Introducción a la Computación Paralela

2 AUTHORS, INCLUDING:

Jose Aguilar
University of the Andes (Venezuela)
213 PUBLICATIONS **560** CITATIONS

SEE PROFILE

Introducción a la

Computación Paralela

J. Aguilar, E. Leiss

UNIVERSIDAD
DE LOS ANDES
MERIDA-VENEZUELA

Introducción a la Computación Paralela

Autores:

Jose Lisandro Aguilar Castro
CEMISID, Departamento de Computación
Universidad de Los Andes
Mérida, Venezuela

Ernst Leiss
Department of **Computer Science**
University of Houston,
Texas, USA

1ra Edición, 2004

Depósito Legal: LF395920040041698
ISBN: 980-12-0752-3

Esta publicación ha sido realizada gracias al apoyo financiero del **Consejo de Desarrollo Científico, Humanístico y Tecnológico de la Universidad de Los Andes**

Diagramación Electrónica: Miguel Rodríguez
Diseño de Portada: Rossana Ríos
Impreso en: Graficas Quinteto
Mérida, Venezuela

Este es uno de los primeros libros en español que hace una descripción amplia del área Computación Paralela. El libro comienza presentando los modelos de paralelismo, así como uno de los problemas fundamentales del área, la comunicación. Después caracteriza las posibles fuentes de paralelismo, así como algunas metodologías para desarrollar aplicaciones paralelas. El libro presenta aspectos fundamentales de las plataformas paralelas, tales como la asignación y planificación de tareas, la dependencia de datos y posibles transformaciones de código, las medidas de rendimiento usadas, entre otros, todos ellos a ser considerados por sus Sistemas Operativos y/o Compiladores. Es un libro que puede ser usado por estudiantes culminando carreras en el ámbito de las Ciencias Computacionales o que están iniciando estudios de Maestría en esa área.

Los autores, Septiembre 2004

Indice

Capítulo 3. Ciertos Problemas en Computación Paralela/Distribuida

Capítulo 4. Análisis de Dependencia y Técnicas de Transformación

Indice de Figuras

Capítulo 1.
Elementos Introductorios

1.1 Motivación

El procesamiento paralelo es un tipo de procesamiento de la información, que permite que se ejecuten varios procesos concurrentemente [5, 10, 17, 30, 35]. El procesamiento paralelo puede ser de diferentes tipos: i) Un tipo es ejecutar procesos independientes simultáneamente, los cuales son controlados por el sistema operativo (usando tiempo compartido, multiprogramación y multiprocesamiento). ii) Otro tipo es descomponer los programas en tareas (controladas por el sistema operativo, los compiladores, los lenguajes de programación, etc.), algunas de las cuales pueden ser ejecutadas en paralelo. iii) Finalmente, el último tipo se basa en usar técnicas de encauzamiento para introducir paralelismo a nivel de instrucciones, lo que implica dividirlas en pasos sucesivos que pueden ser ejecutados en paralelo, cada uno procesando datos diferentes.

Es difícil determinar qué resulta más natural, sí el procesamiento paralelo o el secuencial. Los enfoques paralelos resultan una necesidad, dada la constante afirmación de que la velocidad máxima en procesamiento secuencial se alcanzará prontamente, excepto que aparezcan avances en otras áreas que definan nuevos mecanismos de procesamiento, los cuales pueden venir de nuevos descubrimientos en áreas tales como computación AND, computación cuántica, etc. En los últimos años, el uso extensivo del paralelismo ha estado ligada a varios hechos [2, 10, 13, 20]:

- *La necesidad de mayor potencia de cálculo*: independientemente de que la potencia de los procesadores aumente, siempre habrá un limite que dependerá de la tecnología del momento. Para aumentar la potencia de cálculo, además de los progresos tecnológicos que permitan aumentar la velocidad de cálculo, se requieren nuevos paradigmas basados en cálculo paralelo. Es decir, una manera de aumentar la velocidad de cálculo es usar múltiples procesadores juntos. Así, un problema es dividido en partes, cada una de las cuales es ejecutada en paralelo en diferentes procesadores. La programación para esta forma de cálculo es conocida como programación paralela, y las plataformas computacionales donde pueden ser ejecutadas estas aplicaciones son los sistemas paralelos y distribuidos. La idea que se usa es que al tener *n* computadores se debería tener *n* veces más poder de cálculo, lo que conllevaría a que el problema pudiera resolverse en 1/n veces del tiempo requerido por el secuencial. Por supuesto, esto bajo condiciones ideales que raramente se consiguen en la práctica. Sin embargo, mejoras sustanciales pueden ser alcanzadas dependiendo del problema y la cantidad de paralelismo presente en el mismo.

- *Una mejor relación costo/rendimiento*: en el orden económico, es muy importante tener máquinas con excelente relación costo/rendimiento. Normalmente, el mayor poder de cálculo genera una explosión de costos que a veces lo hacen prohibitivo. Una manera para lograr mayor poder de cálculo sin costos excesivos es hacer cooperar muchos elementos de cálculo de bajo poder, y por consiguiente, de bajos costos.
- *Potencia expresiva de los modelos de procesamiento paralelo:* muchos problemas son más fáciles de modelar usando paradigmas paralelos, ya sea por la estructura que se usa para su resolución o porque el problema es intrínsecamente paralelo. Es decir, si desde el principio se puede pensar en los mecanismos paralelos/concurrentes para resolver un problema, eso puede facilitar la implantación del modelo computacional. Esto podría permitir obtener mejores soluciones para los problemas a resolver, en tiempos razonables de ejecución. Así, este enfoque permite el surguimiento de modelos de cálculos diferentes, a los modelos secuenciales. En pocas palabras, además del procesamiento de paralelismo para mejorar los tiempos de ejecución se agrega la ganancia conceptual, al poder resolver los problemas con nuevos métodos de resolución hasta ahora imposibles de ejecutarse en máquinas secuenciales.

Así, la computación paralela está jugando un papel fundamental en el avance de todas las ciencias, abriendo las puertas para explotar, más allá de las fronteras ya conocidas, impresionantes poderes de cálculo que permiten modelos más realistas (pasar de la segunda a la tercer dimensión, etc.). A pesar de que del área de computación paralela se habla mucho, en realidad es poco lo que se conoce. Quizás el mayor problema en computación paralela y distribuida es que no son ideas fáciles para entender e implementar. Existen diferentes enfoques para aplicar paralelismo. Incluso, para ciertas aplicaciones algunos enfoques pueden ser contraproducentes. Además, existen varios tipos de arquitecturas, modelos de programación, lenguajes y compiladores, compitiendo por conquistar el mercado. Por otro lado, actualmente los precios baratos de los PC, con su incremento en poder de cálculo, los hacen un formidable competidor contra los emergentes computadores paralelos. También las estaciones de trabajo, las cuales ofrecen amigables y poderosas interfaces, son otras fuertes competidoras.

Hay muchas áreas de aplicación donde el poder de cálculo de una computadora simple es insuficiente para obtener los resultados deseados. Las computadoras paralelas y distribuidas pueden producir resultados más rápidos. Por ejemplo, en algunas áreas de cálculo científico, el tiempo estimado de computación para obtener resultados interesantes usando un computador simple, podría ser tan largo que excedería el tiempo esperado en que el mismo puede fallar. O peor aun, en el área industrial una simulación que tome algunas semanas para alcanzar resultados, es usualmente inaceptable en ambientes de diseño, en los cuales los tiempos con los que cuenta el diseñador son cortos. Además, hay ciertos problemas que tienen fechas límites especificas para calcularse (por ejemplo, si el programa de predicción del tiempo para el día de mañana dura dos días perdería el sentido ejecutarlo). Un punto inicial de arranque sería conocer qué áreas de aplicación podrían beneficiarse al usar paralelismo. Indudablemente, el primer grupo serían aquellas áreas donde el paralelismo es aparente, aunque no se pueda explotar en

máquinas secuenciales. A continuación se mencionan algunas áreas [2, 8, 9, 10, 13, 15, 16, 20, 33, 35]:

Procesamiento de imágenes: Con este término se pueden abarcar las transformaciones de imágenes, análisis de imágenes, reconocimiento de patrones, visión por computadora, etc. Existen dos aspectos básicos que hacen apropiado esta área para procesamiento paralelo: el primero tiene que ver con la gran cantidad de datos que están envueltos en el procesamiento. El segundo tiene que ver con la velocidad requerida para procesar esas imágenes. A su vez, este segundo aspecto puede ser caracterizado por dos razones. Primero, muchas aplicaciones de procesamiento de imágenes ocurren en ambientes donde la tasa de procesamiento requerida tiene fuertes restricciones a nivel de tiempo. En otras palabras, la velocidad de respuesta del sistema es crucial para el éxito de la tarea que se esta llevando a cabo. La segunda gran razón es que el procesamiento de imágenes usa estructuras de datos (y operaciones) intrínsecamente paralelas.

Modelado matemático: La computación juega un papel fundamental en las ciencias y las ingenierías, permitiendo a los científicos e ingenieros construir y probar modelos con nuevas teorías para describir fenómenos naturales o para resolver problemas de la vida real. Estos modelos matemáticos altamente complejos son gobernados, por ejemplo, por ecuaciones diferenciales parciales o elementos finitos cuyas soluciones pueden requerir grandes poderes de cálculo. El modelado matemático consiste en describir entidades (estructuras complejas, procesos químicos o biológicos, etc.) en términos de ecuaciones que comprenden variables y constantes. Por ejemplo, para el diseño de la suspensión de un carro su estructura puede ser aproximada por series de elementos planos (por ejemplo, rectángulos, triángulos), cuyas propiedades mecánicas pueden ser descritas por ecuaciones que tienen constantes (tal como la tensión del acero) y variables (tales como la presión aplicada en ciertos puntos). Naturalmente, las variables pueden depender en un punto dado de sus vecinas (los cuales a su vez son dependientes de sus vecinas), tal que las ecuaciones que definen toda la estructura estén enlazadas. La idea es aplicar sobre el modelo fuerzas externas, en forma tal que se pueda medir, por ejemplo, su punto de ruptura. Es claro que para la estructura del carro, la exactitud de los resultados depende de la fineza de los elementos modelados. Esto implica, que para modelos complejos probablemente muchos miles de elementos se requieren usar. Otras aplicaciones son en áreas tales como estudio de las interacciones de partículas en materiales cristalinos o amorfos, cálculo de la estabilidad de partículas, física de plasmas, mecánica cuántica, reacciones químicas, dinámica molecular, etc.

Computación inteligente: Existen muchos aspectos intrínsecamente paralelos en esta área. Quizá el más significativo es que la computación inteligente trata de imitar el comportamiento inteligente de los humanos, lo que hace pensar que para emular la estructura intrínsecamente paralela del cerebro se requiere este tipo de procesamiento. En el área de las redes neuronales artificiales, el procesamiento consiste en la interacción de un grupo de neuronas cuyos elementos van describiendo la dinámica del sistema. Aquí parece que las neuronas pueden aprovechar un procesamiento paralelo para hacer más eficiente su tiempo de ejecución. A nivel de la computación evolutiva existen muchos aspectos intrínsecamente paralelos, por ejemplo, el procesamiento

paralelo de cada individuo que conforma la población bajo manipulación. Así, el procesamiento de los individuos (a nivel de su reproducción o evaluación) es intrínsecamente paralelo.

Manipulación de base de datos: La idea de base es reducir el tiempo de ejecución al asignar segmentos de las base de datos a elementos de procesamiento paralelo. Esto pasa por definir las estructuras de datos adecuadas y las operaciones transaccionales a realizar.

Predicción del tiempo: Ésta es un buen ejemplo de una aplicación que requiere mucho poder de cálculo. Esta aplicación podría haber sido incluida en el grupo de las aplicaciones con modelos matemáticos complejos, pero ésta tiene un segundo elemento clave, la rapidez con que se deben realizar sus cálculos. La atmósfera es modelada dividiéndola en regiones o celdas de tres dimensiones, usando ecuaciones matemáticas complejas para capturar varios efectos. Básicamente, las condiciones en cada celda (temperatura, presión, humedad, velocidad y dirección del viento, etc.) son calculadas en intervalos de tiempo, usando las condiciones existentes en intervalos de tiempo previos. Los cálculos en cada celda son repetidos para modelar el paso del tiempo. La característica clave que hace eficiente esta aplicación es el número de celdas que use. Supongamos que la atmósfera terrestre la quisiéramos dividir en celdas de 1.6 Km*1.6Km*1.6Km, así cada 16 Km. requieren 10 celdas. Para cubrir toda la atmósfera terrestre se requeriría aproximadamente de $5x10^8$ celdas. Supongamos que cada cálculo en un cubo requiere 200 operaciones de punto flotante. Solamente para un intervalo de tiempo se requeriría 10^{11} operaciones de punto flotante. Si quisiéramos predecir el tiempo de los próximos 10 días con intervalos de tiempo de 10 minutos, se necesitarían 10^3 pasos y 10^{14} operaciones de punto flotante. Un computador a 100 Mflop (10^8 operaciones de punto flotante/segundo) tomaría 10^6 segundos para ejecutar el cálculo. Para ejecutarlo en tiempos cortos se requeriría de más de 1 teraflops ($1x10^{12}$ operaciones de punto flotante por segundo).

Otras áreas que podrían aprovecharse de un procesamiento paralelo son las siguientes: biología, ciencias aerospaciales, sismología, diseño de VLSI, robótica, previsión (modelado socioeconómico, etc.), exploración (oceanografía, astrofísica, geología, etc.), máquinas inteligentes (planificación, sistemas expertos, mecanismo de aprendizaje, etc.), medicina (tomografía, síntesis de proteínas, etc.), automatización industrial, aplicaciones militares y multimedia, investigación y diseño farmacéutico, en áreas donde se han establecido bien los modelos de computación en dos dimensiones tal que en un futuro próximo tiendan a tres dimensiones para mejorarlos y puedan parecerse más al mundo real, etc. Las características genéricas de estas aplicaciones son las siguientes: requieren enormes cálculos repetidos sobre grandes cantidades de datos para obtener resultados válidos, y el cálculo debe terminarse en un período de ejecución razonable. Esto implica que para atacar estos problemas se requiere de hardwares rápidos y softwares eficientes.

Durante varios años, científicos e ingenieros han estado confrontándose con máquinas paralelas. Los usuarios han visto como un reto usar estos sistemas, algunas veces teniendo muchos éxitos al usarlas, pero frecuentemente frustrados por las dificultades

para obtener buenos resultados. Parte de los elementos que pueden extender el uso de estas máquinas son [22, 24, 35, 36, 37]:

- *Programación familiar*: programas para estas plataformas deben ser escritos en lenguajes familiares, o variaciones fáciles de entender.
- *Estabilidad de rendimiento*: los rendimientos deben ser estables, tal que la tasa de cálculo no varíe mucho entre un computador y otro.
- *Portabilidad*: el código debe ser fácil de mover entre máquinas.

Mucho esfuerzo se requiere en el desarrollo de software para estas plataformas, más que en la proposición de elementos de hardware eficientes. Así, a nivel de software del sistema es donde se concentran los grandes retos en la actualidad, tanto a nivel de desarrollo de sistemas operativos, compiladores, como de librerías de programación. Algo importante para orientar futuros diseños es que los usuarios aceptan más fácilmente cambios en su software, que cambiar a nuevos productos, aun cuando sean revolucionarios. En pocas palabras, prefieren una evolución en sus herramientas de software a nuevos productos diferentes a los ya conocidos, por mejores que sean. Un clásico ejemplo es a nivel de Fortran, a través de los años, los científicos han preferido adaptar sus códigos a las extensiones que se les han ido incorporando a éste, que pasar a nuevos lenguajes de programación.

Algunos de los actuales problemas en esta área son [2, 13, 22, 24, 34, 35, 36, 37, 40]:

- La *latencia:* la latencia de una máquina paralela es más grande que la de una máquina secuencial. Esto es debido a la propia arquitectura (memoria distribuida, manejo de copias locales que implican problemas de coherencia, etc.). La sola forma de resolverlo es optimizando el uso de la plataforma (por ejemplo, usando esquemas de pre-paginacion en la memoria, protocolos de coherencia global, etc.). Quizás los hilos de ejecución pueden ser otra fuente de ayuda para reducir el problema de latencia.
- Los *anchos de banda*: muchas redes de interconexión se han diseñados (mallas, etc.), algunas con terribles rendimientos y problemas de conflictos.
- La determinación de los *mecanismos de control de* ejecución paralela: SIMD (Single Instruction-Multiple Data) y MIMD (Multiple Instruction-Multiple Data) han estado compitiendo estos últimos años, pero como se ha visto SIMD puede sobrevivir para casos específicos y MIMD es más general.
- La *capacidad de acceso a los datos*: los mecanismos de pase de mensajes o memoria compartida tienen una repercusión directa en el rendimiento del sistema. En particular, el problema de direccionamiento de memoria en cada estilo envuelve diferentes mecanismos en hardware y software, así como específicos estilos de programación y lenguajes.
- Los *modelos de programación*: muchas proposiciones se han hecho sobre cómo pensar en programas paralelos. Los modelos van desde proposiciones de diseño con un bajo nivel de sincronización entre instrucciones, hasta modelos que permiten que el código secuencial se ejecute automáticamente en paralelo. Algunos lenguajes de programación se han ido modificando (por ejemplo, HPF) para aprovechar el rendimiento exhibido por las máquinas paralelas. Pero así como muchos idiomas y

dialectos han sobrevivido, modelos de programación, lenguajes y estilos no convergerán en uno, ni hay razón para que así sea, ya que cada uno responde a una necesidad diferente.

- Los *Compiladores y Sistemas Operativos*: una eficiente explotación de una plataforma paralela pasa por mejores compiladores y sistemas operativos. A pesar de que sustanciales mejoras se han hecho, aun se requieren más; en particular, para manejar grandes cantidades de datos de manera paralela en la jerarquía de memoria, en tareas de planificación (por ejemplo, para lazos de ejecución), entre otras.

Es razonable pensar que la naturaleza humana vislumbre nuevas aplicaciones que excedan las capacidades de los sistemas computacionales. Por ejemplo, las actuales aplicaciones de realidad virtual requieren considerables velocidades de cálculo para obtener resultados interesantes en tiempo real. Esto nos refleja la necesidad permanente de realizar estudios sobre nuevos esquemas de optimización en el ámbito computacional.

1.2 Historia

Grandes progresos en los sistemas computacionales se han hecho en estos últimos 55 años. La velocidad de los computadores secuenciales ha crecido en órdenes de magnitud inimaginables debido a avances a nivel de sus diseños y en las técnicas usadas para construirlos. También, mucho software se ha desarrollado y estandarizado (por ejemplo, el sistema operativo UNIX). Pero a nivel de supercomputadores, desde los *CRAY 1* de 1970 con sus 12.5 ns de periodos de reloj, apenas se pudo mejorar a 4 ns en 15 años. Sólo ha sido posible mantener mejores rendimientos en las supercomputadoras, al introducir paralelismo en la arquitectura del sistema. De esta manera, en vez de estar atados a los constantes progresos a nivel de la tecnología de hardware como en el pasado, los computadores de alto rendimiento están ahora más ligados a innovaciones arquitectónicas y a nivel de software [3, 6, 13, 17, 25, 34].

En general, los avances en tecnología computacional se pueden dividir en los realizados en la era secuencial, y los que están sucediendo ahora en la era de paralelismo (ver figura 1.1). Además, cada una de ellas se puede dividir en 3 fases, una arquitectónica que tiene que ver con los avances en el hardware del sistema, y dos fases de software, una que tiene que ver con los compiladores (pasando de los lenguajes de alto nivel a los compiladores que optimizan los programas para reducir los tiempos de ejecución) y otra con las librerías/paquetes de aplicaciones (que liberan a los usuarios de escribir ciertas partes de código) [3, 6, 13, 17, 25, 34]. Además, cada fase se divide en tres partes: investigación y desarrollo (fase de mucho trabajo teórico y científico, aun sin un uso extendido), comercialización (fase en la que las compañías del área construyen y comercializan los productos) y expansión/consolidación (fase en la cual deja de ser un tópico caliente de investigación, los precios empiezan a ser accesibles para los usuarios, y los productos son fáciles de hacer a nivel industrial).

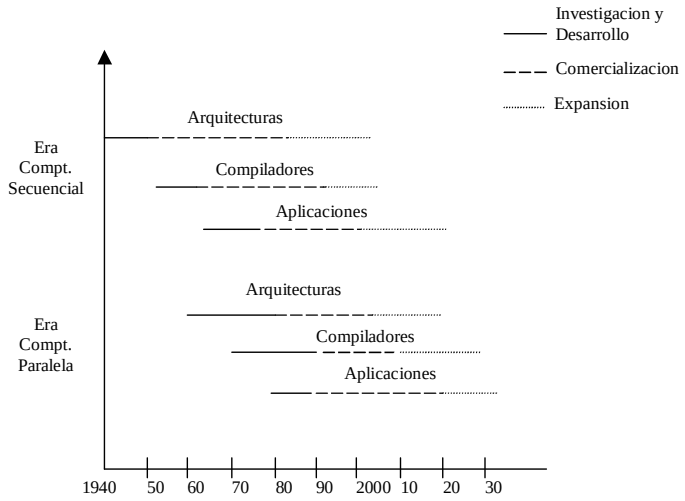


Figura 1.1. Eras de la Computación

En los años 50 ocurrieron los primeros intentos de construcción de máquinas paralelas, pero es hasta los 60-70 que se construyen las primeras. La mayoría de esas máquinas eran máquinas vectoriales mono-procesadoras. A partir de los 70 surgieron las máquinas vectoriales multiprocesadoras. Todas disponían de una memoria compartida, pero eran muy caras. Además, el número de procesadores no llegaba a ser más de 16. A continuación un breve resumen [5, 10, 16, 17, 18, 20, 26, 27, 30, 35]:

- En 1950 Von-Neumann considera analogías entre computador-cerebro y propone un modelo de cálculo paralelo.
- Muchos computadores en los 50 usan procesamiento paralelo en operaciones aritméticas. Por ejemplo, en 1956 Stephe Viger en la Universidad de Columbia propone un arreglo de procesadores compartiendo la memoria, y en 1958 J. Holland en la Univ. de Michigan escribió sobre cómo programar un arbitrario número de subprogramas simultáneamente. Además, a Gil se debe la primera publicación científica en el área al describir sobre cómo debería ser la programación paralela.
- En 1960 Westinghouse Electric construye *Solman*, una máquina paralela bit-send.
- De los 60 a los 70 hay varios proyectos. Se desarrolla Illiac IV en la Universidad de Illinois el cual consiste de 64 procesadores. Bell Labs desarrolla *PEPE* para procesamiento de senales de radares. Esta arquitectura consistía de un computador central que controlaba el arreglo de procesamiento paralelo, ya que la aplicación requería interacción entre diferentes hilos de procesamiento. En los 70 ICL diseño y construyo *Dap*, el cual era una memoria asociativa.
- A mediados de los 70 aparece *CRAY 1*, cuya arquitectura era la de un vector de instrucciones que operaba sobre un vector de 64 palabras. Además, podía realizar la más rápida operación escalar disponible para la época. En los 80 aparecen *CRAY X-MP* y *Y-MP*, que mejoraban a *CRAY 1* al dar adicional bandas de datos de memoria, manejar vectores espaciados eficientemente y mejorar el software del sistema. Además, a diferencia de *CRAY 1* compuesto de un simple procesador, ahora se podía contar con 2, 4, 8 o 16 procesadores.

- En los 80 aparecieron las estaciones de trabajo que eran solamente dos veces más lentas que los CRAY y más baratos. Además, las colas para ejecutar programas en CRAY eran inmensas (perdiéndose mucho tiempo por eso), con tiempos de ejecución costosísimos.
- A finales de los 80 aparecen los sistemas masivamente paralelos de INTEL, nCUBE, Thinking Machines, etc. como una nueva tendencia diferente a los supercomputadores vectoriales o paralelos tradicionales (CRAY, etc.).
- Actualmente, hay una gran extensión de nuevos dominios de aplicaciones, por ejemplo, los nuevos desarrollos en Bases de Datos (Oracle en NCube, etc.). Además, en 1994 IBM asumió un unificado enfoque al unir y conectar sus mainframes y sistemas desktop. Así, hoy en día se presentan nuevas arquitecturas basadas en procesadores poderosos con diferentes tecnologías de interconexión, que dan una diversidad de facilidades a los diferentes tipos de aplicaciones.

Un hecho resaltante es que en cada década la computación paralela ha ofrecido el más alto nivel de rendimiento, al compararla con los otros sistemas disponibles para ese momento (ver tabla 1.1) [3, 5, 8, 10, 16, 20, 27, 30, 35]. El hardware usado para desarrollar computadores paralelos ha cambiado durante cada década, en los 60 se usaba la tecnología más avanzada, lo que resultaba costoso y difícil de construir. En los 70 aparecieron más facilidades para construir computadores paralelos. Ya en los 80, con el uso de los microprocesadores y buses de comunicaciones, el hardware para procesamiento paralelo se abarató, y muchas compañías y universidades han podido construir el suyo. Este hecho se ha extendido durante la década de los 90. Pero un detalle curioso es que su uso masivo ha estado frenado por falta de software eficiente para estas plataformas.

		Relativo a la Epoca		
Décadas	**Máquinas Paralelas**	**Velocidad Pico**	**Tecnología de hardware**	**Calidad del Software**
1960	PEPE, etc.	Alta	Alta	Baja
1970	DAP, etc.	Alta	Media	Baja
1980	Hipercubos, etc.	Alta	Media/Baja	Baja
1990	SP2, etc.	Alta	Baja	Baja

Tabla 1.1 Rendimientos de las Máquinas Paralelas con respecto a la época.

1.3 Clasificaciones

Los computadores paralelos se han desarrollados estos últimos años gracias a los avances en campos tan diversos como el arquitectónico, las tecnologías de interconexión, los ambientes de programación (lenguajes, sistemas, etc.), como de otros más. Muchos esquemas de clasificación se han propuesto, pero el más popular es la taxonomía de Flynn, la cual se basa en la manera como se organiza el flujo de datos y de instrucciones.

Sin embargo, existen otros criterios que se podrían usar para establecer una clasificación a nivel de la programación paralela: el modo operacional de los procesadores, la organización de la memoria, la granularidad de los procesadores, las características de la arquitectura. Así, otras clasificaciones han sido propuestas basadas en otros criterios, algunas de ellas son las de Feng, Kuck, Gajskim Treleaven, etc. Presentaremos aquí a la de Flynn, y mencionaremos otras para tratar de mostrar las arquitecturas paralelas hasta ahora conocidas [5, 10, 13, 16, 17, 18, 27, 30, 32, 34, 40].

1.3.1 Clasificación de Flynn

La taxonomía de Flynn es la clásica clasificación usada en computación paralela, la cual usa ideas familiares al campo de la computación convencional para proponer una taxonomía de arquitecturas de computadores. Esta es una de las más viejas, pero es la más conocida hasta nuestros días. La idea central que se usa se basa en el análisis del flujo de instrucciones y de datos, los cuales pueden ser simples o múltiples, originando la aparición de 4 tipos de máquinas. Es decir, esta clasificación está basada en el número de flujos de instrucciones y de datos simultáneos que pueden ser tratados por el sistema computacional durante la ejecución de un programa. Un flujo de instrucción es una secuencia de instrucciones transmitidas desde una unidad de control a uno o más procesadores. Un flujo de datos es una secuencia de datos que viene desde un área de memoria a un procesador y viceversa. Se pueden definir las variables n_i y n_d como el número de flujos de instrucciones y datos, respectivamente, los cuales pueden ser concurrentemente procesados en un computador. Según eso, las posibles categorías son:

SISD (Single Instruction Stream, Single Data Stream)

Esta representa la clásica máquina de Von-Neumann, en la cual un único programa es ejecutado usando solamente un conjunto de datos específicos a él. Está compuesto de una memoria central donde se guardan los datos y los programas, y de un procesador (unidad de control y unidad de procesamiento), ver figura 1.2. En este caso, $n_i = n_d = 1$. En esta plataforma sólo se puede dar un tipo de paralelismo virtual a través del paradigma de multitareas, en el cual el tiempo del procesador es compartido entre diferentes programas. Así, más que paralelismo lo que soporta esta plataforma es un tipo de concurrencia. Los PC son un ejemplo de máquinas que siguen este tipo de procesamiento.

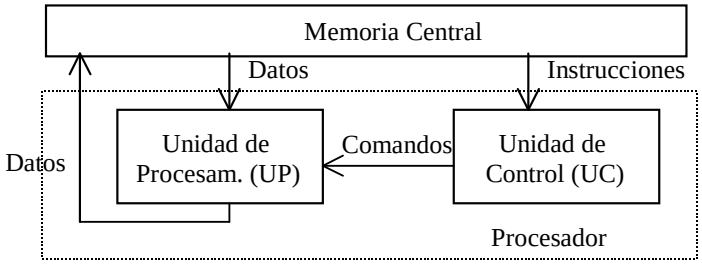


Figura 1.2. Arquitectura tipo SISD (Máquina Von-Neumann).

SIMD (Single Instruction Stream, Multiple Data Stream)

Arreglo de elementos de procesamiento, todos los cuales ejecutan la misma instrucción al mismo tiempo. En este caso, $n_i = 1$ y $n_d > 1$. El enfoque de paralelismo usado aquí se denomina *paralelismo de datos*. Los arreglos de procesadores son típicos ejemplos de esta clase de arquitectura. En estas arquitecturas, un controlador recibe y decodifica secuencias de instrucciones a ejecutar, para después enviarlas a múltiples procesadores esclavos. El arreglo de procesadores procesa los datos que llegan a los diferentes procesadores, usando la instrucción enviada por el controlador. Los procesadores están conectados a través de una red. Los datos a tratar pueden estar en un espacio de memoria que es común a todos los procesadores o en un espacio de memoria propio a cada unidad de procesamiento (ver figura 1.3). Todos los procesadores trabajan con una perfecta sincronización. SIMD hace un uso eficiente de la memoria, y facilita un manejo eficiente del grado de paralelismo. La gran desventaja es el tipo de procesamiento (no es un tipo de procesamiento que aparece frecuentemente), ya que el código debe tener una dependencia de datos que le permita descomponerse.

Su funcionamiento es el siguiente: un simple controlador envía las instrucciones, una a una, a un arreglo de procesadores que operan en el esquema maestro-esclavo. Es decir, las instrucciones son difundidas desde la memoria a un conjunto de procesadores. Así, cada procesador es simplemente una unidad aritmética-lógica y se tiene una sola unidad de control. Todos los procesadores ejecutan cada operación recibida al mismo tiempo, por lo cual, cada uno ejecuta la misma instrucción sobre diferentes datos. Los sistemas modernos SIMD tienen un CPU adicional para procesamiento escalar, de tal manera de mezclar operaciones entre el arreglo de procesadores y el procesador secuencial estándar; esto es debido a que en la mayoría de las aplicaciones paralelas se tienen fases de código secuencial y de código paralelo. De esta forma, la parte de código secuencial es ejecutado en el CPU adicional. Esto también implica dos procesos diferentes para mover los datos entre el arreglo de procesadores y el procesador secuencial: en un caso se *difunden* los datos desde el procesador secuencial al arreglo de procesadores; en el otro caso, al moverse los datos del arreglo de procesadores al procesador secuencial, los datos son *reducidos*.

Para el programador pareciera que se ejecutara su programa de manera secuencial, con la diferencia de que sus intrucciones se ejecutan de manera múltiple sobre diferentes datos y no una sola vez (por lo que para el usuario sigue siendo una máquina secuencial desde el punto de vista de la programación). Este tipo de plataforma computacional se ha desarrollado debido al gran número de aplicaciones científicas y de ingeniería que se adaptan bien a él (procesamiento de imágenes, simulación de partículas, métodos de elementos finitos, sistemas moleculares, etc.), en las cuales se tiene una simple secuencia de instrucciones, operando al mismo tiempo sobre un conjunto de datos. Un ejemplo de máquina que trabaja bajo este enfoque es la CM-2.

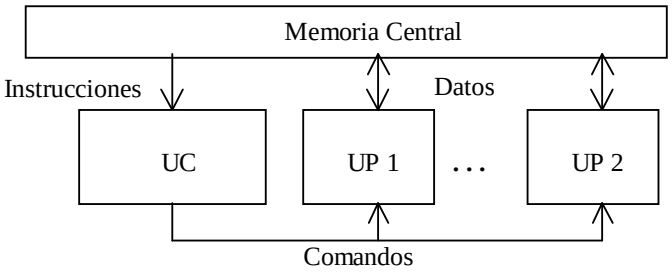


Figura 1.3. Arquitectura tipo SIMD.

MISD (Multiple Instruction Stream, Single Data Stream)

Estas son computadoras con elementos de procesamiento, cada uno ejecutando una tarea diferente, de tal forma que todos los datos a procesar deben ser pasados a través de cada elemento de procesamiento para su procesamiento (ver figura 1.4). En este caso, $n_i > 1$ y $n_d = 1$. Implementaciones de esta arquitectura no existen realmente, excepto ciertas realizaciones a nivel de mecanismos de procesamiento tipo encauzamiento (*pipelining* en ingles) (internamente en los procesadores RISC, etc.) o sistemas de tolerancia a fallas. Muchos autores consideran que esta clase no corresponde a un modo de funcionamiento realista. Otros autores consideran que representan el modo de procesamiento por encauzamiento.

La idea es descomponer las unidades de procesamiento en fases, en donde cada una se encarga de una parte de las operaciones a realizar. De esta manera, parte de los datos pueden ser procesados en la fase 1 mientras otros son procesados en la 2, otros en la tres, y así sucesivamente. El flujo de información es continuo y la velocidad de procesamiento crece con las etapas. Este tipo de clasificación de Flynn puede ser incluida dentro de la clasificación SIMD si se asemeja el efecto de estas máquinas, al tener múltiples cadenas de datos (los flujos) sobre las etapas de procesamiento, aunque también puede ser visto como un modo MIMD, en la medida de que hay varias unidades y flujos de datos y cada etapa está procesando datos diferentes.

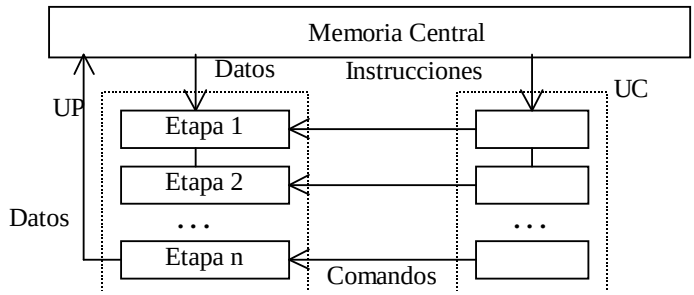


Figura 1.4. Arquitectura tipo MISD.

MIMD (Multiple Instruction Stream, Multiple Data Stream)

Es el modelo más general de paralelismo, y debido a su flexibilidad, una gran variedad de tipos de paralelismo pueden ser explotados. Las ideas básicas son que múltiples tareas heterogéneas puedan ser ejecutadas al mismo tiempo, y que cada procesador opere independientemente con ocasionales sincronizaciones con otros. Está compuesto por un conjunto de elementos de procesamiento donde cada uno realiza una tarea, independiente o no, con respecto a los otros procesadores. La conectividad entre los elementos no se especifica y usualmente se explota un *paralelismo funcional*. La forma de programación usualmente utilizada es del tipo concurrente, en la cual múltiples tareas, quizás diferentes entre ellas, se pueden ejecutar simultáneamente. En este caso, $n_i > 1$ y $n_d > 1$. Muchos sistemas de multiprocesadores y sistemas de múltiples computadores están en esta categoría. Un computador MIMD es fuertemente acoplado si existe mucha interacción entre los procesadores, de lo contrario es débilmente acoplado.

Las MIMD se pueden distinguir entre sistemas con múltiples espacios de direcciones y sistemas con espacios de direcciones compartidos. En el primer caso, los computadores se comunican explícitamente usando pase de mensajes entre ellos según una arquitectura NUMA (non-uniform memory access). En el segundo caso, al usarse una memoria centralizada, se trabaja con modelos UMA. En el primer caso se habla de MIMD a memoria distribuida y en el otro de MIMD a memoria compartida. Por el problema de gestión del acceso concurrente a la información compartida, se origina un cierto no determinismo en las aplicaciones sobre estas plataformas.

1.3.2 Modelo Basado en la Organización de la Memoria

La clasificación anterior no permite distinguir la organización interna de la memoria, por lo cual la taxonomía de Flynn deja por fuera cierta clase de máquinas. En esta sección tratamos de dar una clasificación que cubra a estas máquinas [3, 24, 35, 37]. Para eso, presentaremos una clasificación en la cual combinamos los modos de funcionamiento paralelos más comunes de la taxonomía de Flynn (SIMD y MIMD), con los tipos de organización de la memoria (Memoria Compartida (SM) y Memoria Distribuida (MD)).

SIMD-Memoria Compartida:

Se caracteriza por un control centralizado y datos centralizados. Las máquinas clásicas de este tipo son las máquinas vectoriales mono-procesadores con encauzamiento. Su funcionamiento es el siguiente: se realiza una única operación (por ejemplo, la adición) sobre un conjunto de datos múltiples (por ejemplo, dos vectores escalar). Las operaciones se realizan de manera secuencial pero bajo un modo de funcionamiento de encauzamiento (ver figura 1.3).

SIMD-Memoria Distribuida:

Estas máquinas se caracterizan por un control centralizado y los datos distribuidos (ver figura 1.5). Normalmente, los procesadores de estas máquinas son de bajo poder, y la potencia de cálculo de la máquina paralela es obtenida por el gran número de procesadores usados. En esta arquitectura, cada procesador tiene una memoria local pequeña y recibe las instrucciones de una unidad de control central para ejecutar la misma instrucción, conjuntamente con el resto de procesadores, de manera sincronizada.

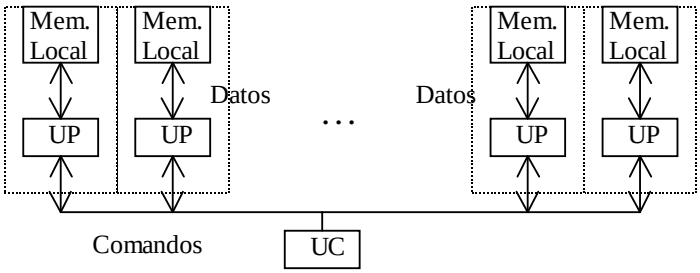


Figura 1.5. Arquitectura tipo SIMD-Memoria Distribuida.

MIMD-Memoria Compartida:

En esta arquitectura, el conjunto de procesadores comparte la misma memoria (ver figura 1.6). Cada procesador puede ejecutar una instrucción diferente y el acceso a la memoria se hace a través de una red de interconexión. La memoria se puede dividir en varios bancos, y en un momento dado, un banco puede solamente ser accesado por un procesador. Este tipo de máquina se desarrolló mucho en los años 80, pues llevar un código secuencial para ser ejecutado en este tipo de máquina resulta muy fácil. La complejidad de la red de interconexión aumenta rápidamente al aumentar el número de procesadores, ya que todos deben poder accesar de manera eficaz todos los bancos de memoria, lo que limita su uso a un número reducido de procesadores. Por poseer pocos procesadores, se usan poderosos procesadores, de tal manera que la potencia de la máquina paralela está dada por la potencia de los procesadores más que por el número de ellos.

Como se dijo antes, el controlador y el proceso son replicados (cada uno se llama nodo), mientras existe una única memoria compartida por todos los nodos. La conexión entre los procesadores y la memoria es a través de la red de interconexión y se usa un espacio de direcciones simples, lo cual indica que cualquier localidad de memoria es vista como una única dirección y esta dirección es usada por cada procesador para accesar la localidad. Una gran ventaja de este enfoque es que no hay que modificar los programas secuenciales en cuanto al acceso a memoria. Programar en estas plataformas implica tener los procesos y sus datos en el espacio de memoria compartida entre los procesadores. Eso también permite la posibilidad de compartir variables y secciones de código entre las aplicaciones. El problema de sincronización es el problema típico de compartir recursos que también se encuentran en las máquinas secuenciales. Así, la

introducción de hilos, mecanismos de sincronización (exclusión mutua, etc.), manejo de secciones críticas, etc., son cruciales en estas plataformas.

Uno de los problemas a resolver tiene que ver con la dificultad que tienen los procesadores para accesar rápidamente a la memoria. Una forma de hacerlo consiste en definir estructuras de memoria jerárquicas o distribuidas, lo que permite accesos físicos a localidades de memoria cercanas a los procesadores (explotar la localidad de los datos). Esto permite que se hagan más rápidos accesos que en el caso de localidades distantes de memoria, lo que hace surgir nuevos problemas, como la coherencia de las memorias (particularmente, de las memorias escondidas). El termino UMA (acceso uniforme a la memoria) se usa en este caso, en oposición al acceso no-uniforme a la memoria (NUMA).

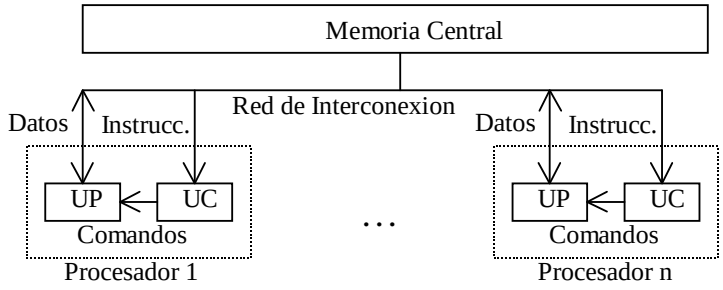


Figura 1.6. Arquitectura tipo MIMD-Memoria Compartida.

La comunicación entre los procesadores se hace a través de la memoria, por lo que se necesita una programación rigurosa a nivel de la organización de la memoria para evitar conflictos de acceso a una misma localidad de memoria. El objetivo de rendimiento es mover los datos a usar a la memoria antes de que sean requeridos y tiene que ver con el hecho de que entre más cerca estén los datos del procesador, más rápido serán accesados. El grave problema es a quién sacar y cuándo moverlo, de tal manera que cuando un procesador los necesite, ya los datos estén en un sitio de acceso rápido. Como también se dijo, la memoria puede ser dividida en módulos (bancos de memorias), cada uno con sus canales de Entrada/Salida, de manera que cada módulo se conecte a los procesadores usando una red de interconexión.

MIMD-Memoria Distribuida:

Este tipo de máquina se caracteriza por un control distribuido y una distribución de los datos. Cada procesador es autónomo, con las mismas características de una máquina secuencial. En este caso se replica la memoria, los procesadores y los controladores (ver figura 1.7). Así, cada procesador tiene su memoria local. Los procesadores se pueden comunicar entre ellos a través de una red de interconexión, la cual puede tener diferentes formas. Esta arquitectura puede explotar la ventaja de las diferentes arquitecturas presentadas (gran número de procesadores, los cuales pueden ser de gran poder). Los procesadores se comunican por pase de mensajes. Los rendimientos de estas máquinas están muy ligados a los rendimientos de las comunicaciones de las máquinas, y el

paralelismo a utilizar debe ser explícito, lo que hace requerir nuevos sistemas de explotación y ambientes de programación.

Como la memoria local de cada procesador no puede ser accesada por otro computador, la red de interconexión es usada para que los procesadores se comuniquen a través de pase de mensajes (los mensajes incluyen datos, requerimientos, etc.). Programar una aplicación para estos sistemas implica dividir el programa en tareas independientes, algunas de las cuales podrán ser ejecutadas concurrentemente. En este caso se habla de espacios de memorias locales, y la comunicación entre los procesadores se hace explícitamente a través de pase de mensajes, usando la red de interconexión.

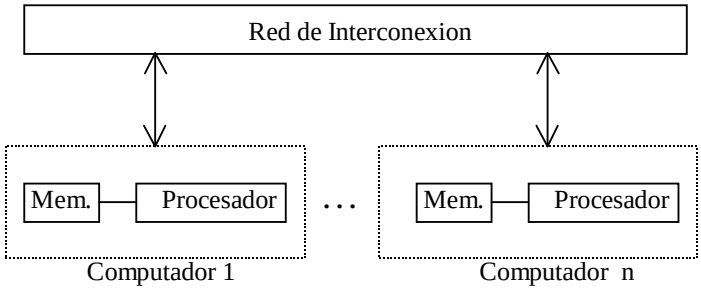


Figura 1.7. Arquitectura tipo MIMD-Memoria Distribuida.

El componente central de esta arquitectura para un buen rendimiento, es la red de interconexión. El tipo de configuración soportada y sus elementos varían increíblemente (desde PC hasta procesadores RISC). Hay dos maneras para realizar las Entradas/Salidas: cada procesador tiene su propio sistema de Entradas/Salidas, o se tiene una red de Entradas/Salidas que se usa ya sea porque hay nodos servidores de Entradas/Salidas o las unidades de Entradas/Salidas están directamente conectadas a la red. Entre sus ventajas está lo fácil que crecen (tanto a nivel de tamaño de memoria, de ancho de banda, como de poder de cálculo), que pueden emular los otros paradigmas (por ejemplo, una SIMD), y que pueden compartir recursos y usar todos los avances en tecnología de computadores. Su mayor desventaja es el ineficiente uso de la memoria y los problemas de sincronización.

MIMD-Memoria Distribuida-Compartida:

Lo mejor de las dos arquitecturas anteriores se pueden combinar para diseñar una máquina distribuida-compartida (ver figura 1.8). Es decir, se desea mantener el acceso a la memoria uniforme sin los problemas de coherencia de memoria escondida (cache) ni manejo de memoria virtual. Así, se logra un espacio de direcciones globales que facilita la programación, con la facilidad de la construcción de las máquinas a memoria distribuidas. La idea es que un procesador pueda accesar toda la memoria como si fuera un simple espacio uniforme de memoria. Cuando un procesador necesita accesar una localidad de memoria que está en otro procesador, ocurre un pase de mensajes que esconde el hecho de que la memoria está distribuida, lo cual se conoce como memoria virtualmente compartida. Por supuesto que los accesos remotos tomarán más tiempo que

los accesos locales. Para eso se agrega una capa de software al hardware de memoria distribuida que permite a todos los procesadores acceder de manera transparente a toda la memoria distribuida. Un acceso a una memoria no local es realizado por un envío de mensajes generados por dicha capa, con el fin de realizar el acceso a distancia. El programador puede usar un lenguaje de programación implícito usando la memoria compartida o un lenguaje de programación explícito con pase de mensajes.

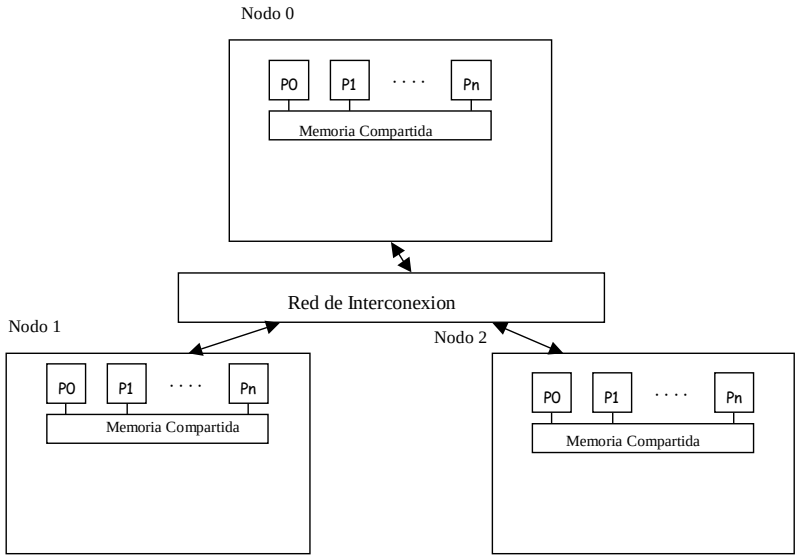


Figura 1.8. Una posible arquitectura tipo MIMD-Distribuida-Compartida.

Una extensión natural a la taxonomía Flynn, en particular al modelo MIMD, es la arquitectura MPMD (múltiples programas, múltiples datos), en la cual en vez de instrucciones, son múltiples programas los que son ejecutados en diferentes sitios, pudiendo ser algunos de ellos copias de otros. Otro modelo es la estructura SPMD (Single Program, Multiple Data), que es un tipo derivado del modelo SIMD, pero a diferencia de éste, es completamente asíncrono, donde se tiene una sola copia de un programa, que es ejecutada sobre diferentes datos. En este caso, quizás diferentes ramificaciones del programa se ejecutan en cada sitio, donde cada uno hace algo distinto del mismo programa. Este caso es interesante, por ejemplo, cuando los datos no son regulares.

1.3.3 Modelo Basado en la Coordinación

Hay dos métodos generales de coordinación en computación paralela: síncrono o asíncrono [13, 18, 25, 26, 28, 29, 34, 39]. En el primer caso se obliga a que todas las operaciones sean ejecutadas al mismo tiempo y se trata de hacer cumplir las dependencias, en los órdenes de ejecución de las tareas para los diferentes programas. En el segundo no existe tal restricción. A continuación presentaremos una clasificación basado en esto.

Arquitecturas síncronas:

Dentro de los computadores sincrónicos se pueden nombrar: vectores/arreglos, máquinas SIMD y sistólicas. A continuación un listado de las máquinas síncronas:

a) *Computador secuencial:* Basado en la máquina de Von-Neumann, las instrucciones son ejecutadas secuencialmente pero pueden ser encauzadas. Esta arquitectura incluye una Unidad de Control, una Unidad Aritmética Lógica y una memoria (ver figura 1.2).

b) *Arquitecturas encauzadas*: El *encauzamiento* divide las operaciones a ejecutar en pasos, los cuales son ejecutados unos detrás de otros. La entrada de una fase es la salida de la precedente (en clasificación Flynn pertenecen a las arquitecturas MISD, ya que la misma data es procesada por un flujo de sucesivas instrucciones elementales). Así, el encauzamiento se basa en la descomposición de la operación (F) en un conjunto de *k* operaciones (fases) F_i, tal que F= F_1, … F_k. La idea de la descomposición de F es que va realizando en secuencia el procesamiento de los datos de entrada, tal que cada dato que llega a la entrada de F_i, éste lo procesa y envía el resultado al siguiente F_{i+1}. A cada unidad de procesamiento que procesa una parte de los datos (un F_i), se le llama una fase, y todos los datos deben transitar las *k* etapas en un orden especifico.

Para clarificar este concepto, consideremos la adición de dos números de puntos flotantes en cuatro fases. Si se quisiera hacer la adición de *n* pares de números, asumiendo que la suma entre ellos es lógicamente independiente, se podría planificar la ejecución de las diferentes fases (F_1, F_2, F_3, F_4) de las *n* instrucciones (I_1, ..., I_n) para hacer ese cálculo, de la siguiente forma:

Reloj	Fase Ejecutada				
1	F_1 de I_1				
2	F_1 de I_2	F_2 de I_1			
3	F_1 de I_3	F_2 de I_2	F_3 de I_1		
4	F_1 de I_4	F_2 de I_3	F_3 de I_2	F_4 de I_1	I_1 termina
...					
n	F_1 de I_n	F_2 de I_{n-1}	F_3 de I_{n-2}	F_4 de I_{n-3}	I_{n-3} termina
n+1		F_2 de I_n	F_3 de I_{n-1}	F_4 de I_{n-2}	I_{n-2} termina
n+2			F_3 de I_n	F_4 de I_{n-1}	I_{n-1} termina
n+3				F_4 de I_n	I_n termina

Como se ve, se necesitan n+3 ciclos de reloj para terminar de ejecutar las *n* instrucciones. En general, se necesitan n+k-1 ciclos de reloj para ejecutar *n*

instrucciones en *k* fases. Si se compara con el rendimiento de máquinas sin encauzamiento, en este último caso se necesitarían *kn* ciclos de reloj para ejecutar *n* instrucciones en *k* fases.

Ejemplo 1.1: Se quiere calcular $f(x,y)=\sqrt{(3(x+y))}$. Los posibles subprocesos son añadir *x+y*, multiplicar ese resultado por *3* y hallar la *raíz cuadrada* del resultado final. Además, se tiene una gran cantidad de información a la cual se le debe dar ese procesamiento. A cada subproceso se le asigna una unidad funcional y la salida de una unidad funcional se conecta a la entrada de la otra de acuerdo al orden en que se deben hacer los cálculos. En el momento 1, x_1 y y_1 son sumados en la unidad 1 dando z_1. En el momento 2 el resultado de la suma de x_1 y y_1 (z_1) se multiplica por 3 en la unidad 2, dando w_1, y en paralelo, en la unidad 1 x_2 e y_2 son sumados (z_2). Este proceso sigue, y en un momento dado las tres unidades están procesando diferentes x_i, y_i, z_i y w_i.

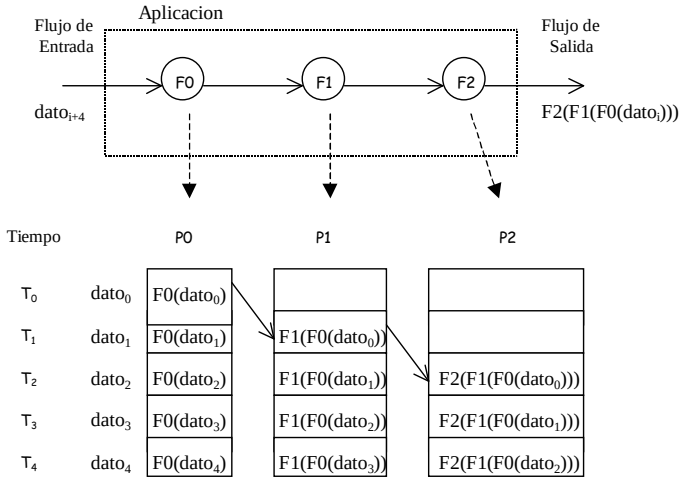


Figura 1.9. Encauzamiento.

El uso de encauzamiento no acelera la ejecución de un solo dato, pero si de un conjunto de datos. Sin encauzamiento, el tratamiento de datos seria $T = \sum_n t_n$, donde t_n es el tiempo de procesamiento de cada uno de los *n* elementos, mientras que con encauzamiento seria $T_e=(k+n-1)T_i^{max}$, donde $T_i^{max} = max_i(T_i)$ y T_i es la duracion de cada fase. Si no se cumple que $T_e < T$, no tendría sentido aplicar encauzamiento. El tiempo $(n-1)T_i^{max}$ es la latencia del encauzamiento, a partir del cual se producen resultados cada T_i^{max} unidades de tiempo.

Discusión: Si suponemos que un encauzamiento ideal corresponde al hecho de que cada f_i necesita el mismo tiempo de cálculo $T_i = T/k$ (k siendo el número de fases), tal que $T_i^{max} = max_i(T_i)= T/k$, entonces la aceleración (ver en el capítulo 5 la expresión de aceleración) seria:

$$k/(1+(k-1)/n)$$

Para n=1 el factor de aceleración sería 1 (lo que confirma que no se puede acelerar una operación escalar). Si n>>k entonces el factor de aceleración tiende a *k* y el paralelismo será máximo. Así, el paralelismo máximo es determinado por el número de etapas, tal que a más etapas es más importante las aceleraciones que se pueden obtener. El termino (k-1)/n es el responsable de la degradación, ya que corresponde al número de pasos iniciales del encauzamiento.

Entre los problemas de esta arquitectura, tenemos:

- Existe una latencia lineal asociada al sistema, que consiste en el tiempo entre cada comienzo de procesamiento y cuando se obtiene el primer resultado.
- Cada problema debe definir su propia organización del encauzamiento (ya que el tipo de subprocesos y el número de unidades requerido puede variar).

Una ventaja sería:

- Como el encauzamiento se basa en un procesamiento secuencial, no se requiere cambios desde el punto de vista de la programación.

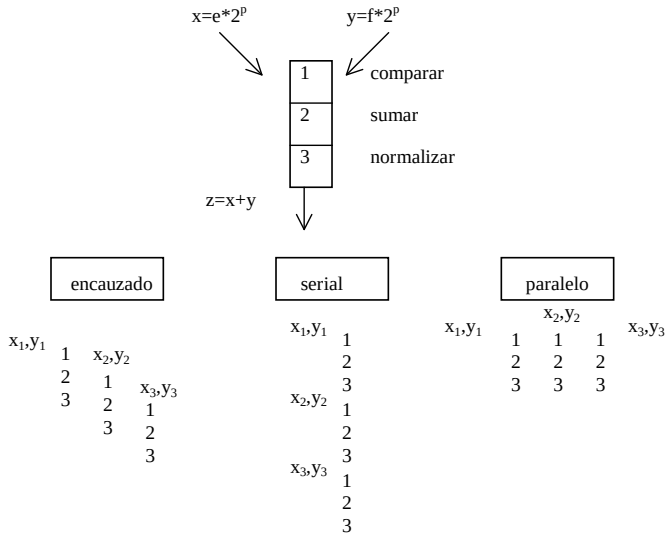


Figura 1.10. Comparación entre procesamiento serial, encauzado y paralelo.

c) *SIMD:* varias unidades de procesamiento son supervisadas por la misma unidad de control. Todos reciben la misma instrucción (o programa, si es una estructura SPMD) de la unidad de control, pero trabajan sobre diferentes datos. Como cada unidad de control ejecuta la misma instrucción al mismo tiempo, las operaciones operan sincrónicamente (ver figuras 1.3 y 1.5).

d) *Máquinas sistólicas*: es un multiprocesador encauzado en el cual los datos son distribuidos desde la memoria a los arreglos de vectores, y cada línea del arreglo (el conjunto de arreglos) ejecuta la misma operación. Así que puede verse como una

mezcla de una máquina SIMD y encauzada. La eficiencia de este enfoque depende de la regularidad de los datos y las fases, y se adapta bien para paralelismo de grano fino. Este esquema se basa en el paradigma de programación basado en el paralelismo de datos. La idea es que la data es impulsada/bombeada a través de estructuras de cálculos paralelos de una manera regular similar.

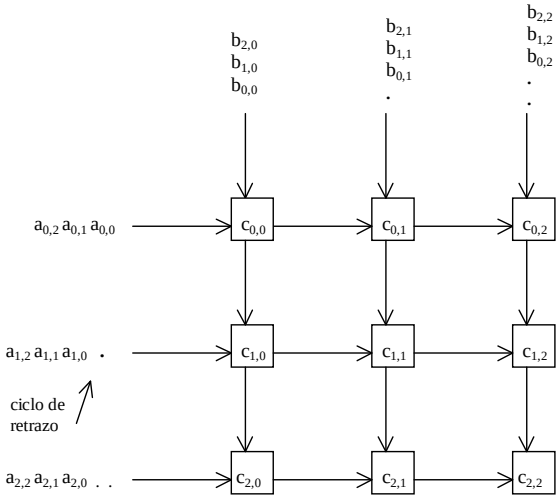


Figura 1.11. Arreglo Sistólico en Multiplicación de Matrices.

En el ejemplo de esta figura se siguen los siguientes principios:

- El flujo de datos va solo en una dirección a través del arreglo.
- Cada elemento del arreglo ejecuta la misma operación.

En ese ejemplo, cada nodo (o procesador) tiene cuatro vecinos y el arreglo tiene la forma de una malla. Los datos entran a cada nodo por dos de sus extremos (superior y lado izquierdo) y se mueven a través del arreglo, donde cada procesador es visitado por cada uno de los datos y pueden así ejecutar las operaciones para las cuales fueron programados. De esta manera, el poder de cálculo del arreglo es aumentado ya que actúa como un encauzamiento generalizado en dos dimensiones. Las operaciones en cada procesador son muy simples y el movimiento de los datos a través de la malla es sincronizado. En el caso particular de la multiplicación de matrices, cada procesador multiplica los dos valores que le llegan y lo suma a un valor local que tiene almacenado.

Arquitecturas asíncronas:

El ejemplo clásico son las máquinas MIMD donde cada procesador puede operar independientemente de los otros. Los intercambios de información son hechos a través de la memoria (SM) o enviando mensajes (DM).

Otra clasificación consiste en cómo se comunican los procesadores. Hasta los años 70, las arquitecturas tenían una memoria compartida por varios procesadores. Esto

generaba problemas de acceso a memoria. Varias soluciones se han dado: dividir la memoria en bancos, introducir organizaciones jerárquicas de la memoria (memorias "cache" (escondidas)), usar eficientemente las redes de interconexión, etc. Como se ha dicho antes, otro tipo de máquina consiste en usar memorias descentralizadas donde los procesadores tengan una conexión limitada (a un cierto número de procesadores).

1.4 Comunicación

Las máquinas paralelas se caracterizan por replicar algunos de sus componentes: unidades de procesamiento, unidades de control, etc. Además, es muy raro que la descomposición que uno haga de una aplicación en tareas, conlleve a que estas sean completamente independientes entre ellas. En este caso se habla de dependencia de tiempo, es decir, un procesador P1 debe culminar la ejecución de la tarea T1 antes que otro procesador P2 pueda comenzar la ejecución de la tarea T2. Normalmente, ellas deben comunicarse información para cumplir sus trabajos, ya sea:

- Compartiendo recursos: datos en la memoria compartida, etc.
- Por la transmisión de la información: envío de mensajes, etc.

En el primer caso se necesita conectar directamente los recursos compartidos (por ejemplo, la memoria compartida) a los diferentes procesadores. Estos recursos a su vez se pueden organizar de manera paralela (por ejemplo, partir la memoria compartida en bancos). En el segundo caso se deben interconectar los procesadores entre ellos, para poder intercambiar información. Así, en ambos casos se requiere de redes de comunicación. Entre más grande es el número de componentes, más importante es la red de interconexión. El problema en el diseño de redes de comunicación radica en la densidad de comunicación que se requiere que ella soporte.

1.4.1 Pase de Mensajes

Supongamos que tenemos un conjunto de procesos con su propia memoria y datos. Los datos son intercambiados entre los procesadores por medio de mensajes. El mensaje es originado en un sitio (el procesador que envía), transmitido a través de la red de interconexión, y recibido en otro lugar (el procesador receptor). Cualquier computador puede actuar como receptor o puede enviar mensajes en un momento dado. Para el *sistema de pase de mensajes* la red es un servidor, y de acuerdo a la ubicación del que envía y del que recibe la operación de comunicación puede hacerse más rápida o no. Muchos trabajos se han hecho sobre los *sistemas de pase de mensajes* para proveer redes de interconexión con topologías eficientes, o librerías con rutinas optimizadas que esconden los detalles de comunicación (PVM, MPI, etc.) [3, 21, 22, 37]. Básicamente, en un sistema de pase de mensajes cada procesador debe:

- Conocer su dirección y la del resto de los procesadores de la red.
- Poder crear y enviar mensajes.

- Poder recibir mensajes, remover los datos del mensaje y manipularlos.

Además, las dos operaciones esenciales son:

Rutina de enviar: usada para transmitir un mensaje al nodo destino. Un mensaje es un conjunto de bytes usualmente con la siguiente información: tamaño, nodo destino y dato. La operación de envío se puede describir genéricamente como:

Enviar (buzón, tamaño, destinatario, tipo)

- Buzón, dirección en memoria donde está el inicio de la información a enviar.
- Tamaño: tamaño del mensaje a enviar. Así, la información a enviar se encuentra desde la dirección inicial indicada antes, más el tamaño indicado aquí.
- Destinatario: a quien se enviara.
- Tipo: tipo de mensaje, para clasificar los mensajes desde el punto de vista de la aplicación.

Rutina de recepción: es usada para recibir mensajes. Normalmente, cada nodo tiene una cola de los mensajes a enviar y otra cola de los mensajes recibidos. Con una llamada a esta rutina se elimina uno de los mensajes de la cola de recepción. La recepción puede ser descrita como:

Recibir(buzón, tamaño, expedidor, tipo)

- Buzón, dirección en memoria donde está el inicio de la información a recibir.
- Tamaño: tamaño del mensaje a recibir. Así, la información a recibir se encuentra desde la dirección inicial indicada antes, más el tamaño indicado aquí.
- Expedidor: de quien se recibe.
- Tipo: tipo de mensaje para clasificar los mensajes desde el punto de vista de la aplicación.

En la ausencia de estas operaciones, no puede haber comunicación.

A este nivel se presentan nuevos problemas: por ejemplo, ¿Qué pasa si hay una instrucción de recepción y aun no se tienen datos? Normalmente, el proceso que recibe se debe bloquear (esperar) ya que debe tener una razón para recibir esos datos. Otra forma es que no se bloquee, pero cada cierto tiempo revisa si llegó un mensaje, o a través de una interrupción del sistema se avisa que llegó un mensaje. Otras capacidades que tiene este sistema son las operaciones de difusión de mensajes (transmitir un valor a un conjunto de sitios), las operaciones de reducción (por ejemplo, calcular la suma global de un conjunto de valores que se encuentran dispersos en diferentes sitios) y las operaciones de sincronización (una instrucción de ejecución no se puede pasar hasta que ella sea alcanzada por el resto de procesos-o un grupo de ellos). En la realidad, las implementaciones de las operaciones de reducción y sincronización son muy semejantes. Diferentes paradigmas pueden usarse con los sistemas de pase de mensajes, por ejemplo,

en una máquina SPMD se usan mensajes para repartir los datos y comunicar resultados parciales.

El envío de un mensaje involucra a un proceso emisor que realiza la operación de envío y a un proceso destinatario que realiza la operación de recepción. Si las dos operaciones se hacen al mismo tiempo se llama comunicación *síncrona* o *rendez-vous*. Es parecido a las comunicaciones telefónicas en la cual los dos interlocutores deben estar presentes al mismo tiempo. Si ambas operaciones se hacen en cualquier instante de tiempo, se habla de comunicación *asíncrona*. Este caso es parecido cuando enviamos una carta por correo a un destinatario dado. El destinatario puede haber estado esperando por ella desde hace tiempo, o recogerla un tiempo más tarde cuando desee.

La comunicación síncrona se realiza cuando los dos procesos están listos para comunicarse, lo que puede implicar tiempos de espera largos, pero permite asegurarle al emisor que el mensaje llegará al destinatario. En el otro caso (asíncrono), el emisor envía el mensaje cuando está listo para hacerlo, mientras que su recepción por parte del destinatario se hará más tarde. Su ventaja es que el emisor no debe esperar, él envía y sigue su ejecución. El receptor quizás deberá esperar, si el mensaje aun no ha llegado cuando él llama la rutina de recepción, aunque como se dijo antes, existen otros modos de operación para él. Este modo tiene tiempos de espera menores, pero el emisor nunca sabe si el receptor recibe el mensaje.

Una extensión al modo de comunicación asíncrono son los modos de comunicación con bloqueo o sin bloqueo. En el modo con *bloqueo*, al llamar la rutina de envío y de recepción, estas operaciones se terminan cuando efectivamente se han realizado. La operación de envío se termina cuando el mensaje sale del emisor, lo que no significa que el mismo mensaje ya llegó al destinatario. En el caso de la rutina de recepción, ésta se termina cuando el mensaje llega y es copiado en el buzón de recepción. En el modo *sin bloqueo* se trata de disminuir el tiempo de espera de las primitivas, así, en ambos casos las rutinas de envío y recepción requieren de una capa de software subyacente que se encarga de enviar y recibir el mensaje cuando esté disponible, y de permitir que el programa pueda continuar su ejecución. Este modo es interesante en los sistemas donde se permite comunicación, y hacer cálculos, simultáneamente, lo que puede permitir esconder los tiempos de comunicación. La dificultad de las operaciones sin bloqueo es que el programador debe verificar que las operaciones se terminen en los tiempos deseados. Además, debe también verificar si las dependencias existentes fueron respetadas (ya que la noción de tiempo no es global).

Un último modo de comunicación es por *interrupción*, el cual es sólo válido en el modo asíncrono. Es basado en un tipo de programación por evento, tal que cada vez que llega un mensaje sobre un sitio se genera una interrupción a nivel de la aplicación. También puede ser declarada en una aplicación a través de una instrucción *IRecibir(buzón, tamaño, expedidor, tipo, procedimiento)*, la cual solicita al sistema operativo de llamar al procedimiento indicado cuando el mensaje llegue. Eso es una declaración de interrupción, y cuando llega un mensaje su ejecución se interrumpe, el mensaje se coloca en el buzón, y el control es pasado al procedimiento de interrupción.

Al terminarse ese procedimiento el programa retoma su ejecución normal. Este esquema es poco utilizado.

En general, existe un conjunto de primitivas de comunicación que ofrecen soluciones a ciertos tipos de esquemas de comunicación, usados frecuentemente (al llamarse a una de ellas, se ejecuta una serie de instrucciones predeterminadas). Las clásicas son las siguientes [1, 4, 10, 11, 13, 15, 17, 20, 30, 33]:

- *Transferencia*: consiste en el envío de un mensaje de un procesador cualquiera a otro.
- *Sincronización:* el procedimiento asegura que todos los procesos lleguen a un punto dado en sus ejecuciones, llamado punto de sincronización.
- *Difusión:* consiste en el envío del mismo mensaje desde un procesador especifico a todos los otros.
- *Distribución:* es una difusión personalizada, es decir, el envío de un mensaje distinto desde un emisor especifico a cada uno de los procesadores.
- *Agrupamiento:* consiste en la operación inversa de distribución. Así, un procesador especifico recibe los mensajes distintos desde cada uno de los otros procesadores.
- *Comunicación total:* la idea es la siguiente, hay *n* personas y cada una conoce un chisme, ellas empiezan a comunicarse para transmitirse el chisme hasta que todas sepan todos los chismes. Así, cada uno de los *n* procesadores posee su propia información, y al final del algoritmo, todos los procesadores conocen todas las *n* informaciones diferentes.
- *Transposición:* es el caso en que cada procesador simultáneamente realiza una distribución.
- *Reducción:* consiste en sintetizar en un dado procesador un dato *d* a partir de *n* datos d_i enviados por los *n* procesadores. Normalmente, se usa una operación conmutativa y asociativa, llamada operador de reducción, para realizar este proceso.

1.4.2 Exclusión Mutua

En las máquinas a memoria compartida, la forma de comunicarse dos procesadores es a través de la memoria compartida entre ellos. Esto implica que se debe controlar el acceso a la memoria para que dos o más procesos no accesen al mismo tiempo la misma dirección de memoria y se creen así situaciones de inconsistencia (por ejemplo, datos modificados simultáneamente por dos procesos distintos). La situación en la que dos procesadores tratan de leer o escribir al mismo tiempo en la misma localidad de memoria es conocida como *condición de competencia*. Una manera de resolver este problema es usando un mecanismo ya conocido en los sistemas operativos, para máquinas secuenciales, para compartir recursos y memorias en ambientes de multiprogramación, la *exclusión mutua*, de tal forma de garantizar que cuando un proceso usa un espacio de memoria nadie más lo esté usando. La parte de los procesos en la que se tiene acceso a la memoria compartida se llama *sección crítica*, por lo cual se deben establecer los mecanismo para que dos procesos dados no accesen al mismo tiempo su sección crítica (así se evitará la condición de competencia).

Se han desarrollado muchos mecanismos, a nivel de lenguajes de programación y sistemas operativos, para resolver este problema [1, 9, 11, 17, 37, 40]: desactivación de interrupciones, alternancia estricta entre los procesos que requieren acceso, contadores de eventos, monitores, semáforos, etc. Quizás los semáforos han sido los más usados en las máquinas a memoria compartida, y la idea básica es que un semáforo puede tener un valor 0 o mayor (si no hay procesos esperando) o un valor negativo igual al número de procesos en espera. Existen dos operaciones atómicas que cada proceso puede realizar:

- La operación *DOWN* verifica si el valor del semáforo es mayor que 0, en cuyo caso lo disminuye y continua. En caso contrario, se va a dormir (esperar o bloquearse).
- La operación *UP* incrementa el valor del semáforo correspondiente, y si uno o más procesos dormían por ese semáforo (no pudieron completar la operación *DOWN*), el sistema elige alguno de ellos (por ejemplo, en forma aleatoria), y le permite desbloquearse y continuar (terminar su ejecución de la operación *DOWN*).

Usando esos mecanismos se puede controlar el acceso a las secciones críticas fácilmente.

1.4.3 Redes de Interconexión

Las máquinas paralelas pueden clasificarse tomando en cuenta la conexión: si es directa entre los procesadores (normalmente la red de interconexión es estática) o a través de bancos de memorias compartidos (en este caso, la red de interconexión es generalmente dinámica) [13, 17, 18, 25, 34]. Así, se identifican dos tipos de máquinas paralelas, cuyo rol de la red de interconexión en cada caso es diferente. En el caso de la memoria compartida, con uno o varios bancos de memoria, la red es usada para enlazar los bancos de memoria con los procesadores; en el caso de la memoria distribuida, la red es usada para enlazar los procesadores entre ellos.

Memoria Compartida:

En las primeras computadoras a memoria compartida, todos los procesadores podían acceder a toda la memoria. Por supuesto, esto generaba cuellos de botellas para acceder la memoria. El siguiente paso fue dividir la memoria en *bancos de memoria*, lo que requería una red de interconexión entre los procesadores y los bancos de memoria. Así, diferentes procesadores podían acceder diferentes bancos al mismo tiempo. Pero si el número de bancos y de procesadores es grande, resulta costoso desde el punto de vista tecnológico, el conectar directamente todos los bancos con los procesadores. Una solución es usar redes del tipo por *etapas*, lo que obliga que para cada acceso a memoria, se debe recorrer una secuencia de enlaces. Si el número de procesadores y bancos crece, este tipo de redes es inmanejable (muchas conexiones y latencia inaceptable), por lo que se propuso la definición de *redes parciales*, donde cada procesador sólo se conecta a ciertos bancos (ya no hay un espacio global de direccionamiento y dos procesadores pueden no tener bancos en común).

Memoria Distribuida:

Otra forma es conectar a los procesadores directamente entre ellos (la memoria esta distribuida entre los procesadores), teniéndose así una máquina a memoria distribuida sin espacio de direcciones globales.

Las principales diferencias entre las redes de máquinas a memoria compartida y a memoria distribuida son:

- *Ancho de Banda:* es la cantidad de información transmitida por unidad de tiempo, es decir, es el número de bits que pueden ser transmitidos por unidad de tiempo. En las máquinas a memoria compartida se deben transmitir un número de datos suficientes para alimentar a los diferentes procesadores, por lo que debe ser grande el ancho de banda; mientras que en las máquinas a memoria distribuida es de órdenes de magnitud más pequeña.
- *Latencia de la red:* es el tiempo que necesita la red para realizar el servicio pedido, es decir, es el tiempo para realizar una transferencia a través de la red. En las máquinas a memoria compartida debe ser pequeño, por lo que se deben tener caminos cortos y rápidos de comunicación entre los procesadores y los bancos. En las máquinas a memoria distribuida es menos crítico, y se pueden tener caminos largos.
- *Información a transmitir:* clásicamente, en las máquinas a memoria compartida se pueden transmitir datos, instrucciones, direcciones de memoria; mientras que en las máquinas a memoria distribuida se transmiten datos.

1.4.3.1 Redes de Interconexión para Máquinas a Memoria Compartida

Estas redes se distinguen por el número de etapas que se requieren para interconectar los procesadores con los bancos de memoria. Todas las redes a etapas usan pequeños

circuitos, llamados cajas de intercambios o conmutadores (o switches), caracterizados por el número *b* de entradas o salidas (casi siempre b=2 o 4, ver figura 1.12). Así, cada red se constituye de un conjunto de capas o etapas, y en cada etapa se tiene en paralelo un conjunto de conmutadores, normalmente del orden de *n/b*, donde *n* es el número de procesadores, para tomar en cuenta todas las entradas. El número de etapas es $\log_b(n)$. Los conmutadores con b=2 (el caso más simple) permiten el tránsito de información de dos maneras: derecho (o recto) y cruzado (o intercambio). De esta manera, si dos procesadores quieren acceder dos bancos diferentes, ellos pueden compartir la red, e incluso el conmutador. El único inconveniente es cuando uno de ellos necesita cruzar y el otro seguir derecho. Normalmente, en ese caso uno de las dos deberá esperar (realmente, esto depende de la implementación tecnológica del conmutador).

Figura 1.12. Un Conmutador con dos posibles modos de funcionamiento

Así, en estas redes, la topología puede variar durante la ejecución de un programa paralelo o entre la ejecución de dos programas (por eso se les llama redes *dinámicas*): por ejemplo, un procesador P puede estar conectado en el momento *t* a la memoria M_i y en el momento *t*+1 a la memoria M_{i+1}. En general, estos tipos de redes son muy comunes en los sistemas multiprocesadores que comparten recursos. Entre los modelos más típicos de estas redes tenemos:

- *Redes de 0 etapa:* Son redes conectadas a un bus (es la red más simple), tal que se tiene una sola memoria y todos los procesadores se conectan a esa única memoria (ver figura 1.13). En este caso, se comparte la red de interconexión entre las diferentes unidades, por lo que se necesita un controlador para el bus. Los accesos a la memoria son de manera secuencial. El único bus de acceso a la memoria es un cuello de botella (debido al uso exclusivo de la memoria por un único procesador en un momento dado). Existen varios algoritmos para asignar el bus: prioridad fija (cada unidad tiene su prioridad de acceso al bus), quantum de tiempo (cada unidad tiene un pedazo de tiempo para usar el bus), FIFO, prioridades dinámicas, etc.

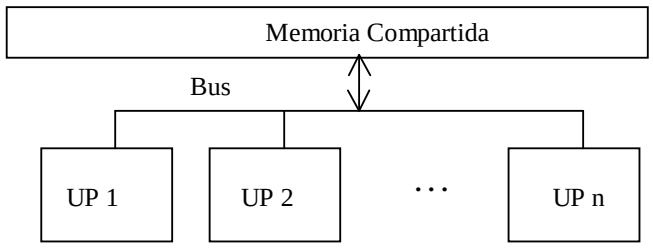


Figura 1.13. Bus.

- *Redes a 1 etapa* (o como es conocida en Ingles, crossbar): es la forma más simple, ya que todas las entradas atraviesan las líneas de salida y hay un conmutador en cada punto de interconexión (ver figura 1.14). Así, cualquier entrada puede comunicarse con cualquier salida. Se caracterizan por el hecho de que la latencia es dependiente del número de procesadores y de bancos de memoria. Por ejemplo, en la figura 1.14 para ir de P1 a M6 se necesita pasar por 8 conmutadores, mientras que para ir de P3 a M2 sólo 2. Además, si dos procesadores tratan de acceder al mismo banco aparece un conflicto.

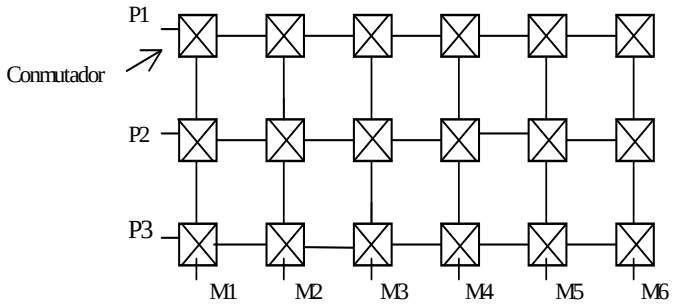


Figura 1.14. Redes a 1 etapa (3x6)

La complejidad de esta red crece rápidamente al crecer el número de procesadores, por ejemplo, para *m* bancos y *n* procesadores se necesitan m*n enlaces.

- *Redes multietapas*: Tienen por objetivo acercarse al comportamiento de las redes crossbar, es decir, tener conectados al mismo momento m*n elementos de cada lado, tratando de usar menos conmutadores, pero con más tiempo de travesía. Normalmente se conectan en forma de tablas rectangulares cuyas dimensiones son generalmente de n líneas por $\log_2(n)$ columnas (número de etapas), es decir, se requieren $n*\log_2(n)$ conmutadores. Para intercambiar información se debe pasar por las $\log_2(n)$ etapas antes de llegar a su destino. Para las definiciones anteriores, hemos supuesto que el número de bancos y procesadores es el mismo (m=n), además de ser potencia de dos. Esto no es problema porque se supone que ciertos procesadores no hacen nada o ciertas memorias nunca se acceden. La interconexión entre las entradas y salidas se realiza a través de una operación biyectiva entre ellas, y para transmitir mensajes entre ellas se debe ir colocando los conmutadores en cada etapa en su correcta posición. Su latencia es función del número de etapas. Las redes multietapas ofrecen caminos de comunicación suficientemente cortos para facilitar su posible implementacion. Entre las etapas sucesivas, ellas pueden estar interconectadas de diferentes formas. Una red multietapa es *sin bloqueo* cuando para todas las entradas inactivas siempre hay un camino hacia toda salida inactiva (por ejemplo, la red crossbar). Tenemos una red *con bloqueo* cuando no existe una total biyección entre las entradas y las salidas, en otras palabras, no toda entrada puede conectarse con toda salida (ejemplo es la red omega). Así, redes multietapas existen de varios tipos:

- *Redes Clos (n,m,r):* son redes a tres etapas, su primera etapa tiene *r* conmutadores con *n* entradas y *m* salidas, la segunda etapa tiene *m* conmutadores con *r* entradas y *r* salidas, y la última es simétrica a la primera con *m* entradas y *n* salidas. La salida *i* del conmutador *k* de la etapa 1 esta conectada a la entrada *k* del conmutador *i* de la etapa 2. Simétricamente, la salida *k* del conmutador *i* de la etapa 2 esta conectado a la entrada *i* del conmutador *k* de la etapa 3. Es una red sin bloqueo si m≥2n-1. Así, estas redes pueden hacer todo tipo de permutaciones entre entradas y salidas, pero la definición de sus conmutadores es costosa en tiempo de cálculo, y debe ser recalculada cada vez que se quieran comunicar diferentes elementos. Sin embargo, para aplicaciones numéricas paralelas en máquinas SIMD a memoria compartida es muy eficiente, ya que usa sólo parte de las permutaciones posibles entre los bancos de memoria y las unidades de procesamiento.

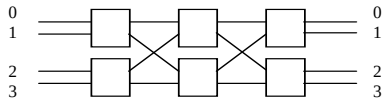


Figura 1.15. Red Clos (2,2,2)

- *Redes conectadas en cubo*: Su principio se basa en la diferencia bit a bit entre el número del procesador y el del banco a acceder. Por ejemplo, si el procesador 4 (100) quiere acceder al banco 7 (111), estos dos números difieren en el primer y segundo bit, así los conmutadores de la primera y segunda etapa a atravesar deben cruzarse, mientras que el de la tercera no (ver figura 1.16). Su construcción es recursiva, así una red de dimensión 1 tiene un conmutador. Una red de dimensión *i* consiste en yuxtaponer dos redes de dimensión *i-1* y agregar una etapa de conmutadores. Las salidas y entradas de las dos redes *i-1* se numeran consecutivamente y se hace entrar a una misma caja en la última etapa a dos entradas que difieren en 2^{i-1} (por ejemplo, en la etapa 2 las entradas 1 y 3 deben entrar al mismo conmutador ya que la diferencia entre ambos es 2). Esto se hace al conectar las salidas del primer subcubo, en el orden, con las primeras entradas de los conmutadores de la última capa; y al conectar las salidas del segundo subcubo, en el orden, con las segundas entradas de los conmutadores de la última capa. Al final, sólo basta con reordenar las salidas de la última etapa.

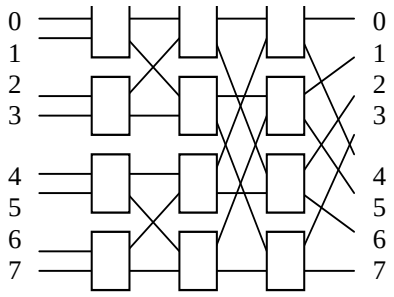


Figura 1.16. Red conectada en Cubo para 8 procesadores y Bancos de Memoria

- *Red shuffle-exchange o red omega*: Las redes de este tipo se basan en un tipo de enlace llamado *shuffle-exchange*, tal que los nodos están interconectados según las operaciones *exchange (intercambiar)* o *shuffle (mezclar)*. Su definición formal es la siguiente (ver figura 1.17): la operación *exchange* conecta dos nodos que solamente difieren en su bit de menor peso en su representación binaria. La operación *shuffle* permuta de manera circular la representación binaria de los nodos, así un nodo $a_{n-1} \ldots a_o$ es transformado en $S(a_{n-1} \ldots a_o) = a_{n-2} \ldots a_o\, a_{n-1}$, para determinar con quien estarán conectados.

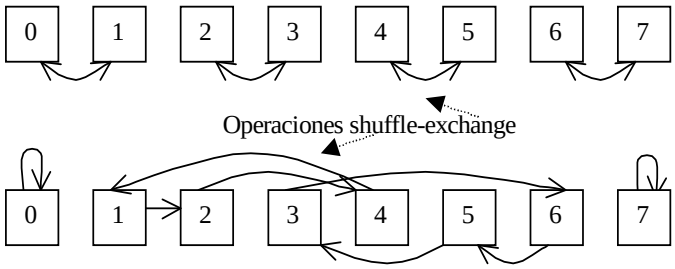


Figura 1.17. Operación *shuffle-exchange*

La red de interconexión basada en el principio shuffle/exchange se llama *Omega*, y su gran ventaja es que esta red esta conectada en cada etapa de la misma forma (ver figura 1.18). Los conmutadores son conectados entre ellos siguiendo el modo shuffle y la operación exchange se hace en los conmutadores. El algoritmo de enrutamiento es muy simple: en la etapa i, el i^{esimo} bit de la fuente y destino es comparado; si son iguales, no se cruza el conmutador, de lo contrario, se cruza (exchange). Esta red es del tipo de redes con bloqueo. Por ejemplo, al tratar de conectar 2 con 7 y 6 con 4 simultáneamente en esta red, conlleva a conflictos de posicionamiento a nivel de los conmutadores.

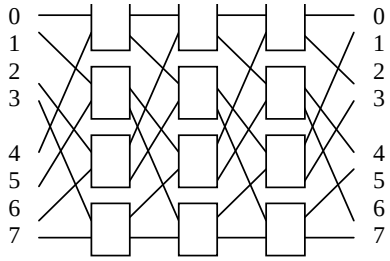


Figura 1.18. Red Omega

En general, la eficacia de estas redes en las máquinas a memoria compartida consiste en tener disponible la memoria, eliminando el mayor número de posibles conflictos. Dos tipos de conflictos existen:

- *Conflictos en la red*: las redes crossbar o bus no tienen este tipo de conflictos que consiste en que dos procesadores que quieren acceder simultáneamente dos bancos de memoria requieren usar enlaces/caminos parcialmente comunes. En crossbar porque tenemos siempre un camino directo entre cada procesador y cada banco, y en bus porque sólo hay un acceso por instante de tiempo.
- *Conflictos de acceso a memoria*: esto sucede cuando dos procesadores tratan de accesar simultáneamente a la memoria, ya que accesos concurrentes a la memoria son imposibles. Si el uso de los bancos de memoria es uniforme, este problema es reducido, pero si hay un banco más usado que otros puede generarse un efecto del tipo *bola de nieve*. Este efecto consiste en lo siguiente: se irán bloqueando procesos que han ido atravesando parcialmente la red, esto provoca el retardo de otros más, y así sucesivamente.

1.4.3.2 Redes de Interconexión para Máquinas a Memoria Distribuida

En este caso, los procesadores se interconectan a través de la red. Normalmente son redes *estáticas*, es decir, redes en las que la topología es definida una vez por todas, cuando se construye la máquina. En estas redes es más reducido el ancho de banda requerido, ya que la información a intercambiar son mensajes. Son típicas de plataformas basadas en el paradigma de pase de mensajes. Se usa un grafo G=(V,E), donde V es el conjunto de vértices/nodos y E el conjunto de arcos, para describir la red de interconexión. Los procesadores son representados por los vértices del grafo y los enlaces de comunicación por los arcos. Clásicamente, se usan grafos simples no orientados (ya que típicamente los enlaces de comunicación son bidireccionales). Varios conceptos se deben introducir a este nivel, los cuales pertenecen a la teoría de grafos:

- Un *camino de longitud k* está compuesto por una secuencia de k arcos adyacentes (arcos que comparten uno de sus nodos). Así, un camino se define por el número y largo de los arcos que interconectan dos nodos dados.
- Un *ciclo* es un camino donde el nodo origen y el extremo son el mismo.
- Un grafo es *conexo* si existe un camino entre dos nodos cualesquiera de G.
- La *distancia* entre dos nodos *u* y *v*, es el camino más corto de los diferentes caminos entre *u* y *v*.
- El *diámetro* de G es la distancia más grande entre dos nodos cualquiera del grafo (indica el camino más largo a recorrer por un mensaje entre dos procesadores dados del sistema). El valor del diámetro nos da el peor caso de retraso, ya que es la máxima distancia a que un mensaje debe viajar. Un valor de diámetro grande implica una mayor distancia entre nodos extremos, lo que hace más larga la fase de comunicación en el sistema modelado por ese grafo. Por ejemplo, si se quiere ordenar un conjunto de números, dispersos en los diferentes sitios de la red, y si el diámetro es d, se necesita como mínimo *d* pasos de comunicación para realizar ese ordenamiento.
- El *grado de un nodo* es el número de vecinos del nodo (nodos interconectados a él a través de un camino de longitud 1), es decir, es el número de arcos que llegan a un nodo. Si todos los grados de los diferentes nodos son iguales se dice que el grafo es *regular* (veremos otra definición de grafo *regular* en el siguiente párrafo).

- El *grado máximo* de un grafo es el nodo con mayor grado del grafo, y el *mínimo* lo inverso. Si ambos son iguales, es un grafo *regular* (es casi siempre el caso en máquinas a memoria distribuida).
- Un grafo es *k-conexo* si hay al menos *k* caminos entre dos nodos cualesquiera del grafo. Este valor indica la *conectividad* del grafo (el número de caminos diferentes que se pueden usar para conectar dos procesadores en el sistema). Así, indica las diferentes maneras en que dos nodos se pueden conectar. Entre más grande es este valor la red es más resistente a fallas.
- *Ancho de Bisección*: número de enlaces que deben ser cortados para dividir una red en dos partes iguales.
- *Red completa*: cada nodo tiene un enlace a cualquier otro nodo. Este tipo de interconexión es lógico cuando el número de procesadores es poco. Para otros casos, el número de interconexiones es impráctico a nivel económico y tecnológico.
- *Simetría:* si los arcos y nodos son intercambiables.

A partir de estas propiedades matemáticas, se pueden comparar las diferentes redes según los siguientes criterios:

- Número de conexiones de cada procesador.
- Número de nodos.
- Diámetro y distancia media.
- Algoritmo de enrutamiento.

A continuación se presentan los grafos más usados para describir las redes de inteconexiones en máquinas a memoria distribuida:

- *Lineal/anillo:* es una fila de nodos con solamente conexiones con sus vecinos. Los procesadores son ordenados de 0 a n-1, cada nodo con 2 vecinos (predecesor y sucesor). En el caso lineal, el diámetro es n-1 (ver figura 1.19). En el caso de los anillos, el último nodo se conecta con el primero. Así, un anillo de orden *n* es un grafo con *n* nodos de tal manera que siempre hay un arco entre el nodo *u* y el nodo *(u+1)mod(n)*. En este caso el diámetro es n/2, y el grado, conectividad y ancho de bisección del grafo es 2. Ambos tipos de topología no son eficientes, ya que los nodos son distantes y la conectividad es pequeña. El opuesto a ellos son las redes punto a punto que requieren n^2 conexiones y cuya conectividad es *n* (pueden ser trasmitidos *n* mensajes simultáneamente).

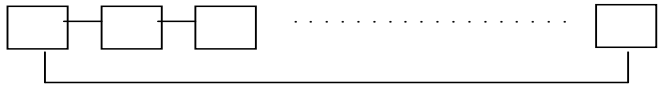


Figura 1.19. Lineal.

- *Malla:* dado n enteros p_i, tal que $\forall i=1, ..., n, p_i \geq 2$, se dice que es una malla de dimensión *n* ($M(p_1, ..., p_n)$), donde cada nodo esta numerado por la n-tupla ($x_1, ..., x_n$), con $x_i = 1, ..., p_i$. Dos nodos estarán unidos por un arco si ellos difieren en 1 en alguna de las coordenadas. Su diámetro es igual a $\Sigma_{i=1}^{n} (p_i-1)$. Es un grafo irregular (los nodos internos tienen grado igual a 2n (grado máximo), un esquinero igual a n (grado mínimo), etc.). En el caso de una malla de dos dimensiones cada nodo interno tiene 4 vecinos y los nodos de los bordes 3, a excepción de los nodos esquineros que tienen 2 (ver figura 1.20). Su algoritmo de enrutamiento se llama X-Y: se va considerando dimensión por dimensión y se recorre en cada dimensión la distancia que separa un nodo del otro.

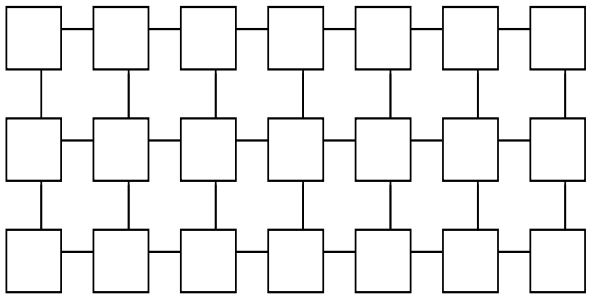


Figura 1.20. Malla M(3, 7).

- *Toro*: es una malla donde los lados (o bordes) se enlazan entre ellos (por ejemplo, en la figura anterior se enlazarían la primera y última fila, y la primera y última columna-ver figura 1.21). Es decir, si los lados de las mallas se enlazan, entonces la red es un toro (de manera circular quedaría la estructura, ya que los lados opuestos se interconectan). Usando la misma nomenclatura que antes, dado *n* enteros p_i, tal que $\forall i=1, ..., n, p_i \geq 2$, se dice que es un toro de dimensión *n* ($T(p_1, ..., p_n)$), donde cada nodo esta numerado por la n-tupla ($x_1, ..., x_n$), con $x_i = 1, ..., p_i$. Su diámetro es igual a $\Sigma_{i=1}^{n} \lfloor p_i/2 \rfloor$. El toro puede ser visto como una malla regular, es decir, que sus bordes se interconectan entre sí. Así, el grado es 2n. El ejemplo clásico de un toro de dimensión 1 es el anillo. Esta arquitectura se usa, ya que existen muchas aplicaciones en ingeniería donde el espacio de soluciones a los problemas puede estar en arreglos de dos o tres dimensiones. En general, en un toro de dimensión 2, donde los nodos se

numeran en parejas (i,j) según la coordenada donde se encuentren, habrá un arco entre un nodo (i,j) y los nodos (((i+1)mod(n)),j) y (i,((j+1)mod(n))).

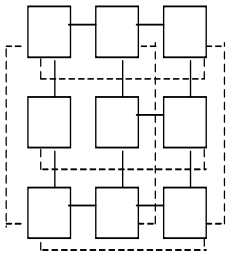


Figura 1.21. Toro T(3, 3).

- *Hipercubo*: un hipercubo de dimensión *d* es una red con $n=2^d$ procesadores, donde cada nodo tiene *d* vecinos (el mismo número de dimensiones), y donde los nodos se numeran de 0 a $2^d–1$ (es un grafo compuesto por 2^d nodos con $d*2^{d-1}$ arcos). Se usa el código de Hamming (es decir, se usa notación binaria) para definir la dirección de los nodos. Cada nodo es descrito por *d* bits y sus vecinos se obtienen complementando cada bit sucesivamente (sólo sí difiere en un solo bit su número de dirección/identificación). Por ejemplo, en un hipercubo de dimensión 3 los nodos están conectados en las dimensiones *x*, *y* y *z*, así que se necesitan 3 bits para definir su dirección. El nodo 1(001), está enlazado al nodo 0(000), 3(011) y 5(101), respectivamente. Su diámetro, grado y conectividad son iguales a d=log(n). El ancho de bisección es n/2. Esto lo hace interesante para diseñar sistemas de multiprocesadores ya que la distancia más larga entre dos nodos cualesquiera es corta, es regular, y tiene un valor de conectividad excelente. El algoritmo de enrutamiento es muy sencillo: para ir del nodo F ($f_{n-1}…f_o$) al nodo D ($d_{n-1}…d_o$) se compara cada uno de sus pares de bits para determinar el índice *k* del bit más significativo de F que será complementado, el cual corresponderá al índice de los pares de bits más significativos donde difieren f_k y d_k (uno es el complemento del otro). Así, ese valor *k* determina el nuevo nodo del camino a seguir, el cual será el identificado por $f_{n-1}f_k^c…f_o$, tal que f_k^c será el complemento de f_k. Esta operación se va aplicando entre el nuevo nodo actual, según la ruta, y el nodo destino. Por ejemplo, la ruta entre el nodo 13 (001101) y el nodo 42 (101010) en un hipercubo de dimensión 6 requiere pasar por el nodo 45 (101101), nodo 41 (101001), nodo 43 (101011) hasta llegar al 42. Los hipercubos se construyen recursivamente, así, un hipercubo de dimensión *d* esta compuesto por dos hipercubos de dimensión *d-1*.

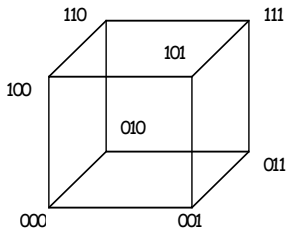


Figura 1.22. Hipercubo de dimensión 3.

- *Red De Bruijn*: una red De Bruijn(d,D) es un grafo donde los nodos son identificados por *D* letras en un alfabeto de *d* letras, tal que habrá un arco entre un nodo $a_0 \ldots a_{D-1}$ y un nodo $a_1 \ldots a_{D-1}i$, para todo $i \in \{0, \ldots, d-1\}$. Por ejemplo, en una red De Bruijn(2, 3), el nodo 001 esta conectado a 010 y 011, y los nodos 000 y 100 están conectados a 001 (100 también estará conectado a 000). Esta red es un buen compromiso ya que crece logarítmicamente al número de vértices en la red, mientras que el grado es independiente de la red. (ver figura 1.23).

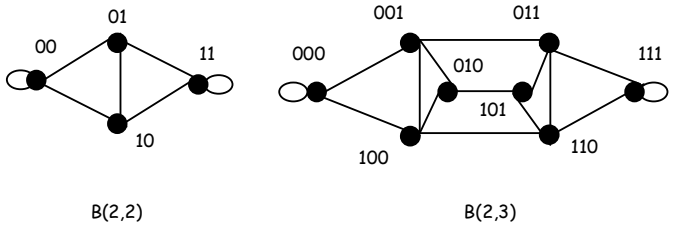


Figura 1.23. Redes De Bruijn

- *Arbol Binario*: comúnmente usado en programación paralela para definir ciertos esquemas de paralelismo (por ejemplo: el maestro-esclavo), también puede ser usado para definir máquinas jerárquicas. Un árbol binario completo tiene *h* niveles y $n=2^h-1$ vértices, cada uno con dos arcos que van al nivel inferior (ver figura 1.24). En el primer nivel esta el nodo raíz, después le siguen dos nodos en el nivel siguiente, después 4 nodos, y así sucesivamente, tal que en el nivel *j* habrán 2^{j-1} nodos (sí el árbol binario es completo). La altura del árbol, también llamado número de niveles, es el número de enlaces desde la raíz hasta los nodos hojas (los que están en el último nivel). Este tipo de interconexión es ideal para los problemas del tipo *divides y conquistarás*, pero la grave dificultad es el crecimiento de la información que ocurre al ir subiendo en el grafo (una solución es añadirles enlaces en paralelos). Los árboles forman parte del grupo de topología jerárquicas, entre las cuales también se pueden nombrar los tipos piramidales, jerarquía de buses, etc. Las topologías jerárquicas tienen la ventaja de disminuir el número de elementos conectados por cada subred interconectada (todos los problemas de contención y densidad de interconexión son reducidos), pero la jerarquía induce problemas de irregularidad (cuellos de botellas, protocolos más difíciles, etc.).

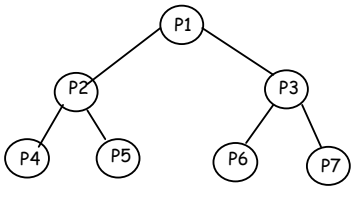


Figura 1.24. Red Arbol binario

1.4.3.3 Otros Aspectos Comunicacionales

El *enrutamiento* es un mecanismo lógico o físico para transmitir información desde una fuente a un destino, cuando un conjunto de elementos de procesamiento se deben comunicar en redes no completamente enlazadas (sean estáticas o dinámicas). Particularmente, es muy usado en las redes estáticas basadas en pase de mensajes de gran dimensión. El algoritmo de enrutamiento efectúa la escogencia del camino a seguir de entre los posibles, de tal manera de evitar los conflictos y compartir los recursos comunicacionales. Los *modos de comunicación* en estas redes (protocolos de comunicación para enviar mensajes a través de la red) requieren dar respuesta a ciertas cosas: ¿Los mensajes serán acotados?, ¿La comunicación será half-duples o full-duplex?, ¿Cómo pasa el mensaje por los diferentes nodos?. Existen varias respuestas:

- *Store and Forward*: el mensaje se para en cada nodo intermedio, éste lo guarda en su memoria local, para después reenviarlo. Además, cada nodo intermedio vuelve a analizar el camino a seguir, lo que puede hacer largo el tiempo de transmisión de mensajes.
- *Circuit switching (conmutación de circuitos):* sólo se envía la cabecera del mensaje con toda la información para establecer la conexión. A medida que va progresando ese mensaje, se va estableciendo el camino a través de los nodos intermedios. Al llegar al destino, el camino establecido se reserva y se envía una notificación de recepción para que el generador del mensaje envíe el resto del mensaje por ese camino. Al terminar el envío se liberan los enlaces del camino establecido. En este mecanismo se requieren tres fases: reservar el camino, transmitir los bloques de mensajes, liberar el camino.
- *Worm-hole:* el que envía manda sólo la cabecera como antes, pero el cuerpo del mensaje es también enviado por paquetes inmediatamente detrás de la cabecera. Si hay un bloqueo, todos son momentáneamente guardados en los nodos intermedios donde se encuentren.
- *Virtual Cut Through:* este caso es similar al anterior, sólo que si hay un bloqueo, y para evitar el posible efecto de bola de nieves en los nodos intermedios, se hace avanzar el resto de los paquetes del mensaje hasta el nodo donde esta bloqueada la cabecera.
- *Packet switching (conmutación de paquetes o mensajes*): en este caso se corta el mensaje en pequeños paquetes de tamaño igual, y cada uno es enviado independientemente entre el expedidor y el destinatario. Los paquetes pueden incluso tomar caminos diferentes en paralelo. Su problema es que se debe agregar cierta información en cada mensaje (pedazo de la cabecera), aumentando así el tamaño del mensaje. Además, se puede crear un interbloqueo entre los diferentes paquetes.

1.5 Tendencias

1.5.1 En Arquitecturas

Dos cosas se pueden decir aquí [13, 17, 34]:

- No ha sido diseñada una arquitectura paralela estándar, como sucedió en computación secuencial, y se continúan proponiendo arquitecturas radicalmente diferentes, en busca de mejoras.
- Poca información se tiene del rendimiento de las primeras máquinas paralelas. Incluso, muy pocas compañías dedicadas al desarrollo de estas máquinas han sobrevivido. Además, nuevos diseños de compañías existentes basan sus enfoques en máquinas de ellos mismos (por ejemplo, la SP2 de IBM basada en sus procesadores RISC 6000). Las compañías sobrevivientes abarcan compañías tradicionales que se mueven hacia estas máquinas (por ejemplo, IBM), y las que siempre han trabajado en el área, que quieren mejorar y extender sus características y precio/rendimiento, para ampliar el rango de posibles usuarios y aplicaciones.

Lo que es seguro es que la tendencia es producir plataformas computacionales eficientes donde se le saque el mayor provecho a la relación costo/rendimiento. Aquella plataforma que resuelva este problema, con abundante software, tendrá un futuro seguro. Los sistemas de memoria distribuidas son baratos y fáciles de escalar a un gran número de procesadores. Pero por el paradigma de pase de mensajes en que están basados, sus rendimientos se degradan mucho (trafico en la red). Así, el acceso a los datos debe mejorarse para hacer más eficiente el procesamiento paralelo en las máquinas a memoria distribuida. Por otro lado, los sistemas a memoria compartida son más costosos, y por usar conexiones complejas, su escalabilidad es difícil. A pesar de que la memoria compartida tiene tiempos de accesos a memoria varios órdenes de magnitud más pequeños que en la memoria distribuida, también tiene dos o tres órdenes de magnitud menos procesadores. Las máquinas a memoria distribuida-compartida son una primera respuesta a estos problemas, la cual es una combinación de ambas arquitecturas, agrupadas según cierta organización, para tomar las ventajas de cada una, y tener comportamientos como máquinas distribuidas y compartidas al mismo tiempo.

Un aspecto clave es que para desarrollar computadores paralelos eficientes, se deben optimizar todos sus elementos (subsistema de memoria, red de interconexión, etc.). Algunos de los aspectos en los que se debe trabajar a nivel de hardware en el futuro son [19, 31]:

- Hacer más óptima las características del hardware que puedan generar ineficiencia (por ejemplo, minimizar los conflictos de acceso a bancos de memorias, mejorar los subsistemas de E/S, etc.).
- Hacer más eficientes las redes de interconexión, ya que la misma, además de permitir a los procesadores comunicarse, también son usadas para soportar operaciones de E/S.

- Proponer estructuras de memoria más eficaces para reducir los tiempos de acceso a ellas (memorias caches, jerarquías de memorias, etc.).
- Desarrollar nuevos diseños de máquinas basados en el uso de procesadores estándar.
- Reducir los costos de desarrollo de estas máquinas para hacerlas más accesible en el mercado.

1.5.2 En Aplicaciones

A nivel de software seguirán ocurriendo los siguientes desarrollos: simuladores (se procesa una entrada de datos para producir un conjunto de resultados de interés), modelos computacionales de cálculo científico e ingenieril más complejos, aplicaciones de Base de Datos. La tendencia es que estas aplicaciones seguirán creciendo explosivamente su demanda de poder de cálculo (por ejemplo, podrán afinarse más los modelos usados en ingeniería y ciencias), y además, también crecerán sus requerimientos de manejo de grandes volúmenes de información. Esto nos tipifica dos tipos de aplicaciones: las aplicaciones orientadas al área de producción (aplicaciones en Base de Datos, etc.) en las cuales se busca mayor tiempo de máquina para ayudar a resolver problemas rápidamente, y las aplicaciones científicas en las cuales se busca explotar las altas velocidades de cálculo para mejorar los modelos teóricos y trabajos experimentales. Algunas de las áreas que podrán explotar o explotan estas plataformas son [37]:

- Inteligencia Artificial: Redes Neuronales Artificiales, Robótica, etc.
- Visualización Científica.
- Modelos basados en Elementos de Estados Finitos, incluyendo Dinámica de Fluidos Computacionales, Ecuaciones Diferenciales Parciales, etc.
- Algoritmos genéricos de computación: Ordenamiento, Procesamiento de Arboles/Grafos, etc.
- Optimización, Programación Entera, etc.
- Base de Datos.
- Simulación.
- Procesamiento de Señales y Sísmico, Computación Gráfica, etc.
- Genoma Humano.
- Correlación Oceánica.
- Modelo Climáticos, etc.

Muchas de estas aplicaciones poseen sustancial paralelismo. Para lograr que usuarios de computadores paralelos (en particular, los tradicionales usuarios: ingenieros y científicos) desarrollen eficiente códigos paralelos, pasa por facilitar poderosos lenguajes de programación, compiladores, ensambladores, sistemas operativos, etc. Los ambientes de programación son aun rudimentarios, por consiguiente, hasta ahora el desarrollo de código paralelo ha sido un trabajo intelectual muy arduo que requiere de expertos. Los fabricantes han tratado de abarcar ambas área (software y hardware), pero aun se necesita más madurez en el área de software para extender las posibles aplicaciones. Aun la gente que desarrolla software en ciertas áreas (financiero, administrativo, etc.) no está ganada

para contratar especialistas en paralelismo, hasta no tener garantías reales de las mejoras sustanciales por ejecutar sus programas en esas plataformas.

En la actualidad, algunos de los más grandes problemas son [22, 37]:

- Poco software para desarrollo de aplicaciones (entre las causas de esto es el hecho de ser un mercado muy reducido),
- Aplicaciones difíciles de portar entre máquinas (falta de portabilidad),
- Falta de estándares (los intentos como MPI-estándar para pase de mensajes-son escasos),
- Complejidad para validar la calidad del software,
- Inestabilidad de los vendedores (tanto a nivel de las empresas como de las máquinas que ellos desarrollan, lo que puede provocar incompatibilidades entre las máquinas nuevas y las viejas).

Esto puede conducir a que un software desarrollado para una máquina requiera rehacerse de nuevo en una nueva generación. Esto indudablemente repercute en la comercialización. Algo a resaltar es que muchos de los programas que se usan actualmente, fueron desarrollados hace 10-20 años. Actualmente, las máquinas que se usan tienen menos tiempo de construidas.

Otra forma de mejorar el software paralelo es desarrollar las aplicaciones basadas en procesamiento de *Hilos* (ver figura 1.25) [37]. En los sistemas de multiprocesamiento el mayor recurso es el tiempo de CPU. La idea básica es compartir eficientemente ese tiempo entre diferentes procesos. La asignación de nuevos procesos al CPU acarrea costos importantes. Ellos deben ser movidos desde y hacia el CPU permanentemente (esto se llama cambios de contextos), lo cual degrada la ejecución de los procesos. Para evitar esto surge la idea de los hilos, los cuales son también procesos, pero comparten con otros (incluyendo su padre) parte de los datos guardados en los registros del CPU necesarios para su ejecución. Un conjunto de hilos es denominado una tarea. Trabajar con tareas reduce el tiempo necesario para realizar los cambios de contextos a nivel del CPU (extracción del actual proceso del CPU para introducir otro, ya sea por expiración del quantum de ese proceso u otra razón), lo que redundará en tiempos de ejecución más eficientes.

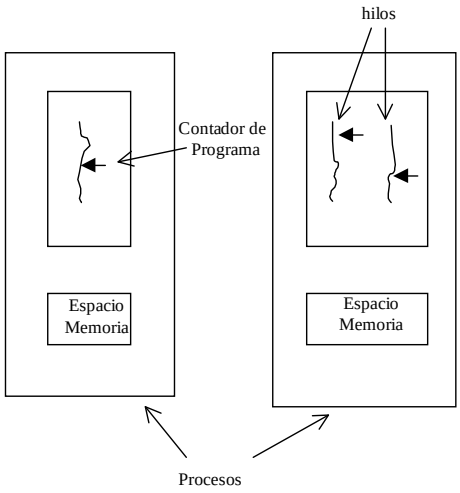


Figura 1.25. Hilos

1.5.3 En Redes de Interconexión

En programación paralela, el desarrollo de redes de computadores eficientes ha posibilitado el uso de grupos de recursos computacionales, interconectados entre sí. Así, redes de estaciones de trabajo o PC ofrecen una atractiva alternativa para usarlas como plataformas de alto rendimiento. Proyectos desarrollados a finales de los años 80 empezaron a proveer software y herramientas de programación paralela para estas plataformas, en especial PVM (Parallel Virtual Machine) fue muy popular. En años más recientes, la librería estándar de pase de mensajes MPI (Message Passing Interface) se definió. Por otro lado, una tendencia actual consiste en máquinas a memoria distribuida basadas en el uso de procesadores ordinarios. Algunas de las ventajas al usarlas son [37]:

- Estaciones de trabajo y PCs de alto rendimiento se pueden conseguir a costos muy bajos, lo que permite tener un gran número de procesadores interconectados.
- Los procesadores, con los últimos avances tecnológicos, pueden ser fáciles de incorporar en los sistemas.
- El software existente puede ser usado. Por ejemplo, estas plataformas pueden hacer uso de núcleos de sistemas operativos distribuidos (LINUX o MARCH) y de bibliotecas de pase de mensajes (PVM y MPI) para el desarrollo de aplicaciones paralelas o distribuidas.

Normalmente, la comunicación es a través de redes de tipo Ethernet (bus que comparten grupos de estaciones de trabajo), pero se han desarrollados otras tales como las redes tipos estrellas, todas ellas basadas en tecnologías tipo FFDI o ATM. Ethernet ha trabajado bien hasta ahora, pero sería más provechoso tener estructuras de interconexión de mayor rendimiento, las cuales puedan responder a muchas comunicaciones simultáneamente. Las redes para estaciones de trabajo tienen cosas en común a las redes estáticas de multicomputadores. Una de sus diferencias es que usualmente en las redes de

multicomputadores todos los computadores son iguales y en estas redes usualmente son diferentes.

A nivel de las redes de interconexión, los trabajos más importantes deben ser en torno a:

- Aumentar la conectividad en las redes (número de vecinos por nodo), lo que mejorara el grado de tolerancia a fallas del sistema.
- Aumentar el ancho de banda de los enlaces.
- Mejorar la latencia de los conmutadores.
- Manejar las comunicaciones más eficientemente (manejo de mensajes recibidos y enviados simultáneamente, etc.).
- Reducir el sobrecosto computacional en los procesadores por tareas de comunicación (mucho hardware actualmente tiene un procesador local específico para eso).

En la actualidad, muchos computadores están conectados a otros más, con el fin de compartir impresoras, software, etc. Estas plataformas son comunes en la industria, centros educativos, etc. Además, también están globalmente interconectadas a través de Internet, lo cual abarca a los computadores de los hogares. Un gran reto en el futuro, será usar esta gran red mundial en tareas de procesamiento paralelo, para explotar el gran tiempo ocioso de las diferentes máquinas extendidas a través del mundo. Otro gran reto para los próximos años consiste en desarrollar herramientas de programación paralela para Internet.

Capítulo 2.
Caracterización del Paralelismo

2.1 Generalidades

En general, la eficiencia de una máquina, para resolver un problema dado, depende de la implementación del algoritmo de resolución. En particular, las máquinas paralelas ofrecen un poder de cálculo enorme, pero explotar todas sus potencialidades no es fácil. Por otro lado, los problemas tienen diferentes características factibles a explotar, cuando se desean desarrollar aplicaciones paralelas para resolverlos. La mejor implementación secuencial de resolución de un problema dado no conlleva obligatoriamente al mejor algoritmo paralelo, ya que para desarrollar un algoritmo paralelo se deben explotar lo que denominaremos las *fuentes de paralelismo* del problema. Básicamente, existen dos modos de ver el paralelismo [3, 4, 17, 18, 35]:

- *Modo Concurrente:* supongamos una aplicación paralela compuesta de tareas, donde además, existe una estructura de intercambio de información entre las tareas. Los sistemas concurrentes buscan ejecutar simultáneamente las que se pueden (si no existen dependencia entre ellas, etc.). La red de intercambio de información es asimétrica y los intercambios asincrónicos. Los problemas de asignación de tareas y gestión de la red de interconexión son dos de los problemas más importantes a resolver. Este enfoque implica que las tareas a paralelizar deben ser explícitamente definidas.

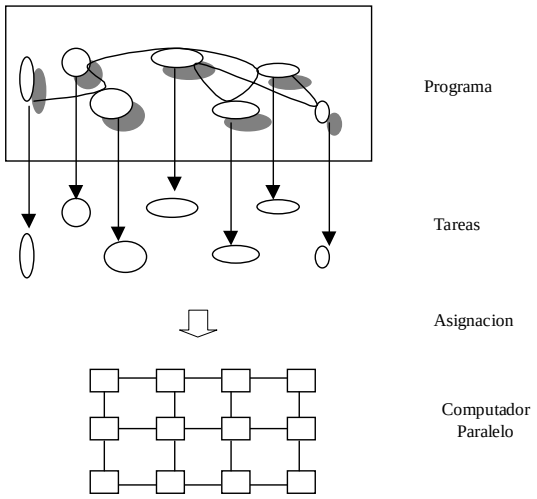


Figura 2.1. Modo Concurrente

- *Modo Paralelo*: las tareas de la aplicación se organizan en forma de una estructura regular como una tabla. Las tareas son lo más parecido posible entre ellas, y la red de interconexión es síncrona y simétrica espacialmente. Muchas veces, esto conlleva a que las aplicaciones pasen por una fase de pre-tratamiento para espaciarlas y transformarlas en tareas simétricas. Es más fácil construir máquinas paralelas para que exploten este modo. Normalmente, en este caso, el paralelismo es implícito, por lo que extraer su paralelismo es uno de los problemas fundamentales a resolver.

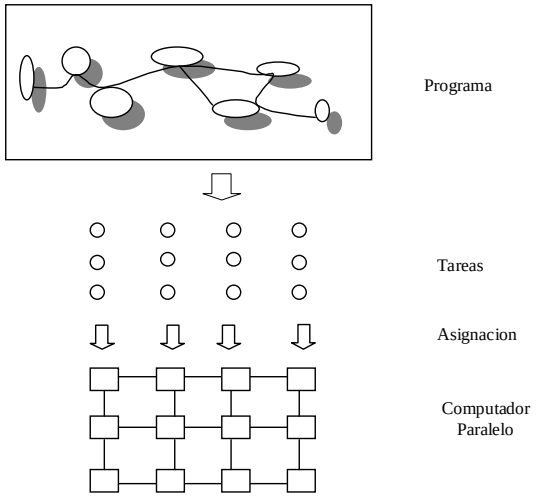


Figura 2.2. Modo Paralelo

Para ambos modos se han desarrollado diferentes lenguajes de programación, ambientes de desarrollo, etc.

2.2 Perfil de Paralelismo

El *perfil de paralelismo* de una aplicación es definido como la "cantidad de paralelismo" que una aplicación posee. Dicha cantidad esta caracterizada por dos parámetros: grado y grano de paralelismo [12, 13, 15, 17, 18].

2.2.1 Granularidad

El grano de paralelismo es definido como el tamaño promedio de las acciones (tamaño promedio de una tarea elemental) en termino de:

- Número de instrucciones ejecutadas.
- Número de palabras de memorias usadas.
- Duración de su tiempo de ejecución.

También puede definirse como la cantidad de procesamiento que una tarea realiza antes de necesitar comunicarse con otra tarea. Por supuesto, un grano puede crecer o decrecer agrupando o desagrupando tareas. Esto establece una relación entre el tiempo de cálculo con respecto al número de eventos de comunicación (*granularidad* de la aplicación). Así, la granularidad, es la relación entre la computación y la comunicación, a lo interno de una aplicación. Una *granularidad pequeña* (grano fino) implica más comunicación y cambios de contextos entre las tareas (es una aplicación con comunicación intensiva). Una *granularidad grande* (grano fuerte) implica menos comunicación, pero puede que potenciales tareas concurrentes queden agrupadas, y por consiguiente, ejecutadas secuencialmente. Así, la determinación del grano ideal, para una aplicación dada, es importante, para definir la mejor relación entre paralelismo (grano pequeño) y comunicación (grano grande). El problema de determinación del tamaño óptimo del grano para una aplicación dada es un problema de optimización del tipo *MaxMin* (Maximizar paralelismo y Minimizar comunicación), y debe estar vinculado a la máquina donde se ejecuta la aplicación. El paralelismo a grano fuerte, es normalmente usado en las arquitecturas tipo MIMD, mientras que el paralelismo a grano fino es especialmente usado en las máquinas SIMD.

Tipo de Código	Granularidad	Modo Paralelo	Arquitectura
Programas	Grueso	Concurrente	MIMD
Rutinas	Mediano-Fuerte	Concurrente	MIMD
Instrucciones	Mediano-Fino	Concurrente-Paralelo	MISD
Operadores y Bits	Fino	Paralelo	SIMD y Hardware

Tabla 2.1. El problema de Granularidad.

Como se ve en la tabla 2.1, el grano de paralelismo va desde el tamaño de un programa entero (miles de millones de instrucciones) a la de un bit. Dentro de un programa es posible variar la granularidad para aumentar los cálculos o reducir las comunicaciones. La idea central es explotar la localidad a lo interno de las aplicaciones: tareas que usan la misma información, deben ser agrupadas y ejecutadas secuencialmente, para reducir las comunicaciones. Estudiemos el problema de granularidad en el siguiente ejemplo.

Ejemplo 2.1: Determinar la descomposición idónea de dos matrices *A* y *B* que se desean multiplicar.

Supongamos la partición de B por columnas para multiplicar las matrices *A* y *B*, de manera de descomponer el cálculo de la matriz resultante *C* por columnas. Si cada una de las columnas de B está en un procesador diferente, para calcular *C* se debe enviar a *A* a cada procesador. Una manera de cambiar la granularidad de este cálculo es agrupando *m* columnas de *B* (ver figura 2.3). La información a comunicar sigue siendo constante (se debe enviar a la matriz *A* a cada procesador), sin embargo, los cálculos que ahora se pueden hacer con la información que se comunicó aumentan *m* veces, aunque esas *m*

columnas se ejecuten secuencialmente. Así, al aumentar la granularidad (disminuir el número de comunicaciones) más trabajo se ejecuta secuencialmente.

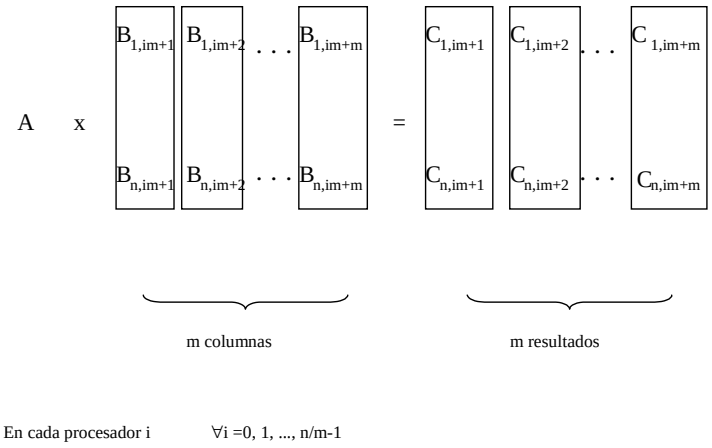


Figura 2.3. Aumento de la granularidad en el problema de multiplicación de matrices

2.2.2 Grado de Paralelismo

El grado de paralelismo es definido como el número de acciones (tareas) que se pueden ejecutar en paralelo durante la ejecución de una aplicación. Es decir, corresponde a una medida del número de operaciones que se pueden ejecutar simultáneamente, y refleja el número de procesadores a utilizar en paralelo durante la ejecución de la aplicación. Puede ser calculado estáticamente, según la estructura de la aplicación, o dinámicamente, a partir de medidas durante la ejecución.

El grado de paralelismo puede ser diferente a través de las diferentes partes de un programa, por lo que se habla de un grado de paralelismo máximo, mínimo y promedio. El *grado máximo* es el límite superior en cuanto al número de procesadores que podrían utilizarse. Con ese número de procesadores la ejecución de la aplicación quizás sería la más rápida posible, pero su eficacidad podría ser mediocre si ese valor máximo excede en mucho al *grado promedio* de paralelismo (muchos procesadores pasarían inactivos durante gran parte de la ejecución del programa). En el Capítulo 5 veremos cómo obtener el número de procesadores idóneos para una aplicación dada, a partir del grado de paralelismo de ésta.

2.3 Fuentes de Paralelismo

Las máquinas paralelas tienen por finalidad explotar el paralelismo inherente en las aplicaciones paralelas. Todas las aplicaciones no tienen el mismo tipo ni la misma cantidad de paralelismo. El primer aspecto tiene que ver con una característica

cualitativa: la manera como el paralelismo puede ser explotado, denominado *fuente de paralelismo*. Básicamente, existen tres tipos de paralelismo [10, 13, 15, 17, 18, 40]: de control, de datos y de flujo. Usaremos el siguiente ejemplo para visualizar los diferentes tipos de paralelismo.

Ejemplo 2.2: Suponga el programa siguiente:

Para i = 0, n-1
$A[i]=a+b*B[i]^2+c*B[i]^3+ d*B[i]^4+ e*B[i]^5+f*B[i]^6+g*B[i]^7$

Una primera posible paralelizacion es notando que cada lazo de ejecución es independiente. Así, cada uno puede ser ejecutado independientemente (en este caso, sí se tienen *n* procesadores, todas las iteraciones podrán ser ejecutadas concurrentemente):

Hacer en Paralelo para i = 0, n-1
$A[i]=a+b*B[i]^2+c*B[i]^3+ d*B[i]^4+ e*B[i]^5+f*B[i]^6+g*B[i]^7$

Por supuesto, si el número de procesadores es menor que *n*, habrá que agrupar algunas de las iteraciones para que sean ejecutadas en un mismo procesador. Otra posible paralelizacion es definir varios polinomios y calcular cada uno en paralelo, por ejemplo:

Para i=0,n-1
Hacer en Paralelo
$X= a+b*B[i]^2+c*B[i]^3$
$Y= d*B[i]^4+ e*B[i]^5+f*B[i]^6$
$Z= g*B[i]^7$
$A[i]=X+Y+Z$

En este caso sólo se requieren tres procesadores, tal que en cada uno se calcule uno de los polinomios del programa (X, Y y Z). Esto es posible ya que cada cálculo es una tarea independiente. La ejecución de las tareas se deben sincronizar en cada iteración, ya que por cada iteración se deben realizar las tres tareas simultáneamente.

La última forma de paralelización es a través de un esquema de descomposición de funciones, de la siguiente manera: Se tiene una función con una pareja (x, y) como argumentos de entradas y (u, v) como argumentos de salidas, tal que u=x. Entonces, el cálculo del polinomio anterior puede ser hecho como:

A[i]=a+B[i](0+B[i](b+B[i] …(f+B[i](g+B[i]*0)…)

Tal que, $A[i] = f_9(f_8(f_7(f_6(f_5(f_4(f_3(f_2(f_1(B[i],0)))))))))$
Con $f_1(x,y) = (x,(g+xy))$, $f_2(x,y) = (x,(f+xy))$, …, $f_8(x,y) = (x,(0+xy))$, $f_9(x,y) = (x,(a+xy))$

Esta forma de descomposición permite la introducción de un procesamiento por encauzamiento, donde cada función corresponde a una etapa del procesamiento por

encauzamiento, y en cada etapa al terminar el cálculo del elemento $f_{j-1}(\ldots(f_1((B[i], 0)))$, se continua con el cálculo del elemento siguiente $f_{j-1}(\ldots(f_1((B[i+1], 0)))$. En este caso, el paralelismo es igual al número de funciones.

Muchas aplicaciones usan los tres tipos de paralelismo al mismo tiempo. A continuación describiremos en detalle cada uno de ellos.

2.3.1 Paralelismo de Control

Consiste en hacer cosas diferentes al mismo tiempo (es el caso del segundo enfoque del ejemplo 2.2). Este tipo de paralelismo surge de la constatación natural de que una aplicación paralela está compuesta de acciones que se pueden hacer al mismo tiempo. Las acciones (tareas o procesos) pueden ser ejecutadas de manera más o menos independiente sobre diferentes procesadores. El paradigma de programación *divide y vencerás* es típico de esta fuente de paralelismo.

En este enfoque se requiere una primera fase de descomposición del programa en tareas, para después definir cuáles se pueden ejecutar simultáneamente o se deben ejecutar secuencialmente. Los criterios importantes a considerar son el número de tareas, el grano de paralelismo, y las dependencias entre las diferentes tareas. Una dependencia surge cuando una acción (tarea) se debe terminar para que otra continúe. Existen dos clases de dependencias:

- *Dependencia de Control de Secuencia*: Es el secuenciamiento clásico de los algoritmos secuenciales.
- *Dependencia de Control de Comunicación*: Es cuando una tarea envía información a otra tarea, la cual no puede continuar hasta recibir dicha información.

Los dos casos de dependencias deben optimizar el siguiente criterio a la hora de asignar las tareas a los procesadores: minimizar los lapsos de tiempo que pasan los procesadores desocupados, o a la espera, debido a dichas dependencias. Bajo la noción de manejo de recursos, la explotación de paralelismo de control consiste en manejar las dependencias entre las tareas de una aplicación, para obtener una asignación de ellas, tan eficaz como sea posible.

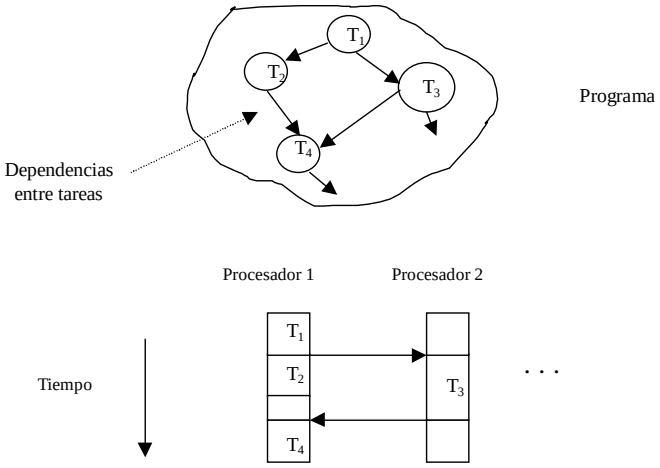


Figura 2.4. Dependencia de Tareas

En la figura 2.4 vemos un ejemplo de dependencias en la ejecución de las tareas. Cuando T_1 termina, el procesador 1 queda libre para ejecutar a T_2 y el procesador 2 puede ejecutar a T_3. T_4 no puede comenzar hasta que T_2 y T_3 terminen. Como T_2 es menos largo que T_3, el procesador 1 estará inactivo después que T_2 termine, hasta que T_3 finalice (en este caso no se han considerado los tiempos de comunicación entre los procesadores, lo cual aumentaría aun más los tiempos de espera).

De la figura anterior podemos concluir que los tiempos de ejecución de las tareas que se ejecutan concurrentemente deben ser más o menos semejantes. Si existen tareas que no pueden comenzar hasta que otras terminen, habrá que esperar a que la tarea que antecede que tome más tiempo termine, antes de poder comenzar la ejecución de la nueva tarea.

Supongamos ahora otro caso de un programa compuesto por tres tareas T_1, T_2 y T_3, tal que T_2 y T_3 pueden ejecutarse simultáneamente, cada tarea dura t_1 (son todas iguales a nivel de tiempo de ejecución), y el tiempo para iniciar una tarea es t_2 (aquí se incluye tiempos de asignación, de comunicación, etc.). El tiempo del programa secuencial seria $3t_1$ y del paralelo $2(t_1+ t_2)$. Si $t_2 > t_1/2$ la ejecución paralela es más lenta que la secuencial. Una manera de mejorarlo es disminuyendo t_2 o aumentando el grano de las tareas.

Las principales limitaciones que se encuentran en este paralelismo son las siguientes [5, 7, 11, 27]:

- *Dependencias Temporales*: ligadas al problema de secuenciamiento y comunicación entre las tareas.
- *Dependencias Espaciales*: ligadas al número limitado de recursos. Si no existen suficientes procesadores, un fenómeno de cuello de botella se produce, tal que es necesario ejecutar secuencialmente parte del trabajo.

- *Sobrecosto por la gestión de los granos*: cuando el tamaño del grano de paralelismo es muy pequeño, los tiempos de gestión (activación, suspensión, etc.) son cada vez más grandes con respecto al trabajo efectuado por el sistema para ejecutar las tareas.
- El *tamaño de las tareas* que se ejecutan concurrentemente debe ser *similar*. Un desequilibrio en el mismo puede conllevar a tiempos en los que sólo pocos procesadores están trabajando.
- En máquinas de memorias distribuidas o máquinas con jerarquía de memoria, hay que añadir el problema de *comunicación* típico de estas arquitecturas.

Las características básicas a considerar en este paralelismo son:

- Un programa paralelo consiste de una o más tareas, algunas de las cuales pueden ejecutarse concurrentemente mientras que otras dependen de la ejecución previa de otras tareas.
- Una tarea encapsula un programa secuencial y normalmente requiere comunicarse con su ambiente u otras tareas del mismo programa (compartiendo espacios de memoria o enviando mensajes).
- El tamaño de las tareas debe ser lo suficientemente grande, para que el tiempo para iniciar la ejecución de una tarea no sea importante.
- Se deben resolver los problemas de planificación de la ejecución de las tareas, de asignación de las tareas a los procesadores, etc., con la idea de reducir sus tiempos de ejecución.
- Las tareas pueden ser asignadas a los procesadores según diferentes criterios: distribuyendo la carga de trabajo, minimizando los costos de comunicación entre ellas, etc.

2.3.2 Paralelismo de Datos

La explotación del paralelismo de datos proviene de la constatación de que ciertas aplicaciones actúan sobre estructuras de datos regulares (vectores, matrices, etc.), repitiendo un mismo cálculo sobre cada elemento de la estructura. Así, normalmente, se habla de programas que manipulan estructuras de datos regulares. La idea es explotar la regularidad de los datos, realizando en paralelo un mismo cálculo sobre datos distintos, por ejemplo: “aumentarle el salario a todos los empleados con más de 5 años de servicio”. Este tipo de paralelismo es explotado en el primer caso del ejemplo 2.2.

En este tipo de paralelismo se asocian directamente los datos a los procesadores. Como los cálculos se realizan en paralelo sobre procesadores idénticos, es posible centralizar el control. Al ser los datos similares, la operación a repetir toma el mismo tiempo sobre todos los procesadores, y el controlador puede enviar de manera síncrona la operación a ejecutar a todos los procesadores. Normalmente, este es un tipo de paralelismo de grano fino, por lo cual se requieren plataformas del tipo SIMD.

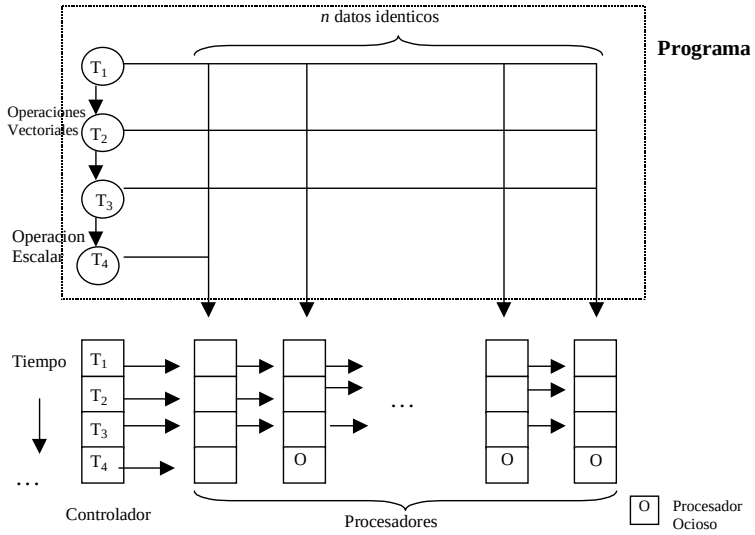


Figura 2.5. Paralelismo de Datos

En la figura 2.5 vemos un ejemplo de paralelismo de datos. Las tareas T_1, T_2 y T_3 se ejecutan secuencialmente como en un algoritmo secuencial clásico. El programador tiene la impresión de tener un programa secuencial donde las operaciones no se ejecutan sobre datos escalares (reales, entero, etc.), sino sobre arreglos de datos. En el caso de la ejecución de T_4, sólo un procesador es usado.

Las principales limitaciones que se encuentran en este paralelismo son las siguientes [13, 18, 20, 27, 30, 33]:

- *Manejo de Vectores:* si el tamaño de la máquina es más pequeño que el tamaño de los datos vectoriales, hay que descomponerlos, lo que agrega un tiempo extra de asignación y de gestión de tareas.
- *Manejo de Escalares:* las aplicaciones que manipulan datos vectoriales, usan frecuentemente datos escalares. Cuando se procesa el escalar, sólo un procesador trabaja y el resto permanece ocioso (caso de la tarea T_4 en la figura 2.5).
- *Operaciones de difusión y reducción:* estas operaciones sobre los datos vectoriales normalmente no son realizadas en paralelo, aunque realmente esto depende de la topología de la plataforma.

Algunos aspectos son importantes a considerar cuando se desea explotar este tipo de paralelismo. Dichos aspectos son los siguientes:

a) Distribución de los Datos

Los datos son normalmente numerosos, mucho mayor al número de procesadores en el sistema. Esto obliga a que los datos sean repartidos entre los diferentes procesadores disponibles. La regularidad de las estructuras de datos permite distribuirlos de manera regular. Por ejemplo, una matriz *n*m* sobre *p* procesadores podría ser distribuida en

forma tal que cada uno tenga *n/p* líneas. El procesador sería responsable de realizar todas las operaciones sobre esas líneas de la matriz. En el caso general, si suponemos *p* el número de procesadores y *n* el tamaño de los datos (*n* múltiplo de *p*), los esquemas más comunes de distribución o repartición sobre los procesadores son (cada procesador es denotado por Pi, $\forall$i =0, p-1, y tendrá asignado todos los datos dj, $\forall$j=0, n-1):

- Por bloques (procesador Pi tendrá asignado todos los datos dj, tal que $i*n/p \leq j < (i+1)n/p$),
- Cíclicos (procesador Pi tendrá asignado todos los datos dj, tal que i = j mod(p)),
- Bloques cíclicos (procesador Pi tendrá asignado todos los datos dj, tal que $i = \lfloor j/b \rfloor$ mod(p), donde *b* es el tamaño de los bloques y es divisor de *p*).

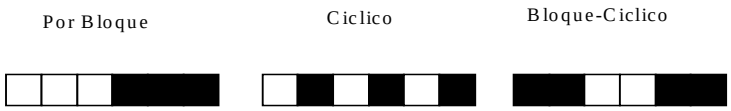


Figura 2.6. Esquemas de distribución de los datos sobre los procesadores

b) Ejecución Síncrona

El paralelismo de datos es clásicamente usado en máquinas SIMD con control centralizado, por lo que se puede ver como un flujo de control sobre múltiples datos. En este caso, la sincronización es automática al existir un sólo controlador. La importancia de la sincronización se ilustra en el siguiente ejemplo:

Hacer en Paralelo para i = 0, n-1
*A[i]=2*i*
Hacer en Paralelo para i = 0, n-1
B[i]=A[i]+A[n-1-i]

Supongamos que los vectores *A* y *B* son repartidos de la misma forma, por ejemplo, por bloques. B[0], que se ejecuta en P0, necesita A[0] y A[n-1], el primero calculado en P0 y el otro en el último procesador. *B* sólo podrá ser ejecutado hasta que los valores necesarios del vector *A* estén disponibles. Esto es garantizado por un cálculo síncrono de los valores de *A* seguido por el cálculo de *B*, fácilmente de implementar en una máquina SIMD síncrona. Esto no implica que sobre una máquina MIMD no se pueda ejecutar, pero hay que asegurarse de que los procesadores trabajen de manera coordinada; por ejemplo, se debe asegurar que los procesadores terminen de calcular los valores de *A* antes de iniciar los cálculos de *B*, por lo que sencillamente se debe sincronizar el conjunto de procesadores entre los dos lazos. El paralelismo de datos sobre una máquina MIMD conlleva a alternaciones entre ciclos de ejecuciones no síncronas entre todos los procesadores y la sincronización del conjunto de procesadores para garantizar la semántica del programa.

c) Dependencia de Datos

Hasta ahora en todos los ejemplos presentados se ha considerado que todos las elementos de un vector pueden ser ejecutados simultáneamente (llamados en la literatura como "lazos paralelos"). Pero existen situaciones menos favorables, donde la dependencia entre los elementos de un mismo vector prohibe ciertos cálculos paralelos, por ejemplo:

```
A[0]=B[0]=0
Para i=0, n-1
        B[i]=A[i]+B[i-1]
```

En este caso, los diferentes cálculos sobre *B* no se pueden hacer en paralelo, ya que existe una dependencia de datos entre sus elementos. En el capítulo 4 nos dedicaremos a estudiar los diferentes esquemas de paralelizacion cuando existe este problema. Aquí nos contentaremos con definir una condición suficiente de independencia, conocida como la *Condición de Bernstein*.

Condición de Bernstein: Supongamos un programa *S*, donde L(S) son lecturas y E(S) son escrituras dentro del código de *S*. Dos instrucciones pueden ser ejecutadas en un orden cualquiera, o en paralelo, si las tres condiciones siguientes se cumplen:

- *Ausencia de cualquier dependencia Verdadera* ($E(S_1) \cap L(S_2)=0$): esto significa que S_2 no utiliza ningún valor producido por S_1. Si no es el caso, no se puede introducir paralelismo, ya que al introducirlo se correría el riesgo de que S_2 calcule resultados errados debido a que leerá al azar un valor producido por S_1 (el valor que había antes de la operación $E(S_1)$, o con suerte, después que esa operación termine).
- *Ausencia de cualquier Antidependencia* ($L(S_1) \cap E(S_2)=0$): si no se respeta esta dependencia, la ejecución paralela puede dar resultados errados, ya que quizás S_2 asigne un nuevo valor a una variable antes que S_1 haya podido leer el viejo valor de esa variable (va en el sentido inverso de la anterior).
- *Ausencia de cualquier dependencia de Salida* ($E(S_1) \cap E(S_2)=0$): en este caso se exige que no existan datos modificados a la vez por S_1 y S_2. Si no se respeta esta condición, la ejecución paralela dará un resultado indeterminado tal, que no se pueda determinar cuál valor final será guardado en la variable común.

Ejemplo 2.3: Hallar las dependencias en el siguiente código:

```
A[0]=B[0]=0
Para i=1,n-1
        S1: A[i]=2*i
        S2: B[i]=A[i]+B[i-1]
```

Existe una *dependencia verdadera* entre S1 y S2, y entre las diferentes instancias de S2. Para verificarlo, consideremos S2(j) una instancia dada de S2 cuando *i* vale *j*, así:

$E(S2(j-1)) \cap L(S2(j)) \neq 0$
$E(S1(j)) \cap L(S2(j)) \neq 0$

Por el contrario, S1 es paralelo a nivel de sus instancias (sólo que el hecho de estar en un mismo lazo con S2 prohibe su paralelizacion). Siguiendo el mismo análisis, no existen *anti-dependencias* ni *dependencias de salida* entre S1 y S2. El código podría ser mejorado como:

A[0]=B[0]=0
Hacer en Paralelo para i = 1, n-1
    *S1: A[i]=2*i*
Para i=1,n-1
    S2: B[i]=A[i]+B[i-1]

d) Perfil de Paralelismo de Datos

Existen dos operaciones básicas en el paralelismo de datos, a través de las cuales se puede expresar todo el paralelismo encontrado en él: φ-notación y β-reducción. φ-notación es descrita por una función *f* de dimensión *n* y *n* vectores del mismo tamaño. φ-notación aplica en paralelo la función *f* sobre el conjunto de vectores. Por ejemplo, la operación r(φ(+v1,v2) suma los vectores v1 y v2 para producir el vector resultado *r*. Esto permite la aparición del paralelismo de datos. La operación β-reducción es una función binaria *f* sobre un único vector *r*, tal que se aplica la función *f* sobre los dos primeros elementos de *v*, y después, se aplica *f* sobre el resultado del cálculo precedente y el elemento siguiente del vector *v*. Por ejemplo, β(+,v) calcula la suma de todos los elementos del vector *v*. Con ciertos cambios sobre esta operación, para explotar la propiedad asociativa (paradigmas del árbol binario o divides y vencerás), es posible generar un esquema paralelo de ejecución.

φ y β son lo suficientemente generales como para expresar un gran número de operaciones sobre datos regulares, como vectores y matrices, al tratar de explotar el paralelismo de datos. Así, por ejemplo, la multiplicación de dos matrices puede ser fácilmente escrita según el siguiente pseudo-codigo:

Para i= 0, n-1
    Para j=0,n-1
        Mat[i, j] = β(+,φ(,A_linea[i],B_columna[j]))*

Por las características particulares de las máquinas MIMD, el paralelismo de datos sobre estas máquinas es diferente. Dos posibles formas para expresar dicho paralelismo en las máquinas MIMD son las siguientes:

Para i=0, n-1
    Si α*n/p ≤ i <α*n/p+n/p
        A[i]=…

*ind=α*n/p*
    Para i=ind, ind+n/p-1
        A[i]= …

donde α es el número identificador de cada procesador. En el lado de la derecha cada procesador determina dinámicamente qué cálculos realizar, mientras que en el otro lo hace estáticamente.

Como hemos visto, la explotación del paralelismo de datos aunque parezca fácil, implica una escritura bien reflexiva del código, para introducir el mínimo de dependencias posibles entre los datos y evitar las dependencias falsas. Los compiladores que paralelizan automáticamente son cada vez más numerosos, y extraen eficientemente este tipo de paralelismo. Un estudio detallado de las técnicas de paralelizacion automática será presentado en el capítulo 4.

2.3.3 Paralelismo del Flujo de Ejecución

Este tipo de paralelismo corresponde al principio de "trabajo en cadena". Está inspirado en la constatación de que muchas aplicaciones funcionan según el modo de trabajo en cadena: se dispone de un flujo de datos, generalmente similares, sobre los cuales se deben efectuar un conjunto de operaciones en cascada/cadena. Es decir, una secuencia de operaciones debe ser aplicada en cadena a una serie de datos similares. Las operaciones a realizar son asociadas a procesadores, los cuales están conectados en cadena de manera de que la entrada de un procesador (operación) sea la salida del procesador (operación) que le antecede. Así, los resultados de las operaciones realizadas en el tiempo T son pasados en el tiempo T+1 a los procesadores siguientes. Este tipo de paralelismo implica un procesamiento de tipo encauzamiento, explotando la regularidad de los datos (es el último caso del ejemplo 2.2). Este paralelismo es muy usado en el cálculo vectorial.

Las mismas operaciones φ y β, anteriormente mencionadas, pueden ser usadas para expresar este paralelismo, sólo que aquí estas operaciones no son realizadas en paralelo sino secuencialmente, y cada elemento es tratado uno después del otro. Así, si se quiere ejecutar en paralelo una adición vectorial, se aplica la operación de adición simultáneamente sobre los diferentes elementos del vector, pero en modo encauzamiento. Esto consiste en descomponer la adición en una cadena de trabajo.

La extracción de operaciones vectoriales en un programa puede ser hecha por el usuario o el compilador. Como los datos y las etapas deben ser independientes para que el paralelismo de flujo sea posible, los otros dos tipos de paralelismo pueden ser explotados. Si los datos se tienen en memoria, es preferible reemplazar el paralelismo de flujo por uno de datos. El paralelismo de control se impone cuando las duraciones de las etapas son muy diferentes entre ellas. El modo de procesamiento por encauzamiento se justifica solamente, si la llegada de los datos es secuencial y si los diferentes elementos de procesamiento son especializados cada uno en una operación particular.

El flujo de datos puede provenir de dos fuentes [29, 38, 39]:

- *Datos de tipo vectorial almacenados en memoria*: en este caso existe una dualidad muy fuerte con el paralelismo de datos. La diferencia reside en la asignación espacio-temporal de los datos con respecto a las acciones.
- *Datos de tipo escalar que provienen de dispositivos de entrada*: el flujo de datos es continuo durante el funcionamiento del sistema y puede ser considerado como una fuente infinita.

En los dos casos, la ganancia que se puede obtener es lineal con respecto al número de fases (procesadores). La figura 1.9 muestra un ejemplo de Paralelismo de Flujo.

Las principales limitaciones que se encuentran en este paralelismo son las siguientes:

- *Transitoriedad:* Al usarse el paralelismo de flujo, hay una fase transitoria cada vez que se realiza un conjunto de cálculos, esto se debe a que en la fase de inicio y de fin, no todos los procesadores trabajan, ya que no tienen elementos. Esta fase dura N ciclos (donde N es el número de etapas). Esta es una pérdida de paralelismo, la cual es inversamente proporcional al tamaño de los datos. Además, si el flujo presenta frecuentes discontinuidades, las fases transitorias de inicio y de fin pueden degradar los rendimientos de una manera importante.
- *Bifurcación:* en una máquina organizada en modo de procesamiento por encauzamiento, las bifurcaciones provocan rupturas del flujo de ejecución, que a su vez llevan implícito transitoriedades para recomenzar las operaciones. Por consiguiente, las bifurcaciones también degradan los rendimientos.

2.4 Comparación entre los Enfoques

Para explicar las ventajas y desventajas de cada una de las fuentes de paralelismo, supongamos el siguiente ejemplo:

```
Para j =1, n
        Si 100=1/(X[j]*Y[j]+Y[j]*Z[j])
                A[j] = A[j](A[j]+1)
        De lo contrario
                A[j] = A[j]-1
```

En la figura 2.7 se compara el paralelismo de control (en el eje y) contra el paralelismo de flujo (en el eje x).

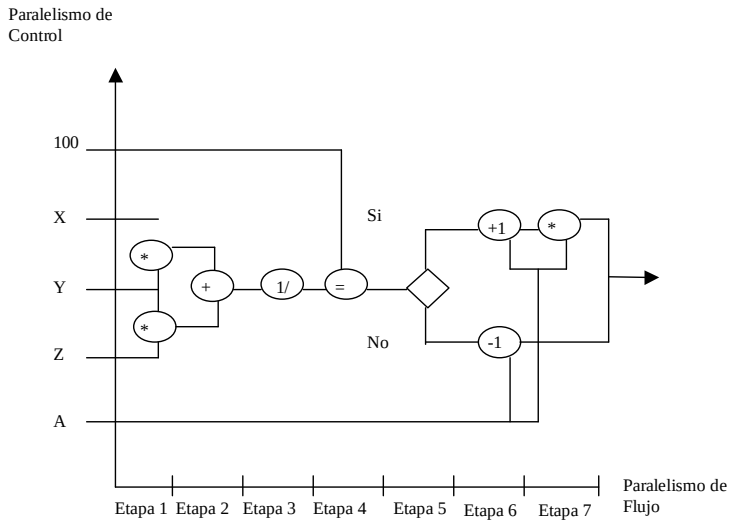


Figura 2.7. Comparación del paralelismo de control contra el paralelismo de flujo

En la siguiente figura, añadimos en el eje *z* el paralelismo de datos:

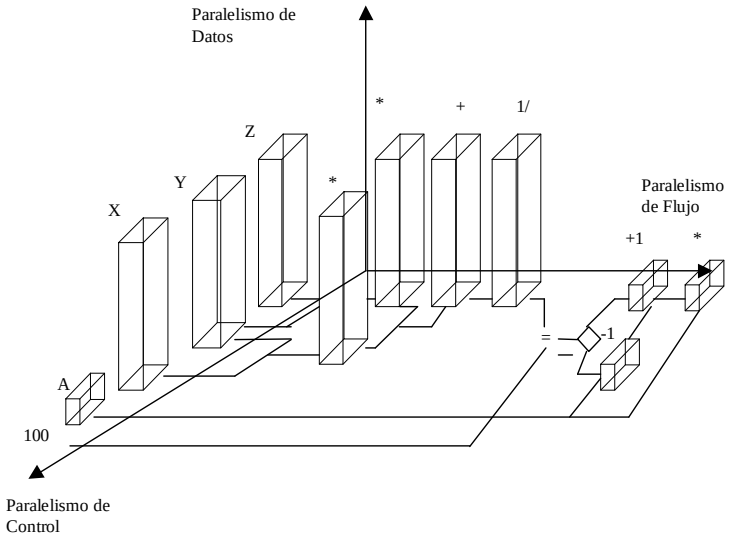


Figura 2.8. Comparación de las diferentes fuentes de paralelismo

Para este ejemplo, el grado promedio de paralelismo de control es de 1.142, mientras que el de flujo viene dado por el número de etapas (7). Finalmente, el de datos en promedio es *n* (número de datos). Una máquina ideal para ejecutar dicho programa sería un cubo con un número de procesadores por cada eje igual al máximo de paralelismo encontrado en cada fuente de paralelismo. Como se ve, aún en problemas con mucho paralelismo no es fácil construir una máquina que pueda explotarlo de manera óptima. La gran diferencia entre las fuentes de paralelismo viene dado por la "cantidad de paralelismo que cada uno genera":

- *Paralelismo de control:* está asociado al número de expresiones funcionales en el programa. Es decir, el paralelismo de control está más ligado a la presencia de tareas independientes en un programa. Así, su grado de paralelismo depende del número de tareas. Normalmente no pasa de más de una docena de tareas (excepto para el caso del paradigma "divides y vencerás"). Numerosas medidas de este paralelismo han determinado que su tamaño promedio es 3, es decir, sorprendentemente pequeño. Se debe tener mucho cuidado al considerar las características de la plataforma computacional para sacarle su máximo provecho. El paralelismo de control es bastante flexible (por ejemplo, se adapta fácilmente para procesar datos dinámicos, es decir, datos creados durante la ejecución del programa).
- *Paralelismo de datos:* está ligado al número de elementos en las estructuras de datos vectoriales. Normalmente, se usan aplicaciones compuestas por vectores con miles de elementos. El grado de paralelismo puede cambiar entre las diferentes partes de un programa. Por eso, una tendencia, es hacer que el número de procesadores a usar, sea igual al grado promedio de paralelismo de la aplicación, más que a su grado máximo. El paralelismo de datos ha sido clásicamente usado en aplicaciones donde los datos son conocidos estáticamente al momento de la compilación. Este paralelismo es fácil de automatizar, pero aun hay que hacer más estudios para el caso cuando se tienen datos dinámicos.
- *Paralelismo de flujo:* es ligado a la profundidad de la expresión funcional del programa, es decir, al número de funciones estáticas imbricadas en el programa, o a la existencia de secuencias de operaciones sobre un vector de datos. El grado de paralelismo depende del número de operaciones a aplicar de manera secuencial. En ciertas aplicaciones pueden haber docenas de operaciones, en otras, apenas algunas. El paralelismo de flujo ha sido usado fundamentalmente para acelerar los circuitos electrónicos y es la base del cálculo vectorial. Además, a nivel programado, normalmente ha sido modificado para explotar el paralelismo de datos o de control implícitos en él.

A pesar de que el paralelismo de control parece menos interesante, no se debe olvidar que es el más común y fácil de explotar. Además, muchas veces los diferentes tipos de paralelismo pueden ser usados simultáneamente, pudiéndose tener a varios grupos de procesadores explotando una fuente de paralelismo, mientras que otros trabajan bajo otro enfoque de paralelismo.

2.5 Aspectos de Diseño de Programas Paralelos

Existen dos estrategias para diseñar aplicaciones paralelas, ellas se diferencian entre sí según las consideraciones iniciales en las que se basan. Por otro lado, las características estructurales de los programas paralelos están basadas en ciertos paradigmas. En esta sección nos dedicaremos a presentar todos estos aspectos [1, 3, 4, 9, 11, 15, 16, 27, 30, 32, 33, 35, 40].

2.5.1 Paralelizando un Programa Secuencial

En esta primera estrategia, se parte de un programa secuencial, el cual se desea paralelizar. Existen varias formas para paralelizar aplicaciones secuenciales:

2.5.1.1 Paralelización Automática

Dependiendo de la máquina paralela, existen un conjunto de tareas que se deben realizar, con el fin de obtener un eficiente código paralelo para un programa secuencial. Algunos de los ejemplos de posibles tareas a realizar son los siguientes:

- Distribuir los datos a través del conjunto de procesadores,
- Establecer referencias a memoria cercanas (entorno a una misma región de memoria) para hacer un uso eficiente de la jerarquía de memoria (problema de localidad),
- Crear múltiples cadenas de control,
- Sincronizar el acceso a las localidades de memoria (en memorias compartidas).

Dichas tareas estarán ligadas a las características de la máquina paralela donde se ejecutara la aplicación. El objetivo de la paralelización automática (compiladores, etc.) consiste en eliminar al programador de la responsabilidad de paralelizar el código, en la medida de lo posible. Por ejemplo, el compilador recibiría una versión de un código secuencial y produciría un eficiente código paralelo sin adicional trabajo para el programador. ¿Qué tan realista es?, es una difícil respuesta, ya que los diseñadores de compiladores pueden construir ejemplos que sus compiladores pueden manejar, pero quizás no son representativos de la gran extensión de los problemas reales que requieren de compiladores paralelos.

Una posible opción ha sido la de formular extensiones a los lenguajes secuenciales existentes (por ejemplo, FORTRAN o C). El esfuerzo mayor en esta área ha sido en torno a FORTRAN y al manejo de arreglo de datos. HPF (High Performance Fortran) y Fortran 90 son ejemplos de ello, cuya idea ha sido facilitarle al programador la incorporación de nuevas primitivas simples, fáciles de aprender, para permitirle al compilador hacer el trabajo difícil (insertar locks o primitivas de pase de mensajes, replicar datos compartidos, etc.). Otro posible enfoque es desarrollar pre-procesadores que tomen el código secuencial y produzcan la correspondiente fuente en lenguaje paralelo (el cual puede entonces ser enviado al compilador). Este tipo de enfoque necesita una interacción sustancial con el usuario.

Figura 2.9. Paralelización Automática

2.5.1.2 Librerías Paralelas

En esta estrategia, la idea de base es que normalmente el tiempo de cómputo en las aplicaciones está concentrada en una sola parte del mismo. Si se trata de optimizar esas partes, por ejemplo, produciendo versiones paralelas eficientes de las mismas en forma de rutinas, se podrían desarrollar librerías paralelas de ellas. Así, a pesar de que no toda la aplicación sea paralelizable, la parte de mayor demanda computacional típicamente si lo es. Este enfoque ha sido usado por mucho tiempo en las máquinas vectoriales, para las cuales se han desarrollado rutinas óptimas. Normalmente, las librerías desarrolladas han estado ligadas a requerimientos que provienen de aplicaciones científicas.

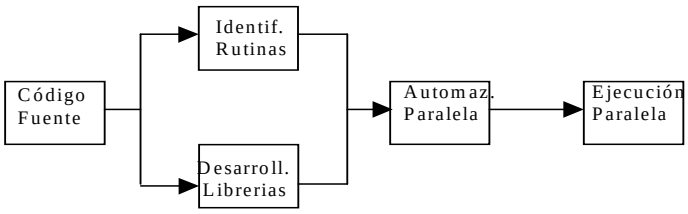


Figura 2.10. Librerías Paralelas

Este enfoque puede trabajar bien, ya que sólo hay que concentrarse en pequeñas partes. Además, es fácil asegurar su escalabilidad ante avances tecnológicos, en las plataformas computacionales (programación multihilos, etc.), ya que el esfuerzo está dedicado a incorporar estos avances en el código desarrollado para esas partes. Por otro lado, esto es completamente invisible al código secuencial (el programador no necesita saber que hay partes paralelas en su código). Incluso, se podría extender esta idea, para que el usuario decida qué versión usar según la plataforma computacional que disponga (por ejemplo, se dispondrían de varias versiones por rutina, una por cada plataforma computacional).

2.5.1.3 Recodificación de Partes

De acuerdo a la plataforma computacional disponible, rehacer parte del código es otra posibilidad. Detalles como no dependencia entre iteraciones de un lazo, si la aplicación es susceptible a explotar un grano fino, pueden ser determinantes a la hora de querer paralelizar un código secuencial. La idea es explotar esas características para maximizar la generación de tareas con posibilidad de ser ejecutadas concurrentemente. Un punto importante a resaltar es el hecho cuando se trabaja en una plataforma de procesamiento distribuida heterogénea (lo que implica diferentes lenguajes, estructuras de control, etc.). En este caso, se añade el problema de tener que portar las porciones de las aplicaciones, a cada una de las diferentes arquitecturas.

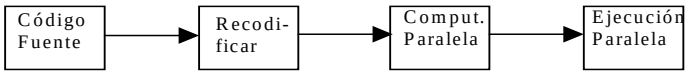


Figura 2.11. Recodificación del Código Secuencial

2.5.2 Desarrollando un Programa Paralelo Nuevo

Con esta estrategia, se desarrolla un programa paralelo, a partir de la definición del problema a resolver. A la pregunta: ¿Es más difícil desarrollar código paralelo que secuencial?, se podría responder, a veces sí y a veces no. Por ejemplo, los lenguajes paralelos ganan en el poder expresivo para manipular grandes arreglos de datos. Así, la posibilidad de escribir código sin preocuparse de definir los lazos de ejecución es hecho naturalmente en lenguajes paralelos. En general, las decisiones que se toman durante el diseño de una aplicación tienen un gran impacto en su implementación, y en el futuro desempeño de la misma. Entre las preguntas a responder se tienen:

- ¿Con que lenguajes de programación se cuentan?
- ¿Qué tan complejo y grande es el problema (las máquinas paralelas necesitan problemas grandes para ser eficiente)?
- ¿Se garantiza la ejecución eficaz de la aplicación?
- ¿Cuánto es el esfuerzo requerido para implementarla?
- ¿El diseño permite un crecimiento a futuro de la aplicación?

Para terminar esta parte, es importante recordar que un algoritmo eficiente para una máquina paralela, puede tener muy malos rendimientos en otras máquinas paralelas. Además, un código eficiente para un problema en una máquina paralela, puede ser muy diferente al código eficiente para el mismo problema en otra máquina. En la sección 2.6 presentaremos una metodología para desarrollar programas paralelos.

2.5.3 Paradigmas de Programación Paralela

A la hora de diseñar algoritmos paralelos, existen un conjunto de paradigmas de programación que pueden ser usados. Estos paradigmas permiten guiar el desarrollo eficiente del código paralelo, algunos de los cuales ya han sido usados con éxito en el desarrollo de código secuencial. Los paradigmas no son algoritmos, son estrategias de resolución de problemas. El propósito de los paradigmas son dos:

- Tener ideas que nos permitan resolver problemas diferentes por medios similares,
- Permitir desarrollar herramientas de diseño que ayuden a los programadores.

En el caso de los paradigmas de programación paralela, estos encapsulan información sobre los patrones de comunicación y el esquema de descomposición de tareas. Existen muchos paradigmas, a continuación presentaremos algunos de ellos:

- *Maestro/Esclavo:* consiste en dividir el trabajo en subtareas que serán realizadas por los diferentes esclavos bajo control del maestro. Esquemas MIMD con Memoria Compartida trabajan perfectamente bajo este enfoque.
- *Divides y Vencerás*: es un esquema en que la resolución del problema se descompone en varios subproblemas, los cuales a su vez se subdividen en subproblemas, hasta llegar a un nivel donde los subproblemas no pueden (o deben) ser más divididos. Cuando los subproblemas son resueltos, ellos regresan sus resultados a sus niveles superiores. Cada subproblema puede ser resuelto independientemente y sus resultados combinados dan el resultado final. En computación paralela, dichos subproblemas pueden ser resueltos al mismo tiempo, si se tienen suficientes procesadores disponibles. Algún pre-procesamiento y post-procesamiento son requeridos para partir el problema en subproblemas y para construir la solución del problema a partir de las soluciones de los subproblemas.
- *Arbol Binario:* es un caso particular del paradigma anterior, tal que los cálculos sobre *N* datos (n_1, …, n_N) son descompuestos en cálculos sobre (n_1, …, $n_{N/2}$) y ($n_{N/2+1}$, ..., n_N) datos. Por ejemplo, para sumar *N* enteros (n_1, …, n_N), donde *N* es múltiple de 2, y si colocamos los datos en las hojas del árbol, dicho cálculo podría hacerse de la siguiente forma: N/2 procesadores son empleados para calcular la suma de pares (n_1, n_2), …, (n_{N-1}, n_N) en un solo paso. Después, N/4 procesadores ejecutan la misma tarea para los pares de elementos de datos recién calculados, y así sucesivamente. El cálculo procede desde las hojas a la raíz del árbol. Existe un problema que consiste cuando un nodo padre se descompone en varios nodos hijos, los cuales duran diferentes tiempos de ejecución. Eso implica que los procesadores con los nodos hijos que duran menos tiempo, pueden llegar a pasar grandes períodos de tiempo ociosos.

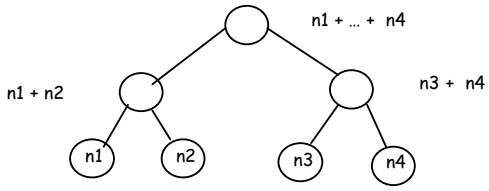


Figura 2.12 Paradigma Arbol Binario.

- *Otros paradigmas:* el paradigma basado en la *programación sistólica* para procesadores a memoria no compartida es otro ejemplo de paradigma. El paradigma sistólico consiste en descomponer el programa en subprogramas que son asignados a procesadores dedicados. Los datos fluyen a través de los procesadores, visitando subconjunto de procesadores para completar los cálculos. Tiene como propiedad

importante la localidad de la comunicación (los procesadores sólo requieren comunicarse con un subconjunto dado de procesadores). Existen otros paradigmas como el de *expandir un árbol* o *Calcular-Agrupar-Propagar*. Este último consiste en tres fases: a) fase de cálculo: que computa operaciones que van desde operaciones simples a programas enteros, las mismas ejecutadas en cada uno de los procesadores que componen el sistema, b) fase de agregación: que combina los datos locales, con los que eventualmente recibe, y c) fase de propagación: que envía los datos almacenados localmente a los diferentes procesadores que componen el sistema.

2.6 Metodología para Desarrollar Programas Paralelos

2.6.1 Generalidades

A la hora de desarrollar aplicaciones paralelas, los siguientes aspectos se deben considerar:

- Definir el enfoque/paradigma apropiado para cada arquitectura.
- Separar los tiempos de comunicación, de Entradas/Salidas y de cálculo en la aplicación, si es posible (esto nos permitirá identificar qué partes de la aplicación se deben optimizar).
- Usar los modelos de rendimiento que incluyan los aspectos más relevantes de la aplicación.
- Realizar pruebas en una variedad de situaciones y configuraciones de máquinas paralelas. Se debe estar seguro que el tamaño de la máquina y las pruebas (benchmark) correspondan entre ellos.
- Definir las características del paralelismo: si es escondido o es explícito, cual es la fuente de paralelismo, etc.

En cuanto al paralelismo escondido o implícito, nos extenderemos a continuación en su explicación. El paralelismo escondido está relacionado a los tipos de datos vectoriales y arreglos. Por ejemplo, el programa

```
integer A[i,j], B[i,j], C[i,j]
repeat i=0; i++; i<n
        repeat j=0; j++; j<n
                C[i,j]=A[i,j]+B[i,j]
```

podría ser implementado en un lenguaje paralelo como:

```
declare array A;B;C
C=A+B
```

El paralelismo implícito refleja el paralelismo que provee la máquina. En este caso, el programador no necesita preocuparse de los limites del arreglo o de pasear por el arreglo cíclicamente o sincrónicamente. Pero hay casos donde es mejor que el programador explícitamente controle el conjunto de operaciones paralelas, ejemplo:

Ejemplo 2.4: Hacer un programa paralelo que se ejecutará en una máquina tipo malla MIMD, el cual calcule la siguiente expresión:

$$f(x,y,z) = \sqrt{(x+z)^3 - (x+z)^2} \Big/ yz$$

Un posible código paralelo seria:

Hacer en <u>Paralelo</u>
r1=x+y,
r2=x+z,
r3=yz
Hacer en <u>Paralelo</u>
*r4=r1*r1*r1,*
*r5=r2*r2*
r6=r4-r5
r7=sqrt(r6)
r8=r7/r3

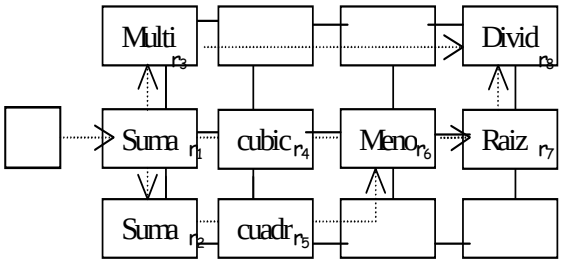


Figura 2.13. Flujo de Ejecución del programa que calcula $f(x,y,z) = \sqrt{(x+z)^3 - (x+z)^2} \Big/ yz$.

Suponiendo que cada procesador en el sistema tiene un número que lo identifica y que un compilador sólo asigna recursos siguiendo el orden numérico de los procesadores, tendremos la siguiente asignación:

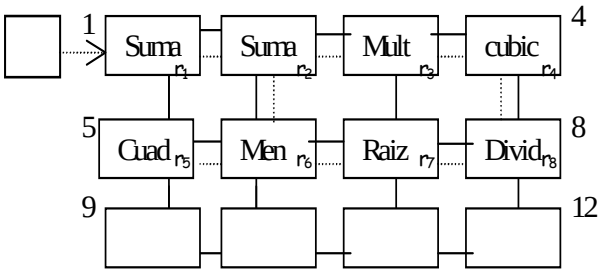


Figura 2.14. Asignación para el programa que calcula $f(x,y,z) = \sqrt{(x+z)^3 - (x+z)^2} \Big/ yz$

Esta asignación está muy lejos de la ideal, ya que los procesadores deben ejecutar operaciones adicionales de envío de datos. Una manera de evitar esto es que cada programador explícitamente defina cual proceso será asignado a cada procesador. Esto, por supuesto, no es posible en muchas aplicaciones, por lo que es más fácil que el usuario defina los enlaces de comunicación que requiere. Esto daría:

en serie: x -> E1, y -> E1, z -> E1
Hacer en Paralelo
E1 -> T2
E1 -> T3
Hacer en Paralelo
(x+y) -> T1
(x+z) -> T5
(yaz) -> T4
Hacer en Paralelo
T1 -> [cubo] -> T7
T5 -> [cuadrado] -> T6
T4 ->T4
Hacer en Paralelo
T4 ->T4
T7 -> T6 -> [-] -> T8
T8 -> [raiz] -> T9
Fi -> T4 -> T9->[/]

De esta manera, el usuario explícitamente indica el tipo de procesamiento paralelo, así como las comunicaciones, que se requieren. Esto permite que el computador pueda hacer la siguiente asignación, la cual es más eficiente:

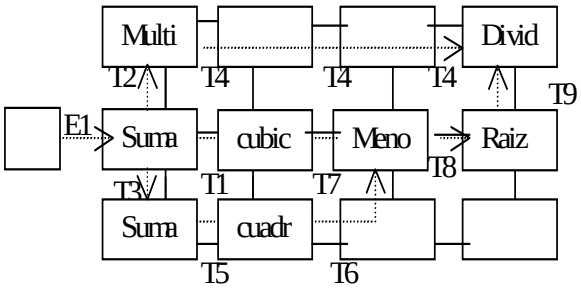


Figura 2.15. Asignación Optima para el programa que calcula $f(x,y,z) = \sqrt{(x+z)^3 - (x+z)^2} \Big/ yz$ en una Máquina MIMD tipo Malla.

2.6.2 Metodología

El diseño de programas paralelos no es fácil y requiere de una gran creatividad, así que el conjunto de pasos que daremos a continuación sólo debe ser entendido como una guía para maximizar la definición de eficientes programas paralelos [1, 3, 4, 5, 9, 10, 11, 12, 14, 15, 16, 27, 30, 32, 33, 35, 36, 37, 40]. El algoritmo paralelo final puede ser muy diferente a su versión secuencial, si es que existe.

Los pasos que se presentarán a continuación, se repiten en ciclos de diseño, a través de una metodología tipo espiral que va retrasando decisiones de hardware, considerando inicialmente aspectos más vinculados al software (las primeras decisiones de diseño no consideran la máquina donde se ejecutará la aplicación). Existen cuatro pasos que se repiten en esta metodología tipo espiral: descomposición, comunicación, agrupamiento y asignación. Los dos primeros pasos estudian el problema de escalabilidad y concurrencia, mientras que los dos últimos estudian el problema de localidad y mejoras en el rendimiento computacional. Los cuatros pasos, mezclados con la metodología espiral, hacen que se repitan estos pasos las veces que sean necesarios, por lo que se va haciendo un diseño en forma incremental y, redefiniendo las decisiones tomadas previamente.

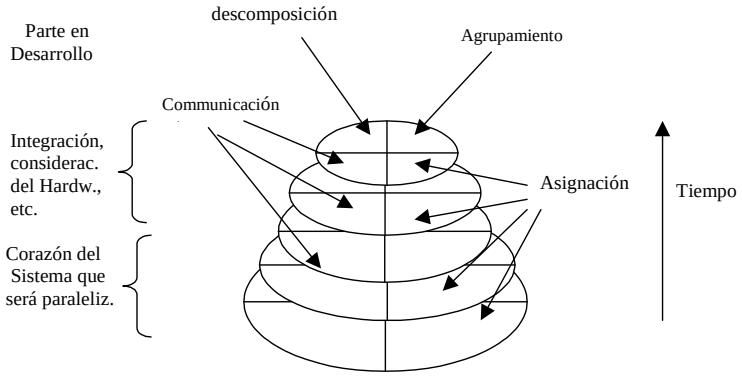


Figura 2.16. Metodología de Desarrollo de Aplicaciones Paralelas.

La flexibilidad de esta metodología radica en que se permite que las tareas puedan ser creadas o eliminadas dinámicamente, para diferentes tipos de plataformas computacionales. Las cuatros fases, a pesar de que parecieran secuenciales, son realizadas permanentemente, según la metodología espiral, con diferentes niveles de agregación y sobre diferentes componentes, por lo que se va haciendo en forma incremental. En las primeras fases de la metodología espiral, los pasos de descomposición y comunicación son los más relevantes, mientras que en las últimas fases de dicha metodología los pasos más relevantes son agrupamiento y asignación.

Un aspecto genérico a considerar tiene que ver con el diseño lógico (ver figura 2.17). El diseño lógico captura el paralelismo máximo que existe en un algoritmo dado. Su mapeo a una plataforma especifica puede reducir ese paralelismo máximo (además, puede ser muy complicado).

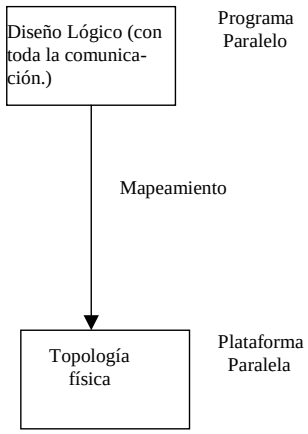


Figura 2.17. Pasos claves del proceso de diseño.

2.6.2.1 Mecanismo Espiral

El mecanismo espiral permite realizar un diseño óptimo de programas paralelos basado en dos premisas:

- Explotar el conocimiento que se va adquiriendo durante el desarrollo.
- Diseñar inicialmente las partes claves de la aplicación, y después, poco a poco, añadirle las otras partes.

Aprovechar la experiencia y conocimiento ganado sobre el hardware y software disponible, durante el desarrollo de la aplicación, permite un diseño final más eficiente. Se parte del hecho de que al minimizar el número de restricciones posibles (sobre las características de la máquina, con respecto al software disponible, etc.), se pueden generar ideas iniciales de diseño, que pueden tener un impacto fundamental en las características finales del código a generar. Al final, se deben considerar los aspectos específicos de la arquitectura y software disponible para acoplar el diseño a las especifidades de la plataforma donde se ejecutará. Esta última etapa aprovechará todo el conocimiento y experiencias obtenidas en los ciclos previos.

Por otro lado, el diseño incremental permite ir mejorando el código, además de considerar sólo al inicio las partes más importantes del mismo. La idea es que de los módulos claves que se tengan que desarrollar, se pueda tener un prototipo temprano, en el ciclo de diseño. Una vez que el corazón de la aplicación esté correcto, otras piezas de código pueden ser poco a poco incorporadas. Esto permite, por ejemplo, que la fase de desarrollo de interfaces a otros sistemas y usuarios (algo común con las aplicaciones secuenciales), puedan ser añadidas fácilmente al final, una vez que la parte clave de la aplicación paralela esté optimizada.

2.6.2.2 Descomposición

La idea en esta fase es vislumbrar las oportunidades de paralelismo, al diseñar el mayor número de pequeñas tareas posibles, las cuales determinarán el máximo paralelismo que se puede encontrar en dicha aplicación. Esta fase consiste en dividir o descomponer un programa en componentes que pueden ser ejecutados concurrentemente. Esto no implica necesariamente descomponer el programa en un número igual al número de procesadores que se tenga. El principal objetivo en esta fase es asegurar un eficiente uso de los recursos por parte de la aplicación y esconder la latencia. Un buen mecanismo de partición, debe considerar tanto a los datos como a los cálculos.

Un *eficiente uso de los recursos* se refiere a cómo el rendimiento de una aplicación dada varía según el número de procesadores disponibles. Por ejemplo, en ciertos casos, un programa secuencial podría ejecutarse más rápido que uno paralelo, cuando el número

de procesadores es pequeño (se puede atribuir a los costos de comunicación, sincronización, etc.). El tiempo de ejecución del programa paralelo puede ser mejorado añadiendo procesadores, hasta un límite dado, donde no hay más mejoras, ya que no hay suficiente trabajo para tener ocupados a todos los procesadores por la influencia de la comunicación.

Por otro lado, normalmente es preferible tener mucho más tareas que procesadores, tal que se pueda esconder la latencia. *La latencia* es el tiempo de comunicación entre los diferentes componentes de la arquitectura (procesadores, memoria, etc.), derivado de la ejecución de la aplicación (bloqueos en las tareas, acceso concurrente a la memoria, etc.). Al usar la multiprogramación, para mantener ocupados a los procesadores mientras otras tareas esperan, estaremos escondiendo la latencia.

Los esquemas de partición están muy atados a las fuentes de paralelismo. Básicamente, tres esquemas de partición pueden ser usados:

- *Descomposición por Dominio*: en este caso, el diseñador se concentra en partir los datos y en como asociar los cálculos a esos datos. La idea central es dividir dichas estructuras de datos en componentes que pueden ser manipulados independientemente (concurrentemente). Si es posible, los datos se dividen de igual tamaño a condición de que los procesadores sean de igual poder. Además, los cálculos a realizar también se descomponen típicamente, para asociar los cálculos con los datos sobre los cuales se ejecutan. Diferentes particiones son posibles, según la estructura de datos que se tenga y los cálculos que se requieren realizar. Esto implica que tanto las estructuras de datos como los cálculos a realizar deben ser estudiados simultáneamente. Se pueden usar en aplicaciones con una gran cantidad de datos regulares. Ejemplos de estos casos son las matrices. La figura 2.18 muestra dos formas clásicas de partir una matriz, por columnas o una descomposición por bloques.

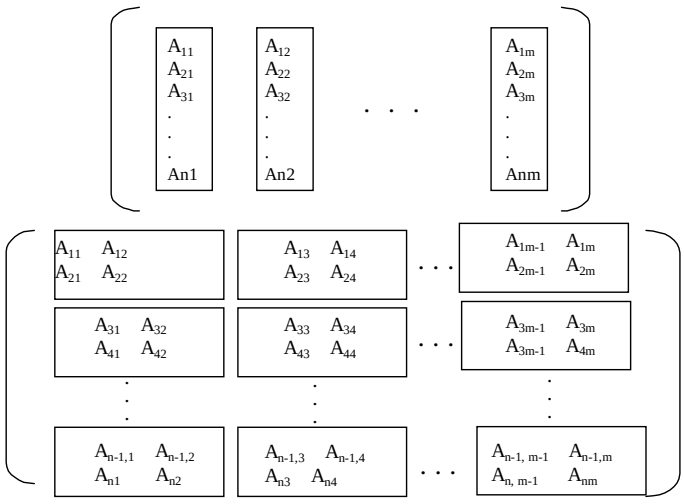


Figura 2.18. Posibles particiones de una matriz.

Supongamos la descomposición de *B* por columnas en un problema de multiplicación de matrices (AxB=C). Vemos que para calcular una columna de la matriz resultado *C*, los cálculos entre las correspondientes columnas de *B* por cada fila de *A* pueden ser ejecutados completamente independientes, por consiguiente, en paralelo (ver figura 2.18).

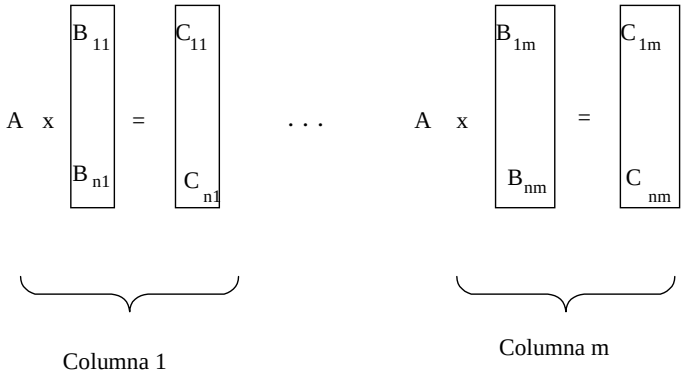


Figura 2.19. Multiplicación de matrices según una partición por Columnas.

La escalabilidad de esa partición está gobernada por el número de elementos de la partición (en nuestro ejemplo, por el número de columnas de B). Esto implica que para mantener las mejoras en rendimiento, es necesario que el número de las particiones se corresponda con el número de procesadores (si el número de componentes de la partición es menor, con más procesadores no se obtienen mejoras). Hay una implícita suposición en este tipo de descomposición, y tiene que ver con el tamaño de los datos y la regularidad en el patrón de comunicación; una partición regular de los datos supone una división regular del trabajo, por consiguiente, no es necesario balancear la carga. Este es un punto de inicio para el diseño de programas, pero muchas veces el patrón de comunicación envuelto para resolver el problema tiene poca relación con la descomposición de los datos, y puede generar desequilibrios de cargas de trabajo. Por ejemplo, en dinámica de fluidos, con una estructura de datos regular, algunos puntos en el campo de fluidos pueden tener una única propiedad, los cuales pueden necesitar de cálculos globales.

- *Descomposición Funcional:* aquí, el trabajo se concentra en descomponer los cálculos a realizar. La descomposición funcional es una herramienta fundamental de diseño estructurado de programas, ya que esta descomposición del cálculo lleva consigo, en muchos casos, la descomposición del código también. La estrategia consiste en dividir las funciones de un programa y operarlas concurrentemente. Por ejemplo, para una función f(x) se quiere calcular su integral en el intervalo [a, b]. Una descomposición funcional consiste en dividir el intervalo en *n* trozos de área *w*. Cada área podrá ser calculada independientemente y concurrentemente, y la área total será simplemente la suma de cada una. En este ejemplo también se suponen cálculos semejantes, por lo que no se requiere balancear la carga y la escalabilidad depende

del número de trozos. Este tipo de partición es complementaria a la anterior. Si se pueden dividir los cálculos, entonces se pasa a estudiar los datos, si los datos son descompuestos, entonces la partición es completa.

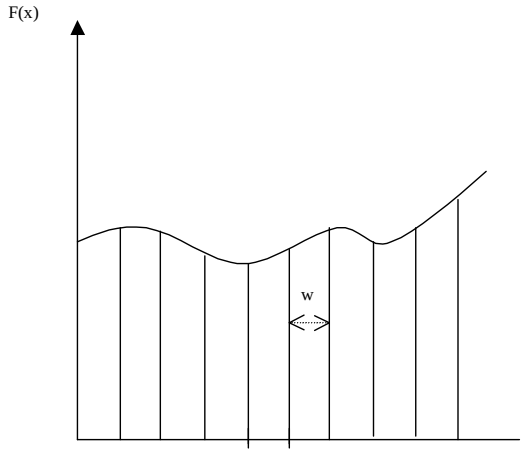


Figura 2.20. Descomposición Funcional para el Problema del Cálculo del Integral

En general, la descomposición funcional es más compleja que el caso presentado en el ejemplo anterior, donde los componentes de la descomposición no son similares, y por consiguiente, no auto-balanceados. Por ejemplo, consideremos la estructura típica de un compilador de un lenguaje de alto nivel (ver figura 2.21).

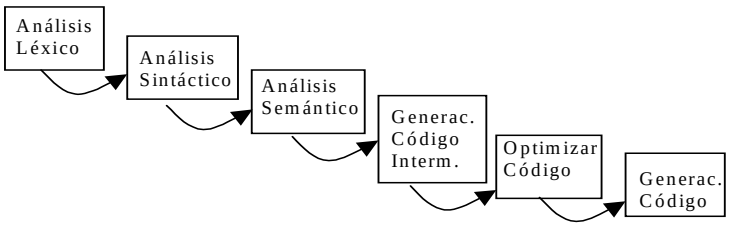


Figura 2.21. Estructura Programada de un compilador de un lenguaje de alto nivel

Una descomposición funcional podría primero focalizarse en la descomposición completa del compilador, y después revisar a cada uno de sus componentes. La descomposición funcional abarca el procesamiento por encauzamiento. El uso de encauzamiento es limitado por la velocidad del elemento más lento, por lo que se debe decidir qué aspectos del procesamiento por encauzamiento deben ser mejorados (descompuestos) para mejorar los tiempos. Otro ejemplo, es en el mismo cálculo del integral, para $\int_{b}^{a} f(x)$, cuando sucede lo siguiente:

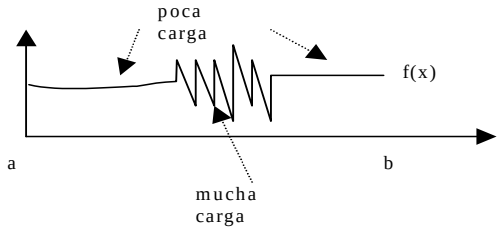


Figura 2.22. Cálculo del Integral con Desequilibrio de la Carga de Trabajo.

- *Descomposición de problemas irregulares:* en este caso, la estructura del programa evoluciona dinámicamente durante su ejecución y no puede ser determinada a priori. Ejemplos son los cálculos en dinámica de fluidos, etc. La idea es definir un modelo que cambie dinámicamente cuando el programa se ejecuta (por ejemplo, un árbol), de esta manera dicho modelo puede ser usado para partir dinámicamente el programa. Esto conlleva a que cuando el tamaño de las particiones varía, algún método de equilibrio de la carga es necesario, El esquema para usar en la división, depende de la estructura del modelo, del costo de las comunicaciones y del esquema de equilibro de la carga usado. Por ejemplo, la figura 2.23 muestra dos estrategias para descomponer árboles, en una se escoge un nivel en el árbol, donde el número de hojas es grande, y se descompone en subárboles, en el segundo se divide el árbol regularmente.

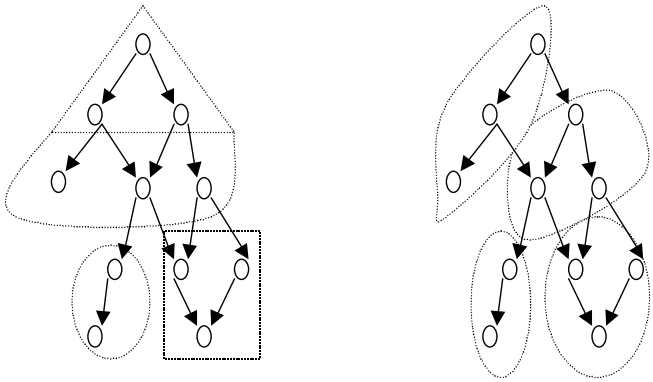


Figura 2.23. Posibles particiones de un Arbol

Detalles como que la mejor descomposición en una fase es de una forma y en la otra es diferente, son factibles de considerar, lo que puede conllevar a la necesidad de adaptar los datos, para poder ser usados en ambas fases (amoldar las estructuras de datos de cada fase, lo que implica añadir una fase explícita de reestructuración de datos, etc.).

2.6.2.3 Comunicación

Las tareas que se van definiendo, son diseñadas para que en su mayoría puedan ser ejecutadas concurrentemente, pero muchas veces eso no es posible, ya que no pueden ejecutarse independientemente. Los cálculos a realizar en una tarea, pueden requerir datos de otras tareas, lo que implica que se deben transferir esos datos entre las tareas. Definir la estructura comunicacional de la aplicación es clave en el futuro desempeño de una aplicación. El diseño de la estructura comunicacional se puede descomponer en dos fases: en la primera se definen los enlaces entre las tareas. Después, se especifican los mensajes a intercambiarse entre ellos.

En la descomposición por dominio, la comunicación puede ser difícil de determinar (hay que identificar las operaciones que requieren datos de diferentes fuentes). En contraste, en la descomposición funcional, es fácil de determinar, ya que las interrelaciones entre las tareas vienen dadas por las funciones que realiza cada una, y de las cuales quizás la otra depende. A continuación pasamos a categorizar los tipos de comunicación (para más detalles, ver el Capítulo 1):

- *Comunicación local y global*: en la comunicación local una tarea se comunica solamente con sus vecinos. En la comunicación global cada tarea se comunica con todas, o casi todas. Así, en una comunicación global, muchas tareas pueden participar, lo que puede conllevar a muchas comunicaciones.
- *Comunicación estructurada y no estructurada*: es un tipo de comunicación donde el patrón de comunicación de una tarea con sus vecinos forma una estructura regular, por ejemplo, del tipo árbol o malla. Las no estructuradas pueden generar grafos de comunicación arbitrarios.
- *Comunicación estática o dinámica*: la comunicación estática establece patrones de comunicación que no cambian en el tiempo, mientras que la dinámica puede ser solamente determinada durante el tiempo de ejecución de la aplicación.
- *Comunicación síncrona o asíncrona*: en el caso de la síncrona implica que las tareas se ejecutan coordinadamente (es decir, se deben sincronizar entre ellas), en contraste, la asíncrona significa que cada tarea puede ejecutarse al obtener los datos que requiere, sin tener que esperar por otras tareas.

2.6.2.4 Agrupamiento

El algoritmo hasta ahora es una abstracción, en el sentido de que aún no están especificados los detalles para asegurar su eficiente ejecución, sobre un computador particular. De hecho, puede ser bastante ineficiente, si se crean muchas tareas pequeñas y la arquitectura que se posee no es apta para este tipo de diseño de programas. Las tareas y estructuras comunicacionales definidas previamente, deben ser evaluadas según los requerimientos de rendimiento y costos de implementación. Así, en esta fase se revisan las decisiones de descomposición y comunicación, para obtener un algoritmo que se ejecuta eficientemente sobre un computador paralelo dado. En particular, se determina si se deben agrupar tareas identificadas en la fase de descomposición o si se deben replicar

datos y/o cálculos. La reducción de tareas puede llevar consigo el objetivo de que exista una tarea por procesador, dependiendo del tipo de plataforma computacional que se tenga. Los procesos de agrupamiento y replicación deben considerar:

- *Aumentar la cantidad de cálculos y reducir los costos comunicacionales*: al llegar a esta fase, normalmente se tiene un número grande de tareas (ya que se ha tratado de maximizar el paralelismo que se puede extraer), pero un gran número de tareas, no implica un eficiente paralelismo, en particular, por los costos comunicacionales, por lo que se deben reducir estos costos. Estos costos pueden ser reducidos de dos formas: enviando menos datos o menos mensajes. Esto porque el costo comunicacional depende tanto de la cantidad de datos a transferir como del costo inicial por establecer una comunicación. Ambos costos pueden ser reducidos por agrupamiento. La replicación también puede reducir los requerimientos comunicacionales, introduciendo cálculos redundantes se pueden evitar costos comunicacionales.

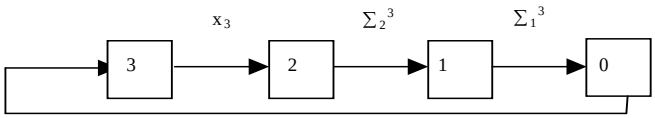


Figura 2.24. Costos Comunicacionales

Suponga el problema de sumar *n* números, tal que la distribución de la suma es distribuida entre n tareas Ti, tal que 0≤i≤n-1. Las tareas están conectadas en un anillo (ver figura 2.24, para n=4). Cada tarea espera la suma parcial desde su vecino izquierdo, añade esta a su resultado local, y lo envía a su vecino a la derecha. Finalmente, la tarea T0 recibe la suma parcial y lo añade a su valor local para obtener la suma completa. Este algoritmo distribuye las comunicaciones y adiciones, y permite concurrente ejecutar múltiples sumas. El arreglo de tareas es usado como un encauce, por el cual fluyen sumas parciales, así una suma simple toma *n* pasos. Ahora, supongamos que los resultados de la suma de las *n* tareas se deben replicar en cada tarea, una forma de hacerlo es seguir el mismo procedimiento y al final difundir los resultados a las *n* tareas (se puede usar la misma estructura de comunicación), lo que implica que se deben realizar 2(n-1) pasos y comunicaciones. Una variante es suponer que todas las tareas conectadas en el anillo ejecutan el mismo algoritmo, tal que *n* sumas parciales se hacen simultáneamente. Después de n-1 pasos/comunicaciones la suma completa es replicada en cada tarea, lo que evita la fase de difusión de resultados, pero con $(n-1)^2$ cálculos redundantes (ver figura 2.25, siempre para n=4).

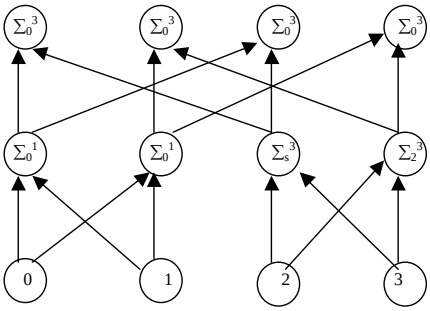


Figura 2.25. Cálculos Redundantes

Una posible extensión para el ejemplo anterior con *n* números y *m* procesadores (m<<n), tal que m=3 y n=300, es que cada procesador ejecute una parte de los números (ver figura 2.26).

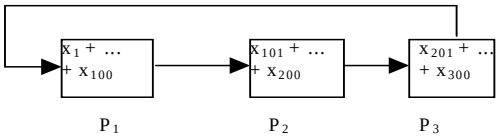


Figura 2.26. Distribución del trabajo

Si el tiempo de comunicar un valor al vecino es igual a 10 veces el tiempo de una adición, entonces para m=3 y n=300, una nueva distribución no tiene sentido, por consiguiente, pasa a ser más interesante que cada procesador haga el cálculo completo para todos los elementos.

La agrupación es también importante, cuando se analizan los requerimientos comunicacionales y se determina que ciertas tareas no pueden ejecutarse concurrentemente, por sus dependencias. Por ejemplo, en el caso de un árbol de ejecución sólo las tareas que están en el mismo nivel pueden ejecutarse concurrentemente, así, tareas en diferentes niveles se agrupan sin reducir las posibilidades de ejecución concurrente entre las tareas.

- *Aumentar la flexibilidad con respecto a la escalabilidad y proceso de decisión de las asignaciones*: un buen algoritmo debe ser diseñado para ser inmune a cambios en el número de procesadores. Detalles como solapar la ejecución de una tarea, con la comunicación de otra tarea en un mismo procesador, puede ser crítico a la hora de determinar el agrupamiento. La flexibilidad no necesariamente requiere que existan más tareas que procesadores, pero sí que la granularidad pueda ser controlada al momento de la compilación o en tiempo de ejecución. Esto evita la innecesaria

incorporación de límites en el número de tareas a crear, y permite que se explote más eficientemente la plataforma computacional.
- *Reducir costos de diseño de software:* a este nivel se consideran los costos de desarrollo. Si la aglomeración lleva consigo la posibilidad de reutilizar el código secuencial eficiente dentro del programa paralelo, bienvenida sea. Lo importante es facilitar el desarrollo de la aplicación, evitando el código extensivo (tamaño) de las diferentes partes y facilitando su integración.

2.6.2.5 Asignación

En este paso se especifica donde será ejecutada cada tarea. Las tareas son asignadas a los procesadores de manera de maximizar la utilización de los procesadores y minimizar los costos comunicacionales. Este problema no existe en sistemas uniprocesadores o computadores a memoria compartida y/o distribuida con automática planificación de tareas. En el último caso, el sistema operativo es el que realiza la distribución de las tareas. En general, este es un problema difícil y su objetivo es minimizar el tiempo total de ejecución de la aplicación a través de dos ideas:

a) Colocar tareas concurrentes en diferentes procesadores.
b) Colocar tareas que se comunican frecuentemente en el mismo procesador.

En general, este problema es NP-completo, sin embargo muchas técnicas heurísticas se han desarrollado con resultados eficientes. La asignación puede ser estática o dinámica, usando algoritmos de equilibro de la carga de trabajo u otros criterios. Los mecanismos de asignación dinámicos son más interesantes, cuando los problemas no tienen una estructura comunicacional regular (particularmente, cuando la cantidad de cálculo o comunicación cambia dinámicamente durante la ejecución de un programa). El sobrecosto que se introduce, debe ser controlado para maximizar los beneficios y evitar que el costo para determinar la optima asignación sea grande (por la periodicidad con que debe ser ejecutado). En el capítulo 3 estudiaremos en detalles el problema de asignación de tareas.

Capítulo 3.
Ciertos Problemas en Computación Paralela/Distribuida

En este capítulo presentaremos algunos de los aspectos a considerar, cuando se trabaja en plataformas distribuidas y paralelas, a nivel del problema de gestión de los diferentes recursos que componen a estos sistemas (procesadores, archivos, memorias, unidades de entrada/salida, etc.). Normalmente, dichos problemas de gestión, se refieren al problema de asignación de los recursos a los diferentes usuarios, a la planificación del uso de los recursos, a los esquemas para hacer tolerante a fallas a los sistemas, entre otros. La finalidad es optimizar el uso de los recursos y reducir los tiempos de ejecución de las aplicaciones.

3.1 Problema de Planificación en los Sistemas Distribuidos/Paralelos

El problema de planificación surge cuando hay que escoger el orden en el cual un número dado de actividades deben ser realizadas [14, 30]. Este problema ha sido descrito en diferentes formas en diversos campos, lo cual abarca desde trabajos que deben ser procesados en una línea de producción, pasando por clientes de un banco que esperan para ser atendidos. Claramente, esto puede ser extendido a tareas de programas que deben ser ejecutados en plataformas distribuidas/paralelas. Quizás, el problema que más ha influenciado esta área, es el clásico problema de secuencias de trabajos en una planta de producción. Los procesos de manufacturación envuelven varios tipos de operaciones para transformar el material que se va recibiendo en un nuevo producto. El problema es determinar la secuencia ideal de esas operaciones de acuerdo a cierto criterio, por ejemplo, económico. Muchas técnicas han sido usadas para resolver este problema, desde la enumeración completa de las posibles soluciones (búsqueda exhaustiva), pasando por la programación entera y técnicas heurísticas.

En general, el problema de planificación asume un conjunto de recursos y un conjunto de clientes que requieren servicios de esos recursos. Basado en la naturaleza y restricciones de los clientes y recursos, el problema consiste en encontrar una forma eficiente para usar dichos recursos, con el objeto de optimizar ciertas medidas de desempeño, normalmente basadas en la longitud total de la planificación. Las tareas de un programa, los clientes de un banco, y los trabajos en una fabrica, son ejemplos de clientes; los procesadores de un sistema computarizado, las máquinas en una fabrica, y los cajeros de un banco, son ejemplos de recursos (ver figura 3.1). El aspecto crítico de un esquema de planificación es la política usada para asignar los recursos. Generalmente,

la política representa una mezcla de compromisos entre objetivos conflictivos. Un ejemplo de política de planificación es "primero que llega-primero servido", no muy eficiente la mayoría de las veces.

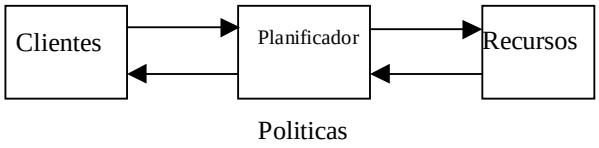


Figura 3.1. Sistema de Planificación

El rendimiento es la característica usada para evaluar un sistema de planificación. Dicho rendimiento es medido en qué tan buena es la planificación producida y cuánto dura el planificador para lograr dicha planificación. Así, la planificación producida es evaluada en términos del criterio de rendimiento que se desea optimizar, el cual, como se dijo antes, normalmente tiene que ver con la minimización del tiempo para culminar un programa (longitud de la planificación). En cuanto al planificador, este es evaluado basado en su tiempo de ejecución para lograr la planificación (sí dos planificadores producen el mismo resultado, aquel que sea más rápido será más eficiente).

En los sistemas paralelos/distribuidos, el problema de planificación es un problema NP-completo, por lo cual se han propuesto un gran número de heurísticas que dan soluciones aproximadas. Las técnicas de resolución de este problema pueden ser usadas por los compiladores, los sistemas operativos, en el diseño de las arquitecturas, etc. Los aspectos que caracterizan a las diferentes estrategias de planificación son [5, 14, 30, 40]:

- *Deterministas o No Deterministas*: Las estrategias de planificación pueden ser clasificadas según la disponibilidad de la información sobre el sistema. En el caso de la planificación determinista, toda la información requerida (detalles de las tareas, plataforma computacional, etc.) se supone que es conocida antes y durante la ejecución del planificador. En el caso no determinista, dicha información no es conocida.
- *Estáticas o Dinámicas*: las estrategias de planificación pueden ser clasificadas según el momento de hacer la planificación. En el caso *estático*, las decisiones se toman antes de arrancar la ejecución de las tareas de un programa dado. En el caso *dinámico*, las decisiones se van haciendo durante la ejecución del sistema. Esto es clásico cuando se desea tomar en cuenta la información actualizada sobre el sistema. Sin embargo, esta técnica impone un sobrecosto importante sobre el sistema. Un enfoque *híbrido* es una mezcla de los dos anteriores, donde algún preprocesamiento es hecho estáticamente para guiar el proceso de planificación dinámico. El enfoque estático es muy usado en el tipo de paralelismo SPMD.
- *Múltiples programas o no*: En el segundo caso sólo hay una aplicación ejecutándose en el sistema, la cual puede consistir en varias tareas cooperando y comunicándose entre sí. En ese caso el objetivo es minimizar el tiempo de ejecución de la aplicación. En el primer caso puede haber varias aplicaciones, paralelas o no, que se ejecutan al

mismo tiempo, por lo cual deben compartir entre ellas los recursos computacionales. En ambos casos, el problema de balance de la carga de trabajo juega un papel relevante.

- *Apropiativas o No Apropiativas:* Una planificación apropiativa permite que una tarea sea interrumpida y removida su asignación a un recurso, bajo la hipótesis de que más adelante volverá a usar dicho recurso. En el segundo caso, una tarea no puede ser interrumpida una vez que ella comienza su ejecución. El uso de técnicas apropiativas introduce cierto sobrecosto sobre el sistema, pero resulta generalmente en un rendimiento superior.
- *Adaptativas o No Adaptativas*: Un planificador adaptativo cambia su comportamiento de acuerdo a la información actualizada que recibe de su entorno. En otras palabras, modifica sus políticas de decisión de acuerdo al comportamiento presente y pasado del sistema, donde se encuentre trabajando. En contraste a este esquema, un planificador no adaptativo no cambia sus decisiones de planificación. El enfoque adaptativo es típicamente dinámico porque colecciona información sobre el sistema y toma decisiones simultáneamente.
- *Centralizadas o Descentralizadas*: en el caso centralizado, se basa en un esquema maestro/esclavo. El maestro es responsable de resolver el problema de planificación. Cada esclavo ejecuta las tareas asignadas por el maestro. Además, los esclavos pueden también enviar trabajos al maestro para ser asignados. Una variante del anterior es una jerarquía de maestros/esclavos. Este esquema divide los esclavos en conjuntos disyuntos, cada uno con su maestro. Cada maestro le asigna trabajo a sus esclavos, y se comunica periódicamente con los otros maestros para balancear la carga. En el esquema descentralizado no hay un maestro, sino que cada procesador mantiene una lista separada de tareas por ejecutar, y, por ejemplo, procesadores sin trabajo piden tareas a otros procesadores. La lista de tareas es una estructura de datos distribuida accesada de una manera asincrónica. Variedades de este enfoque pueden ser definidas, por ejemplo, cuando un procesador sólo puede requerir trabajos a un cierto grupo de procesadores, etc.

En el ámbito de la computación paralela/distribuida, el problema de planificación abarca, entre otros, los siguientes subproblemas [14, 30, 31, 36, 37, 40]:

1. Planificación de tareas de programas paralelos.
2. Planificación de lazos de ejecución (lazos paralelos o con dependencias entre ellos).
3. Planificación de operaciones de Entrada/Salida (E/S).

En esta sección haremos una presentación de los más importantes aspectos de estos subproblemas.

3.1.1 Planificación de Tareas de Programas Paralelos/Distribuidos

Informalmente, el problema de planificación se origina porque las tareas que componen a un programa paralelo/distribuido deben ser ordenadas en el tiempo y espacio, tal que el tiempo de ejecución del programa sea minimizado. Este problema es NP-completo en su

forma general [14, 30]. En la práctica, no se exige una minimización, por lo tanto es posible llegar a una solución adecuada usando algoritmos heurísticos.

Clásicamente, un sistema de planificación de tareas está compuesto por cuatro componentes: el sistema computacional, el programa paralelo, la planificación generada y el criterio de rendimiento. El sistema computacional esta compuesto por *m* procesadores conectados a través de una red de interconexión. Cada procesador puede ejecutar cualquier tarea y la interconexión entre ellos puede ser representada usando un grafo no dirigido (ver figura 3.2.b). Un programa paralelo puede ser modelado por un grafo de tareas (grafo dirigido acíclico), donde cada nodo representa una tarea y las relaciones de precedencia entre ellas son definidas por los arcos entre los nodos (ver figura 3.2.a). Un arco que conecta al nodo *u* con el nodo *v*, significa que la tarea *u* debe ser ejecutada antes que *v*, y su resultado conocido por *v*. Además, se le pueden incluir pesos a los nodos (que especifican las unidades de tiempo requeridas para su procesamiento) como a los arcos (que especifican el tamaño del mensaje/información a enviar o el costo/tiempo de comunicación entre las tareas). De esta manera, tanto el tiempo de comunicación, como de ejecución de las tareas, pueden ser representados. Si no hubiera relaciones de precedencias entre ellos, simplemente no existirían arcos. Una función de planificación *f* asigna cada tarea sobre cada procesador, y determina el momento en que se debe iniciar su ejecución. Dicha planificación debe preservar las relaciones de precedencias y las restricciones de comunicación. Para describir la planificación dada por un sistema de planificación, se usan los *Diagramas de Gantt*. Un Diagrama de Gantt da una noción del esquema de planificación propuesto (donde cada tarea se ejecutará, cuándo comenzará y cuándo terminará). Las medidas de desempeño describen los objetivos a optimizar durante la planificación. Por ejemplo, balancear la carga de trabajo o minimizar el tiempo de ejecución de los programas. Como se dijo antes, la última medida es la más usada, la cual es también conocida como la minimización de la longitud de la planificación.

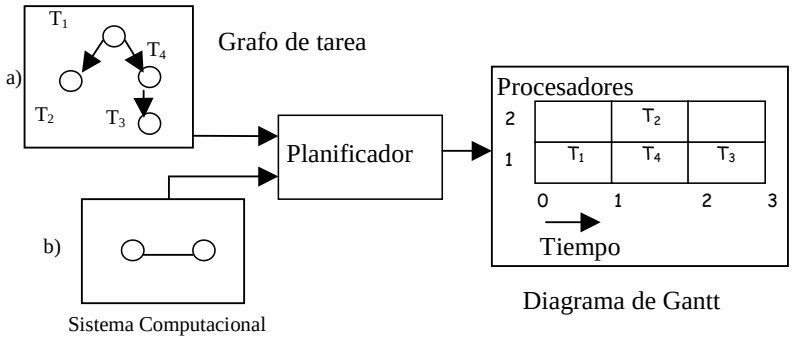


Figura 3.2. Elementos necesarios en la tarea de Planificación

Cuando los tiempos de comunicación no son considerados, es posible obtener una solución óptima en los siguientes casos:

- Cuando el programa paralelo es modelado por un grafo de tareas tipo árbol.
- Cuando hay solamente dos procesadores en el sistema.

Una heurística produce una planificación con menos esfuerzo, pero no garantiza una solución óptima. Heurísticas buenas generan soluciones cercanas al óptimo, a través del uso de un conjunto de intuiciones o suposiciones. Una de las clásicas heurísticas de planificación, es la heurística llamada *lista de planificación* (list scheduling). En esta heurística, a cada tarea se le asigna una prioridad. Después, una lista de tareas es construida en un orden decreciente de prioridad. Cuando un procesador es disponible, la tarea con más alta prioridad, es seleccionada desde dicha lista, para ser asignada a ese procesador. Si más de una tarea tiene la misma prioridad, se escoge alguna al azar. Las diferentes versiones de esta heurística difieren entre si en la manera en que las prioridades son asignadas. Al introducir los lapsos de comunicación, el problema se puede convertir en un problema min-max, es decir, en conseguir la mejor relación entre el máximo de paralelismo con la mínima comunicación. Si las tareas son asignadas, considerando maximizar el número de tareas a ejecutar simultáneamente, sin considerar los costos de transmisión de la información, se puede llegar a situaciones donde el programa se ejecuta más lentamente que si se ejecuta en un solo procesador. Un procedimiento genérico de la heurística *lista de planificación* es el siguiente [14, 30]:

1. A cada nodo del grafo de tareas se le asigna una prioridad: normalmente, se crea una lista con las tareas que están listas para ser ejecutadas (al inicio, dicha lista estará sólo conformada por las tareas que no tienen predecesores). A cada tarea a insertar en la lista se le asigna una prioridad. Además, las tareas en la lista se ordenan según sus prioridades.
2. Mientras existan elementos en la lista:
   2.1 Obtener una tarea de la lista.
   2.2 Seleccionar un procesador sin trabajo para asignarle dicha tarea.
   2.3 Cuando las ejecuciones de todos los predecesores inmediatos de una tarea dada han sido computadas, esta tarea está lista para ser ejecutada, por lo que se inserta en la lista con una prioridad dada.

Las prioridades entre las tareas pueden ser definidas de diferentes formas. Dos ejemplos de esquemas de asignación de prioridades son:

- Las tareas con los más grandes tiempos de ejecución estimados, tendrán la más alta prioridad.
- Las tareas tendrán asignadas sus prioridades aleatoriamente.

El primer esquema de prioridades es óptimo, en plataformas, donde los tiempos de comunicación no son importantes. Si no se consideran los tiempos de comunicación, todas las tareas que están listas para ejecutarse se asignarán sobre todos los procesadores disponibles. El costo de comunicación debe ser considerado cuando se trabaja en plataformas distribuidas. Al considerar los tiempos de comunicación, si dos tareas tienen un largo tiempo de comunicación entre ellas, se tiende a asignar a ambas tareas en el mismo procesador. Por ejemplo, en la figura 3.3, dependiendo del tiempo de

comunicación entre T1 y T3 o T1 y T2, aquel que tenga el mayor tiempo de comunicación entre ellos debe ser asignado al mismo procesador donde es asignado T1. Si suponemos que ese es el caso entre T1 y T2, y si el tiempo de comunicación entre T1 y T3 es menor que el tiempo para ejecutar T2, T3 podría ser asignado en un procesador diferente; en caso contrario T3 debe ser asignado al mismo procesador de T1 y T2. Es decir, todas las tareas serian asignadas al mismo procesador.

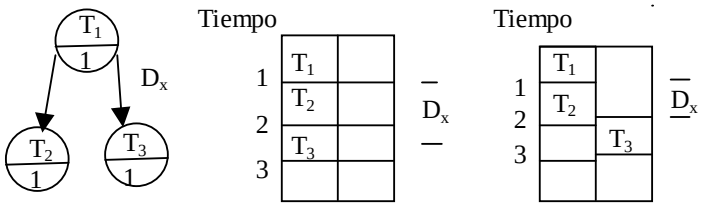


Figura 3.3. Planificación de tareas considerando los tiempos de comunicación

Una modificación a la heurística *lista de planificación*, para incorporar los lapsos de comunicación entre las tareas, es la siguiente: cuando una tarea está lista para ser ejecutada y existe un procesador libre, si la asignación de esta tarea al procesador no introduce un tiempo de comunicación con alguno de sus antecesores, que es mayor al tiempo estimado para que concluya la tarea en ejecución en el procesador donde dicho antecesor culminó, es preferible esperar a que se libere dicho procesador para asignarle esta tarea, y pasar a estudiar la asignación de la siguiente tarea. De lo contrario, la tarea actual puede ser asignada al procesador libre. Otras técnicas más sofisticadas han sido propuestas.

3.1.2 Planificación de las Operaciones de Entrada/Salida

El procesamiento masivamente paralelo es en la actualidad la respuesta más prometedora en la búsqueda por incrementar el rendimiento de los computadores. Sin embargo, muchas de las actuales aplicaciones requieren de un supercomputador que sea capaz de procesar grandes cantidades de datos en forma eficiente. El cuello de botella debido a las operaciones de E/S en los computadores paralelos, ha generado diferentes esquemas para resolver este problema, por ejemplo, usando paralelismo de bajo nivel (a través de técnicas tales como partición de discos), solapando el acceso a los datos con cálculos, paralelizando el acceso a los datos (ya sea explotando el paralelismo a nivel del programa o incrementando los recursos), y otros más. La planificación de las operaciones de E/S es otra alternativa [31].

Como cualquier computador, los computadores masivamente paralelos deben balancear la capacidad de cálculo, de acceso a memoria, de comunicación y de operaciones de E/S. Casi todos los sistemas paralelos/distribuidos proveen alguna clase de hardware y software para soportar E/S paralelas, usualmente en la forma de un subsistema de E/S. A pesar de eso, los subsistemas de E/S han sido los huérfanos en el

área de arquitecturas de computadores y han recibido muy poca atención. Idealmente, un subsistema de E/S en estas plataformas permite la transferencia de datos en paralelo, entre los procesadores y los discos, y distribuye la carga repartiendo operaciones de E/S a través de múltiples nodos. El problema se genera cuando existe un gran número de operaciones de E/S en una aplicación paralela dada. Es decir, cuando la transferencia de datos domina la ejecución. Este es el caso de muchas aplicaciones científicas que se ejecutan en los computadores paralelos, lo que puede degradar severamente sus tiempos de ejecución, si no hay los recursos para realizarlas rápidamente. Algunas de las aplicaciones con fuertes requerimientos de E/S son: aplicaciones que deben analizar grandes volúmenes de datos (modelos climáticos, procesamiento sísmico, etc.), aplicaciones que deben procesar imágenes (visualización de imágenes, etc.), aplicaciones multimedias, etc., las cuales se ejecutan sobre una gran variedad de arquitecturas paralelas y distribuidas. Está claro, que los órdenes de magnitud en cuanto a los requerimientos de E/S seguirán creciendo. Además, con el aumento del poder computacional de los procesadores, cada vez más aplicaciones se convertirán en programas de este tipo.

El modelo básico de un subsistema de E/S comprende un conjunto de procesadores de E/S y un conjunto de discos. Cada procesador de E/S controla uno o más discos. Los procesadores de E/S pueden ser procesadores que no corren ningún tipo de trabajo, o pueden ser procesadores que también realizan cómputos. Otra organización común, son supercomputadores con grandes discos, o arreglos de discos, conectados a las unidades de cálculo a través de un bus común. En general, los datos de cada archivo son distribuidos a través de los discos, usando un método conocido como descomposición. En este método los archivos son divididos en un número de unidades, cada una de las cuales es una unidad lógicamente contigua de la otra, en el archivo de datos. Normalmente, las unidades son distribuidas a través de los discos, usando una estrategia “round robin” (se van asignando de manera circular una a una en cada disco hasta asignarlas a todas). La ventaja de este método es que si múltiples procesadores requieren diferentes partes de un archivo, el requerimiento de cada uno puede ser distribuido a través de los diferentes discos, para así, recuperarlos en paralelo. Además, al repartir los datos entre los diferentes discos se reduce la probabilidad de que varios procesadores intenten leer el mismo disco simultáneamente.

El número y la complejidad de las operaciones de E/S dependen de la manera en que los datos son distribuidos sobre los discos y la memoria. Frecuentemente, es mejor distribuir explícitamente los datos en la memoria para minimizar el número de requerimientos de E/S. El objetivo del problema de optimización de las operaciones de E/S es proveer un camino de comunicación libre de cuellos de botella entre los procesadores y las unidades de E/S. Este problema se ha convertido en crítico, y la necesidad por anchos de banda amplios para las operaciones de E/S ha emergido con una gran importancia en muchos computadores paralelos. Este problema puede ser visto desde diferentes ángulos: lenguajes, compiladores, sistemas de archivos, sistemas de almacenamiento, sistemas de redes, sistemas operativos, etc. A nivel de sistemas de archivos, los estudios conducen en darle soporte a las operaciones de E/S paralelas, a través de sistemas paralelos de archivos, los cuales consideran factores tales como

descomposición de archivos, pre-búsquedas de información, memorias caches, y otras técnicas más. A nivel de compiladores y sistemas de ejecución se debe explotar la información provista por los usuarios, para optimizar la asignación de los datos. A nivel de los compiladores, el reconocer las operaciones de E/S es clave para pre-planificarlas, solaparlas con operaciones de cálculos, etc. Desde el punto de vista del hardware, se han propuestos varios modelos como el de *sin restricciones paralelas*, en el cual la memoria principal esta compuesta por M bloques, cada uno de tamaño igual a B páginas. La idea es que M bloques, compuestos por B páginas contiguas, puedan ser transferidos en una simple operación de E/S. Aquí, existen dos formas de paralelismo, a nivel de la transferencia de los bloques tal que B páginas puedan ser simultáneamente transferidas, y a nivel de la transferencia de los discos tal que M bloques puedan ser transferidos. Una extensión a este modelo es el Modelo de *disco paralelo*, tal que M bloques puedan ser transferidos simultáneamente sólo si provienen de M discos diferentes. Este modelo es más realista.

Hay muchos estudios sobre la necesidad de planificar las operaciones de E/S para aumentar el rendimiento de las plataformas de multiprocesadores. Básicamente, es importante la planificación en ciertas aplicaciones, como en la visualización 3D que pasa a través de fases con diferentes grados de paralelismo de cálculo y requerimientos de E/S. En estos casos, los requerimientos de datos deben ser satisfechos antes que la computación pueda proceder. Esta familia de aplicaciones puede beneficiarse sustancialmente de enfoques de planificación de operaciones de E/S fuera de línea, para evitar las fuentes de cuello de botella existentes en el sistema de E/S.

El problema de planificación de las operaciones de E/S puede ser resuelto modelándolo como el problema de comunicación entre los procesadores y las unidades de E/S, tales como los discos (ver figura 3.4). Viéndolo así, el problema consiste en maximizar la utilización del camino entre ambos conjuntos y minimizar el tiempo para comenzar cada operación de E/S.

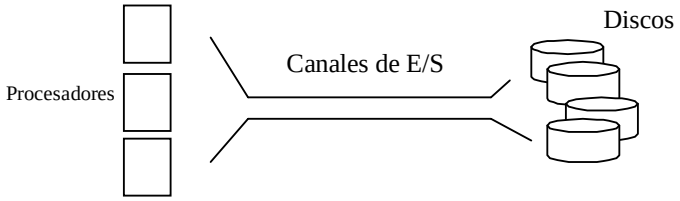


Figura 3.4. Problema de Optimización de Ejecución de las Operaciones de E/S

Las operaciones de E/S pueden ser consideradas como tareas de transferencias de datos (T1-T4), las cuales son requeridas entre un conjunto de procesadores y un conjunto de recursos de E/S (ver figura 3.5.a). Para una situación como ésta, podrían proponerse varios esquemas de planificación con el fin de reducir el tiempo empleado en las operaciones de E/S. Por ejemplo, supongamos que se tienen dos canales de E/S independientes. En la figura 3.5.b, la *planificación 1* muestra un esquema de planificación basada en el orden en que ellas llegan al sistema. El esquema de

planificación 2 muestra que las transferencias pueden ser completadas en un tiempo más corto al reorganizar el orden en las cuales ellas son ejecutadas.

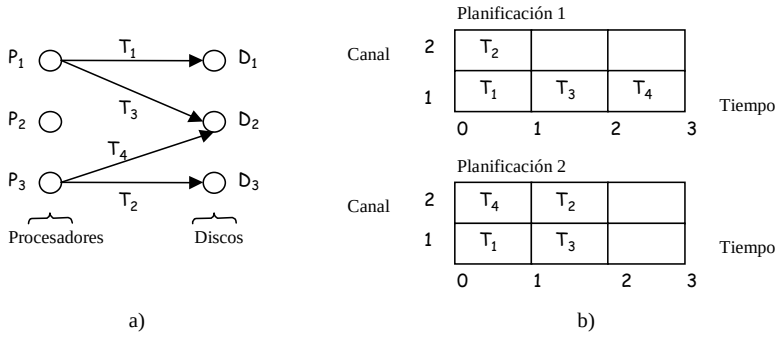


Figura 3.5. Planificación de E/S.

Para explicar el modelo de planificación de las operaciones de E/S propuesto en la figura 3.5.a, consideremos un grupo de operaciones de E/S las cuales son realizadas entre un conjunto de procesadores y un conjunto de recursos de E/S. Cada recurso puede representar un único disco independiente o un arreglo de discos, o cualquier otra unidad de E/S. Además, las peticiones de transferencias de E/S son numeradas en el orden en que ellas llegan al sistema, pero pueden planificarse en cualquier orden. Por otro lado, cada procesador y cada disco pueden participar en una transferencia de datos en un momento dado. Para modelar esto, se utilizan grafos como la estructura matemática, de la siguiente manera (ver figura 3.5.a): por un lado, un grupo de nodos representan los procesadores de donde se generan las transacciones, del otro lado, otro grupo de nodos representan los recursos de E/S, y los arcos representan las transferencias de datos. El peso de los arcos representa la cantidad de datos a transferirse. Así, el grafo a usar es bipartido, en el cual los vértices están divididos en dos conjuntos y los arcos conectan los vértices de ambos conjuntos. Los elementos del grafo son los siguientes:

P1, P2, P3: representan los procesadores.
D1, D2, D3: representan los recursos de E/S.
T1, T2, T3, T4: representan las transacciones entre un procesador y una unidad de E/S. Pueden tener un peso para representar la cantidad de datos a transferir entre ellos (Peso(T_j)).

El problema de planificación modelado de esta forma puede representarse como un problema de mínimo grafo k-coloreado, donde k es el ancho de banda entre los procesadores y los recursos de E/S. Así, k determina el volumen máximo de datos que se pueden transferir en un momento dado, entre los recursos de E/S y los procesadores. El problema de mínimo grafo coloreado es definido como:

Definición 3.1: Dado un grafo G=(V, E) no dirigido, el problema de mínimo grafo coloreado consiste en encontrar el mínimo conjunto de colores necesarios

para colorear los arcos E, tal que dos arcos con el mismo color no comparten un vértice en común.

Mientras que el problema de mínimo grafo k-coloreado puede ser definido como:

Definición 3.2: El problema de mínimo grafo k coloreado consiste en resolver el problema de mínimo grafo coloreado, tal que cada color pueda ser usado para colorear k arcos como máximo.

Resolver el problema del mínimo grafo k-coloreado es equivalente a resolver el problema de optimización de las operaciones de E/S (mínimo número de transacciones, es equivalente al mínimo número de colores), de tal manera de ejecutar en el mínimo tiempo posible el mayor número de transacciones u operaciones de E/S, maximizando el uso de los recursos de E/S y de los canales de comunicación. Así, un grafo coloreado de G representa una planificación de las operaciones de E/S, donde todos los arcos con *color=i* representan el grupo de transferencias de datos que pueden tomar lugar en el mismo instante de tiempo. El número de colores requerido para colorear los arcos determina la longitud de la planificación.

3.1.3 Planificación de Lazos de Ejecución

Generalmente, los lazos son la fuente de multiprocesamiento más rica en muchas aplicaciones paralelas. Una manera de explotar este paralelismo es ejecutar las iteraciones de los lazos en paralelo sobre los diferentes procesadores. Por ejemplo, al haber N iteraciones independientes a ejecutar en un sistema multiprocesador con P procesadores, el objetivo de la planificación consiste en distribuir esas N iteraciones sobre los P procesadores de la manera más eficiente y con el mínimo costo. Esto no es fácil, ya que la cantidad de trabajo en cada iteración puede ser diferente, el valor de N puede ser desconocido durante el tiempo de la compilación, el número de procesadores P es dependiente de la máquina, y la cantidad de sobrecosto introducida depende de la máquina y del planificador.

El problema de planificación de lazos puede ser visto como parte de un problema general de planificación de tareas sobre multiprocesadores, donde se busca minimizar el tiempo de ejecución de las aplicaciones paralelas. La planificación de lazos consiste en asignar las diferentes iteraciones de un lazo sobre diferentes procesadores, definiendo el orden en que deben ser ejecutadas [14, 30]. Una planificación de lazos debe explotar el paralelismo que puede existir entre diferentes iteraciones de lazos y dentro de cada iteración, minimizando la comunicación. Los factores a considerar en la planificación de lazos son: la dependencia de datos entre sus iteraciones, la carga entre los procesadores, el sobrecosto debido a la sincronización y a la comunicación entre las iteraciones, y el costo introducido por el planificador, entre otros.

Cuando hay una independencia completa entre las iteraciones de un lazo, se habla de un *lazo paralelo*. Iteraciones independientes en un lazo son perfectas para la ejecución en

paralelo. Esto es debido a que las iteraciones se pueden ejecutar en cualquier orden y siempre producen el resultado correcto. Las comunicaciones entre procesadores son necesarias, si una dependencia de datos existe entre iteraciones del lazo, y las iteraciones han sido asignadas en procesadores separados. En este caso, un solapamiento parcial entre lapsos de comunicación y de ejecución, debido a pase de información entre las iteraciones, podría ser requerido. Nosotros estudiaremos ambos problemas de planificación de lazos, cuando se tiene un lazo paralelo o cuando se tienen lazos con iteraciones, las cuales tienen dependencias de datos entre ellas.

3.1.3.1 Planificación de Lazos Paralelos

Un lazo paralelo consiste en un lazo, cuyas iteraciones no tienen dependencia de datos entre ellas, es decir, las iteraciones pueden ser ejecutadas en cualquier orden o simultáneamente. Un ejemplo de lazo paralelo es:

```
For (i=0; i≤12; i++)
        A[i]=B[i]+C[i]*D[i]
```

Los lazos paralelos son perfectos para paralelizar (dos ejemplos de posibles paralelizaciones son dados en la figura 3.6 para el caso de un sistema con 4 procesadores). Normalmente, el número de iteraciones es mucho mayor que el número de procesadores disponibles, por lo que un procesador dado debe ejecutar más de una iteración. Si los tiempos de ejecución de las diferentes iteraciones son semejantes, el lazo se llama uniformemente distribuido y se podría usar un enfoque tal que las N iteraciones sean distribuidas uniformemente entre los P procesadores. Así, P iteraciones siempre estarán ejecutándose en paralelo. En los otros casos, cuando los lazos son no-uniformes, en cuyo caso las iteraciones tienen diferentes tiempos de ejecución, asignar igual número de iteraciones sobre los procesadores, no siempre resulta en que cada procesador tenga la misma cantidad de trabajo, en este caso se deben seguir otros enfoques.

Figura 3.6. Dos ejemplos de Asignación de Lazos Paralelos

Métodos sofisticados que combinan procesos de descomposición y de planificación de lazos pueden ser usados. Por ejemplo, para el siguiente programa:

```
For (i=0; i≤3; i++)
        T1[i]
        T2[i]
```

Si suponemos tres procesadores y las tareas T_1 y T_2 independientes entre sí, una primera planificación podría ser la dada en la figura 3.7.a., donde uno de los procesadores requerirá ejecutar dos iteraciones (ver figura 3.7.a). Una manera de optimizar dicha ejecución es redefiniendo los lazos paralelos a ejecutar (ver figura 3.7.b). En este caso, el tiempo de ejecución del lazo puede ser reducido a través de un proceso de separación de las diferentes tareas que componen a cada lazo, las cuales ahora se podrán ejecutar independientemente (en el caso de la figura 3.7.a, la longitud de la planificación es de 4 unidades de tiempo, mientras que en el caso de la figura 3.7.b, en el cual se descomponen las tareas de cada lazo antes de hacer la planificación, la longitud de la misma es de 3 unidades de tiempo).

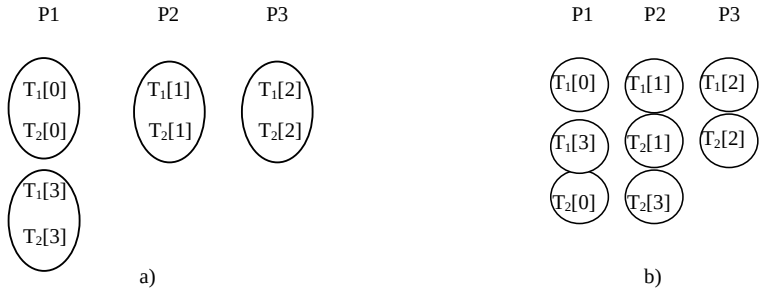


Figura 3.7. Planificación sobre 3 procesadores basada en una descomposición de los lazos

Una clasificación de los diferentes esquemas de planificación de lazos paralelos es la siguiente [14, 30]:

- *Planificación de Lazos Paralelos Estática*: depende del comportamiento promedio en el sistema, y no del estado actual del mismo, y es aplicable a lazos distribuidos uniformemente. Asigna iteraciones a procesadores estáticamente, es decir, durante el tiempo de compilación, buscando minimizar los problemas de sincronización durante la ejecución, pero sin equilibrar la carga de trabajo. El más simple algoritmo de planificación estático consiste en dividir el número de iteraciones entre los procesadores disponibles tan equitativos como posibles, con la esperanza de que cada procesador reciba la misma carga de trabajo. El desequilibrio en la carga de trabajo, puede ser grave, en el caso de lazos no-uniformemente distribuidos. Una planificación estática no es aplicable, si los bordes de un lazo son desconocidos durante el tiempo de compilación.
- *Planificación de Lazos Paralelos Dinámica*: difiere de la anterior en la manera de asignar las iteraciones, la cual se realiza durante el tiempo de ejecución de la aplicación. No asume un conocimiento preliminar del tiempo de ejecución de las iteraciones y es interesante cuando las iteraciones pueden tomar diferentes tiempos de ejecución. En este esquema los procesadores intercambian periódicamente información sobre sus cargas de trabajo, además de iteraciones. Estos métodos dan mejores rendimientos en el caso de cargas transitivas no predecibles y lazos no uniformemente distribuidos. Claramente, cierto sobrecosto es introducido en este esquema. Los esquemas dinámicos se pueden agrupar en dos categorías:

- *Basados en una Cola Central*: en este caso, las iteraciones del lazo son guardadas en una cola central compartida y cada procesador saca algunas iteraciones desde ella para ejecutar (ver figura 3.8). Así, las iteraciones son tomadas del frente de la cola compartida siguiendo un orden FIFO. La mayor ventaja, es la posibilidad de balancear la carga de trabajo. Entre sus problemas, tenemos que ellos no facilitan la afinidad de una iteración por un procesador, porque pueden ser asignados a cualquier procesador, e introducen un sobrecosto a nivel de la comunicación con la cola central, la cual, a su vez, puede convertirse en un cuello de botella. Un esquema dinámico basado en este enfoque consiste en asignar, un número inicial de iteraciones grandes sobre los procesadores, y luego decrecer/aumentar el número de iteraciones subsecuentemente a asignar, dependiendo de la carga de trabajo en los procesadores y del número de iteraciones por asignar, hasta asignar a todas las iteraciones que están en la cola.

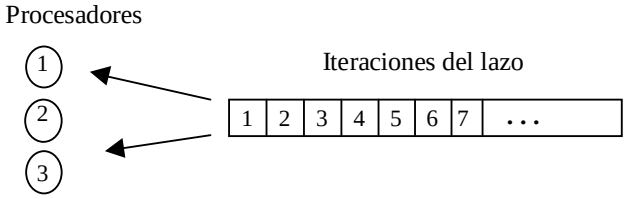


Figura 3.8. Un esquema de Planificación de Lazos Paralelos basado en una cola central

- *Basado en Colas Distribuidas*: permite explotar la afinidad que muchas iteraciones tienen durante su ejecución en un procesador dado y eliminar el cuello de botella de las colas centrales al distribuir la cola en los diferentes procesadores. Generalmente, las iteraciones son divididas estáticamente en colas locales, lo que hace que un procesador deba hacer accesos remotos, cuando ocurren desequilibrios.

3.1.3.2. Lazos con Dependencia de Datos entre Iteraciones

Cuando existe una dependencia entre diferentes iteraciones, se necesita definir un enfoque de planificación, que pueda explotar el paralelismo dentro de cada iteración y entre diferentes iteraciones del lazo. Claramente, un enfoque basado en descomponer un lazo determina todo el paralelismo posible en un lazo, pero este enfoque es impráctico cuando los bordes de los lazos son muy grandes. En el caso general de lazos, la dependencia de datos en las iteraciones puede ser de dos tipos [7, 11, 14, 33]:

- *Intradependencia:* es cuando los datos son pasados entre diferentes iteraciones.
- *Interdependencia*: es cuando los datos son pasados desde una tarea a otra dentro de la misma iteración.

El segundo tipo de dependencia puede ser representado usando *grafos de tareas*. El primero no puede ser representado usando grafos de tareas, ya que expresa las

dependencias entre tareas que están en diferentes iteraciones. Daremos ahora algunas definiciones [7, 11, 14, 33]:

- *Vector de Iteraciones:* cada instancia de ejecución de un lazo n-anidado es descrito por un vector de iteraciones $\{I_1, \ldots, I_n\}$. Los elementos del vector de iteración son enumerados desde el lazo externo al interno, y cada valor en el vector representa el valor de la iteración actual de cada lazo en dicha instancia. Por ejemplo, el vector de iteraciones {1, 2} representa la instancia de ejecución de un lazo 2-anidado para el caso cuando el valor de iteración del lazo interno es *2* y del externo es *1*.
- *Vector Distancia:* suponga dos tareas v y w en un lazo 1-anidado. Si la tarea w ejecutada en la iteración I_w depende de la tarea v ejecutada en la iteración I_v, el vector distancia es $D=I_w-I_v$. Existe una inter-dependencia entre las tareas v y w si y sólo si $I_w=I_v$, de lo contrario es un lazo con intra-dependencia. En el caso n-anidado, por cada lazo se determina su distancia, y el conjunto de ellos conformaran el vector distancia.
- *Par de Dependencia*: existe entre dos tareas v y w, con w ejecutada en la iteración I_w y v ejecutada en la iteración I_v. La nomenclatura que se usa es (D, W), donde D es el vector distancia y W es el tamaño del mensaje que w recibe desde v. Así, cada vector distancia conforma un par de dependencia.
- *Conjunto de Dependencia*: es el conjunto de todos los pares de dependencia entre dos tareas.
- *Vector de Bordes Superiores de un lazo n-anidado* será igual a $\{b_1, b_2, \ldots, b_n\}$, donde b_i es el borde superior del lazo del nivel i.
- *Vector Descomposición:* es igual a $\{u_1, \ldots, u_n\}$, es decir, el lazo i es descompuesto u_i veces.

Normalmente se asume que los lazos están normalizados, es decir, los valores de sus iteraciones arrancan en uno, hasta llegar al valor de sus bordes superiores, a través de pasos de uno. También se asume que todos los tareas están dentro de los lazos. En la literatura se han desarrollado muchas técnicas para normalizar lazos.

Las dependencias presentes en un lazo pueden ser representadas usando *grafos de lazos de tareas*. Estos son grafos dirigidos con pesos G=(V, A), donde V es el conjunto de nodos y A de arcos. Un nodo es una tarea y un arco entre dos nodos representa la dependencia entre las tareas. El peso en el nodo es la cantidad de cálculo de la tarea y en los arcos el conjunto de dependencia entre las tareas. Dicho lazo puede contener ciclos.

Ejemplo 3.1. Suponga el siguiente programa:

```
For i= 1 to 2
        For j=1 to 2
                T1(i, j)
                T2(i, j)
```

donde,

Tarea T1(i,j):
X[i, j]= F1(V[i-1, j], X[i-1, j-1])
Z[i, j]=constant1

Tarea T2(i, j):
Y[i, j]=F2(Z[i, j], Y[i-1, j])
V[i, j]=constant2

Y F1 y F2 no tienen efectos colaterales, es decir no modifican sus parámetros ni a variables globales. En dicho ejemplo, el grafo de lazos de tareas está compuesto por dos nodos T1 y T2 (ver figura 3.9). Los pesos de los nodos son los tiempos para calcular las funciones F1 y F2 (4 y 8 unidades de cálculo, respectivamente) y las asignaciones (1 unidad de cálculo cada una). De esta manera, la cantidad de cálculo de la tarea T1 es 7 y de la tarea T2 es 9 (pesos de las tareas T1 y T2). Para obtener los pesos de los arcos se asume que los arreglos X, Y, Z y V requieren 10, 12, 20 y 5 unidades de almacenamiento, respectivamente. Además, T1 usa durante la iteración <i,j> a V[i-1,j] que es calculado por la tarea T1 en la iteración <i-1,j>, y a X[i-1,j-1] que es calculado por la tarea T1 en la iteración <i-1,j-1>. Los vectores distancias para esas dos dependencias son <1,0> y <1,1>, respectivamente. De la misma manera se le pueden definir a T2 sus arcos. T2 usa en la iteración <i, j> a Z[i,j] (el conjunto de dependencias entre T1 y T2 es {<0,0>,20)}) y en la iteración <i-1, j> a Y[i-1,j] (el conjunto de dependencias consigo mismo es {<1,0>,12}).

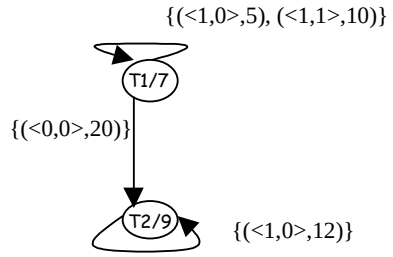


Figure 3.9. Un grafo de lazos de tareas

3.1.3.2.1 Descomposición de Lazos

Consiste en reemplazar las iteraciones de un lazo por su código no iterado. La idea es descomponer el lazo de manera de eliminar las dependencias de lazos tal que varias iteraciones solapen sus ejecuciones. Por ejemplo, para el lazo siguiente:

For i= 1 to 4
X[i+2]= X[i+1]+X[i]

Si se descompone una vez nos daría:

For i= 1 to 4 step 2
X[i+2]= X[i+1]+ X[i]
X[i+3]= X[i+2]+ X[i+1]

De esta manera, algún paralelismo puede aparecer. Cuando un lazo es descompuesto u veces, u+1 copias del cuerpo son generadas, la variable de control del lazo es ajustada por cada copia, y el paso del lazo es multiplicado por u+1. Así, cuando un conjunto de n lazos anidados, con vector de borde superior igual a $\{b_1, \ldots, b_n\}$, es descompuesto según el vector $u=\{u_1, \ldots, u_n\}$, $\prod_{i=1}^{n} (u_i+1)$ copias del cuerpo son generadas y la variable del lazo es ajustada por cada copia, tal que el valor del paso del lazo i^{iesimo} es multiplicado por u_i+1. Un *grafo de tareas replicadas* ($G_u=V_u, A_u$) es un grafo acíclico que representa el cuerpo del lazo después de ser descompuesto usando un vector $u=\{u_1, \ldots, u_n\}$. El conjunto de nodos V_u es el conjunto de tareas replicadas según el proceso de descomposición ($|V_u|=|V|*\prod_{i=1}^{n} (u_i+1)$). Su conjunto de arcos esta compuesto por el conjunto de arcos que representan las inter-dependencias del grafo de lazos de tareas originales y las nuevas inter-independientes como resultado del proceso de descomposición. El peso de los nodos y arcos replicados es igual al de los originales (ver la figura 3.10, que muestra al grafo de la figura 3.9 para u={1, 1}, es decir, cuando se descomponen los lazos externo e interno una vez).

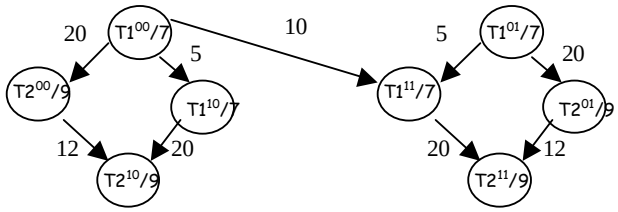


Figura 3.10. Un Grafo de Tareas Replicadas para el Grafo de Lazos de Tareas de la figura 3.9 y u={1, 1}.

El proceso de descomposición de un lazo permite explotar el paralelismo existente en diferentes iteraciones. Una manera de hacerlo es:

1. Generar el *grafo de lazos de tareas*.
2. Definir el vector de descomposición *u* para descomponer el lazo.
3. Generar el *grafo de tareas replicadas* del grafo de lazos de tareas previo.
4. Planificar dicho grafo sobre los procesadores disponibles

La clave es definir cómo descomponer el lazo. Para el grafo de la figura 3.10, dos posibles planificaciones son presentadas en la figura 3.11, si suponemos dos procesadores, y la velocidad de procesamiento y tasa de transferencia iguales a uno.

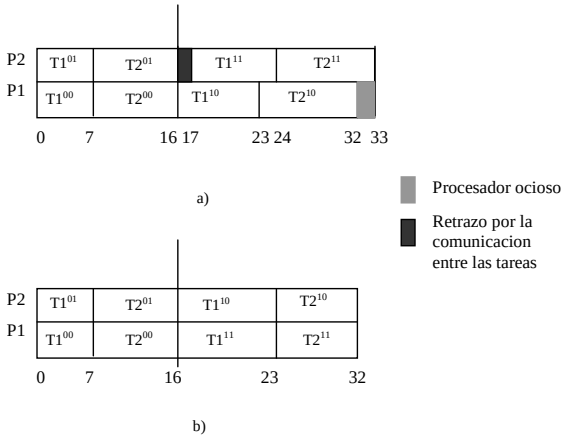


Figura 3.11. Posibles Planificaciones del Grafo de tareas replicados para u={1, 1}

La planificación óptima es presentada en la figura 3.11.b. Esta planificación asigna $T1^{11}$ (que se ejecuta en la segunda iteración de *i* y de *j*) en el mismo sitio donde se ejecuta $T1^{00}$, para minimizar los costos de comunicación. La planificación propuesta en 3.11.a, es el caso cuando $T1^{00}$ y $T1^{11}$ se ejecutan en procesadores diferentes. En este caso, hay que esperar una unidad de tiempo más para que se pueda comenzar con la tarea $T1^{11}$. Esto es debido a la dependencia de T1 consigo misma (<1,1>, 10), es decir, $T1^{11}$, que se ejecuta en la segunda iteración de *i* y de *j*, depende de los resultados de $T1^{00}$ obtenidos en la primera iteración de *i* y de *j*. El tiempo de comunicación entre $T1^{11}$ y $T1^{00}$ (10 unidades de cálculo) es más grande que el tiempo de ejecución de $T2^{00}$ (9 unidades de tiempo), que se ejecuta en la primera iteración de *i* y de *j*, por lo cual estos tiempos sólo se solapan parcialmente.

3.1.4 Planificación de Estructuras Condicionales

Esta es una típica situación donde el no-determinismo está presente, ya que la ejecución de un conjunto de instrucciones depende del valor condicional previo a ellas [14]. En ese sentido, un enfoque dinámico de planificación sería más apropiado, según el cual, al saber que el conjunto de instrucciones debe ser ejecutado, se pasa a asignar dicho conjunto de instrucciones/tareas sobre el sistema. Otra vía, al usar un enfoque determinista, es:

1. Generar todas las posibles instancias de ejecución para un programa dado
2. Construir el grafo de tareas respectivo para cada instancia
3. Obtener un esquema de planificación para cada una de las instancias
4. Mezclar las diferentes planificaciones obtenidas

El problema es cuando existen muchas instancias para un programa dado, donde resulta poco práctico combinar las posibles planificaciones de cada uno. La única vía es

tratar de minimizar el grado de no-determinismo en el programa paralelo. Una forma de minimizar dicho grado, es determinando aquellas secuencias en una estructura de decisión que tengan equivalentes tiempos de cálculo y de comunicación, en cualquiera de sus ramificaciones, es decir, buscando equivalencias entre las tareas de los grafos de las diferentes instancias. Al conseguir conjuntos de tareas con dichas características, éstas podrían ser representadas por una misma tarea simple. Otros mecanismos más complejos pueden ser propuestos.

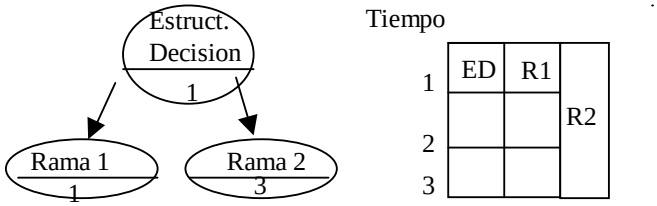


Figura 3.12. Un ejemplo de Planificación de una Estructura Condicional

3.2. Problema de Asignación de Tareas en los Sistemas Distribuidos/Paralelos

En el ciclo de vida de un programa en los Sistemas Distribuidos/Paralelos, su descomposición en tareas y la asignación de estas sobre los diferentes procesadores, son aspectos a considerar, para optimizar el rendimiento de esos sistemas [14, 15, 36, 40]. En particular, la asignación de tareas es uno de los problemas más importantes a considerar, por esa razón ha sido objeto de numerosos estudios. Este problema es NP-completo, por lo que en la mayoría de los estudios se han simplificado las dimensiones del problema, eliminando ciertos criterios y/o imponiendo ciertas restricciones.

El término de "asignación de tareas" ha sido muchas veces confundido con el término de "planificación de tareas". El problema de "asignación de tareas" es un caso especial del problema de planificación de tareas, en el cual no se consideran las relaciones de precedencia entre las tareas que forman el programa, por lo que el orden en que se ejecutan las tareas del programa paralelo, no es importante. En este caso, el programa distribuido/paralelo debe ser asignado sobre los procesadores para hacer un uso eficiente de los recursos. Este problema es más crítico en las arquitecturas tipos MIMD con memoria distribuida, donde el costo comunicacional es crucial. En el caso de los MIMD a memoria compartida, dichos costos de comunicación se reducen de una manera significativa. En el caso de las SIMD se encuentra otro tipo de problema de asignación, el cual en este caso es de datos. El objetivo principal al tratar de resolver este problema, consiste en desarrollar algoritmos de asignación de tareas, que minimicen el tiempo total de ejecución de una aplicación paralela/distribuida dada, a través de una utilización eficiente de los recursos del sistema.

La asignación de un programa compuesto por *n* tareas sobre una arquitectura compuesta por *P* procesadores, es equivalente a la proyección F: T->S según un objetivo predefinido, donde *T* es el conjunto de las *n* tareas y *S* es el conjunto de los *P* procesadores. Existen P^n posibles asignaciones. El rendimiento del sistema depende de la asignación escogida. Para realizar esta selección varios criterios pueden ser utilizados, tales como equilibrar la carga de trabajo entre los procesadores, optimizar el grado de paralelismo, minimizar el costo de comunicación entre los procesadores, minimizar el costo de ejecución de las tareas, etc. Igualmente, la selección debe tomar en cuenta las restricciones y características impuestas por el programa, tales como las restricciones temporales típicas de las aplicaciones tiempo real; así como las impuestas por los sistemas, tales como los limites de la memoria principal en cada sitio, la capacidad de los canales de comunicación, la topología de interconexión entre los procesadores, etc.

Existe una gran cantidad de trabajos dedicados al problema de asignación de tareas, pero la mayoría ha reducido las dimensiones del problema eliminando ciertos criterios y/o definiendo ciertas hipótesis. La asignación de tareas en sistemas multiprocesadores puede ser abordada de diferentes maneras. Un enfoque consiste en considerar aisladamente a cada aplicación, y en base a eso se determina dónde se deben ejecutar sus tareas; otro considera la información global sobre el sistema y de las diferentes aplicaciones, para que en base a ello se tomen las decisiones de donde se ejecutarán las tareas.

El problema de asignación de tareas, por ser NP-completo, es difícil de resolver usando métodos exactos. Los métodos exactos permiten siempre encontrar una solución óptima, pero con tiempos de cálculo muy grandes. A partir de ciertos valores de los parámetros de este problema (por ejemplo, para n=100 o $P \geq 2$), la resolución por estos tipos de métodos, se hace imposible. Uno se debe contentar con soluciones aproximadas. Así, los estudios realizados hasta ahora han sido orientados hacia estrategias que permiten obtener buenas asignaciones (soluciones subóptimas), a veces óptimas, en un tiempo de cálculo reducido. Estos métodos son conocidos como métodos heurísticos o aproximativos. A continuación se presentan dos tipos de clasificación de los métodos aproximativos. La primera clasificación es inspirada por la teoría en la que se basan los métodos de resolución.

a) *Basados en la Teoría de grafos:* estos métodos usan grafos para modelar el problema (los programas, los procesadores y sus enlaces, etc.) y aplican algoritmos basados en la teoría de grafos (corte mínimo-flujo máximo, etc.) para encontrar la asignación. Normalmente, se han usado grafos dirigidos para representar los programas (los nodos representan las tareas y los arcos las comunicaciones entre ellas) y no dirigidos para representar los sistemas computacionales (los nodos representan los sitios o procesadores y los arcos los enlaces de comunicación entre ellos). Por su simplicidad, son muy usados.
b) *Basados en Análisis Empíricos e Intuiciones:* consisten en agrupar las tareas tomando en cuenta las restricciones del problema, hasta tener un cierto número de ellas en cada grupo. Generalmente, estos métodos permiten tener soluciones rápidas, quizás óptimas. Se hacen suposiciones sobre el sistema y programas, y a partir de ellas se

desarrollan heurísticas de resolución. Normalmente, las hipótesis reducen la complejidad del problema, pero se llegan a proponer soluciones no-óptimas.

c) *Basados en la Programación Matemática:* estos métodos formulan el problema de asignación de tareas como un problema de optimización combinatoria y lo resuelven usando técnicas de programación matemática. La idea consiste en definir una función de costo que constituirá la función a optimizar. Entre las técnicas de programación matemática que se pueden usar, están las técnicas de programación dinámica y los procedimientos de separación y evaluación.

La segunda clasificación está basada en el proceso de búsqueda que sigue el algoritmo para obtener la solución:

a) *Algoritmos Iterativos:* son métodos que parten de una solución inicial completa y tratan de mejorarla. Un ejemplo de estos métodos es el *Recocido Simulado* (simulated annealing en ingles).

b) *Algoritmos Constructivos:* son métodos que parten de soluciones iniciales parciales y buscan completarlas. Este procedimiento se detiene cuando la solución es completa. La mayoría de las técnicas empíricas e intuitivas son de este tipo.

Otras técnicas se han desarrollado para resolver este problema, algunas basadas en las redes neuronales artificiales, la computación evolutiva, etc. En general, en el diseño de un algoritmo para resolver el problema de asignación de tareas se pueden mezclar diferentes métodos. La escogencia de un método depende esencialmente de la exactitud deseada y de la rapidez con que se quiera obtener la asignación. Los métodos de asignación de tareas se pueden clasificar como [14, 15, 30]:

- *Estáticos o Dinámicos*, según el momento en que se toma la decisión de donde asignar las tareas. La asignación estática se hace al inicio de la ejecución del programa, y normalmente no cambiará durante la ejecución del mismo. La dinámica consiste en que las asignaciones de las tareas se van haciendo durante la ejecución de la aplicación, en función de la información presente, sobre el estado actual del sistema. Normalmente, el algoritmo de asignación dinámico es distribuido y heurístico, y se ejecuta de manera concurrente con el programa del usuario. Además, si las tareas pueden cambiar de procesador durante su ejecución, ocurre un proceso llamado *migración de tareas*. La asignación dinámica ofrece una gran flexibilidad para manejar tareas heterogéneas o modelos de cálculos dinámicos. Sin embargo, se generan importantes costos de gestión. En el caso de migraciones, puede suceder el problema de que si las tareas migran muy seguido, éstas dejan de hacer cálculo útil por estar migrando. Además, hay que elaborar mecanismos eficientes de reexpedición de mensajes para las tareas que migran. En general, estos últimos algoritmos permiten un mejor balance de la carga.
- *Determinísticos o Heurísticos*: Los determinísticos son adecuados, cuando se tiene toda la información sobre el sistema, por lo que se podrían definir todas las posibles soluciones para escoger la mejor. En el otro extremo están los sistemas en donde la carga es totalmente impredecible. En este caso se requieren heurísticas para obtener una solución razonable.

- *Centralizados o Distribuidos*: Se refiere a cómo se realiza la recolección de la información y donde se ejecuta el algoritmo de asignación. En el caso en que la recolección de toda la información se hace en un determinado sitio se elimina un sobrecosto sobre el sistema, permite tomar una decisión centralizada, pero es menos robusta y coloca una carga pesada sobre dicho procesador. También existe el problema de baja tolerancia a fallas de esa plataforma.
- *Optimos o no óptimos:* La decisión es saber si se quiere encontrar la mejor asignación o sólo una que sea aceptable. Para obtener las soluciones óptimas se requiere de más información y mayor tiempo para procesarla. Los algoritmos óptimos resuelven casos específicos del problema de asignación y normalmente el principio reposa sobre una exploración de todas las posible soluciones. Estos algoritmos son muy costosos. Los algoritmos no óptimos conducen a una solución aproximada.
- *Locales o globales:* Este aspecto tiene que ver con las políticas de toma de decisión. Las opciones consisten en basar la decisión en información local o no. En el caso local, se basa en que si la carga de la máquina está por debajo de cierto umbral, se conserva el nuevo proceso. En el caso global, se basa en recolectar toda la información sobre el sistema, y en base a eso, determinar si una máquina está cargada o no. Los algoritmos locales son muy sencillos, pero muchas veces están muy lejos del óptimo. Por otro lado, en el caso global generalmente se usa un esquema "centralizado".

Otros aspectos claves a considerar al implantar un algoritmo de asignación son:

- ¿Cómo determinar su carga de trabajo?, por ejemplo:
  - Contando el número de procesos en cada máquina.
  - Calculando la fracción de tiempo que el CPU esta ocupado.
- ¿Cómo recolectar la información y mover los procesos de aquí para allá?.
- ¿Cuál es la estabilidad de la decisión tomada?. En pocas palabras, si al cambiar permanentemente los estados de las máquinas, las decisiones que se toman en un momento dado, pueden dejar de ser válidas un momento después.

Un ejemplo de algoritmo de asignación determinista basada en la teoría de grafos es el siguiente: supongamos que se conocen los requerimientos de comunicación entre parejas de tareas. El objetivo que se persigue es asignar las tareas en forma tal que se minimice el tráfico en el sistema y se repartan las tareas equitativamente entre los diferentes procesadores. Para crear el grafo de tareas, cada nodo es una tarea y los arcos representan el flujo de mensajes entre dos tareas. En general, se debe partir el grafo en tantos subgrafos como procesadores existan en el sistema. Los arcos entre subgrafos representaran el tráfico en el sistema, así que el objetivo es encontrar la partición que minimice dicho tráfico, pero a su vez, que asigne un número más o menos igual de procesos en los procesadores (ver figura 3.13). En este caso, la partición óptima es el de la de la figura 3.13.b ya que su costo de comunicación es menor (7) que el costo de comunicación de la partición de la figura 3.13.a (10) y mantiene un número casi igual de procesos en cada uno de los procesadores.

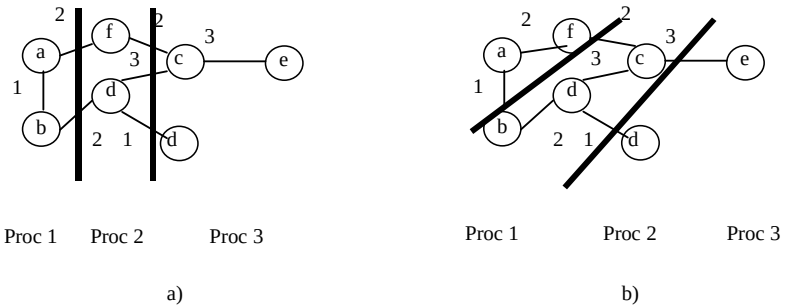


Figura 3.13. Asignación de Tareas

Un caso particular de asignación ocurre en las redes de interconexión estáticas (hipercúbicas, etc.). En estos casos, se usa el término incrustar/fijar (embedding) para describir la asignación de las tareas sobre los procesadores. Dicha asignación consiste en colocar los nodos de un grafo, que representa la aplicación, sobre los nodos de otro grafo, que representa la topología de interconexión de la arquitectura computacional. Un incrustamiento es perfecto, si cada arco del primer grafo puede ser plasmado directamente en un arco del otro grafo. Cuando una asignación de tareas puede ser colocada perfectamente sobre los nodos que describen la plataforma computacional, eso conlleva a reducciones de tiempo de comunicaciones ya que los mensajes pueden alcanzar sus destinos a través de enlaces que minimizan dichos tiempos (eso no significa que se trate de una asignación que minimice los costos comunicacionales, ya que los mismos quizás podrían ser reducidos asignando ciertas tareas en el mismo procesador, según ciertos criterios que veremos más adelante). La calidad de un incrustamiento es medida en términos del número máximo de enlaces del sistema que corresponden a un enlace de la aplicación. Un ejemplo de asignación de un árbol binario sobre un red tipo malla es mostrada en la figura 3.14. El nodo *A* es un nodo adicional que hubo que colocar para poder mantener la estructura del árbol binario al asignarlo sobre la red.

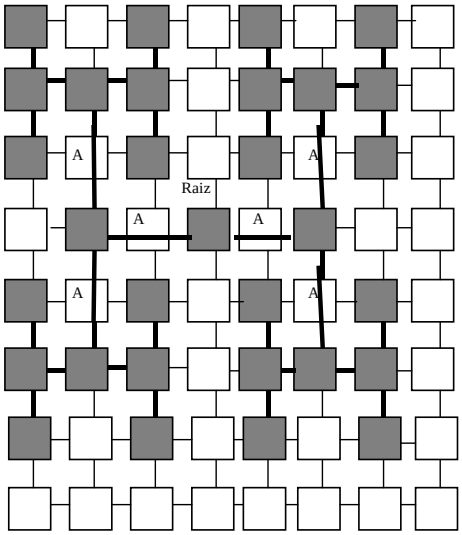


Figura 3.14. Asignación de un Arbol Binario sobre una Red tipo Malla

3.2.1. Definición de una Función de Costo General

Una asignación puede caracterizarse por una función de costo que permita medir su calidad. Si utilizamos una función de costo, la asignación optima será la que minimice dicha función. Los costos describen las características de cada uno de los elementos del sistema, así como de las tareas que componen los programas. Estos costos constituyen los criterios que deben ser optimizados por la asignación final. Clásicamente, las funciones de costo han sido definidas por dos criterios: el costo de comunicación y el costo de ejecución de las tareas. Esta definición no toma en cuenta todos los aspectos presentes a la hora de querer mejorar los rendimientos en los Sistemas Distribuidos/Paralelos, tales como el equilibrio de la carga entre los procesadores, etc. Durante el diseño de una función de costo, todas las características que definen a los sistemas y a los programas deben ser consideradas, sin perder de vista que esta función debe ser fácil de evaluar.

- *Costo de Ejecución (C_E):* Este costo depende de la complejidad de la tarea a ejecutar y de la capacidad de cálculo del procesador donde la tarea reside. El costo total de ejecución de un programa es definido, como la suma de los costos de ejecución de cada tarea del programa, sobre los procesadores del sistema donde residen.

- *Costo de Comunicación (C_C):* Este costo es considerado como el criterio más importante a optimizar en un Sistema Distribuido. El depende de la cantidad de información a intercambiar entre las tareas, de la topología del sistema y de las características de los canales de comunicación. Este costo existe a partir del momento en que dos tareas que se encuentran en procesadores diferentes, desean comunicarse.

- *Costo de Referencia a un Archivo (C_A):* Es el costo que se origina cuando una tarea *i* referencia a datos contenidos en un archivo *f*, el cual reside en un procesador diferente a donde reside la tarea *i*. Este costo esta íntimamente ligado al problema de asignación de datos.

- *Costo de Interferencia (C_I):* Es el costo que se incurre cuando dos o más tareas, las cuales pueden ser ejecutadas en paralelo, residen en el mismo procesador, por lo que deben compartir los recursos. También es conocido como el costo por perdida de paralelismo. Puede ser atribuido a dos factores:

    - La competencia por usar los componentes del procesador: CPU, memoria, etc.
    - La competencia por usar los servicios de comunicación del procesador.

- *Costo por el Desequilibrio de la Carga de Trabajo (C_D):* Una buena asignación debe asegurar el uso mínimo, pero a su misma vez equitativo, de los recursos del sistema. Como en una buena asignación se deben considerar los dos objetivos, nosotros debemos definir un costo que asegure una distribución equilibrada de la carga de trabajo entre los procesadores del sistema. Este costo depende de la complejidad de las tareas a ejecutarse y de las capacidades de procesamiento de los procesadores.

Tomando en cuenta los diferentes costos definidos en la sección anterior, se puede definir la siguiente función de costo general:

$$F = A_1C_E + A_2C_C + A_3C_I + A_4C_D + A_5C_A$$

donde A_1, A_2, A_3, A_4 y A_5 son constantes que definen las relaciones y el grado de importancia de los costos. Igualmente, ellos permiten normalizar las unidades en la función de costo. Con esta función, se puede definir el problema de asignación óptima como:

$$\text{MIN}(F)$$

La minimización de esa función de costo convierte al problema en NP-completo, ya que la idea es obtener la asignación que minimice dicha función de costo.

3.2.2. Diferentes Restricciones en los Sistemas Distribuidos/Paralelos

Las características de un sistema imponen ciertas restricciones que son necesarias considerar para obtener una asignación realizable. Estas restricciones son de dos tipos, las que dependen de las características de la arquitectura (CPU, memoria, etc.) y las que dependen de los rendimientos deseados (plazos de tiempo, grado de paralelismo, etc.). Algunas de ellas son [5, 13, 17, 25]:

a) *Restricciones de Memoria:* Esta restricción hace referencia a la capacidad de la memoria en cada procesador. La suma de los espacios de memoria requeridos por las tareas residentes sobre un procesador debe ser inferior a la capacidad máxima en dicho procesador.
b) *Plazos de Tiempo:* Esta restricción es utilizada en los sistemas en tiempo real, ya que el tiempo de respuesta es el criterio de rendimiento más importante. Esta restricción impone que el tiempo de ejecución de una tarea *i* sobre un procesador *p* sea inferior o igual al tiempo de respuesta límite permitido para esa tarea.
c) *Relación Tarea-Procesador:* Es una matriz lógica que define si una tarea *i* debe/puede ser asignada sobre un procesador *p*. Hay varias razones para definir si una tarea *i* debe/puede ser asignada a un procesador *p*. Por ejemplo, una tarea puede necesitar un recurso que no se encuentra en todos los procesadores.

La asignación de tareas depende directamente de las restricciones consideradas, ya que estas influyen sobre el espacio de soluciones. Dos tipos de problemas pueden surgir en cuanto a la escogencia de las restricciones, soluciones interesantes son eliminadas o soluciones que son poco interesantes son consideradas. Ambas situaciones desmejoran o alargan el proceso de búsqueda de la solución óptima.

3.2.3. Ejemplos de Uso de la Función de Costo sobre Diferentes Arquitecturas Distribuidas/Paralelas

La arquitectura de base es un sistema tipo MIMD con memoria distribuida.

a) Procesadores idénticos usando un bus compartido para comunicarse: Por ser los procesadores homogéneos, C_E no interviene en este tipo de arquitectura. Por el contrario, los problemas de equilibrio de la carga, costo de comunicación, costo de referencia a archivos y costo de interferencia son importantes.

$$F = C_C + C_D + C_A + C_I$$

b) Procesadores Homogéneos sin que estén completamente interconectados: En este caso, C_E continúa sin intervenir. En el cálculo de C_C, C_I y C_A se debe considerar la topología de interconexión entre los procesadores para reflejar el problema de enrutamiento de la información.

c) Procesadores Heterogéneos usando un bus compartido para comunicarse: En este caso, se debe utilizar la función de costo general de *F* definida en la sección 3.2.1, considerándose la heterogeneidad entre los procesadores al evaluar cada costo.

d) Procesadores Heterogéneos sin que estén completamente interconectados: En este caso, se debe utilizar la función de costo general de *F* definida en la sección 3.2.1, considerándose la topología de interconexión entre los procesadores para reflejar el problema de enrutamiento de la información.

3.3 Asignación de Datos

Un importante tema en el manejo y diseño de sistemas distribuidos/paralelos es el diseño del sistema de archivo. En los sistemas distribuidos/paralelos, los archivos son accesados por operaciones de actualización o recuperación de usuarios o aplicaciones, dispersos geográficamente. El sistema de archivo debe resolver problemas tales como la asignación de los archivos, sus niveles de replicación o de fragmentación, etc., para alcanzar óptimos niveles de rendimiento. Aquí revisaremos algunos problemas ligados a esto.

3.3.1 Problema de Asignación de Archivos

El problema de asignación de archivos consiste en que, dado un número de sitios que usan archivos comunes, cómo se deben asignar los mismos para que se optimice el uso de los recursos sobre el sistema [19, 31]. Es decir, es el problema de asignar *f* archivos sobre *n* sitios de manera de optimizar ciertos criterios de rendimiento. El problema está caracterizado por los siguientes criterios:

- La comunicación entre los diferentes sitios en el sistema debe ser mínima.
- La carga en los diferentes sitios debe ser balanceada.

- El tiempo de ejecución total efectivo de las aplicaciones debe ser minimizado.

Esta definición puede adaptarse para el caso de los problemas de fragmentación o replicación de archivos. En cada caso, la asignación óptima que se busca es la de los fragmentos o réplicas. En general, para este problema, se definen cuatro costos:

$$F = C_a + C_m + C_c + C_d$$

donde

C_a: costo de almacenamiento
C_m: costo por actualizaciones
C_c: costo por consultas
C_d: costo por el desequilibrio de la carga de trabajo

3.3.2 Problema de Replicación de Archivos

El problema consiste en que, dado un número de archivos, determinar el número de copias/réplicas por cada archivo y dónde ellas deben ser asignadas para maximizar el rendimiento sobre el sistema [19, 31]. Este problema es típico cuando los usuarios están dispersos y las costos de comunicación son importantes. Este problema depende de varios criterios: costo de comunicación, costo de almacenamiento, costo de actualización, número de requerimientos de actualizaciones, costo del mecanismo de coherencia, relación entre leer y escribir (sí se lee mucho y se escribe poco; tener réplicas es mucho más atractivo que el caso contrario), etc. En este caso el problema de la consistencia entre las diferentes copias de un mismo archivo, en diferentes lugares, es muy importante a considerar. Dicho problema se debe reflejar en la expresión de C_m. Una posible función de costo para este problema seria

$$CF = C_a + C_m + C_c + C_d + C_{no}$$

donde, C_{no} es el costo por el número no óptimo de copias.

3.4 Distribución de Carga de Trabajo

Este problema puede ser definido de la siguiente manera: cómo distribuir las tareas sobre el sistema, dado un conjunto de tareas y un sistema computacional [14, 30, 36]. Es de gran importancia en los sistemas multiprocesadores, donde muchas veces algunos procesadores están fuertemente cargados mientras que otros están ociosos. Si durante la ejecución de un programa, algunos procesadores están ociosos o subutilizados, mientras que otros están sobreutilizados, eso puede llevar a que los tiempos de ejecución de las aplicaciones estén lejos del óptimo. Ciertas ganancias en rendimiento se podrían obtener si se transfiere parte de la carga de trabajo, desde los procesadores sobrecargados a los subcargados. Esta repartición de la potencia de cálculo es lo que se conoce como la distribución de la carga de trabajo. Su principal objetivo es aprovechar toda la potencia

de cálculo en el sistema. Sus principales obstáculos son los costos de comunicación y el tiempo para determinar la carga.

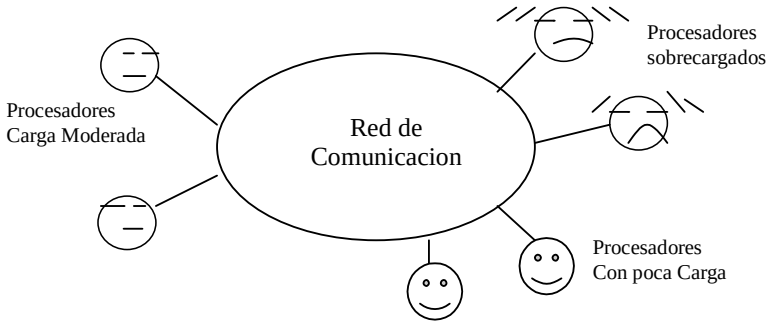


Figura 3.15. Problema de la Distribución de la carga de trabajo

La distribución de la carga puede verse como parte del problema de asignación de tareas en un sistema de multiprocesamiento, en el cual se busca mejorar la utilización de los recursos, y a su vez, minimizar los tiempos de ejecución de las aplicaciones. En general, se busca ganar en procesamiento paralelo guardando todos los procesadores ocupados. La distribución de la carga puede formar parte del Sistema Operativo, de los Compiladores, o de los Lenguajes de programación. La distribución de la carga puede hacerse según dos técnicas [14, 15, 30, 36]:

- *Compartir la carga*: se usa para disminuir las sobrecargas de trabajo, o los períodos de congestión, que aparecen temporalmente sobre ciertos procesadores. Esto se hace transfiriendo parte de la carga de un procesador a otro. De esta manera se garantiza que ningún procesador esté ocioso mientras que otros están sobrecargados. La transferencia ocurre cuando la carga de trabajo sobrepasa un límite dado en cierto sitio. Su objetivo es maximizar el uso de los recursos, compartiéndolos entre todos los procesos del sistema.
- *Equilibrar la carga*: ésta es más específica, se busca permanentemente que las cargas de trabajo entre los diferentes procesadores sean casi idénticas. Así, el equilibrio de la carga busca repartir equitativamente la carga en el sistema. El desplazamiento de procesos se requerirá cada vez que las condiciones globales del sistema cambien. En este caso, el objetivo consiste, en guardar balanceada la carga entre los diferentes procesadores, buscando de esta forma minimizar los tiempos de ejecución de las aplicaciones.

La diferencia entre las dos técnicas quizás no es tan marcada. Donde es más clara dicha diferencia, es en las redes heterogéneas. En el caso del equilibrio de la carga de trabajo, la distribución de la carga se hará en función de la rapidez de cada procesador. En el caso de compartir la carga, dicha técnica se contentará con pasar parte de la carga de trabajo desde un procesador sobrecargado a uno no cargado, sin importarle la potencia en cada procesador. En general, un sistema de distribución de la carga de trabajo debe tener [14, 30, 36]:

- *Políticas de información:* determinan la naturaleza de la información que se requiere, además dónde deben buscarse en el sistema. Poseen *módulos de recolección de la información*, responsables de la búsqueda de información.
- *Políticas de transferencia:* determinan si las condiciones corrientes, son las idóneas para realizar una transferencia de tareas, y seleccionan las tareas a migrar. La distribución de la carga puede ser hecha por la misma aplicación (ella misma define dónde deben ejecutarse sus tareas), la plataforma de ejecución (por ejemplo, PVM determina cómo repartir las tareas que le son sometidas, usando su propio sistema de manejo de tareas), o el sistema operativo.
- *Políticas de asignación:* determina los procesadores donde deben asignarse las tareas.

La distribución de la carga puede ser usada en muchas situaciones, por lo que existen diferentes esquemas para clasificarlas:

a) *A nivel de los objetivos:* qué objetivos se persiguen alcanzar.
- Si son específicos a un dominio o no.
- Si son de uso exclusivo de una aplicación o pueden ser usadas por todo el sistema.
- Si entre sus funciones se deben realizar tareas de particionamiento.

b) *A nivel de implementación*: cuál será la plataforma de programación que se usará para construir el algoritmo de asignación.
- Si las estrategias de monitoreo y mecanismo de decisión forman parte de la aplicación del sistema donde se ejecutan (por ejemplo, PVM) o del Sistema Operativo.
- Si el compilador les dará soporte o no.

c) *Estructura*: cuáles son los elementos a considerar.
- Orientadas a programas paralelos, procesos, hilos, acceso a objetos, acceso a datos, o mezclas de diferentes tipos de entidades (hilos, consultas a Base de Datos, etc.)
- Plataforma de recursos (Servidores de Base de Datos, Sistemas Multiprocesadores, Estaciones de Trabajo, etc.)

d) *Modelo de transferencia de la carga*:
- Espacio de transferencia: si la transferencia es entre un nodo y sus vecinos, o no existen límites a ese nivel.
- Política de transferencia.

e) *Intercambio de información:* además del sobrecosto debido a la transferencia de la carga, la carga de comunicación también crece sobre la red, por los intercambios de los estados en el sistema. Es importante minimizar este costo, pero es también claro que una buena distribución de la carga pasa por tener información suficientemente buena sobre el estado global del sistema.
- Espacio de la información: se refiere a determinar el espacio a que cada sitio envía su información; y puede ser solamente a sus vecinos, a cierto grupo no muy lejano de él, o a toda la red.

- Ambito de la información: es importante definir qué será coleccionado, la información parcial o total del sistema.

f) *Coordinación:* en este caso, se refiere a cómo las decisiones son alcanzadas eficientemente y de manera coordinada en el sistema
- Modo de decisión: si cada sitio puede decidir independiente y autónomamente, si ellos cooperan entre sí, o si se realiza en un modo competitivo.
- Participación: si todos los nodos participan en el proceso de distribuir la carga, lo que puede llevar a procesos de sincronización entre ellos que pueden degradar los rendimientos. La otra forma es una participación parcial, de tal manera que sólo los nodos involucrados participen.

g) *Algoritmo:* las propiedades de los algoritmos de distribución de la carga son semejantes a las propiedades de los algoritmos de planificación y asignación:
- *Proceso de decisión estático o dinámico:* En el caso estático se decide al inicio, y cada procesador ejecuta las tareas asignadas a él. Así, se debe determinar a priori la carga de trabajo que engendrará la aplicación y se ignoran los estados futuros sobre el sistema. Funciona bien si se puede caracterizar la carga de trabajo de las aplicaciones y si no existen muchas fluctuaciones de la carga sobre el sistema. Además, el sobrecosto que se introduce sobre el sistema es mínimo. En el caso dinámico, las decisiones se hacen durante la evolución del sistema, es decir, no se requiere ningún conocimiento antes de la ejecución, y la información que se utiliza es información obtenida durante la ejecución. Además, dichas asignaciones podrán cambiar según la evolución del sistema. Tiene sobrecostos importantes a nivel de sus tiempos de ejecución. Un caso particular de la dinámica es la estrategia adaptativa, en la cual el algoritmo adapta sus parámetros/estrategias de decisión, etc., de acuerdo a los cambios que van sucediendo en el sistema. Una interesante estrategia dentro de este grupo, es la que aprende a mejorar la efectividad del proceso de distribución de la carga.
- *Toma de Decisiones centralizada o descentralizada*: Consiste en conocer quién toma la decisión de localidad; en el caso *centralizado* hay un único procesador que escoge la localidad; normalmente es un procesador dedicado a esa tarea. Sus problemas son el cuello de botella que puede generarse y el bajo nivel de tolerancia a fallas. En el caso *distribuido*, cada procesador puede tomar la decisión. El problema es el tráfico generado para mantener informado a todo el mundo sobre las decisiones locales de cada uno. En general, en el caso *centralizado* se siguen estructuras tipo maestro-esclavo, en el cual un maestro controla la asignación de la carga sobre todos los esclavos. Normalmente, se tiene una cola de tareas en espera de ser ejecutadas, de la cual se van extrayendo las tareas a asignar en cada procesador. La ventaja de este enfoque, es que se comporta bien para tareas de diferentes tamaños, si hay algunas tareas más prioritarias que otras (reordenamiento de la cola), o si una tarea produce nuevas tareas (se insertan en la cola). La forma de extraer los elementos de la cola, puede seguir cualquier esquema de planificación que se defina. Un enfoque sencillo descentralizado consiste en dividir la cola de tareas en subcolas (se crean sub-maestros para controlar cada subcola). Una vez distribuida la carga de tareas, grupos de esclavos trabajan con un sub-maestro en particular. Además, si algún sub-maestro

vacía su cola, puede ayudar a otro sub-maestro. En otro enfoque distribuido, cada procesador tiene su propio mecanismo de selección de procesadores a ayudar. Cuando un procesador no tiene carga, requiere tareas de otro procesador, si éste no le puede proveer, sigue con el siguiente, y así sucesivamente. Otros enfoques de búsqueda de carga de trabajo son factibles de definir.

- *Recolección centralizado o descentralizado de la Información*: una estrategia centralizada usa una entidad única hacia la cual se envía toda la información. Esta entidad central tiene un papel muy sensible en el proceso. En el caso descentralizado, cada cual puede estimar el estado global, o recibir el estado de cada sitio, pero a un costo de comunicación bastante elevado. Esto hace pensar que las soluciones híbridas son las ideales, a través del uso de niveles de reagrupamientos de nodos. Esta característica está muy ligada al proceso de toma de decisiones que se acaba de presentar.
- *Difusión de la información*: en el caso del esquema descentralizado se pueden difundir las informaciones locales como sigue:
  - Estrategias de difusión total
  - Estrategias de difusión parcial
    - Utilizando la noción de vecindad: un nodo envía su estado global solamente a sus vecinos.
    - Utilizando testigos circulantes: contiene el estado global, el cual es actualizado con la información local cada vez que un procesador lo recibe, para después reenviarlo.
    - Utilizando la comunicación entre tareas para incorporar información sobre la carga de trabajo en un nodo.
- *Decisión a la iniciativa del que envía o del que recibe:* determina quién toma la iniciativa para distribuir la carga, ya sea quién va a recibir o quién va a enviar, o combinadas, o a través de un servidor central. Esta es una clasificación especifica a los esquemas dinámicos y distribuidos. A la iniciativa del que envía o activa, es cuando un procesador esta sobrecargado y toma la iniciativa de repartir su carga de trabajo. Esta estrategia no da buenos rendimientos ya que agrega una nueva carga en el procesador ya sobrecargado. Además, los procesadores ociosos deberán esperar hasta que se les asigne carga de trabajo. Si el sistema está muy cargado, las iniciativas del que envía son más costosas (pierden mucho tiempo buscando procesadores con poca carga). En el caso de la iniciativa del que recibe o pasiva, los procesadores ociosos piden que se les dé trabajo, en este caso, molestando a los procesadores activos. En general, estos sistemas tienen muy mal rendimiento en sistemas con poca carga global (ya que los nodos con poca carga inundan la red con solicitudes de trabajo), pero eliminan el sobrecosto cuando todos los procesadores tienen trabajo. El primer tipo no necesita de mecanismos de migración de tareas. Mezclas de ellas son posibles, por ejemplo, un procesador ocioso está en espera hasta que algún procesador esté sobrecargado. Así, el procesador ocioso informa a los otros que está libre, y cuando los otros estén sobrecargados, lo usan directamente. Si la carga es muy grande es mejor la estrategia pasiva, si la carga es débil la estrategia activa es mejor, y si hay grandes fluctuaciones es mejor una mixta.
- *Actualización de la información:* para determinar cuándo se va a enviar la información local desde un procesador, se distinguen dos casos: envío voluntario o a

la demanda. En el primer caso se envían los estados según una decisión local (cada cierto intervalo de tiempo, si el estado cambia mucho, etc.).

- *Sensibilidad del costo*: Si una estrategia guarda balanceada la carga, pero su costo de ejecución es inmenso, la misma deja de ser interesante.
- *Control de estabilidad*: uno de los principales problemas en los que se puede caer, consiste en tener sistemas de distribución de la carga inestables. Esto sucede cuando la carga cambia sustancialmente de tiempo en tiempo. En este caso, puede ocurrir que todos los procesadores estén ocupados en tareas de distribución de la carga sin que progresen los trabajos de los usuarios; y eso se debe evitar. La estabilidad es garantizada si el algoritmo de distribución de la carga guarda estable el sistema bajo todas las condiciones (por ejemplo, un algoritmo puede restringir el número de migraciones de una entidad simple).
- *Reglas de decisión deterministas, probabilísticas o a ciega:* decisiones deterministas dependen del estado actual del sistema. Decisiones probabilísticas dependen de un conjunto de posibilidades generadas de experiencias pasadas. En el caso *a ciega* no se usa información sobre el estado global del sistema, por lo que se pueden asignar las tareas de manera aleatoria. Esta estrategia funciona bien con una estrategia en la cual un procesador no acepta la asignación de una nueva tarea si tiene mucha carga.
- *Determinación del estado global del sistema:* Esto es particularmente importante si el proceso de distribución toma mucho tiempo, por lo que el sistema puede haber cambiado mucho. Esto se resuelve al usar un sistema que al transferir una tarea, también envíe la situación que esperaba encontrar (carga esperada en el sitio destino), y sitios alternativos en caso de cambios bruscos.

Una estrategia simple de distribución de la carga es la *difusión* [36]. La difusión consiste en coleccionar las cargas de los vecinos y si la carga de uno de los vecinos es menor, entonces se le envía cierta parte de la carga local. La cantidad de carga a enviar/recibir depende de la diferencia entre su carga y la carga de sus vecinos. Por ejemplo, en la figura 3.16 se difunden las cargas entre vecinos, tal que las diferencias entre sus cargas no sean mayores a las que cada sitio tenga asignado.

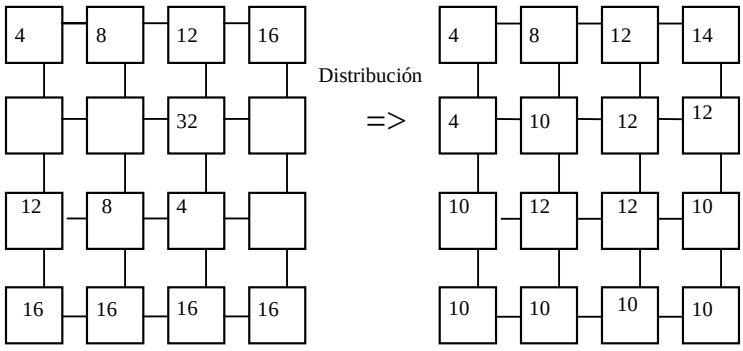


Figura 3.16. Distribución de la carga usando la técnica de *Difusión*

Existen otros métodos posibles, tales como los métodos pares (es decir, dos procesadores, uno cargado y el otro no, trabajan en conjunto para distribuirse la carga), modelo basado en la oferta y demanda (las tareas son consumidores y los procesadores

proponen precios para el consumo de sus tiempos de cálculo, de esta manera la ley de oferta y demanda establece una regulación para el consumo de los procesadores, por parte de las tareas), etc. Antes de incorporar un algoritmo de distribución de la carga, se debe pesar el potencial de reducción en tiempo para completar un trabajo, contra el tiempo requerido para distribuir la carga. Un algoritmo de distribución de la carga debe ser:

- Independiente: no depende de las características físicas del sistema.
- Estable: es la habilidad del algoritmo para evitar soluciones malas u obsoletas en el breve tiempo, en pocas palabras, proveer soluciones persistentes.
- Transparente a los usuarios.
- Tolerante a fallas.
- Poco sensible (adaptarse fácilmente a cambios bruscos en la carga de trabajo).

3.4.1 Equilibrio de la Carga de Trabajo

En el equilibrio de la carga se busca guardar los procesadores equitativamente ocupados todo el tiempo, para que ellos finalicen sus ejecuciones, aproximadamente al mismo tiempo [36]. Ciclos de procesadores pueden ser perdidos si algunos nodos deben esperar que otros terminen. Esta técnica mitiga el efecto de diferencias a nivel de las velocidades de los procesadores. La figura 3.17 muestra los efectos de reducción del tiempo de ejecución, si se usa una técnica que equilibre la carga de trabajo.

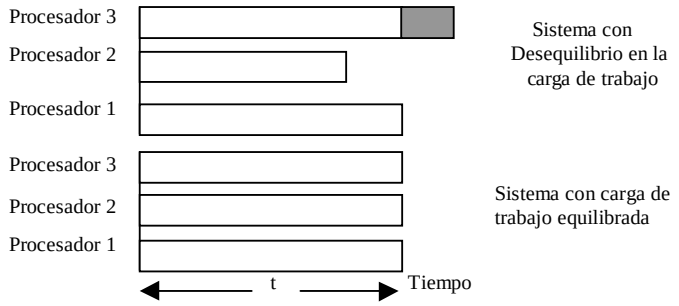


Figura 3.17. Efecto por equilibrar la carga de trabajo

Algunas técnicas de equilibrio de la carga son [15, 36, 40]:

- *Modelos de presión:* en este caso los trabajos son tratados como un fluido que fluye a través de la arquitectura. Un computador tiene una presión interna (trabajos a realizar localmente) y externa (trabajos que hacen otros computadores). El equilibrio de la carga de trabajo consiste en dirigir los trabajos a lo largo de la red a áreas que corresponden a bajas presiones (grupos de computadores con poca carga). En este caso no se requiere información global, un computador sólo debe tener información de sus vecinos y alguna indicación de su propia carga. Cuando un computador alcanza cierto valor crítico, entonces busca cuáles de sus vecinos tienen carga baja para transferirles trabajo. En el tiempo, los trabajo tienden a propagarse a áreas con baja carga. Es parecido a la técnica de Difusión, pero aplicada repetidamente.
- *Algoritmo Round Robin:* Consiste en colocar secuencialmente una tarea sobre cada procesador y regresar al primero una vez recorrido todos.

- *Bisección recursiva*: recursivamente se divide el problema (carga) en subproblemas con más o menos iguales esfuerzos computacionales.

Otros enfoques de equilibrio de la carga de trabajo han sido desarrollados, por ejemplo, al usar estructuras de árboles para repartir la carga de trabajo entre los procesadores, etc.

3.5 Partición de Datos/Programas

El problema de partición de datos y programas consiste en cómo dividir un programa o archivo en componentes que puedan ser ejecutados concurrentemente. Un programa paralelo puede llegar a ser más lento que uno secuencial, principalmente debido a los costos de comunicación y de sincronización que no ocurren en los programas secuenciales. Después que el programa paralelo sobrepasa ese sobrecosto, este provee mejores rendimientos al aumentar el número de procesadores. Por otro lado, la latencia es el tiempo que toma un mensaje para viajar a través de la arquitectura. Reducir la latencia en las plataformas distribuidas/paralelas consiste en usar el multiprocesamiento dentro de cada computador, para guardar los computadores ocupados durante la comunicación. Un óptimo particionamiento de programas/archivos es la base para optimizar tales aspectos.

3.5.1 Descomposición de Programas

Para que una máquina ejecute tareas paralelas, se debe antes que nada exhibir dichas tareas sobre la máquina, es decir, partir el programa paralelo en las respectivas tareas a ejecutar en paralelo. El problema para descomponer un programa en módulos concurrentes, normalmente es conocido, como el problema de definición del tamaño del grano para una aplicación dada. Un grano es una secuencia de instrucciones empaquetadas, las cuales deben ser ejecutadas secuencialmente en un procesador. Si el grano es demasiado grande, el paralelismo es reducido, porque tareas potencialmente concurrentes son agrupadas, y por consiguiente, ejecutadas secuencialmente por un procesador. Si el grano es demasiado fino, mucho sobrecosto debido a cambios de contexto, tiempos de planificación y costos de comunicación, se añade. Este es un problema min-max, que consiste en buscar el equilibrio entre maximizar el paralelismo (grano fino) y minimizar la comunicación (grano grueso). La partición se puede hacer de manera manual (el usuario lo especifica al escribir el código de su programa) o automáticamente (compiladores, sistema operativo, etc.)

Una versión especial del problema de particionamiento de programas es la técnica de "divides y vencerás", la cual consiste en descomponer un problema en subproblemas de manera recursiva, es decir, el nuevo subproblema es de nuevo descompuesto hasta llegar a un punto, donde el actual subproblema no pueda ser más descompuesto. Finalmente, los subproblemas son ejecutados y sus resultados combinados para generar la respuesta final del problema. Esta técnica es usada por muchos programas, tales como el algoritmo de ordenamiento “quicksort”. Una definición recursiva de esta técnica es:

Procedimiento Ejecutar Función(s)
Si el número de elementos en la lista s <= 2
    Ejecutar función sobre dichos elementos
    Regresar resultado
De lo contrario
    Dividir (s, s1, s2)
    Ejecutar Función(s1)
    Ejecutar Función(s2)
    Regresar (resultado ejecutar función para s1 + resultado ejecutar función para s2)

Entre los problemas que pueden ser resueltos, usando esta técnica, tenemos: problemas de ordenamiento, de integración numérica, etc.

3.5.2 Problema de Descomposición de Archivos

Este problema consiste en determinar, dado un número de archivos, cuál es la división óptima de los mismos. Este problema es típico en los sistemas con usuarios dispersos, donde cada usuario o aplicación accesa un fragmento diferente. El rendimiento de una descomposición dada de un archivo, puede ser medido en términos de los tiempos de recuperación por diferentes consultas, la distribución de la carga, y la utilización de los medios de almacenamiento. Este problema incluye un problema de asignación de los fragmentos del archivo recién descompuesto. Una extensión a este problema es la descomposición multidimensional, es decir, el espacio de atributos es dividido y colocado sobre las diferentes localidades físicas.

Un problema particular de particionamiento de datos es la asignación de arreglos, la cual consiste en tomar bloques de columnas o filas de los arreglos, y colocarlos sobre diferentes medios de almacenamiento. Este es una manera de partir estáticamente las operaciones de E/S, y es una repartición de la carga de trabajo entre los procesadores, tal que cada procesador accesa sólo una parte del arreglo.

3.6 Mecanismo de Migración

La migración es requerida en los enfoques dinámicos de asignación para equilibrar la carga. Una primera decisión es la entidad que migrará (migrante), la cual puede ser un objeto activo (procesos, hilos) o pasivo (datos). Otro aspecto es cuando se tiene un límite en el tamaño del objeto que puede migrar o no. Otro aspecto es la heterogeneidad, es decir, si las entidades migrantes pueden ser movidas entre máquinas heterogéneas o no, lo cual es particularmente importante en el caso de migrantes activos. Otro aspecto es cuando se toma en cuenta un requerimiento de migración, si se debe esperar hasta que

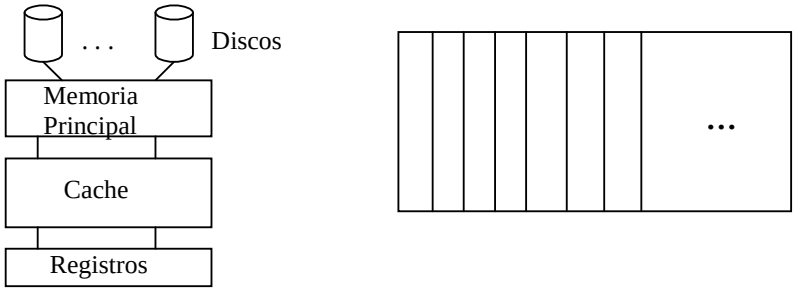
una aplicación dada esté en cierto estado, o cuando se recibe la solicitud se inicia la migración (inmediata). Existen varias políticas que se deben definir [14, 15, 30, 36]:

- *Políticas de localización*: quiénes trabajarán en el proceso de migración.
- *Política de selección:* cuáles son los procesadores ideales para transferirles un proceso (nodos destinos).
- *Política de transferencia*: determina cómo se realiza la transferencia de la información sobre el objeto migrante. El problema es saber hasta cuándo un procesador debe guardar datos o funcionalidades de un proceso que ya migró. Esto se llama *dependencia residual* y ocurre si los migrantes siguen requiriendo servicios del sitio desde donde migró. En general, los datos pueden ser transferidos completamente o parcialmente. En el caso parcial, existen varias estrategias:
  - *Copia Perezosa*: sólo se transfiere inicialmente una mínima cantidad de información. Futuras informaciones que se requieran, deben solicitarse al nodo fuente, para que las transfiera en ese momento.
  - *Copia Perezosa Centralizada*: es similar al anterior, sólo que ahora hay un específico nodo por cada objeto migrante, llamado *nodo casa*, que provee los futuros requerimientos de transferencia de información sobre ese objeto.
  - *Pre-copiado*: en este caso, si un proceso está activo puede continuar con sus cálculos, y si es pasivo puede seguir siendo usado por procesos. El proceso de transferencia es como sigue: al principio se envía toda la información sobre el objeto migrante, al terminar se reenvía la información que se modificó durante el lapso de transferencia, después las partes que se siguieron cambiando, y así hasta llegar a una situación en la que el sitio destino, tenga la información actualizada.
- *Política de Transparencia*: Existen dos tipos de transparencias, transparencia del proceso de migración y transparencia del código fuente. La primera se refiere a si el proceso de migración es transparente para el usuario. La otra se refiere a si se debe modificar el código de las aplicaciones existentes para beneficiarse del proceso de migración.

3.7 Manejo de la Memoria

En general, el propósito de los dispositivos de almacenamiento es guardar programas o datos. En los sistemas computacionales existe una jerarquía a nivel de los dispositivos de almacenamiento: unidades de cintas, unidades de discos, memoria principal, memoria cache y registros. Todos ellos, en su conjunto, forman el sistema de almacenamiento. Este orden está basado en un orden decreciente de capacidades y creciente de velocidades. En el caso particular de la memoria, la figura 3.18 muestra dos maneras para organizarla, horizontal o verticalmente. La primera es la más común (figura 3.18.a), la misma se estructura jerárquicamente desde memorias pequeñas y de rápido acceso, hasta memorias más lentas y grandes. Al fondo están los registros de CPU, que operan tan rápido como el procesador. La idea es guardar en ellos los operandos que están en uso. Después viene la memoria cache que guarda subcontenidos de la memoria principal. La idea es tener en la memoria cache el subconjunto de la memoria principal que se está usando en un momento dado. Después viene la memoria principal. Finalmente vienen los

discos y la memoria virtual (ésta última no es mostrada en la figura 3.18.a). El funcionamiento de los mismos es más lento, pero pueden almacenar una mayor cantidad de información. Por otro lado, la organización vertical facilita el acceso paralelo ya que divide la memoria en varios módulos independientes (ver figura 3.18.b).



a) Horizontal b) Vertical

Figura 3.18. Organización de la Memoria

Usualmente, los procesadores pueden procesar instrucciones más rápido, que leer y guardar la información en la memoria principal. El ciclo de memoria, conocido como el tiempo mínimo requerido para realizar operaciones de lectura o escritura sobre la memoria, es un típico cuello de botella. Aumentar el ancho de banda de la memoria, es una manera para mejorar el rendimiento del sistema. Para resolver este problema varias técnicas han sido usadas. En esta parte estudiaremos algunas de ellas.

3.7.1 Bancos de Memoria

Generalmente, la memoria puede ser dividida en subbloques, llamados *bancos de memoria* [13, 17, 34]. Ellos están conectados a los procesadores a través de una red de interconexión. Cada banco tiene su canal de E/S y cada uno se puede accesar independientemente (ver figura 3.19). El controlador y el generador de direcciones controlan el acceso a los bancos de memoria.

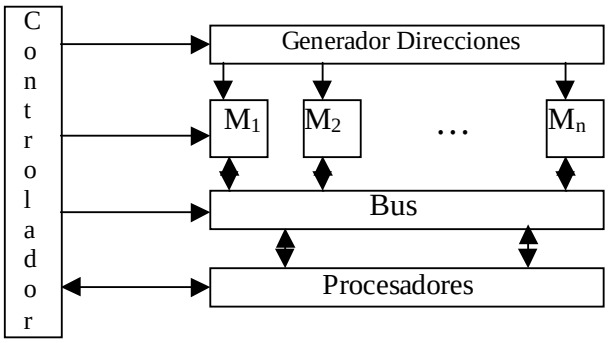
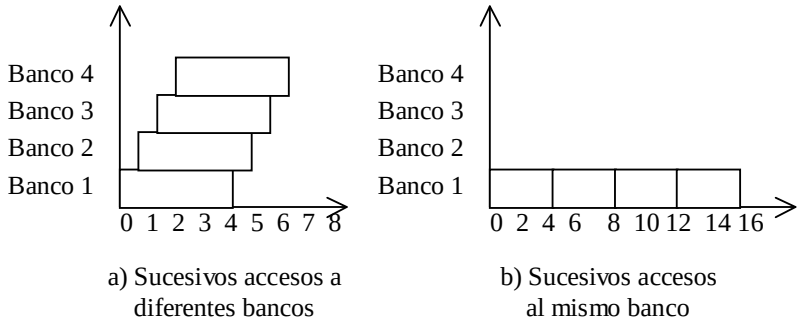


Figura 3.19. Bancos de Memoria

La idea es la siguiente: si la memoria principal se puede estructurar como una colección de módulos de memorias físicas independientes, las operaciones de acceso a memoria pueden proceder en más de un módulo al mismo tiempo. Así, la tasa de transmisiones de palabras promedio (ancho de banda) desde y hacia el sistema de memoria aumenta. Esto ocurre por distribuirse las referencias sobre los módulos, lo que permite que sucesivos accesos puedan ser solapados y una velocidad N puede ser alcanzada en un sistema con N módulos.

Ejemplo 3.2. Suponga un sistema con 4 bancos de memoria. Además, el tiempo de acceso a la memoria es igual a 4 unidades de tiempo. De esta manera, usando bancos de memoria, se pueden realizar 4 accesos consecutivos a diferentes bancos, en los tiempos 0, 1, 2 y 3 (ver figura 3.20.a). Los bancos se activan de manera solapada (el segundo banco se activa un ciclo después del primero, y así sucesivamente el resto). En cambio, si todos los accesos se hacen al mismo banco (ver figura 3.20.b), los mismos deben hacerse secuencialmente, ya que el banco a accesar (banco 1) está ocupado (conflicto de acceso), excepto para el primer acceso.



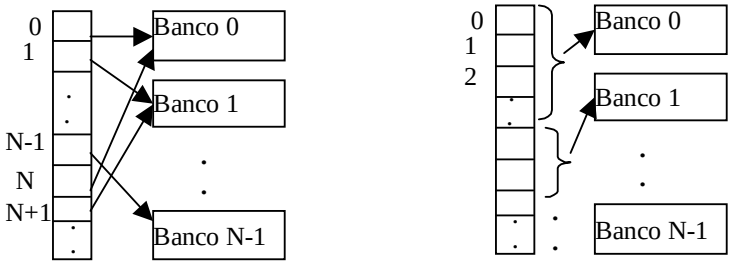
a) Sucesivos accesos a diferentes bancos

b) Sucesivos accesos al mismo banco

Figura 3.20. Eficiencia de los Bancos de Memoria

La definición de las direcciones de almacenamiento en los módulos de memoria juega un papel importante, las mismas pueden ser de bajo-orden o de alto-orden. Consideremos una dirección formada por K bits, tal que K=K1+K2 (ver figura 3.21):

- *Bajo-orden*: los bits de bajo-orden (K2) son usados para identificar el número del banco y los bits de alto-orden (K1) indican la dirección dentro del banco. Este enfoque permite repartir contiguas localidades de memoria a través de contiguos bancos de memoria.
- *Alto-orden:* los bits de alto-orden son usados para identificar el número del banco y los de bajo-orden la dirección dentro del banco. Así, contiguas localidades de memoria son asignadas al mismo banco de memoria.



a) Bajo Orden b) Alto Orden

Figura 3.21. Asignación de datos continuos en los Bancos de Memoria según diferentes esquemas de direccionamiento.

Muchos programas referencian contiguos elementos de vectores en la memoria, por lo que el direccionamiento de bajo-orden, en estos casos, es más interesante para facilitar el acceso a contiguas palabras. Esta técnica puede ser usada, tanto en sistemas uniprocesador como multiprocesadores.

Dividir una memoria en bancos, no asegura un rápido acceso, porque pueden presentarse algunos conflictos, tales como que varios procesadores requieran accesar datos contenidos en el mismo banco de memoria. Por ejemplo, si cada columna de una matriz es guardada en un banco diferente de memoria, diferentes procesadores pueden tener accesos paralelos a los datos por fila o diagonal, pero no a los datos de la misma columna. Entre los conflictos de acceso bajo este esquema organizacional tenemos:

- *Conflictos de bancos ocupados*: Cuando un requerimiento de acceso a memoria es generado a un banco de memoria ocupado, el requerimiento en conflicto será retrasado, hasta que el banco ocupado esté libre.
- *Conflicto de bancos simultáneos*: cuando dos o más procesos en diferentes procesadores (o incluso, en un mismo procesador) intentan accesar el mismo banco de memoria durante el mismo ciclo del reloj, una cola de prioridad se podrá usar para asignar prioridades a los procesos. El de más alta prioridad se procesa primero y el otro debe esperar.

Algunos mecanismos de resolución del problema de conflictos de bancos son:

1. Tener un número grande de bancos de memoria. Esta es una solución muy costosa.
2. Cuando las dimensiones de las matrices son declaradas, se debe añadir una fila o columna para evitar un conflicto de acceso (ver ejemplo 3.3). La matriz nueva, es más grande que la original, pero además, ofrece un mejor rendimiento.

Ejemplo 3.3. Suponga una matriz (4,4), 4 bancos de memoria, y tiempo de acceso a la memoria igual a 4 unidades. Además, los datos son colocados en la memoria siguiendo un patrón por filas. En la figura 3.22.a se muestra en qué banco es colocado cada elemento de la matriz (se crea un conflicto de acceso por columnas), y en la figura 3.22.b la asignación de los elementos, cuando se agrega una columna a la matriz (se elimina el conflicto de acceso por columnas)

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

a) Asignación clásica

0	1	2	3	0
1	2	3	0	1
2	3	0	1	2
3	0	1	2	3

b) Agregar una columna

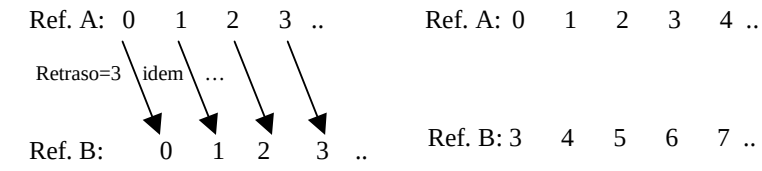
Figura 3.22. Efecto de agregar una columna en una matriz

3. Escoger el mejor banco de inicio de una matriz, es decir, en qué banco será la referencia al primer elemento de una matriz dada. El banco donde será la primera referencia de un arreglo, puede generar conflictos de acceso (ver ejemplo 3.4 y figura 3.23)

Ejemplo 3.4. Suponga dos matrices A y B, 8 bancos de memoria, tiempo de acceso a la memoria de 4 unidades y el siguiente código:

```
Para i=0, n
        B[i]=A[i]*B[i]
```

En este caso la referencia al primer elemento es igual a 0 para ambos arreglos (banco 1), lo que genera un retraso de acceso de 3 unidades cada vez que se trata de accesar a un elemento del arreglo B después de un acceso a un elemento de A (ver figura 3.23.a). En el caso en que se mueva la referencia del primer elemento del vector B al banco 3, el acceso a los elementos de ambos arreglos requerido por el código se puede hacer simultáneamente (ver figura 3.23.b).



a) Banco de Inicio A=B=0 b) Banco de Inicio A=0, B=3

Figura 3.23. Definición del Banco de Inicio

3.7.2 Memorias Cache

Uno de los mayores objetivos de optimización en un sistema de memoria, es cómo diseñar memorias cercanas al procesador, que sean lo suficientemente rápidas y grandes, pero además, económicas [13, 17, 34]. Una de las soluciones más comunes ha sido agregar *memorias cache* entre el procesador y la memoria principal para reducir los tiempos de acceso. Esta solución es posible al reconocer que los programas no accesan los datos aleatoriamente siempre, sino que, normalmente siguen un principio de localidad inherente a los patrones de acceso a los datos y al código.

La idea detrás de las memorias cache es similar a cuando se trabaja con una memoria virtual, aumentar el rendimiento de la memoria principal, en este caso, duplicando áreas de datos en módulos de rápido acceso. La memoria cache es la forma más simple y efectiva para alcanzar altas velocidades dentro de la jerarquía de la memoria a bajo costo. Una memoria cache provee, con alta probabilidad, instrucciones y datos necesarios por el procesador a una tasa que está más en línea con la tasa de demanda del procesador. Así, los retrasos por el uso de la memoria, pueden ser limitados al agregar al sistema una memoria cache a cada procesador. Esta memoria local rápida tiene 2 ventajas: acelera el acceso, y en el caso de las memorias compartidas, disminuye la congestión, debido a procesos que quieren acceder a la memoria común al mismo tiempo. Pero la gestión de la memoria cache, tanto en los sistemas multiprocesadores como en los sistemas uniprocesadores, es bastante compleja. La manera como trabaja es simple, los datos pedidos son buscados primero en la memoria cache, y después, si no están disponibles en ella, en la memoria principal. Además, los datos son movidos de la memoria principal a la memoria cache para futuros accesos. Así, habrá dos valores de la misma variable, por lo que se debe mantener la coherencia de dichos valores en ambos sitios. En el caso de múltiples procesadores, si una variable compartida por varios procesadores existe en diferentes caches, se deberá guardar la coherencia de dicha variable en los diferentes sitios en el sistema.

Físicamente, una memoria cache esta organizada en bloques de datos. Como se dijo antes, cuando un requerimiento es enviado desde el procesador y no existe en la memoria cache, un bloque completo es transferido, desde la memoria principal a la memoria cache.

Esto se llama “fallo de página”, y ocurre cada vez que el dato que se busca no está en la memoria cache. Un ejemplo de organización de la memoria cache es mostrada en la figura 3.24.

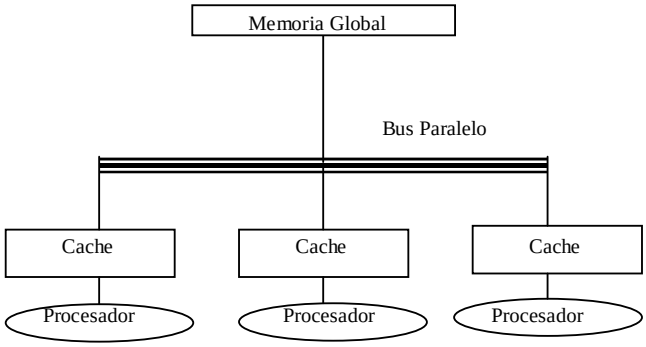


Figura 3.24. Organización de una Memoria Cache en un entorno Distribuido

La tasa de fallas de página es la fracción de referencias que no pueden ser satisfechas por la memoria cache, por lo que deben ser copiadas desde la memoria principal a la memoria cache. Uno de los objetivos de la memoria cache es mantener una baja tasa de fallas. De esta manera, se minimiza el número de veces que un procesador intenta usar el bus de acceso a la memoria principal. De lo contrario, la velocidad del procesador está limitada por el ancho de banda para accesar la memoria principal (en el caso de la *memoria compartida*, por el ancho de banda de la red de interconexión que interconecta las memorias caches con la memoria compartida; en el caso de la *memoria distribuida*, por el costo de la red de interconexión de los procesadores, debido a los mecanismos de coherencia necesarios a implantar).

El mapeo entre la memoria cache y la memoria principal se hace explícitamente: si no se encuentra en la memoria cache cierta información, la referencia se debe recuperar desde la memoria principal. Hay varios esquemas para mapear bloques, entre la memoria principal y la memoria cache:

- *Mapeo directo:* Bloques de la memoria principal se envían directamente a bloques específicos en la memoria cache. Es decir, se mapea cada bloque de la memoria principal en un único bloque de la cache, así existan otros bloques de la cache libres. Un gran número de direcciones se mapean al mismo bloque en la cache, por lo que será fuente de un innecesario número de reemplazos de datos que quizás serán usados en un futuro cercano. Es decir, son forzados los reemplazos entre bloques de la memoria principal que comparten el mismo bloque en la cache, cada vez que alguno lo requiere. Este es el más simple de implementar y el más rápido, sin embargo, es el que produce peores tasas de fallas.
- *Completamente Asociativas:* No se aplica ningún mapeo ya que cualquier bloque de la memoria cache puede ser referenciado por los bloques de la memoria principal. Así, se permite que un bloque de la memoria principal sea mapeado a cualquier bloque de la cache vacío, si alguno existe. Si no hay bloques de la cache vacíos, una *política de reemplazo* debe ser usada para seleccionar el bloque de la cache a ser

reemplazado. Esta es la más difícil de implementar, pero es la que produce mejores tasas de fallas.

- *Conjunto Asociativos*: En este caso, son agrupados los bloques de la memoria cache que pueden ser referenciados por un bloque dado de la memoria principal. Es decir, se asocian bloques de la memoria principal a bloques de la memoria cache. De esta manera, ciertos bloques de la memoria principal comparten bloques de la memoria cache. Este es un compromiso entre las otras dos organizaciones de la memoria cache. El procedimiento que se sigue es el siguiente: un bloque de la memoria principal puede ser mapeado a cualquier bloque dentro de su conjunto de la cache, cuando existe un bloque vacío. De lo contrario, un bloque de la cache dentro de su conjunto será seleccionado para ser reemplazado. En cuanto a su rendimiento (implementación y tasa de fallas), da valores intermedios a los dos anteriores.

La escogencia de alguno de los esquemas tiene un impacto inmenso en el rendimiento y costo de la cache. Algunas organizaciones de la memoria cache, para lograr rápidos acceso a ella, son las siguientes:

- *Cache con Múltiples Niveles*: la cache de mapeo directo es usada en un primer nivel, de manera de reducir los tiempos de acceso a la cache. En adición a la cache de mapeo directo, una cache completamente asociativa es usada en un segundo nivel, para disminuir la tasa de falla sin degradar demasiado el tiempo de acceso. Así, el bajo nivel es más rápido, y el alto nivel es más lento. Esto permite combinar la rápida velocidad de la cache de mapeo directo, pero con tasas altas de fallo de página, con la completamente asociativa que intenta minimizar este último aspecto.
- *Mecanismos Predictivos*: En memorias caches tradicionales, más del 50% de los datos son sacados de la cache antes de que sean usados, por lo que usar mecanismos predictivos para evitar sacar datos de ella antes de que sean usados, es fundamental.
- *Cache que explota la Localidad Temporal*: La principal idea de la cache es tener solamente los datos que son usados. En este caso, la idea es mantener el dato actualmente usado, más los datos recientemente usados. De esta manera, a la cache se le puede añadir un pequeño buzón en el que se guarden los datos clasificados como no temporales, por si ellos se han presumido erróneamente, y de esa forma reducir la penalidad de tal error.
- *Cache que explota la Localidad Espacial*: Esta tradicionalmente explota las técnicas de preenvio y cache sectorizada. Los preenvios que se hacen, son de páginas vecinas a la página actualmente accesada. Esto aumenta el tráfico en el sistema y el almacenamiento de información no interesante dentro de la cache. La cache sectorizada se basa en que no hay suficiente espacio de cache para explotar, por lo que los bloques de la cache se dividen en subbloques (usualmente en 2 o 4) que comparten la misma etiqueta, pero diferentes bits de valides. Así, cuando ocurre un acceso no espacial, sólo una página es llevada a la memoria, reduciendo el tráfico, aunque se pierda espacio en el caso de bloques no espaciales, ya que los subbloques no usados, no pueden ser utilizados para guardar otros datos.

Cuando los datos son modificados en la memoria cache, existen varios mecanismos para actualizar dicha información, en la memoria principal:

a) *Respaldo de escritura*: datos escritos en la memoria cache, se guardan en ella hasta que sean reemplazados, en ese momento es cuando se actualizan en la memoria principal. Es decir, sólo se actualizan en la memoria principal cuando se deben eliminar de la cache.
b) *Inmediata escritura:* datos actualizados en la memoria cache, son inmediatamente copiados en la memoria principal. Así, los datos se escriben simultáneamente en la memoria cache y en la principal. Este enfoque es mejor en el caso de sistema distribuidos a memoria compartida. Como todas las caches ven lo que sucede en la memoria principal, por permanente monitoreo del tráfico en los buses, pueden verificar si un dato que se está actualizando está en sus caches, para actualizarlo o invalidarlo.

Quizás los aspectos más importantes que influyen en los rendimientos sobre una cache son las políticas de coherencia y de reemplazo. Estas son importantes a considerar, ya que la manera como una cache es manejada, tiene una directa influencia en el rendimiento de los algoritmos.

3.7.2.1 Coherencia de Memorias en Sistemas Multiprocesadores

Uno de los grandes problemas, es la coherencia de la memoria (ya sea la memoria principal o la memoria cache), la cual aparece cuando 2 o más computadores tienen en sus memorias, el mismo dato compartido, es decir, cuando múltiples copias de los datos están repartidos a través de las memorias. Las copias son coherentes si ellos tiene igual valor. Si esos datos son leídos, la coherencia es mantenida. Sin embargo, si algún procesador quiere cambiar el valor del dato compartido, entonces las copias se convierten en inconsistentes. Si los datos permanecen inconsistentes (o incoherentes), resultados incorrectos serán propagados en el sistema. Algoritmos de coherencia, son necesarios para mantener el nivel de consistencia a través del sistema. Existen muchos algoritmos, los cuales se basan en dos políticas:

a) *Protocolo Entrometido (más adecuado para mantener la coherencia de la memoria cache):* En este protocolo, la información es distribuida permanentemente, usando alguna de las dos operaciones siguientes:

- *Escribir-Invalidar*: Todos pueden leer, pero sólo uno escribir, en un intervalo de tiempo dado. Cuando algún procesador actualiza un valor, todas las copias son invalidadas en el resto de las memorias donde hay una copia de ésta. Cuando otro procesador quiere leer dicho dato, este debe esperar hasta que su copia en la memoria sea actualizada. En el caso de coherencia en la memoria cache, la copia en la memoria principal puede ser invalidada (se actualiza cuando se reemplaza el dato en la memoria cache) o inmediatamente actualizada.
- *Escribir-Actualizar*: Permite que múltiples escrituras distribuidas se realicen. Además, las actualizaciones son inmediatamente difundidas a los otros que comparten los datos. Esta es más costosa (se requieren buses adicionales), genera más

tráfico y puede actualizar datos que nunca se usaran, sin embargo, no hay que esperar llegar al ciclo de lectura/escritura, sino que se tendrán disponibles los datos antes que se necesiten (evitándose el costoso ciclo de falla de lectura). Aquí, al igual que antes, en el caso de coherencia en la memoria cache la copia en la memoria principal puede ser invalidada o inmediatamente actualizada.

b) *Protocolo Directorio (más adecuado para mantener la coherencia de la memoria principal):* La coherencia de la memoria es controlada en forma centralizada, usando un directorio con una entrada por bloque de memoria, que contiene toda la información para el mantenimiento de la consistencia. Si sobre una memoria se requiere realizar alguna operación, se consulta al sistema centralizado de mantenimiento de la coherencia de la memoria, para saber el estado actual del dato, en la memoria que se quiere accesar. Las diferentes organizaciones del directorio pueden ser:

- Protocolo con directorio que contiene todos los bloques de todas las memorias distribuidas.
- Protocolo con directorios que contienen limitada información (por ejemplo, sólo de un grupo de memorias distribuidas).
- Protocolo con encadenamiento de los directorios locales.

Entre los mecanismos específicos de coherencia, tenemos:

- *Descarga Inmediata*: Cuando un valor es generado, la actualización es hecha sobre un buzón. Así, todas las actualizaciones son coleccionadas en el buzón, y la actualización real es hecha en un momento dado, en todos los sitios.
- *Descarga Perezosa*: es un protocolo de múltiples escrituras. Cada cual registra en una estructura de datos local llamada *diff,* todos los cambios que hace en todos los datos compartidos desde el último punto de actualización global. En cada punto de actualización global, los datos modificados por otros procesos se invalidan en cada procesador. Al primer acceso a un dato invalido, los *diff* son coleccionados y aplicados en el correcto orden causal, para reconstruir la coherencia del dato.
- *Descarga Actualización Automática*: en el anterior esquema, los *diff* se guardan y se envían cuando son solicitados, para ser aplicados en las copias locales para actualizarlas. En este caso, las actualizaciones son enviadas a un procesador *hogar*, quien reconstruye el orden causal entre los *diff*, y desde donde se recupera el dato válido, cuando un procesador lo solicita, evitándose de esta manera los múltiples envíos de *diff* desde diferentes usuarios, cada vez que alguno lo requiera.

Los estados clásicos de los bloques de datos en una memoria cache son:

Modificado: se tiene un dato que ha sido actualizado en él.
Exclusivo: se tiene un dato que ninguna otra cache tiene.
Compartido: hay varias copias de ese dato en diferentes caches.
Invalido: no se tiene una copia válida de ese dato.

Un ejemplo de protocolo de acceso a una memoria cache, como la de la figura 3.24, usando esos estados, es el siguiente:

1. Memoria cache pregunta a la memoria principal por un particular dato que no tiene almacenado.
2. La memoria principal lo regresa en respuesta a un fallo de página, y cambia su estado a *Exclusivo.*
3. Si otra memoria cache solicita a la memoria principal compartida el mismo dato, el requerimiento es interceptado por la primera memoria cache, y lo retorna desde ella. El dato es marcado como *Compartido*, en ambos sitios.
4. Al actualizarse en algún sitio el dato marcado como compartido, esta operación es precedida por un mensaje difundido a todas las otras cache indicando que se *Invalidan* las copias. Finalmente, el estado de este dato se pasa a *Modificado,* y se actualiza en la memoria principal.

3.7.2.2 Políticas de Reemplazo

Una cache tiene un tamaño finito de almacenamiento. Cuando este espacio está lleno, la cache debe escoger un conjunto de objetos (o conjunto de bloques) víctimas para ser sacados y hacer espacio para nuevos objetos o bloques. La escogencia de una política de reemplazo es uno de los aspectos más críticos en el diseño de las memorias caches, ya que tiene un gran impacto en el rendimiento del sistema. El objetivo de la política de reemplazo es hacer el mejor uso de la memoria cache. Dicha política indica, cuándo un nuevo bloque debe ser colocado en la memoria cache, y cuando la memoria cache está llena, cual bloque debe ser removido para hacer espacio a uno nuevo. El bloque a remover debe ser escogido, para asegurar que los bloques que probablemente serán referenciados en el futuro, sean retenidos en la cache. Este problema, al igual que el de coherencia de memorias, también puede presentarse a nivel de la memoria principal. En general, la política de reemplazo debe anticipar las referencias futuras a la cache. El objetivo es disminuir el número de fallos de página en la memoria cache. Las políticas más comunes de reemplazo son:

- *Primero en Entrar-Primero en Salir (First In-First Out, FIFO):* Reemplaza el bloque de la cache que tiene más tiempo en ella. Esta es la más simple, ya que se puede manejar como una cola FIFO. Cuando un reemplazo es necesario, el primer bloque que está en la cola será escogido para reemplazar.
- *Menos Recientemente Usado (Least Recently Used, LRU):* Reemplaza el bloque de la cache que no ha sido usado por más largo tiempo. La premisa básica, es que el bloque que ha sido referenciado en el pasado, probablemente será referenciado de nuevo en el futuro (localidad temporal). Esta política usa un patrón de acceso a memoria, que supone que el bloque que es menos probable a ser accesado en el futuro, es el que menos recientemente ha sido accesado. Esta política trabaja bien cuando hay una alta localidad temporal en las referencias de la actual carga de trabajo.

- *Mas Recientemente Usado (Most Recent Used, MRU):* reemplaza el bloque de la cache que ha sido usado recientemente. Este no es muy usado en los sistemas de memoria cache, porque tiene una mala propiedad de localidad temporal, la cual es típica de los patrones de referencias a la memoria en los procesadores.
- *Menos Frecuentemente Usado (Least Frequently Used, LFU):* está basada en la frecuencia con la cual un bloque es accesado. Esta política requiere que un contador de referencia sea mantenido por cada bloque en la cache. Dicho contador es incrementado en cada referencia a él. La motivación de esta técnica es que el contador puede ser usado como una probabilidad que estima cuando un bloque será referenciado. Cuando un reemplazo es necesario, la política LFU reemplaza el bloque con el más bajo contador de referencia. Una extensión a este enfoque, es la política llamada *LFU-Aging*, esta es debido a que en la política LFU la cache se puede llenar de páginas que eran populares y se convirtieron en impopulares con el tiempo. Estas permanecerán en la cache, conllevando a que nuevas populares páginas con bajo contador al inicio sean reemplazadas. Una forma de solucionar este problema es considerando algún tipo de contador de referencia con edad. El contador es mantenido dinámicamente, y cuando se pasa cierto tiempo en el sistema, todos los contadores son disminuidos en cierta cantidad (o simplemente reinicializados).
- *Magnitud Doble Codiciosa (Greedy Dual Size, GDS):* El algoritmo asigna un costo a cada bloque para calcularle la prioridad de ser removido de la memoria cache. El costo combina la localidad temporal, el tamaño del objeto, su contador de referencias, y otros costos más. Por ejemplo, la versión *GDSlifetime* usa el tiempo que un objeto ha estado en la memoria cache como parte de la información, para calcularle su prioridad (*lifetime/size*). La versión *GDStype* se basa en que cada tipo de aplicación tiene su propio patrón de referencia, usando eso como parte de la información, para calcular la prioridad. Por ejemplo, las páginas web y las aplicaciones que cambian frecuentemente, podrían tener una alta prioridad (*2/size*) de ser reemplazadas, y para el resto de aplicaciones, la prioridad seria más baja (*1/size)*. Otra versión es la *GDSlatency*, la cual usa como parte de la información para calcular la prioridad, la siguiente expresión (*1/latency)*size,* donde la latencia, es el lapso que duró la última recuperación de dicho objeto.

Hay varios criterios para medir el rendimiento de una cache:

- *Tasa de falla:* número de referencias no encontrados en la memoria. Es la medida de eficiencia más popular de una memoria cache.
- *Tiempo para recuperar un objeto:* es fundamental en las caches distribuidas, ya que ese tiempo depende de donde está y del tamaño del objeto (cuando los objetos no son de tamaños fijos).
- *Utilización del CPU, sistema de E/S y red*: fracción del tiempo total del CPU, memoria o red, consumido por la técnica de reemplazo.
- *Tasa de objetos inválidos*: el número de objetos inválidos almacenados en la memoria cache.
- *Tasa de bytes encontrados:* El número de bytes directamente recuperados desde la memoria cache, en función del total de bytes solicitados. Esta medida puede ser

importante, cuando los bloques/objetos no son de tamaños fijos. En Internet, esta medida es interesante porque el ancho de banda de la red es un valioso recurso.

Un procedimiento general de manejo de la memoria cache en plataformas distribuidas y paralelas es:

1. Si falla la búsqueda de la referencia y la memoria cache está llena
   1.1 Aplicar un mecanismo de reemplazo y de coherencia.
2 Si la referencia es un éxito
   2.1 Aplicar un mecanismo de coherencia.
3 Si falla la búsqueda de la referencia y la memoria cache está vacía
   3.1 Asignar un nuevo bloque en un espacio libre de la cache.
   3.2 Aplicar un mecanismo de coherencia.

3.8 Tolerancia a Fallas

Un sistema falla cuando no sigue sus especificaciones funcionales. En ciertos casos, una falla no es muy importante, pero en otros casos puede ser catastrófica (por ejemplo, en un sistema de control aéreo). Como los computadores se han convertido en elementos críticos, existe una creciente necesidad de considerar sistemas donde algunos de sus componentes fallen (memoria, CPU, unidades de E/S, sistema operativo, programas de usuarios, etc.). La tolerancia a fallas es la mejor garantía para resolver este problema. Hay varios enfoques de diseño de sistemas tolerantes a fallas. En general, las técnicas basadas en el hardware proveen un mejor rendimiento pero aumentan los costos. Las técnicas basadas en el software son más flexibles porque pueden modificarse constantemente.

Una falla puede ser por un error de diseño, de programación, de fabricación, de operación, etc. No todas las fallas de un componente generan una falla total en el sistema. Una falla puede ser:

- *Transitiva*: aparece y después desaparece, por lo que al repetirse la operación que falló, puede que la segunda vez se ejecute correctamente.
- *Intermitente*: fallas que se repiten cada cierto tiempo.
- *Permanente:* es una falla que continua hasta que sea reparada.

El objetivo de diseñar y construir sistemas tolerantes a fallas, es asegurar que el sistema pueda continuar funcionando correctamente, a pesar de la presencia de fallas. En el caso de los procesadores, dos tipos de fallas pueden ocurrir:

- *Fallas silenciosas*: Un procesador se para y no responde a nuevas entradas.
- *Fallas bizantinas*: En este caso el procesador continua funcionando, dando malas respuestas, e incluso, pudiendo trabajar maliciosamente dando la impresión de que trabaja correctamente.

En el contexto de tolerancia a fallas en sistemas distribuidos, un sistema que tiene la propiedad de responder siempre a un mensaje en un tiempo finito se dice que trabaja sincrónicamente. Un sistema sin esa propiedad es asincrónico. Es más difícil hacer tolerante a fallas a los sistemas asincronos ya que se debe determinar si realmente hay una falla en el sistema o si es lento en responder. El enfoque clásico para hacer tolerante a fallas a un sistema es la redundancia. Tres clases de redundancia son posibles:

- *Redundancia de información*: En este caso, extra bits son añadidos para permitir la recuperacion. Por ejemplo, un mecanismo de chequeo de sumas puede ser añadido para la transmisión de datos, de tal forma de recuperarse de ruidos en la línea de transmisión. También, copias de datos pueden guardarse en diferentes sitios. Este último caso es conocido como redundancia espacial.
- *Redundancia de tiempo:* Una acción se ejecuta, y si se requiere, se vuelve a ejecutar de nuevo. También es conocida como redundancia temporal (redundancia en los cálculos). La redundancia temporal normalmente requiere recálculos, por lo que tiene un proceso de recuperacion de fallas lento. Esta es efectiva cuando las fallas son transitivas o intermitentes.
- *Redundancia física:* En estos casos equipos extras son añadidos para hacer posible que el sistema como un todo sea tolerante a malfuncionamientos de sus componentes. Hay varias formas de organizar esos componentes extras:
  - *Réplicas activa:* es cuando varios componentes son usados al mismo tiempo (en paralelo) haciendo lo mismo. Las réplicas activas son muy usadas en la realidad (los cuatro motores de un avión y con tres se puede volar; los dos ojos y orejas de los humanos y con uno de cada uno se puede ver y oír, etc.). En las réplicas activas se requieren protocolos de votación. El mecanismo de votación recibe las salidas de los diferentes componentes y determina la correcta salida. Las salidas se comparan entre ellas, y aquella salida que sea dada por más componentes, será considerada la correcta. Así, se sigue un esquema de votación por consenso, donde una operación es aceptada solamente, si un número dado de copias obtienen el mismo resultado. Un importante aspecto es determinar el número de réplicas necesarias, es decir, qué tan tolerante queremos hacer el sistema. Un sistema *k* tolerante a fallas sobrevivirá a *k* fallas de un mismo componente. Esta técnica puede tolerar fallas de hardware y de software. Una versión de esta técnica, es la *Programación de N-versiones*. La idea básica es correr una versión sustancialmente diferente de una aplicación dada, en cada uno de los componentes replicados. El mecanismo de votación es el mismo.
  - *Respaldos primarios*: Este protocolo consiste en que un servidor es el primario y hace todo el trabajo, pero tiene un respaldo. Si el servidor falla, el respaldo lo sustituye. En la realidad existen muchos ejemplos: los vice-presidentes de organizaciones o estados, los copilotos de aviones, etc. Este esquema tiene dos ventajas con respecto al anterior: son más fáciles de implantar y requieren menos recursos (sólo para el primario y su respaldo, el cual debe estar permanentemente actualizado). La Figura 3.25 muestra un ejemplo de cómo puede trabajar el protocolo de respaldo primario: un cliente envía un mensaje que recibe el servidor, éste envía un mensaje de actualización a su respaldo, el cual responde

haberlo recibido. En algún momento el servidor le responde al cliente, y si este falla, el respaldo será quien le responde.

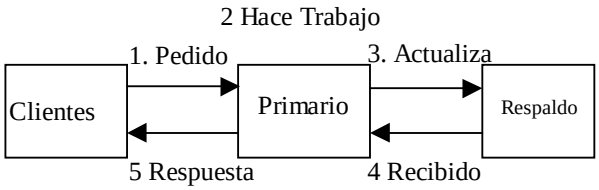


Figura 3.25. Protocolo de Respaldo Primario en la *Operación Escribir*

- *Réplicas pasivas o semi-activas:* es usada para hacer al sistema tolerante a fallas de software. Consiste en que el sistema tiene réplicas de una tarea dada (llamada tarea principal), las cuales no son ejecutadas (réplicas pasivas) o son ejecutadas, pero sus resultados no son tomados en cuenta (réplicas semi-activas). Sólo cuando ocurre una falla en la ejecución de la tarea principal, las réplicas son tomadas en cuenta. En el caso de las réplicas pasivas, es en ese momento que se inicia la ejecución de las réplicas.

Algunas otras técnicas son:

- *Códigos de Error*: Para ciertos elementos del sistema (RAM, buses, etc.), códigos de errores pueden ser eficientes y suficientes para implementar mecanismos de recuperacion.
- *Puntos de Chequeo:* Un punto de chequeo es una copia del estado actual de una aplicación en alguna unidad de almacenamiento, la cual es inmune a fallas. Cuando la falla ocurre, el estado de la aplicación es recuperado, desde el último punto de chequeo para que sea recomenzada la aplicación desde ese punto. Esta técnica es comúnmente usada para recuperaciones de fallas transitivas y permanentes de hardwares, y ciertos tipos de fallas de softwares.
- *Test de Aceptación*: Esta técnica usa múltiples módulos para ejecutar la misma función. Un módulo es un módulo primario, los otros son secundarios. Cuando el módulo primario completa la ejecución, sus salidas son chequeadas por un test de aceptación. Si la salida no es aceptada, un módulo secundario es ejecutado, y así sucesivamente, hasta que alguna salida aceptable sea obtenida o todos los módulos se revisen. Los tests de aceptación pueden tolerar fallas de software, porque los módulos son usualmente implementados usando diferentes enfoques.

Capítulo 4.
Análisis de Dependencia y Técnicas de Transformación

Los lenguajes de programación secuencial, han evolucionado de tal forma, que los programadores se pueden abstraer completamente de las arquitecturas, donde se ejecutarán sus programas. Algunas de las razones son: los progresos hechos a nivel de los compiladores, la unificación de los modelos de arquitecturas secuenciales, etc. La situación es muy diferente en los casos de la programación paralela y distribuida. El programador puede difícilmente ignorar la plataforma, a la hora de escribir su código, si desea que el mismo sea bueno. Aunque existen muchos trabajos en las áreas de compilación de programas paralelos y conversión automática de programas secuenciales en programas paralelos, la paralelización eficiente debe permitir que el programador participe en dicha tarea. En estas áreas, dos aspectos teóricos son fundamentales: el análisis de dependencias y las técnicas de transformación [1, 5, 7, 11, 15, 29, 30, 33, 38, 39]. Este capítulo presenta estos dos aspectos teóricos.

4.1 Generalidades

En este capítulo se presenta un conjunto de técnicas para extraer paralelismo de las aplicaciones. Para explicar esto, comencemos con un simple ejemplo:

Ejemplo 4.1: En el siguiente código:

```
For I = 1, n
        A[I-1] = nuevo[I]
        viejo[I] = A[I]
```

Sí reescribimos este código de la siguiente manera:

```
Parallel For I = 1, n
        A[I-1] = nuevo[I]
        viejo[I] = A[I]
```

ó

```
Parallel For I = 1, n
        A[I-1] = nuevo[I]
```

```
Parallel For I = 1, n
        viejo[I] = A[I]
```

este lazo podría ejecutarse en sólo dos operaciones si tuviéramos *n* procesadores. Pero esta transformación es errónea. Para entender el por qué, debemos comprender cómo ocurre el flujo de datos. En el código original, cada elemento de *A* con índice de *1* a *n* es guardado en *viejo* sin que se pierda su valor, antes de que sea sobreescrito por la nueva asignación de *nuevo* (con índices de *1* a *n*) sobre *A* (con índices de *0* a *n-1*). Estas operaciones de sobreescritura y respaldo de *A* se hacen de manera intercalada, sin perderse la información de *A*. Bajo el esquema de paralelización planteado anteriormente, eventualmente se pierde el valor viejo de *A*. Una manera de garantizar que el comportamiento de la transformación es similar al código original sería:

```
Parallel For I = 1, n
        viejo[I] = A[I]
        A[I-1] = nuevo[I]
```

ó

```
Parallel For I = 1, n
        viejo[I] = A[I]
Parallel For I = 1, n
        A[I-1] = nuevo[I]
```

Aunque esto es fácil de hacer manualmente, lo mismo parece difícil de hacer automáticamente. Lo ideal es tener una herramienta que determine, en el momento de la compilación, todos los cambios a realizar en una aplicación, para explotar el máximo de paralelismo presente en ella. Para realizar esto, dos pasos son necesarios: un análisis de dependencias sobre las diferentes instrucciones que componen el programa, y una fase de transformación del programa basada en la información generada en la fase anterior. Ambos pasos no son triviales.

La *dependencia de datos* refleja la ausencia o presencia de dependencias entre las diferentes instrucciones que componen el programa. Después que las dependencias revelan el paralelismo potencial, *transformaciones* son hechas sobre el código en un intento por generar paralelismo. Aun cuando estamos interesados en paralelizar todas las partes de un programa, el potencial de aceleración por paralelizar los lazos, es tan grande, que le prestaremos una mayor atención (al cambiar la ejecución serial de las iteraciones de cada lazo por la ejecución paralela de todas las iteraciones del lazo, la aceleración puede ser tan grande como el número máximo de iteraciones en el lazo). Las transformaciones más comunes sobre los lazos son: fusión, distribución, intercambio, y torsión. Estas transformaciones son aplicadas independientemente y en orden arbitrario. En general, el éxito de la paralelización automática viene dado por:

- La exactitud y eficiencia de la dependencia de datos.

- La reestructuración del programa para explotar el máximo de paralelismo y mantener la semántica del programa.
- La preservación de la correctitud del programa paralelo que resulta, dado que el programa secuencial era correcto.

4.2 Dependencia de Datos

El análisis de dependencia consiste en determinar si dos o más referencias a memoria, son a la misma localidad y al menos una de ellas escribe en esa localidad, tal que se debe seguir un orden para garantizar el cálculo correcto. Así, la dependencia genera un orden parcial o una relación de precedencia entre las instrucciones de un programa y determina si dos instrucciones dadas son independientes entre ellas, es decir, si ambas se pueden ejecutar en cualquier orden, en particular, en paralelo. La ausencia de dependencias permite definir la reestructuración de los programas, mientras que la presencia de dependencias puede prevenir transformaciones erróneas de los programas.

Además, el análisis de dependencia permite descomponer un programa secuencial en bloques de código que contengan dependencias bien definidas. Posteriormente, estos bloques se pueden convertir en tareas, las cuales se pueden planificar para ser ejecutadas en diferentes procesadores y correr quizás en paralelo.

Las dependencias pueden clasificarse en diferentes tipos [7, 15, 29, 30, 38, 39]. Algunas corresponden a las dependencias entre los datos. Estas dependencias se deben al orden en que el programa accede, tanto en lectura como en escritura, a las diferentes localidades de memoria. Otras se deben al flujo de ejecución de los programas. En general, las dependencias de datos determinan el orden de ejecución de las instrucciones (determina el flujo de los valores escalares a través de un programa) mientras que las de control determinan el conjunto de todos los caminos de ejecución de un programa. Veamos en detalles cada una de esas dependencias comenzando con el ejemplo 4.2.

Ejemplo 4.2. Supongamos el siguiente fragmento de instrucciones de un programa:

S1: A=C-A
S2: A=D+C
S3: B=A+C
S4: A=C

- *Dependencia Verdadera:* Existe este tipo de dependencia entre una instrucción I1 y otra instrucción I2 si I2 se ejecuta después de I1 y I2 accesa en lectura a una localidad de memoria que ha sido modificada por I1. Un ejemplo de esta dependencia es presentado en el ejemplo 4.2 entre S2 y S3 por el valor de *A*. En este caso, el orden en el código entre S2 y S3 deben ser mantenido, de lo contrario, S3 accesará el valor viejo de *A* (el cual, en el ejemplo 4.2 es el calculado por S1, pero en algunos casos puede ser indefinido). Como esto podría modificar el efecto del programa, tales transformaciones son semánticamente inválidas.

- *Antidependencias:* Existe esta dependencia entre una instrucción I1 y otra instrucción I2 si I2 se ejecuta después de I1 y I2 accesa en escritura a una localidad de memoria que ha sido leída por I1. Estas son dependencias que podrían resultar en dependencias verdaderas si el orden de ejecución de las instrucciones es invertido. Un ejemplo de esta dependencia existe entre S3 y S4 por el valor de *A*, ya que S3 usa el actual valor de *A* y S4 lo modifica posteriormente. Si se invierte, S3 usaría un valor erróneo de *A*.
- *Dependencia de Salida:* Estas son dependencias donde dos instrucciones calculan un valor de la misma variable, y si ambas instrucciones son invertidas, resultados erróneos serían obtenidos. Así, existe este tipo de dependencia entre una instrucción I1 y una instrucción I2 si I2 esta después de I1 en el código y I2 modifica la localidad de memoria precedentemente modificada por I1. Este es el caso entre S1 y S2 en el ejemplo 4.2, porque al cambiar el orden de ejecución de esas dos instrucciones lleva a que S3 use un valor erróneo de *A*. El orden entre S1 y S2 debe ser mantenido para preservar la semántica del programa.

  La dependencia verdadera es diferente a las otras dos, en el sentido de que las otras dos dependencias son generadas por la posibilidad de usar y rehusar espacios de memoria a través de variables. Estas dependencias podrían, en principio, ser eliminadas en una forma semánticamente válida, al introducir nuevas variables. Por consiguiente, estas dos últimas, son llamadas dependencias artificiales. Todas estas son dependencias de datos. A continuación presentamos otras dos dependencias.

- *Dependencias de Control:* en un programa, la ejecución de una instrucción puede estar condicionada por el resultado de un test. Cuando la condición *C* controla la ejecución de la instrucción *I* se dice que hay una dependencia de control de *C* hacia I (cuya notación es C $\&^{c}$ I). Más específicamente, una instrucción I tiene una dependencia de control de la instrucción *C* si la ejecución de *C* determina si la instrucción I será ejecutada.

Ejemplo 4.3. Supongamos el siguiente fragmento de un programa:

```
C:      If (X≠0) then
S2:             A=A/X
        Else
S3:             A=A+1
```

Las ejecuciones de S2 o S3 dependen del test realizado en *C*. Este tipo de dependencia puede ser fácilmente eliminado. Esto puede hacerse al crear nuevas variables lógicas, las cuales son verdaderas si las instrucciones son ejecutadas, y falsas en caso contrario. Estas variables se incorporan a cada instrucción (ver ejemplo 4.4).

Ejemplo 4.4. Suponga el siguiente programa:

```
X=A+B
If (X>0)
        X=X+E
Else
    {   X=X+C
        If (D>0)
                X=X+F
        Else
                X=X+D
    }
X=X+G
```

Dicho código puede ser transformado para eliminar la dependencia de control de la siguiente forma,

```
X=A+B
b1= X>0
X=X+E when b1 = true
X=X+C when b1 = false
b2=D>0
X=X+F when b2 = true and b1= false
X=X+D when b1 = b2 = false
X=X+G
```

En general, se requieren más instrucciones para ejecutar el nuevo código. En el código viejo, si X>0 se requieren ejecutar 4 instrucciones, de lo contrario (si X≤0) se deben ejecutar 6 instrucciones. En el nuevo código, más de 6 instrucciones (8) se deben ejecutar. Este el precio a pagar al eliminar la dependencia de control.

Otra forma de estudiar la dependencia de control, es considerar todas las dependencias existentes, en los diferentes caminos de ejecución (ver ejemplo 4.5).

Ejemplo 4.5. Suponga el siguiente código:

```
S1:     A=B+C
C:      If X>0
S2:             A=D+E
        Else
S3:             F=A+2
```

En este ejemplo existen diferentes dependencias según la evaluación de la condición *C*. Así, habrá una dependencia de salida entre S1 y S2 o una dependencia verdadera entre S1 y S3. Generalmente, como la condición no puede ser precalculada antes de la

ejecución del programa, es necesario considerar todas las dependencias potenciales. De esta manera, para ese código habrá dos dependencias.

- *Dependencias de Procedimientos:* Analizar los efectos de las llamadas a un procedimiento o función consiste en determinar cuáles parámetros son cambiados y qué variables globales son usadas o cambiadas, lo cual definirá si la llamada al procedimiento prevendrá la generación de código paralelo. Un método alternativo para manejar las funciones y procedimientos consiste en expandirlos en línea, de tal forma que ciertas transformaciones consideren simultáneamente el código de la llamada como el del programa que lo llama.

Las únicas dependencias que continuaremos estudiando, son las dependencias de datos, ya que son de ellas desde donde más paralelismo se puede extraer. Definamos estas dependencias. Dada una instrucción S, se define al conjunto EN(S) como el conjunto de todas las variables que son usadas por la variable S, y SAL(S) al conjunto de todas las variables a las cuales se les asigna un nuevo valor en S. Para ser más específicos, una instrucción de asignación S consiste del conjunto EN(S) compuesto por las variables usadas del lado derecho, y del conjunto SAL(S) compuesto por la variable usada del lado izquierdo. Con estos conjuntos, podemos redefinir las dependencias de datos, suponiendo un programa que llega a la instrucción S2 a través de S1:

- S2 tiene una dependencia verdadera con S1 (denotada como S1&S2) si $SAL(S1) \cap EN(S2) \neq 0$, es decir, S2 usa la salida de S1.
- S2 tiene una antidependencia con S1 (denotada como $S1\&^{-1}S2$) si $EN(S1) \cap SAL(S2) \neq 0$, es decir, S1 usaría erróneamente la salida de S2 si el orden de ejecución de ellos dos es invertido.
- S1 y S2 tienen una dependencia de salida entre ellos (denotada como $S1\&^{0}S2$) si $SAL(S1) \cap SAL(S2) \neq 0$. Si el orden de ejecución de ambos es invertido, subsecuentes instrucciones que usan la variable de salida modificada por ambos usaran un valor erróneo. Esta última dependencia es importante solo en el caso de que dicha variable sea subsecuentemente usada.

Ellas imponen un orden en el cual las dos instrucciones deben ser ejecutadas. Existe un cuarto tipo de dependencia de dato (*dependencia de entrada*: $EN(S1) \cap EN(S2) \neq 0$) que no es grave, ya que la semántica del programa no es modificada, si se cambia el orden de ejecución de las instrucciones. Por consiguiente, no consideraremos esta dependencia. Sin embargo, ella juega un papel importante a la hora de querer optimizar el manejo de la memoria.

En general, las dependencias anteriores pueden ser representadas por un grafo dirigido G=(N, E), llamado *grafo de dependencia*, en el cual un nodo $N_i \in N$ corresponde a una instrucción y un arco $E_{ij} \in E$ a la dependencia entre dos instrucciones, etiquetado con el tipo especifico de dependencia. Para los códigos sin lazos o recursiones, sus grafos de dependencias son acíclicos. Para los otros casos, el grafo de dependencia puede contener ciclos.

Es necesario distinguir la diferencia entre una instancia de una instrucción (por ejemplo, la ejecución de una instrucción dentro de un lazo para una iteración dada) y la instrucción misma. Puede existir una dependencia entre instrucciones (las clásicas) pero también entre instancias de una instrucción (escondida). De hecho, existe una dependencia entre instrucciones, cuando existe una dependencia entre instancias de ellas. Un *grafo de dependencia entre instancias de una instrucción*, es un grafo cuyos nodos corresponden a una instancia de una instrucción dada y cada arco es una dependencia entre dos instancias. La ejecución secuencial de las instancias, define un orden total sobre el conjunto de instancias. Algunas de las relaciones de orden son arbitrarias, por lo cual podría cambiarse sin afectarse el resultado final. Hay otras relaciones que deben imperativamente conservarse, ya que de lo contrario, se modificaría el resultado, es a esto a lo que se conoce como dependencia entre instancias. El grafo de dependencia entre instancias de instrucciones no contiene ciclos. El grafo presentado en el párrafo anterior es el *grafo de dependencias entre instrucciones*, al cual llamamos simplemente *grafo de dependencia*, el cual consiste en la agrupación de los nodos del *grafo de dependencia entre instancias* que representan instancias de la misma instrucción. Es decir, todos los nodos del grafo de instancia, correspondientes a una misma instrucción, se fusionan en un mismo nodo, en el grafo de dependencias entre instrucciones. Además, este grafo contiene arcos entre dos instrucciones, si el grafo de instancia de instrucciones contiene una dependencia (arco) entre dos instancias de esas instrucciones. Como se dijo antes, un grafo de dependencia entre instrucciones puede contener ciclos.

Ejemplo 4.6. Suponga el siguiente programa:

S1: A= B+C
S2: D=E+F
S3: G=A+C
S4: E=H+C
S5: E=1

El grafo de dependencia es:

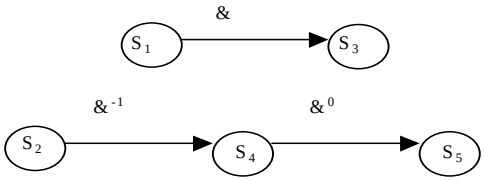


Figura 4.1 Grafo de dependencia del ejemplo 4.6

Ejemplo 4.7. El grafo de dependencia del ejemplo 4.2 es:

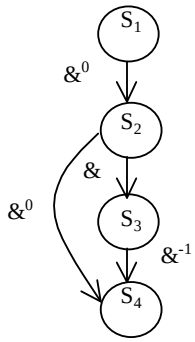


Figura 4.2 Grafo de dependencia del ejemplo 4.2

Ejemplo 4.8. Suponga el siguiente programa:

S1: *A=B+C*
S2: *D=A+E*
S3: *A=A+D*
S4: *If (X<0) then F=D+1*
S5: *If (X>0) then G=F+A*

El grafo de dependencia es mostrado en la figura 4.3. En este grafo solo se muestran las dependencias verdaderas y las dependencias de salida. El grafo muestra una dependencia entre S4 y S5, aunque realmente esto no es cierto, ya que las dos condiciones, son mutuamente excluyentes. La razón es que en la práctica no es posible determinar si las dos condiciones son mutuamente excluyentes. Por eso, es usual suponer que existe una dependencia, de manera de garantizar no cambiar el significado del programa (ver el problema de dependencia de control). Esta suposición es llamada “no fatal”. En contraste, al suponer erróneamente la ausencia de una dependencia, el programa puede ser reestructurado sobre la base de ello, y en consecuencia, generar resultados incorrectos.

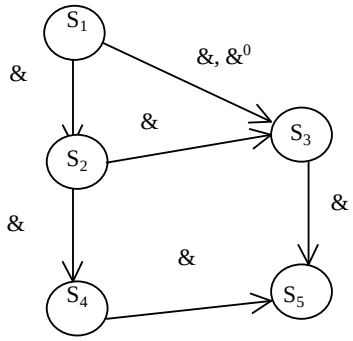


Figura 4.3. Grafo de dependencia del ejemplo 4.8

4.2.1 Análisis Escalar

Se refiere al conjunto de métodos, para realizar un análisis estático en un programa, para generar suposiciones, sobre el uso de las variables escalares en el mismo, durante su tiempo de ejecución [29, 30, 38, 39]. Las variables analizadas pueden incluir arreglos, pero solamente en el sentido de que cada variable con índice, represente a todo el arreglo. El análisis escalar es un prerequisito para el análisis de dependencia de datos. Es necesario para determinar la aplicabilidad y efecto de los métodos de transformación que se discuten más adelante. Veamos un ejemplo.

Ejemplo 4.9. Considere el código siguiente:

S1: A=…
S2: B=…
…
S3: C = A+B
S4: For I=1, C

Las instrucciones S1, S2 y S3 son definiciones de las variables A, B, y C, respectivamente. Al determinar si el valor de C es un valor conocido cuando se usa en el lazo (S4), se podría mejorar significadamente el análisis de dependencia dentro del lazo. Existen varios métodos para eso. La técnica de *propagación de constantes* es uno de ellos, y consiste en determinar el valor de una variable en un punto dado del programa. El ejemplo 4.9 consiste en buscar todas las definiciones de A y B, las cuales alcancen S3. Si nosotros podemos probar que A y B en S1 y S2 son las únicas definiciones que alcanza S3, y si se conocen los valores de A y B, se puede calcular el valor de C (se conocería el límite superior del lazo).

Existen tres escenarios en los que se puede hacer el análisis escalar:

- *Análisis Local*: se basa en bloques básicos, secciones de código que pueden ser accesados sólo a través de la primera instrucción y terminan solamente a través de la última instrucción.
- *Análisis Interprocedimientos*: es ejecutado a nivel interno de los procedimientos.
- *Análisis Completo*: toma todo el programa en cuenta.

4.2.2 Dependencia en los Lazos

El análisis de dependencias entre instrucciones que no tienen lazos es bastante fácil de realizar. Cuando existen lazos e instrucciones que usan variables de tipo arreglo, es difícil de hablar directamente de dependencias entre instrucciones, ya que cada instancia de la instrucción puede, según los valores de los índices del lazo, trabajar sobre localidades de memoria diferentes. Mientras que el análisis de flujo de datos reconoce a las variables con índices como una única variable, el análisis de dependencia de datos debe examinarlos individualmente, es decir, considerar el valor del índice.

Ejemplo 4.10. Considere el siguiente código:

For I=1, 10
*S1: A[2*I]=B[I]+1*
*S2: D[I]=A[2*I+1]*

Si A[2*I] y A[2*I+1] son reconocidas como referencias a *A* (caso del análisis de flujo de datos) se asume la existencia de una dependencia entre S1y S2. Al hacer un análisis de las expresiones de los índices 2*I (siempre par) y 2*I+1 (siempre impar), se puede ver que nunca toman el mismo valor, así, S1 y S2 son independientes. Ahora, son necesarios análisis más detallados, porque aquí aparecen otros tipos de dependencias a considerar.

Suponga que S1 y S2 están en un lazo y se desea saber si hay dependencias de datos entre S1 y S2. En este caso, debemos determinar si para algún par de iteraciones (i_1, i_2) de S1 y S2, respectivamente (es decir, S1(i_1) e S2(i_2)), ellos accesan variables comunes y al menos una de los dos escribe en dicha variable.

Ejemplo 4.11. Considere el ejemplo 4.10. Los conjuntos EN y SAL de S1 y S2 son:

EN(S1(I))={B[I], I} *EN(S2(I))={A[2*I+1], I}*
*SAL(S1(I))={A[2*I]}* *SAL(S2(I))={D[I]}*

Con esos conjuntos podemos hacer un análisis de dependencia usando los mismos formalismos presentados en la sección anterior. Así, existe una relación de dependencia de datos entre instancias de dos instrucciones si S1(i)<<S2(i') ("<<" significa que S1(i) se ejecuta antes que S2(i'), tal que *i* e *i'* son las específicas iteraciones de S1 y S2, respectivamente), y SAL(S1(i))∩EN(S2(i')) o SAL(S1(i))∩SAL(S2(i')) o EN(S1(i))∩SAL(S2(i')). En estos casos, como el valor que causa la dependencia es generado en una iteración del lazo, esta dependencia se llama *dependencia generada por el lazo*. En general, los tres tipos de dependencias son ahora especificados como:

- *Dependencia Verdadera*: (S1(i)&S2(i')): SAL(S1(i))∩EN(S2(i'))
- *Antidependencia* (S1(i)&$^{-1}$ S2(i')): EN(S1(i))∩SAL(S2(i'))
- *Dependencia de Salida* (S1(i)&^{o} S2(i')): SAL(S1(i))∩SAL(S2(i'))

Es necesario definir ciertos conceptos, algunos de los cuales se trataron brevemente en capítulos anteriores [29, 30, 38, 39]. Para eso, consideremos *n* lazos anidados cuyos índices son i_1, …, i_n, partiendo del lazo más externo al más interno.

- *Lazo Normalizado*: cuando tanto el borde inferior, como el paso del lazo, son iguales a 1. Por ejemplo, el lazo *For I=1, n* es normalizado, no así el lazo *For I=2, n, 3*. Muchas técnicas de transformación, detección de paralelismo y análisis de dependencias de datos son más fáciles, si los lazos están normalizados. Existen varios métodos para normalizar un lazo, aquí presentaremos uno.

Definición 4.1. Suponga el siguiente ejemplo:

For I= e1, e2, e3
Cuerpo del lazo

Donde e_i son expresiones. La normalización de ese lazo sería:

For In= 1, (int(e2)-int(e1)+int(e3))/int(e3)
cuerpo del lazo normalizado

Donde $int(e_i)$ es la parte entera del valor e_i.

y en el cuerpo del lazo normalizado se reemplaza cada I por int(e1)+(In-1)*int(e3)

Ejemplo 4.12. Suponga el siguiente programa:

For I=1000, 1, -1
A[I]= …

Al normalizarlo sería:

For In=1, (1-1000+(-1))/(-1))
A[1000+(In-1)(-1)]=…*

Que se puede simplificar como:

For In=1, 1000
A[1001-In]= …

- *Vector de iteración* de una instancia de S es el vector $(i_1, i_2, \ldots, i_n)$ de valores de i_j que toman los *n* índices de los lazos que engloban a S al momento de que esa instancia es ejecutada. De esta manera se identifica a las instancias de una instrucción. Así, si S está encerrada en un lazo *n*-anidado con índices $I_1, \ldots, I_n$, entonces se escribe $S(i_1, \ldots, i_n)$ para referirse a la instancia de S durante la iteración particular $I_1= i_1, \ldots I_n= i_n$.

Ejemplo 4.13. Suponga el siguiente programa:

```
For I=1, n
{       For J = 1, m
S1      {       …
                For K= 1, r
S2:             {       …
S3                      …}
S4              …}}
```

En este caso los vectores de iteración de S1 y S4 son del tipo (i_1, i_2) donde $1 \leq i_1 \leq n$ y $1 \leq i_2 \leq m$, mientras que para S2 y S3 son del tipo (i_1, i_2, i_3), tal que $1 \leq i_3 \leq r$.

Ejemplo 4.14. Suponga el siguiente programa:

```
For I=1, 2
        For J= I+ 1, 4
                For K=J+2, 7, I+1
                        S
```

Un posible vector de iteración es (1, 4, 6). El conjunto de los vectores de iteración para este ejemplo esta compuesto por los siguientes elementos:

{ (1, 2, 4), (1, 2, 6), (1, 3, 5), (1, 3, 7), (1, 4, 6), (2, 3, 5), (2, 4, 6)}

- *Orden textual:* corresponde al orden de aparición de las instrucciones en el texto. Ese orden determina el orden de ejecución en el programa secuencial.
- *Orden lexicográfico:* (denotado $<_{lex}$) es basado en el análisis del vector de iteración. Sean dos instancias de instrucciones S1(i) y S2(i'). Si i' $<_{lex}$ i, S2(i') se ejecuta primero que S1(i) si el paso del lazo es positivo. En caso de que las partes comunes del vector sean iguales, es el orden textual que indica el orden de ejecución. El orden lexicográfico sirve para describir la ejecución secuencial del lazo anidado.

Ejemplo 4.15. Supongamos las instancias de instrucciones S1(1,2), S2(1,2,2), S2(1,2,3,), S3(1,2,3) y S4(1,2) del ejemplo 4.13. S1(1,2) se ejecuta antes que S2(1,2,2), ya que los vectores son lexicográficamente idénticos y S1 precede a S2 en el texto. S2(1,2,2) se ejecuta antes S2(1,2,3) y S3(1,2,3), según el orden lexicográfico. Finalmente, S3(1,2,3) precede S4(1,2), ya que las partes comunes son idénticas y S3 precede a S2 textualmente.

- *Un nido de lazos es perfecto* si todos los lazos de ese nido están anidados los unos en los otros sin instrucciones intermediarias. Es decir, un nido de *n* lazos L={L_1, …L_n} es perfecto, si el cuerpo completo del lazo L_i es el lazo L_{i+1}, para $1 \leq j \leq n-1$. Esta noción es importante ya que muchos mecanismos de transformación sólo se aplican a estos tipos de lazos anidados. Por otro lado, un lazo es *anidado una-via* si hay solo un lazo por cada nivel de anidamiento, aunque pueden existir otras instrucciones, y es *anidado multi-vias* si hay dos o más lazos en un mismo nivel.

Ejemplo 4.16. Suponga el siguiente código:

```
For I=1, n
        For J=2, n
                T[I]=T[I]+1
                A[I,J]=B[I,J]+A[I,J-1]*T[I]
```

Es un lazo perfectamente anidado. Mientras que:

```
L1:  For I=1, n
S1:      T[I]=T[I]+1
L2:      For J = 2, n
S2:              A[I,J]=B[I,J]+A[I,J-1]*T[I]
```

Es un lazo no perfectamente anidado (pero si es una-via), ya que S1 es parte del cuerpo de L1.

- *Vector Distancia*: Para entender la dependencia en un nido de lazos, hay que medir la distancia entre las instrucciones fuentes y destinos de la relación de dependencias. Suponga que S es una instrucción en un lazo anidado y existe dependencia entre las diferentes instancias de ejecución de S en cada iteración. La primera instancia de S ocurre cuando el índice del lazo es i_1, y la segunda cuando el índice del lazo es i_2, tal que la iteración i_1 es la fuente de la dependencia e i_2 es el destino de la dependencia. El vector distancia de esa dependencia es Dis=i_1- i_2. El vector distancia indica el número de iteraciones, entre donde la variable es usada por primera vez y cuando ella es usada por segunda vez (lo cual genera la dependencia). La diferencia entre dos vectores de iteración lexicográficamente ordenados, da siempre un vector distancia, donde el primer componente no nulo es positivo. Si los vectores distancia son los mismos para todas las dependencias del lazo, se habla de una distancia de dependencia constante. Además, si i_1-i_2=0 se habla de una *Dependencia Independiente del Lazo*, de lo contrario, se habla de una *Dependencia Dependiente del Lazo*.

Ejemplo 4.17. Suponga el siguiente código:

```
For I=1, 5
        For J=1,4
S1:             A[I, J]=B[I, J]+C[I, J]
S2              B[I, J+1]=A[I, J]+B[I, J]
```

Para S1, su vector distancia con S2 será <0,0> (por B[I, J]), y para S2, su vector distancia con S1 y S2 serán iguales a <0, 1> (por A[I, J] y B[I, J]).

- El *Vector de Dirección* de una dependencia S1&S2 es el vector de dirección D(i, j) entre el vector de iteración *i* de S1 y el vector de iteración *j* de S2 entre los cuales la dependencia existe. La determinación de este vector consiste en examinar los signos (positivos, nulos o negativos) de los valores del vector distancia. Este vector D=(s_1, …, s_n) es un vector cuyos elementos pertenecen al conjunto {<, =, >}, definidos a partir de Dis=(Dis_1, …, Dis_n) según las condiciones siguientes:

$$s_i = \begin{cases} < \text{ si } Dis_i < 0 \\ = \text{ si } Dis_i = 0 \\ > \text{ si } Dis_i > 0 \end{cases}$$

El vector de direcciones permite caracterizar la validez de ciertas transformaciones de código y especificar la relación entre dos pares de vectores de iteración. El símbolo * representa cualquier tipo de relación entre los correspondientes componentes de dos vectores de iteración. Así, se pueden usar para definir clases de vectores de direcciones. Por ejemplo, (=, <, *) define el conjunto {(=,<,>), (=,<,=), (<,=,<)}

Ejemplo 4.18. Suponga dos vectores de iteración i_1=(10,20,30) e i_2=(20,20,2). Dis(i_1,i_2)=(10,0,-28) y D(i_1,i_2)=(<,=,>).

- *Par de Dependencia*: se define un par de dependencia entre dos instrucciones S1 y S2, caracterizadas por sus vectores de iteración i_1 e i_2, respectivamente, como (Dis(i_1,i_2), W(i_1,i_2)), tal que W(i_1,i_2) es el tamaño del mensaje que S2 recibirá de S1. S2 puede tener más de un par de dependencia con S1.

- *Conjunto de Dependencia*: Es el conjunto de todos los pares de dependencias entre dos instrucciones S1 y S2.

- *Vector de Bordes Superiores*: para un conjunto de *n* lazos anidados, es el vector {b_1, b_2, …, b_n}, donde b_i es el borde superior del lazo en el nivel *i*.

- *Grafo del Espacio de Iteraciones*: es un espacio cartesiano discreto n-dimensional, para *n* lazos anidados, donde cada eje del espacio de iteración corresponde a un lazo (su índice). Cada punto en el espacio representa la ejecución de la instrucción anidada para una iteración de los lazos anidados (es una instancia caracterizada por su vector de iteración). Así, el espacio de iteraciones es una región cartesiana discreta n-dimensional que abarca el conjunto de valores de los índices de las instrucciones generadas por todas las iteraciones de los lazos anidados. La figura 4.4 muestra el espacio de iteraciones para el lazo del ejemplo 4.19.

Ejemplo 4.19. Suponga el programa:

```
For I=1, 4
        For J=I, 4
                A[I, J]=A[I, J]+B[I, J]
```

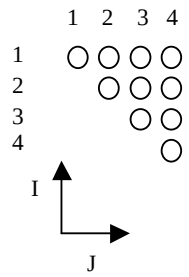


Figura 4.4. Espacio de iteraciones del ejemplo 4.19.

El grafo de dependencia del espacio de iteraciones representa las relaciones de dependencias en el lazo anidado, es decir, entre las diferentes instancias de las instrucciones del nido de lazos. En dicho grafo, hay una flecha desde un punto (i_1, …, i_n) a un punto (j_1, …, j_n) si hay dos instrucciones S1 y S2 en el nido de lazos, tal que S1(i_1, …, i_n)&*S2(j_1, …, j_n), donde * representa cualquier tipo de dependencia. Esto es válido para cualquier par de instrucciones en un lazo. La figura 4.5 muestra esas dependencias con flechas desde el punto correspondiente a la iteración fuente de la dependencia I=(i_1, …, i_n) al punto correspondiente a la iteración destino de la dependencia J=(j_1, …, j_n) para la dependencia entre las instrucciones S[I]&S[J] del ejemplo 4.20.

Ejemplo 4.20. Suponga el programa:

```
For I=1, 5
        For J=1,4
                X[I+1,J+1]=X[I,J]+Y[I,J]
```

El grafo de dependencia en el espacio de iteraciones es:

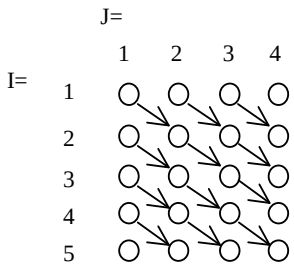


Figura 4.5. Grafo de Dependencia en el Espacio de iteraciones del ejemplo 4.20.

Informalmente, la dirección de la dependencia de datos determina las direcciones de las flechas en el espacio de iteraciones. En términos del espacio de iteraciones, una dirección hacia adelante apunta hacia un valor más grande de índice en alguna de las dimensiones, y una dirección hacia atrás apunta hacia un menor índice en alguna de las dimensiones. Así, se pueden definir una variedad de direcciones de dependencias:

< significa que la dependencia atraviesa un borde de iteración hacia adelante, de I a I+1.
= significa que la dependencia no atraviesa bordes de iteraciones.
> significa que la dependencia atraviesa un borde de iteración hacia atrás, de I a I-1.

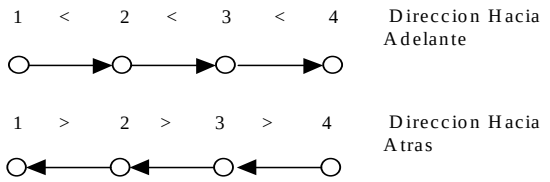


Figura 4.6. Ejemplos de direcciones de dependencias.

- *Grafo de dependencia:* es un grafo dirigido G(V, E), donde V es un conjunto de nodos que corresponden a las instrucciones del programa y E los arcos que representan las dependencias de datos entre las instrucciones. Típicamente, cada arco es etiquetado con su tipo de dependencia.

Ejemplo 4.21. Suponga el programa:

S1: X=Y+1
For I=2, 30
S2: C[I]=X+B[I]
S3: A[I]=C[I-1]+Z
*S4 C[I+1]=B[I]*A[I]*
For J=2, 50
S5 F[I,J]=F[I,J-1]+X
S6: Z=Y+3

Su grafo de dependencia es:

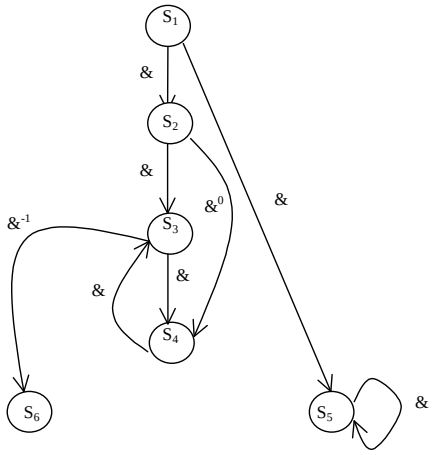


Figura 4.7. Grafo de Dependencia para el ejemplo 4.21.

Al tener definido los diferentes conceptos que caracterizan un lazo, se pueden formalizar las dependencias entre instancias. La clasificación de dependencias basada en la relación entre los conjuntos de entrada y salida, será extendida para considerar ahora

las dependencias usando los vectores de iteración de las instancias. Al considerar las instrucciones S1 y S2 del lazo, con los vectores de iteración *i* y *i'*, respectivamente, tal que S1(i)<<S2(i'), se permite definir las siguientes relaciones:

$S1(i) <<_{\mu} S2(i')$ si μ=Dis(i,i') podemos hablar de la dependencia $S1(i) \&^{\mu} S2(i')$
$S1(i) <<_{c} S2(i')$ si $i <_{c} i'$ podemos hablar de la dependencia $S1(i) \&^{c} S2(i')$
$S1(i) <<_{\infty} S2(i')$ si i = i' y S1*bef* S2 podemos hablar de la dependencia $S1(i) \&^{\infty} S2(i')$

Donde, $i <_{c} i'$ es $D(<_{c})=\{=_{c-1},<,*,\ldots *\}$
S1 *bef* S2 si S1ocurre textualmente antes de S2.

Con esas relaciones, podemos definir:

- $S1(i) \&^{c} S2(i')$ es una *dependencia dependiente del lazo* que esta en el nivel *c* entre S1(i) y S2(i'). Informalmente, existe esta dependencia entre S1 y S2 si las dependencias entre ellas ocurren en diferentes iteraciones del lazo.
- $S1(i) \&^{\infty} S2(i')$ es una *dependencia independiente del lazo* entre S1(i) y S2(i'). Informalmente, existe esta dependencia entre S1 y S2 si las instrucciones no se ejecutan en un lazo o si las dependencias entre ellas ocurren en la misma iteración del lazo.

Ahora, se definen formalmente estas dependencias:

- La instrucción S2 tiene una *dependencia dependiente del lazo* con la instrucción S1 si:
  - S1 referencia a una localidad de memoria en la iteración i y S2 referencia a la misma localidad de memoria en la iteración i'.
  - D(i, i') contiene al menos un componente diferente de =, el cual corresponde al símbolo < y debe estar más a la izquierda de cualquier símbolo > que aparezca.

  Es decir, esta dependencia existe, si ella depende de los índices de los lazos que las engloban, y los datos deben ser pasados entre instrucciones ejecutadas en diferentes iteraciones. Una dependencia de este tipo prohibe la ejecución paralela del lazo en cuestión.

- La instrucción S2 tiene una *dependencia independiente del lazo* con la instrucción S1 si:

  - S1 referencia a una localidad de memoria en la iteración i y S2 referencia a la misma localidad de memoria en la iteración i'.
  - Cada componente de D(i,i') es igual a =.
  - S2 va después que S1 textualmente en el programa.

Esta dependencia existe aun si se ignoran los lazos que los engloban, ya que corresponde a datos que son pasados entre instrucciones en una misma iteración.

La diferencia esencial entre los dos tipos de dependencias, es que en el primer tipo las referencias a la memoria ocurren en diferentes iteraciones, mientras que en la otra ocurren en la misma iteración.

Ejemplo 4.22. Suponga el programa:

```
For I=1, n
S1:    A[I]=A[I]+10
S2:    B[I]=A[I-1]*B[I]
```

En este ejemplo existe una *dependencia generada por el lazo* para las iteraciones *i* e *i-1*, tal que S1(i-1)&S2(i). No así para S1(i) y S2(i).

Ejemplo 4.23. Suponga el programa:

```
For I=1, 3
S1:    A[I]=B[I-1]
S2:    B[I]=A[I-1]
S3:    B[I]=D[I]
```

cuyo grafo de dependencia es:

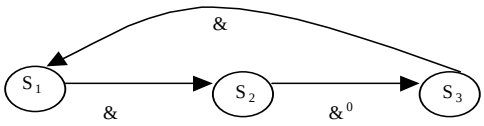


Figura 4.8. Grafo de Dependencia del ejemplo 4.23.

Al desenrollar el lazo se obtienen las instancias de las instrucciones. Con ellas se pueden establecer las específicas dependencias presentes en el lazo (ver figura 4.9).

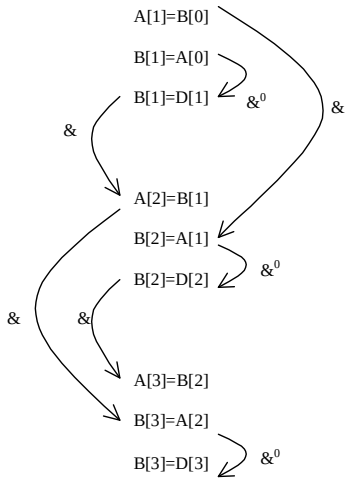


Figura 4.9. Dependencias especificas del ejemplo 4.23.

En el caso general, no es posible desenrollar los lazos para encontrar las dependencias, sino que se deben usar sistemas de ecuaciones, basados en los índices de los lazos, para encontrar las dependencias.

Ejemplo 4.24. Suponga el programa:

```
For I= 1 to 2
        For J=1 to 2
                T1(I,J)
                T2(I,J)
```

donde,

Tarea T1(I,J):
X[I, J]= F1(V[I-1,J], X[I-1,J-1])
Z[I, J]=constante1
V[I, J]=constante2

Tarea T2(I,J):
Y[I,J]=F2(Z[I, J], Y[I-1, J])

y F1 y F2 no tienen efectos colaterales, es decir no modifican ni a sus parámetros ni a variables globales. En dicho ejemplo, el grafo de lazos de tareas está compuesto por dos nodos T1 y T2 (ver figura 4.10). Los pesos de los nodos son los tiempos para calcular las funciones F1 y F2 (4 y 8 unidades de cálculo, respectivamente) y las asignaciones (1 unidad de cálculo cada una). De esta manera, la cantidad de cálculo de la tarea T1 es 7 y de la tarea T2 es 9 (pesos de las tareas T1 y T2). Para obtener los pesos de los arcos se asume que los arreglos X, Y, Z y V requieren 10, 12, 20 y 5 unidades de almacenamiento, respectivamente. Además, T1 usa durante la iteración <i,j> a V[i-1,j] que es calculado por la tarea T1 en la iteración <i-1,j>, y a X[i-1,j-1] que es calculado por la tarea T1 en la iteración <i-1,j-1>. Los vectores distancias para esas dos dependencias son <1,0> y <1,1>, respectivamente. Así, considerando el tamaño de los elementos de V y X, el

conjunto de dependencia de T1 a él mismo es {(<1, 0>, 5), (<1, 1>, 10)}. De la misma manera se le puede definir a T2 sus arcos. T2 usa en la iteración <i, j> a Z[i,j] (el conjunto de dependencias entre T1 y T2 es {<0,0>,20)}) y en la iteración <i-1, j> a Y[i-1,j] (el conjunto de dependencias consigo mismo es {<1,0>,12}).

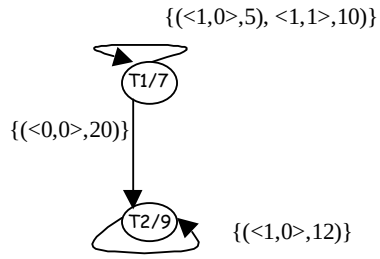


Figure 4.10. Grafo de dependencia para el ejemplo 4.24.

Ejemplo 4.25. Suponga el programa:

```
For J=1, 100
        For I=1, 100
S1:             C[I,J]=0
                For K=1, 100
S2:                     C[I,J]=C[I,J]+A[I,K]*B[K,J]
```

S1 es una instrucción del nivel 2 y S2 del nivel 3. El conjunto de índices de ejecución son, para S1 $[1:100]^2$ y para S2 $[1:100]^3$. El número de índices comunes es 2. Aquí hay dependencias $S1\&^{(=,=)}S2$ o $S1\&^{\infty}S2$, y $S2\&^{(=,=,<)}S2$ o $S2\&^{3}S2$. El grafo de dependencia sería:

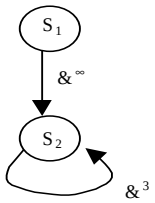


Figure 4.11. Grafo de dependencia del ejemplo 4.25

4.2.3 Test de Dependencias de Datos

En esta fase, el mayor objetivo es probar que no hay dependencia entre dos instrucciones S1 y S2 de un lazo, y si esto no es posible, caracterizar el conjunto de dependencias entre las dos instrucciones. Desafortunadamente no siempre es posible determinar con exactitud si un lazo tiene dependencias de datos o no. Normalmente, para determinar esto en una complicada referencia de arreglos, se requiere de específicas técnicas [29, 30, 38, 39]. Estas técnicas pueden ser tests exactos, ya que se determinan las condiciones necesarias y suficientes para que exista una dependencia. Si ellas no pueden ser

aplicadas, se aplican los tests llamados inexactos, ya que estos proveen solamente una necesaria condición para la existencia de una dependencia. El objetivo es determinar que no exista una dependencia, por lo tanto, tests inexactos siempre indican la presencia de una dependencia si realmente existe, y tal vez, si no existe. Eso es importante, porque la presencia equivocada de una dependencia solamente impide ciertas transformaciones del código, pero la semántica se preserva. En el caso contrario, es decir, la ausencia errónea de dependencia resultaría en códigos que no producen resultados correctos. Revisaremos algunas de esas técnicas en esta sección.

4.2.3.1 Análisis Diofantino

Se puede analizar la dependencia en un grafo del espacio de iteraciones usando una técnica basada en ecuaciones diofantinas lineales. Suponga dos enteros *a* y *b* ($b\neq0$), se dice que *a* divide a *b* ($a|b$) si hay un entero x tal que $a=bx$. Por otro lado, el máximo común divisor (MCD) del conjunto de números enteros $i_1, \ldots, i_n$, los cuales son diferentes de 0, para $n\geq1$, es definido como $MCD(i_1, \ldots, i_n)=\max\{b: i_j|b \ \forall j \in [1:n]$. Por ejemplo MCD(24, 18)=6, MCD(57, 33, 24)=3, MCD(4,6)=2, MCD(12,-18)=6, MCD(1561, 32)=1. Una ecuación lineal diofantina es de la forma $\sum_{i=1}^{n} a_i x_i=c$, para $n\geq1$, tal que *c* y a_i son números reales y no todos los a_i son iguales a 0, y los x_i son variables enteras, es decir, que solamente admiten valores enteros. Una ecuación diofantina puede o no tener una solución.

Ejemplo 4.26. Suponga la ecuación diofantinas en *x* e *y* siguiente:

x+4y=1

Ella tiene una solución para x=5, y=-1.

Por otro lado, la ecuación diofantina

5x-10y=2

no tiene una solución. Una sola ecuación diofantina tiene solución si $MCD(a_1, \ldots a_n)|c$, es decir, si el MCD de los coeficientes del lado izquierdo divide al lado derecho. En general, un sistema de ecuaciones no tiene solución cuando existe una ecuación $\Sigma_{i=1}^{n} a_i=c$, y no se cumple que $MCD(a_1, \ldots a_n)|c$. Para la primera ecuación del ejemplo 4.26, MCD (1,4)=1|1. Para el segundo ejemplo, MCD(5,-10)=5 el cual no divide a 2. Existen muchas técnicas para resolver ecuaciones diofantinas. Aquí presentaremos una técnica para el caso de dos variables (ax+by=c). Esta técnica se basa en definir *x* e *y* de la siguiente forma:

x= - b*t/MCD(a,b)+ algún múltiplo de c
y= a*t/MCD(a,b)+ otro múltiplo de c para t= 0, 1, ,2, …

t puede ser cualquier valor entero.

Ejemplo 4.27. Resolver la ecuación diofantina 2x-3y=198:

MCD(2,-3)=1 Así, x= 3t+ algún múltiplo de 198
y=2t+ otro múltiplo de 198

Un múltiplo de 198 es 396 y otro múltiplo es el mismo 198. Así,

x= 3t+396
y= 2t+198

y para t=0, 1, …, tenemos:

x=96, 399, …
y=198, 200, ..

Ahora, veamos su aplicación para el problema de dependencias. Asumamos conocidos S1(i), S2(i'), *i* e *i'*. Además, las funciones sobre los índices de los lazos f(i) y f'(i') pueden ser descritas como:

$f(i)= a_0 + \sum_{j=1}^{n} a_j i_j$
$f'(i')= b_0 + \sum_{j=1}^{n} b_j i'_j$

Se puede definir una ecuación diofantina de dependencia como:

$a_0 + \sum_{j=1}^{n} a_j i_j - (b_0 + \sum_{j=1}^{n} b_j i'_j)=0$

Esta ecuación puede ser usada para determinar si existe una dependencia de lazos, usando los índices de los arreglos de las instrucciones presentes en el lazo. Normalmente, es bastante eficiente en el caso de dos lazos anidados (2 dimensiones). Para poderla usar, los índices en los lazos deben ser lineales y no pueden haber otros términos. La solución de la ecuación será en función de los índices definidos como parámetros, tal que las restricciones sobre las variables de la ecuación deben ser evaluadas después (los posibles valores de los índices). En síntesis, la idea de este test, es examinar si la ecuación diofantina tiene una solución en una cierta región, la cual esta definida por los bordes del lazo.

Ejemplo 4.28. Determine la ecuación diofantina para el siguiente ejemplo:

```
For I=10, 20
S1:     A[2*I-1]= …
S2:     …       = A[4*I-7]
```

Para determinar si hay dependencia, hay que definir los valores de *x*, *y* e *I*, donde las variables accesadas por las instancias S1(x) son las mismas que las accesadas por S2(y). *x* e *y* representan valores independientes de la variable I, los cuales son asociados con las

respectivas ocurrencias de I en las variables índices [2*I-1] y [4*I-7]. Eso implica que debe haber una solución para la ecuación diofantina de dependencia 2x-1=4y-7. Supongamos el caso S1&$^{\infty}$S2 (dependencia independiente del lazo). Así, si *x* debe ser igual a *y*, la sola solución es x=y=3. Como este valor está fuera de la región de iteración (10≤x,y≤20), no hay una dependencia independiente del lazo.

Veamos el caso general para el ejemplo anterior, si se escribe la ecuación 2x-4y=-6, MCD=(2,-4)=2|-6. En este caso existen varias soluciones. La solución general sería y=-3-t y x=-9-2t. Sin embargo, la existencia de una solución para la ecuación lineal diofantina no es suficiente condición para la existencia de una dependencia. Se deben satisfacer aun ciertos criterios, como el de la región de iteración en el ejemplo anterior. Las únicas dos soluciones que cumplen esta adicional condición son (x,y)=(17,10) y (19,11). Así, tenemos que S2(10)&$^{-1}$S1(17) es causada por el acceso común a A[33], y S2(11)&$^{-1}$S1(19) es causado por el acceso común a A[37].

Ejemplo 4.29. Para el caso siguiente:

For I=30, 100
*S1: A[3*I-5]=…*
*S2: …= A[6*I]*

La ecuación de dependencia es 3x-6y=5. Como el MCD(3,6)=3 no divide a 5, podemos establecer que no hay dependencia entre S1 y S2.

Ejemplo 4.30. Para el caso siguiente:

For I=1, 101
*S1: A[2*I]= …*
*S2: …= A[3*I+198]*

La ecuación diofantina sería 2x=3y+198 donde 1 ≤ x, y ≤ 101, la solución general sería y= 3t+396 e y=2t+198. Las restricciones sobre *x* e *y* se convierten en restricciones para *t*, tal que:

1≤ 3t+396 ≤ 101 es decir, -131≤ t ≤ -99
1 ≤ 2t+198 ≤ 101 es decir, -98 ≤ t ≤ -49

Ya que t≤-99 contradice a –98≤t, la ecuación diofantina no tiene una solución que satisfaga esas restricciones. Así, el par de referencias no tiene dependencia de datos.

4.2.3.2 Test Inexacto

Resolver el sistema de ecuaciones diofantino puede ser muy complejo para un número arbitrario de variables. Incluso cuando se llega a una solución, el chequeo de las restricciones es difícil. Alternativas a este método, se pueden dar usando técnicas

inexactas, es decir, técnicas que determinan condiciones necesarias para una dependencia, no así suficientes.

- *Test de MCD:* Este test determina una condición necesaria para una dependencia, pero no suficiente, ya que ignora la región en el espacio asociado con la dependencia. Suponga una instrucción S con $f(i)=a_o+a_1i$ y $g(i)=b_o+b_1i$, ella depende de ella misma sí $MCD(a_1,b_1)|(b_0-a_0)$.

Ejemplo 4.31. Para el caso siguiente:

```
For I=…
        For J= …
S1:             A[2*I+2*J+101]= …
S2:             … = A[2*I-2*J]
```

La ecuación diofantina es $2x_1+2x_2-2y_1+2y_2+101=0$. El MCD de los coeficientes es 2, pero 2 no divide a 101. Así, la ecuación no tiene solución, y por consiguiente, no hay dependencia de datos entre las dos referencias.

Ejemplo 4.32. Para el caso siguiente:

```
For I = 0, 10
        For j=0,10
S1:             A[2*I+J]= …
S2:             …= A[-I+2*J-21]
```

Al evaluar las dependencias entre S1 y S2, la ecuación de dependencia es $2x_1+x_2+y_1-2y_2=-21$. El test de MCD daría (MCD(2,1,1,-2)=1|21). Pero en realidad, no hay una solución para la ecuación diofantina en la región cubierta por los índices de los lazos $\{0\leq x1,x2,y1,y2\leq 10\}$

- *Test del Borde:* Este es otro test que puede ser usado, en el que no se necesita resolver la ecuación diofantina. Este test trata la ecuación diofantina como una ecuación con valores reales, cuyo dominio es un conjunto convexo definido por un lazo constante bordeado. Esta ecuación tendrá solución si el mínimo del lado izquierdo no es mayor que 0 y el máximo no es menor que 0.

Ejemplo 4.33. Para el caso siguiente:

```
For I=1, 30
        For J=1, 30
S1:             A[200-I]= …
S2:             …= A[I+J]
```

La ecuación diofantina es $x_1+y_1+y_2-200=0$. La ecuación será evaluada con valores reales entre $1 \leq x_1, y_1, y_2 \leq 30$. No hay valores de I, J dentro del rango permitido, que den una solución a la ecuación, por lo tanto no hay dependencia de datos en ese lazo.

4.2.3.3 Test de Dependencia usando el Vector de Dirección

El test del vector de dirección es usado para encontrar la independencia de datos que MCD no puede encontrar, particularmente, cuando el MCD concluye que hay dependencia de datos, pero quizás no necesariamente en la región del espacio de iteraciones, en la que el lazo esta iterando. El test del vector de direcciones busca intersecciones en un arreglo. En general, se quiere probar si un arreglo es accesado por dos referencias que se interceptan. Una intersección ocurre cuando los subíndices de las dos referencias son simultáneamente iguales. Si la independencia puede ser probada con el vector de direcciones, significa que las regiones acccesadas por las dos referencias son disjuntas. El vector de direcciones usa una jerarquía de direcciones de vectores (la figura 4.12 muestra la jerarquía de direcciones para dos lazos anidados). Este test puede ser más difícil de aplicar que el MCD, pero es menos costoso que resolver las ecuaciones diofantinas. Además, es más exacto que el test MCD cuando los lazos tienen relaciones de dependencias y los índices quedan fuera del espacio de iteración.

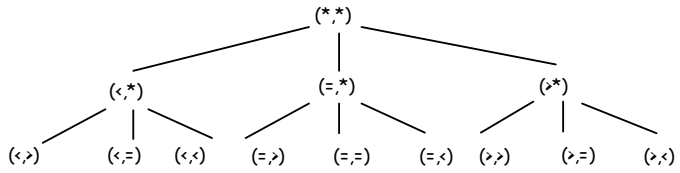


Figure 4.12. Jerarquía de Direcciones para dos lazos anidados.

Ejemplo 4.34. Suponga el siguiente lazo:

```
For I=1 ,10
        For J=1, 10
                A[10*I+J]= …
                … = A[10*I+J-1]
```

El espacio de iteraciones esta en una región 10 por 10 que deja por fuera los índices de A. En concreto, el espacio de iteraciones es definido por [I,J]=[1:10, 1:10], pero los subíndices de A están entre [11:110] y [10:109]. MCD dice que hay una dependencia porque MCD(10, 1)=1. De hecho, hay dependencia de datos en la región del espacio de iteraciones que nos interesa (donde aparecen los índices de A).

4.2.3.4 Teoremas de Verificación de Dependencias

Teorema 1: Consideremos las instrucciones Sr y St en un anidamiento perfecto de *n* lazos L. A(f(i)) es una variable de Sr y A(g(i')) es una variable de St, y al menos una de ellas es una variable de salida. Además,

$f(i)= a_o + \sum_{k=1}^{n} a_k i_k$
$g(i')=b_o+ \sum_{k=1}^{n} b_k i'_k$

donde a_k y b_k son constantes enteras $\forall k=0$, n, y n es el número de lazos anidados. Además, en cada lazo L_k las iteraciones van de 0 a μ_k, para k=1, n.

Si S1&S2 son causados por A(f(i)) y A(g(i')), entonces las dos siguientes condiciones deben ser ciertas:

Condición 1: El máximo común divisor (MCD) de (a_1, … a_k, b_1, …, b_k) divide a b_o-a_o.
Condición 2: Existe un α con $1 \le \alpha \le n$ para $r \ge t$ y con $1 \le \alpha \le n+1$ para $r<t$, tal que:

$$-b_\alpha - \sum_{k=1}^{\alpha-1} (a_k - b_k)^- \mu_k - (a_\alpha^- + b_\alpha)^+ (\mu_k - 1) - \sum_{k=\alpha+1}^{n} (a_k^- + b_k^+)\mu_k \le b_0 - a_0 \le$$
$$-b_\alpha + \sum_{k=1}^{\alpha-1} (a_k - b_k)^+ \mu_k + (a_\alpha^+ - b_\alpha)^+ (\mu_k - 1) + \sum_{k=\alpha+1}^{n} (a_k^+ + b_k^-)\mu_k$$

Para un número q, $q^+=\max(q, 0)$ y $q^-=\max(-q, 0)$, y μ_k es el borde superior del lazo k. Este teorema es verdadero tanto para dependencias verdaderas como de salida y antidependencias. En general, para que una dependencia exista, ambas condiciones deben ser verdaderas. Si alguna de las dos es falsa, se garantiza que S1&S2 es falso. Las dos condiciones son necesarias pero no suficientes para que una dependencia S1&S2 exista.

Ejemplo 4.35. Suponga el siguiente código:

```
For I=0, 2
S1:    A[4-3*I]=B[I]
S2:    C[I]=A[2*I]
```

Queremos determinar si existe $S1\&^{-1}S2$ tal que r=2, t=1, a_o=0, a_1=2, b_o=4, b_1=-3. Existe un solo valor de α (α=1) que satisface ambas condiciones:

Condición 1: MCD (2, -3)=1 divide a 4
Condición 2: $3-(0-3)^+ \le 4 \le 3+(2+3)^+$

Por esta razón, se desconoce la validez de $S1\&^{-1}S2$. De hecho, es fácil verificar que esta dependencia no es cierta, porque es imposible encontrar valores de I y J tal que

$0 \le I \le J \le 2$ y $2I=4-3J$

Así que este teorema no es suficiente, por lo cual se requiere de otro teorema.

Teorema 2: Condiciones 1 y 2 del teorema 1 son necesarias y suficientes, sí para algún valor positivo de q:

$a_k=q*\gamma_k$, $b_k=q*\beta_k$ y $\gamma_k, \beta_k \in \{-1, 0, +1\}$ $\forall k=1, \ldots n$.

Ejemplo 4.36. Suponga el siguiente código:

```
For I=0, 10
      For J=0, 5
            For K= 0, 8
S1:                 A[-40+4*I-8*J]=e1
S2:                 A[48-6*I+4*J+16*K] =e2
```

donde e1 y e2 son algunas expresiones. Supongamos que queremos verificar si $S1\&^0S2$ es cierto. $\mu_1=10$, $\mu_2=5$, $\mu_3=8$, $a_0=-40$, $a_1=4$, $a_2=-8$, $a_3=0$, $b_0=48$, $b_1=-6$, $b_2=4$, $b_3=16$. La condición 1 del teorema 1, para MCD(4, -8, 0, -6, 4, 16) = 2 que divide a 88, es verdad. Para $\alpha=1$, la condición 2 daría la inigualdad:

$-182 \leq 88 \leq 96$

En tal situación, análisis adicionales son requeridos para determinar si la dependencia $S1\&^0S2$ es cierta (por ejemplo, si i=(10, 0, 0) y i'=(10,3,0) se ve que es verdadera). Si b_0 se cambia de 48 a 49, la condición 1 garantiza que $S1\&^0S2$ no es verdadero. En forma similar, si se cambia b_0 de 48 a 68 ninguno de los cuatro valores de α (1, 2, 3, 4) genera una inigualdad válida para la condición 2. En ambos casos se puede concluir que $S1\&^0S2$ no es cierto.

Ejemplo 4.37. Suponga el siguiente código:

```
For I= 0, 10
      For J= 0, 10
S1:         A[5+3*I-3*J]=I+J
S2:         D[5+I+11*J]=A[11+3*J]
```

Se quiere verificar si S1&S2 es verdad. La condición 1 se reduce a MCD(3, -3, 0, 3)=3, que divide a 11-5=6, así esa condición es verdadera. En el caso de la condición 2, existen tres posibles valores de α=1, 2, o 3. Al escoger α=1, se llega a la inigualdad $-60 \leq 6 \leq 27$, lo cual también es verdadero. Conclusión, S1&S2 es verdad. El teorema 2 se puede aplicar para garantizar esa dependencia.

Ahora verifiquemos si $S2\&^0S2$ es verdad. Condición 1 es obviamente satisfecha ya que $b_0-a_0=0$. Hay dos posibles valores de α en condición 2, α=1 o 2. Al escoger α=1 llegamos a la inigualdad $-120 \leq 0 \leq 109$, así que también la condición 2 es verdad. En este ejemplo el teorema 2 no sirve, así que análisis adicionales se requieren, lo que revelará que $S2\&^0S2$ no es verdadero ya que no hay dos iteraciones $i=(i_1, i_2)$, $j=(j_1, j_2)$, tal que, $i \neq j$ y $5+ i_1+11* i_2=5+ j_1+11* j_2$

Los teoremas antes presentados han sido aplicados a arreglos de una dimensión. Supongamos arreglos de t-dimensiones, para t≥2. Asumiendo que μ_k y φ_k indican el límite superior e inferior en la dimensión k, se puede generar un arreglo B de una dimensión desde A bijectivamente de la siguiente forma:

$A(i_1, \ldots, i_t) <-> B(\theta(i_1, \ldots, i_t))$

Tal que: $\theta(i_1, \ldots, i_t)= \sum_{k=1}^{t} w_k (i_k-\varphi_k)$

Y $w_k = \prod_{s=k+1}^{t} (\mu_s-\varphi_s+1)$

Este es un mapeo por fila, pero el mapeo por columna es similar. Después de esta transformación, los teoremas 1 y 2 pueden ser aplicados. La función θ es una función lineal de los índices. Para más detalles sobre estos teoremas, ver a [29, 39].

4.3 Técnicas de Transformación

Solamente para pocos casos, la paralelización puede hacerse haciendo una división funcional, donde diferentes segmentos del programa ejecutan tareas diferentes, ya sea sobre datos compartidos o no. Ejemplos de programas de este tipo se encuentran en el área de Bases de Datos, Control de Procesos, etc. En esta sección no estamos interesados en ese paralelismo, sino en la ejecución concurrente de lazos. La idea es generar lazos paralelos, es decir, lazos cuyas iteraciones puedan ser ejecutadas simultáneamente por diferentes procesos. Como los dependencias independientes de los lazos no dependen de ellos, sólo las dependencias generadas por los lazos nos interesan.

En general, en función de la presencia o no de cierta dependencias, ciertos lazos pueden ser paralelizados. Muchas veces es necesario reescribir el lazo para paralelizarlo. Esta reescritura es conocida como transformación de lazos y tiene por objetivo aprovechar el paralelismo disponible en el sistema computacional [29, 30, 38, 39]. Una transformación es válida, si ella no cambia los resultados del programa. Las transformaciones se clasifican en función del tipo de modificación que ellos efectúan. Existen las transformaciones que cambian el orden de ejecución, las que efectúan redefiniciones de dependencias, las que modifican el acceso a la memoria, etc. Por ejemplo, las transformaciones que buscan mejorar el rendimiento del acceso a memoria, tratan de optimizar los diferentes niveles de la jerarquía de memoria al definir, por ejemplo, que cada instancia de un lazo trabaje con datos que pueden ser completamente contenidos en la memoria cache. Las transformaciones que se presentan aquí, se deben realizar en la última fase del proceso de compilación (después de las fases de análisis léxico, sintáctico, semántico, y generación de la representación intermedia, es decir, en la fase de optimización del código). También pueden ser usadas por herramientas generadoras de paralelismo automático.

Las transformaciones pueden ser aplicadas en cualquier orden y en forma independiente. Desafortunadamente, es frecuente el caso en que el orden de aplicación es muy importante. Así, si se aplica el mismo conjunto de *n* transformaciones en los n! diferentes posibles órdenes, tendremos n! semánticamente equivalentes programas. Debido a las diferencias en la sintaxis, estos programas tendrán aspectos diferentes de paralelismo, dando por resultado, posiblemente, diferentes rendimientos sobre una misma máquina. El mejor orden de aplicación de un conjunto de transformaciones es solamente conocido para un número reducido de aplicaciones. En general, encontrar el mejor orden de aplicación de las transformaciones es un problema abierto de investigación.

Básicamente existen tres tipos de ejecución de lazos:

- Ejecución secuencial de cada instancia del lazo (DOSEQ)
- Ejecución paralela de las diferentes instancias del lazo (DOALL). Esto es posible si no hay dependencias entre las instancia de las instrucciones del lazo, es decir, si las iteraciones se pueden ejecutar completamente en forma independiente. Un lazo secuencial se puede transformar en este tipo de lazo si no existen dependencias generadas por el lazo.

Ejemplo 4.38. Suponga el siguiente código:

```
For I=1 ,100
        A[I]=B[I]*C[I]+D[I]
        B[I]=C[I]/D[I-1]+A[I]
        If C[I] <0 then
                C[I]=A[I]*B[I]
```

Como no contiene dependencias generadas por el lazo, se puede paralelizar la ejecución del lazo:

```
Parallel For I=1,100
        A[I]=B[I]*C[I]+D[I]
        B[I]=C[I]/D[I-1]+A[I]
        If C[I] <0 then
                C[I]=A[I]*B[I]
```

- Ejecución a través del lazo que permite una cobertura parcial del procesamiento del lazo (DOACROSS). Es una clase de encauzamiento vectorial que solapa iteraciones de lazos como si ellas se ejecutaran en diferentes etapas de un encauzamiento. Es decir, permite un parcial solapamiento en la ejecución de sucesivas iteraciones, al realizar un encauzamiento de las iteraciones sincronizadamente. Hay ciertas sincronizaciones que se deben incorporar explícitamente en el código de los lazos. Así, al cuerpo de instrucciones del lazo, se le agregan instrucciones *send_signal(S), EXP)* y *wait_signal(S, EXP),* donde S es una instrucción y EXP una expresión que indica el índice de la instancia de la instrucción que se esta esperando. Cuando un proceso llega a una instrucción *send_signal(S, EXP),* envía una señal a los otros

procesos indicando que terminó la instrucción S con el índice de iteración EXP. Cuando un proceso llega a una instrucción *wait_signal(S, EXP)*, se bloquea en espera de mensajes provenientes de la instancia de ejecución S(EXP). Si todas las iteraciones son independientes, este paralelismo se reduce a un DOALL. Un DOACROSS se puede convertir en un DOALL usando la técnica de transformación por torsión, que se presenta más adelante. Un lazo secuencial puede ser transformado en un DOACROSS si todas las dependencias son satisfechas por sincronizaciones apropiadas entre las iteraciones del lazo.

Ejemplo 4.39. Suponga el siguiente código:

```
For I=1, n                          Su transformación sería     DOACROSS I=1, n
S1:    A[I]=B[I]-1                                      S1:    A[I]=B[I]-1
S2:    C[I]=A[I-3]/4                                           send_signal(S1, I)
                                                               wait_signal(S1,I-3)
                                                        S2:    C[I]=A[I-3]/4
```

En este ejemplo, S2 depende de S1 (S1&S2) con un vector distancia 3, eso implica que S2(i) no se puede calcular antes que S1(i-3). Es decir, no se puede transformar en un lazo paralelo. Sin embargo, si se puede transformar en una lazo DOACROSS. En DOACROSS, cuando se ejecuta S1(i) hay que enviar un mensaje al proceso encargado de calcular S2(i+3) para que lo autorice a calcular. Así, las iteraciones en este caso se sincronizan.

Ejemplo 4.40. Suponga el siguiente código:

```
For I=1, n
S1:    A[I]=B[I-1]+2
S2:    B[I]=A[I-3]/C[I-2]
```

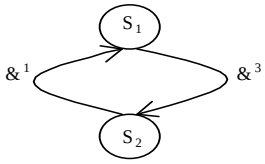


Figure 4.13. Grafo de Dependencia del ejemplo 4.40.

La transformación sería:

```
DOACROSS I=1, n
       wait_signal (S2,I-1)
S1:    A[I]=B[I-1]+2
       send_signal(S1,I)
       wait_signal(S1,I-3)
S2:    B[I]=A[I-3]/C[I-2]
       send_signal(S2, I)
```

Para cada iteración, la ejecución es bloqueada por *wait_signal(S2,I-1)* hasta que se envíe *send_signal(S2, I)* en la iteración previa. Sólo la primera iteración no se bloquea. Así, la ejecución de este lazo es más costosa que ejecutarlo secuencialmente. Normalmente, los lazos secuenciales se pueden transformar en lazos DOACROSS, pero en la practica hay que analizar el grado de paralelismo que se obtendrá para determinar el interés del mismo.

En general, en el caso de sistemas a memoria distribuidas, los lazos DOACROSS generan pases de mensajes que incrementan sus costos de ejecución. Los lazos DOACROSS, debido a las sincronizaciones, son más idóneos para máquinas a memoria compartida. En la sección siguiente estudiaremos algunas de las transformaciones que generan lazos paralelos.

4.3.1 Transformación DOALL

Esta transformación convierte cada iteración de un lazo en un proceso que es independiente de todos los otros, por consiguiente, se asume que no hay una dependencia generada por el lazo.

Teorema 3: Cada iteración de un lazo, puede ser ejecutada en un procesador separado, sin requerir sincronización entre los procesadores si no existen dependencias generadas por el lazo.

La ausencia de esa dependencia asegura que no se requiere comunicación entre los procesadores, si cada iteración es ejecutada en un procesador diferente. Si hubiera una dependencia generada por el lazo, se debe asegurar la ejecución de algunas operaciones, antes que otras operaciones inicien su procesamiento.

Ejemplo 4.41. Suponga el siguiente código:

```
For I= 1, n
S1:     b[I] = A[I] < 0
S2:     X[I] = B[I] when b[I]=true
S3:     X[I] = C[I] when b[I]=false
S4:     D[I] = X[I] + 1
```

Aquí no hay dependencias generadas por el lazo. Si hay *n* procesadores, cada iteración (ejecución de las instrucciones S1 a S4 para un particular valor de I) se asigna a un procesador diferente. En este caso, sólo se requiere de 4 unidades de tiempo en vez de las 4n unidades de tiempo, cuando se ejecuta secuencialmente el lazo. La transformación DOALL es muy eficiente (ignorando los costos de inicialización), sin embargo, normalmente hay muchas dependencias generadas por el lazo, por lo que es poco factible aplicarla directamente.

4.3.2 Distribución de Lazos

La transformación tipo distribución es la operación inversa de la transformación tipo fusión que veremos más adelante. En concreto, esta transformación divide la ejecución de un lazo en varios grupos de instrucciones de su cuerpo, para ser ejecutadas en diferentes lazos. El objetivo es distribuir las operaciones dentro de un lazo, en forma tal de reducir las necesidades de una sincronización explícita. Es decir, el propósito es obtener un conjunto de lazos más pequeños que el original, pero equivalentes, los cuales requieran menos sincronismo. Como la distribución de los lazos cambia solamente el orden de ejecución, no elimina las dependencias; al contrario, las reordena de tal forma que convierte las dependencias generadas por el lazo en dependencias independientes del lazo. La distribución se puede usar para crear nidos de lazos perfectos, lazos con menos dependencias, reducir el número de instrucciones en el nido de lazos, o mejorar el uso de la memoria.

Ejemplo 4.42. Suponga el siguiente código y sus dos transformaciones:

```
For I= …          For I= …          For I= …
S1:    …          S1:    …          S2:    …
S2:    …          For I= …          For I=…
                  S2:    …          S1:    …
```

En general, esta transformación alrededor de dos instrucciones S1 y S2 es legal si no hay dependencia entre ellas, o si las hay en una sola dirección. De hecho, en el ejemplo 4.42 existen cuatro formas de organizar la ejecución entre S1 y S2 (ver figura 4.14). Si no hay dependencia entre ellas, el orden en la distribución puede ser cualquiera. Si hay una dependencia de S1 hacia S2, entonces S1 debe ejecutarse antes que S2. Si hay una dependencia de S2 hacia S1, entonces S2 debe ejecutarse antes que S1. Finalmente, si hay un ciclo de dependencia entre S1 y S2, se prohibe la distribución del lazo.

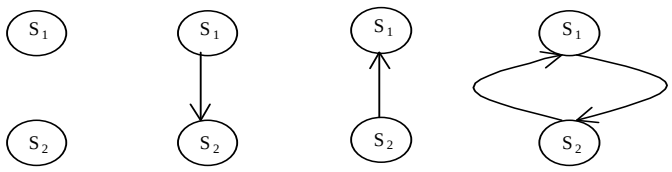


Figure 4.14. Los cuatro tipos de dependencias para el ejemplo 4.42.

Ejemplo 4.43. Suponga el siguiente código:

For I= 1, n
S1: C[I] = A[I] + B[I]
S2: D[I] = C[I-1] + B[I]

Como hay una dependencia generada por el lazo de S1 sobre S2, las iteraciones del lazo no pueden ser ejecutadas en paralelo. La ejecución de ese lazo puede ser reorganizada y distribuida en dos nuevos lazos, y la dependencia generada por el lazo es convertida en una dependencia independiente del lazo. Como resultado de ello, cada uno de los dos nuevos lazos puede ser ejecutado en paralelo, si todos los procesadores se sincronizan después de la culminación de la ejecución del primer lazo. Indudablemente que la ejecución será más rápida para este caso, que para el código original, porque sincronizar un lazo con otro, es mucho más fácil que sincronizar iteraciones de un lazo con iteraciones de otro lazo.

For I= 1, n
S1: C[I] = A[I] + B[I]
For I= 1, n
S2: D[I] = C[I-1] + B[I]

Se puede pensar que con una distribución de lazos, todas las dependencias dependientes del lazo pueden ser reemplazadas por dependencias independientes del lazo; esto no es cierto. A continuación damos un contraejemplo:

Ejemplo 4.44. Suponga el siguiente código:

For I= 1, n
S1: A[I] = B[I-1]+ 1
S2: C[I] = D[I-1]+ A[I]
S3: B[I] = A[I] + 1
S4: D[I] = C[I]+ 1
S5: E[I] = D[I] + F[I-1]
S6: F[I] = D[I-1] - F[I-1]

Se puede distribuir ese código de la siguiente forma:

```
For I= 1, n
S1:     A[I] = B[I-1]+ 1
S3:     B[I] = A[I] + 1

For I= 1, n
S2:     C[I] = D[I-1]+ A[I]
S4:     D[I] = C[I]+ 1

For I= 1, n
S6:     F[I] = D[I-1] - F[I-1]

For I= 1, n
S5:     E[I] = D[I] + F[I-1]
```

Como se ve en los tres primeros lazos, las dependencias generadas por el lazo, no pueden ser eliminadas por una simple distribución de los lazos. Estas dependencias son conocidas como dependencias hacia atrás (para diferenciarlas de las dependencias hacia adelante mostradas en el ejemplo 4.43). Así, si la dependencia generada por el lazo es hacia atrás, la misma no puede ser eliminada por este tipo de transformación.

4.3.3 Intercambio de Lazos

Es una transformación que intercambia (permuta) un par de lazos anidados, tal que el lazo más externo se convierte en el más interno y viceversa. Es decir, consiste en cambiar el orden o nivel de los diferentes lazos en el nido de lazos. Esto se puede aplicar repetidamente para intercambiar más de dos lazos de un conjunto de lazos anidados. Esta operación se realiza con el objeto de hacer posible que el lazo más interno se paralelice. Por consiguiente, se busca colocar los lazos que contienen las dependencias en el exterior del nido de lazos y aquellos que no tienen dependencias en el interior. De esta forma, el paralelismo se lleva a cabo sobre el lazo que está en la posición más externa para maximar dicho paralelismo.

Ejemplo 4.45. Suponga el siguiente código:

```
For I=1, n
        For J=1, n
                A[I,J]=A[I-1,J]+C[I,J]
```

Este lazo contiene una dependencia generada por el lazo en el nivel 1 y la paralelización es posible para el nivel 2. Si se paraleliza el lazo interno, al planificar la ejecución de ese lazo, se realiza la misma *n* veces, una por cada iteración del lazo externo. Al hacer el intercambio del lazo, la planificación se debe realizar una sola vez!.

Ejemplo 4.46. Suponga los siguientes códigos:

```
For I=1 , m                          For J=1, n
      For J= 1, n                          For I=1, m
S:          A[I,J]=A[I,J-1]          S:          A[I,J]=A[I,J-1]
```

En el código de la izquierda hay una dependencia entre las instancias de S generada por el lazo de índice J. Las instancias S(i,j) deben ser ejecutadas antes que las de S(i,j'), con j'>j, para cada i. Esto hace imposible paralelizar completamente ese nido de lazos. Por lo contrario, no hay ninguna restricción entre S(i,j) y S(i',j), sin importar los valores de *i, i'* y *j*. Así, se puede cambiar el orden de los lazos y paralelizar seguidamente el lazo más interno. La figura 4.15 indica el orden de ejecución en ambos casos. El mismo se ha cambiado respetándose las dependencias.

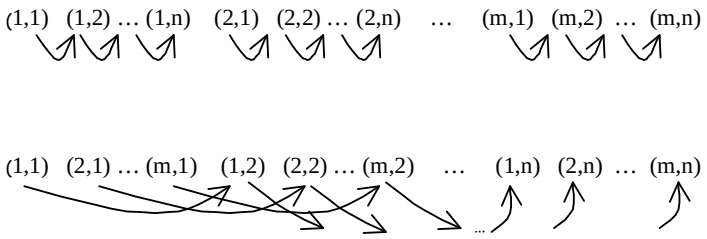


Figure 4.15. Ordenes de ejecución del ejemplo 4.46 antes y después del intercambio.

Para lograr realizar el intercambio de lazos correctamente, se debe determinar primero, cuál de los lazos anidados es responsable de la introducción de la dependencia. En general, se debe preservar las dependencias existentes. Para esto se usan los siguientes teoremas:

Teorema 4: Si una instrucción S2 tiene una dependencia generada por el lazo con una instrucción S1, esta dependencia será preservada por una transformación basada en un reordenamiento alrededor de las dos instrucciones que no cambian su orden de ejecución.

Se supone ahora que el nivel de un lazo, en un grupo de lazos anidados, es el número de los lazos que lo contienen más 1. Entonces se puede definir otro teorema:

Teorema 5: Cualquier transformación de reordenamiento, la cual no altere los lazos entre los niveles 1 y k, preserva cualquier dependencia de nivel k (nivel del lazo donde ocurre la dependencia).

Este teorema plantea lo importante del nivel de una dependencia; cuando la reestructuración de un código deja los lazos en el nivel menor o igual que el nivel de la dependencia, sin cambiar, la dependencia es preservada.

Ejemplo 4.47. En el siguiente código:

```
For I=1 , n-1
S1:     A[I+1] = F[I]
S2:     F[I+1]=A[I]
```

El orden en que S1 y S2 son ejecutados no es importante. Lo importante es preservar las dependencias entre ellas. Así, el siguiente código es equivalente:

```
For I=1 , n-1
S2:     F[I+1]=A[I]
S1:     A[I+1] = F[I]
```

Como se indicó anteriormente, esta transformación puede revertir una dependencia existente. Revertir una dependencia puede conllevar a cambiar el significado del código; por esta razón, para esta transformación el reordenamiento que revierte dependencias no es aceptable. Así, el intercambio de lazos, no es siempre semánticamente válido.

Ejemplo 4.48. En el siguiente código:

```
For J=1, 100
        For I=1, 100
                For K=1 ,100
S:                      C[I,J]=C[I,J]+A[I,K]*B[K,J]
```

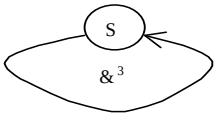


Figure 4.16. Grafo de dependencia del ejemplo 4.48.

Este lazo tiene un ciclo de dependencia de nivel 3 ya que el cálculo de C depende de un valor de C calculado en una iteración previa. Al intercambiar los lazos 3 y 1 tenemos:

```
For K=1,100
        For J=1, 100
                For I=1, 100
S:                      C[I,J]=C[I,J]+A[I,K]*B[K,J]
```

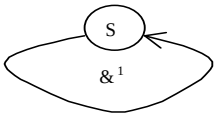


Figure 4.17. Nuevo Grafo de dependencia del ejemplo 4.48.

Esta transformación es válida al preservar las dependencias originales, y los dos lazos internos pueden entonces ser paralelizados (este ejemplo es una formulación alternativa de la multiplicación de matrices).

Ejemplo 4.49. Suponga el siguiente código:

```
For I= 1, 3
        For J= 1, 3
S:              A[I,J]=A[I-1,J+1]
```

Estos lazos no pueden ser intercambiados. Para eso veamos la figura 4.18. La figura 4.18.a muestra las dependencias entre las instrucciones, la figura 4.18.b muestra la ejecución del código original, aquí S(1,2) se ejecuta antes que S(2,1), mientras que la figura 4.18.c muestra el código después de intercambiar los lazos. En esa figura, S(1,2) se ejecuta después de S(2,1), por lo que S(2,1) usa el viejo valor de A[1,2], lo cual es erróneo.

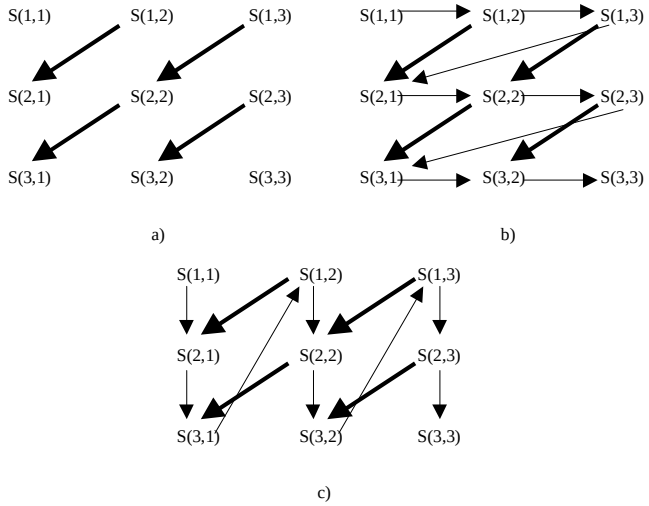


Figure 4.18. Ordenes de ejecución del ejemplo 4.49.

Un análisis similar se puede hacer con el lazo:

```
For I= 1, 3
```

```
    For J= 1, 3
        A[I,J]=A[I-1,J-1]
```

En este caso, haciendo un análisis parecido al ejemplo anterior, se ve que al intercambiar los lazos se mantienen las precedencias en los cálculos de los valores usados por las sentencias (se deja al lector hacer ese análisis).

Ejemplo 4.50. Suponga el siguiente código:

```
For I=1, 100
        For J=1, 100
S:          A[I+1,J]=A[I,J+1]*B[I,J]
```

Aquí hay una dependencia verdadera que envuelve a S en el nivel 1 ($S\&^{(<,>)}S$). Por ejemplo, entre S(1,2)&S(2,1), es decir, S(2,1) lee el valor de *A* previamente definido por S(1,2). Al intercambiar los dos niveles:

```
For J=1, 100
        For I=1, 100
S:          A[I+1,J]=A[I,J+1]*B[I,J]
```

la dependencia desaparece, pero la instancia para I=2 y J=1 es ejecutada antes de la instancia para I=1 y J=2. Así, esta transformación es invalida.

Ejemplo 4.51. En el siguiente código:

```
For I=1, 5
        For J= 1+I, 5+I
            A[I,J]=2*A[I,J]
```

El espacio de iteraciones se muestra a continuación:

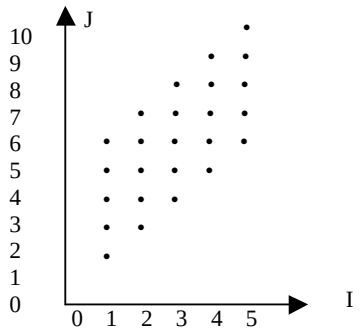


Figure 4.19. Espacio de iteraciones del ejemplo 4.51.

Para cada iteración del lazo de índice J se recorren los elementos correspondientes a los puntos (I, I+1) a (I, I+5). El espacio es recorrido columna por columna de abajo hacia arriba comenzando por el lado izquierdo. Al invertir los dos lazos, el espacio ya

no podrá ser recorrido columna por columna, sino línea por línea. Para recorrer el mismo espacio, el lazo J, que ahora es el más externo, debe tener sus índice variando de 1+min(I) a 5+max(I), donde min(I) y max(I) son los valores mínimos y máximos que puede tomar el índice I. Así, J varía de 1+1 a 5+4, y los valores que toma el índice I en el lazo interno, para un valor fijo de J, dependerán del valor de J. Para J=2 se calcula únicamente en el punto (1,2), para J=3 se exploran los puntos (1, 3) y (2, 3), y así sucesivamente.

```
For J=1+1, 5+5
        For I= max(1, J-5), min(5, J-1)
                A[I,J]=2*A[I,J]
```

Es fácil determinar si un intercambio es válido al inspeccionar el vector de direcciones correspondiente a la dependencia: una dependencia prohibe el intercambio entre dos lazos, si la entrada más a la izquierda del vector de direcciones, que es diferente de = es < antes del intercambio, es > después del intercambio. En general, dos lazos perfectamente anidados pueden ser intercambiados, si no hay dependencia de datos en el vector de direcciones (<, >), lo contrario no es cierto.

Ejemplo 4.52. Suponga el siguiente código:

```
Parallel For I=2, n
        For J= 2, n
S:              A[I,J]= (A[I,J-1] + A[I,J+1])/2
```

Las dependencias son del tipo $S\&^{(<,=)}S$ y $S\&^{-1(=,<)}S$. Al no haber dependencias (<,>) en el vector de direcciones se puede aplicar el intercambio:

```
For J= 2, n
        Parallel For I=2, n
                A[I,J] = (A[I,J-1] + A[I,J+1])/2
```

Ahora las dependencias son $S\&^{(<,=)}S$ y $S\&^{-1(<,=)}S$.

Un inconveniente de esta técnica, es porque no hay una manera única para determinar el mejor intercambio.

Teorema 6: Un intercambio de lazos al nivel *c* es válido si no hay un c-intercambio que previene la dependencia. Un c-intercambio que previene la dependencia es una dependencia del tipo $S1\&^{\theta}S2$ tal que $\theta=(=^{c-1},<,>,*,..*)$

Ejemplo 4.53. En el siguiente código:

```
For I=1, 100
        For J=1, 100
S:              A[I,J+1]=A[I,J]*A[I-1,J+1]
```

Se tienen las siguientes dependencias $S\&^{(=,<)}S$ y $S\&^{(<,=)}S$. Así, el intercambio al nivel 1 es válido. Al cambiar el orden resulta:

```
For J=1, 100
        For I=1, 100
S:              A[I,J+1]=A[I,J]*A[I-1,J+1]
```

Las nuevas dependencias son $S\&^{-1(<,=)}S$ y $S\&^{-1(=,<)}S$, respectivamente. Así, la dependencia de nivel 2 se mueve hacia afuera y la dependencia del nivel 1 hacia adentro.

4.3.4 Eliminación de Dependencias de Salidas y Antidependencias

Se ha visto cómo las dependencias condicionales se pueden eliminar en un programa, ahora se estudia cómo eliminar las antidependencias y las dependencias de salidas. Para eso, se usan dos técnicas:

- *Renombrar:* elimina ambas dependencias al cambiar el nombre de la variable que es subsecuentemente usada.

Ejemplo 4.54. Suponga el siguiente código:

```
S1: A = B+C
S2: D = A+ E
S3: A = A + D
S4: if x<0 then F=D+1
S5: if x>0 then G=F+A
```

En ese código existe una dependencia de salida entre S1 y S3 y una antidependencia entre S2 y S3. Si reemplazamos S3 y S5 por:

```
S3: A'=A+D
S5: if x>0 then G=F+A'
```

Así, A' es una nueva variable, y las dependencias de salida y antidependencia son eliminadas. Si después de S5 A vuelve a ocurrir, el valor será reemplazado por A' como se hizo con la instrucción S5.

- *Expansión Escalar:* es un caso especial de mencionar. Consiste en una transformación que puede aplicarse a variables escalares que aparecen en un lazo. Ella crea una copia de la variable para cada iteración del lazo anidado, la cual es usada para reemplazar la variable en cada iteración. Para ello se usa un arreglo con una apropiada dimensión. De esta manera, se eliminan dependencias asociadas con la variable, preservando la semántica e impidiendo ciclos de dependencias.

Ejemplo 4.55. Suponga el siguiente código:

For I = 1, n
S1: b = A[I] < 0
S2: x = B[I] when b=true
S3: x = C[I] when b=false
S4: D[I] = x + 1

Este programa tiene el grafo de dependencia mostrado en la figura 4.18.

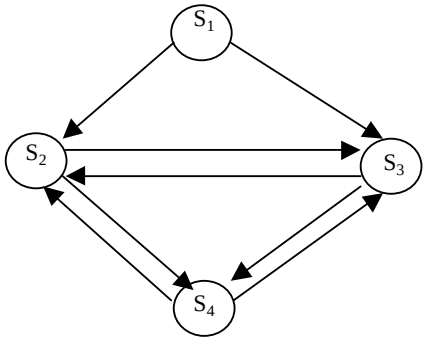


Figure 4.20. Grafo de dependencia del ejemplo 4.55.

Si se reemplaza en el código, la ocurrencia de *b* y *x* por b[I] y x[I], resulta:

For I = 1, n
S1: b[I] = A[I] < 0
S2: x[I] = B[I] when b[I]=true
S3: x[I] = C[I] when b[I]=false
S4: D[I] = x[I]+ 1

donde todos los ciclos de dependencias se han eliminados (ver figura 4.21).

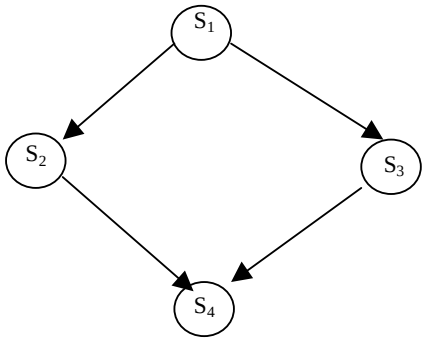


Figure 4.21. Grafo de dependencia del código modificado del ejemplo 4.55.

Ejemplo 4.56. En el siguiente código:

```
For I=1, 100
S1:     A=B[I]+C[I]
S2:     D[I]=A+1
S3      E[I]=A*(D[I]-2)
```

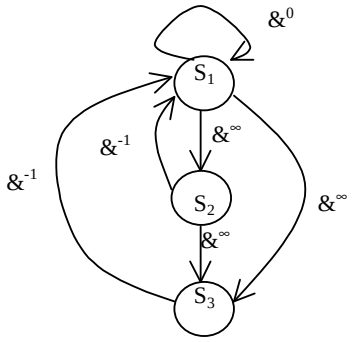


Figure 4.22. Grafo de dependencia del ejemplo 4.56.

Aquí todas las instrucciones forman parte del ciclo de dependencia, al aplicar la expansión escalar se eliminaría el ciclo de dependencia (ver figura 4.23).

```
For I=1, 100
S1:     temp[I]= B[I]+C[I]
S2:     D [I]= temp[I]+1
S3      E[I]= temp[I]*(D[I]-2)
```

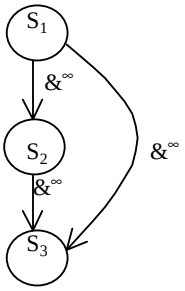


Figure 4.23. Grafo de dependencia del código modificado del ejemplo 4.56.

Esta transformación debe ser aplicada con cuidado, ya que si no se conocen las dimensiones de los lazos, podrían generarse requerimientos de memoria exagerados. Muchas veces, no se usa un vector sino la existencia de *temp* solamente en los registros, es decir, no se usa ningún espacio en la memoria. Por eso, normalmente se restringe su uso a los lazos más internos. El uso de esta técnica en el caso de paralelización de lazos sobre máquinas a memoria distribuida, resulta diferente. Esta técnica es usada en estos casos para colocar réplicas en las memorias locales de los procesadores. Así, la variable no es expandida en un arreglo (como en el ejemplo

anterior), sino que ahora será propia de cada procesador. Otra variante de mencionar, llamada *sustitución*, puede ser usada para eliminar dependencias verdaderas.

Ejemplo 4.57. Suponga el siguiente código:

```
For I=1, n
S1:    A[I] = B[I]-C[I]
S2:    D[I] = A[I]+A[I]*A[I-1]
```

Aquí existe una dependencia generada por el lazo entre S1 y S2. Si se sustituyen las tres ocurrencias de A en S2 por su correspondiente expresión S1, quedaría:

```
For I=1, n
S1:    A[I] = B[I]-C[I]
S2:    D[I] = B[I]-C[I] + (B[I]-C[I])*( B[I-1]-C[I-1])
```

De esta manera se elimina la dependencia generada por el lazo entre S1 y S2. Además, si *A* no se utiliza más adelante, la instrucción S1 puede ser eliminada. Sin embargo, es importante resaltar, que se requiere de más operaciones a ser realizadas. Por tanto, se debe hacer un análisis de rendimiento para aplicar esta técnica.

4.3.5 Fusión de Lazos

Esta transformación consiste en combinar dos lazos adyacentes en uno. En general, fusionar bloques paralelos aumenta la granularidad de las tareas, reduce los costos de gestión de los lazos, y puede reducir el sobrecosto debido a la sincronización. Pero los lazos que se generan pueden reducir sus rendimientos por un uso indebido de la jerarquía de memoria. Los dos lazos iniciales permiten eventualmente a un gran número de iteraciones accesar sus datos en la memoria cache, lo que quizá no sería el caso al fusionar los lazos. Hay que buscar un compromiso entre ambos aspectos.

Ejemplo 4.58. En el siguiente código:

```
For I = 2, n                    su fusión sería:        For I = 2, n

S1:    A[I]=B[I]+1                                      S1:    A[I]=B[I]+1
For I=2, n                                              S2:    D[I]=E[I]+F[I]
S2:    D[I]=E[I]+F[I]
```

Ejemplo 4.59. Suponga el siguiente código:

```
For I = 2, N
S1:     A[I]=B[I]+C[I]
For I=2, N
S2:     D[I]=A[I-1]
```

Al usar compiladores convencionales, estos dos lazos no son fusionados. Sin embargo, después de un análisis de dependencias, es claro que la fusión es posible, ya que la dependencia entre S2 y S1 no es violada:

```
For I = 2, N
S1:     A[I]=B[I]+C[I]
S2:     D[I]=A[I-1]
```

La inexistencia de dependencias entre los lazos, es una condición necesaria, pero no suficiente, para aplicar esta transformación. Hay que determinar las condiciones cuando dos lazos, ya sean paralelos o secuenciales, pueden ser combinados. La fusión es posible si dos lazos tienen el mismo espacio de iteraciones y si no existe una dependencia entre una instrucción del primer lazo con una instrucción del segundo lazo. Al existir tal dependencia, ciertas instancias de las instrucciones del primer lazo deben ser efectuadas antes de las del segundo. Esta dependencia no es necesariamente respetada, si se fusionan los lazos. Así, una fusión de lazos será válida si no existe una dependencia que previene la "fusión-serial" entre los lazos a funcionar. Una dependencia $S1\&^{\infty}S2$ previene la "fusión-serial", si hay un vector de iteraciones *i* para S1 e *i'* para S2, tal que la dependencia es causada por las instancias de S1(i) y S2(i') e $i_c > i'_c$ (note que el nivel máximo de lazos en común entre S1 y S2 es c-1, lo que implica que para todo j, $1 \leq j < c$, $i_j = i_j'$).

Ejemplo 4.60. En el siguiente código:

```
For I=1, n
        A[I]=A[I-3]+C[I]
For I= 1, n
        D[I]=A[I+1]*D[I-2]
```

A[I] y A[I+1] generan una dependencia que no permite la fusión serial. Así, esta transformación produce un lazo con una semántica diferente. Ahora, si el código es:

```
For I= 1, n
        A[I]=A[I-3]+C[I]
For I= 1, n
        D[I]=A[I-1]*D[I-2]
```

pueden ser fusionados sin problemas. Note que la dependencia independiente del lazo A[I] y A[I-1] de la versión secuencial, se transforma en una dependencia generada por el lazo al fusionarlos, lo cual no cambia la semántica del programa.

4.3.6 Torsión de Lazos

Esta transformación tiene por objeto, modificar la forma del espacio de iteraciones al desplazar el trabajo por iteración. Así, esta transformación modifica los índices de los lazos originales, creando un nuevo espacio de iteraciones. La torsión es siempre válida, ya que ella no cambia el orden de ejecución de las instancias de las instrucciones. El interés de la torsión reside en el hecho de poder modificar las distancias de las dependencias de los lazos, en tal forma que todas las dependencias son generadas por ciertos lazos, mientras que otros lazos no generan dependencias. Estos últimos pueden ser paralelizados de manera segura. Así, esta transformación extrae el paralelismo de un nido de lazos, en casos donde el paralelismo no puede ser encontrado para un simple lazo.

Definición 4.2. En el siguiente código:

```
For I=1, n                    su torsión sería    For I=1, n
    For J=1, m                                    For J=1+α*I, m+α*I
        A[I, 2*J] = …                                 A[I, 2*(J-α*I)]=…
```

Una torsión de paso α, donde α es un entero, consiste en agregar α-veces el índice del lazo externo a los bordes del lazo interno, conjuntamente con la sustracción de α-veces el índice del lazo externo de cada ocurrencia del índice del lazo interno en el cuerpo del lazo. Ilustremos su uso con el siguiente ejemplo:

Ejemplo 4.61. Suponga el siguiente código:

```
For I=1, n-1
    For J=1, n-1
        A[I, J] = (A[I+1, J]+A[I-1, J]+A[I, J+1]+A[I, J-1])/4
```

En este caso, ninguno de los dos lazos puede ser paralelizado. Sin embargo, al ver el grafo del espacio de direcciones para n=5 (ver figura 4.24), se nota una ola hacia el frente en forma de diagonal. De esta manera, las iteraciones a través de esa diagonal pueden ser paralelizadas (ver figura 4.25).

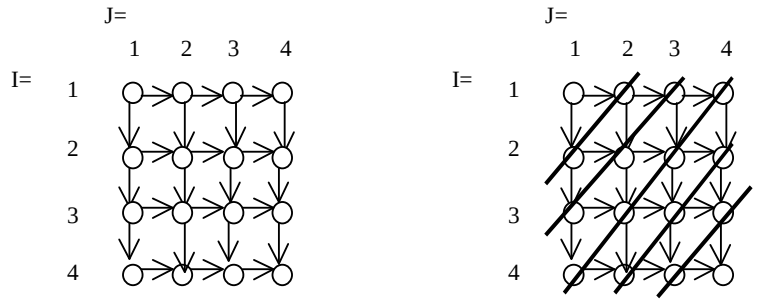


Figure 4.24. Grafo del espacio de direcciones del ejemplo 4.61.

Al hacer una torsión para α=1, el código quedaría

For I= 1, n-1
For J=I+1, I+n-1
A[I,J-1]=(A[I+1,J-I]+A[I-1,J-I]+A[I,J+1-I]+A[I,J-1-I])/4

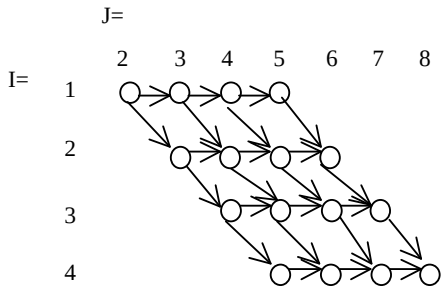


Figure 4.25. Nuevo grafo del espacio de direcciones del ejemplo 4.61.

Ejemplo 4.62. Suponga una pequeña modificación del código anterior:

For I=1, 5
For J=1, 5
A[I,J]= A[I+1, J]+ A[I-1,J]+A[I,J+1]+A[I,J-1]

Su espacio de iteraciones se muestra en la figura 4.26. Las rectas punteadas indican cálculos que se pueden hacer en paralelo, gracias a la ausencia de dependencias en esas direcciones.

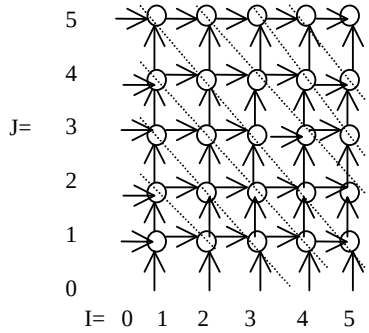


Figure 4.26. Espacio de iteraciones del ejemplo 4.62.

Al efectuar una torsión en el espacio de iteraciones, con un paso α=1, se eliminan las dependencias horizontales. Su nuevo código es (el espacio de iteraciones es mostrado en la figura 4.27):

```
For I=1, 5
        For J=1+I, 5+I
                A[I,J-I]=A[I+1,J-I]+A[I-1,J-I]+A[I,J+1-I]+A[I,J-1-I]
```

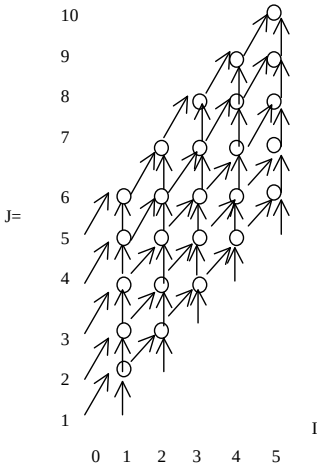


Figure 4.27. Nuevo espacio de iteraciones del código transformado.

Normalmente, esta transformación se usa conjuntamente con otras transformaciones. En el ejemplo anterior, el lazo externo no tiene dependencias después de la torsión, lo que permite intercambiar el orden de los lazos I y J en el código, para obtener un lazo paralelo sobre el índice del lazo I.

```
For J=1+1, 5+5
        For I=max(1,J-5), min(5,J-1)
                A[I,J-I]=A[I+1,J-I]+A[I-1,J-I]+A[I,J+1-I]+A[I,J-1-I]
```

4.3.7 Otras Técnicas de Transformación

A continuación presentaremos otras técnicas menos utilizadas [29, 30, 38, 39].

4.3.7.1 Alineación y Replicación

La *alineación* de lazos intenta transformar una dependencia generada por el lazo en una dependencia independiente del lazo, sin distribuir el lazo. Esto se logra al mover referencias a variables de una iteración a otra, es decir, alineándolas. Además, es necesario hacer otras modificaciones, en particular, algunas instrucciones se deben ejecutar condicionalmente para preservar la semántica del programa.

Ejemplo 4.63. En el siguiente código:

```
For I=1, n
S1:     A[I] = B[I]+C[I]
S2:     D[I] = A[I-1]*2
```

Claramente, hay una dependencia generada por el lazo entre S1 y S2. Si se quiere planificar cada iteración del lazo, en diferentes procesadores, se necesitarán puntos de sincronización entre S1 y S2 para asegurar la exactitud de las operaciones (que S2 use el más reciente valor de A). Se puede usar distribución de lazos y crear dos independientes lazos paralelos, pero se seguirá requiriendo el punto de sincronización entre S1 y S2. Una forma de resolver el problema es alinear los cálculos de A[I] y A[I-1] que ocurren en diferentes iteraciones, para que se realicen en la misma iteración. Por ejemplo:

```
For I=0, n
        If (I>0) then A[I] = B[I]+C[I]
        If (I<n) then D[I+1] = A[I]*2
```

En este caso, no se requiere de sincronizaciones, si cada iteración es asignada a un procesador diferente. Es decir, este lazo puede ser ejecutado como un lazo paralelo sin sincronizaciones.

Esta transformación no es siempre válida. Por ejemplo, si S2 se reemplaza en el código del ejemplo 4.63 por D[I]=A[I-1]+A[I], entonces algún intento de alineación cambiaría la semántica del programa ya que destruiría la dependencia independiente del lazo. Se puede usar *Replicación* del código en este caso, al introducir arreglos temporales para remover cualquier dependencia generada por el lazo que no sea parte del ciclo de dependencia. Veamos con detalles este ejemplo.

Ejemplo 4.64. Suponga la modificación del ejemplo 4.63 siguiente:

For I= 1 ,n
S1: A[I]=B[I] + C[I]
S2: D[I]=A[I]+A[I-1]

En este caso es necesario aplicar alineación del lazo en conjunto con replicación. Así, si se réplica S1 (y se cambia S2, respectivamente), la misma instrucción no tendrá instancias con A[I] y A[I-1] en el mismo momento.

For I=1 , n
S1: A[I]=B[I]+C[I]
S1': A1[I] = B[I]+C[I]
S2': D[I] = A1[I] + A[I-1]

Ahora, la alineación de lazos puede ser aplicada, y un simple lazo se obtiene donde cada iteración del lazo puede ser asignada a un procesador diferente, sin necesidad de puntos de sincronización:

For I=0 , n
T1: If (I >0) A[I] = B[I]+C[I]
T1': If (I<n) A1[I+1] = B[I+1]+C[I+1]
T2': If (I<n) D[I+1] = A1[I+1] + A[I]

Como A1 es temporal y no se usa subsecuentemente, se puede usar una variable escalar en cada iteración (procesador).

Así, se define el siguiente teorema:

Teorema 7: La alineación de lazos y replicación de código pueden ser usados para eliminar cualquier dependencia verdadera, generada por el lazo que no forma parte de un ciclo de dependencia.

4.3.7.2 Destapar Minas y Unir lazos

La aplicación de *Destapar Minas* sirve para distribuir un largo lazo sobre diferentes procesadores. En este caso, el lazo más externo es paralelizado, donde a cada procesador se le asigna una instancia del lazo más interno. *Destapar Minas* también se conoce como seccionamiento de un lazo, y tiene mucho que ver con el tamaño de la memoria local. Dependiendo de la misma, se puede enviar la cantidad de datos que puede ser contenida en ella.

Ejemplo 4.65. En el siguiente código:

```
For I=1, n
S:      A[I]=B[I]+C[I]
```

Si el registro para almacenar un vector es de *64k* y *n* es mucho mayor que eso, se debe dividir. Al Suponer que $n=64q+r$, tal que $q \geq 0$ y $0 \leq r < 64$, entonces el lazo se divide en un lazo externo y un lazo interno de operaciones, como sigue:

```
For J=1, q+1
        For K=1, min(64,n-(J-1)*64)
                I=(J-1)*64 + K
                A[I]=B[I]+C[I]
```

o

```
For J=1, q
        For K=1, 64
                A[-64+64*J+K]=B[-64+64*J+K]+C[-64+64*J+K]
If r ≠0
        for K=1, r
                A[64*q+K]=B[64*q+K]+C[64*q+K]
```

En este caso, el lazo externo es paralelizado sin problemas.

Ejemplo 4.66. En el siguiente código:

```
For  I=1, n
        A[I]=B[I]+1
        D[I]=B[I]-1
```

Al asumir que la longitud del registro de vectores es 128, una transformación de destapar minas es:

```
For J= 1, n , 128
        For I= J, min(J+127, n)
                A[I] = B[I]+1
                D[I] = B[I]-1
```

Unir lazos transforma varios lazos paralelos anidados en un simple lazo. Es decir, se intenta crear un gran lazo a partir de dos o más pequeños lazos anidados. Esta transformación es usada normalmente por los sistemas de planificación de lazos, para reducir los procesos de creación (overhead) en algunos sistemas. Es decir, cuando el tamaño de dos lazos es muy pequeño para paralelizar, pero el lazo resultante (que es más grande) puede ser paralelizado efectivamente. Esta técnica lineariza el espacio de

iteraciones multidimensional. Por ejemplo, un espacio de iteraciones bidimensional se convierte en un espacio uno-dimensional.

Ejemplo 4.67. En el siguiente código:

```
For J=1, n
        For I= 1, m
                A[I,J]=B[I,J]+2
```

Se transforma en:

```
For L=1, n*m
        J=(L-1)/m+1
        I=mod(L-1,m)+1
        A[I,J]= B[I,J]+ 2
```

Ejemplo 4.68. En el siguiente código:

```
Parallel For I=1, n
        Parallel For J= 1, n
                A[I,J]= …
```

Se transforma en:

```
Parallel For K=1, n*n
        A[Redondeo+(K/n), K-n*Redondeo-((K-1)/n)]= …
```

donde Redondeo^{+}(e) significa que el número real *e* se redondea hacia arriba, y en Redondeo^{-}(e) hacia abajo.

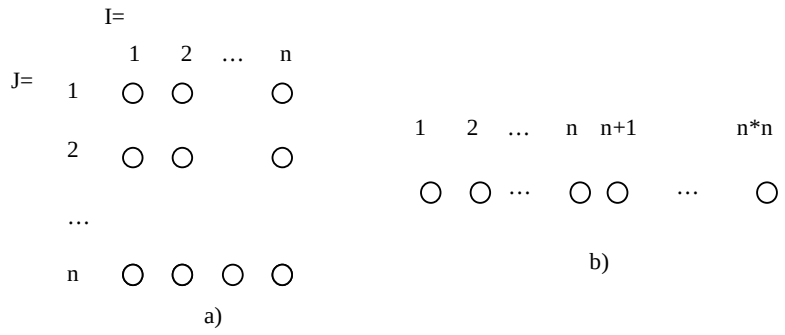


Figure 4.28. Espacio de iteraciones del ejemplo 4.68 antes (a) y después (b) de la unión.

\

4.3.7.3 Partición de Nodos

Corrientemente una instrucción es considerada como una unidad atómica, con la consecuencia, que variables que ocurren en S, que contribuyen a un ciclo de dependencia, hacen que toda la instrucción no sea paralelizable. Esta transformación trata de aislar las operaciones de una instrucción que no participan en un ciclo de dependencia. De esta forma, el ciclo de dependencia puede ser roto, resultando en una paralelización. Normalmente, esta transformación es seguida por una distribución del lazo.

Ejemplo 4.69. Suponga el siguiente código:

For I=1, n
*S1: B[I]=A[I]+C[I]*D[I]*
S2: A[I+1]=B[I](D[I]-C[I])*

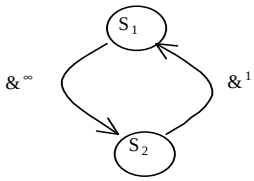


Figure 4.29. Grafo de Dependencia del ejemplo 4.69.

Ese ciclo es causado exclusivamente por A[I] y B[I] en S1, y A[I+1] y B[I] en S2. Los cálculos de C[I]*D[I] y D[I]-C[I] pueden ser separados e individualmente paralelizados. Así, el nuevo código es:

For I=1, n
*S1': T1[I]=C[I]*D[I]*
For I=1, n
S2': T2[I]=D[I]-C[I]
For I=1, n
S1'': B[I]=A[I]+ T1[I]
*S2'': A[I+1]=B[I]*T2[I]*

Ejemplo 4.70. Suponga el siguiente código:

For I=1, n
S1: A[I]= B[I]+ C[I]
*S2: D[I]=A[I-1]*A[I+1]*

Se puede eliminar el ciclo que causa la antidependencia partiendo el nodo y usando un renombramiento

```
For I=1, n
S3:     temp[I]=A[I+1]
S1:     A[I]= B[I]+ C[I]
S2:     D[I]=A[I-1]* temp[I]
```

Para después hacer un distribución de lazos:

```
Parallel For I=1, n
S3:     temp[I]=A[I+1]
Parallel For I=1, n
S1:     A[I]= B[I]+ C[I]
Parallel For I=1, n
S2:     D[I]=A[I-1]* temp[I]
```

4.3.7.4 Encoger Lazos

Cuando todas los dependencias en un ciclo son dependencias de flujo, partir un nodo no ayuda. Sin embargo, dependiendo de las distancias de cada dependencia se puede paralelizar parte de esos lazos usando esta transformación.

Ejemplo 4.71. Suponga el siguiente código:

```
For I=1, n
S1:     A[I] = B[I-2]-1
S2:     B[I] = A[I-3]*k
```

Después de aplicar esta transformación:

```
For J=3, n, 2
        Parallel For I=J, J+1
S1:             A[I] = B[I-2]-1
S2:             B[I] = A[I-3]*k
```

el espacio de iteraciones en cada caso es:

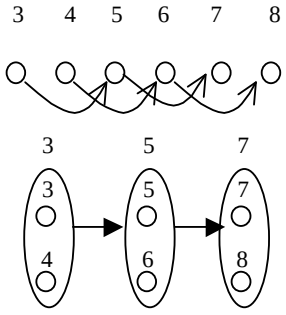


Figure 4.30. Espacio de iteraciones del ejemplo 4.71 antes (a) y después (b) de la transformación.

4.3.7.5. Desenrollar Lazos

Esta técnica se refiere al proceso de hacer una o más copias del cuerpo del lazo. Es decir, se reestructura el lazo, escribiendo las iteraciones como segmentos de código lineal. Al hacer k copias del cuerpo, se debe modificar el salto del lazo a un valor igual a k y disminuir el límite superior (suponiendo que el salto es positivo) por k-1. Esta transformación es usada para reducir el sobrecosto, al ejecutar el test de control en la ejecución de un lazo, o para decrecer la cantidad de movimientos de datos en la jerarquía de memoria.

Ejemplo 4.72. En el siguiente código:

```
For I=1, 100
        A[I]=B[I+2]*C[I-1]
```

Desenrollar una vez daría:

```
For I=1, 99,2
        A[I]=B[I+2]*C[I-1]
        A[I+1]=B[I+3]*C[I]
```

Ejemplo 4.73. En el siguiente código:

```
J=n
For I=1, n
        A[I]= (B[I]+B[J])/2
        J=I
```

Después de desenrollar la primera iteración del lazo se obtiene:

```
A[1]= (B[1]+B[n])/2
For I=2, n
        A[I]= (B[I]+B[I-1])/2
```

La operación inversa es reenrollar un lazo. Su efecto consiste en deshacer un desenrollamiento para hacer más eficiente el código.

Ejemplo 4.74. El siguiente código:

```
Q=0
For K=1, 996, 5
        Q=Q+Z[K]*X[K]+Z[K+1]*X[K+1]+Z[K+2]*X[K+2]+Z[K+3]*X[K+3]+Z[K
+4]*X[K+4]
```

Es reenrollado a:

```
Q=0
For K=1, 1000
        Q=Q+Z[K]*X[K]
```

4.4 Remarcas Finales

Las diferentes representaciones de dependencias de datos son más aptas para ciertas transformaciones. A continuación se resume la relación entre ellas [29, 30, 38, 39]:

Notación de Dependencia	*Que Modela*	*Técnica*
Grafo de Dependencia	General	Distribución de Lazos
Espacio de Iteraciones	Lazos Anidados	Partición de Nodos Torsión de Lazos Unir Lazos Encoger Lazos
Vector Distancia	Lazos Anidados	Torsión de Lazos
Vector de Dirección	Lazos Anidados	Fusión de Lazos Intercambio de Lazos Distribución de Lazos

Después de hacer las transformaciones, se debe mapear la nueva estructura de lazos sobre las máquinas. En el caso de máquinas a memoria compartida. es fácil, y se puede usar cualquiera de las técnicas de planificación de lazos conocidas. El compilador sólo necesita agregar las instrucciones de sincronización en los puntos apropiados. En el caso de la memoria distribuida resulta más difícil. La razón es la siguiente:

- Los datos deben ser explícitamente distribuidos en cada procesador.
- La comunicación debe ser explícitamente especificada.

La partición de datos influye en el costo de comunicación, lo que a su vez redunda en la velocidad de ejecución del programa paralelo. Al suponer que la partición de datos esta hecha, se tiene aun el problema de encontrar la topología de interconexión lógica para ese programa, y mapearla al hardware de la red. Algo que puede ayudar en estas tareas es colocar puntos de comunicación dentro de los lazos (parecidos a los puntos de sincronización en las máquinas a memoria compartida). Por supuesto, dichos puntos deben estar ligados a la topología lógica de interconexión y al patrón de distribución de los datos de la aplicación.

Ejemplo 4.75. Se tiene una topología lógica tipo anillo con 4 nodos, además, se desea distribuir dos arreglos A[40] y B[40] según los patrones de distribución tipo simple, preenvío e intercalado. El objetivo es minimizar el costo debido al pase de mensajes.

- *Distribución Simple:* se utiliza cuando las iteraciones del lazo son independientes, así los datos se pueden repartir equitativamente entre los procesadores. Por ejemplo:

```
Parallel For I=1, 40
        A[I] = A[I]+B[1]
```

El compilador puede generar el siguiente código:

```
For I=1, 4
        pl=10*(I-1)+1
        pu=10*I
        Send(A[pl:pu], B[pl:pu], I)
```

Y cada procesador tendría:

```
Receive (A[1:10],B[1:10])
For I=1, 10
        A[I]= A[I]+B[I]
```

Aquí se observan ciertas ineficiencias, como por ejemplo, cada procesador debe esperar hasta recibir ¼ de A y B, los elementos seleccionados de A y B son secuencialmente enviados desde el procesador 0 (es uno de los 4 procesadores), después de que cada procesador termina la suma, el resultado parcial debe ser coleccionado, y así otros más. Todos estos aspectos, de alguna manera, pueden reducir las ganancias de paralelismo.

- *Preenvío:* cuando la comunicación es más complicada, por ejemplo, cuando se requiere comunicación entre los vecinos, por acceso a datos vecinos en un arreglo, se requiere de otros métodos. Ejemplo:

```
Parallel For I=0, 39
        A[I]= 0.5(A[I-1]
```

Los datos son partidos y distribuidos equilibradamente a través de los procesadores como en el ejemplo anterior: P0 tendrá A(0:9), P1 tendrá A(10:19), P2 tendrá A(20:29), P3 tendrá A(30:39). Esta simple distribución falla debido a las condiciones de los bordes del arreglo para el cálculo, note que P0 necesita B[10] de P1; P1 necesita B[9] de P0, etc. Esto se toma en cuenta de la siguiente manera:

```
For I=1, 4
        pl=10*(I-1)
        pu=10*I-1
        Send(A[pl:pu], I)
```

Y cada procesador tendría:

```
Dimension A[0:10]
if id=0 then
        Receive (A[0,9], id)
        Send(A[9],id+1)
        for I=0, 9
                A[I]= 0.5*A[I-1]
if id=3 then
        Receive (A[1,10], id)
        Receive(A[0],id-1)
        for I=1, 10
                A[I]= 0.5*A[I-1]
if id=1 or id=2 then
        Receive (A[1,10], id)
        Receive(A[0],id-1)
        Send(A[10],id+1)
        for I=1, 10
                A[I]= 0.5*A[I-1]
```

Sin duda que las operaciones de envío/recepción producen un retraso en la ejecución. Aquí se aprovecha la ventaja de la topología tipo anillo en la red lógica, y que no hay dependencia con respecto al contador del lazo. Además, si no hubiera restricción al nivel del espacio de memoria, se puede enviar todo B a todos los procesadores.

- *Intercalar:* en muchas aplicaciones el patrón de distribución es una difusión a todos los procesadores, por ejemplo:

```
Parallel For I=0, n-1
        For J=0, n-1
                A[I]=A[I]+B[J]
```

En este ejemplo cada procesador necesita a B, donde a cada procesador se le puede preenviar todo el vector B a su memoria local, antes de entrar al lazo paralelo, para esto, se debe asumir que se tiene suficiente espacio de memoria para guardar B en cada memoria local. Para evitar el tiempo de transferencia de memoria, el compilador puede generar código para intercalar la difusión con el cálculo. Así, el costo por la difusión es solapado con el cálculo, si se permite comunicación asincrónica. De esta manera, se envía un bloque de B a todos los procesadores, después se inicia en paralelo el cálculo parcial de A en cada sitio, al terminar se borra el bloque de B y se espera recibir otro bloque. Cada procesador recibe a un subsecuente bloque, continua a sumar, y borra el bloque, así hasta que todos los elementos de B hayan sido enviados, totalizados y borrados.

Hemos concluido la presentación de las técnicas de transformación. Ellas son necesarias si se quieren construir códigos eficientes. Algunas herramientas se han construido para ayudar en este proceso. Por otro lado, han existido muchos intentos por construir compiladores que generen código paralelo, ya sea creando código paralelo o

extrayendo automáticamente paralelismo de programas seriales existentes. Una automatización completa de la generación de programas paralelos aun no es posible, y probablemente nunca lo será, pero herramientas semiautomáticas de reestructuración han sido construidas para realizar interactivamente análisis de dependencias que ayuden al programador a identificar paralelismo, y poder así, ejecutar las transformación necesarias.

La tarea de extracción automática de paralelismo resulta más fácil, si el usuario aporta cierto paralelismo explícito. Es decir, si el programador introduce más información semántica que pueda ser explotada por la herramienta. En general, la vectorizacion se hace sin problemas y con éxito cuando se basa en la información sintáctica. Sin embargo, para una paralelización efectiva y eficiente es necesario usar información semántica, la cual es mucho más difícil de tener. Además, al usar herramientas de paralelización se debe permitir interacciones entre el usuario y el sistema, por lo que el usuario debe tener cierto conocimiento sobre el análisis de dependencia y de las técnicas de transformación. Para conocer qué información es más beneficiosa para la herramienta, el programador debe estar consciente de las capacidades de dicha herramienta. También, puede suceder que aparezcan especificaciones erróneas de dependencias, que sólo el programador que conoce el código puede resolver y/o eliminar (en realidad, el problema es que las técnicas no son suficientemente finas, por lo tanto, el compilador no puede determinar que no existe una dependencia. En ese caso, tiene que asumir que si existe una dependencia). Integrar aspectos de automatización, con el conocimiento íntimo del problema, conducirá, en el futuro, a muy interesantes desarrollos.

Capítulo 5.
Medidas de Rendimiento

En los Sistemas Distribuidos/Paralelos, uno de sus principales problemas es la degradación de sus rendimientos. En un ambiente ideal, el rendimiento crece linealmente al crecer el número de procesadores. Si se duplica el número de procesadores en un sistema, el rendimiento debería igualmente duplicarse. Esto no sucede en la realidad, ya que a partir de un cierto número de procesadores los excesivos intercambios de mensajes de control y de transferencia de datos entre los procesadores, provocan un efecto de saturación en el sistema, lo que genera automáticamente una degradación del rendimiento. Parece entonces evidente la relación entre la optimización del rendimiento y la minimizacion de los intercambios entre los procesadores. Existen otros aspectos que juegan un papel importante para optimizar el rendimiento en un sistema Distribuido/Paralelo, como es el equilibrio de la carga de trabajo, y la minimización del número de operaciones de entradas/salidas, entre otros. En este capítulo se estudia algunos de los aspectos relativos al problema de optimización de rendimientos para estas plataformas computacionales.

5.1. Introducción

En la programación paralela/distribuida, como en otras disciplinas de ingeniería, el objetivo del proceso de diseño es optimizar un conjunto de medidas, las cuales están en función del problema específico a resolver, como es el caso de los tiempos de ejecución, los requerimientos de memoria, los costos de implementación, los costos de mantenimiento, etc. En particular, el rendimiento de un programa paralelo/distribuido es un tema complejo y multifacético. Se deben considerar, en adición a los tiempos de ejecución, a la portabilidad y a la escalabilidad, los mecanismos para guardar datos, trasmitirlos a través de la red, moverlos desde/hacia los discos, etc. Así, las medidas de rendimiento pueden ser muy diversas. La importancia relativa de esas medidas varía de acuerdo a la naturaleza del problema. Incluso, algunas de ellas requieren que otras sean optimizadas o ignoradas. Por ejemplo, en el caso de un sistema de predicción de tiempo se requieren tiempos de ejecución mínimos y que la fidelidad del modelo sea maximizada con costos de implementacion bajas. Además, la escalabilidad para futuras generaciones de computadores es importante. En contraste, en un sistema de procesamiento de imágenes de manera encauzada, el número de imágenes que se procesan es fundamental, cuando se comprimen imágenes de videos, mientras que la latencia (duración de una imagen en cada fase del encauzamiento) es más importante, si el sistema es parte de un sensor, que debe reaccionar en tiempo real a eventos detectados en las cadenas de imágenes.

Es importante entender la manera en que las medidas de rendimiento son obtenidas y la manera de ser interpretadas. Parte de las preguntas a responder son:

- ¿Cómo se caracteriza el rendimiento de las aplicaciones y sistemas?
- ¿Cuáles son los requerimientos de los usuarios a nivel de rendimiento?
- ¿Cómo se mide el rendimiento?
- ¿Cuáles son los factores que afectan al rendimiento?

La computación paralela/distribuida define nuevas medidas de rendimiento, tales como unidades MIPS (Millones de Instrucciones por Segundo) y MFLOPS (Millones de Operaciones de Punto Flotante por Segundo). Por ejemplo, para un cierto programa, la correspondiente medida de rendimiento está dada en MIPS. Para aplicaciones científicas o de ingeniería, donde los cálculos numéricos dominan, una medida de rendimiento natural es el MFLOPS. Por otro lado, el pico de rendimiento de una máquina es la tasa de MFLOPS la cual debe estar garantizada por el fabricante del computador. Pero en general, el acceso a memoria, los pases de mensajes, las barreras de sincronización etc., no son considerados por estas unidades, por lo que esas medidas de rendimiento no son suficientes. Además, el número de procesadores a usar y el tamaño de los datos son escogidos independientemente por el usuario, según el cálculo a hacer. Todos estos factores tienen implicaciones en el rendimiento ya que ayudan a determinar el porcentaje de un programa secuencial que puede ser ejecutado en paralelo. Así, muchos factores, incluyendo la arquitectura, el software del sistema, la estructura de datos y las estructuras de los programas, afectan los rendimientos de los sistemas paralelos/distribuidos.

Una de las medidas de rendimiento más usada es la *aceleración*. Esta es calculada al dividir el tiempo para calcular una solución de cierto problema en un procesador, por el tiempo para calcular la solución usando N procesadores en paralelo. Cuando el costo-efectividad del cálculo es la medida a usar, más que la aceleración, se extienden los criterios de rendimiento para incorporar la noción de *capacidad de ejecución* (*throughput* en ingles). Este criterio no se relaciona directamente con supercomputación, pero tiene que ver con la ganancia en rendimiento a bajo costo, en sistemas multiprocesadores multitareas (ella se utiliza cuando múltiples trabajos son ejecutados simultáneamente). La capacidad de ejecución de un sistema es definida como el número de trabajos procesados por unidad de tiempo. Si sólo un trabajo es ejecutado en un momento, la capacidad de ejecución, es el recíproco del tiempo de ejecución. Por otro lado, los *puntos de referencia*, o *programas de comparación del rendimiento* (*benchmark* en ingles), son diseñados para medir el rendimiento de un computador bajo situaciones más o menos reales. En lo que resta de este capítulo, hablaremos de estos aspectos.

5.2 Aceleración y Eficiencia

Durante años, los conceptos de aceleración y eficiencia, han sido usados como medidas de rendimientos [1, 11, 23, 33]. A pesar de existir algunas dificultades al usar esas definiciones en ambientes reales (por ejemplo, para calcular la aceleración, cuando los datos son muy grandes para almacenarlos en las memorias locales de los procesadores en

un sistema a memoria distribuida, o cuando hay pocos procesadores), trabajan bien en muchas aplicaciones prácticas. La frase "Una implementación de un algoritmo para un problema de tamaño N=100 sobre un computador paralelo X con 12 procesadores, alcanza una aceleración de 10.8", puede ser interpretado de diferentes maneras. ¿Qué sucede con 1000 procesadores?, ¿Sí N=1000? ¿Cuándo el costo de comunicación es 10 veces más alto?. Finalmente, una simple medida de rendimiento sirve solamente para determinar el rendimiento en una situación, pero no es indicador del rendimiento en otras situaciones. Así, es importante resaltar que estos dos parámetros son valores específicos (es decir, para un cálculo dado sobre una máquina dada) y no necesariamente una función.

En 1967, Gene Amdahl, empleado de IBM, dijo:

"Desde hace más de una década los profetas han vociferado que las mejoras en la organización de un simple computador, han llegado a su límite, y que los próximos avances significativos serán hechos interconectando múltiples procesadores, de manera de permitir soluciones cooperativas".

Formalmente, la aceleración es la relación entre el tiempo de ejecución sobre un procesador secuencial y el tiempo de ejecución en múltiples procesadores. Esto es conocido como la "ley de Amdahl". Un aspecto importante a considerar es que todo programa paralelo tiene una parte secuencial que eventualmente limita la aceleración que se puede alcanzar en una plataforma paralela. Por ejemplo, si el componente secuencial de un algoritmo es *1/s* de su tiempo de ejecución, entonces la aceleración máxima posible que se puede alcanzar en el computador paralelo es *s*. Así, si el componente secuencial es 5%, la aceleración máxima que se puede alcanzar es 20.

Matemáticamente, la "ley de Amdahl" puede ser definida como [1, 11, 23, 30, 33]:

$$\mathrm{S(P,\,N)} = \frac{T(1, N)}{T(P, N)} \qquad (5.1)$$

donde T(1, N) es el mejor tiempo secuencial conocido para resolver un problema de tamaño N, y T(P, N) es el tiempo requerido en el sistema con P procesadores para resolver el mismo problema. A nivel del tiempo de ejecución paralelo, suponiendo que β es la fracción del programa que es serial, y 1-β es la fracción del programa que es paralelo, la parte secuencial puede ser computada en un tiempo igual a $\beta T(1)$ y la paralela en $(1-\beta)T(1)/P$, asumiendo un caso ideal de paralelismo. Así, la "ley de Amdahl" puede ser redefinida como:

$$\mathrm{S} = \frac{1}{\beta + \frac{(1-\beta)}{P}} = \frac{P}{\beta P + (1-\beta)} \qquad (5.2)$$

Además,

$$S = \frac{P}{1+(P-1)\beta} \rightarrow \frac{1}{\beta} \; cuando \; P \rightarrow \infty$$

La "ley de Amdahl" tiene varias implicaciones:

- Para una carga dada, la aceleración máxima tiene un borde superior de 1/β. Así, el componente secuencial del programa es un cuello de botella. Al aumentar β, la aceleración decrecerá proporcionalmente.
- Para alcanzar buenas aceleraciones, es importante reducir β.
- Cuando un problema contiene esas dos porciones, se debe ejecutar la más larga porción lo más rápidamente posible.

Por otro lado, la ideal eficiencia es:

$$E(P, N) = \frac{S(P, N)}{P} \qquad (5.3)$$

Formalmente, la eficiencia es definida como la fracción del tiempo que el procesador consume haciendo trabajo útil. Tanto la aceleración como la eficiencia son *relativas* cuando se definen con respecto al algoritmo paralelo, ejecutado en un simple procesador. La eficiencia y aceleración son *absolutas* cuando usan el mejor tiempo secuencial conocido de un algoritmo, para hacer esos cálculos. Claramente, una eficiencia de 100% significa que el problema se ejecuta en un tiempo con aceleración lineal.

Las fórmulas anteriores no toman en cuenta el sobrecosto por preparar las tareas paralelas, o cualquier degradación de rendimiento debido a sincronizaciones, o la comunicación entre procesadores, o la jerarquía de memoria, entre otras cosas. Así, en la realidad, para valores grandes de P y/o pequeño de β, las aceleraciones alcanzadas son muy lejanas de la aceleración ideal. En general, la "ley de Amdahl" es muy pesimista ya que predice solamente una mejora de 50% sobre un simple procesador, al usar una centena de procesadores, lo que ha hecho que muchos programadores eviten usar máquinas paralelas, así el programa paralelo tenga poca cantidad de parte secuencial (ver figura 5.1). El problema es que supone una situación, que no refleja la realidad, es decir, que el tamaño del problema no crece con el número de procesadores disponibles.

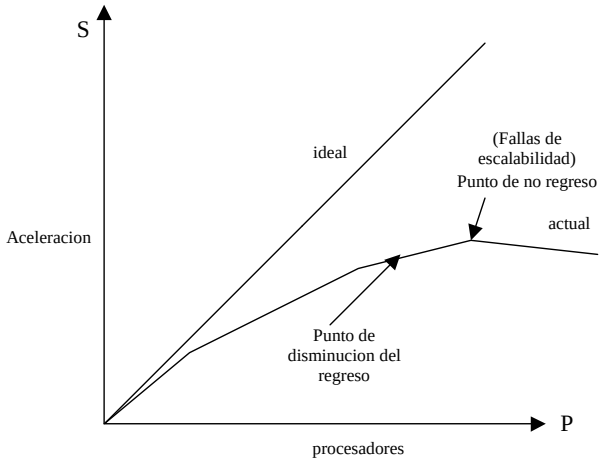


Figura 5.1. Rendimiento de Sistemas Paralelos (S vs P)

La figura 5.1 indica lo que se espera cuando más procesadores se usan en un cálculo dado. La aceleración no excede P, para un sistema P-procesadores, salvo que la computación sea cambiada. La pregunta es, ¿Qué tan cerca de esa línea puede estar un cálculo al crecer P?, y en particular, ¿Qué valor de P disminuye los rendimientos lejos de esa línea ideal?. La figura 5.1 muestra un punto que indica el número de procesadores, a partir del cual al añadir más procesadores al cálculo, no produce adicionales ganancias de rendimiento (punto de no regreso). Un sistema paralelo o distribuido moderno sirve múltiples tareas de un programa (por ejemplo, la ejecución simultanea de múltiples iteraciones de un lazo), pero al mismo tiempo permite servir a múltiples usuarios. Esto conlleva a la necesidad de planificar la asignación de cada tarea sobre el sistema, y en muchos casos, a esquemas de partición del sistema, impuestos por el sistema operativo, de manera de simplificar la asignación de los recursos. Esto genera redefinir las particiones permanentemente (por ejemplo, cuando el tamaño de un nuevo programa es grande y la actual partición del sistema que le es asignada es demasiada pequeña), lo que adiciona sobrecostos que hacen más lentas las ejecuciones de las tareas. En general, se deben añadir nuevos grupos de procesadores cuando el grupo existente, alcanza un punto, en el cual no se producen mejoras de rendimiento (ver el punto de disminución del regreso en la figura 5.2).

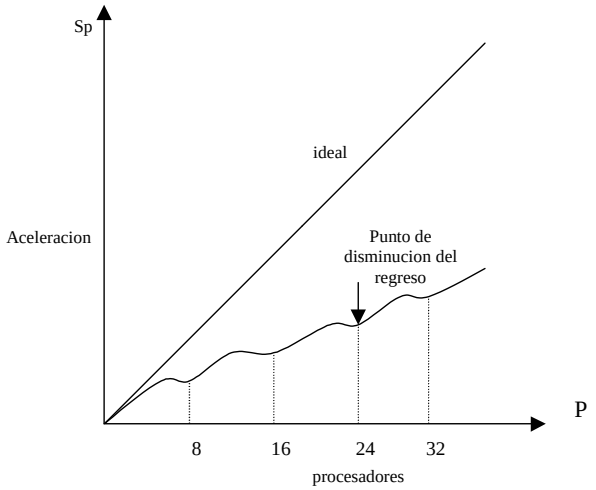


Figura 5.2. Cuando se puede escalar una arquitectura.

La "ley de Amdahl" es vista desde dos extremos: una consideración pesimista y otra optimista. La consideración optimista establece que la parte paralela del cálculo se ejecuta completamente en los P procesadores, y la pesimista es que el remanente se ejecuta en justo un procesador. Se puede suponer que las consideraciones optimistas y pesimistas de la "ley de Amdahl" se pueden balancear entre ellas: esto generaría el límite de rendimiento de los sistemas paralelos caracterizado por la curva de la figura 5.3. Veamos como se llega a ese límite. Al suponer que una fracción γ de T(1) puede completamente explotar P procesadores, mientras que los remanentes de T(1) corren en justo un procesador, se establece el siguiente límite

$$S(P) = \frac{T(1)}{\gamma T(1)/P + (1-\gamma)T(1)} = \frac{P}{\gamma + (1-\gamma)P} \leq \frac{1}{1-\gamma} \qquad (5.4)$$

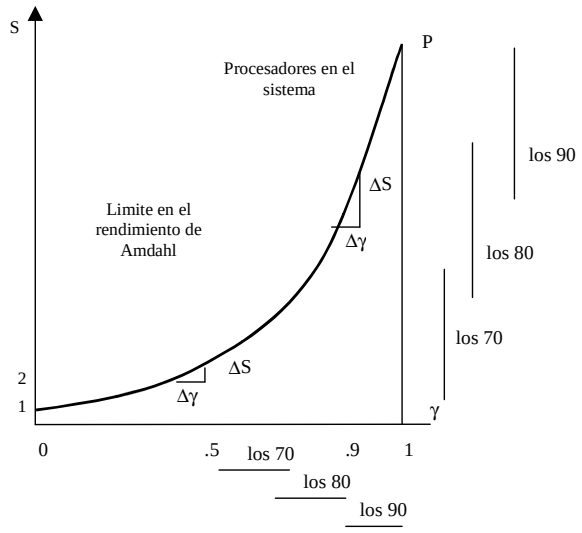


Figura 5.3. Límite de la aceleración paralela de la "ley de Amdahl" y su desarrollo en el tiempo

Históricamente, la figura 5.3 es usada para indicar que el procesamiento paralelo es difícil, si no una imposible alternativa para hacer más rápido a los sistemas secuenciales. En particular, en los 60 y 70, la forma de la curva era un gran reto, para los entusiastas en procesamiento paralelo. Como lo muestra esa figura, cuando γ era cercano al valor .5 y la inigualdad de la ecuación de aceleración era $S \leq 2$; un cambio razonable de $\Delta\gamma$ daba solamente una mejora pequeña en la aceleración (ΔS). Sin embargo, en los últimos años mejoras en el procesamiento paralelo han sido hechas en muchos frentes, y hoy los cálculos operan frecuentemente con γ alrededor de 0.9. Con esta forma, el mismo (o más pequeño) $\Delta\gamma$ que hace 20 años realizaba una diferencia de rendimiento poco significativa, puede causar en un sistema nuevo con 16 o 32 procesadores, doblar o cuadruplicar su rendimiento. En general, para cualquier tamaño de P, $S \leq 1/(1-\gamma)$ define el borde de la ecuación de aceleración (Por la inigualdad, sí γ=0.990, este modelo llega al borde superior de aceleración).

Desde los años 60 a los 80, la "ley de Amdahl" ha sido una guía para los diseñadores de sistemas de procesamiento paralelo, mientras se estuvo en la porción baja de la curva. A partir de los 90, este modelo es un estímulo para los diseñadores de sistemas paralelos, para escalar la aceleración a la porción alta de la curva.

La figura 5.4 muestra un estudio de la carga de trabajo en el sistema. Se observa que en los procesadores vectoriales, la longitud del vector crece con N, hasta que las unidades de encauzamientos son saturadas. En el caso de los procesadores paralelos, estos son explotados hasta que el propio sistema esté saturado. Esto ocurre cuando los procesadores accesan la memoria cache para muchos de sus datos, pero al aumentar el tamaño del problema, la memoria cache es un cuello de botella, ya que muchos de los datos son acccesados desde la memoria principal. Además, al exceder el tamaño de los datos al tamaño físico de la memoria principal, puede ocurrir que existan accesos a la memoria virtual, lo que implica accesos a las unidades de discos, conllevando a que el sistema de

entrada/salida determine el rendimiento del sistema. Estas características de rendimiento son también típicas de los sistemas secuenciales. La clave es explotar el rango de rendimiento óptimo, lo cual puede hacerse a nivel del software. Por ejemplo, para tamaños de datos grandes, se debe explotar la localidad de referencia en ellos. Así, la caída de rendimiento causada por el tamaño de la memoria cache puede evitarse al obligar que el código repetidamente referencie los mismo datos en la memoria cache. Otro ejemplo es que la caída de rendimiento, por el tamaño de la memoria principal, se evita reestructurando el cálculo y focalizándose en conjuntos de páginas de datos de la memoria virtual, que pueden simultáneamente colocarse en la memoria principal. Igualmente que en las localidades de la cache, los programas poseen fases que consisten en conjuntos de trabajos sobre paginas en la memoria principal que satisfacen todos los requerimientos de acceso en dicha fase, y después, entonces, moverse a una nueva fase.

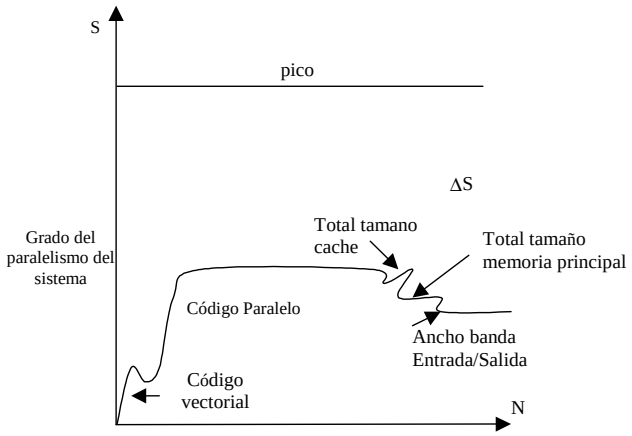


Figura 5.4. Saturación arquitectónica *S vs. N*

Compiladores y librerías modernas buscan hacer usos eficientes de la jerarquía de memoria, con estrategias como las planteadas anteriormente, buscando que los programas hagan referencias a memorias más rápidas para acelerar sus ejecuciones. Al dejar que lo hagan los usuarios, aun a través de extensiones en lenguajes amigables, les coloca un enorme peso a ellos. A pesar de que las librerías y los compiladores modernos pierden potenciales operaciones concurrentes, normalmente los usuarios no tienen la capacidad para detectar dichas perdidas. Aún peor, les resulta difícil portar los programas a través de nuevas generaciones de sistemas, ya que requieren permanentemente hacer importantes reestructuraciones en sus programas.

Ahora extenderemos la "ley de Amdahl" con un sobrecosto de comunicación (To):

$$S = \frac{T(1)}{\beta T(1) + (1-\beta)T(1)/P + To} = \frac{P}{1 + (P-1)\beta + \frac{PTo}{T(1)}} \rightarrow \frac{1}{\beta + \frac{To}{T(1)}} \text{ cuando } P \rightarrow \infty$$

Esta extensión indica que no solamente se debe reducir el cuello de botella de la parte secuencial, sino que además se debe aumentar la granularidad promedio, de manera de reducir el impacto adverso del sobrecosto.

Un aspecto interesante, es que al aumentar el tamaño de la máquina para tener más poder de cálculo, se puede aumentar la complejidad del problema, de manera de crear más carga de trabajo, lo que puede producir una más exacta solución, aun guardando, sin cambiar, el tiempo de ejecución. Ejemplos son los métodos de elementos finitos para ejecutar análisis estructurales, o los métodos de diferencia finita para resolver problemas computacionales de dinámicas de fluidos. Así, al hacer más finos los cálculos, ya sea aumentando los pasos de tiempos o reduciendo los espacios en la malla, conllevará a aumentar la carga de trabajo, con la posibilidad de obtener mejores resultados. En esos casos, el motivo no es disminuir los tiempos de ejecución, sino producir mejores resultados (más exactos). La "ley de Amdahl" es inválida en estos casos. Este problema de escalar por exactitud es lo que motivó a Gustafson a desarrollar su modelo de aceleración a tiempo fijo, conocida como la "ley de Gustafson" [30]. La idea consiste en que el problema escalado, guarde todos los recursos aumentados ocupados, dando como resultado una mejor utilización del sistema.

Los orígenes de esta última ley son descritos a continuación: la pesimista predicción de la "ley de Amdahl" disminuyó el interés por estas plataformas, por lo que apareció un premio llamado el premio de Korp que tenia por finalidad premiar con US$ 1000 a quien realizara exitosos programas paralelas para resolver problemas reales, los cuales debían alcanzar aceleraciones de 200, sin considerar el número de procesadores usados. En 1988, un equipo del laboratorio Sandia ganó el premio, revirtiendo las predicciones de la "ley de Amdahl". J. Gustafson y E. Barsis, dos de los investigadores de Sandia quienes compartieron el premio, derivaron una alternativa a la "ley de Amdahl" para explicar cómo ellos podían hacer algo que era imposible según esa ley. La clave es que β y N no son independientes. Gustafson dijo: "la expresión (para la "ley de Amdahl") contiene la implícita suposición que $(1-\beta)$ es independiente de P, lo que nunca es el caso". Ellos partieron de la misma fórmula de aceleración $S=T(1)/T(P)$ e interpretaron la fórmula de la siguiente manera: Si un procesador es usado, tanto la parte serial como la paralela es calculada como:

$$T(1)= \beta+(1-\beta)P \qquad (5.5)$$

Pero si P procesadores paralelos son usados, el problema puede ser escalado de la siguiente forma:

$$T(P)=\beta+(1-\beta)=1 \qquad (5.6)$$

Sustituyendo a T(P) y T(1) en la fórmula de aceleración, da la "ley de Gustafson-Barsis"

$$S = P-(P-1)\beta \qquad (5.7)$$

Así, contrario a la "ley de Amdahl", en este caso T(1) es el tiempo para calcular tanto la fracción serial como paralela del programa, en un procesador simple. Esta ley dice que cierto tipo de paralelismo puede vencer a la "ley de Amdahl", en particular, los modelos de paralelismo tipo SIMD y SPMD se adaptan bien a esta ley. Esta fórmula aún ignora los costos de comunicación, los sobrecostos asociados a las funciones del sistema operativo (creación de procesos, manejo de la memoria, etc.).

A continuación los detalles de deducción de esta ley. En el supuesto de que un programa dura W en una máquina paralela, al escalar el tamaño de la máquina, la idea es que el programa paralelo siga ejecutándose en la misma cantidad de tiempo, por lo que se debe escalar la carga de trabajo a W'=βW+(1-β)PW. Así, la ejecución paralela será ejecutada en P nodos aun en W unidades de tiempo. Sin embargo, el tiempo secuencial para ejecutar la carga de trabajo escalada será W'. Además, la aceleración con la carga de trabajo escalada será ahora definida como:

$$S' = \frac{\text{tiempo secuencial de la carga trabajo}}{\text{tiempo paralelo de la carga trabajo}} = \frac{\beta W + (1-\beta)PW}{W} = \beta + (1-\beta)P$$

La "ley de Gustafson" puede ser explicada como la aceleración para un tiempo fijo, la cual es una función lineal de P si la carga de trabajo es escalada para mantener el tiempo de ejecución fijo. De esta manera, cuando el problema es escalado para aprovechar el poder de cálculo disponible, la fracción secuencial deja de ser un cuello de botella. Note que para que sea válida la "ley de Gustafson", es importante escalar la porción paralela de la carga de trabajo desde (1-β)W a (1-β)PW, mientras que la porción secuencial se mantiene igual (βW). Al incorporar los sobrecosto a la "ley de Gustafson", la aceleración escalada sería:

$$S' = \frac{\beta W + (1-\beta)PW}{W + To} = \frac{\beta + (1-\beta)P}{1 + To/W} \qquad (5.8)$$

La "ley de Gustafson generalizada" puede alcanzar una aceleración lineal al definir el sobrecosto como una función decreciente con respecto a P. Pero esto es difícil de alcanzar.

X. Sun y L. Ni desarrollaron un modelo de aceleración que generaliza las leyes de Amdahl y Gustafson, de manera de maximizar el uso de las capacidades de CPU y de memoria. Este modelo ataca el problema del límite de la memoria, el cual aparece en aplicaciones científicas y computacionales que requieren grandes espacios de memoria. De hecho, muchas aplicaciones de cálculo paralelo son aplicaciones con necesidades de memoria más que con necesidades de CPU o de Entradas/Salidas. Esto es un problema en los sistemas a memoria distribuida, donde la memoria local en cada nodo es relativamente pequeña por lo que cada nodo puede manejar solamente un pequeño subproblema. Cuando un gran número de nodos son usados colectivamente para resolver un problema, la capacidad total de memoria aumenta proporcionalmente. Esto permite al sistema resolver problemas escalados a través de la descomposición del dominio de los

datos. Así, en vez de guardar los tiempos de ejecución fijos, la idea es usar toda la memoria aumentada escalando el problema.

A continuación veamos en detalle lo antes dicho. Al suponer que cada nodo tiene una capacidad de memoria de tamaño M, un sistema paralelo con P nodos tendrá PM como capacidad de memoria total. Si además se supone que la carga de trabajo de un cálculo secuencial es W', tal que W'=βW + (1-β)W, al tener P nodos, es decir, con una capacidad de memoria PM, y al asumir que la porción paralela de la carga de trabajo se ha escalado a G(P), entonces la carga de trabajo de un programa paralelo se escalaría a: W'=βW + (1-β)G(P)W. El factor G(P) refleja el aumento en carga de trabajo, cuando la capacidad de memoria aumenta P veces. El borde de memoria para la aceleración es dado por:

$$S' = \frac{\text{tiempo secuencial de la carga trabajo}}{\text{tiempo paralelo de la carga trabajo}} = \frac{\beta W + (1-\beta)G(P)W}{\beta W + (1-\beta)G(P)W / P} = \frac{\beta + (1-\beta)G(P)}{\beta + (1-\beta)G(P)/P} \qquad (5.9)$$

Hay cuatro casos especiales de esta ley:

- G(P)=1, corresponde al caso donde el tamaño del problema es fijo. Así, el borde superior de la aceleración es equivalente al de la "ley de Amdahl".
- G(P)=P, esto es cuando la carga de trabajo aumenta P veces al aumentar la memoria P veces. Este caso es equivalente a la "ley de Gustafson".
- G(P)>P, corresponde a la situación donde la carga de trabajo computacional aumenta rápidamente.
- 1< G(P)<P es el caso cuando la carga de trabajo aumenta pero en una magnitud menor a la capacidad del sistema, de tal forma de no sobrecargarlo.

Gelenbe analizó la "ley de Amdahl" para entender la aceleración en términos más precisos [30]. En particular, él definió límites en el potencial de mejoras de rendimientos que se podían obtener. El límite superior asume perfecciones, mientras que el inferior disminuye la utilización de los procesadores, cuando el número de estos crece. Ambas definiciones de límites se definen como:

$$P/\log_2 P \leq S \leq P \qquad (5.10)$$

El límite inferior se obtiene al asumir que un problema puede ser resuelto en un tiempo proporcional a P en un simple procesador, y proporcional a $\log_2$ P en un procesador paralelo. El límite superior es el caso ideal, se dice que es una aceleración lineal. Si existe una aceleración que supere ese límite se llama una aceleración superlineal, excediendo el factor de P para P procesadores, y significa que el algoritmo paralelo es superior al mejor código secuencial, aun ejecutándose en un procesador. Estas suposiciones son irreales en máquinas reales, hay varias razones para ello: las fuentes de desequilibrio de la carga de trabajo en los procesadores, la ineficiencia de las comunicaciones entre los procesadores, la capacidad de memoria que se posee, etc. En la figura 5.5 vemos los dos límites de paralelismo.

Gelenbe también sugirió una fórmula de aceleración para aplicaciones intensivas en comunicación (los tiempos de comunicación dominan a los tiempos de ejecución), dando una expresión, donde el tiempo de comunicación es función del número de procesadores. Así,

$$S=1/C(P) \qquad (5.11)$$

donde C(P) es alguna función del sobrecosto comunicacional entre P procesadores. Por ejemplo, C(P) puede ser igual a 0.1P+0.9.

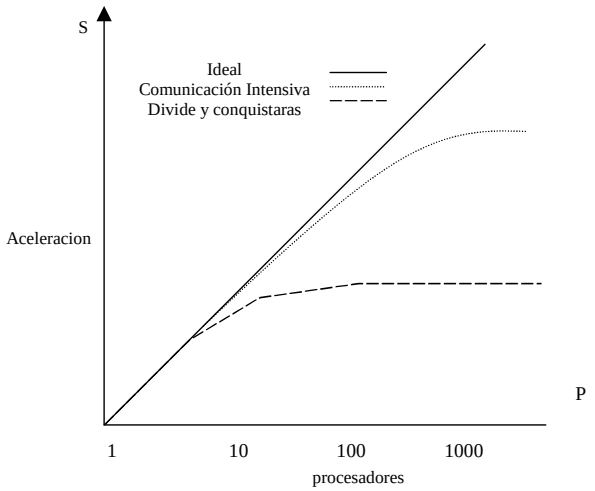


Figura 5.5. Curvas de la aceleración que miden la utilidad de la computación paralela.

El paralelismo ideal es raro, y no asume ningún sobrecosto debido a la comunicación o a la naturaleza serial de las aplicaciones. El tamaño del problema es aumentado en estos casos por la programación del paralelismo de datos. Un paralelismo $S=P/\log_2 P$ es para problemas que tienen como flujo de ejecución a árboles binarios (como el problema divides y conquistarás). Las hojas del árbol representan tareas a correr en paralelo. A medida que se sube en el árbol, el número de tareas paralelas decrece hasta ejecutarse una sola tarea en la raíz. Así, la aceleración es limitada por el logaritmo del número de procesadores (ejemplos son algunos de los algoritmos de búsqueda, ordenamiento, etc.). Finalmente, para S=1/C(P) se considera que C(P) es una relación inversa (A+B/P) o logarítmica ($A+B\log_2 P$), donde A y B son constantes. Si A y B son grandes, es posible que un programa paralelo se ejecute más lento que su equivalente secuencial (cálculo de matrices donde grandes porciones de ellas deben pasar entre los procesadores o entre las unidades de Entrada/Salida y las memorias locales constantemente, o programas donde los datos son globales para todos los procesadores, o cuando el tamaño de la memoria es pequeño con respecto a lo que requiere una aplicación).

Se debe recalcar que al usar la aceleración como criterio de rendimiento, se entiende como los sistemas paralelos responden a cambios en la arquitectura y en la carga de trabajo. Para ese análisis se hacen varias suposiciones. Por ejemplo, si un procesador

contiene una memoria cache, entonces P procesadores deben tener P memorias cache, o quizás lo suficiente para contener todos los datos para el cálculo, y así poder computar con un rápido y efectivo tiempo de acceso a la memoria.

5.3. Escalabilidad

Un importante aspecto del análisis de rendimiento es estudiar cómo el algoritmo rinde al variar parámetros tales como tamaño del problema, costo para iniciar el envío de mensajes, etc. En particular, un aspecto a evaluar es la escalabilidad de un algoritmo paralelo, es decir, cómo efectivamente él puede usar un número mayor de procesadores [24, 25]. Una manera de cuantificar la escalabilidad es determinar cómo el tiempo de ejecución T y la eficiencia E varía al aumentar el número de procesadores P para un tamaño del problema y parámetros de máquina fijos. Esto permite responder a preguntas, tales como, ¿ Cuál es el número de procesadores más grande que se pueden usar si se debe mantener una eficiencia de 50%?. La escalabilidad puede ser limitada por los costos de comunicación, los tiempos de ocios, los cálculos replicados, etc. El análisis de escalabilidad no tiene sentido para todos los problemas. Por ejemplo, en ciertas aplicaciones el escalamiento no es posible por las restricciones físicas del tamaño del problema. En esta sección se presentan tres modelos de escalabilidad, los cuales caracterizan cómo un sistema paralelo escala, mientras se mantiene una buena eficiencia y aceleración en el sistema [17].

- *Modelo de Isoeficiencia:* es una medida que caracteriza la escalabilidad del sistema, donde la eficiencia de cualquier sistema es expresada como una función de la carga de trabajo (W) y del tamaño de la máquina (P), así E=f(W,P). Si se fija la eficiencia como una constante y se usa la ecuación de E para estudiar como escalar W al variar P, la función resultante es una función isoeficiente. Entre más pequeña es la Isoeficiencia, menor es la carga de trabajo requerida para que un aumento del tamaño de la máquina guarde la misma eficiencia. Así, será mejor la escalabilidad del sistema. En general, un algoritmo con una función lineal de isoeficiencia O(P) es altamente escalable, ya que la cantidad de cálculo necesita crecer solamente linealmente con respecto a P para guardar la eficiencia constante. En contraste, un algoritmo con una función de Isoeficiencia cuadratica o exponencial es pobremente escalable.

  Ejemplo 5.1: Dado dos esquemas de multiplicaciones de matrices A y B para el mismo número de procesadores. Los tiempos de ejecución paralelos y las eficiencias de estos esquemas son mostrados en la tabla 5.1.

Esquema	Tiempo de Ejecución Paralelo (Tp)	Eficiencia (E)
A	$cN^3/P+bN^2/\sqrt{P}$	$1/(1+(b\sqrt{P})/(cN))$
B	$2cN^3/P+bN^2/(2\sqrt{P})$	$1/(2+(b\sqrt{P})/(2cN))$

Tabla 5.1. Tiempo Paralelo y Eficiencia para dos esquemas de multiplicación de Matrices.

donde b y c son constantes. En cada expresión del tiempo de ejecución paralelo, el primer término es el tiempo computacional y el segundo el sobrecosto comunicacional. Aproximadamente, la diferencia de los dos sistemas es que B tiene sólo la mitad del sobrecosto del sistema A, pero a expensas de doblar el tiempo computacional. Ahora, ¿Cuál es más escalable?. Para eso tomemos:

- E=1/3, en este caso la función de isoeficiente de A es $W(A)=(b/2c)3P^{1.5} = 2$. Similar es la expresión para el sistema B. Así, ambos son sistemas igualmente escalables.
- E=1/4, en este caso la función de isoeficiente de A es $W(A)=(b/3c)3P^{1.5}$ y de B es $W(B)=(b/4c)3P^{1.5}$. Así, B tiene una más pequeña isoeficiencia, por consiguiente, es más escalable que A.

La isoeficiencia es una herramienta poderosa para predecir la tasa de crecimiento de la carga de trabajo, con respecto al aumento del tamaño de la máquina. Sin embargo, esta medida se debe usar con cuidado, ya que en un sistema más escalable según la isoeficiencia, no implica necesariamente que corre más rápido. La isoeficiencia implica que responde bien a un mayor número de procesadores (eficiencia), pero no indica cuanto será el tiempo de ejecución, para procesar esa carga extra.

- *Modelo de Isoaceleración*: El concepto es básicamente igual al anterior, pero en lugar de mantener una eficiencia constante, se puede preservar una aceleración constante, mientras escala tanto al tamaño de la máquina como al tamaño del problema, al mismo tiempo.

- *Modelo de Isoutilización*: Una medida ideal de escalabilidad debe tener dos propiedades: i) Predecir la tasa de crecimiento de la carga de trabajo requerida, al aumentar el tamaño de la máquina; y ii) Ser consistente con el tiempo de ejecución. ¿Hay alguna medida con esas dos propiedades? Sí la hay. Supongamos que la utilización de un sistema paralelo puede ser definida como U=W/(PTpPpico), donde Ppico es el máximo rendimiento que se puede obtener en la plataforma en uso. Si se fija la utilización como una constante (por ejemplo, 25%) y se usa la ecuación anterior para estudiar como escalar W al variar P, la función resultante se llama función de isoutilización del sistema. Sistemas con pequeñas isoutilizaciones son más escalables que aquellos con grandes isoutilizaciones, y tendrán un tiempo de ejecución más corto.

Ejemplo 5.2. Calcular la isoutilización para los esquemas de multiplicación de matrices en paralelo del ejemplo 5.1:

$$\mathrm{U} = \frac{W}{PT_p(A)P_{pico}} = \frac{N^3}{P\left(cN^3/P + bN^2/\sqrt{P}\right)P_{pico}} = \frac{1/(cP_{pico})}{1+(b\sqrt{P})/(cN)} \qquad (5.12)$$

Para una utilización fija $U=1/(4cP_{pico})$, tenemos en el esquema A que

$$b\sqrt{P}/(cN) = 3 \Rightarrow W(A) = (b/(3c))^3 P^{1.5}$$

Similarmente, la utilización del esquema B es

$$\mathrm{U} = \frac{W}{PT_p(B)P_{pico}} = \frac{N^3}{P\left(2cN^3/P + bN^2/(2\sqrt{P})\right)P_{pico}} = \frac{1/(2cP_{pico})}{1+(b\sqrt{P})/(4cN)} \qquad (5.13)$$

Para la misma cantidad fija de utilización anterior, tenemos

$$b\sqrt{P}/(4cN) = 1 \Rightarrow W(B) = (b/(4c))^3 P^{1.5}$$

Como W(B)<W(A), el sistema B es más escalable.

En general, los modelos de escalabilidad permiten responder a:

- ¿Qué tan adaptable es el algoritmo a diferentes tipos de arquitectura?
- ¿Qué tan sensible es a los parámetros de la máquina?

Si la escalabilidad sugiere que el rendimiento es pobre para ciertos tamaños del problema, o para ciertas computadoras, se pueden usar modelos para identificar fuentes de ineficiencia, y así, áreas donde un algoritmo puede ser mejorado.

5.4 Complejidad Algorítmica Paralela

Una forma de medir la complejidad de un algoritmo secuencial, para resolver un problema dado, es a través del número de operaciones que él ejecuta [1, 13]. Por ejemplo, dada dos matrices cuadradas n*n clásicamente se requieren del orden de n^3 operaciones escalares para multiplicar dos matrices de ese tipo, es decir, la complejidad es $O(n^3)$. La notación O(g) es una función que define a *g* como el límite superior del número de operaciones para resolver un problema dado. Decir que un algoritmo que resuelve un problema según una complejidad O(g) significa que el número de operaciones efectuadas por el algoritmo tiende a *g* asimptoticamente como el límite superior. Por ejemplo, es útil pensar que un algoritmo que para una entrada *n* tiene una complejidad de O(nlog(n)) necesita alrededor de nlog(n) operaciones. Así, para un programa secuencial la complejidad algorítmica es una forma de medir la calidad de dicho algoritmo. En el caso

de los algoritmos paralelos, el número de procesadores es un nuevo parámetro, por lo que se necesitan nuevas definiciones de complejidad.

En la programación secuencial, la *Máquina de Turing* (y sus variantes) normalmente se usa como modelo de cálculo secuencial. De manera práctica, el modelo RAM (Random Access Machine) es una adaptación de la *Máquina de Turing*. Para el caso paralelo se usa una versión del mismo llamada modelo PRAM (Parallel Random Access Machine). Este es el modelo teórico de máquinas paralelas más conocidos, usado para la paralelización abstracta de problemas y evaluación de la complejidad de los algoritmos paralelos. El modelo PRAM no considera al mismo tiempo al cálculo como a las comunicaciones en todas sus formas, ya que es demasiado complejo para ser usado eficazmente. Este es el caso de casi todos los modelos de cálculo paralelo, normalmente, no se considera la parte comunicacional, por esto casi todos se basan en máquinas paralelas a memoria compartida.

Como se ve, la máquina PRAM es una simplificación de las máquinas reales, pero para muchos casos es un modelo suficiente, porque es bastante próximo a éstas, para estudiar la complejidad de los programas paralelos. El estudio consiste en extraer el paralelismo máximo posible para un algoritmo dado en el modelo PRAM. Esto permite clasificar los algoritmos según sus complejidades paralelas. Dichos modelos de complejidad son normalmente usados para mostrar que un algoritmo dado es óptimo, o para determinar la complejidad mínima de un problema. Tales estudios pueden ser de dos formas: para establecer el límite superior del número de operaciones necesarias para resolver un problema dado, y para determinar si el algoritmo dado es óptimo (si es del mismo orden que del límite inferior, es óptimo).

Una PRAM es un conjunto de computadores secuenciales independientes (RAM) que tienen memoria privada/local y se comunican con los otros al usar una memoria global que comparten. Es decir, el modelo PRAM es un conjunto de unidades de cálculo sincronas (del tipo modelo RAM) que se comunican vía una memoria compartida. En cada ciclo del reloj, cada procesador puede hacer: un cálculo con los datos en su memoria local, una lectura en una localidad en la memoria global, o una escritura de un dato en la memoria global. El acceso simultáneo de varios elementos a una misma localidad de memoria puede ser hecho de varias maneras, dando origen a diferentes tipos de máquinas PRAM que veremos más adelante. En general, en un modelo PRAM se tiene:

- Un conjunto ilimitado *p* de procesadores P_1, … P_p indexados, donde cada procesador conoce su índice (que es único).
- Cada procesador tiene un controlador y una memoria local no accesible al resto de los procesadores.
- Una memoria global compartida. La memoria puede ser accesible en lectura y escritura, y es descompuesta en *m* localidades de memorias M_1, … M_m.
- Un programa PRAM consiste de una secuencia finita de instrucciones (operaciones atómicas) etiquetadas (lectura, escritura en memoria, estructura condicional, etc.). La máquina tiene dos operaciones adicionales: “halt” para terminar la ejecución del

proceso ejecutándose, y "fork" que al ser llamado por un procesador Pi arranca el cálculo en un procesador Pj, colocando en Pj el acumulador de Pi.
- Cada procesador puede leer o escribir en una localidad *i* de memoria (read (M_i) y write (M_i)) en una operación atómica que toma una unidad de tiempo. Estas operaciones son realizadas sincrónicamente (es decir, todos los procesadores leen, escriben o computan al mismo tiempo).

Como la memoria global es compartida, pueden ocurrir conflictos entre los procesadores al tratar de accesar simultáneamente al mismo lugar de memoria, ya sea en lectura o escritura. Al trabajar la PRAM sincrónicamente, puede darse cuenta en cada paso de cálculo de esta situación. Para modelar esto, se han introducido diferentes clases de PRAM que se distinguen entre ellas por el hecho de autorizar o no el acceso concurrente en modo lectura o escritura. Si un solo procesador puede leer en una localidad es ER (Exclusive Read), si varios pueden leer concurrentemente la misma localidad de memoria es CR (Concurrent Read). Igualmente para el caso de escribir, EW (Exclusive Write) si sólo se puede escribir en una localidad de memoria en un momento dado y CW (Concurrent Write) en el caso concurrente. Así, generalmente los modelos PRAM usuales son las posibles combinaciones en el acceso a memoria: EREW, CREW, ERCW, y CRCW con varias variantes para el caso CW (aleatorio, prioritario, etc.):

EREW: prohibe que dos procesadores puedan accesar la misma localidad de la memoria global al mismo tiempo (concurrentemente). Es el modelo más restrictivo, pero el más cercano a las máquinas reales (en la práctica, es la única forma de hacer accesos a memoria).

CREW: este modelo permite a cualquier procesador accesar simultáneamente la misma localidad de la memoria compartida, sólo para leer, y prohibe el acceso concurrente para escritura.

ERCW: prohibe las lecturas concurrentes, pero permite las escrituras simultáneas en una misma localidad de la memoria global. Eso obliga a especificar cual es el resultado después que varios procesadores escriben simultáneamente, en la misma localidad de la memoria global. Hay varias formas de resolver ese conflicto, las más comunes:

- *Modelo común*: todos los procesadores escriben el mismo valor en la localidad de la memoria global.
- *Modelo aleatoria*: cualquier cosa puede suceder al escribir, es decir, cualquier procesador ganá (cualquier dato de uno de ellos es el que se guarda). Así, el valor de alguno de los procesadores es almacenado en la memoria, sin saber el de cual (arbitrario).
- *Modelo prioridad*: los procesadores son clasificados según el orden de prioridad, y sólo el valor del procesador con más alta prioridad, entre todos los que quieren escribir a la misma localidad de memoria, será el que queda escrito en esa localidad de memoria.

- *Modelo fusión*: El valor de la memoria es el resultado obtenido luego de aplicar una función asociativa, tal como la adición o el máximo, sobre la lista de los valores escritos por los diferentes procesadores.

CRCW: es una generalización del anterior. Por lo tanto, no tiene sentido tener una máquina ERCW. Cualquier procesador puede accesar la misma localidad de memoria al mismo tiempo, ya sea para leer o para escribir.

Esta clasificación no considera el caso, cuando un procesador accesa una localidad de memoria en modo lectura y otro accesa la misma localidad de memoria en modo escritura. Se puede considerar en este caso que un procesador lee el valor anciano antes que el otro escriba uno nuevo, o que el procesador lee el nuevo valor, o cualquiera de las dos situaciones (es la más usada). En general, estos modelos son muy cercanos y se puede pasar de uno a otro fácilmente. Solamente EREW y CREW tienen utilidad, y sólo EREW se usa en máquinas reales.

El funcionamiento de una máquina PRAM es el que sigue: se inicializan todos los contadores de los procesadores. Después, en cada paso, todas las unidades ejecutan la instrucción correspondiente a su contador. La noción de paso corresponde a la hipótesis síncrona con duración unitaria de las operaciones. Este modelo es válido para máquinas paralelas a memoria compartida, como para máquinas distribuidas con una red de conexión completamente conectada. En el caso de la memoria distribuida, el modelo PRAM se debe modificar para modelar el acceso a memorias remotas (DRAM) usando una red de interconexión. Una DRAM es un conjunto de procesadores P_i , un conjunto de localidades de memoria M_j y un conjunto de pasos sincronizados $X_f(k,l)$, tal que $k \in P_i$ y $l \in M_j$. X_f es una operación (x(k,.)) lo que implica que P_i puede accesar a cualquier memoria en cualquier sitio.

Según la literatura, es difícil de determinar qué tipo de algoritmos se pueden desarrollar bajo este modelo, ya sea para máquinas SIMD o MIMD. Se supone aquí que son del tipo SIMD. En el siguiente ejemplo, se parte de un algoritmo secuencial para generar uno PRAM.

Ejemplo 5.3: Calcular el máximo de un conjunto de enteros. Se tienen $n=2^m$ números enteros, además, se usa $\sum_{i=0}^{m} 2^i = 2n-1$ localidades de memoria numeradas de 1 a 2n-1 y llamadas a[i]. La localidad a[1] guarda el resultado final, las localidades a[2] a la a[n-1] los valores intermedios y las localidades a[n] a a[2n-1] los números enteros. Esto nos da el siguiente algoritmo secuencial:

Para k=m-1, 0
    Para j=2^k, 2^{k+1} -1
        a(j)=max(a(2j),a(2j+1))

Este algoritmo necesita O(n) pasos. La paralelización del código anterior consiste en ejecutar en paralelo el lazo interno, por implica cálculos independientes en cada una de sus iteraciones. Para eso, se cambia el índice *j* por $i+2^k$, lo que da el código:

Para k=m-1, 0
Para i=0, 2^k-1
$a(i+2^k)=max(a(2(i+2^k)), a(2(i+2^k)+1))$

El lazo interno ahora se puede reemplazar por una ejecución paralela sobre $p=2^k$ procesadores, tal que *i* es el índice de los procesadores. Al no existir ningún conflicto de memoria (ni a nivel de escritura ni a nivel de lectura), este algoritmo se puede ejecutar sobre una PRAM EREW (ver figura 5.6).

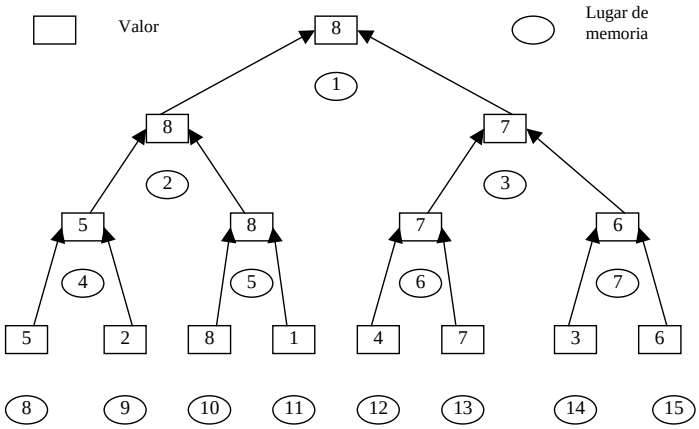


Figura 5.6 El árbol del cálculo del máximo del ejemplo 5.3.

El número máximo de procesadores activos en un momento dado es $2^{m-l}=n/2$, lo que da un orden de complejidad igual a O(n). Además, se requieren m=log(n) etapas de cálculo (cada etapa es una llamada del lazo), ejecutándose el algoritmo en forma de un árbol muy típico usado en paralelismo.

No sólo el número de etapas es un criterio para evaluar un algoritmo paralelo. Existen otros criterios para medir la calidad de un algoritmo paralelo. En los modelos secuenciales RAM, dos criterios son normalmente considerados para evaluar un programa: uno físico y otro temporal. Usaremos la siguiente notación: P es el problema y P(n) una instancia de P de tamaño n. Un algoritmo secuencial que resuelve el problema P(n) es llamado As(n).

- El *tiempo secuencial* Ts(As(n)) es definido como el número de operaciones elementales ejecutadas sobre un computador dado (por ejemplo, las operaciones de punto flotantes, las lecturas y escrituras en memoria, etc.).
- El *espacio de memoria* S(As(n)) es definido como el número de localidades de memorias necesarias para la ejecución del algoritmo.

Por analogía, la calidad de un programa paralelo Ap al resolver el mismo problema P usando el modelo PRAM, se basa en la relación entre el tiempo de cómputo en la máquina paralela y el espacio usado (número máximo de procesadores ocupados). Usaremos la siguiente notación: Ap(n) es el algoritmo paralelo que resuelve el problema P(n).

- La *superficie* H(Ap(n)) es definida como el número máximo de procesadores utilizados en cada uno de los pasos del algoritmo Ap(n) (Ap(n) es el algoritmo paralelo que resuelve el problema P(n)). En cada paso de cálculo paralelo, cada procesador ejecuta, o no, una instrucción del algoritmo.
- El *tiempo* Tp(Ap(n)) es definido como el número de pasos necesarios para la ejecución del algoritmo con H(Ap(n)) procesadores.

Así, podemos definir a Op(Tp(Ap(n)), H(Ap(n))) como la superficie (H(Ap(n))) y el tiempo (Tp(Ap(n))) requerido por el algoritmo paralelo Ap(n). Para los algoritmos paralelos también se puede cuantificar el espacio de memoria usado. Pero los dos primeros criterios (número de procesadores y tiempo de ejecución) para este caso son más relevantes. Otro criterio que nos interesa para este modelo, es la noción de trabajo. El *trabajo* de un algoritmo paralelo Ap sobre un problema P(n) es la cantidad Wp(An) definida por

$$Wp(An)=H(Ap(n))*Tp(Ap(n))$$

En el caso de algoritmos secuenciales, el trabajo llamado W es igual a Ts(As(n)), ya que la superficie (número de procesadores usados) en ese caso siempre tendrá el valor de 1. Para terminar esta parte, definiremos algunos conceptos teóricos en torno a la aceleración y eficacia:

- Un algoritmo paralelo es eficaz si su tiempo de ejecución es polinomial y si su trabajo es igual al mejor tiempo del mejor algoritmo secuencial conocido, multiplicado por un factor polinomial. Así, un algoritmo eficaz permite obtener un tiempo de resolución imposible de obtener por un algoritmo secuencial, quizás con una mayor carga de trabajo, pero la misma razonable.
- Un algoritmo paralelo es bueno si es eficaz, y su trabajo es al menos del mismo orden que el trabajo del mejor algoritmo secuencial conocido.

La jerarquía entre los modelos PRAM viene dada por las relaciones entre los diferentes modelos de ella. Si definimos que $x<<y$ indica el hecho de que PRAM*y* puede ejecutar todos los algoritmos escritos para la PRAM*x*. En general, se tiene las inigualdades siguientes:

EREW<<CREW<<CRCW y EREW<<ERCW<<CRCW

Podemos verificar esas inigualdades con el siguiente ejemplo:

Ejemplo 5.4: Verificar si un elemento *e* dado está en el conjunto $E=\{e_1, \ldots, e_n\}$.

En una PRAM-CREW un procesador inicializa una variable lógica en falso. Después, cada procesador Pi, para i=1, n, lee el elemento *e* y lo compara con e_i. Si son iguales, Pi posiciona la variable lógica en verdad, de lo contrario no hace nada. Como los e_i son diferentes dos a dos, sólo un proceso escribe un valor en la localidad de memoria que contiene la variable lógica (por eso es una CREW). En una EREW los procesadores no pueden accesar simultáneamente el elemento *e*, por lo que se requieren copias de *e*, lo cual requiere un tiempo extra de ejecución (EREW<<CREW).

Ahora la pregunta es saber cómo realizar operaciones de escritura y lectura concurrentes en máquinas a lectura y escritura exclusiva (simularlas). Por ejemplo, para crear *n* copias se requiere O(log(n)) como tiempo de ejecución utilizando O(n/log(n)) procesadores. Así, todas las etapas CREW usando *p* procesadores pueden ser simulados en O (log(p)) pasos sobre una EREW. Esta simulación no requiere más procesadores que el programa inicial, ya que se necesitan O(p/log(p)) procesadores, menos que los *p* disponibles. Algo análogo se puede hacer para simular la escritura concurrente (PRAM ERCW) y la lectura y escritura concurrente (PRAM CRCW). El factor de multiplicación del tiempo de simulación es del orden del logaritmo del número de procesadores, pero independiente del tiempo de ejecución del algoritmo a simular.

A pesar de parecer que la definición del modelo PRAM a escritura y lectura concurrente fuese de poco interés, por las suposiciones que se deben hacer, la posibilidad de su simulación en PRAM-EREW, con el bajo costo que se añade, a nivel de tiempo de ejecución, hace interesante ese modelo. Por supuesto, conseguir un algoritmo óptimo EREW nos da una manera directa de paralelizar un código, sin necesidad de simular los accesos concurrentes, y así, los tiempos de ejecución extras.

Se pueden definir diferentes técnicas para, a partir de un algoritmo secuencial, desarrollar programas paralelos PRAM:

a) Estudiar el Grafo de Precedencia:

El análisis del grafo de precedencia permite explorar el paralelismo que existe en el algoritmo inicial. La profundidad de ese árbol es el límite superior del tiempo de ejecución del programa paralelo que se requiere para ese problema, y el grado máximo del árbol nos determina el número de procesadores requeridos como mínimos.

Ejemplo 5.5: Resolver el sistema lineal

Ax=b.

donde A es una matriz n*n, b es un vector n y se quiere calcular el vector x de *n* elementos. El algoritmo secuencial consiste en el proceso iterativo siguiente:

$\forall$ i=1, …, n $\quad x_i = (b_i - \sum_{k=1}^{i-1} a_{ik}x_k)/a_{ii}$

El orden es $O(n^2)$, suponiendo que todas las operaciones pueden ser ejecutadas en tiempo constante. Su paralelización es difícil por las dependencias fuertes entre cualquier x_i y x_j, $\forall$ i,j=1, …, n. Un posible código secuencial es:

Para i=1, n
x(i)=0
Para k=1,i-1
x(i)=x(i)+a(i,k)x(k)
x(i)=(b(i)-x(i))/a(i, i)

Ese código puede ser parcialmente paralelizado. Las iteraciones *i* se van a realizar secuencialmente para respetar las dependencias entre diferentes x_i, pero el lazo interno puede ser paralizado realizándose al final una operación de reducción. En una máquina EREW esto nos da un requerimiento para la solución del problema del tipo Op(n log(n), n/log(n)) con un número máximo de procesadores necesarios iguales a O(n/log(n)).

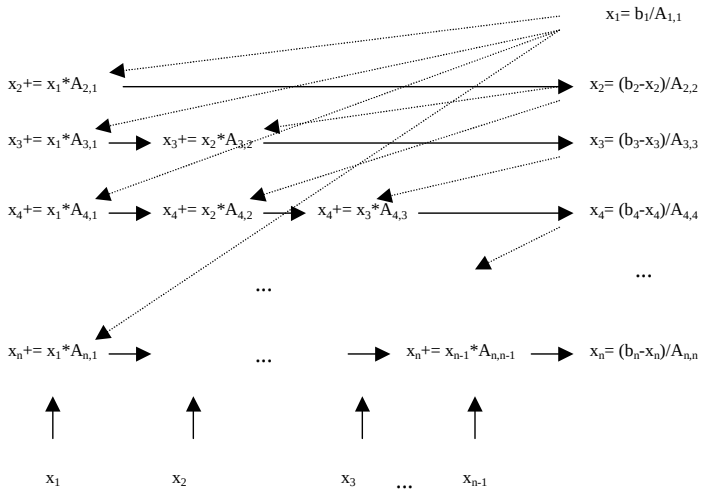


Figura 5.7. Grafo de dependencia del ejemplo 5.5.

Esta paralelización puede ser mejorada en tiempo, al remarcarse que para cualquier x_i es posible calcular en paralelo a_{ik} x_k,$\forall$i>k (esto se deduce del grafo de precedencia, al observarse que se pueden calcular en paralelo en los diferentes procesadores encargados de calcular los x_i, ver figura 5.7). Así, todos los x_k deben ser leídos en paralelo por los procesadores encargados de calcular los x_i, requiriéndose por eso una PRAM-CREW. Reescribiendo la secuencia de cálculos $x_i = x_i + a_{i1}x_1$; $x_i =(b_i - x_i)/a_{ii}$, $\forall$i>1, en la secuencia equivalente $b_i = b_i – a_{i1}x_1$; …; $b_i = b_i - a_{ii-2}x_{i-2}$; $x_i =(b_i - x_{i-1})a_{ii-1}$, se obtienen las siguientes etapas:

Etapa 1: $x_1 = b_1/a_{11}$
Etapa 2: $x_2 = (b_2 – a_{21}x_1)/a_{22}$ // $b_3 = b_3 – a_{31}x_1$ // $b_4 = b_4 – a_{41}x_1$ // …
Etapa 3: $x_3 =(b_3 – a_{32}x_2)/a_{33}$ // $b_4 = b_4 – a_{42}x_2$ // $b_5 = b_5 – a_{52}x_2$ // …
.

.
Etapa n-1: $x_{n-1} = (b_{n-1} - a_{n-1,n-2}x_{n-2})/a_{n-1,n-1}$// $b_n = b_n - a_{n,n-2}x_{n-2}$
Etapa n: $x_n = (b_n - a_{n,n-1}x_{n-1})/a_{n,n}$

En cada nueva etapa, un nuevo componente x_i es calculado, donde el vector x puede ser calculado en un tiempo O(n) en una PRAM-CREW con *n* procesadores. En este caso, los requerimientos del algoritmo PRAM-CREW serán de Op(n, n), lo que lo hace más eficaz y rápido.

b) Dividir para Paralelizar:

Es la versión paralela de la técnica secuencial divide y vencerás, la cual permite definir el grado de paralelismo a usar. La idea es tratar de resolver un problema P(n) a través de instancias más simples del mismo, tal que cada instancia pueda ser tratada independientemente. El procedimiento es el siguiente:

- Reducir el problema en instancias independientes más pequeñas.
- Resolver paralelamente y recursivamente las subinstancias.
- Construir la solución del problema inicial a partir de las soluciones de cada subinstancia.

Ejemplo 5.6: Calcular $P_k = a_1 * \ldots * a_k$, $\forall k=1, n$, donde los a_k corresponden a *n* entradas y * es una operación asociativa (reducción).

La solución secuencial conduce a un algoritmo O(n). Pero en este caso se puede aplicar la técnica divide y vencerás:

- Separar la cadena original en dos subcadenas de tamaño n/2: s_1 (a_1 , ..., $a_{n/2}$) y s_2 ($a_{n/2+1}$, …, a_n).
- Resolver cada subcadena en paralelo. En este caso, en cada una se aplica la recursividad (es decir, cada subcadena se vuelve a dividir). Al final se obtendrán los resultados parciales (U_1, …, $U_{n/2}$) para s_1 y ($U_{n/2+1}$, …, U_n) para s_2.
- Construir la solución del problema inicial, tal que:

$$P_i = \begin{cases} U_i & \text{si } i \le n/2 \\ U_{n/2} * U_i & \text{si } i > n/2 \end{cases}$$

Este algoritmo puede ser resuelto por una PRAM-CREW (lectura concurrente en la fase de construcción) que requiere Op(log(n), n). Sin embargo, se pueden realizar las siguientes transformaciones para ser ejecutadas en una PRAM-EREW:

- Construir una tabla b_1, …, $b_{n/2}$, donde todos los elementos tienen el valor de $U_{n/2}$.
- Construir la solución del problema inicial, tal que:

$$P_i = \begin{cases} U_i & \text{si } i \leq n/2 \\ b_{i-n/2} * U_i & \text{si } i > n/2 \end{cases}$$

Estos cálculos se pueden hacer en un tiempo del orden O(n), sin que exista ahora conflicto de acceso a memoria, quedando por definir la complejidad de la construcción de la tabla b_i. Esta es del orden Op(log(n), n) si se supone una etapa inicial donde se recopia $P_{n/2}$ en b_1. Después, se usa un esquema repetitivo. En la primera etapa un procesador lee b_1 y escribe una copia en b_2. En la segunda, dos procesadores leen b_1 y b_2, respectivamente, y escriben en b_3 y b_4, y así sucesivamente, hasta llenar los $b_{n/2}$ en O(log(n)) etapas con O(n) procesadores. El algoritmo PRAM-EREW tiene requerimientos del tipo Op($\log^2$(n), n). En resumen, el trabajo en los procesadores es O(n) en cualquiera de los dos enfoques, mientras que el tiempo en la PRAM-EREW es O($\log^2$(n)) y en la PRAM-CREW es O(log(n)).

c) Romper la Precedencia por Redundancia:

Los métodos anteriores se inspiran en técnicas empleadas sobre los algoritmos secuenciales. Otra forma de buscar paralelismo es haciendo varias veces el mismo cálculo. Esta técnica parte de la idea de hacer cálculos redundantes de manera de disminuir las dependencias de datos. La introducción de redundancia es una manera de disminuir el tiempo de procesamiento paralelo, a pesar de no conducir a un algoritmo óptimo, ya que la introducción de más cálculo aumenta el trabajo en el sistema.

Ejemplo 5.7: Calcular el entero R=A+B, tal que A y B son dos enteros representados por una secuencia de *n* bits: A= a_{n-1}, …, a_o; B= b_{n-1}, …, b_o. Además, R será representado por la secuencia de n+1 bits: r_n, …, r_o.

El algoritmo secuencial consiste en sumar dos a dos las cifras de A y B, desde la menos significativa, haciendo propagar los resultados parciales, lo cual puede ser expresado así:

$c_{-1} = 0$
$r_k = (a_k + b_k + c_{k-1}) \bmod 2$
$c_k = (a_k + b_k + c_{k-1}) \text{ div } 2$

donde *div* y *mod* son las operaciones de división entera y del módulo. Además, c_k es el acarreo. Con esta relación de recurrencia es imposible extraer paralelismo, en particular por la propagación del acarreo (r_k no puede ser calculado hasta no tener el acarreo c_{k-1}, que depende a su vez de todos los cálculos anteriores). Para evitar esto e introducir paralelismo, se puede partir A y B en secuencias de pesos fuertes (Af y Bf) y pesos débiles (Ad y Bd). Igual se hace con R, separándolo en Rf y Rd. Dos cálculos son posibles para Rf: suponer que el acarreo al calcular Ad+Bd es nulo o igual a 1. Así, el algoritmo es:

- Reducción: separar en paralelo las secuencias Af y Ad, Bf y Bd.

- Resolución: calcular recursivamenete Ad+Bd, Af+Bf+1 y Af+Bf.
- Fusionar: si la suma de Ad y Bd genera un acarreo, fusionar Ad+Bd con Af+Bf+1, de lo contrario, fusionar Ad+Bd con Af+Bf.

Se tiene así, un cálculo redundante, ya que las cifras más significativas son sumadas dos veces, pero los tres cálculos son independientes. En este algoritmo se necesita una PRAM-CREW, ya que hay accesos concurrentes a los bits más fuertes. Como se pueden crear en tiempos constantes copias de las secuencias de pesos fuertes con O(n) procesadores, el tiempo de ejecución en una PRAM-EREW es O(log(n)).

5.5 Puntos de Referencia o Programas de Comparación del Rendimiento (Benchmarks)

Un punto de referencia, es un programa cuyo propósito es medir una características de rendimiento de un computador, por ejemplo, la rapidez con que se hacen los cálculos de puntos flotantes, o la rapidez con que se hacen las operaciones de entradas/salidas [1, 11, 17, 23, 30, 33]. Particularmente, un punto de referencia emula las características de los movimientos de datos y de procesamiento de clases de aplicaciones. En general, los puntos de referencia son usados para medir y predecir el rendimiento de sistemas computacionales, y así determinar sus debilidades y fortalezas. Un punto de referencia puede ser un programa que hace trabajos reales o un programa sintético específicamente diseñado para hacer pruebas de rendimiento. *Linpack y Livermore loops* son ejemplos de estos tipos de aplicaciones, las cuales fueron extraídas de aplicaciones grandes. Estos programas son escritos en lenguajes de alto nivel como Fortran. Los puntos de referencia son más interesantes cuanto más portables y fáciles sean para ejecutarse en una gran variedad de computadores, aun más que la propia exactitud de los cálculos. Cuando una colección de estos programas es agrupada en un conjunto, a este conjunto se le denomina conjuntos de puntos de referencia (Benchmarks). Los conjuntos de puntos de referencia tienen reglas específicas que gobiernan las condiciones y procedimientos de las pruebas. Dichas reglas indican los datos de entrada, los resultados de salidas y las medidas de rendimiento.

Hay un gran número de puntos de referencia, desde programas en Fortran77 para cálculo científico, hasta programas que se preocupan por explotar específicas operaciones (por ejemplo, operaciones de entrada/salida). Los puntos de referencia pueden ser clasificados de acuerdo a clases de aplicaciones, tales como de computación científica, comerciales, de servicios de redes, multimedias, procesamiento de señales, etc. También pueden ser divididos en macro y micro-puntos de referencia. Los macro-puntos de referencia miden el rendimiento de un sistema como un todo. Ellos comparan diferentes sistemas con respecto a una clase de aplicaciones. Sin embargo, no revelan por que el rendimiento de un sistema es bueno o malo. Un micro-punto de referencia mide un aspecto específico de un sistema computacional, tal como la velocidad de CPU, la velocidad de la memoria, la velocidad de las unidades de Entrada/Salida, el rendimiento del sistema operativo, etc. Muchos conjuntos de puntos de referencias han sido

propuestos, algunos de ellos son mostrados en la tabla siguiente [7, 20, 24, 25, 28, 29, 34, 38, 39].

Tipo	Nombre	Medida
Micro-programas de comparación	LINPACK	Computación numérica
	LIMBENCH	Llamadas a sistemas y operaciones de movimiento de datos en UNIX
	STREAM	Ancho de Banda de la memoria
Macro-programas de comparación	NAS	Computación Paralela
	PARKBENCH	Computación Paralela
	SPEC	Una mezcla de clases de programas de comparación
	Splash	Computación Paralela
	STAP	Procesamiento de Señales

Tabla 5.2. Algunos ejemplos de conjuntos de programas de comparación

Se explica a continuación algunos de ellos. Linpack es uno de los de puntos de referencia más usados. Es una colección de rutinas de álgebra lineal usadas para resolver sistemas de ecuaciones lineales. El punto de referencia LINPACK fue creado y es mantenido por J. Dongarra, primero en el laboratorio nacional Argonne y ahora en la Universidad de Tennessee. Es un buen indicador de las capacidades de procesamiento numérico de un sistema. Otro ejemplo de punto de referencia es el mantenido por McVoy para SGI, llamada LIMBENCH. Este es un punto de referencia portable usado para medir el sobrecosto de los sistemas operativos y las capacidades de transferencias de datos entre los procesadores, la memoria, la cache, las redes, y el disco. Permite identificar cuellos de botellas a nivel de rendimiento.

SPEC (System Performance Evaluation Cooperative Effort) es un conjunto de puntos de referencias para estaciones de trabajo, creado por un grupo de consorcios de vendedores de computadoras (Compaq, HP, DEC, Sun, Unisys, IBM, etc.). SPEC consiste en aplicaciones reales que reflejan la carga de trabajo de aplicaciones comunes. Por cada conjunto de puntos de referencia, SPEC define las medidas de rendimientos a usar. SPEC comenzó con puntos de referencia para medir los rendimientos del CPU, y fue extendido para el modelo cliente/servidor, subsistema entrada/salida, etc. Algunos de los conjuntos de puntos de referencia que componen SPEC son los siguientes:

- *SPEC'95:* mide el rendimiento de CPU, el sistema de memoria y la generación de código de los compiladores.
- *SPEChpc92:* mide el rendimiento del sistema para cálculos intensivos corriendo aplicaciones industriales. Está compuesto por dos puntos de referencia, uno de procesamiento sísmico (SPECseis96) y otro de cálculo químico (SPECchem96).
- *SPECweb96:* mide el rendimiento de servidores web basado en la carga de trabajo generada por servidores www del mundo real. Mide el tiempo de respuesta versus la capacidad de ejecución de los servidores de archivos NFS para varios niveles de carga.

- *SPEC77* es un programa que simula flujos atmosféricos para predecir patrones de tiempo.
- *SDM* (System Development Multitasking): mide como el sistema maneja un ambiente con un gran número de usuarios utilizando comandos basados en Unix (por ejemplo, make, spell, etc.)
- *GPC* (Graphics Performance Characterization): mide el rendimiento gráfico. Para eso, se basa en los siguientes puntos de referencia: PLB (Picture Level Benchmark) que mide las capacidades gráficas, Xmark93 que mide el rendimiento de Xwindow, y Viewperf que mide el rendimiento de OpenGL, entre otros.

El comité PARKBENCH (PARallel Kernels and BENCHmarks) fue fundado en el congreso Supercomputing'92 por un grupo de personas interesadas en la evaluación de rendimientos de computadores paralelos. Una contribución del grupo fue establecer un conjunto consistente de medidas de rendimientos y de notaciones. PARKBENCH mide las comunicaciones punto-a-punto y las barreras de sincronización para sistemas a memoria distribuida, usando aplicaciones codificadas en Fortran77, con PVM o MPI para el pase de mensajes, además de versiones en HPF (High Performance Fortran) para arquitecturas a memoria compartida. El grupo definió 4 clases de programas de comparación:

- Puntos de referencia de bajo nivel: estos micro puntos de referencia miden parámetros básicos tales como velocidad de las operaciones aritméticas, velocidad de la memoria principal y cache, sobrecosto de las sincronizaciones, etc.
- Puntos de referencia de kernel: ellos utilizan rutinas frecuentemente usadas en computación científica, tales como operaciones sobre matrices, resolución de ecuaciones diferenciales, etc.
- Puntos de referencia para aplicaciones compactas: incluye una aplicación paralela para modelar la transformación espectral de la superficie del agua y simulaciones que emulan los movimientos característicos de computación y de datos de grandes aplicaciones de cálculo de fluidos dinámicos. Estas aplicaciones fueron desarrolladas en la NASA.
- Puntos de referencia para compiladores HPF. Son simples aplicaciones sintéticas usadas para medir los compiladores HPF.

Otros programas de comparación son los siguientes. *Livermore loops* es una colección de segmentos de códigos de Fortran extraídos de varias aplicaciones científicas. *Nas* es una colección de siete rutinas de Fortran típicas de cálculos de dinámicas de fluidos. Entre esas rutinas se tiene MMX (Multiplicación de matrices) y GMTRY (eliminación gausiana). *SLALOM* (Scalable Language independent, Ames Laboratory, One-minute Measurement) es un punto de referencia basado en cálculo científico general, el cual fue propuesto por Gustafson para medir la eficiencia del cálculo con paralelismo de datos. Este punto de referencia permite medir el rendimiento de un computador paralelo, en función del tamaño del problema. Este punto de referencia siempre se corre un minuto, lo que se pueda computar en ese tiempo es lo que determina la aceleración del sistema.

5.6 Otros Aspectos de Rendimiento

A continuación se presentan, otras medidas de rendimiento para arquitecturas multicomputadores, que no toman en cuenta ciertos aspectos, por ejemplo, el comportamiento de la memoria cache. Una de las medidas es el *tiempo de ejecución*, el cual está en función del tamaño del problema, del número de procesadores P, del número de tareas U, y de otras características de los algoritmos y del hardware. En general, el tiempo de ejecución de un programa paralelo es el tiempo que se toma cuando el primer procesador comienza la ejecución del programa hasta cuando el último procesador completa su ejecución. Así, el tiempo de ejecución T puede ser definido de dos formas: como la suma de los cálculos, comunicaciones y tiempos de ocio en un procesador arbitrario j (si se ejecuta en un solo procesador)

$$T= Tj^{comp}+Tj^{comun} + Tj^{ocio} \qquad (5.14)$$

o como el máximo de estos tiempos sobre todos los procesadores

$$T= maximo_j\ (Tj^{comp}+ Tj^{comun} + Tj^{ocio}) \qquad \forall j=1, P \qquad (5.15)$$

Como se observa en la última fórmula, al repartir el cálculo en diferentes procesadores, el tiempo de computación dependerá del número de tareas o procesadores. Además, si el sistema de computación es heterogéneo, los tiempos de computación varían en los diferentes procesadores. Los tiempos de cálculo también dependen de las características de los sistemas de memorias. El tiempo de comunicación es el tiempo que las tareas gastan enviando y recibiendo mensajes. Existen dos tiempos de comunicación: interprocesadores (comunicación entre tareas en diferentes procesadores) e intraprocesadores (tareas que se comunican que están en el mismo procesador). Muchas veces, para ciertos sistemas estos tiempos pueden ser comparables (cuando los accesos a las regiones críticas o los cambios de contextos son comparables a los costos de comunicación interprocesadores). En una idealizada arquitectura, los costos de enviar mensajes pueden ser representados por: el tiempo para iniciar la comunicación (T_{init}) y el tiempo de transferencia por palabra (T_m). Así, para enviar L palabras:

$$T_{mensaje}=T_{init} + T_mL \qquad (5.16)$$

La anterior expresión es muy simple y requiere adicionales costos que consideren, por ejemplo, el manejo del buzón de mensajes.

Los tiempos anteriores son explícitamente indicados en los algoritmos paralelos, mientras que los tiempos de ocio son más difíciles de determinar. Un procesador puede tener tiempos de ocio debido a largos cálculos en otros procesadores o a datos que él requiere para sus cálculos. El primer caso se puede evitar usando técnicas de equilibro de la carga. En el segundo caso, el ocio es debido a cálculos que requieren de datos remotos. Para evitar eso se hacen programas que solapen los cálculos y las comunicaciones. Tal solapamiento puede ser creado asignando múltiples tareas en cada procesador, así,

cuando una tarea espera por datos pase a ejecutar otra tarea (esto es válido si el tiempo de planificación es bajo) (ver figura 5.8).

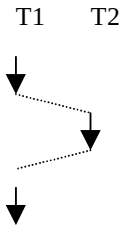


Figura 5.8. Solapar la comunicación con la computación.

En general, los tiempos de ejecución dependen de muchos factores:

- *Los algoritmos, las estructuras de datos y los datos de entrada*: Programas, tales como los de ordenamiento y búsqueda, pueden tener diferentes tiempos para diferentes datos, porque el flujo de control de un programa puede cambiar para las diferentes entradas.
- *Las plataformas*: tanto el hardware como el sistema operativo afectan el rendimiento. Otros factores que afectan también, incluyen la jerarquía de memoria (cache, memoria principal y discos) y si las aplicaciones hacen un uso exclusivo del computador o comparten con otros recursos.
- *Los lenguajes*: los compiladores y las opciones que se usen, juegan un rol importante.

¿ Qué ocurre cuando un computador paralelo es compartido por varios usuarios? En estos casos se puede usar la *capacidad de ejecución* como medida de rendimiento, que significa el número de trabajos que han sido utilizados por unidad de tiempo. En general, la capacidad de ejecución de un sistema puede ser aumentada por los siguientes dos métodos: i) por encauzamiento, tal que los trabajos solapen entre sí sus tiempos de ejecución. La capacidad de ejecución es el tiempo de ejecución de la más larga fase del encauzamiento, ii) por la asignación de diferentes trabajos en diferentes nodos del sistema.

Otros aspectos a considerar a nivel de los rendimientos computacionales son:

- *La jerarquía de memoria*: Los rendimientos de un moderno computador dependen de la rapidez con que los datos se pueden mover entre los procesadores y las memorias. En un subsistema de memoria, para cada nivel se deben considerar 3 parámetros:
  - ¿Cuantos bytes de datos se pueden almacenar en una unidad?
  - ¿Cuanto tiempo es necesario para traer una palabra desde alguna unidad?
  - ¿Cuantos bytes pueden ser transferidos desde una unidad en un segundo?
- *Los costos de paralelismo y de interacción*: El tiempo para ejecutar una instrucción implícitamente coloca una barrera después de cada instrucción. Si no se asumiera solapamiento, el tiempo de ejecución de los programas paralelos sería:

$$T= T^{comp} + T^{par} + T^{interac} \qquad (5.17)$$

donde T^{comp}, T^{par} y $T^{interac}$ son los tiempos para cálculo, del paralelismo y por las operaciones de interacción, respectivamente. Hay tres tipos de operaciones de paralelismo que son fuentes de sobrecosto de paralelismo:

- Manejo del paralelismo: creación, cambio de contexto, etc.
- Operaciones de agrupamiento.
- Operaciones de consulta a procesos: identificación de procesos, etc.

Para las operaciones de interacción hay tres tipos de fuentes de sobrecosto

- Sincronización: tales como barreras, regiones críticas, etc.
- Agregación: tales como reducción
- Comunicación: tales como comunicaciones punto-a-punto y colectivas, así como variables compartidas.

Con frecuencia, los sobrecostos son causados por el sistema operativo o software del sistema. Consecuentemente, los sobrecostos son diferentes de un sistema a otro, aun cuando se use la misma arquitectura. Según esos sobrecostos, el programador puede definir su estrategia para desarrollar programas paralelos. Por ejemplo, cuando el sobrecosto de paralelismo es pequeño, el programador puede realizar programas paralelos dinámicos, en el cual sus procesos son frecuentemente creados, destruidos, etc. Cuando el sobrecosto es grande, programas paralelos estáticos son usados, donde los procesos son creados solamente al inicio y destruidos al final. Si las barreras son costosas, es mejor usar algoritmos asíncronos, de lo contrario, si son eficientemente soportados, uno puede usar algoritmos síncronos, y así sucesivamente.

Se culmina este capítulo con la presentación de los pasos a seguir en el proceso de análisis de rendimientos, los cuales son: colección de los datos, transformación de los datos y visualización de los datos.

a) *La colección de los datos* es el proceso por el cual los datos sobre los rendimientos de los programas son obtenidos desde ellos, cuando se ejecutan. Los datos son coleccionados en un archivo. Tres técnicas de colección de datos se conocen:

- *Perfiles:* como ya se ha dicho, un pobre rendimiento puede ocurrir por el exceso de cálculos replicados, a tiempos de ocio, al costo de transferencia de mensajes, etc. Un importante paso para identificar esos factores es determinar cuál de ellos toma más tiempo. Eso se puede hacer, calculando un esperado perfil (*profile* en ingles) de ejecución para el algoritmo, indicando las contribuciones de cada uno de los factores sobre el tiempo de ejecución, es decir, registrando la cantidad de tiempo que duran las diferentes partes del programa. Esa información puede usarse para guiar el rediseño de un algoritmo. Con frecuencia, puede ser un motivo para reconsiderar una decisión hecha tempranamente en el proceso de diseño. Por ejemplo, si los cálculos distribuidos reducen el rendimiento, entonces se puede reconsiderar un algoritmo alternativo que aumente los cálculos distribuidos a expensas de aumentar el costo de

comunicación. Alternativamente, si el costo de transferencia de datos es alto, se puede pensar en mensajes más pequeños. En general, los perfiles tienen dos ventajas: pueden ser obtenidos automáticamente a bajo costo y proveen una vista del comportamiento del programa que permite al programador identificar partes problemáticas del programa. Una de sus desventajas es que no incorpora aspectos temporales de la ejecución de un programa (por ejemplo, si todas las tareas de un programa que están en diferentes sitios envían en el mismo momento o en diferentes momentos, sólo indica que envían una cantidad de datos). Los perfiles están disponibles en muchos computadores paralelos.

- *Contadores:* registran frecuencias de eventos o tiempos acumulativos. Un contador es una localidad de almacenamiento que puede ser incrementada cada vez que un evento especifico ocurre. Los contadores pueden ser usados para registrar el número de llamadas a procedimientos, el número total de mensajes, el número de mensajes enviados entre cada par de procesadores, etc. Una variante es usar el contador para medir el intervalo de tiempo que determina la longitud del tiempo utilizado en ejecutar una específica parte del código.
- *Trazas de eventos*: registran cada ocurrencia de varios eventos. Una traza de ejecución es el enfoque más detallado de colección de datos. Los sistemas basados en estas técnicas, típicamente generan archivos que contienen estampillas, con las ocurrencias de eventos significativos, durante la ejecución de un programa (por ejemplo, un pase de mensaje, una llamada a un procedimiento, etc.). Ellos se usan para determinar las relaciones entre comunicaciones, localizar fuentes de tiempos de ocio, etc. También, ellos se pueden procesar para obtener contadores, perfiles, etc. La gran cantidad de información que generan es su mayor desventaja.

b) *La transformación de los datos* se aplica con el objetivo de reducir el volumen de los mismos. Las transformaciones se usan para determinar el valor promedio u otras estadísticas, cuando se usan los contadores o perfiles. Por ejemplo, un perfil que registra el tiempo utilizado, en cada subrutina en cada procesador, puede ser transformado para determinar el tiempo promedio gastado en cada subrutina en cada procesador. Una observación interesante en este punto es que los datos de rendimiento son inherentemente multidimensionales, y consisten en tiempos de ejecución, costos de comunicación, etc., para múltiples componentes del programa, sobre diferentes procesadores, y para diferentes tamaños de un problema dado. A pesar de que técnicas de reducción de datos pueden ser usadas en algunas situaciones para comprimir los datos de rendimientos, es frecuentemente necesario explorar el espacio multidimensional de los datos.

c) El proceso de *exploración del espacio multidimensional* de los datos puede beneficiarse enormemente al usar técnicas de visualización de datos. En general, al seleccionar una herramienta de evaluación de rendimiento, se debe considerar:

- La exactitud.
- La simplicidad: las mejores herramientas son las que recolectan los datos automáticamente.

- La flexibilidad: la manera de extenderse para coleccionar datos adicionales o proveer diferentes vistas de un mismo dato.
- La mínima intrusión de la herramienta en el sistema: la colección de datos produce un inevitable sobrecosto que debe ser minimizado.

Ejemplos de herramientas de rendimiento son los siguientes: *Paragraph* (un paquete de visualización y análisis de trazas desarrollado por el laboratorio nacional Oak Ridge para programas de pase de mensajes), *Upshot* (un paquete de visualización y análisis de trazas desarrollado por el laboratorio nacional Argonne para programas de pase de mensajes), *Pablo* (un sofisticado sistema con una variedad de mecanismos para coleccionar, transformar y visualizar datos, el cual fue desarrollado por la Universidad de Illinois), etc.

Capítulo 6.
Ejemplos de Programas Paralelos

En este capítulo se presentan varios ejemplos de programas paralelos [1, 2, 6, 8, 11, 20, 27, 28, 32, 33, 37]. Los mismos son presentados en pseudo-código, dejando al lector su posible implementación, en cualquiera de los lenguajes o librerías de programación paralela (por ejemplo, HPF, MPI, etc.). Se presta una especial atención en estudiar las fuentes de paralelismo, en cada uno de los problemas que se plantean.

6.1 Producto de Dos Vectores

Este problema es un buen ejemplo para demostrar el paralelismo de datos al no existir dependencia de datos. El producto entre dos vectores A y B de tamaño *n* es dado por:

$D = \sum_{i=0}^{n} A(i)*B(i)$

Su algoritmo secuencial es:

```
Res=0
Para i=1,n
        Res = Res + A(i)*B(i)
```

La parte a paralelizar es el lazo principal al no existir dependencias entre la instancia *i* e *i+1* del lazo. Así, los vectores A y B pueden ser divididos en fragmentos iguales, donde cada fragmento es enviado a un procesador diferente. Entonces, los procesadores calculan el producto parcial para sus respectivas porciones de vector. Finalmente, el procesador maestro colecciona los resultados parciales y las sumas para obtener el vector con el producto total. La figura 6.1 muestra una posible partición de los vectores entre los diferentes procesadores, asumiendo un tamaño de vectores de 10000 y 4 procesadores.

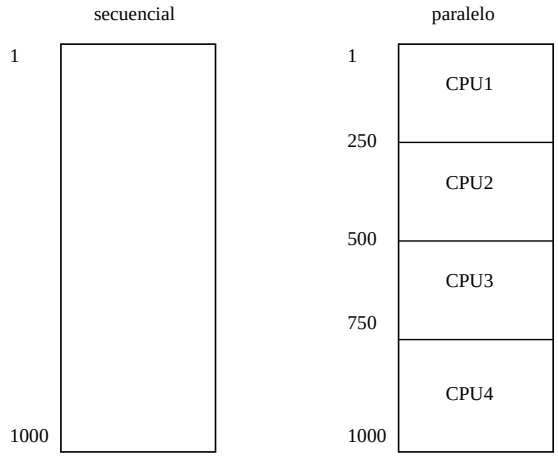


Figura 6.1. Partición de los vectores a multiplicar

Los pasos del programa paralelo son:

1. Dividir los vectores A y B en partes iguales entre todos los procesadores. Así, el número de elementos por partición será igual al número de filas, entre el número de procesadores.
2. Cada procesador calcula el producto parcial usando su porción de los vectores A y B.
3. El procesador maestro (uno de los 4 procesadores) suma todos los productos parciales para obtener el producto total de los vectores A y B enteros.

6.2 Paralelización de un Lazo *Livermore*

Este programa es un ejemplo de paralelismo de datos donde los dominios se sobreponen. En particular, este programa ha sido usado con otros *conjuntos de referencias* para comparar el rendimiento de sistemas computacionales. Este es un programa que permite probar las capacidades de encauzamiento de datos en los sistemas programados (particularmente, en los procesadores vectoriales). El programa secuencial consiste de un lazo central, el cual se ejecuta un gran número de veces. El tiempo tomado para su ejecución es la medida de rendimiento. Básicamente, ese lazo es el siguiente:

Para k=1, n-6
x[k]=f(u[k],z[k],y[k],u[k+1],u[k+2],u[k+3],u[k+4],u[k+5],u[k+6])

Como se observa, un valor particular de x(k) depende de *u, z* e *y*, y 6 elementos extras de *u* (u[k+1] a u[k+6]). Así, si dividimos el arreglo *u*, seis elementos adicionales de *u* deben ser enviados a cada procesador, además de los respectivos intervalos definidos por la partición del arreglo. De esta manera, los dominios de los procesadores son solapados

con sus vecinos. Por ejemplo, si tenemos 4 procesadores y el tamaño del arreglo es 40000, entonces:

Procesador 1 recibirá el arreglo *u* desde el índice 1 al 10006
Procesador 2 recibirá el arreglo *u* desde el índice 10001 al 20006
Procesador 3 recibirá el arreglo *u* desde el índice 20001 al 30006
Procesador 4 recibirá el arreglo *u* desde el índice 30001 al 40000

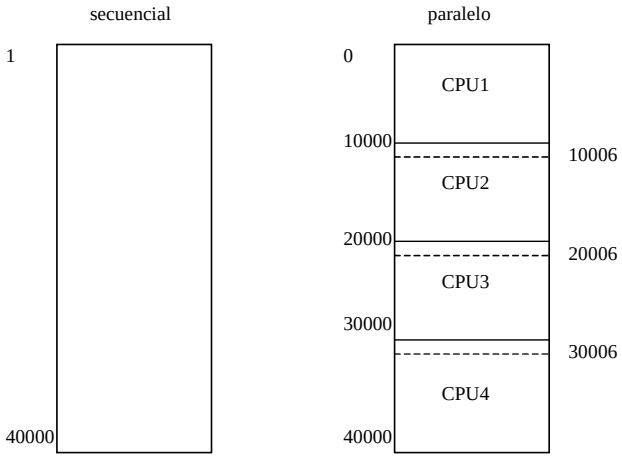


Figura 6.2. Paralelización de un lazo *Livermore*

Es bueno resaltar que aunque la distribución del arreglo *u* es solapada, el código del programa no es modificado. Así, el corazón del programa permanece sin modificarse en la versión paralela, excepto por los limites superiores e inferiores que tendrá cada lazo *k*. El pseudo-código paralelo es:

1 Dividir los vectores *x*, *y* y *z* en partes iguales entre todos los procesadores. Así, el número de elementos por partición será igual al número de filas entre el número de procesadores
2 Dividir el vector *u* en partes iguales y agregarles los elementos que se solapan. Así, el número de elementos por partición será igual al número de filas entre el número de procesadores, más los números de elementos solapados.
3 Cada procesador calcula la función *f* usando su porción de datos.
4 Al final, se recopilan los cálculos de los diferentes *f* de cada uno de los procesadores, en un solo procesador.

6.3 Paralelización de la Multiplicación de Matrices

La multiplicación de dos matrices es un procedimiento que se usa en muchas aplicaciones científicas y de ingeniería. El producto es definido como C= A*B, donde A, B y C son matrices de orden *nxn*. La expresión clásica es:

$C_{ij}= \sum_{k=1}^{n} A_{ik}*B_{kj}$ donde $1\leq i\leq n$ y $1\leq j\leq n$

A continuación presentamos el pseudo-código del programa secuencial:

Para j=1, n
  Para i=1, n
    $C_{ij} = 0$
    Para k=1, n
      $C_{ij} = C_{ij} + A_{ik} * B_{kj}$

Para dividir el cálculo de la matriz C entre los diferentes procesadores, una posibilidad consiste en descomponer el cálculo a través de sus filas o columnas. Como Fortran guarda las matrices de acuerdo a un orden basado en columnas, es mejor partir dichas matrices por columnas, es decir, cada procesador tomara un bloque de columnas para calcular. En el caso del lenguaje C, la situación es diferente, ya que la manera de guardar la información es por filas, por lo que es mejor partirlas por bloques de filas. Volviendo al caso de Fortran, se toma en cuenta el lazo principal del pseudo-código, para calcular C_{ij},. Toda la fila *i* de la matriz A y la columna *j* de la matriz B son necesarias. Así, la matriz A no debe ser dividida y la matriz B debe ser dividida en bloques de columnas entre los procesadores. La siguiente figura muestra el método para dividir las matrices entre los procesadores:

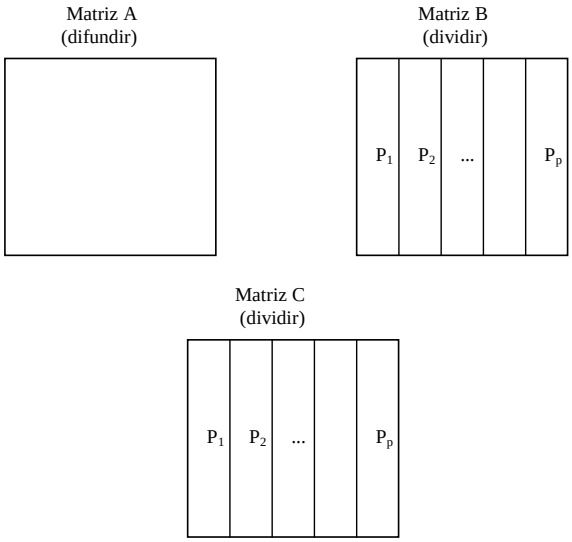


Figura 6.3. Paralelización de la Multiplicación de Matrices

En dicha figura, P1, P2, etc. son los procesadores utilizados. Los pasos a seguir para elaborar un código paralelo, para el caso anterior, son:

1. Enviar a todos los procesadores la matriz A.
2. Enviar los requeridos bloques de columnas de la matriz B a los respectivos procesadores (dividir a B).
3. Cada procesador calcula las correspondientes columnas de la matriz C.
4. Coleccionar los bloques de las columnas de C calculadas en los diferentes procesadores, en un solo procesador, y unirlos en una única matriz C.

Existen otras implementaciones paralelas de la multiplicación de matrices. Por ejemplo, frecuentemente el intercambio de lazos anidados permite un aumento significativo en la cantidad de paralelismo que puede ser extraído de un programa. Esto puede ser aplicado para la multiplicación de matrices:

Para j = 1, n
    Para i= 1, n
        Para k = 1, n
            $C_{ij} = C_{ij} + A_{ik} * B_{kj}$

Como los lazos pueden ser intercambiados, tal que las *k* iteraciones del lazo interno puedan pasar ahora a ser las más externas, esto facilita la tarea de paralelización del lazo externo:

Para k = 1 ,n en Paralelo
    Para i= 1, n
        Para j= 1, n
            $C_{ij} = C_{ij} + A_{ik} * B_{kj}$

Así, con esta modificación en el código se pueden calcular en paralelo todos los elementos de C. Previamente, se deben difundir los valores de A y B.

Otro ejemplo de implementación paralela de la multiplicación de matrices fue dado en el capítulo 1, para el caso de las maquinas sistólicas. Otro ejemplo es el algoritmo de multiplicación de matrices por bloques, basado en el paradigma *divides y conquistaras* propuesto por Horowitz y Zorat [1, 8, 11, 13, 33]. Este es ideal para plataformas a memoria compartida, y matrices cuadraticas de poder 2. La idea es dividir las matrices a multiplicar A y B en 4 submatrices (App, Apq, Aqp, Aqq) y (Bpp, Bpq, Bqp, Bqq). Para calcular la multiplicación se requiere de 8 pares de submatrices las cuales se pueden usar para calcular en paralelo en 8 procesadores una parte de la multiplicación, y pares de ellos se deben sumar para el cálculo final de la matriz C resultante. Esto mismo se podría hacer en los grupos de las submatrices de manera recursiva para calcular los resultados de cada una de ellas. El código es:

P0: mat_mul(App, Bpp)
P1: mat_mul(Apq, Bqp)

P2: mat_mul(App, Bpq)
P3: mat_mul(Apq, Bqq)
P4: mat_mul(Aqp, Bpp)
P5: mat_mul(Aqq, Bqp)
P6: mat_mul(Aqp, Bpq)
P7: mat_mul(Aqq, Bqq)

Generalmente, el número de procesadores requeridos, al usar el enfoque recursivo, es poder de 8.

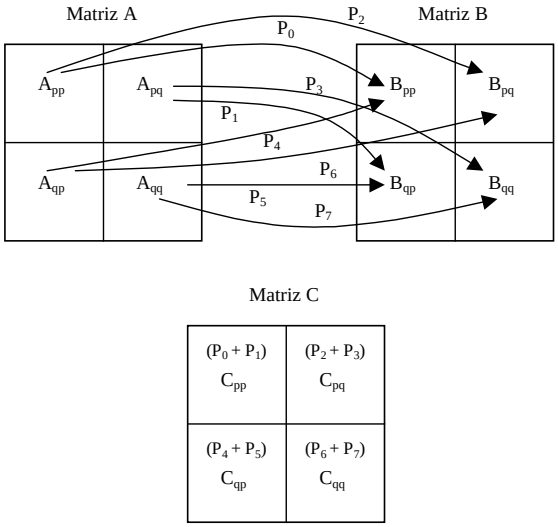


Figura 6.4. Otro esquema paralelo de Multiplicación de Matrices

6.4. Paralelización del Algoritmo de los Números Primos

Un eficiente equilibrio de la carga de trabajo entre los procesadores, es una parte importante del procesamiento paralelo. El programa para determinar los números primos en una secuencia de números, muestra la importancia del equilibrio de la carga de trabajo.

Un número primo es un número entero que tiene como únicos factores a 1 y a él mismo. En otras palabras, un número primo no es divisible por cualquier otro número que no sea 1 y el mismo número. Ejemplos son los números 1, 2, 3, 5, 7, 11, etc. El siguiente programa secuencial encuentra todos los números primos entre 1 y un dado limite superior *n*:

Para i=1, n
    Para j=2, sqrt(i)
        Si j divide a i, entonces i no es un número primo

Desde el punto de vista del código anterior, el limite superior del nodo interno (*j*) es función del índice del nodo externo (*i*). Así, cuando el valor de *i* aumenta, el lazo interno es ejecutado más veces. A continuación discutiremos en detalle dos formas de paralelizar este código:

Método 1: En este método se divide la carga entre los procesadores partiendo el lazo externo (*i*) en partes iguales. Así, el rango de i del 1 al *n* es dividido en intervalos iguales, tal que la variable *intervalo* es definida como intervalo=n/número_procesadores. El pedazo del lazo a enviar a los diferentes procesadores será:

Procesador 1: del 1 al intervalo
Procesador 2: del intervalo+1 al 2*intervalo
...
Procesador número_ procesadores: del (número_procesadores-1)*intervalo+1 a n

Se observa que en los últimos procesadores, el lazo interno *j* toma más tiempo. Así, a pesar que el lazo externo es dividido en partes iguales, la carga no es balanceada.

Método 2: En este caso se asigna la carga entre los procesadores al dividir el lazo externo (*i*) de una manera interpaginada, con un factor de interpaginamiento igual al número de procesadores. Así, el lazo externo en los diferentes procesadores se asigna en tal forma que sus iteraciones son distribuidas de la siguiente forma:

Procesador 1: 1, número_procesadores+1, 2*número_procesadores+1,
3*número_procesadores+1, ...
Procesador 2: 2, número_procesadores+1, 2*número_procesadores+2,
3*número_procesadores+2, ...
...
Procesador número_procesadores: número_procesadores,
número_procesador+número_procesadores, 2*número_procesadores+
número_procesadores, 3*número_procesadores+ número_procesadores, ...

El número de veces que el lazo externo es ejecutado en cada sitio, es el mismo, como en el caso anterior, es decir, el número de veces en que se divide el lazo, es el mismo. Además, el tiempo de ejecución del lazo interno es más o menos igual en todos los procesadores, ya que el valor de distribución del índice del lazo externo es más uniforme, así, la carga es más balanceada.

6.5 Dependencia del Vecino más Cercano

La característica de depender de los datos vecinos más cercanos, es común en muchas aplicaciones que resuelven ecuaciones diferenciales parciales. Esta dependencia puede ser descrita de la siguiente forma: el cálculo de un elemento de una matriz *A* en una iteración *k*, puede ser expresado por la ecuación siguiente:

$$a_{ij}^{k} = f(a_{ij-1}^{k-1}, a_{ij+1}^{k-1}, a_{i-1j}^{k-1}, a_{i+1j}^{k-1})$$

donde *f* es alguna función

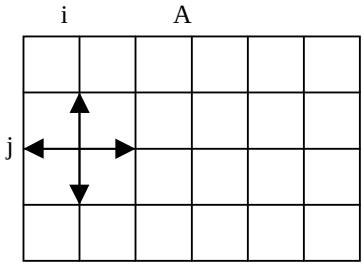


Figura 6.5. Dependencia de Vecinos

Suponiendo *m* iteración y una matriz de orden *n*n*, el procedimiento secuencial de un hipotético programa para el caso donde $a_{ij}^{k} = f(a_{ij-1}^{k-1}, a_{ij+1}^{k-1})$ sería:

```
Para k=1,m
    Para i=1, n
        Para j=1,n
            Si j=1
                bij=f(aij, aij+1)
            de lo contrario
                Si j=n
                    bij=f(aij-1, aij)
                de lo contrario
                    bij=f(aij-1, aij+1)
    Copiar B en A
```

La matriz intermedia *B* es usada para preservar los valores previos de la matriz *A* durante el cálculo de la iteración *k*. Como puede ser inferido del lazo principal del programa, el cálculo de la columna *j* de la matriz *B* requiere de las columnas *j-1* y *j+1* de *A*. De hecho, el cálculo de la columna *j* de *A* en la iteración *k* requiere de los valores de las columnas vecinas *j-1* y *j+1* de la iteración *k-1*. Así se refleja la dependencia entre los vecinos más cercanos. El algoritmo paralelo para este programa puede ser definido como:

- Dividir el dominio de los datos de la matriz *A* entre los procesadores, tal que cada procesador reciba dos columnas extras de *A* (una extra por cada borde). Estas columnas extras son requeridas para calcular las columnas de los bordes para cada dominio asignado en cada procesador.
- Calcular los valores de *B* por iteración.
- Después de cada iteración, intercambiar las columnas de los bordes de cada dominio en cada procesador. Esto es hecho, de tal manera que las columnas extras en cada procesador son actualizadas y están disponibles para la próxima iteración.

El último paso del procedimiento anterior puede ser implementado según el siguiente pseudo-código, donde P_1 a P_p son los procesadores usados, y las variables *baja, alta* son los limites inferiores y superiores de cada dominio asignado para ser calculado en cada procesador:

Para todo Procesador P_i
Si i=1
    Envio columna alta a P_2
    Recibo columna alta+1 de P_2
De lo contrario
    Si i=p
        Envio columna baja a P_{p-1}
        Recibo columna baja-1 de P_{p-1}
    De lo contrario
        Recibo columna baja-1 de P_{i-1}
        Envio columna alta a P_{i+1}
        Recibo columna alta+1 de P_{i+1}
        Envio columna baja a P_{i-1}

Este pseudo-código implementa un conjunto de envíos de comunicaciones entre los procesadores.

Ejemplo 6.1. El problema de las diferencias finitas en una dimensión, es tal que para el vector *X* de tamaño *n* se debe calcular X^t de la siguiente forma:

$$x_i^t = (x_{i-1}^{t-1} + 2x_i^{t-1} + x_{i+1}^{t-1})/4 \qquad \text{para } 1 \le i \le n \text{ y } 1 \le t \le T$$

Esto se debe repetir para cada elemento de *X*, tal que ningún elemento puede ser actualizado en el paso *t+1,* hasta que sus vecinos hayan sido actualizados en el paso *t*.

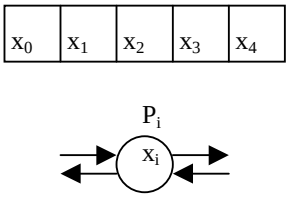


Figura 6.6. Paralelizacion del cálculo de la diferencia finita en una dimensión

Un algoritmo paralelo para este problema crea *n* tareas, una para cada punto de *X*. La tarea *i* es responsable por calcular los valores de x_i^1, ..., x_i^T, y para cada paso *t*, él debe recibir los valores de x_{i-1}^t, x_{i+1}^t, desde las tareas *i-1* e *i+1*. El algoritmos en cada procesador *i* sería:

1. Enviar el valor de x_i^t a sus vecinos derecho e izquierdo.
2. Recibir x_{i-1}^t e x_{i+1}^t desde sus vecinos derecho e izquierdo.
3. Calcular x_i^{t+1}

Se puede notar que las *n* tareas se pueden ejecutar simultáneamente, con una fase de sincronización debido a la recepción de datos. Esta sincronización asegura que ningún dato es actualizado en la iteración *t+1* hasta que sus vecinos hayan actualizado sus datos en la iteración *t*.

6.6 Búsqueda

Aquí se presenta un algoritmo que explora en un árbol de búsqueda los nodos que representan soluciones a un problema dado. Este algoritmo tiene la característica de crear tareas dinámicamente. El procedimiento secuencial, llamado *Buscar*, es:

Si solución(A) entonces
    Evaluar(A)
De los contrario
    Para cada hijo A(i) de A
        Buscar (A(i))

El algoritmo principal para este problema podría ser:

- *Crear una tarea para la raíz del árbol.*
- *Si no es una solución,*
  - *Crear una tarea por cada hijo.*

  De lo contrario
  - *Regresar su solución.*

Y para cada tarea recién creada:

- *Si no es una solución,*
  - *Crear una tarea por cada hijo.*

  De lo contrario
  - *Regresar su solución a sus padres.*

Esto se repite hasta que todas las soluciones lleguen al nodo raíz.

6.7 Algoritmo de Búsqueda del Camino más Corto

Este es un problema muy importante en la teoría de grafos, por sus aplicaciones en comunicación, transporte, electrónica, etc. Este problema consiste en encontrar el camino más corto entre todos los pares de vértices de un grafo. El grafo es definido por G=(V, E),

donde V representa al conjunto de vértices y E al conjunto de arcos. Además, G tiene definida una matriz de adyacencia A tal que a_{ij}=1 si hay un arco entre los vértices v_i y v_j, y a_{ij}=0 en caso contrario. Un camino desde v_i a v_j es una secuencia de arcos donde los vértices extremos de la secuencia son estos dos vértices. El camino más corto entre dos vértices, es el camino con el menor número de arcos. El problema del *camino más corto entre todos los pares* requiere que se encuentre el camino más corto de todos los pares de vértices en un grafo. Un ejemplo del algoritmo secuencial para este problema es el método denominado, algoritmo de Floyd:

si $i=j$
    $I_{ij}(0)=0$
si $a_{ij}=1$ *y además,* $i \neq j$
    $I_{ij}(0)=E(v_i,v_j)$
De lo contrario
    $I_{ij}(0)=\infty$
Para $k=0, n-1$
    Para $i=0, n-1$
        Para $j=0,n-1$
            $I_{ij}(k+1)=min(I_{ij}(k), I_{ik}(k)+I_{kj}(k))$
$S=max(I_{ij}(n)) \quad \forall i,j$

Este programa construye cada paso *k* en la matriz intermedia *I(k)*, la cual contiene la distancia más corta conocida hasta ese momento entre cada par de nodos. Cada paso *k* del algoritmo considera a cada I_{ij} y determina si el camino más corto conocido desde v_i a v_i es más largo que las longitudes combinada desde el camino más corto conocido desde v_i a v_k y desde v_k a v_j. Si es así, la entrada I_{ij} es actualizada para reflejar el camino más corto. Una posible paralelización de ese algoritmo se basa en una descomposición del dominio de las filas de la matriz intermedio *I*. Así, el algoritmo puede usar *n* procesadores. Cada procesador tiene una o más filas de *I* y es responsable por calcular esas filas. Así, el algoritmo paralelo sería:

Para $k= 0, n-1$
    Para i= valor inicial local de i, valor final local de i
        Para $j=0,n-1$
            $I_{ij}(k+1)=min(I_{ij}(k), I_{ik}(k)+I_{kj}(k))$

En el paso *k*, cada procesador requiere, además de los datos locales, los valores para calcular I_{kj}, es decir, la fila *k* de *I*. Así, el procesador con esta fila debe difundirlo a todos los otros procesadores.

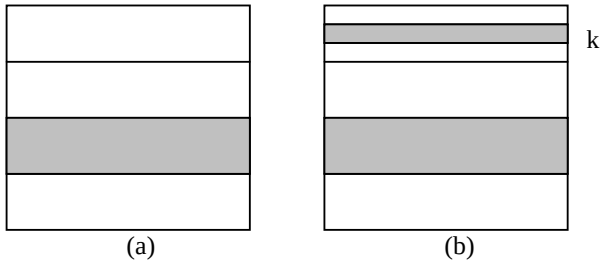


Figura 6.7. Versión paralela del Algoritmo de Floyd basada en una descomposición en una dimensión

Otro esquema parecido al anterior, pero usando una descomposición en dos-dimensiones, es la siguiente:

Para k= 0, n-1
Para i= valor inicial local de i, valor final local de i
Para j= valor inicial local de j, valor final local de j
$I_{ij}(k+1)= min(I_{ij}(k), I_{ik}(k)+I_{kj}(k))$

Ahora, para cada paso cada tarea requiere, además de sus datos locales, N/sqrt(P) valores desde las tareas con sus mismas filas o columnas del arreglo de tareas de dos dimensiones. Así, se requieren dos operaciones de difusión: desde la tarea en cada fila que posee la parte de la columna *k* a todas las otras tareas que tengan esa fila, y desde la tarea en cada columna que posee parte de la fila *k* a todas las otras tareas que tengan esa columna.

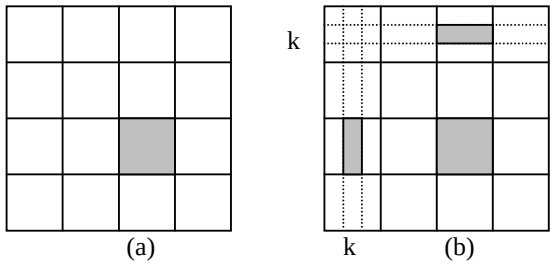


Figura 6.8. Versión paralela del Algoritmo de Floyd basada en una descomposición en dos dimensiones

6.8 Paralelización de LINPACK

La librería LINPACK es una colección de programas de Fortran que resuelven sistemas de ecuaciones lineales de la forma AX=B, donde *A* es una matriz del tamaño n*n y *B* es un vector del tamaño *n*. Esta librería ha sido usada como un programa de prueba estándar para comparar el rendimiento de diferentes sistemas de computadores. Supongamos el sistema de ecuaciones lineales:

$a_{n-1,0}x_0 + a_{n-1,1}x_1 + ... + a_{n-1,n-1}x_{n-1} = b_{n-1}$
.
.
.
$a_{0,0}x_0 + a_{1,0}x_1 + ... a_{0,n-1}x_{n-1} = b_0$

El objetivo es hallar x_0, ..., x_{n-1} dados $a_{0,0}$, ..., b_n. Es bueno resaltar que la matriz *A* puede ser una matriz densa o esparcida, dependiendo del número de 0s que tenga. Será densa si el número de 0s es poco, de lo contrario será esparcida. Uno de los métodos para resolver este problema es el de Eliminación Gaussiana. Se basa en el hecho de que cualquier fila puede ser reemplazada por esa fila añadida a otra fila multiplicada por una constante. El procedimiento comienza en la primera fila y sigue hasta el final. En la fila *i*, cada fila *j* antes de *i* ha sido reemplazada por fila j + (fila i)($-a_{ji}/a_{ii}$). La constante usada por la fila *j* es $-a_{ji}/a_{ii}$. Esto tiene el efecto de hacer a todos los elementos de la columna *i* antes de la fila *i* iguales a 0, ya que:

$a_{ji}=a_{ji}+a_{ii}(-a_{ji}/a_{ii})=0$

En particular, en esta sección se presenta la paralelización del método estándar de eliminación de Gauss, el cual consiste en reducir los coeficientes de las matrices para formar la triangular superior. La complejidad de este método para resolver sistemas de ecuaciones lineales es de $O(n^3)$ operaciones de punto flotante.

Dos procedimientos claves del método a paralelizar son los de triangularización y de sustitución. Por otro lado, el programa principal genera la matriz *A* y el vector *B* usando un generador de números aleatorios, llama las dos rutinas anteriores y verifica los resultados. La rutina de triangularización reduce la matriz de coeficientes aumentados (*A* aumentada con *B*) a la forma de triangular superior. Para eso suponemos $a_{ii}\neq 0$, $\forall i=1,n$ y que $A=(a_{ij})$, $1\leq i\leq n$, $1\leq j\leq n+1$ es la matriz original A aumentada en la columna n+1 por el vector B. El código principal de la rutina de triangularización es:

```
Para i=1,n-1
        Para j=i+1,n
                Manipular columna j (transformación lineal) usando columna i
```

La transformación sobre cada columna es realizada según el siguiente cálculo:

A(j)=A(j)+A(i) escalar*

donde A(j) es la columna *j* de la actual matriz *A,* A(i) es la columna *i* de la actual matriz *A* y el escalar es igual a -a(i,j)/a(i,i). Así, al final se tendrá la matriz A en forma triangular superior. Ese cálculo es hecho de la siguiente forma:

```
Para i=1,n-1
        Para j=i+1, n+1
                temp=a(i,j)/a(i,i)
                Para k=1,n-i
```

```
a(j,k)=a(j,k)-temp*a(i,k)
```

En cuanto a la rutina de sustitución, su código principal es:

```
x(n,n)=a(n,n+1)/a(n,n)
Repite para i=n-1,1, -1
        temp=0
        Repite para j=i+1,n
                temp=temp+a(i,j)*x(j)
        x(i)=(a(i,n+1)-temp)/a(i,i)
```

En dichas rutinas, lo que realmente toma mucho tiempo es el cálculo de la triangular superior. Como se observa en esas rutinas, el número de cálculos envueltos disminuye a medida que progresa el lazo externo. De manera de darle una carga semejante a todos los procesadores, es útil distribuir las columnas según un esquema *round robin* (asignación circular), como se indica a continuación:

Si hay *p* procesadores y *n+1* columnas entonces el procesador P_1 procesará las columnas 1, p+1, 2*p+1, ..., el procesador P_2 procesará las columnas 2, p+2, 2*p+2, ..., y el procesador Pp procesará las columnas p, 2*p, 3*p, ...

Para iniciar el procesamiento, el primer procesador enviará a todos, la primera columna, y todos los procesadores manipularan sus columnas. En el segundo lazo, el segundo procesador es la fuente y enviará a todos, la segunda columna ya modificada, y todos los procesadores manipularán sus columnas. Este procesamiento es repetido para las n+1 columnas. Una vez que ha sido completado, todos los procesadores deben enviar al procesador maestro sus columnas procesadas para reconstruir la matriz triangular superior.

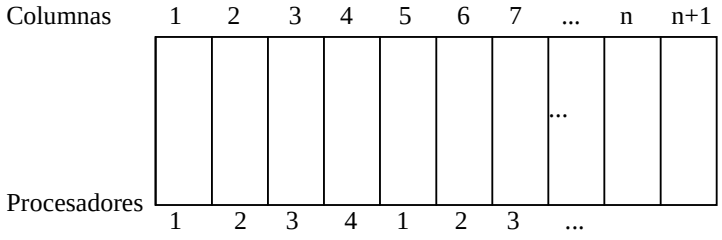


Figura 6.9. Paralelización del algoritmo de LINPACK según un esquema round robin de asignación de trabajo, para 4 procesadores.

Este procedimiento se puede ver de la siguiente forma en cada procesador:

```
Ilazo= índice de la primera columna en ese procesador
Iprev= identificador del procesador que lo antecede
Yo= identificador del procesador donde se está ejecutando esta rutina
Para i=1,n-1
        aux=i+1
        Ifuente=mod(i-1,p)
        Si (Ifuente=yo) entonces
                Enviar columna i a todos los procesadores
        De lo contrario
                Recibir columna i desde procesador Ifuente
        Para j=Ilazo, n+1, p
                temp=a(i,j)/a(i,i)
                Para k=1,n-i
                        a(j k)=a(j,k)-temp*a(i,k)
        Si Ifuente=Iprevio
                Ilazo=Ilazo+p
```

Además, para acumular todas las transformaciones en el procesador central para la sustitución, se requiere ejecutar en él a:

```
ncol=mod(n-1,p)
Aux2=mod(n,p)
Si ncol =0
        Recibir columna n del procesador P_ncol
Si Aux2≠0
        Recibir columna n+1 del procesador P_Aux2
```

En ese momento ya se habrá calculado la triangular superior de la matriz A y podrá ser resuelto por el método de sustitución.

Otro esquema de paralelización del método de eliminación de Gauss es dado a continuación. Un código clásico secuencial es:

```
Para i=1, i<n-1
        Para j=i+1, j<n
                m=a_ji/a_ii
                Para k=i, k<n
                        a_jk=a_jk-a_ik*m
                b_j=b_j-b_i*m
```

El lazo interno envuelve operaciones independientes sobre los elementos de la fila *j*. Una manera de paralelizar este problema es colocar en cada procesador una fila con los elementos de las otras filas que requiere y trabajar sobre esa fila. Así, antes de que un

procesador que tiene la fila i+1 pueda operar sobre ella, debe recibir los elementos de las filas <*i+1*. Primero, el procesador P_1 (que tiene a la fila 1) difunde los elementos de su fila a cada uno de los otros p-1 procesadores. Entonces, ellos realizan sus cálculos (multiplicaciones) para modificar sus filas. El procedimiento es repetido con cada procesador P_2,... P_p difundiendo los elementos de sus filas. Los elementos de la fila *i* difundidos por P_i son a_{ii}, ..., $a_{i,n}$, y $a_{i,n+1}$, es decir, un total de n-i+1 elementos. Los procesadores P_{i+1} a P_p recibirán el mensaje difundido. El código de esa parte será:

Para i=1, n-1

Procesador P_i difunde su fila a los procesadores P_{i+1} a P_p

Procesadores P_{i+1} a P_p realizan su cálculos

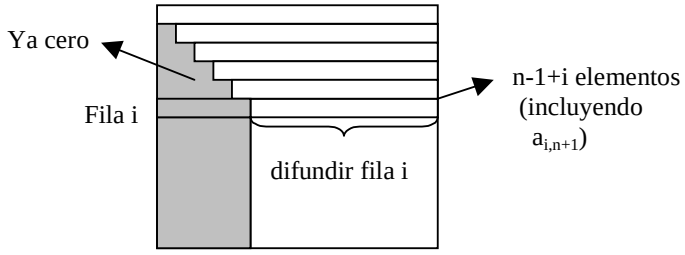


Figura 6.10. implementación paralela del método de Eliminación de Gauss

Si *n* es mayor que el número de procesadores *p*, se deben hacer particiones de las filas para ser agrupadas y procesadas en cada uno de los procesadores. Como en el caso anterior, para equilibrar la carga, ya que los elementos a la derecha se van haciendo 0, se debe hacer una partición cíclica que permita repartir la carga de trabajo equitativamente entre todos los procesadores.

6.9 Inversión de una Matriz

Una matriz se puede invertir usando, por ejemplo, el método de eliminación de Gauss. El algoritmo de ese método consiste en aumentar la matriz de entrada con una matriz de identidad del mismo orden. Entonces, transformaciones elementales son aplicadas sobre dicha matriz aumentada, hasta reducir la parte que representa la matriz de entrada a una forma de identidad. Al concluir esta última operación, la parte aumentada resultante, será la inversa de la matriz original.

El método de Gauss parte de la siguiente propiedad: $A*A^{-1}=I$, suponiendo que $a(i,i)\neq 0$. Así, la entrada *A* será aumentada con *I*. Si suponemos una matriz cuadrática n*n, la operación será:

```
Para i=1, n
        Dividir fila i de A y A-1 entre a(i,i)
        Para j=2, n
                Si j ≠ i
                        temp=a(j,i)
                        Para k=1,n
                                a(j,k)=a(j,k)-a(i,k)*temp
                                a-1(j,k)=a-1(j,k)-a-1(i,k)*temp
```

Se evidencia de ese pseudo-código, que para cada instancia del lazo externo *i*, las operaciones son ejecutadas para todas las filas y columnas de la matriz aumentada. Así, la mejor y más simple vía de paralelizarlo es partir la matriz de entrada (A) y la matriz de identidad (A^{-1}) por filas según el número de procesadores, es decir, cada procesador tendrá un bloque de filas a calcular. Dicha partición no necesita ser entrepaginada porque el número de operaciones por instancia del lazo externo es constante. Después de dividir la matriz, durante cada instancia del lazo *i*, el procesador que tiene la fila *i* debe enviarlo a todos los otros, para que todos los procesadores puedan procesar sus respectivas filas. El pseudo-código en cada procesador será:

```
Para i=1,n
        Ifuente = identidad del procesador con fila i
        Si yo=Ifuente
                Dividir fila i entre a(i,i)
                Enviar fila i modificada a todos
        De lo contrario
                Recibir fila i
        Para j=iinicial, ifinal (limite superior e inferior del bloque que tiene procesador actual)
                Si j≠i
                        temp=a(j,i)
                        Para k=1,n
                                a(j,k)=a(j,k)-a(i,k)*temp
                                a-1(j,k)=a-1(j,k)-a-1(i,k)*temp
```

6.10 Operaciones de Reducción

Estas operaciones son frecuentemente encontradas en situaciones donde se tiene una estructura, por ejemplo un vector, y se reduce a otra más simple estructura, por ejemplo a un escalar. Un ejemplo es la suma de los elementos de un vector. En general, asumamos *n* números escalares $x_1, ..., x_n$ y una operación binaria asociativa &, se pueden realizar operaciones asociativas entre ellos en vías del cálculo final. Por ejemplo, supongamos el vector X compuesto por tres elementos escalares, entonces se pueden hacer las siguientes operaciones sobre ellos: $x_1\&(x_2\&x_3)$ o $(x_1\&x_2)\&x_3$. Las operaciones típicas & son suma, producto, maximización, etc. Esquemas asociativos más complejos se pueden hacer, por ejemplo para:

$(...[(x_1 \& x_2) \& (x_3 \& x_4)] \& ... \& [(x_{n-3} \& x_{n-2}) \& (x_{n-1} \& x_n)]...)$

tenemos:

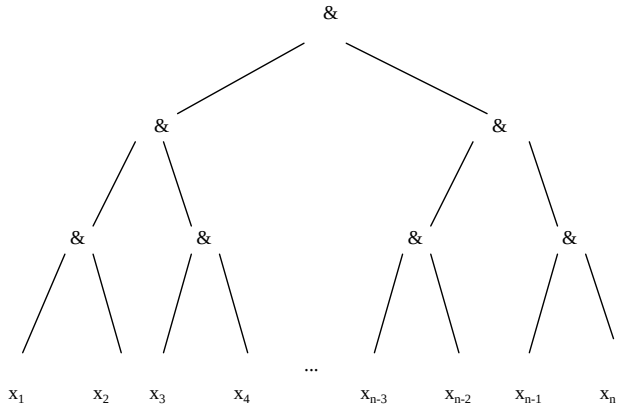


Figura 6.11. Esquema paralelo de Reducción

Donde la raíz almacena el resultado final. Esto significa que el resultado se obtiene en $\log_2(n)$ pasos, donde en el primer paso hay n/2 operaciones, después n/4, y así sucesivamente. Suponiendo la lista de *n* elementos $x_1, ..., x_n$, un posible esquema de solución es:

Para j=1, n en paralelo
    S[0,j]=x_j
Para i=1, log(n)
    Para j=1, $2^{log(n)-i}$ en paralelo
        S[i,j]=S[i-1,2j-1]&S[i-1,2j]

Es claro que el resultado final se almacenará en la raíz S[log(n),1] después de O(log(n)) pasos.

Bibliografía

[1] S. Akl, "The Design and Analysis of Parallel Programs, Prentice-Hall, 1989

[2] G. Almasi, A. Gottlieb, "Highly Parallel Computing", Benjamin/Cummings, 1994.

[3] J. Baek, "Concurrent Systems", Addison Wesley, 1998.

[4] L. Baker, B. Smith, "Parallel Programming", McGraw-Hill, 1996.

[5] J. Banâtre, "La programmation parallèle: outils, méthodes et éléments de mise en oeuvre", Eyrolles, Francia, 1991.

[6] P. Banerjee, "Parallel Algorithms for VLSI Computer-Aided Design", Prentice-Hall, 1994.

[7] U. Banerjee, "Dependence Analysis for Supercomputing", Kluwer Academic, 1988.

[8] D. Bertsekas, J. Tsitsiklis, "Parallel and Distributed Computation: Numerical Methods", Prentice-Hall, 1989.

[9] N. Carriero, D. Gelernter, " How to Write Parallel Programs", MIT, 1990.

[10] K. Chandy, S. Taylor, "An Introduction to Parallel Programming", Jones and Bartlett, 1992.

[11] P. Chaudhuri, "Parallel Algorithms: Design and Analysis", Prentice-Hall, 1992

[12] M. Cosnard, M. Nivat, Y. Robert, "Algorithmique parallèle", North Holland, 1992

[13] M. Cosnard, D. Trystran, "Parallel Algorithms and Architectures", International Thomson Computer, 1995.

[14] H. El-Rewini, T. Lewis, H. Ali, "Task Scheduling in Parallel and Distributed Systems", Prentice-Hall, 1994.

[15] I. Foster, "Designing and Building Parallel Programs", Addison Wesley, 1995.

[16] T. Fountain, "Parallel Computing: Principles and Practice", Cambridge University, 1994

[17] M. Gengler, S. Ubeda, F. Desprez, "Initiation au parallélisme: concepts, architectures et algorithmes", Masson, 1996.

[18] C. Germain, J. Sansonnet, "Les ordinateurs massivement parallèles", Armand Colin, 1991.

[19] G. Gibson, "Redundant Disk Array: Reliable, Parallel Secondary Storage", MIT, 1992.

[20] G. Golub, J. Ortega, "Scientific Computing: An Introduction with Parallel Computing", Academic, 1993.

[21] W. Gropp, E. Lusk,, A. Sjellum, "Using MPI: Portable Parallel Programming with the Message Passing Interface", MIT, 1995.

[22] W. Gropp, E. Lusk, R. Thakur, "Using MPI-2: Advanced Features of the Message Passing Interface", MIT, 1999.

[23] G. Haring, G. Kotsis, "Performance Measurement and Visualization of Parallel Systems", Elsevier Science, 1993.

[24] K. Huang, "Scalable Parallel Computing", McGraw-Hill, 1998.

[25] K. Hwang, "Advanced Computer Architecture: Parallelism, Scalability, Programmability", McGraw-Hill, 1993.

[26] J. Jacquemin, "Informatique Parallèle et Systèmes Multiprocesseurs", