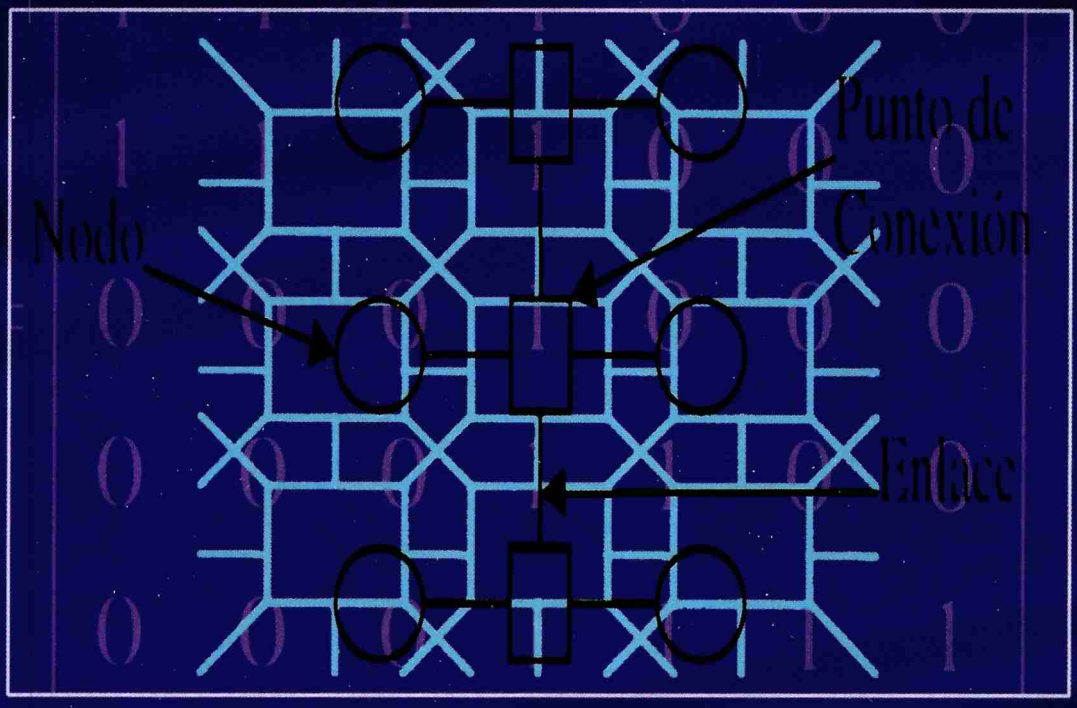
Procesamiento Paralelo

Conceptos de Arquitecturas y Algoritmos

Fernando G. Tinetti
Armando De Giusti



Editorial
Exacta

Fernando G. Tinetti nació en Tapalqué (Buenos Aires) en 1968.
Analista de Computación de la Universidad Nacional de La Plata (UNLP).
Licenciado en Informática de la UNLP.
Master en Informática de la Universidad Autónoma de Barcelona, y en el Programa de Doctorado de la misma universidad.
Docente del Departamento de Informática de la Facultad de Ciencias Exactas de la UNLP.
Autor de más de 20 trabajos publicados.
Temas de Investigación: Procesamiento Concurrente y Paralelo, especialmente en Arquitecturas NOW (Network of Workstations).

Armando E. De Giusti nació en Gualeguay (Entre Ríos) en 1950.
Egresó como Calculista Científico y como Ingeniero en Telecomunicaciones de la UNLP a los 22 años.
Es profesor Titular con dedicación exclusiva del Departamento de Informática de la Facultad de Ciencias Exactas de la UNLP e Investigador Principal del CONICET en el área Informática.
Ha sido Profesor Visitante de diversas universidades del país y del exterior.
Tiene 3 libros y más de 150 trabajos publicados en el país y en el exterior.
Dirige el Laboratorio de Investigación y Desarrollo en Informática (LIDI) de la UNLP, uno de los grupos más importantes del país en IyD en Informática.
Sus temas de investigación son Procesamiento Concurrente y Paralelo, especialmente en aplicaciones de Tiempo Real.
Es apasionado por el fútbol, el mate y Racing Club... en fin.

Procesamiento Paralelo

Conceptos de Arquitecturas y Algoritmos

Fernando G. Tinetti
Universidad Nacional de La Plata

Armando De Giusti
Universidad Nacional de La Plata

Procesamiento Paralelo. Conceptos de Arquitecturas y Algoritmos
Fernando G. Tinetti, Armando De Giusti

Primera Edición: Agosto de 1998.
ISBN 987-99858-5-0
Depósito que marca la Ley 11723
La Plata, Buenos Aires, República Argentina

A dos de mis mejores profesoras...

Lía Oubiña y Mirta Salerno.

Fernando G. Tinetti

A mis hijos Laura, Eduardo y Verónica que me alientan en el trabajo académico de todos los días y al Profesor Marcelo Naiouf con quien he discutido el enfoque y contenido de la mayoría de los temas tratados en el texto.

Armando De Giusti

Contenido

1. Introducción

El objetivo de este libro es presentar temas de procesamiento paralelo desde dos puntos de vista muy básicos: hardware y software. En todos los casos, tanto la presentación misma como el análisis de cada tema se enfocan al menos en una de dos direcciones: el rendimiento en la ejecución de aplicaciones y/o la naturalidad con que las aplicaciones paralelas se pueden resolver.

La intención no es abarcar *todos* los temas posibles de lo que tradicionalmente se ha denominado procesamiento paralelo. Lo que sí se ha intentado conservar es la idea de *aplicación* en el sentido de *utilización* del procesamiento paralelo. En el caso en que es utilizado para lograr mejores tiempos de ejecución en la resolución de aplicaciones con grandes requerimientos de cómputo, no se puede perder de vista el rendimiento de las máquinas paralelas. En el caso de utilizar procesamiento paralelo para acercar las soluciones computacionales a problemas que suceden en forma paralela, no es bueno distanciarse de la especificación y diseño de los programas paralelos. En medio de estas dos ideas de aplicación del cómputo paralelo (rendimiento y diseño de aplicaciones) hay una amplia gama de problemas a resolver.

La idea subyacente de aplicación del cómputo paralelo guía en cierto sentido la presentación de los temas. Los análisis teóricos, por ejemplo en términos de rendimiento o de posibles soluciones a un problema de hardware o software, son presentados haciendo explícitas las simplificaciones o las suposiciones necesarias. Los ejemplos de aplicación se presentan intentando extraer de ellos al menos algunas ideas comunes a la clase de problemas a la que pertenecen.

1.1 Por qué Cómputo Paralelo

Históricamente, la única forma de tratar algunos problemas de procesamiento ha sido por medio de cómputo paralelo [Akl89]. Existen muchas aplicaciones en las que es necesario procesar grandes cantidades de datos en muy poco tiempo. En todas estas aplicaciones la velocidad es crucial y, en principio, no alcanzable con computadoras secuenciales con un único procesador (monoprocesador) tal como se las conoce.

Es usual que los satélites recojan datos de la tierra a razón de 10^{10} bits por segundo, con información meteorológica, de contaminación, de agricultura y recursos naturales, etc. Para que esta información pueda utilizarse, necesita ser procesada a una velocidad de al menos 10^{13} operaciones por segundo. Las aplicaciones médicas, y particularmente las relacionadas con cirugía, necesitan una velocidad de procesamiento mínima de 10^{15} operaciones por segundo. El desarrollo de nuevas drogas, verificación y prueba de modelos de aviones, astronomía, procesamiento de señales, etc. son algunos de los campos en los que también se necesita gran cantidad de operaciones por segundo para obtener resultados aceptables al menos en un período razonable de tiempo.

Las dos opciones básicas para aumentar la velocidad de procesamiento a partir de los

monoprocesadores han sido: a) mejorar tecnológicamente el hardware monoprocesador, para que en sí mismo sea más rápido en procesar los mismos datos, o b) aumentar la cantidad de recursos de procesamiento (procesadores) sin grandes cambios en la tecnología utilizada. Estas dos opciones no necesariamente son disjuntas, aunque se refieren a conceptos diferentes. A grandes rasgos, mejorar tecnológicamente es casi lo mismo que aumentar la velocidad de ejecución de cada instrucción. Aumentar los recursos de procesamiento implica aumentar la cantidad de instrucciones que se ejecutan de forma simultánea. En ambas direcciones se tienen ventajas, límites e inconvenientes.

Aumentar la velocidad del hardware monoprocesador mejorando la tecnología tiene la ventaja de que se parte de una arquitectura básicamente idéntica, pero con mayor capacidad. En términos teóricos, lo que se refiere a la programación no debería actualizarse en función de la nueva tecnología, porque las instrucciones a ejecutar serían similares, o incluso las mismas. Por ejemplo, el paso a la nueva tecnología podría resolverse con un nuevo compilador. Se debe mencionar que la mejora tecnológica no es ilimitada. También hay otros inconvenientes en esta dirección. Por un lado, los problemas orientados naturalmente al paralelismo no son fáciles de resolver computacionalmente, y por el otro, el costo de las mejoras tecnológicas no es necesariamente proporcional a la mejora en el rendimiento [Sto93].

Aumentar la cantidad de elementos de procesamiento tiene la ventaja de aproximarse a los problemas en los que la paralelización es inmediata. Muchos de estos problemas, requieren además una gran cantidad de cómputo para responder a eventos que necesitan ser resueltos con un límite máximo de tiempo. Por otro lado, en términos de costo, no se deben desarrollar nuevos componentes complicados, se tiende a la utilización de los mismos componentes en mayores cantidades. Uno de los problemas más conocidos es la programación de este nuevo hardware. No solamente de las nuevas aplicaciones, sino de las aplicaciones que se deben "traducir" del monoprocesador. El paralelismo tiene también sus propios límites: a) la velocidad de comunicación entre los elementos de procesamiento, b) la necesidad de sincronización en la ejecución de las aplicaciones, c) la capacidad de paralelización de las aplicaciones.

A pesar de los avances en términos de velocidad de procesamiento que se han obtenido, es evidente que hay un límite: la velocidad de la luz en el vacío. Esta velocidad es aproximadamente 3×10^8 metros por segundo. Se puede aprovechar al máximo esta velocidad reduciendo la distancia entre los componentes y, por lo tanto, aumentando la velocidad de operación. Aunque dentro de un componente electrónico se puedan realizar operaciones muy rápidamente, si se necesita comunicar con otro componente, el tiempo de comunicación de las señales está limitado y se perdería la ganancia de velocidad. Por otro lado, usualmente no es posible integrar en un componente electrónico todas las funciones necesarias, porque no es posible reducir las distancias entre dispositivos electrónicos más allá del límite en donde comienzan a interactuar.

A partir de lo explicado anteriormente, se llega a que la única forma de tratar algunos problemas implica la utilización de procesamiento paralelo. Si varias operaciones pueden ser ejecutadas simultáneamente, el tiempo total de procesamiento se verá reducido, aún cuando cada una de las operaciones no se lleve a cabo más rápidamente. En este sentido, desde el punto de vista de las aplicaciones, es importante comprender que: *A mayor complejidad de cálculo y mayor compromiso con la ejecución en tiempo real (inteligencia artificial, redes*

neuronales, robótica, reconocimiento de patrones, visualización científica, modelos de elementos finitos y de fluidos, manejo de grandes bases de datos, etc.), se hace imprescindible utilizar procesamiento paralelo para obtener tiempos de respuesta aceptables. En la gran mayoría de los casos, las computadoras utilizadas para los problemas con los mayores requerimientos de cómputo se han denominado *supercomputadoras*, e históricamente también este término se ha relacionado de una forma u otra con el *procesamiento paralelo* [Hoc88].

También desde el punto de vista de las aplicaciones, muchos de los problemas que se encuentran para resolver computacionalmente son intrínsecamente paralelos. Se puede afirmar en este caso, que la naturaleza de algunos problemas tiende a la aplicación de procesamiento paralelo. En este contexto de aplicaciones, la idea de utilizar procesamiento paralelo no hace más que acercarse a la definición de los problemas del mundo real y por lo tanto reducir de forma natural la complejidad de las soluciones.

Por otra parte, la evolución de la tecnología tiende al procesamiento paralelo (en la actualidad, prácticamente no existen máquinas secuenciales puras) y asociada con la evolución tecnológica, se debe considerar que:

- Los costos favorecen la reducción de tamaño de los procesadores para incrementar el rendimiento. Alcanzar 1 Gigaflop (1000 millones de operaciones de punto flotante por segundo) es mucho más económico sobre 1000 procesadores de 1 Mflop cada uno que disponer de una supercomputadora capaz de realizar todas las operaciones sobre 1 procesador.
- La máxima velocidad alcanzable por el reloj de cualquier procesador implica un límite al mínimo ciclo de operación de un procesador sincrónico, lo que hace inevitable el paralelismo.
- La velocidad de procesamiento requiere componentes de tecnología especial que consumen más, lo cual disminuye la confiabilidad. En procesamiento paralelo se pueden utilizar tecnologías relativamente más lentas y también más confiables.
- Distribuir el procesamiento implica distribuir la memoria local, lo cual incrementa el ancho de banda global alcanzable por el sistema.

1.2 Definiciones y Conceptos Generales

Paralelismo: Ejecución concurrente (en el mismo instante de tiempo) sobre diferentes componentes físicos (procesadores). El paralelismo es un concepto asociado con la existencia de múltiples procesadores ejecutando un algoritmo en forma coordinada y cooperante. Al mismo tiempo se requiere que el algoritmo admita una descomposición en múltiples procesos ejecutables en diferentes procesadores (concurrencia).

Cuando el sistema de hardware está formado por un conjunto de procesadores o elementos de procesamiento vinculados (por ejemplo una red), con capacidad de ejecutar coordinadamente un algoritmo general, se obtiene paralelismo. Una arquitectura paralela es el soporte de hardware para poder tener procesamiento concurrente real.

El concepto de arquitectura paralela se asocia con varios procesadores, homogéneos o no, con un soporte de sistema operativo y con un subsistema de comunicaciones (hardware y software). Se pueden tener miles de procesadores interconectados (como en los hipercubos o las redes neuronales) donde cada procesador tiene una limitada capacidad y memoria local (grano fino), o decenas de procesadores de mayor potencia y heterogeneidad (grano grueso).

Objetivos del Procesamiento Paralelo:

- Disminuir los tiempos de ejecución.
- Incrementar la eficiencia.
- Atender fenómenos del mundo real que suceden en paralelo.

Proceso y Procesador: Un proceso es un bloque de programa secuencial, con su propio seguimiento de control. El concepto de proceso es el concepto básico e inicial de la programación concurrente: si en el sistema existen procesos independientes, existe la concurrencia. Cada proceso puede residir en un procesador independiente o dedicado. También se pueden tener múltiples procesos sobre el mismo procesador. Se debe notar que en este último caso se tiene concurrencia pero no paralelismo, o simultaneidad de ejecución.

Interacción, Comunicación y Sincronización de procesos: N procesos que residen en un procesador o en varios procesadores interactúan para ejecutar los aspectos del algoritmo global que requieran cooperación. La notación que normalmente se utiliza para definir este comportamiento es [Hoa86]:

$$P_1 / P_2 / .../ P_N$$

La interacción requiere comunicación para el intercambio de datos entre los procesos. La comunicación entre dos procesos puede ser por memoria compartida, a través de un mensaje explícito entre los procesos, o de un mensaje implícito por medio de un proceso servidor intermedio. La forma en que se lleve a cabo la comunicación depende de los mecanismos que se definan, que a su vez se relacionan de alguna manera con la arquitectura de procesamiento.

Cuando dos procesos necesitan ajustar el orden de ejecución de sus secuencias de instrucciones al estado de la ejecución del otro, se deben sincronizar. Desde el punto de vista de la programación de algoritmos sobre arquitecturas paralelas la sincronización es uno de los aspectos más relevantes.

Speed-Up ***(o Factor de Speed-Up)***: La relación entre el mejor tiempo de ejecución de un algoritmo sobre un procesador (T_1) y el tiempo de ejecución sobre una arquitectura paralela con N procesadores (T_N) se denomina factor de Speed-Up (S).

$$S = T_1 / T_N \tag{1.1}$$

- El óptimo que se puede esperar para S es N.
- Normalmente, resulta prácticamente imposible alcanzar el óptimo.
- Además, parece razonable pensar que más allá de un cierto N, para un dado problema algorítmico PA, las ineficiencias propias del algoritmo harán inútil el agregado de nuevos procesadores, es decir que S tendrá una cota máxima distinta de N.

La Ley de Amdahl [Amd67] expresa que para un problema dado, existe un Speed-Up máximo $Smax$, independiente de la cantidad de procesadores de la máquina paralela.

$$S \xrightarrow[N \to \infty]{} Smax \tag{1.2}$$

donde N representa la cantidad de procesadores. Quizás esta visión del Speed-Up no sea la correcta, o del todo correcta, y eso es lo que se puntualiza en [Gus88a] [Gus88b], al afirmar que el Speed-Up no debería observarse para un problema con un tamaño aislado. En este sentido, se tendría que tener en cuenta que una máquina paralela mayor puede ejecutar problemas de mayores dimensiones. Por ejemplo, no sería correcto evaluar el Speed-Up que se logra con una máquina paralela con mil procesadores en función de un problema que se puede resolver con una máquina paralela con 100 procesadores. Como alternativa a la ley de Amdhal se propone, entonces, un modelo de representación del Speed-Up que sea escalable con el tamaño del problema [Gus88b].

Eficiencia: La relación entre el Speed-Up alcanzado S y el óptimo teórico Sop se define como eficiencia (E).

$$E = S / Sop \tag{1.3}$$

O, lo que es igual,

$$E = S / N \tag{1.4}$$

tal como se la encuentra definida en [Kum94], y donde N representa la cantidad de procesadores. Es claro que

$$E \leq 1 \tag{1.5}$$

La eficiencia depende normalmente del tamaño del problema, tal como sucede muchas veces con el factor de Speed-Up. En este factor de eficiencia están englobados varios aspectos, como el balance de la carga computacional. Cuando la carga computacional de la aplicación está balanceada, se obtiene un grado de ocupación similar en cada uno de los procesadores que forman parte de la arquitectura paralela. El desbalance de carga tiende a secuencializar la ejecución de la solución algorítmica.

Impacto del procesamiento paralelo sobre el sistema operativo: para el desarrollo de los sistemas operativos, se debe pensar que la distribución física de algoritmos y datos significa una complejidad creciente en la administración de los recursos. Lo más usual es que los sistemas operativos ya desarrollados y probados en computadoras monoprocesador no funcionen sobre máquinas con arquitectura paralela, a menos que se introduzcan serias modificaciones en el núcleo (kernel) del software. Esta realidad se mantiene aunque el procesador de base de la máquina paralela sea el mismo que el de la computadora con arquitectura monoprocesador [Bac90].

Impacto del procesamiento paralelo sobre los lenguajes de especificación y programación de aplicaciones: los lenguajes de programación deben hacer posible la definición de procesos y su sincronización y comunicación. Se debe asegurar la posibilidad de cooperación o competencia por recursos según corresponda. Es deseable que esta especificación sea independiente de la arquitectura paralela de soporte. Naturalmente (como se ve al estudiar la programación concurrente), esto exige construcciones semánticas más complejas para los lenguajes de programación y/o especificación. También crece la complejidad requerida para verificar y validar algoritmos ejecutables sobre arquitecturas paralelas.

Es bastante claro que no alcanza con tener una máquina paralela muy potente para obtener buen rendimiento, Speed-Up, o tiempos de respuesta. Sí es cierto que es uno de los requisitos, pero no el único. Como mínimo, se debe programar la aplicación de forma tal que utilice adecuadamente todos los recursos disponibles del sistema y, en la medida de lo posible, que esta utilización sea óptima.

Como ya se puntualizó en parte, desde el punto de vista de las aplicaciones mismas es necesario que sean *paralelizables*: que sean posibles de programar soluciones de forma paralela. En este sentido, ya no alcanza que la máquina paralela sea muy potente y que se pueda programar adecuadamente, sino que las aplicaciones deberían permitir que el programa paralelo utilice todos los recursos de la forma más cercana a la óptima posible.

Por lo que se ha explicado, para lograr rendimiento cercano al óptimo, o el óptimo, es necesario que confluyan tres aspectos en la utilización de las computadoras paralelas:

1. Capacidad de procesamiento del hardware.
2. Capacidad de programación paralela (software) sobre el hardware.
3. Aplicación paralelizable.

1.3 Organización del Contenido

Como paso inicial dentro del área de procesamiento paralelo, se deben conocer las arquitecturas de cómputo paralelo para explotar al máximo la capacidad de procesamiento. Una vez que se conoce la arquitectura de una computadora paralela se debe programar de forma eficiente, como en el caso de las computadoras secuenciales. Es por esta razón que como paso inicial, en el capítulo siguiente, se verán algunas clasificaciones de las arquitecturas de procesamiento paralelo. Estas clasificaciones intentan describir los distintos modelos de cómputo subyacentes que se establecen para el procesamiento paralelo.

El Capítulo 3 describe las memorias cache y su aplicación en las arquitecturas paralelas. En general, las memorias cache se asocian a la mejora en la velocidad de acceso a los datos desde el procesador. Las computadoras paralelas rápidamente aprovecharon sus ventajas y las incorporaron como parte de su arquitectura. Algunos autores [Hwa93], realizan la clasificación de las arquitecturas tomándola como punto de referencia (Arquitecturas COMA). Teniendo en cuenta solamente esto, ya sería muy útil conocer las distintas alternativas de diseño de las memorias cache. De todas maneras, en las arquitecturas paralelas hay nuevos problemas para las memorias cache. Se hace necesario, por lo tanto, un mejor conocimiento del funcionamiento y los parámetros de diseño a tener en cuenta para poder proponer soluciones y/o comprender mejor las que se han propuesto.

En el Capítulo 4 se describe y analiza la utilización de buses de comunicación para interconectar procesadores a una memoria compartida común. Los buses de comunicación suelen usarse de forma bastante extendida en los multiprocesadores. Un único bus se utiliza cuando la cantidad de procesadores es pequeña, y no se necesita gran cantidad de accesos simultáneos a memoria. Se necesita más de un bus de comunicaciones cuando la demanda de accesos a memoria crece, que es usual a medida que crece la cantidad de procesadores. Se incluye asimismo una forma de realizar análisis de rendimiento para conocer los límites de utilización y aprovechamiento de este sistema de conexión procesadores-memoria.

El Capítulo 5 se dedica a la descripción de las redes de interconexión. Las redes de interconexión se utilizan en dos formas: para interconectar procesadores a memoria y para interconectar a los propios procesadores entre sí. El primer caso se da en las máquinas paralelas con arquitectura MIMD de memoria compartida, y generalmente se utilizan cuando no es posible conectar todos los elementos por medio de buses. El segundo caso se encuentra en las máquinas paralelas con arquitectura MIMD de memoria distribuida, donde es necesario transportar los mensajes que se envían desde un procesador a otro. La descripción se enfatiza en las características topológicas de las redes de interconexión. La interconexión entre procesadores de una máquina paralela con arquitectura SIMD también se realiza utilizando redes de interconexión.

En el Capítulo 6 se analizarán los problemas clásicos de tratamiento paralelo de datos, sobre los cuales se discutirán conceptos teóricos vistos en los capítulos anteriores. A medida que sea necesario, también se introducirán conceptos que clarifiquen ideas sobre los algoritmos paralelos, las arquitecturas paralelas a utilizar y/o el análisis de rendimiento que se realice. Los primeros problemas son clásicos en el contexto del tratamiento de datos organizados en *una dimensión*, o en *vectores*. Los últimos problemas tratan sobre el procesamiento clásico de datos organizados en *dos dimensiones*, o en *matrices*. No hay un tratamiento exhaustivo de todos los problemas de procesamiento de datos sino que se intenta presentar las bases sobre las cuales se apoya el procesamiento de datos organizados en una o en dos dimensiones.

El Capítulo 7 trata de enfocar detalladamente la atención en clases de problemas que se acercan a aplicaciones no numéricas. En cierto sentido, los problemas que se tratan en este capítulo son de más alto nivel de abstracción que los presentados en el capítulo anterior porque resuelven problemas más cercanos a los reales. Aún así, la forma en que se proponen soluciones a estos problemas en una máquina paralela y el análisis que se hace de las soluciones es muy similar. Se analizan problemas clásicos encontrados en estructuras de tipo grafo, en el tratamiento de mensajes distribuidos en una red estática de procesadores y algunos algoritmos de procesamiento de imágenes, poniendo énfasis en su paralelización. Todos estos problemas se han estudiado ampliamente desde el punto de vista algorítmico en general y también desde el punto de vista de los algoritmos paralelos.

En el Capítulo 8 se describen algunas características con respecto a la administración de recursos compartidos entre procesos de una aplicación paralela. En general, la presencia en el sistema paralelo de recursos compartidos degrada el máximo rendimiento teórico de una arquitectura paralela. La degradación de rendimiento relacionada con los recursos compartidos depende del tipo de la forma en que se utilizan y comparten los recursos y de la administración. Por lo tanto, El desarrollo de algoritmos eficientes de administración (scheduling) de los recursos es un aspecto importante, pocas veces tratado en los textos de programación paralela. Se utiliza el lenguaje de programación Ada como herramienta de especificación del esquema de los algoritmos por su claridad sintáctica y su precisión semántica. También se discuten algunos aspectos de los sistemas cliente-servidor de recursos.

En los últimos cuatro capítulos se describe de manera detallada un ejemplo de aplicación e investigación en el contexto de las máquinas paralelas MIMD con memoria distribuida. Por un lado, se presenta una discusión completa de lo que se ha realizado, dando una idea de la complejidad de la aplicación. Por otro lado, se intentan extraer ideas generales

y/o comunes a las aplicaciones paralelas al menos en este contexto. Estas ideas se refieren no solamente al diseño y especificación de aplicaciones paralelas, sino también al análisis y solución de los problemas encontrados y al análisis de los resultados obtenidos.

En el Capítulo 9 se presenta y se describe en términos generales la utilización de la migración de procesos para lograr balance de carga en una máquina paralela. Generalmente la migración es de aplicación en las máquinas paralelas con arquitectura MIMD de memoria distribuida, donde se deben gestionar los recursos de forma balanceada para obtener el máximo rendimiento posible. También en este capítulo se presenta y se discute el Problema de Arribo de Mensajes que se produce por las migraciones de procesos en máquinas con arquitectura MIMD de memoria distribuida. Se dan algunas características generales del problema y también se presentan algunas políticas de solución. En cada caso se intentan conocer los puntos favorables y desfavorables de cada solución propuesta.

El Capítulo 10 se dedica a la descripción del diseño de un entorno dedicado a la experimentación en el contexto de las arquitecturas MIMD con memoria distribuida, al que se le agrega la capacidad de migración dinámica de procesos. Se presenta también la adaptación del diseño presentado a las políticas propuestas en el capítulo anterior. Para conocer el comportamiento de cada política se deben definir los índices que se pueden utilizar para la evaluación y comparación de las políticas propuestas para resolver el Problema de Arribo de Mensajes. Es decir qué índices de rendimiento se tienen en cuenta y por qué. Dado que la experimentación se lleva a cabo en una máquina paralela real, debe ser posible la ejecución de aplicaciones paralelas de usuario. Por esta razón se define también un modelo sintético de aplicaciones de usuario para poder especificar distintos tipos de problemas que pueden presentar los programas paralelos.

En el Capítulo 11 se describen algunos detalles de implementación del entorno de experimentación que se presenta a nivel de diseño en el capítulo anterior. También se describe la implementación, experimentación y evaluación de las políticas propuestas para resolver el Problema de Arribo de Mensajes. Se deben definir los valores de los parámetros del entorno para conocer cómo se comporta cada política en distintos contextos de ejecución de aplicaciones de usuario.

En el Capítulo 12 se describen y analizan los resultados obtenidos con el entorno de experimentación presentado desde dos puntos de vista: (a) comparación de las distintas alternativas de resolución del Problema de Arribo de Mensajes, y (b) comparación de los valores obtenidos en trabajos previos. Ambos tipos de análisis se presentan para cada uno de los índices de rendimiento que se tienen en cuenta. También en este capítulo se dan algunas conclusiones con respecto al entorno de experimentación de migración dinámica de procesos en el contexto de las máquinas paralelas con arquitectura MIMD de memoria distribuida.

El Apéndice A presenta una introducción al análisis del tiempo de ejecución de algoritmos, y en particular al análisis de los algoritmos paralelos. También se introduce la forma de calcular los índices de rendimiento más utilizados, tales como el factor de Speed-Up. El análisis del tiempo de ejecución de los algoritmos constituye un mecanismo muy valioso de evaluación y comparación para obtener criterios de elección entre varias alternativas.

En el Apéndice B se hace un breve resumen de los aspectos útiles del lenguaje Ada

para la especificación de tareas concurrentes que se ejecutan sobre una arquitectura paralela. El lenguaje de programación Ada posee una gran riqueza expresiva que se origina especialmente para cumplir requerimientos del desarrollo de sistemas de tiempo real (mono o multiprocesador, distribuídos o no). Con Ada se cumplieron las etapas de especificación formal rigurosa del lenguaje antes de desarrollar compiladores y se cumple una validación estricta de los nuevos compiladores, lo cual favorece la portabilidad y la estandarización.

Finalmente se proporciona la bibliografía a la cual se hace referencia a lo largo de todo el material presentado.

2. Clasificaciones de las Arquitecturas de Procesamiento

Se han propuesto múltiples formas de organizar las arquitecturas de procesamiento paralelo, e incluso se considera un problema definir qué significa de manera precisa una "arquitectura paralela" [Dun90]. El problema de definir qué es, y luego dar una taxonomía de las arquitecturas de procesamiento paralelo, se encuentra por la gran cantidad de características que se tienen en una computadora paralela y que no todas son de fácil descripción, comparación y clasificación.

La clasificación establecida inicialmente en [Fly66] y [Fly72] se centra en la forma en que se ejecutan las instrucciones sobre los datos. Cualquier computadora, sea paralela o no, opera ejecutando instrucciones sobre datos. Un flujo (secuencia o stream) único de instrucciones indica qué hacer en cada paso de procesamiento. Un flujo (secuencia o stream) único de datos, o datos de entrada, son los procesados por estas instrucciones. Dependiendo de si hay uno o más de cada uno de estos elementos, se distinguen cuatro clases básicas:

1. SISD: **S**ingle **I**nstruction stream, **S**ingle **D**ata stream.
2. MISD: **M**ultiple **I**nstruction stream, **S**ingle **D**ata stream.
3. SIMD: **S**ingle **I**nstruction stream, **M**ultiple **D**ata stream.
4. MIMD: **M**ultiple **I**nstruction stream, **M**ultiple **D**ata stream.

Estas cuatro clases suelen utilizarse como modelo de referencia, cada autor en algunos casos las define de manera distinta de otros, y también se le han agregado múltiples características y propuesto alternativas de clasificación. Se describirán a continuación de forma no exhaustiva y sin mencionar ejemplos de utilización. La clase MIMD es la que se describirá con mayor nivel de detalle, por al menos dos razones: (a) es la clase que se considera más flexible por cuanto son útiles para una amplia gama de problemas, y (b) los capítulos siguientes tienen muchas referencias a esta clase de arquitecturas, aunque también se hará referencia a la clase SIMD.

Las computadoras con arquitectura SISD son las de procesamiento secuencial clásico: las instrucciones se ejecutan una después de otra, en serie. En la organización de estas computadoras se pueden encontrar: una Unidad de Procesamiento (PU), una Unidad de Control (CU) y la Unidad de Memoria (MU) donde se almacenan tanto datos como instrucciones. Se pueden describir esquemáticamente como en la Fig. 2.1.

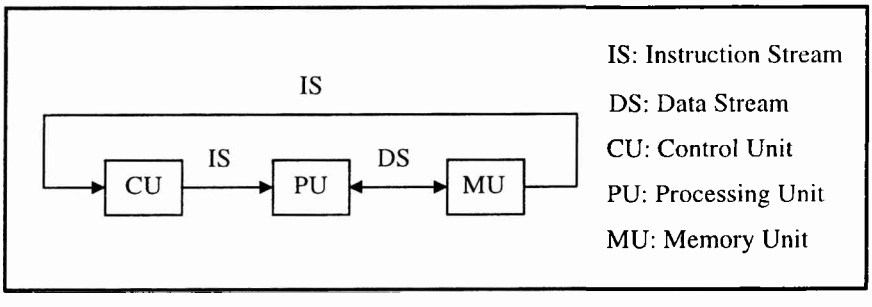


Figura 2.1: Arquitectura SISD Monoprocesador.

En las computadoras con arquitectura SISD (Fig. 2.1), la unidad de procesamiento es la encargada de llevar a cabo la ejecución efectiva de las instrucciones sobre los datos. La unidad de control es la encargada de decodificar las instrucciones para enviar las señales de ejecución correspondientes a la unidad de procesamiento. La unidad de memoria es relativamente *pasiva*: recibe y almacena los datos de escritura (Write) y proporciona los datos de lectura (Read) por requerimientos de las demás unidades.

En las computadoras con arquitectura MISD un mismo dato es procesado por múltiples instrucciones en distintas unidades de procesamiento. En [Hwa93] se las describe esquemáticamente como lo muestra la Fig. 2.2.

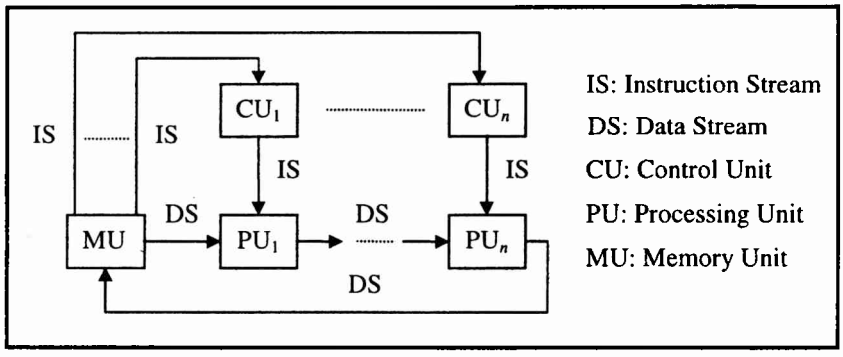


Figura 2.2: Arquitectura MISD (Systolic Array) en [Hwa93].

Por su parte, en [Akl89] se describen las arquitecturas MISD como en la Fig. 2.3. Se puede notar por comparación de ambas figuras, que ya al nivel de definición de la misma clase de computadoras las diferencias son muy significativas.

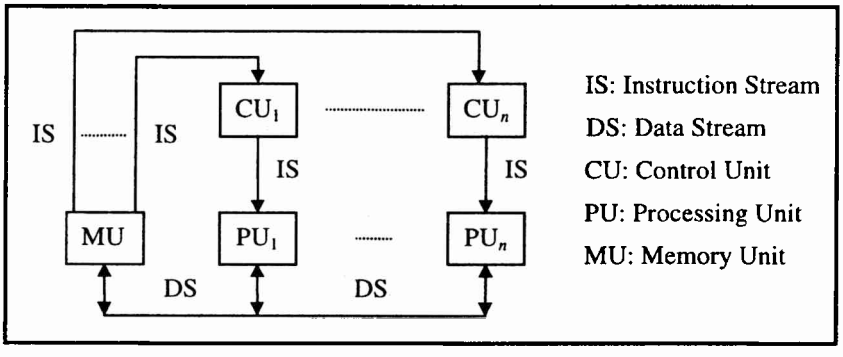


Figura 2.3: Arquitectura MISD en [Akl89].

Por lo tanto, aunque las clasificaciones tengan el mismo origen de referencia ([Fly66] y [Fly72]), la forma en que se describen las clases pueden diferir. Es por esta razón que luego se pueden producir confusiones, y encontrar la misma computadora en clases diferentes según el autor. En el caso particular de esta clase (MISD), en general se está de acuerdo en que por lo menos se trata de arquitecturas de procesamiento paralelo específicas. Este tipo de arquitecturas se adapta a una clase de problemas y no se considera de propósito general. En muchos casos, se la considera solamente como una posibilidad hipotética y no muy práctica [Dun90], aunque en [Akl89] se pueden encontrar algunos ejemplos de utilización en los que no se entrará en detalles.

Las computadoras con arquitectura SIMD constan de un conjunto de elementos de procesamiento (PEs: **P**rocessing **E**lements) idénticos, todos controlados por una única Unidad de Control (CU: Control Unit). La ejecución es sincrónica (lock-step operation), dado que hay una CU compartida y también un reloj global. En algunos casos se pueden habilitar o deshabilitar PEs para que ejecuten o no la siguiente instrucción distribuida desde la CU. Cada instrucción que se ejecuta en un PE procesa dato/s distinto/s de los demás. Cada PE puede, a su vez, tener memoria local a la cual accede con exclusividad. La Fig. 2.4 esquematiza las computadoras de arquitectura SIMD con memoria local en cada PE. Los elementos de procesamiento pueden estar comunicados por medio de memoria compartida o de una red de comunicaciones.

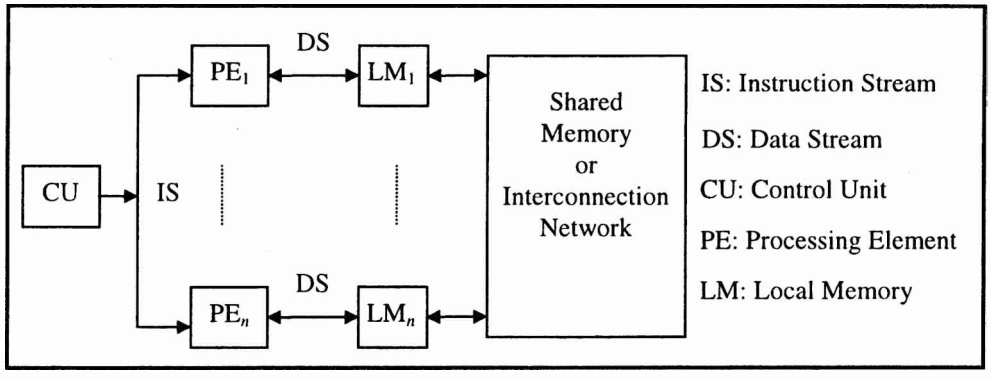


Figura 2.4: Arquitectura SIMD con Memoria Distribuida

La clase de arquitectura MIMD es la que se considera intrínsecamente paralela en [Hwa93], y sin lugar a dudas es aceptada por la mayoría de autores como de propósito general. Las computadoras que se incluyen dentro de esta clase constan de *n* procesadores del tipo que se mostró en la clase MISD, es decir con su propia unidad de control. Por esta razón, cada uno de los procesadores puede ejecutar su propia secuencia de instrucciones. Además, y a diferencia de las computadoras de la clase MISD, cada secuencia de instrucciones actúa (se ejecuta) en un procesador con diferentes datos de los que se utilizan en los demás procesadores. La forma en que se conectan los procesadores a la memoria y también entre sí permite diferenciar al menos dos subclases: los *multiprocesadores* y las *multicomputadoras*.

Los multiprocesadores, también llamados computadoras fuertemente acopladas (tightly coupled machines) son máquinas con arquitectura MIMD donde todos los procesadores comparten una única memoria. La estructura básica de la arquitectura MIMD de memoria compartida (multiprocesador) se puede esquematizar como en la Fig. 2.5.

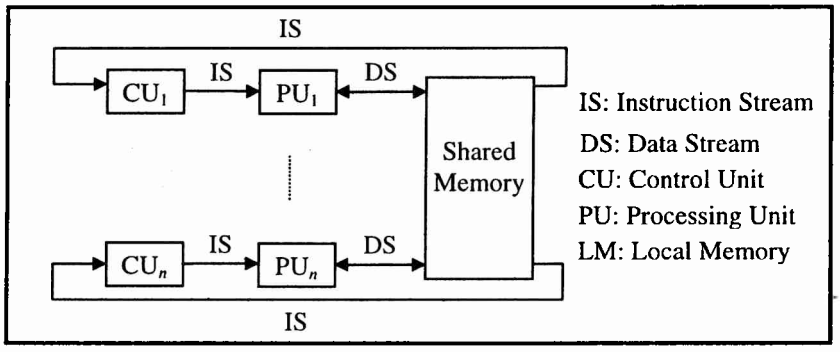


Figura 2.5: Arquitectura MIMD con Memoria Compartida.

En los multiprocesadores (Fig.2.5), el espacio de direccionamiento es común para todos los procesadores, una posición de memoria tiene el mismo contenido para todos ellos. Por lo tanto, la comunicación entre los procesadores se lleva a cabo por medio de la memoria compartida, donde se proveen métodos de acceso sincronizado para no tener interferencias en las áreas de datos comunes.

Las multicomputadoras reciben varios nombres alternativos: DMPC (**D**istributed **M**emory **P**arallel **C**omputers) [Sim97], computadoras débilmente acopladas (losely coupled machines), multicomputadores, message-passing multicomputers, y también computadores con arquitectura de memoria distribuida [Lew92]. Esquemáticamente, la arquitectura MIMD con memoria distribuida se puede describir como lo muestra la Fig. 2.6.

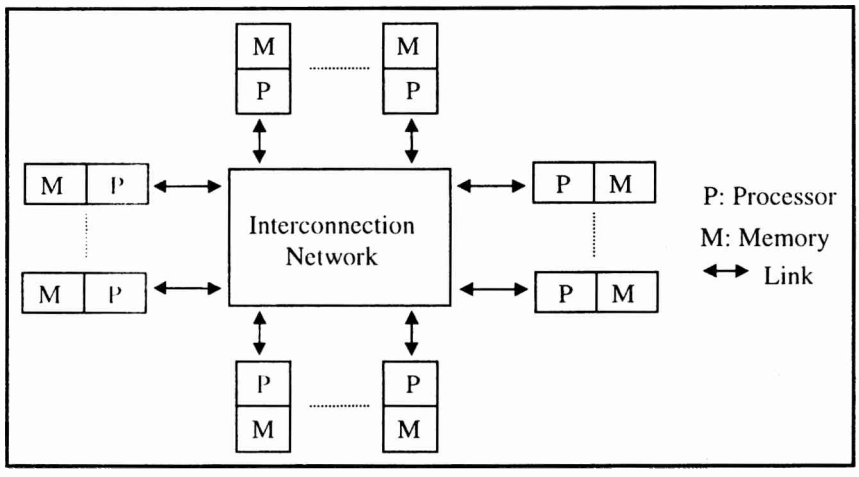


Figura 2.6: Arquitectura MIMD con Memoria Distribuida.

Cada procesador (P) equivale a la unidad de control (CU) más la unidad de procesamiento (PU). Los streams de datos y de instrucciones (DS y DI) de cada procesador provienen de su propia memoria. A través de los enlaces (*links*) de comunicaciones más la red de interconexión se pueden enviar mensajes a los demás procesadores. Se podría definir como múltiples SISD interconectadas.

Con respecto a la interpretación de esta clasificación inicialmente dada en [Fly66] y [Fly72], en [Hwa93] se define la clase MISD según la Fig. 2.2 tal como los Systolic Arrays. También en [Hwa93] no se incluyen a los procesadores vectoriales en la clase de arquitecturas SIMD, pero quedan como una clase fuertemente relacionada con ella. Los procesadores vectoriales son los que poseen múltiples pipelines de procesamiento vectorial de datos que pueden ser utilizados de forma concurrente. De esta manera se define que los procesadores vectoriales hacen uso intensivo del procesamiento pipeline para explotar el paralelismo temporal que se encuentra en el cómputo con vectores. La clase SIMD se describe como dedicada al procesamiento vectorial sincrónico, y se enfoca en el aprovechamiento del paralelismo espacial que se puede aprovechar en el cómputo con vectores de datos. También puntualiza que con memoria asociativa se pueden construir computadoras que se incluyen dentro de la clase SIMD.

Dentro de los Multiprocesadores, se diferencian tres modelos de acuerdo a la forma en que se accede a la memoria compartida por todos los procesadores. En el modelo UMA

(Uniform-Memory-Access), la memoria física es uniformemente compartida por todos los procesadores. Todos los procesadores tienen igual tiempo de acceso a todas las posiciones de memoria. Este caso sigue las líneas generales de descripción de la Fig. 2.5. Si todos los procesadores acceden de igual forma a todos los dispositivos periféricos, el sistema se denomina Multiprocesador Simétrico. En un Multiprocesador Asimétrico solamente uno, o un subconjunto de los procesadores puede ejecutar el sistema operativo completo y manejar la entrada/salida.

En el modelo NUMA (Nonuniform-Memory-Access) el tiempo de acceso a la memoria depende de la posición a la cual se acceda. Estos multiprocesadores se pueden definir como de memoria compartida pero físicamente distribuida. Cada procesador tiene algunas posiciones de memoria más cercanas y de menor tiempo de acceso que otras. Una posibilidad es compartir las memorias locales de cada procesador haciéndolas accesibles por medio de una red de interconexión, y la otra posibilidad es establecer una jerarquía de accesos a memoria donde el nivel en la jerarquía indique de alguna manera el tiempo de acceso que se tiene desde cada procesador.

La Fig. 2.7 esquematiza el caso en que las memorias locales de cada procesador de una arquitectura MIMD se comparten. El tiempo de acceso mínimo se logra cuando un procesador accede a su memoria local (LM), y el tiempo de acceso a las demás memorias locales depende de la topología y de la carga de la red de interconexión. En ambos casos (accesos a memoria local o no local) se debe tener en cuenta la posibilidad de interferencias, es decir, de accesos desde otros procesadores. En todos los casos en que dos o más procesadores acceden al mismo módulo de memoria, necesariamente los accesos se deben secuencializar y, por lo tanto, también se secuencializa en este punto la ejecución de los procesos.

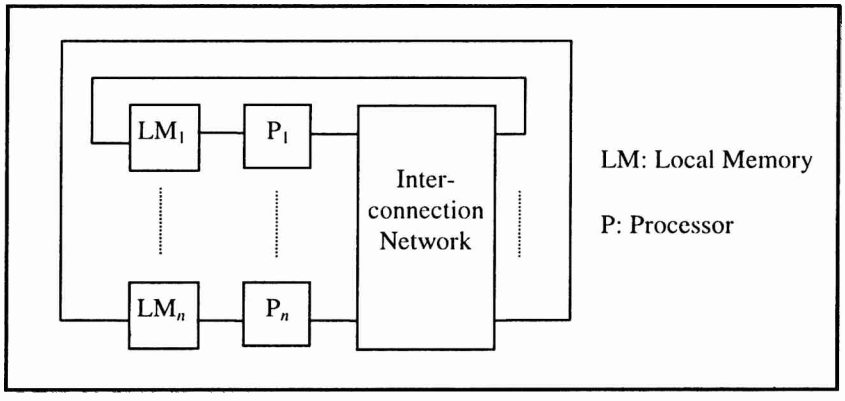


Figura 2.7: Multiprocesador con Memorias Locales Compartidas.

La Fig. 2.8 esquematiza el caso en que se establece una jerarquía de memorias a las cuales se accede desde un procesador. Si se organizan los accesos por grupos (clusters), se pueden encontrar tres tipos de accesos a memoria: local, global y remoto (a otras memorias locales). Los procesadores se organizan en clusters, donde cada cluster puede ser considerado un multiprocesador UMA o NUMA. En la Fig. 2.8, el tiempo de acceso de cada procesador a una posición de memoria en particular depende del tipo de acceso que debe realizar. La mayor velocidad en alcanzar un dato se produce cuando el acceso es local. Los accesos a memoria global (GSM en la Fig. 2.8) son más lentos que los accesos locales, pero más rápidos que los accesos remotos. El tiempo de cada acceso remoto no solamente es mayor que los demás

(acceso local y acceso global), sino que además puede depender de la cantidad de clusters y de la posición relativa de los clusters.

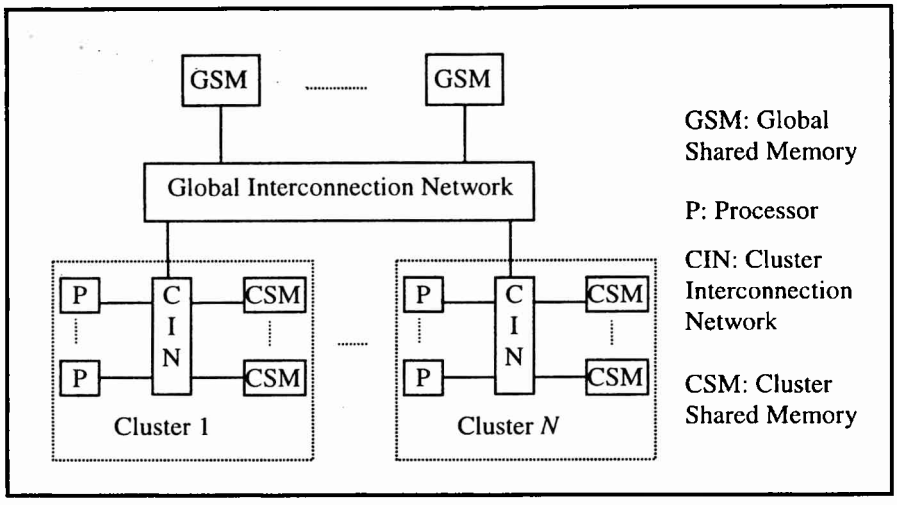


Figura 2.8: Multiprocesador con Acceso a Memoria Jerárquico.

El modelo COMA (**C**ache-**O**nly-**M**emory-**A**ccess) puede considerarse una caso especial de computadora NUMA, donde las memorias compartidas principales son convertidas en memoria cache. Todas las memorias cache forman un espacio de direccionamiento global. El acceso a memoria cache no local es resuelto por medio de directorios de cache distribuidos. El mantenimiento y utilización de la información en los directorios agrega una gran complejidad de hardware, lo cual a su vez aumenta el costo total de la arquitectura. Los datos eventualmente migran a la memoria cache del procesador que los utiliza. La Fig. 2.9 esquematiza las computadoras con este tipo de arquitecturas.

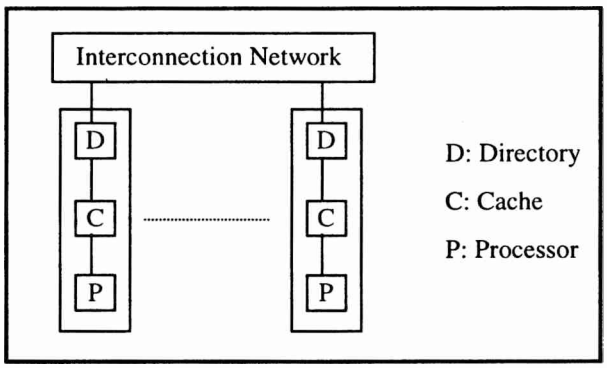


Figura 2.9: Modelo de Arquitectura COMA.

La mayoría de las clasificaciones de arquitectura que se han propuesto, utilizan o incluyen la clasificación que ha sido mencionada [Fly66] [Fly72]. Por ejemplo, en [Dun90] las arquitecturas de procesamiento paralelo se clasifican como lo muestra la Fig. 2.10. A grandes rasgos, esta clasificación inicialmente separa las arquitecturas sincrónicas (asociadas a un único reloj y ejecución lock-step) de las MIMD. Por otro lado, dentro de las arquitecturas de la clase sincrónica separa lo que originalmente se propuso como SIMD de lo que son las arquitecturas de procesamiento vectorial con pipeline. También mantiene dentro de la clase MIMD la separación entre las arquitecturas con memoria compartida o memoria distribuida.

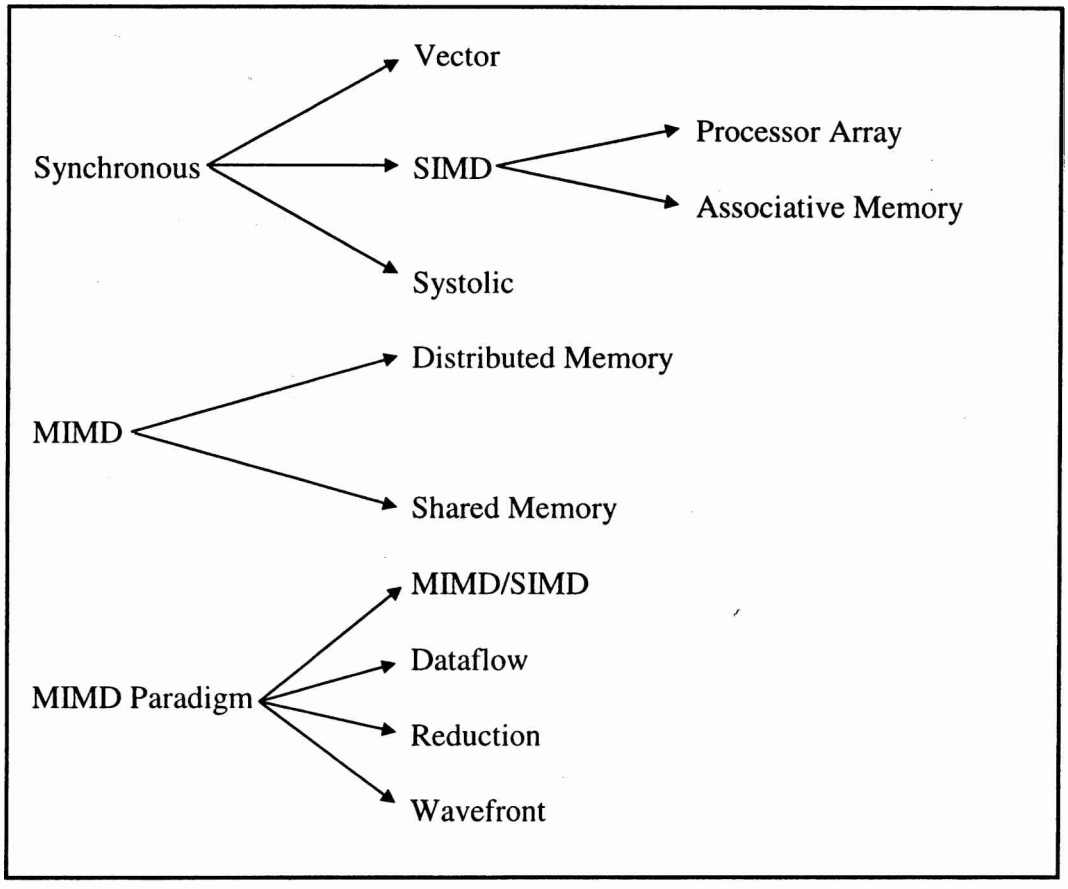


Figura 2.10: Taxonomía de Arquitecturas Paralelas propuesta en [Dun90].

Como se ha dicho anteriormente, las clasificaciones pueden ser confusas y ambiguas. Por ejemplo, se puede dar el caso en el que una computadora paralela no pueda ser incluida en ninguna clase de las propuestas, o se pueda incluir en más de una clase. En [Dec89], por ejemplo, la clase SIMD se subdivide como lo muestra la Fig. 2.11. También en [Dec89], las máquinas paralelas con arquitectura MIMD se subdividen dos clases: (a) las que utilizan pipeline (pipelined MIMD) y (b) las que se diferencian según la red de interconexión (Connectivity differentiation). Esta última subclase se subdivide una vez más para cada tipo de interconexión. Cuando las clasificaciones no tienen claros los parámetros sobre los cuales se definen las clases, se llega a una confusión no solamente en las máquinas paralelas que contiene cada clase, sino también en los términos que se utilizan. De esta manera se llega a una gran confusión de términos y de separación de similitudes o diferencias entre las máquinas paralelas que sea innecesaria.

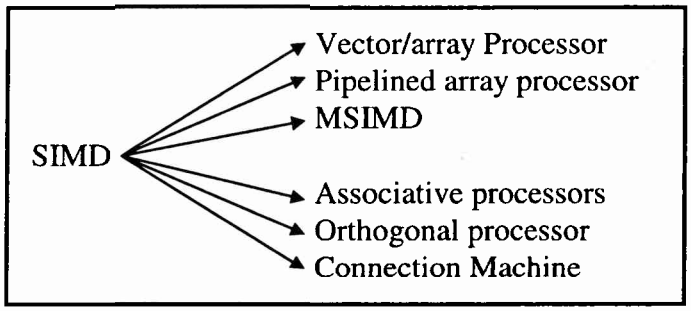


Figura 2.11: Clasificación de las Arquitecturas SIMD en [Dec89].

Las dificultades de descripción y clasificación son extensivamente puntualizadas en [Hoc88] y es por eso que propone una notación estructural de tipo algebraica (ASN: Algebraic-Style Notation) para describir las arquitecturas. Por ejemplo, un procesador Z80 con su memoria se describe en [Hoc88] con la fórmula

$$C(Z80 + \text{memoria}) = C(Z80)_{\bar{8}}M_{64k*8};\ C(Z80) = I_8^{250}\left[B_{88}M_{18*8}\right]$$

Sin entrar en los detalles de la notación, sí se puede afirmar que suele llegar a ser muy complicada cuando las características aumentan y la complejidad de la máquina es mayor. Por ejemplo, la arquitectura de la CRAY X-MP puede describirse de forma simplificada con las ecuaciones que se dan a continuación. La primera ecuación de alguna manera describe la arquitectura "completa" y las demás ecuaciones refinan el nivel de detalle de la descripción.

$$C(\text{CRAY X - MP / nm}) = nC1(\text{CPU}) \times (8m)M1_{128k*64}^{38}(\text{common});$$

$$C1(\text{CPU}) = Iv_{16,32}^{9.5}\left[14\,Ep - M2 - \left\{\left\langle\frac{9.5}{64}, \left\langle\frac{9.5}{64}, \frac{9.5}{64}\right\rangle\right\rangle\right\}\right]_r;$$

$$M2(\text{registers}) = \left\{8M_{64*64}(\text{vector}),\ 72M_{1*64}(\text{scalar}),\ 72M_{1*24}(\text{address})\right\};$$

$$14Ep = \left\{3Fp_{64},\ 5Bp_{64},\ 4Bp_{64},\ 2Bp_{24}\right\};$$

$$3Fp_{64}(\text{floating - point}) = \left\{Fp_{64}(+),\ Fp_{64}(*),\ Fp_{64}(\div)\right\};$$

$$5Bp_{64}(\text{vector}) = \left\{Bp_{64}(\text{integer} +),\ Bp_{64}(\text{shift}),\ 2Bp_{64}(\text{logical}),\ Bp_{64}(\text{pop.count})\right\};$$

$$4Bp_{64}(\text{scalar}) = \left\{Bp_{64}(\text{integer} +),\ Bp_{64}(\text{shift}),\ Bp_{64}(\text{logical}),\ Bp_{64}(\text{pop.count})\right\};$$

$$2Bp_{64}(\text{address}) = \left\{Bp_{24}(\text{integer} +),\ Bp_{24}(\text{integer} *)\right\}$$

Otras clasificaciones se enfocan en diferentes formas de describir las arquitecturas. Por ejemplo, en [Hoc88] se describe una clasificación basada en la forma en que se organizan las computadoras de acuerdo a las partes que la constituyen. En [Ni91] se da una clasificación que se basa en la descripción de las arquitecturas por niveles de detalle (layered development).

3. Memoria Cache

Como se explicó previamente, las memorias cache inicialmente se relacionan más con la mejora en el tiempo de acceso a los datos de los procesadores que con el procesamiento paralelo [Sto93]. El principal interés al introducir una memoria cache en el sistema, es el de reducir el tiempo efectivo de acceso a los datos desde el procesador. De todas maneras, tanto la memoria cache como el procesamiento paralelo tienen, en general, un objetivo común: aumentar la velocidad de procesamiento de las computadoras. Pero la relación es mucho más evidente cuando se trata de diseñar y/o analizar las arquitecturas que se han denominado MIMD con un fuerte acoplamiento, es decir donde todos los procesadores comparten la misma memoria principal. Para comprender cómo es utilizada y aprovechada la memoria cache en este contexto, es necesario al menos hacer una definición general del funcionamiento de este tipo de memoria.

3.1 Principio de Funcionamiento de la Memoria Cache

La velocidad de acceso a los datos almacenados en la memoria principal es de vital importancia para determinar el tiempo de procesamiento de un procesador y también de una computadora paralela. Los índices de rendimiento de las memorias que están más relacionados con la velocidad de acceso a los datos son el *tiempo de acceso a memoria* y el *tiempo de ciclo de memoria* [Wil91]. El tiempo de acceso a memoria se define como el tiempo que transcurre desde que se envía un requerimiento a memoria hasta que se completa la transferencia de información con la dirección de memoria direccionada. El tiempo de ciclo de memoria se define como el tiempo mínimo que debe transcurrir entre dos operaciones de acceso a memoria. En todos los casos, lo que se presenta como una situación general es la diferencia entre las velocidades de funcionamiento de los procesadores y la memoria principal. Otra de las medidas de velocidad de acceso a memoria muy relacionada con las anteriores es el ancho de banda (bandwith) de la memoria [Sto93]. El ancho de banda se puede definir como la cantidad de bits (bytes, o palabras) por segundo que se pueden acceder a memoria.

Los procesadores pueden operar sobre los datos más rápidamente de lo que permite el tiempo de acceso de la memoria. Comúnmente, los procesadores pueden ejecutar instrucciones a mayor velocidad de la que se puede acceder a los datos, y por lo tanto la velocidad de acceso a memoria principal es aproximadamente la que se puede lograr en el procesamiento de los datos. En términos de velocidades relativas, la velocidad de procesamiento de los datos en el procesador es mayor que la velocidad en la cual la memoria principal puede responder a los requerimientos. Aunque es posible fabricar memorias que pueden operar a velocidades comparables con las de los procesadores, en general no es económicamente viable. Lo que se propone para que se puedan equiparar las velocidades es introducir una memoria intermedia de menor tamaño que la memoria principal, la *memoria cache*, entre el procesador y la memoria principal, que sea capaz de realizar operaciones a velocidad comparable con la del procesador.

La memoria cache es de mayor velocidad que la memoria principal, y para que sea

económicamente viable, es necesariamente de mucho menor tamaño que la memoria principal. Todos los requerimientos que se realicen desde el procesador serán respondidos por la memoria cache que se ubica entre el procesador y la memoria principal. Tanto los datos como las instrucciones se transfieren desde la memoria principal hacia la memoria cache y son accedidos desde allí por el procesador. Esquemáticamente, el tráfico de información entre el procesador, la memoria cache y la memoria principal se puede describir con la Fig. 3.1.

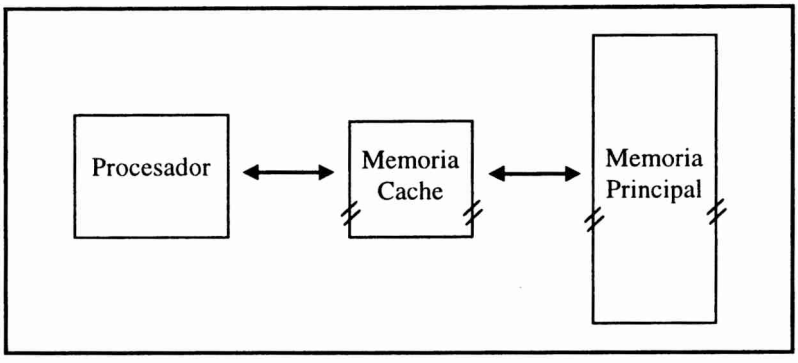


Figura 3.1: Memoria Cache.

Este modo de funcionamiento no resuelve por sí solo la posibilidad de que el procesador tenga que esperar por la menor velocidad de transferencia de información de la memoria principal. Si cada vez que se debe transferir información se debe acceder a la memoria principal y luego a la memoria cache, no solamente no se gana en velocidad sino que por el contrario todo el proceso se vuelve más lento, porque ahora hay una memoria intermedia que interviene en la transferencia de información.

Como sucede con la memoria virtual [Sil94] [Tan88], el *principio de localidad*, o *principio de localidad de referencia*, hace posible que la memoria cache logre su objetivo: mayor velocidad de acceso a datos e instrucciones desde el procesador. Una vez que la información se halla en la memoria cache, es utilizada muy frecuentemente desde el procesador, sin necesidad de nuevas transferencias entre la memoria principal y la memoria cache. Aunque el código de los programas se ejecuta de manera secuencial, en la gran mayoría de los programas se repite la ejecución de secciones de código y se accede a los mismos datos o a datos en posiciones cercanas de memoria. Por lo tanto, el principio de localidad se puede encontrar en la mayoría de los programas, y tiende a ser aplicado tanto a las referencias a instrucciones como a datos. Tiene dos aspectos principales:

1. *Localidad Temporal* (localidad en el tiempo): una vez que se referencia una dirección de memoria, es altamente probable que se vuelva a referenciar en el futuro cercano. Es decir que es altamente probable que la información que se utilizará en un futuro cercano haya sido referenciada muy recientemente.
2. *Localidad Espacial* (localidad en el espacio): es altamente probable que la próxima referencia a memoria sea cercana a la última referencia que se ha realizado. Las partes del espacio de memoria que se referencian durante un período relativamente corto de tiempo, generalmente son una pequeña cantidad de segmentos individuales de memoria consecutiva.

La localidad temporal se encuentra principalmente en iteraciones, tipos de datos *pila*, y accesos a variables. En períodos cortos de tiempo, un programa distribuye sus referencias a

memoria de manera no uniforme sobre el espacio de direccionamiento total. La localidad espacial describe el comportamiento de la ejecución de los programas en los que se accede a las instrucciones y a los datos asignados por *regiones* de memoria. La ejecución de las instrucciones es básicamente secuencial, y la mayoría de los saltos (branchs) no son a posiciones de memoria muy lejanas de la que apunta el contador de programa. Los datos son generalmente asignados a posiciones contiguas de memoria, principalmente los vectores y las matrices.

Suponiendo que una referencia a memoria es repetida n veces (por ejemplo dentro de una iteración) y, después de la primera referencia la información se encuentra siempre en la memoria cache, entonces el tiempo promedio de acceso a la información será [Wil91]

$$\overline{t_a} = \frac{(nt_c + t_m)}{n} = t_c + \frac{t_m}{n} \tag{3.1}$$

donde $\overline{t_a}$ es el tiempo promedio de acceso a la información, n es la cantidad de referencias, t_m es el tiempo de acceso a memoria principal, y t_c el tiempo de acceso a memoria cache. De esta manera, el tiempo de acceso a los datos desde el procesador se verá reducido en proporción a la cantidad de referencias se resuelvan en memoria cache (n).

Utilizando la Ec. (3.1), si por ejemplo, el tiempo de acceso a memoria cache está dado por t_c = 25 ns, el tiempo de acceso a memoria principal es t_m = 200 ns, y la cantidad de referencias es n = 10, el tiempo de acceso promedio a la información es de 45 ns gracias a la utilización de la memoria cache. El tiempo promedio de acceso a la información sin memoria cache es de 200 ns.

Cuanto mayor sea n (la cantidad de referencias a un dato que se encuentre en la memoria cache), el tiempo promedio de acceso será más aproximado al tiempo de acceso a memoria cache, dado que

$$\frac{t_m}{n} \xrightarrow[n \to \infty]{} 0 \tag{3.2}$$

Gracias a la localidad temporal y a la localidad espacial se puede afirmar que, en promedio, la información es accedida varias veces una vez que ha sido asignada en memoria cache.

3.2 Memoria Cache en las Arquitecturas MIMD

En los sistemas de cómputo MIMD donde todos los procesadores comparten una memoria principal común, se pueden observar al menos tres problemas [Ste90]:

- Interferencia de accesos a memoria. En general, la memoria no puede ser accedida simultáneamente por dos o más procesadores, y por lo tanto los requerimientos se deben secuencializar. De esta manera, el acceso a memoria se realiza más lentamente cuanto mayor sea la cantidad de conflictos entre los procesadores por accesos a la memoria principal.

- Interferencia en la red de comunicación, o carga de la red de comunicación. La red que conecta a los procesadores con la memoria compartida es otro de los recursos a los cuales se debe acceder y no tiene capacidad ilimitada. Se puede encontrar que dos requerimientos se deban secuencializar en un enlace de comunicación aunque estén dirigidos a distintos módulos de memoria.
- Tiempo de latencia. A medida que la cantidad de procesadores crece, la red de interconexión con la memoria se hace más compleja, y por lo tanto el tiempo de latencia de las redes de comunicación es mayor.

Los tres problemas mencionados contribuyen a aumentar el tiempo de acceso a memoria y por lo tanto disminuyen el tiempo de ejecución de los procesadores. La introducción de una memoria cache para cada procesador puede resolver gran parte de los tres problemas.

El objetivo cuando se introduce una memoria cache es que la mayoría de los accesos a memoria se realicen sobre ella. Como consecuencia inmediata, la cantidad de accesos a memoria principal se reduce considerablemente. Es bajo estas circunstancias que la memoria cache puede reducir la cantidad de conflictos entre procesadores y la utilización de la red de comunicaciones. La relación entre la frecuencia de acceso a memoria principal por parte de los procesadores y la probabilidad de conflictos es inmediata. Con memoria cache, cada procesador accede a memoria principal menos frecuentemente, y por lo tanto la probabilidad de que haya conflictos disminuye considerablemente.

El mayor conflicto que se encuentra para las memorias cache en las arquitecturas MIMD es lo que se ha denominado *coherencia de cache* (cache coherence), o *multicache consistency*. Dado que todos los procesadores tienen acceso a la memoria compartida, un mismo dato puede ser utilizado por más de un procesador. Con la utilización de memoria cache, cada procesador tendrá en su memoria cache los datos que utilice, y allí pueden ser actualizados. Si no se provee ningún método de control de la actualización de los datos en la memoria cache, cuando un procesador escribe una posición de memoria, puede suceder que ni la memoria principal ni las memorias cache de los demás procesadores puedan conocer el valor correcto del dato. Se llegaría de esta manera a que una misma posición de memoria tenga más de un valor, de acuerdo a las escrituras que se hayan llevado a cabo de forma local en las memorias cache de cada procesador. La Fig. 3.2 muestra una secuencia de pasos por las que se puede llegar a tener más de un valor para el mismo dato.

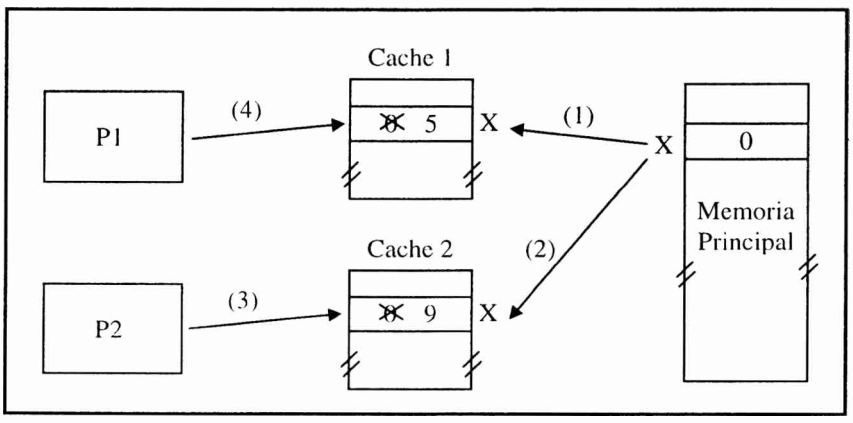


Figura 3.2: Falta de Coherencia de Memorias Cache.

Los dos primeros pasos, (1) y (2), son los de traslado del valor de la variable X a las memorias cache de los procesadores P1 y P2. En ambos casos se trae a memoria cache el valor de X en la memoria principal: 0. El tercer paso, (3), corresponde a la asignación que se hace en el procesador P2, X = 9. En el procesador P1, se lleva a cabo el cuarto paso, (4), que corresponde a la asignación X = 5. En este momento se tienen 3 valores para la variable X:

a) 0 en la memoria principal, con lo cual si otro procesador accede a la variable X en la memoria principal, obtendría este valor.
b) 5 para el procesador P1, ya que es el valor que obtendrá de su memoria cache.
c) 9 para el procesador P2, ya que es el valor que almacena su memoria cache para esa variable.

Teniendo en cuenta el ejemplo de la Fig. 3.2, después de ejecutado el paso (4), el valor de la variable X debería ser 5. Este valor debería ser el que se lea desde el procesador P1, desde el procesador P2, o desde cualquier otro procesador.

Un sistema de memorias cache es coherente si y solo si una lectura realizada por cualquier procesador de una ubicación de memoria principal X asignada en la cache (la cual puede estar asignada en la memoria cache de otro procesador), siempre proporciona el valor global más reciente de la posición de memoria X [Dec89]. La coherencia de memoria cache se hace posible sincronizando las operaciones de lectura y escritura de las memorias cache de cada procesador.

Si se ha decidido los procesadores compartirán la memoria principal, el objetivo inicial es que los datos que se utilicen sean los mismos en todos los procesadores. Por lo tanto, se debe mantener la coherencia de cache, y un dato debe tener siempre el mismo valor para cualquier procesador en cualquier instante de tiempo.

3.3 Parámetros de Diseño de Memoria Cache

Cuando se quiere optimizar el diseño de la memoria cache generalmente abarca cuatro aspectos [Smi82]:

1. Maximizar la probabilidad de encontrar un dato en la memoria cache (hit ratio). Es equivalente a reducir la probabilidad de no encontrar un dato en memoria cache (miss ratio).
2. Minimizar el tiempo de acceso a los datos que se encuentran en memoria cache (tiempo de acceso de la memoria cache)
3. Minimizar el retardo debido a la ausencia (miss) de un dato en memoria cache
4. Minimizar la sobrecarga que se produce por la actualización de la memoria principal, mantenimiento de la consistencia de múltiples memorias cache, etc.

Todos estos aspectos de optimización deben ser considerados dentro del límite impuesto por las restricciones de costo de las memorias cache. En las secciones que siguen se analizarán las diferentes alternativas para el diseño de una memoria cache. Para cada parámetro se explicará en qué consiste, si tiene relación con otros parámetros, y cómo afecta el modo de funcionamiento de la memoria cache, ya sea en cuanto al manejo de los datos como así también en cuanto a la complejidad de la implementación. Como suele suceder cuando se tienen múltiples índices a optimizar con múltiples condiciones a satisfacer (y en

particular el costo), los parámetros no son independientes. La situación para el análisis se torna más difícil cuando se tiene en cuenta que en la mayoría de los casos no se conoce el grado de correlación entre los distintos parámetros de diseño.

3.3.1 Tamaño de Memoria Cache

Este parámetro indica la cantidad de memoria cache. Es de esperar que a mayor cantidad de memoria cache se aumentará la probabilidad de encontrar un dato en memoria cache una vez que se ha referenciado. Lamentablemente el tamaño de la memoria cache no se puede aumentar indefinidamente y por lo tanto suele ser mucho menor que la memoria principal. Es por esta razón que se debe buscar, y se ha realizado de manera extensiva, la mejor relación costo/rendimiento.

En general para todos los programas y más aún en los entornos de procesamiento con multiprogramación, se debería lograr contener en la memoria cache la cantidad de datos necesarios para aprovechar el principio de localidad. Estos datos se denominan usualmente el working set del programa. El tamaño del working set no es uniforme, depende del programa, y para el mismo programa varía a medida que avanza la ejecución. Si la memoria cache es muy pequeña, cada vez que se deba asignar un dato de la memoria principal se deberá desalojar otro ya asignado en memoria cache. En el peor caso, para cada referencia habrá que recurrir a la memoria principal y, como se afirmó anteriormente, la velocidad de acceso a los datos se reduce de manera considerable. Muchos de los demás parámetros de diseño de la memoria cache dependen del tamaño, pero sin duda el costo total es el índice que más pesa a la hora de elegir la cantidad de memoria cache que se utilizará en el sistema.

3.3.2 Tamaño de Bloques y Cantidad de Bloques

La transferencia de información entre la memoria cache y la memoria principal sigue las líneas generales de la memoria virtual, y se realiza por conjuntos de posiciones de memoria consecutivas. El nombre de este conjunto suele ser *bloque*, aunque puede cambiarse según el diseño de la memoria cache por *sub-bloque* o *sector*. Con esta forma de transferencia se intenta aprovechar el principio de localidad espacial y además puede aprovechar el hecho de que la memoria esté dividida en módulos.

Tanto la memoria principal como la memoria cache se subdividen en bloques. Los bloques de la memoria cache suelen denominarse *frames*, *block frames* [Hwa93] o *líneas de cache*, para diferenciarlos de los bloques de la memoria principal. El tamaño de los bloques de la memoria principal y de los de la memoria cache es el mismo. A menos que se defina lo contrario, la unidad de transferencia entre la memoria principal y la memoria cache es un bloque.

Dado que la cantidad de bloques de memoria principal es mucho mayor que la cantidad de bloques de la memoria cache, se debe llevar a cabo algún método de identificación y asignación de bloques de memoria principal en memoria cache. Para identificar los bloques de memoria principal en la memoria cache, se asocia a cada uno de

ellos un *identificador*, *etiqueta* [Sta97], *index/tag* [Hwa93], o *tag* único que lo identifica. Es por esta razón que se puede describir la memoria cache compuesta por:

1. Directorio de la memoria cache, que indica para cada bloque de cache qué bloque de memoria principal lo ocupa o si está libre, más algunos bits de control. Si un bloque de memoria cache está ocupado por un bloque de memoria principal, en el directorio se almacena el tag que identifica el bloque de memoria principal.
2. Memoria de almacenamiento de datos, que es la que contiene los valores a los cuales se accede desde el procesador.

La Fig. 3.3 muestra de forma esquemática la relación entre la información en memoria principal y la memoria cache. La longitud de un bloque se determina en k posiciones consecutivas de memoria, que se transfiere de forma completa entre la memoria principal y la memoria cache. El tamaño de los bloques se define en términos de la cantidad de palabras de memoria que contiene. Si la memoria se puede direccionar de a bytes, el tamaño de un bloque generalmente se expresa en cantidad de bytes. En la Fig. 3.3 se tienen en cuenta los bloques, tags, y el directorio. La longitud de un bloque se determina en k posiciones consecutivas de memoria. El tamaño de los bloques generalmente se define en términos de la cantidad de palabras de memoria que contiene. Si la memoria se puede direccionar de a bytes, el tamaño de un bloque generalmente se expresa en cantidad de bytes.

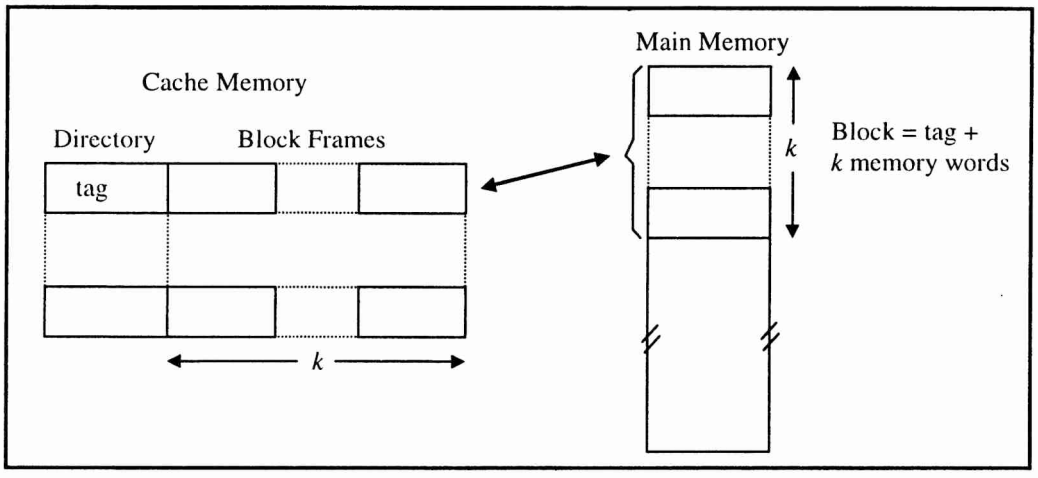


Figura 3.3: Bloques en Memoria Principal y en Memoria Cache.

De acuerdo con esta organización, para que un bloque de memoria principal se encuentre en memoria cache, debe encontrarse en el directorio el tag de identificación del bloque. Por lo tanto, antes de acceder a los datos de la memoria cache se debe verificar que el tag de la dirección a la que se quiere acceder (el bloque al que pertenece la dirección de memoria), coincida con uno de los tags almacenados en la memoria cache. Si la posición de memoria a la que se quiere acceder no se encuentra en la memoria cache (miss), se debe acceder a la memoria principal. Usualmente, el directorio de la memoria cache se implementa con alguna forma de memoria asociativa para poder conocer con rapidez si la dirección a la que se intenta acceder se encuentra en la memoria cache o no.

La longitud de los bloques se elige en potencias de 2. De esta forma se logra que, por un lado, el tag de identificación del bloque está formado por los bits más significativos de la dirección de memoria virtual. Por el otro, una palabra de memoria se accede por medio del

desplazamiento dentro del bloque al cual se quiere acceder. Por lo tanto, las direcciones tanto virtual como real se consideran como la concatenación de (bloque, desplazamiento dentro del bloque).

Manteniendo fijo el tamaño de la memoria cache, el tamaño de cada bloque determina la cantidad total de bloques, tanto de la memoria cache como de la memoria principal. Siempre se cumple la ecuación

$$Tam\_Cache = \#Bloques_C \times Tam\_Bloque \quad (3.3)$$

donde *Tam_Cache* es el tamaño total de memoria cache para datos (no incluye el directorio), $\#Bloques_C$ es la cantidad total de bloques de la memoria cache y *Tam_Bloque* es el tamaño de cada bloque.

En general, se fija el tamaño de los bloques y queda determinada la cantidad de bloques, aunque la decisión no es completamente independiente. Por un lado, se debe evitar que haya muy pocos bloques de memoria cache para evitar que un mismo programa con varias regiones de referencia produzca el reemplazo continuo de bloques de memoria cache. Por otro lado, tampoco es deseable tener demasiada cantidad de bloques, dado que el directorio de la cache se haría demasiado costoso. Manteniendo fija la cantidad de memoria cache, de la longitud de bloque dependen de forma directa o indirecta

1. El tráfico con la memoria principal.
2. La cantidad de tags de la memoria cache y, por lo tanto, el tamaño de la memoria asociativa.
3. La longitud de cada tag de bloque.

Es por esto que usualmente se dan los resultados de rendimiento fijando la longitud de la memoria cache y en función de la longitud de bloque.

3.3.3 Política de Búsqueda

La política de búsqueda de datos para transferir de la memoria principal hacia la memoria cache se utiliza para decidir cuándo poner información en la memoria cache. El método más utilizado ha sido "por demanda" (on demand): cuando un dato se debe acceder, se le asigna una ubicación en la memoria cache y se trae desde la memoria principal. Aquí el término "dato" equivale siempre a "bloque", que es la unidad de transferencia entre la memoria cache y la memoria principal, a menos que se especifique otra cosa.

La principal alternativa a la búsqueda de información por demanda, consiste en la realización de prebúsqueda (prefetching) [Smi82] [Hil84], es decir la asignación de un dato en la memoria cache antes que se necesite. Los algoritmos de prebúsqueda realizan estimaciones acerca de qué datos se necesitarán en un futuro cercano, y los asigna en la memoria cache para tenerlos disponibles inmediatamente en el momento en que se referencian.

Cuando la búsqueda de información se realiza por demanda, siempre que no se encuentre un dato en la memoria cache se accederá a la memoria principal. Si, por el contrario, se utiliza una política de prebúsqueda, se debería definir como mínimo:

- Qué cantidad de información buscar.

- Cuál información buscar.
- Cuándo buscar la información.

Tres valores usuales para estos parámetros, suelen ser:

- Cantidad de datos de prebúsqueda: uno o dos bloques.
- Información de prebúsqueda: El o los bloques inmediatamente siguientes al último referenciado.
- Tiempo de prebúsqueda: cada vez que se realiza una referencia desde el procesador o cada vez que hay un miss (el dato a buscar no se encuentra).

En términos generales, la prebúsqueda aumenta el tráfico de información entre la memoria cache y la memoria principal con el objetivo de reducir la probabilidad de no encontrar un dato en memoria cache. La prebúsqueda de información tiene varios problemas que se deben resolver, además de la implementación que se debe realizar en la propia memoria cache.

Un inconveniente bastante restrictivo consiste en que para realizar prebúsqueda de un dato se debe saber si el dato ya está en memoria cache. Para saber si un dato se encuentra o no en memoria cache se debe utilizar el directorio (la memoria asociativa). Esta utilización del directorio no debería interferir con las referencias que se realicen desde el procesador, ya que de este modo, el procesador debería esperar a que la memoria cache termine la fase de prebúsqueda para acceder a los datos que referencia (que incluso pueden no estar).

Otro de los inconvenientes cuando se realiza prebúsqueda de información es la mayor cantidad de bloques que suelen asignarse en la memoria cache. Asignar un bloque en memoria cache como consecuencia de la prebúsqueda puede ocasionar al menos dos inconvenientes:

- Asignar en memoria cache información que no se utilizará. No siempre se realiza la prebúsqueda de la información que se utilizará, porque no siempre se puede saber qué información se utilizará.
- Desplazar de la memoria información que se va a utilizar y que por causa de la prebúsqueda ya no se encuentra en la memoria cache.

En términos generales, se estudia mucho el entorno de procesamiento antes de decidir utilización de prebúsqueda. Se ha estudiado esta alternativa, y se ha encontrado que es útil bajo algunas circunstancias y contextos especiales [Hil84].

Una alternativa más que se considera cuando se puede tener en cuenta es la de búsqueda selectiva (selective fetching). Se considera de utilidad en los entornos de multiprocesadores donde se omite la búsqueda de algún tipo de información, tal como los semáforos [Smi82]. Tal información se denomina *unfetchable*, nunca se asigna en memoria cache. En particular, toda escritura se realiza en memoria principal y cada acceso puede considerarse como un miss porque no se encuentra el dato en memoria cache.

3.3.4 Modelos de Direccionamiento de Memoria Cache

Dado el contexto de utilización de memoria cache en sistemas de procesamiento con uso de memoria virtual, se plantea el interrogante de cómo acceder a memoria cache. Las alternativas son: con la dirección de memoria virtual o con la dirección de memoria real del dato al que se

hace referencia. En el primer caso, la memoria se denomina cache de dirección virtual, y en el segundo caso, cache de dirección real.

Acceder a la memoria cache con la dirección virtual tiene un gran atractivo, dado que es la dirección que sale del procesador. No se debe esperar ninguna traducción de memoria virtual a memoria real. Además, la búsqueda de la información en la memoria cache se puede realizar en paralelo con la traducción de la dirección de memoria virtual a la dirección de memoria real. Las direcciones que se mantienen en memoria cache (básicamente en el directorio) son las direcciones virtuales. La Fig. 3.4 esquematiza la forma básica en que se utiliza la memoria cache con direcciones virtuales.

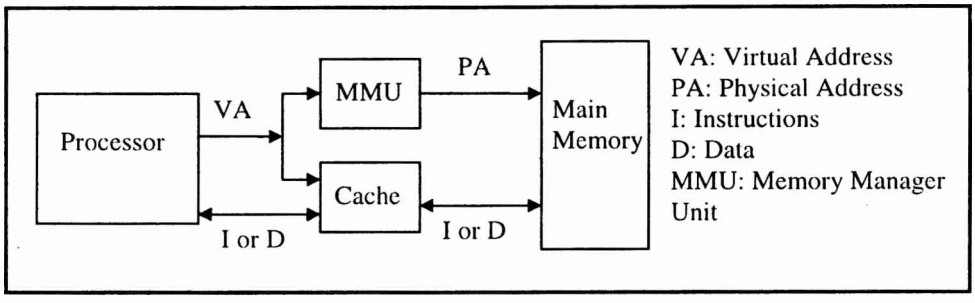


Figura 3.4: Memoria Cache de Direcciones Virtuales.

El problema principal que se encuentra en las memorias cache de direcciones virtuales es la aparición bastante frecuente de sinónimos (synonyms) [Wil91], también llamado problema de aliasing [Hwa93]. Suele suceder que dos procesos referencian la misma dirección real utilizando distintas direcciones virtuales. Para la memoria cache, distintas direcciones virtuales implican distintos datos, y por lo tanto la misma posición de memoria real podría estar en distintos bloques de memoria cache y ser considerada como información distinta. Se podría llegar a algo semejante a la falta de coherencia de cache tal como se explicó en la sección 3.2: un mismo dato puede tener más de un valor dependiendo del contexto (en este caso procesos). El problema de aliasing hace que usualmente se descarte la aplicación de memorias cache de direcciones virtuales. En algunos casos se ha utilizado la conversión de dirección de memoria virtual a dirección de memoria real (utilizando los denominados RTB: **R**everse **T**ranslation **B**uffers o ITB: **I**nverse **T**ranslation **B**uffers) para identificar todas las direcciones virtuales que se dan a la misma dirección real.

En el caso de acceder a la memoria cache con direcciones de memoria real, como es el caso más común, por lo menos una parte de la dirección de memoria virtual debe ser traducida a dirección de memoria real. Es por esta razón que hay cierto retraso en el acceso a memoria cache desde que la referencia es producida por el procesador hasta que la memoria cache es inspeccionada por el dato que se está buscando.

La Fig. 3.5 muestra esquemáticamente la forma en que se accede a la memoria cache de direcciones reales. La dirección de memoria virtual se traduce primero a dirección de memoria real en una unidad de manejo de memoria (MMU: **M**emory **M**anagement **U**nit), o TLB: **T**ranslation **L**ook-aside **B**uffer. Por causa de esta traducción, la dirección de memoria real llega a la memoria cache algunos ciclos de reloj posteriores a la generación de la referencia a memoria virtual por parte del procesador. Este retraso en la llegada de la dirección desde que se genera una referencia hasta que es procesada en la memoria cache es

una de las desventajas principales de la memoria cache de direcciones reales en comparación con la memoria cache de direcciones virtuales.

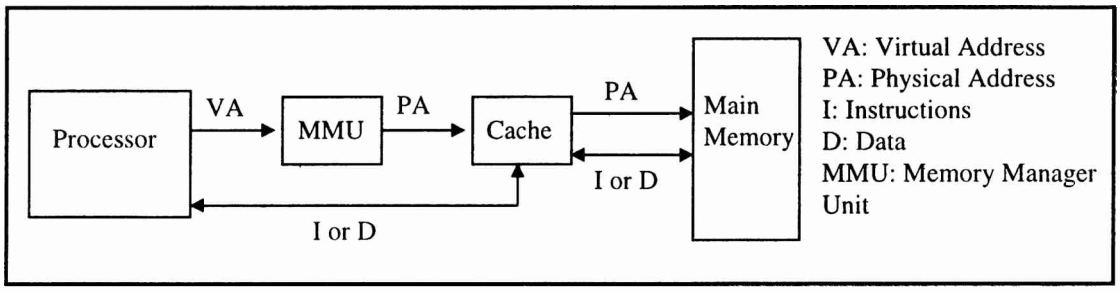


Figura 3.5: Memoria Cache de Direcciones Reales.

3.3.5 Método de Asignación de Bloques: Asociatividad

El método de asignación de bloques de memoria principal a bloques de memoria cache define en qué bloques de memoria cache puede ser asignado cada bloque de memoria principal. Este parámetro también se denomina *asociatividad* dado que *asocia* un bloque de memoria principal a uno o más bloques de memoria cache. En principio, se podría establecer que un bloque de memoria principal se pueda asignar a cualquier bloque de memoria cache. Este caso se denomina usualmente asociatividad completa (full associativity). La Fig. 3.6 muestra esquemáticamente esta forma de asignación de bloques de memoria principal a bloques de memoria cache.

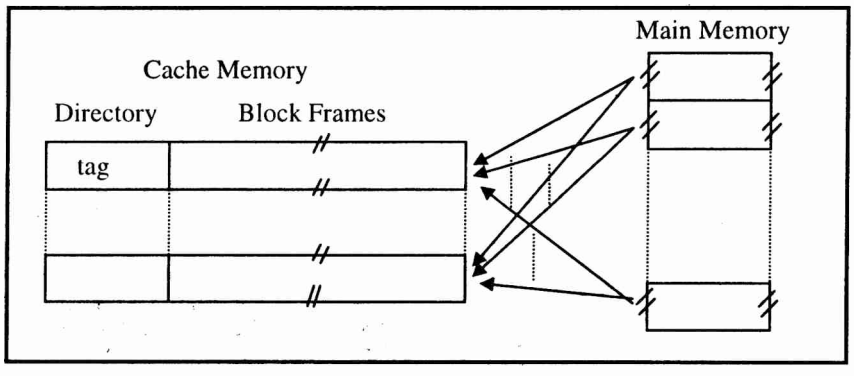


Figura 3.6: Memoria Cache con Asociatividad Completa.

En el método de asignación de asociatividad completa, la dirección que llega a la memoria cache se considera formada por el par: (tag, desplazamiento), tal como lo muestra la Fig. 3.7. El tag de la dirección que se debe buscar en la memoria cache se debe comparar con todos los tags asignados en el directorio de la memoria cache. Si estas comparaciones se realizan de manera secuencial, el retardo en conocer si una dirección de memoria está o no asignada en memoria cache puede ser demasiado grande, y es por eso que el directorio se implementa utilizando memoria asociativa, también denominada memoria direccionable por contenido o CAM: Content Addressable Memory. Con memoria asociativa, todas las comparaciones se realizan a la vez y por lo tanto el retardo es mínimo. Desde otro punto de vista, la Fig. 3.7 muestra la forma en que se ubica una dirección de memoria dentro de una

memoria cache en caso de que se encuentre (el tag de memoria referenciado se encuentra en el directorio de la memoria cache).

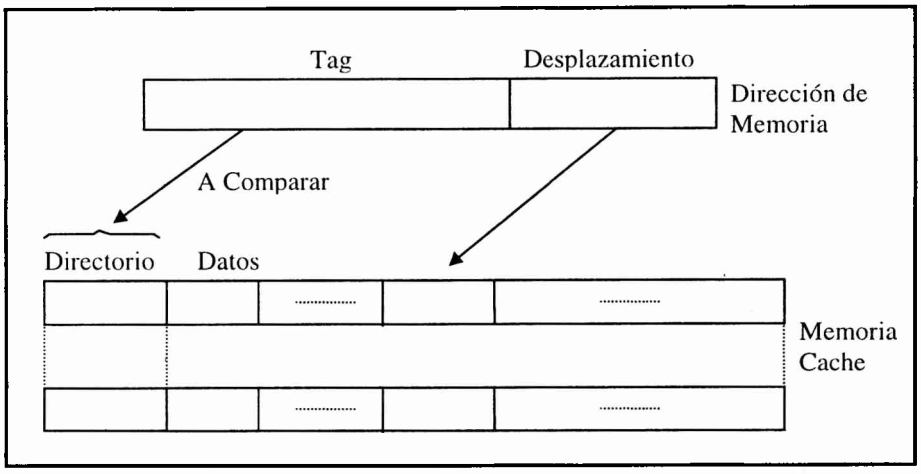


Figura 3.7: Direccionamiento de Memoria Cache con Asociatividad Completa.

El gran problema de este tipo de asignación es que a mayor cantidad de tags que se deban almacenar y cuanto mayor sea la longitud de cada tag, el costo crece de manera no proporcional, y además la memoria se hace más lenta en responder a los requerimientos. Aunque provee mucha flexibilidad a la hora de asignar un bloque en memoria cache, el método de asociatividad completa se descarta, en principio, por razones de costo y de eficiencia.

El método opuesto al de asociatividad completa se denomina asignación directa (direct-mapping). En este método de asignación, un bloque de memoria principal puede ser asignado en un sólo bloque de memoria cache. Tiene menos flexibilidad que el anterior, ya que no se tiene en cuenta qué bloques de memoria cache están libres o no.

La Fig. 3.8 muestra esquemáticamente el método de asignación directa de bloques de memoria principal en memoria cache.

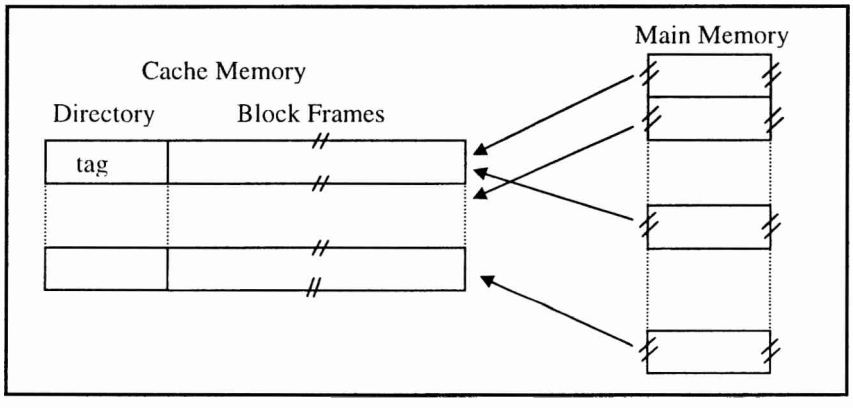


Figura 3.8: Memoria Cache con Asignación Directa.

En el método de asignación directa, la dirección que llega a la memoria cache se considera compuesta por: (tag, bloque, desplazamiento). Tanto el campo tag como el campo

desplazamiento tienen las mismas funciones que en el método de asociatividad completa. El campo bloque identifica el bloque de cache que se debe inspeccionar para conocer si el dato se encuentra en memoria cache o no, comparando los tags (el del directorio y el de la dirección de memoria).

Para definir el método de asignación directa se pueden numerar los bloques de memoria cache como $\overline{B_i}$ (i = 1, 2, ..., m), y los bloques de memoria principal de igual forma como B_j (j = 1, 2, ..., n). Se asume que $n >> m$, $n = 2^s$ y $m = 2^r$. El método de asignación ($\overline{B_i} \rightarrow B_j$) directo está basado en la función módulo m de la siguiente forma:

$$\overline{B_i} \rightarrow B_j \quad \text{si } i = j \text{ (módulo } m) \tag{3.4}$$

Si la cantidad de palabras de un bloque de la memoria principal está definida como 2^w, la dirección de memoria tiene en total $s+w$ bits. Para implementar la asignación directa

- el tag de la dirección está compuesto por los s-r bits más significativos
- los siguientes r bits más significativos indican el bloque de memoria cache al cual se asigna el bloque de memoria principal, y
- los w bits menos significativos indican el desplazamiento dentro del bloque.

La principal consecuencia de utilizar el método de asignación directa es que se hace necesaria solamente una comparación entre el tag de la dirección de memoria principal y un tag del directorio de memoria cache. Esto facilita la implementación del directorio de la memoria cache significativamente. La segunda consecuencia de utilizar este método de asignación es que, siguiendo las definiciones hechas previamente, ya no serían necesarios s, sino s-r bits para identificar un tag.

Como métodos intermedios de asignación se encuentran los de asociatividad por conjuntos (set associativity). Un bloque de memoria principal se puede asignar a cualquiera de un conjunto de bloques de memoria cache. Se dividen los bloques de memoria cache en 2^t conjuntos, con $k = 2^{r-t}$ bloques en cada conjunto. Por lo tanto, cada bloque de memoria tiene k bloques de memoria cache asociados en los cuales puede ser asignado. Por esta razón, a este método también se la llama k-way set associativity. Para este método, la dirección de entrada a la memoria cache se divide en (tag, conjunto, desplazamiento). Teniendo en cuenta que un bloque de memoria principal B_j tiene asociado un conjunto SB_j de bloques de memoria cache, para saber si B_j está asignado en memoria cache, el tag de B_j se debe comparar con todos los tags de los bloques de SB_j. Es por esta razón que el directorio de la memoria cache se puede implementar como un grupo de memorias asociativas pequeñas, cada una de ellas para un conjunto diferente. Según los estudios realizados en [Smi82] y [Hil84] no es necesario implementar memorias cache con asociatividad completa, dado que con 4-way u 8-way set associativity se puede obtener rendimiento similar con mucho menor costo.

Como método apropiado para las memorias cache que se incluyan dentro de los procesadores se propone el método de asignación por sub-bloques o sectores (sub-block placement, sector placement) [Hil84], [Hen90], [Hwa93]. En este caso un bloque se subdivide en sub-bloques, y la unidad de transferencia con la memoria principal es el sub-bloque o sector. Los tags siguen asociados a los bloques, pero en el directorio de la memoria cache se debe tener registrado si un sector del bloque se ha asignado a memoria cache o no. Al menos

se necesita un bit por sub-bloque para mantener esta información en memoria cache. Con este método de asignación, los pasos a seguir para llegar a un dato en memoria cache son:

- buscar el bloque en memoria cache. Se utiliza el tag del bloque de memoria principal y puede ser por asociatividad completa, asignación directa, o k-way set associativity (k = 2, 4, u 8). Si el tag del bloque se encuentra en memoria cache, entonces
- buscar el sub-bloque dentro del bloque en memoria cache. Si el sub-bloque se encuentra, entonces
- utilizar el desplazamiento dentro del sub-bloque para direccionar la palabra de memoria buscada.

Cualquiera de los primeros dos pasos que falle provoca un miss de memoria cache, y se debería acceder a memoria principal.

Con el método de asignación de sub-bloques, se logra reducir la cantidad de datos a los que se accede en memoria principal por cada miss de memoria cache y por lo tanto se reduce el tráfico en el bus. También como resultado de asociar cada tag a un bloque y no a un sub-bloque se obtiene una reducción en la cantidad de bits que se necesitan en el directorio de la cache para los tags [Hil84].

Los métodos de prebúsqueda de información suelen ser utilizados (o al menos tenidos en cuenta) cuando se utiliza el método de asignación de sub-bloques. Dado que se espera que en este caso el primer sub-bloque de cada bloque sea asignado por un miss de memoria cache, y uno o más sub-bloques sean asignados por prebúsqueda, sin esperar a que se produzcan más misses en memoria cache. Es por esta razón que en general, traer por adelantado un sub-bloque a memoria cache no implica desplazar otro bloque ya asignado. De todas maneras, se mantienen los problemas de sobrecarga de utilización de memoria cache y de accesos a la memoria principal. En las arquitecturas MIMD de memoria compartida se debe recordar que acceder a memoria principal puede llegar a producir interferencias con accesos de los demás procesadores.

3.3.6 Algoritmo de Reemplazo de Bloques

Cuando un bloque de memoria principal debe ser asignado en memoria cache, puede suceder que todos los bloques de memoria cache que sean posibles destinos ya estén en uso. Por esta razón se debe implementar un algoritmo de reemplazo de bloques como sucede con los sistemas de procesamiento que utilizan memoria virtual.

Como sucede cuando se analizan alternativas de reemplazo de páginas y/o segmentos de memoria virtual, los algoritmos más referenciados son LRU (**L**east **R**ecently **U**sed), FIFO (**F**irst **I**n, **F**irst **O**ut), y también Aleatorio o Random. La forma en que funcionan estos algoritmos es ampliamente conocida dada la extensiva utilización y referencia a estos métodos para implementar mecanismos de memoria virtual [Sta97] [Sil94] [Tan92] . Por esta razón no se hará la descripción de su funcionamiento, aunque sí se detallará lo que sucede en el contexto de la memoria cache, cuando se utilizan para reemplazo de bloques.

Usualmente los bloques de memoria cache que son posibles destinos de un bloque de memoria principal son muy pocos, por lo que se ha detallado en la sección de métodos de asignación de bloques. Por esta causa, a diferencia de lo que sucede en el contexto de

utilización de memoria virtual, el algoritmo FIFO y aún el método Aleatorio pueden ser utilizados con buenos resultados. Al tener menos posibilidad de elegir bloques a reemplazar por la asociatividad (usualmente hasta ocho bloques posibles), estos algoritmos dan resultados aceptables comparándolos con los demás. Por otro lado, tienen la ventaja de ser mucho más fáciles de implementar en hardware. También el algoritmo LRU se tiene normalmente en cuenta al menos cuando se realiza el análisis de distintas alternativas, aunque en general, las implementaciones son variaciones de este algoritmo.

Solamente se pueden tener en cuenta implementaciones por hardware de los algoritmos de reemplazo de bloques dada la velocidad a la cual se necesita que operen. Esta es otra de las diferencias teniendo en cuenta los entornos de memoria cache y memoria virtual con respecto a estos algoritmos.

En el caso de asignación directa de bloques de memoria principal en bloques de memoria cache no hay posibilidad de aplicar ningún método de reemplazo. El destino de un bloque de memoria principal es único y no hay posibilidad de elección. En el caso de 2-way set associativity (o incluso 4-way set associativity) suele utilizarse el método random porque es el más sencillo de implementar.

3.3.7 Actualización de Memoria Principal

El método de actualización de memoria principal define cuándo las escrituras (writes) realizadas por el procesador se reflejarán en la memoria principal. Hay dos formas básicas de realizar una escritura de un dato en memoria principal:

- Write-Back, Copy-Back o Copyback: todas las escrituras se realizan en memoria cache. Cuando un bloque se debe desalojar de memoria cache se lo copia en memoria principal para que los nuevos valores queden en memoria principal y no se pierdan al desalojar el bloque de memoria cache.
- Write-Through: todas las escrituras se realizan de manera inmediata en la memoria principal. Siempre el valor de una posición de memoria será el mismo en memoria cache y en memoria principal.

