retraso de 250 ns y con enlaces de 40 MB/s ¿cual es el tiempo de transferencia mínimo, máximo y medio para un mensaje de 64 bytes? ¿y para uno de 256 bytes? Calcule los mismos tiempos para el caso del enrutamiento mediante el algoritmo cut-through.



: Procesador —— : Enlace

A4.7 Considere un simple esquema de diferencias finitas en dos dimensiones en el que en cada paso cada punto de la matriz es actualizado con la media ponderada de sus cuatro vecinos:

$$A[i,j] = A[i,j] - w(A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])$$

Todos los valores son números en coma flotante de 64 bits. Asumiendo que cada procesador calcula un elemento y la matriz tiene tamaño 1024×1024, ¿qué cantidad de datos debe transmitirse entre procesadores en cada paso? Explicar cómo se puede dividir este cálculo entre 64 procesadores para poder minimizar la cantidad de datos transmitidos, y calcular cuantos datos se deberían transmitir en cada paso.

A4.8 Considere un multiprocesador de memoria compartida distribuida. Considere un modelo simple de coste donde los accesos a la caché local tardan 10 ns, los accesos a memoria local tardan 100 ns y los accesos a memoria remota tardan 400 ns. Si ejecutamos en este sistema un programa paralelo correctamente balanceado con un 80 % de accesos a caché, 10 % de accesos a memoria local y 10 % de accesos remotos, ¿cuál será el tiempo efectivo de acceso a memoria? Si el cálculo tiene un límite de memoria, ¿cuál será la máxima tasa de cálculo?

Considere el mismo programa ejecutándose en un sistema uniprocesador. Ahora el procesador tiene un ratio de acierto de caché del 70 % y el 30 % de accesos a memoria local. ¿Cuál será la tasa efectiva de cálculo para un procesador? ¿Cuál será la tasa de cálculo?

A4.9 Considere el enrutamiento de mensajes en un sistema paralelo que utiliza el algoritmo de almacenamiento y reenvío. En este sistema, el coste de envío de un mensaje de tamaño m desde P_{origen} hasta $P_{destino}$ a través de un camino de longitud d es $t_s + t_w \times d \times m$. Un método alternativo para el envío de

un mensaje de tamaño m sería el siguiente. El usuario divide el mensaje en k partes, cada uno de tamaño $\frac{m}{k}$, y envía las distintas partes una por una desde P_{origen} hasta $P_{destino}$. Para este nuevo método, encontrar una expresión para el tiempo de transferencia del mensaje de tamaño m hasta un destino situado a una distancia de d saltos, teniendo en cuenta los siguientes casos:

- Asumir que un mensaje puede enviarse desde P_{origen} tan pronto como el mensaje anterior ha alcanzado el siguiente nodo en el camino.
- Asumir que un mensaje puede enviarse desde P_{origen} únicamente cuando el mensaje anterior ha alcanzado $P_{destino}$.

Para cada caso, analizar el comportamiento de la expresión cuando k varía entre 1 y m. A su vez, calcular el valor óptimo de k si t_s es muy grande, y si $t_s = 0$.

A4.10 Calcular el diámetro, número de enlaces y ancho de bisección de un d-cubo k-ario con p nodos. Definir l_{av} como la distancia media entre dos nodos de la red. Calcular l_{av} para el d-cubo k-ario.

A4.11 Las etiquetas de los nodos de un hipercubo d-dimensional utilizan d bits. Fijando un bit k cualquiera de la etiqueta, demostrar que los nodos cuyas etiquetas difieren en los $d - k$ bits restantes forman un subcubo $(d - k)$-dimensional compuesto por $2^{(d-k)}$ nodos.

A4.12 Dibuje una red estática con los siguientes valores en sus parámetros: diámetro = 2, conectividad de arco = 4 y ancho de bisección = 8. ¿Cuál sería el coste de la red que ha dibujado?

A4.13 Defina los siguientes conceptos de redes estáticas: distancia de Hamming y ancho de banda del canal.

A4.14 Dibuje una red butterfly de 8×8 y describa el enrutamiento desde el procesador 011 hasta el 101 y desde el procesador 000 hasta el 110.

Apéndice A

MEMORIA VIRTUAL

El sistema de memoria de los procesadores actuales no está constituido únicamente por memoria física, esto es, los registros, los niveles de memoria caché y la memoria RAM del procesador. Cuando un programa se ejecuta en un procesador, lo que realmente visualiza es la existencia de un espacio de direccionamiento muchísimo mayor que el proporcionado por la memoria RAM. Es un espacio de direccionamiento lógico, físicamente inexistente, que se denomina *memoria virtual*.

Hoy en el día, los procesadores se diseñan para trabajar con memoria virtual. La memoria virtual es un mecanismo por el que un procesador puede trabajar con un espacio de direccionamiento muy superior al ofertado por la memoria principal instalada físicamente junto al procesador. Gracias a ello, los programas se pueden ejecutar casi sin limitaciones en lo referente al espacio de direccionamiento con que cuentan ya que el procesador percibe que todo el espacio de direccionamiento virtual de que dispone se encuentra residente en la parte de la memoria física asignada a ese programa y que el acceso es transparente. Esta ilusión se genera mediante *mecanismos de traducción de las direcciones de memoria virtual en direcciones reales y mecanismos de paginación bajo demanda* que aseguran que la porción del espacio de direcciones virtuales que maneja el programa en un instante dado se encuentra alojado en la memoria principal y si no lo está, se le proporciona.

De esta forma, cuando un programa se está ejecutando en el procesador, la memoria principal constituye solo una parte del espacio privado de direcciones virtuales de que dispone, estando el resto almacenado en los discos duros. En muchos aspectos, el funcionamiento de la memoria virtual es similar a la memoria caché ya que cuando se intenta acceder a una dirección virtual que no está contenida en la memoria principal se producen fallos y es necesario reemplazar una parte del espacio de direcciones virtuales por otro almacenado en los discos duros.

Por ejemplo, la aparición de arquitecturas de 64 bits ha incrementado el espacio de direccionamiento virtual a 2^{64} direcciones, lo que equivale a 17.179.869.184 Gbytes o 16 Ebytes de memoria RAM, algo muy difícil de alcanzar utilizando la tecnología disponible actualmente para construir la memoria

principal. Un ordenador personal dotado de un procesador de 32 bits y 4 Mbytes de memoria principal cuenta con un espacio de direcciones virtuales que es 1000 mayor que el espacio de direcciones físicas. Por ejemplo, el PowerPC 970 que soporta direcciones virtuales de 65 bits y maneja direcciones reales de 42 bits.

Cuando se emplea memoria virtual, el procesador maneja direcciones virtuales que tiene que traducir para convertirlas en direcciones físicas y poder operar con la memoria principal. La traducción puede producir dos tipos de resultados: una dirección real de memoria (es decir, una dirección física) si se encuentra esa parte del espacio de direccionamiento virtual ubicado en la memoria principal, o una indicación de que la dirección de memoria virtual no se encuentra en la memoria principal. En este último caso se dice que se ha producido un *fallo de página* ya que la memoria virtual se organiza en *bloques* de tamaño fijo denominados *páginas* (observe la similitud con la memoria caché que se organiza en bloques). Si el sistema de memoria virtual utiliza bloques de longitud variable, los bloques pasan a denominarse *segmentos* y se habla de *memoria segmentada*. Hoy en día, la utilización de memorias segmentadas está en desuso debido a los problemas que plantea el reemplazo de bloques y cuando se habla de memoria virtual se entiende que se trata de una organización basada en páginas. El mecanismo de traducción de direcciones y protección de memoria se gestiona por una unidad hardware denominada Unidad de Manejo de Memoria o MMU (*Memory Management Unit*). Para acelerar todo este proceso de gestión de memoria, la mayoría de los procesadores ya incluyen una MMU en el mismo chip,

Como ya se ha indicado, si la dirección de memoria virtual se encuentra ubicada en la memoria principal, la MMU entrega una dirección física y el acceso a memoria se realiza de forma completamente normal. Por el contrario, si la dirección de memoria virtual no se encuentra en la memoria principal entonces se genera un fallo de página. Los fallos de página disparan una *excepción de fallo de página* que provoca que el sistema operativo tome el control para resolver la situación. Debido a que el fallo de página provoca el acceso a los discos duros para transferir una página, esto da lugar a una penalización de cientos de miles de ciclos de reloj. Por este motivo, el sistema operativo suspende la ejecución del programa causante del fallo y entrega el procesador a otro proceso para que se ejecute mientras se realiza la lectura de la página.

La traducción de direcciones se efectúa mediante una estructura de almacenamiento que contiene las equivalencias entre las direcciones virtuales y reales. Esta estructura se denomina *tabla de páginas* (PTE – *Page Table Entry*) y se almacena en la memoria principal (Figura A.1). Ya que la memoria virtual y la memoria principal se organizan en páginas (de igual forma que la caché y la memoria principal se organizan en bloques), la traducción consiste, de forma simplificada, en obtener el número de página física que corresponde a la página virtual en que se encuentra la dirección virtual. Por ello, una dirección virtual consta de dos partes o campos: el *Número de página virtual* y el *Desplazamiento* dentro de la página, siendo la longitud de este campo lo que determina el tamaño de la página. Un campo *Desplazamiento* de 10 bits de longitud indica que el tamaño de la página es de 1024 bytes o 1 Kbyte.

En cada entrada de la PTE se almacena el número de la página de memoria física asociado a la página virtual y un conjunto de bits con información sobre el estado de la página en memoria principal. Algunos de estos bits son el de *presencia* (P) que indica si la página está en memoria principal, el bit

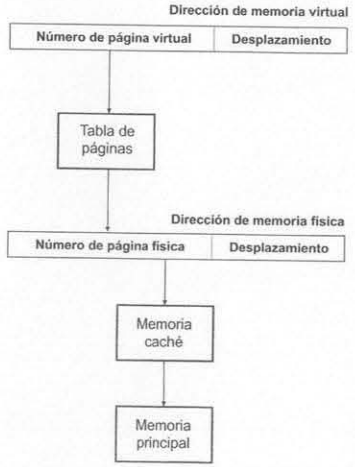


Figura A.1: Traducción de una dirección de memoria virtual en una dirección física.

de *modificada* (M) que especifica si la página ha sido escrita desde que fue traída y debe volcarse en el disco cuando sea reemplazada, el bit de *referencia* (R) que se utiliza para saber qué página es la menos utilizada y, por tanto, la mejor candidata para cuando sea necesario sustituir una página de la memoria principal ante un fallo de página, el bit de *permiso de lectura* (L), el bit de *permiso de escritura* (E) o el bit de *inhibición* (I) que permite evitar que las direcciones de esa página no se almacenen en caché por razones de rendimiento. Cuando se accede a la PTE para realizar la traducción de una dirección virtual y en la entrada la página está marcada como no presente (bit P) en la memoria principal se produce el fallo de página y la consabida excepción de fallo de página. Solventar el problema implica buscar una página a reemplazar en la memoria principal (bit R) y si la página selecciona fue escrita (bit M) actualizarla en el disco duro; tras esto ya se puede reemplazar la página seleccionada en memoria principal con la página de memoria virtual ubicada en el disco duro. El proceso finaliza con la actualización de la PTE para reflejar el nuevo estado de la memoria principal.

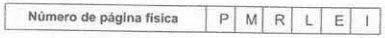


Figura A.2: Estructura de una entrada en la tabla de páginas.

El problema que surge al almacenar la tabla de páginas en la memoria principal es que la necesidad de realizar dos accesos a memoria para las instrucciones de carga y almacenamiento. Un primer acceso

es a la memoria principal para consultar la PTE y realizar la traducción y un segundo acceso, ya con la dirección física para la lectura o escritura del dato, se realiza a la caché. La solución adoptada por los procesadores actuales para reducir el tiempo de traducción es aplicar un mecanismo de memoria caché a la tabla de páginas. Esta memoria caché es parte de la MMU y se denomina TLB (*Translation Lookaside Buffer – Buffer de Traducción Anticipada*).

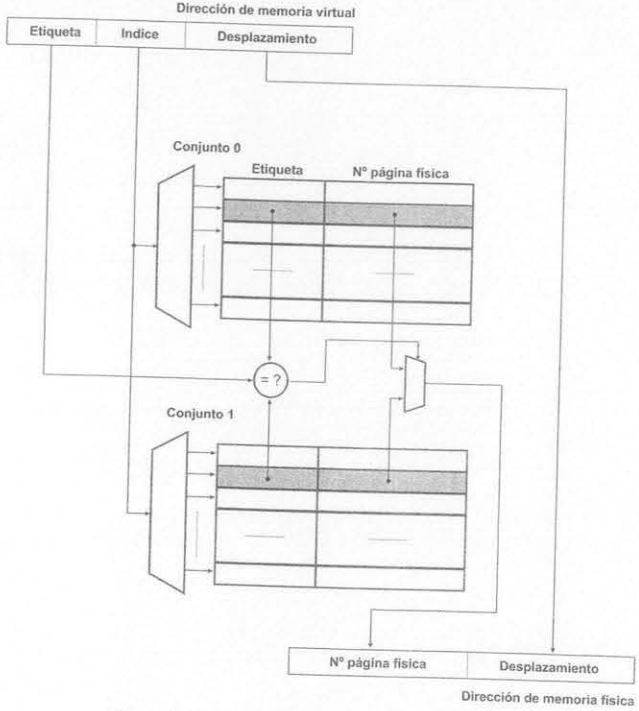


Figura A.3: TLB asociativa por conjuntos de dos vías.

La TLB, al igual que la memoria caché clásica, se puede organizar de varias formas: correspondencia directa, asociativa y asociativa por conjuntos. La Figura A.3 muestra un esquema de organización de la TLB asociativa por conjuntos de dos vías. Ahora, para acceder a la TLB, la dirección de página de memoria virtual se divide en *Etiqueta* e *Índice*. El resultado devuelto por la TLB es el número de página

física y se concatena con el desplazamiento para completar la dirección de memoria física. Recuerde que una organización de la TLB con asociatividad total implica que el campo *Índice* desaparece quedando únicamente *Índice* y *Desplazamiento*.

Dado que la TLB es una memoria caché de la tabla de páginas que se almacena en la memoria principal, también pueden aparecer *fallos de TLB*. Un fallo de TLB se produce cuando la equivalencia entre el número de página de memoria virtual y el de memoria física no está almacenada en la TLB y hay que recuperar otra parte de la tabla de páginas que se encuentra en la memoria principal. Esto provocará una detención de varios ciclos de reloj ya que la TLB debe acceder a memoria principal para reemplazar una parte de ella y recomenzar la traducción.

Otro fallo que se puede producir en la TLB es el ya conocido fallo de página. En este caso, no es que la equivalencia entre direcciones virtuales y físicas no esté en la TLB sino que la equivalencia entre página virtual y página física no existe en la tabla de páginas. Esto significa que la página virtual referenciada forma parte del espacio de direcciones virtuales que no está ubicado en la memoria principal, sino en los discos. Esto provoca la intervención del sistema operativo y su tratamiento implica:

- Decidir qué página de las comprendidas en la TLB hay que reemplazar.
- Si la página a reemplazar fue modificada, escribirla en el disco duro.
- Copiar la nueva página desde el disco duro a la memoria principal.
- Actualizar la tabla de páginas y la TLB.
- Recomenzar de nuevo el acceso a la TLB para obtener la traducción.

El resultado de la traducción mediante TLB es una dirección de memoria física que se emplea para acceder a la memoria caché. Por lo tanto, cualquier operación de memoria en un computador dotado de memoria virtual implica, como mínimo, dos accesos a caché: uno a la TLB para realizar la traducción de virtual a física y un segundo acceso a la caché de datos o instrucciones utilizando el resultado de la TLB. La relación entre la TLB y la memoria caché se muestra en la Figura A.4.

En este ejemplo, la dirección de memoria virtual tiene una longitud de $(v + g)$ bits, de los cuales v bits corresponden a la página de memoria virtual y g bits establecen el desplazamiento dentro de la página. De acuerdo con esto, el espacio de direccionamiento virtual constaría de 2^v páginas de 2^g bytes de longitud. En la Figura A.4, la TLB es asociativa por conjuntos por lo que el campo correspondiente al número de página virtual se divide en *Etiqueta* ($v - k$ bits) e *Índice* (k bits). El acceso a la TLB tiene como resultado un número de página física de f bits que concatenados a los g bits del desplazamiento de página forman la dirección de memoria física. Por lo tanto, el espacio de direccionamiento físico está formado por 2^f páginas de 2^g bytes de longitud. Dado que el espacio virtual es muy superior al físico se cumple que $2^f < 2^v$. La dirección de memoria física de $(f + g)$ bits se emplea para acceder a la caché. El desglose de esta dirección en los campos *Índice*, *Etiqueta* y *Desplazamiento* para acceder a la caché depende de la organización de la memoria caché que se utilice. En el ejemplo de la Figura A.4, la caché es asociativa por conjuntos con un tamaño del bloque de 2^b bytes.

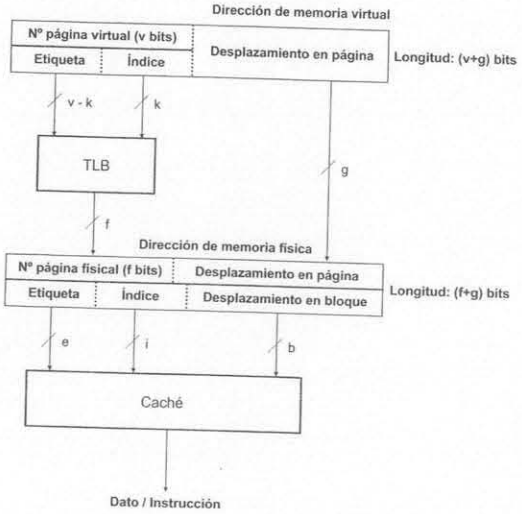


Figura A.4: Relación entre la TLB y la memoria caché.

Para entender mejora la relación entre memoria virtual, memoria física y memoria caché, lo mejor es un sencillo ejemplo como el que refleja la Figura A.5. Suponga un procesador superescalar con un espacio de direccionamiento virtual de 32 bits (4 Gbytes) y un espacio de direccionamiento físico de 23 bits que equivale a una memoria principal de 8 Mbytes. Si se utiliza un tamaño de página de 2 Kbytes, esto significa que la longitud del desplazamiento debe ser $g = 11$ bits, lo que deja $v = 21$ bits de la dirección virtual para referenciar una página. Por lo tanto, el total de páginas de que consta el espacio de direccionamiento virtual es $2^{21} = 2{,}097{,}152$ páginas. Dado que se conoce el desplazamiento de página, $g = 11$ bits, y la longitud de una dirección física, 23 bits, el número de páginas de que consta el espacio de direccionamiento físico es $2^{(23-11)} = 212 = 4096$ páginas. Para almacenar la TLB el procesador utiliza una memoria caché de 8 Kbytes con asociatividad por conjuntos de 2 vías y un tamaño de bloque de 2 bytes, suficiente para almacenar los $f = 12$ bits de longitud del número de página física y 4 bits con información de estado. Acceder a este tipo de memoria caché implica dividir los $v = 21$ bits del número de página virtual en *Etiqueta* e *Índice* ya que el desplazamiento dentro del bloque se considera 0. De acuerdo con estos datos, ya que cada conjunto de 2 vías ocupa 4 bytes, la caché tendrá 2^{11} conjuntos por lo que el tamaño del *Índice* será $k = 11$ bits y la *Etiqueta* $(v - k)$=10 bits. La memoria caché del procesador también es asociativa por conjuntos de 2 vías con un tamaño de 128 Kbytes y bloques de 128 bytes. Esto implica un campo *Desplazamiento* en bloque de $b = 7$ bits, dejando para los campos *Etiqueta*

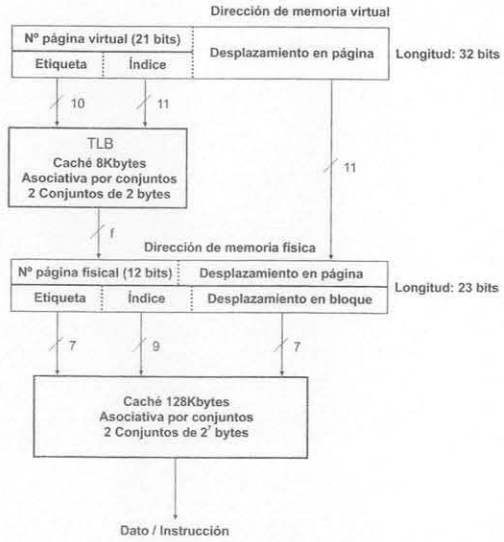


Figura A.5: Ejemplo de dimensionamiento de la TLB y la memoria caché.

e *Índice* un total de $23 - 7 = 16$ bits. Dado que la caché tiene 128 Kbytes (2^{17}) y la asociatividad es de 2 bloques de 128 bytes por conjunto (lo que implica 2^8 bytes por conjunto), habrá $2^{(17-8)} = 2^9$ conjuntos de 2^8 bytes, por lo que tamaño del *Índice* será $i = 9$ bits lo que deja $e = 16 - 9 = 7$ bits para el campo *Etiqueta*.

El esquema de la Figura A.4 corresponde a las etapas segunda y tercera de la unidad funcional de carga de un procesador superescalar. En una unidad funcional de carga, en el segundo paso o segmento se realiza la traducción mediante TLB y en el tercer paso se accede a la memoria caché. Si no existen fallos de TLB, fallos de página y fallos de bloque, el proceso de traducción consume un ciclo y el acceso a la caché para la lectura de un dato otro ciclo.

Para finalizar este apéndice, se describe de forma simplificada cómo funciona el sistema de memoria virtual del PowerPC 970. El PowerPC 970 utiliza un mecanismo de traducción de dos pasos basado en páginas segmentadas. En el primer paso, una dirección efectiva de 64 bits, generada por software, se traduce en una dirección virtual de 65 bits mediante una tabla de segmentos almacenada en memoria principal. Las entradas de la tabla de segmentos, denominadas STEs (*Segment Table Entry*), contienen descriptores de segmentos que definen las direcciones virtuales de los segmentos. En el segundo paso, la dirección virtual de 65 bits se traduce en una dirección física de 42 bits mediante una tabla de páginas,

también residente en memoria principal y formada por PTEs (*Page Table Entry*). Tanto la tabla de segmentos como la de páginas son utilizadas indistintamente para traducir direcciones de instrucciones y direcciones de datos. El tamaño de un segmento es de 256 Mbytes y el tamaño de la página puede oscilar entre los 4 Kbytes y los 16 Mbytes.

Para reducir el impacto en el rendimiento que producirían los accesos a memoria para consultar la tabla de segmentos y la de páginas, el PowerPC 970 incluye dentro del chip del procesador dos memorias cachés para la tabla de segmentos y la tabla de páginas, denominadas SLB (*Segment Lookaside Buffer*) y TLB (*Translation Lookaside Buffer*), respectivamente. La SLB es una memoria caché de 64 entradas completamente asociativa, mientras que la TLB cuenta con 1024 entradas y es asociativa por conjuntos de 4 vías. La TLB es gestionada por hardware mientras que la SLB es gestionada por el sistema operativo.

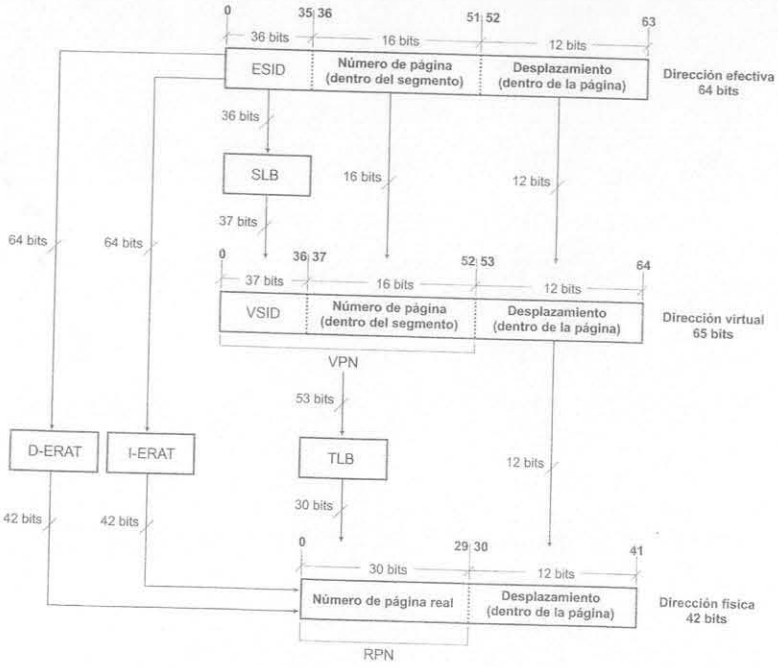


Figura A.6: Esquema de traducción de la MMU del PowePC 970.

La Figura A.6 corresponde al esquema de traducción de direcciones efectivas (65 bits) en físicas

(32 bits) que realiza la MMU del PowerPC 970. Una parte de la dirección efectiva, el ESID (*Effective Segment ID*) se emplea para acceder a las entradas STE de la SLB. Si se produce un fallo en la búsqueda en la SLB, la MMU accede a la tabla de segmentos almacenada en la memoria principal, y si se sigue produciendo un fallo, se produce un fallo de acceso a memoria que gestiona el sistema operativo. Si la STE necesaria se localiza en la tabla de segmentos, se copia en la SLB y continúa el proceso de traducción. Cada STE de la SLB contiene un descriptor de un segmento y se utiliza para generar una dirección virtual de 65 bits. La dirección virtual consta de un identificador de segmento virtual de 37 bits (*Virtual Segment ID* - VSID). La concatenación del VSID y del índice de página, obtenido de la dirección efectiva, forma el número de página virtual (*Virtual Page Number* - VPN) que se utiliza para acceder a la TLB que, como resultado, produce un número de página real (*Real Page Number* – RPN). Si se produce un fallo de TLB, se accede a la tabla de páginas en memoria principal para recuperar la entrada que contenga un número de página real. La unión del número de página real junto con el desplazamiento obtenido de la dirección efectiva forma la dirección física de 42 bits.

Con el fin de maximizar el rendimiento del mecanismo de traducción de direcciones, parte de la información contenida en la SLB y en la TLB se encuentra almacenada en un nuevo nivel de caché compuesto por dos pequeñas memorias: la I-ERAT (*Instruction Effective-to-Real Address Translation*) y la D-ERAT (*Data Effective-to-Real Address Translation*). Ambas memorias constan de 128 entradas y su organización es asociativa por conjuntos de dos vías. En la Figura A.6 se puede apreciar que las ERATs constituyen un mecanismo rápido de traducción cuando la dirección efectiva se encuentra almacena en una de las ERATs, evitando la penalización de acceso a la SLB y TLB. Las dos ERATs se rellenan a partir de las traducciones realizadas mediante la SLB y la TLB y son gestionadas por hardware.

A diferencia del esquema secuencial de acceso TLB-memoria caché-memoria principal descrito previamente, el PowerPC 970 utiliza una variante en la que la TLB y la SLB se encuentran por encima de la caché L1. Esto se debe a que el direccionamiento de la caché L1 se basa en índices derivados de direcciones efectivas y de etiquetas derivadas de direcciones físicas mientras que la caché de nivel 2 se direcciona únicamente mediante direcciones físicas. Por esta razón, el procesador se comunica directamente con la caché L1, la cual accede a la TLB y a la SLB cuando le es necesario.

Índice