El método write-back reduce el tráfico de información entre la memoria principal y la memoria cache. Tiene en principio dos inconvenientes: (a) información de control en memoria cache, y (b) desactualización o inconsistencia de memoria principal.

Si se decide que cada bloque que se desaloja de memoria cache se copie en memoria principal (una forma de implementar el método write-back), probablemente se tendría demasiado tráfico de información. Puede suceder que se copien bloques que no se han cambiado en memoria cache. Por lo tanto, se debe tener registrado para cada (sub-)bloque si se debe copiar en memoria principal o no cuando se desaloje de memoria cache. Usualmente esta información se mantiene en un bit asociado al (sub-) bloque, que se denomina "dirty bit". Tanto el dirty bit como la lógica de actualización necesaria para mantener consistente la información contenida en él se agregan en el directorio de la memoria cache.

Si bien en los sistemas monoprocesador no es muy problemático que los valores en memoria principal no coincidan con los valores en memoria cache, en los sistemas

multiprocesadores esto genera inconsistencias. En este caso, se debe proveer un mecanismo de sincronización de memorias cache para mantener la consistencia (coherencia de memorias cache). La lógica suele ser complicada y se debe incluir siempre que más de un procesador acceda a la misma memoria principal.

Bajo el método de actualización write-back normalmente la escritura de un dato genera que éste sea asignado primero en la memoria cache, lo que usualmente se denomina fetch-on-write. Este esquema de búsqueda coincide con el detallado previamente de búsqueda de datos por demanda desde el procesador.

En el método de actualización write-through, toda escritura genera un acceso a memoria principal, es como si se tuviera un miss en memoria cache por cada escritura. De esta manera, el tráfico con la memoria principal se aumenta considerablemente, aunque se mantiene la consistencia de la información. Con este método de actualización se puede no hacer fetch-on-write sino realizar la escritura en la memoria principal sin asignar el dato en memoria cache, con lo cual no se estaría siempre cumpliendo la búsqueda de datos por demanda. No se puede evitar el acceso a memoria cache para verificar la existencia del dato, pero sí se pueden realizar los accesos (a memoria cache y a memoria principal) de manera simultánea.

En los dos casos se deben mantener buffers para copia de la información en memoria principal. En el caso de utilizar write-back, se necesitan buffers para que el ciclo de búsqueda de un dato no interfiera con la lectura de otro. La copia de un bloque siempre se realiza cuando se debe transferir un bloque desde memoria principal a memoria cache y se debe desalojar un bloque de memoria cache. En el caso de utilizar write-trough, son necesarios varios buffers para que la escritura a memoria principal no tenga como consecuencia que el procesador deba esperar siempre que la transferencia con la memoria principal se realice para cada escritura de un dato. Es este último caso también se debe controlar la información que está en los buffers para no leer de memoria un dato que aún no se ha escrito (se encuentra en los buffers de escritura).

3.3.8 Homogeneidad

La información que se asigna en la memoria cache puede ser clasificada de muchas maneras y de hecho es utilizada de distintas formas. Por esta razón es que se propone una memoria cache para cada tipo de información o, lo que es equivalente, dividir la memoria cache en partes dedicadas a distintas clases de información. Como el costo sigue siendo una restricción importante nunca se han propuesto más de dos clases de información y, por lo tanto, dos "tipos" de memoria cache. El funcionamiento básico de cada memoria cache es el mismo, aunque algunos parámetros, como el de tamaño de memoria cache o de bloque, pueden tener distintos valores.

La división más usual de memoria cache que se tiene en cuenta es la que destina una memoria cache para **datos** y otra para **instrucciones**. De esta manera se tiene una memoria cache "de datos" y una memoria cache "de instrucciones". Cada tipo de memoria cache es en sí misma una memoria cache completa, con sus propios parámetros de diseño, tales como el tamaño de la memoria. Este tipo de división de memoria cache tiene como consecuencia la

posibilidad de aumentar el ancho de banda con la memoria cache. Se puede tener acceso simultáneo a datos y a instrucciones, como en las arquitecturas de tipo Harvard. El primer inconveniente se encuentra cuando se debe dividir la información que originalmente reside en un lugar. Las instrucciones pueden ser modificadas, y esto debe realizarse antes de que sean ejecutadas. Más aún, datos e instrucciones pueden estar alojados en el mismo bloque de memoria principal, por la distribución de información que realicen los compiladores o por las instrucciones de carga inmediata de operandos.

El sistema operativo es utilizado por todos los procesos de una manera u otra, y este factor se tiene en cuenta para proponer que una memoria cache se dedique al sistema operativo. De esta manera, la memoria cache se divide en dos partes: una para el **sistema operativo** y otra para los **procesos de usuario** [Smi82]. Uno de los primeros inconvenientes para la implementación de esta división es similar al encontrado cuando se divide la memoria cache en memoria cache de datos y memoria cache de instrucciones. No toda la información es de uso exclusivo por una de los dos partes (sistema operativo o procesos). Quizás el mayor problema que se ha encontrado es que no existen muchos datos para evaluar esta alternativa, dada la dificultad de obtener información de lo que ocurre en modo supervisor.

Quizás la dificultad mayor al dividir la memoria cache es que de alguna manera se cambia la posibilidad de redistribución del working set de los programas. En el caso de memorias cache de datos y de instrucciones, no siempre la distribución del working set es la misma. En el caso de memorias cache para programas y para sistema operativo, las variaciones en el tamaño del working set dependen de la carga de trabajo y del tipo de trabajo. La cache unificada permite la redistribución automática de la información porque no pone más límite que el tamaño total de la memoria cache.

Para todas las divisiones de la memoria cache que se propongan, se debe determinar el tamaño de cada división, así como también otros parámetros como el tamaño de bloque, método de reemplazo de bloques, etc.

3.3.9 Niveles

Cuando se ha observado la necesidad de mayor tamaño de memoria cache, dado el crecimiento de costo y de tiempo de acceso, se ha propuesto la construcción de memoria cache de segundo nivel. La Fig. 3.9 muestra la forma en que se transfiere la información entre el procesador y la memoria principal.

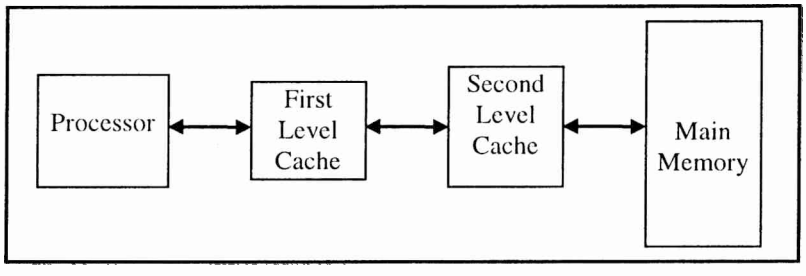


Figura 3.9: Memoria Cache Multinivel.

La memoria cache que se comunica con el procesador se denomina usualmente

memoria cache de nivel 1, y la memoria cache que se comunica con la memoria principal se denomina usualmente memoria cache de segundo nivel, o de nivel 2. Por razones de consistencia, se debe cumplir el principio de inclusión: toda la información que se encuentre en la memoria cache de nivel 1 debe encontrarse en la memoria cache de nivel 2.

La memoria cache de nivel 1 suele ser de tamaño muy reducido y se propone que se integre con el procesador (on-chip cache) [Hil84]. También se suele proponer que la actualización de memoria sea con el método write-through entre la memoria cache de nivel 1 y la memoria cache de nivel 2, y write-back entre la memoria cache de nivel 2 y la memoria principal.

3.4 Coherencia de Memorias Cache

En el contexto de multiprocesadores, como se ha indicado en la sección 3.2, el uso de memoria cache presenta el problema de coherencia (o consistencia) de memorias cache. El caso más simple que se puede encontrar consiste en un multiprocesador con dos procesadores, P1 y P2, que comparten una memoria común, y cada uno con una memoria cache. El procesador P1 lee una dirección de memoria y luego modifica su valor. El procesador P2 lee la misma posición de memoria y también modifica su valor. Aún cuando el método de actualización de memoria principal sea write-through, el procesador P1 no tendrá el último valor de la posición de memoria, que es el que asignó el procesador P2. Algunas soluciones se han implementado y otras se han propuesto.

La semántica de la memoria compartida se debe mantener, independientemente de la utilización de memoria cache. Esta semántica indica que cuando un proceso cambia el valor de una posición de memoria, las siguientes lecturas de esa posición de memoria deben retornar el nuevo valor [Tan90].

Las soluciones varían entre las implementadas en hardware, en las cuales la memoria cache continúa siendo transparente para el software, y las políticas que se deben implementar por software con la asistencia de hardware específico para esta tarea. En términos generales, se han propuesto tres soluciones:

1. Protocolos de hardware
2. Datos de escritura compartidos no asignables en cache
3. Esquemas basados en software.

Tanto para las soluciones dentro de la clase (2) como para las de la clase (3) se requiere el soporte de hardware, pero los programadores deben tener en mente el problema de consistencia de memorias cache.

En la primera clase, se incluyen todas las soluciones al problema de coherencia de memoria cache que permiten que todos los datos sean asignados en la memoria cache de cualquier procesador y el hardware del sistema se encarga de mantener la coherencia. El principal inconveniente es la complejidad del hardware. La complejidad suele crecer de forma no lineal cuando se aumenta la cantidad de procesadores.

El problema de coherencia de memorias cache se produce por los datos compartidos que pueden ser escritos por distintos procesadores. Por esta razón, una solución es mantener

estos datos *siempre* en memoria principal. Se define, por lo tanto, que algunos datos no pueden ser asignados en memoria cache (non-cacheable data). Siguiendo las directivas del programador, el compilador asigna los datos compartidos que se pueden escribir en regiones de memoria principal desde las cuales no se pueden transferir datos a memoria cache. El principal problema de esta solución consiste en la cantidad de memoria principal que se designa con estas características. La cantidad de datos que se comparten y que se pueden escribir puede ser muy dependiente del problema.

La tercera solución, los esquemas basados en software, propone que los datos compartidos de escritura sean asignados en memoria cache solamente cuando se puede asegurar que no se producirá inconsistencia. Cuando se prevé que pueden producirse inconsistencias, se realiza la escritura de los datos en memoria principal (cache flushing). La ejecución se divide en intervalos en los cuales es seguro asignar los datos en cache, y al término de cada uno de esos intervalos la memoria principal se actualiza con los datos que han cambiado.

3.4.1 Protocolos de Hardware

La base para conservar la coherencia de cache por hardware consiste en la comunicación a todas las memorias cache (broadcast) de cada escritura. Si la dirección destino de la escritura se encuentra asignada en otra memoria cache, puede ser actualizada o invalidada.

La política denominada write-update es la que define que para cada escritura en una memoria cache se actualice el dato en las demás memorias cache en las que se encuentre. Se debe tener en cuenta la situación en la que dos procesadores asignan la misma posición de memoria y se producen las dos actualizaciones a la vez. En este caso se puede llegar a tener dos valores para la misma posición de memoria y se pierde también la coherencia de las memorias cache.

La política denominada write-invalidate es la que define que para cada escritura en una memoria cache se invalide el dato en las demás memorias cache en las que se encuentre. Se debe recordar que para invalidar un dato se invalida el bloque de cache al cual pertenece. A partir del momento de la invalidación, si se referencia un dato perteneciente al bloque invalidado, se debe volver a asignar en memoria cache.

La invalidación del dato en memoria cache necesita menor transferencia de información que la actualización. Cualquiera sea el método que se elija, se debe resolver desde dónde leer un dato cuando no existe en memoria cache. Un dato puede haber sido escrito en memoria cache y aún no haber llegado la actualización a memoria principal. Esta decisión también depende del método de actualización de la memoria principal: write-back o write-through.

Las políticas write-update y write-invalidate requieren los comandos de actualización de memoria cache y de invalidación de memoria cache, que se denominan colectivamente comandos de consistencia (consistency commands). Los comandos de consistencia deben ser enviados al menos a las memorias cache que posean una copia del bloque de memoria. En las redes de conexión basadas en buses de comunicación, es posible realizar un broadcast para

cada comando de consistencia. En las redes de comunicación en las que el costo de un broadcast es demasiado alto, se realiza un multicast de los comandos de consistencia únicamente a las memorias cache que poseen una copia del bloque de memoria.

En el caso de realizar broadcast de los comandos de consistencia, cada memoria cache debe procesar todos los comandos de consistencia que le lleguen para descubrir si se refiere a sus datos. Estos protocolos se denominan *snoopy cache protocols*, porque cada memoria cache debe estar continuamente observando los comandos de consistencia que circulan por la red [Tan90] [Sto93].

En el caso de realizar multicast, se debe conocer qué memorias cache tienen una copia del dato. Esta información se debe mantener en directorios, y a estos protocolos se los denomina esquemas de directorio (directory schemes) [Maa91] [Sto93].

Una última alternativa que se puede considerar para mantener la consistencia de memoria cache por métodos de hardware consiste en que todos los procesadores accedan a una única memoria cache. Si todos los procesadores comparten la misma memoria cache, no hay problema de coherencia. El dato al que se accede en memoria cache es físicamente el mismo, en la misma memoria cache. El problema que genera este método para conservar la coherencia es el mismo que se detalló en la sección 3.2 en cuanto al acceso a memoria de múltiples procesadores. Los requerimientos a memoria cache en este caso se resuelven secuencialmente y eso tiene como consecuencia que algunos procesadores deban esperar para acceder a la información. La utilización de este método es muy limitada y la cantidad de procesadores que comparten la misma memoria cache no debería ser mayor que dos. Aún así, se ha implementado por ejemplo en la computadora UNIVAC 1100/80 [Bor79], donde dos procesadores comparten una memoria cache.

La Fig. 3.10 muestra la clasificación que se podría hacer con los métodos para conservar la coherencia de memoria cache por hardware. Los protocolos snoopy y los esquemas de directorio pueden ser considerados las formas de implementar las políticas de write-invalidate y write-update.

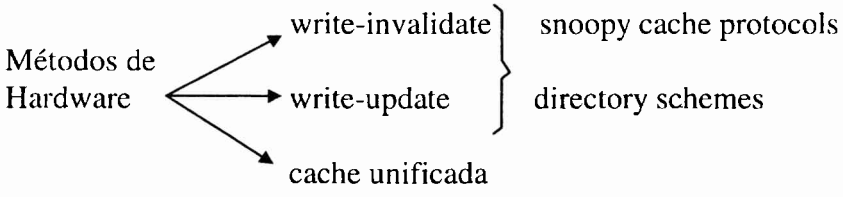


Figura 3.10: Métodos de Hardware para Consistencia de Memoria Cache.

3.5 Indices de Rendimiento de la Memoria Cache

El propósito de la memoria cache es mejorar el rendimiento del subsistema de memoria reduciendo el tiempo efectivo de acceso a memoria y el requerimiento de ancho de banda entre el procesador y la memoria principal. El tiempo efectivo de acceso a memoria es definido de forma simplificada como [Hil84]

$$t_e = t_c\ (1 - m) + t_m\ m \tag{3.5}$$

donde t_e es el tiempo efectivo de acceso a memoria, t_c es el tiempo de acceso a la memoria cache, m (miss ratio) es la probabilidad de no encontrar un dato en memoria cache, y t_m es el tiempo de acceso a memoria principal. Se debe recordar que

$$m = 1-h \tag{3.6}$$

si h (hit ratio) es la probabilidad de acierto, la probabilidad de encontrar un dato en memoria cache.

Los tiempos de acceso a memoria cache (t_c) y de acceso a memoria principal (t_m) no son fáciles de obtener. Dependen de la tecnología, complejidad y organización de cada una de las memorias, y por lo tanto dependen de la implementación. En las experimentaciones con memorias cache, suele utilizarse como índice de rendimiento la probabilidad de acierto (h) o la probabilidad de no acierto (m). En realidad se asume y se asumirá en lo que sigue que la probabilidad de encontrar (no encontrar) un dato en memoria cache es igual a la relación entre la cantidad de veces que se han encontrado (no encontrado) los datos que se buscaron con respecto a la cantidad total de veces que se han buscado datos. Es decir

$$h = \#aciertos / \#accesos \tag{3.7}$$

donde *#accesos* es la cantidad de datos que se han requerido desde el procesador, y *#aciertos* es la cantidad de datos que se han requerido desde el procesador y se han encontrado en memoria cache. De forma similar,

$$m = \#no\_aciertos / \#accesos \tag{3.8}$$

La relación entre m y h sigue siendo la que se dio en la Ec. (3.6). Tanto m como h son independientes de la implementación y por lo tanto son los más utilizados en la experimentación con memorias cache.

Otro de los índices de rendimiento más utilizados suele ser la relación de tráfico de memoria principal con memoria cache con respecto al tráfico sin memoria cache [Hil84]. Dos factores afectan el tráfico con la memoria principal. En primer lugar, las referencias a posiciones de memoria que se hallan en la memoria cache reducen el tráfico con la memoria principal. En segundo lugar a medida que se aumenta el tamaño de un (sub-)bloque, se tiende a incrementar el tráfico, porque cada acceso a memoria principal implica la transferencia de un bloque, aunque originariamente se necesite transferir una posición de memoria. En realidad, el tráfico que se incrementa de forma directa es el que se produce cuando se referencia una posición de memoria que no está en memoria cache (miss). El tráfico que se produce en un miss tiene dos aspectos:

- Se debe transferir un (sub-)bloque completo desde memoria principal a memoria cache, sin importar la cantidad de datos del (sub-)bloque que efectivamente se utilizará.
- Si se desaloja de memoria cache un (sub-)bloque que ha sido escrito y el método de actualización de memoria principal es write-back, se debe transferir el (sub-)bloque desde memoria cache a memoria principal.

Por lo tanto, lo que seguramente se incrementará es el tiempo que lleva resolver una falta del dato que se referencia en memoria cache. No se puede afirmar categóricamente que la

transferencia total con la memoria se incrementa, porque todos los datos que se asignan en memoria cache pueden llegar a utilizarse. De esta manera, el costo de transferir mayor cantidad de datos puede ser útil para reducir la cantidad de veces que los datos no se encuentran en memoria cache.

Los estudios y experimentos que se han realizado sobre aspectos y diseño de memorias cache se han basado principalmente en la simulación dirigida por trazas (trace-driven simulation). Una de las principales dificultades que se encuentran para definir el comportamiento de una memoria cache y para asignar valores a los parámetros de diseño consiste en que su efectividad depende de la carga de trabajo que se tenga. Así como no hay formas muy aceptadas de caracterizar la carga de trabajo, tampoco hay formas de caracterizar el efecto del comportamiento de los programas sobre el funcionamiento de la memoria cache [Smi82].

La simulación de memoria cache dirigida por trazas es uno de los métodos más efectivos (y sin dudas el más aceptado), en la evaluación del comportamiento de la memoria cache. Una traza es una secuencia de referencias a memoria y usualmente se obtiene ejecutando de manera interpretada un programa y almacenando todas las referencias a memoria que realiza. Cada referencia a memoria puede ser asociada a un tipo de acceso como por ejemplo: referencia a dato, referencia a instrucción, escritura. Se pueden utilizar una o más trazas para simular el comportamiento de la memoria cache. Las trazas, además, permiten la simulación de la misma ejecución bajo distintas alternativas de diseño de memoria cache, lo cual a su vez facilita la evaluación y la comparación de alternativas.

Dado el crecimiento de tamaño de las memorias cache, se debe tener una cantidad de referencias notable para que los resultados sean confiables. En este sentido, las simulaciones dirigidas por trazas tienen al menos tres inconvenientes: (1) la cantidad de referencias que se deben almacenar, (2) el tiempo durante el cual se deben recoger las trazas, es decir el tiempo en que se ejecutan los programas de forma interpretada para recoger las referencias a memoria y (3) el tiempo de simulación de la memoria cache. Para crear las trazas, los programas se ejecutan entre 10000 y 1000000 de veces más lentamente [Sto93] y esto hace que sea muy difícil crear trazas que representen períodos (suficientemente) largos de tiempo real de ejecución. A esto se debe sumar el período de inicialización del sistema, para recoger datos que no estén afectados por efectos secundarios.

4. Buses de Interconexión en Multiprocesadores

En este capítulo se considerará la utilización de buses de interconexión para conectar los procesadores con la memoria compartida de un multiprocesador (arquitectura MIMD con memoria compartida). A medida que crece la cantidad de procesadores y los módulos de memoria crece la necesidad de interconexión entre ellos. Además, junto con la necesidad de interconexión crece la importancia de los buses en lo referente a la eficiencia de los multiprocesadores.

En primer término se tendrá en cuenta la utilización de un único bus que los procesadores deben compartir de la misma manera en que comparten la memoria. Se verán algunos aspectos de manejo de la asignación del bus así como también algunos aspectos relacionados con el rendimiento que se puede obtener de la máquina paralela.

En segundo término se comentará la utilización de más de un bus de comunicaciones (múltiples buses) para conectar a los procesadores con la memoria compartida. También en el contexto de los buses múltiples se verán algunos aspectos de manejo de asignación de los buses y de rendimiento. En el capítulo siguiente, se verá el método de interconexión por cross-bar como si fuera un sistema de múltiples buses. Casi todo el análisis y la descripción de estos temas pueden encontrarse en [Wil91].

4.1 Multiprocesadores con un Unico Bus

En los sistemas basados en microprocesadores, es común que se utilice un único bus de comunicaciones. Este bus conecta el microprocesador con el o los módulos de memoria y con las unidades de entrada/salida. Este método de interconexión también puede aplicarse en un sistema multiprocesador como se muestra en la Fig. 4.1, para conectar todos los procesadores con la memoria compartida a la cual tienen acceso. En cualquier instante de tiempo, se lleva a cabo una única transferencia de información sobre el bus.

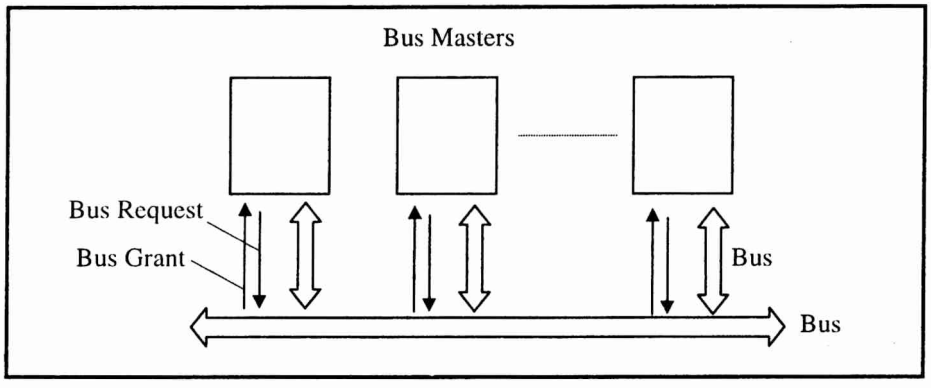


Figura 4.1: Bus Compartido.

Cada procesador o unidad que puede requerir la utilización del bus de comunicación se denomina Bus Master. Los procesadores deben requerir el uso del bus al mecanismo de asignación del bus (bus arbitration), y acceden solamente cuando están habilitados por este

mecanismo para hacerlo. Las dos líneas de control principales que se utilizan para la petición y asignación del bus se denominan Bus Request (Señal de Requerimiento del Bus) y Bus Grant (Señal de Asignación del Bus) respectivamente. Todos los procesadores deben competir por la utilización del bus de comunicaciones. En términos de tiempo, el bus provee el mismo tiempo de comunicación para todos los bus masters (procesadores) con la memoria. Por otro lado, se debe recordar que la velocidad de transferencia de información entre el procesador (o la memoria cache asociada al procesador) y la memoria principal no superará la velocidad de transferencia del bus.

Dado que más de un procesador puede requerir la utilización del bus de comunicaciones, se debe implementar (en hardware) un mecanismo de acceso. Este mecanismo, independientemente de la forma en que asigne el bus (por ejemplo, con prioridades o no) y de la forma en que se implemente, debe satisfacer al menos dos requisitos elementales:

1. A lo sumo un bus master puede utilizar el bus en un instante de tiempo. Se deben tener en cuenta dos casos: (1) si se producen peticiones simultáneas los accesos al bus se deben secuencializar, y (2) si el bus ya está asignado a un procesador P1 y otro procesador P2 requiere el acceso, se debe esperar a que P1 libere el bus para que P2 pueda utilizarlo.
2. Si el bus está libre, se debe dejar acceder al procesador que lo requiera. No se debe permitir que el bus esté libre y un procesador esté esperando para utilizarlo.

En términos de rendimiento, para que un único bus de comunicaciones mejore la velocidad de procesamiento de un sistema monoprocesador, se debe cumplir que cada procesador con acceso al bus tenga intervalos de tiempo en los que no lo utilice. Si todos los procesadores requieren la utilización del bus todo el tiempo, no se podrá mejorar la velocidad de ejecución porque solamente un procesador podrá llevar a cabo la ejecución, y todos los demás deberán esperar para acceder al bus de comunicaciones. En este caso, la ejecución se ha secuencializado porque habrá solamente un procesador ejecutando en cada instante de tiempo.

La mayoría de los procesadores poseen intervalos de tiempo en los que no requieren el bus, pero los procesadores sin más memoria que la memoria compartida (sin memoria local ni memoria cache) suelen requerir el bus entre el 50% y el 80% del tiempo de ejecución. Si cada procesador del sistema multiprocesador requiere el bus $1/n$ del tiempo, entonces se puede incrementar la velocidad de ejecución del sistema a lo sumo n veces. Lamentablemente, el tiempo de acceso al bus no depende exclusivamente de los procesadores sino que depende también de las aplicaciones, y por lo tanto se deben tener en cuenta a la hora de evaluar el rendimiento.

Como paso inicial se describen las formas de arbitrar el bus, el manejo de la asignación del bus de comunicaciones en un sistema multiprocesador. Como se afirmó anteriormente, el procesador, o cualquier unidad que pueda controlar el bus, se denominará bus master. El procesador que controla el bus en un instante se denominará bus master actual (current bus master). En cualquier instante de tiempo, podrá haber a lo sumo un bus master actual. El método de asignación de bus normalmente se implementa en hardware. Por esta razón la descripción de las alternativas se realiza en términos de las líneas de control del bus de comunicaciones.

4.1.1 Esquemas de Asignación del Bus

En general, se puede afirmar que la competencia que se establece entre los procesadores de un multiprocesador por el acceso a la memoria compartida, se traslada a la competencia por el acceso a la utilización del bus de comunicaciones. Tal como se mostró en la Fig. 4.1, las líneas de control más importantes para llevar a cabo la asignación del bus a los bus masters son dos: bus request y bus grant. Normalmente existen otras líneas de control asociadas, y los nombres de las líneas de control pueden variar según la implementación particular.

Cuando un bus master necesita utilizar el bus, lo indica por medio de la línea bus request. De esta manera, el mecanismo de asignación del bus conoce que existe un requerimiento pendiente para la utilización del bus. Cuando se ha decidido que el procesador puede acceder al bus, recibirá la notificación a través de la señal bus grant. Dado que el bus master actual puede no haber terminado la transferencia sobre el bus cuando el mecanismo de asignación ya decidió satisfacer el requerimiento de otro bus master, se agrega otra línea de control, denominada usualmente Bus Busy (Señal de "Bus Ocupado"). Las tres líneas de control se muestran en la Fig. 4.2.

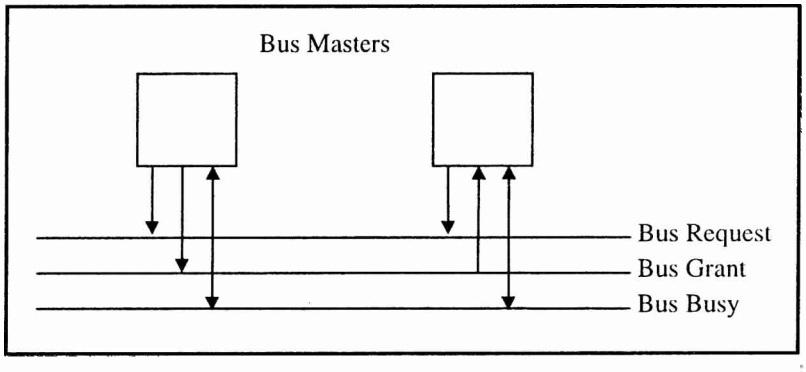


Figura 4.2: Señales de Control de Bus.

De acuerdo a las señales de control que se muestran en la Fig. 4.2, el current bus master mantiene la línea bus busy activa mientras realiza transferencias utilizando el bus. Solamente desactiva la señal bus busy cuando recibe la señal bus grant no activa desde el mecanismo de asignación del bus. En términos de líneas de control, el current bus master es el que mantiene la línea bus busy activa. El procesador que recibe la señal bus grant activa será el current bus master en la próxima transferencia.

Cuando el bus master actual recibe la señal bus grant no activa, debe dejar de utilizar el bus, solamente completa la transferencia actual y no importa que necesite realizar más transferencias o no. Cuando el bus master actual deja libre el bus, desactiva la señal bus busy, y los demás bus masters pueden reconocer que el bus está libre. Una vez que un bus master ha requerido el bus de comunicaciones y recibe la activación de la señal bus grant, no accede al bus inmediatamente, sino que espera hasta que la señal bus busy esté no activa. Si el bus está libre (señal bus busy no activa), y un bus master realiza una petición, entonces la activación de la señal bus grant equivale al acceso inmediato del bus master al bus. Cuando el bus master actual no necesita utilizar más el bus, lo libera desactivando las líneas bus request y bus busy cuando termina de realizar la última transferencia sobre el bus. La línea bus request puede

haber sido desactivada una vez que ha activado la línea bus busy, que es la que efectivamente indica la utilización del bus.

Cuando se debe decidir la asignación del bus entre múltiples procesadores que lo requieren, las posibilidades a evaluar son varias. En general, siempre hay un concepto de prioridad subyacente en la forma en que se asigna el bus a uno de los bus masters que lo ha requerido. La forma en que se asignan las prioridades puede ser estática o dinámica. Si las prioridades son estáticas (también llamadas fijas), un bus master siempre tiene la misma relación de precedencia con respecto a los demás para el acceso al bus. Por el contrario, si las prioridades son dinámicas las relaciones de precedencia se cambian a medida que transcurre el tiempo siguiendo algún método específico. Las prioridades asignadas dinámicamente tienen como objetivo que todos los procesadores tengan acceso equitativo al bus, especialmente en los sistemas en los que ningún procesador debería acceder al bus con más probabilidad que los demás. Los algoritmos más utilizados para establecer prioridades dinámicamente son:

- Rotación de Prioridades Simple: en cada ciclo de asignación del bus, todos los niveles de prioridad se reducen en una posición, y el de menor prioridad pasa a tener la prioridad más alta.
- Rotación de Prioridades Dependiente de la Aceptación (Rotación de Prioridades): el procesador al cual se le asigna el bus es el que recibe la menor prioridad, y los demás incrementan sus prioridades de forma lineal.
- Prioridades Aleatorias: en cada ciclo de asignación del bus los niveles de prioridad se asignan aleatoriamente (pseudo-aleatoriamente).
- Prioridades Iguales: todos los requerimientos que se realizan tienen la misma probabilidad de ser aceptados.
- Prioridades LRU (**L**east **R**ecently **U**sed): la mayor prioridad es asignada al bus master que no ha utilizado el bus por mayor período de tiempo.

De acuerdo a la forma en que se transmiten las señales de control de asignación del bus (bus request y bus grant), los esquemas de asignación del bus pueden ser:

1. Esquemas de Asignación Paralelos
2. Esquema de Asignación Serie.

En los esquemas de asignación paralelos, las señales de requerimiento del bus (bus request), entran a la lógica de asignación del bus por separado, y las señales de asignación del bus (bus grant) se generan por separado. Siempre se puede reconocer el bus master que genera un requerimiento y siempre se genera la señal bus grant para un determinado bus master que pasará a ser el bus master actual.

En los esquemas de asignación serie, una de las señales de control de asignación del bus (bus request o bus grant) se transmite de un master a otro para decidir a cuál de los bus masters se asigna el bus si alguno lo ha requerido. Este esquema también suele denominarse esquema *daisy chain*.

De acuerdo a la forma (*lugar*) en que se decide la asignación del bus, los esquemas de asignación de bus pueden ser

1. Esquemas de Asignación Centralizados
2. Esquemas de Asignación No Centralizados (Decentralized)

En los esquemas centralizados, las señales de requerimiento del bus (bus requests) directa o indirectamente se centralizan (llegan a un único lugar) para resolver la asignación. Desde el punto donde se centralizan las señales de requerimiento se genera la señal bus grant de asignación del bus.

En los esquemas no centralizados, las señales de requerimiento del bus no se centralizan para resolver el procesador al cual se asignará el bus. La decisión de asignación se puede hacer en varios lugares a la vez, normalmente en los mismos procesadores.

Aunque son los más utilizados, los esquemas centralizados tienen la desventaja de no ser muy tolerantes a las fallas (normalmente tienen un punto simple de falla), mientras que los esquemas no centralizados suelen ser más tolerantes a las fallas. El punto simple de falla se da en la lógica de asignación, donde se centralizan las señales de bus requests provenientes de los procesadores. Los esquemas serie y paralelos pueden tener formas centralizadas o no centralizadas.

La Fig. 4.3 muestra el esquema de asignación paralelo centralizado, también denominado esquema de prioridades paralelo centralizado. Cada bus master genera una señal de bus request que entra a la lógica de asignación del bus separada de las demás señales de requerimiento del bus. Desde la lógica de asignación se genera una señal de bus grant hacia el bus master al que se le asigna el bus. Tanto las señales bus request como la lógica de asignación del bus se centralizan donde se realiza el *Arbitraje*. Las señales bus request, bus grant y bus busy llegan a un mismo punto donde se realiza la asignación del bus de comunicaciones. En la lógica de asignación de deben tomar todas las decisiones con respecto a la asignación del bus cuando se tiene más de un requerimiento pendiente desde los procesadores. La complejidad de esta lógica de asignación se incrementa de manera considerable en función del crecimiento de la cantidad de procesadores y de las características propias de la asignación del bus.

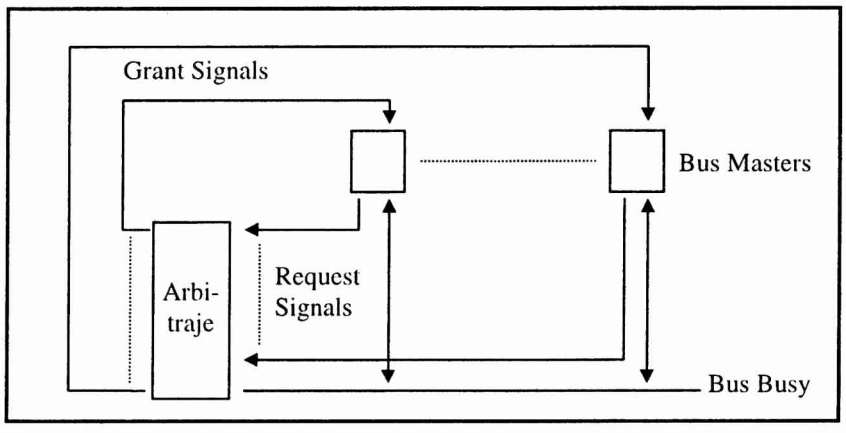


Figura 4.3: Esquema de Asignación Paralelo Centralizado.

De acuerdo al esquema de la Fig. 4.3, cada bus master genera una señal de bus request que entra a la lógica de asignación del bus separada de las demás señales de requerimiento del bus. Desde la lógica de asignación se genera una señal de bus grant hacia el bus master al que se le asigna el bus. Tanto las señales bus request como la lógica de asignación del bus se

centralizan donde se realiza el *Arbitraje*. Las señales bus request, bus grant y bus busy llegan a un mismo punto donde se realiza la asignación del bus de comunicaciones.

Como se ha definido previamente, la señal bus busy es mantenida activa por el bus master actual mientras utiliza el bus, y un bus master que ha requerido la utilización del bus pasa a ser el bus master actual cuando recibe la señal bus grant activa y la señal bus busy no está activa. Cuando la lógica de asignación del bus identifica que un bus master de mayor prioridad que el bus master actual ha requerido el bus, desactiva la señal bus grant del bus master actual y activa la señal del bus master con mayor prioridad. El bus master que ha recibido la activación de la señal bus grant espera hasta que se desactive la señal bus busy y comienza a utilizar el bus. La señal bus busy es necesaria porque el bus master actual no necesariamente puede liberar el bus de forma inmediata cuando recibe la señal bus grant no activa.

El esquema de asignación paralelo centralizado es quizás el más flexible en cuanto a la implementación de prioridades, pero también es el más costoso. Como mínimo, se deben mantener todas las líneas separadas, tanto las señales bus request como las señales bus grant. El costo de la lógica de asignación depende de la complejidad en lo que se refiere al manejo de las prioridades. La complejidad de la lógica de asignación suele crecer no linealmente con respecto a la cantidad de bus masters.

La Fig. 4.4 muestra el esquema de asignación paralelo no centralizado, donde la lógica de asignación del bus (arbitraje) se asocia a cada procesador. La lógica de asignación del bus asociada a cada procesador es la que produce la señal de bus grant que le corresponde.

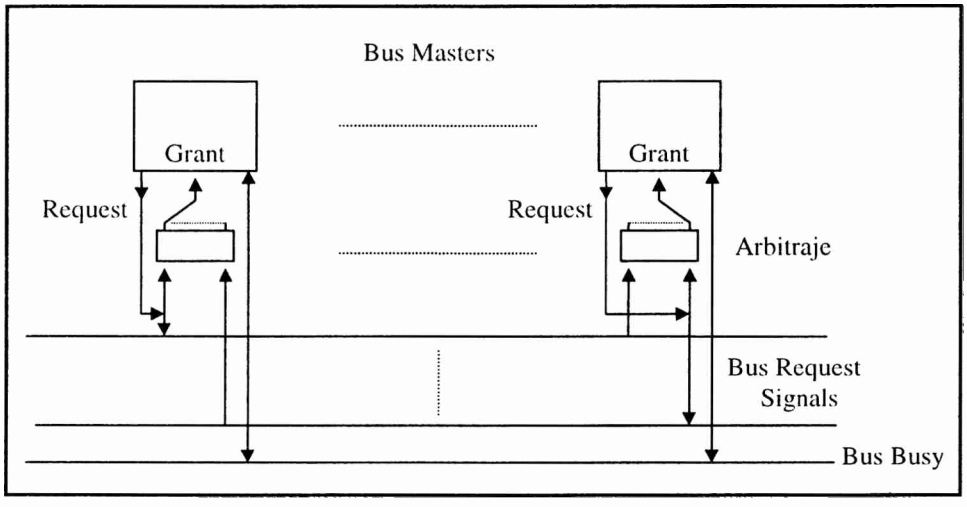


Figura 4.4: Esquema de Asignación Paralelo No Centralizado.

Tal como lo muestra el esquema de la Fig. 4.4, no hay un único lugar en donde se centralicen las señales y se decida la asignación del bus. Se necesita que todas las señales de bus request lleguen a cada procesador, pero las señales de bus grant se asocian localmente a cada procesador. La lógica de asignación será la misma para todos los procesadores, pero las entradas y las salidas serán las apropiadas para cada uno de ellos. Cada procesador utilizará la señal de entrada para bus request y la señal de salida bus grant que le corresponde.

En lo referente a la tolerancia a fallas, el esquema de asignación paralelo no centralizado es potencialmente más confiable que el centralizado. La falla en uno de los lugares donde se implementa la lógica de asignación (puntos de arbitraje en la Fig. 4.4), debería afectar solamente al procesador asociado y no a todo el sistema multiprocesador. Aún así, algunas fallas podrían afectar a todo el sistema. Una ventaja de este esquema con respecto al anterior es que requiere menos señales, las señales bus grant son locales a cada procesador.

La Fig. 4.5 muestra el esquema de asignación del bus centralizado serie con la señal correspondiente al bus grant que se transmite de un bus master al siguiente (daisy chained grant scheme). Suele ser el más utilizado de los esquemas de asignación serie. Los requerimientos de los bus masters se transmiten por una línea común (wired-OR) de requerimientos. La línea bus busy es también común y, como se definió anteriormente, permanece activa mientras un bus master esté utilizando el bus. Como se puede notar en la figura, la lógica de asignación es bastante sencilla. Por ejemplo, el bus controller puede no estar y de esta manera, el crecimiento en la cantidad de procesadores a asignar no aumenta la complejidad de la lógica de asignación del bus. Cada procesador tiene las mismas líneas para requerir, utilizar y liberar el bus. Además, la generación de las señales es la misma para cualquier cantidad de procesadores que utilicen el mismo bus de comunicaciones. De esta manera, agregar procesadores solamente implica agregar sus propias líneas de control.

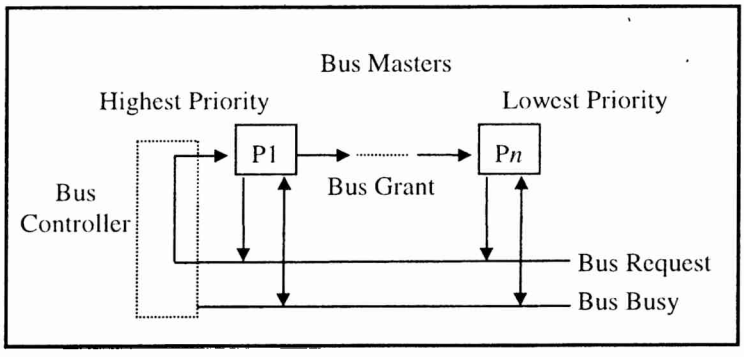


Figura 4.5: Esquema Centralizado Serie. Daisy-Chained Grant Signal.

Cuando uno o más bus masters activan la señal bus request, esta señal es transportada al bus master con mayor prioridad. El bus master con mayor prioridad es el que se ubica al principio de la cadena de bus grant (procesador P1 en la Fig. 4.5). En algunos casos se utiliza un controlador dedicado a esta tarea (Bus Controller) y en otros casos la señal se conecta directamente al bus master. La señal se transmite desde un bus master al siguiente por la cadena de bus grant hasta llegar al bus master de mayor prioridad que realizó un requerimiento del bus. Este bus master no continúa con la transmisión de la señal bus grant y de esta manera será el próximo bus master con acceso al bus. El bus master actual, si es de menor prioridad, no recibirá la señal bus grant y por lo tanto liberará el bus cuando termine la transmisión en curso.

La Fig. 4.6 muestra el esquema de asignación del bus centralizado serie con la señal correspondiente al bus request que se transmite de un bus master al siguiente (daisy chained request scheme). Nuevamente, la cadena de señales comienza en el bus master con mayor prioridad (procesador P1) y finaliza en el bus master de menor prioridad (procesador P*n*). La

señal que se transmite a lo largo de la cadena es la que corresponde a la línea de control bus request.

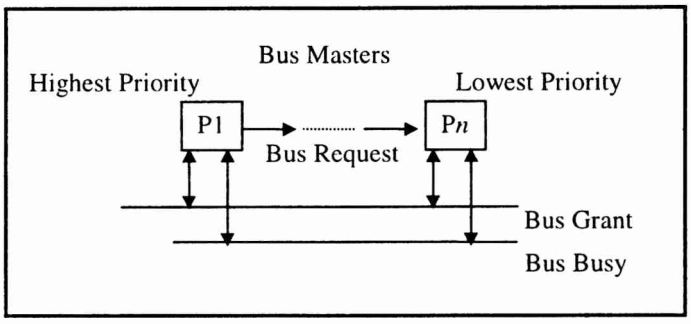


Figura 4.6: Esquema Centralizado Serie. Daisy-Chained Request Signal.

Cada bus master que requiere la utilización del bus genera la señal bus request y la transmite al siguiente bus master. Cada bus master activa la señal bus request si la recibe activa o si requiere la utilización del bus. De esta manera la señal bus request se transmite a lo largo de la cadena hasta que llega al bus master actual o hasta que llega al final de la cadena. Si el bus master actual recibe la señal bus request activa, debe liberar el bus porque un bus master con mayor prioridad ha realizado un requerimiento. Para liberar el bus, el bus master actual activa la señal bus grant que es común a todos los bus masters (wired-OR) y desactiva la señal bus busy cuando termina la transacción en curso sobre el bus. Todos los bus masters recibirán la señal bus grant porque es común, pero solamente accederá al bus el bus master que no tenga la señal bus request de entrada activa y además tenga activa la misma señal de salida. Los demás bus masters con la señal bus request de salida activa tendrán la señal bus request de entrada también activa y por esta razón pueden identificar que un bus master con mayor prioridad requiere la utilización del bus.

Ambos esquemas serie que se han esquematizado en la Fig. 4.5 y en la Fig. 4.6 se clasifican como centralizados porque la cadena que se construye para una de las líneas (bus request o bus grant) se originan en un procesador. El procesador de mayor prioridad es el origen físico de la cadena y en cada ciclo de asignación del bus la señal que se transmite por la cadena (bus request o bus grant) tiene un único origen. Desde el punto de vista de las prioridades de acceso al bus, el único método que se puede implementar, en principio, es el de prioridades estáticas. Con respecto a los esquemas paralelos, la disminución en cantidad de líneas del bus de control es significativa.

La Fig. 4.7 muestra el esquema de asignación del bus serie no centralizado, que permite la rotación de prioridades. Se denomina también *rotating daisy chain*. Se puede considerar que este esquema es la descentralización del esquema de asignación centralizado serie con la señal de bus grant daisy-chained. En la lógica de asignación correspondiente al bus master actual se genera la señal bus grant, que se comunica de manera circular a todos los demás bus masters. El bus master actual, de esta manera, es el de menor prioridad y el bus master a la derecha del bus master actual es el de mayor prioridad. Se implementa así la política de asignación de prioridades Rotación de Prioridades Dependiente de la Aceptación (o Rotación de Prioridades). Siguiendo el esquema de la Fig. 4.7 el bus master actual genera la señal bus grant que se transmite de un bus master a otro si no hay ningún requerimiento pendiente (señal bus request). Cuando la señal bus grant llega activa a un bus master con un

requerimiento pendiente, la lógica de asignación del bus de este bus master impide que la señal bus grant continúe activa a lo largo de la cadena. Sin embargo, no necesariamente este bus master puede utilizar el bus de forma inmediata. Debe esperar hasta que la señal bus busy (no mostrada en la Fig. 4.7) que es común a todos los bus masters (wired-OR) esté no activa. Si la señal bus busy no está activa, entonces el nuevo bus master puede utilizar inmediatamente el bus, activa esta línea y la de bus grant. Si el bus master actual recibe desactiva la línea bus grant, debe dejar de utilizar el bus lo antes posible y señalizar esta liberación del bus desactivando la línea bus busy. Si hay más de un requerimiento pendiente, el bus master más cercano en la dirección de la cadena de bus grant al bus master actual será el siguiente en utilizar el bus, porque tendrá la mayor prioridad.

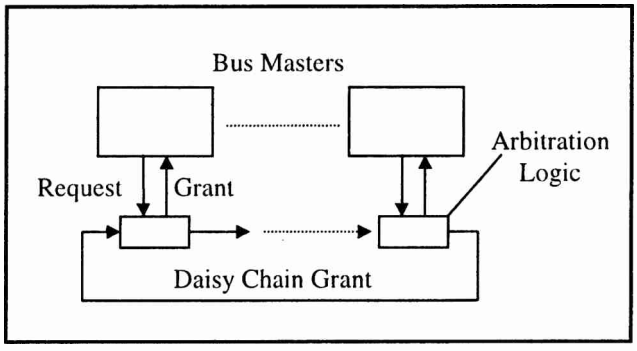


Figura 4.7: Esquema de Asignación Serie no Centralizado.

En vez de que los bus masters requieran el bus y un controlador decida cuál de los requerimientos se va a aceptar, el esquema alternativo propone que el controlador del bus sea el que "pregunte" a los bus masters si tienen un requerimiento pendiente. Estos esquemas se denominan polling schemes, o esquemas de encuesta. Los esquemas de encuesta pueden ser centralizados (un controlador del bus) o no centralizados (varios controladores). Se deben agregar líneas de encuesta desde el controlador del bus a los bus masters para que se pueda llevar a cabo este esquema de asignación. Las señales de control bus request y bus grant deben mantenerse.

Como ejemplo de los esquemas de encuesta, la Fig. 4.8 muestra el esquema de asignación del bus que se denomina esquema de encuesta centralizado.

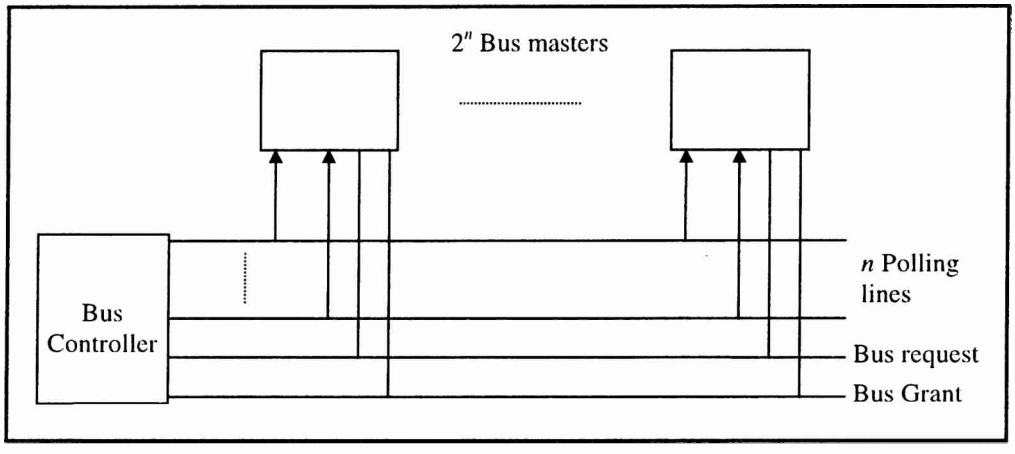


Figura 4.8: Esquema de Asignación de Encuesta Centralizado.

En el esquema de asignación del bus de encuesta centralizada se proveen n líneas de encuesta para 2^n bus masters. Tal como lo muestra la Fig. 4.8, los bus masters se identifican de acuerdo a la combinación de valores de las n líneas de encuesta. Desde el controlador del bus se generan las direcciones de los bus masters para que éstos respondan con la señal bus request si tienen un requerimiento pendiente o no. Si un bus master recibe su dirección de encuesta y tiene un requerimiento pendiente activará la señal bus request. Con esta activación, el controlador identifica que el bus master utilizará el bus y no genera otras direcciones de bus masters. La liberación del bus puede producirse normalmente (el bus master actual termina todas las transferencias pendientes) o porque detecta la dirección de otro bus master en las líneas de encuesta. Bajo este esquema de asignación del bus se pueden implementar distintos métodos de asignación de prioridades. Toda la lógica de asignación se centraliza en el controlador del bus.

4.1.2 Análisis de Rendimiento

Para el análisis de rendimiento se hacen algunas suposiciones sobre el sistema que simplifican el entorno de los multiprocesadores con un único bus de comunicaciones. En principio, se asume que los requerimientos de memoria se generan aleatoriamente, y la probabilidad de que un procesador genere un requerimiento de memoria es uniforme. Por lo tanto, si un procesador genera un requerimiento de acceso a memoria con probabilidad r, la probabilidad de que un procesador no genere un requerimiento es $1-r$. Si se tienen p procesadores, la probabilidad de que ningún procesador genere un requerimiento es $(1-r)^p$. Po lo tanto, la probabilidad de que uno o más procesadores haga un requerimiento está dada por $1-(1-r)^p$. Dado que se tiene un único bus de comunicaciones, se puede aceptar solamente un requerimiento a la vez en cada ciclo de asignación del bus, y por lo tanto la cantidad promedio de requerimientos aceptados en cada ciclo de asignación será

$$BW = 1 - (1 - r)^p \tag{4.1}$$

donde BW es el ancho de banda (**B**and**w**idth) del sistema, que en este contexto se define como la cantidad promedio de requerimientos de memoria que se aceptan en cada ciclo de asignación del bus.

La Fig. 4.9 muestra el índice BW calculado según la Ec. (4.1) para distintos valores de r en función de la cantidad de procesadores p [Wil91]. Se puede notar que cuando la probabilidad de requerimientos de memoria es alta, el bus llega a estar saturado con muy pocos procesadores. Esto significa que agregar procesadores no mejorará el tiempo de ejecución porque para cada requerimiento de los procesadores es altamente probable que se deba esperar hasta que el bus esté disponible. Se debe recordar que cuando hay más de un procesador, la cantidad de peticiones de acceso al bus es mayor de lo que indica la probabilidad de requerimiento del bus r. Esto se debe a que el bus master al que se le negó el acceso al bus en un ciclo de asignación necesariamente seguirá con la transferencia pendiente y, por lo tanto, hará la petición del bus en el ciclo siguiente. En condiciones normales, se da que ningún bus master y en particular ningún procesador, puede seguir operando sin llevar a cabo las transferencias sobre el bus que necesita. Por lo tanto, la Fig. 4.9 es quizás *el mejor caso* en cuanto al rendimiento de un bus de comunicaciones compartido por varios (hasta 16) procesadores.

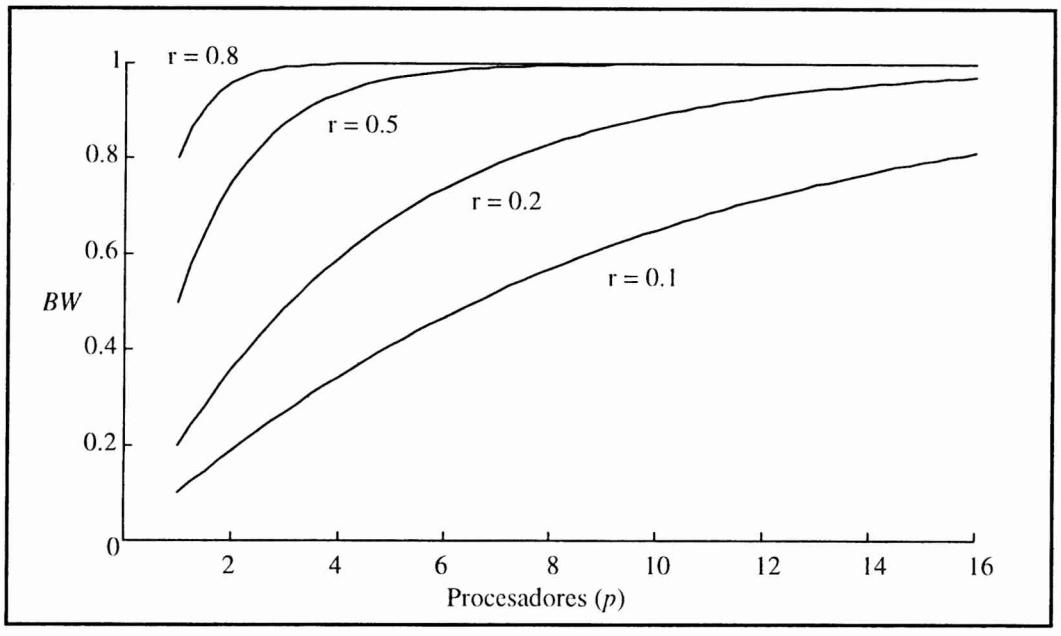


Figura 4.9: Ancho de Banda de un Multiprocesador con Bus Unico.

Si un requerimiento no es aceptado, normalmente se reintentará el acceso en el ciclo siguiente, y por la tanto la probabilidad de requerimientos r se aumenta a una probabilidad ajustada, a, de requerimientos de utilización del bus. La Fig. 4.10 muestra el tiempo de ejecución T de un procesador con todos los requerimientos aceptados de forma inmediata, y el tiempo de ejecución T', con algunos requerimientos bloqueados y reenviados en ciclos siguientes [Yen82].

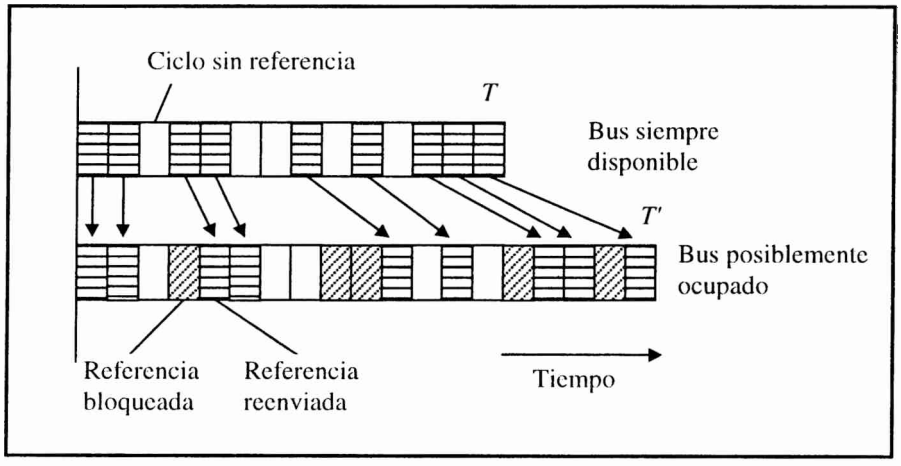


Figura 4.10: Referencias a Memoria sin y con Ocupación del Bus.

Como la cantidad de ciclos sin requerimientos es $T(1-r)$ si el bus siempre puede ser utilizado por el procesador, o $T'(1-a)$ en caso contrario,

$$T'(1-a) = T(1-r) \tag{4.2}$$

Considerando que BW_a es el ancho de banda del bus teniendo en cuenta la probabilidad ajustada, a, de requerimientos de memoria, y la política de asignación del bus es

ecuánime,

$$P_a = \frac{BW_a}{pa} \tag{4.3}$$

donde P_a es la probabilidad de que un requerimiento sea aceptado teniendo en cuenta la probabilidad ajustada, a, de requerimientos de memoria. Como se definió anteriormente, BW_a es la cantidad de requerimientos aceptados en un ciclo de asignación del bus, y P_a es la cantidad de requerimientos en un ciclo de asignación del bus. Utilizando la Ec. (4.1), se llega a que

$$P_a = \frac{1-(1-a)^p}{pa} \tag{4.4}$$

En la Fig. 4.11 se muestra P_a calculado según la Ec. (4.4) para los valores de a: 0.1, 0.2, y 0.5, y hasta 16 procesadores (bus masters).

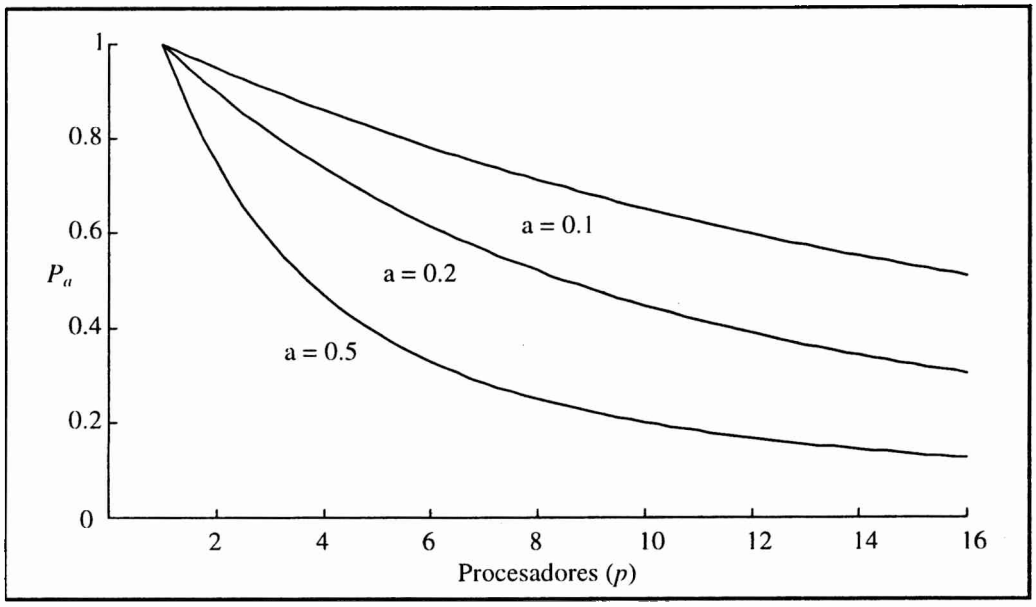


Figura 4.11: Probabilidad de Acceso al Bus.

Se debe notar que los valores significativos de P_a en la Fig. 4.11 son los que se corresponden con los valores enteros de p, que es la cantidad de bus masters (o procesadores) conectados al bus común. De forma análoga a los valores de BW que se obtienen para distintos valores de r (Fig. 4.9), se debe recordar que la probabilidad ajustada, a, es mayor que la probabilidad de requerimiento del bus (r). Además, las colisiones por requerimientos simultáneos (en un mismo ciclo de asignación del bus) aumentan junto con r, y estas colisiones son acumulativas dado que los bus masters continúan requiriendo la utilización del bus. En cualquier caso, el aumento de la probabilidad de requerimientos del bus implica aumentar la probabilidad ajustada de requerimientos del bus, lo cual a su vez implica reducir la probabilidad de acceso al bus en un ciclo de asignación, tal como se puede ver en la Fig. 4.11.

La cantidad de requerimientos que un procesador debe realizar antes de que se acepte la utilización del bus, incluyendo el requerimiento aceptado es $1/P_a$, y por lo tanto se tiene que

$$T' = (1 - r)\,T + (1/P_a)\,r\,T \tag{4.5}$$

porque

- $(1 - r)\,T$ es la cantidad de ciclos en los que no se realiza ningún requerimiento de memoria,
- $r\,T$ es la cantidad de ciclos en los que se requiere la utilización del bus, y
- $1/P_a$ es la cantidad de veces que se debe requerir el acceso al bus cuando hay más de un procesador que lo utiliza.

Por lo tanto, el tiempo de ejecución teniendo en cuenta que los procesadores compiten por la utilización del bus se expresa como

$$T' = ((1 - r) + r/P_a)\,T \tag{4.6}$$

Hasta ahora no hay más que una explicación intuitiva de la relación entre r: probabilidad de requerimiento del bus de cada procesador, y a: probabilidad ajustada teniendo en cuenta la competencia de los procesadores por el acceso al bus. Si se reemplaza T' definido en la Ec. (4.6) en la Ec. (4.2), se llega a que

$$((1 - r) + r/P_a)\,(1 - a) = (1 - r) \tag{4.7}$$

y, por lo tanto,

$$a = 1 - \frac{1 - r}{1 - r + r/P_a} \tag{4.8}$$

Expresado de otra manera,

$$a = \frac{1}{1 + P_a\,(1/r - 1)} \tag{4.9}$$

De esta manera, y utilizando la definición de P_a de acuerdo a la Ec. (4.4) se puede hallar el valor de la probabilidad ajustada de requerimiento del bus, a, respecto a los valores de r y p que se necesiten. La Fig. 4.12 muestra algunos valores de la probabilidad ajustada dependiendo de los valores de p (cantidad de procesadores) y r (probabilidad de requerimiento del bus de cada procesador).

$r \setminus p$	2	4	8	16
0.1	0.105	0.117	0.158	0.438
0.2	0.219	0.276	0.502	0.750
0.5	0.586	0.751	0.875	0.938

Figura 4.12: Valores de Probabilidad Ajustada a.

Lo más significativo de destacar de los valores de la probabilidad ajustada es, como se explicó previamente, el crecimiento de a en función del crecimiento de la cantidad de

procesadores del multiprocesador y de la probabilidad de requerimiento del bus de cada uno de ellos. Por ejemplo, en la Fig. 4.12 se nota que aunque cada procesador requiera el bus el 10% del tiempo si tuviera acceso exclusivo al bus, la probabilidad ajustada indica que cuando debe competir con otros 15 procesadores, el porcentaje de requerimientos aumenta al 44% del tiempo aproximadamente.

En un sistema monoprocesador, el tiempo de ejecución debería ser Tp, si se considera que todos los requerimientos son aceptados. Esto significa que T es el tiempo de ejecución en el multiprocesador y p es el factor de Speed-Up (aceleración). Cuando se debe competir por el bus reenviando requerimientos retrasados, el tiempo de ejecución es T' y el factor de Speed-Up está dado por

$$S = Tp / T' \tag{4.10}$$

donde S es el factor de Speed-Up. Utilizando la Ec. (4.6), se llega a que

$$S = \frac{p}{(1 - r) + r / P_a} \tag{4.11}$$

Si se calculan a y P_a en función de r y p, se pueden obtener los valores de S que le corresponden. En ese caso, se puede notar que el factor de Speed-Up es cercano al lineal hasta que llega a un punto en que no sigue aumentando. Este punto es el denominado punto de saturación, y se debe a que el bus no puede realizar todas las transferencias que se le requieren. Por ejemplo, para $r = 0.1$ (un procesador requiere la utilización del bus de comunicaciones el 10% de los ciclos), el factor de Speed-Up crece de forma lineal con el crecimiento en la cantidad de procesadores hasta llegar a alrededor de 10 procesadores. A partir de este punto, su punto de saturación, el Speed-Up crece muy lentamente o no crece aumentando la cantidad de procesadores. Esto significa que a partir de 10 procesadores no se logrará aumentar la velocidad de procesamiento del sistema multiprocesador aunque se agreguen procesadores.

Una vez que se ha llegado al punto de saturación del factor de Speed-Up, no se mejora sustancialmente la velocidad de procesamiento por el agregado de procesadores al sistema. El problema no es la capacidad en cuanto a la cantidad de datos que se puedan operar en los procesadores, sino en la capacidad del bus de transferir los datos y los resultados entre los procesadores y la memoria. En este caso, el bus compartido es el cuello de botella que determina la velocidad máxima de procesamiento de todo el sistema multiprocesador.

La cantidad de veces que se debe reenviar un requerimiento una vez que no ha sido aceptado puede derivarse de la cantidad de requerimientos totales que se deben realizar para que un requerimiento sea aceptado ($1/P_a$). La cantidad de reenvíos se puede definir en cantidad de ciclos de asignación del bus, se denomina también "tiempo de acceso", y está dada por

$$T_a = (1 - P_a) / P_a \tag{4.12}$$

donde T_a es el tiempo de acceso definido en cantidad de ciclos de asignación, y P_a es la probabilidad ajustada de requerimientos del bus. La Fig. 4.13 muestra T_a para distintas

cantidades de procesadores y distintas probabilidades de requerimiento del bus. Los valores de la probabilidad ajustada (a), utilizados son los que se muestran en la Fig. 4.12 dependientes de la probabilidad de requerimiento del bus (r), y de la cantidad de procesadores del multiprocesador.

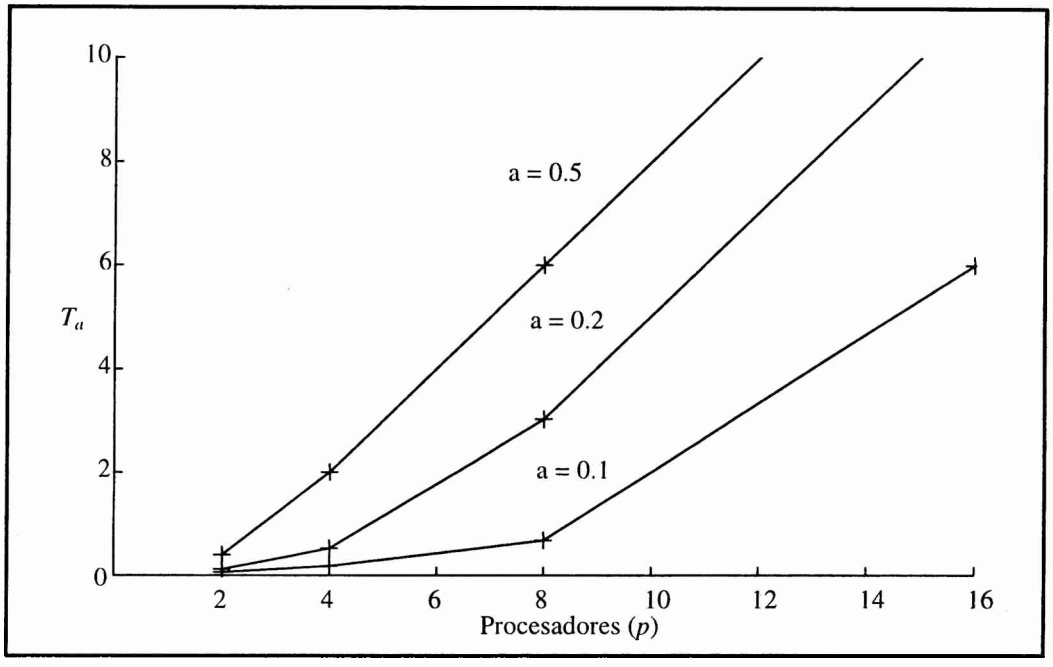


Figura 4.13: Tiempo de acceso (Ciclos).

En la Fig. 4.13, los valores de T_a calculados consecutivamente (para un mismo valor de r, p = 2, 4, 8, y 16), se unieron con líneas rectas para facilitar la visualización. Es inmediata la relación entre la probabilidad ajustada y el tiempo de acceso definido en ciclos de asignación del bus. También es claro que aún para una probabilidad de requerimiento del bus muy baja, como lo es r = 0.2, el tiempo de acceso al bus en ciclos de asignación crece muy rápidamente en función del crecimiento de la cantidad de procesadores.

En las Figs. 4.10-13 se incluyen probabilidades de acceso al bus que no parecen ser muy realistas. Las probabilidades de requerimiento del bus menores de 0.5 suelen ser muy raras de encontrar en casos reales, a menos que se tengan en cuenta otros factores como memoria local asignada a cada procesador o memoria cache. Quizás lo más utilizado para disminuir la competencia con el bus sea la memoria cache. Si se utiliza memoria cache para cada procesador, la relación entre la probabilidad de acierto (hit ratio) y la probabilidad de acceso al bus, es directa.

Todo el análisis de rendimiento de un único bus se ha realizado asumiendo que la política de asignación del bus es ecuánime (fair), todos los procesadores tienen la misma probabilidad de acceso al bus. En el caso de realizar análisis con prioridades fijas, las probabilidades de acceso al bus o de tiempo de acceso al bus, por ejemplo, son distintas según el procesador y la prioridad que se le ha asignado. Lamentablemente es muy difícil incorporar al análisis las probabilidades ajustadas de requerimientos (P_a), porque dependen de la relación entre los procesadores. La situación se complica aún más si las prioridades no solamente son distintas sino que además varían en el tiempo. En estos casos quizás lo más razonable sea

simular el sistema. En caso contrario, las decisiones de simplificación que se deban asumir sobre el sistema para poder tratarlo de forma analítica podrían hacer que los resultados sean demasiado lejanos a la realidad.

También como aclaración general del análisis realizado, se debe recordar que los requerimientos de accesos a memoria, en general, no se realizan al azar sino que suelen seguir un patrón dependiente de la aplicación. Más aún, la probabilidad ajustada que se calcula de forma analítica puede no representar de forma exacta el comportamiento de los requerimientos rechazados que se vuelven a intentar en el ciclo siguiente de asignación del bus. En este sentido, la simulación permitiría (no sin aumentar considerablemente la complejidad) analizar las situaciones que se producen por la ejecución de algunas aplicaciones en particular y por la competencia entre los procesador para la utilización del bus compartido.

4.2 Multiprocesadores con Múltiples Buses

El sistema de un único bus de comunicaciones se puede extender de manera inmediata a un sistema con múltiples buses de comunicación tal como se muestra en la Fig. 4.14. De esta forma, se puede expresar el sistema multiprocesador con p procesadores, m módulos de memoria, y b buses de comunicación. Se asume directamente que la memoria está dividida en módulos porque de otra forma no tendría sentido utilizar más de un bus de comunicación. Solamente un procesador puede utilizar un bus en un instante de tiempo determinado. De igual manera, solamente un procesador puede acceder a un módulo de memoria en un instante de tiempo determinado.

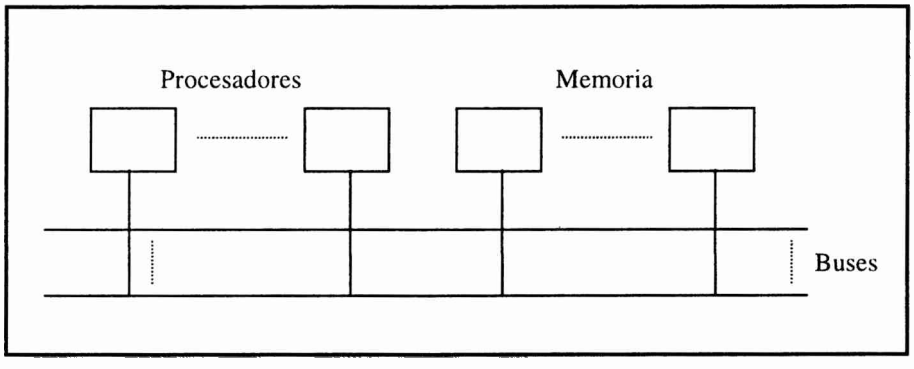


Figura 4.14: Multiprocesador con Múltiples Buses de Interconexión.

En principio, todos los procesadores pueden acceder a cualquier módulo de memoria utilizando cualquier bus de comunicación que esté libre. Usualmente se cumple que $b < m \leq p$, o $b < p < m$, y se pueden realizar b accesos simultáneos a b módulos de memoria distintos. Las razones para utilizar múltiples buses pueden ser varias. Las dos razones más importantes suelen ser: rendimiento y tolerancia a fallas.

Es inmediato que al disponer de varios buses de comunicación se podrá dar acceso a memoria a varios procesadores a la vez, siempre que no haya dos procesadores que accedan al mismo módulo de memoria. Si bien los mecanismos de asignación de buses se hacen más complicados, todo el resto de la lógica de conexión entre cada procesador y la memoria permanece igual. Como se vio en la sección de análisis de rendimiento, un único bus de

comunicaciones llega rápidamente a saturarse aún con la utilización de memorias cache asociadas a cada procesador. Teniendo un mayor número de buses de comunicación, se podría aumentar la cantidad de procesadores en la cual el sistema multiprocesador se satura.

Con la lógica de hardware suficiente para detección de errores de los buses, si hay un bus que no se puede utilizar, es posible continuar con el, o los otros buses de comunicación. En principio todos los buses proveen el mismo tipo de comunicación con la memoria y la misma velocidad de transferencia. Si bien la lógica de hardware necesaria se hace más complicada, teniendo más de un bus de comunicaciones se reduce la probabilidad de que todo el sistema falle por la falla de un bus.

Para reducir la cantidad de conexiones (switches de comunicación) entre procesadores-buses y buses-módulos de memoria, se han propuesto alternativas de configuración. Por ejemplo, los módulos de memoria no necesariamente tienen que estar completamente interconectados, como lo muestra la Fig. 4.15, y agruparse en cuanto a la conexión a los buses. Con esta forma de interconexión se tiene menos flexibilidad en la conexión procesadores-memoria, pero no necesariamente implica una pérdida de rendimiento. En [Lan83] se muestra que se pueden eliminar hasta el 25% de las interconexiones a los buses (switches) y de todas maneras se puede mantener similar el ancho de banda de los múltiples buses.

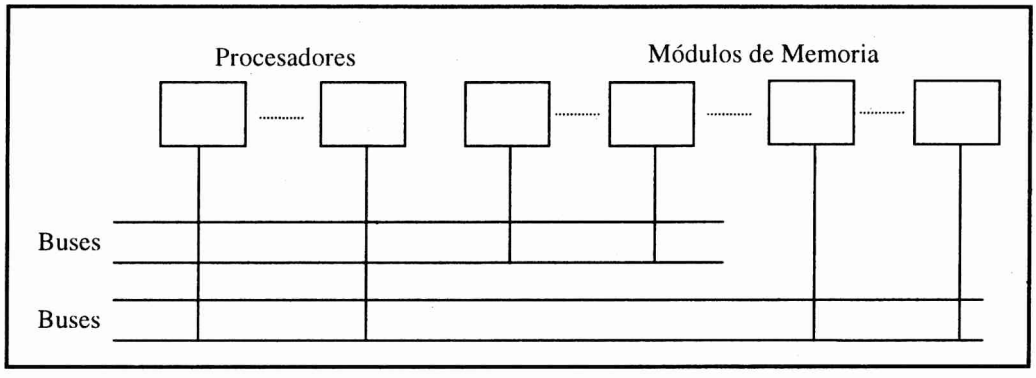


Figura 4.15: Múltiples Buses con Conexiones Parciales.

Como en el caso de los multiprocesadores con un único bus que comunica los procesadores con la memoria compartida, se detallarán algunos aspectos relacionados con la asignación de los buses de comunicación, y con el rendimiento que es posible obtener.

4.2.1 Esquemas de Asignación de los Buses

En un sistema multiprocesador con múltiples buses, se deben resolver dos problemas en lo que se refiere a la asignación de los buses en respuesta a los requerimientos que realizan los procesadores:

1. Los módulos de memoria no realizan más de una transferencia de información en un instante de tiempo.
2. Los buses de comunicación no pueden ser utilizados por más de un procesador en un instante de tiempo.

El primer problema implica elegir solamente un procesador entre todos los que realicen requerimientos para un mismo módulo de memoria. Todos los demás procesadores deberán esperar hasta que el módulo de memoria quede libre para tener la posibilidad de acceder. Se deben secuencializar los accesos a un mismo módulo de memoria.

El segundo problema implica elegir solamente un procesador entre todos los que realizan accesos a memoria para utilizar un bus de comunicaciones que esté libre. Se debe tener en cuenta que pueden haber menos buses libres de los necesarios para realizar todos los accesos a memoria permitidos. En este caso se secuencializan los accesos para acceder a los buses de comunicaciones que estén (o queden) libres.

El problema de asignación de buses se divide en dos fases: en la primera fase se eligen a lo sumo m accesos a memoria, para cada módulo de memoria distinto se elige un procesador. En la segunda fase se eligen hasta b accesos a memoria, para cada bus de comunicación libre se elige un procesador. En la Fig. 4.16 se muestra esta selección de accesos a memoria en dos fases.

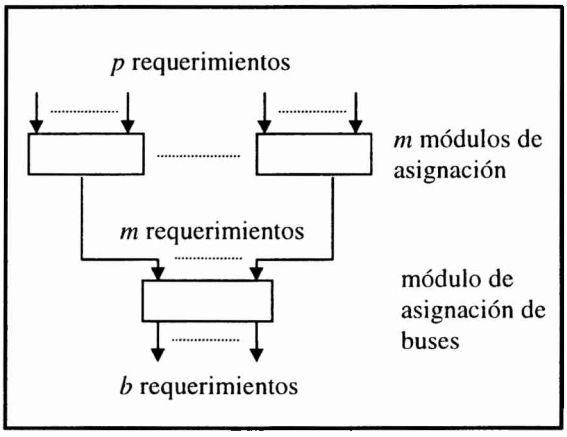


Figura 4.16: Esquema de Asignación de Buses en Dos Fases.

Cada uno de los m módulos de memoria es gestionado por un módulo de asignación que tiene en cuenta todos los requerimientos de los procesadores. La gestión de asignación de los buses se lleva a cabo en un módulo que tiene como entrada m requerimientos (para cada uno de los m módulos de memoria), y genera a lo sumo b requerimientos de utilización de los buses. Si se dispone de más buses que módulos de memoria (por razones de tolerancia a fallas), nunca se utilizarán todos los buses a la vez.

Cada etapa de asignación tiene que tener en cuenta qué recursos ya están ocupados para no asignarlos. La etapa de selección de los módulos de memoria no debe seleccionar un acceso a un módulo de memoria que ya esté en uso. De igual forma no se debe asignar un bus de comunicaciones que ya esté realizando una transferencia de datos entre un módulo de memoria y un procesador.

Los esquemas de asignación de buses y el acceso a los buses por parte de los procesadores no difieren de lo que se detalló en el contexto de un único bus de comunicación. Asimismo, las líneas de control son similares, aunque lo lógica de asignación es más compleja.

4.2.2 Análisis de Rendimiento

En un sistema multiprocesador con múltiples buses de interconexión, los p procesadores y los m módulos de memoria están conectados a b buses de comunicación. El ancho de banda depende de las interferencias entre los procesadores que se produzcan sobre los módulos de memoria y sobre los buses de comunicación. Asumiendo que se cumple que $b \leq m$ y $b \leq p$, un máximo de b transferencias entre procesadores y memoria se pueden llevar a cabo en un ciclo de bus, y esto solamente si los b requerimientos involucran módulos de memoria distintos.

Para el análisis de rendimiento se tienen en cuenta inicialmente las dos fases de selección de los requerimientos:
1. selección de hasta m requerimientos para módulos de memoria distintos y
2. selección de hasta b buses disponibles.

Asumiendo que todos los procesadores (P_1, ..., P_p) tienen la misma probabilidad, r, de realizar una referencia a memoria y que todos los módulos de memoria (M_1, ..., M_m) pueden ser referenciados con la misma probabilidad, esta probabilidad está dada por r/m. La probabilidad de que un procesador, P_i, no realice un requerimiento a un módulo de memoria M_j es $(1- r/m)$. La probabilidad de que ningún procesador realice un requerimiento a un módulo de memoria M_j es $(1- r/m)^p$. Por lo tanto, la probabilidad de que uno o más procesadores realice/n un requerimiento a un módulo de memoria M_j (el módulo de memoria tiene como mínimo un requerimiento desde un procesador) es

$$q = 1- (1- r/m)^p \tag{4.10}$$

La probabilidad de que se realicen exactamente i requerimientos para módulos de memoria distintos en un ciclo de bus está dada en [Mud84] [Goy84]

$$f(i) = \binom{p}{i} q^i (1-q)^{p-i} \tag{4.11}$$

donde se debe recordar que se cumple que $i \leq p$ porque con p procesadores no habrá más de p requerimientos a memoria.

Debido a que se tienen b buses para ser asignados a todos los requerimientos de memoria, se llega a que el ancho de banda o BW (**B**and**w**idth) es

$$BW = \sum_{i=1}^{b-1} i f(i) + \sum_{i=b}^{p} b f(i) \tag{4.12}$$

El primer término se refiere al caso en que se realizan menos de b requerimientos y se utilizan menos de b buses distintos, y el segundo término se refiere al caso en que se realizan más de b requerimientos distintos y se utilizan todos los buses, los b buses disponibles. Como en el caso de los multiprocesadores con un único bus de comunicaciones, el ancho de banda representa la cantidad de requerimientos que se aceptan en un ciclo de asignación.

Calculando *BW*, se encuentra una situación similar a la de un único bus de comunicaciones, tal como se puede ver en la Fig. 4.17. Dependiendo de la probabilidad de requerimiento del bus de cada procesador (r), y de la cantidad de procesadores (p), se llega a la utilización completa de todos los buses, y a partir de ese punto de saturación no se mejora en la cantidad de transferencias de datos entre la memoria compartida y los procesadores. Claramente, en todos los casos la cantidad de módulos de memoria tiene que ser mayor o igual que la cantidad de procesadores.

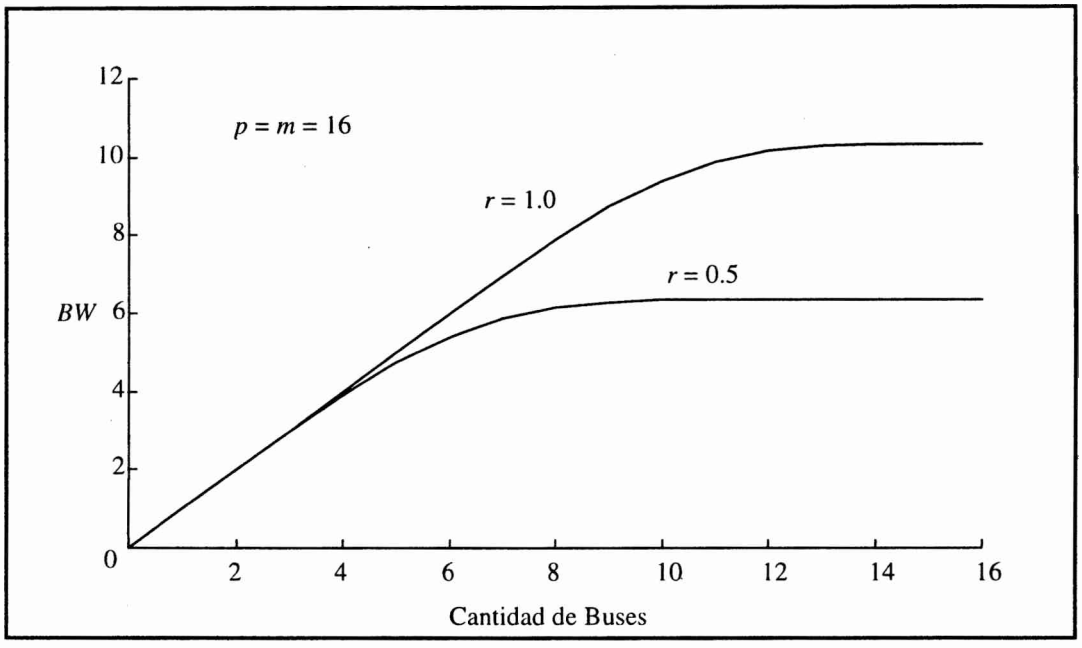


Figura 4.17: Ancho de Banda de un Multiprocesador con Múltiples Buses.

Como en el caso de los multiprocesadores con un único bus de comunicación, la incorporación de prioridades en el método analítico no es trivial. Aunque siempre es posible recurrir a las simulaciones, no conviene sobredimensionar el sistema porque cualquier alternativa de diseño tiene impacto directo sobre el hardware y, por lo tanto, sobre el costo del sistema.

5. Redes de Interconexión

Las redes de interconexión son de fundamental importancia en el diseño y operación de las máquinas paralelas. Aumentar las unidades de procesamiento de un sistema de cómputo paralelo no tiene utilidad a menos que se puedan conectar estas unidades de procesamiento para que cooperen [Hoc88]. En los sistemas MIMD multiprocesador, por ejemplo, se deben interconectar los procesadores con la memoria compartida, y en los sistemas MIMD de memoria distribuida se deben interconectar los procesadores entre sí. Por otro lado, en los sistemas SIMD también es necesario interconectar PEs (Processing Elements), y en [Hwa84] por ejemplo, las redes de interconexión se dan en el contexto de las arquitecturas SIMD.

Tanto para la descripción de una red de interconexión, como para conocer las capacidades de comunicación como el rendimiento, se han establecido diferentes clasificaciones. En la sección siguiente, se definirán algunos términos relacionados con las redes de interconexión y también se detallarán los parámetros de clasificación a tener en cuenta. Como último paso se describen las redes de interconexión más significativas y se detallan algunas de sus características más importantes.

5.1 Definiciones Básicas

Para facilitar el análisis, las redes de interconexión se describirán en términos de nodos de *entrada* y nodos de *salida*, aunque las redes puedan realizar conexiones bidireccionales. En los sistemas multiprocesadores, por ejemplo, los procesadores son los nodos de entrada de la red de comunicaciones y los módulos de memoria son los nodos de salida. Las redes de interconexión proveen un conjunto de interconexiones o *mappings* entre los elementos de los dos conjuntos de nodos, los de entrada y los de salida.

En todas las redes de interconexión se distinguen dos tipos principales de recursos de hardware que llevan a cabo la transmisión de información:

- Los enlaces de comunicación (links)
- Los puntos de conexión entre los enlaces (switches).

Los enlaces de comunicación se dedican al "transporte" de la información, y los puntos de conexión proveen la capacidad y la flexibilidad para conectar los nodos de entrada con los nodos de salida. En la Fig. 5.1 se muestra un ejemplo de red de interconexión que se puede utilizar para la comunicación de seis nodos.

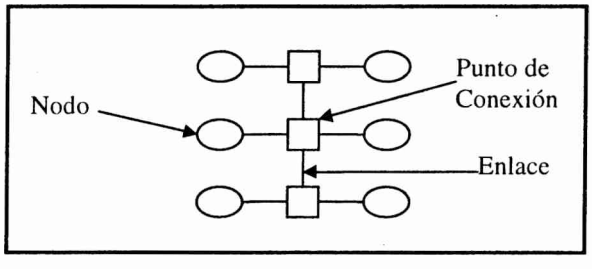


Figura 5.1: Red de Interconexión.

Cada uno de los nodos, en principio, podría comunicarse con todos los demás. La capacidad de interconectar todos los nodos (cada uno de ellos puede enviar información a cualquiera de los otros), depende de la asignación de los puntos de conexión y de la capacidad de los enlaces de comunicación de transmitir información de forma unidireccional o bidireccional.

Una interconexión entre las entradas y las salidas de una red es *bien definida* si cada salida está conectada a una y sólo a una entrada [Hoc88]. Si se tienen N entradas y M salidas, la cantidad de conexiones bien definidas son N^M. Una red de comunicaciones que provea todas las conexiones bien definidas se denomina Red de (Inter)Conexiones Generalizada, o GCN: **G**eneralised **C**onnection **N**etwork. En el contexto de los sistemas de cómputo paralelos, es muy usual encontrar que $N = M$ y que las conexiones necesarias o permitidas, sean uno a uno (una entrada conectada a una salida). En este contexto, por lo tanto, la cantidad de conexiones posibles son $N!$. Una red de comunicaciones que provea cada una de las $N!$ posibles conexiones (una entrada a una salida) se denomina Red de (Inter)Conexiones o CN: **C**onnection **N**etwork. Aunque estas definiciones no son muy tenidas en cuenta, dan un marco de referencia al menos parcial de las características y capacidades de una red de interconexión.

5.2 Diseño y Clasificaciones

Las decisiones a tomar para el diseño de una red de comunicaciones pueden ser divididas en cuatro aspectos [Hwa84]:

1. Modos de Operación
2. Estrategias de Control
3. Metodologías de Conexión entre Entrada y Salida
4. Topologías de la Red

Estos cuatro factores también son muy tenidos en cuenta para la clasificación de las redes de interconexión de los nodos.

El modo de operación de una red de interconexión se refiere a la existencia o no de un reloj global que coordine el control de acceso y se utilice en los enlaces y los puntos de comunicación de la red de interconexión. En este sentido, los dos tipos de operación más clásicos a los que se hace referencia son: sincrónico y asincrónico. El modo de operación sincrónico es el más común en los sistemas SIMD, donde toda la operación de la red de comunicaciones se sincroniza con un reloj común. Los requerimientos de comunicación se producen todos al mismo tiempo. Por otro lado, la operación asincrónica es la más encontrada en los sistemas MIMD, donde los requerimientos de comunicación entre los nodos se producen dinámicamente y de forma independiente.

Las estrategias de control son las que determinan la asignación de operación de los puntos de conexión (switches). Si el estado de todos los puntos de conexión se determina (asigna) en un controlador central, las redes de interconexión son de *control centralizado*. Si, por otro lado, en los mismos puntos de conexión se determina su estado, la red de interconexión es de control distribuido. Las arquitecturas SIMD tienden a tener control centralizado y las MIMD tienden al control distribuido.

Las metodologías de conexión entre una entrada y una salida de la red son básicamente dos: (a) circuito de conexión o circuit switching, y (b) conexión por paquetes o packet

switching. En el caso de elegirse el circuito de conexión, se establece un camino físico entre la entrada y la salida. Este camino físico involucra enlaces y puntos de conexión que se *dedican* a la comunicación entre la entrada y la salida. Si la conexión se realiza por paquetes, los datos se introducen en la red desde la entrada en forma de paquetes. La transmisión de estos paquetes en la red no necesariamente sigue un mismo camino, sino que los paquetes pueden llegar a la salida utilizando distintos enlaces y pasando por distintos puntos de conexión.

Las topologías de conexión tienen relación con la representación de las redes de conexión como grafos, donde los nodos representan los puntos de conexión y los arcos representan los enlaces de comunicación. Esta representación tiene impacto directo sobre las formas de conectar las entradas con las salidas. Las topologías tienden a ser regulares y se pueden agrupar en dos categorías: estáticas y dinámicas. Dado que este es el criterio más tenido en cuenta en las clasificaciones, es el que será enfocado con más atención.

Como se ha afirmado previamente, el criterio de clasificación más utilizado en las redes de interconexión, es la capacidad de conectar los nodos de entrada con los nodos de salida (topología, en [Hwa84]). También se tiene en cuenta si todas las conexiones se pueden realizar a la vez, o si, habiendo algunas conexiones que ya se han establecido, se pueden llevar a cabo las que faltan. En general, no se tienen en cuenta la cantidad de enlaces de comunicación y de puntos de conexión en las clasificaciones, aunque sí son importantes cuando se debe evaluar el costo de la red de interconexión.

Teniendo en cuenta los cuatro criterios de decisión que se han detallado, las clasificaciones de las redes de interconexión se podrían esquematizar como lo muestra la Fig. 5.2.

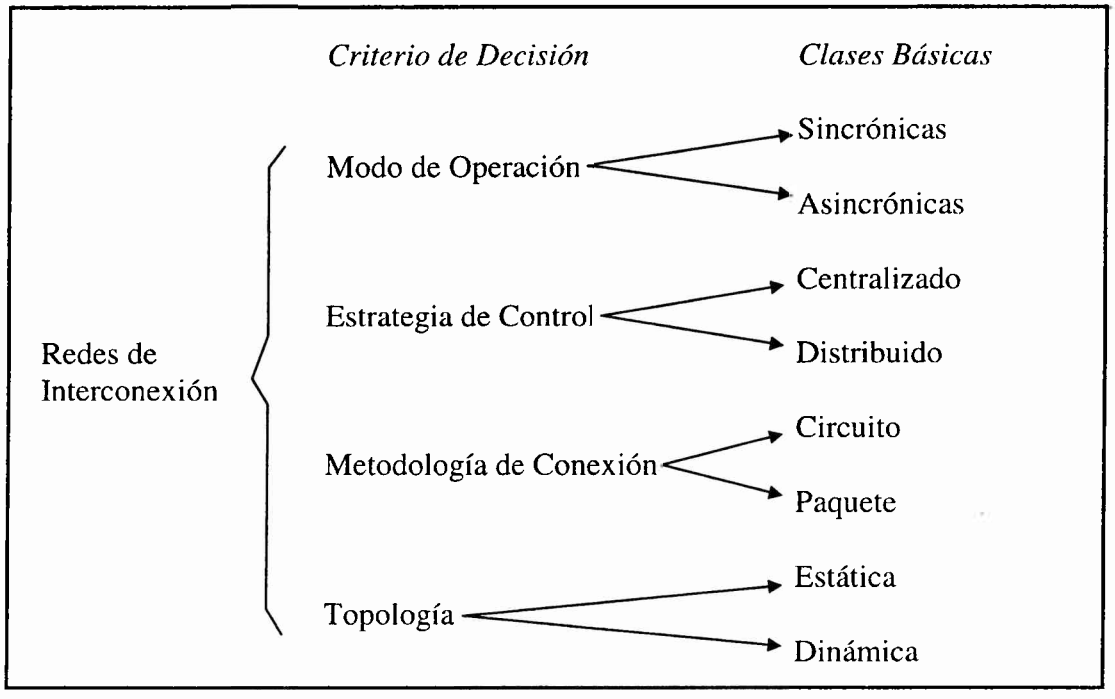


Figura 5.2: Criterios de Decisión en las Redes de Interconexión.

Quizás muchas de las redes que se han desarrollado no sean presentadas en la

bibliografía teniendo en cuenta los *Criterios de Decisión* y las *Clases Básicas* que se muestran en la Fig. 5.2. Se debe tener en cuenta que la mayoría de las clasificaciones intenta poner orden en las características relevantes a la hora de evaluar y comparar distintas alternativas. Si bien es cierto que muchas de las redes de interconexión se han desarrollado con fines específicos y en este sentido se podrían dar nuevas necesidades para las redes de interconexión, siempre conviene utilizar o al menos comparar lo que se va a diseñar con lo que ya se ha hecho y probado en casos reales.

La descripción y el análisis de las topologías se realizará en función de la flexibilidad para conectar los nodos de entrada y de salida de la red. Desde el punto de vista de las aplicaciones paralelas y la programación de la máquina paralela, mayor flexibilidad implica mayor facilidad para la ejecución de la aplicación sobre la máquina paralela. Claramente, a mayor flexibilidad en la conexión de los nodos, mayor será la complejidad y el costo de la red de interconexión.

5.2.1 Topología de las Redes de Interconexión

Ya en el contexto de las topologías, las redes de interconexión dinámicas son las que permiten cambiar las conexiones entre los nodos de entrada y de salida dinámicamente, en tiempo de ejecución de aplicaciones [Wil91]. Es posible cambiar el estado de los puntos de conexión (switches) para llevar a cabo diferentes conexiones entre nodos de entrada y nodos de salida. Las redes de interconexión dinámicas permiten la conexión directa de una entrada con cualquier salida. Puede suceder que se utilicen varios enlaces y puntos de conexión, pero la comunicación no se debe transferir por nodos intermedios de entrada y/o de salida.

La Fig. 5.3 muestra de forma esquemática una red de interconexión dinámica y su relación con los nodos de entrada y de salida. Como se ve en la figura, la red es usualmente transparente para los nodos que se comunican.

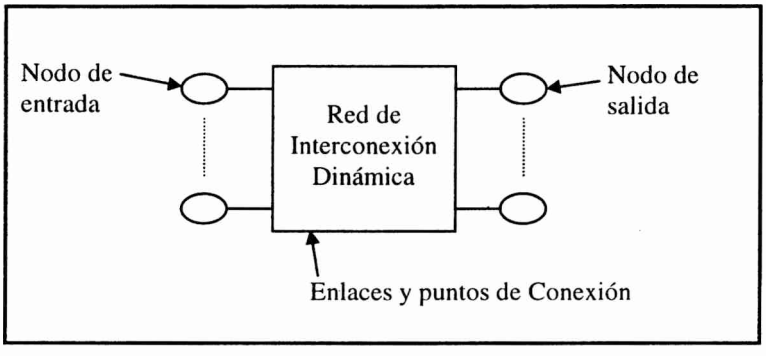


Figura 5.3: Red de Interconexión Dinámica.

Los nodos de entrada de la Fig. 5.3 solamente envían requerimientos de comunicación a la red, y por esta razón las redes de interconexión dinámicas también suelen denominarse redes indirectas de interconexión. Los mensajes entre los nodos de procesamiento son manejados indirectamente a través de los puntos de conexión [Sei84]. La utilización más extendida se ha dado entre las computadoras paralelas con arquitectura MIMD de memoria compartida para comunicar los procesadores con los módulos de memoria. También se han

utilizado en la interconexión de elementos de procesamiento (PEs) de una máquina con arquitectura SIMD.

En las redes de interconexión estáticas, cada nodo a comunicar tiene un conjunto de enlaces establecidos previamente con otros nodos y cada enlace comunica siempre el mismo par de nodos. Hay nodos conectados directamente a través de enlaces, pero normalmente un nodo no se puede comunicar en forma directa con todos los demás, sino con un subconjunto de los demás nodos. La única forma de comunicar directamente a todos los nodos entre sí, es haciendo posible que cada nodo tenga un enlace con cada uno de los demás. Para N nodos, la cantidad de enlaces necesarios sería N (N-1) si cada enlace fuera unidireccional, o N (N-1) / 2 si los enlaces fueran bidireccionales. Las redes estáticas completamente conectadas son posibles para cantidades pequeñas de nodos a comunicar (por ejemplo, 4). La Fig. 5.4 muestra una red de interconexión estática, donde cada nodo tiene a lo sumo 4 enlaces directos con otros nodos.

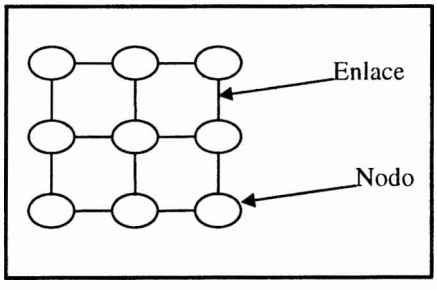


Figura 5.4: Red de Interconexión Estática.

Las redes de interconexión estáticas también suelen denominarse directas, porque toda la transferencia de información de un enlace a otro se lleva acabo en los nodos de procesamiento [Sei84]. Los nodos que se comunican están *directamente* conectados entre sí y no a *la red*. Son las más utilizadas en los sistemas de cómputo MIMD con memoria distribuida para conectar procesadores. También son utilizadas en los sistemas SIMD para conectar los elementos de procesamiento (PEs). Como se puede notar en la Fig. 5.4, no hay puntos de conexión que se puedan manipular para cambiar conexiones. Si se consideran a los nodos de una red de interconexión estática como puntos de conexión (switches), entonces se puede llegar a considerar como una red de interconexión dinámica. En estas redes, para comunicar dos nodos que no están conectados por un enlace se debe pasar por nodos intermedios de comunicación.

Si las redes de interconexión dinámicas permiten la conexión simultánea de todas las combinaciones posibles de nodos de entrada y de nodos de salida se denominan *no bloqueantes* (non-blocking). A su vez, las redes no bloqueantes se denominan *estrictamente no bloqueantes* si una nueva conexión entre un nodo de entrada y un nodo de salida no conectados, siempre se puede realizar sin cambiar los caminos ya establecidos. Las redes estrictamente no bloqueantes no se ven afectadas por las conexiones ya realizadas entre nodos de entrada y nodos de salida, una nueva conexión siempre se puede realizar por caminos disponibles si los nodos no están conectados. Algunas redes no bloqueantes requieren que los caminos se rehagan para que se puedan conectar nuevos nodos, y estas redes se denominan no bloqueantes en sentido amplio o *rearrangeable networks*. Rehacer caminos significa reasignar

el estado de los puntos de conexión, la información será transferida por distintos enlaces y pasará por distintos puntos de conexión.

Aunque en algunas redes dinámicas es posible lograr que un nodo de entrada se pueda conectar a cualquier nodo de salida, no se permiten realizar simultáneamente todas las conexiones posibles entre nodos de entrada y nodos de salida. Estas redes se denominan bloqueantes. Es posible que algunos requerimientos de comunicación no se puedan satisfacer porque no hay recursos disponibles para todas las conexiones a la vez, y por lo tanto las comunicaciones se deben realizar en serie.

En las redes de interconexión dinámicas, los puntos de conexión (switches) suelen agruparse por etapas y por esta razón las redes se clasifican como de una sola etapa o de más de una etapa de puntos de conexión. Las de una sola etapa se denominan redes de etapa única (single stage), y las de más de una etapa se denominan redes multietapa (multistage). En las redes de etapa única, la información pasa a través de un sólo punto de conexión (una sola etapa de conexiones) desde la entrada hasta la salida. En las redes multietapa, la información pasa a través de más de un punto de conexión (más de una etapa de conexiones) desde el nodo de entrada hasta el nodo de salida.

La cantidad de puntos de conexión y por lo tanto de etapas de puntos de conexión es importante porque puede dar una idea del tiempo de latencia en la transmisión de la información. La latencia se define como el tiempo que se necesita para enviar una unidad de información desde un punto de entrada hasta un punto de salida de la red. A mayor cantidad de etapas, se tiene que transmitir la información por mayor cantidad de enlaces y puntos de comunicación y esto en general implica un mayor tiempo de latencia.

También dentro de la topología de las redes de interconexión, se pone especial atención a la forma en que se conectan los nodos de las redes estáticas, así como las formas en que se conectan los puntos de conexión de las redes dinámicas. Se recurre a diferentes formas, aunque todas o casi todas se basan en la numeración (o dirección) de cada nodo o enlace y utilizan la notación binaria de la numeración. En [Hoc88], por ejemplo, se presentan las topologías (sin discriminación de redes estáticas o dinámicas), en términos de *permutaciones* de las representaciones binarias de los nodos o links, y se recurre al álgebra de permutaciones para definir las topologías. En [Dec91] se presentan las topologías en términos de funciones de interconexión que asignan una dirección de entrada a una de salida de forma matemática.

5.3 Redes con Buses de Comunicación

En el contexto de los multiprocesadores, la utilización de buses para la interconexión de los procesadores con la memoria compartida tal como se lo describió en el capítulo anterior, puede ser considerada como una red de interconexión. En la Fig. 5.5 se muestra un sistema multiprocesador con múltiples buses de interconexión desde el punto de vista de las redes de interconexión. Los nodos de entrada de la red de interconexión son los procesadores. Los nodos de salida de la red son los módulos de la memoria compartida. Los puntos de conexión son los que unen a los procesadores y a los módulos de memoria con el o los buses. Un punto de conexión (switch) es necesario entre cada procesador y cada bus de comunicación. De la

misma manera, un punto de conexión es necesario entre cada bus de comunicación y cada módulo de memoria.

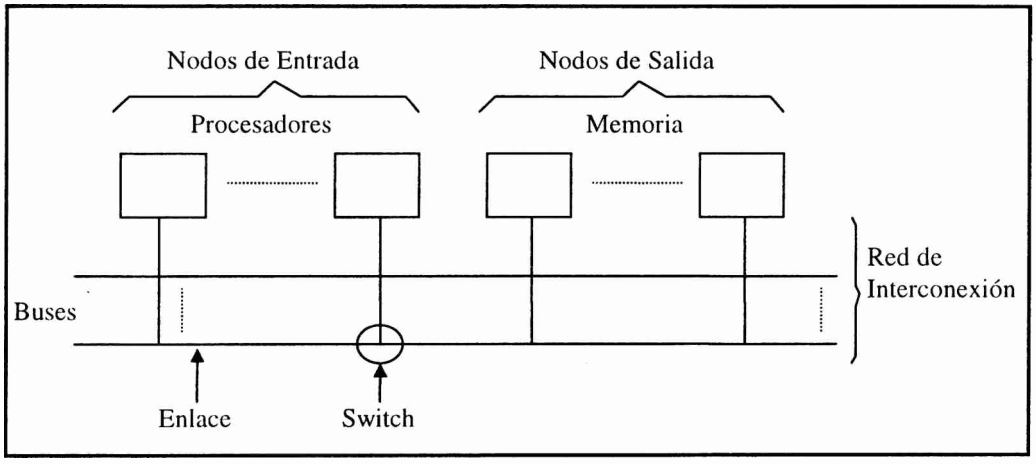


Figura 5.5: Red de Interconexión de Múltiples Buses.

De acuerdo con el esquema de la Fig. 5.5, si p es la cantidad de procesadores, b es la cantidad de buses y m es la cantidad de módulos de memoria,

$$\#SWI = b\,(p + m) \tag{5.1}$$

donde $\#SWI$ es la cantidad de puntos de conexión (switches) necesarios para una red de interconexión tal como la de la Fig. 5.5. Debe notarse que, por ejemplo, la red de la Fig. 5.5 tiene todos los procesadores y todos los módulos de memoria conectados a todos los buses de comunicación. También desde el punto de vista de las redes de interconexión, el único tipo de conexión en los multiprocesadores es uno a uno: un procesador con un módulo de memoria. Cada módulo de memoria puede ser accedido por un sólo procesador.

De acuerdo con los criterios de definición y clasificación de las redes, un sistema de múltiples buses de comunicación como el de la Fig. 5.5 se puede considerar como una red de interconexión

- Dinámica: cualquier nodo de entrada (procesador) puede conectarse con cualquier nodo de salida (módulo de memoria), cambiando el estado (conexión - no conexión) de los puntos de conexión (switches).
- Bloqueante: si la cantidad de buses es menor que la cantidad de procesadores, no todos los requerimientos de conexión (procesador - módulo de memoria) se podrán satisfacer de forma simultánea.
- Multietapa: Todas las comunicaciones deben pasar por dos etapas de puntos de conexión. Una es la que une un procesador a un bus de comunicaciones. La segunda etapa es la que une el bus de comunicaciones con el módulo de memoria que se requiere. Ambas etapas son resueltas por el mecanismo de asignación de buses, tal como se detalló en el capítulo anterior.

La estrategia de control puede ser centralizada o no, dependiendo del esquema de asignación que se utilice, tal como ya se ha detallado.

Aunque un único bus de comunicaciones puede ser clasificado de la misma manera

que un sistema de múltiples buses de comunicación (dinámica, bloqueante y multietapa), la cantidad de conexiones permitidas simultáneamente que se pueden llevar a cabo no es la misma. A mayor cantidad de buses que se utilicen, mayor será la cantidad de conexiones simultáneas entre un procesador y un módulo de memoria permitidas.

5.4 Red Cross-Bar

Las redes de interconexión de tipo cross-bar conectan los elementos utilizando una matriz de puntos de conexión (switches), con un punto de conexión por cada par de elementos a comunicar, tal como lo muestra la Fig. 5.6. La red de interconexión cross-bar permite la comunicación entre cualquier par de elementos a comunicar cambiando el estado de los puntos de conexión (conexión - no conexión). Esta característica hace que las redes cross-bar sean clasificadas como dinámicas.

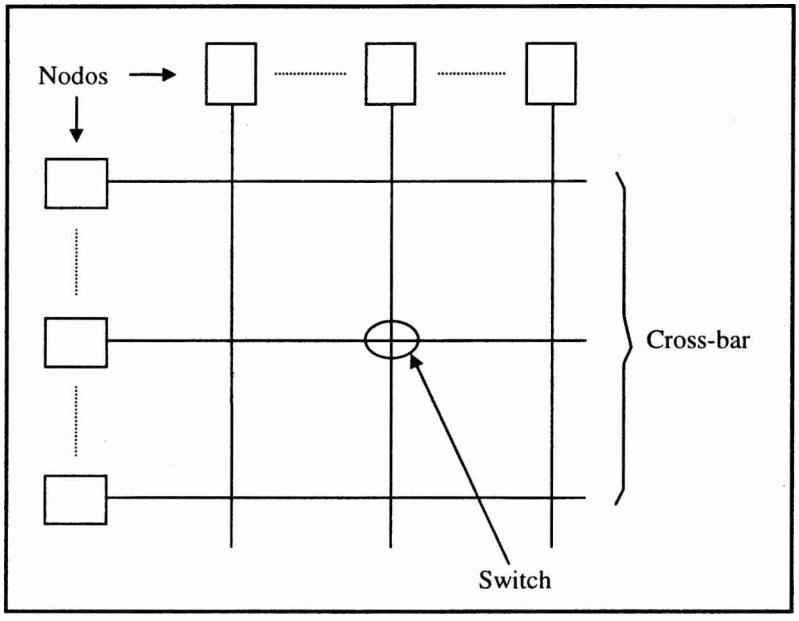


Figura 5.6: Red de Interconexión Cross-bar.

Las conexiones entre cualquier par de nodos no comunicados se puede llevar a cabo sin importar las conexiones que ya estén hechas y sin cambiarlas. Por esta razón, las redes cross-bar se incluyen dentro de la clase de redes de interconexión dinámicas estrictamente no bloqueantes. Los elementos a comunicar pueden ser procesadores entre sí, o procesadores con módulos de memoria. Las conexiones que usualmente se establecen son uno a uno, aunque se podrían realizar conexiones *bien definidas* tal como se definen en [Hoc88].

Desde el punto de vista de las conexiones en un sistema multiprocesador, las redes cross-bar conectan los procesadores con los módulos de memoria. Nuevamente en este caso las conexiones posibles son uno a uno, porque a lo sumo un procesador puede acceder a cada módulo de memoria. En este contexto, las redes cross-bar se pueden considerar como el caso extremo de utilización de múltiples buses de comunicación, con $b = p$, igual cantidad de buses de comunicación y de procesadores.

Desde el punto de vista de la asignación de los buses, la Fig. 5.7 muestra un esquema de asignación de buses no centralizado que se aplica al control de las redes cross-bars. La asignación de cada bus que conecta a los procesadores con un módulo de memoria (vertical en la Fig. 5.7), se realiza independientemente de los demás y se asocia al mismo bus. También se podría implementar un sistema de control centralizado, a diferencia del esquema de la Fig. 5.7. Los sistemas cross-bar con control centralizado [Wil91] suelen denominarse master-slave, dado que un procesador es el que controla todos los puntos de conexión y por lo tanto asigna las conexiones procesador-memoria que se llevarán a cabo. Este procesador es el que determina qué procesadores se podrán comunicar y es el que se denomina procesador master. Los demás procesadores son los slaves, se comunican a requerimiento del master.

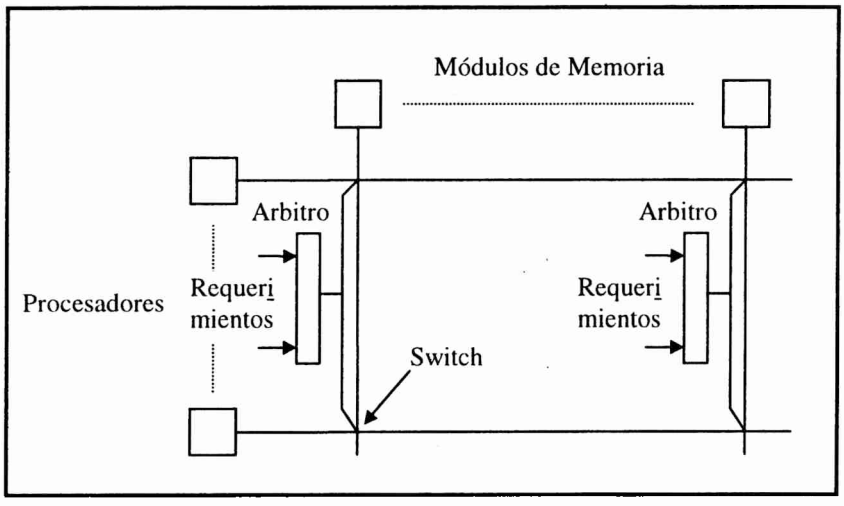


Figura 5.7: Cross-bar con Control no Centralizado.

Si N es la cantidad de nodos a comunicar y CB_{NxN} es la matriz de puntos de conexión, para conectar el nodo i con el nodo j, el único punto de conexión por el que se debe pasar es el CB_{ij}. Por lo tanto la red cross-bar es de etapa única, teniendo en cuenta la cantidad de puntos de conexión por los que la información debe pasar, porque la red cross-bar *es* la etapa de puntos de conexión.

En resumen, la red de interconexión cross-bar se clasifica como (a) dinámica, (b) estrictamente no bloqueante, y (c) de etapa única.

5.4.1 Análisis de Rendimiento en Multiprocesadores

El análisis de rendimiento de la red de interconexión cross-bar en el contexto de los sistemas multiprocesadores sigue las líneas generales del análisis con múltiples buses de comunicación. Los nodos de entrada de la red (cross-bar) son los procesadores y los nodos de salida de la red son los módulos de memoria.

Siendo p la cantidad de procesadores, donde cada uno de ellos tiene la misma probabilidad, r, de requerir el acceso a cualquiera de los m módulos de memoria, ya se ha detallado que

$$q = 1 - (1 - r/m)^p \tag{5.2}$$

en la Ec. (3.10). Donde q es la probabilidad de que un módulo M_j (j = 1, ..., m), tiene como mínimo un requerimiento de acceso. Por otro lado, dado que la red cross-bar es no bloqueante, el ancho de banda BW, que se define como la cantidad de accesos a memoria que se pueden realizar por ciclo [Wil91], es exactamente igual a la cantidad de requerimientos. Por lo tanto, el ancho de banda de la red cross-bar está dado por

$$BW = m\,q \tag{5.3}$$

porque se tienen m módulos de memoria. Utilizando la Ec. (5.2) se tiene que

$$BW = m\,(1 - (1 - r/m)^p) \tag{5.4}$$

La Fig. 5.8 muestra BW calculado según la Ec. (5.4) para distintas cantidades de procesadores y módulos de memoria (manteniendo $p = m$) y distintos valores de r.

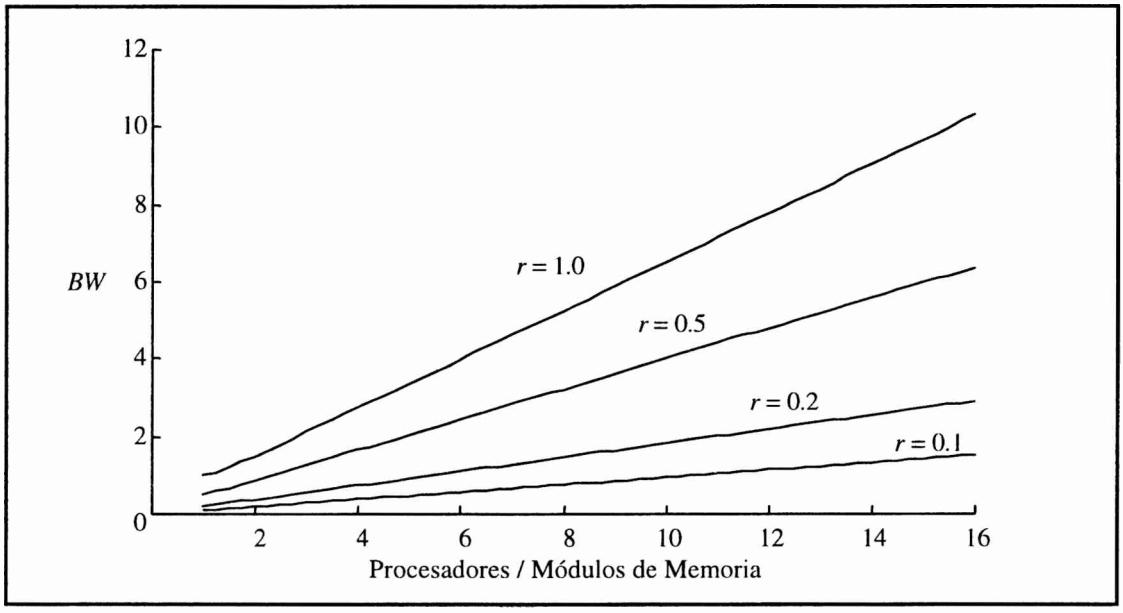


Figura 5.8: BW de un Multiprocesador con Interconexión Cross-Bar.

Se debería recordar que la probabilidad de requerimientos r no tiene en cuenta las interferencias posibles entre los procesadores que intentan acceder al mismo módulo de memoria. También se debe notar que r solamente es una aproximación en que se asume que los procesadores requieren accesos al azar a módulos de memoria al azar. En este sentido, los resultados analíticos se deben tomar como referencia a un hipotético valor promedio de las aplicaciones, pero que puede diferir de forma arbitraria con el rendimiento que se obtenga en algunas aplicaciones.

Si se comparan los resultados de rendimiento de los múltiples buses con los de la red cross-bar se verá que el ancho de banda que se logra con la red cross-bar es la cota superior del rendimiento que se puede lograr con los múltiples buses. Otra de las comparaciones significativas se da en el *comportamiento*: los múltiples buses se saturan debido a la cantidad

de requerimientos, mientras que la red cross-bar puede gestionar todos los requerimientos de conexión permitidos entre procesadores y módulos de memoria.

Quizás una de las características más atractivas que se puede observar en la Fig. 5.8 es el crecimiento lineal de *BW* desde el punto de vista analítico, en función del crecimiento de los procesadores y los módulos de memoria. Esta característica, junto con la de ser una red de interconexión dinámica no bloqueante, hace que las demás redes de interconexión tiendan a comparase con la red cross-bar.

5.4.2 Costo de las Redes Cross-Bar

Aunque las características de conexión y los índices de rendimiento de las redes cross-bar las hacen muy atractivas, el costo que suponen suele ser un problema sin solución posible. Teniendo en cuenta solamente la cantidad de puntos de conexión (switches), si se tienen N nodos de entrada y M nodos de salida,

$$\#SWI = N \times M \tag{5.5}$$

donde *#SWI* indica la cantidad de puntos de conexión de la red cross-bar para conectar todos los nodos. Si, como antes, se asume que $N = M$, entonces

$$\#SWI = N^2 \tag{5.6}$$

Aún suponiendo que el crecimiento lineal de ancho de banda logrado con las redes cross-bar se traduce en un crecimiento también lineal en la velocidad de procesamiento, el costo del sistema no parece seguir la misma línea. Si bien el costo de la red cross-bar no es el costo de todo el sistema de multiprocesamiento, dado que crece de forma cuadrática con respecto al los recursos de hardware del sistema (procesadores y módulos de memoria), según la Ec. (5.6), llegará a dominarlo.

En el costo de las redes cross-bar también se debe incluir la lógica de asignación de los switches. Aunque este costo es importante, suele crecer de forma lineal con respecto a la cantidad de nodos a comunicar, sobre todo si la lógica de asignación es "no centralizada".

Es usual que las redes cross-bar no se propongan como el único hardware de interconexión de más de 8 elementos a comunicar. Cuando de debe superar esa cantidad, se la utiliza al menos de dos maneras: a) como punto de referencia del óptimo rendimiento que se puede lograr, y b) como celda básica para la construcción de otro tipo de redes de interconexión.

5.5 Permutaciones

Se verán en esta sección algunas de las permutaciones [Hoc88] más utilizadas tanto en la interconexión de los nodos de una red estática como en la interconexión de los puntos de conexión de una red dinámica. Como se afirmó previamente, se recurre a la numeración de los nodos o enlaces de la red, que a partir de ahora se describirán en términos de elementos. Esta

numeración se define entre 0 y N-1 para N nodos, el número asociado a cada nodo es su *dirección*, y se utiliza la notación binaria de cada dirección. Todas las permutaciones definen conexiones uno a uno entre los elementos. De esta manera,

$$x = \{b_n, b_{n-1}, ..., b_1\} = b_n\, 2^{n-1} + b_{n-1}\, 2^{n-2} + ... + b_1 \tag{5.7}$$

es la dirección binaria de uno de los elementos. Las permutaciones del conjunto de entradas, ahora pueden ser definidas por operaciones sobre su dirección binaria.

Debido a la notación binaria empleada y a la forma de definir conexiones en términos de permutaciones de bits, se asume que la cantidad de nodos a comunicar es una potencia de dos.

5.5.1 Permutaciones de Intercambio

Las permutaciones intercambio están definidas por

$$\varepsilon_k(x) = \{b_n, ..., \overline{b_k}, ..., b_1\} \tag{5.8}$$

donde $\overline{b_k}$ indica el complemento de b_k. La Fig. 5.9 muestra las dos permutaciones de intercambio posibles para cuatro elementos.

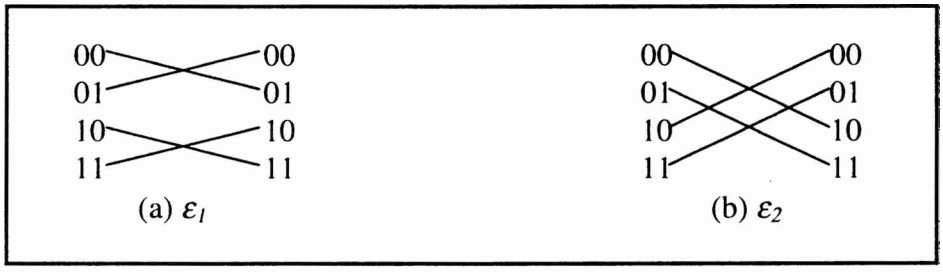


Figura 5.9: Permutaciones de Intercambio.

En la Ec. (5.8), k representa la posición del bit que se permutará. La cantidad de dígitos binarios que se utilicen en la representación de los elementos será la cantidad de posibles permutaciones de intercambio. Si se siguen las definiciones de nodos de entrada y nodos de salida. En la Fig. 5.8 (a), el nodo 00 se comunica con el nodo 01, el nodo 01 con el nodo 00, el nodo 10 con el nodo 11, y el nodo 11 con el nodo 10.

5.5.2 Permutación Perfect Shuffle

Esta permutación es llamada así (mezcla perfecta) porque puede ser realizada dividiendo el conjunto de elementos en dos partes iguales e intercalando los elementos de las dos partes. Con respecto a las operaciones sobre las direcciones binarias, está definida por

$$\sigma(x) = \{b_{n-1}, b_{n-2}, ..., b_1, b_n\} \tag{5.9}$$

o, lo que es similar, una rotación circular a izquierda de la notación binaria.

En la Fig. 5.10 se muestra esta permutación para ocho elementos a comunicar. A diferencia de las permutaciones de intercambio, la permutación perfect shuffle es única, no depende de la cantidad de dígitos binarios que se utilicen en la representación de los elementos ni tiene variaciones.

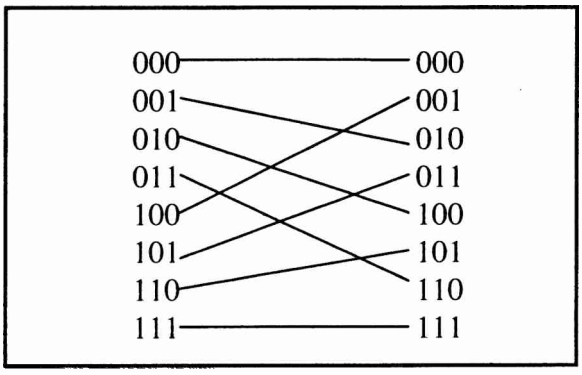


Figura 5.10: Permutación Perfect Shuffle.

Esta permutación no se considera muy útil por sí sola, sino combinada con otras permutaciones en una red multietapa, por ejemplo. Siguiendo el ejemplo de la Fig. 5.10, esta red por sí sola dejaría incomunicados a los elementos 000 y 111. Si se piensa que cada nodo es un elemento de procesamiento, estos elementos no podrían participar en la ejecución de una aplicación paralela a menos que se provea de alguna otra forma de interconexión.

También se pueden definir permutaciones k-ésimo sub-shuffle, denotado como $\sigma_k(x)$, y k-ésimo super-shuffle $\sigma^k(x)$, como la rotación circular a izquierda de los k bits menos y más significativos respectivamente.

5.5.3 Permutación Mariposa (Butterfly)

Esta permutación está definida por el intercambio del primer bit con el último bit de la notación binaria de los elementos. Por lo tanto

$$\beta(x) = \{b_1, b_{n-1}, ..., b_2, b_n\} \tag{5.10}$$

La permutación butterfly para ocho elementos (tres bits) se muestra en la Fig. 5.11. Como en el caso de perfect shuffle, se pueden definir permutaciones sub-butterfly y super-butterfly. También como la permutación perfect shuffle para ocho elementos, quedan definidos varios subconjuntos de elementos disjuntos (que no se comunican entre sí). Cada uno de estos subconjuntos está formados por:

1. El elemento 000.
2. Los elementos 001 y 100.
3. El elemento 010.
4. Los elementos 011 y 110.
5. El elemento 101.
6. El elemento 111.

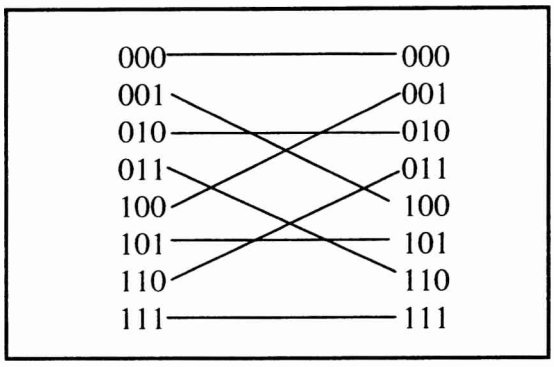


Figura 5.11: Permutación Butterfly.

5.5.4 Permutación Baseline

Esta permutación corresponde a la rotación circular a derecha de los bits de la representación binaria de los elementos. Por lo tanto

$$BL(x) = \{b_1, b_n, ..., b_2\} \tag{5.11}$$

La Fig. 5.12 muestra esta permutación con ocho elementos a comunicar.

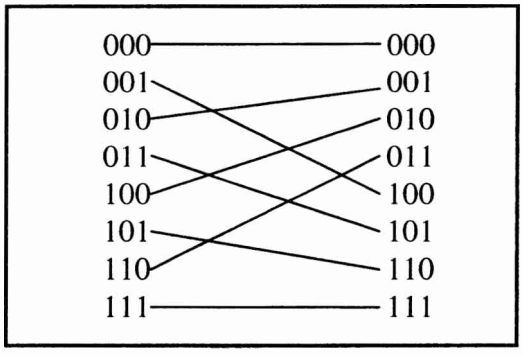


Figura 5.12: Permutación Baseline.

En principio, también se podrían definir variaciones de esta permutación, tal como la permutación sub-baseline k-ésima,

$$BL_k(x) = \{b_n, ..., b_{k+1}, b_1, b_k..., b_2\} \tag{5.12}$$

5.6 Redes Dinámicas Multietapa

En esta sección se verán algunas de las redes dinámicas multietapa más significativas y se detallarán sus características básicas. En general, las redes multietapa se diferencian en su arquitectura por la utilización de puntos de conexión de tipo cross-bar general o de tipo cross-bar de dos entradas y de dos salidas. En el primer caso se denominan redes basadas en cross-

bar (cross-bar switch-based) y en el segundo caso se denominan redes basadas en celdas (cell-based).

5.6.1 Redes de Clos

Las redes de Clos se originaron con el fin de obtener una red dinámica no bloqueante como la red cross-bar, pero con menor cantidad de puntos de conexión. Las redes de Clos tienen tres o más etapas. La Fig. 5.13 muestra de forma esquemática este tipo de redes con tres etapas, para N entradas y M salidas.

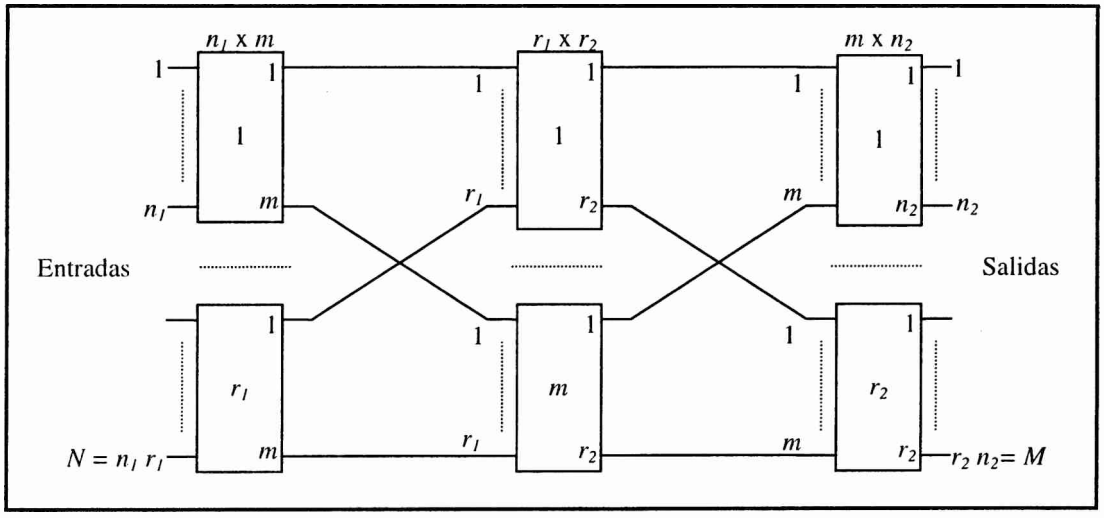


Figura 5.13: Red de Clos de Tres Etapas.

Para las redes de Clos de tres etapas (Fig. 5.13), las N entradas se dividen en r_1 grupos, y cada uno de estos grupos de entradas se asocian a r_1 puntos de conexión de tipo cross-bar, que forman la primera etapa de la red. Cada grupo contiene n_1 elementos, y por lo tanto, cada punto de conexión de la primera etapa es de tipo cross-bar con n_1 entradas y m salidas. La segunda etapa de las redes de Clos se compone de m puntos de conexión de tipo cross-bar, cada uno de ellos con r_1 entradas y r_2 salidas. Finalmente, la tercera etapa de las redes de Clos tienen r_2 puntos de conexión de m entradas y n_2 salidas, donde $r_2 n_2 = M$, que es la cantidad de salidas de la red.

Las redes de Clos de tres etapas están totalmente definidas por los parámetros n_1, n_2, r_1, r_2 y m. Se ha demostrado que las redes de Clos de tres etapas son no bloqueantes si se satisface que

$$m \geq n_2 + n_1 - 1 \tag{5.13}$$

Si la cantidad de entradas y salidas de la red es la misma, entonces $r_1 n_1 = r_2 n_2$, y la red es no bloqueante si se cumple que

$$m \geq 2n - 1 \tag{5.14}$$

Las redes de Clos son multietapa y basadas en puntos de conexión de tipo cross-bar.

Pueden ser bloqueantes o no, dependiendo del cumplimiento de la Ec. (5.13) o la Ec. (5.14) según sea el caso.

Las redes de Clos de cinco etapas se construyen reemplazando la segunda etapa de las redes de tres por una red de Clos de tres etapas. Esta nueva red de Clos deberá ser de r_1 m entradas y r_2 m salidas. De esta forma, se pueden obtener redes de Clos con cinco, siete, nueve, etc. etapas.

5.6.2 Redes Baseline

Las redes baseline son un ejemplo de las redes basadas en celdas. Cada punto de conexión (celda) es de tipo cross-bar con dos entradas y dos salidas. Normalmente cada una de estas celdas proveen dos estados posibles, como lo muestra esquemáticamente la Fig. 5.14: conexión directa de las entradas a las salidas o intercambio de entradas a salidas. Las redes baseline son también un ejemplo de las redes en las que el patrón de interconexión entre las distintas etapas varía. En este caso, los enlaces de salida de una etapa son los enlaces de entrada de la siguiente, y por lo tanto se establece una permutación distinta entre los enlaces para cada etapa.

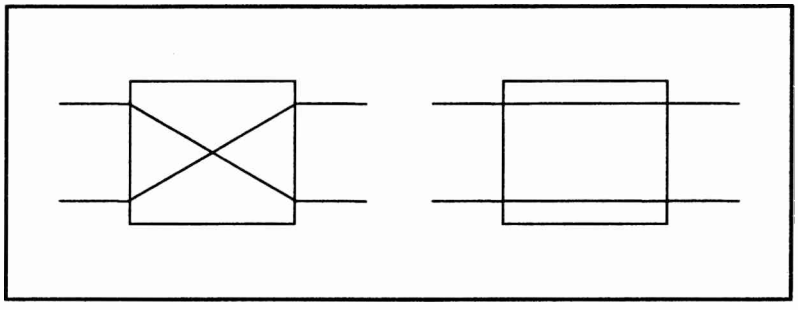


Figura 5.14: Celda de Dos entradas y Dos Salidas.

La Fig. 5.15 muestra una red baseline de tres etapas, donde el patrón de interconexión de los enlaces entre la primera y la segunda etapa se corresponde con la permutación baseline que se definió en una de las secciones anteriores.

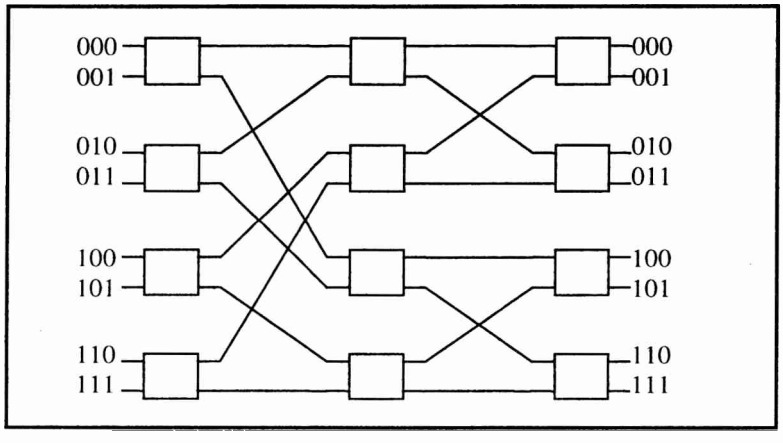


Figura 5.15: Red Baseline de Tres Etapas.

Los enlaces entre la segunda y la tercera etapa de la Fig. 5.15 se podrían ver como dos conjuntos de cuatro enlaces conectados. En cada conjunto se sigue la misma permutación baseline, pero ahora con cuatro entradas y cuatro salidas. Los primeros cuatro enlaces de salida de la segunda etapa son considerados como el primer conjunto de entradas, y los últimos cuatro enlaces de salida de la segunda etapa forman el segundo conjunto de cuatro entradas.

Otra de las características a tener en cuenta de las redes baseline es que se puede implementar en forma directa un algoritmo de ruteo de información basado en la dirección destino. El algoritmo es incluido en la clase de los *destination tag self-routing*, y utiliza la representación binaria del destino de la información más la etapa en la que se ubica el punto de conexión para decidir qué tipo de conexión es necesaria en la celda. Cada bit de la dirección destino es utilizado en una etapa distinta, y cada etapa divide todos los destinos posibles en dos partes iguales. La primera etapa, divide el total de destinos en dos mitades, y el bit más significativo de la dirección es utilizado para elegir hacia cuál de las dos mitades se debe dirigir la información. En la segunda etapa, los destinos vuelven a dividirse en dos, y por lo tanto, se debe elegir la cuarta parte a la que pertenece el destino. El segundo bit más significativo es utilizado en esta tarea. El proceso se repite en tantas etapas como sean necesarias, y se llegará al destino. De esta forma, las celdas tienen control distribuido y el control de cada celda no tiene complicaciones de implementación en hardware. Si se considera N igual a la cantidad de entradas y la cantidad de entradas es igual a la cantidad de salidas, entonces la cantidad de etapas necesarias es $\log_2(N)$. Cada etapa tiene $N/2$ celdas.

La red baseline, como la mayoría de las redes basadas en celdas es bloqueante. La razón es que habiendo s celdas, cada una con 2 estados posibles, entonces la red de interconexión tendrá 2^s estados posibles en total. Si la cantidad de entradas y salidas es N, entonces $N!$ es la cantidad de combinaciones (conexiones) posibles. Normalmente $N!$ es mucho mayor que 2^s. El crecimiento de la cantidad de conexiones necesarias crece en forma factorial con respecto a la cantidad de elementos a conectar. En el ejemplo de la red que se muestra en la Fig. 5.15, si la entrada 000 está conectada con cualquier salida 0xx, la entrada 001 no podrá conectarse con ninguna de las otras tres salidas de la forma 0yy. En este caso, para la red de la Fig. 5.15, la cantidad de combinaciones posibles es 8!, que es mucho mayor que 2^{12}.

5.6.3 Redes Omega

Las redes Omega se basan en la conexión de las etapas siguiendo el patrón de la permutación perfect shuffle, tal como se la describió en una de las secciones anteriores. Como las redes baseline, son redes basadas en celdas de dos entradas y dos salidas, con dos estados posibles.

La Fig. 5.16 muestra una red Omega de tres etapas, que conecta ocho entradas con ocho salidas. Al igual que con las redes baseline, en las redes Omega se puede implementar un algoritmo de ruteo de la información de la clase *destination tag self-routing*, que utilizan la dirección destino de la información con representación binaria. En este caso, el bit menos significativo de la dirección destino es analizado en la primera etapa para decidir si la información saldrá por el enlace superior o inferior de la celda. En la segunda etapa, el

segundo bit más significativo es utilizado para definir cuál de los dos enlaces de salida se utilizará para la transmisión de la información.

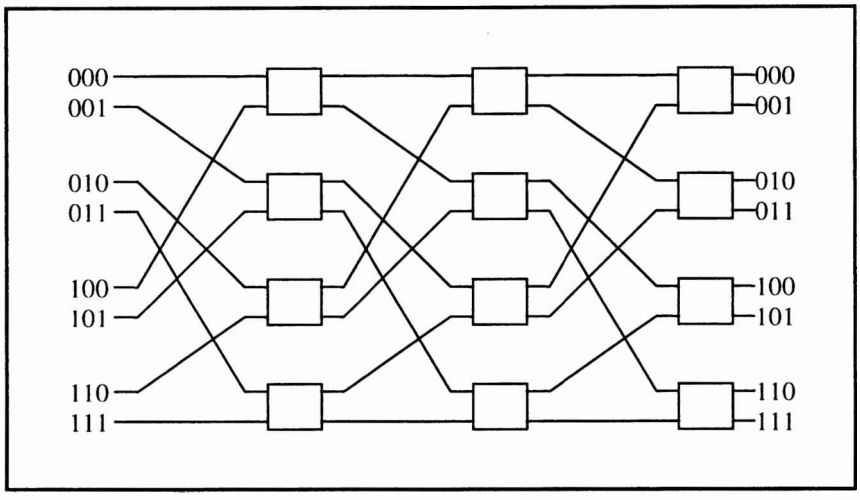


Figura 5.16: Red Omega de Tres Etapas.

Las redes Omega son bloqueantes. Por ejemplo, en la Fig. 5.16, los nodos 000 y 100 no se pueden comunicar a la vez con dos nodos cuya dirección tenga un uno en el bit menos significativo (xx1). En este caso, la primera celda de la primera etapa de la red se convierte en un cuello de botella donde estos requerimientos de conexión se deben secuencializar. Lo mismo puede suceder en las demás celdas.

5.7 Redes Estáticas

Las redes estáticas son las más difundidas entre las máquinas paralelas con arquitectura MIMD de memoria distribuida. Cada nodo tiene conexión directa solamente con un subconjunto de los demás nodos. Los enlaces de comunicación no pueden ser redirigidos.

Lo más usual en las redes estáticas es considerar que cada nodo a comunicar es un procesador, sea en una arquitectura MIMD o también en una arquitectura SIMD. También es muy frecuente encontrar que todos los procesadores tienen la misma cantidad de enlaces. La cantidad de enlaces define el "vecindario" de un elemento a comunicar, es decir los elementos que están directamente comunicados con él. Como se ha afirmado al principio de este capítulo las conexiones exhaustivas entre todos los nodos no son económicamente viables.

En [Hoc88] se realiza una comparación de algunas redes estáticas con la red cross-bar. Esta comparación se basa en la capacidad de comunicación de la red estática, es decir qué conexiones provee y qué conexiones no provee en forma directa con respecto a la red cross-bar.

Una de las formas más utilizadas en la descripción de las redes estáticas es la de clasificarlas por dimensiones. Así, por ejemplo, se definen las redes estáticas como *k-ary n-cube*, con n dimensiones y k elementos en cada dimensión. En principio, se seguirá esta forma de descripción, pero también es posible conectar los nodos de muchas de las formas que se

han definido en las redes estáticas. Por ejemplo, los nodos se podrían conectar de acuerdo a las permutaciones butterfly, o combinando la permutación butterfly y la permutación intercambio. Cualquiera sea la forma de interconectar los nodos, es muy importante que no se produzcan subconjuntos sin comunicación posible. Si así fuera, los nodos de cada subconjunto quedarían sin la posibilidad de interactuar con los nodos que se ubican en los demás subconjuntos.

Aunque la cantidad de nodos conectados en forma directa es limitada, nunca se descarta la comunicación con los demás nodos. Es por esta razón que en [Hoc88], por ejemplo, no se distinguen las redes entre dinámicas o estáticas. Se está considerando que cada nodo pueda ser a la vez un destino de información o un punto de conexión intermedio, tal como en las redes dinámicas.

5.7.1 Redes Unidimensionales

Las redes unidimensionales más clásicas son el arreglo lineal y el anillo, como se los muestra en la Fig. 5.17. La única variación que agrega el anillo con respecto al arreglo lineal es la conexión directa entre los extremos. En el arreglo lineal, todos los nodos tienen dos enlaces, excepto los que se ubican en los extremos. En ambos casos, con un enlace bidireccional o con dos enlaces unidimensionales (en sentido contrario) entre cada par de nodos, se puede enviar información desde un nodo hacia cualquier otro.

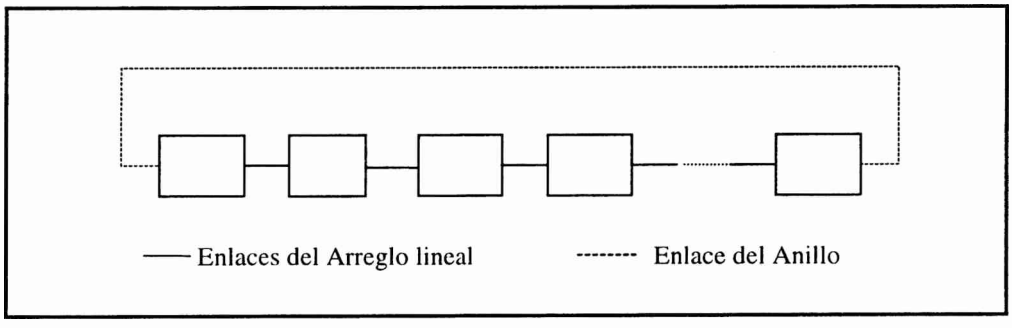


Figura 5.17: Arreglo Lineal.

En general, agregar el enlace a un arreglo lineal para formar un anillo no es decisivo en el costo ni en la operación de la red, pero sí hace más flexible la comunicación en la red.

En términos comparativos con la red cross-bar, un anillo con cuatro elementos a comunicar y con enlaces bidireccionales, tiene los puntos de conexión que se muestran en la Fig. 5.18 [Hoc88]. Los puntos de conexión marcados en la Fig. 5.18 son los puntos de conexión que posee el anillo con respecto a la red cross-bar. Desde este punto de vista, el anillo es una red cross-bar incompleta. A medida que la cantidad de nodos crece, la cantidad de puntos de conexión crece en forma lineal y no cuadrática, como en el caso de la red cross-bar. Las redes estáticas incompletas también se pueden representar por matrices de nxm elementos, donde n es la cantidad de nodos de entrada de la red y m es la cantidad de nodos de salida. Cada posición (i, j) de dicha matriz representa un enlace que comunica al nodo de entrada i con el nodo de entrada j. De acuerdo al valor (puede ser un valor lógico) de la posición (i, j) el nodo i está conectado al nodo j o no.

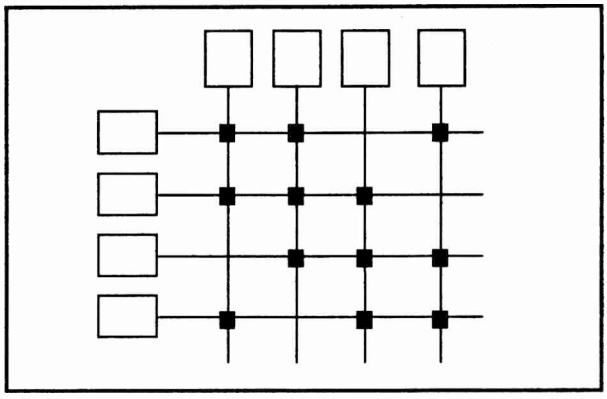


Figura 5.18: Comparación de la Red Cross-Bar con la Red Anillo.

Siguiendo la definición de redes *k*-ary *n*-cube, la red de la Fig. 5.17 es *k*-ary 1-cube, donde *k* es la cantidad de elementos del anillo.

5.7.2 Redes Bidimensionales

Las redes bidimensionales más comunes son las de tipo arreglo malla, donde cada elemento tiene cuatro enlaces que lo conectan con sus vecinos en las dos dimensiones. La Fig. 5.19 muestra una red de este tipo, también denominada *wraparound mesh network* [Dec89],

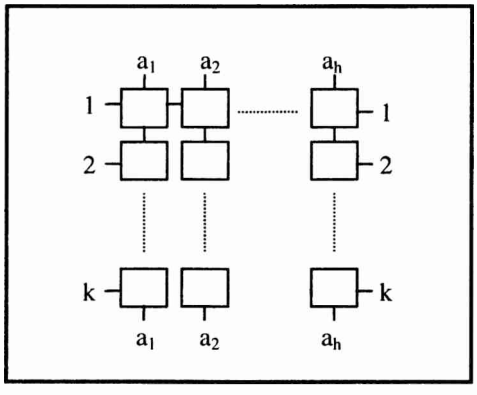


Figura 5.19: Red de Interconexión Bidimensional Mesh.

donde los enlaces 1, 2, ..., k y a_1, a_2, a_h de cada extremo se conectan entre sí.

Otras redes bidimensionales definen la disposición de los nodos a comunicar en forma de estrella, árbol, hexagonal (también denominada X), tal como se las muestra en la Fig. 5.20. En todos los casos, las redes de la Fig. 5.20 se pueden asociar a algún tipo de procesamiento en particular y de hecho han sido propuestas inicialmente con el objetivo de resolver algún tipo de aplicaciones. Cualquiera sea el origen de una red de interconexión, sule ser útil tenerla en mente para su posible utilización o, por lo menos comparación con otras posibilidades. Aunque una red de interconexión estática parezca muy relacionada con una aplicación en particular, puede ser muy adecuada para otras aplicaciones y, por lo tanto, no conviene descartar su utilización de antemano. De hecho, la idea básica de describir y conocer distintas

redes apunta a la optimización en la búsqueda de la más adecuada para un problema en particular.

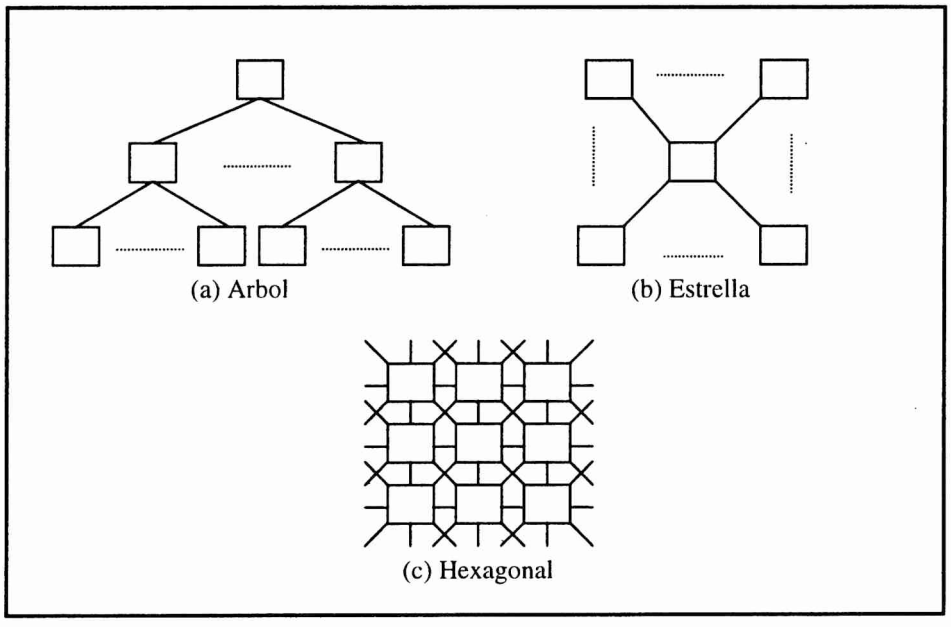


Figura 5.20: Configuraciones de Redes Estáticas Bidimensionales.

Cada red estática puede ser definida de distintas formas, por ejemplo en [Dec89] la red bidimensional wraparound mesh se define por medio de cuatro funciones de interconexión que se asocian a cada nodo a conectar:

$$\begin{aligned} M_{+1}(x) &= x + 1 \text{ Mod } N \\ M_{-1}(x) &= x - 1 \text{ Mod } N \\ M_{+n}(x) &= x + n \text{ Mod } N \\ M_{-n}(x) &= x - n \text{ Mod } N \end{aligned} \qquad (5.15)$$

donde N es la cantidad de elementos a comunicar y es una potencia de dos, y $n = \sqrt{N}$. Los elementos se numeran desde 0 hasta N-1, y si se disponen como una matriz de NxN, se comienzan a numerar desde la esquina superior izquierda hacia la derecha y por filas.

En las redes hexagonales (o redes X), tal como se las muestra en la Fig. 5.20 (c), también es usual interconectar los nodos de los extremos del hexágono.

5.7.3 Redes Hipercubo de Orden *n*

Las redes definidas en n-dimensiones se relacionan en forma directa con la cantidad de dígitos binarios que se utilizan para numerar los nodos a comunicar. Cada nodo está comunicado en forma directa con todos los demás nodos cuya dirección difiera en un sólo bit.

La cantidad de dimensiones está dada por la cantidad de bits que sean necesarios para numerar los nodos. Por ejemplo, en una red cúbica (hipercubo de orden 3), el nodo 000 estará

comunicado en forma directa con los nodos 001, 010, y 100. También se puede afirmar que la cantidad de enlaces de un nodo es igual a la cantidad de dimensiones de la red.

Las redes hipercubo de orden *n* también tienen un algoritmo de ruteo de la información basado en la dirección del nodo destino. En cada nodo se compara la dirección destino con la dirección del nodo, si las direcciones son iguales, entonces la información ha llegado al nodo destino. Si las direcciones son distintas, entonces se debe redirigir la información hacia otro nodo. El criterio de elección del nodo al cual se reenvía la información se basa en los valores de los dígitos binarios del destino y de los vecinos. Suponiendo que la cantidad de dígitos binarios iguales entre la dirección del destino de la información y la dirección del nodo es k, entonces se reenvía la información a uno de los nodos vecinos cuya dirección sea igual en k+1 dígitos binarios. Si se debe enviar, por ejemplo, información desde el nodo 000 al nodo 101, desde el nodo 000 se puede enviar al 100 o al 001. Si se supone que se ha enviado al 100, desde allí es reenviada al 101. Si, por el contrario, se envió la información desde el 000 al 001, desde allí se puede reenviar al 101. En ambos casos se llega al nodo destino.

La Fig. 5.21 muestra un hipercubo de orden 3 [Hwa84], sobre el cual se pueden verificar los ejemplos anteriores de conexión entre los nodos y de ruteo de la información.

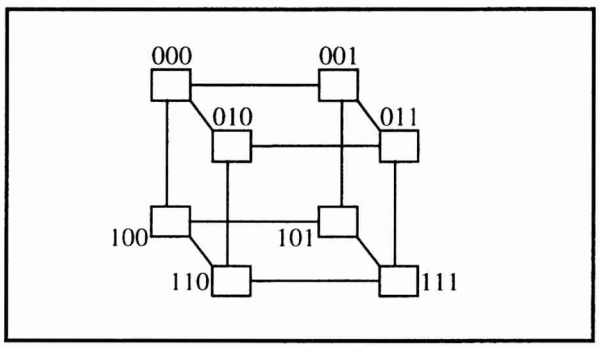


Figura 5.21: Hipercubo de Orden 3.

5.7.4 Evaluación de Redes Estáticas

Según las definiciones y ejemplos que se han visto, hay formas muy variadas de redes estáticas. La cantidad de enlaces y la forma en que se conectan los nodos pueden ser muy dispares entre sí. Es por esta razón que se necesita algún método de comparación de las redes estáticas o al menos algún método de caracterización de cada red, más allá de la forma en que se conectan los nodos.

Una primera medida que puede ser útil para evaluar una red es el *grado* de cada nodo de la red, que se define como la cantidad de enlaces que tiene cada nodo. En principio, este índice se calcula por nodo, pero se puede calcular el promedio con el fin de caracterizar a la red. El grado de un nodo tiene relación directa con la complejidad del nodo y, por lo tanto, con el costo del nodo.

Uno de los factores críticos en una red de interconexión es la mínima cantidad de enlaces que se deben utilizar para transferir información entre dos nodos. Esta medida es

conocida como distancia entre los nodos. La cantidad de nodos o enlaces intermedios es de interés porque tiene relación directa con el retardo en la transmisión de la información y con la posibilidad de colisiones. La distancia promedio de una red se define como [Wil91] [Agr86]

$$AvgD = \frac{\sum_{d=0}^{Max} dN_d}{N-1} \tag{5.16}$$

donde N es la cantidad de nodos, Max es la distancia máxima para interconectar dos nodos, y N_d es la cantidad de nodos separados por d enlaces. Para una red en particular, se deberían calcular todas las conexiones posibles entre todos los pares de nodos.

Otro de los índices también relacionado con las distancias entre los nodos, es el *diámetro* de la red. Se lo define como la distancia máxima que separa dos nodos teniendo en cuenta todos los caminos mínimos entre los nodos. Escrito de otra forma,

$$D = \max_{i,j \in N} (\min_{p \in Pij} \text{length}(p)) \tag{5.17}$$

donde N es el conjunto de nodos, y Pij es el conjunto de caminos entre los nodos i y j.

6. Algoritmos Paralelos Clásicos

En este capítulo se analizarán algunos problemas simples de tratamiento paralelo de datos, sobre los cuales se discutirán conceptos teóricos vistos en los capítulos anteriores. A medida que sea necesario, también se introducirán conceptos que clarifiquen ideas sobre los algoritmos paralelos, las arquitecturas paralelas a utilizar y/o el análisis de rendimiento que se realice.

En el primer problema presentado, todo el análisis de rendimiento seguirá las ideas del Ap. A con respecto a la cantidad de pasos de ejecución y su relación con el tiempo de ejecución. El Ap. A presenta una introducción al análisis de algoritmos, y en particular al análisis de los algoritmos paralelos. También se introduce la forma de calcular los índices de rendimiento más utilizados, tales como el factor de Speed-Up.

En los sistemas SIMD y en las arquitecturas sincrónicas, los pasos de ejecución tienen relación directa con la cantidad de *ciclos de reloj*, dada la ejecución de instrucciones de modo *lock-step* (lock-step operation mode). Es por esta razón que el análisis de rendimiento para el segundo problema y los siguientes se hace en términos de ciclos de reloj. En cualquier caso, tanto el orden de magnitud del tiempo de ejecución de los algoritmos paralelos como los índices de rendimiento que se presentan quedan completamente especificados y explicados en cada problema.

Los primeros problemas son clásicos en el tratamiento de datos organizados en *una dimensión*, o en *vectores*. Los últimos problemas tratan sobre el procesamiento clásico de datos organizados en *dos dimensiones*, o en *matrices*. Como se explicó al principio del libro, no hay un tratamiento exhaustivo de todos los problemas de procesamiento de datos. En este capítulo lo que se intenta es presentar las bases sobre las cuales se apoya el procesamiento de datos organizados en una o en dos dimensiones. Como una sección intermedia se agrega un breve comentario acerca de la granularidad de los algoritmos y/o de las arquitecturas.

6.1 Suma de los Elementos de un Vector

El primer problema es de tipo aritmético y consiste en realizar la suma de los elementos de un vector, es decir,

$$Sum = x_1 + x_2 + \ldots + x_n \tag{6.1}$$

o, lo que es lo mismo,

$$Sum = \sum_{i=1}^{n} x_i \tag{6.2}$$

La resolución de la suma de los elementos de un vector en una computadora monoprocesador es inmediata. Se llevan a cabo las n-1 sumas necesarias de forma secuencial. Esto implica llevar a cabo n-1 pasos de ejecución, donde cada paso de ejecución es una suma. Esquemáticamente, los pasos de ejecución se pueden especificar como

Paso 1: $\sum_1^2 = Sum_{12} = x_1 + x_2$

Paso 2: $\sum_1^3 = Sum_{13} = Sum_{12} + x_3$

... ...

Paso n-1: $Sum = \sum_1^n = Sum_{1n} = Sum_{1(n-1)} + x_n$

En términos de tiempo de ejecución (ver Ap. A), la cantidad de operaciones a realizar para obtener la suma es $O(n)$. Más específicamente, la función que proporciona la cantidad de pasos de ejecución dependiente del tamaño del problema es $g_1(n) = n$-1, y la función que determina el *orden de complejidad* es $f(n) = n$. El tamaño del problema es, en este caso, la cantidad de elementos a sumar, la dimensión del vector.

6.1.1 Utilizando Dos procesadores

En el caso de disponer de $N = 2$ procesadores para realizar la suma de los elementos de un vector, la tarea se puede dividir. Un procesador puede llevar a cabo la suma parcial de la primera mitad de las sumas, y el otro la suma parcial de la segunda mitad. Es decir que el procesamiento se divide en los dos procesadores de forma tal que

Procesador P_1: $\sum_1^{n/2} = Sum_{1(n/2)} = x_1 + ... + x_{n/2}$

Procesador P_2: $\sum_{n/2+1}^{n} = Sum_{(n/2+1)n} = x_{n/2+1} + ... + x_n$

Esquemáticamente, se podría describir esta división del procesamiento en dos procesadores como lo muestra la Fig. 6.1.

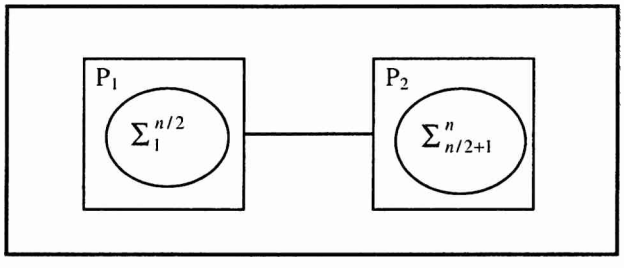


Figura 6.1: Sumas Parciales de los Elementos de un Vector.

donde $\sum_1^{n/2} = Sum_{1(n/2)} = x_1 + ... + x_{n/2}$, y $\sum_{n/2+1}^{n} = Sum_{(n/2+1)n} = x_{n/2+1} + ... + x_n$. Pero esto no resuelve el problema, porque se necesita $Sum = Sum_{1(n/2)} + Sum_{(n/2+1)n}$. Para llegar a esto, los procesadores se deberían comunicar de alguna manera para que en uno de ellos se realice la última suma necesaria para el resultado final. Esta comunicación se puede llevar a cabo mediante la utilización de memoria compartida común o un enlace de comunicación, y está representada en la Fig. 6.1 con la línea que une ambos procesadores. Independientemente de la forma en que se transmitan los resultados parciales de un procesador a otro, se debe llevar a cabo la última suma en uno de los procesadores.

La cantidad de sumas que se llevan a cabo con dos procesadores es exactamente la misma, n-1 sumas, pero la diferencia se encuentra en:

- Cantidad de operaciones.

- Cantidad de pasos de ejecución.

Y ambas cantidades afectan el tiempo de ejecución. Por lo tanto el tiempo de ejecución será también diferente. La cantidad de operaciones varía porque se debe agregar la comunicación entre los procesadores que originalmente no estaba. La cantidad de pasos de ejecución debe tener en cuenta que ahora hay operaciones que se llevan a cabo de forma simultánea porque los dos procesadores pueden realizar las sumas parciales al mismo tiempo.

La cantidad de pasos de ejecución para resolver ambas sumas parciales se representa con la función $g_{1/2}(n) = n/2\text{-}1$, porque con dos procesadores se pueden ejecutar dos sumas al mismo tiempo (una en cada procesador). Una vez que se tienen las sumas parciales, se necesitan dos pasos de ejecución más, que son secuenciales: (1) la comunicación de una suma parcial desde un procesador hacia el otro, y (2) la última suma para obtener el resultado final. De esta manera, la cantidad total de pasos de ejecución está representada por $g_2(n) = n/2+1$.

El tiempo de ejecución secuencial, dado por la cantidad de pasos de ejecución $g_1(n) = n\text{-}1$ y el tiempo de ejecución paralelo calculado en función de dos procesadores interconectados dado por $g_2(n) = n/2+1$, son $O(n)$. Sin embargo, el tiempo real de ejecución será diferente. El factor de Speed-Up, por ejemplo, está dado por

$$S_2 = (n\text{-}1) / (n/2+1) \xrightarrow[n\to\infty]{} 2 \qquad (6.3)$$

donde S_2 denota el factor de Speed-Up que se obtiene utilizando dos procesadores. La tendencia de la Ec. (6.3) indica que cuanto más grande sea la cantidad de elementos del vector (n), el factor de Speed-Up se acercará más a 2. Por ejemplo, si la dimensión del vector está dada por $n = 128$ implica $S_2 \cong 1.95$, y si $n = 1024$ implica $S_2 \cong 1.99$.

6.1.2 Más Procesadores

Agregar procesadores puede, en principio, mejorar el tiempo de ejecución, o lo que es lo mismo, aumentar el factor de Speed-Up. De la misma manera que al utilizar $N = 2$ procesadores se llega a que la cantidad de pasos de ejecución es $g_2(n) = n/2+1$, al utilizar $N = 3$ procesadores se llega a que la cantidad de pasos de ejecución está dada por $g_3(n) = n/3+3$. En este caso, los pasos de ejecución que se consideran realizados de forma secuencial son

- $n/3\text{-}1$ sumas en cada uno de los tres procesadores. Se obtienen $\sum_1^{n/3} = Sum_{1(n/3)} = x_1 + ... + x_{n/3}$ en el procesador P_1, $\sum_{n/3+1}^{2n/3} = Sum_{(n/3+1)2n/3} = x_{n/3+1} + ... + x_{2n/3}$ en el procesador P_2, y $\sum_{2n/3+1}^{n} = Sum_{(2n/3+1)n} = x_{2n/3+1} + ... + x_n$ en el procesador P_3.
- 1 comunicación y una suma (ejecutadas en secuencia) para obtener en el procesador P_1 o en el procesador P_2: $\sum_1^{2n/3} = Sum_{1(2n/3)} = x_1 + ... + x_{2n/3}$.
- 1 comunicación y una suma (ejecutadas en secuencia) para obtener el resultado final.

A medida que se agregan procesadores, la cantidad de comunicaciones necesarias aumenta, porque los datos y las sumas parciales están más repartidas entre ellos. En principio, la cantidad máxima de procesadores que tendría sentido utilizar está dada por la cantidad mínima de sumas que se pueden llevar a cabo en cada uno de ellos. Si se considera que cada procesador puede hacer, como mínimo, una suma entre dos elementos de un vector y en este

caso se utilizan $N = n/2$ procesadores. En el primer paso de ejecución cada procesador lleva a cabo una suma de la siguiente manera:

Procesador P_1:	$\sum_1^2 = Sum_{12} = x_1 + x_2$
Procesador P_2:	$\sum_3^4 = Sum_{34} = x_3 + x_4$
...	...
Procesador P_i:	$\sum_{2i-1}^{2i} = Sum_{(2i-1)2i} = x_{2i-1} + x_{2i}$
...	...
Procesador $P_{n/2}$:	$\sum_{n-1}^{n} = Sum_{(n-1)n} = x_{n-1} + x_n$

En el segundo paso de ejecución, se deberían comunicar las primeras sumas parciales, por ejemplo

Procesador P_1:	Recibe $\sum_3^4$ de P_2
...	...
Procesador P_i:	Recibe $\sum_{2i+1}^{2i+2}$ de P_{i+1}
...	...
Procesador $P_{n/2-1}$:	Recibe $\sum_{n-1}^{n}$ de $P_{n/2}$

En el tercer paso de ejecución se harían las segundas sumas parciales, y así se continúa hasta el último paso de ejecución en el que se llevaría a cabo la suma $\sum_1^{n/2} + \sum_{n/2+1}^{n}$.

Esquemáticamente, las sumas parciales y las comunicaciones necesarias para sumar los elementos de un vector pueden verse como en la Fig. 6.2. Se tiene resuelto el problema porque el resultado final es el mismo, el de la Ec. (6.2). Sin embargo, la forma de calcular y ejecutar las operaciones en una computadora paralela tal como se ha presentado es radicalmente distinta de la manera secuencial en que se lleva a cabo en una computadora monoprocesador.

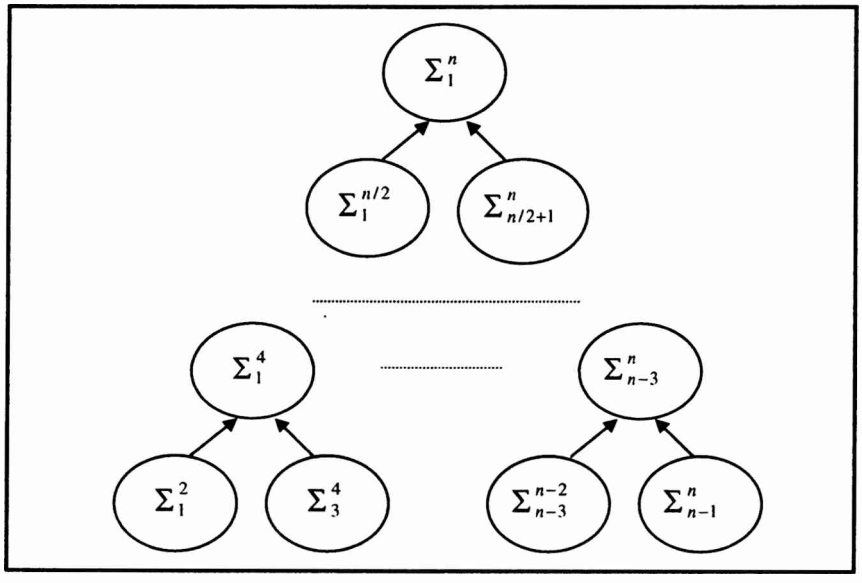


Figura 6.2: Sumas Parciales y Comunicaciones.

6.1.3 Consideraciones de Rendimiento

A medida que se aumenta la cantidad de procesadores, es de esperar que disminuya el tiempo de ejecución. En esta sección se analizará cuál es la relación entre la cantidad de procesadores y el tiempo de ejecución. Este análisis se lleva a cabo de acuerdo al estudio del tiempo de ejecución de los algoritmos dependiente de la cantidad de procesadores, tal como se lo presenta en el Ap. A.

Retomando el ejemplo de la suma de los elementos de un vector, es claro que para n (cantidad de elementos del vector) potencia de dos, la cantidad de pasos de ejecución correspondientes a operaciones de suma es $\log_2(n)$. De la misma manera, la cantidad de comunicaciones es $\log_2(n)$-1. En cada paso de suma se obtienen sumas parciales con el doble de elementos del vector sumados que en el paso de suma anterior. Esto se debe a que:

- En el primer paso de suma se tienen sumas parciales de dos elementos del vector.
- En el segundo paso de suma se tienen sumas parciales de cuatro elementos del vector.
- En el paso i-ésimo de suma se tienen sumas parciales de 2^i elementos del vector.
- Finalmente, en el paso $\log_2(n)$ de suma se tiene la suma de los n elementos del vector.

En cada paso de suma la cantidad de operaciones simultáneas se reduce a la mitad, y por lo tanto es posible realizar todas las operaciones de cada paso de suma simultáneamente, porque siempre hay procesadores disponibles. Esto es así porque se asumió la utilización de $P_{n/2}$ procesadores ($N = n/2$), que es la cantidad de sumas simultáneas necesarias en el primer paso de suma, y la cantidad de sumas simultáneas se reduce a la mitad cada vez que se avanza.

Por otro lado, los pasos de ejecución que corresponden a las comunicaciones son necesarios para avanzar de un paso de suma al siguiente. Como hay $\log_2(n)$ pasos de suma, son necesarios $\log_2(n)$-1 pasos de comunicaciones para llegar a la última suma en la que se obtiene el resultado final. La cantidad de datos que se pueden comunicar de manera simultánea se va reduciendo a la mitad a medida que se avanza de un paso de comunicación al siguiente, de la misma manera que se reduce la cantidad de operaciones de sumas simultáneas en cada paso de suma.

Finalmente, a cada paso de suma le sigue un paso de comunicación de manera secuencial, porque no se pueden transmitir los resultados de las sumas parciales antes de que se lleven a cabo las sumas correspondientes. De esta manera se intercalan de forma secuencial $\log_2(n)$ pasos de suma con $\log_2(n)$-1 pasos de comunicación de datos. Por lo tanto, la cantidad de pasos que corresponde al tiempo de ejecución paralelo utilizando $n/2$ procesadores está dada por $g_{n/2}(n) = \log_2(n)+\log_2(n)-1 = 2\log_2(n)-1$. Claramente, $g_{n/2}(n)$ es $O(\log_2(n))$, que además coincide con una de las funciones de complejidad más utilizadas en la bibliografía, tal como se lo menciona en el Ap. A.

El factor de Speed-Up que se obtiene con la resolución paralela utilizando $n/2$ procesadores con respecto a la resolución secuencial, tal como se ha analizado, está definido por

$$S_{n/2} = (n-1) / (2\log_2(n)-1) \xrightarrow[n\to\infty]{} \infty \qquad (6.4)$$

Comparando la Ec. (6.3) con la Ec. (6.4), las diferencias y la conclusión son evidentes. Si se determina la cantidad de procesadores en función de la cantidad de elementos a procesar

($n/2$), según la Ec. (6.4), teóricamente el factor de Speed-Up para este problema crece indefinidamente junto con la cantidad de datos a procesar. Por ejemplo, si $n = 128 = 2^7$ implica el factor de Speed-Up $S_{64} \cong 9.77$, y si $n = 1024 = 2^{10}$ implica $S_{512} \cong 53.84$. Aunque el factor de Speed-Up crezca en función de la cantidad de datos a procesar, se deben considerar desde el principio otros factores como la complejidad del hardware que determina su viabilidad y la relación costo/beneficio del sistema de cómputo paralelo.

Es interesante considerar el caso en el cual la cantidad de procesadores disponibles sea menor de la que se ha definido, es decir $N < n/2$. Si, por ejemplo $N = n/4$, se tendría un caso típico de bisección del algoritmo: una estructura de N procesadores podría sumar N números, obteniendo la suma parcial de una mitad del vector de números de entrada. Se debería repetir este proceso de suma para tener dos resultados parciales (las sumas parciales de dos mitades del vector de entrada). Todavía faltaría una suma para llegar al resultado final tal como el de la Ec. (6.2).

6.1.4 Consideraciones de Hardware

Varias características de hardware se deben tener en cuenta a la hora de evaluar una solución propuesta, independientemente de que desde el punto de vista algorítmico el análisis resulte satisfactorio. Además, el estudio de la variación del hardware permite tener una visión más precisa de lo que sucede en el sistema de cómputo a medida que se avanza en la paralelización de una aplicación.

Una de las primeras características que se puede observar a medida que se avanza en la división de tareas para llevar a cabo la suma de los elementos de un vector es el grado de complejidad e interdependencia de los procesadores. En la solución monoprocesador, el único elemento de procesamiento debe ser capaz de hacer *todo*. Esto implica almacenar los elementos del vector, hacer todas las sumas intermedias y almacenarlas hasta llegar al resultado final. Cuando se decide llevar a cabo la tarea utilizando dos procesadores esta situación cambia. Cada uno de los dos procesadores debe ser capaz de realizar la mitad de las sumas intermedias. Además, se debe agregar un medio de comunicación entre los dos procesadores. Finalmente, cuando se utilizan $n/2$ procesadores, cada uno de ellos debe ser capaz de realizar solamente una suma, pero la red de comunicaciones se hace más compleja y cada procesador depende de los resultados de al menos otro procesador para avanzar en el cómputo.

Al hacer el análisis de rendimiento de la paralelización, además de aumentar la cantidad de procesadores en función del tamaño del problema (n), se asume que todas las comunicaciones se pueden realizar simultáneamente. Con los procesadores se logra la ejecución simultánea de todas las sumas en cada paso de cálculo de sumas parciales. Además, se debe tener hardware suficiente en la red de comunicaciones para transmitir todos los datos en cada paso de comunicación. Tal como se analizó en algunos capítulos anteriores, la interconexión de los procesadores no es un problema que se pueda dejar de lado.

Siguiendo con el ejemplo de la suma de elementos de un vector, se debe decidir cómo transmitir los datos de las sumas parciales entre los procesadores. Si se decide la interconexión de los procesadores siguiendo una arquitectura de memoria compartida, se deben tener en cuenta al menos dos problemas asociados. Por un lado habría que tener como

mínimo la misma cantidad de módulos de memoria que procesadores, y por el otro, la red de interconexión procesadores – memoria debería permitir que todas las transferencias de información de las sumas parciales se puedan llevar a cabo de forma simultánea. Este último requerimiento llevaría a utilizar las redes de interconexión más costosas, tales como las cross-bar, y las redes de interconexión dinámicas. Esto a su vez muy probablemente haría que el sistema de cómputo sea demasiado costoso y, por lo tanto, podría llevar a desechar la utilización de una máquina paralela, que era la idea inicial.

Como alternativa a las redes de interconexión dinámicas, siempre es posible recurrir a las redes estáticas si se conoce el patrón de interconexión entre procesadores (también llamado topología), que necesita el algoritmo. Para el caso que se está tratando, la red que se proponga debería ser capaz de llevar a cabo la transmisión simultánea de todas las sumas parciales de cada paso de comunicación. Esto significa que deben existir los enlaces necesarios entre procesadores para que en todos los pasos de comunicación las transferencias de datos se puedan llevar a cabo simultáneamente. Para el primer paso de comunicación, por ejemplo, se definen las conexiones de forma esquemática tal como lo muestra la Fig. 6.3. En este primer paso de comunicación se transmiten las primeras sumas parciales, que corresponden a la suma de dos elementos consecutivos del vector, es decir $\sum_{2i-1}^{2i} = Sum_{(2i-1)2i} = x_{2i-1} + x_{2i}$.

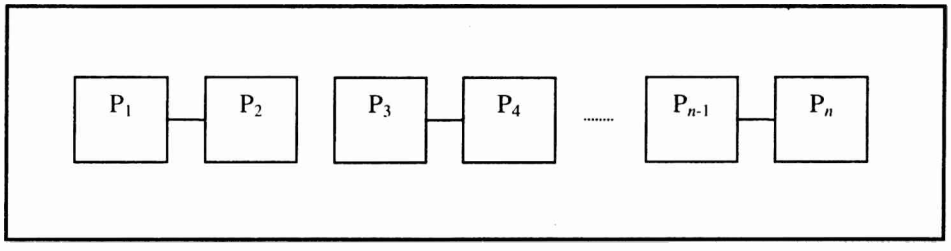


Figura 6.3: Interconexiones para el Primer Paso de Comunicación.

Para el segundo paso de comunicación, en cambio, se definen las conexiones que se muestran de manera esquemática en la Fig. 6.4.

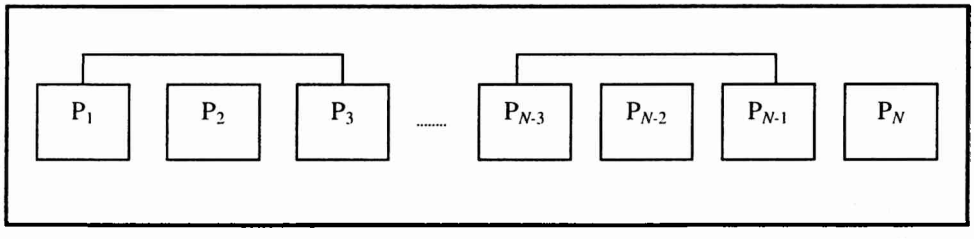


Figura 6.4: Interconexiones para el Segundo Paso de Comunicación.

Los requerimientos de comunicaciones se definen en función del paso que corresponde. Cada paso de comunicación implica reunir en un mismo procesador el resultado de dos sumas parciales para obtener una suma parcial más, con el doble de elementos del vector de entrada sumados. Finalmente, en el último paso se comunica el procesador P_1 con el procesador $P_{N/2+1}$ y con una suma más en el procesador P_1 se obtiene el resultado final, que corresponde a la Ec. (6.2).

Por ejemplo, si se tiene un vector de entrada de 16 elementos, las conexiones

quedarían definidas como lo muestra esquemáticamente la Fig. 6.5.

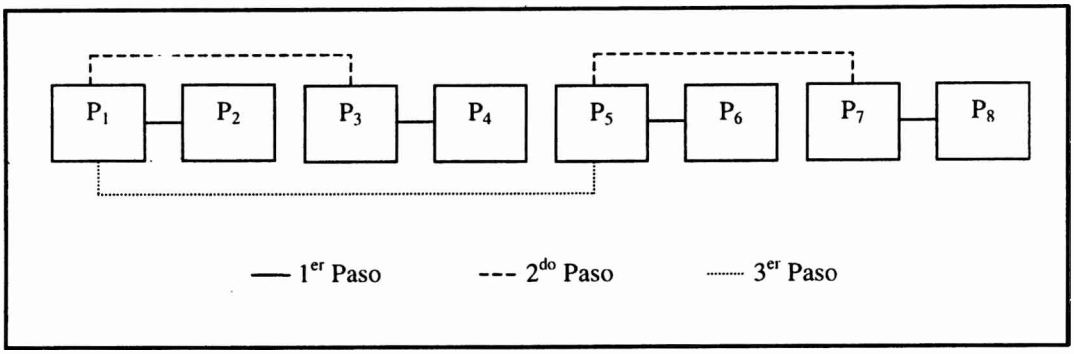


Figura 6.5: Conexiones para Efectuar las Sumas Parciales.

Junto con la paralelización propuesta, queda definido que en cada paso de suma los procesadores que participan activamente son cada vez menos a medida que se avanza en las sumas parciales. Más específicamente, en cada paso de suma la cantidad de procesadores que realizan cómputo efectivo es la mitad con respecto al paso de suma anterior. Quizás esto sea más evidente al analizar la Fig. 6.5, donde en el primer paso de suma intervienen todos los procesadores. En el segundo paso los procesadores P_1, P_3, P_5 y P_7 son los que tienen instrucciones de suma para ejecutar, en el tercer paso los procesadores P_1 y P_5, y en el cuarto y último paso ($\log_2(16) = 4$) solamente el procesador P_1. Esto también denota una realidad común en los algoritmos paralelos: a mayor cantidad de procesadores es más difícil lograr que los algoritmos los utilicen a todos todo el tiempo.

Una última característica sobre el hardware necesario a utilizar resulta de analizar los requerimientos de entrada/salida (E/S o I/O: Input/Output). En el caso de la computadora monoprocesador, alcanza con que los datos lleguen secuencialmente al procesador y por lo tanto a la computadora. En la máquina paralela con dos procesadores se necesita que dos datos lleguen a la vez (uno hacia cada procesador) para utilizar ambos procesadores simultáneamente. En general, a medida que se utilizan más procesadores, se necesita que más datos lleguen simultáneamente a ellos. De esta manera, los requerimientos de entrada al conjunto de los procesadores se hacen mayores y esto afecta a los requerimientos de E/S de toda la máquina paralela.

6.1.5 ¿Aún Más Procesadores?

La Fig. 6.5 de alguna manera indica una red de interconexión estática con estructura de comunicaciones de tipo arborescente. En base a esto, se puede tomar la decisión de construir un *árbol binario lleno* [Aho88] de procesadores tal como lo muestra de forma esquemática la Fig. 6.6. La estructura que se muestra en la figura es un árbol porque es un conjunto de nodos, uno de los cuales se distingue como *raíz*, junto con una relación (también llamada "relación de paternidad"), que impone una estructura jerárquica sobre los nodos. En este caso, los nodos son los procesadores, el nodo raíz es el nodo ubicado en la parte superior de la figura, y la relación de paternidad está indicada por los enlaces de comunicaciones. Los nodos que no tienen hijos se denominan *hojas* del árbol. Además, el árbol de la Fig. 6.6 es *binario* porque cada nodo tiene a lo sumo dos hijo. Se denomina *lleno* porque todos los nodos que no son hojas tienen exactamente dos hijos y todos los nodos ubicados en las hojas tienen la misma

distancia en cantidad de enlaces al nodo raíz. Expresado de otra manera, el árbol binario de la Fig. 6.6 es lleno porque tiene la cantidad máxima posible de nodos.

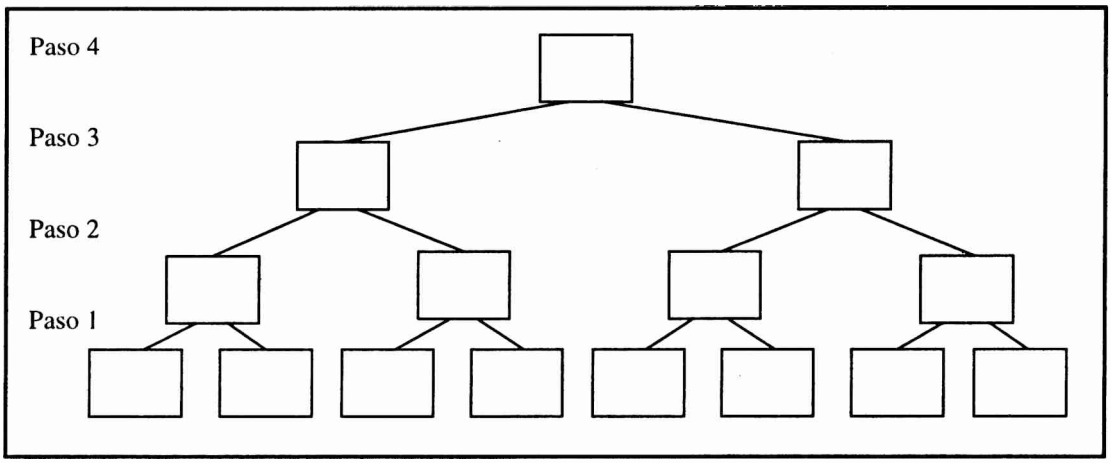


Figura 6.6: Arbol de Procesadores.

Volviendo a la tarea que se debe realizar, los procesadores ubicados como hojas del árbol de la Fig. 6.6 son los que ejecutarán el primer paso de suma. Cada paso de suma subsiguiente implica la utilización de los procesadores cada vez más cercanos al procesador ubicado en la raíz del árbol.

Siguiendo el esquema de la Fig. 6.6 se tiene una cantidad similar de enlaces que en una red de interconexiones estáticas tal como lo muestra el esquema de la Fig. 6.5. Sin embargo, la cantidad de procesadores no es la misma, es mucho mayor. Si, como se comentó, no todos los $n/2$ procesadores se pueden usar en todos los pasos para sumar los elementos de un vector ¿Por qué definir una estructura con mayor cantidad de procesadores? En particular, la cantidad de procesadores que se utilizan para construir el árbol que obtiene la suma de los elementos de un vector es n-1. Siguendo la terminología clásica de las estructuras de tipo árbol binario [Aho88], un árbol binario lleno que ejecute h pasos tales como los que se han definido, es de altura h-1. Además, es demostrable (por inducción) que un árbol binario lleno de altura h-1 tiene 2^h-1 nodos. Recordando que $h = \log_2(n)$, donde n es la cantidad de elementos a sumar, se llega a que el árbol binario definido tiene n-1 procesadores.

El árbol binario de procesadores puede ser muy útil si el problema a resolver consiste en obtener la suma de los elementos de varios vectores. La resolución repetitiva del mismo problema con diferentes datos suele ser algo muy común en el contexto de utilización de computadoras y también de computadoras paralelas. El *nuevo* problema ahora define que para una secuencia de vectores de entrada $v_1, \ldots v_k$, se debe obtener una secuencia de valores escalares de salida $s_1, \ldots, s_k$, tales que

$$s_i = \sum_{j=1}^{n} v_i(j), \text{ para todo } i = 1, \ldots, k \qquad (6.5)$$

¿Cómo llevar a cabo esta tarea en un árbol binario? Se utiliza la idea de una línea de ensamblado, también denominada *pipelining* en el contexto de las arquitecturas de cómputo. Cada paso de suma parcial de elementos del vector constituye una etapa de la línea de ensamblado, es decir, una etapa del pipeline de procesamiento [Hwa93].

Una vez que los procesadores ubicados en las hojas del árbol ejecutan el primer paso de suma del vector v_1, estos procesadores quedan libres. Por lo tanto, pueden utilizarse para ejecutar el primer paso de suma del vector v_2, al mismo tiempo en que se ejecuta el segundo paso de suma del vector v_1. De la misma manera, cuando se ejecuta el tercer paso de suma de los elementos del vector v_1, se puede ejecutar el segundo paso de suma del vector v_2 y el primero de v_3. A medida que el tiempo avanza, todos los procesadores del árbol ejecutarán un paso de suma parcial sobre distintos datos de distintos vectores. Esquemáticamente, para n = 16 y k vectores a procesar, el pipeline podría verse como lo muestra la Fig. 6.7.

Paso de + 4				v_1	v_2	...	v_{k-3}	v_{k-2}	v_{k-1}	v_k
Paso de + 3			v_1	v_2	v_3	...	v_{k-2}	v_{k-1}	v_k	
Paso de + 2		v_1	v_2	v_3	v_4	...	v_{k-1}	v_k		
Paso de + 1	v_1	v_2	v_3	v_4	v_5	...	v_k			
Paso de ejec.	1	2	3	4	5	...	k	k+1	k+2	k+3

Figura 6.7: Pipeline de Pasos de Suma.

Donde "Paso de + i" indica el i-ésimo paso de suma parcial de elementos del vector.

En general, si se utiliza un árbol de altura $\log_2(n)$-1 para la ejecución de $\log_2(n)$ pasos de suma, se obtendrá la suma de los elementos del vector v_k en el paso de suma k+$\log_2(n)$-1. Teniendo en cuenta que cada paso de suma parcial excepto el último implica un paso siguiente de comunicación, la cantidad de pasos de ejecución puede expresarse como $g_{n-1}^{(k)}(n)$ = k+2$\log_2(n)$-2, donde el superíndice (k) indica la cantidad de vectores que se procesan. ¿Qué se ha ganado por utilizar más procesadores? Con respecto a la solución secuencial en una máquina monoprocesador, la cantidad total de pasos de ejecución para obtener las sumas de los elementos de k vectores es $kg_1(n) = k(n-1)$. Por lo tanto, el factor de Speed-Up está dado por:

$$S_{n-1}^{(k)} = k(n-1) / 2(k+\log_2(n)) \tag{6.6}$$

Donde el superíndice (k) nuevamente indica la cantidad de vectores que se deben procesar.

Por otro lado, si las sumas de los elementos de los vectores se llevan a cabo con $n/2$ procesadores tal como se explicó anteriormente, el factor de Speed-Up está dado por la ecuación

$$S_{n/2}^{(k)} = (n-1) / (2\log_2(n)-1) = S_{n/2} \tag{6.7}$$

porque el procesamiento entre los distintos vectores debe ser secuencial, es decir que no se comienza con el cálculo de las sumas parciales de v_i sino hasta que se haya calculado la suma de todos los elementos de v_{i-1}. Por lo tanto se debe repetir secuencialmente k veces el procesamiento de la suma de elementos de un vector.

De acuerdo con la Ec. (6.6) y la Ec. (6.7), si k = 50 y $n = 2^7$, se tienen los factores de Speed-Up $S_{64}^{(50)} \cong 9.77$ y $S_{127}^{(50)} \cong 113$. Si k = 250 y $n = 2^{10}$, se tienen los factores de Speed-Up

$S_{512}^{(250)} \cong 53.84$ y $S_{1023}^{(250)} \cong 954$. Claramente, las diferencias entre los factores de Speed-Up se hacen mayores a mayor cantidad de vectores que se procesan.

6.2 Granularidad

Tal como se detalla en la sección anterior, a medida que la cantidad de procesadores aumenta, normalmente la cantidad de procesamiento en cada uno de ellos disminuye y los requerimientos de comunicaciones aumentan. Esta relación cómputo/comunicación es una constante a medida que se aplica procesamiento paralelo a distintos problemas computacionales. Dada su importancia no solamente en la resolución de problemas sino también en el análisis de rendimiento y de la complejidad del hardware, se la referencia usualmente bajo el término *granularidad*.

Sin entrar en demasiados detalles, se puede definir la granularidad de una aplicación o de una máquina paralela como la relación entre la cantidad mínima o cantidad promedio de operaciones aritmético-lógicas con respecto a la cantidad mínima o promedio de datos que se comunican. La cantidad de operaciones aritmético-lógicas que se contabilizan se llevan a cabo dentro de un mismo procesador y las comunicaciones se hacen necesarias para avanzar con el cómputo.

De alguna manera, la relación cómputo/comunicación tiene impacto directo sobre la complejidad de los procesadores. A medida que los procesadores son más independientes y llevan a cabo más operaciones aritmético-lógicas entre operaciones de comunicación también deben ser más complejos. Esta complejidad creciente se debe a que el procesador necesita almacenar más datos dentro del propio procesador y también en la memoria local. Además, debe ser capaz de ejecutar una mayor cantidad de instrucciones, lo cual implica más capacidad de control de ejecución también.

Algunos autores dan por hecho que a medida que la cantidad de operaciones aritmético-lógicas se acerca a la cantidad de comunicaciones la granularidad disminuye [Lei92]. Esto implica a su vez que a medida que la cantidad de operaciones aritmético-lógicas se hace mayor que la cantidad de comunicaciones, la granularidad aumenta. Expresado de otra manera, aumentar la granularidad implica aumentar también la cantidad de operaciones aritmético-lógicas mínima o promedio entre dos operaciones de comunicación entre los procesadores.

Quizás un último comentario puede ser útil con respecto a la relación existente entre la granularidad del algoritmo que se define y la granularidad de la arquitectura de la máquina paralela sobre la cual se ejecutará el algoritmo. Es bastante claro que si la granularidad del algoritmo es diferente a la granularidad de la arquitectura se tendrá alguna pérdida de rendimiento.

Si un algoritmo determina más operaciones de comunicación de las que se pueden llevar a cabo de manera eficiente en la computadora paralela, es muy probable que el rendimiento decaiga porque la mayor cantidad de tiempo de ejecución se dedicará a las comunicaciones. En este caso, se dice que el algoritmo es de granularidad más fina que la granularidad de la arquitectura. De forma análoga, se dice que la granularidad de la arquitectura es más gruesa que la granularidad del algoritmo.

Si la arquitectura paralela es de granularidad más fina que la granularidad que indica el algoritmo, es muy probable que ni siquiera se pueda ejecutar el algoritmo sobre esa máquina paralela. Esto se debe a que las arquitecturas de granularidad muy fina implican procesadores muy sencillos, con poca capacidad de procesamiento. Si el algoritmo determina una gran cantidad de operaciones aritmético-lógicas o al menos una cantidad mayor de las que cada procesador de la arquitectura puede resolver, entonces no se podrá implementar sobre la máquina paralela.

6.3 Ordenación de un Vector Sobre un Arreglo Lineal de Procesadores

El segundo problema a considerar consiste en la ordenación de los elementos de un vector. El orden de los elementos (menor a mayor o mayor a menor) no altera el tipo de solución, y se utilizará como ejemplo la ordenación de menor a mayor. Este es un problema ampliamente estudiado en la literatura clásica de algorítmica [Aho74] [Aho88] [Knu73], y también en el contexto de los algoritmos paralelos [Lei92].

La presentación de la solución paralela a este problema difiere en varios aspectos si se la compara con la solución propuesta para la suma de los elementos de uno o varios vectores que se presentó en la sección anterior. Las tres diferencias básicas son

- Se parte de la utilización de una arquitectura paralela en particular: un arreglo lineal de procesadores.
- Se establece desde el principio el modo de operación sincrónico de los procesadores y de las comunicaciones (lock-step operation mode). Esto implica la utilización de un reloj global que determina el tiempo durante el cual se lleva a cabo cada una de las operaciones.
- El análisis del tiempo de ejecución se expresa en términos de ciclos o de ciclos de reloj, lo cual es una consecuencia inmediata de asumir el modo de operación sincrónico.

La ordenación de los elementos de un vector se llevará a cabo en un esquema de $N = n$ procesadores fijos, tal como el *arreglo lineal* de la Fig. 6.8, donde n es la cantidad de elementos del vector a ordenar. Cada procesador interior P_i tiene dos enlaces (conexiones) definidos, C_{ii} (*a izquierda*) y C_{id} (*a derecha*), con la excepción del procesador extremo P_N tiene una única conexión a izquierda. La conexión a izquierda de P_1 permite la E/S de datos.

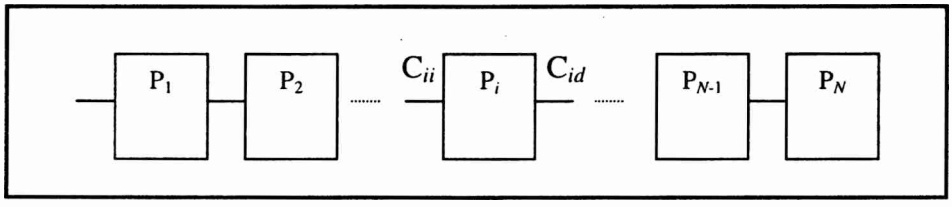


Figura 6.8: Arreglo Lineal de Procesadores.

En principio se supone que cada procesador P_i tiene un programa de control simple, con una reducida memoria local. El sincronismo de tiempo de todos los procesadores es manejado por un *reloj global*, por lo que cada acción de los N procesadores está coordinada y se propaga con los pulsos del reloj. Esto conduce a denominar este arreglo lineal como *arreglo sistólico* (por la semejanza con el bombeo de sangre del corazón humano).

En términos generales, la operación del arreglo sistólico se puede definir por las operaciones a realizar en cada. De esta manera cada procesador P_i: (1) recibe un dato de sus vecinos, (2) analiza el dato en su memoria local, (3) calcula, (4) actualiza la memoria local, y (5) genera datos de salida para sus vecinos.

Se analizará ahora sobre esta arquitectura la ordenación de N números que ingresan al sistema por el procesador P_1. Se pueden obtener los N números ordenados de menor a mayor en dos fases. Durante la primera fase, cada procesador P_i: (1) recibe un dato por C_{ii}, (2) compara el dato con el número almacenado en su memoria local, (3) actualiza la memoria local dejando el menor de los dos números, y (4) envía por C_{id} el mayor de los dos números. En la segunda fase se realiza la salida por el procesador P_1 de los N números ordenados de menor a mayor.

La primera fase del algoritmo explicado conduce a que el procesador P_1 analice los N números de entrada y almacene el menor. De forma análoga, el procesador P_2 analiza los N-1 números restantes (recibidos desde el procesador P_1) y almacena el segundo menor. En general, el procesador P_i analiza N-i+1 números y almacena el i-ésimo menor. La primera fase concluye después de $2N$-1 ciclos de reloj, con los números correctamente ordenados y almacenados en el arreglo de procesadores. La Fig. 6.9 muestra la secuencia de once pasos que se suceden para ordenar el vector de entrada que se corresponde con la secuencia de los seis números 5, 0, 6, 8, 3, 1 en un arreglo lineal con $N = 6$ procesadores.

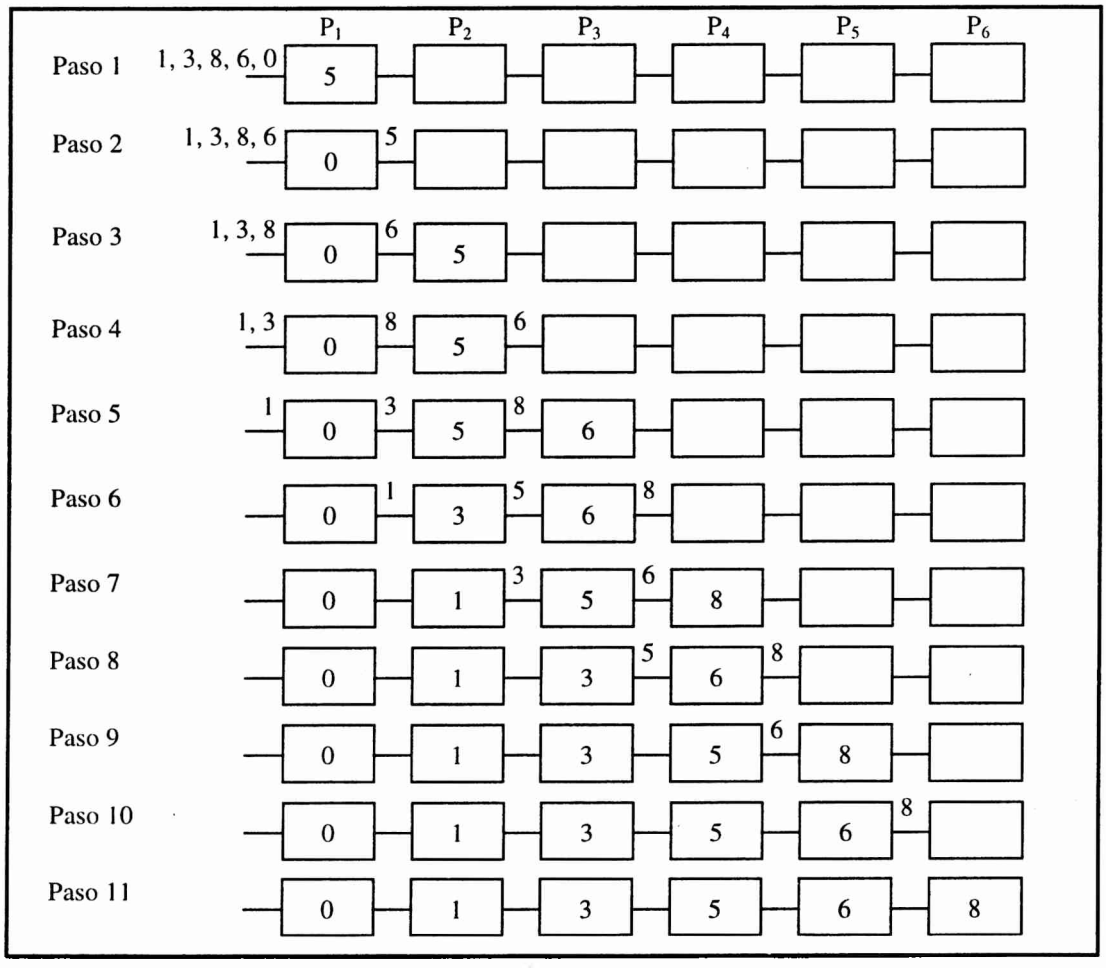


Figura 6.9: Ordenación de un Vector de Seis Números.

Siguiendo la Fig. 6.9, en el primer paso se almacena el primer número de entrada (5) en el primer procesador. Más específicamente, se podría decir que, inicialmente, los procesadores tienen almacenado en su memoria local el máximo número de entrada posible. De esta forma, cualquier número de entrada será menor o igual que el valor almacenado en la memoria local del procesador. En el segundo paso, el número de entrada (0) es menor que el almacenado en el procesador P_1 (5), y por lo tanto el valor de entrada (0) queda almacenado y el otro (5) se transmite al procesador siguiente por medio de la conexión C_{1d}. La sucesión de pasos continúa hasta tener los números almacenados de forma ordenada de menor a mayor en los procesadores en el paso 11.

En la segunda fase se debe llevar a cabo la salida de los números ordenados en el arreglo lineal. Para ello se pueden utilizar varios métodos, según el hardware disponible o la complejidad del algoritmo de control de cada procesador:

- La solución más simple requiere que el procesador P_N sea especial. Cuando este procesador recibe el dato por C_{Ni}, comienza a trasmitirlo hacia la izquierda. Cada uno de los procesadores P_i, cuando recibe un dato por C_{id} trasmite lo que tiene en su memoria hacia la izquierda (al procesador P_{i-1}) y almacena el dato recibido de P_{i+1}. A partir del comienzo de la recepción de datos por C_{id}, si en un ciclo de reloj el procesador P_i no recibe datos, trasmite lo que tiene en memoria y termina. Cuando el procesador P_1 trasmite el último dato que tiene en memoria, el arreglo ordenado ha terminado la salida.
- Una solución similar, sin que necesariamente el procesador P_N sea especial, requiere que luego de k ciclos sin recibir datos por C_{ii}, cada procesador P_i comience la transmisión de datos a izquierda. Suponga el lector que $k = 1$ y analice la evolución de la segunda fase en el arreglo de la Fig. 6.8. ¿Cuántos ciclos de reloj se requieren?
- Una pequeña variante de la solución anterior es que cada procesador P_i *conozca su posición en el arreglo*. Tal como se ve al analizar la Fig. 6.8, luego de haber recibido l datos por C_{ii}, si $l+i = N+1$, el procesador P_i puede comenzar la transmisión de datos hacia la izquierda.

6.3.1 Breve Análisis de Rendimiento

Factor de Speed-Up: se tratará de analizar el factor de Speed-Up de la estructura de ordenación lineal de N procesadores que se ha visto. Se considera que todo el hardware necesario para optimizar la paralelización está disponible.

Se ha visto en el análisis previo que el tiempo total de ejecución paralela T es función lineal de N, es decir que T es $\Theta(N)$. La mejor solución secuencial (que se puede ejecutar en una computadora monoprocesador) encontrada para ordenar un vector de N números requiere un tiempo [Aho74] [Aho88] Ts que es $\Theta(N \log(N))$. Por lo tanto, el factor de Speed-Up que se alcanza con el circuito de N procesadores es

$$\Theta(S_N) = \Theta(\text{Ts}) / \Theta(\text{T}) = \Theta(\log(N)) \qquad (6.8)$$

En el caso de los algoritmos de ordenación es difícil alcanzar el máximo factor de Speed-Up teórico, es decir $S_N = N$. Alcanzar este factor de Speed-Up significaría que *todo* el trabajo de ordenación, durante *todo* el tiempo de procesamiento se distribuye homogéneamente entre los procesadores. Es claro que la estructura de la Fig. 6.7 no puede/debe hacer trabajar a *todos* los procesadores en *todos* los ciclos de reloj.

Trabajo realizado por el arreglo de procesadores: otro aspecto importante para analizar el rendimiento es considerar el trabajo W realizado por la arquitectura de procesamiento. Se define el trabajo W como el producto de la cantidad de procesadores por el tiempo de ejecución (haciendo una analogía con la función trabajo de la física). De esta manera, se intenta capturar la idea de la cantidad de procesamiento que todos los procesadores realizan durante la ejecución de un algoritmo. En realidad se está asumiendo que todos los procesadores están ejecutando instrucciones durante todo el tiempo de ejecución. En este caso se tiene que

$$\Theta(W) = \Theta(N^2). \tag{6.8}$$

Naturalmente, la noción de trabajo conduce al concepto de *eficiencia* de los procesadores que se utilizan en la solución paralela de un algoritmo. Para la definición de eficiencia se intenta tener en cuenta que aunque todos los procesadores ejecuten instrucciones continuamente, la cantidad de instrucciones de la solución secuencial (monoprocesador) no varía. Como es de esperar, no tiene sentido agregar potencia de cálculo cuando la que ya se tiene alcanza (o sobra) para resolver un problema. De otra manera, se pueden agregar procesadores que no tengan ninguna tarea (instrucciones) para realizar. Teniendo en cuenta esto, la eficiencia se define como la relación entre el tiempo de ejecución del mejor algoritmo secuencial (a ejecutarse en una máquina monoprocesador) con respecto al trabajo del algoritmo paralelo. Es decir

$$E = \mathrm{Ts} / W \tag{6.9}$$

Por razones de costo, en el procesamiento paralelo interesa maximizar el factor de Speed-Up utilizando el mínimo posible de procesadores. De forma alternativa a la Ec. (6.9) pero con idéntico resultado, se define entonces la eficiencia como la relación entre el factor de Speed-Up obtenido y el máximo factor de Speed-Up posible de alcanzar, es decir

$$E = S / N \tag{6.10}$$

Tal como fue definida en el primer capítulo. Se debe recordar que el mejor valor de eficiencia se obtiene cuando $E = 1$, es decir que todos los procesadores están trabajando todo el tiempo. De forma análoga, valores de E cercanos a 0 indican que los procesadores tienen muy poca carga de trabajo para realizar.

En el ejemplo presentado de ordenación de los elementos de un vector, la eficiencia E es $\Theta(\log(N) / N)$ lo cual tiende a ser muy ineficiente cuando N crece. Esto se debe a que

$$\log(N) / N \xrightarrow[N \to \infty]{} 0 \tag{6.11}$$

Lo cual a su vez significa que, en promedio, a medida que se agregan procesadores al arreglo lineal, cada procesador estará sin ejecutar instrucciones durante más tiempo. Este resultado se confirma claramente al observar la Fig. 6.8 y el ejemplo de la Fig. 6.9. Uno de los factores que contribuyen es el tiempo de inactividad en cada procesador hasta la llegada de los datos.

Si bien es natural que se intente obtener la eficiencia máxima ($E = 1$), o lo que es lo mismo el Speed-Up lineal $S = N$, se puede tener una definición más "suave" del factor de Speed-Up. De esta manera, se puede comparar la solución paralela que se ejecuta sobre el

arreglo lineal de procesadores con una solución secuencial elemental, tal como la de N pasadas. En este caso se llega a que S es $\Theta((N^2) / N)$, es decir que S es $\Theta(N)$ y también que la eficiencia $E = 1$.

6.3.2 El Caso de Utilizar N Procesadores con $N < n$

En muchos casos se tienen menos procesadores que datos a procesar y, en particular, menos procesadores que datos a ordenar. Se supone para simplificar el análisis que $N < n$ y que $n/N = K$, donde n es la cantidad de elementos a ordenar. Si la granularidad de cada uno de los N procesadores lo permite, en ellos se pueden almacenar K números del vector de entrada.

La Fig. 6.10 muestra el caso en el cual $n = 12$ y $N = 4$, y por lo tanto cada procesador debe almacenar y manejar $n/N = K = 3$ de los números a ordenar. En general, se deben utilizar *K ciclos internos* de cada procesador para ejecutar el mismo algoritmo de ordenación definido para $N = n$ procesadores y realizar una comunicación entre procesadores. Esto significa que el tiempo total de ejecución del algoritmo es T' = 3T donde T es el tiempo de ejecución de la ordenación del arreglo con $N = n$ procesadores.

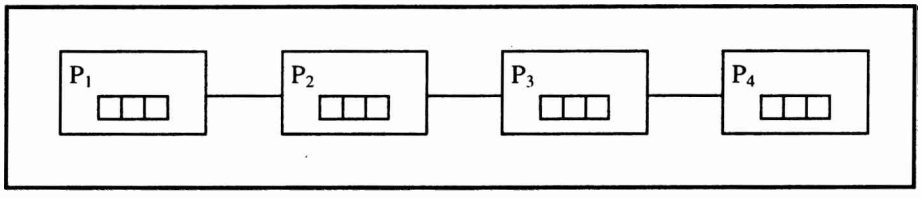


Figura 6.10: Distribución de los Elementos de un Vector.

Se debe notar que el mayor tiempo de ejecución *no afecta la eficiencia* pues se ha reducido en la misma proporción la cantidad de procesadores. En general, siempre se puede migrar un algoritmo paralelo que se ejecuta sobre N_1 procesadores a N_2 procesadores, con $N_1 < N_2$, manteniendo su eficiencia. En cambio el proceso inverso no siempre es posible, porque normalmente se pierde eficiencia al agregar procesadores y no se los puede ocupar a todos durante todo el tiempo de ejecución.

6.4 Ordenación Binaria

En todos los ejemplos anteriores se ha supuesto que cada uno de los procesadores utilizados es capaz de manejar un dato numérico, *independientemente de la cantidad de bits necesarios para su representación interna*. Además, en el ejemplo anterior de ordenación también se ha asumido que la unidad aritmético-lógica interna es capaz de comparar dos números cualquiera en un ciclo de reloj interno. Es decir que los procesadores tienen un ancho de palabra de datos suficientemente grande como para representar y procesar un número cualquiera.

En las arquitecturas paralelas reales se dan muchos casos en los cuales los procesadores solamente manejan bits. A estos procesadores se los suele denominar *procesadores binarios*. Naturalmente, los procesadores se pueden organizar y comunicar con el fin de procesar datos numéricos de cualquier cantidad de bits, con el consiguiente costo en ciclos de reloj. Es interesante analizar cómo difiere el factor de Speed-Up y la eficiencia

alcanzables, según la estructura del arreglo de procesadores binarios que se tengan.

La primera propuesta que se analizará, tal como se ve en la Fig. 6.11 consiste en un arreglo lineal de procesadores que compara los números bit a bit, comenzando por el bit más significativo. El bit más significativo es el que corresponde al procesador P_1. La salida de cada procesador depende del valor de la entrada a izquierda y de los valores de los bits que tiene almacenados y que pertenecen a los números a comparar.

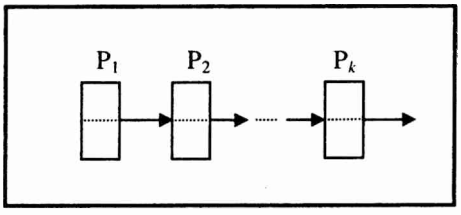


Figura 6.11: Procesadores de Comparación de Bits.

En la Fig. 6.11, el procesador P_1 no tiene entrada a izquierda porque el resultado de la comparación del bit más significativo no depende de ningún otro resultado. Cada procesador P_i con $i > 1$ produce la salida teniendo en cuenta que si a su izquierda ya se ha encontrado que uno de los dos números es menor (mayor), el resultado de la comparación entre los dos números almacenados localmente no afecta el resultado final. Esto se debe a que los procesadores ubicados a su izquierda (de los cuales recibe su entrada) contienen los bits más significativos de los números. De este modo, un procesador de números de k bits se puede construir a partir de k procesadores binarios. Luego de k ciclos de reloj se tiene el resultado de la comparación. El resultado final de la comparación es la salida del procesador P_k.

En la Fig. 6.12 se muestra la comparación de los números num_1 = 00100101 con num_2 = 00011111 (k = 8). La notación (=, S, I) se establece para determinar si los números (bits) son iguales o si el número (bit) superior o inferior respectivamente es el menor de los dos.

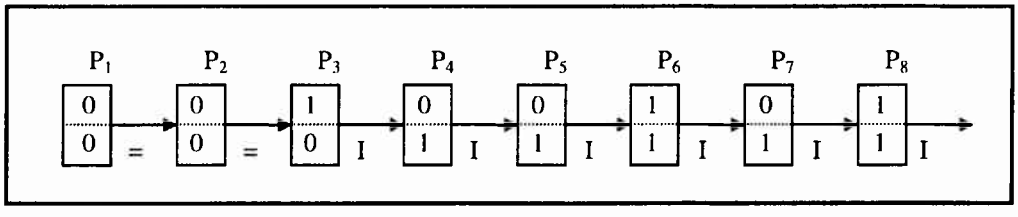


Figura 6.12: Comparación de Dos Números de 8 Bits.

La Fig. 6.13 muestra una segunda alternativa (para k = 8), la cual propone construir un árbol binario lleno de procesadores binarios. Si bien un primer análisis de la estructura de la Fig. 6.13 puede parecer menos eficiente porque requiere más procesadores, al construir un árbol de procesadores para realizar la comparación se tienen dos ventajas:

- En vez de k ciclos de reloj, se necesitan $\log_2(k)$ ciclos para obtener el resultado de la comparación.
- Si cada uno de los procesadores "de números" de la Fig. 6.8 se reemplaza por un árbol de k procesadores binarios, la notificación "hacia atrás" acerca de cuál es el número a transmitir a derecha es realizable por las mismas interconexiones, y también utiliza $\log_2(k)$ ciclos de reloj.

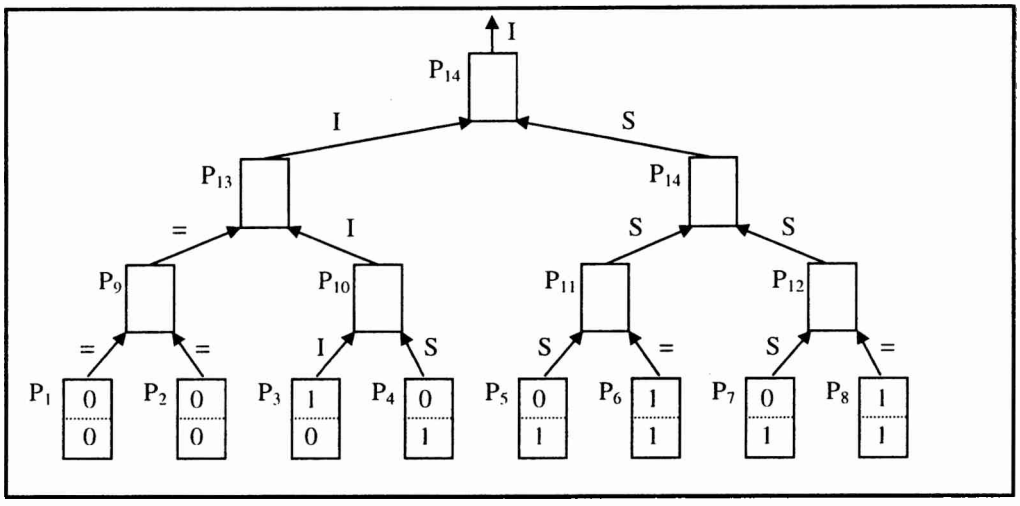


Figura 6.13: Arbol Binario de Comparación.

En la Fig. 6.14 se muestra el proceso de notificación desde el procesador ubicado en la raíz del árbol binario hasta los procesadores ubicados en las hojas. Esta notificación es necesaria para efectuar luego la transmisión del menor de los dos números. Una vez que el procesador raíz tiene el resultado, puede transmitirlo a sus dos hijos en paralelo, utilizando los enlaces definidos por la estructura de árbol. En el siguiente paso, se vuelve a transmitir el mismo resultado final a un nivel más cercano a los procesadores ubicados en las hojas. Se continúa de la misma manera hasta llegar a los procesadores que tienen almacenados los bits de los números comparados.

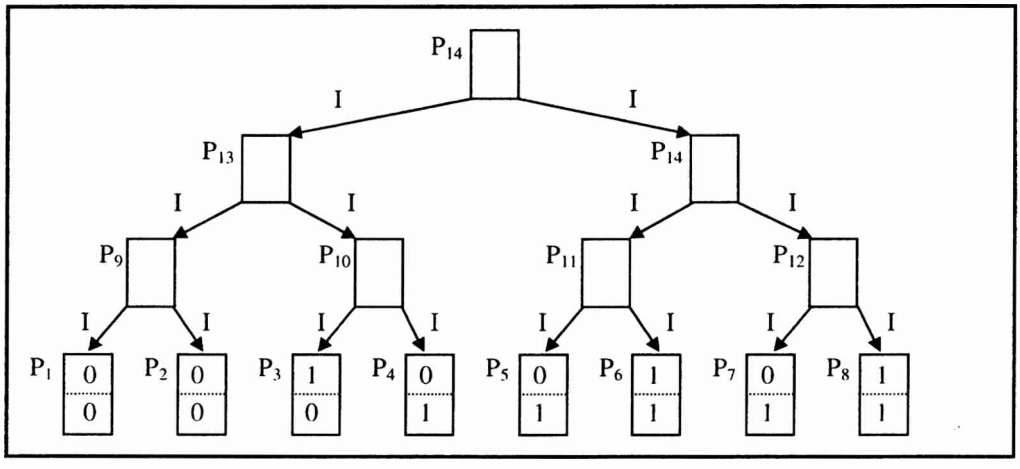


Figura 6.14: Notificación en el Arbol Binario de Comparación.

Por lo tanto, la comparación de dos números de k bits se completa en $2\log_2(k)$ ciclos de reloj. En el ciclo siguiente *todo el número* puede ser transmitido a derecha siguiendo la estructura de la Fig. 6.8.

La Fig. 6.15 muestra la forma de interconectar los procesadores binarios de dos procesadores consecutivos de un arreglo lineal de comparación tal como el que se encuentra en la Fig. 6.8. Los procesadores de comparación P_i y P_{i+1} se muestran con línea de puntos. Finalmente, se tiene que para ordenar $N = n$ números de k bits se construye un arreglo de N

árboles binarios llenos, con $2k$-1 procesadores binarios cada uno, que completan la primera fase de la ordenación en $(2N\text{-}1)*2\log_2(k)$ ciclos de reloj.

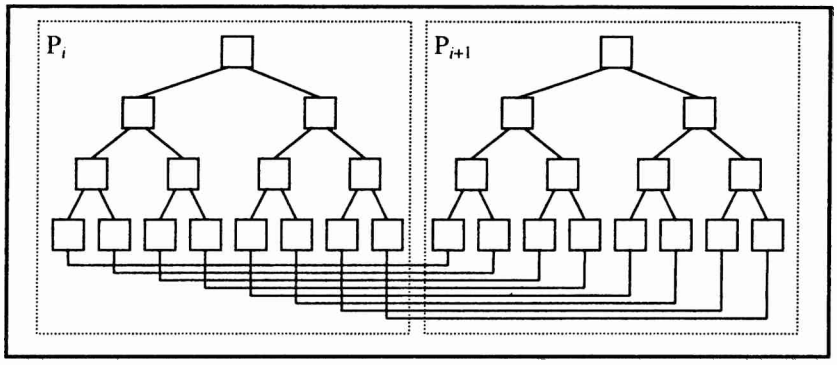


Figura 6.15: Interconexión Entre Dos Procesadores Consecutivos.

Se debe notar que se continúa haciendo el análisis de tiempo de ejecución como si se utilizara un único reloj global. Sin embargo, se podría tener un reloj interno más rápido en cada procesador construido de procesadores binarios y otro en los ciclos de comunicación entre árboles de la estructura. De todos modos, la hipótesis adoptada favorece el análisis conceptual de los algoritmos en relación con la arquitectura.

Aunque la proposición de construir árboles binarios de comparación significa un avance en tiempo de ejecución con respecto al arreglo lineal de procesadores, esta solución inicial (Fig. 6.8) puede mejorar sustancialmente *si se adapta la arquitectura a la clase de problema en cuestión.*

Desde el punto de vista conceptual se está utilizando cada procesador un bajo porcentaje de tiempo y en un sólo eje espacial. La idea de la solución que se va a analizar a continuación trata de explotar dos ejes simultáneos, construyendo una tabla (mesh) de procesadores que trabajan coordinadamente. Se comenzará por analizar una estructura de k procesadores binarios que trata de encontrar el mínimo entre n números de k bits.

La Fig. 6.16 muestra la forma en que se disponen los bits de los números a procesar, el estado inicial de la búsqueda y el primer paso en la evolución para los valores $n = 4$, $k = 4$ y los números 2, 4, 0, 9. los bits de los números de entrada ingresan a los procesadores de comparación de acuerdo a su posición en el número y a la posición del número en el vector de entrada.

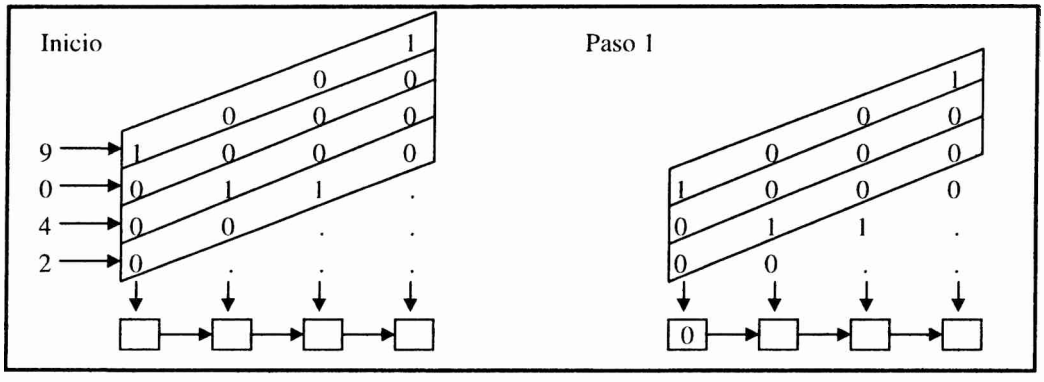


Figura 6.16: Inicio y Primer Paso en la Búsqueda del Menor Número.

La Fig. 6.17 muestra el segundo y tercer paso de la evolución. En el tercer paso de la evolución se encuentra la primera comparación significativa en el segundo procesador. De esta comparación se concluye que el segundo número de la secuencia (2) es menor que el primero (4) por la relación del segundo bit más significativo de ambos números. Es por esta razón que el tercer procesador de comparaciones binarias recibirá el valor de entrada I. Cuando este procesador recibe la el valor de entrada I, almacena el bit del número inferior (el bit que ya tiene almacenado en su memoria local) independientemente de los valores que tengan el bit de entrada y el bit almacenado en la memoria local.

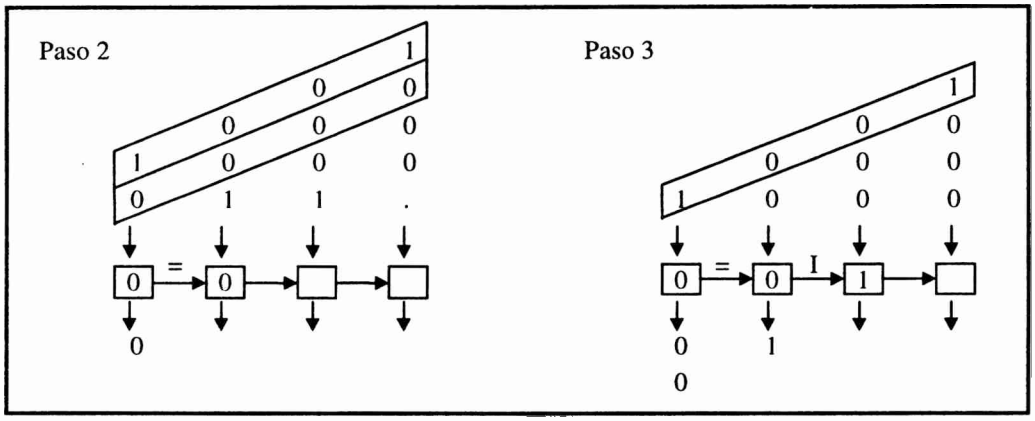


Figura 6.17: Segundo y Tercer Paso en la Búsqueda del Menor Número.

La Fig. 6.18 muestra cómo evoluciona el arreglo de procesadores entre el cuarto y el último paso (Paso 7).

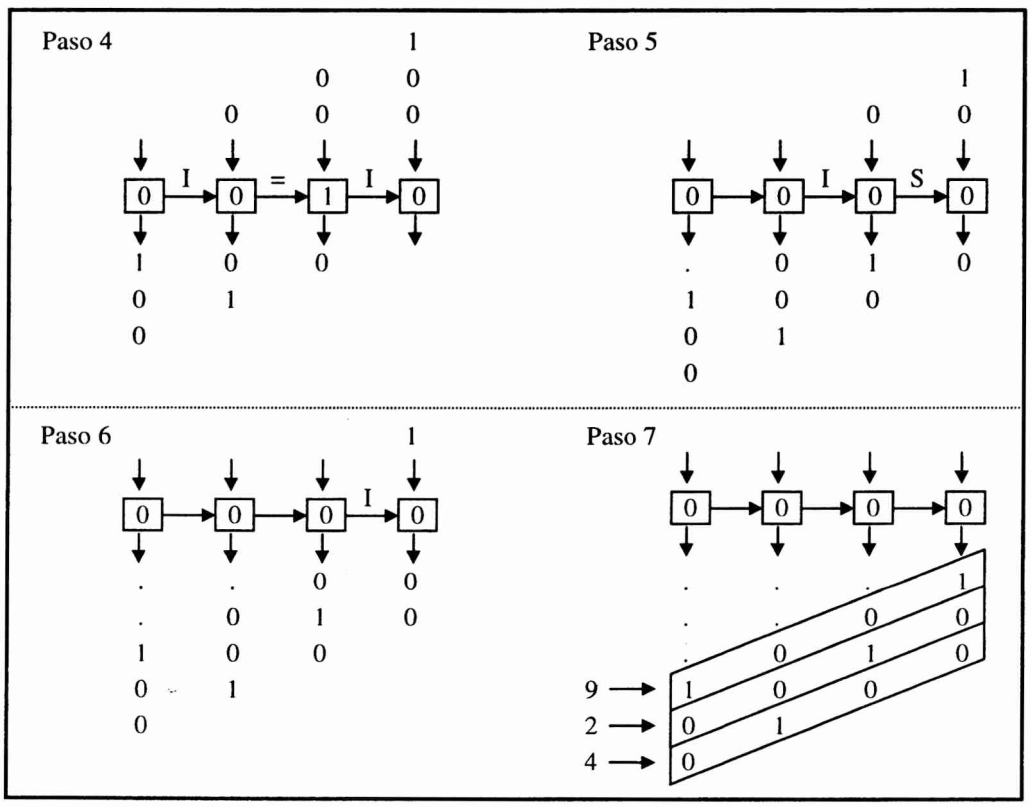


Figura 6.18: Cuarto al Séptimo Paso en la Búsqueda del Menor Número.

En general, al cabo de $n+k-1$ ciclos de reloj (7 ciclos en el ejemplo), el arreglo de k procesadores binarios contiene *el mínimo de los n números.* Además, el arreglo de k procesadores ha entregado los $n-1$ números de entrada restantes, con otro ordenamiento relativo (ver la salida de la Fig. 6.18).

Si se regresa al problema original de la ordenación de n números, la extensión de la estructura lineal mostrada es bastante clara: con n filas de k procesadores se debería poder tratar *en paralelo* la ordenación de n números de k bits cada uno. Con esta estructura, la primera fila busca y almacena el menor de los n números del vector de entrada. Los $n-1$ números restantes del vector de entrada salen de la primera fila y son procesados por la segunda fila que busca y almacena el menor de ellos. De esta manera, se tiene en la primera fila el menor y en la segunda fila el segundo menor de los números del vector. En general, la fila i de la estructura busca y almacena el número i-ésimo menor del vector de entrada entre los $n-i+1$ números que le llegan de la fila $i-1$ de la estructura.

Cada fila de la estructura bidimensional propuesta puede realizar su tarea junto con las demás una vez que le llegan los datos. No es necesario, por ejemplo, que la segunda fila espere para comenzar su tarea hasta que la primera fila haya realizado *completamente* la suya. A medida que los datos salen de la primera fila, la segunda puede procesarlos en su propia búsqueda del mínimo de los valores. En general, una vez que los números comienzan a salir de la fila i se pueden procesar en la fila $i+1$. De esta forma, nuevamente se procesan los datos en modo pipeline tal como se presentó en el problema de la suma de los elementos de vectores de entrada. Gracias al procesamiento pipeline de los datos se gana, en términos de velocidad de ejecución, en proporción a la cantidad de etapas (en este caso filas) del pipeline.

La Fig. 6.19 muestra la estructura bidimensional de hardware de ordenación para el problema con $n = 4$, $k = 4$, y el vector de entrada con los valores: 2, 4, 0, 9. Según el esquema de la Fig. 6.19 y tal como se ha explicado, la búsqueda del mínimo de los valores de entrada que realiza la primera fila de procesadores se solapa en el tiempo con la búsqueda del mínimo de los valores que salen de la primera fila de procesadores y que lleva a cabo la segunda fila. Finalmente, cuando se ha finalizado el procesamiento, los números quedan almacenados en las memorias locales de los procesadores de forma ordenada por filas.

Es interesante notar que cada una de las filas de procesamiento de la Fig. 6.19 es prácticamente igual a la estructura de la Fig. 6.18. En este sentido, no se agrega hardware de procesamiento de datos ni de control adicional. Solamente se conecta cada fia de procesadores de ordenación con la previa (para recibir los datos de entrada) y con la siguiente (para enviar los datos de salida).

En general, dados n números de k bits, el tiempo de $n+k-1$ ciclos para obtener el mínimo en la primera fila se solapa con el procesamiento del segundo mínimo y así sucesivamente hasta la última fila. Se aprovecha de esta manera la propiedad de pipelining que tiene el algoritmo. Se puede demostrar que luego de $2n+k-2$ ciclos la primera fase de la ordenación se completa. El último paso de ejecución está determinado por la llegada al procesador P_{nk} del bit menos significativo del mayor de los números de entrada. En términos de la propia estructura, el último paso de ejecución es aquel en el cual se asigna la memoria local del procesador P_{nk} de la estructura. La segunda fase (la salida de los números ordenados de la estructura) se completa con $n-2$ ciclos al igual que en el arreglo lineal de procesadores no binarios.

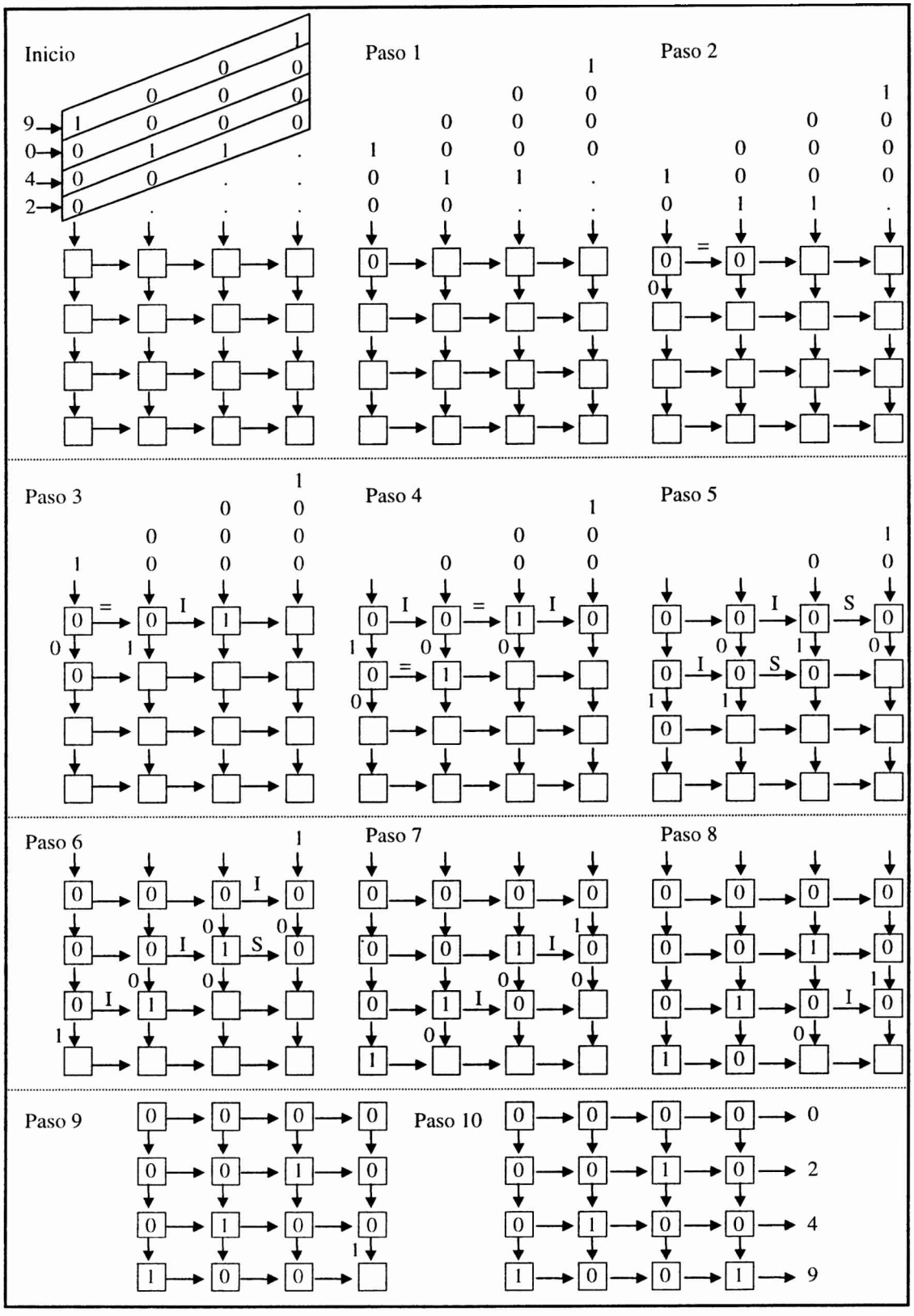


Figura 6.19: Estructura Bidimensional de Ordenación.

Es interesante comparar la solución con la estructura de árbol de procesadores binarios con la tabla para distintos valores de n y k. Si, por ejemplo, $n = 1000$ y $n = 8$,

- la solución con la estructura de árbol requiere $(2*n\text{-}1)*(2\log_2(k)) \equiv 12.000$ ciclos.

- la solución con el arreglo bidimensional de procesadores requiere $2*1000+8-2 \cong 2.000$ ciclos, es decir, aproximadamente 6 veces menos.

También se debe recordar que la cantidad de procesadores e interconexiones entre ellos que requiere el arreglo bidimensional son mayores que los del árbol.

6.4.1 Conclusiones y Comentarios de Rendimiento

- El algoritmo presentado y analizado previamente es un buen ejemplo de construcción de una arquitectura orientada a resolver problemas complejos, *a partir de procesadores elementales muy simples.*
- En la medida en que el hardware soporta entrada/salida de datos independiente en cada procesador, se puede mejorar el tiempo de procesamiento. En el arreglo bidimensional, solamente se tienen k procesadores que hacen E/S, con lo cual se requieren muchos ciclos para que todos los bits pasen por la estructura de procesadores y sean procesados.
- Se denomina *diámetro* de una arquitectura de procesamiento a la máxima distancia entre dos procesadores. En el esquema bidimensional es claro que esta distancia es $n+k-2$. Este diámetro es importante porque fija un límite mínimo al tiempo de comunicación entre estos dos procesadores: $n+k-2$ ciclos de reloj para transmitir un dato desde el procesador P_{11} al procesador P_{kn}. Naturalmente, más allá del diámetro que tenga la topología, es importante que los algoritmos que se desarrollen requieran la mínima comunicación posible entre procesadores distantes.
- Los problemas de *bordes* aparecen al dividir un espacio de datos a procesar por una arquitectura paralela en dos o más subespacios. Típicamente, realizar esta división (por ejemplo en los problemas de tratamiento de imágenes) facilita la paralelización y el tratamiento independiente de subconjuntos de los datos. El problema es que luego hay que resolver la comunicación entre los procesadores que manejan subáreas de la información. En particular, cuando se resuelve un problema de ordenación mediante la división del espacio de datos se puede llegar a un algoritmo de *sorting by merging*.
- Hasta ahora se ha obviado el problema de *sincronización* entre los procesadores, porque se ha hecho referencia a un reloj global único para todos. Sin embargo, en la medida que se subdivide la arquitectura y el espacio de datos, puede suceder que la carga de trabajo no sea homogénea y que los procesadores terminen su tarea en tiempos diferentes. También puede suceder que cada procesador tenga una carga de trabajo dependiente del subespacio de los datos que se le asigne. En este caso se tienen dos alternativas: aceptar la sincronización con el más procesador lento, o bien establecer mecanismos que lleven a cabo la sincronización y reasignación de tareas *dentro del algoritmo paralelo.*

6.5 Algoritmos de Cálculo sobre Matrices

En lo que sigue se analizarán algunos algoritmos simples de procesamiento de datos organizados en matrices. En general serán fáciles de implementar sobre arreglos y/o árboles de procesadores, cada uno de los cuales con la capacidad de manejar operaciones simples sobre números (multiplicación, suma, división). En el caso de migrar este tipo de algoritmos a procesadores binarios, se tendrían que aplicar los conceptos explicados en la sección anterior.

Los algoritmos que se verán constituyen la primera generación de aplicaciones de

procesamiento paralelo. De ellos nació el concepto de computadora vectorial, es decir las computadoras con instrucciones para el procesamiento de vectores. A diferencia de las instrucciones tradicionales que operan sobre valores escalares, el alcance de una instrucción vectorial es, justamente, un vector de datos, un arreglo lineal de valores.

Una de las características más interesantes de los algoritmos de procesamiento de matrices es que al aplicar procesamiento paralelo se puede llegar al factor de Speed-Up ideal, aunque con un alto costo en número de procesadores y por ende una baja eficiencia.

Los lectores interesados en analizar implementaciones paralelas de algoritmos más complejos (como la resolución de sistemas de ecuaciones) basados en este subconjunto básico pueden consultar, por ejemplo, [Lei92].

6.5.1 Multiplicación de una Matriz por un Vector

Dada una matriz $A = a_{ij}$ con $i = 1, ..., n$ y $j = 1, ..., n$; y un vector $X = x_j$ se debe calcular el vector producto $Y = y_i = A\ X$ con $i = 1, ..., n$, definido por :

$$y_i = \sum_{j=1}^{n} a_{ij}x_j \text{ con } i = 1, ..., n. \tag{6.12}$$

El algoritmo secuencial para el cálculo del vector Y es simple, y requiere n operaciones de multiplicación y n-1 de suma para cada y_i. Esto significa que el cálculo total requiere $n(2n-1) = 2n^2-n$ operaciones. Si, por ejemplo, $n = 1000$ y cada operación se realiza en 100 nseg, se tendrían aproximadamente 0.2 segundos de procesamiento para obtener el vector resultado del producto.

Desde el punto de vista de complejidad, la solución secuencial es $O(n^2)$. La función que proporciona la cantidad de pasos de ejecución es $g(n) = 2n^2-n$, donde n es la cantidad de filas y columnas de la matriz A y también la cantidad de elementos de los vectores X e Y.

La primera propuesta para la resolución del problema en paralelo consiste en la utilización de un arreglo lineal de $N = n$ procesadores con los datos de entrada dispuestos como se muestra en la Fig. 6.20.

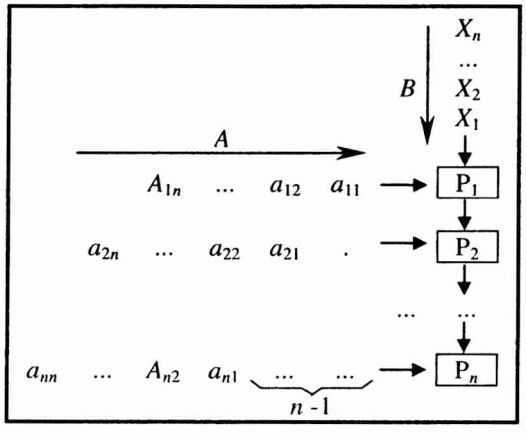


Figura 6.20: Matriz por Vector en un Arreglo Lineal de Procesadores.

La tarea de cada procesador P_i de la Fig. 6.20 se puede definir en cada paso de ejecución como sigue:

- Recibir las entradas: por el enlace superior un valor del vector X y por el enlace a izquierda un valor de una fila de la matriz A.
- Multiplicar los valores de entrada.
- Sumar el resultado de la multiplicación en su memoria local.
- Transmitir por el enlace inferior el valor del vector X que recibió por el enlace superior.

Por lo tanto, el procesador P_i resuelve el resultado para y_i según la ecuación

$$y_i = \sum_{j=1}^{n} a_{ij}x_j \tag{6.13}$$

Se debe notar que los valores a_{ij} y x_j llegan al procesador P_i en el paso $i+j-1$ y permiten calcular (haciendo uso de la memoria local) el valor y_i. El último valor del vector Y calculado es y_n, que se resuelve en el paso $2n-1$.

Volviendo al ejemplo de $n = 1000$ con 100 nseg para realizar cada paso de ejecución, el arreglo lineal de la Fig. 6.20 lleva a cabo el procesamiento de forma paralela en un tiempo aproximado de 0.2 milisegundos. En este caso se obtiene un factor de Speed-Up aproximadamente igual a 1000, que se corresponde con la cantidad de procesadores utilizados en el arreglo lineal. En este cálculo se asume que cuatro operaciones (las definidas anteriormente para cada procesador P_i), se realizan en el mismo tiempo que se estableció para una operación en la computadora secuencial. Para ser más exactos en el cálculo se debería establecer que el tiempo de un paso de ejecución en cada P_i se corresponde con el tiempo de una operación multiplicado por cuatro, es decir 400 nseg. De esta manera se llega a que el tiempo de ejecución en el arreglo lineal es de aproximadamente 0.8 milisegundos, y que el factor de Speed-Up es aproximadamente igual a 250. Por otro lado, el arreglo lineal se construye con 1000 procesadores y 999 enlaces de comunicación entre ellos.

En general, el tiempo de ejecución de la solución secuencial es $O(n^2)$ y el tiempo de ejecución de la solución paralela es $O(n)$. Por lo tanto, el factor de Speed-Up al que se llega es $O(n)$, que es el mejor al que se puede aspirar con $N = n$ procesadores. Como siempre, se debe recordar que el costo de llegar a tal factor de Speed-Up es construir una estructura con tantos procesadores como filas de la matriz y elementos del vector a multiplicar.

6.5.2 Producto de dos Matrices

Dadas dos matrices, $A = a_{ij}$ y $B = b_{ij}$, con $i = 1, ..., n$, y $j = 1, ..., n$; se define la matriz producto $C = c_{ij} = AB$ con $i = 1, ..., n$, y $j = 1, ..., n$ como

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \text{ con } i = 1, ..., n, \text{ y } j = 1, ..., n. \tag{6.14}$$

De forma similar al cálculo de una matriz por un vector, el algoritmo secuencial para la multiplicación de matrices es simple y requiere n multiplicaciones y $n-1$ sumas para cada c_{ij}. Esto significa que el cálculo total de la multiplicación de matrices requiere $g(n) = n^2(2n-1)$

$= 2n^3-n^2$ operaciones. Si $n = 1000$ y cada operación se lleva a cabo en 100 nseg, se tendrían aproximadamente 200 segundos de tiempo de ejecución.

La propuesta para resolver la multiplicación de matrices en paralelo consiste en la construcción de un arreglo bidimensional (matricial) de n^2 procesadores con las matrices ingresando a la estructura tal como se muestra en la Fig. 6.21.

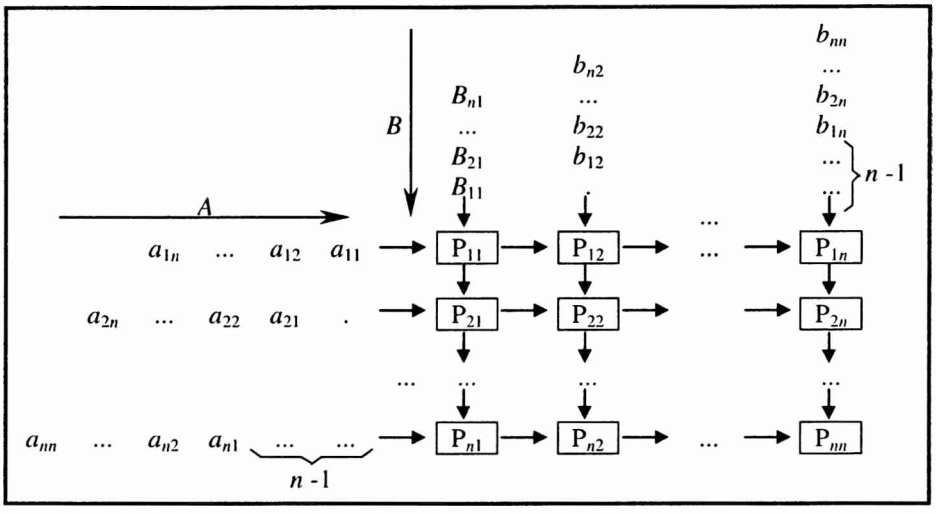


Figura 6.21: Estructura Bidimensional para Multiplicación de Matrices.

En el esquema de la Fig. 6.21, cada procesador P_{ij} es similar a los procesadores del arreglo lineal que se ha detallado previamente, con el mismo tipo de operaciones que se llevan a cabo en un paso de ejecución. Lo único que se agrega en cada procesador P_{ij} es que, además de transmitir por el enlace inferior lo que recibe por el enlace superior, transmite por el enlace a derecha lo que recibe por el enlace a izquierda. En este caso, el procesador P_{ij} recibe por el enlace a izquierda los valores a_{ik} de la matriz A, con $k = 1, ..., n$; y recibe por el enlace superior los valores b_{kj} de la matriz B con $k = 1, ..., n$. De acuerdo con la Fig. 6.21 cada procesador P_{ij} resuelve el resultado para c_{ij}.

La Fig. 6.22 muestra la evolución de la estructura de la Fig.6.21 en el quinto paso de ejecución de la multiplicación de dos matrices $A = a_{ij}$ y $B = b_{ij}$, con $i = 1, ..., 5$, y $j = 1, ..., 5$. En la Fig. 6.22 también se pueden identificar los procesadores ocupados (con datos) y los procesadores ociosos (están en blanco). Nótese que el k-ésimo par de datos llega al procesador P_{ij} en el paso de ejecución $i+j+k-2$, el cual permite llevar a cabo el cálculo (haciendo uso de la memoria local) de c_{ij} de acuerdo con la ecuación

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad (6.15)$$

Por lo tanto, los datos terminan de llegar a cada procesador P_{ij} en el paso de ejecución $i+j+n-2$. El último c_{ij} que se calcula es c_{nn}, lo cual se puede verificar gráficamente en la Fig. 6.21 y en la Fig. 6.22. El valor de c_{nn} se obtiene en el procesador P_{nn} en el paso de ejecución (como se ha explicado antes), $n+n+n-2$. Se llega de esta manera a que, para terminar el cálculo de la multiplicación de matrices se necesitan $3n-2$ pasos de ejecución. En términos de complejidad, la solución paralela presentada es $O(n)$.

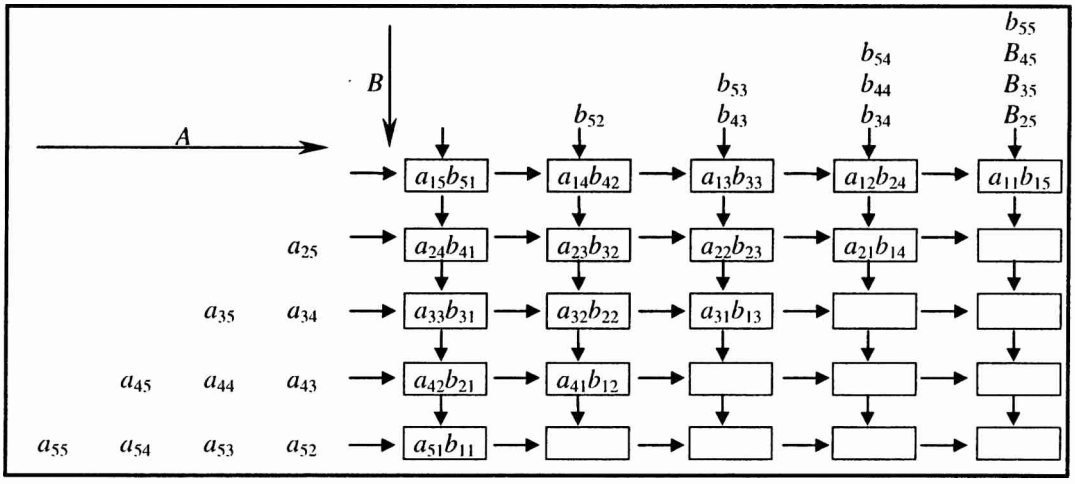


Figura 6.22: Quinto Paso en la Multiplicación de Matrices.

Con respecto a la solución secuencial, si se considera que el tiempo en ejecutar un paso del algoritmo en la estructura bidimensional es 100 nseg y n = 1000, se tiene un tiempo de ejecución aproximadamente igual a 0.3 milisegundos. Relacionando este tiempo con el de la solución secuencial calculado previamente, se obtiene un factor de Speed-Up con valor aproximado de 1000000. Nuevamente se debe recordar que un paso de ejecución del algoritmo paralelo involucra varias operaciones elementales y también que la cantidad de procesadores debe ser la misma que la cantidad de datos de las matrices A y B, es decir en este caso, 1000000 de procesadores.

En general, se tiene que la solución secuencial es $O(n^3)$ y que la solución paralela presentada es $O(n)$, por lo tanto el factor de Speed-Up es $O(n^2)$. También en general, la cantidad de procesadores que requiere la solución paralela presentada es igual a la cantidad de datos de cada una de las matrices, esto significa tener n^2 procesadores interconectados en forma de tabla (recordar las definiciones de trabajo y de eficiencia).

6.5.3 Consideraciones de Rendimiento

Para mejorar la eficiencia de los algoritmos, es interesante explotar la posibilidad de hacer *pipelining* de aplicaciones, de modo que los procesadores estén continuamente trabajando. Si bien la mayoría de las veces no es posible que todos los procesadores tengan instrucciones para ejecutar simultáneamente todo el tiempo, por lo menos se tiende a incrementar la proporción de tiempo en la cual la mayoría de ellos está ocupado.

Otra de las formas para mejorar el tiempo de ejecución consiste en utilizar la realimentación en la estructura de procesadores. Se analiza esta alternativa para el caso del producto de una matriz por un vector, donde la realimentación en el arreglo lineal se establece entre el procesador P_1 y el procesador P_n, tal como lo muestra la Fig. 6.23. De esta manera, los datos de salida del procesador P_n se convierten en los datos de entrada del procesador P_1. En el principio de la ejecución, cuando ningún procesador tiene datos en su memoria local, la entrada del procesador P_1 no proviene del procesador P_n sino del exterior de la estructura. De forma análoga, al finalizar la ejecución, cuando todos los resultados ya han sido calculados, la salida del procesador P_n no se envía al procesador P_1 sino al exterior de la estructura de procesadores.

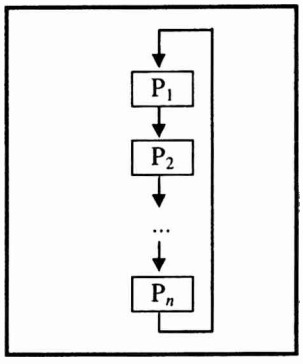


Figura 6.23: Anillo de Procesadores.

Al convertir el arreglo lineal de procesadores en un *anillo* se puede obtener el producto de la matriz por el vector en n pasos, si se cumplen las siguientes dos condiciones, tal como se muestra en la Fig. 6.24:

- El vector X está previamente cargado en el anillo de procesadores, de forma tal que el valor x_j se encuentra en el procesador P_{n-j+1}. Es como si el vector X se ingresara en el anillo con n pasos de comunicación.
- Cada fila i de la matriz A se rota circularmente a izquierda i veces.

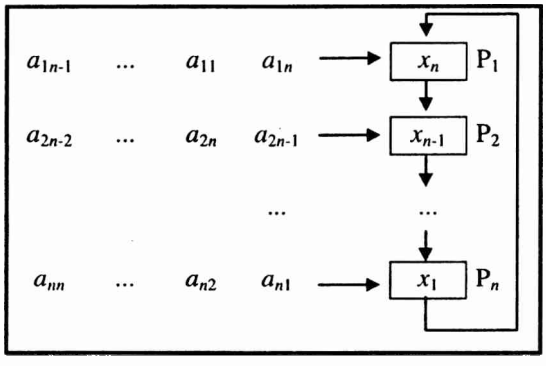


Figura 6.24: Multiplicación de una Matriz por un Vector en un Anillo.

En el esquema de la Fig. 6.24, en cada paso de ejecución cada procesador P_i recibe los datos, realiza un producto, acumula el resultado en la memoria local y transfiere el valor correspondiente a un x_i por el enlace inferior.El esquema de realimentación que convierte un arreglo lineal en un anillo puede extenderse para el caso del producto de dos matrices. La Fig. 6.25 muestra la estructura *toroide* resultante.

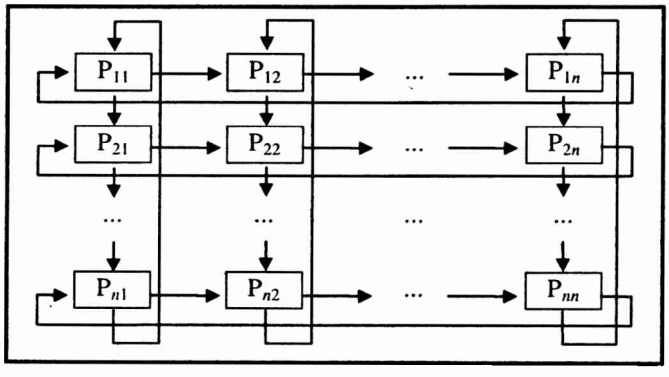


Figura 6.25: Estructura Toroide de Procesadores.

En cada paso de ejecución los valores de la matriz A se mueven hacia la derecha, mientras que los valores de la matriz B lo hacen hacia abajo. Las filas y las columnas de las matrices A y B respectivamente se disponen de manera similar a la mostrada para la multiplicación de una matriz por un vector. Luego de n pasos de ejecución, cada procesador P_{ij} contiene el valor correspondiente de la matriz C, c_{ij}, para todo $i = 1, ..., n$ y $j = 1, ..., n$. Por lo tanto, todo el procesamiento que se necesita para multiplicar dos matrices se completa en n pasos de ejecución. En este caso, aunque el factor de Speed-Up es $O(n)$ tal como para la estructura bidimensional de la Fig. 8.21, el valor del factor de Speed-Up será mayor. Esto de se debe a que la cantidad de pasos de ejecución de la solución paralela es menor, está dada por $g(n) = n$.

7. Algoritmos Clásicos: Grafos, Ruteo, Imágenes

En este capítulo se dejarán de lado los algoritmos paralelos de cálculo que operan sobre números (suma, ordenación, producto, etc.), para enfocar detalladamente la atención en clases de problemas que se acercan a otras aplicaciones, no numéricas. En cierto sentido, los problemas a tratar en este capítulo son de más alto nivel de abstracción que los presentados en el capítulo anterior porque resuelven problemas más cercanos a los reales. Aún así, la forma en que se proponen soluciones a estos problemas en una máquina paralela y el análisis que se hace de las soluciones es muy similar a lo que ya se ha presentado al respecto.

Se analizarán problemas clásicos encontrados en estructuras de tipo grafo, en el tratamiento de mensajes distribuidos en una red estática de procesadores y algunos algoritmos de procesamiento de imágenes, poniendo énfasis en su paralelización. Todos estos problemas se han estudiado ampliamente desde el punto de vista algorítmico en general y también desde el punto de vista de los algoritmos paralelos. A medida que se ha desarrollado y ampliado la capacidad de las computadoras en general y de las computadoras paralelas en particular, siempre se ha buscado la forma de aprovechar las nuevas características en algunas o a todas estas áreas de aplicación.

Los problemas que se encuentran en estructuras de tipo grafo se han tratado en el contexto de las estructuras de datos [Aho88] [Wei91] [Mel84] y también en el contexto del flujo en redes y de la teoría de grafos [Eve79] [For62] [Har69]. El manejo de los mensajes distribuidos en una red de procesadores es clásico en el contexto de transmisión de datos en las redes de interconexión [Val81] [Kri88] [Gra89]. Los problemas de procesamiento de imágenes se iniciaron en el contexto del procesamiento de señales, pero por su magnitud ya pueden ser considerados como una clase de problemas en sí mismos [Gon92] [Hus91] [Jai89].

Los problemas que se presentan sobre grafos tienen la particularidad de tener una solución en computadoras paralelas que, sino idéntica, es muy similar para todos. Esto no se puede extender a *toda* la clase de algoritmos sobre grafos, pero por lo menos es de destacar en el contexto de aplicación de los algoritmos paralelos.

En uno de los capítulos anteriores se han visto con bastante nivel de detalle las características generales y también ejemplos significativos de las redes de interconexión. Para algunas redes se presentaron algoritmos de manejo de la transmisión de mensajes, pero más como una característica ventajosa de esas redes que como un problema a resolver. En este capítulo se intenta describir el problema y algunas posibles soluciones con respecto al ruteo de paquetes en una red de interconexión. También se analizarán las soluciones propuestas desde el punto de vista del rendimiento.

Como ya se ha afirmado, el área del procesamiento de imágenes es muy amplia tanto en aplicaciones como en soluciones computacionales. En este caso también se intenta la identificación de las características comunes a varios problemas y soluciones aplicando computadoras paralelas. Los tres problemas que se han elegido para la descripción, presentación de soluciones aplicando computadoras paralelas y análisis de las soluciones son: labelling de componentes en imágenes, identificación de patrones geométricos simples y compresión.

7.1 Algoritmos sobre Grafos

Un grafo dirigido $G = (V, E)$ está formado por un conjunto V de n nodos y un conjunto E de arcos dirigidos entre los nodos de V. Los nodos se identifican con números enteros, es decir $i = 1, ..., n$ y los arcos dirigidos se definen con pares ordenados de la forma (i, j) si hay un arco dirigido desde el nodo i al nodo j. La Fig. 7.1 muestra un grafo con siete nodos, el conjunto de nodos V y el conjunto de arcos dirigidos E.

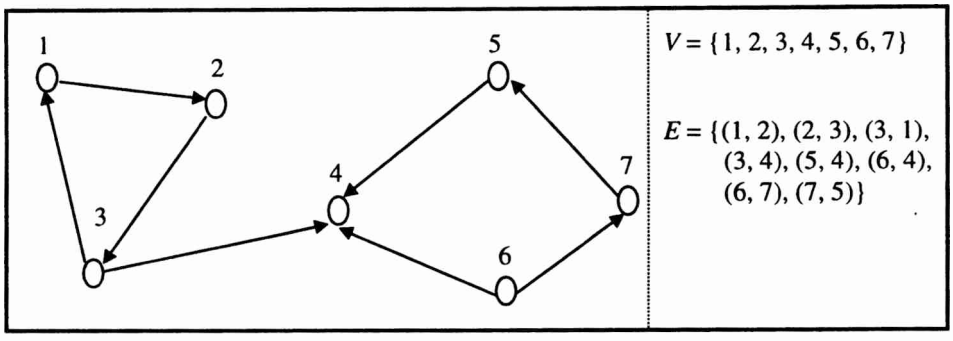


Figura 7.1: Grafo Dirigido, Conjuntos de Nodos y de Arcos Dirigidos.

La matriz de adyacencia de un grafo G tal como se ha definido es $A = a_{ij}$, con $i = 1, ..., n$ y $j = 1, ..., n$; y representa las conexiones entre nodos pues $a_{ij} = 1$ si existe un arco dirigido desde el nodo i al nodo j, $a_{ij} = 0$ en caso contrario. La Fig. 7.2 muestra la matriz de adyacencia para el grafo de siete nodos de la Fig. 7.1.

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Figura 7.2: Matriz de Adyacencia.

Los valores (binarios) que se asignan en cada posición de la matriz de adyacencia se pueden expresar también como: se define que $a_{ij} = 1$ si $(i, j) \in E$, y $a_{ij} = 0$ en caso contrario.

Tal como se puede apreciar en la Fig. 7.2 se considera que existen los arcos dirigidos (i, i) para todos los nodos $i = 1, ..., n$. Como ya se hizo en la Fig. 7.1 estos arcos dirigidos no se mostrarán explícitamente en las figuras ni en los conjuntos de arcos dirigidos.

7.1.1 Clausura Transitiva de un Grafo

La clausura transitiva G^* de un grafo G con matriz de adyacencia A es un grafo formado por

el mismo conjunto de vértices V del grafo G, pero con un conjunto de arcos dirigidos E^* tal que existe un arco entre los nodos i y j si en el grafo G existe un camino desde el nodo i al nodo j. La Fig. 7.3 muestra G^* para el grafo de la Fig. 7.1.

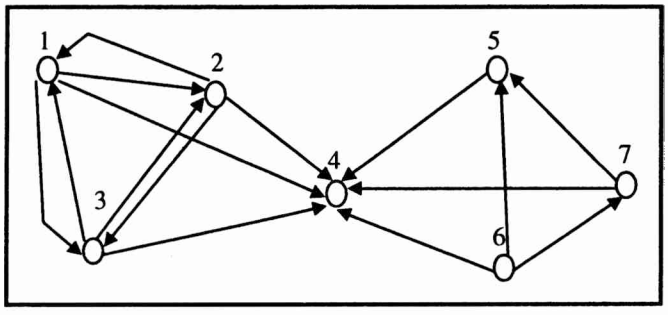


Figura 7.3: Clausura Transitiva.

La Fig. 7.4 muestra el conjunto de arcos dirigidos E^* y la matriz A^* de la clausura transitiva del grafo de la Fig. 7.1.

$$A^* = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

$E^* = \{(1, 2), (2, 3), (3, 1), (3, 4), (5, 4), (6, 4), (6, 7), (7, 5), (1, 3), (1, 4), (2, 1), (2, 4), (3, 2), (3, 4), (6, 5), (7, 4)\}$

Figura 7.4: Matriz A^* y Conjunto E^* de la Clausura Transitiva.

El algoritmo para obtener la clausura transitiva de un grafo $G = (V, E)$ con matriz de adyacencia A tiene n fases. Se comienza con el grafo original G:

- En la primera fase se inserta el arco (i, j) en G^* si los arcos $(i, 1)$ y $(1, j)$ están en el grafo original.
- En la segunda fase se inserta el arco (i, j) en G^* si los arcos $(i, 2)$ y $(2, j)$ están en el grafo formado en la fase anterior.
- En general, en la fase h se inserta el arco (i, j) en G^* si los arcos (i, h) y (h, j) están en el grafo formado en la fase h-1. Notar que en la etapa h se agrega un arco entre dos nodos si existe un camino que los vincule que pase por *los h primeros nodos* del grafo.
- Al llegar a la fase n se ha obtenido G^*.

El algoritmo anterior se puede definir por el procesamiento sobre la matriz de adyacencia A del grafo G hasta llegar a la matriz de adyacencia A^* correspondiente a la clausura transitiva:

- En la primera fase se genera la matriz de adyacencia $A^{(1)} = a_{ij}^{(1)}$. El valor de cada $a_{ij}^{(1)}$ se puede definir como $a_{ij}^{(1)} = a_{ij} \vee (a_{i1} \wedge a_{1j})$.

- En la segunda fase se genera la matriz de adyacencia $A^{(2)} = a_{ij}^{(2)}$. El valor de cada $a_{ij}^{(2)}$ se puede definir como $a_{ij}^{(2)} = a_{ij}^{(1)} \vee (a_{i2}^{(1)} \wedge a_{2j}^{(1)})$.
- En general, en la fase h se genera la matriz de adyacencia $A^{(h)} = a_{ij}^{(h)}$. El valor de cada $a_{ij}^{(h)}$ se puede definir como $a_{ij}^{(h)} = a_{ij}^{(h-1)} \vee (a_{ih}^{(h-1)} \wedge a_{hj}^{(h-1)})$.
- Al llegar a la fase n se ha obtenido $A^{(n)} = A^*$ correspondiente a G^*.

La cantidad de fases del algoritmo es igual a n, la cantidad de nodos del grafo G. En cada fase del algoritmo se analiza si hay un nuevo arco dirigido entre dos nodos por haber un camino entre ellos que pase por otro nodo del grafo. El tipo de operaciones que se deben llevar a cabo en cada paso son operaciones binarias elementales: $\wedge$ y $\vee$. La cantidad de operaciones a realizar en cada paso es $O(n^2)$ porque se deben inspeccionar todos los posibles pares de nodos origen y destino de un nuevo arco dirigido. Como el algoritmo define n fases a realizar y la cantidad de operaciones de cada fase es $O(n^2)$, se llega a que la cantidad de pasos de ejecución de una implementación simple secuencial (monoprocesador) es de $O(n^3)$.

En la Fig. 7.5 se muestra la secuencia de siete fases para llegar a la clausura transitiva del grafo G de la Fig. 7.1.

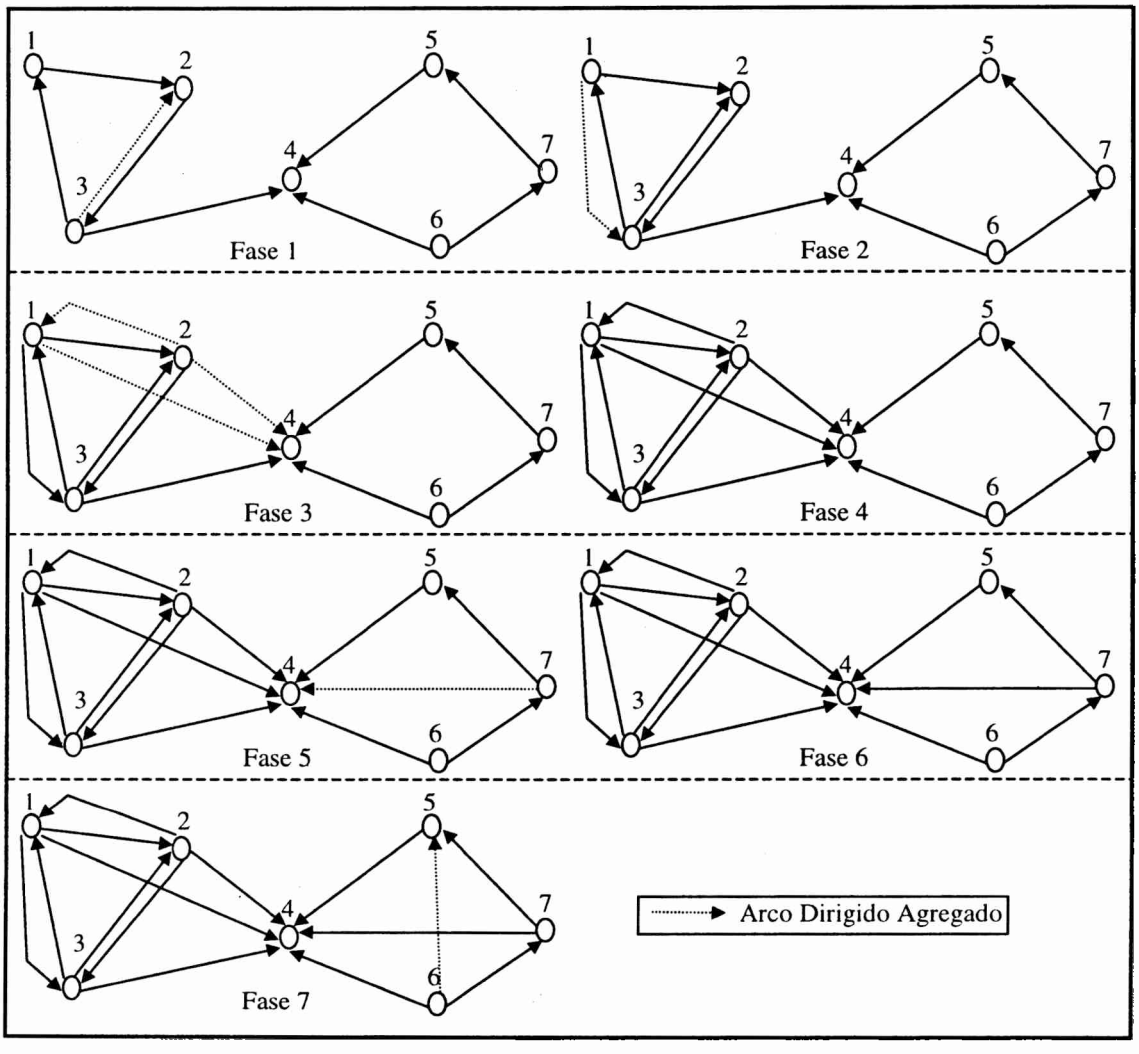


Figura 7.5: Fases del Algoritmo de Clausura Transitiva.

De acuerdo con la Fig. 7.5, se comienza en la primera fase con los arcos dirigidos del grafo original G, tal como el algoritmo lo establece. Los arcos dirigidos que se insertan en cada fase se muestran en la Fig. 7.5 con líneas de puntos para una mejor identificación del funcionamiento del algoritmo. En términos del grafo resultante, los arcos dirigidos que se agregan en cada fase del algoritmo tienen el mismo significado que los demás arcos. En los términos de la descripción del algoritmo en función de la matriz de adyacencia, al completarse la fase i, la matriz de adyacencia que corresponde al grafo es $A^{(i)}$. En las fases en que no se inserta ningún arco dirigido el grafo no se transforma, queda como en la fase anterior. En general, cuando en la fase h no se insertan nuevos arcos dirigidos se cumple que $A^{(h)} = A^{(h-1)}$. El grafo que resulta de la séptima fase (en general, cuando se completa la fase n), es G^*, la clausura transitiva del grafo original G.

Una implementación paralela del algoritmo presentado se puede realizar sobre un arreglo bidimensional de nxn procesadores. En vez de procesar los datos sobre la representación gráfica, se utilizará la matriz de adyacencia A del grafo $G = (V, E)$. Por lo tanto, se busca obtener la matriz A^* utilizando el arreglo bidimensional de procesadores.

La implementación sobre la máquina paralela se define en términos del movimiento de datos y de las operaciones que se realizan en cada paso. El movimiento de datos, a su vez, se puede definir como:

- La matriz de adyacencia A del grafo ingresa al arreglo de procesadores por filas, esto es, todos los valores de una fila a la vez, y la primera fila en ingresar es la primera fila de la matriz. La primera fila de la estructura de procesadores es la que recibe los datos de la matriz, y una vez que ingresaron, los datos continúan hacia las filas de procesadores que siguen.
- Cada fila i de la matriz que entra en la estructura de procesadores es transmitida hasta llegar a la fila i de procesadores libres (que está libre), donde se almacena. En este sentido, cada vez que una fila i entra, "pasa por sobre" las i-1 filas anteriores ya almacenadas en el arreglo bidimensional hasta llegar a la fila i de procesadores, donde queda almacenada. En términos de cantidad de pasos, la fila i de la matriz A alcanzará la fila i de procesadores luego de $2i - 1$ pasos.
- La fila i de la matriz de adyacencia queda almacenada en la fila i de procesadores hasta que la fila anterior (i-1) pasa sobre ella. En el caso de la primera fila, queda almacenada hasta que la fila n (la última) pasa sobre ella. Una vez que esto sucede (la fila "i-1" de la matriz pasa sobre la fila i), la fila de la matriz comienza a transmitirse nuevamente hacia las filas de procesadores siguientes hasta salir de la estructura por la fila n de procesadores. En general, la fila i de la matriz comenzará a moverse nuevamente en el paso $2i$-1+n y terminará de salir del arreglo de procesadores en el paso $2n+i-1$. Por lo tanto, la fila n de la matriz saldrá de la estructura de procesadores al completarse el paso $3n$-1 y entonces toda la matriz habrá sido procesada.

La Fig. 7.6 muestra la secuencia de pasos para una matriz de 7x7 correspondiente a un grafo de siete nodos como el de la Fig. 7.1. La fila i-ésima de la matriz de adyacencia A se denota como A_i. Por ejemplo, A_3 denota la tercera fila de la matriz A. El movimiento de los datos ya sugiere por sí mismo una idea del tipo de procesamiento paralelo a llevar a cabo. De la misma forma, la Fig. 7.6 indica que la matriz de procesadores debe tener tantas filas como filas tiene la matriz de adyacencia A. Como mínimo, los procesadores deben conectarse de forma tal que los datos de la matriz puedan transmitirse desde la fila 1 hacia la fila n de la estructura bidimensional.

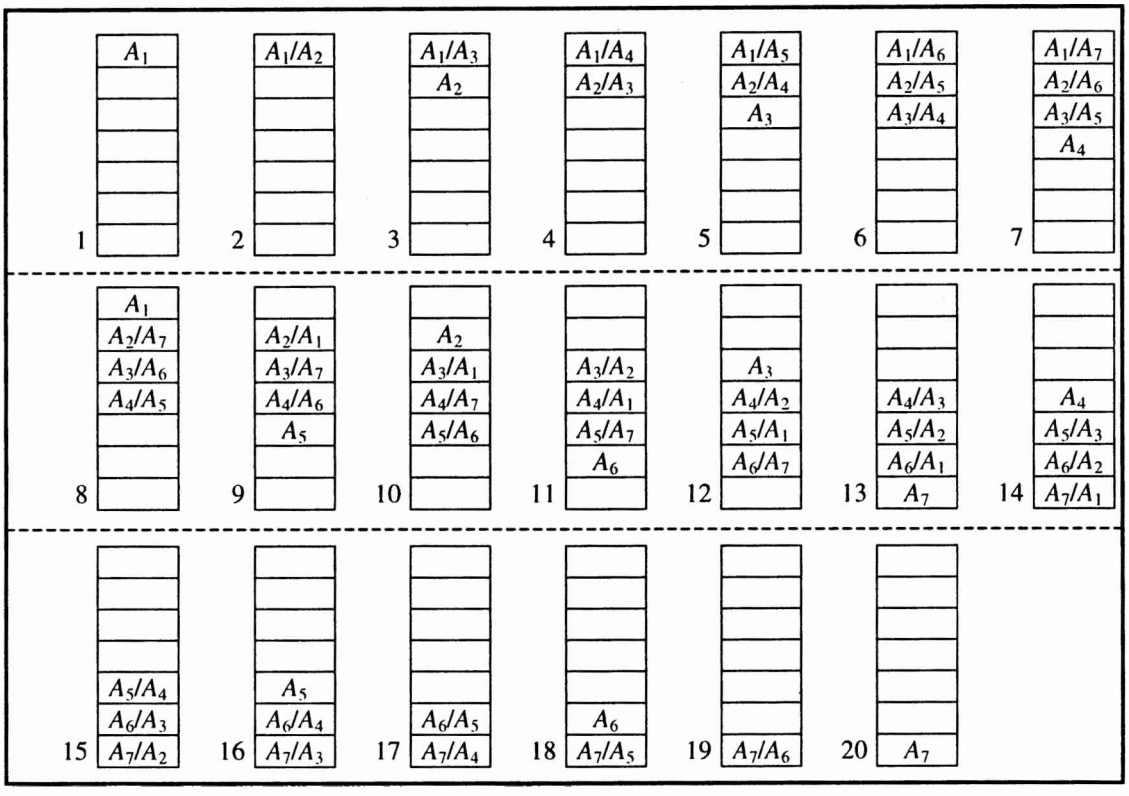


Figura 7.6: Movimiento de Filas de la Matriz de Adyacencia.

Una vez que se tiene definido el movimiento de los datos de la matriz de adyacencia se pueden describir y analizar las operaciones a realizar en cada paso para implementar el algoritmo:

- La primera fase del algoritmo se lleva a cabo cuando las filas 2 a n de la matriz pasan sobre la fila 1 almacenada en la primera fila de procesadores. Cuando la fila i pasa sobre la fila 1, el procesador P_{11} envía el valor de a_{i1} a todos los demás procesadores de la fila 1. De este modo, si el procesador P_{1j} tiene almacenado un 1 y a_{i1} es 1, el valor $a_{ij}^{(1)}$ que está pasando sobre el procesador P_{1j} se fija en 1. Cuando las filas 2 a n de la matriz A pasan sobre la primera fila se obtiene $A^{(1)}$, es decir la matriz de adyacencia en correspondencia con la primera fase del algoritmo.
- En general, la fase h se completa cuando las filas 1 a h-1 de la matriz de adyacencia pasan sobre la fila h que está almacenada en la fila h de procesadores. Cuando la fila h de la matriz de adyacencia alcanza la fila h del arreglo de procesadores, ha sido actualizada pasando sobre las filas 1 a h-1 y, por lo tanto, corresponde a la fase h-1 del algoritmo de clausura transitiva. De hecho, en esta fila h los elementos en 1 indican caminos formados por los arcos que unen nodos entre 1 y h del grafo original G. En la fase h, cuando la fila i de la matriz de adyacencia pasa sobre la fila h actualizada, el procesador P_{hh} transmite el valor $a_{ih}^{(h-1)}$ a todos los demás procesadores de la fila h y si existe un procesador P_{hj} con $a_{hj}^{(h-1)} = 1$ y $a_{ih}^{(h-1)} = 1$ entonces se fija $a_{ij}^{(h)} = 1$ actualizando la fila i de la matriz que está pasando.
- Al completarse el paso $3n-1$, la matriz A^* ha salido por la parte inferior del arreglo de procesadores (por la última fila de procesadores).

Para cualquier fila h de procesadores, un paso de ejecución de la fase h abarca las

siguientes operaciones básicas:

1. Recibir y almacenar una fila i de datos de la matriz de adyacencia, provenientes de la fila anterior de procesadores. En el caso de la primera fila de procesadores, recibe los datos de entrada de la red de procesadores.
2. El procesador P_{hh} transmite el valor de la matriz de adyacencia $a_{ih}^{(h-1)}$ a todos los demás procesadores de la misma fila.
3. Cada procesador computa $a_{ij}^{(h)} = a_{ij}^{(h-1)} \vee (a_{ih}^{(h-1)} \wedge a_{hj}^{(h-1)})$.
4. Transmitir la fila i de datos de la matriz de adyacencia a la fila siguiente de procesadores del arreglo bidimensional. La última fila de procesadores es la que efectúa la salida de la matriz A^*.

Las excepciones a esta descripción de operaciones se producen cuando la fila h de la matriz de adyacencia llega a la fila h de procesadores (solamente se almacena) y cuando la abandona (solamente se transmite). Estas excepciones se pueden visualizar claramente en la Fig. 7.6. Todas las operaciones se llevan a cabo cuando una fila de la matriz de adyacencia pasa sobre otra almacenada en el arreglo bidimensional de procesadores.

En la descripción de la implementación paralela y en la Fig. 7.6 en particular se puede notar cómo se procesa en paralelo por el pasaje de varias filas sobre otras al mismo tiempo. En el paso 7 de la Fig. 7.6 por ejemplo, hay tres filas de procesadores ejecutando pasos de diferentes fases al mismo tiempo. En particular, la primera fila de procesadores actualiza la fila 7 de la matriz $A^{(1)}$, la segunda fila de procesadores actualiza la fila 6 de la matriz $A^{(2)}$ y la tercera fila de procesadores actualiza la fila 5 de la matriz $A^{(3)}$.

En definitiva se ha obtenido una implementación paralela del algoritmo de cálculo de la clausura transitiva de un grafo que es $O(n)$. Las soluciones secuenciales son $O(n^3)$ [Lei92], por lo tanto, con n^2 procesadores se obtiene un factor de Speed-Up que es $O(n^2)$.

Un buen ejercicio para el lector puede ser seguir la implementación paralela del algoritmo con la Fig. 7.6 y los datos de la matriz de adyacencia A de la Fig. 7.2, para obtener A^*.

7.1.2 Pipelining

De acuerdo con el movimiento de los datos (filas de la matriz de adyacencia) que establece la implementación paralela del algoritmo, y tal como se ve en la Fig. 7.6, las primeras filas de la estructura bidimensional de procesadores son las que se utilizan primero. También son las primeras filas de procesadores las que quedan libres una vez que la matriz de adyacencia ha sido procesada pasando por ellas. A medida que se suceden los pasos de ejecución, el procesamiento se "traslada" a las filas inferiores del arreglo bidimensional de procesadores dejando inactivas las primeras filas. Siguiendo la Fig. 7.6 por ejemplo, a partir de la finalización del paso ocho la primera fila de procesadores queda libre, y a partir de la finalización del paso diez queda libre la segunda. Por un lado, esto puede ser considerado ineficiente, y por otro se podría aprovechar para llevar a cabo procesamiento pipeline de varias clausuras transitivas de grafos dirigidos.

Siempre que la primera fila de procesadores queda libre, se puede comenzar el procesamiento de una nueva clausura transitiva de un grafo utilizando su matriz de

adyacencia. De esta forma, no solamente se siguen utilizando los procesadores de las primeras filas del arreglo bidimensional, sino que se utilizan todos a la vez, porque todos tienen datos para procesar. La Fig. 7.7 muestra el procesamiento pipeline con matrices de 3x3 siguiendo esta idea.

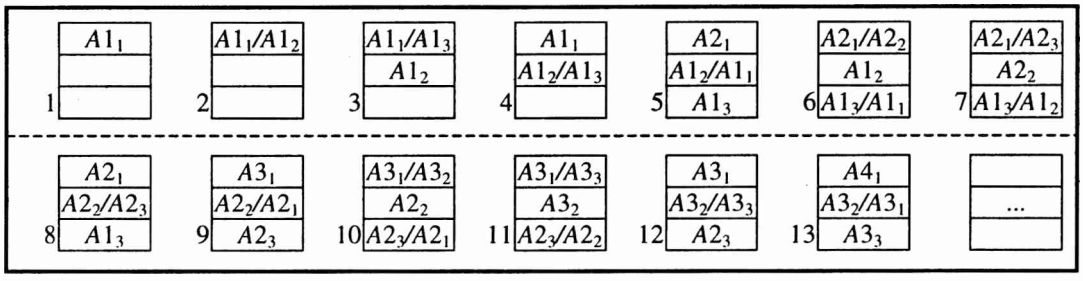


Figura 7.7: Procesamiento Pipeline de Matrices de 3x3.

En la Fig. 7.7, Ah_i denota la fila i-ésima de la h-ésima matriz de adyacencia. Por ejemplo, $A2_3$ denota la tercera fila de la segunda matriz de adyacencia que se procesa en el arreglo bidimensional de procesadores.

La primera matriz de adyacencia a procesar, $A1$, comenzará la entrada (la primera fila por la primera fila de procesadores), en el primer paso de ejecución. Esta primera matriz se transformará en la clausura transitiva del grafo correspondiente, y saldrá del arreglo bidimensional de procesadores al completarse el paso $3n$-1, como ya se ha explicado. Si se denota con $E(h)$ el paso de ejecución en el que la matriz h-ésima comienza la entrada (ingresa la primera fila) y con $S(h)$ el paso de ejecución en el que completa la salida (egresa la fila n de la matriz), entonces

$$E(1) = 1 \tag{7.1}$$

y

$$S(1) = 3n\text{-}1 \tag{7.2}$$

En general, la primera fila de la matriz Ah no puede ingresar a la estructura de procesadores inmediatamente después de la entrada de la última fila de la matriz $A(h\text{-}1)$. Además, y de acuerdo con el movimiento de los datos (filas de la matriz de adyacencia), la diferencia en el tiempo de entrada entre las matrices será también la diferencia en el tiempo de salida de las clausuras transitivas. Esto significa que si la matriz Ah ingresa a la estructura de procesadores d pasos de ejecución después del ingreso de la matriz $A(h\text{-}1)$, la matriz Ah^* correspondiente a la clausura transitiva de Ah terminará de salir de la estructura d pasos de ejecución después de completarse la salida de $A(h\text{-}1)^*$, la matriz correspondiente a la clausura transitiva de $A(h\text{-}1)$. Expresado de otra forma, se cumple que

$$S(h) = S(h\text{-}1) + (E(h) - E(h\text{-}1)), \quad \forall\, h > 1 \tag{7.3}$$

donde

$$E(h) - E(h\text{-}1) = d, \quad \forall\, h > 1 \tag{7.4}$$

Siguiendo la Fig. 7.7 se puede verificar la Ec. (7.3), ya que $E(1) = 1$, $E(2) = 5$, y $E(3) = 9$; y $S(1) = 8$, $S(2) = S(1) + (E(2) - E(1)) = 12$, y $S(3) = S(2) + (E(3) - E(2)) = 16$;

siguiendo tres pasos más de los que se muestran en la figura.

La cantidad de pasos de ejecución que debe haber entre el comienzo de la entrada de la matriz Ah y el comienzo de la entrada de la matriz $A(h\text{-}1)$ es la cantidad de pasos de ejecución que la primera fila de procesadores está ocupada procesando la matriz $A(h\text{-}1)$.

La Fig. 7.8 muestra la secuencia de pasos que está ocupada la primera fila de procesadores desde que una matriz de adyacencia A comienza la entrada a la estructura de procesadores en el i-ésimo paso de ejecución. Nuevamente, A_i denota la fila i-ésima de la matriz de adyacencia A.

Paso de Ejec.	Fila 1 de Proc.
i	A_1
i+1	A_1/A_2
i+2	A_1/A_3
i+3	A_1/A_4
...	...
$i+(n\text{-}1)$	A_1/A_n
$i+n$	A_1
$i+(n+1)$	*Libre*

Figura 7.8: Ocupación de la Primera Fila de Procesadores.

Según la Fig. 7.8, la primera fila de procesadores queda libre en el paso de ejecución $i+(n+1)$ si una matriz de adyacencia comienza su entrada en el arreglo de procesadores en el paso de ejecución i. Por lo tanto, la matriz de adyacencia Ah puede comenzar su ingreso $d = n+1$ pasos de ejecución después del comienzo del ingreso de la matriz $A(h\text{-}1)$. Expresado de otra forma,

$$E(h) = E(h\text{-}1) + (n+1), \quad \forall\, h > 1 \tag{7.5}$$

Se puede volver a la Fig. 7.7 para verificar la Ec. (7.5): $n = 3$ y $E(1) = 1$, $E(2) = E(1) + 4 = 5$, $E(3) = E(2) + 4 = 9$, y $E(4) = E(3) + 4 = 13$.

En general, utilizando iterativamente la Ec. (7.5) se tiene que

$$E(h) = E(h\text{-}1) + (n+1) = E(h\text{-}2) + 2(n+1) = \ldots = E(1) + (h\text{-}1)(n+1), \quad \forall\, h > 1 \tag{7.6}$$

Es decir que, recurriendo a la Ec. (7.1), se tiene

$$E(h) = (h\text{-}1)(n+1) + 1, \quad \forall\, h > 1 \tag{7.7}$$

De forma análoga, utilizando la Ec. (7.3) y la Ec. (7.5) se llega a que

$$S(h) = S(h\text{-}1) + (n+1) = S(h\text{-}2) + 2(n+1) = \ldots = S(1) + (h\text{-}1)(n+1), \quad \forall\, h > 1 \tag{7.8}$$

Y utilizando la Ec. (7.2) se obtiene

$$S(h) = (h\text{-}1)(n+1) + (3n\text{-}1), \quad \forall\, h > 1 \tag{7.9}$$

La Ec. (7.9) nuevamente se puede verificar en el ejemplo de la Fig. 7.7, ya que $n = 3$ y además $S(1) = 8$, $S(2) = 4 + 8 = 12$, y $S(3) = 8 + 8 = 16$, siguiendo tres pasos más de los que se muestran en la figura.

Con el análisis realizado hasta aquí se puede calcular ahora la cantidad de pasos de ejecución necesarios para llevar a cabo la clausura transitiva de los grafos G_1, G_2, ..., G_k, con matrices de adyacencia $A1$, $A2$, ... Ak de nxn elementos. Esta cantidad de pasos está dada por la salida de la última matriz de adyacencia, Ak^*. Se puede decir que se computan k clausuras transitivas sobre matrices de nxn al completarse el último paso de salida de la matriz de adyacencia k-ésima, es decir en el paso $g_p(n) = S(k) = (k\text{-}1)\ (n+1) + (3n\text{-}1)$. Si las clausuras transitivas se calculan de forma secuencial en el arreglo de procesadores, es decir que el procesamiento de G_h comienza cuando está totalmente terminado el procesamiento de $G_{h\text{-}1}$, se tienen $g_s(n) = k\ (3n\text{-}1)$ pasos de ejecución. La ganancia en velocidad por el procesamiento pipeline está dada por $g_s(n)/g_p(n)$ sin utilizar ningún recurso adicional de hardware en el arreglo bidimensional de procesadores. Si, por ejemplo, $n = 100$ y $k = 20$ el procesamiento pipeline hace que se obtengan las clausuras transitivas aproximadamente 2,7 veces más rápido que si se llevaran a cabo de forma secuencial. Si $n = 1000$ y $k = 500$, el factor de ganancia en velocidad es aproximadamente 3.

7.1.3 Consideraciones de Hardware

Con respecto al hardware necesario para realizar el cálculo de la clausura transitiva de grafos dirigidos, se ha detallado que se utiliza un arreglo bidimensional de procesadores interconectados. En esta sección se avanzará en el nivel de detalle con respecto a la complejidad de cada procesador y, lo que es más interesante, la red de interconexiones.

Los requerimientos de cada procesador con respecto a la capacidad de ejecutar operaciones y almacenar datos surgen claramente de las operaciones que se ejecutan en cada paso de la implementación. Como se ha detallado, en cada paso de la fase h se debe resolver $a_{ij}^{(h)} = a_{ij}^{(h-1)} \vee (a_{ih}^{(h-1)} \wedge a_{hj}^{(h-1)})$. Por lo tanto, cada procesador debe tener la capacidad de almacenar tres datos y ejecutar las operaciones binarias $\wedge$ y $\vee$.

La red de interconexión de procesadores necesaria para la ejecución tiene detalles más interesantes que la capacidad de cada procesador para ejecutar instrucciones y almacenar operandos. De acuerdo con la definición del movimiento de los datos (filas de la matriz de adyacencia) en el arreglo, y siguiendo la Fig. 7.6, cada procesador debe tener dos enlaces: un enlace superior, por el cual recibir un dato de la matriz de adyacencia y un enlace inferior para transmitir el dato "transformado". El enlace superior de cada procesador P_{ij} lo comunica con el procesador correspondiente de la fila anterior, es decir con el procesador $P_{(i-1)j}$. En el caso de los procesadores de la primera fila, el enlace superior los habilita para llevar a cabo la entrada de los datos desde el exterior del arreglo de procesadores. El enlace inferior de cada procesador P_{ij} lo comunica con el procesador correspondiente de la fila siguiente, es decir con el procesador $P_{(i+1)j}$. En el caso de los procesadores de la última fila (n), el enlace inferior los habilita para llevar a cabo la salida de los datos hacia el exterior del arreglo de procesadores. La Fig. 7.9 muestra los enlaces necesarios para llevar a cabo el movimiento de las filas de la matriz de adyacencia entre las filas de procesadores.

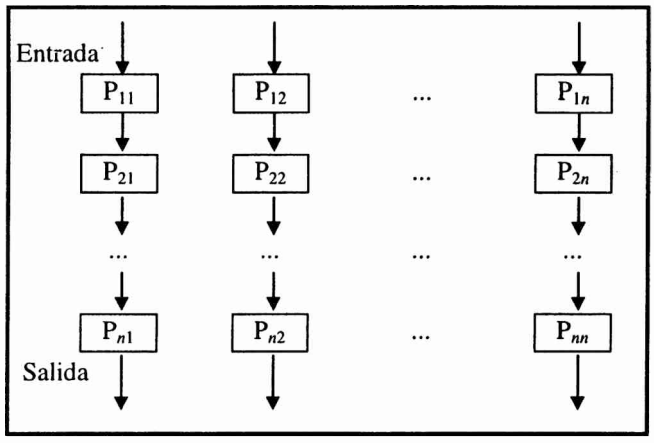


Figura 7.9: Enlaces Superiores e Inferiores del Arreglo Bidimensional.

Según la descripción de las operaciones que se llevan a cabo en cada paso de ejecución, cada procesador de la diagonal del arreglo bidimensional debe transmitir el valor de la matriz de adyacencia recibido hacia todos los demás procesadores de la misma fila. En el análisis de la cantidad de pasos de ejecución se asume que este multicast (o broadcast en [Lei92]), es $O(1)$, es decir que no depende de la cantidad de procesadores hacia los cuales transmitir.

Una de las primeras opciones para la comunicación de cada P_{ii} con los demás procesadores de la fila consiste en definir y utilizar enlaces de comunicación hacia los vecinos inmediatos a la izquierda y derecha de cada procesador. Esto significa que cada procesador P_{ij} está comunicado con el procesador $P_{i(j+1)}$ y con el procesador $P_{i(j-1)}$. El sentido de la transmisión debe ser tal que el dato se envíe desde P_{ii} hacia P_{i1}, y desde P_{ii} hacia P_{in}. La Fig. 7.10 muestra los enlaces y el sentido de la comunicación siguiendo este esquema para una red de 4x4 procesadores, es decir que en este caso $n = 4$. Se puede ver que para llegar desde el procesador P_{11} al procesador P_{14} se necesitan tres comunicaciones, como para llegar desde el procesador P_{44} al procesador P_{41}.

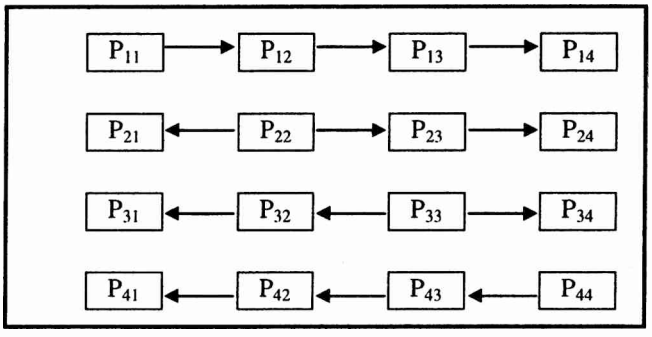


Figura 7.10: Enlaces a Izquierda y Derecha del Arreglo Bidimensional.

Siguiendo el esquema de conexiones por enlaces hacia los procesadores vecinos no es difícil calcular la cantidad de pasos secuenciales de comunicación necesarios para enviar un dato desde cada procesador P_{ii} hacia todos los demás de la misma fila. Esta cantidad de comunicaciones depende de la cantidad de procesadores que se utilicen y está dada por

$$|i - j|$$

si el procesador P_{ii} debe enviar un dato al procesador P_{ij}. Tal como se ha definido el arreglo bidimensional, cada fila es de n procesadores, por lo tanto, el tiempo que lleva realizar este multicast es $O(n)$. De esta manera, la utilización de enlaces hacia los procesadores vecinos debe descartarse si se quiere/necesita mantener la cantidad de pasos de procesamiento en $O(n)$.

Una segunda alternativa para comunicar un dato desde cada procesador P_{ii} hacia todos los demás de la misma fila puede ser que todos ellos compartan una memoria común. De acuerdo con esta idea, el procesador P_{ii} almacena el dato en la memoria compartida y luego cada procesador P_{ij} (j = 1, ..., i-1, i+1, ..., n), puede leer simultáneamente de la memoria para utilizar el dato. Este "multicast" es de orden constante, $O(1)$, porque implica llevar a cabo dos operaciones que son elementales: un almacenamiento en memoria y una lectura de memoria. Con respecto al hardware necesario, se debería agregar la memoria compartida y sincronizar el acceso de los procesadores. El sincronismo (primero el procesador P_{ii} debe escribir y luego todos los demás de la misma fila deben realizar la lectura) no representa un problema porque de hecho todo el arreglo bidimensional de procesadores puede ejecutar sincronizado por un mismo reloj. El hardware necesario para agregar una memoria compartida por fila de procesadores es significativo, porque la cantidad de memorias compartidas y de buses de comunicación desde los procesadores hacia esas memorias depende de n, la cantidad de nodos de los grafos que se procesan.

Es posible que siguiendo la idea de la memoria compartida se pueda llegar a una tercera opción, menos costosa en términos de hardware necesario para implementarla. Esta tercera posibilidad surge de observar que la memoria compartida interviene en una sola "transacción": un procesador escribe y los demás leen. Desde este punto de vista, la memoria no es más que un canal de comunicaciones unidireccional 1 a n. El procesador P_{ii} envía un dato que todos los demás procesadores en la misma fila reciben. De acuerdo con esto, la memoria compartida puede ser reemplazada de forma inmediata por un bus de comunicaciones. El resultado es el mismo y los requerimientos de hardware se reducen de forma considerable: un bus compartido por fila de procesadores.

La Fig. 7.11 muestra cómo quedaría interconectado el arreglo bidimensional donde cada procesador P_{ij} posee los enlaces superior e inferior que se muestran en la Fig. 7.9 y la conexión al bus de comunicaciones.

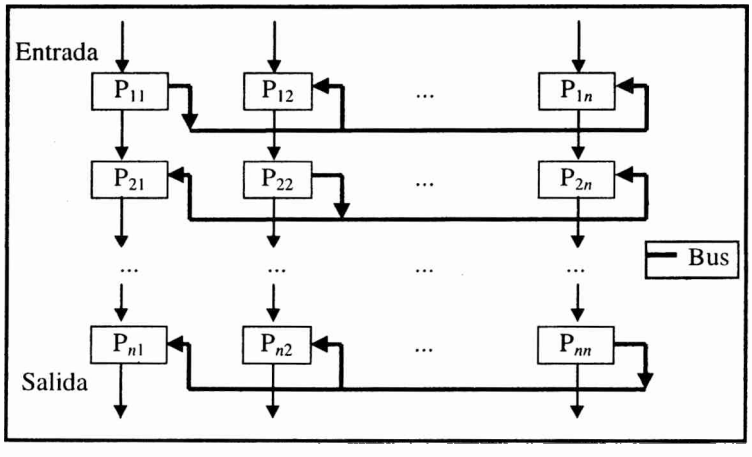


Figura 7.11: Red de Interconexión del Arreglo Bidimensional.

Como ya se ha explicado, en la Fig. 7.11 el enlace superior de cada procesador P_{ij} se utiliza para recibir datos del procesador $P_{(i-1)j}$ y el enlace inferior para enviar datos al procesador $P_{(i+1)j}$. Ambos enlaces se necesitan para llevar a cabo el movimiento de las filas de la matriz de adyacencia sobre la cual se procesa. El bus de comunicaciones es común para todos los procesadores de una misma fila, es decir que todos los procesadores de cada fila i-ésima tienen acceso al mismo bus de comunicaciones. En el contexto del problema a resolver (clausura transitiva de un grafo dirigido), la conexión al bus de comunicaciones se utiliza para que el procesador perteneciente a la diagonal del arreglo bidimensional, P_{ii}, envíe un dato a todos los demás procesadores P_{ij} (j = 1, ..., i-1, i+1, ..., n), en la misma fila. Esta operación se ejecuta en $O(1)$ cantidad de pasos, es decir independientemente de n, la cantidad de elementos del grafo a procesar y de procesadores en una misma fila. En la Fig. 7.11 también se muestra el sentido en el que se transmite la información por el bus de comunicaciones.

Con la red de interconexión de la Fig. 7.11 se logra ejecutar la clausura transitiva de un grafo G en $O(n)$ pasos y también se pueden procesar las clausuras transitivas de k grafos dirigidos G_1, ..., G_k de modo pipeline tal como se ha explicado.

7.1.4 Determinación de Componentes Conectadas en un Grafo

Un grafo no dirigido $G = (V, F)$ está formado por un conjunto V de n nodos y un conjunto F de arcos no dirigidos entre los nodos de V. Es similar a un grafo dirigido, pero los arcos son tales que, si existe un arco entre el nodo i y el nodo j se utiliza en los dos sentidos: desde el nodo i al nodo j y desde el nodo j al nodo i. La matriz de adyacencia $A = a_{ij}$, con i = 1, ..., n y j = 1, ..., n, de un grafo no dirigido es tal que

$$A = A^T \tag{7.10}$$

donde A^T es la matriz traspuesta de A, es decir que $A^T = a_{ij}^T = a_{ji}$.

Dado un grafo no dirigido $G = (V, F)$, los componentes conectados Cc de G son los conjuntos de nodos que se pueden formar con los nodos de G, donde en cada conjunto están los vértices entre los cuales existe un camino de conexión en G. Si Cc_1, Cc_2 y Cc_3 son los componentes conectados de G, se cumplen

$$Cc_i \subset V, \quad \forall\, i = 1, 2, 3$$

$$Cc_i \cap Cc_j = \varnothing, \quad \text{con } 1 \le i < j \le 3$$

$$\bigcup_{i=1}^{3} Cc_i = V$$

En la Fig. 7.12 se muestra un grafo no dirigido de siete nodos y los conjuntos Cc que le corresponden. Se puede notar claramente que desde cualquier nodo h en un conjunto Cc_i se puede llegar a cualquier otro nodo k en el mismo conjunto Cc_i. Si no existe un arco directo entre los nodos h y k, entonces se pueden utilizar los arcos existentes entre los nodos pertenecientes a Cc_i en cualquier sentido partiendo desde el nodo h y pasando por todos los nodos intermedios que sean necesarios hasta llegar al nodo k..

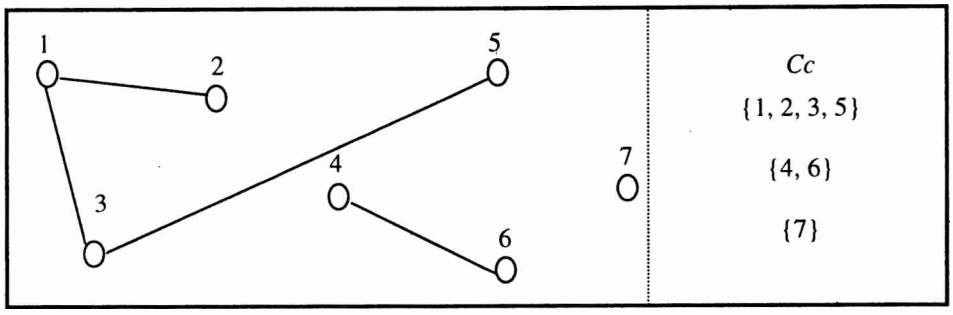


Figura 7.12: Grafo no Dirigido y Conjuntos *Cc*.

El cálculo de los componentes conectados de un grafo no dirigido *G* resulta inmediato si se tiene la clausura transitiva A^* de *G* porque $a_{ij}^* = 1$ significa que hay un camino entre los nodos *i* y *j*, es decir que están en el mismo conjunto de *Cc*. Los pasos a seguir a partir de la matriz A^* son:

- Se busca en cada fila de A^* el primer uno que corresponde a la columna j_1.
- Todas las filas que tengan el mismo j_1 corresponden a los nodos que pertenecen al mismo conjunto *Cc*.

En la Fig. 7.13 se muestra la matriz A^* correspondiente a la clausura transitiva del grafo no dirigido de la Fig. 7.12. El lector puede, a partir de la matriz A^*, hallar los conjuntos *Cc* utilizando los pasos anteriores. Se puede utilizar la Ec. (7.10) para procesar solamente la parte triangular superior o triangular inferior de la matriz A^* correspondiente a la clausura transitiva del grafo *G*.

$$A^* = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

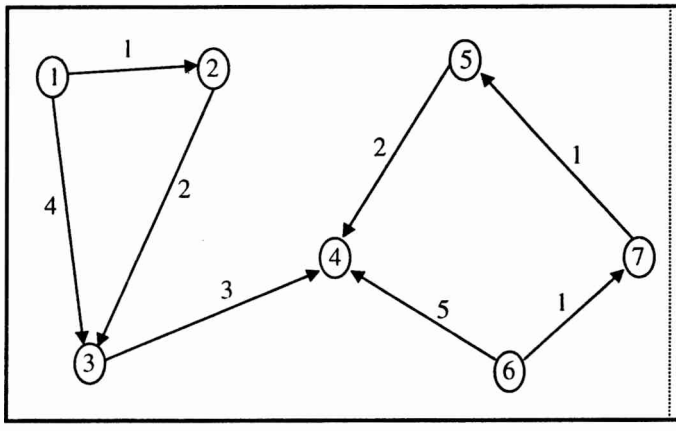
Figura 7.13: Matriz de Adyacencia de la Clausura Transitiva.

Es muy interesante que el mismo arreglo bidimensional de procesadores que se ha presentado puede utilizarse para hallar los componentes conectados de un grafo no dirigido. No se necesitan mayores cambios ni en la implementación del algoritmo ni en el hardware del arreglo de procesadores. Lo que se debe realizar es un procesamiento posterior para hallar los conjuntos *Cc* a partir de la matriz A^*.

7.1.5 Algoritmos de Camino Mínimo en un Grafo

Uno de los problemas más interesantes y analizados sobre grafos es el de encontrar el camino

de costo mínimo entre dos nodos. En este caso, se considera como punto de partida un grafo $G = (V, E, W)$ donde V es el conjunto de n nodos del grafo, E es el conjunto de arcos dirigidos, y $W = w_{ij}$ con $i = 1, ..., n$ y $j = 1, ..., n$; indica el costo de utilizar el arco dirigido que une al nodo i con el nodo j. Si w_{ij} es ∞ entonces no existe el arco dirigido que une al nodo i con el nodo j. La Fig. 7.14 muestra un grafo dirigido con costos asociados a cada arco y la matriz W que le corresponde.



$$W = \begin{pmatrix} 0 & 1 & 4 & \infty & \infty & \infty & \infty \\ \infty & 0 & 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 2 & 0 & \infty & \infty \\ \infty & \infty & \infty & 5 & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 \end{pmatrix}$$

Figura 7.14: Grafo Dirigido y Matriz de Costo Asociada.

El objetivo de la búsqueda del camino mínimo entre cada par de nodos del grafo G es la obtención de la matriz de costo óptimo $W^* = w_{ij}^*$ con $i = 1, ..., n$ y $j = 1, ..., n$; donde w_{ij}^* indique el costo mínimo para llegar al nodo j partiendo desde el nodo i. Si w_{ij}^* es ∞ significa que no existe un camino que une al nodo i con el nodo j en el grafo G.

La Fig. 7.15 muestra la matriz de costo óptimo W^* correspondiente al grafo de la Fig. 7.14. Nótese que el costo de algunos enlaces directos entre dos nodos del grafo G ha sido reemplazado por el costo de un camino entre esos dos nodos que es de menor costo. Por ejemplo, se puede ver que el valor $w_{13}^* = 3 < w_{13} = 4$, porque existe un camino desde el nodo 1 al nodo 3 pasando por el nodo 2 que es de menor costo que el arco dirigido que une al nodo 1 cono el nodo 3. También aparecen en la matriz de costo óptimo W^* los costos de los caminos entre dos nodos que no están directamente conectados en el grafo G. En este caso se cumple que w_{ij}^* ya no es ∞, y en la Fig. 7.15 se pueden ver, por ejemplo, $w_{14}^* = 6$ y $w_{65}^* = 2$ entre otros.

$$W^* = \begin{pmatrix} 0 & 1 & 3 & 6 & \infty & \infty & \infty \\ \infty & 0 & 2 & 5 & \infty & \infty & \infty \\ \infty & \infty & 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 2 & 0 & \infty & \infty \\ \infty & \infty & \infty & 4 & 2 & 0 & 1 \\ \infty & \infty & \infty & 3 & 1 & \infty & 0 \end{pmatrix}$$

Figura 7.15: Matriz de Costo de Camino Mínimo.

El algoritmo utilizado para hallar los caminos mínimos tiene muchos puntos en común con el cálculo de la clausura transitiva que ya se ha explicado. En cierto sentido esto resulta lógico, porque se busca un camino entre dos nodos y *al mismo tiempo* se calcula su costo *para elegir el camino de costo mínimo*. Expresando el algoritmo en función de las operaciones sobre la matriz de costo W, se comienza con los costos originales del grafo G y se llevan a cabo n fases:

- En la primera fase se asigna el costo de unir el nodo i con el nodo j con el costo del menor camino que sale del nodo i y llega al nodo j y pasa a lo sumo por el nodo 1. En términos de la matriz de costo, se genera la matriz $W^{(1)} = w_{ij}^{(1)}$ donde el valor de cada $w_{ij}^{(1)}$ es tal que $w_{ij}^{(1)} = min(w_{ij}, (w_{i1} + w_{1j}))$.
- En la segunda fase se asigna el costo de unir el nodo i con el nodo j con el costo del menor camino que sale del nodo i y llega al nodo j y que pasa a lo sumo por los nodos 1 y 2. En términos de la matriz de costo, se genera la matriz $W^{(2)} = w_{ij}^{(2)}$ donde el valor de cada $w_{ij}^{(2)}$ es tal que $w_{ij}^{(2)} = min(w_{ij}^{(1)}, (w_{i2}^{(1)} + w_{2j}^{(1)}))$.
- En general, en la fase h se asigna el costo de unir el nodo i con el nodo j con el costo del menor camino que sale del nodo i y llega al nodo j y que pasa a lo sumo por los nodos 1, ..., h-1. En términos de la matriz de costo, se genera la matriz $W^{(h)} = w_{ij}^{(h)}$ donde el valor de cada $w_{ij}^{(h)}$ es tal que $w_{ij}^{(h)} = min(w_{ij}^{(h-1)}, (w_{ih}^{(h-1)} + w_{hj}^{(h-1)}))$.
- Naturalmente luego de n fases, se tiene en $W^{(n)}$ el mínimo costo para unir cada nodo i con cada nodo j. Por lo tanto, $W^{(n)} = W^*$.

La implementación del algoritmo puede hacerse con un arreglo de procesadores y una secuencia de pasos *idéntica a la del cálculo de la clausura transitiva del grafo G,* reemplazando la operación $\wedge$ por la operación +, y reemplazando la operación $\vee$ por la operación *min*. Dado que las operaciones + y *min* son del mismo orden de complejidad para el arreglo de procesadores que las operaciones "originales" $\wedge$ y $\vee$, se puede implementar el cálculo de W^* en el mismo tiempo en el que se obtiene la clausura transitiva de un grafo, es decir en $g(n) = 3n$-1 pasos de ejecución, lo cual es $O(n)$.

Como en el caso de la búsqueda de las componentes conectadas de un grafo no dirigido, la solución al problema de encontrar los caminos mínimos en un grafo dirigido con costos asociados a cada arco es casi idéntica a la resolución de la clausura transitiva. En este caso, lo único que se necesita cambiar con respecto a la implementación de la clausura transitiva es el tipo de operación elemental que se lleva a cabo en cada paso de ejecución. Tanto la estructura de la implementación como el hardware requerido y el tiempo necesario para resolver el problema permanecen iguales.

7.2 Algoritmos de Ruteo de Paquetes

Una de las características de las implementaciones que se han presentado en éste y en los capítulos precedentes es que *los datos adecuados están disponibles en el procesador que los requiere*. En las consideraciones de hardware que se hicieron con respecto a la implementación de la clausura transitiva de un grafo sobre un arreglo bidimensional de procesadores ya se observa que esto no siempre se puede afirmar. En muchos problemas reales, los datos se deben (re)transmitir por la red de interconexión de los procesadores. Por lo

tanto, *no siempre* se puede asegurar que en el procesador P_i está disponible un conjunto específico de datos D sobre el cual operar en un instante de tiempo t determinado.

Los problemas de ruteo son clásicos en las computadoras paralelas con redes de interconexión estáticas, donde un procesador no está *directamente* conectado a todos los demás. En las implementaciones sobre tales máquinas, es usual que un procesador necesite enviar o recibir datos hacia o desde un procesador utilizando otros procesadores para transmisiones intermedias. Si bien es clásico en el contexto de las redes estáticas de interconexión, la discusión del problema de ruteo de paquetes se puede trasladar sin muchos cambios al contexto de las redes dinámicas. En este caso, los puntos de conexión (switches) actúan como los procesadores intermedios a través de los cuales se transmite la información para llegar al procesador destino. En cualquier caso, para simplificar la presentación y el análisis, en general se hace referencia a una red de interconexión estática de procesadores.

En el problema de ruteo de paquetes se tiene un conjunto de m bloques de información (paquetes), almacenados en m nodos o procesadores *origen* Po_i, $i = 1, ..., m$, donde cada paquete tiene un nodo o procesador *destino* Pd_i, $i = 1, ..., m$, al cual debe llegar utilizando la red de interconexión. En principio se supone que para distintos paquetes los nodos origen son distintos entre sí, $Po_i \neq Po_j$, $1 \leq i < j \leq m$; y también que los nodos destinos son distintos entre sí, $Pd_i \neq Pd_j$, $1 \leq i < j \leq m$. El problema consiste en "rutear" adecuadamente los m paquetes, es decir transmitirlos desde el nodo origen Po_i en el que están almacenados hasta el nodo destino Pd_i al que deben llegar pasando por todos los procesadores intermedios que sean necesarios. Las restricciones usuales para los algoritmos de ruteo son dos:

- utilizar control local en cada procesador (nodo), y
- minimizar el tiempo, es decir la cantidad de pasos (intermedios) necesarios para que cada paquete llegue al nodo destino desde el nodo origen.

Dentro de los objetivos de este texto, el interés por esta clase de problema es doble. Por un lado y tal como se ha explicado, es un problema que se debe resolver en una amplia gama de implementaciones sobre computadoras paralelas con redes de interconexión estáticas. Por otro lado, es en sí mismo un problema de procesamiento paralelo ya que se intenta que la mayoría de los paquetes sean transmitidos de forma simultánea entre los procesadores.

Se tratará una clase de solución conocida como *greedy algorithm* (algoritmo *avaro* o *ávido*) al cual se hará referencia a partir de ahora como algoritmo greedy. En general, en términos de la función de costo asociada a un problema, los algoritmos greedy buscan el mínimos local más cercano a partir de un punto dado. En el caso del problema de ruteo de paquetes, la función de costo asociada es el tiempo, o sea la cantidad de pasos intermedios para que cada paquete llegue al nodo destino Pd_i partiendo desde el nodo origen Po_i. Esta cantidad de pasos intermedios está compuesta por:

- La cantidad de pasos intermedios de comunicación para que el paquete se transmita desde el nodo Po_i hasta el nodo Pd_i.
- La cantidad de pasos en los que el paquete debe esperar para utilizar enlaces de comunicación que encuentra ocupados.

Los algoritmos greedy para el ruteo de paquetes pueden mejorarse de varias formas, tal como se menciona en [Lei92].

7.2.1 Algoritmos de Ruteo Fijo

Un algoritmo que rutea m paquetes a sus nodos destinos Pd_i *recorriendo un camino mínimo*, se denomina algoritmo greedy. En el caso de las redes de interconexión estáticas, el camino mínimo entre un nodo origen Po_i y un nodo destino Pd_i es el que utiliza la menor cantidad de enlaces de comunicación entre los dos nodos.

En la Fig. 7.16 se muestra un arreglo lineal de $N = 8$ nodos y el ruteo de $m = 5$ paquetes en dicho arreglo hasta alcanzar los nodos destinos correspondientes. Los paquetes están inicialmente almacenados en m procesadores origen. Cada paquete puede transmitirse en un paso del ruteo a izquierda o a derecha del procesador en el que está ubicado. Para esto, se asume que cada uno de los procesadores del arreglo lineal posee canales bidireccionales a izquierda y derecha. En este caso, el algoritmo de ruteo determina que, si el paquete i está almacenado en el procesador P_j del arreglo lineal

- Si $Pd_i = j$ el paquete ha llegado al procesador destino y debe ser almacenado.
- Si $Pd_i \neq j$ entonces el paquete debe ser enviado al procesador P_{j+1} si $Pd_i > j$ o al procesador P_{j-1} en caso contrario ($Pd_i < j$).

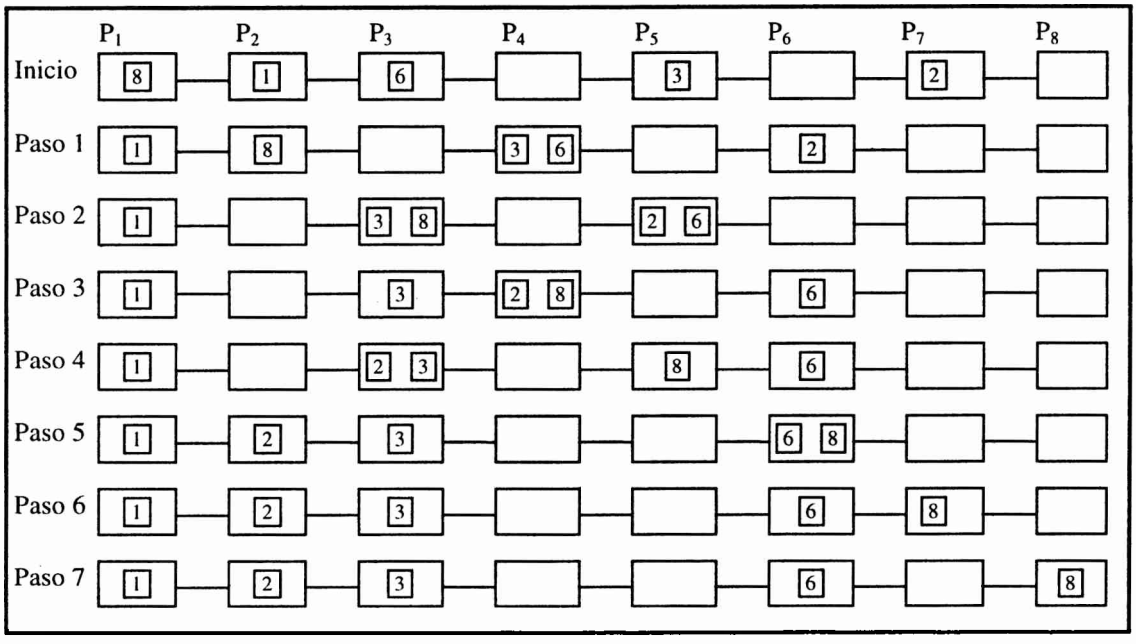


Figura 7.16: Ruteo en un Arreglo Lineal de Procesadores.

Se debe hacer notar que con las hipótesis precedentes:

- Un paquete alcanza su nodo destino a lo sumo en N-1 pasos. En general requiere d pasos, con $d < N$ representando la distancia al nodo destino.
- Dos paquetes pueden coexistir en memoria del mismo nodo, pero nunca competirán en el mismo paso por el mismo enlace de comunicaciones.
- Al principio y al final del algoritmo cada nodo tendrá a lo sumo 1 paquete.

Por primera vez se da la cantidad de pasos de ejecución estrictamente en función de la arquitectura (cantidad de procesadores o nodos en este caso), sin tener en cuenta la aplicación que se ejecuta. Todo lo que se conoce de la aplicación es que necesita resolver el problema de ruteo de paquetes. De alguna manera, este problema es "interno" y como tal independiente del tamaño del problema que se necesita resolver con la arquitectura paralela.

Cuando la topología de la red de interconexión no es tan simple como el arreglo lineal, *aparece competencia por los enlaces de comunicaciones* (*contention*). La Fig. 7.17 muestra dos sucesiones de paquetes que llegan a un nodo común Pc. Cada sucesión de paquetes llega al nodo común Pc por caminos (enlaces) diferentes y deben utilizar el mismo enlace para salir de él. Cada paso de ruteo implica que al menos uno de los paquetes deberá encolarse y esperar, lo que degrada el rendimiento del algoritmo. Uno de los criterios más comunes para decidir qué paquete utilizará un enlace en caso de haber más de uno que lo requiere es el *farthest-first,* que prioriza el paso al paquete que tiene el destino más lejano. Si bien en esta figura se muestran dos enlaces por los cuales llegan los paquetes al nodo común Pc, en general, si un nodo tiene k enlaces de comunicación, pueden entrar hasta k-1 sucesiones de paquetes que compiten por un mismo enlace de salida del nodo Pc. En este caso se tiene que en cada paso de ruteo se agregan a la cola de salida hasta k-1 paquetes y solamente un paquete (uno de los que llegó recientemente o en algún paso anterior) puede abandonar el nodo.

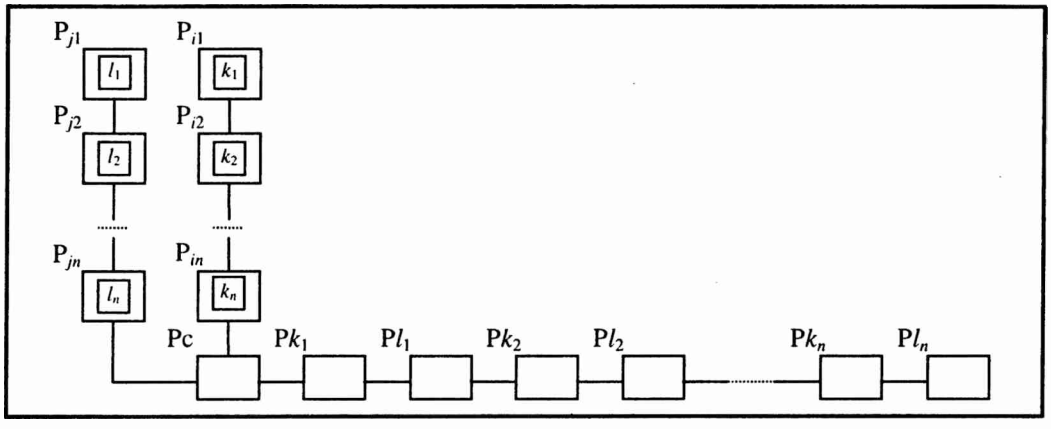


Figura 7.17: Competencia por Enlaces de Comunicación.

Se puede considerar ahora una estructura de arreglo bidimensional de NxN nodos. Manteniendo las hipótesis de ruteo ya mencionadas, se tiene que cada paquete h está almacenado en un procesador $Po_h = (i, j)$ y tiene un procesador destino $Pd_h = (k, l)$. Los valores Po_h y Pd_h hacen referencia a la posición del procesador origen y procesador destino del paquete h respectivamente. Un algoritmo greedy de ruteo se puede definir en dos fases :

- En la primera fase se transmite cada paquete por los procesadores de la fila del procesador origen (i) hasta llegar a la columna del procesador destino (l). Esto implica $|l - j|$ transmisiones intermedias.
- En una segunda fase se transmiten los paquetes por los procesadores de la columna del procesador destino (l) hasta llegar a la fila del procesador destino (k) donde termina su recorrido. Esto implica $|k - i|$ transmisiones intermedias.

La primera fase del algoritmo se puede cumplir en N-1 pasos, dado que no hay conflicto de ruteo *dentro de las filas*. De hecho cada fila del arreglo bidimensional se comporta como el arreglo lineal de N procesadores que se ha detallado anteriormente. La diferencia principal es que se puede producir competencia por un enlace de comunicación y esto tiene una consecuencias inmediatas: hay pasos de ruteo en los cuales un paquete debe esperar a que el enlace que requiere para seguir su camino hacia el procesador destino esté disponible (desocupado).

Al iniciar la segunda fase del algoritmo se pueden tener varios paquetes que deben moverse en la misma dirección, *desde el mismo nodo.* Si se aplica el criterio *farthest-first* para la utilización de los enlaces, se puede asegurar que todos los paquetes alcanzan su nodo (procesador) destino a lo sumo en N-1 ciclos. Las razones son:

(1) Al iniciar la segunda fase del algoritmo, en la posición (Ix, Jx) se tienen k paquetes que deben moverse por la columna Jx desde la fila Ix hacia la fila N (el análisis en el caso en que los paquetes deben transmitirse hacia la fila 1 es análogo). La cantidad de nodos ubicados entre Ix y N debe ser mayor que k ($k \leq N\text{-}Ix$), y el nodo destino de cada paquete es diferente.

(2) El caso peor al iniciar la segunda fase es que haya exactamente $k = N\text{-}Ix$ paquetes cuyos destinos son las posiciones (N, Jx), (N-1, Jx), ..., (Ix+1, Jx).

(3) Respetando el criterio *farthest-first*, el primer paquete en salir del nodo en la posición (Ix, Jx) es el que tiene como destino el nodo en la posición (N, Jx), que debe realizar $N\text{-}Ix$ pasos. El segundo paquete en salir es el que tiene como destino (N-1, Jx), que debe realizar $N\text{-}Ix$-1 pasos. El último paquete en salir del nodo en la posición (Ix, Jx) es el que debe ir en 1 paso hasta el nodo en la posición (Ix+1, Jx). En el peor caso, la cantidad total de pasos para el paquete i-ésimo que transcurren desde que llega al nodo en la posición (Ix, Jx) hasta el nodo destino es, por lo tanto, (i-1) + ($N\text{-}Ix\text{-}i$+1) = $N\text{-}Ix$.

(4) La cantidad de pasos en el peor caso ($N\text{-}Ix$) alcanza el máximo con Ix = 1, por lo tanto la cantidad máxima de pasos de la segunda fase es N-1.

Así se llega a que el algoritmo greedy de ruteo en un arreglo de NxN nodos requiere 2N-2 pasos *en el peor caso*. La Fig. 7.18 muestra el ruteo de diez paquetes en un arreglo bidimensional de 4x4 nodos.

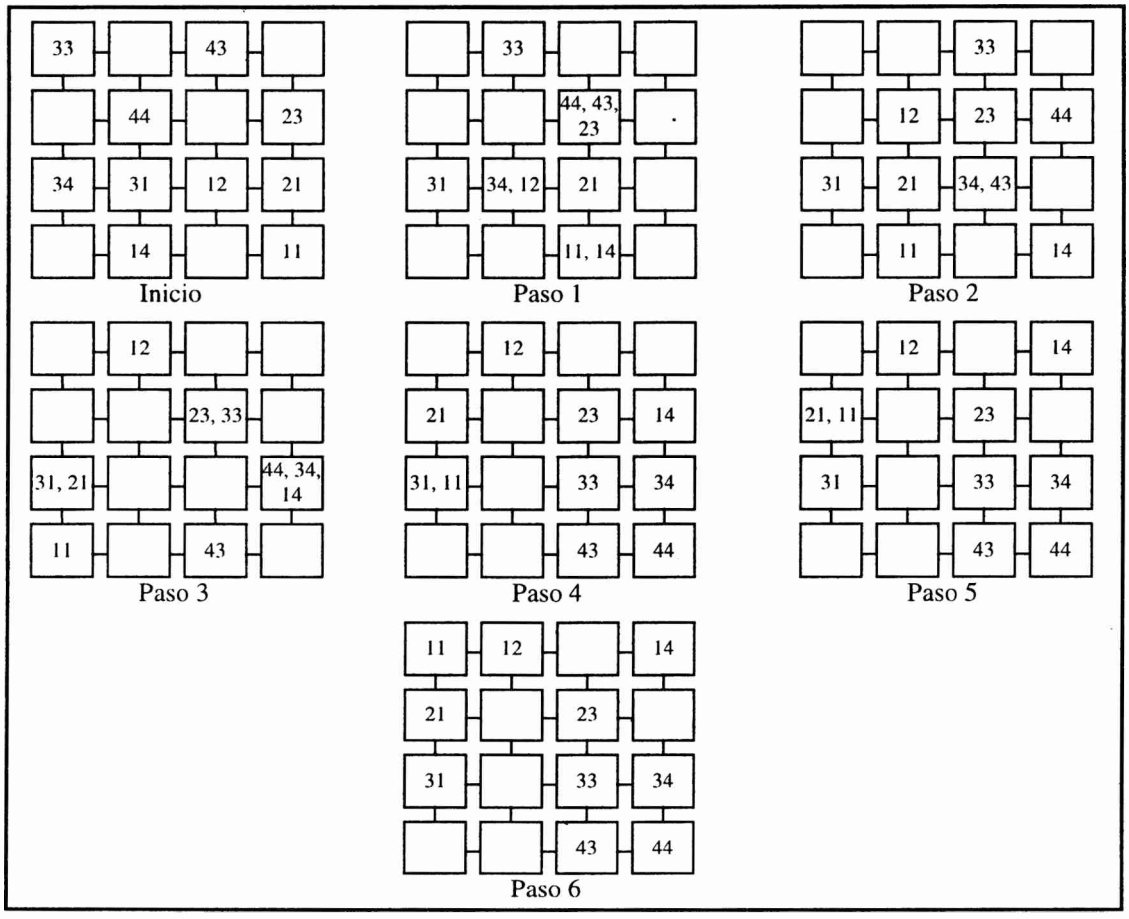


Figura 7.18: Ruteo en un Arreglo Bidimensional de Procesadores.

Siguiendo la evolución del ruteo de los paquetes por separado en la Fig. 7.18, se puede notar claramente que no todos los paquetes necesitan los $2N$-1 pasos de ruteo del peor caso para llegar al nodo destino. La cantidad de pasos de ruteo que necesita un paquete en particular depende de la distancia desde el nodo origen al nodo destino y de la competencia por los enlaces de comunicación por los que se necesita transmitir. En el caso de la Fig. 7.18, ningún paquete tiene que esperar ningún paso por la liberación de un enlace que se necesita.

En el análisis de cantidad de pasos de ruteo se afirma de antemano que en general, el i-ésimo paquete en ser transmitido desde el nodo (Ix, Jx) es el que tiene como destino el nodo en la posición (N-i+1, Jx) que espera (i-1) pasos y debe llevar a cabo (N-Ix-i+1) transmisiones intermedias para llegar a destino. Esto es posible en el peor caso, pero también se puede estudiar estadísticamente el *caso promedio*, tal como en [Lei92].

7.2.2 El Problema del Ruteo Dinámico

Las dos hipótesis más importantes que se han hecho en la sección anterior son: (1) cada paquete tiene como destino un nodo diferente y (2) los paquetes tienen una ubicación conocida al iniciar el proceso de ruteo.

En problemas reales, muchas veces ninguna de las hipótesis que se asumieron es verdadera. El destino de los paquetes puede ser coincidente o incluso centralizado. Por ejemplo en un administrador de base de datos centralizada, claramente muchos paquetes se dirigirán al nodo administrador desde los nodos de la red, mientras que el flujo entre los demás nodos es bajo. Por otro lado, el arribo de los paquetes a un nodo inicial puede ocurrir en cualquier instante de tiempo.

Si bien no se presentará el análisis estadístico dependiente de la tasa de arribos y la probabilidad de destino, se considera el caso peor referido al destino de m paquetes fijos en un arreglo bidimensional de NxN nodos. Este caso es el de m paquetes que tienen un mismo nodo origen y tienen como destino el nodo más lejano. Tal como se ha planteado el problema, m podría ser mayor que N.

Cuando m paquetes parten de un mismo nodo origen hacia el mismo nodo destino utilizando el algoritmo greedy ya presentado, en general todos los paquetes necesitan $d1$+$d2$ pasos de transmisión, y se transmiten de forma secuencial por el canal que utilizan para salir del nodo. El último paquete en salir tendrá una demora de m-1 pasos, y por lo tanto llegará después de ($d1$+$d2$+m-1) pasos, donde

- $d1$ representa la distancia a la columna destino, que en el peor caso es N-1.
- $d2$ representa la distancia a la fila destino, que en el peor caso es también N-1.

Por lo tanto, el tiempo total del algoritmo en el peor caso es ($2N$-3+m) pasos.

La Fig. 7.19 muestra un ejemplo del problema con cuatro paquetes que se deben enviar desde el nodo en la posición (1,1) al nodo en la posición (3,3) en un arreglo de 3x3 nodos. Todos los paquetes siguen el mismo algoritmo greedy de ruteo y, por lo tanto, pasan por los mismos nodos intermedios y utilizan los mismos enlaces desde el nodo origen hasta el nodo destino. Se puede notar que con una pequeña variación del algoritmo el ruteo de los paquetes se podría llevar a cabo en menos pasos de ruteo. Por ejemplo, unos paquetes se podrían transmitir primero hacia la fila del nodo destino mientras otros se rutean hacia la

columna del nodo destino. Si bien en este caso se reduce la cantidad de pasos de ruteo para los paquetes, cambios como estos en el algoritmo de ruteo ponen en peligro otras características. Por ejemplo, en el análisis general de cantidad de pasos que se ha hecho, ya no se puede asegurar que en la primera fase no haya conflictos por la utilización de recursos (enlaces) y por lo tanto ya no se puede asegurar que la cantidad máxima de pasos sea ($2N$-3+m) pasos. Por otro lado, habría que analizar con mucha más atención la posibilidad de aparición de *deadlocks* en el ruteo de los paquetes, o lo que es casi lo mismo, la longitud de las colas que se necesita para que el algoritmo pueda resolver todos los conflictos.

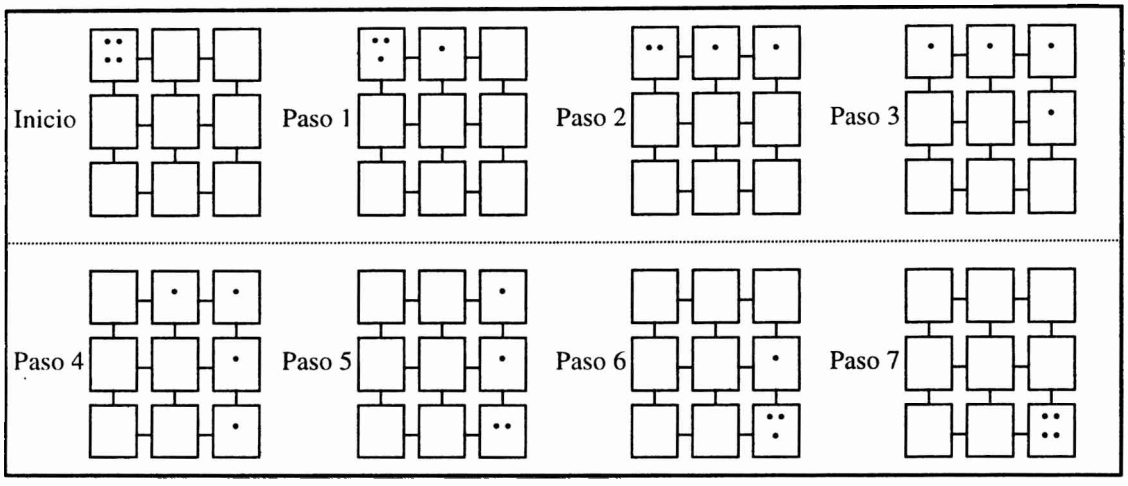


Figura 7.19: Ruteo de Cuatro Paquetes con Igual Nodo Origen y Destino.

7.3 Algoritmos de Procesamiento de Imágenes

Una imagen es la representación espacial de un objeto o de una escena de dos o tres dimensiones. Una imagen se puede almacenar de distintas formas, según el medio que se utilice y, en el caso de las computadoras, se deberá utilizar un formato digital. Como en el caso de las señales acústicas (sonido), las imágenes pueden se pueden describir como señales analógicas (visuales, en este caso), y se deben discretizar las variables libres que intervienen. Específicamente, una imagen es una representación bidimensional donde intervienen tres variables: eje horizontal, eje vertical y color o nivel o valor de la imagen en un punto. En otros términos, una imagen es una función aplicada sobre una región plana y tiene como resultado la intensidad de luz o el color de cada punto [Koe94] [Ste95].

Una imagen digital es representada por una matriz de valores numéricos [Lei92], donde cada uno de esos valores (muestra de la imagen) es la cuantificación del valor de intensidad. Cada uno de los puntos en los cuales una imagen es muestreada se conoce como *Picture Element* y usualmente se abrevia como *Pixel*. El problema inmediato que se presenta al tratar de manejar las imágenes en computadoras es la variedad de datos y la cantidad de datos que hay capturar, almacenar, procesar y transferir [Kim92]. Por esta razón, desde el inicio mismo de la incorporación y procesamiento de imágenes digitales la representación o codificación intenta realizar a la vez minimizar al máximo la cantidad de datos que se utilizan.

En general, el valor de un pixel se ha denominado usualmente color del pixel, y éste puede ser la codificación de dos valores solamente (imágenes en blanco y negro o *bilevel*),

niveles de grises y varias cantidades posibles de colores alternativos (desde 2^8 hasta 2^{24}). En todos los casos, el valor de un pixel surge de la cuantificación del valor de intensidad de la imagen original en un punto dado.

Las dimensiones horizontal y vertical se han discretizado de diferentes formas y esto tiene como resultado distintas resoluciones de las imágenes. La resolución se puede definir como la capacidad de identificar un punto de otro de la imagen original en la imagen digitalizada. Para la discretización de las dimensiones horizontal y vertical, lo más común ha sido utilizar una malla con los pixels igualmente espaciados a lo largo de ambos ejes. La distancia entre los puntos obviamente afecta la precisión con la cual la imagen original es representada, es decir cuánto nivel de detalle de la imagen original puede representarse.

El formato de la imagen suele especificarse con dos parámetros: *Resolución Espacial* y *Codificación de Color*. La resolución espacial se da en *pixels x pixels*, los valores más usuales para imágenes estáticas han sido 320 x 240 y 640 x 480, aunque también se utilizan valores como 720 x 480. Cuando se trata de imágenes de video, se suelen agregar otros valores y esta variación se debe a la utilización de placas conversoras (analógico-digitales y digital-analógicas) específicas. La codificación de las imágenes con la resolución espacial y el valor de cada pixel se suele denominar codificación PCM: Pulse Code Modulation.

Teniendo en cuenta la cantidad de datos que se necesitan para una imagen es razonable que se intente tanto la compresión como la paralelización de cualquier procesamiento. Por ejemplo, una imagen de 320 x 240 pixels en blanco y negro requiere 9600 bytes y una imagen de 640 x 480 pixels con 24 bits por pixel requiere aproximadamente 921600 bytes. Hasta el tratamiento más simple de las imágenes requiere la utilización de cada pixel al menos una vez y, por lo tanto, la cantidad de operaciones que se deben llevar a cabo para procesar una imagen es muy elevada en la mayoría de los casos.

Agregado a la cantidad de datos que se deben procesar, en algunos contextos de procesamiento de imágenes se agregan restricciones de tiempo de cómputo disponible para obtener una respuesta. Este es el caso típico de un robot que debe identificar las piezas que le llegan por una cinta transportadora y que, según la pieza que llega es la tarea que debe realizar el robot. Puede ser tan sencillo como clasificar las piezas para almacenarlas en lugares diferentes o enviarlas a talleres de armado de distintos productos.

El procesamiento de imágenes digitales ha aprovechado todo lo referente al procesamiento de señales en general, y ha incluido sus propias características. En general, los problemas de tratamiento de imágenes tienen una solución secuencial conocida o derivable de alguna solución ya desarrollada. La paralelización de estas soluciones muchas veces es inmediata por la división de la imagen en distintos procesos de cálculo sobre los pixels. En otros casos se debe observar con mucha atención el procesamiento que necesita toda la imagen en su conjunto para llevar a cabo la tarea que se requiere.

Como ya se ha adelantado, no se cubrirán *todas* las clases de problemas que se relacionan con el procesamiento de imágenes, y en vez de ello se intentará mostrar las características más importantes en cuanto al procesamiento paralelo de las imágenes. Se tomaron como ejemplos tres problemas: (1) afinado y separación de curvas de nivel de planimetría, (2) reconocimiento de patrones geométricos simples, y (3) compresión de imágenes utilizando JPEG.

7.3.1 Afinado y Separación de Curvas

En este problema se deben procesar imágenes digitalizadas de curvas de nivel de planimetría como paso previo a una representación tridimensional del perfil del terreno real [Cos94a] [Cos94b]. Cada curva de nivel representa una altura con respecto al nivel del mar, por ejemplo.

El tipo de tratamiento que requiere este problema es muy común en el contexto de las imágenes porque se debe procesar cada uno de los pixels, y el procesamiento requiere el conocimiento de los pixels adyacentes o *vecinos inmediatos*. Todos los pixels se procesan de la misma manera. Ejemplos similares son el realce de los bordes, la eliminación de ruido y la identificación del esqueleto de los objetos de una imagen [PRA78] [JAI89].

Las imágenes a procesar corresponden a la digitalización bilevel (en blanco y negro) de cartas con las curvas de nivel de la superficie terrestre. Cada punto que pertenece a una curva es negro y el fondo es blanco. Las dos tareas más importantes a realizar son:

1. Afinar las curvas de la imagen digitalizada, para que cada punto de la curva real se asocie con un solo pixel. Expresado de otra forma, la curva de la imagen digitalizada debe tener solamente *un pixel de ancho* y se debe mantener la conectividad de los puntos que pertenecen a una misma curva.
2. Separar las curvas, es decir, identificar cada curva como una secuencia de puntos o lo que es lo mismo identificar unívocamente a qué curva pertenece cada punto. Esta secuencia de puntos junto con la altura asociada a cada curva se utiliza para la proyección tridimensional.

El proceso de afinado de las curvas debe ser anterior al reconocimiento y separación de las mismas. Por un lado, el reconocimiento de curvas no afinadas (con más de un pixel de ancho) se hace más difícil algorítmicamente que con curvas afinadas. Por otro lado, las curvas no afinadas necesitan mayor capacidad de almacenamiento porque tienen mayor cantidad de pixels.

La resolución de digitalizado de la imagen original suele tener incidencia en el afinado porque a mayor resolución cada curva tendrá mayor cantidad de pixels y por lo tanto el proceso de afinado normalmente requerirá mayor tiempo. En contrapartida, si la resolución de digitalizado de la imagen es muy baja se corre el riesgo de producir la discontinuidad de una misma curva. Esto a su vez ocasiona que el proceso de reconocimiento identifique dos o más curvas de nivel donde originalmente se tenía solamente una.

El proceso de reconocimiento y separación de las curvas de nivel se simplifica considerablemente sobre la base de dos hipótesis muy fuertes:

- Las curvas tienen un pixel de ancho, porque son el resultado del proceso de afinamiento. De esta manera se elimina la ambigüedad resultante de la correspondencia de varios pixels de la imagen digitalizada con un mismo punto de una curva de nivel.
- Las curvas de nivel no tienen intersección porque corresponden a distintos niveles del terreno. Por lo tanto, se elimina la ambigüedad que surge cuando desde un punto de intersección parten dos curvas diferentes.

Inicialmente se describe el algoritmo de afinado de las curvas digitalizadas y luego el algoritmo de separación de las curvas. En ambos casos se incluye también un posible

paralelización del procesamiento.

Algoritmo de Afinado de Curvas

Se deben borrar de la imagen digital (asignar su valor en *blanco*) los pixels que son redundantes. Por lo tanto, cada pixel que tiene el valor *negro* en la imagen debe ser analizado para verificar si se cambia a *blanco* o permanece sin cambio. La imagen se recorre tantas veces como sea necesario para la eliminación de pixels negros que son redundantes. En cada barrida de la imagen se erosionan las curvas hasta que son de un pixel de ancho. El procesamiento de cada pixel p de la imagen se define en función de la numeración de los pixels vecinos tal como lo muestra la Fig. 7.20.

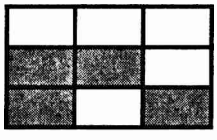
p2	p1	p8
p3	p	p7
p4	p5	p6

Figura 7.20: Numeración de Pixels para Procesar.

Para cada pixel a procesar se definen también Z0(p) como la cantidad de transiciones de cero a uno en el orden p1, p2, ..., p8, p2; y NZ(p) como la cantidad de vecinos distintos de cero de p. En función de estos datos un pixel negro (p = 1) se borra (se asigna p = 0) si se cumple que [Jai89]:

$(NZ(p) \geq 2) \wedge$ /* (1) */
$(NZ(p) \leq 6) \wedge$ /* (2) */
$(Z0(p) = 1) \wedge$ /* (3) */
$((p1 * p3 * p7 = 0) \vee (Z0(p1) \neq 1)) \wedge$ /* (4) */
$((p1 * p3 * p5 = 0) \vee (Z0(p3) \neq 1))$ /* (5) */

(1) Al menos dos vecinos negros para no desconectar una curva.
(2) No más de seis vecinos negros para no quitar un punto interior de una curva.
(3) Para evitar que la curva se desconecte en casos del tipo:



(4) y (5) Son necesarias para llevar un orden en la erosión de los puntos, es decir no borrar un punto si no fue borrado antes uno vecino.

Cada pixel negro del borde de la imagen digital se borra si tiene más de un pixel negro vecino y además su borrado no desconecta localmente la curva a la que pertenecen.

La imagen se procesa pixel por pixel hasta que en un barrido de la imagen no se elimina ningún pixel negro. Cuando no se pudo eliminar ningún pixel negro entonces cada curva de nivel de la imagen tiene un pixel de ancho. La cantidad de veces que se procesa cada pixel o, lo que es igual, la cantidad de veces que se procesa la imagen, no es conocida de antemano.

Paralelización del Proceso de Afinado de las Curvas de Nivel

Según el algoritmo de afinado de las curvas de nivel, para el tratamiento de un pixel se necesita conocer el valor de los pixels vecinos de una columna previa y una posterior y dos filas previas y una posterior. Además, los pixels de una fila no pueden ser procesados para el afinamiento si antes no se procesaron los pixels de las filas previas. Esto determina que el proceso de afinado es secuencial por filas.

Dada la secuencialidad en el proceso de afinado por filas de la imagen se puede pensar paralelizar el algoritmo aplicando procesamiento pipeline por filas. Una vez que se procesa una fila se la transmite a la siguiente etapa de afinado. Dada la dependencia del procesamiento de una fila con las filas cercanas no conviene que una etapa contenga *solamente* una fila de la imagen. Dado que para el procesamiento de los pixels de una fila de la imagen se necesitan tres filas más (dos previas y una posterior) cada procesador contiene en memoria local cuatro filas.

La estructura de procesadores necesaria para el procesamiento en modo pipe de las filas de la imagen es, en principio, un arreglo lineal. Dado que el algoritmo de afinado no puede determinar de antemano la cantidad de recorridas de la imagen que se deben llevar a cabo, se determina que las filas de la imagen vuelvan a ingresar a la estructura de procesadores si aún no se ha terminado el procesamiento. Por esta razón el arreglo lineal se realimenta uniendo el último procesador del arreglo con el primero y la estructura de interconexión de los procesadores se convierte en un anillo tal como lo muestra la Fig. 7.21. Para determinar cuándo las filas de la matriz se dejas de procesar y transmitir, uno de los procesadores debe controlar la condición de finalización: n filas consecutivas no se modifican en el procesamiento local, donde n es la cantidad de filas de la matriz.

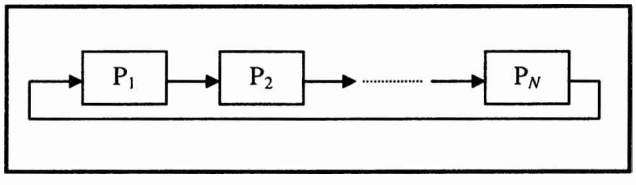


Figura 7.21: Anillo de Procesadores.

Es interesante notar que a diferencia de los ejemplos de procesamiento pipeline que se han dado, todas las etapas en este caso son idénticas. Cada etapa tiene que realizar la misma tarea con la misma cantidad de datos. Esto evita algunos problemas de sincronización y de tiempos diferentes de comunicación entre las etapas. Sin embargo, el tiempo de cómputo de cada etapa no necesariamente es el mismo, porque depende de los valores de los pixels de cada fila. A mayor cantidad de pixels negros, por ejemplo, mayor será el tiempo de cómputo para procesar una fila.

Resultados de Implementación del Afinado de Curvas

Los resultados reportados en [Cos94a] con la implementación del algoritmo de afinado paralelizado sobre transputers muestran el factor de Speed-Up promedio de $0.6N$, donde N es la cantidad de procesadores. La diferencia de este valor con respecto al teórico (N) está dada mayoritariamente por el costo de las comunicaciones en el anillo de procesadores.

Si bien hay dependencia en el procesamiento de los datos de distintas filas de la imagen, el costo en tiempo de la transmisión de una fila puede ser mayor que el costo del procesamiento de dicha fila. La relación entre el costo de la comunicación y de procesamiento depende de la cantidad de datos de la fila (cantidad de columnas de la imagen) y de los valores de los pixels. Esto se hace más evidente en las últimas etapas de la erosión, cuando las filas permanecen casi sin cambios, es decir que hay pocos pixels a eliminar de la imagen original.

Algoritmo de Separación de Curvas

A partir de una imagen con curvas de un pixel de ancho se debe generar una secuencia de puntos para cada curva. El proceso de identificación de las curvas se encarga de aislar cada una de estas curvas y generar una secuencia de puntos por curva. No se conoce de antemano la cantidad de curvas de nivel que contiene una imagen digital y por lo tanto no se conoce de antemano la cantidad de secuencias de pixels que se generan ni la cantidad de pixels que pertenecen a cada curva.

En el caso específico de imágenes con curvas de nivel, se tienen curvas cerradas o abiertas. Las curvas de nivel abiertas tienen cada extremo en uno de los bordes de la imagen digital y además nunca hay intersección de dos curvas. La Fig. 7.22 muestra una imagen ejemplo con una curva abierta y una cerrada.

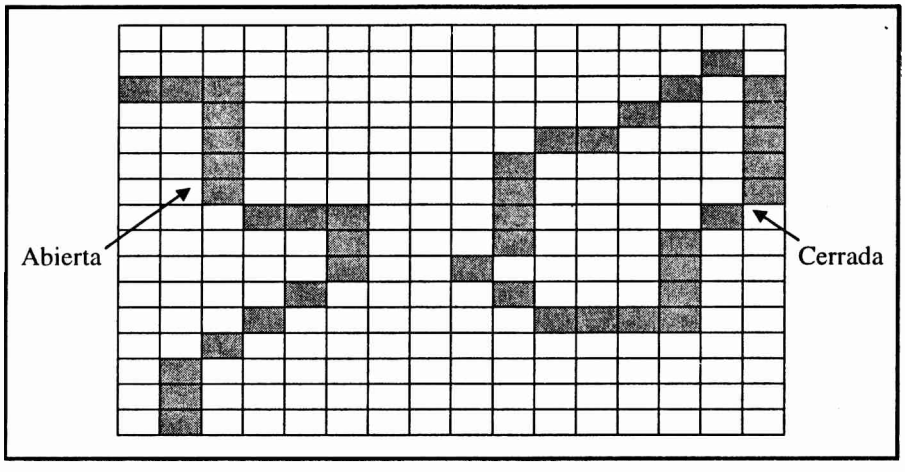


Figura 7.22: Curva de Nivel Abierta y Curva de Nivel Cerrada.

El algoritmo de reconocimiento de curvas también recorre la imagen pixel a pixel y una vez que encuentra un pixel negro determina cuál es el siguiente punto de la curva, si existe. Si un punto de una curva se encuentre en la posición (i, j) el siguiente punto de la curva puede estar (en orden de prioridad):

1. en la misma fila pero en distinta columna $(i, j\pm1)$ o en la misma columna pero en distinta fila $(i\pm1, j)$.
2. en una diagonal a partir de la posición del punto, es decir una fila y una columna adyacentes $(i\pm1, j\pm1)$.

La Fig. 7.23 muestra un ejemplo de identificación de los puntos de una curva. A medida que se encuentran y se procesan los puntos de una curva se los almacena en una secuencia y se los borra de la imagen original. El borrado de los pixels permite que un punto

no se procese más de una vez, lo cual sucedería al menos en el caso de las curvas cerradas. De esta manera, al cabo de una recorrida de toda la imagen se tienen las secuencias de puntos que pertenecen a cada curva.

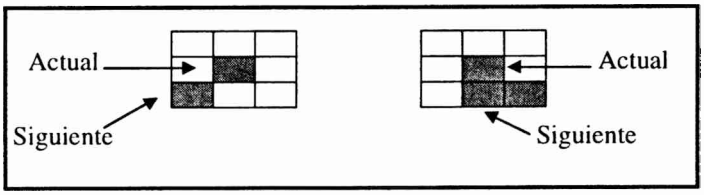


Figura 7.23: Seguimiento de los Pixels de una Curva.

Paralelización del Proceso de Separación de las Curvas de Nivel

La paralelización es inmediata dividiendo el espacio de búsqueda (la imagen) en tantas partes iguales como unidades de procesamiento se tengan disponibles. El algoritmo de identificación de las curvas se aplica sin alteraciones a cada una de las porciones de la imagen. El resultado de cada parte de la imagen es un conjunto de subcurvas SC_i que pueden ser partes de una misma curva C_i de la imagen original. Esta forma de paralelización del proceso es posible por la independencia en el reconocimiento de las curvas C_i entre las distintas partes de la imagen. Una vez aplicado el procesamiento en paralelo (simultáneamente) a las distintas partes de la imagen se debe agregar la unión de las distintas curvas. Cada subcurva SC_i que resulta del procesamiento de un área de la imagen puede ser una parte de una curva que quedó dividida en las distintas áreas de la imagen original.

El costo de cómputo que se agrega en el algoritmo paralelo resulta de la unión de las curvas. El costo de comunicaciones que se agrega en el algoritmo paralelo resulta de la distribución de las partes de la imagen y la concentración de los resultados para la unión de las curvas. La unión de las curvas puede implicar un reordenamiento de la secuencia de algunas partes de la curva para que la secuencia de toda la curva completa quede ordenada como en el algoritmo secuencial.

Cada proceso de reconocimiento recibe una porción de la imagen, aísla las partes de las curvas que encuentra en la "imagen" recibida y retorna tantas secuencias de puntos como subcurvas encuentra. Un proceso posterior debe recibir los resultados de los procesos de reconocimiento (secuencias de puntos correspondientes a partes de curvas) y unir las curvas de la imagen original. La unión de las curvas de nivel se lleva a cabo analizando los extremos de las subcurvas y ordenando y uniendo las secuencias de puntos si así fuera necesario.

Resultados de Implementación de la Separación de Curvas

La Fig. 7.24 muestra la implementación en tres procesadores (P_1, P_2 y P_3) con un proceso (M) que divide la imagen y recoge y procesa los resultados de los procesos que reconocen (R) las subcurvas de cada porción. El proceso M debe reconstruir las curvas preprocesadas por los procesos R y por lo tanto esta unión de las curvas de nivel necesariamente debe realizarse después de la ejecución de los procesos reconocedores de (sub)curvas. Todos los procesos reconocedores de curvas son idénticos y esto simplifica notablemente la implementación paralela. En uno de los procesadores se puede asignar un proceso reconocedor y el proceso de reconstrucción de las curvas de nivel (M) porque no tienen interferencia en cuanto al tiempo

en que necesitan ejecutar. Además, se facilita la comunicación entre ellos porque es local al procesador y por lo tanto mucho más rápida.

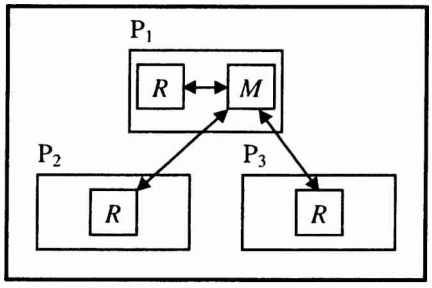


Figura 7.24: Implementación de la Paralelización.

Los resultados reportados en [Cos94b] de este tipo de paralelización sobre transputers muestran un factor de Speed-Up promedio de $0.46N$, donde N es la cantidad de procesadores utilizados. En este caso se debe notar que la paralelización implica una parte importante de cómputo secuencial que corresponde a la unión de las curvas.

7.3.2 Reconocimiento de Patrones Geométricos Simples

En el contexto de las imágenes digitales, un patrón se define como una descripción estructural o cuantitativa de un objeto o cualquier otra entidad de interés. En general, un patrón está formado por una disposición de uno o más descriptores (características) [Gon92].

El reconocimiento de patrones en imágenes digitales consiste en identificar la presencia de los mismos dentro de la imagen. Los patrones que se analizan pueden ser formas, texturas, imágenes complejas con formas y colores, un conjunto de pares atributo-valor, etc.

Existen diferentes técnicas para realizar el reconocimiento de un patrón Px en una imagen, una de éstas consiste en hallar exactamente Px dentro de la imagen (por ejemplo en *pattern matching*). Otra alternativa consiste en hallar una subimagen Yx similar a Px sin necesidad de que sea idéntica. Las diferencias que se toleran pueden referirse a tamaño, rotación, pequeñas deformaciones y otras variaciones que no alteran sustancialmente las características de Px.

En los procesos de reconocimiento de patrones en general se identifican dos etapas. En la primera fase se extraen los atributos que describen a la imagen que se analiza y en la segunda fase se clasifican dichos atributos con el fin de determinar si se corresponden o no con el patrón. Sin embargo, los procesos de reconocimiento de patrones son variados, ya que se pueden tener:

- Patrones que se reconocen por la coincidencia exacta con la imagen analizada. Rara vez se requiere un análisis de este tipo, ya que la forma en que se generan las imágenes digitales hacen imposible obtener imágenes idénticas.
- Búsqueda en la imagen de alguna porción que se parezca mucho al patrón, aunque no sea exactamente igual; el patrón buscado no podrá entonces estar rotado o escalado con respecto al patrón original. En este caso es muy apropiada la técnica de correlación [Jai89].
- Buscar en la imagen la presencia del patrón como una porción de la imagen donde ciertos atributos clasificadores del patrón (valores de esos atributos) se encuentran con valores

similares dentro de cierta tolerancia admisible. Para esto se debe: (1) extraer los atributos del patrón (o recibir el patrón de esta manera), (2) analizar qué valores toman esos atributos para el patrón dado, (3) definir un margen de tolerancia para cada atributo, y (4) luego analizar si una imagen satisface el patrón si ocurre que al extraerle esos mismos atributos, el valor de cada uno de ellos se halla dentro de los límites de tolerancia para el patrón dado.

Como una primera etapa para problemas de reconocimiento de patrones simples se analiza la identificación de tres tipos de figuras geométricas simples: cuadrados, círculos y triángulos, a partir de imágenes digitalizadas de las mismas [Cos95] [Fel97]. Todo el reconocimiento se lleva a cabo por medio de atributos. Esta clase de soluciones se orienta al funcionamiento de robots simples de manipulación, que pueden reconocer un objeto en el plano para tomarlo o realizar otra acción controlada por software. Se considera adecuado utilizar una técnica de reconocimiento basada en atributos, a fin de no depender de ubicación, orientación o tamaño de la figura específica.

Los atributos a analizar pueden basarse en técnicas conocidas como correlación, *signature* o identificación de esqueleto [Gon92]. En el caso que se presenta, se define un algoritmo apropiado para las clases de formas geométricas de interés y que no depende del tamaño o posición relativa de las figuras. Los pasos del procesamiento de una imagen bilevel son:

- Extraer el borde de la figura junto con la superficie y el perímetro. La superficie es la cantidad de puntos negros que se encuentran sobre la imagen, y el perímetro es la longitud (cantidad de pixels) del borde de la figura.
- Buscar los dos puntos más extremos del borde de la figura.
- Identificar si la figura es un círculo, un cuadrado o un triángulo teniendo en cuenta el perímetro y la superficie de la figura. Tanto el perímetro como la superficie son conocidos para cada una de las figuras de la imagen, y también se conoce el *valor teórico* para las figuras buscadas.
- Si una figura no es reconocida como círculo, cuadrado o triángulo, se dice que la figura no tiene clasificación.

Reconocimiento de Círculos

Si la figura que se analiza es un círculo, la distancia entre los dos puntos extremos del borde será el diámetro (d) y el radio está definido por $r = d/2$. Sabiendo que el perímetro de un círculo está dado por $p = \pi * d$, y la superficie por $s = \pi * r^2$, se calculan ambos, y se comparan con la superficie y perímetro reales de la figura. Si la comparación de ambas medidas es exitosa, es decir que tanto superficie como perímetro reales (los tomados de la figura), y los de un círculo con diámetro d coinciden con un margen de error acotado de forma apropiada, entonces se afirma que la figura es un círculo.

Reconocimiento de Cuadrados

Si la figura que se analiza es un cuadrado, la distancia entre los dos puntos más extremos del borde constituye la distancia de la diagonal (dg). Tanto para calcular la superficie como el perímetro, lo que se necesita es la longitud del lado del cuadrado (l), el cual se obtiene a partir de la diagonal utilizando el Teorema de Pitágoras. La superficie del cuadrado está dada por $s = l^2$ y el perímetro por $p = 4l$. Para decidir si la figura de la imagen es un cuadrado se

comparan superficie y perímetro reales (los extraídos de la figura) con los de un cuadrado, análogamente a lo analizado para el círculo. Si esta comparación es exitosa con un margen de error acotado apropiadamente, no se asegura que la figura sea un cuadrado, ya que perímetro y superficie pueden compensarse de manera que un rectángulo pueda ser aceptado. Para solucionar este problema se verifica que los lados del cuadrado tengan la misma longitud. La forma de llevar a cabo este cálculo consiste en hallar los cuatro puntos del cuadrado (dos de éstos son los puntos más extremos de la figura) y verificar que el borde los contenga. Para esta verificación nuevamente se tienen en cuenta pequeñas variaciones de la figura con respecto al cuadrado *teórico*. Si los lados tienen la misma longitud se considera que la figura es un cuadrado.

Reconocimiento de Triángulos

Dados los dos puntos más alejados del borde, si se supone que la figura es un triángulo, la distancia entre estos dos puntos la longitud de la base (b). Para calcular la longitud de los otros dos lados del triángulo se necesita conocer el punto que corresponde al tercer vértice. Para hallar el tercer punto se toma el punto más distante a la base en línea perpendicular a ella. Una vez obtenido este punto, se tienen la altura (h) y las longitudes de los lados del triángulo. El perímetro está dado por la suma de las longitudes de los lados y la superficie se calcula como $s = (b*h)/2$. Como en los casos anteriores, estos atributos del triángulo *teórico* se comparan con los atributos de la figura de la imagen y se decide en base a un margen de error acotado.

Paralelización del Proceso de Reconocimiento

La clase de algoritmo presentada plantea posibilidades interesantes de paralelización, de las cuales se explica una que consta de dos fases. En la primera fase, dada una figura F en una imagen, un proceso M la divide por filas en tantas partes como unidades de procesamiento se dispongan. En cada una de estas unidades reside un proceso E_i que extrae el borde y calcula la superficie de la porción de la imagen que le fue asignada. A continuación, los datos extraídos son devueltos al proceso M donde se arma el borde tomando los subbordes recibidos y se calcula la superficie total. Además, se determina la longitud del perímetro y se identifican los dos puntos más alejados de F.

En la segunda fase, los datos obtenidos en la fase anterior (borde, superficie y perímetro) son transmitidos a tres procesos (en lo posible, residentes en tres procesadores distintos). Cada uno de los procesos se encarga de reconocer una de las posibles figuras de clasificaciones:

- R_{Cir}: analiza si es círculo.
- R_{Cua}: analiza si es cuadrado.
- R_{Tri}: analiza si es triángulo.

Estos tres procesos devuelven el resultado de éxito o no en el reconocimiento al proceso M, el cual proporciona el resultado final.

La Fig. 7.25 muestra esquemáticamente el desarrollo en el tiempo de las dos fases del algoritmo. Cada proceso de extracción de borde y superficie de una porción de la imagen (E_i) se puede ejecutar simultáneamente con respecto a los demás. Cada uno de los procesos que reconoce un tipo de figura geométrica también se puede ejecutar de forma simultánea con respecto a los otros dos.

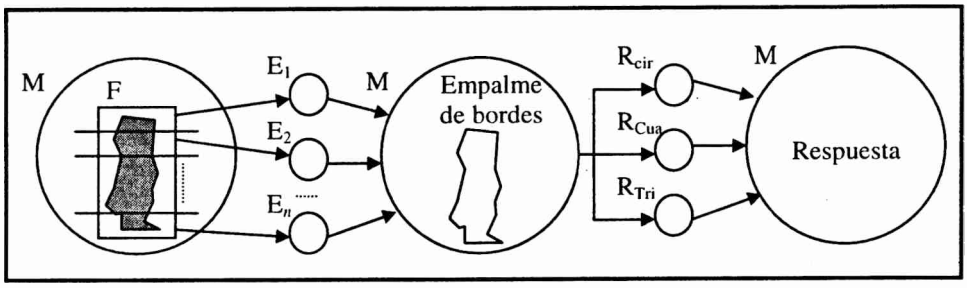


Figura 7.25: Reconocimiento de una Figura Geométrica Simple.

Extensión del Reconocimiento

A partir de haber resuelto el problema de clasificación de una figura dentro de una imagen, se puede plantear una extensión considerando la clasificación de más de una figura por imagen. La Fig. 7.26 muestra una imagen con más de una figura geométrica a reconocer.

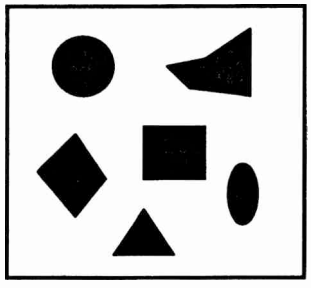


Figura 7.26: Figuras Geométricas a Reconocer en una Imagen.

Para resolver esta extensión, la idea es llevar a cabo un preprocesamiento sobre la imagen de entrada. En este preprocesamiento se aísla cada figura y una vez aislada, se procesa por separado de las demás como en la Fig. 7.25. La manera de aislar las figuras consiste en que un proceso M divida la imagen de entrada por filas en tantas partes como procesadores se tengan para que en cada uno de ellos un proceso E_i extraiga las porciones de borde de las figuras que allí se encuentren. A continuación, cada proceso E_i retorna el conjunto de bordes que halló. El proceso M entonces, con todos los subbordes hallados en la imagen, realiza el empalme de los bordes consiguiendo de esta manera obtener el marco de cada figura. A partir de estos bordes distintos procesos se encargarán de realizar la clasificación como se explicó anteriormente.

La Fig. 7.27 muestra esquemáticamente el procesamiento paralelo de una imagen para el reconocimiento de las figuras geométricas que contiene.

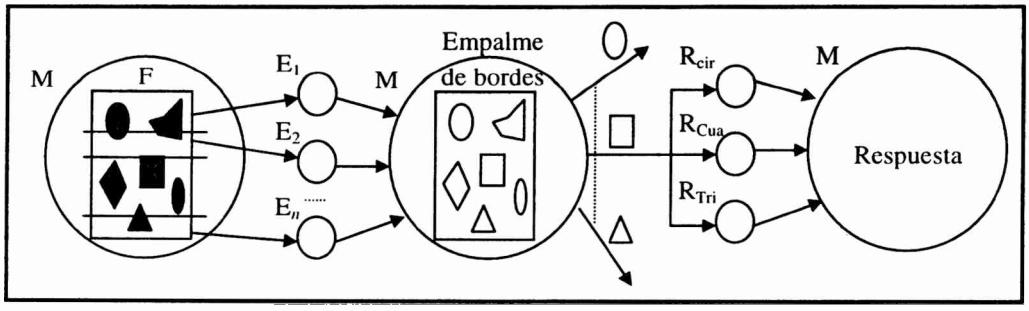


Figura 7.27: Reconocimiento de Varias Figuras Geométricas Simples.

La eficiencia de la paralelización que muestra la Fig. 7.27 depende fundamentalmente del hardware disponible. Para el proceso simple de análisis de una sola figura, resulta aceptable utilizar un mínimo de tres procesadores. A partir de la extensión a varias figuras por imagen de entrada, es conveniente contar al menos con $3n$ procesadores, donde n es la cantidad de figuras en la imagen, para aprovechar al máximo la independencia en el análisis de las distintas figuras de la imagen.

En este caso, se debe aclarar que el proceso M tiene bastante más tarea para realizar porque debe empalmar los perímetros de todas las imágenes. Tanto la paralelización de la Fig. 7.25 como la paralelización de la Fig. 7.27 se puede implementar independientemente de la cantidad de procesadores disponibles. La diferencia se notará esencialmente en el tiempo de ejecución de la aplicación. En ambos casos también, el límite máximo para la cantidad de procesadores a utilizar en la primera fase (extracción de bordes de la/s figura/s) se debe analizar con respecto al tiempo de empalme de los bordes hallados. De forma análoga, el límite máximo para la cantidad de procesadores a utilizar en la segunda fase tiene relación directa con la cantidad de figuras geométricas que se intentan reconocer.

7.3.3 Compresión de Imágenes Utilizando JPEG

Como se ha detallado antes, el volumen de las imágenes digitales es importante aún con baja resolución. Desde el principio de la incorporación de las imágenes en las computadoras se ha intentado almacenarlas y/o transmitirlas con la mínima cantidad de datos posible. Como resultado, se han desarrollado y utilizado varios métodos de compresión, y uno de los más importantes y representativos es el producido por el Joint Photographic Expert Group (JPEG) [ISO93] [Wal91].

El estándar establecido por el JPEG es el considerado más completo en compresión de imágenes estáticas y de hecho se han desarrollado chips que ejecuten *compresión JPEG* [Kim92]. Al reunirse para establecer un estándar de compresión de imágenes estáticas, el grupo se propuso una serie de objetivos, entre los cuales pueden destacarse:

- Parametrizable: el usuario puede elegir el grado de compresión deseada.
- Aplicable a prácticamente todas las imágenes, sin importar la fuente.
- Implementable en hardware y software.
- Soporte de diferentes modos de operación: secuencial, progresivo, sin pérdida y jerárquico.

Estas características fijadas a priori resultaron en el estándar que soporta cuatro modos de operación, tres de los cuales son con pérdidas. En los métodos con pérdidas se utiliza la transformada discreta coseno (DCT: Discrete Cosine Transform) y en el método sin pérdidas se utiliza predicción. En los tres modos se utiliza compresión estadística [Ger92].

En la Fig. 7.28 se muestra esquemáticamente el modo de operación secuencial, también denominado *baseline*. Este modo de operación es con pérdidas, que se producen fundamentalmente en la cuantificación de los coeficientes que resultan de la DCT. Aunque este modo se identifica como *secuencial* no necesariamente debe ser implementado en una computadora monoprocesador. La sucesión de etapas del procesamiento es sin lugar a dudas es secuencial, ya que, por ejemplo, la cuantificación debe realizarse sobre los coeficientes que resultan de la DCT, y la codificación estadística se lleva a cabo en último término, una vez que se han ordenado de forma zig-zag los coeficientes cuantificados.

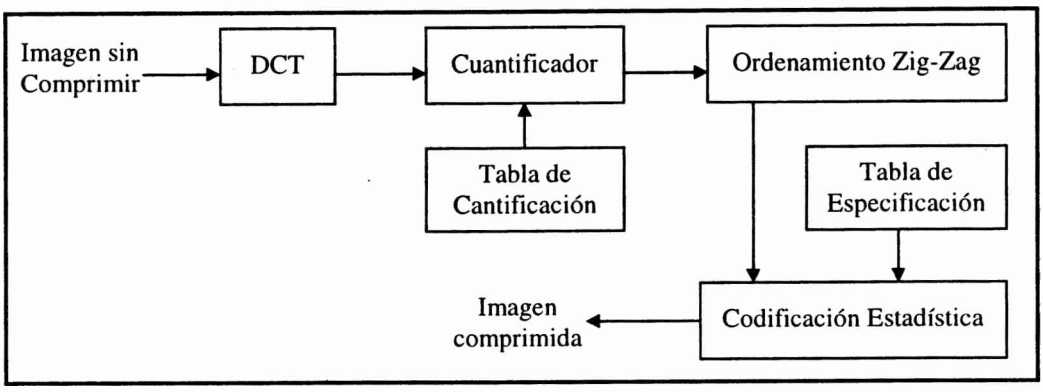


Figura 7.28: Modo Secuencial de Compresión JPEG.

La DCT se realiza sobre un bloque de 8x8 pixels adyacentes, y produce 64 valores (coeficientes) de salida. Estos 64 coeficientes representan las amplitudes de los componentes de frecuencias espaciales en las dos dimensiones, como se muestra en la Fig. 7.29 y se denominan coeficientes DCT.

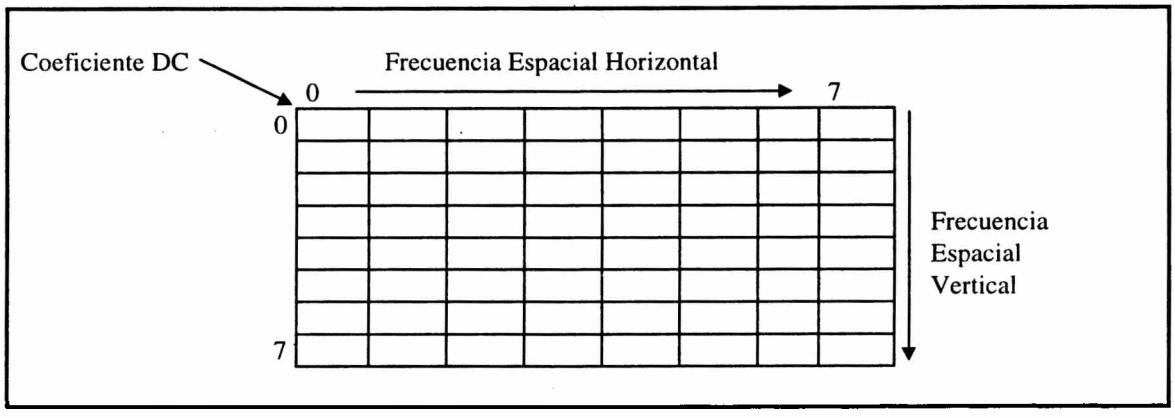


Figura 7.29: Matriz de Coeficientes DCT.

El coeficiente de la frecuencia espacial cero es llamado coeficiente DC (Direct Current) y es el valor promedio de todos los pixels en el bloque. Los 63 coeficientes restantes son denominados coeficientes AC (Alternate Current) y representan las amplitudes de las frecuencias espaciales horizontales y verticales progresivamente mayores. Como los valores de pixels adyacentes tienden a ser muy similares, el procesamiento DCT provee una buena vía de compresión porque hace que el componente de menor frecuencia espacial tenga la mayor parte de la energía y sea el más importante de codificar. En la mayoría de los casos, muchos de los coeficientes AC de mayor frecuencia pueden ser ignorados en la codificación.

El ordenamiento Zig-Zag posterior a la DCT y a la cuantificación se utiliza para que la codificación estadística sea más efectiva. Los coeficientes que son más probablemente distintos de cero se ponen antes en la secuencia que aquellos que serán cero con mayor probabilidad. El coeficiente DC de cada bloque se codifica de modo DPCM con respecto al coeficiente DC del bloque anterior. En la codificación estadística se pueden utilizar el método Huffman o el método aritmético y la codificación *runlength* [Ger92]. El método más utilizado ha sido la codificación Huffman, y para la para la secuencia de coeficientes DCT iguales a cero se ha utilizado *runlength*.

Las arquitecturas de los demás modos de operación JPEG con pérdidas son muy similares al modo secuencial. El modo de operación sin pérdida utiliza un predictor para comprimir (además de la posterior codificación estadística) y puede esquematizarse como en la Fig. 7.30.

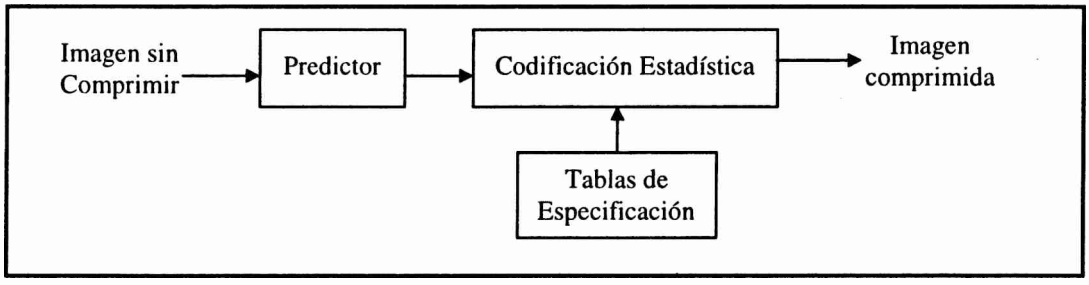


Figura 7.30: Modo de Compresión JPEG Sin Pérdida.

El predictor de un pixel tiene siete clases de predicción disponibles y elige cuántos valores de pixels adyacentes puede utilizar. El predictor se define en función de los vecinos de un pixel, y el vecindario de un pixel X a su vez se define como se muestra en la Fig. 7.31.

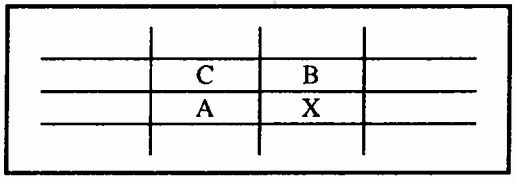


Figura 7.31: Vecindario de Predicción de un Pixel.

La tabla de predicción de un pixel se muestra en la Fig. 7.32, donde se puede ver que para cada pixel se tienen siete valores posibles de predicción. El valor de predicción cero (no predicción), hace posible que este modo de operación no tenga pérdida. El resultado Una vez que la imagen comprimida se descomprime, el resultado es la imagen original.

Valor de Selección	Tipo de Predicción
0	No hay predicción
1	X = A
2	X = B
3	X = C
4	X = A + B - C
5	X = A + (B - C) / 2
6	X = B + (A - C) / 2
7	X = (A + B) / 2

Figura 7.30: Tabla de Predicción JPEG.

Paralelización

La cantidad de operaciones a realizar para cualquier modo de compresión JPEG justifica que se busque la posibilidad de paralelización del procesamiento. Por ejemplo, para realizar la DCT en un bloque de 8x8 pixels utilizando un algoritmo de multiplicación de matrices son necesarias 896 sumas y 1024 multiplicaciones [Kim92]. Si se multiplica esta cantidad de

operaciones por la cantidad de bloques de 8x8 pixels de una imagen se llegará rápidamente a justificar la paralelización del procesamiento siempre que sea posible.

El cálculo de cada DCT sobre un bloque de la imagen es totalmente independiente del procesamiento de todos los demás bloques de la imagen y por lo tanto es un buen candidato para la paralelización. En este sentido, el mayor grado de paralelismo estaría dado por el cálculo simultáneo de todas las DCT sobre los bloques de 8x8 pixels de la imagen. Si, por ejemplo, se tiene una imagen de entrada de 640 x 480 pixels se podrían llevar a cabo 4800 DCTs simultáneamente. Además, si la imagen está codificada en planos RGB o YUV la cantidad de DCTs simultáneas puede multiplicarse por tres.

El modo de operación sin pérdidas también puede paralelizarse. En este caso se puede paralelizar de distintas formas y no necesariamente dividiendo la imagen en bloques fijos de 8x8 pixels. Lo único que se necesita para la codificación (predicción) de un pixel son los valores de los pixels vecinos y dado que la codificación es sin pérdida, no se necesita conocer el resultado de la codificación de los pixels del vecindario sino solamente sus valores en la imagen original. Por ejemplo, si se divide la imagen por filas, cada proceso de codificación (predicción) necesita una fila previa a la primera que codifica y esto a lo sumo produce una copia más de algunas filas pero no cambia el resultado final ni secuencializa el procesamiento. La paralelización en este caso depende de la cantidad de procesadores y de la granularidad óptima de la arquitectura. Se deben balancear los requerimientos de comunicación de un conjunto de datos con respecto a los requerimientos de cómputo para procesar esos datos.

El proceso de cuantificación de los coeficientes DCT no es tan inmediata con respecto a la paralelización, porque los coeficientes DC se codifican de modo DPCM (Differential Pulse Code Modulation). Esto significa que para codificar (en este caso cuantificar) el coeficiente DC de un bloque de 8x8 pixels, se necesita el valor del coeficiente DC del bloque anterior. No sucede lo mismo con respecto a los coeficientes AC, que pueden cuantificarse independientemente de los demás.

En los modos de operación con pérdida o sin pérdida, la codificación estadística con el método Huffman puede paralelizarse. Una vez que se tiene disponible la tabla de correspondencia de símbolos, cada coeficiente DCT o cada valor de predicción se puede codificar independientemente de los demás.

Se debe notar que todo el análisis realizado para la paralelización se basa en la independencia o no de los datos con respecto al procesamiento. Esto es usual en el contexto del procesamiento de imágenes en general y en particular en los algoritmos de compresión. En general, se buscan los procesos con mayor carga de trabajo, que operan sobre toda la imagen y se analiza si el mismo proceso se puede aplicar a un bloque de pixels vecinos de forma más o menos independiente de los demás. La independencia o no indicará la forma de paralelizar o al menos proporciona una pauta importante en la posibilidad de la paralelización del proceso.

A nivel un poco más general, se podría pensar en la implementación de un pipeline para llevar a cabo todas las tareas de la compresión JPEG sobre varias imágenes. Por ejemplo en los modos con pérdida cada uno de los procesos que se aplican a la imagen y que se muestran esquemáticamente en la Fig. 7.28 puede ser una etapa del pipeline. A su vez, cada etapa se puede implementar de forma paralela o no, dependiendo de la etapa y de los requerimientos de la aplicación. En este caso, se debe aclarar que las etapas resultantes son

muy heterogéneas, y por lo tanto el tiempo de procesamiento de cada una de ellas puede ser muy distinto al de las demás. Como en todo pipeline, el paso de una etapa a la siguiente depende del mayor tiempo de operación entre *todas* las etapas y de esta manera se puede perder tiempo de procesamiento general por el desbalance de tiempos entre las etapas. Este problema no es fácil de solucionar porque depende en principio del procesamiento propio de las etapas y de la implementación que se haga de cada una de ellas.

8. Algoritmos de Administración de Recursos

Desde el punto de vista del software, la tarea más importante para el procesamiento paralelo es la *especificación de la ejecución concurrente de procesos.* Los procesos residentes en distintos procesadores cooperan y compiten: cooperan cuando pueden ejecutarse en forma independiente (simultánea o verdaderamente paralela) en cada procesador y compiten cuando deben sincronizarse para intercambiar datos o control.

Como mínimo, conviene tener en cuenta que ningún sistema de software de aplicación puede ser especificado en forma totalmente paralela y por lo tanto se produce una degradación del máximo rendimiento teórico de una arquitectura paralela. En principio, la degradación es función del porcentaje de código secuencial presente en la aplicación [Amd67]. Por otro lado, se tiene una segunda fuente real de reducción en el máximo factor de Speed-Up alcanzable: la utilización de recursos compartidos entre los procesos.

En un sistema paralelo siempre es posible la existencia de recursos compartidos (memoria, datos en disco, canal de comunicaciones, dispositivos periféricos de E/S, etc.). El desarrollo de algoritmos eficientes de administración (scheduling) de estos recursos es un aspecto importante, pocas veces tratado en los textos de programación paralela.

La intención en este capítulo no es llegar a programas paralelos con la expresión matemática del factor de Speed-Up asociado, tal como en los algoritmos que suponen que todos los datos están disponibles en los procesadores y que el procesamiento no requiere recursos compartidos. Se analizarán algunos problemas de administración de recursos compartidos que se pueden encontrar en las arquitecturas paralelas, con el objetivo de analizar el rendimiento real de toda la aplicación.

Se utiliza el lenguaje de programación Ada como herramienta de especificación del esquema de los algoritmos por su claridad sintáctica y su precisión semántica. Para el lector no familiarizado con el lenguaje Ada, en el Ap. B se hace un breve resumen de los aspectos útiles para la especificación de tareas concurrentes que se ejecutan sobre una arquitectura paralela.

En general, los problemas de administración de recursos pueden tener soluciones centralizadas o distribuidas. En una solución centralizada se tiene un proceso Cc (y por lo tanto su procesador de residencia) que controla el recurso. Para acceder al recurso, los demás procesos deben comunicarse con Cc y requerir la asignación. Se suele decir que el proceso de administración del recurso es un servidor y que los procesos que solicitan el acceso al recurso compartido son clientes.

En las soluciones distribuidas, la administración del recurso compartido puede estar repartida entre m procesos residentes en diferentes procesadores. Normalmente la administración se puede llevar a cabo de forma más eficiente para los clientes, pero es más compleja para el mantenimiento, integridad y consistencia del recurso compartido. Por ejemplo, en una base de datos distribuida puede ser modificada al mismo tiempo por distintos clientes sobre diferentes servidores.

Los sistemas cliente-servidor (C-S) se basan en el modelo conceptual de interacción

entre procesos. Dos procesos remotos interactúan intercambiando datos y/o control a través de mensajes. El proceso que controla recursos compartidos es el servidor. En los ejemplos que siguen se tratarán aplicaciones de administración de recursos compartidos, utilizando el modelo C-S. En el lenguaje Ada resulta natural la especificación de la comunicación entre procesos (TASKs) que se ejecutan en paralelo, cada uno de los cuales puede ser cliente o servidor.

8.1 Administrador/Servidor Centralizado

En esta sección se analiza un administrador centralizado ProcAC1, por ejemplo de una base de datos (BD), con m funciones y n clientes replicados con diferentes tipos de requerimientos y prioridades. Para distinguir los n clientes entre sí, se los define como n_1 procesos lectores de la BD y n_2 procesos escritores, con $n = n_1 + n_2$. Todos los procesos cliente deben comunicarse con el proceso ProcAC1 para acceder a la base de datos (BD) que es el recurso compartido.

Los procesos lectores y los procesos escritores se excluyen mutuamente, del mismo modo que los escritores entre sí. Esto significa que en cualquier instante de tiempo puede haber más de un proceso lector operando simultáneamente en la BD, pero solamente un proceso escritor operando. Expresado en términos de restricciones, no puede haber procesos escritores y procesos lectores, ni más de un proceso escritor operando simultáneamente en la BD. Estas restricciones de operación simultánea de los procesos clientes en la BD definen en gran parte la política de acceso al recurso que se comparte.

En la Fig. 8.1 se muestra la solución básica del administrador centralizado ProcAC1, tal como es tratada en [Hab93], priorizando el acceso de los procesos escritores a la BD. Se puede afirmar que ProcAC1 tiene tantas funciones como cláusulas ENTRYs tiene la tarea o proceso (TASK) de Ada. En este caso, el proceso ProcAC1 tiene $m = 4$ funciones:

1. Permitir el acceso de los procesos lectores de la BD: ENTRY InicioLeer en la definición del proceso y ACCEPT InicioLeer en la implementación del proceso. Se debe asegurar que no haya un proceso escritor operando en la BD y como consecuencia de la entrada de un proceso lector, se debe inhibir la entrada a la BD de cualquier proceso escritor.
2. Registrar la finalización del trabajo de los procesos lectores de la BD: ENTRY y ACCEPT FinLeer.
3. Permitir el acceso de los procesos escritores de la BD: ENTRY y ACCEPT InicioEscribir. Se debe tener en cuenta que no haya ningún otro proceso escritor ni lector operando en la BD. A su vez, la entrada de un proceso escritor debe inhibir la entrada a la BD de cualquier otro proceso.
4. Registrar la finalización del proceso escritor de la BD: ENTRY y ACCEPT FinEscribir.

Cada proceso lector, antes de operar sobre de la BD requiere el acceso al administrador ProcAC1 y éste debe ejecutar el ACCEPT InicioLeer. Al concluir la operación de lectura sobre la BD, el proceso debe notificar al proceso administrador que no necesita acceder al recurso compartido, y el proceso ProcAC1 Debe ejecutar el ACCEPT FinLeer. De forma análoga, cada proceso escritor debe utilizar el ENTRY InicioEscribir antes de operar sobre la BD y debe utilizar el ENTRY FinLeer una vez finalizado el acceso al recurso compartido que administra el proceso ProcAC1. Sin este *protocolo* de acceso a la BD los procesos lectores y los procesos escritores pueden interferir en sus tareas respectivas y producir inconsistencia en los datos.

```
TASK TYPE ProcAC1 IS          {Definición del administrador de acceso}
  ENTRY InicioLeer;           {Entrada a la BD de procesos lectores}
  ENTRY FinLeer;              {Salida de la BD de procesos lectores}
  ENTRY InicioEscribir;       {Entrada a la BD de procesos escritores}
  ENTRY FinEscribir;          {Salida de la BD de procesos escritores}
END ProcAC1;

TASK BODY ProcAC1 IS
  NumLectores: INTEGER := 0;

BEGIN
   LOOP
     SELECT
          WHEN (InicioEscribir'COUNT = 0) =>   {Prioriza los escritores}
          ACCEPT InicioLeer Do
               NumLectores := NumLectores+1;
          END InicioLeer;
     OR
          ACCEPT FinLeer DO
               NumLectores := NumLectores-1 ;
          END FinLeer;
     OR
          WHEN (NumLectores = 0) => {Exclusión mutua con los lectores}
          ACCEPT InicioEscribir;
          ACCEPT FinEscribir;     {Bloquea hasta que termina el escritor}

          {Esta iteración da prioridad a los lectores pendientes}
          FOR i IN (1..InicioLeer'COUNT) LOOP
               ACCEPT InicioLeer DO
                    NumLectores := NumLectores +1;
               END InicioLeer;
          END LOOP;
     END SELECT;
   END LOOP;
END ProcAC1;
```

Figura 8.1: Administrador Centralizado con Prioridad para los Procesos Escritores.

En la Fig. 8.2 se muestra la solución para un administrador centralizado ProcAC2 que es similar al administrador ProcAC1 pero prioriza el acceso de los procesos lectores a la BD. Tanto la prioridad que tienen los procesos escritores en el administrador ProcAC1 como la prioridad que tienen los procesos lectores en el administrador ProcAC2 se producen por las guardas en las instrucciones ACCEPT InicioLeer y ACCEPT InicioEscribir respectivamente. En el caso del proceso ProcAC1, no se habilita la ejecución de ACCEPT InicioLeer si hay uno o más procesos escritores esperando el acceso a la BD (en el ENTRY InicioEscribir). De forma análoga, en el proceso ProcAC2 no se habilita la ejecución de ACCEPT InicioEscribir si hay uno o más procesos lectores esperando el acceso a la BD (en el ENTRY InicioLeer).

```
TASK TYPE ProcAC2 IS          {Definición del administrador de acceso}
  ENTRY InicioLeer;           {Entrada a la BD de procesos lectores}
  ENTRY FinLeer;              {Salida de la BD de procesos lectores}
  ENTRY InicioEscribir;       {Entrada a la BD de procesos escritores}
  ENTRY FinEscribir;          {Salida de la BD de procesos escritores}
END ProcAC2;

TASK BODY ProcAC2 IS
    NumLectores: INTEGER := 0;

BEGIN
   LOOP
      SELECT
          WHEN (NumEscritores = 0) =>  {Exclusión mutua con los escritores}
              ACCEPT InicioLeer DO
              NumLectores := NumLectores+1;
          END InicioLeer;
      OR
          ACCEPT FinLeer DO
               NumLectores := NumLectores-1 ;
          END FinLeer;
      OR
          {Dar prioridad a los lectores pendientes a los escritores}
          WHEN ((InicioLeer'COUNT = 0) AND (NumLectores = 0)) =>
              ACCEPT InicioEscribir;
              ACCEPT FinEscribir;     {Bloquea hasta que termina el escritor}

          {Esta iteración da prioridad a los lectores pendientes}
          FOR i IN (1..InicioLeer'COUNT) LOOP
              ACCEPT InicioLeer DO
                  NumLectores := NumLectores+1;
              END InicioLeer;
          END LOOP;
      END SELECT;
   END LOOP;
END ProcAC2;
```

Figura 8.2: Administrador Centralizado con Prioridad para los Procesos Lectores.

Por último, se puede discutir la posibilidad de restringir el número de procesos lectores simultáneos a un máximo de k y limitar el tiempo de acceso a la BD por parte de los procesos lectores a un máximo *tmax*. Para limitar el número de lectores simultáneos a k bastaría en el administrador ProcAC1 cambiar la condición de acceso al cliente lector:

```
WHEN (InicioEscribir'COUNT = 0) =>      {Prioriza los escritores}
```

por

```
WHEN ((InicioEscribir'COUNT = 0) AND (NumLectores < k)) =>
```

Para limitar cada proceso lector a un tiempo máximo *tmax*, una posible solución puede definir que al recibir el control de acceso a la BD cada proceso lector dispare un timer por el período de tiempo *tmax*. Si el proceso excede el tiempo máximo (conocido por el timer), debe autolimitarse por medio del requerimiento FinLeer al administrador. Esta solución puede denominarse *distribuida* entre los procesos, ya que cada proceso controla la cantidad de tiempo que utiliza la BD.

La solución centralizada a la restricción de tiempo máximo de utilización de la BD no es muy eficiente. El administrador debería (en el momento que da el control de la BD a un proceso lector) crear un proceso timer asociado con el cliente al que acaba de otorgar el acceso a la BD y permanentemente controlar si alguno de los timers excede el tiempo límite. En el caso de suspender el acceso a un proceso lector debe cancelar su permiso de acceso a la BD o enviarle un mensaje para que el proceso lector mismo se autolimite. Estos procesos timer pueden a su vez ser lógicos o residir físicamente en otro procesador, lo cual mejora el rendimiento del administrador pero incrementa la sobrecarga (overhead) de comunicaciones.

Hasta ahora se tienen dos políticas de administración del acceso a la BD y ambas respetan las restricciones que se enunciaron al principio: (1) no pueden operar de forma simultánea sobre la BD procesos lectores y procesos escritores, y (2) no puede operar más de un proceso escritor de forma simultánea sobre la BD. Una de las políticas se corresponde con el administrador centralizado AC1, que se basa en dar prioridad a los procesos escritores en el acceso de la base de datos. Esto significa que (3) no se puede permitir el acceso a la BD de un proceso lector si hay algún proceso escritor esperando que se le otorgue el permiso de acceso. La otra política se corresponde con el administrador centralizado AC2, que se basa en dar prioridad a los procesos lectores en el acceso a la BD. Esto significa que (3') no se puede permitir el acceso a la BD de un proceso escritor si hay algún proceso lector esperando que se le otorgue el permiso de acceso.

Las distintas políticas de administración de la BD tienen impacto directo sobre el tiempo de respuesta de los procesos clientes que comparten la BD. Se puede analizar una secuencia temporal de pedidos de procesos lectores y procesos escritores para luego determinar el máximo retardo en tiempo de respuesta dependiendo del administrador AC1 y del administrador AC2. La Fig. 8.3 muestra un ejemplo de secuencia temporal referente a requerimientos de procesos clientes al proceso administrador.

Tiempo de Llegada	Cliente	Tiempo de Uso
0	L_1	5
1	E_1	6
2	L_2	4
7	L_3	5
8	L_4	2
9	L_5	3
10	E_2	2

Figura 8.3: Ejemplo de Requerimientos al Administrador de la BD.

En la Fig. 8.3 cada proceso cliente se identifica con L_i para los procesos lectores de la BD o con E_j para los procesos escritores de la BD. Los índices asociados a cada proceso cliente (i o j) solamente se utilizan para identificar el orden en que se llevan a cabo los

requerimientos. El tiempo de llegada 0 identifica el tiempo del primer requerimiento de acceso a la BD, en el cual se asume que está libre, no hay ningún otro proceso que esté operando sobre ella. El "Tiempo de Uso" que aparece en la Fig. 8.3 se refiere al lapso de tiempo en el cual el proceso cliente debe operar sobre la BD.

En la Fig. 8.4 se muestra la secuencia temporal de acceso a la BD que se deriva de los requerimientos de la Fig. 8.3 y el administrador ProcAC1 de la Fig. 8.1. Las líneas de puntos indican la espera de los procesos clientes por el permiso de acceso a la BD que debe otorgar el proceso ProcAC1.

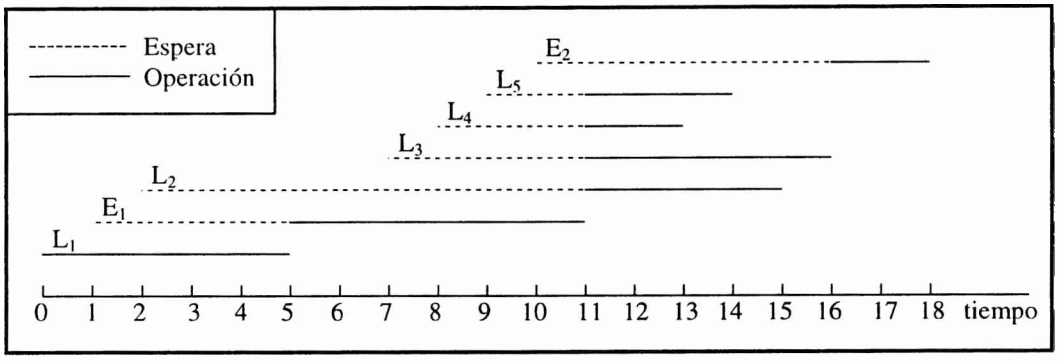


Figura 8.4: Acceso de los Clientes a la BD con Prioridad de Escritores.

En la Fig. 8.4 se puede ver por ejemplo, que el proceso lector L_2 no puede ingresar a la BD mientras L_1 está operando porque el proceso escritor E_1 también está esperando por el permiso de acceso. El administrador centralizado AC1 no le otorga el permiso de acceso al recurso compartido al proceso lector L_2 porque de acuerdo con su política de asignación no puede ingresar un proceso lector mientras está esperando un proceso escritor.

En la Fig. 8.5 se muestra la secuencia temporal de acceso a la BD que se deriva de los requerimientos de la Fig. 8.3 y el administrador ProcAC2 de la Fig. 8.2. De forma similar a la Fig. 8.4, las líneas de puntos indican la espera de los procesos clientes por el permiso de acceso a la BD que debe otorgar el proceso ProcAC2.

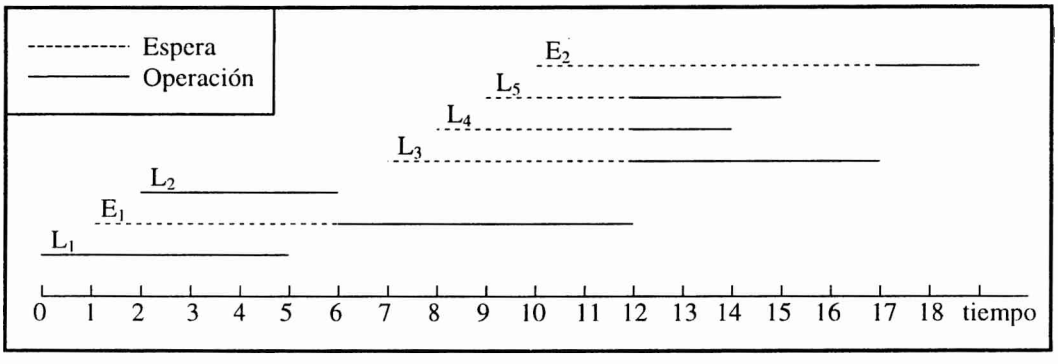


Figura 8.5: Acceso de los Clientes a la BD con Prioridad de Lectores.

De la comparación de la Fig. 8.4 con la Fig. 8.5 surge de forma casi inmediata que una mayor cantidad de procesos paralelos durante un intervalo de tiempo no necesariamente reduce el tiempo total de ejecución de un conjunto de tareas que comparten un recurso. Por esta razón, se debe analizar muy bien cada política de asignación de recursos y se deben tener

en cuenta los requerimientos de la aplicación.

En la Fig. 8.6 se muestran los tiempos de finalización de cada proceso dependiendo del administrador que se utilice para gestionar el acceso al recurso compartido.

Proceso Cliente	Tiempo de Finalización	
	ProcAC1	ProcAC2
L_1	5	5
L_2	15	6
L_3	16	17
L_4	13	14
L_5	14	15
E_1	11	12
E_2	18	19

Figura 8.6: Tiempos de Finalización de los Procesos Clientes.

El mayor grado de paralelismo (mayor cantidad de tareas paralelas en un intervalo de tiempo), se logra con el administrador ProcAC1 y coincide con el mejor tiempo de finalización de la ejecución de todos los procesos clientes. Por otro lado, las restricciones de acceso a la BD reduce desde el principio el grado de paralelismo ya que no pueden operar a la vez las tareas lectoras con las escritoras ni más de una tarea escritora. En este caso, la arquitectura no impone ningún límite al grado de paralelismo, sino que es la propia aplicación la que no lo permite. Esta es una de las razones por las cuales el factor de Speed-Up pierde importancia en las aplicaciones donde se requiere la administración de recursos compartidos.

Aunque el ejemplo que se ha analizado es muy sencillo, se puede notar que la diferencia en el tiempo de respuesta del proceso L_2 por ejemplo, es notable. Según el caso, esta diferencia en el tiempo de respuesta puede no ser aceptable a nivel de tiempo de respuesta de la aplicación al usuario. En este caso, en el que se trata el acceso a una base de datos se debe tener en cuenta que los procesos lectores son de consulta y los procesos escritores son de actualización y por lo tanto se debe tratar con especial atención la posibilidad de leer un dato por el cual un proceso escritor está esperando el acceso para actualizarlo. Dada la gran cantidad de casos posibles entre procesos clientes y combinación de requerimientos al administrador se deben tomar la mayor cantidad de precauciones posibles que surgen del contexto de aplicación, en este caso de la administración del acceso a la base de datos.

8.2 Administrador/Servidor Replicado

La utilización de un único administrador que controla el acceso a un recurso compartido no siempre es posible. En muchos casos, el recurso está distribuido físicamente en distintas computadoras y centralizar la administración implica que la ejecución de los procesos se secuencializa inútilmente. En este contexto (recurso distribuido físicamente) se requiere la utilización de varios servidores o administradores (distribuidos) del recurso que cooperan en la atención de los procesos clientes. Si bien la interface con los procesos clientes puede

permanecer sin cambios, la complejidad en la operación y cooperación de los procesos administradores crece considerablemente.

Para analizar el caso de procesos administradores replicados se continúa en el contexto del acceso a una base de datos (BD) por parte de n procesos clientes. Para simplificar, se supone que la BD está constituida por registros o archivos distribuidos en varias computadoras, de modo que no hay repetición del espacio de datos. Esto significa que cada proceso administrador de datos (AD_i) tiene un conjunto de archivos/registros locales propios y cuando se le demanda un archivo o registro que no está a su alcance (en el sistema de archivos o *filesystem* local), debe comunicarse con el servidor AD_j que dispone del mismo. También para simplificar la aplicación se supone que los n procesos clientes son todos procesos lectores de la base de datos.

En todos los casos, los requerimientos de lectura local se pueden satisfacer con los datos a los que accede cada administrador directamente. Los requerimientos de lectura remotos (lectura de datos no alcanzables localmente) exigen al proceso administrador que los recibe comunicarse con el servidor adecuado. En todos los casos también, el proceso administrador debe aceptar pedidos de consulta remota que lo obligan a trasmitir un dato local.

Cada proceso administrador $ProcAD_i$ debe atender dos tipos de clientes: los procesos lectores locales y los otros servidores que requieren la transmisión de un dato local. Los procesos lectores pueden realizar requerimientos locales, o pueden realizar requerimientos de datos remotos. En principio, el proceso $ProcAD_i$ no puede atender más de un servicio (lectura o consulta) remoto por vez, pero puede admitir múltiples requerimientos de lectores locales. Puede haber un requerimiento de un servidor remoto y otro requerimiento a un servidor remoto.

En la Fig. 8.7 se muestra la solución básica del administrador ProcAD1. A diferencia del problema de procesos lectores y procesos escritores de la BD, mientras un proceso servidor transmite o recibe datos de otro proceso servidor (servidor remoto), pueden existir procesos clientes lectores locales consultando la BD *local*. Se deben tener tantos procesos administradores de datos $ProcAD1_1$, ..., $ProcAD1_m$, como cantidad de divisiones *físicas* tiene la base de datos. En cada computadora donde se administra un subconjunto de datos se debe asignar un proceso administrador $ProcAD1_i$.

El proceso ProcAD1 tiene definidas seis cláusulas de tipo ENTRY, tres para requerir la *entrada* a la BD y tres para notificar la finalización de la operación o la *salida* de la BD. Un proceso siempre requiere el acceso a la BD a su proceso administrador $ProcAD1_i$ local. Si el proceso lector necesita acceder a los datos locales, entonces requiere el acceso por medio del ENTRY InicioLeerDatoLocal, y cuando finaliza la operación notifica la salida de la BD por medio del ENTRY FinLeerDatoLocal. Si el proceso lector necesita acceder a datos no locales, entonces requiere el acceso por medio del ENTRY InicioLeerDatoRemoto, y cuando finaliza la operación notifica la salida de la BD por medio del ENTRY FinLeerDatoRemoto. Los procesos administradores se *comunican* entre sí utilizando el ENTRY ConsultaRemota para requerir el envío de un dato de otro proceso servidor y el ENTRY FinConsultaRemota para notificar la finalización de la operación de consulta remota. Este último ENTRY puede utilizarse como una confirmación satisfactoria (*acknowledge*) de la operación.

```
TASK TYPE ProcAD1 IS                    {Permite crear un arreglo de TASKs con el mismo molde}
  ENTRY InicioLeerDatoLocal;            {Lector local – dato local}
  ENTRY FinLeerDatoLocal;
  ENTRY InicioLeerDatoRemoto;           {Lector local – dato remoto}
  ENTRY FinLeerDatoRemoto;
  ENTRY ConsultaRemota;                 {Requerimiento de otro ProcAD1 (lectura local)}
  ENTRY FinConsultaRemota;
END ProcAD1;

TASK BODY ProcAD1 IS
  NumLectoresLocales: INTEGER := 0;
  NumLectoresRemotos: INTEGER := 0;
  NumConsultasRemotas: INTEGER := 0;

BEGIN
   LOOP
      SELECT
         ACCEPT InicioLeerDatoLocal DO
              NumLectoresLocales := NumLectoresLocales+1;
              {El cliente local lee la BD local}
         END InicioLeerDatoLocal;
      OR
         ACCEPT FinLeerDatoLocal DO
              NumLectoresLocales := NumLectoresLocales-1;
         END FinLeerDatoLocal;
      OR
         WHEN (NumLectoresRemotos = 0) =>      {A lo sumo una consulta a otro ProcAD1}
              ACCEPT InicioLeerDatoRemoto DO
                   NumLectoresRemotos := 1;
                   {Requiere la transmisión del dato al servidor remoto}
              END InicioLeerDatoRemoto;
      OR
         WHEN (NumConsultasRemotas = 0) =>     {A lo sumo una consulta de otro ProcAD1}
              ACCEPT ConsultaRemota DO
                   NumConsultasRemotas := 1;
                   {Trasmite el dato solicitado al servidor remoto}
              END ConsultaRemota;
      OR
         ACCEPT FinLeerDatoRemoto DO
              NumLectoresRemotos := 0;
              {Aviso de recepción de datos al servidor remoto}
         END FinLeerDatoRemoto;
      OR
         ACCEPT FinConsultaRemota DO
              NumConsultasRemotas := 0;
         END FinConsultaRemota;
      END SELECT;
   END LOOP;
END ProcAD1;
```

Figura 8.7: Administrador Distribuido AD1 con Lectores Locales y Remotos.

En la Fig. 8.8 se muestra la solución para un administrador ProcAD2 que sólo puede aceptar una operación remota por vez. Un tipo de operación remota es el pedido de lectura de datos remotos por parte de un proceso lector local. El otro tipo de operación remota es la consulta de datos locales por parte de otro servidor).

```
TASK TYPE ProcAD2 IS                    {Permite crear un arreglo de TASKs con el mismo molde}
  ENTRY InicioLeerDatoLocal;            {Lector local – dato local}
  ENTRY FinLeerDatoLocal;
  ENTRY InicioLeerDatoRemoto;           {Lector local – dato remoto}
  ENTRY FinLeerDatoRemoto;
  ENTRY ConsultaRemota;                 {Requerimiento de otro ProcAD2 (lectura local)}
  ENTRY FinConsultaRemota;
END ProcAD2;

TASK BODY ProcAD2 IS
  NumLectoresLocales: INTEGER := 0;
  NumRemotos: INTEGER := 0;

BEGIN
  LOOP
    SELECT
        ACCEPT InicioLeerDatoLocal DO
            NumLectoresLocales := NumLectoresLocales+1;
            {El cliente local lee la BD local}
        END InicioLeerDatoLocal;
    OR
        ACCEPT FinLeerDatoLocal DO
            NumLectoresLocales := NumLectoresLocales-1 ;
        END FinLeerDatoLocal;
    OR
        WHEN (NumRemotos = 0) =>        {Solamente una operación remota}
            ACCEPT InicioLeerDatoRemoto DO
                NumRemotos := 1;
                {Requiere la transmisión del dato al servidor remoto}
            END InicioLeerDatoRemoto;
    OR
        WHEN (NumRemotos = 0) =>        {La misma variable de control del ACCEPT previo}
            ACCEPT ConsultaRemota DO
                NumRemotos := 1;
                {Trasmite el dato solicitado al servidor remoto}
            END ConsultaRemota;
    OR
        ACCEPT FinLeerDatoRemoto DO
            NumRemotos := 0;
            {Aviso de recepción de datos al servidor remoto}
        END FinLeerDatoRemoto;
    OR
        ACCEPT FinConsultaRemota DO
            NumRemotos := 0;
        END FinConsultaRemota;
    END SELECT;
  END LOOP;
END ProcAD2;
```

Figura 8.8: Administrador Distribuido AD2 con Lectores Locales Remotos.

El proceso administrador ProcAD1 y el proceso administrador ProcAD2 siguen distintas políticas de servicio de requerimientos. Estas distintas políticas producen distintos tiempos de finalización de los procesos clientes. En la Fig. 8.9 se muestra una secuencia de pedidos de servicio por clientes locales y remotos. Los procesos clientes locales se identifican como LL_i (lectura local) o LR_j (lectura remota) y los procesos clientes remotos se identifican

como CR_k. Los índices asociados a cada proceso cliente (i, j o k) solamente se utilizan para identificar el orden en que se llevan a cabo los requerimientos. El tiempo de llegada 0 identifica el tiempo del primer requerimiento de lectura de la BD, en el cual se asume que está libre, no hay ningún otro proceso que está operando sobre ella. El "Tiempo de Uso" de la Fig. 8.9 se refiere al lapso de tiempo en el cual el proceso cliente debe operar sobre la BD.

Tiempo de Llegada	Cliente	Tiempo de Uso
0	LL_1	3
1	LR_1	5
2	CR_1	7
3	LL_2	3
4	LR_2	6
5	LL_3	3
10	LL_4	3

Figura 8.9: Requerimientos para un Administrador Distribuido.

En la Fig. 8.10 se muestra la secuencia temporal de acceso a los datos locales que se deriva de la secuencia de requerimientos de la Fig. 8.9 y el administrador ProcAD1 de la Fig. 8.7. Las líneas de puntos indican la espera de los procesos clientes por el permiso de acceso a la BD que debe otorgar el proceso ProcAD1. Como ya se ha definido, todos los procesos clientes locales pueden leer simultáneamente la BD *local* y, por lo tanto, se pueden satisfacer todos estos requerimientos de forma inmediata. El proceso LR_2 es el único al que el proceso administrador ProcAD1 no otorga el permiso de entrada de forma inmediata. Este proceso LR_2 debe esperar mientras opera en la BD *local* el proceso LR_1. Con esto se respeta la restricción mencionada anteriormente: no se permite a dos procesos de consulta remota operar simultáneamente sobre un mismo administrador de datos (en sus datos locales).

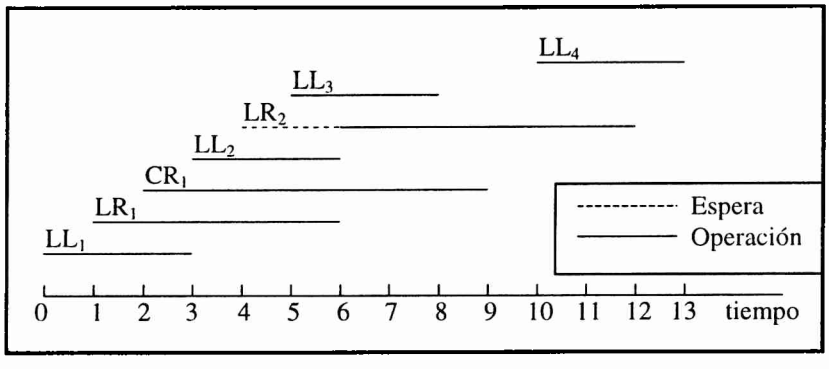


Figura 8.10: Acceso de los Clientes a los Datos Locales de ProcAD1.

En la Fig. 8.11 se muestra la secuencia temporal de acceso a los datos locales que se deriva de los requerimientos de la Fig. 8.9 y el administrador ProcAD2 de la Fig. 8.8. De forma similar a la Fig. 8.10, las líneas de puntos indican la espera de los procesos clientes por el permiso de acceso a los datos locales que debe otorgar el proceso ProcAD2. Se debe aclarar que cuando el proceso LR_1 finaliza su tarea (en la sexta unidad de tiempo), hay dos procesos esperando por el permiso de operación sobre el administrador ProcAD2: el proceso CR_1 y el proceso LR_2. El proceso administrador ProcAD2 no define ninguna prioridad entre estos

requerimientos y por lo tanto se otorga el acceso a los datos dependiendo del orden de evaluación de las cláusulas ACCEPT. En el caso de la Fig. 8.11 se le otorga permiso de operación al proceso LR_2.

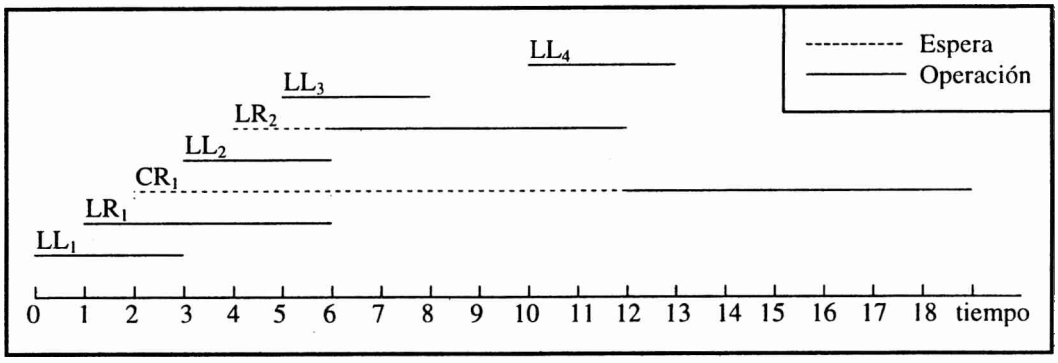


Figura 8.11: Acceso de los Clientes a los Datos locales de ProcAD2.

En la Fig. 8.12 se muestran los tiempos de finalización de cada proceso dependiendo del administrador que se utilice para gestionar los permisos de operación sobre la BD. En este caso, el tiempo en que finalizan todas las tareas depende mucho de la política que se define para otorgar el permiso de operación. La diferencia es muy clara con respecto a la cantidad de permisos *simultáneos* de operación y por lo tanto a la cantidad de tareas clientes que pueden operar a la vez sobre la BD. La diferencia entre ambas políticas la establecen los procesos que requieren el acceso de datos remotos (LR_j) junto con los procesos remotos que requieren el acceso a datos locales (CR_k).

Proceso Cliente	Tiempo de Finalización	
	ProcAD1	ProcAD2
LL_1	3	3
LL_2	6	6
LL_3	8	8
LL_4	13	13
LR_1	6	6
LR_2	12	12
CR_1	9	19

Figura 8.12: Tiempos de Respuesta de los Procesos Clientes.

Se debe notar que estos resultados son parciales, porque corresponden a un sólo servidor de la Base de Datos y se supone que tanto el canal de comunicaciones como el/los servidor/es remoto/s al/los que se consulta/n está/n libres. Se puede afirmar que estas suposiciones simplifican excesivamente la aplicación. Si se modeliza un problema real con un canal de comunicaciones compartido, como podría ser un bus por ejemplo, se debe tener otro proceso administrador de recurso compartido (el bus de comunicaciones). El acceso al bus depende de la demanda global de requerimientos de comunicaciones y por lo tanto tiende a deteriorar el rendimiento global del sistema. En contextos de aplicación reales, como mínimo se debe tener en cuenta una mayor cantidad de posibilidades (y complejidad asociada) en las secuencias de requerimientos que le llegan a un proceso administrador.

8.3 Procesos que son a la vez Clientes y Servidores

Los procesos servidores de la Base de Datos distribuida que se han presentado en el ejemplo anterior tienen la particularidad de ser a la vez procesos clientes y procesos servidores. Son procesos idénticos que atienden clientes locales o consultas remotas y *también* solicitan atención a otros servidores remotos.

En esta sección se analiza un problema donde se tienen múltiples funciones integradas en un sistema en el cual los procesos pueden ser clientes o servidores. El manejo del cambio en cajeros, por ejemplo, es un problema típico de los bancos. En este contexto, existen procesos Usuario que piden cambio en las cajas. Para simplificar se supone que los procesos Usuario piden cambio de un tipo de billetes contra otro tipo de billetes, por ejemplo 2 de 100$ en billetes de 10$. Los procesos Caja pueden dar el cambio o rechazar el pedido por no tener billetes disponibles. A su vez los procesos Caja pueden pedir cambio a un proceso CajaGeneral. Tanto el proceso Caja como el proceso CajaGeneral tienen que informar el monto total de dinero que contienen y la distribución del cambio en los tipos de billetes existentes como respuesta a requerimientos de un proceso MasterBanco.

Para ordenar las características de la aplicación conviene llevar a cabo un resumen del análisis del problema en términos de procesos clientes y procesos clientes/ servidores identificando los requerimientos:

- Los procesos Usuario son clientes de algún proceso Caja.
- Los procesos Caja son servidores de los procesos Usuario y del proceso MasterBanco y a su vez son clientes del proceso CajaGeneral.
- El proceso CajaGeneral es servidor de cada uno de los procesos Caja y del proceso MasterBanco.
- El proceso MasterBanco es cliente del proceso Caja y del proceso CajaGeneral.
- Los procesos Usuario podrían elegir la caja de interés o hacer un *Conditional Call* con un *timeout* a todas las cajas.
- Los procesos Caja pueden ser un arreglo de procesos del mismo tipo, que conocen su identificación. Deben procesar inteligentemente los pedidos y mantener una información global del estado de sus billetes de cada tamaño. Un adicional interesante es que al llegar a un determinado nivel de billetes de cada denominación deben pedir cambio.
- El proceso CajaGeneral maneja un servicio con modificación de sus datos internos (pedido de cambio de las cajas) y otro de consulta pura (desde el master general). Esto puede funcionar al modo de Lectores-Escritores con la particularidad de que hay sólo un proceso lector.
- El proceso MasterBanco no requiere una alta prioridad. Se lo puede esquematizar con una consulta permanente a cada caja. Es interesante es analizar *cómo* puede conocer si las cajas están activas o no; también es interesante tener en cuenta que puede hacer un recorrido por todas las cajas activas, utilizando *Conditional Call*s.

La Fig. 8.13 muestra de forma esquemática el código en lenguaje Ada del proceso correspondiente a las cajas. No se detallan los parámetros y cálculos para simplificar y evitar los detalles de implementación. Como en el caso de los procesos administradores, se definen los puntos de entrada del proceso (cada uno con un ENTRY) para todos los demás procesos de la aplicación. La Fig. 8.14 muestra de forma esquemática el proceso CajaGeneral, y la Fig. 8.15 el proceso MasterBanco.

```
TASK TYPE Caja IS
   ENTRY PedidoDeCambioUsuario;          {Requerimiento de Usuario}
   ENTRY FinCambioUsuario;
   ENTRY ReciboCambioCajaGeneral;        {Requerimiento de CajaGeneral}
   ENTRY FinCambioCajaGeneral;
   ENTRY PedidoConsultaMaster;           {Requerimiento de MasterBanco}
   ENTRY FinConsultaMaster;
END Caja;

BEGIN
  {Se obvian todos los demás procesos que puede realizar la Caja.
  Tampoco se detalla el pedido de cambio a la Caja General.
  Se supone que la atención de un pedido de cambio de Usuario es prioritaria respecto
  de la consulta del proceso MasterBanco}

     LOOP
        SELECT

            ACCEPT PedidoDeCambioUsuario DO;
               {Chequear si el cambio pedido se puede entregar}
               {Si está OK ==> orden de emisión del dinero al puesto Usuario
                               actualización de saldos por billetes en la Caja}
               {Si NO está OK ==> Mensaje informando al Usuario}
               ACCEPT FinCambioUsuario;
            END PedidoDeCambioUsuario;

        OR

            ACCEPT ReciboCambioCajaGeneral DO;
               {Actualizar el estado de los saldos por billete}
               {Mensaje a la Caja General de cambio recibido OK}
               ACCEPT FinCambioCajaGeneral;
            END ReciboCambioCajaGeneral;

        OR

           WHEN (PedidoDeCambioUsuario'COUNT = 0) =>

              ACCEPT PedidoConsultaMaster DO;
                 {Mensaje al proceso Master con el detalle de estado de  la Caja}
                 ACCEPT FinConsultaMaster;
              END PedidoConsultaMaster;

        END SELECT;
     END LOOP;
END Caja;
```

Figura 8.13: Esquema del Proceso Caja.

```
TASK TYPE CajaGeneral IS
   ENTRY PedidoDeCambioCaja;              {Requerimiento de Caja}
   ENTRY FinCambioCaja;
   ENTRY PedidoConsultaMaster;            {Requerimiento de MasterBanco}
   ENTRY FinConsultaMaster;
END CajaGeneral;

BEGIN
  {Se obvian todos los demás procesos que puede realizar la Caja General.
   No se detalla la lógica de envío de cambio a la Caja.
   Se supone que la atención de un pedido de cambio de Caja es prioritaria respecto de
   la consulta del proceso MasterBanco}

  LOOP

     SELECT

          ACCEPT PedidoDeCambioCaja DO;
              {Chequear si el cambio pedido se puede entregar}

              {Si está OK ==> orden de emisión del dinero al puesto Caja
                              actualización de saldos por billetes en la CajaGeneral}

              {Si NO está OK ==> Mensaje informando al proceso Caja.
                                 Mensaje informando al proceso Master}

              ACCEPT FinCambioCaja;

          END PedidoDeCambioCaja;

       OR

          WHEN (PedidoCambioCaja'COUNT = 0) =>

            ACCEPT PedidoConsultaMaster DO;

              {Mensaje al proceso Master con el detalle de estado de  la Caja General}

              ACCEPT FinConsultaMaster;

            END PedidoConsultaMaster;

     END SELECT;

  END LOOP;

END CajaGeneral;
```

Figura 8.14: Esquema del Proceso CajaGeneral.

```
TASK TYPE MasterBanco IS
   ENTRY RespuestaConsultaCaja;                    {Requerimiento de Caja}
   ENTRY FinConsultaCaja;
   ENTRY RespuestaConsultaCajaGeneral;             {Requerimiento de CajaGeneral}
   ENTRY FinConsultaCajaGeneral;
END MasterBanco;

BEGIN

   {Se obvian todos los demás procesos que puede realizar el proceso MasterBanco}

   {Se supone que pedida una consulta se queda esperando la respuesta}

   {Podrían hacerse los pedidos a las cajas en forma sincrónica en tiempos determinados}

   LOOP

      SELECT

            ACCEPT RespuestaConsultaCaja DO;
               {Actualizar las tablas de estado de cada Caja}

               {Mensaje de recepción de datos al proceso Caja}

               ACCEPT FinConsultaCaja;

            END RespuestaConsultaCaja;

       OR

            ACCEPT RespuestaConsultaCajaGeneral DO;
               {Actualizar las tablas de estado de la CajaGeneral}

               {Mensaje de recepción de datos al proceso CajaGeneral}

               ACCEPT FinConsultaCajaGeneral;

            END RespuestaConsultaCajaGeneral;


      END SELECT;

   END LOOP;

END MasterBanco;
```

Figura 8.15: Esquema del Proceso MasterBanco.

El proceso correspondiente al Usuario es simple, y de alguna manera es el que inicia la mayoría de las transacciones en cada uno de los procesos Caja. La Fig. 8.16 muestra de forma esquemática el código correspondiente a este proceso.

```
TASK TYPE Usuario IS
   ENTRY ReciboCambio;
   ENTRY FinCambio;
END Usuario;

TASK BODY Usuario IS

BEGIN
  {Se obvia el proceso de pedido de cambio.
   Tratándose de un arreglo de procesos Caja podría pedir y recibir indistintamente de
   cualquiera de ellos}

     LOOP
        ACCEPT ReciboCambio DO;
           {El usuario puede recibir el dinero o un mensaje de que la Caja no tiene
            el tipo de cambio pedido.
            El Usuario responde con un mensaje de dinero/información recibida OK y
            espera confirmación de la Caja}
            ACCEPT FinLeerDatoLocal;
        END ReciboCambio;
     END LOOP;

  {Si el usuario NO recibió el dinero puede reiterar el pedido}

END Usuario;
```

Figura 8.16: Esquema del Proceso Usuario.

8.3 Procesamiento (Paralelo) en Tiempo Real

Las clases de problemas que se han analizado en este capítulo muestran que *el grado de paralelismo alcanzable* puede ser dependiente del contexto y de la secuencia temporal de eventos. Las relaciones de dependencia y prioridad entre procesos determinadas por la política de administración elegida, condiciona el número de procesos concurrentes que pueden ejecutarse, con independencia de la cantidad de procesadores disponibles.

En esta sección se discute un aspecto característico de los sistemas de tiempo real duros [Lap93]: se debe tener en cuenta el requerimiento de que determinados procesos (o acciones dependientes de procesos) terminen en un tiempo máximo, o se ejecuten un tiempo determinado, o se retarden un tiempo especificado. En cualquiera de estos casos, se introduce una restricción adicional a la solución paralela a desarrollar: con los procesos propios de la aplicación *coexiste uno o varios procesos* ***reloj*** *con el que se debe sincronizar la aplicación.* En ocasiones, estos procesos *reloj* son prioritarios y pueden interrumpir, demorar o cancelar

un proceso activo. El impacto del agregado de restricciones de tiempo sobre el grado de paralelismo alcanzable resulta obvio: para cumplir la especificación de tiempo real se puede reorganizar dinámicamente la ejecución de procesos, sacrificando eficiencia.

Como ejemplo de procesos paralelos que además deben tener en cuenta restricciones de tiempo real se presenta el manejo de robots que cooperan en una tarea compleja. El trabajo de un conjunto de robots se presenta en forma abstracta y sumamente simplificada en ambiente industrial, tratando de analizar la clase de problemas que se ha mencionado:

- Se supone un conjunto de 3 clases de robots que participan de una línea de producción de un automóvil por ejemplo: el automóvil se traslada por una cinta transportadora a una velocidad fija que determina un tiempo máximo de operación de cada robot en su tarea específica.
- El primer conjunto de robots R1 trabaja en paralelo en la inserción de accesorios en el vehículo (tornillos, plásticos, protectores, etc.). El conjunto R1 tiene un tiempo máximo de trabajo y si alguno de ellos no hubiera terminado su tarea (por una falla local por ejemplo) deberá reportarlo al proceso organizador como un mensaje de error. Este mensaje de error puede significar un aviso a control de producción, reemplazar al robot $R1_i$ que falla, abortar ese vehículo o detener la línea de producción según el caso.
- Cuando el proceso organizador de los robots R1 lo indica, o cuando se cumple un tiempo conocido T_1, el segundo conjunto de robots R2 comienza a trabajar, por ejemplo en el pintado del vehículo, durante un tiempo fijo T_2.
- Al cumplirse T_2, el tercer conjunto de robots (R3) puede realizar una tarea post-pintura (inserción de espejos, secado rápido de la pintura, colocación de limpiaparabrisas, etc.), tarea que nuevamente debe tener un tiempo máximo (T_3) y determinadas condiciones de error.
- Independientemente, *en cualquier momento* puede haber una tarea prioritaria (por ejemplo una alarma de incendio) que obligue a **todos** los robots a pasar a otra función. Por ejemplo detener su tarea normal y operar lanzando un líquido anti-fuego con un ángulo determinado según la posición (calculada) del vehículo que están atendiendo. En este caso hay un evento claramente prioritario que debe tener la posibilidad de interrumpir a todos los procesos que se están ejecutando en paralelo.

El problema exige dos aspectos claves de tiempo real: un reloj centralizado y sincronizado. Si bien podrían distribuirse relojes locales en cada proceso-procesador robot, esto obliga a que todos tengan exactamente la misma hora (salvo algún error permitido). Si el reloj está físicamente centralizado, significa que cada T segundos se puede trasmitir la hora real *a todos los procesadores* con prioridad y mínimo error. La segunda exigencia del problema es que cada proceso-procesador debe saber cuándo comenzar a operar sobre un vehículo. Como es prácticamente imposible manejarlo en forma sincrónica sobre la base de la velocidad de desplazamiento de la cinta transportadora, es necesario tener un sistema sensor propio de cada robot o un sensor general múltiple que sea prioritario para avisar a cada robot el tiempo de inicio de su tarea.

Toda la especificación del problema anterior sirve para explicar algunos aspectos de la solución que se esquematiza a nivel especificación en Ada:

1. A medida que aumenta la cantidad de robots, será conveniente que los sistemas de sensado de inicio y de reloj local coexistan físicamente con el procesador robot.
2. Se necesitan al menos dos líneas de comunicación con todos los robots, que tengan alta prioridad y constituyan canales globales para todos los procesos robot: la línea con el

mensaje de alarma de incendio y la línea por la cual se trasmiten señales de sincronismo del reloj global para ajustar todos los relojes locales a la hora general.

3. Para la toma de decisiones del modo de funcionamiento normal se necesita un proceso organizador, que en este caso *no* trabaja sobre la administración de recursos comunes, sino sobre la información del avance del trabajo de cada robot para controlar acciones de control global.

Las figuras que siguen describen brevemente en Ada la especificación de las diferentes tareas que se pueden ejecutar en paralelo. Se debe hacer notar nuevamente que para estudiar el grado de paralelismo real de la ejecución se debe realizar una simulación (o al menos un análisis) en función de las *trazas* o secuencias temporales de eventos del problema. La Fig. 8.17 muestra la especificación de los procesos que controlan a los robots.

```
TASK TYPE Robot IS
   ENTRY FinTiempo;              {El proceso reloj local indica fin de tiempo}
   ENTRY SeñalVision;            {Se recibe la indicación de vehículo presente}
   ENTRY AlarmaGeneral;          {Se recibe la señal global de alarma}
   ENTRY FinAlarmaGeneral;
   ENTRY AjusteHoraGeneral;      {El proceso reloj local recibe hora general y
                                  envía una corrección}
END Robot;
```

Figura 8.17: Especificación de los Procesos de Control de Robots.

La Fig. 8.18 muestra la especificación del proceso correspondiente al reloj local del robot. El único ENTRY permite ajustar la hora local de acuerdo al reloj global.

```
TASK TYPE RelojRobot IS
    ENTRY HoraGeneral;      {Se recibe la hora global}
END RelojRobot;
```

Figura 8.18: Especificación de los Procesos del Relojes Locales al Robot.

La Fig. 8.19 muestra la especificación del proceso correspondiente al reloj global de la aplicación.

```
TASK TYPE RelojGeneral IS
  ENTRY TicRelojExterno;              {Se recibe el tic o la hora exacta de un reloj externo}
  ENTRY ConfirmacionHoraRecibida;     {Cada proceso reloj local de robot confirma el ajuste
                                       de la hora local a la hora global}
END RelojGeneral;
```

Figura 8.19: Especificación de los Procesos de Reloj Global de la Aplicación.

La Fig. 8.20 muestra la especificación del proceso correspondiente a la alarma de la aplicación. Además de las señales provenientes de los sensores externos de alarma (ENTRY

SeñalAlarma y ENTRY FinSeñalAlarma) recibe la confirmación de recepción de la señal de fin de la alarma por parte de los Robots.

```
TASK TYPE AlarmaGeneral IS
   ENTRY SeñalAlarma;        {Señal externa de los sensores de alarma}
   ENTRY FinSeñalAlarma;
   ENTRY AckFinAlarma;       {Cada proceso robot local confirma la recepción del fin
                              de la alarma y puede reiniciar el modo normal}
END AlarmaGeneral;
```

Figura 8.20: Especificación de los Procesos de Reloj Global de la Aplicación.

La Fig. 8.21 muestra la especificación de los procesos de control de Robots en lo referente a la presencia de un vehículo sobre el cual trabajar.

```
TASK TYPE VisionRobot IS
   ENTRY SeñalVehículo;      {Señal de los sensores que indican presencia de vehículo}
   ENTRY SeñalRecibida;      {El robot confirma la recepción de la señal}
   ENTRY AlarmaGeneral;
   ENTRY FinAlarmaGeneral;
END VisionRobot;
```

Figura 8.21: Especificación de los Procesos de Sensado Visual.

Finalmente, la Fig. 8.22 muestra la especificación del proceso de organización de todos los conjuntos de Robots (la línea de construcción/armado de vehículos).

```
TASK TYPE SchedulerRobot IS
   ENTRY InformeDeRobot;     {Informe de tarea y condiciones de error de Robot}
   ENTRY HoraGeneral;        {El scheduler actualiza su reloj interno}
   ENTRY AlarmaGeneral;      {El scheduler recibe la señal de alarma}
   ENTRY FinAlarmaGeneral;
END SchedulerRobot;
```

Figura 8.22: Especificación del Proceso Organizador de los Robots.

Por otra parte es de hacer notar que si bien por cada vehículo los conjuntos de robots R1, R2 y R3 *no pueden trabajar en paralelo*, hay robots de las tres clases *trabajando en paralelo sobre diferentes vehículos*. Por lo tanto, se tiene que en el análisis global del sistema los conjuntos de robots trabajan de modo pipeline.

9. Migración de Procesos y Canales en DMPCs

Una de las grandes áreas dentro de las arquitecturas de procesamiento paralelo la constituyen las computadoras paralelas de memoria distribuida (DMPC: **D**istributed **M**emory **P**arallel **C**omputers) [Sim97]. También denominadas en la bibliografía como multicomputadoras de memoria distribuida [Hwa93], computadoras con arquitectura de memoria distribuida [Lew92], o computadoras con arquitectura MIMD de memoria distribuída. Por razones de simplicidad, en éste y en los capítulos siguientes, se hará referencia a este tipo de computadoras como DMPC.

Un sistema DMPC se define como una arquitectura MIMD con memoria distribuida. Básicamente, los sistemas DMPC está formado por múltiples nodos, cada uno de ellos constituido por un procesador y memoria local, que se interconectan por medio de una red de comunicaciones. La red de comunicaciones permite el envío y recepción de mensajes entre procesadores, y usualmente pertenece a la clase de redes estáticas de interconexión. Cada nodo tiene, adicionalmente, la posibilidad de realizar entrada/salida. Cada nodo tiene, de forma adicional, la posibilidad de realizar también entrada/salida.

La característica más atractiva de los DMPCs cuando se los compara con otras alternativas de procesamiento paralelo es la escalabilidad. La escalabilidad consiste en la replicación de la arquitectura para obtener mayor rendimiento de la computadora paralela. A su vez, la replicación de la arquitectura es posible gracias al relativamente bajo nivel de dependencia, recursos compartidos, o *acoplamiento* entre los nodos.

A nivel de procesamiento y programación de los DMPC, usualmente se toma como referencia el modelo de CSP: **C**ommunicating **S**equential **P**rocesses [Hoa86]. Una aplicación está constituida por procesos que se ejecutan en paralelo y que se comunican explícitamente por envío y recepción de mensajes [Bal89] [Cha91] [Pan90]. La unidad de paralelismo es el proceso, que ejecuta código secuencial y que se comunica por medio de la utilización de canales. Los canales son los que llevan a cabo la comunicación y, según el modelo de comunicación por mensajes que se elija (de tipo bloqueante o no bloqueante), pueden también proveer sincronización entre procesos.

Para aprovechar al máximo el rendimiento que pueden proveer los DMPC se deben utilizar los recursos de forma *balanceada* [Qui87]. Los recursos básicos disponibles para la aplicación son principalmente dos: el conjunto de los procesadores, dedicados al cómputo; y la red de comunicaciones, dedicada a la transmisión de mensajes. Por lo tanto se deben balancear los requerimientos de cómputo y de comunicaciones de los procesos que se ejecutan en cada procesador. La idea de utilización balanceada de los procesadores implica que todos los procesadores estén ocupados en la ejecución de procesos. En la ejecución de aplicaciones, no se deberían encontrar procesadores sobrecargados, siempre ocupados, mientras que otros se encuentran libres, sin procesos para ejecutar. De la misma manera, la utilización balanceada de la red de comunicaciones implica que los requerimientos de comunicación entre procesadores sea similar. No se deberían producir congestiones por los mensajes que deben llegar a, o salir de, uno o algunos procesadores (*hot-spots*).

Hay dos líneas principales de investigación en lo que se refiere a la búsqueda de balancear la carga de un DMPC:

1. Estática: se intenta descubrir el comportamiento de la aplicación paralela en términos de los requerimientos de cómputo y de comunicación de cada proceso que la compone. Una vez estimado el comportamiento de cada proceso, se realiza la asignación de procesos a procesadores (mapping) y de esta manera se ejecuta la aplicación [Cor92]. No se realizan cambios de la asignación proceso-procesador durante la ejecución de la aplicación. La mayoría de las estrategias de mapping usualmente estiman solamente los requerimientos de cómputo de cada proceso.
2. Dinámica: durante la ejecución de la aplicación se monitoriza el estado de ocupación de los procesadores y/o de la red de comunicaciones y se decide si es necesario cambiar la ubicación de los procesos en los procesadores. Se espera obtener una mejor utilización de los recursos, y por lo tanto mayor rendimiento, por la introducción en los DMPCs de técnicas de Migración Dinámica de Procesos. El soporte para la migración de procesos permite cambiar la asignación de procesos a procesadores de manera dinámica, en tiempo de ejecución [Smi88].

El principal inconveniente de las estrategias de mapping consiste en que es muy difícil conocer a priori el comportamiento dinámico (de ejecución) de la aplicación del usuario. Más aún, el comportamiento puede ser dependiente de los datos y, por lo tanto, muy distinto entre distintas ejecuciones del mismo programa paralelo.

Los inconvenientes de la migración dinámica de procesos inicialmente hicieron descartar esta posibilidad [Tan92], aunque actualmente se la está retomando. Se ha llegado a la conclusión de que posiblemente los problemas que se generan por la migración dinámica de procesos hayan sido sobredimensionados [Sta96]. De la misma forma en que los mecanismos de traslación de direcciones y el movimiento automático de datos a través de la jerarquía de memoria (memoria virtual y memoria cache) se han impuesto como características fundamentales de cualquier computadora, es posible que también será necesario incluir un mecanismo de migración de procesos, para mejorar de esta forma el rendimiento de las computadoras paralelas.

La política de migración así como el mecanismo de migración que se incorpore a un sistema paralelo deben satisfacer ciertos requisitos para considerarse aceptables. Algunos de los requisitos más importantes son [Dou91] [Corr92]

- Transparencia: el comportamiento de un proceso no debe modificarse por la posibilidad de migraciones en el sistema. Tanto la tarea de cada proceso como la visión que tiene el proceso del resto del sistema no deben verse afectados.
- Rendimiento: la presencia del mecanismo de migración no debe impedir el crecimiento de la velocidad de ejecución de las aplicaciones. El sistema debería ser escalable de la misma forma en que lo era sin la posibilidad de realizar migración de procesos.
- Selección automática: tanto la selección del proceso que se migre como del procesador destino del proceso deben realizarse sin la intervención del usuario.
- Cantidad de migraciones: debe ser posible migrar un proceso tantas veces como sean necesarias.
- Independencia de la configuración del sistema: no se debería requerir la sintonización de parámetros para adecuarse a la configuración del sistema.

Como suele suceder, algunos de los requisitos son difíciles de satisfacer, y se pueden encontrar requisitos contrapuestos. Se debería encontrar la solución que satisfaga de la mejor manera posible a todos ellos.

El balance de carga de la computadora paralela no es la única motivación para considerar que es necesario un mecanismo de migración de procesos, aunque en algunos casos puede ser considerada suficiente. También es deseable contar con un mecanismo de migración de procesos por razones de [Smi88]:

- Tolerancia a fallas y confiabilidad: cuando es posible detectar fallas con cierta anticipación en un procesador de un sistema paralelo, se pueden migrar los procesos que se verían afectados.
- Acceso a datos o recursos no disponibles localmente: en el caso de los datos, es posible que sea conveniente migrar un proceso a otro procesador y no transmitir una gran cantidad de datos por la red de comunicaciones. Puede suceder lo mismo con el acceso a recursos remotos o, también, que no sea posible acceder a algunos recursos de forma remota, como suele ocurrir con dispositivos físicos de propósito especial.

Las tareas a llevar a cabo para poder contar con un mecanismo de migración dinámica de procesos en un DMPC se pueden resumir en:

- Evaluar la carga de los procesadores y de la red. Asimismo, se debe determinar cómo utilizar esta información para decidir qué proceso debe migrar, a dónde y cuándo se va a migrar (Política de Migración o de Balance Dinámico de Carga).
- La migración física propiamente dicha, es decir cómo migrar procesos. Se incluye aquí el desalojo de un proceso en el procesador desde el que se migra, y su reubicación en el procesador destino (Mecanismo de Migración de Procesos).
- Gestionar las comunicaciones que involucran al proceso que migra, el cual debe continuar recibiendo mensajes independientemente de su ubicación en la red (Gestión del Arribo de los Mensajes) [Zhu90].

Si bien este capítulo y los que siguen tienden a centrarse en el tercer punto, no se puede independizar totalmente de los dos anteriores. Si se quiere experimentar en entornos paralelos reales para obtener datos que guíen las decisiones, será necesario implementar una política de migración, un mecanismo de migración física de procesos y una gestión del arribo de mensajes. De la misma forma, se debe contar con una forma de definir y ejecutar aplicaciones de usuario para poder ponderar cuál es el resultado de las decisiones que se tomen.

El Problema de Arribo de Mensajes (PAM) aparece como consecuencia de la migración dinámica de procesos. Cuando los procesos pueden cambiar de ubicación durante el tiempo de ejecución, ya no es posible asumir que recibirán todos los mensajes en el mismo procesador. Se deben mantener las conexiones entre procesos de usuario aunque éstos migren dinámicamente y por lo tanto es necesario gestionar los mensajes entre procesos de la aplicación de usuario. En la sección que sigue se describe de forma más detallada tanto el Problema de Arribo de Mensajes como las políticas propuestas para realizar la gestión de los mensajes: *Follow Me*, *Global Server*, *Home Processor*, *Mailbox*, *Message Rejection* y *Migration Protocol*.

Para poder evaluar y comparar cada política así como para proponer otras, es necesario contar con un entorno de experimentación que ejecute aplicaciones paralelas [Tin97]. El diseño de este entorno paralelo que implementa cada política y ejecuta aplicaciones de usuario sintéticas se define a continuación de las políticas con las cuales se gestionan los mensajes. Por su relación con el entorno de experimentación, también se definen en esta sección el modelo considerado de las aplicaciones de usuario y de la política de migración que se implementa.

Como último punto de este capítulo se presentan algunos índices de evaluación a tener en cuenta con respecto a las políticas de gestión de mensajes que se presentan. Se intenta que estos índices no solamente muestren la interferencia de las políticas con las aplicaciones de usuario, sino también permitan comparar las distintas políticas.

9.1 El Problema de Arribo de Mensajes

Mantener consistente el procesador y proceso de usuario de arribo de los mensajes es uno de los problemas a solucionar cuando se decide incorporar a un sistema paralelo la posibilidad de migración dinámica de procesos. Dados los procesos de una aplicación de usuario pu_{i1}, ..., pu_{ik} que se comunican enviando mensajes al proceso pu_i asignado a un procesador P*m*, Solucionar el Problema de Arribo de los Mensajes en un DMPC consiste en que los procesos pu_{i1}, ..., pu_{ik} puedan seguir enviando mensajes al proceso pu_i aún cuando éste sea migrado desde el procesador P*m* a otro procesador P*n*. Se deben mantener las conexiones entre los procesos (canales) aunque éstos migren de forma dinámica.

De forma esquemática, se puede ver la situación en la que los procesos de usuario migran dinámicamente como lo muestra la Fig. 9.1. Los procesos de usuario pu_{i1}, ..., pu_{ik} pueden estar asignados a cualquier procesador de la red de procesadores, inclusive en los procesadores P*m* y P*n*. De la misma manera, el proceso pu_i puede estar asignado en cualquier procesador y puede migrar a cualquier otro. La salida y llegada de los mensajes no debería verse afectada por la migración de uno o más procesos, a lo sumo podría variar (y de hecho lo hará) el tiempo requerido para la transmisión de los datos. Desde este punto de vista, el Problema de Arribo de Mensajes podría denominarse el Problema de Migración de Canales. De allí el título de este capítulo: "Migración de Procesos y *Canales* en DMPCs".

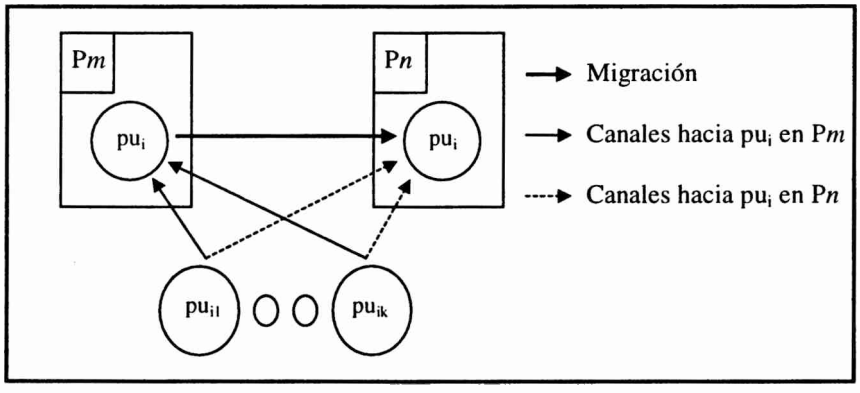


Figura 9.1: Problema de Arribo de Mensajes.

Se deben tener en cuenta varios aspectos asociados a este problema. En principio, los procesos de la aplicación de usuario no deberían conocer la posibilidad de migración dinámica, que debe ser transparente para ellos. Los procesos de la aplicación de usuario no se modifican ni incorporan comportamiento especial para facilitar o hacer parte de la migración dinámica ni conservar consistente el arribo de los mensajes.

El entorno paralelo de ejecución de programas debe contar con mecanismos que provean la solución a cada problema que se encuentre en la migración dinámica de los procesos de usuario. En el caso específico del arribo de mensajes, esto a su vez implica

manejar los mensajes entre procesos de usuario para evitar que al producirse la migración de uno de ellos los mensajes se pierdan o lleguen a un procesador en el que no se puedan manejar. La comunicación entre procesos de usuario no se realizará en forma directa, sino como lo muestra esquemáticamente la Fig. 9.2 para tres procesos de usuario (pu_i, pu_j, pu_k) asignados a dos procesadores (P*m* y P*n*).

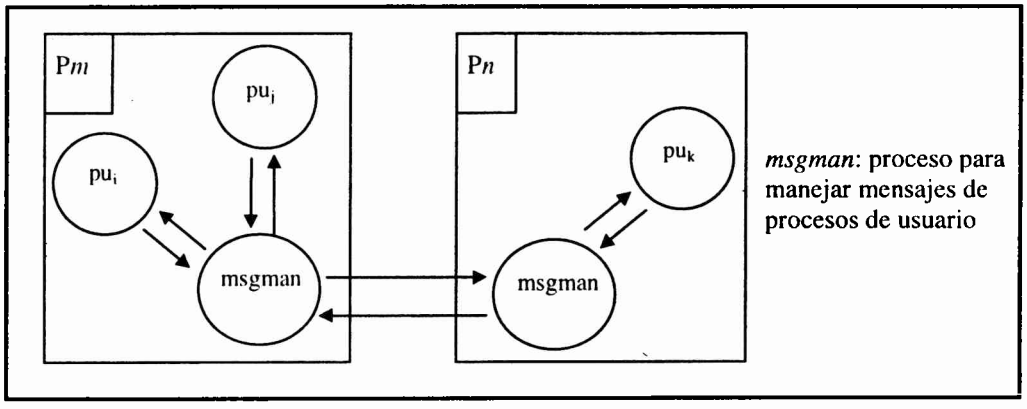


Figura 9.2: Manejo de los Mensajes entre Procesos de Usuario.

Los procesos dedicados a manejar mensajes (llamados *msgman* en la Fig. 9.2), implican no solamente un paso intermedio que se agrega en la comunicación entre los procesos de usuario. También utilizan memoria y tiempo de procesamiento, que originalmente se podían dedicar a las aplicaciones de usuario. Aún así, esta sobrecarga (overhead), para lograr la transparencia en el manejo de los mensajes no significa necesariamente una gran pérdida de rendimiento. Las comunicaciones que se agregan son locales, por lo tanto no hay más comunicaciones entre procesadores (que son las que utilizan la red de comunicaciones), de las que la misma aplicación de usuario define, y el tiempo de CPU necesario para la tarea de cada proceso *msgman* no es más que el que se necesita para identificar la ubicación de un proceso de usuario. Lo que realmente queda por decidir es si este overhead que se agrega es compensado por el mayor rendimiento al utilizar la computadora paralela de forma balanceada.

El hecho de agregar procesos que manejan los mensajes entre procesos de usuario no soluciona por sí solo el Problema de Arribo de Mensajes. No siempre se puede asumir que se conoce en cada procesador la ubicación correcta de todos los procesos de usuario cuando éstos migran dinámicamente. Por lo tanto, puede suceder que un mensaje se envíe a un procesador en el cual el proceso de usuario destino del mensaje ya no esté asignado. Una alternativa que se presenta para que estos mensajes no se pierdan es la retransmisión, es decir que los mensajes se reenvíen a otro procesador en el cual se conozca la ubicación o esté asignado el proceso de usuario destino del mensaje. En este punto sí aparecen mensajes entre procesadores que no existen en la aplicación de usuario. Si bien los procesos de usuario no manejan ni conocen la existencia de estas retransmisiones, es probable que tengan un retraso en su ejecución porque:

1. encuentran la red de comunicaciones más congestionada, o
2. el acceso al proceso *msgman* que procesa los requerimientos de mensajes es más lenta, o
3. el acceso a la ejecución misma es aplazada porque el proceso *msgman* utiliza más tiempo el procesador ya que tiene más tareas para realizar.

La idea de utilizar solamente información local en cada procesador es la más atractiva para la independencia de procesamiento, escalabilidad del algoritmo que se utilice, y para evitar mensajes entre procesadores de un DMPC. Pero también tiene el riesgo de que la información local se desactualice a medida que transcurre el tiempo. En el caso particular del Problema de Arribo de Mensajes, la información crítica es la que proporciona la ubicación de los procesos de usuario, que puede variar en el tiempo por causa de las migraciones dinámicas. Como ya se ha detallado, enviar un mensaje a un procesador incorrecto implicará generalmente una retransmisión del mensaje.

La alternativa clásica a manejar solamente información local consiste en transmitir a cada procesador actualizaciones de esta información. Estos mensajes suelen denominarse *mensajes de control* porque no contienen datos de mensajes entre procesos de usuario sino información para el funcionamiento de los procesos que manejan los mensajes de procesos de usuario. Como en el caso de las retransmisiones, los mensajes de control son mensajes que los procesos de usuario no manejan ni conocen pero con los cuales tendrán que competir por los recursos disponibles en la red de procesadores y de comunicación.

En el contexto del Problema de Arribo de Mensajes, usualmente la cantidad de mensajes de control es contrapuesta a la cantidad de retransmisiones de mensajes. A mayor cantidad de mensajes de control suelen ser necesarias menos retransmisiones y viceversa. Los dos extremos a considerar en un DMPC para resolver el Problema de Arribo de Mensajes son, por un lado, utilizar mensajes de control que de alguna manera *globalizan* la información de asignación de procesos de usuario a procesadores; y por el otro, utilizar solamente retransmisiones de mensajes. Es común encontrar estos dos extremos en los sistemas distribuidos [Tan92], y también suele buscarse una alternativa intermedia para que en cada procesador no tenga que conocerse el estado de todo el sistema ni utilizarse solamente información local que puede ser muy parcial.

Se presentan seis políticas propuestas [Hey95] [Tin97] para resolver el Problema de Arribo de Mensajes, donde se definen diferentes alternativas en cuanto a la retransmisión de mensajes, mensajes de control, y utilización de información local disponible en cada procesador. En las secciones que siguen se describe cada una de las políticas propuestas.

9.2 Política Follow Me

En esta política solamente se utiliza información conocida y generada localmente en lo que respecta a la asignación de procesos de usuario a procesadores. Cada vez que se produce una migración del proceso de usuario pu_i desde el procesador P*m* hacia el procesador P*n*, la nueva ubicación se conoce y se registra solamente en estos dos procesadores (P*m* y P*n*).

A medida que migra, cada proceso construye un camino que deben seguir los mensajes destinados a él, como se puede ver en la Fig. 9.3. Inicialmente, el proceso de usuario pu_i ha sido migrado desde el procesador P*m* al procesador P*n*, y luego al procesador P*q*. Cuando llega un mensaje al procesador P*m* para el proceso de usuario pu_i que ha migrado, el mensaje es retransmitido al procesador P*n* hacia el cual pu_i migró. Las retransmisiones se suceden hasta que el mensaje llega al procesador en el cual está ubicado pu_i. De esta manera, el mensaje es retransmitido por los mismos procesadores en los que ha estado asignado el

proceso pu_i. La excepción se produce cuando pu_i migra a un procesador en el cual ha estado previamente asignado.

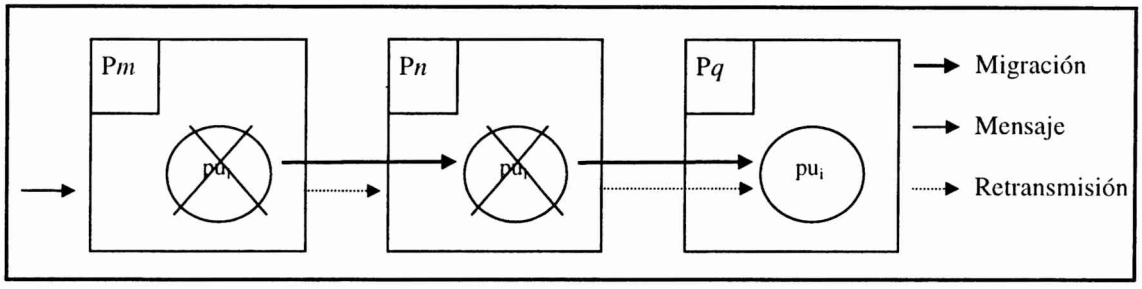


Figura 9.3: Retransmisiones con la Política Follow Me.

En esta política propuesta para el Problema de Arribo de Mensajes no se generan ni manejan mensajes de control, todo se resuelve con las retransmisiones de los mensajes entre procesos de usuario.

Cuando se aumenta la cantidad de procesadores, la cantidad de posibles destinos de un proceso también es mayor. Por lo tanto, la probabilidad de que un proceso migre a un procesador en el cual ha sido previamente asignado es cada vez menor. Se debe recordar en este punto que las retransmisiones de los mensajes, además de utilizar los recursos tal como lo ocupan los mensajes de control, tienen información que generalmente ocupa mayor espacio que los mensajes de control. En este sentido, una retransmisión de cada mensaje de usuario implica el doble de utilización de la red de comunicaciones. Un mensaje de control por cada mensaje de usuario implica en la mayoría de los casos un porcentaje de utilización agregado a la red de comunicaciones mucho menor.

9.3 Global Server

En esta política, la información sobre la ubicación de procesos de usuario que no es conocida localmente se debe consultar a un proceso que se dedica a concentrar esta información. Este proceso es el que se denomina comúnmente Servidor Central o Servidor Global.

Para cada mensaje que se genera en un procesador P*m*, si el proceso de usuario destino pu_i no está ubicado localmente se debe realizar lo siguiente

1. Consultar al proceso Servidor Global sobre la ubicación de pu_i (a qué procesador está asignado pu_i)
2. Recibir la respuesta del proceso Servidor Global, es decir el procesador P*n* que es la última ubicación conocida de pu_i
3. Enviar el mensaje al procesador P*n*

los dos primeros pasos se resuelven con mensajes de control hacia y desde el proceso Servidor Global.

En la Fig. 9.4 se pueden identificar los tres pasos de la enumeración anterior (1, 2 y 3). Para que esta política funcione correctamente el proceso Servidor Global debe conocer la ubicación de todos los procesos de usuario la mayor marte del tiempo de ejecución y por lo tanto, cada vez que se produce una migración se genera un mensaje de control hacia el proceso Servidor Global para actualizar su información.

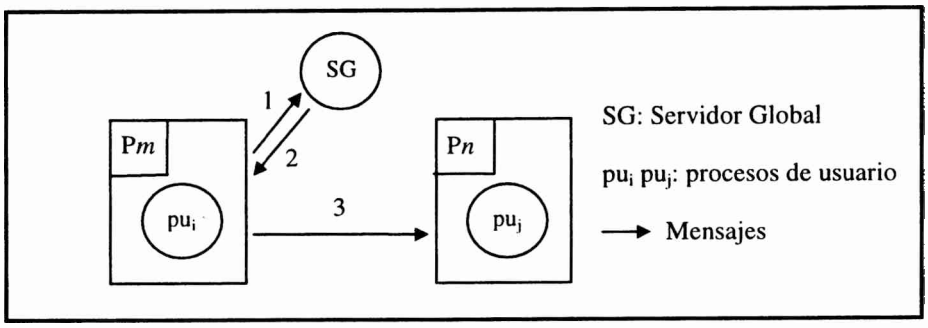


Figura 9.4: Envío de Mensajes con Global Server.

La respuesta que proporciona el proceso Servidor Global no siempre es correcta y puede darse el caso en el que un mensaje es enviado desde un procesador P*m* a un procesador P*n* incorrecto. Esta situación se produce cuando el mensaje con la nueva ubicación de un proceso de usuario pu_i llega al proceso Servidor Global después de haber respondido a una consulta sobre la ubicación de pu_i. Desde el procesador P*n* se genera un mensaje de control, de no reconocimiento (NACK: *not acknowledge*) del mensaje de usuario hacia el procesador P*m*, y todo el procedimiento para enviar el mensaje se repite. En este caso también se producirá una retransmisión del mensaje de usuario.

Según la descripción que se ha realizado, esta política genera cuatro tipos de mensajes de control

1. Petición de la ubicación de un proceso de usuario pu_i al proceso Servidor Global.
2. Respuesta del Servidor Global a la petición de ubicación (procesador) de un proceso de usuario pu_i.
3. Mensaje de rechazo de un mensaje de usuario, cuando llega a un procesador en el cual no está asignado el proceso de usuario destino del mensaje.
4. Mensaje de actualización de la ubicación de un proceso de usuario pu_i que se envía al proceso Servidor Global.

También esta política puede generar la retransmisión de un mensaje, que se produce en un procesador después de haber recibido un mensaje de control rechazando un mensaje de usuario.

Es de destacar que el Servidor Global tiene información actualizada durante gran parte del tiempo de ejecución de la aplicación. Por otra parte, se debe también destacar que este algoritmo tiene un punto de falla: si por alguna razón el proceso Servidor Global no se puede ejecutar, todo el sistema deja de funcionar. También se debe tener en cuenta que el proceso Servidor Global es en sí mismo un cuello de botella para toda la aplicación, principalmente en lo que se refiere a comunicaciones.

9.4 Home Processor

En esta política es necesario definir un Home Processor para cada proceso de usuario. Usualmente el Home Processor del proceso de usuario pu_i es el procesador P*m* al cual es asignado pu_i en la carga de la aplicación en la máquina paralela (mapping). Cuando se genera un mensaje destinado a pu_i (que no está asignado al procesador local), el mensaje es enviado al procesador P*m* que es el Home Processor de pu_i. Cuando el mensaje llega a P*m*, y pu_i ha sido migrado, el mensaje se retransmite desde P*m* a la nueva ubicación de pu_i.

Una de las condiciones necesarias para llevar a cabo esta política es que el Home Processor (P*m*) de un proceso de usuario pu_i debe conocer siempre la ubicación de pu_i, por lo tanto, cada vez que pu_i es migrado se genera un mensaje de control hacia el procesador P*m* con la nueva ubicación (procesador P*n*) de pu_i. La Fig. 9.5 muestra el caso en el cual pu_i ha migrado dos veces y se le envía un mensaje. El procesador Pm es el Home Processor del proceso de usuario pui. Inicialmente, el proceso de usuario migra al procesador Pn y luego al procesador Pq. Con respecto a la ubicación de los procesos de usuario, en cada procesador se conoce si un proceso de usuario está ejecutándose localmente o no, y también conoce cuál es el Home Processor de todos los procesos de usuario.

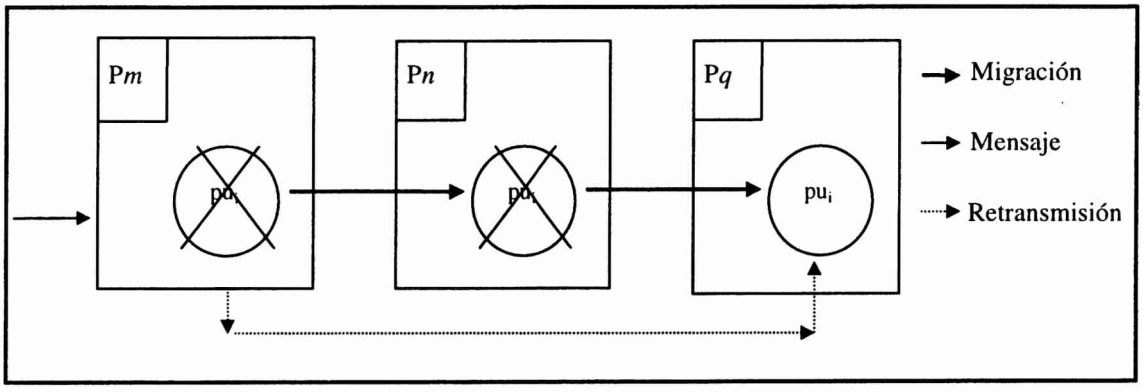


Figura 9.5: Retransmisión desde el Home Processor.

La retransmisión de mensaje desde el Home Processor P*m* al proceso de usuario pu_i no siempre van dirigidas al procesador correcto. Un mensaje de control con una nueva ubicación para pu_i puede llegar al procesador P*m* inmediatamente después de que se haya retransmitido un mensaje hacia el procesador desde el cual pu_i migró. En este caso, se genera un mensaje de control de no reconocimiento (NACK: *not acknowledge*) del mensaje de usuario hacia el procesador P*m*. Una vez que P*m* recibe la nueva ubicación de pu_i se producen retransmisiones adicionales de mensajes hacia el procesador a donde pu_i fue migrado.

En esta política hay dos tipos de mensajes de control que circulan entre los procesadores:

1. Actualización de la ubicación de un proceso de usuario pu_i, que se envía al Home Processor de pu_i, cuando éste es migrado.
2. Mensaje de rechazo de un mensaje de usuario, cuando llega a un procesador en el cual no está asignado el proceso de usuario destino del mensaje.

Las retransmisiones se producen en un procesador cuando se recibe un mensaje de rechazo de un proceso de usuario. Las retransmisiones hacia un proceso de usuario siempre se realizan desde el Home Processor del proceso de usuario.

9.5 Mailbox

En esta política se decide que toda la comunicación entre procesos de usuario se realice mediante la utilización de buzones. Cada proceso de usuario pu_i tiene asociado un buzón en donde se almacenan los mensajes que van destinados a él. Asimismo se cambia la semántica de la primitiva de comunicación *receive* para la recepción de mensajes. Usualmente, el buzón se mantiene en el procesador al cual pu_i fue asignado en la carga de la aplicación (mapping).

Cuando un mensaje es enviado al proceso de usuario pu_i, se envía y almacena en su buzón. Cuando pu_i ejecuta un *receive*, se produce una petición de un mensaje almacenado en el buzón de pu_i. No se necesita conocer la ubicación de pu_i en el procesador en el que se mantiene su buzón, porque cada petición de mensaje de pu_i proporciona la ubicación (el procesador) a la cual el mensaje debe ser enviado.

Cuando un proceso de usuario está ubicado en un procesador diferente al que mantiene su buzón, cada recepción implicará un mensaje de control y una retransmisión de mensaje. El mensaje de control contiene la petición de un mensaje del buzón y la retransmisión del mensaje se produce hacia el procesador desde el que se produjo la petición. La Fig. 9.6 muestra el envío y recepción de un mensaje de un proceso de usuario que ha migrado.

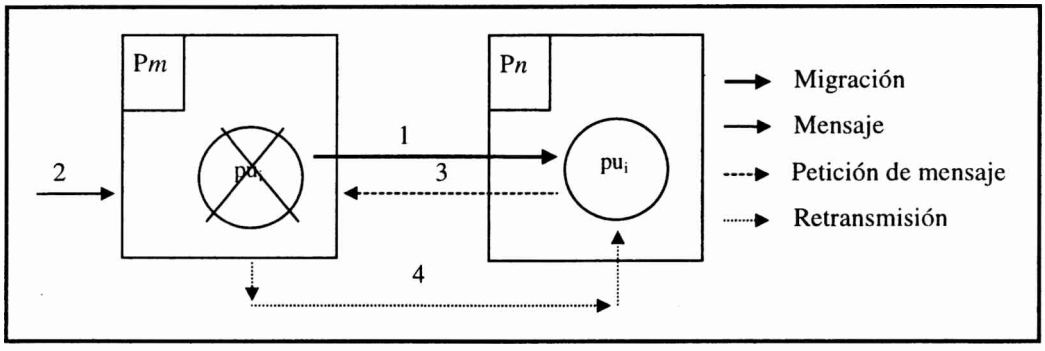


Figura 9.6: Envío y recepción de Mensajes Utilizando Mailboxes.

El proceso de usuario pu_i ha migrado del procesador P*m* al P*n* (1), se le envía un mensaje al procesador P*m* en el que está su buzón (2), pu_i ejecuta un *receive* (3) y se le envía un mensaje almacenado en su buzón (4).

Con esta política se agrega una restricción con respecto a las migraciones de procesos de usuario. Un proceso de usuario no puede ser migrado mientras tenga una o más recepciones de mensajes pendientes.

Solamente hay un tipo de mensaje de control que se maneja en esta política: el que se produce como consecuencia de un receive ejecutado por un proceso de usuario pu_i. Si el buzón de pu_i se gestiona localmente, entonces no hay generación de mensaje de control. Si, por el contrario, el buzón de pu_i no se gestiona localmente, entonces el receive se transforma en un mensaje de control de petición más una retransmisión de mensaje de usuario.

En ningún caso un mensaje se envía a un destino (procesador) equivocado. El envío se dirige al procesador que contiene el buzón del proceso de usuario destino, y como el buzón permanece siempre en el mismo procesador, llegará al procesador correcto. Cuando un mensaje de usuario es pedido desde otro procesador, se puede enviar con la seguridad de que llegará al proceso de usuario destino porque es este mismo proceso el que lo requiere. Además, un proceso de usuario no se puede migrar mientras tenga pendientes recepciones de mensajes. De esta manera, no son necesarios los mensajes de rechazo de mensajes de usuario.

9.6 Message Rejection

Como en el caso de la política Follow Me, cada vez que se produce una migración de un proceso de usuario pu_i desde un procesador P*m* a un procesador P*n*, la nueva ubicación se conoce y almacena en estos dos procesadores. Si pu_i migra nuevamente desde P*n* a otro procesador distinto de P*m*, esta nueva ubicación es desconocida en P*m*. Los mensajes que lleguen destinados para pu_i al procesador P*m* serán retransmitidos a P*n*. Cuando la retransmisión llega a P*n*, si pu_i ha sido migrado, se genera un mensaje de control de no reconocimiento (NACK: *not acknowledge*) o *rechazo* del mensaje de usuario hacia el procesador P*m*, con la nueva ubicación de pu_i conocida en P*n*. Desde P*m* se vuelve a retransmitir el mensaje hasta que se reciba en el procesador al cual pu_i está asignado. La Fig. 9.7 muestra el caso en el cual pu_i ha migrado dos veces y se le envía un mensaje

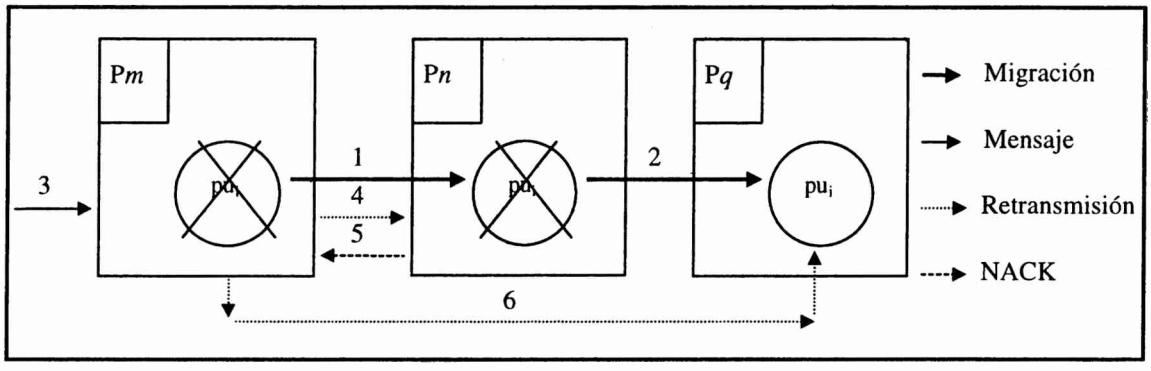


Figura 9.7: Retransmisiones y NACKs con Message Rejection.

La actualización de información que no es conocida localmente en un procesador se realiza cuando se recibe el rechazo (NACK) de un mensaje, es decir cuando recibe un mensaje de control NACK. Esta información es la que asocia a cada proceso de usuario con el procesador al cual está asignado. Las retransmisiones se producen hacia los procesadores a los cuales los procesos de usuario migran o hacia los procesadores indicados cuando se reciben los NACKS.

Se maneja un tipo de mensaje de control, el mensaje de rechazo de un mensaje de usuario, que tiene información asociada: el procesador hacia el cual migró el proceso de usuario destino. Como sucede en las políticas anteriores exceptuando la política Mailbox, las retransmisiones se producen después de recibir un rechazo de mensaje (NACK).

9.7 Migration Protocol

Toda la actividad para resolver el Problema de Arribo de Mensajes se realiza cuando se decide que un proceso de usuario pu_i va a migrar. De hecho, puede considerarse que se establece un protocolo de migración de procesos de usuario para conservar la consistencia en el arribo de mensajes.

Cuando se ha decidido que un proceso de usuario pu_i migrará, se deben llevar a cabo una sucesión de pasos para intercambiar información con los procesadores desde los cuales se pueden generar mensajes para pu_i. Definiendo que Pi_j, ..., Pi_k (no necesariamente $Pi_i \neq Pi_j$ si

$i \neq j$) son los procesadores a los cuales están asignados los procesos de usuario pu_{i1}, ..., pu_{ik} que tienen canales hacia el proceso de usuario pu_i que se migrará, los pasos a seguir son:

1. Enviar un mensaje de control a cada procesador Pi_1, ..., Pi_k para que no envíen más mensajes para pu_i.
2. Recibir de cada procesador Pi_1, ..., Pi_k la cantidad de mensajes que ya han sido enviados para pu_i.
3. Esperar que lleguen todos los mensajes dirigidos a pu_i.
4. Migrar físicamente el proceso de usuario pu_i.
5. Comunicar a Pi_1, ..., Pi_k la nueva ubicación de pu_i. A partir de este momento se pueden generar más mensajes dirigidos a pu_i.

Este protocolo asegura que no se producirá ninguna retransmisión de mensajes, pero genera $3k$ mensajes de control por cada migración, donde k es la cantidad de procesadores en que están asignados los procesos de usuario que envían mensajes al que se migra. Si todos los procesos de usuario están interconectados entre sí, k suele ser el número total de procesadores del DMPC en todas las migraciones que se generen.

Una alternativa para definir esta política suele ser que se realice la migración sin que se tenga en cuenta el tercer paso de la enumeración anterior. En esta alternativa se deben retransmitir los mensajes que no hayan llegado hasta el momento de la migración, y por lo tanto agrega más mensajes a la red de comunicaciones además de los mensajes de control del primer, segundo y quinto los paso, que siguen realizándose sin cambios.

Esta política es particularmente orientada a la utilización de broadcasts de mensajes. Lamentablemente, en la mayoría de los sistemas DMPC no se dispone de un mecanismo de broadcast de mensajes, ni tampoco de mecanismos de multicast de mensajes. En estos sistemas, aunque se provea una primitiva de comunicación de esta clase (broadcast o multicast) en general se implementa como una serialización de mensajes uno a uno, en este caso: un proceso origen y un proceso destino. El costo de implementar broadcast y multicast es demasiado alto para las necesidades clásicas de comunicación en un DMPC. Volviendo a la política de gestión de mensajes, tanto el primer paso, como el último paso de la enumeración anterior son orientados al broadcast de los mensajes (en este caso, mensajes de control desconocidos por las aplicaciones de usuario). En el caso de no contar ni siquiera con un mecanismo de broadcast o multicast a nivel de primitiva de comunicación, se deben enviar los mensajes de forma serie explícitamente, y esto podría causar problemas entre los procesos que gestionan los mensajes.

Esta política define tres tipos de mensajes de control:

1. Mensaje de inicio de migración de un proceso de usuario pu_i.
2. Mensaje de respuesta al mensaje de inicio de migración de pu_i, donde se responde con la cantidad de mensajes dirigidos a pu_i en cada procesador.
3. Mensaje de fin de migración de pu_i.

10. Diseño de un Entorno para Experimentación de Migración en DMPCs

Este capítulo se dedica a la descripción del diseño de un entorno dedicado a la experimentación en el contexto de arquitectura MIMD con memoria distribuida, en la que se agrega la capacidad de migración dinámica de procesos. Se presenta también la adaptación del diseño presentado a las políticas propuestas en el capítulo anterior. Para conocer el comportamiento de cada política se deben definir los índices que se pueden utilizar para la evaluación y comparación de las políticas propuestas para resolver el problema de integridad de mensajes. Es decir qué índices de rendimiento se tienen en cuenta y por qué. Dado que la experimentación se lleva a cabo en una máquina paralela real, debe ser posible la ejecución de aplicaciones paralelas de usuario. Por esta razón se define también un modelo sintético de aplicaciones de usuario para poder especificar distintos tipos de problemas que pueden presentar los programas paralelos.

El objetivo inicial para el entorno para experimentación de migración en DMPCs consiste en probar y comparar cada política propuesta para resolver el Problema de Arribo de Mensajes (PAM). Estas políticas de manejo de mensajes de usuario fueron estudiadas en principio mediante un simulador de eventos discretos, lo cual permitió conocer el comportamiento intrínseco de cada una de ellas sin tener en cuenta la carga de la red de comunicaciones ni de los procesadores [Hey95]. La carga de la red de comunicaciones se compone de los mensajes propios de la aplicación de usuario más los mensajes que genere cada política. La carga de los procesadores está compuesta por los procesos de la aplicación de usuario más los procesos necesarios para llevar a cabo cada política propuesta para resolver el Problema de Arribo de Mensajes.

En [Hey95] se asumieron algunas características simplificadas con respecto a las aplicaciones de usuario, como así también al entorno de ejecución de las aplicaciones. Estas simplificaciones permitieron, en principio, simular las políticas sin conocer detalles tales como la congestión de la red de comunicaciones, mensajes que generan conflictos, ni procesamiento necesario en cada procesador que se añade para cada política. Como resultado, la información que proveen esas simulaciones no tienen en cuenta estas características que son relevantes a la hora de evaluar y comparar comportamientos.

La simulación por eventos discretos se hace muy complicada una vez que se ha decidido conocer el comportamiento de cada política teniendo en cuenta la (sobre)carga de la red de comunicaciones y de los procesadores. Para simular teniendo en cuenta estas características se deben incluir detalles no sólo de la aplicación paralela de usuario sino también de la máquina paralela misma (básicamente de los procesadores y de la red de interconexión), y los procesos necesarios para llevar a cabo las políticas propuestas para resolver el Problema de Arribo de Mensajes. Aunque las aplicaciones de usuario se pueden describir y simular de forma sencilla, no es tan claro que lo mismo se puede hacer para la red de comunicaciones y los procesadores de la máquina paralela.

Entre los objetivos que guiaron el diseño de este entorno para experimentación de migración en DMPCs se pueden mencionar:

- Flexibilidad: esencial para implementar las políticas definidas, cambiarlas, y agregar otras características al entorno paralelo para medir el impacto en las aplicaciones de usuario con el mínimo esfuerzo. Esto implica al menos una fuerte modularización.
- Transparencia: como mínimo, los procesos de usuario no tienen en cuenta ni hacen nada para que sea más fácil el mecanismo de migraciones ni las políticas que resuelven el PAM. Se intenta seguir la idea de "Transparencia de Ubicación" como se la define en [Tan92] con respecto a los procesos de la aplicación. Los procesos de usuario solamente procesan (secuencialmente) y se comunican (a través de canales) con otros procesos de usuario.
- Facilidades para la Experimentación: Debe ser posible simular las aplicaciones de usuario definiendo el comportamiento de los procesos en términos de la relación procesamiento/comunicación y las interconexiones.

Con respecto a la flexibilidad del entorno para experimentación, podría agregarse que dentro de las principales extensiones se podrían encontrar: las políticas específicas de migración, otras políticas para mantener la consistencia en el arribo de los mensajes y también los diferentes mecanismos de migración de procesos. En todos los casos, además de la implementación, debe ser posible instrumentar, experimentar y evaluar las distintas alternativas que se propongan.

Aunque el diseño se orientó a la experimentación con las políticas de gestión de mensajes, no se puede dejar de establecer una política de migración ni un método de migración de procesos de usuario. Para que las políticas de gestión tengan sentido, los procesos deben ser transferidos entre los procesadores con algún método de migración, y para que esto sea posible se debe definir una política de migración de procesos de usuario. Es por esta razón también que se deben definir las aplicaciones de usuario con las cuales se realizará la experimentación.

10.1 Modelo Sintético de las Aplicaciones de Usuario

Las aplicaciones de usuario a ejecutar sobre la máquina paralela se definen de acuerdo a varios parámetros que la caracterizan:

- la cantidad de procesos que la componen,
- el patrón de interconexiones entre los procesos, y
- el comportamiento de cada proceso.

El comportamiento de cada proceso de usuario es muy importante para definir el funcionamiento de toda la aplicación, por lo tanto se lo define por: (a) la frecuencia de comunicaciones (mensajes) y (b) el cómputo que requiere, que equivale a tiempo de utilización de CPU.

La definición de requerimientos de cómputo y de comunicación de los procesos de usuario es inmediatamente adaptable a un entorno de multiprogramación. Cuando un proceso de usuario tenga su turno de posesión de CPU (en un *time slice*), si no genera comunicación utilizará todo el tiempo que tiene asignado. Si genera comunicación, el proceso de usuario utilizará una parte del tiempo asignado y luego enviará un requerimiento de mensaje. La generación o no de mensajes dependerá del requerimiento de comunicación que se defina para el proceso de usuario. El tiempo de que se utilice de la CPU cuando se genera un mensaje dependerá del requerimiento de cómputo que se defina para el proceso de usuario.

Esquemáticamente, un proceso de usuario en un entorno multiprogramado puede comportarse como lo muestra la Fig. 10.1 una vez que tiene asignada la CPU. En el caso mostrado en la parte (a), el proceso de usuario utiliza todo el tiempo de CPU que tiene asignado, y el control vuelve al proceso encargado de asignar la CPU a los procesos de usuario (scheduler). En el caso mostrado en la parte (b), el proceso de usuario genera un mensaje entre el tiempo de inicio de su quantum (en que se le asigna la CPU) y el tiempo de fin de su quantum.

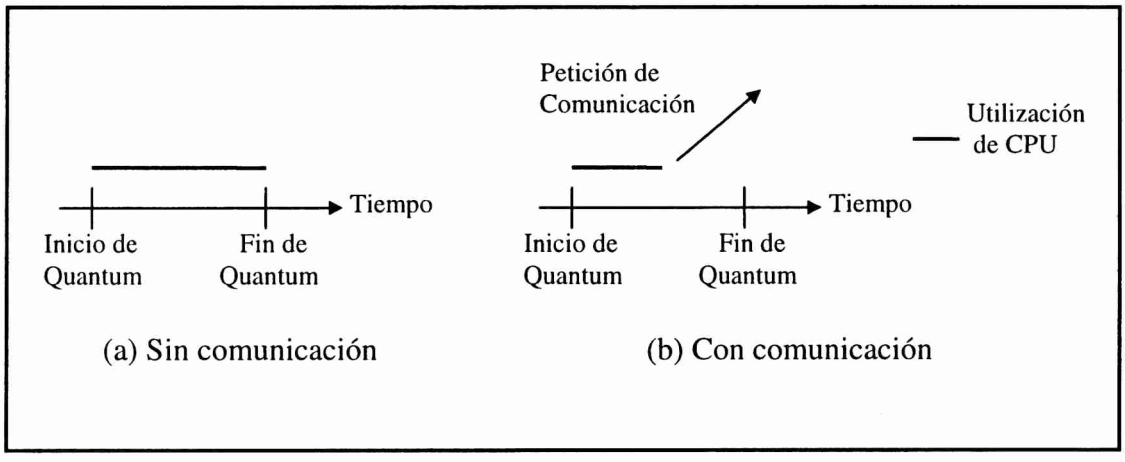


Figura 10.1: Multiprogramación: Proceso de Usuario *Sintético*.

La probabilidad con que un proceso de usuario genera un mensaje depende de sus requerimientos de comunicaciones. El intervalo de tiempo durante el cual utiliza la CPU cuando genera un mensaje depende de sus requerimientos de cómputo. Lo más usual en la definición de los requerimientos de un proceso de usuario es que sean inversamente proporcionales, es decir: a mayores requerimientos de comunicación se tendrán menores requerimientos de cómputo. Cuando el proceso de usuario genera un requerimiento de mensaje o se termina su tiempo asignado, libera la CPU y se la puede asignar a otro proceso de usuario.

Se evita la utilización de aplicaciones de usuario reales por varias razones. En principio no obliga a elegir, diseñar e implementar un conjunto de aplicaciones de usuario que deban "representar" a todas las posibles. También permite definir aplicaciones (sintéticas) rápidamente, lo que a su vez permite probar las políticas para distintos casos de aplicaciones de usuario. Se evita además entrar en detalles muy específicos con respecto al manejo de los procesos de usuario, tales como la asignación de memoria, cambios de contexto y migración física.

Otra de las ventajas de simular aplicaciones de usuario consiste en evitar el conocimiento en detalle de la forma de ejecución de cada máquina paralela en particular. Si se tiene que implementar, por ejemplo, el método de planeamiento de ejecución (scheduling) round-robin, se debería conocer en detalle la forma en la que a un proceso de usuario se le asigna el recurso CPU y también cómo se le quita el recuso (preemption). También se deberían conocer detalles tales como el cambio de contexto entre los procesos: registros del procesador a salvar/recuperar, registros del procesador a asignar con nuevo estado, etc., sea entre procesos de usuario o entre los procesos que gestionan los mensajes.

Con respecto a la comunicación entre los procesos de usuario se asume, como en [Hey95], que no hay esperas por sincronización, es decir que cuando un proceso de usuario envía un mensaje, el proceso de usuario destino del mismo lo recibe inmediatamente. Este comportamiento es similar al que se encuentra cuando las aplicaciones de usuario utilizan primitivas de comunicación por mensajes no bloqueantes (el proceso que envía no se detiene para esperar al que recibe). Si por el contrario se consideran primitivas de comunicación bloqueantes, es como si se tuviera una aplicación de usuario óptima en cuanto a la sincronización para las comunicaciones. De esta manera se pueden comparar y evaluar las políticas que resuelven el Problema de Arribo de Mensajes sin tener en cuenta los retrasos producidos para la recepción de mensajes. Siempre la idea subyacente es analizar y comparar las políticas de la forma más independiente de la aplicación de usuario que sea posible.

10.2 Migraciones de los Procesos de Usuario

Dado que el objetivo inicial es la comparación de las políticas que preservan la consistencia en el arribo de los mensajes, se decide que la política de migraciones sea lo más sencilla posible, y que no introduzca desbalance de carga en la ejecución de la aplicación de usuario.

Para evitar que la política de migraciones produjera demasiada carga en la red de comunicaciones o en la cantidad de cómputo necesaria, se resolvió seguir la idea de [Hey95], de generar migraciones al azar según los mensajes que se hayan generado. Esta decisión implica que la tasa o el porcentaje de migración se define con relación a los mensajes que se generen en la aplicación de usuario. Si la tasa de migración es m, por cada mensaje generado por un proceso de usuario, la probabilidad de que se genere una migración es m. Otra de las implicaciones consiste en la distribución uniforme de las migraciones durante el tiempo que lleva simular la aplicación paralela de usuario.

Si bien esta política es discutible en el contexto de la migración de procesos [Art89] [Del91], es la que produce menos interferencia con las políticas que resuelven el Problema de Arribo de Mensajes, porque casi no consume los recursos disponibles de cómputo ni de comunicación. Por otro lado, dado que todos los procesos de usuario tienen los mismos requerimientos de comunicación, mantiene el balance de carga de cómputo y de comunicaciones en cada procesador.

Definir las migraciones en términos de la probabilidad de migrar por mensaje de la aplicación, también tiene la ventaja de poder contabilizar la cantidad total de migraciones que se realizan. Si se conoce a priori la cantidad de mensajes que se generarán en la aplicación de usuario, también se conocerá a priori la cantidad de migraciones que se producirá. Si además se define que todos los procesos de usuario tienen la misma probabilidad de ser elegidos para migrar, entonces también se conoce la cantidad de migraciones promedio de cada proceso de usuario.

10.3 Máquina Paralela

La máquina paralela considerada para la experimentación es básicamente un DMPC (*Distributed Memory Parallel Computer*) con la red de comunicaciones virtualizada, como la que se muestra en la Fig. 10.2 con N procesadores.

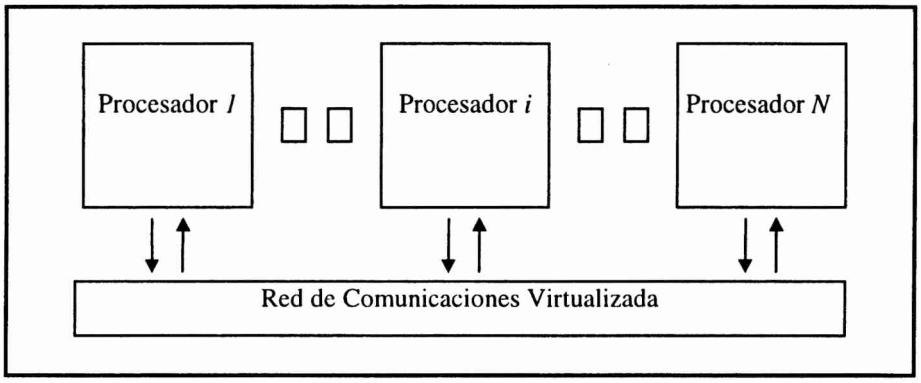


Figura 10.2: DMPC con Red de Comunicaciones Virtual.

Los procesadores tienen su propia memoria local a la cual acceden con exclusividad, toda la comunicación se produce a través de mensajes que se envían entre procesadores. Los procesadores se comunican a través de canales y cada procesador puede comunicarse con cualquier otro con el mismo tiempo promedio de transmisión de mensajes [Fra93] [Izu92].

La virtualización de la red de comunicaciones es especialmente útil para las redes estáticas de comunicación, ya que evita conocer en detalle la forma en que los procesadores se comunican. Por esta razón, en algunos lenguajes paralelos ya se provee de manera estándar. En el lenguaje C de INMOS para los transputers, por ejemplo, cualquier proceso se puede comunicar con cualquier otro proceso, independientemente de la ubicación de ambos (asignación a procesadores) [INM92a] [INM92b] [INM92c].

La virtualización de la red de comunicaciones no solamente permite disponer de un grafo completo de canales de comunicación, con todos los procesadores conectados entre sí, sino también otras características como [Fra93] [Mou93]:

- Utilización de todo el ancho de banda de la red de comunicaciones. Se pueden aprovechar todos los caminos disponibles para enviar mensajes desde un procesador a otro.
- Es posible obtener latencia de mensajes uniforme cuando la carga de la red de comunicaciones no cambia, todos los procesadores están comunicados a la misma distancia en términos de tiempo de comunicación.
- Definición de nuevos tipos de canales de comunicación disponibles para las aplicaciones. Se pueden definir canales *1* a *N*, y *N* a *1* entre otros, cuando los canales de comunicación más comunes que se pueden encontrar en un DMPC son canales uno a uno, es decir que conectan solamente dos procesos.

En una máquina paralela con estas características, migrar un proceso de usuario desde un procesador a otro, tiene el mismo costo en tiempo de comunicación, independientemente de los procesadores origen y destino de la migración. Lo mismo sucede con los mensajes que se envían entre los procesos de usuario. Esto se debe a que no hay distintas "distancias" desde un procesador hacia los demás de la red, el tiempo que lleva comunicarse con cualquiera de ellos es el mismo.

Considerando el modelo sintético de la aplicación de usuario, migrar un proceso de usuario de un procesador a otro no implica mucho más que enviar la descripción sintética del proceso de usuario que se quiere migrar. Avanzando un poco más en el nivel de detalle de la ejecución de aplicaciones de usuario, se puede enviar un mensaje mayor al de la descripción

sintética para simular la carga que se produce en la red de comunicaciones por la migración de procesos de usuario reales. Estos mensajes pueden simular el envío de información asociada al proceso de usuario que migra tal como: área de memoria para datos, área de memoria para código, estadísticas que se utilicen para la planificación de la ejecución, etc.

La carga de cómputo que produce una aplicación en la máquina paralela está dada de forma directa por los requerimientos de cómputo de los procesos de usuario más la utilización de la CPU por parte de los procesos de gestión de las migraciones (política de migración, método de migración y gestión de los mensajes). La carga en la red de comunicaciones está dada por los requerimientos de comunicación de los procesos de usuario, más los mensajes generados para la gestión de las migraciones (mensajes de control de las políticas, por ejemplo), más la simulación del envío de los procesos de usuario entre los procesadores. De esta manera se puede simular la carga de la red de comunicaciones que se produce por las migraciones dinámicas de procesos de usuario, sin conocer ni manejar las particularidades de la máquina paralela que se utilice en cuanto a la asignación de memoria e información asociada a cada proceso de usuario.

10.4 Procesos Básicos del Entorno para Experimentación

El diseño del entorno para experimentación de migraciones en DMPCs desarrollado puede describirse con los procesos de la Fig. 10.3, que pueden modificarse para cada política o característica del entorno paralelo que se quiera agregar.

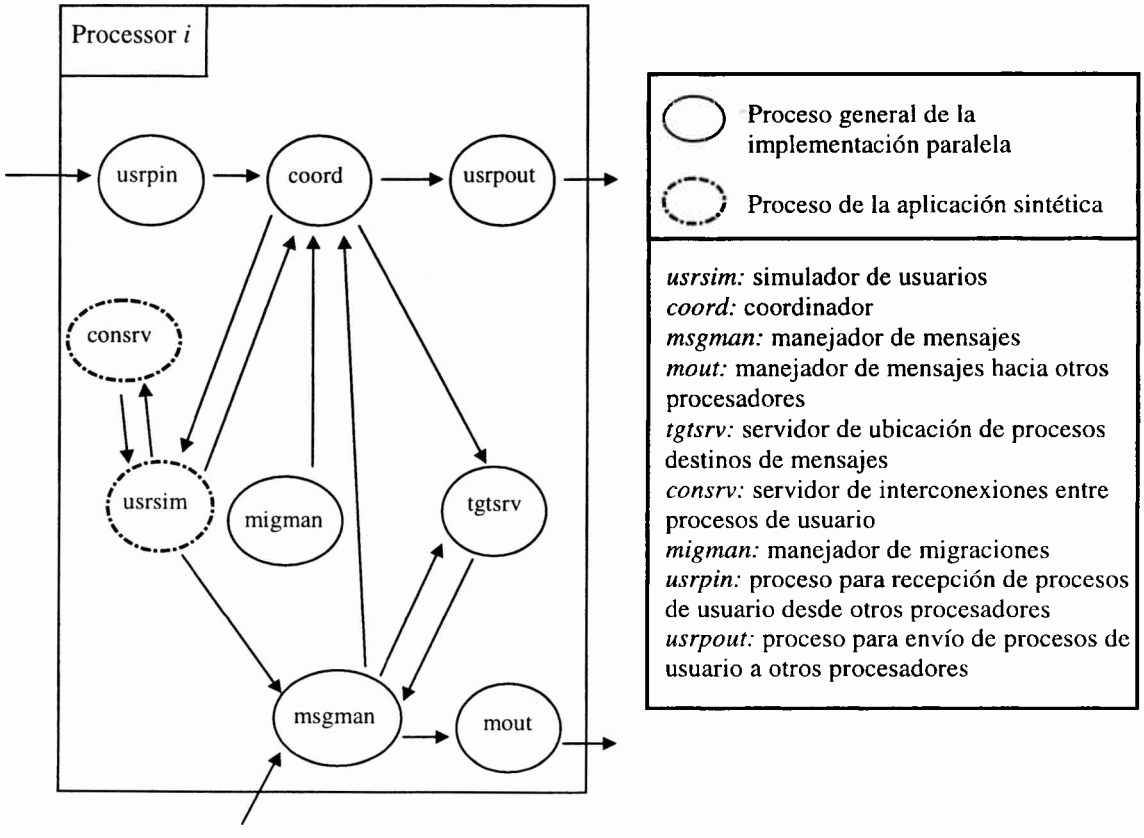


Figura 10.3: Procesos del Entorno para Experimentación.

Los procesos que se muestran en la Fig. 10.3 con línea discontinua son los que se necesitan para manejar la aplicación sintética de usuario. Los demás procesos, que se muestran con línea continua, son propios del entorno para experimentación, para la ejecución de aplicaciones de usuario: para las políticas que conservan la consistencia en el arribo de mensajes, y para las migraciones de procesos de usuario.

El proceso ***usrsim*** (proceso simulador de procesos de usuario) simula el comportamiento de los procesos de usuario en uso de CPU, como en los sistemas de multiprogramación. Dicho proceso recibe la descripción de un proceso de usuario desde el proceso *coord* (probabilidad de envío de mensajes y valor de cómputo), y simula el proceso de usuario hasta que abandone la CPU.

Si un usuario no se comunica en el tiempo que se simula (tiene asignada la CPU), el proceso *usrsim* simula el cómputo del proceso de ese usuario y pide al proceso *coord* el siguiente usuario para simular. Si un proceso de usuario que se está simulando se comunica, el proceso *usrsim* simula el uso de CPU de acuerdo a los requerimientos de cómputo del usuario y realiza una petición de mensaje al proceso *msgman*. Si se debe generar un mensaje, el proceso *usrsim* pide al proceso *consrv*, que conoce el grafo de interconexión entre procesos de usuario, el otro proceso de usuario con el cual se comunicará y envía un requerimiento de mensaje al proceso *msgman*. Después del envío del requerimiento de mensaje, requiere del proceso *coord* la descripción del siguiente proceso de usuario a simular.

El proceso *usrsim* es el que en realidad implementa el comportamiento de cada proceso de usuario en términos de requerimientos de cómputo y de comunicación. El proceso *usrsim* es el que en realidad utiliza la CPU y el que genera los requerimientos de comunicación hacia el proceso *msgman* tal como se describe de manera esquemática en la Fig. 10.1.

El proceso ***coord*** (proceso coordinador) mantiene a los procesos de usuario ejecutándose tanto como sea posible y además envía y recibe procesos de usuario por migraciones. Para que los procesos de usuario se ejecuten, el proceso *coord* se comunica con el proceso *usrsim*, y para llevar a cabo las migraciones envía y recibe procesos de usuario hacia y desde otros procesadores. Los procesos de usuario son enviados para ejecutar al proceso *usrsim* cuando éste está listo, esto es, cuando ha finalizado la simulación del proceso de usuario anterior.

Las peticiones de migración hacia otros procesadores son recibidas en el proceso *coord* desde el proceso *migman* y se resuelven enviando un proceso de usuario a otro procesador, ambos elegidos aleatoriamente, vía el proceso *usrpout*. Los procesos de usuario que llegan al procesador se reciben en el proceso *coord* desde el proceso *usrpin* el cual los recibe desde otros procesadores. Para todas las migraciones, se debe actualizar la información del proceso *tgtsrv* con la nueva ubicación del proceso de usuario que deja el procesador o llega al procesador.

El proceso ***msgman*** (proceso para manejar mensajes), se encarga de gestionar todos los requerimientos de mensajes. Estos pedidos llegan desde el proceso *usrsim* que se ejecuta en el mismo procesador o desde los otros procesadores. Para todos los requerimientos, el proceso *msgman* solicita del proceso *tgtsrv* la ubicación (el procesador) del proceso de usuario destino del mensaje. Si el proceso de usuario destino está asignado al mismo procesador, el

mensaje es consumido, en caso contrario, el requerimiento es manejado de acuerdo a la política que se utilice para resolver el PAM. Cuando el mensaje tiene que ser enviado a otro procesador, el proceso *msgman* envía el requerimiento al proceso *mout*. Cuando el requerimiento proviene del proceso *usrsim*, y una vez que ha sido procesado, el proceso *msgman* notifica al proceso *coord* que el requerimiento de comunicación del proceso de usuario local ya ha sido manejado.

El proceso ***mout*** (proceso para manejar mensajes hacia otros procesadores) se dedica a distribuir los requerimientos de mensajes del proceso *msgman* hacia los demás procesadores. Esta tarea podría ser realizada por el mismo proceso *msgman*, pero se intenta que el proceso *msgman* solamente se dedique a manejar los requerimientos de mensajes para procesos de usuario y no se encargue de la distribución física de los requerimientos para los procesadores que correspondan. Esta es, justamente, la tarea a la que se dedica el proceso *mout*.

El proceso ***tgtsrv*** (proceso servidor de ubicación de procesos de usuario) responde a los pedidos del proceso *msgman* referidos a la ubicación de procesos de usuario y mantiene actualizada la información, dependiendo de la política utilizada para resolver el PAM, al menos de los procesos de usuario que se ejecutan localmente. Este proceso recibe del proceso *coord* la información de los procesos de usuario que llegan y que salen del procesador.

El proceso ***consrv*** (proceso servidor de interconexiones entre procesos de usuario) conoce el grafo de interconexión entre procesos de usuario y responde a los requerimientos del proceso *usrsim* sobre procesos de usuario a comunicar. Una vez que un proceso de usuario ha decidido enviar un mensaje, el proceso *usrsim* requiere del proceso *consrv* el proceso de usuario destino del mensaje. En realidad, en el proceso *conrsrv* se define la interconexión entre los procesos de usuario. Dado un proceso de usuario que enviará un mensaje, el proceso *consrv* selecciona aleatoriamente un proceso de usuario destino para este mensaje.

El proceso *consrv* no es necesario en una aplicación real, ya que cada proceso de usuario conoce con quién se comunica en cada mensaje. Teniendo en cuenta que se trata de un entorno para experimentación, el proceso *consrv* es útil para tener la posibilidad de probar distintos patrones de interconexión entre procesos de usuario que se podrían dar en las aplicaciones reales.

El proceso ***migman*** (proceso para manejar migraciones) solamente genera requerimientos de migración para el proceso *coord*. En realidad, el proceso *migman* implementa la política de migración de los procesos de usuario. Decisiones tales como qué proceso de usuario migrar, hacia qué procesador migrarlo y cuándo realizar la migración deben estar reflejadas en este proceso [Cor92] [Corr92].

Los procesos ***usrpin*** y ***usrpout*** (procesos para recepción y envío de procesos de usuario desde y hacia otros procesadores respectivamente) manejan la migración física de los procesos de usuario. Como los procesos de usuario son simulados, la migración solamente produce un mensaje desde un proceso *usrpout* hacia un proceso *usrpin* en otro procesador que representa al proceso de usuario que se migra desde un procesador a otro. Este mensaje contiene principalmente la identificación del proceso de usuario e información asociada a él. El proceso *usrpout* corresponde al procesador desde el cual se migra, y el proceso *usrpin* corresponde al procesador en el cual el proceso de usuario continuará ejecutándose.

Con los procesos *usrpin* y *usrpout* se evita entrar en los detalles de la migración física de procesos de usuario a la vez que se tiene, como en los sistemas reales, la carga sobre la red de comunicaciones que se produce por las migraciones. Decisiones tales como dejar o no parte o todo el proceso de usuario en el procesador desde el cual migra, o liberar y asignar memoria para el proceso de usuario, se deberían reflejar en estos procesos (*usrpin* y *usrpout*) en una implementación con procesos de usuario reales (no simulados) [Tan92], [Sta92].

Cada proceso *mout* tiene que estar comunicado con cada proceso *msgman* de los otros procesadores, y asimismo cada proceso *usrpout* tiene que estar comunicado con cada proceso *usrpin* de los otros procesadores. De esta manera, se definen dos circuitos de información separados: uno para datos de los mensajes de procesos de usuario (*mout* - *msgman*), y otro para los datos de los procesos de usuario que migran (*usrpin* - *usrpout*). La forma en que los procesos *mout*, *msgman*, *usrpin* y *usrpout* se comunican, así como los dos circuitos de información se pueden ver en la Fig. 10.4 para cuatro procesadores.

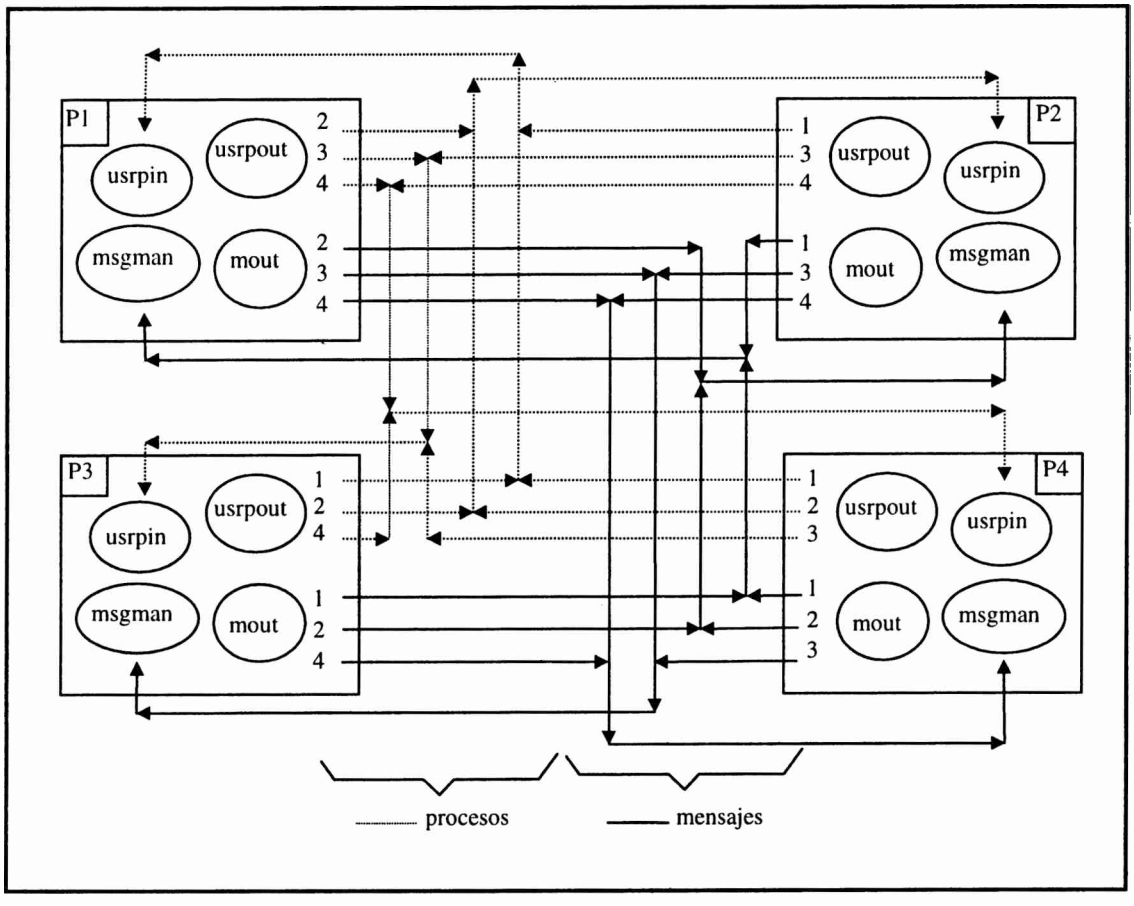


Figura 10.4: Circuitos de Información con cuatro procesadores.

En la Fig. 10.4, los números asociados a cada proceso *usrpout* o *mout* indican el procesador destino de los mensajes que circulan por los canales. La interconexión mostrada se realiza por medio de la utilización de canales *N* a *1*, en el caso de no contar con ellos se agrega en cada procesador un proceso que cumpla esta función. Por un lado, los mensajes que circulan desde un proceso *usrpout* hacia un proceso *usrpin* tendrán como contenido un proceso de usuario que migra. Por el otro, los mensajes que circulan desde un proceso *mout* hacia un proceso *msgman* tendrán como contenido un mensaje enviado desde un proceso de usuario hacia otro.

Dependiendo de la política que resuelva el Problema de Arribo de Mensajes, el circuito de datos de los mensajes de procesos de usuario podría incluir los mensajes de control. La información asociada a la migración de los procesos de usuario (código, datos del proceso, cantidad de memoria asignada, etc.) nunca se mezcla con los mensajes de procesos de usuario. Los cambios que se hagan en uno de los circuitos no afectan al otro circuito.

La estructura de procesos del entorno presentada es necesaria en cada procesador para que los procesos de usuario realicen su tarea de procesar y enviar mensajes. La asignación inicial de los procesos de usuario a los procesadores es realizada por los procesos cargadores de la aplicación. Las alternativas posibles para cargar la aplicación de usuario en la máquina paralela son dos como primera aproximación: (a) un proceso cargador general que le haga llegar la información a cada proceso de cada procesador o (b) un proceso cargador general que se comunique con cargadores locales y que son los que distribuyen la información en cada procesador.

La estructura de procesos del entorno en cada procesador permite la simulación de la ejecución de aplicaciones de usuario de acuerdo a los parámetros:

- Cantidad de Procesadores.
- Cantidad de procesos de usuario.
- Comportamiento en lo referente a comunicación y cómputo de los procesos de usuario.
- Patrón de interconexión entre los procesos de usuario.
- Política de migración.
- Política de *scheduling* de los procesos de usuario.

Las políticas de migración y planeamiento de ejecución quizás no sean paramétricas en el sentido de cambiar los comportamientos con valores numéricos o valores de parámetros. En ambos casos es posible que sean necesarios cambios en los procesos *coord* y *migman*, pero estos cambios no afectarían a los demás procesos, se encuentran concentrados en los procesos *coord* y *migman*, y por lo tanto serían menores con respecto al entorno para experimentación de migración en DMPCs completo.

La cantidad de procesadores es casi independiente de los demás parámetros y está directamente relacionada con la interconexión entre los procesos *mout* - *msgman* y *usrpout* - *usrpin*.

La cantidad de procesos de usuario no afecta mucho más que a la cantidad de memoria que se necesita para almacenar toda la información en cada procesador. En principio, dada la descripción sintética de la aplicación de usuario, hasta se podrían simular aplicaciones mayores de las que la computadora paralela podría ejecutar en realidad.

El comportamiento de los procesos de usuario modificará la cantidad de tiempo que el proceso *usrsim* utiliza para simular los tiempos de estos procesos en posesión del procesador y la cantidad de mensajes que generarán por unidad de tiempo. El patrón que siguen los procesos de usuario para su interconexión se verá reflejado en las respuestas a los requerimientos que provea el proceso *consrv*, ya que es el encargado de responder con un proceso de usuario destino de un mensaje dado un proceso de usuario origen del mismo.

La política de migración se implementa directamente en el proceso *migman*, y de ella dependen decisiones tales como cuándo migrar un proceso de usuario, hacia qué procesador

migrar el proceso de usuario, y cuál de los procesos de usuario migrar.

La planificación de la ejecución (*scheduling*) de los procesos de usuario se define e implementa en el proceso *coord*, ya que éste es el que decide qué proceso de usuario se envía para su ejecución simulada en el proceso *usrsim* y el que efectivamente lo envía.

Desde el punto de vista de los simuladores [Law91] [Mac87], lo único que se está simulando son los procesos de usuario, ya que los demás procesos deben estar presentes en un entorno paralelo que provea la posibilidad de migrar dinámicamente procesos de usuario. Algunos procesos no son muy distintos (aunque sí simplificados) de los que pueden verse en los sistemas operativos tradicionales [Tan88], [Sil94]. La decisión de simular los procesos de usuario simplifica el diseño del entorno para experimentación. Esto a su vez hace posible concentrarse en las políticas que resuelven el Problema de Arribo de Mensajes, sin entrar en detalles propios de la máquina de simulación utilizada y de la forma en que se debería producir la migración física de los procesos de usuario.

10.5 Adaptación del Diseño a cada Política

Volviendo a las políticas para resolver el Problema de Arribo de Mensajes, no es necesario cambiar la estructura de procesos general que se muestra en la Fig. 10.3 para implementar cada una de ellas. A continuación se describen los principales cambios para cada política.

10.5.1 Follow Me

No hay ningún cambio en el esquema de la Fig. 10.3 para implementar la política Follow Me, la estructura general de procesos la implementa directamente. Como cada vez que hay una migración del procesador P_i al P_j se actualiza la información de los procesos *tgtsrv* de P_i y de P_j, cada proceso *tgtsrv* considera la ubicación de un proceso de usuario en un momento dado, como:

- El procesador local, si el proceso de usuario está asignado al procesador.
- El procesador al que fue asignado inicialmente, si el proceso de usuario nunca estuvo asignado al procesador local.
- El procesador al cual fue migrado el proceso de usuario, si no es local pero estuvo asignado alguna vez al procesador local anteriormente.

Todos los procesos mantienen la definición ya realizada en la sección de descripción de los procesos básicos del entorno para experimentación de migración en DMPCs.

Para conservar la consistencia en el arribo de los mensajes en cada procesador se utiliza y mantiene solamente información local, tal como queda establecido por la política. Todos los mensajes entre procesadores se producen por mensajes que se envían entre procesos de usuario o por migraciones de procesos de usuario.

Las retransmisiones de los mensajes que se deban producir por migraciones del proceso de usuario destino del mensaje se realizan automáticamente de acuerdo al funcionamiento definido para los procesos *tgtsrv* y *msgman*. En el tiempo en que se carga la

aplicación de usuario en cada procesador se le proporciona al proceso *tgtsrv* la información sobre la asignación inicial de los procesos de usuario a cada procesador.

10.5.2 Global Server

Los cambios para el Global Server consisten en agregar el proceso Servidor Global para responder a las peticiones de ubicación de procesos de usuario. Las peticiones llegarán desde todos los procesadores. Este proceso servidor es similar al proceso *tgtsrv*, y el proceso *tgtsrv* se deja en cada procesador para resolver solamente los requerimientos locales (es decir, para determinar si un proceso de usuario es local o no). De esta manera, los procesos *tgtsrv* de cada procesador estarán comunicados con el proceso Servidor Global para pedir la ubicación de procesos de usuario que no estén asignados al procesador local y para informar sobre los procesos de usuario que migren.

El único proceso que cambia de la definición realizada en la descripción general es el proceso *tgtsrv*, además de incorporarse el proceso *glbsrv*. El proceso *tgtsrv* tendrá que comunicarse con el proceso *glbsrv* para realizar algunas de las tareas que se le requieren desde los procesos *msgman* y *coord*. Ninguno de estos procesos que se comunican localmente con el proceso *tgtsrv* debe cambiarse porque el comportamiento en términos de requerimiento-respuesta es el mismo.

Esquemáticamente, los procesos y conexiones para implementar esta política son los que muestra la Fig. 10.5.

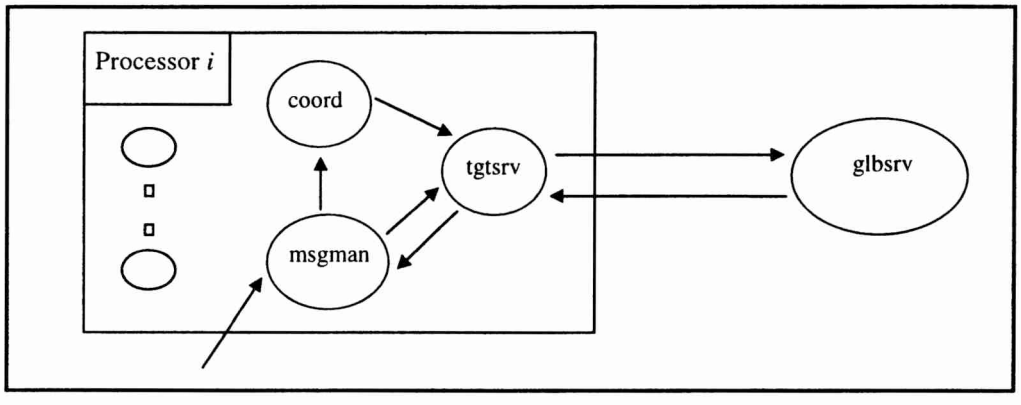


Figura 10.5: Diseño de la Política Global Server.

El proceso Servidor Global (*glbsrv*) genera un cierto desbalance de carga si se lo asigna a un procesador de los que contienen procesos de usuario. El desbalance se produce en términos de carga de procesamiento y de comunicaciones. Por un lado, el proceso *glbsrv* debe utilizar el procesador para responder a los requerimientos de ubicación y de actualización de procesos de usuario. Por otro lado, todos los mensajes de control que se generan en esta política tienen que llegar o salir del procesador al cual se asigne el proceso *glbsrv*.

La virtualización de las comunicaciones es particularmente útil para comunicar a todos los procesadores con aquel al que se asigne el proceso *glbsrv*. Sin esta virtualización, todos los procesadores deberían incluir procesos de ruteo de la información que se dirige al proceso *glbsrv*.

10.5.3 Home Processor

En esta política, el proceso *tgtsrv* siempre conoce el Home Processor de todos los procesos de usuario que no están ejecutándose localmente. Usualmente el Home Processor de un proceso de usuario es el procesador al cual es asignado inicialmente (mapping). Esta información es conocida y asignada en tiempo de carga de la aplicación en la máquina paralela.

De acuerdo a la definición de la política, se puede afirmar que para cada proceso *tgtsrv*, la ubicación de un proceso de usuario en un momento dado es:

- El procesador local, si el proceso de usuario está asignado al procesador.
- El Home Processor del proceso de usuario, si el proceso de usuario no está asignado al procesador local.

Dado que el Home Processor de un proceso de usuario siempre debe conocer la ubicación de ese proceso de usuario, siempre que hay una migración desde el procesador P_i al P_j, es posible que no alcance con actualizar la información de los procesos *tgtsrv* en P_i y en P_j. También se debe actualizar la información de ubicación del proceso de usuario en el Home Processor si no es ni P_i ni P_j. La información de actualización del proceso *tgtsrv* que corresponda, se envía desde el proceso *coord* cada vez que hay una migración. Esta información de actualización es como la que se envía al proceso *tgtsrv* local.

Para que la información en el Home Processor de cada proceso de usuario se mantenga actualizada, cuando hay una migración, el proceso *coord* recibe del proceso *tgtsrv* la identificación del Home Processor del proceso de usuario que se está migrando. Lo único que se agrega en el proceso *coord* es este envío de un mensaje de control para que el Home Processor conozca la nueva ubicación del proceso de usuario.

Desde el punto de vista del proceso *tgtsrv*, cuando recibe un pedido de actualización para un proceso de usuario que migra del procesador local, debe responder al proceso *coord* con la identificación del Home Processor del proceso de usuario que migra. De esta forma se agregan vías de comunicación entre los procesos *coord* y los procesos *tgtsrv* que no están en el esquema general de la Fig. 10.3. La Fig. 10.6 muestra esquemáticamente la forma en que se interconectan los procesos *coord* y *tgtsrv*, tanto en el mismo procesador como entre distintos procesadores.

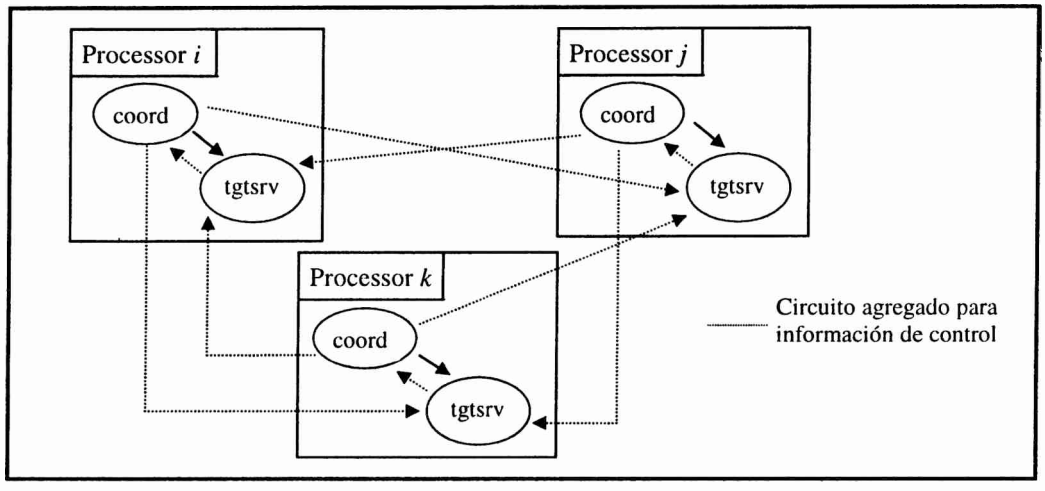


Figura 10.6: Diseño de la Política Home Processor.

En la Fig. 10.6, el proceso *tgtsrv* tiene un nuevo canal de salida hacia el proceso *coord*, para enviarle la identificación del Home Processor de un proceso de usuario que se enviará a otro procesador. El proceso *coord* debe tener un canal de salida hacia cada proceso *tgtsrv* en los demás procesadores para comunicarles la identificación del procesador hacia el cual un proceso de usuario migra. Este mensaje de actualización se produce solamente hacia el Home Processor del proceso de usuario que migra. La Fig. 10.6 lo muestra para tres procesadores *i*, *j*, y *k*.

Se podría afirmar que se agrega un tercer circuito de información, para los mensajes de control necesarios entre los procesos *coord* y *tgtsrv*. Los mensajes de control agregados son dos: (a) identificación del Home Processor desde el proceso *tgtsrv* al proceso *coord* local, y (b) actualización de la ubicación de un proceso de usuario desde un proceso *coord* a un proceso *tgtsrv* no local.

Los procesos *tgtsrv* y *coord* son los que se cambian en cuanto a la definición realizada para la estructura general del entorno para experimentación de migración en DMPCs. El proceso *tgtsrv* responde a cada requerimiento del proceso *coord* con el Home Processor del proceso de usuario que se está migrando. El proceso *coord* recibe del proceso *tgtsrv* el Home Processor y le notifica la nueva ubicación del proceso de usuario que migra. Todos los demás procesos permanecen sin cambios.

10.5.4 Mailbox

Si bien el esquema general de la Fig. 10.3 permanece sin cambios para esta política, algunos de los procesos deben cambiar su definición de forma significativa. Los procesos de usuario deben generar recepción de mensajes para recibir mensajes desde sus buzones. Esta generación debe añadirse en el proceso *usrsim* y debe ser manejada por el proceso *msgman*. Además, el proceso *msgman* tiene que mantener los buzones, es decir recibir y enviar mensajes desde y hacia los buzones. Esta política puede pensarse como si los procesos de usuario enviaran y recibieran mensajes hacia y desde sus buzones, y no hacia y desde otros procesos de usuario, y esta analogía hace que los cambios sean más fáciles de comprender. Los buzones que se mantienen en cada proceso *msgman* no se cambian durante la ejecución de la aplicación, y son asignados en tiempo de carga. El proceso *tgtsrv* solamente debe conocer en qué procesador está ubicado el buzón de cada proceso de usuario.

Los procesos *usrsim*, y *msgman* son los que tienen los cambios sustanciales en su definición, y los procesos *tgtsrv* y *coord* son los que se adaptan con menos cambios de su definición para esta política.

El proceso *usrsim*, como ya se ha explicado, debe generar recepción de mensajes además de los envíos que se han definido previamente. Siempre se debe requerir del proceso *consrv* el otro proceso de usuario a comunicar, sea para recepción o envío de mensajes, por lo tanto la relación con el proceso *consrv* no cambia. Todos los requerimientos de mensajes deben ser manejados por el proceso *msgman* y se deben seguir recibiendo las descripciones de los procesos de usuario desde el proceso *coord*. Por lo tanto, el funcionamiento general del proceso *usrsim* no cambia sustancialmente, sólo se agrega la generación de recepciones de mensajes por parte de los procesos de usuario.

El proceso *msgman* debe cambiarse para manejar las peticiones de envíos y ecepciones de mensajes y también para mantener los buzones de procesos de usuario. No hay cambios sustanciales para manejar los envíos de mensajes, excepto que se envía al procesador donde se mantiene el buzón del proceso de usuario destino y no al proceso mismo. Las peticiones de recepción de mensajes generan un mensaje de control entre procesos *msgman*: desde el que recibe la petición del usuario hacia el que maneja el buzón del proceso de usuario del cual se quiere recibir. La respuesta a estos mensajes de control es el envío de un mensaje de datos al proceso *msgman* que lo requiere. Este mensaje de control y la respuesta no son necesarios si el usuario que realiza la petición de recepción está asignado al procesador donde se mantiene su buzón.

El proceso *tgtsrv* ya no necesita información sobre la ubicación de los procesos de usuario en los procesadores, solamente necesita conocer dónde (en qué procesador) se maneja el buzón de cada proceso de usuario. Esta información se inicializa durante la carga de la aplicación y no se cambia dinámicamente en tiempo de ejecución.

No es necesario que el proceso *coord* comunique al proceso *tgtsrv* los cambios de asignación de procesos de usuario. Por lo tanto, el proceso *coord* no envía mensajes al proceso *tgtsrv* cada vez que se produce una migración y el canal que comunica a estos dos procesos (Fig. 10.3) se debe eliminar porque ya no es necesario.

10.5.5 Message Rejection

Esta política maneja los mensajes que llegan a un procesador erróneo enviando un mensaje de no reconocimiento NACK (mensaje de control para rechazar un mensaje de datos) al procesador desde el que se envió el mensaje. Los mensajes de control para rechazar mensajes de datos se envían entre procesos *msgman* y utilizan las conexiones entre los procesos que se muestran en el diseño general, correspondiente a la Fig. 10.3. No hay cambios en la estructura, sino en el comportamiento del proceso *msgman*. Todos los demás procesos no se alteran así como tampoco se alteran las conexiones entre los procesos.

El proceso *msgman* genera y maneja los mensajes de control para rechazar mensajes de usuario. La generación se produce cuando el proceso *msgman* recibe un mensaje destinado a un proceso de usuario que no está asignado al procesador local. En el mensaje de rechazo se incluye la nueva ubicación conocida del proceso de usuario, es decir hacia qué procesador migró desde el procesador local. Cuando un proceso *msgman* recibe un mensaje de rechazo, envía un pedido de actualización al proceso *tgtsrv* y reenvía el mensaje al nuevo procesador.

El proceso *tgtsrv* no necesita identificar el origen de las peticiones de actualización de procesos de usuario ni responder a tales peticiones, y por lo tanto no es necesario ningún cambio. Estos pasos se repiten hasta que el mensaje llega al procesador a donde está asignado el proceso de usuario destino.

10.5.6 Migration Protocol

Esta política es la más difícil de implementar bajo el esquema general de la Fig. 10.3, y puede ser vista como un protocolo de migración. Se deben llevar a cabo varios pasos desde que un

proceso de usuario es elegido para migrar hasta que es enviado a otro procesador. Con esto en mente, dos procesos fueron agregados en cada procesador para manejar la migración, los procesos *protmanout* (manejador de salida de información del protocolo) y *protmanin* (manejador de entrada de información del protocolo).

En términos generales, el proceso *protmanout* recibirá los requerimientos de migración del proceso *coord* y se comunicará con cada proceso *protmanin* en los otros procesadores. Cada proceso *protmanin* responderá con la información necesaria para llevar a cabo la migración. El proceso *protmanout* debe recibir de todos los procesos *protmanin* la cantidad de mensajes enviados al proceso de usuario que se migra. La suma de todas las cantidades recibidas se envía al proceso *protmanin* local. Cuando un proceso de usuario llega al procesador destino, el proceso *protmanout* genera todas las notificaciones a los procesos *protmanin* para que actualicen la información de ubicación del proceso de usuario.

La estructura general de procesos para esta política es aumentada con respecto a la que se muestra en la Fig. 10.3, como se muestra en la Fig. 10.7.

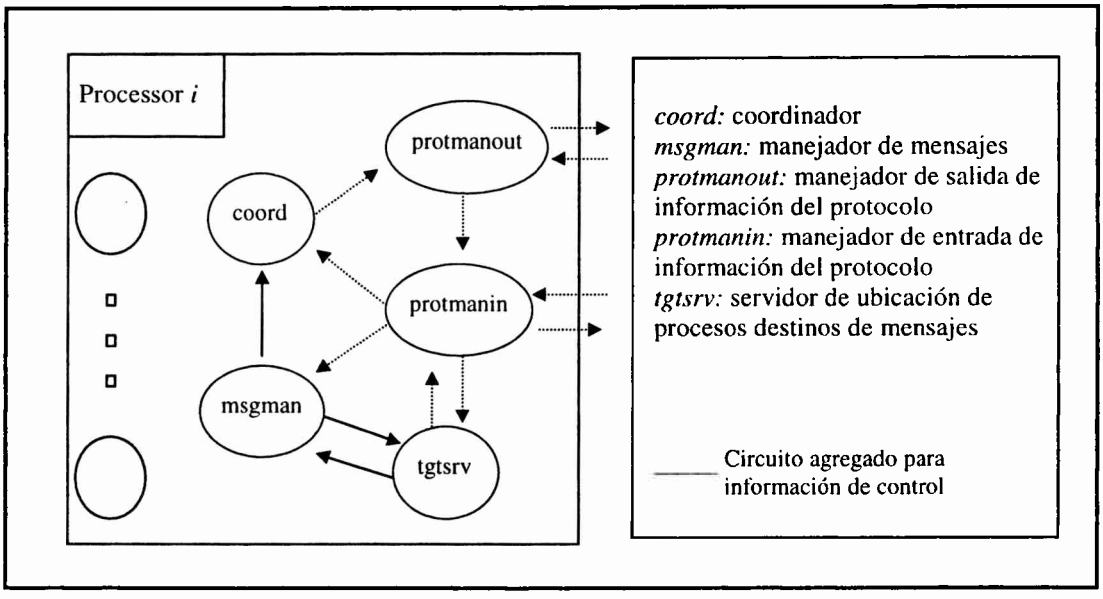


Figura 10.7: Diseño de la Política Migration Protocol.

El proceso *protmanout* solamente interviene en la migración de procesos de usuario que están asignados al procesador local y que se deben enviar a otro procesador. El proceso *protmanin* interviene en todas las migraciones de procesos de usuario. Para las migraciones de procesos de usuario que están en otros procesadores, el proceso *protmanin* enviará a los procesos *protmanout* (cuando éstos lo requieran) la cantidad de mensajes enviados desde el procesador local al proceso de usuario que migra. Para la migración de procesos de usuario que se ejecutan en el mismo procesador, el proceso *protmanin* enviará al proceso *coord* ejecutándose en el mismo procesador la cantidad de mensajes que el proceso de usuario debe recibir antes de ser migrado. También el proceso *protmanin* será el encargado de notificar la finalización de la migración de cada proceso de usuario a los procesos *tgtsrv* y *msgman*.

El proceso *tgtsrv* será el encargado de contabilizar la cantidad de requerimientos de mensajes para los procesos de usuario. Se debe conocer la cantidad de mensajes que se han

enviado a, o recibido de, los procesos de usuario que están asignados a otros procesadores, para poder proporcionar esta información al proceso *protmanin*.

El proceso *msgman* dejará suspendidos a los procesos de usuario si intentan enviar mensajes a otros procesos de usuario que están migrando. Estos mensajes se enviarán cuando se reciba la notificación de finalización de migración del proceso de usuario desde el proceso *protmanin* local.

Como para la política Home Processor, se agrega un circuito más de información de control, que es el manejado principalmente por los procesos *protmanin* y *protmanout*. Cada proceso *protmanout* debe estar conectado a todos los procesos *protmain* asignados a cada procesador. De la misma forma, cada proceso *protmanin* debe estar conectado a todos los procesos *protmanout*.

Dada la complejidad del protocolo con respecto a las demás políticas propuestas para resolver el Problema de Arribo de Mensajes, se hace necesaria una descripción un poco más exhaustiva de la tarea que realizan los procesos que se muestran en la Fig. 10.7. Los párrafos que siguen describen el comportamiento de cada uno de esos procesos con respecto a la migración de procesos de usuario.

Los pasos que se llevan a cabo en el proceso ***coord*** para la migración de un proceso de usuario a otro procesador son:

- Enviar una petición de "inicio de migración" del proceso de usuario al proceso *protmanout*.
- Recibir del proceso *protmanin* la cantidad de mensajes que el proceso de usuario debe recibir antes de ser migrado.
- Recibir desde el proceso *msgman* la notificación de cada uno de los mensajes pendientes (si los hay) del proceso de usuario.
- Enviar el proceso de usuario a otro procesador (corresponde a la migración física del proceso de usuario).

Cuando un proceso de usuario se recibe desde otro procesador, en principio lo el único necesario paso a seguir en el proceso ***coord*** es:

- enviar una petición de "fin de migración" del proceso de usuario que se ha recibido al proceso *protmanout*.

Los pasos que se llevan a cabo en el proceso ***protmanout*** cuando se recibe desde el proceso *coord* un "inicio de migración" de un proceso de usuario son:

- Enviar a todos los procesos *protmanin* el "inicio de migración" del proceso de usuario para el comienzo de intercambio de información.
- Recibir desde todos los procesos *protmanin* asignados a otros procesadores la cantidad de mensajes que fueron enviados al proceso de usuario.
- Enviar al proceso *protmanin* local la suma de mensajes que se han enviado al proceso de usuario desde otros procesadores.

Cuando se recibe desde el proceso *coord* un "fin de migración" de un proceso de usuario en el proceso ***protmanout*** el único paso a seguir consiste en

- Enviar a todos los procesos *protmanin* el "fin de migración" con la identificación del procesador local como la nueva ubicación del proceso de usuario.

Los pasos que se llevan a cabo en el proceso ***protmanin*** cuando se recibe un "inicio de migración" de un proceso de usuario son:

- Enviar una actualización de estado del proceso de usuario al proceso *tgtsrv*, para que éste pueda notificar al proceso *msgman* si es posible para el proceso de usuario seguir recibiendo mensajes.
- Recibir desde el proceso *tgtsrv* la cantidad de mensajes asociados al proceso de usuario.

Tanto la cantidad de mensajes asociados al proceso de usuario, como los pasos que se realizan a continuación en el proceso *protmanin* se discriminan según la ubicación del proceso de usuario que se está migrando. Si el proceso de usuario es local, la cantidad de mensajes asociados a él recibida desde el proceso *tgtsrv* es la cantidad de mensajes ya recibidos por el proceso de usuario, y los pasos a seguir son:

- Recibir desde el proceso *protmanout* local la cantidad de mensajes enviados al proceso de usuario desde los demás procesadores.
- Enviar al proceso *coord* la cantidad de mensajes que le falta recibir al proceso de usuario hasta que se pueda migrar.

Si el proceso de usuario que se va a migrar no es local, la cantidad de mensajes asociados a él recibida desde el proceso *tgtsrv* es la cantidad de mensajes que se le han enviado desde el procesador local al proceso de usuario, y el paso a seguir es:

- Enviar al proceso *protmanout* que envió el "inicio de migración" esta cantidad de mensajes que se le han enviado al proceso de usuario desde el procesador local

Los pasos que se llevan a cabo en el proceso ***protmanin*** cuando se recibe un "fin de migración" de un proceso de usuario son:

- Enviar una actualización de estado del proceso de usuario al proceso *tgtsrv*, con la nueva ubicación del proceso de usuario.
- Enviar al proceso *msgman* la notificación de que el proceso de usuario ya ha sido migrado, para que éste le pueda enviar los mensajes que tiene suspendidos.

El proceso ***tgtsrv*** debe llevar a cabo las siguientes tareas para manejar el protocolo de migración de procesos de usuario:

- Contar la cantidad de requerimientos de mensajes para cada proceso de usuario (mensajes asociados al proceso de usuario). Si el proceso de usuario está asignado al procesador local, debe contar los requerimientos de los mensajes que le llegan. Si el proceso de usuario no es local, debe contar los requerimientos de mensajes que se le envían.
- Cuando recibe del proceso *protmanin* una actualización de estado que no indique la nueva ubicación del proceso de usuario, debe (a) registrar el estado del proceso de usuario migrando, y (b) enviar al proceso *protmanin* la cantidad de mensajes asociados al proceso de usuario.
- Inicializar el contador asociado al proceso de usuario cuando recibe una nueva ubicación para él desde el proceso *protmanin*.
- Contestar a cada petición del proceso *msgman* sobre la ubicación de un proceso de usuario si el proceso de usuario está en proceso de migración o no (además de dar la ubicación del proceso de usuario si no está migrando, como en el caso general).

El proceso ***msgman*** debe llevar a cabo las siguientes tareas para manejar el protocolo de migración de procesos de usuario:

- Indicar el proceso de usuario que genera un mensaje en las peticiones al proceso *tgtsrv* sobre la ubicación del proceso destino del mensaje.
- Suspender los mensajes que se le envíen a un proceso de usuario que está migrando.

- Recibir las notificaciones del proceso *protmanin* sobre la finalización de migración de procesos de usuario, y enviarles los mensajes que estén suspendidos.
- Notificar al proceso *coord* cuando llegan mensajes para un proceso de usuario local que está migrando.

Todos los demás procesos que se corresponden con la Fig. 10.3 permanecen con la misma definición e interconexiones que fueron dadas en la descripción general del diseño del entorno para experimentación de migración en DMPCs.

10.6 Instrumentación y Control de Interferencias

Con el fin de evitar que la monitorización interfiera con la aplicación de usuario, en cada procesador se toman solamente medidas locales. Las medidas locales se realizan de tal forma que no necesiten ninguna comunicación con procesos en otros procesadores ni en el procesador local. Esto significa que no utilizan la red de comunicaciones ni se necesita sincronización entre procesos. De esta manera no se retrasa de forma.considerable el procesamiento necesario para que la aplicación de usuario que se está monitorizando pueda continuar su ejecución.

Los promedios de cada medida no se toman localmente porque no se tiene toda la información necesaria para hacerlo. Al final de la simulación de la aplicación de usuario se envía lo calculado en cada procesador a un proceso asignado al procesador que realiza la entrada/salida (procesador 0) de la red de procesadores. Este proceso es el que se encarga de calcular los promedios, dado que recibe la información de toda la ejecución. En ningún momento este proceso de recepción de datos interfiere con la aplicación, porque está asignado a un procesador que solamente se utiliza para la entrada/salida de toda la ejecución de procesos de usuario. De esta manera se intenta que las métricas y cálculos necesarios para analizar resultados interfieran lo menos posible con la aplicación de usuario que se está monitorizando.

Al esquema general de la Fig. 10.3 no se agrega ningún proceso, sino que algunos de ellos serán los encargados de monitorizar lo que sucede en la ejecución de los procesos de usuario. Sí se agrega un proceso en el procesador 0, el proceso ***datacol*** (proceso receptor de las mediciones de monitorización de la aplicación de usuario), que está destinado principalmente a recibir las mediciones hechas localmente en cada procesador y calcular los promedios de cada una de ellas para toda la aplicación. También realiza la entrada/salida necesaria para leer los valores de los parámetros de la aplicación de usuario y para escribir los resultados.

En el procesador 0 no se realiza ninguna tarea propia de la aplicación de usuario. No hay procesos de usuario asignados ni ningún proceso de manejo de mensajes. Se utiliza solamente para cargar la aplicación de usuario en la red de procesadores (como se detalla en la sección siguiente) y recibir, calcular y escribir en almacenamiento permanente los resultados de la ejecución de la aplicación. Estos resultados de la aplicación son los que se utilizarán para calcular los índices de evaluación, y analizar y comparar las políticas propuestas para resolver el Problema de Arribo de Mensajes. Es de destacar que esta estructura general de carga y recepción de datos en el procesador 0 de esta máquina paralela

también se puede utilizar para calcular y evaluar otros aspectos que se quieran incorporar en la ejecución de aplicaciones de usuario.

10.7 Procesos de Carga de la Aplicación de Usuario

Es necesario un mecanismo para cargar la aplicación de usuario en la máquina paralela. Con el fin de evitar interferencias, el procesador 0 es el que se utiliza para ejecutar el proceso de carga global de toda la aplicación de usuario, aunque por no disponer de un mecanismo de *broadcast* o *multicast* de datos sigue siendo necesario un proceso de carga local en cada procesador. Con *broadcast* o *multicast* se podría, desde el proceso cargador global, enviar la información de carga de la aplicación a cada uno de los procesos de cada procesador con el mismo mensaje. Por ejemplo, desde el proceso cargador global se podría enviar información a los procesos *coord* de todos los procesadores con un solo mensaje.

Esquemáticamente, los procesos de carga global y local, recolección y cálculo de cada medida local se pueden describir como en la Fig. 10.8 para el procesador 0 y cualquier otro procesador *i* de la red de procesadores. En proceso *gloader* realizaría la tarea de broadcast entre los distintos procesadores, y el proceso *lloader* realizaría la tarea de broadcast entre los distinto procesos de un mismo procesador.

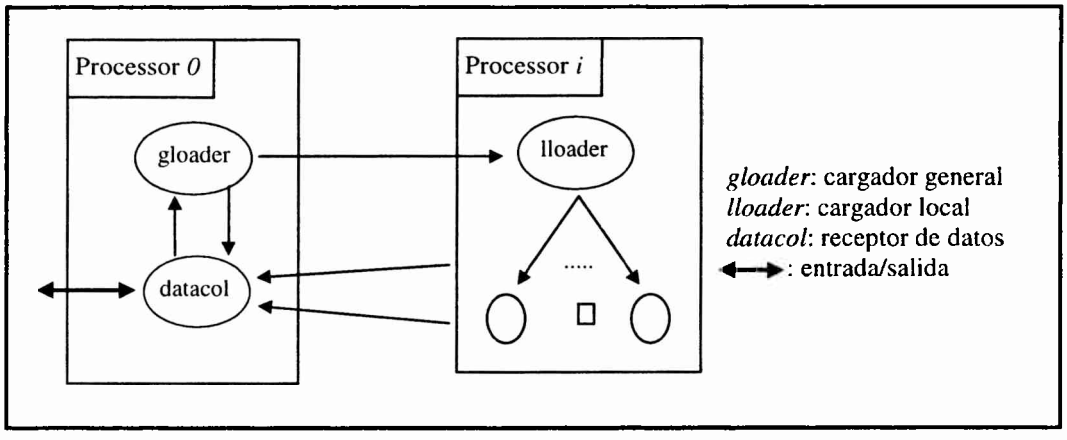


Figura 10.8: Procesos de Carga y Recepción de Datos.

Se debe aclarar que aunque la máquina paralela no disponga de un procesador que se encargue de la entrada/salida (tal como el procesador 0 que se ha mencionado), la carga de procesamiento y de comunicaciones que se manejan en este procesador en general no interfieren con la aplicación de usuario. El proceso de carga global (*gloader*), tiene su tarea circunscripta al tiempo de carga de la aplicación, por lo tanto la aplicación no está ejecutándose aún y por esta razón no hay interferencia posible. El proceso *datacol* tiene su mayor parte de tiempo de ejecución cuando la aplicación finaliza. Puede considerarse el caso en que no todos los procesadores hayan finalizado la ejecución de todos los procesos de usuario, pero esto no ocurre sino al final de la aplicación y la interferencia es mínima.

El proceso ***gloader*** (proceso cargador general de la aplicación de usuario), se encarga de enviar la información necesaria para que cada procesador pueda ejecutar la aplicación de usuario. La información es enviada a los procesos cargadores de la aplicación locales de cada procesador. Los mensajes que se envían se refieren principalmente a la cantidad de procesos

de usuario, comportamiento de cada proceso de usuario y tasa de migración. El proceso *gloader* recibe del proceso *datacol* los datos que necesita del sistema de entrada/salida.

El proceso ***lloader*** (proceso cargador local de la aplicación de usuario) en cada procesador, distribuye localmente la información de los procesos de usuario entre los procesos que la necesitan. Por ejemplo, envía la cantidad, identificación y descripción de comportamiento de los procesos de usuario al proceso *coord* local y la identificación de los procesos de usuario asignados inicialmente al procesador local. Esta información que distribuye es recibida desde el proceso *gloader*.

El proceso ***datacol*** (proceso receptor de las mediciones de monitorización de la aplicación de usuario), está destinado principalmente a recibir las mediciones hechas localmente en cada procesador y calcular los promedios de cada una de ellas para toda la aplicación. También realiza la entrada/salida necesaria para leer los valores de los parámetros de la aplicación de usuario y para escribir los resultados.

Los procesos que realizan la carga de la aplicación de usuario (*gloader - lloader*) son los que se verán afectados por el cambio o agregado de parámetros en la descripción de las aplicaciones de usuario o en el funcionamiento de los procesos de gestión de migraciones. Las aplicaciones de usuario se podrían definir utilizando más parámetros, por ejemplo para incorporar la creación dinámica de procesos o de canales de comunicación entre los procesos. En cuanto a los parámetros para los procesos del entorno para experimentación de migración enb DMPCs, se pueden incorporar parámetros para, por ejemplo,

- definición de las políticas de migración,
- definición de los métodos de migración física,
- control del comportamientode las políticas que gestionan la consistencia en el arribo de mensajes.

Cabe recordar en este contexto que a mayor cantidad de posibilidades de variación del entorno para experimentación la complejidad aumenta de forma considerable.

10.8 Descripción de la Experimentación y de los Indices de Evaluación

Toda la experimentación con aplicaciones de usuario se ha realizado con el fin de evaluar y comparar las políticas presentadas para solucionar el Problema de Arribo de Mensajes. En todos los casos es seguro que los mecanismos para migrar procesos de usuario y resolver el Problema de Arribo de Mensajes producirán una sobrecarga en la red de comunicaciones y en los procesadores disponibles.

Uno de los fenómenos más importantes a estudiar con el entorno para experimentación de migración en DMPCs, es la sobrecarga que se produce en la red de comunicaciones. Esta sobrecarga es el resultado de los mensajes que se agregan para transmitir los propios procesos de usuario que migran y para que estos procesos que migran sigan recibiendo los mensajes como si su ubicación en la red de procesadores no cambiara dinámicamente. Los procesadores se sobrecargan porque en cada uno de ellos se deben agregar los procesos mencionados en el diseño del entorno de experimentación, y cada uno de ellos consume tiempo de CPU para

realizar su tarea. El interés está puesto en identificar (si existe) la política que produce menor degradación en la ejecución de las aplicaciones de usuario.

Como se ha explicado previamente, las aplicaciones de usuario se definen en términos de los requerimientos de cómputo y de comunicación de los procesos de usuario. Si bien en el entorno para experimentación es posible hacer que cada proceso de usuario se defina en forma distinta de los demás, se decidió continuar con lo establecido en [Hey95], que todos los procesos de usuario tengan los mismos requerimientos de cómputo y de comunicación y con conectividad completa: todos los procesos de usuario están conectados entre sí. Cabe aclarar en este punto que el patrón de interconexión entre procesos de usuario puede establecerse de forma paramétrica en el proceso *consrv*.

Como lo que se estableció en [Hey95], el trabajo límite de cada proceso de usuario se fijó en el envío de 150 mensajes. Por razones de simplicidad se asume que no hay creación dinámica de canales ni de procesos, todos los procesos de usuario que se quieran simular se definen estáticamente con todos los canales que utilizarán.

En cuanto a la política de planeamiento de la ejecución (*scheduling*) llevada a cabo por el proceso *coord* en cada procesador, se eligió la política de asignación circular de CPU (*Round-Robin*, [Sil94]). Esta elección se debe a que

- Es una política apropiativa, no deja que los procesos de usuario ocupen indefinidamente la CPU.
- Todos los procesos de usuario tienen los mismos requerimientos de cómputo y comunicaciones, y por lo tanto no es necesario complicar la planificación de la ejecución.
- Es la política que consume menor cantidad de recursos (principalmente CPU) para las aplicaciones de usuario definidas.

Siempre es posible el cambio de política de planeamiento de ejecución [Cas88] dentro del proceso *coord* del entorno para experimentación.

En el caso de la política Global Server, es necesario decidir dónde ubicar al proceso Servidor Global (*glbsrv*), que es el que proporciona la ubicación (procesador) de cada proceso de usuario destino de un mensaje. Para poder comparar los resultados con los obtenidos en por simulación en [Hey95], se decidió asignarlo a un procesador distinto de los que ejecutan procesos de usuario. El procesador 0, que solamente contiene a los procesos *gloader* (carga global) y *datacol* (recolección de datos), es el más indicado para que se le asigne también el proceso *glbsrv*. Se hicieron algunas pruebas preliminares, asignando el proceso *glbsrv* a otro procesador (de los que ejecuta también procesos de usuario) y el desbalance de carga que se produce es notable.

10.8.1 Experimentación: Relación con Parámetros de Diseño

Los parámetros que se variaron en la experimentación con el fin de evaluar las políticas presentadas para resolver el Problema de Arribo de Mensajes fueron:

- Cantidad de procesadores.
- Cantidad de procesos de usuario por procesador.
- Tasa de migración de los procesos de usuario, definida en función de la probabilidad de migración por mensaje generado por la aplicación. El valor 0.0 se utiliza como referencia y

representa el caso en el que se soportan los mecanismos de migraciones y consistencia en el arribo de mensajes, pero los procesos de usuario no migran.

Todas las políticas fueron analizadas con distintos valores de cada uno de los parámetros. En cada caso (combinación de valores elegidos), se ejecuta la aplicación sintética de usuario y se monitoriza la ejecución.

Se analiza el comportamiento de las políticas a medida que se utilizan más procesadores con el fin de estudiar la *escalabilidad* de cada política con respecto a la cantidad de procesadores. Como en cualquier aplicación paralela en un DMPC, se espera que al utilizar más procesadores la aplicación de usuario se ejecute más rápidamente. Es de esperar que al tener que manejar más información y más alternativas posibles para tomar decisiones, las políticas degraden la ejecución de la aplicación paralela cuanto mayor sea la cantidad de procesadores que se utilicen. Esta degradación puede estar dada por la cantidad de procesamiento (tiempo de CPU) y/o por la comunicación que requieran para llevar a cabo su tarea. Si una política degrada los tiempos de ejecución en proporción mayor, o peor aún, la degradación es no lineal con respecto a la cantidad de procesadores agregados, se llegará a un punto en el que agregar procesadores no mejorará la velocidad de ejecución de la aplicación de usuario o la degradará. Si las políticas no son escalables se puede perder la ventaja de replicar la arquitectura de un DMPC para lograr mejor rendimiento.

Se analiza el comportamiento de las políticas aumentando la cantidad de procesos de usuario por procesador para estudiar el comportamiento de cada política con respecto a la carga de la máquina paralela. En este caso la idea es descubrir si la política tiene un comportamiento dependiente de la cantidad de procesos de usuario que se ejecuten por procesador.

La tasa de migración varía en la experimentación para analizar cómo se comportan las políticas con respecto a la frecuencia de migración de los procesos de usuario. También con la variación de este parámetro se pueden descubrir las políticas que se comporten mejor o peor en problemas de aplicaciones con distintas características en cuanto a la frecuencia de migración de los procesos de usuario.

La asignación inicial de procesos a procesadores se realizó en todos los casos con la misma cantidad de procesos para todos los procesadores. Todos los procesos de usuario tienen los mismos requerimientos de cómputo y de comunicaciones. Los procesos de usuario están interconectados por un grafo completo, es decir, todos los procesos de usuario están conectados dos a dos.

10.8.2 Indices de Evaluación

Los índices que se consideraron más importantes para evaluar y comparar las políticas propuestas para resolver el Problema de Arribo de Mensajes son:

- *Latencia* de los mensajes, es decir el tiempo promedio que utiliza un mensaje desde que es enviado desde el proceso de usuario que lo origina hasta que llega al proceso de usuario destino.
- *Carga* sobre la red de comunicaciones que añade cada política. Esta carga está dada por la cantidad promedio de *Retransmisiones* de mensajes más la cantidad promedio de *Mensajes*

de control que se transmitieron por mensaje entre procesos de usuario.

- *Tiempo de reacción* de los procesos de usuario, definido como el tiempo promedio que transcurre desde que un proceso de usuario es elegido para migrar hasta que efectivamente es migrado.
- *Modificación del patrón de comunicaciones*, es decir, si una vez que se soportan las migraciones entre procesos, los mensajes de un proceso de usuario a otro, circulan por el mismo camino que en la aplicación sin que se realicen migraciones. Aunque la red de comunicaciones de la máquina paralela se utilice virtualizada [Luq95a] y los caminos por los cuales un mensaje se transfiera físicamente no se conozcan, sí se puede saber si un mensaje pasa por los mismos procesadores desde el proceso de usuario que lo emite hasta el que lo recibe una vez que los procesos se han migrado.
- *Escalabilidad*, o sea el comportamiento de las políticas cuando se añaden procesos y procesadores.

Se agregaron algunos otros indicadores con el fin de monitorizar situaciones que pudieran generar una mala interpretación de resultados. Uno de los índices más importantes que fue agregado es la cantidad de migraciones de procesos de usuario que se producen. La razón principal para monitorizar la cantidad de migraciones consiste en verificar que en todas las políticas se tiene aproximadamente la misma carga de la red de comunicaciones por la transmisión de procesos de usuario y también que los procesadores reciben en promedio la misma carga de procesamiento a lo largo de cada ejecución.

La latencia de los mensajes es quizás el primero de los índices de evaluación en el que se puede pensar, porque es el que se nota más inmediatamente en los procesos de usuario. También es uno de los índices que aporta más información unificada, dado que todos los demás índices que se han escogido tienen impacto directo o indirecto en la latencia de los mensajes. Por ejemplo, si una política carga demasiado la red de comunicaciones con mensajes de control o con retransmisiones de mensajes, la latencia de los mensajes aumentará, porque la red de comunicaciones no podrá atender de forma inmediata a los requerimientos, hasta no resolver los requerimientos previos.

La cantidad de retransmisiones así como la cantidad de mensajes de control que produzca cada política indican los requerimientos de comunicaciones. La cantidad de retransmisiones y de mensajes de control constituyen la parte del ancho de banda de la red de comunicaciones que utiliza cada política.

Es necesario prestar atención al tiempo de reacción de los procesos de usuario porque puede tener interferencia con la política de migraciones. Cuando se integre/n la/s política/s para aplicaciones reales, deben implementarse ambos aspectos: la consistencia en el arribo de mensajes y la política de migraciones. La política de migración debería tener en cuenta el tiempo de reacción para no asumir que se ha migrado un proceso de usuario (cambiando el estado de asignación de procesos de usuario a procesadores) antes de que la migración real se haya producido. Por otro lado, es deseable que la política que se utilice para conservar la consistencia en el arribo de los mensajes tenga el menor tiempo de reacción posible para no interferir en este sentido con la política de migraciones.

La forma en que se modifica o no el patrón de comunicaciones en la red de comunicaciones es importante para analizar si es posible lograr el balance de las comunicaciones. Si bien los procesos de usuario se comunican entre sí de forma

independiente de la forma en que se resuelve el Problema de Arribo de Mensajes, la cantidad de mensajes que circulan entre los procesadores depende de cada política. El objetivo para lograr el balance de comunicaciones es que todos los recursos de la red de comunicaciones se utilicen en forma similar, sin sobrecarga de algunas partes y baja utilización de otras. La sobrecarga de algunas partes de la red de comunicaciones (*hotspot*) suele ser el mayor problema dado que su efecto se expande rápidamente a toda la red y termina por congestionarla [Sto93].

11. Entorno para Experimentación en DMPCs: Detalles de Implementación

En este capítulo se describirán algunos puntos relevantes que tienen relación con la implementación paralela de un entorno de experimentación de migración de procesos, tales como:

1. el entorno de programación de la máquina paralela,
2. el control de deadlocks entre los procesos de la implementación paralela,
3. la explicación de los procesos más relevantes de la aplicación, con mayor nivel de detalle,
4. cómo se realiza la monitorización y el cálculo de los índices de evaluación.

Con respecto al último punto de la enumeración anterior, se debe tener en cuenta que el objetivo inicial del entorno para experimentación de migración en DMPCs es el de probar y comparar cada política de gestión de mensajes. Por esta razón se debe monitorizar lo que sucede durante la ejecución de la aplicación de usuario para mantener la consistencia en el arribo de los mensajes, y analizar los valores obtenidos para cada uno de los índices propuestos. Para cada índice de rendimiento, se debe conocer la forma de monitorizar la ejecución y, si el índice no se puede monitorizar de manera directa, se debe conocer qué otros índices monitorizar para calcularlo.

11.1 Entorno de Programación

La máquina paralela que utilizada en la implementación es una red de 33 transputers, donde uno de ellos se utiliza casi con exclusividad para la entrada/salida de toda la red. Varias características fueron agregadas al entorno de programación original [Luq95], las más importantes se explican a continuación. La máquina paralela (principalmente los recursos de la red de comunicaciones) es utilizada por medio de TransComm [Luq95].

La red de comunicaciones entre transputers se utilizó virtualizada, todos los procesadores se consideran comunicados entre sí. El lenguaje utilizado para programar es básicamente C [Ker91], con la posibilidad de definir procesos que se comunican a través de canales, similar al lenguaje C desarrollado por INMOS [INM92a] [INM92b] [INM92c]. Los canales de comunicación se definen en la cabacera de los procesos y tienen longitud fija, es decir: la cantidad de información que se transmite utilizando un canal es siempre la misma para todas las comunicaciones. Las primitivas de comunicación disponibles utilizando TransComm son:

- send(canal_de_salida, área_de_memoria): para enviar información contenida en la región de memoria área_de_memoria indicada por el canal de salida canal_de_salida del proceso. La cantidad de datos es la definida en la declaración del canal de salida. La comunicación es bloqueante, el proceso se suspenderá hasta que los datos hayan salido hacia el proceso receptor de la información.
- recv(canal_de_entrada, área_de_memoria): para recibir información por el canal de entrada canal_de_entrada, que se almacenará en la región de memoria área_de_memoria. La cantidad de datos es la definida en la declaración del canal de entrada. La comunicación es bloqueante, el proceso se suspenderá hasta que los datos estén disponibles en la memoria del procesador.

- nb_recv(canal_de_entrada, área_de_memoria): es similar a recv(...), excepto que la comunicación es no bloqueante y la función retorna la cantidad de datos que se recibieron. Si el proceso que debe enviar los datos no está listo, y por lo tanto la comunicación no se lleva a cabo, la función retorna 0.

No es posible tener estructura de datos que contengan canales ni está disponible una función del tipo ProcAlt de C para este tipo de máquinas paralelas [INM92b] donde se indica una lista de canales y el proceso se bloquea hasta recibir por cualquiera de ellos. Como en el lenguaje C desarrollado por INMOS, se pueden definir canales *1* a *1*; pero a diferencia de éste se agregó en TransComm la posibilidad de definir canales *1* a *N* y *N* a *1*. Los canales *N* a *1* fueron utilizados para realizar la tarea de la función del tipo ProcAlt. La Fig. 11.1 muestra esquemáticamente un canal *N* a *1*, cada vez que uno de los procesos emisores (p_1, ..., p_N, en la Fig. 11.1) genera un mensaje, este mensaje se transmite al único proceso receptor del canal (p_r en la Fig. 11.1).

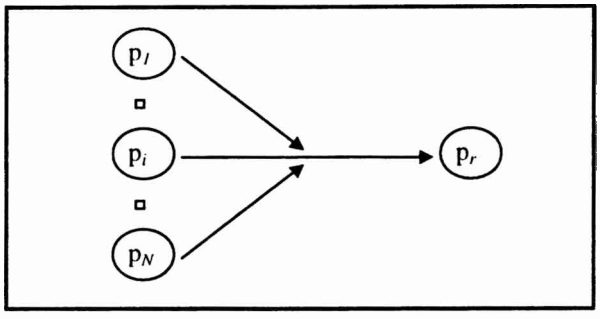


Figura 11.1: Canal *N* a 1.

La virtualización de la red de comunicaciones hace posible que los procesos p_1, ..., p_N, y p_r se puedan asignar a cualquier procesador de la máquina paralela.

Una característica más que fue agregada por TransComm y que se utilizó en la implementación paralela, es la posibilidad de asociar memoria intermedia (*buffers*) para almacenamiento de mensajes de un canal. De esta manera se evita que los dos procesos que se comunican tengan que sincronizarse para realizar la transferencia de datos al mismo tiempo. La Fig. 11.2 muestra esquemáticamente un canal con memoria asociada para el almacenamiento de los mensajes que se transmiten desde un proceso p_e a otro p_r. La cantidad de memoria se especifica en cantidad de mensajes que se quieran almacenar.

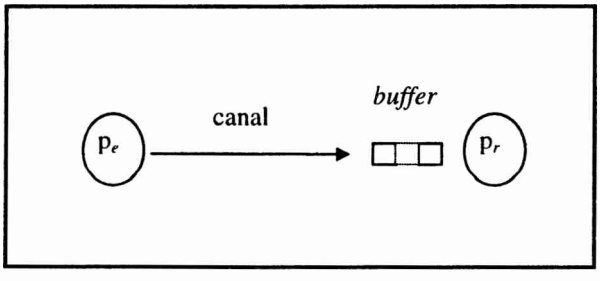


Figura 11.2: Canal con *Buffer*.

En este contexto de implementación y programación de la máquina paralela, el hecho de simular procesos de usuario y no mantener una aplicación real evita entrar en detalles

como la planificación de la ejecución (*scheduling*) de procesos y asignación de espacios de memoria para los procesos en los transputers [INM88a] [INM88b] [Tyr89].

11.2 Control de Deadlocks

Según la descripción realizada de la estructura general del diseño del entorno para experimentación de migración en DMPCs, y dado que se utilizan primitivas de comunicación esencialmente bloqueantes, hay al menos dos fuentes claras de deadlocks entre los procesos del entorno. En ambos casos lo más fácil de observar son los ciclos del grafo de comunicaciones entre procesos de la Fig. 6.3, que pueden producir una espera circular [Tan88] [Sil94]. El control de deadlocks entre procesos de la aplicación de usuario no tiene sentido dada la definición de las aplicaciones y la forma en que se asume la comunicación entre los procesos.

Una posible fuente de deadlock está dada por la comunicación entre los procesos *coord*, *usrpout* y *usrpin* que forman el circuito de información para migración de procesos de usuario. Si desde un procesador P*i* se intenta migrar un proceso de usuario al procesador P*j* y desde P*j* se está esperando para migrar un proceso de usuario a P*i*, se tiene una espera circular entre los procesos mencionados. Si bien los procesos *usrpout* y *usrpin* actúan a la vez como memoria intermedia (*buffer*) entre los procesos *coord*, cuando se produzcan muchas migraciones, estos procesos y su almacenamiento intermedio no serán suficientes para evitar la espera circular y el deadlock.

La Fig. 11.3 muestra una secuencia de mensajes (migraciones de procesos de usuario) con la cual se produce el deadlock de los procesos *usrpin, coord* y *usrpout* de dos procesadores (P*m* y P*n*). Por razones de simplicidad no se muestran los canales no utilizados entre procesos. Los números que se muestran dentro de un rectángulo (4, 9, 7, 6, 10 y 8) representan los mensajes *bloqueados*, esto implica que los procesos que los envían deben esperar a los procesos receptores. Por ejemplo, el mensaje 4 que se envía desde el proceso *coord* en el procesador P*m* debe esperar a que el proceso *usrpout* finalice la tarea correspondiente a la petición previa (mensaje 3), de envío de un proceso de usuario hacia el procesador P*n*. Aunque los procesos del procesador P*q* intervienen en la secuencia de pasos mostrada, no intervienen en el deadlock, no forman parte de la espera circular.

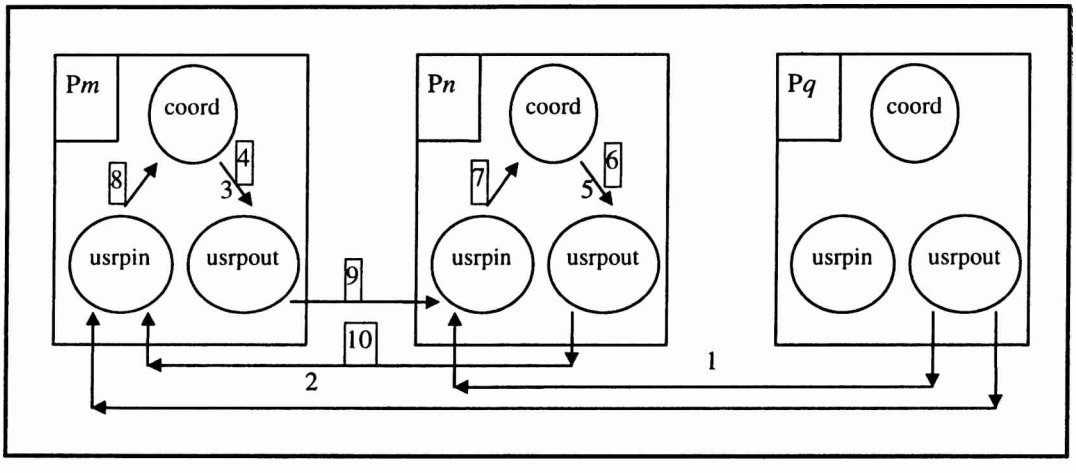


Figura 11.3: Bloqueo por Migraciones de Procesos.

Para evitar este tipo de deadlocks se incluyó un mensaje de control que se envía desde el proceso *usrpout* al proceso *coord*. En la Fig. 11.4 se muestra el canal de comunicación que se agrega desde el proceso *usrpout* hacia proceso *coord*. El mensaje de control indica al proceso *coord* cuándo el proceso *usrpout* está listo para enviar un proceso de usuario a otro procesador.

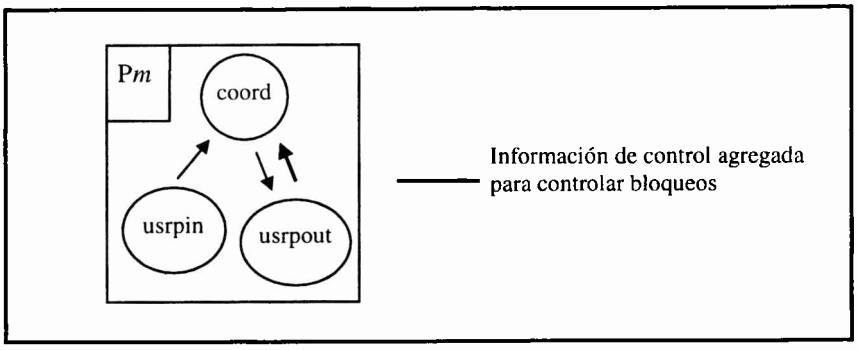


Figura 11.4: Control de Bloqueos por Migración de Procesos.

De esta manera el proceso *coord* nunca esperará para comunicarse con el proceso *usrpout* porque solamente le enviará un proceso de usuario cuando se pueda recibir y por lo tanto se rompe la espera circular y el deadlock. Volviendo al ejemplo de la Fig. 11.3, los mensajes 4 y 6 no se enviarán sino hasta que los procesos *usrpout* de cada procesador notifiquen a los procesos *coord* que pueden seguir recibiendo requerimientos de migración de procesos de usuario.

La otra posible fuente de deadlocks se encuentra de forma similar a la anterior entre los procesos *msgman* y *mout* de distintos procesadores (que forman el circuito de información para datos de mensajes de procesos de usuario). En este caso la probabilidad de deadlocks es mayor porque está dada por la cantidad de mensajes entre los procesos de usuario. En este caso se hizo uso de la memoria que se puede asociar a los canales de comunicación entre procesos (*buffers*). Las bases para la elección de este método de solución fueron:

1. Tal como se define el modelo sintético de la aplicación de usuario, se asume el mejor caso de comunicación entre procesos de usuario: no hay espera por comunicación para ningún proceso de usuario. Esto implica que todos los requerimientos de envío de mensajes que reciba un proceso *msgman* podrán ser recibidos por los procesos de usuario a los cuales van dirigidos, y por lo tanto la memoria no se llenará por los mensajes que no pueden ser recibidos. Todos los requerimientos serán recibidos en los procesadores a los cuales estén asignados los procesos de usuario que son destinos de los mensajes.
2. Se asume que hay memoria suficiente para mantener todos los requerimientos de usuario. Por un lado, todo manejador de mensajes siempre tendrá memoria para contener en espera requerimientos pendientes, y por el otro se intenta evaluar el efecto de las propias políticas que resuelven el Problema de Arribo de Mensajes. Si bien la memoria que requieren podría ser un índice para comparar, no fue de los que se consideraron más importantes y por lo tanto se está asumiendo que siempre hay memoria suficiente.

Con memoria suficiente asociada a los canales que se utilizan para los mensajes de datos de procesos de usuario, el problema de deadlock se resuelve porque los procesos *msgman* nunca esperarán para enviar un requerimiento de mensaje a otro procesador, y se rompe la espera circular. En el caso en que la memoria no fuera suficiente para contener todos

los requerimientos de mensajes, se llega a lo que se podría denominar como *estado inseguro*; se podría producir una espera circular entre los procesos *msgman* y *mout* de distintos procesadores, y en este caso se llegaría al deadlock entre estos procesos.

Una tercera fuente de deadlocks puede darse entre los procesos *coord*, *usrsim* y *msgman* de un mismo procesador. En este caso, el proceso *coord* solamente enviará un proceso de usuario para ser simulado en el proceso *usrsim* si éste está libre. El proceso *coord* recibe la notificación de que el proceso *usrsim* está listo para recibir un nuevo proceso de usuario desde el mismo proceso *usrsim* o desde el proceso *msgman*. El proceso *usrsim* requiere un nuevo proceso de usuario si finalizó de simular el recibido previamente y no generó ninguna comunicación. El proceso *msgman* notifica al proceso *coord* el caso en el que el proceso de usuario que estaba siendo simulado envió un mensaje, y por lo tanto el proceso *usrsim* está libre. Los dos tipos de mensajes están indicados en la Fig. 6.3 por los canales que van desde el proceso *usrsim* al proceso *coord* y desde el proceso *msgman* al proceso *coord* respectivamente.

En la implementación de la política Migration Protocol hay otra posible fuente de deadlocks entre los procesos *coord*, *protmanout* y *protmanin*. El problema se agrava (las probabilidades de que se produzcan bloqueos es mayor que en los casos anteriores) debido a que cada petición desde el proceso *coord* implica que el proceso *protmanout* se comunique al menos una vez con todos los procesos *protmanin* en los demás procesadores. Este caso se resolvió como el que se podía dar entre los procesos *usrpin*, *coord* y *usrpout* que se mencionó anteriormente. Se decide agregar un mensaje desde el proceso *protmanout* para notificar al proceso *coord* que puede enviar otra petición porque ya ha resuelto la anterior. La Fig. 11.5 muestra el canal de comunicaciones que se agrega con respecto a la Fig. 6.7 desde el proceso *protmanout* hacia el proceso *coord.*

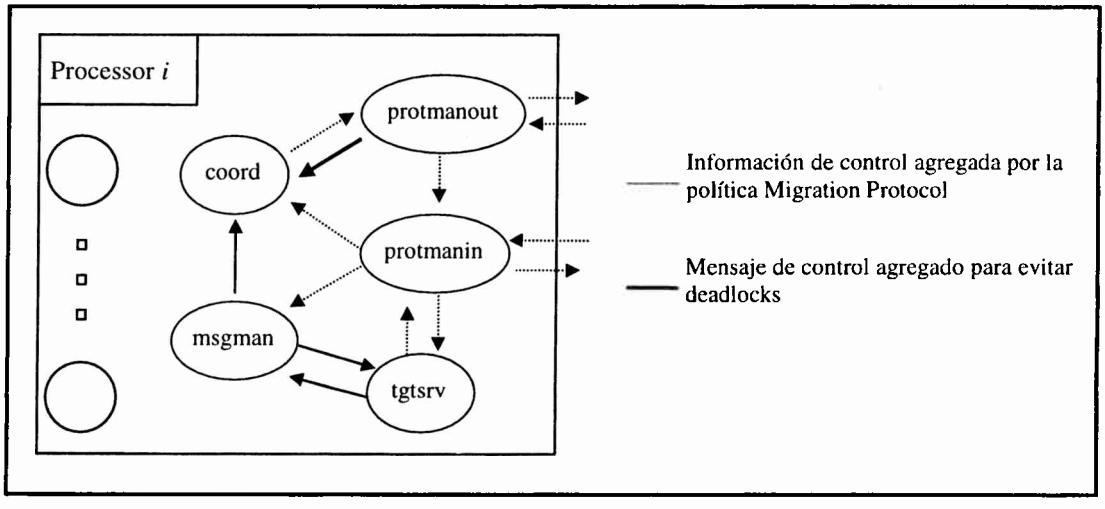


Figura 11.5: Control de Bloqueos en la Política Migration Protocol.

Con esto se vuelve a romper la espera circular, porque el proceso *coord* nunca quedará bloqueado por esperar para comunicarse con el proceso *protmanout*.

De esta manera, cada vez que el proceso *protmanout* finaliza su tarea (sea de un requerimiento de "comienzo de migración" o de un "fin de migración"), debe enviar un mensaje al proceso *coord* para notificar que está libre. Asimismo, el proceso *coord* debe tener

en cuenta no solamente enviar peticiones al proceso *protmanout* cuando esté libre, sino también recibir las notificaciones del proceso y mantener el estado del proceso *protmanout* (ocupado o libre).

Los procesos *coord* y *msgman* fueron los más controlados para evitar deadlocks, dado que ambos tienen una importancia sustancial en la implementación paralela. Toda, o casi toda, la ejecución de aplicaciones de usuario se encuentra controlada por estos dos procesos. En el caso de deadlock, seguramente al menos uno de esos dos procesos estaría formando parte de la espera circular. La idea general que se siguió para evitar deadlocks consiste en que ni el proceso *coord* ni el proceso *msgman* tuvieran que esperar para comunicarse con otro proceso para resolver un requerimiento. El proceso *coord* se encarga de resolver los requerimientos que se refieren a la migración o envío a ejecución de un proceso de usuario. El proceso *msgman* se encarga principalmente de los requerimientos de envío de mensajes. Para estos procesos en particular se intentó seguir la idea de procesamiento cliente-servidor [Tan92], con la particularidad de que para llevar a cabo una determinada tarea, el servidor no depende de ninguna comunicación que pueda mantenerlo bloqueado.

Como alternativa para evitar los bloqueos por mensajes, se podría haber utilizado la primitiva de comunicación no bloqueante, nb_recv(,,,). Con esta primitiva de comunicación se puede controlar que ningún proceso quedara esperando de forma indefinida para llevar a cabo una comunicación. Esta alternativa se descartó por al menos dos razones:

1. Utilizar nb_recv(,,,) implica tener de alguna manera una espera ocupada (busy waiting), es decir que el proceso está esperando para hacer una comunicación pero no libera la CPU. Tener uno o más procesos con esta característica en la implementación paralela puede ser muy problemático, ya que los procesos de usuario, por ejemplo, se verían imposibilitados de ser simulados y el procesador estaría libre sin hacer cómputo *útil*, solamente esperando.
2. Se debería controlar el uso nb_recv(,,,). Hay varias decisiones a tomar que no tienen una respuesta clara: cantidad de intentos, cuándo volver a intentar en caso de no haber llevado a cabo la recepción del mensaje, qué hacer después de intentar sin éxito, etc.

La posibilidad de utilizar canales con memoria asociada para mensajes (*buffers*), más la conexión de canales de *N* posibles fuentes a un destino (*N* a *1*), más los mensajes de notificación de procesos libres (*usrpout*, y *protmanout* en la política Migration Protocol), permitió resolver el problema de deadlock aún cuando la cantidad de procesos de la implementación paralela puede llegar a ser considerable: alrededor de 450 procesos ejecutándose concurrentemente, simulando la ejecución de alrededor de 650 procesos de usuario computando y enviando mensajes, en las pruebas con 32 procesadores.

11.3 Procesos del Entorno para Experimentación

Para conocer más en detalle la tarea de cada proceso conviene hacer una descripción en términos de los datos de *entrada* y *salida* de cada uno de ellos, o al menos de los procesos más importantes. La entrada de un proceso está compuesta por los datos de los mensajes que le llegan al proceso y, de la misma manera, la salida está compuesta por los datos de los mensajes que salen. Los tipos de datos se mostrarán con más nivel de detalle para cada uno de los procesos más importantes, y se seguirá la sintaxis del lenguaje C [Ker91]. Se tomarán como referencia los procesos de la política Follow Me.

El proceso ***usrsim*** se define con cuatro canales de entrada y cuatro canales de salida. Los canales de entrada se utilizan para recibir mensajes desde los procesos *lloader*, *coord*, *consrv*, *randsrv* y los canales de salida son utilizados para enviar mensajes a los procesos *randsrv*, *consrv*, *coord* y *msgman*.

El proceso *usrsim* recibe desde el proceso *lloader* el identificador del procesador local (un número entero). Desde el proceso *coord* recibe descripciones de usuario para simular, la información le llega en una estructura del tipo:

```
struct usrprocdes   /* Descripción de un proceso de usuario      */
{
  int usrprocid;    /* Identificación                            */
  float pc;         /* Requerimiento de Cómputo                  */
  float ps;         /* Probabilidad de envío de mensajes         */
};
```

Figura 11.6: Información desde el Proceso *coord* al Proceso *usrsim*.

El proceso *usrsim* recibe desde el proceso *consrv* una identificación de proceso de usuario (un número entero). Es la respuesta al requerimiento que se produce desde el proceso *usrsim* cuando se ha decidido que un proceso de usuario envía un mensaje. Desde el proceso *randsrv* se reciben números pseudo-aleatorios que se requieren del proceso *randsrv*. Estos números son del tipo *float* de C y se consideran muestras de una variable aleatoria uniforme con valores en el intervalo [0, 1].

Los mensajes que se envían desde el proceso *usrsim* a los procesos *randsrv* y *consrv* son peticiones de un número aleatorio o de una identificación de proceso de usuario respectivamente. En el caso de la petición al proceso *randsrv*, el proceso *usrsim* le envía su propia identificación porque son varios procesos los que pueden requerir números pseudo-aleatorios por el mismo canal de comunicaciones (definido como un canal *N* a *1*). De esta manera el proceso *randsrv* conoce a cuál de ellos debe responder. La petición al proceso *consrv* incluye la identificación del proceso de usuario que va a comunicarse. La petición al proceso *coord* se produce cuando el proceso de usuario que se simula no genera ninguna comunicación, y la información consiste en

```
struct coordreq             /* Requerimientos al proceso coord   */
{
  enum reqproc requester;   /* Identificación                    */
  int datum;                /* Dato asociado a la petición       */
};
```

Figura 11.7: Información desde el Proceso *usrsim* al Proceso *coord*.

donde la identificación de cada proceso que puede realizar las peticiones se define como lo muestra la Fig. 11.8.

```
enum reqproc {usrsimp, consrvp, msgmanp, migmanp, usrpinp, usrpoutp};
```

Figura 11.8: Identificadores de los Procesos.

La identificación es necesaria porque como en el caso de los requerimientos que se realizan al proceso *consrv*, varios procesos realizan las peticiones por el mismo canal de entrada del proceso *coord* (otro de los canales *N* a *1*). En este caso no es necesaria ninguna información asociada al requerimiento, pero el campo *datum* se asigna con la identificación del último proceso de usuario simulado. Los mensajes hacia el proceso *msgman* tienen la estructura de información

```
struct msgreq
{
  int requester;          /* Origen del requerimiento                    */
  int usrsrcid;           /* Identificación de proceso de usuario fuente */
  int usrtgtid;           /* Identificación de proceso de usuario destino */
  int msglength;          /* Longitud del mensaje                        */
  int retnum;             /* Cantidad de retransmisiones del mensaje     */
  int msg[MSGMAXLEN];     /* Datos de los Mensajes                       */
};
```

Figura 11.9: Información de los Mensajes.

que es necesaria para manejar cada mensaje entre procesos de usuario en el proceso *msgman*. El campo *msg* de la estructura de la Fig. 11.9 es el que produce la carga real de la red de comunicaciones.

El proceso ***coord*** posee tres canales de entrada y cuatro de salida. Por dos de los canales de entrada le llegan mensajes de los procesos *lloader*, *usrpin*, y el tercer canal (definido como *N* a *1*) corresponde a los requerimientos que son enviados desde los procesos *usrpout*, *usrsim*, *msgman*, *migman*, *usrpin* y *usrpout*. Los canales de salida son utilizados para enviar mensajes a los procesos *usrpout*, *usrsim*, *tgtsrv* y *datacol*.

El canal de entrada del proceso *coord* que proviene del proceso *lloader* se utiliza para recibir la información de carga de procesos de usuario al procesador local. Esta información incluye la cantidad de procesos de usuario asignados inicialmente al procesador, así como también las descripciones de requerimientos de cómputo y de comunicaciones. La estructura de la información que le llega desde el canal de requerimientos de los otros procesos (que está definido como *N* a *1*), es la que se muestra en la Fig. 11.7 y en la Fig. 11.8. En cada caso el identificador del que realiza el requerimiento define qué hacer en el proceso *coord*.

En general, las peticiones que se reciben en el proceso *coord* desde el proceso *usrsim* y del proceso *msgman* se responden con el envío de un proceso de usuario a simular en el proceso *usrsim*. En el caso de las que llegan desde el proceso *msgman*, también se contabiliza la cantidad de mensajes que envía cada proceso de usuario. Las peticiones desde el proceso *usrpin* contienen el proceso de usuario que se ha migrado al procesador local, y por el canal *1* a *1* desde ese proceso se recibe la cantidad de mensajes que el proceso de usuario ya ha enviado (para conocer cuándo se debe terminar su ejecución). Las peticiones desde el proceso *usrpout* solamente se utilizan para evitar deadlocks, como se ha explicado en una de las secciones anteriores. Las peticiones desde el proceso *migman* contienen un procesador elegido al azar al cual se debe enviar un proceso de usuario que también se elige al azar.

El canal de salida del proceso *coord* hacia el proceso *usrpout* se utiliza para requerir la

migración *física* de procesos de usuario hacia otros procesadores. La estructura de la información se muestra en la Fig. 11.10.

```
struct outpreq        /* Requerimientos de coord a usrpout */
{
  int usrpid;         /* Identificador del Proceso de usuario */
  int msgnum;         /* Cantidad de mensajes ya enviados */
  int tgtprocessor;   /* Procesador destino */
};
```

Figura 11.10: Información desde el Proceso *coord* al Proceso *usrpout*.

Como el proceso *usrpout* es el encargado de la migración física, sólo es necesario enviar la identificación del proceso de usuario que se debe migrar, junto con información que es conocida solamente por el proceso *coord* local. A la identificación del proceso de usuario se le añaden el procesador destino de la migración y la cantidad de mensajes que el proceso ya ha enviado para saber cuándo finaliza su tarea. Todos los procesos *coord* conocen el comportamiento de todos los procesos de usuario y por lo tanto no se envía esta información con la migración del proceso. Si por alguna razón (memoria disponible, por ejemplo) se decide que cada proceso *coord* conozca sólo el comportamiento de los procesos de usuario que se ejecutan localmente, entonces se debería agregar esta información a los mensajes que se envían al proceso *usrpout*.

La información que se envía desde el proceso *coord* al proceso *usrsim* para simular un proceso de usuario ya se mostró en la Fig. 11.6. Al proceso *tgtsrv* se le deben notificar los cambios de ubicación de procesos de usuario que se produzcan, y esta información está contenida en la Fig. 11.11.

```
struct tgtreq         /* Requerimientos al proceso tgtsrv */
{
  int usrprocid;      /* Identificación del proceso de usuario */
  int where;          /* Identificador de procesador */
};
```

Figura 11.11: Información desde el Proceso *coord* al Proceso *tgtsrv*.

Básicamente, la información que se envía desde el proceso *coord* al proceso *datacol* es la necesaria para concentrar los datos recogidos localmente en cada procesador y conocer cuándo se finaliza la ejecución simulada de la aplicación de usuario. La estructura de la información se muestra en la Fig. 11.12.

```
struct datacoord      /* Información desde coord a datacol */
{
  int requester;      /* Identificador de procesador */
  int suspnum;        /* Cantidad de usuarios que han finalizado */
  int mignum;         /* Cantidad de migraciones realizadas */
};
```

Figura 11.12: Información desde el Proceso *coord* al Proceso *datacol*.

donde

- el campo *requester* identifica el procesador desde el cual se envía el mensaje (es decir el procesador al cual está asignado el proceso *coord*),
- el campo suspnum contiene la cantidad de procesos de usuario que se han suspendido en el procesador desde el que se envía el mensaje, y
- el campo mignum contiene la cantidad de migraciones que se han realizado en el procesador desde el que se envía el mensaje.

El proceso ***msgman*** posee tres canales de entrada y cinco canales de salida. Uno de los canales de entrada es el utilizado por los requerimientos de mensajes que se envían desde el proceso *usrsim* local y desde los procesos *mout* de los demás procesadores, y está definido como *N* a *1*. Otro canal de entrada se utiliza para recibir información desde el proceso *lloader*, y el tercer canal de entrada se utiliza para recibir mensajes enviados desde el proceso *tgtsrv*. Los canales de salida se utilizan para enviar mensajes a los procesos *tgtsrv*, *migman*, *coord*, *mout* y *datacol*.

La información contenida en los requerimientos de mensajes es la que se muestra en la Fig. 11.9, y el manejo de cada uno de ellos es el mismo: (a) requerir la ubicación (procesador) del proceso de usuario destino; (b) si el proceso de usuario destino está asignado al procesador local se recibe (consume) el mensaje, y sino, (c) manejar el requerimiento según la política que se utilice para resolver el Problema de Arribo de Mensajes. Desde el proceso *lloader* se recibe información de carga de la aplicación de usuario. Desde el proceso *tgtsrv* se recibe en el proceso *msgman* la ubicación *conocida* del proceso de usuario que se requiera. Esta información depende también de la política utilizada para conservar la consistencia en el arribo de los mensajes.

El canal desde el proceso *msgman* hacia el proceso *tgtsrv* se utiliza para requerir la ubicación (procesador) de procesos de usuario. La información tiene la estructura mostrada en la Fig. 11.11, donde el campo *where* contiene una marca que indica que el mensaje es enviado desde el proceso *msgman* y no una actualización de ubicación de proceso de usuario enviada desde el proceso *coord*. El canal que llega al proceso *tgtsrv* desde los procesos *coord* y *msgman* está definido también como un canal *N* a *1*.

El canal de salida del proceso *msgman* hacia el proceso *migman* solamente se utiliza para que este último genere peticiones de migración dependientes de la tasa de migración utilizada. Los mensajes que se envían por el canal de salida hacia el proceso *coord* tienen la estructura de la Fig. 11.7 y de la Fig. 11.8, y se utilizan para informar de los mensajes que se envían desde los procesos de usuario locales.

El canal de salida del proceso *msgman* hacia el proceso *coord* se utiliza para notificar a este último dos cambios de estado de la simulación. Estos dos cambios de estado se notifican cuando se termina de procesar un requerimiento de mensaje que fue generado por un proceso de usuario local. Estos dos cambios de estado son:

- El proceso de usuario que generó un mensaje está libre como para ser elegido para migración. Se piensa en esta estrategia debido a que hay políticas que requieren que un proceso de usuario no sea elegido para migrar o sea suspendido mientras tiene requerimientos de mensajes que están pendientes de ser procesados (políticas Mailbox y Migration Protocol).

- El proceso *usrsim* está libre, es decir que puede recibir un nuevo proceso de usuario para simular. Esto se debe a que el último proceso de usuario que el proceso *coord* envió al proceso *usrsim* es el que generó el mensaje que se ha procesado.

La estructura de la información que se envía desde el proceso *msgman* al proceso *mout* es la que muestra la Fig. 11.13.

```
struct moutreq          /* Requerimientos desde msgman a mout    */
{
  struct msgreq msg;  /* Mensaje a enviar                      */
  int tgtprocr;         /* Procesador destino del mensaje        */
};
```

Figura 11.13 Información desde el Proceso *msgman* al Proceso *mout*.

donde la estructura *msgreq* es la que se muestra en la Fig. 11.9. El canal de salida desde el proceso *msgman* hacia el proceso *datacol* se dedica a enviar información recogida para evaluar el rendimiento de cada política y poder realizar comparaciones. La estructura de la información que se envía al proceso *datacol* es la que se muestra en la Fig. 11.14. Cada uno de los campos correspondientes a la estructura contiene las mediciones realizadas localmente en el proceso *msgman* y serán utilizados para calcular los índices de rendimiento generales.

```
struct datamsgman /* Datos recogidos en el proceso msgman      */
{
  int requester;      /* Procesador origen de los datos        */
  int retrnum;        /*              Información             */
  int hopnum;
  int ctrlnum;        /*              Recogida                */
  int reqnum;
  int maxlength;      /*              Localmente              */
  int numlenmax;
  float qlength;
  float msglty;       /*              Por mensajes            */
};
```

Figura 11.14: Información desde el proceso *msgman* al Proceso *datacol*.

El proceso ***mout*** posee dos canales de entrada y *N-1* canales de salida, donde *N* es la cantidad de procesadores. Recibe mensajes desde los procesos *lloader* y *msgman*, y envía mensajes a todos los procesos *msgman* que están asignados a otros procesadores.

Desde el proceso *lloader*, el proceso *mout* recibe información para iniciar su tarea en el procesador local. Desde el proceso *msgman* recibe requerimientos de mensajes que deben ser enviados a procesos *msgman* asignados a otros procesadores. La estructura de los requerimientos que llegan desde el proceso *msgman* es la que se muestra en la Fig. 11.13, donde el campo *tgtprocr* identifica el procesador en el cual está asignado el proceso *msgman* destino del requerimiento.

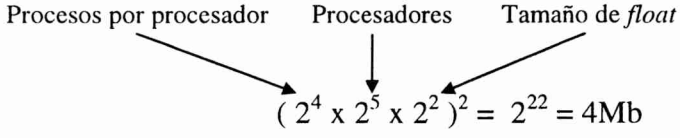
Cada uno de los canales de salida del proceso *mout* forma parte de los canales que se

definen como *N* a *1* y permiten enviar requerimientos de mensajes a los procesos *msgman* en otros procesadores.

El proceso ***tgtsrv*** ha sido definido casi en su totalidad por su relación con procesos anteriores, porque su tarea es: (a) recibir los requerimientos de actualización de ubicación de procesos de usuario del proceso *coord*, y (b) responder a los requerimientos de ubicación de procesos de usuario del proceso *msgman*. Solamente habría que agregar que recibe información inicialmente del proceso *lloader*.

Como en el caso del proceso *tgtsrv*, los procesos ***consrv***, ***migman***, ***usrpin*** y ***usrpout*** ya han sido definidos por su relación con los procesos de la implementación paralela que se han explicado.

Con respecto a la implementación del proceso ***consrv***, se puede comentar que podría implementar la interconexión entre los procesos de usuario con una matriz de *N*x*N*, donde *N* es la cantidad total de procesos de usuario. Cada posición (*i*, *j*) de la matriz podría contener la probabilidad de que el proceso de usuario pu_i envíe un mensaje al proceso de usuario pu_j. La cantidad de memoria necesaria para una matriz de estas características teniendo 32 procesadores puede llegar a ser considerable y suele exceder los límites impuestos por la memoria real de los procesadores en un DMPC, haciéndose necesaria la utilización de algún tipo de manejo de memoria virtual. En el caso particular de los transputers, no manejan memoria virtual, y por lo tanto esta implementación no es posible para la cantidad de procesos de usuario que se controlan en una red de 32 procesadores. Por ejemplo: si se asignan 16 procesos de usuario en cada procesador (en promedio, ya que el número varía en tiempo de ejecución), y se utilizan los 32 procesadores disponibles, la cantidad de memoria real necesaria en cada transputer es



El proceso ***randsrv*** tiene dos canales de entrada y tres de salida. Por medio de los canales de entrada recibe información del proceso *lloader* y recibe las peticiones de números pseudo-aleatorios muestras de una variable aleatoria uniforme en el intervalo [0, 1]. Por los canales de salida envía las muestras requeridas a los procesos que las solicitan: *usrsim*, *consrv*, y *migman*.

11.3.1 Procesos Relacionados con la Generación de Migración

Con respecto a la implementación de los procesos *msgman*, *migman* y de la política de migraciones es necesario aclarar que, a diferencia de [Hey95], se decidió generar migraciones a partir de la llegada de un mensaje y no en el tiempo en que se realiza el envío.

Las pruebas preliminares mostraron que generar migraciones en el tiempo del *send* del mensaje producía desbalance de carga la mayoría de las veces. Esto se debe a que cuando uno de los procesadores, por ejemplo P*m*, tiene algunos procesos de usuario más que los demás (lo cual puede suceder temporariamente), más mensajes serán destinados a este procesador. Por

lo tanto, el proceso *msgman* de P*m* estará más tiempo ocupado para gestionar los mensajes que le llegan, y esto es así independientemente de la política que se proponga para conservar la consistencia en el arribo de los mensajes. Si el proceso *msgman* ocupa por más tiempo la CPU para gestionar mensajes, esto a su vez implica que los procesos de usuario tienen menos tiempo de CPU disponible para ejecutarse y para realizar envíos de mensajes. Si se define que las migraciones dependen de los envíos (*send*) de mensajes, habrá menos posibilidades de que el generador de migraciones, el proceso *migman* de P*m*, actúe, y por lo tanto todos los procesos de usuario que están asignados al procesador P*m* así como los lleguen por migraciones desde otros procesadores, permanecerán en P*m*.

Las pruebas preliminares, que se realizaron para 2, 4 y 8 procesadores mostraron que aproximadamente en el 95% de las ejecuciones se produce desbalance. Una vez producido el desbalance, un procesador llega a tener más del 90% de los procesos de usuario asignados localmente y casi con exclusividad se dedica a gestionar los mensajes destinados a los procesos de usuario locales.

El problema de desbalance de carga no se produce en la simulación de las políticas tal como se hizo en [Hey95] porque no hay restricciones de tiempo real con respecto a la utilización de la CPU. Es como si la gestión de los mensajes en cada procesador siempre se pudiera realizar a la vez que avanza la ejecución de cada uno de los procesos de usuario.

Para resolver el problema de desbalance, el proceso *msgman* envía una señal al proceso *migman* cuando se produce la recepción de un mensaje, es decir, cuando un mensaje llega al procesador donde está ubicado el proceso de usuario destino. De esta manera, cuando un procesador tiene más procesos de usuario que los demás y reciba más mensajes que los demás, el proceso *migman* recibirá más notificaciones de mensajes y se generarán más migraciones que en los demás procesadores, manteniendo así la carga relativamente balanceada a lo largo de toda la ejecución de la aplicación de usuario.

Desde el punto de vista de la codificación de ambas alternativas, tanto generar migraciones cuando se envía un mensaje como cuando se recibe, son similares. Más aún, el proceso *migman* ni siquiera conoce cuándo recibe la señal de generar migración aleatoriamente.

11.3.2 Adaptación de los Procesos a cada Política

La variación que se produce en cada proceso para implementar cada política, puede comprenderse a partir de la adaptación del diseño del entorno para experimentación de migración en DMPCs a cada política. Siempre se toman como referencia los procesos que implementan la política Follow Me, tanto los procesos que cambian, así como los procesos nuevos.

Se hace necesario explicar con un poco más de nivel de detalle lo que sucede en la implementación de la política Migration Protocol. En las pruebas preliminares de la implementación se hizo notorio que tanto el propio protocolo de migración como el mecanismo de prevención de bloqueos por migración de procesos tenían como consecuencia que los procesos de usuario casi no migraran en el tiempo de ejecución de la aplicación. Esto se produce porque mientras se lleva a cabo la migración, todos los demás procesos continúan

ejecutando y se generan migraciones, pero no hay procesos de usuario disponibles para migrar. Esta situación es particularmente crítica cuando se asignan, en promedio, pocos procesos por procesador.

En la política Migration Protocol tal como se la describe en la sección de Adaptación del Diseño a cada Política. Migration Protocol, es común que haya más migraciones pendientes que procesos de usuario ejecutándose localmente. La razón para que esto ocurra es que pueden llegar al procesador más mensajes que procesos de usuario locales y, por lo tanto, se pueden generar más requerimientos de migración que procesos de usuario locales. Como cada migración bajo la política Migration Protocol lleva mucho más tiempo en llevarse a cabo comparado con el tiempo de las demás políticas, la cantidad de migraciones pendientes se acumulan y no se llegan a realizar todas las migraciones que se generan en *migman*.

Para lograr que la cantidad de migraciones se correspondiera con la tasa de migración, se decidió cambiar el requerimiento "fin de migración" desde el proceso *coord* al proceso *protmanout* a una de las dos alternativas que siguen: (a) "fin de migración para continuar ejecutando" o (b) "fin de migración para seguir migrando". De esta manera, cuando el proceso llega a un procesador que tiene más migraciones pendientes que procesos de usuario locales, se genera un "fin de migración para seguir migrando" de tal forma que el proceso *msgman* no envía los mensajes que tiene pendientes para el proceso de usuario que ha llegado a un procesador, porque seguirá migrando. Así, se logra que las migraciones se generen de forma más distantes en el tiempo y, por lo tanto, todos (o la mayoría de) los requerimientos de migración puedan llevarse a cabo.

La decisión de no enviar los mensajes pendientes desde el proceso *msgman* hacia la nueva ubicación del proceso de usuario que se migra tiene como primera consecuencia que los mensajes estén esperando más tiempo del que sería necesario. Este tiempo mayor de espera se "amortigua" haciendo que el proceso de usuario que envía el mensaje quede suspendido momentáneamente hasta que el proceso de usuario destino deje de migrar (se genere un "fin de migración para continuar ejecutando"). De esta forma se generan menos mensajes por unidad de tiempo y por lo tanto se evita que más mensajes esperen hasta que el proceso de usuario destino deje de migrar. Quizás esta decisión sea muy discutible en términos de precisión de las medidas que se tomen por mensajes, pero es la única forma que se encontró de hacer que las cantidad de migraciones para la política Migration Protocol se correspondiera más o menos directamente con la tasa de migración.

11.3.3 Estructura General del Entorno para Experimentación

Si se tiene en cuenta ahora la interconexión que se ha definido entre los procesos de la implementación paralela, más los canales y procesos necesarios para la carga de la aplicación de usuario, se puede aumentar el nivel de detalle de la Fig. 6.3 al que se muestra en la Fig. 11.15. Se puede ver la implementación paralela para dos procesadores con el nivel de detalle que se ha definido en esta sección. Quizás en este punto se pueda tener una idea de la complejidad del entorno para experimentación de migración en DMPCs. Esta complejidad está dada en forma directa con la cantidad y complejidad de los procesos asignados a cada procesador. Si bien la cantidad de procesadores aumenta los canales de comunicación entre los procesadores y la cantidad de memoria requerida por algunos procesos, no aumenta considerablemente la complejidad total de todo el entorno de experimentación.

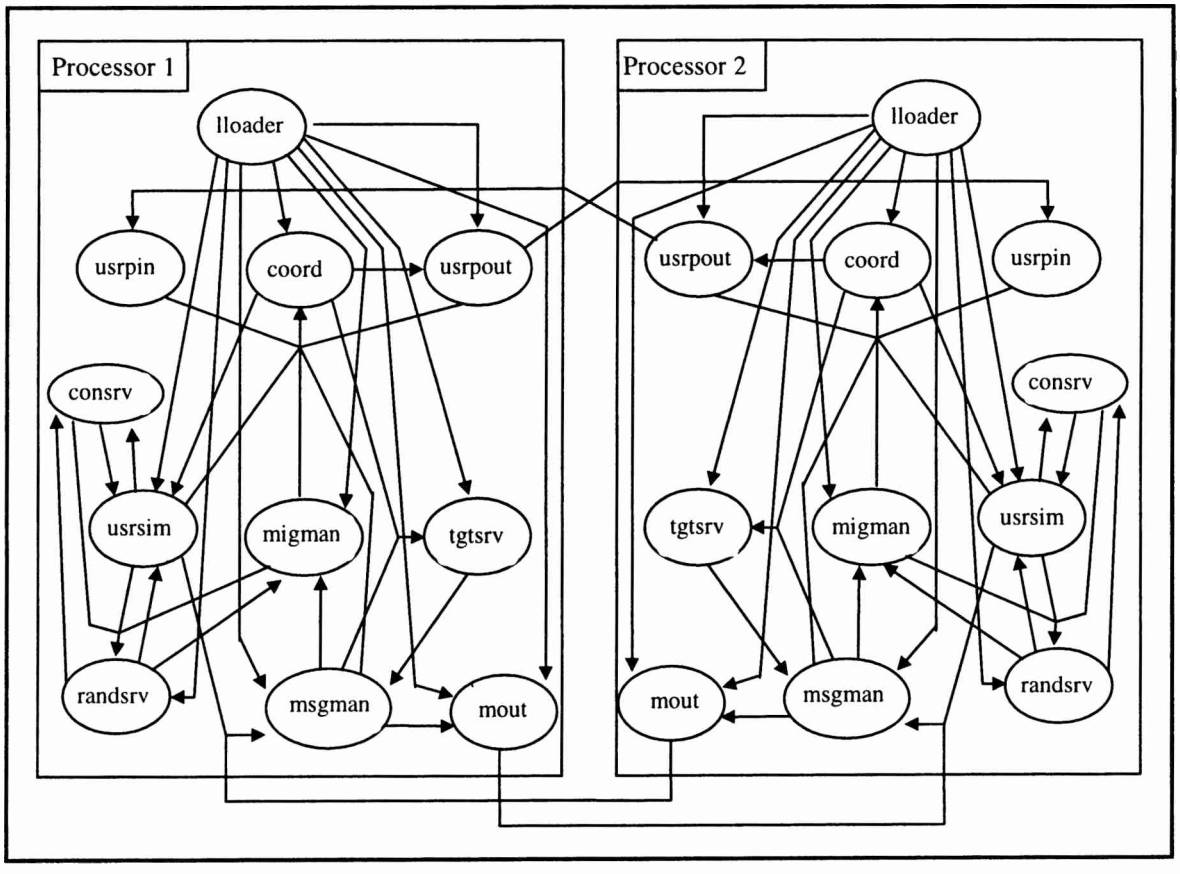


Figura 11.15: Entorno de Experimentación en Dos Procesadores.

Los datos que debe distribuir el proceso *lloader* (cargador local de la aplicación de usuario) en cada procesador le llegan desde el proceso *gloader* (cargador global de la aplicación de usuario), tal como lo muestra la Fig. 6.8. Los canales hacia el proceso *datacol* no se muestran, pero como se ha explicado, proceden de los procesos *msgman* y *coord* asignados a cada procesador.

En la Fig. 11.15 se puede ver cómo se conservan los dos circuitos de información, formados respectivamente por los procesos *usrpin* - *usrpout* y *msgman* - *mout*. Asimismo, se debe notar que los canales de llegada a los procesos *coord*, *msgman*, *tgtsrv* y *randsrv* se definen como canales *N* a *1* de TransComm. Esta implementación se puede extender a la cantidad de procesadores que posea el DMPC que se utilice, replicando los procesos en cada uno de ellos.

11.4 Cálculo de los Indices de Evaluación

Algunos de los índices de evaluación definidos son observables de forma inmediata en los procesos del entorno de experimentación, pero otros se deben calcular de acuerdo a mediciones que se tomen localmente en cada procesador. En todos los casos, los índices de evaluación generales se calculan en el proceso *datacol* en base a las mediciones locales que se realicen en cada procesador.

La cantidad de **mensajes de control** que se generan es contable en cada procesador, en

los procesos que los producen. En las políticas Global Server, Message Rejection y Mailbox, los mensajes de control son generados y pueden ser contabilizados en los procesos *msgman* de cada procesador. En la política Home Processor, los procesos *msgman* y *coord* generan y contabilizan los mensajes de control en cada procesador. Para la política Migration Protocol, los mensajes de control pueden ser contabilizados en el proceso *coord*, porque es proporcional a la cantidad de procesos de usuario que se migran y al patrón de interconexiones entre los procesos de usuario. Es posible calcular el promedio de mensajes de control por mensaje de usuario porque se conoce la cantidad total de mensajes que se generan en la aplicación de usuario

$$\#Mensajes = 150 \times \#Procesos\_de\_Usuario \tag{11.1}$$

donde *#Procesos_de_Usuario* es la cantidad total de procesos de usuario que se ejecutan en la máquina paralela y cada proceso de usuario genera 150 mensajes. Por lo tanto, la cantidad promedio de mensajes de control por mensajes de usuario se puede calcular como

$$\overline{\#Mensajes\_de\_Control} = \left(\sum_{i=1}^{M} \#Mensajes\_de\_Control_i \right) / \#Mensajes \tag{11.2}$$

donde $\#Mensajes\_de\_control_i$ es la cantidad de mensajes de control que se generaron en el procesador i y M es la cantidad total de procesadores.

La cantidad de **retransmisiones** que se producen por mensaje entre procesos de usuario no es calculable de forma tan inmediata como el caso de los mensajes de control. Cuando son posibles las retransmisiones (en todas las políticas excepto en Migration Protocol), es posible que el requerimiento de mensaje pase por varios procesadores (procesos *msgman*) hasta que llega al proceso de usuario destino. En ninguno de los procesadores intermedios es posible conocer cuántas retransmisiones más se necesitarán para el mensaje. Por esta razón es que en el propio requerimiento de mensaje se dedica un campo (campo *retrnum* de la estructura de la Fig. 11.9), para contener la cantidad de retransmisiones que han sido necesarias. Este campo se incrementa cada vez que el mensaje debe retransmitirse a un nuevo procesador.

Cuando un mensaje llega al proceso *msgman* donde se recibe (el proceso de usuario destino está asignado al procesador local), se puede almacenar la cantidad de retransmisiones que ha necesitado para llegar al proceso de usuario destino. De esta forma es posible tener la cantidad de retransmisiones que fueron necesarias para los mensajes que han llegado a cada procesador. El cálculo de la cantidad de retransmisiones por mensaje se puede hacer en el proceso *datacol* de forma similar a la cantidad de mensajes de control por mensaje de usuario

$$\overline{\#Retransmisiones} = \left(\sum_{i=1}^{M} \#Retransmisiones_i \right) / \#Mensajes \tag{11.3}$$

donde $\#Retransmisiones_i$ es la suma de las retransmisiones de los mensajes que han llegado al procesador i, y M es la cantidad total de procesadores.

La **latencia de los mensajes** no puede ser calculada en forma directa para cada mensaje, porque no se dispone de un reloj global de toda la red de procesadores al cual hacer

referencia antes de enviar un mensaje y después de ser recibido. Esta carencia de información general (en este caso no se dispone de un reloj global), es algo común en las multicomputadoras. Lo que sí se puede calcular por mensaje, y de hecho se puede hacer en el proceso *msgman* de cada procesador, es

- La cantidad de procesadores intermedios por los que el mensaje pasa hasta llegar al procesador destino (*#saltos*), que incluye las retransmisiones. Cuando el proceso de usuario origen y destino del mensaje están en el mismo procesador no se realiza ningún salto.
- La cantidad de requerimientos por los que tiene que esperar un mensaje hasta ser analizado en el proceso *msgman* (*#cola*). Cuando un requerimiento llega al proceso *msgman* se almacena en una estructura de cola del tipo FIFO (*First In First Out)*, y debe esperar que todos los requerimientos previos sean procesados hasta que llegue su turno.
- El tiempo que se necesita para la transmisión de un requerimiento de mensaje desde un procesador a otro (*latencia_de_salto*). Debe aclararse que no es siempre el mismo porque depende de la carga de la red de comunicaciones.

En todos los casos se deben calcular los promedios por mensaje de cada una de estas medidas locales de cada procesador.

Cada proceso *msgman* puede calcular su propio promedio de las medidas *#cola* y *latencia_de_salto*, porque son medidas estrictamente locales. En el proceso *datacol* solamente se deben calcular los promedios de lo que se reciba desde cada procesador.

El cálculo de la cantidad de saltos promedio de los mensajes se lleva a cabo conociendo la cantidad de retransmisiones, y no es inmediato como para las medidas de *#cola* y *latencia_de_salto*. Aún así, la relación entre la cantidad de saltos de un mensaje y la cantidad de retransmisiones es casi inmediata. Cuando un mensaje llega al proceso *msgman* donde se recibe (el proceso de usuario destino está asignado al procesador local), la cantidad de saltos de un mensaje se calcula como

$$\#saltos = \begin{cases} 0, \text{ si el requerimiento es local} \\ retrnum+1, \text{ en caso contrario} \end{cases} \tag{11.4}$$

donde *retrnum* es la cantidad de retransmisiones del mensaje (campo de la estructura para los requerimientos de mensajes de la Fig. 11.9). De esta manera se tienen en cuenta los saltos que no son retransmisiones, y se puede tener en el proceso *msgman* la cantidad de saltos de los mensajes que se han recibido en el procesador. El promedio de saltos de mensajes puede calcularse como el promedio de mensajes de control por mensaje de la Ec.(11.2), tal como lo muestra la Ec. (11.5).

$$\overline{\#Saltos} = \left(\sum_{i=1}^{M} \#Saltos_i \right) / \#Mensajes \tag{11.5}$$

donde $\#saltos_i$ es la suma de los saltos entre procesadores de los mensajes que han llegado al procesador i, y M es la cantidad total de procesadores.

Teniendo la cantidad promedio de la longitud de cola de los requerimientos de mensajes, y el tiempo promedio de transmisión de un requerimiento de mensaje de usuario

desde un procesador a otro, se puede calcular la cantidad de tiempo de espera promedio de un requerimiento como

$$\overline{Espera\_en\_cola} = \overline{\#cola} \times \overline{\#latencia\_de\_salto} \tag{11.6}$$

La cantidad de tiempo total desde que un mensaje es generado hasta que llega al proceso de usuario destino debe tener en cuenta la cantidad de saltos que fueron necesarios y, según las políticas, los mensajes de control intermedios.

En la política Follow Me no hay mensajes de control de ninguna clase y por lo tanto solamente son necesarios los saltos y la espera promedio de los requerimientos en cada procesador. En la política Migration Protocol, hay mensajes de control, pero el tiempo que es necesario para llevar a cabo todo el protocolo está incluido en la espera dentro de cada procesador porque los mensajes se suspenden hasta que el proceso de usuario destino sea migrado. Esto implica que tanto para la política Follow Me como para la política Migration Protocol, la latencia promedio de los mensajes entre procesos de usuario puede calcularse como

$$\overline{Latencia} = \overline{\#\text{Saltos}} \times \overline{\#Espera\_en\_\text{cola}} \tag{11.7}$$

donde la cantidad de saltos promedio y la cantidad de tiempo de espera promedio en cada procesador se calculan de acuerdo a la Ec. (11.5) y la Ec. (11.6) respectivamente.

En el caso de las políticas Global Server, Home Processor y Message Rejection, se deben tener en cuenta los mensajes de control que pueden aparecer como respuesta a los mensajes que se envían a un procesador donde no está ubicado el proceso de usuario destino. Para estas políticas, se deben tener en cuenta, por lo tanto, los mensajes de control NACKs, que deben ser procesados en los procesos *msgman* de la misma manera en que son procesados los requerimientos de mensajes.

El tiempo de espera de los mensajes de control NACK, hasta ser procesados por el proceso *msgman* se puede calcular como el tiempo de espera en cola de los requerimientos de mensajes de la Ec. (11.6). Como esta espera en cola se repite para cada NACK, y cada NACK se responde con una retransmisión del mensaje (un NACK implica una retransmisión), la latencia de los mensajes se ve aumentada por el tiempo de los mensajes de rechazo

$$\overline{T\_de\_NACKs} = \overline{\#Retransmisiones} \times \overline{\#Espera\_en\_\text{cola}} \tag{11.8}$$

Por lo tanto, la latencia de los mensajes para estas tres políticas (Global Server, Home Processor y Message Rejection), se puede calcular como

$$\overline{Latencia} = \overline{\#saltos} \times \overline{\#Espera\_en\_cola} + \overline{\#T\_de\_NACKs} \tag{11.9}$$

o, utilizando la igualdad de la Ec. (11.8), se reemplaza el promedio de tiempo que se necesitan para los rechazos de mensajes (NACKs), y se llega a la ecuación

$$\overline{Latencia} = \left(\overline{\#saltos} + \overline{\#Retransmisiones}\right) \times \overline{\#Espera\_en\_cola} \tag{11.10}$$

Para la política Mailbox sucede algo semejante a las políticas anteriores. La diferencia es que el mensaje de control que se debe contabilizar es el de petición del mensaje que se corresponde con la primitiva *receive* para recepción generada por el proceso de usuario. Como la cantidad de mensajes de control (peticiones de mensajes de los buzones) es igual a la cantidad de retransmisiones, el cálculo de la latencia para la política Mailbox es como se define en la Ec. (11.10). La diferencia principal es que tanto la recepción del mensaje en el buzón como la petición de recepción que se envía desde el proceso de usuario pueden solaparse en el tiempo. En este sentido, la Ec. (11.10) considera que se produce un mensaje después del otro y por lo tanto es el peor caso, sin solapamiento en el tiempo de ambos mensajes.

El **tiempo de reacción** de los procesos de usuario siempre se puede medir en el proceso *coord*. Este proceso es el que: (a) recibe las peticiones de migración desde el proceso *migman*, y (b) conoce cuándo el proceso de usuario elegido es enviado a otro procesador. El tiempo que transcurre entre ambos eventos es, tal como se lo ha definido, el tiempo de reacción de los procesos de usuario.

En el caso de la política Migration Protocol, el tiempo de reacción es el tiempo que se utiliza para llevar a cabo los primeros cuatro pasos definidos para llevar a cabo la migración de un proceso de usuario. En los términos de la descripción hecha del diseño de la política, es el tiempo que transcurre entre los requerimientos "inicio de migración" y "fin de migración" que se envían desde el proceso *coord* hacia el proceso *protmanout*. Para el resto de las políticas, el tiempo de reacción puede analizarse sin realizar mediciones locales, ya que en todas ellas está relacionado en forma directa con el tiempo en que un proceso de usuario se comunica con los demás.

Para las aplicaciones de usuario que se han definido, el análisis de la **modificación en el patrón de comunicaciones** entre procesadores que produce cada política, puede realizarse sin necesidad de monitorizar la aplicación de usuario simulada. En todo caso, la monitorización permite conocer más a fondo cómo evoluciona cada política.

Si se necesita realizar una medición exhaustiva del patrón de comunicaciones entre procesadores, en una primera aproximación se podrían contabilizar la cantidad total de requerimientos de mensajes de usuario que llegan a, y/o salen de cada *msgman* por unidad de tiempo, y comparar estas cantidades provenientes de distintos procesadores. Si se necesitan medidas tomadas con más nivel de detalle, por ejemplo la cantidad total de mensajes (y/o datos) que llegan a, o salen de cada procesador, se podrían contabilizar también los procesos de usuario que llegan y/o salen a/de cada procesador por unidad de tiempo, por causa de las migraciones dinámicas.

Finalmente, la **cantidad de migraciones** de procesos de usuario locales se pueden contabilizar en el proceso *coord* de cada procesador. Cada proceso *coord* incrementa la cantidad de migraciones locales siempre que se envía un proceso de usuario hacia otro procesador. Al finalizar la ejecución de la aplicación de usuario, estas medidas locales se deben enviar al proceso *datacol*, que es el que puede calcular la cantidad promedio de migraciones por proceso de usuario. El promedio no tiene en cuenta la cantidad de mensajes sino la cantidad total de procesos de usuario. La cantidad promedio de migraciones por proceso de usuario se puede calcular como

$$\overline{\#\,Migraciones} = \left(\sum_{i=1}^{M} \#\,Migraciones_i \right) / \#\,Procesos\_de\_usuario \tag{11.11}$$

donde $\#Migraciones_i$ es la cantidad de migraciones que se realizaron en el procesador i, M es la cantidad total de procesadores, y *#Procesos_de_usuario* es la cantidad total de procesos de usuario.

12. Entorno para Experimentación en DMPCs: Experimentación y Análisis de Resultados

En esta sección se muestran los resultados obtenidos al realizar las experimentaciones que se describieron anteriormente. Los resultados se analizan desde dos puntos de vista:

1. Comparativo entre las políticas que se presentaron para resolver el Problema de Arribo de Mensajes.
2. Comparativo con los resultados obtenidos en simulación y que se detallan en [Hey95].

En cada caso, los gráficos que se muestran se realizaron para una cantidad determinada de procesadores y procesos por procesador, variando la tasa de migración (probabilidad de migración por mensaje de proceso de usuario) en el eje X del gráfico. La cantidad de procesos por procesador es en realidad la cantidad inicial que se asignan y la cantidad promedio en tiempo de ejecución, ya que la cantidad de procesos que están asignados a un procesador puede variar en el tiempo debido a las migraciones.

12.1 Tasa de Migración y Cantidad de Migraciones

Como paso previo para la comparación de las políticas, es necesario conocer qué sucede en promedio en los procesadores con respecto a carga de procesamiento y de comunicaciones a lo largo de todos los experimentos. Con esta información se pueden analizar luego los resultados obtenidos para los demás índices de evaluación. Si bien la tasa de migración se varía de la misma manera para todas las políticas en todas la ejecuciones de aplicaciones de usuario, es útil verificar la cantidad de migraciones que se producen con la misma tasa de migración en todos los casos en los que se experimenta. Se muestra en la Fig. 12.1 y 12.2 la cantidad de migraciones promedio de cada proceso de usuario.

Sería de esperar que las cantidades de migraciones promedio por proceso fueran iguales para igual tasa de migración. Se pueden relacionar la cantidad de migraciones por proceso con la carga de procesamiento y comunicaciones que tiene cada procesador, porque todo lo demás de las aplicaciones de usuario permanece sin cambios. Cada migración genera carga de la red de comunicaciones porque se transmite el proceso, y también se cargan los procesadores porque deben tener en cuenta agregar o eliminar un proceso de usuario de los que se ejecutan localmente. Si las cantidades de migraciones por proceso de usuario son similares, los demás resultados pueden compararse. Si las cantidades que se obtienen son muy distintas entre sí, para algunas políticas se tendría mayor tiempo utilizado en procesar y comunicar las migraciones mientras que para otras los recursos se habrían utilizado para la ejecución del programa de usuario.

Primero se muestran los resultados obtenidos con 5 procesos de usuario por procesador y luego con 20 procesos de usuario por procesador. En el eje Y de cada gráfica se muestran las cantidades de migraciones promedio de cada proceso de usuario. Se comentan brevemente los resultados obtenidos. Los resultados obtenidos para los demás índices de evaluación (latencia de mensajes, mensajes de control, etc.), que son los que se consideran más importantes, serán discutidos más detalladamente.

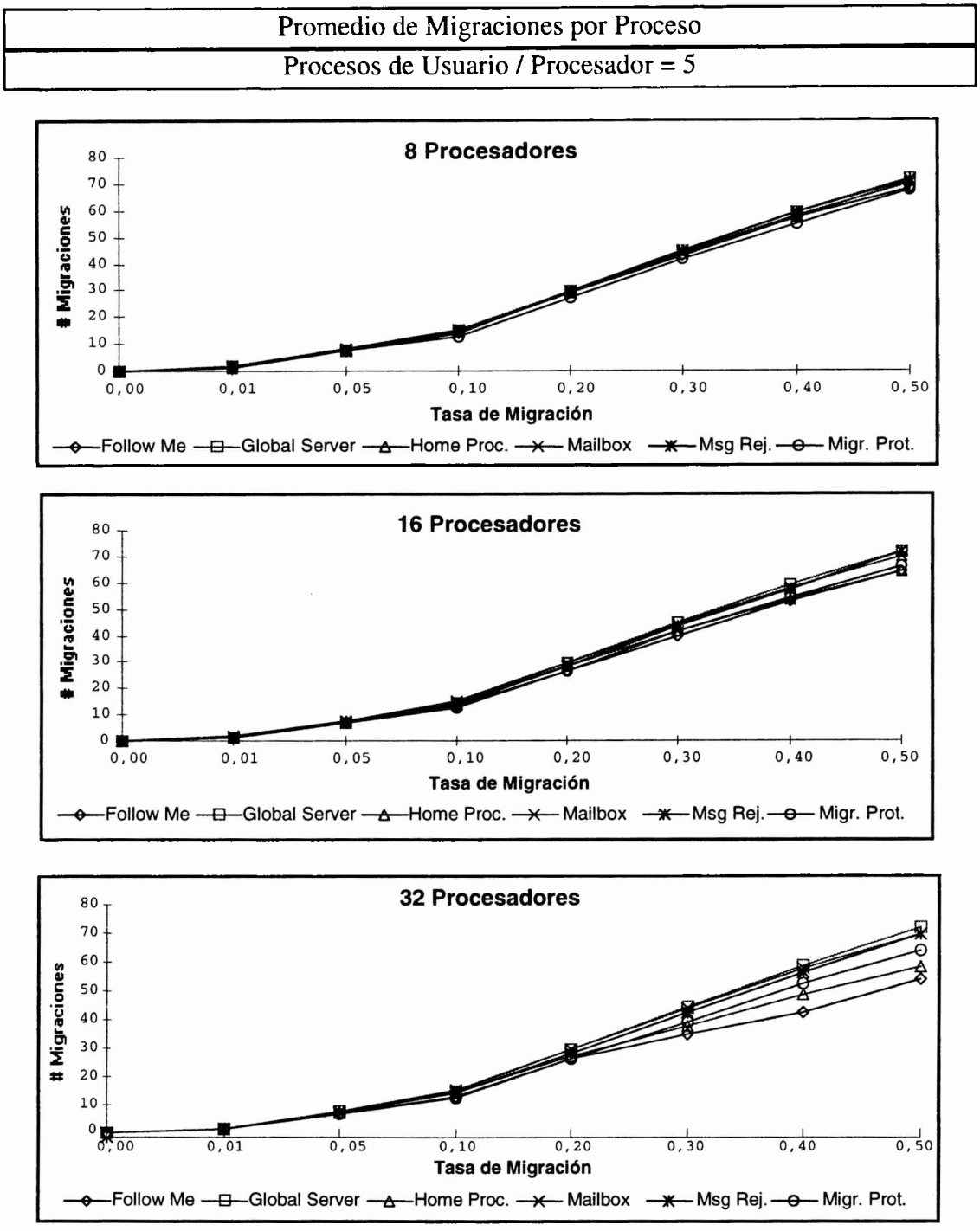


Figura 12.1: Cantidad de Migraciones. 5 Procesos por Procesador.

En todos los resultados que se muestran en la Fig. 12.1 aparecen las mismas tendencias para todas las políticas. Puede haber algunas excepciones para los resultados obtenidos con 32 procesadores y tasas altas de migración (mayores de 30%). En todo caso, cuando se analicen los demás índices de rendimiento habría que considerar la cantidad promedio de migraciones por proceso para estos casos excepcionales.

Cuando los procesadores tienen asignado un mayor número de procesos de usuario, como lo muestra la Fig. 12.2, los resultados que se obtienen son similares para todas las

políticas con la excepción de Migration Protocol. Tanto para 8, 16, como para 32 procesadores la cantidad promedio de migraciones por proceso que se realizan en la política Migration Protocol es menor. Esto se debe a lo que se explicó en la sección de detalles de implementación relacionado al comportamiento de esta política con respecto a las demás. Para 16 y 32 procesadores se encuentran diferencias notables con respecto a todas las demás políticas.

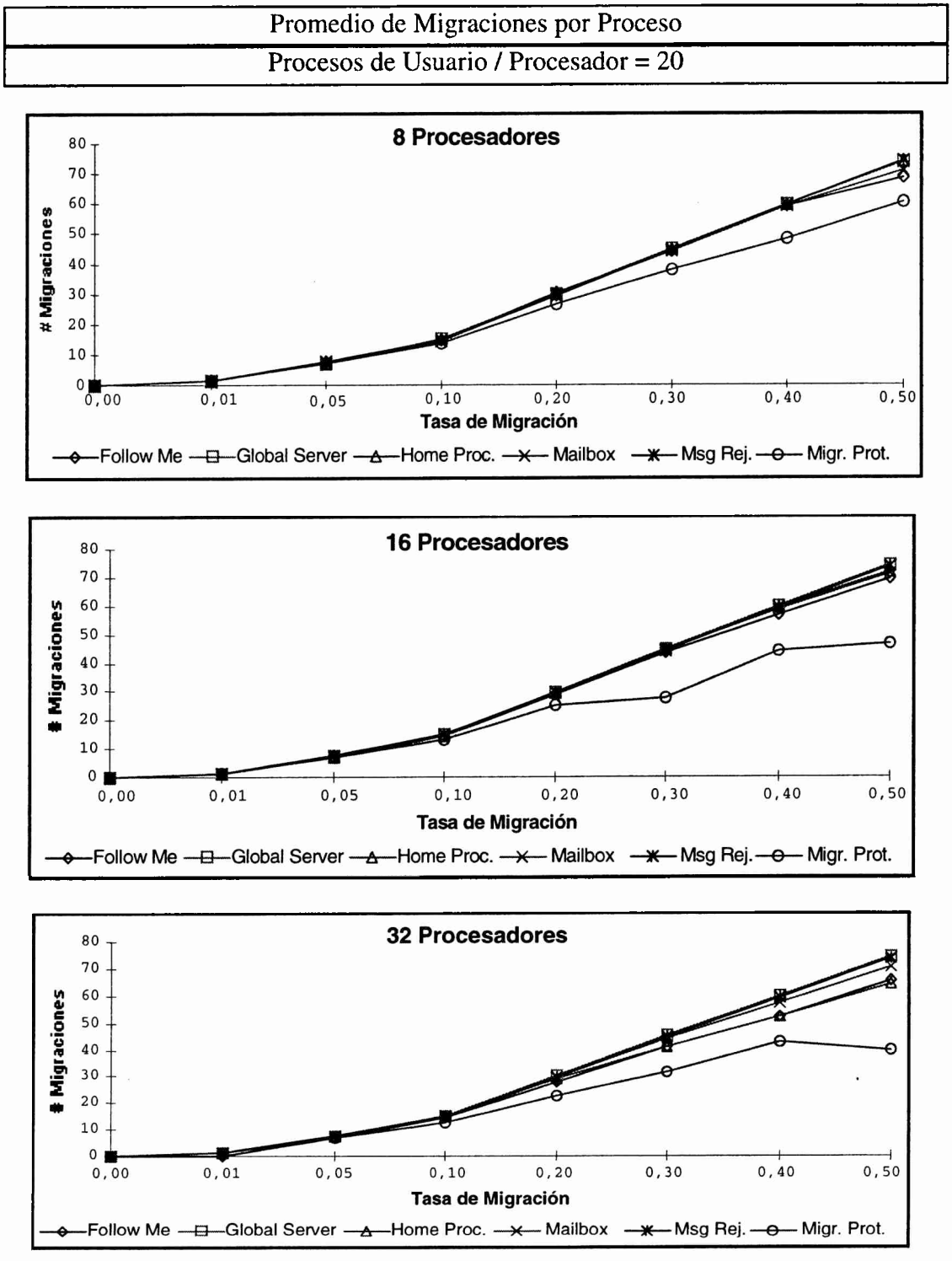


Figura 12.2: Cantidad de Migraciones. 20 Procesos por Procesador.

En cuanto a la comparación de estos resultados (Fig. 12.1 y Fig. 12.2) con los obtenidos en [Hey95] por simulación, en líneas generales se obtienen los mismos valores, con la excepción de la política Migration Protocol.

Los resultados obtenidos en las simulaciones de todas las políticas, en todos los casos dan aproximadamente la misma cantidad de migraciones promedio por proceso. Como ya se ha explicado, en las simulaciones no se tienen en cuenta los problemas de carga de la red de comunicaciones ni de los procesadores. Por esta razón, siempre que se debe simular el protocolo de migración para la política Migration Protocol, todas las comunicaciones se realizan en el menor tiempo posible, es decir, como si no hubiera otros mensajes en la red de comunicaciones ni el procesador tuviera que realizar otras tareas. Estas otras tareas del procesador incluyen la ejecución de procesos de usuario, la migración de procesos de usuario, y la gestión de los mensajes entre procesos de usuario.

12.2 Latencia Promedio de los Mensajes

En esta sección se muestran y comentan los resultados obtenidos con respecto a la latencia promedio de los mensajes. El cálculo de este índice de evaluación se hizo de acuerdo a lo detallado en la sección de Cálculo de los Indices de Evaluación. En el eje Y de cada gráfica se muestra el tiempo (t) promedio de latencia de los mensajes. Primero se muestran los resultados para 5 procesos de usuario por procesador en la Fig. 12.3.

Como se explicó anteriormente, este índice de evaluación de rendimiento es uno de los más importantes por el impacto directo que tiene sobre el tiempo de ejecución de los programas de usuario. Por otro lado también es el que proporciona más información unificada, porque todos los demás índices tienen relación directa o indirecta con la latencia. Es por esta razón que algunos resultados de latencia de mensajes se pueden comprender cuando se comprendan los resultados de los demás índices de rendimiento.

Como primera comparación entre lo que sucede para las distintas cantidades de procesadores utilizados de la Fig. 12.3, se pueden ver los diferentes rangos en los valores de latencia promedio de mensajes. Si bien se pueden notar diferencias entre las distintas políticas con respecto a estos valores, el incremento de las latencias para cada cantidad de procesadores se corresponde con la mayor utilización de la red de comunicaciones, y con la mayor cantidad de colisiones de los mensajes que se producen.

En todos los casos se puede notar que los resultados obtenidos para las diferentes políticas no difieren demasiado hasta el valor de tasa de migración igual a 0.05. Dentro del rango de valores de tasa de migración entre 0.0 hasta 0.05, el tiempo de latencia promedio de los mensajes para la política Global Server es el mayor para las tres cantidades de procesadores (8, 16, y 32). La mayor latencia de la política Global Server es producida por las consultas al Servidor Global que se deben realizar para enviar un requerimiento de mensaje a otro procesador.

Para el rango de valores de tasa de migración entre 0.10 y 0.50, los valores de latencia de las políticas se separan, identificándose mejor las diferencias que se encuentran para cada política. En todos los casos la política Follow Me tiende a ser la peor en este rango, usualmente seguida por la política Home Processor. La política Follow Me es afectada en

forma directa por la cantidad de retransmisiones que se producen, las cuales a su vez implican tiempo real de comunicaciones y congestión de la red de comunicaciones. Como se verá en una de las siguientes secciones, la política Follow Me es la que más retransmisiones produce y por lo tanto es explicable la diferencia de latencia promedio obtenida con respecto a las demás políticas. La política Global Server se mantiene relativamente constante, al igual que la política Migration Protocol.

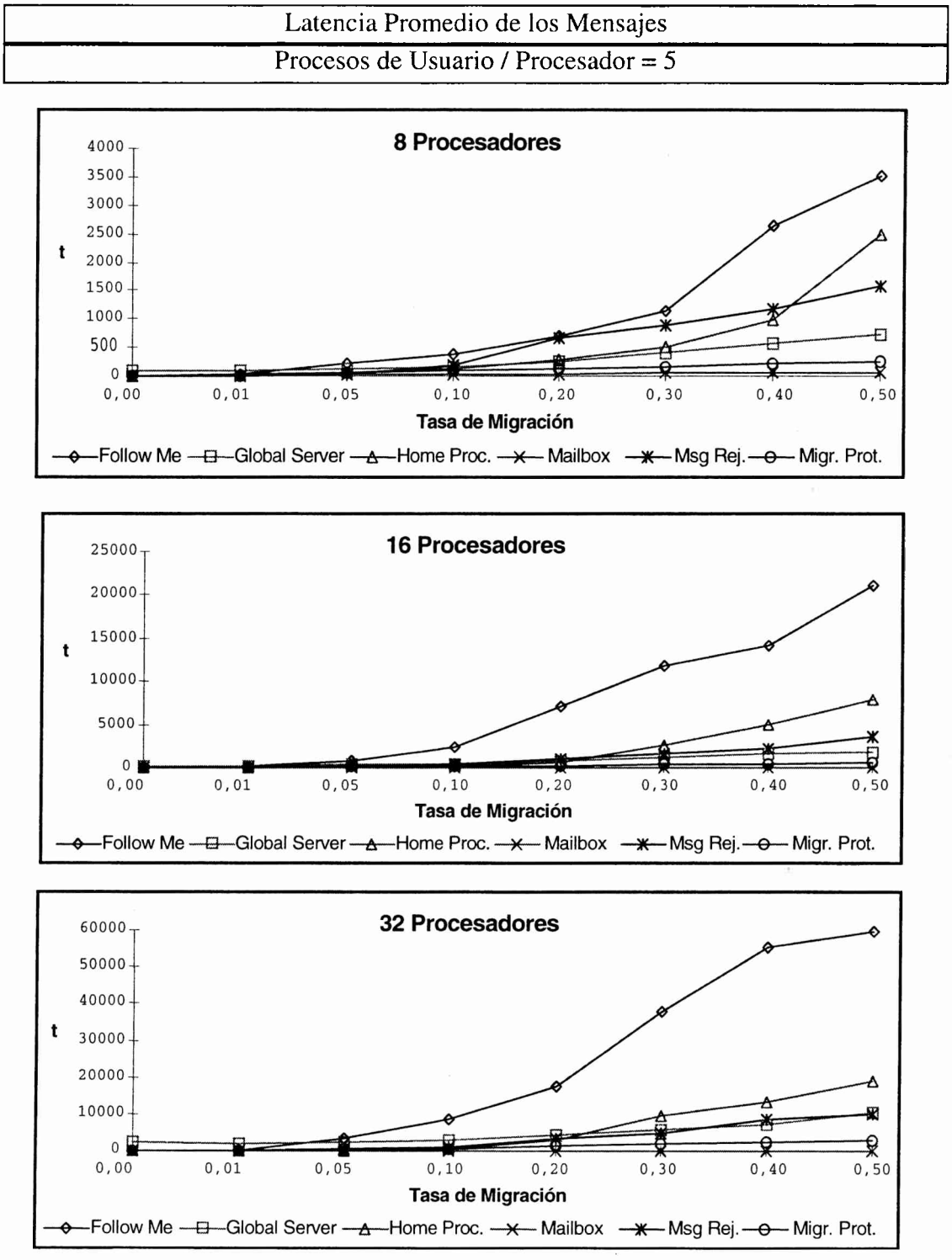


Figura 12.3: Latencia de Mensajes. 5 Procesos por Procesador.

En la Fig. 12.4 se muestran los valores de latencia de mensajes cuando se asignan en promedio 20 procesos de usuario por procesador, para 8, 16 y 32 procesadores.

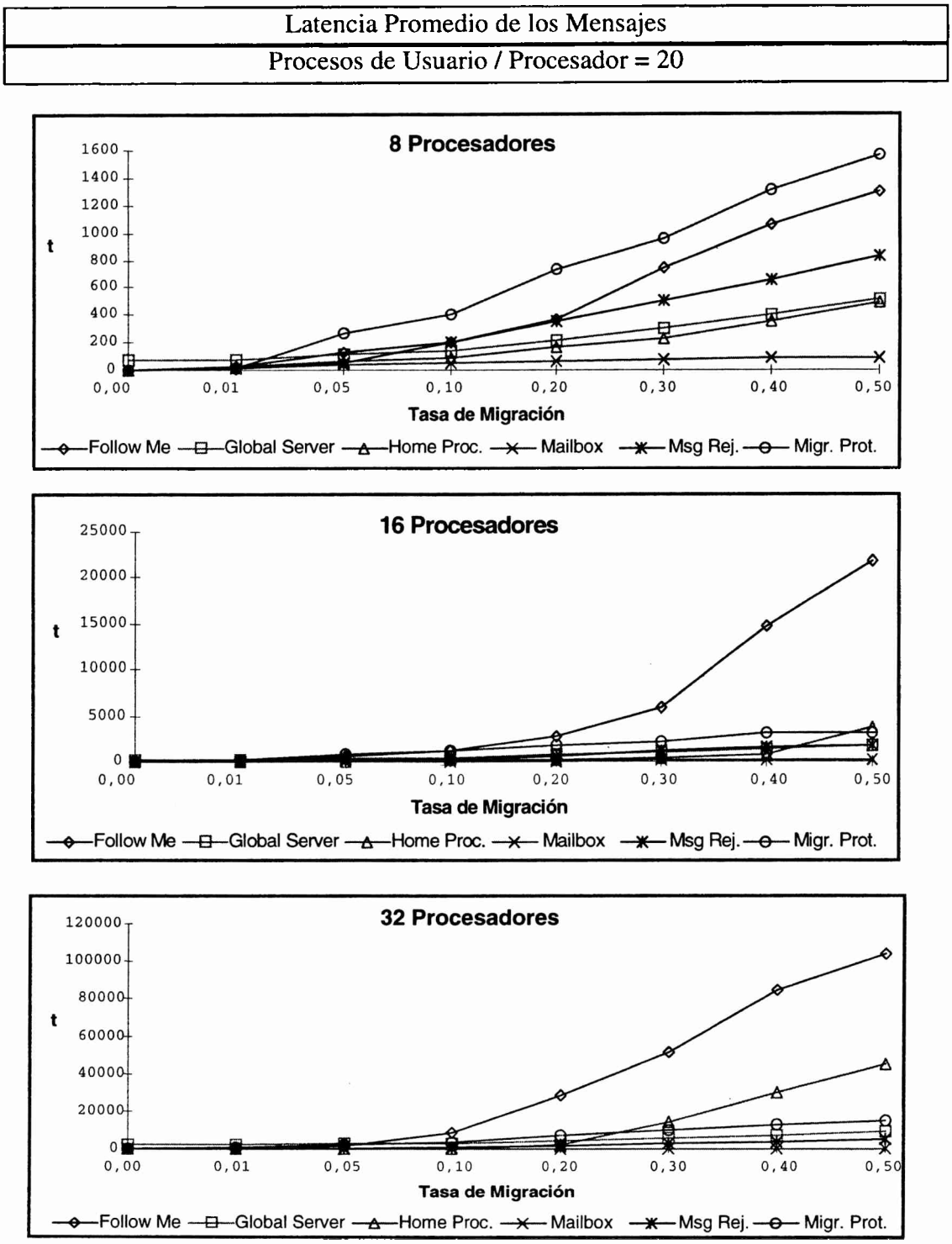


Figura 12.4: Latencia de Mensajes. 20 Procesos por Procesador.

El comportamiento de las políticas es similar al que se encuentra en el caso de asignar en promedio 5 procesos de usuario por procesador, aunque para 8 procesadores los resultados obtenidos tienen una distribución algo distinta a los demás. En el caso de ejecutar 20 procesos

de usuario en 8 procesadores hace que se tenga mucha carga de procesamiento tanto por la ejecución de los procesos de usuario como por la gestión de los mensajes de toda la aplicación que se reparten entre los 8 procesadores.

Se puede apreciar una diferencia notable comparando los valores obtenidos que se muestran en la Fig. 12.4 con respecto a los que se muestran en la Fig. 12.3. Esta misma característica se puede encontrar en la cantidad de retransmisiones promedio por mensaje, y la cantidad de retransmisiones afecta en forma directa el tiempo de latencia de los mensajes: cuantas más retransmisiones sean necesarias para un mensaje, más tiempo transcurrirá entre el envío y la recepción del mensaje. En la sección que sigue se explica por qué hay mayor cantidad de retransmisiones de los mensajes cuando se asignan más procesos de usuario a cada procesador. Las más afectadas en los resultados obtenidos por este comportamiento son las políticas Follow Me y Message Rejection.

La diferencia más significativa con respecto a los resultados obtenidos por simulación de las políticas en [Hey95], corresponde a las magnitudes de los valores de latencia. Esta diferencia era de esperar, ya que en la implementación paralela se tienen las mediciones realizadas sobre la red de comunicaciones real y por lo tanto se está considerando la carga de la red de comunicaciones y también las colisiones de los mensajes. Por ejemplo, para 5 procesos de usuario por procesador y 8 procesadores, los valores de latencia variaban entre 2 y 20 unidades de tiempo, y para 20 procesos de usuario por procesador y 64 procesadores, los valores variaban entre 5 y 60 unidades de tiempo.

En términos de comparación entre políticas y el comportamiento de cada una encontrado en [Hey95], los comportamientos se mantienen similares a lo que se encontró en la implementación paralela, hecha la salvedad de las magnitudes. Las diferencias más notables son:

- Las políticas Global Server y Migration Protocol tienen en la simulación tiempo promedio de latencia constante, no así en la implementación paralela.
- La mayoría de las políticas analizadas según las simulaciones llegan a un punto de *saturación* después del cual los valores de las latencias no cambian demasiado. Esta saturación se produce para valores de tasa de migración de aproximadamente 0.05. En la implementación paralela, para algunas políticas como por ejemplo Home Processor se sigue incrementando la latencia cuando se incrementa la tasa de migración.

En ambos casos de los enumerados anteriormente, siempre está presente en la implementación paralela la carga de los procesadores y de la red de comunicaciones como factor de peso en las diferencias. Como en el caso de las diferentes magnitudes, dado que se están tomando mediciones en tiempo real, es de esperar que las políticas no sean independientes de la carga que significa migrar más procesos de usuario en términos de cómputo y comunicación que se agrega por cada migración.

12.3 Carga de la Red de Comunicaciones: Cantidad de Retransmisiones

En esta sección se muestran y comentan los resultados obtenidos en la implementación paralela con respecto a la cantidad de retransmisiones de mensajes entre procesos de usuario. El cálculo de este índice se realizó como se detalla en la sección de Cálculo de los Indices de

Evaluación. En el eje Y de cada gráfica se muestra la cantidad promedio de retransmisiones de los mensajes. Primero se muestran los resultados obtenidos con 5 procesos de usuario por procesador, en la Fig. 12.5, y luego con 20 procesos de usuario por procesador, en la Fig. 12.6.

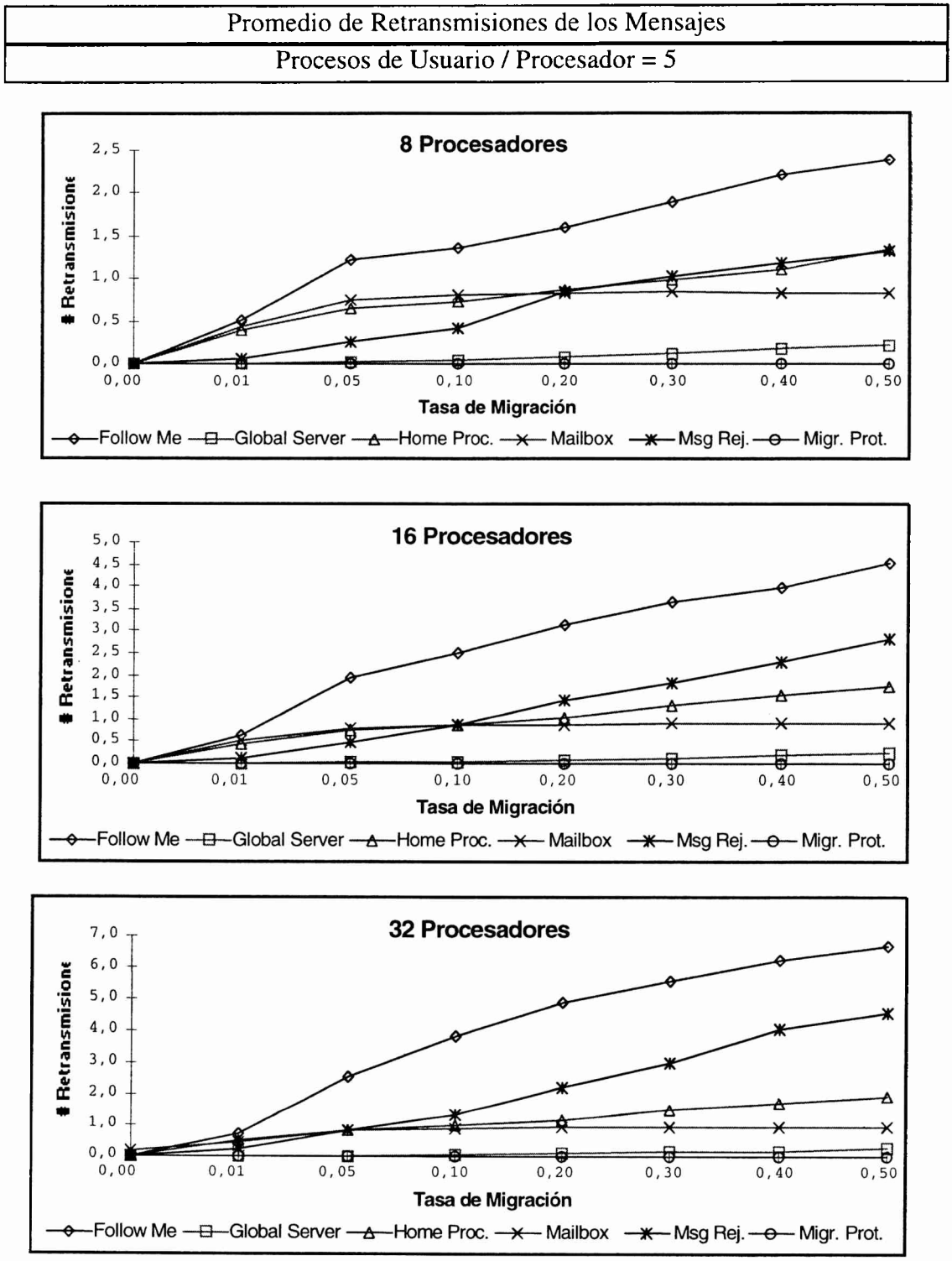


Figura 12.5: Retransmisiones. 5 Procesos por Procesador.

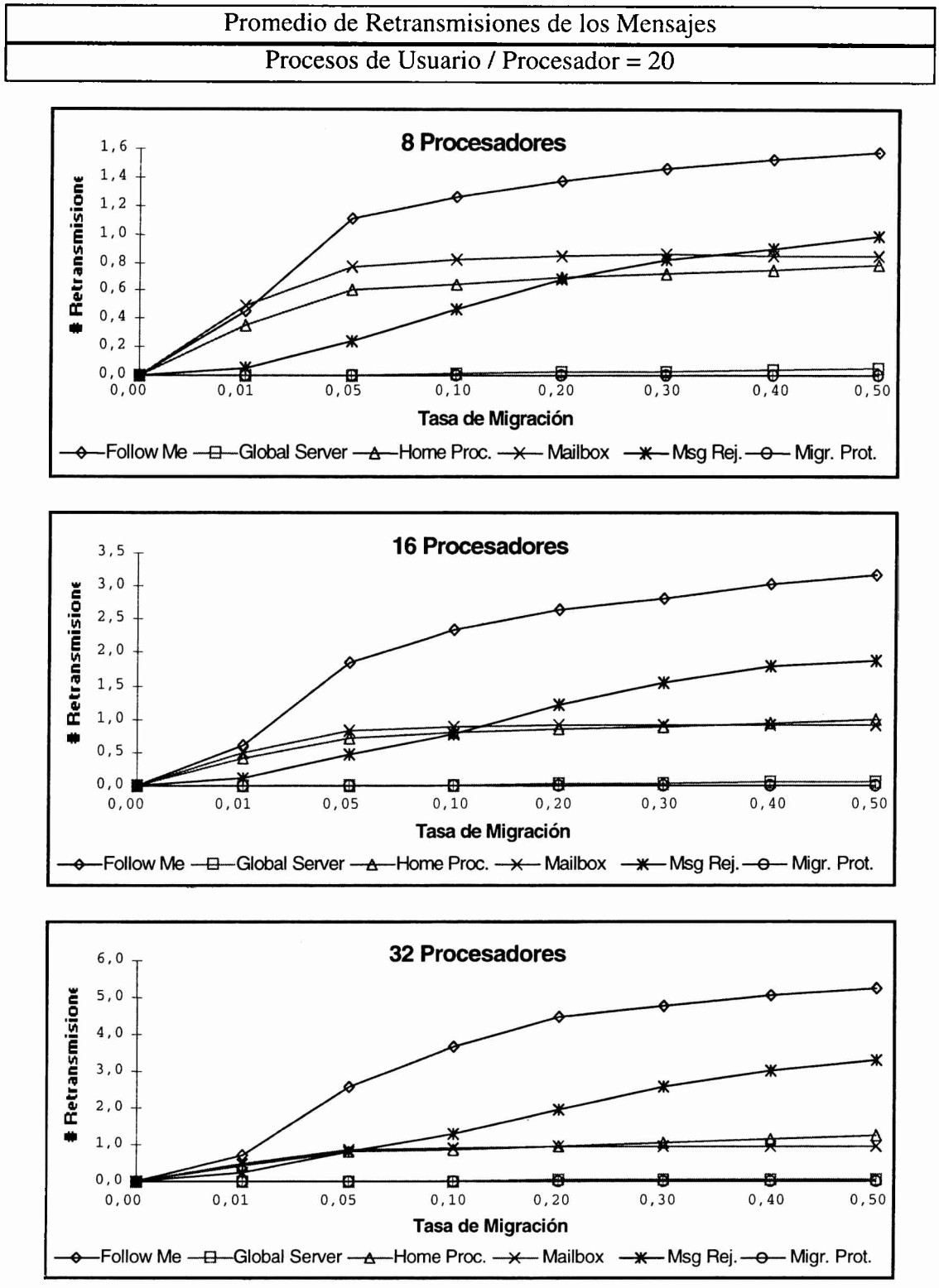


Figura 12.6: Retransmisiones. 20 Procesos por Procesador.

El comportamiento de los resultados en la Fig. 12.5 es casi igual para las tres cantidades de procesadores (8, 16, y 32). En todos los casos la política Follow Me es la que produce mayor cantidad de retransmisiones, y esto es lo que hace que los mensajes tengan más tiempo de latencia que en todas las demás políticas. Como era de esperar, la política

Global Server casi no produce retransmisiones, y la política Migration Protocol no produce ninguna retransmisión.

En la Fig. 12.6 se nota que, como para 5 procesos de usuario por procesador, los comportamientos de las políticas son similares para 8, 16 y 32 procesadores. Nuevamente la mayor cantidad de retransmisiones corresponde a la política Follow Me.

Comparando los valores obtenidos para 5 y 20 procesos de usuario por procesador, la única diferencia se nota en los valores, no en el comportamiento relativo de las políticas entre sí. En general, la cantidad de retransmisiones de mensajes cuando se asignan más procesos de usuario a cada procesador es menor, cuando puede llegar a esperarse lo contrario.

Lo que sucede con respecto a la cantidad de retransmisiones cuando se asignan más procesos de usuario a cada procesador puede explicarse de la siguiente manera: la cantidad de mensajes que se generan por unidad de tiempo no depende de la cantidad de procesos de usuario que se asignen por procesador, porque todos los procesos de usuario tienen los mismos requerimientos de cómputo y de comunicaciones. Tener más procesos de usuario por procesador implica que el procesador estará ocupado durante un período más extenso de tiempo, pero no implica que habrá más mensajes por unidad de tiempo.

En el caso de la experimentación realizada, ejecutar una vez por un *quantum* de tiempo (time slice) 20 procesos de usuario en un procesador, equivale a ejecutar 4 veces por un *quantum* de tiempo 5 procesos de usuario. Esto es así porque en cada *quantum* los procesos de usuario se comportan de igual manera en cuanto a la utilización de CPU y generación de mensajes. Como las migraciones se generan en función de los mensajes, la cantidad de migraciones por unidad de tiempo permanecerá similar aunque se asignen más procesos a cada procesador. Cada vez que se genera una migración, la cantidad de procesos de usuario a elegir es mayor si hay más procesos de usuario en cada procesador. Por lo tanto, cada proceso de usuario es más *estable* en cada procesador, permanece por más tiempo una vez que ha sido asignado a un procesador. Esta característica es más notable para las políticas Follow Me y Message Rejection, que son las que no tienen ningún mecanismo para evitar retransmisiones de mensajes.

La mayor cantidad de retransmisiones para menor cantidad de procesos de usuario asignados a cada procesador también puede explicar por qué las magnitudes de latencia de mensajes para 5 procesos de usuario por procesador son mayores que para 20 procesos de usuario por procesador. Se observa que la relación de cantidad de retransmisiones - latencia de mensajes, es directa. A mayor cantidad de retransmisiones se tendrá mayor tiempo de latencia. El tiempo de latencia de cada mensaje es dominado por el tiempo de comunicación de las retransmisiones que sean necesarias para el mensaje. La misma comparación de resultados de retransmisiones y latencia de mensajes puede realizarse entre la Fig. 12.3 y la Fig. 12.5 y también entre la Fig. 12.4 y la Fig. 12.6.

Excepto por el comportamiento de las retransmisiones cuando se asignan más procesos de usuario, los valores obtenidos de retransmisiones de mensajes son similares a los que se dan en [Hey95]. La diferencia nuevamente se basa en la carga de la máquina paralela, en este caso con respecto a la utilización de CPU. En las simulaciones no se tiene en cuenta si los procesos de usuario pueden ser ejecutados o no, solamente se tiene en cuenta la cantidad

de procesos de usuario. En este sentido, es como si los procesadores fueran lo suficientemente potentes como para ejecutar más rápido, cuando se le asignan más procesos de usuario.

12.4 Carga de la Red de Comunicaciones: Cantidad de Mensajes de Control

En esta sección se muestran y comentan los resultados obtenidos en el entorno de experimentación de migración con respecto a la cantidad de mensajes de control por mensaje entre procesos de usuario. En el eje Y de cada gráfica se muestra la cantidad promedio de mensajes de control por mensaje de procesos de usuario.

La Fig. 12.7 muestra los resultados obtenidos asignando inicialmente 5 procesos de usuario por procesador. Todos los resultados tienen características semejantes referentes al comportamiento de cada política para 8, 16 y 32 procesadores. La política Migration Protocol es la que presenta mayor cantidad promedio de mensajes de control por mensaje entre procesos de usuario, y esto se debe a la cantidad de mensajes de control que se generan para cada migración de un proceso de usuario.

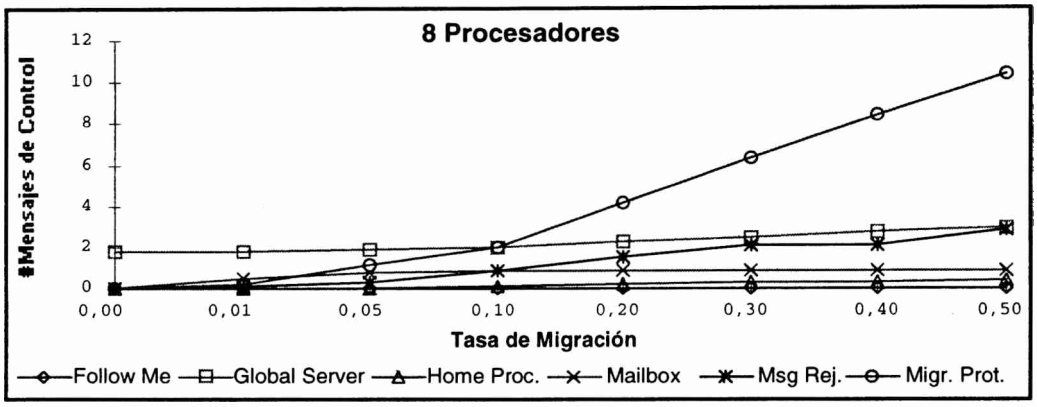
La política Message Rejection es la que produce más mensajes de control después de la política Migration Protocol, aunque se mantiene en valores cercanos a los obtenidos en la política Global Server para tasas de migración mayores de 0.10.

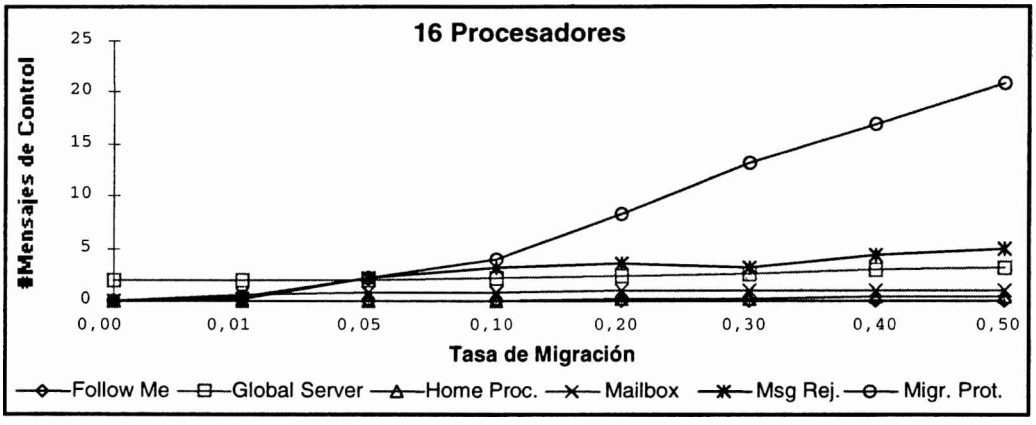
La cantidad de mensajes genrados en la política Global Server es más o menos constante en todos los casos. Cuando la cantidad de migraciones (tasa de migración) aumenta, tiene un leve incremento en la cantidad de mensajes de control. Esto se debe a las migraciones que se deben actualizar en el Servidor Global, y también es posible que se produzca un incremento leve de los rechazos de mensajes.

Las políticas Home Processor y Mailbox casi no generan mensajes de control, o muy pocos. Se debe recordar que la cantidad de mensajes de control está dada con respecto a la cantidad de mensajes de usuario. Por lo tanto en este contexto "pocos" significa que la cantidad de mensajes de control es uno o más órdenes de magnitud menor que la cantidad de mensajes generados por la aplicación de usuario. La política Follow Me no genera ningún mensaje de control, lo cual viene dado por su especificación.

Puede ser interesante comparar estos resultados de la Fig. 12.7 con los que se obtienen en las mismas ejecuciones en cuanto a la cantidad de migraciones que se llevan a cabo de la Fig. 12.1. Para algunas políticas, como Migration Protocol y Message Rejection se puede ver la relación de mensajes de control generados por cada política. En el caso de la política Migration Protocol, la cantidad de mensajes de control por cada migración es mucho mayor. En el caso de la política Message Rejection, los mensajes de control se producen para rechazar un mensaje entre procesos de usuario, o sea que depende de que se envíe un mensaje al usuario que ha migrado. Además, el procesador que recibe el rechazo (NACK) actualizará la información que se refiere a la nueva ubicación del proceso de usuario migrado. Por lo tanto la cantidad de mensajes de control que se generan en la política Message Rejection no es tan directamente proporcional con respecto a la cantidad de migraciones que se llevan a cabo para cada proceso de usuario.

Promedio de Mensajes de Control
Procesos de Usuario / Procesador = 5





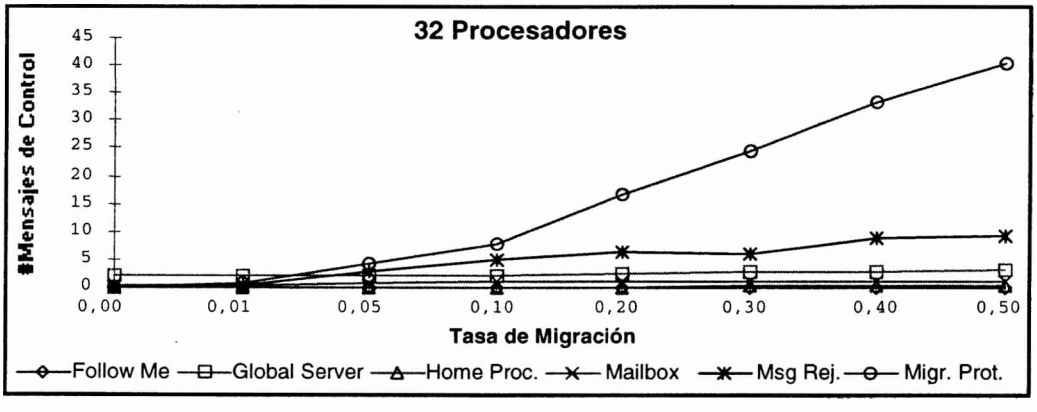
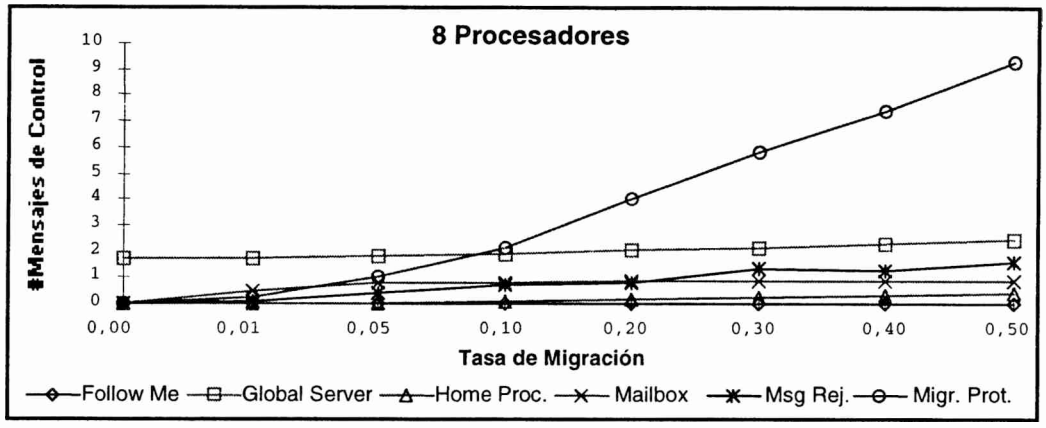
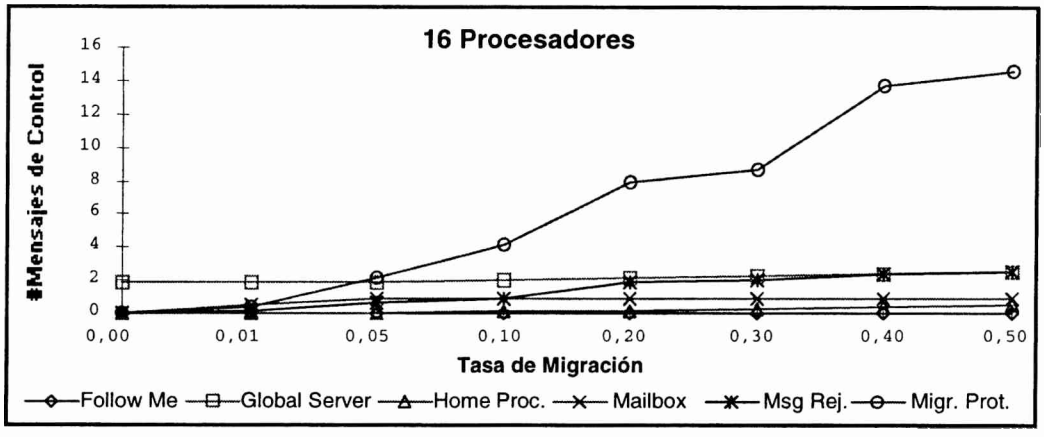


Figura 12.7: Mensajes de Control. 5 Procesos por Procesador.

En la Fig. 12.8 se muestran los promedios de mensajes de control obtenidos cuando se asignan 20 procesos de usuario en cada procesador. Los resultados obtenidos para 8 procesadores siguen las líneas generales de lo que se obtiene con 5 procesos de usuario promedio asignados a cada procesador. Para 16 y 32 procesadores los resultados muestran un comportamiento similar para todas las políticas excepto para la política Migration Protocol. En la mayoría de los casos (tasa de migración y cantidad de procesadores), la política Global Server es la que produce mayor cantidad de mensajes de control (por mensaje de usuario).

Promedio de Mensajes de Control
Procesos de Usuario / Procesador = 20





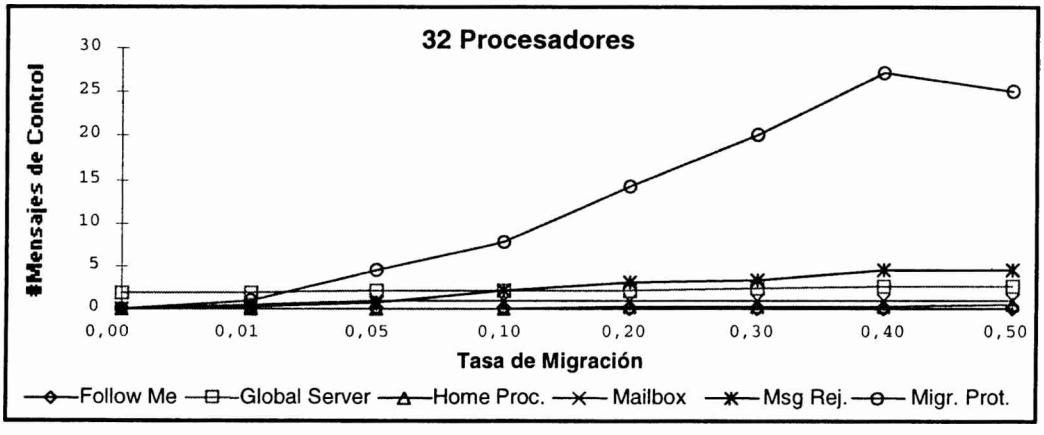


Figura 12.8: Mensajes de Control. 20 Procesos por Procesador.

En el caso de la política Migration Protocol, se pueden comparar estos resultados con los de cantidades de migraciones promedio de los procesos de usuario (Fig. 12.2), y se verá el mismo comportamiento. La explicación de los resultados obtenidos en la política Migration Protocol es, por lo tanto, la misma que para la cantidad de migraciones: los procesos de usuario migran menos porque las migraciones que genera el proceso *migman* no se pueden llevar a cabo debido al tiempo que se necesita para cada una de ellas. En ese tiempo, los

demás procesos continúan ejecutándose y pueden llegar al final de su trabajo con migraciones que no se realizaron.

Previamente se explicó la dependencia entre la cantidad de retransmisiones y la latencia de los mensajes que se obtienen en la política Follow Me. La conclusión a la que se llegó es que la cantidad de retransmisiones implican una mayor latencia de los mensajes. Quizás sea de esperar que suceda lo mismo con la cantidad de mensajes de control y la latencia. De hecho es lo que sucede con las políticas Message Rejection y Global Server: la latencia de los mensajes tiene aproximadamente el mismo comportamiento que los mensajes de control. El caso excepcional más significativo es el de la política Migration Protocol, a pesar de generar una gran cantidad de mensajes de control esto no afecta demasiado a la latencia de los mensajes (Fig. 12.7 y Fig 12.3, Fig. 12.8 y Fig. 12.4). Este comportamiento se debe a que los mensajes de control son relativamente mucho más cortos que los mensajes de datos, y por lo tanto su influencia en cuanto a la carga que introducen en la red de comunicaciones no es tan grande.

Comparando estos resultados con los obtenidos en [Hey95], nuevamente las características generales se repiten. Las mayores diferencias se encuentran en las políticas:

1. Global Server: en la simulación los valores se mantienen constantes, y en la experimentación sobre la máquina paralela los valores se incrementan levemente.
2. Message Rejection: los valores que se obtienen por simulación de esta política son siempre menores que para la política Global Server, mientras que en la experimentación sobre la máquina paralela los resultados de Message Rejection suelen ser similares o mayores que los de Global Server.

En los dos casos mencionados, las diferencias pueden explicarse como la mayoría de las comparaciones entre la simulación y la implementación paralela, por la carga de procesadores y/o red de comunicación.

En el primer caso de la enumeración anterior, los valores constantes para la política Global Server se pueden explicar, ya que a medida que los procesos migran más frecuentemente, es más probable que la información del proceso Servidor Global quede desactualizada. En la simulación, la notificación de los cambios llega a este proceso con el retraso de tiempo que implica realizar la comunicación de los cambios. No se tiene en cuenta la carga de la red de comunicaciones. En la implementación paralela se tiene en cuenta esta carga, incluyendo las posibles colisiones de salida del procesador desde el que se envían las notificaciones. También con relación a las colisiones y la carga de la red, las respuestas del Servidor Global llegarán con más retraso a los procesadores desde los cuales se realizaron los requerimientos. Por estas dos razones (desactualización de la información del Servidor Global y tiempo de llegada de la respuesta desde el Servidor Global), es más probable que en la experimentación sobre la máquina paralela se generen más rechazos de mensajes (NACKs). Estos NACKs son los que se cuentan en la experimentación y no estarían presentes con el mismo peso en la simulación.

En el segundo caso de la enumeración anterior, la relación entre los valores de las políticas Global Server y Message Rejection, es explicable por el tiempo que lleva a un requerimiento de mensaje llegar a un determinado procesador. En el contexto de la experimentación sobre la máquina paralela el requerimiento de mensaje debe superar la congestión de la red de comunicaciones. Esto implica que le llevará más tiempo llegar a un procesador. Durante este tiempo añadido, el proceso de usuario destino del mensaje podría

migrar, y eso implicaría un mensaje de rechazo (NACK). La congestión de la red afecta más a la política Message Rejection porque es la que produce más retransmisiones (si se la compara con la política Global Server) y de hecho, los resultados comparativos entre ambas políticas podrían analizarse a partir de los valores de las retransmisiones.

12.5 Cantidad de Mensajes de Control y Cantidad de Retransmisiones

La idea básica de monitorizar la cantidad de mensajes de control y la cantidad de retransmisiones es la de medir la carga en la red de comunicaciones que produce cada política. Tanto las retransmisiones como los mensajes de control constituyen la sobrecarga (*overhead*) de la red de comunicaciones. Ninguno de estos dos tipos de mensajes estaría presente si no se implementara la gestión necesaria para preservar la consistencia en el arribo de los mensajes entre procesos de usuario.

Tal como se encontró en [Hey95], hay una relación inversa de comportamiento entre las cantidades de retransmisiones de mensajes y los mensajes de control. Las políticas que generan mayor cantidad de mensajes de control son las que producen menor cantidad de retransmisiones, y viceversa.

Para tasas de migración relativamente bajas, la cantidad de mensajes de control generadas por la política Global Server es inicialmente más alta que en cualquier otra política (Fig. 12.7 y Fig. 12.8). También se puede observar que aún en el caso de pocos procesadores y pocos procesos de usuario asignados a cada procesador, la cantidad de mensajes de control que se generan en la política Migration Protocol tiende a crecer con factores de multiplicación muy elevados en relación con la tasa de migración. Por otro lado, la política Global Server casi no produce retransmisiones aún cuando la tasa de migraciones es muy alta (0.4 ó 0.5), y en la política Migration Protocol no se produce ninguna retransmisión, independientemente de la cantidad de procesadores, procesos de usuario asignados a cada procesador y tasa de migración.

Por la definición de la política Follow Me, no se generan mensajes de control, y toda la gestión que se realiza para conservar la consistencia en el arribo de los mensajes se lleva a cabo por medio de las retransmisiones. Por esta razón, la cantidad de mensajes de control para esta política es siempre cero independientemente de la cantidad de procesadores y procesos de usuario asignados a cada procesador. Por el contrario, la mayor cantidad de retransmisiones que se observan corresponden a esta política.

Con respecto a la cantidad de mensajes de control que produce la política Migration Protocol se debe aclarar que la experimentación se realizó con el peor caso posible de interconexión entre procesos de usuario para esta política, y que en principio no afecta tanto a las demás. Todos los procesos de usuario se consideran conectados entre sí, y por lo tanto una migración implica varios mensajes de control por cada procesador:

1. Envío de "inicio de migración" del proceso de usuario.
2. Recepción de la cantidad de mensajes enviados desde cada procesador.
3. Envío de "fin de migración" del proceso de usuario.

En este contexto de interconexión de procesos de usuario, es de esperar que la política Migration Protocol genere mayor cantidad de mensajes de control que todas las demás políticas.

12.6 Escalabilidad

La forma en que una mayor cantidad de procesadores afecta el rendimiento de cada política puede analizarse para los índices de latencia de mensajes, cantidad de mensajes de control y cantidad de retransmisiones que producen.

Según los resultados que se han mostrado, se puede apreciar que todas las políticas incrementan la latencia cuando se utilizan más procesadores. Si bien es justificado este incremento de latencia por la carga de la red de comunicaciones y de los procesadores, no debería impedir el incremento de la velocidad de ejecución de las aplicaciones de usuario. En las simulaciones realizadas en [Hey95] también se incrementan las latencias, pero en forma menos significativa que en la experimentación sobre la máquina paralela.

Con respecto a la escalabilidad de las políticas y su relación con la latencia, se podría tomar como ejemplo el caso de 8 procesadores con 5 procesos de usuario por procesador (Fig. 12.3) y 32 procesadores con 5 procesos de usuario por procesador (Fig. 12.3). La cantidad de procesadores en este caso se multiplica por 4 y no se cambia ningún otro parámetro. Observando los valores de las latencias, se puede ver que el factor de multiplicación de los tiempos es como mínimo 4, y puede llegar a valores de hasta 10 para la política Home Processor o, peor aún, de hasta 15 para la política Follow Me. Las demás políticas muestran factores de multiplicación cercanos a 4. El mismo caso analizado por simulación [Hey95], muestra un decremento de rendimiento que varía por un factor de entre 1.5 y 2.

La cantidad de retransmisiones muestran un comportamiento aceptable para todas las políticas, siendo la peor en este sentido la política Follow Me. Aunque la cantidad de retransmisiones aumentan cuando se utilizan más procesadores, en ningún caso los factores de incremento son iguales ni superan al factor por el que se incrementa la cantidad de procesadores. En todos los casos la política Migration Protocol es la que presenta mejor escalabilidad de latencia seguida de la política Global Server.

Como ya se ha comentado, los valores obtenidos de cantidad de mensajes de control están directamente relacionados con los que se refieren a la cantidad de retransmisiones de mensajes. Esto sigue siendo válido para observar la escalabilidad de cada política con respecto a los mensajes de control y las retransmisiones de mensajes. En el caso de la cantidad de mensajes de control, habría que remarcar que cuando los procesos de usuario están todos interconectados entre sí, tal como se realizó la experimentación, la política Migration Protocol es la más afectada por el incremento de la cantidad de mensajes de control.

12.7 Tiempo de Reacción

En la mayoría de las políticas a excepción de la política Migration Protocol, el tiempo de reacción depende más o menos de la implementación, pero siempre se puede relacionar con los mensajes entre procesos de usuario o con los mensajes de control.

Dependiendo de la implementación, en las políticas Global Server y Home Processor, la nueva ubicación del proceso que migra tiene que ser actualizada en el Servidor Global o en el Home Processor respectivamente, antes de que se produzca la migración. También se puede enviar la nueva ubicación después de la migración. En ambos casos la diferencia se encontraría en el tiempo que lleva comunicar un cambio de ubicación al Servidor Global o al Home Processor.

En la política Mailbox, el proceso de usuario debe esperar la llegada de los mensajes que ha requerido de su buzón. Usualmente hay a lo sumo una recepción pendiente y por lo tanto se debe esperar la llegada de un mensaje antes de que el proceso de usuario pueda ser migrado.

Los procesos de usuario en las políticas Follow Me y Message Rejection reaccionan inmediatamente, todos los eventos se llevan a cabo en el mismo procesador donde se generan.

La política Migration Protocol es la que tiene el mayor tiempo de reacción. Se debe cumplir una sucesión de pasos desde que se elige un proceso de usuario para migrar hasta que se envía a otro procesador bajo esta política. Las medidas que se han tomado en la experimentación sobre la máquina paralela muestran que el tiempo de reacción para la política Migration Protocol varía entre el tiempo de 20 y 30 mensajes de usuario. Esto significa que si el proceso de usuario fuera migrado de forma inmediata, podría enviar entre 20 y 30 mensajes. De la misma manera, mientras se está llevando a cabo el protocolo de migración para un proceso de usuario, podría recibir entre 20 y 30 mensajes de otros procesos de usuario.

12.8 Patrón de Comunicaciones

Si un algoritmo no modifica adecuadamente el patrón de comunicación en la red, no será capaz de obtener el balance de las comunicaciones. Se debe lograr que todos los procesadores tengan aproximadamente los mismos requerimientos de comunicaciones (entrada y salida de mensajes o datos).

La política Migration Protocol, después del intercambio de información entre los procesadores, realiza una modificación efectiva del patrón de comunicaciones. Es decir que por un período relativamente corto de tiempo las comunicaciones entre procesadores son muchas, y luego se cambia el patrón. La cantidad, así como el patrón que siguen los mensajes de control durante el intercambio de información dependen de la conexión entre procesos de usuario.

En la política Message Rejection también se cambia el patrón que siguen los mensajes entre los procesadores, porque los mensajes de control (NACKs) generan la actualización de información en el procesador que emite los mensajes para el proceso de usuario.

El patrón de comunicaciones entre procesadores permanece más o menos sin alteraciones en la política Follow Me. Más aún, se agrega carga entre cada par de procesadores a medida que los procesos de usuario son migrados, porque los mensajes pasan por los mismos procesadores en los que los procesos de usuario estuvieron asignados. La excepción se produce cuando un proceso de usuario vuelve a estar a un procesador donde ya

estuvo previamente. En la mayoría de los casos, a mayor cantidad de migraciones suele ser mayor la carga de mensajes entre procesadores.

Independientemente de la ubicación de un proceso de usuario que ha migrado, las políticas Home Processor y Mailbox mantienen la carga alrededor del Home Processor y del procesador que contiene al buzón para migración respectivamente. Desde estos procesadores hacia los demás pueden aparecer mensajes debido a las retransmisiones. Los mensajes continúan llegando al Home Processor y al procesador que contiene los buzones de los procesos de usuario. Por lo tanto, la carga entre procesadores se mantiene o aumenta para ambas políticas.

La política Global Server produce una sobrecarga de transferencia de mensajes alrededor del procesador en el que se asigna el Servidor Global. Esta sobrecarga se produce por el envío de requerimientos de (a) la ubicación de procesos de usuario, o (b) actualización de la información que contiene el Servidor Global. La sobrecarga desde el procesador en el cual está ubicado el Servidor Global hacia los demás procesadores se produce por las respuestas sobre la ubicación de procesos de usuario que se envían a cada procesador.

12.9 Conclusiones del Entorno para Experimentación

Para poder balancear la carga de procesamiento y de las comunicaciones en un DMPC es necesario contar con un mecanismo de migración dinámica de procesos en tiempo de ejecución. Para lograr que los procesos puedan ser migrados de forma dinámica, se deben elegir e implementar

- la política de migración,
- el mecanismo de migración física, y
- la política para preservar la consistencia en el arribo de los mensajes.

Aunque este trabajo se ha concentrado en el tercer punto, las posibilidades de elección de cada política y/o mecanismo son múltiples y no necesariamente independientes.

Se necesita contar con un entorno de experimentación en una máquina paralela, donde no solamente sea posible implementar cada política y/o mecanismo que se proponga, sino también poder diseñar variaciones, probarlas con distintas aplicaciones de usuario, y monitorizar la ejecución.

Se muestra el diseño, algunos detalles de la implementación, y también la instrumentación y comparación de seis políticas que conservan la consistencia en el arribo de los mensajes. Se ha utilizado para ello una máquina paralela sobre la cual experimentar. Las políticas utilizadas para la preservación de la consistencia en el arribo de los mensajes fueron: Follow Me, Global Server, Home Processor, Mailbox, Message Rejection y Migration Protocol. En cada una de ellas, los procesos que migran siguen recibiendo los mensajes destinados a ellos de forma independiente de su ubicación en la red.

El entorno de experimentación que se presenta permite la implementación de distintas alternativas para las políticas de migración, distintos mecanismos de migración física de procesos de usuario y distintas políticas de migración de procesos de usuario. También es posible cambiar otras características del entorno de ejecución, como la política de planificación de la ejecución (scheduling).

El entorno de experimentación presentado permite la definición de aplicaciones de usuario sintéticas por medio de: (a) Cantidad de procesos, (b) patrón de interconexión entre procesos, y (c) comportamiento de cada proceso. En el contexto de la ejecución de aplicaciones de usuario es muy importante el comportamiento de cada proceso, y por lo tanto se hace necesaria una definición más detallada de lo que significa el punto (c) anterior. En el entorno de experimentación, cada proceso de usuario se define por su (c.1) frecuencia de comunicación, o requerimiento de comunicación (mensajes), y su (c.2) carga de cómputo o requerimiento de cómputo (utilización de CPU).

Con la definición de aplicaciones de usuario sintéticas se evita tener que elegir, diseñar e implementar un conjunto de aplicaciones de usuario reales que deban "representar" a todas las posibles. También hace posible definir aplicaciones de usuario (sintéticas) rápidamente, y no desviar la atención de las características que se quieran evaluar de la migración dinámica. Otra de las ventajas de esta decisión es que se evita entrar en detalles propios de la máquina paralela que se utilice en cuanto a la asignación de memoria para los procesos y los cambios de contexto de ejecución. Se puede entonces abstraer los detalles, y trabajar en el contexto de los DMPCs (*Distributed Memory Parallel Computers*) en general.

La experimentación con la implementación paralela se realizó variando los parámetros: (a) cantidad de procesadores, (b) cantidad de procesos de usuario de la aplicación, y (c) tasa de migración de los procesos de usuario. Todas las políticas presentadas para la preservación de la consistencia en el arribo de los mensajes cuando los procesos migran dinámicamente fueron monitorizadas y analizadas en base a estos parámetros.

Los índices de rendimiento que se definieron para el análisis de las políticas de preservación de la consistencia en el arribo de los mensajes fueron: (a) latencia promedio de los mensajes, (b) sobrecarga de la red de comunicaciones que produce (en cantidad promedio de mensajes de control y retransmisiones de mensajes), (c) tiempo de reacción de los procesos de usuario, (d) modificación del patrón de comunicaciones, y (e) escalabilidad de cada política. La sobrecarga de la red de comunicaciones se definió en términos de (b.1) cantidad de mensajes de control que se generan, y (b.2) cantidad de retransmisiones de mensajes que se realizan.

El entorno de experimentación en una máquina paralela y la ejecución de aplicaciones de usuario han permitido tener en cuenta características tales como la carga de la red de comunicaciones y de los procesadores, considerando condiciones reales de tráfico de mensajes y de procesos.

Ninguna de las políticas propuestas para resolver el Problema de Arribo de Mensajes ha resultado ser suficientemente buena para todas las situaciones. En todo caso, es posible descartar inicialmente algunas de ellas o proponer mejoras y/o combinaciones.

Dentro de las líneas abiertas de investigación con las políticas que se han propuesto, sería útil estudiar el impacto de distintas aplicaciones de usuario reales o no uniformes sobre cada política. Se podría variar el patrón de interconexión entre procesos de usuario y, por ejemplo, experimentar con aplicaciones que conectan los procesos en una malla (mesh), y/o extraer patrones de aplicaciones reales. También se podrían definir aplicaciones donde todos los procesos no fueran iguales, e introducir de esta manera desbalance de requerimientos de cómputo y comunicación entre los procesos de usuario.

En el contexto general de las políticas para conservar la consistencia en el arribo de los mensajes, las líneas abiertas de investigación que se pueden mencionar son:

- Diseño de políticas que incorporen las mejores características de cada una de las que se han propuesto, o nuevas.
- Nuevos índices de evaluación, que permitan obtener una mejor caracterización de las políticas. Asimismo, se podría buscar también un modelo formal de caracterización de las políticas, que permita realizar estimaciones de rendimiento fuera del rango en que se realizan las simulaciones (por ejemplo, para mayor cantidad de procesadores de los que se pueden utilizar en una máquina paralela).

Siempre se debe tener en cuenta que el objetivo consiste en encontrar una política que no introduzca una sobrecarga tal que impida obtener mejoras de rendimiento de un DMPC por introducir migración dinámica de procesos.

Con respecto a las líneas abiertas de investigación en el campo de la migración dinámica de procesos se pueden mencionar el diseño, implementación y evaluación de (a) políticas de migración y (b) mecanismos de migración dinámica. En todos los casos se deberían evaluar las relaciones o dependencias existentes, es decir cómo afecta una política a las demás. En este contexto, se podrían implementar y comparar las políticas y mecanismos que ya hay propuestos, así como también proponer otros.

Con respecto a la máquina paralela utilizada y la cantidad de procesadores disponibles, siempre es deseable tener un mayor rango de resultados en lo que respecta a la utilización de una mayor cantidad de procesadores. En este sentido, el límite físico de cantidad de procesadores de la máquina paralela será el límite para la experimentación.

Quizás el objetivo más general que aún quedaría por resolver consiste en identificar la relación costo/beneficio por la introducción de la migración dinámica. Como se ha explicado previamente, se deben resolver varios problemas que, en principio, se están analizando por separado. Si bien se han estudiado las soluciones propuestas para resolver el Problema de Arribo de Mensajes, en todos los casos se puede apreciar que las soluciones propuestas agregan (sobre)carga de cómputo y/o de comunicaciones (overhead) que se deben llevar a cabo. Esta sobrecarga no está presente en las aplicaciones de usuario cuando no se soporta migración dinámica.

Apéndice A: Análisis de Algoritmos

Como ya se explica en la introducción de este libro, se recurre a las máquinas paralelas para resolver una gran cantidad de aplicaciones debido a los requerimientos de cómputo. Para este tipo de aplicaciones, muchas veces se necesita conocer antes de llevar a cabo la implementación el tiempo de la ejecución del algoritmo. Por otro lado, tanto para estas aplicaciones como en general, es muy útil conocer el costo de la ejecución, y la eficiencia del algoritmo en la utilización de los recursos disponibles. También se necesita un mecanismo de evaluación y comparación de algoritmos para elegir entre varias alternativas de implementación.

Una vez que se ha desarrollado un nuevo algoritmo paralelo para resolver un problema, se suele evaluar en relación a tres criterios: (a) tiempo de ejecución, (b) cantidad de procesadores que necesita, y (c) costo [Akl89].

A.1 Tiempo de Ejecución

El tiempo de ejecución, o tiempo de ejecución paralelo se define como el tiempo que necesita un algoritmo para resolver un problema en una computadora paralela. Expresado de otra manera, es el tiempo que transcurre desde que el programa paralelo cominenza hasta que finaliza la ejecución del último procesador de la máquina paralela [Akl89] [Kum94].

Como paso previo a la implementación y monitorización del tiempo de ejecución de un programa paralelo, es usual llevar a cabo un análisis teórico del tiempo que requiere resolver el problema en una computadora (monoprocesador o paralela). Este análisis se enfoca en averiguar la cantidad de operaciones básicas o *pasos* que se ejecutarán en la computadora y que indican el tiempo de ejecución del algoritmo. Por esta razón, el análisis de algoritmos en este sentido se denomina también análisis de la complejidad temporal de los algoritmos (time-complexity analysis).

La cantidad de pasos de ejecución de un algoritmo queda normalmente especificada en función del tamaño del problema. Para una gran cantidad de problemas el tamaño es directamente la cantidad de datos que se deben procesar. Por ejemplo, para los problemas en los que se deben procesar vectores o matrices, el tamaño de la entrada es la cantidad de datos que los vectores o matrices contengan.

En las máquinas monoprocesador, los pasos de ejecución que normalmente se cuentan son los que corresponden a las operaciones computacionales básicas que se llevan a cabo dentro de un elemento de procesamiento: las aritmético-lógicas. En las máquinas paralelas los elementos de procesamiento deben comunicarse para llevar a cabo la ejecución del programa. Además, estas comunicaciones son importantes desde el punto de vista del tiempo que consumen. Por esta razón, en los algoritmos paralelos se cuentan, además de las operaciones computacionales, las operaciones de *ruteo* [Akl89] o de comunicaciones de los datos entre los elementos de procesamiento.

En las computadoras monoprocesador las operaciones se ejecutan de forma secuencial,

por lo tanto el tiempo de ejecución es proporcional a la cantidad de operaciones. En las computadoras paralelas varias operaciones se pueden ejecutar al mismo tiempo (simultáneamente) y por lo tanto el tiempo de ejecución total no necesariamente está determinado por la cantiodad de operaciones. En este contexto, entonces, es más apropiado expresar el tiempo de ejecución en función de la cantidad de pasos de ejecución que se deben realizar. Un paso de ejecución se define como un conjunto de operaciones computacionales y/o de comunicaciones que se ejecutan simultáneamente. De esta manera, el tiempo de ejecución de un programa es proporcional a la cantidad de pasos de ejecución, lo cual a su vez se obtiene en función (depende) de la cantidad de procesadores y de la red de interconexión entre ellos.

A.2 Límites

La cantidad de pasos de ejecución de un algoritmo para un problema dado se tiene en cuenta en dos aspectos básicos: (a) para saber si es posible diseñar un nuevo algoritmo que tenga mejor tiempo de ejecución, y (b) para saber si es el algoritmo más rápido desarrollado para el problema que resuelve [Akl89].

El mejor tiempo de ejecución se obtiene cuando se ejecuta en la mínima cantidad posible de pasos de ejecución para un problema dado. Para muchos problemas, la cantidad mínima de pasos a ejecutar puede ser conocida o estimada. Por ejemplo, si se debe llevar a cabo la multiplicación de dos matrices de nxn elementos, como la matriz resultado tiene n^2 elementos, esta cantidad es la que se considera como mínima para un algoritmo de multiplicación de matrices. Si un algoritmo resuelve un problema en la cantidad mínima de pasos, se dice que es *óptimo*, esto significa que es el más rápido posible. La cantidad mínima de pasos de ejecución de conoce como el *límite inferior* de la cantidad de pasos de ejecución de un problema.

Independientemente de que se haya alcanzado el límite inferior de pasos de ejecución con un algoritmo, si este algoritmo es más rápido que todos los existentes se dice que ha establecido un nuevo *límite superior* para el problema. Por lo tanto, para que un nuevo algoritmo sea aceptado para el mismo problema, la cantidad de pasos de ejecución de este nuevo algoritmo no debería superar el *límite superior*.

Dada la complejidad de las expresiones que se pueden encontrar para la cantidad de pasos de ejecución, se intenta enfocar con atención solamente el término dominante de esas expresiones. Es así que es muy común encontrar la frase "del orden de" en el contexto del análisis de algoritmos. Normalmente, se recurre a la notación que establece [Akl89] [Lei92] [Kum94]:

- Se define que la función $g(n)$ es *de orden de al menos* $f(n)$, denotado como $g(n)$ es $\Omega(f(n))$, si existen constantes positivas c y n_0 tales que $g(n) \geq cf(n)$ para todo $n \geq n_0$.

- Se define que la función $g(n)$ es *de orden de a lo sumo* $f(n)$, denotado como $g(n)$ es $O(f(n))$, si existen constantes positivas c y n_0 tales que $g(n) \leq cf(n)$ para todo $n \geq n_0$.

- Se define que la función $g(n)$ es $\Theta(f(n))$ si existen constantes positivas c_1, c_2, y n_0 tales que $c_1 f(n) \leq g(n) \leq c_2 f(n)$ para todo $n \geq n_0$.

En todos los casos, $g(n)$ es la función que expresa la cantidad de pasos de ejecución del algoritmo que se analiza. Los símbolos Ω, O y Θ se conocen también en la bibliografía como los símbolos de *orden de magnitud* [Lee94], en el contexto del análisis de la complejidad temporal de los algoritmos. Dado que esta notación se utiliza para conocer el comportamiento de la cantidad de pasos de ejecución a medida que el tamaño del problema aumenta, se la denomina también notación asintótica [Aho88] [Aho74].

Por ejemplo, si $g(n) = 2n^2 + 40$ es la función que define la cantidad de pasos de ejecución de un algoritmo para un problema de tamaño n, se puede afirmar que $g(n)$ es $O(n^2)$. En este caso, $f(n) = n^2$, y los valores de las constantes pueden ser $c = 3$, y $n_0 = 7$, porque $g(7) = 138 < 3x7^2 = 147$, y $g(n) < 3n^2$ con $n \geq 7$. La Fig. A.1 muestra la relación entre las funciones resultantes del análisis de $g(n)$ realizado en términos de O.

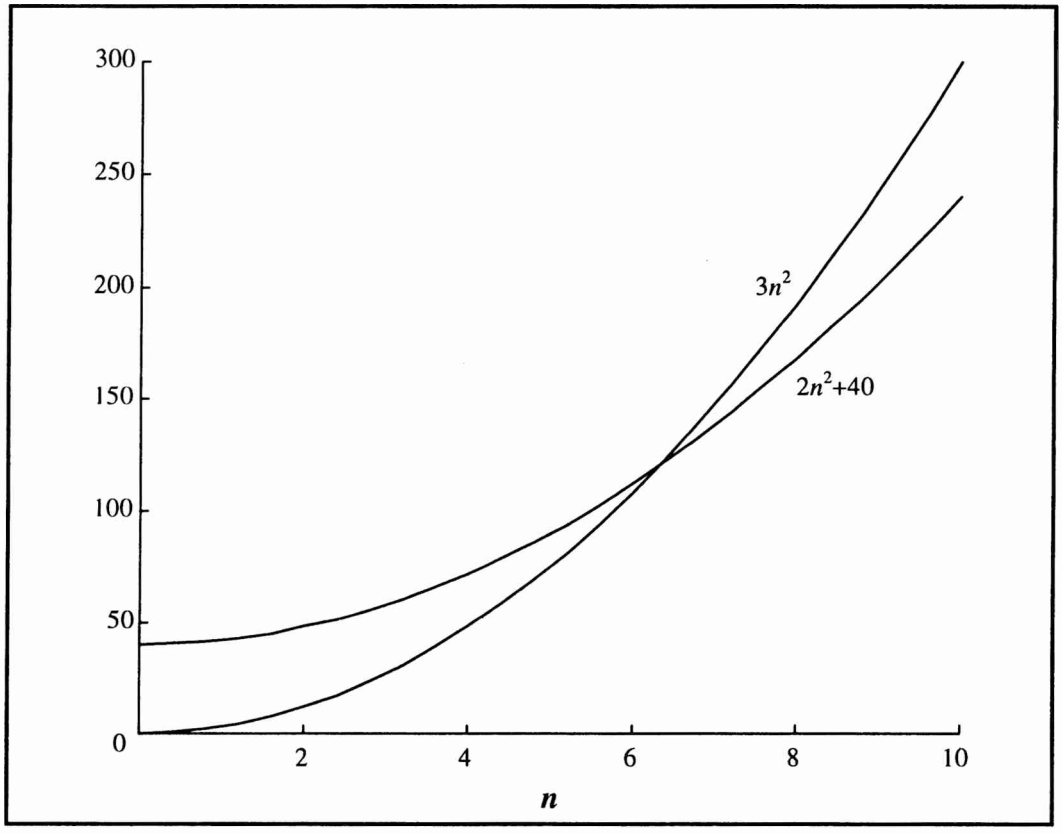


Figura A.1: Ejemplo para $O(\ )$.

En orden de menor a mayor en cuanto al tiempo de ejecución, las funciones de complejidad más utilizadas en la bibliografía suelen ser

$$1 \qquad \log(n) \qquad \log^2(n) \qquad \sqrt{n} \qquad N \qquad n\log(n) \qquad n^2 \qquad 2^n$$

donde $O(1)$ implica que el tiempo de ejecución no depende del tamaño del problema, y la base del logaritmo es irrelevante, aunque con frecuencia se asume $\log_2$. Las constantes que se asumen con el análisis en función de $O(\ )$ tienen relevancia solamente para tamaños de problema relativamente pequeños ($n < n_0$).

El análisis en función de $g(n)$ es $O(f(n))$ puede ser poco *preciso* en cuanto a la relación existente entre la función $g(n)$ y el tiempo de ejecución. Siguiendo la definición de O, si $g(n)$ es $O(f(n))$, esto no implica que no existen c_h constante positiva y $h(n)$ tales que $g(n) \leq c_h h(n) \leq cf(n)$ para todo $n \geq n_0$. En el caso de existir c_h y $h(n)$ que cumplan tales condiciones, es más preciso afirmar que $g(n)$ es $O(h(n))$ en vez de $g(n)$ es $O(f(n))$, pero esto no está obligado por la definición de O. Por ejemplo la función $g(n) = 2n^2 + 40$ dada enteriormente es $O(n^2)$, pero también es cierto que $g(n)$ *es* $O(n^3)$. Aunque la tendencia general es expresar el orden de magnitud más cercano al real, que en este caso sería $O(n^2)$, la definición de $O(\ )$ no es precisa en este sentido. Por esta razón surge la definición y utilización de $\Theta(\ )$, para tener no solamente la idea de límite superior dada por el $\leq$ de la definición de $O(\ )$, sino también de límite inferior de la cantidad de pasos de ejecución.

Siguiendo con la definición de $g(n) = 2n^2 + 40$ que ya se ha dado, $g(n)$ es $\Theta(n^2)$, además de ser $O(n^2)$. En este caso, los valores de las constantes pueden ser: $c_1 = 1$, $c_2 = 3$, y $n_0 = 7$. La relación entre las funciones resultantes $c_1 f(n) = n^2$, $c_2 f(n) = 3n^2$, y $g(n) = 2n^2 + 40$, se puede ver en la Fig. A.2.

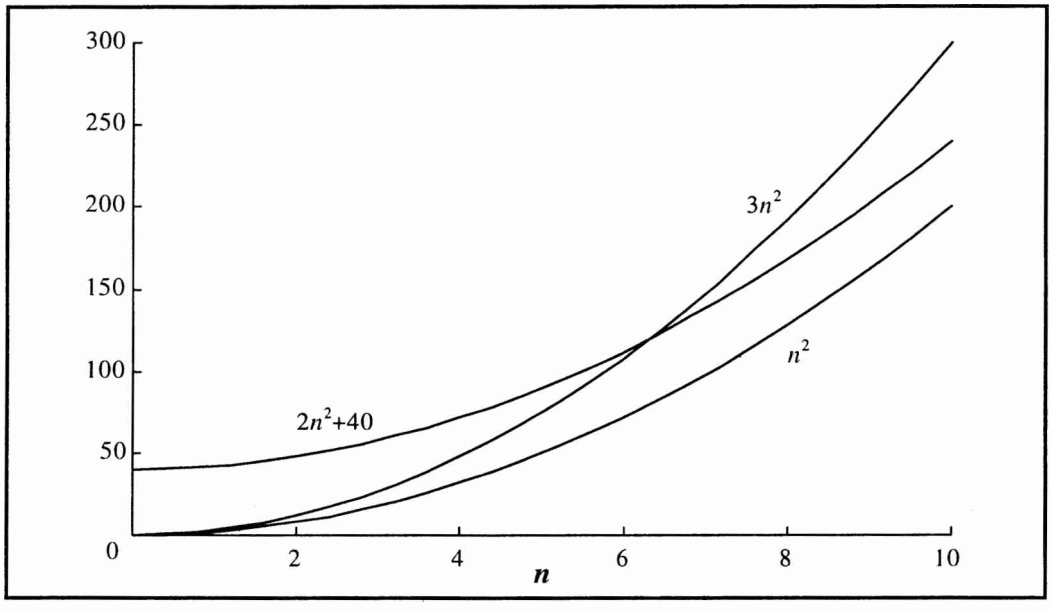


Fig A.2: Ejemplo para $\Theta(\ )$.

A diferencia de lo que sucede con O, la función $g(n) = 2n^2 + 40$ no puede ser $\Theta(n^3)$. Siguiendo la definición de Θ, $f(n) = n^3$, e independientemente del valor de c_1 que se elija, siempre se puede llegar a que $c_1 n^3 > g(n)$ a partir de algún n_1, con lo cual no se cumpliría una de las condiciones de la definición de Θ. Por lo tanto, se llega a que $g(n)$ *no* es $\Theta(n^3)$. Esto se debe a que $c_1 n^3$ *crece más rápidamente* que $g(n)$. En la Fig. A.3 se muestran las funciones $c_1 f(n) = c_1 n^3$ y $g(n)$ con $c_1 = 1$.

La definición y utilización de Θ proporciona una caracterización precisa de la tasa de crecimiento de las funciones que expresan la cantidad de pasos de ejecución de un algoritmo. Esta característica se pudo comprobar en el ejemplo anterior, en el que se llega a que $g(n) = 2n^2 + 40$ no es $\Theta(n^3)$, aunque pueda ser $O(n^3)$.

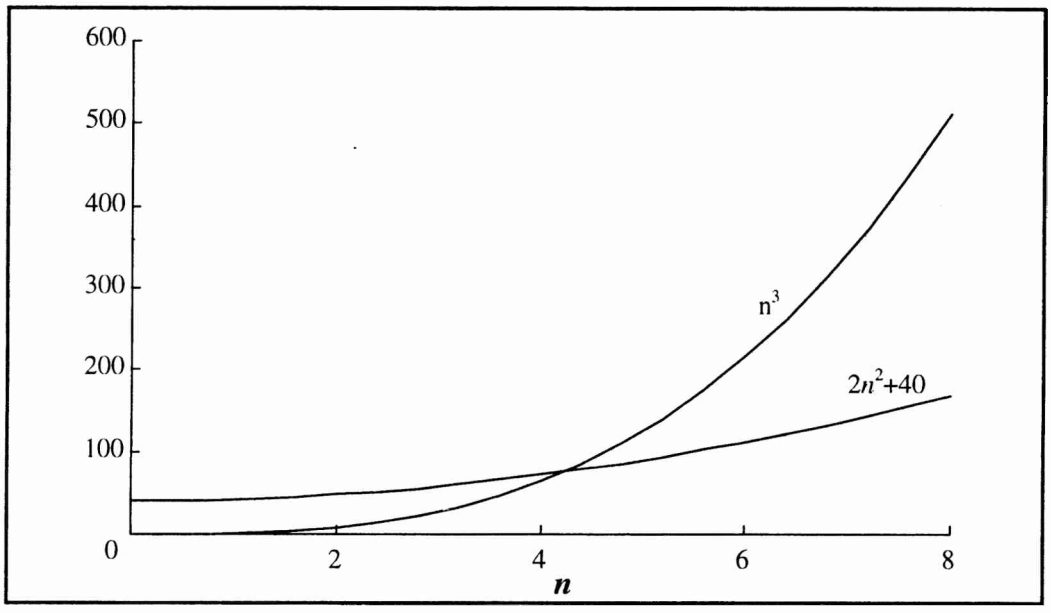


Figura A.3: Ejemplo para *no* Θ( ).

La utilización de Ω( ) y *O*( ) también permite expresar los límites de cantidad de pasos de ejecución inferior y superior de un problema respectivamente. Como se explicó antes, el análisis se concentra en los términos dominantes de las expresiones, sin tener en cuenta las constantes de multiplicación. Normalmente, a medida que se definen algoritmos para un problema dado, se detallan el o los límites que sean útiles en el análisis que se realice. El índice de comparación de algoritmos más utilizado es el límite superior *O*( ), porque permite analizar la situación en el peor caso en cuanto al tiempo de ejecución de los algoritmos. La frase que anteriormente se mencionaba, "del orden de", se refiere a la mayoría de los casos a *O*( ), o más apropiadamente, a: "de orden de a lo sumo". Aunque son diferentes, *O*( ) y Θ( ) se suelen utilizar para expresar lo mismo: el orden de magnitud de la cantidad de pasos de un algoritmo (sea secuencial o paralelo) [Akl89] [Kum94].

A.3 Speed-Up, Eficiencia y Costo

Las definiciones de Speed-Up y Eficiencia que se realizaron en el primer capítulo pueden tomar forma explícita una vez que se conoce cómo calcular el tiempo de ejecución de un algoritmo. De acuerdo a las definiciones de órdenes de magnitud de la cantidad de pasos de ejecución, los índices de rendimiento tales como el Speed-Up y la Eficiencia se pueden expresar haciendo uso de Ω( ), *O*( ) y/o Θ( ).

Por la definición del factor de Speed-Up en particular, se utiliza la propiedad de *O*( ) y de Θ( ) por la cual

$$O(f_1(n)) / O(f_2(n)) = O(f_1(n) / f_2(n)) \quad \text{(A.1)}$$

$$\Theta(f_1(n)) / \Theta(f_2(n)) = \Theta(f_1(n) / f_2(n)) \quad \text{(A.2)}$$

De forma similar a lo que sucede con el factor de Speed-Up, en la definición de eficiencia se encuentra el tiempo de ejecución del algoritmo. En este caso, como la cantidad

de procesadores no siempre se aumenta en función del tamaño del problema, se debe recordar que tal como se definen $O(\ )$ y $\Theta(\ )$, las constantes no se tienen en cuenta, no alteran el resultado. Si, por el contrario, la cantidad de elementos de procesamiento crece junto con el tamaño del problema, entonces se debería utilizar la Ec. (A.1) o la Ec. (A.2) según sea el caso.

El costo de un algoritmo paralelo se define como

$$C = T_N \times N \tag{A.3}$$

donde T_N es el tiempo de ejecución paralelo con N elementos de procesamiento.

Apéndice B: Lenguaje de Programación Ada

El lenguaje de programación Ada posee una gran riqueza expresiva que se origina especialmente para cumplir requerimientos del desarrollo de sistemas de tiempo real (mono o multiprocesador, distribuídos o no). Con Ada se cumplieron las etapas de especificación formal rigurosa del lenguaje antes de desarrollar compiladores y se cumple una validación estricta de los nuevos compiladores, lo cual favorece la portabilidad y la estandarización. En este apéndice se verán de forma concisa algunas caracterísiticas generales del lenguaje y luego un detalle de las instrucciones específicas para programación de aplicaciones paralelas.

Además de los módulos clásicos de los lenguajes imperativos (procedimientos y funciones), Ada provee abstracción de datos a través de los *Packages*. Un package encapsula operaciones y datos, separando la parte visible de especificación de la implementación de cada operación. Los datos pueden ocultarse a través de la especificación *Private* y la especificación puede compilarse por separado, dando facilidad al desarrollo de prototipos. Los packages pueden ser genéricos, constituyendo un molde básico a partir del cual se instancian nuevos packages que heredan parcial o totalmente las operaciones definidas del molde general. La Fig. B.1 muestra un ejemplo de definición de un package con un dato private.

```
GENERIC TYPE elemento IS PRIVATE;
PACKAGE PilaGeneral IS
    TYPE pila IS PRIVATE;
    PROCEDURE Push(f: IN elemento; s: IN OUT pila);
    PROCEDURE Pop(f: OUT elemento; s: IN OUT pila);
END PilaGeneral;
```

Figura B.1: Definición de un Package.

El dato definido se puede instanciar como lo muestra la Fig. B.2.

```
PACKAGE Pila_de_Reales IS NEW PilaGeneral(FLOAT);
PACKAGE Pila_de_Cuentas IS NEW PilaGeneral(cuenta);
```

Figura B.2: Instancias de un Package.

En Ada se puede descomponer un sistema en procesos que se ejecutan de forma concurrente (posiblemente en múltiples procesadores) y se comunican, sincronizando sus acciones. Para esto se utiliza la primitiva TASK con la cual se definen procesos que se pueden ejecutar concurrentemente, y dentro de ellos se permite especificar los puntos de comunicación entre TASKs a través de la primitiva ENTRY. Estos puntos de entrada constituyen puntos de invocación (o de comunicación) entre TASKs. El mecanismo de sincronización de procesos de Ada se conoce como rendezvous extendido.

La Fig. B.3 muestra un ejemplo de definición de un proceso (TASK), que se dedica al control de una lista de vuelo.

```
TASK Lista_de_Vuelo IS
    ENTRY Reservar(Num: OUT asiento);
    ENTRY Liberar(Num: IN asiento);
    ENTRY CancelarVuelo;
END Lista_de_Vuelo;
```

Figura B.3: Ejemplo de Definición de un Proceso.

La Fig. B.4 muestra cómo se invoca el proceso definido en la Fig. B.3, o lo que es lo mismo, cómo se le envía un mensaje.

```
Lista_de_Vuelo.Reservar(x);
Lista_de_Vuelo.Liberar(5);
```

Figura B.4: Ejemplo de Comunicación con un Proceso.

Siguiendo con el ejemplo de la Fig. B.3, si es necesario mantener múltiples listas de vuelo, se puede declarar un TASK TYPE Lista_de_Vuelo que actúa como un *molde* para la definición de procesos. Con el *molde* (TASK TYPE) definido se puede llevar a cabo la definición de instancias, por ejemplo un arreglo de TASKs. Cada elemento del arreglo es una lista de vuelo en particular. Desde el punto de vista de un programa en Ada, un TASK TYPE puede considerarse un tipo de dato como cualquier otro, y puede utilizarse como tal.

El lenguaje Ada también provee facilidades para manejo de excepciones. De un modo muy flexible se pueden especificar los manejadores de excepción de cada procedimiento, función o tarea del sistema. También es posible tener diferentes manejadores para una misma excepción, según la instancia de desarrollo en la que se encuentre la aplicación.

El lenguaje Ada permite manejar directamente la interface de hardware, y en particular, operar con tiempos absolutos y relativos, obtenidos del reloj real de la arquitectura de procesamiento donde se lleva a cabo la ejecución. Esto facilita la programación de aplicaciones de tiempo real y permite "mapear" funciones lógicas con la arquitectura física que las soporta, controlando el tiempo en el caso de ser necesario.

Este apéndice no se dedica a realizar una descripción extensiva del lenguaje Ada [Hab93] [Ols83], sino a introducir los aspectos fundamentales para la programación concurrente. Una vez definido un programa concurrente, se tiene un conjunto de procesos (TASKs) que se pueden ejecutar de forma paralela (simultánea) dependiendo de la arquitectura utilizada y de la asignación de procesos en los procesadores.

B.1 Instrucciones Específicas del Lenguaje Ada para Aplicaciones Paralelas

La construcción básica en Ada para procesos concurrentes es el TASK. Los procesos concurrentes pueden especificarse como TASKs que se sincronizan a través del mecanismo conocido como rendezvous extendido [Hab93] [Ols83]. Por ejemplo, se puede definir un

TASK que proporciona los servicios de un recurso (por ejemplo un buzón) y k procesos clientes que depositan o retiran mensajes (datos) del buzón tal como se esquematiza en la Fig. B.5.

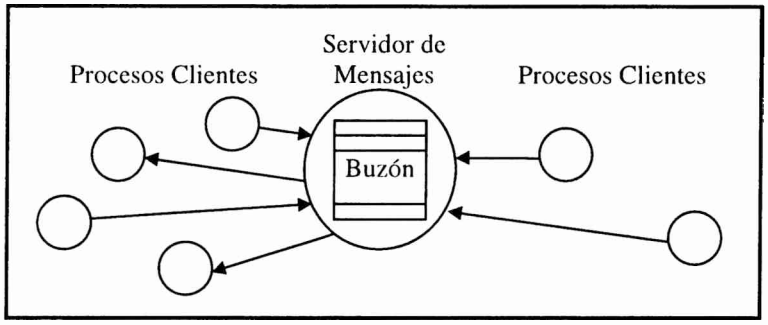


Figura B.5: Proceso Servidor de un Buzón y Procesos Clientes.

La especificación del proceso servidor sería tal como se muestra esquemáticamente en la Fig. B.6.

```
TASK Mailbox IS
   ENTRY Depositar(msg: IN mensaje);
   ENTRY Retirar(msg: OUT mensaje);
END Mailbox;

TASK BODY Mailbox IS
-- Declaraciones
BEGIN
   -- Implementación de los entries Depositar y Retirar
END Mailbox;
```

Figura B.6: Especificación de un Proceso Servidor de Buzón.

Por su parte los procesos cliente tendrán una estructura similar, pero sin ENTRIES. Para *invocar* al proceso servidor indican Mailbox.Depositar(x) o Mailbox.Retirar(x) donde x es de tipo mensaje. Cada vez que se ejecuta un Mailbox.Depositar(x) se envía un mensaje al proceso servidor para que sea almacenado en el buzón. Cada vez que se ejecuta un Mailbox.Retirar(x) se requiere del proceso servidor que retorne un mensaje que tiene almacenado en el buzón. La especificación de cada proceso cliente sería tal como se muestra esquemáticamente en la Fig. B.7.

```
TASK BODY Cliente IS
   -- Declaraciones
BEGIN
   --
   Mailbox.Depositar(x)
   ---
   Mailbox.Retirar (x)
END Cliente;
```

Figura B.7: Proceso Cliente del Proceso Servidor de Buzón.

Las especificaciones de la Fig. B.6 y de la Fig. B.7 corresponden a un TASk de Ada y por lo tanto definen solamente un proceso. Se puede obtener mayor generalidad declarando un *tipo TASK* para el servidor de buzón. Posteriormente, *instancias* del tipo TASK permiten tener un conjunto de procesos servidores de buzones similares, cada uno de los cuales puede tener una función dentro de un sistema de software. Como cualquier otro tipo de un lenguaje de progrmamación, para utilizar el tipo de datos se deben definir *variables* de tipo TASK. Volviendo al ejemplo, la Fig. B.8 define el tipo TASK correpondiente al servidor de buzón.

```
TASK TYPE Mailbox IS
    ENTRY Depositar(msg: IN mensaje);
    ENTRY Retirar(msg: OUT mensaje);
END Mailbox;
```

Figura B.8: Tipo TASK para el Servidor de Buzón.

En la Fig. B.9 se muestra la declaración de tres variables de tipo Mailbox y, por lo tanto, tres instancias del TASK TYPE.

```
A, B, C : Mailbox; -- Declaración de variables.
```

Figura B.9: Declaración de Tres Instancias de Proceso Servidor.

Las tres variables definidas en la Fig. B.9 son tres servidores de buzón distintos y se pueden utilizar estos servidores desde diferentes tareas cliente. Los mensajes se envían y reciben de los servidores del siguiente modo:

```
A.Depositar(x1);
B.Depositar(x2);
C.Retirar(x3);
```

En todos los casos, las operaciones afectan a servidores diferentes y los mensajes no se mezclan, un mensaje depositado en un servidor no se puede retirar desde ningún otro.

El TASK TYPE, así como el TASK define la comunicación de un proceso con los demás. En otras palabras, todo lo que se identifica desde los demás procesos son los entries del TASK (TYPE). El TASK BODY implementa estos entries visibles al usuario, teniendo en cuenta que la implementación en sí misma está oculta como en el caso de la implementación de un PACKAGE. La principal diferencia entre la implementación de un PASK y un PACKAGE es que en un TASK se debe especificar la sincronización necesaria, porque un TASK es un *proceso concurrente.*

Un aspecto importante en el ejemplo de proceso servidor es que cuando se implementan los entries Retirar y Depositar, naturalmente las dos operaciones van a ser excluyentes entre sí dentro de un mismo TASK. Como ya se ha explicado, dentro del TASK BODY se implementan los entries, y se lleva a cabo mediante la utilización de la instrucción ACCEPT. La estructura básica para implementar un ENTRY es la que muestra esquemáticamente la Fig. B.10.

```
ACCEPT E1(parámetros formales) DO
    -- Cuerpo de instrucciones para el Entry E
END E1;
```

Figura B.10: Implementación de un ENTRY.

Si, por ejemplo, se tiene una secuencia de código

```
ACCEPT E1(parámetros formales) DO cuerpo de E1 END E1;
ACCEPT E2(parámetros formales) DO cuerpo de E2 END E2;
ACCEPT E3(parámetros formales) DO cuerpo de E3 END E3;
```

Se define que se debe esperar un pedido por el ENTRY E1, una vez que se produce el pedido, es atendido y *luego* se espera un pedido por el entry E2 y se antiende y *luego* se espera un pedido por el entry E3 y se atiende.

Se puede ejecutar repetidamente una secuencia de atención de pedidos como la anterior, mientras se cumpla una condición lógica:

```
WHILE (condición) LOOP
   ACCEPT E1(parámetros) DO cuerpo de E1 END E1;
   ACCEPT E2(parámetros) DO cuerpo de E2 END E2;
   ACCEPT E3(parámetros) DO cuerpo de E3 END E3;
END LOOP
```

Mientras la condición sea TRUE se repite la secuencia de esperas y atención de los entries E1, E2, E3, E1, E2, E3... Aplicado al ejemplo del proceso servidor, supondría una iteración como la de la Fig. B.11.

```
LOOP
   ACCEPT Depositar (msg: IN mensaje) DO ... END Depositar;
   ACCEPT Retirar (msg: OUT mensaje) DO ... END Retirar;
END LOOP;
```

Figura B.11: Iteración de Esperas y Atenciones de Entries.

Se debe notar que esta manera de escribir el código supone una secuencia exacta: a cada Depositar le seguirá un Retirar. Si hay dos requerimientos consecutivos de Depositar, el segundo quedará en espera, porque *no* puede ser atendido hasta que no haya un requerimiento de Retirar.

En muchas ocasiones no se puede restringir un proceso servidor a la atención de un pedido de cada tipo alternativamente, como en el caso que se muestra en la Fig. B.11. Por un lado se pierde flexibilidad pero, aún más importante, en la mayoría de los casos no se conoce de antemano la secuencia de pedidos que se le enviarán a un proceso, sea servidor o no. Se debe especificar la posibilidad de atender un ENTRY (en general, un requerimiento) u otro/s según lo que otros procesos (posiblemente procesos clientes), requieren. Para esto, el lenguaje

Ada ofrece la instrucción SELECT que permite una elección entre dos o más entries, tal como lo muestra de forma esquemática la Fig. B.12.

```
SELECT
   ACCEPT E1(parámetros) DO cuerpo de E1 END E1;
OR
   ACCEPT E2(parámetros) DO cuerpo de E2 END E2;
OR
   ACCEPT E3(parámetros) DO cuerpo de E3 END E3;
END SELECT;
```

Figura B.12: Atención Alternativa de Entries.

Además, se puede incluir una condición (o "guarda") para realizar el ACCEPT. Esto significa "atender el pedido (ENTRY) si se cumple una condición dada". La condición se especifica inmediatamente antes del ACCEPT mediante la cláusula WHEN. De forma esquemática, el ACCEPT con guarda se define

```
WHEN (condición) =>
   ACCEPT E(parámetros) DO cuerpo de E END E;
```

Lo cual a su vez puede utilizarse dentro de la instrucción SELECT.

La Fig. B.13 muestra la combinación de estas instrucciones en el ejemplo del servidor de buzón con los entries Depositar y Retirar.

```
LOOP
  SELECT
    WHEN (cantidad < capacidad) =>
      ACCEPT Depositar(msg: IN mensaje) DO
        -- Acciones
        -- Incrementar cantidad
      END Depositar;
  OR
    WHEN (cantidad > 0) =>
      ACCEPT Retirar (msg: OUT mensaje) DO
      -- Acciones
      -- Decrementar cantidad
      END Retirar;
  END SELECT;
END LOOP;
```

Figura B.13: Implementación del Servidor de Buzones.

Uno de los atributos importantes de un TASK en Ada es que, por definición del lenguaje, los pedidos pendientes de cada ENTRY se "encolan". Cada ENTRY tiene una cola asociada de procesos que esperan por ser antendidos. Además se puede saber *cuántos*

requerimientos pendientes se tienen de un ENTRY en un determinado momento de la ejecución. La cantidad de requerimientos encolados de un ENTRY E se obtiene por medio de

E'COUNT

Si, por ejemplo, se quiere dar prioridad a los procesos que retiran un mensaje por sobre los que depositan en el servidor de buzón, se puede cambiar en el ejemplo de la Fig. B.13

```
WHEN (cantidad < capacidad) =>
   ACCEPT Depositar(msg: IN mensaje) DO
```

por

```
WHEN (cantidad < capacidad) AND (Retirar'COUNT = 0) =>
   ACCEPT Depositar(msg: IN mensaje) DO
```

En este caso se deshabilita la atención de requerimientos Depositar cuando haya uno o más requerimientos esperando en la cola del ENTRY Retirar. De forma análoga, se puede dar prioridad de atención a los procesos que depositan un mensaje por sobre los que retiran mensajes del buzón.

Bibliografía

[Agr86] Agrawal P, Janakiram V, Pathak G, "Evaluating the Performance of Multicomputer Configurations", IEEE Computer, 19, No. 5, 1986.

[Aho74] Aho A, Hopcroft J, Ullman J, "The Design and Analysis of Computer Algorithms", Addison-Wesley Publishing Company, 1974.

[Aho88] Aho A, Hopcroft J, Ullman J, "Estructuras de Datos y Algoritmos", Addison-Wesley Iberoamericana, S. A., Wilmington, Delaware, E. U. A.

[Akl89] Akl S, "The Design and Analysis of Parallel Algorithms", Prentice-Hall, Inc., 1989.

[Amd67] Amdahl G, "Validity of the Single-Processor Approach to Achieving Large-Scale Computer Capabilities", AFIPS Conference Proceedings 30, 1967.

[And91] Andrews G, "Concurrent Programming: Principles and Practice", The Benjamin/ Cummings Publishing Company, Inc., 1991.

[Art89] Artsy Y, Finkel R, "Designing a Process Migration Facility. The Charlotte Experience", IEEE Computer, Sep.1989.

[Bac90] Bach M, "The Design of the UNIX Operating System", Prentice Hall, Inc., 1990.

[Bal89] Bal H, et alter. "Programming Languages for Distributed Computing Systems", ACM Computing Surveys, vol. 31, N° 3, 1989.

[Bor79] Borgerson B, Godfrey M, Hagerty P, Rykken T, "The Architecture of the Sperry Univac 1100 Series Systems", Proc. 6th Annual Symposium Computer Architecture, ACM, New York, April 1979.

[Cas88] Casavant T, Kuhl J, "A Taxonomy of Scheduling in General Purpose Distributed Operating Systems", IEEE Transactions on Software Engineering, vol. 14, N° 2, Feb. 1988.

[Cha91] Chandy K, Kasselman C, "Parallel Programming in 2001", IEEE Software, Nov. 1991.

[Cor92] Cortés A, "La asignacion Dinámica de Tareas en Computadores Paralelos: Un Análisis Crítico", Trabajo de Tercer Ciclo dentro del programa de Arquitectura de Ordenadores, Universidad Autónoma de Barcelona, Jul. 1992.

[Corr92] Corradi A, Leonardi L, Zambonelli F, "How to Apply Locality to Achieve Load Balancing: Migration in Massively Parallel Architecture", IEEE 1992.

[Cos94a] Cosentino A, De Andrea M, "Algoritmo Paralelo para el Afinado de una Imagen Digital", First International Congress of Information Engineering, Universidad de Buenos Aires, Bs. As., 1994.

[Cos94b] Cosentino A, De Andrea M, "Algoritmo Paralelo para el Reconocimiento de Curvas en Planimetría", First International Congress of Information Engineering, Universidad de Buenos Aires, Bs. As., 1994.

[Cos95] Cosentino A, Acosta Burllaile L, De andrea M, De Giusti A, "Algoritmo Paralelizable para Reconocimiento de Patrones de Figuras Geométricas", I Congreso Argentino de Ciencias de la Computación, Bahía Blanca, Octubre de 1995.

[Dec89] DeCegama A, "Parallel Processing Architectures and VLSI Hardware", Volume I, Prentice-Hall International, Inc., Englewood Cliffs, 1989.

[Del91] Delaplace F, Giavitto J-L, "An Efficient Routing Strategy to Support Process Migration", Microprocessing and Microprogramming 32, 1991.

[Dou91] Douglis F; Ousterhout J, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", Software Practice and Experience, vol. 21, n° 8, Aug. 1991.

[Dun90] Duncan R, "A Survey of Parallel Computer Architectures", Computer, Feb. 1990.

[Eve79] Even S, "Graph Algorithms", Computer Science Press, Potomac, MD, 1979.

[Fel97] Felice R, Ruscitti F, Naiouf M, "Reconocimiento y Clasificación de Objetos en Paralelo", III Congreso Argentino de Ciencias de la Computación (III CACIC), Universidad Nacional de La Plata, Argentina, Septiembre de 1997.

[Fly66] Flynn M, "Very High Speed Computing Systems", Proc. IEEE, Vol. 54, 1966.

[Fly72] Flynn M, "Some Computer Organizations and Their Effectiveness", IEEE Trans. Computers, 21(9), 1972.

[For62] Ford L, Fulkerson D, "Flows in Networks", Princeton University Press, Princeton, NJ, 1962.

[Fra93] Franco D, "Virtualización de las Comunicaciones en Multicomputadores", Trabajo de Tercer Ciclo dentro del programa de Arquitectura de Ordenadores. Universidad Autónoma de Barcelona, Jul. 1993.

[Ger92] Gersho A, Gray R, "Vector Quantization and Signal Compression", Kluwer Academic Publishers, 1992.

[Gon92] Gonzalez R, Woods R, "Digital Image Processing", Addison-Wesley, 1992.

[Goy84] Goyal A, Argewala T, "Performance Analysis of Future Shared Storage Systems", IBM J. Res. Develop., 28, No. 1, 1984.

[Gra89] Graham R, Knuth D, Patashnik O, "Concrete Mathematics: A Foundation Computer Science", Addison-Wesley, Reading, Ma, 1989.

[Gus88a] Gustafson J, "Reevaluating Amdahl's Law", Communications of the ACM, Vol. 32, Num 5, May 1988.

[Gus88b] Gustafson J, "The Scaled-Sized Model: A Revision of Amdahl's Law", Proc. Supercomputing 1988, Vol. II.

[Hab93] Habermann A, Perry D, "Ada for Experienced Programmers", Addison-Wesley Publishing Company, Inc., 1993.

[Har69] Harary F, "Graph Theory", Addison-Wesley, Reading, MA, 1969.

[Hen90] Hennessy J, Patterson D, "Computer Architecture: A Cuantitative Approach", Morgan Kaufmann Publishers, Inc., 1990.

[Hey95] Heymann E, "Soporte para la Migración de Procesos en Computadores Paralelos de Memoria Distribuida", Trabajo de Tercer Ciclo dentro del programa de Arquitectura de Ordenadores, Universidad Autónoma de Barcelona, Jul. 1995.

[Hil84] Hill M, Smith A, "Experimental Evaluation of On-Chip Microprocessor Cache Memories", SIGARCH Newsletter, Vol 12, Issue 3, The 11th Annual International Symposium on Computer Architecture, Conference Proceedings, IEEE, June 1984.

[Hoa86] Hoare C, "Communicating Sequential Processes", Englewood Cliffs, Prentice-Hall, 1986.

[Hoc88] Hockney R, Jesshope C, "Parallel Computers 2", Adam Hilger, Bristol and Philadelphia, IOP Publishing Ltd, 1988.

[Hus91] Hussain Z, "Digital Image Processing: Practical Applications of Parallel Processing Techniques", Ellis Horwood Limited, 1991.

[Hwa93] Hwang K, "Advanced Computer Architecture: Parallelism, Scalability, Programmability", McGraw-Hill, Inc., 1993.

[Hwa84] Hwang K, Briggs F, "Computer Architecture and Parallel Processing", McGraw-Hill, Inc., 1984.

[ISO93] ISO/IEC 10918-1, "Digital Compression and Coding of Continuous-Tone Still Image", 1993. Además ITU-T Rec. T.81.

[Izu92] Izu C, Arruabarrena A, Beibide R, "Analysis and Evaluation of Message Management Functions for Bidimensional Transputer Networks", Microprocessors and Microsystems, vol. 16, n° 6, 1992.

[INM88a] INMOS Limited, "Transputer Reference Manual", Prentice Hall International, 1988.

[INM88b] INMOS Limited, "Transputer Instruction Set. A Compiler Writer's Guide", Prentice Hall International, 1988.

[INM92a] INMOS Limited, "ANSI C Toolset Reference Manual", INMOS Limited, 1992.

[INM92b] INMOS Limited, "ANSI C Language and Libraries Reference Manual", INMOS Limited, 1992.

[INM92c] INMOS Limited, "ANSI C Toolset User Guide", INMOS Limited. 1992.

[Jai89] Jain A, "Fundamentals of Digital Image Processing", Prentice Hall, Englewood Cliffs, NJ, 1989.

[Kar92] Karonis N, "Timing Parallel Programs That Use Message Passing", Journal of Parallel and Distributed Computing, 14, 1992.

[Ker91] Kernighan B, Ritchie D, "El Lenguaje de Programación C", Segunda Edición, Prentice-Hall Iberoamericana S. A., 1991.

[Kim92] Kim Y, "Chips para Multimedia", Binary, Marzo 1992.

[Knu73] Knuth D, "The Art of Computer Programming vol. III: Sorting and Searching", Addison-Wesley, Reading Wassachusetts, 1973.

[Koe94] Koeger Buford J, "Multimedia Systems", ACM Press, Addison-Wesley Publishing Company, 1994.

[Kri88] Krizanc D, Rajasekaran S, Tsantilas T, "Optimal Routing Algorithms for Mesh-Connected Processor Arrays", Reif J, Ed., Proceedings 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures, volume 319 Lecture Notes in Computer Science, Springer-Verlag, July 1988.

[Kum94] Kumar V, Grama A, Gupta A, Karypis G, "Introduction to Parallel Computing. Design and Analyisis of Algorithms", The Benjamin/Cummings Publishing Company, Inc., 1994.

[Lan83] Lang T, Valero M, Fiol M, "Reduction of Connections for Multibus Organizations", IEEE Trans. Comput. C-32, No. 8, 1983.

[Lap93] Laplante P, "Real-Time Systems Design and Analysis: an Engineer's Handbook", IEEE PRESS, 1993.

[Law91] Law A, Kelton D, "Simulation Modeling & Analysis", McGraw-Hill, Inc. Second Edition, 1991.

[Lee94] Leeuven J. van, ed., "Handbook of Theoretical Computer Science. Volume A: Algorithms and Complexity", Elsevier Science Publishers, The MIT Press, 1994.

[Lei92] Leighton F, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes", Morgan Kaufmann Publishers, Inc., San Mateo, California, 1992.

[Lew92] Lewis T, El-Rewini H, "Introduction to Parallel Computing", Prentice-Hall, Inc., Englewood Cliffs, 1992.

[Luq95] Luque E, Franco D, Heymann E, Moure J, "TransCom: a Communication Micro-kernel for Transputers", Reporte Interno, Universidad Autónoma de Barcelona, Departamento de Informática. Jun. 1995.

[Maa91] Maa Y-C, Pradhan D, Thiebaut D, "Two Economical Directory Schemes for Large-Scale Cache Coherent Multiprocessors", ACM-CAN, Vol. 19, No. 5, 1991.

[Mac87] MacDougall M, "Simulating Computer Systems", Mit Press, 1987.

[Mel84] Melhorn K, "Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness", Springer-Verlag, Berlin, 1984.

[Mou93] Moure J, "Run-Time Kernel para Multicomputadores", Trabajo de Tercer Ciclo dentro del programa de Arquitectura de Ordenadores, Universidad Autonóma de Barcelona, Jul. 1993.

[Mud84] Mudge T, Hayes J, Buzzard G, Winsor D, "Analysis of Multiple Bus Interconection Networks", Proc. 1984 Int. Conf. on Parallel Processing, IEEE, 1984.

[Ni91] Ni L, "A Layered Classification of Parallel Computers", Proc. 1991 Int. Conf. for Young Computer Scientists, Beijing, China, May 1991.

[Ols83] Olsen E, Whitehill S, "Ada for Programmers", Reston Publishing Company, Inc., Prentice-Hall, 1983.

[Pan90] Pancake C, Bergman D, "Do Parallel Languages Respond to the Needs of Scientific Programmers?", IEEE Computer, vol. 23, N° 12, Dec. 1990.

[Pra78] Pratt W, "Digital Image Processing", John Wiley & Sons, Inc., 1978.

[Qui87] Quinn M, "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill, 1987.

[Sei84] Seitz C, "Concurrent VLSI Architectures", IEEE Transactions on Computers, Vol C-33, No. 2, Dec. 1984.

[Sil94] Silberschatz A, Peterson J, Galvin P, "Sistemas Operativos. Conceptos Fundamentales", Tercera Edición, Addison-Wesley Iberoamericana S. A., 1994.

[Sim97] Sima D, Fountain T, Kacsuk P, "Advanced Computer Architectures. A Design Space Approach", Addison Wesley Longman Limited, 1997.

[Smi82] Smith A, "Cache Memories", Computing Surveys, Vol. 14, No. 3, Sep. 1982.

[Smi88] Smith J, "A survey of Process Migration Mechanisms", Operating Systems Review, A.C.M. SIGOPS, Jul. 1988.

[Sta97] Stallings W, "Sistemas Operativos", 2 Ed., Prentice Hall International (UK) Ltd, 1997.

[Ste95] Steinmetz R, Nahrstedt K, "Multimedia: Computing Communications and Applications", Prentice Hall, 1995.

[Ste90] Stenström P, "A Survey of Cache Coherence Schemes for Multiprocessors", IEEE Computer, June 1990.

[Sto93] Stone H, "High-Performance Computer Architecture", Third Edition, Addison-Wesley Publishing Company, 1993.

[Tan88] Tanenbaum A, "Sistemas Operativos. Diseño e Implementación", Prentice-Hall, 1988.

[Tan90] Tanenbaum A, "Structured Computer Organization", 3rd Edition, Prentice-Hall, 1990.

[Tan92] Tanenbaum A, "Modern Operating Systems", Prenctice-Hall, Inc. 1992.

[Tin94] Tinetti F, Bria O, "DSPs for DSP", II Jornadas de Investigación del Grupo Montevideo, realizado en Concordia, Argentina y Salto, Rep. Oriental del Uruguay, Septiembre 1994.

[Tin97] Tinetti F, "Migración de Procesos e Integridad de Mensajes. Un Entorno para Diseño y Experimentación", Tesis de Master en Informática, Universidad Autónoma de Barcelona, Jul 1997.

[Tyr89] Tyrrell A, Nicoud J, "Scheduling and Parallel Operations on the Transputer", Microprocessing and Microprogramming 26, 1989.

[Wei91] Weiss M, "Data Structures and Data Analysis", Benjamin/Cummings Publishing Company, Inc., 1991.

[Val81] Valiant L, Brebner G, "Universal Schemes for Parallel Communication", Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, May 1981.

[Wal91] Wallace G, "The JPEG Still Picture Compression Standard", Communications of the ACM, Vol. 34, Num 4, April 1991.

[Wil91] Wilkinson B, "Computer Architecture: Design and Performance", Prentice Hall International (UK) Ltd, 1991.

[Wit87] Witten I, Neal R, Cleary J, "Arithmetic Coding for Data Compression", Communications of the ACM, 30, 1987.

[Yen82] Yen D, Patel J, Davidson E, "Memory Interference in Synchronous Multiprocessors Systems", IEEE Trans. Comput. C-31, No. 11, 1982.

[Zhu90] Zhu W, Goscinski A, Gerrity G, "Process Migration in RHODOS", Australian Defence Force Academy, Canberra. Technical Report CS90/9, 1990.

La Editorial Exacta constituye el órgano editorial de la Fundación Ciencias Exactas, entidad creada para apoyar el funcionamiento de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata. Su objetivo primordial es el de propiciar la difusión de obras originales de la autoría de los docentes e investigadores de la Facultad de Ciencias Exactas.

Procesamiento Paralelo

Conceptos de Arquitecturas y Algoritmos

Históricamente, la única forma de tratar algunos problemas de procesamiento ha sido por medio de cómputo paralelo. A mayor complejidad de cálculo y mayor compromiso con la ejecución en tiempo real (inteligencia artificial, redes neuronales, robótica, reconocimiento de patrones, visualización científica, modelos de elementos finitos y de fluidos, manejo de grandes bases de datos, etc.), se hace imprescindible utilizar procesamiento paralelo para obtener tiempos de respuesta aceptables.

En los primeros capítulos de este libro se analizan aspectos de la arquitectura (organización del control, memoria y comunicaciones) de los sistemas de procesamiento paralelo.

En los capítulos intermedios se tratan algorítmicamente clases de problemas de cómputo paralelo como los de cálculo numérico, ordenación, grafos e imágenes. Se discuten los temas de concurrencia y sincronización entre procesos que se ejecutan en paralelo. Asimismo se analizan parámetros de rendimiento tales como el factor de Speed-Up y el grado de paralelismo alcanzable en los ejemplos que se exponen.

En los últimos capítulos se desarrolla extensivamente el problema de migración de procesos en arquitecturas paralelas distribuidas. Se presenta un entorno de experimentación (diseño e implementación) de migración de procesos, así como resultados de experimentación con distintas políticas de manejo de mensajes.

La intención de este libro no es abarcar *todos* los temas posibles de lo que tradicionalmente se ha denominado procesamiento paralelo. Lo que sí se intenta conservar son las ideas de *aplicación y rendimiento* del procesamiento paralelo, estudiando casos y problemas con el enfoque de optimizar el diseño de la arquitectura y del software.

La idea subyacente de *aplicación* del cómputo paralelo guía en cierto sentido la presentación de los temas. Se hacen explícitas las simplificaciones y suposiciones en los análisis teóricos presentados y se discuten los ejemplos tratando de extraer de ellos las ideas comunes con respecto a la *clase de problemas* a la que pertenecen.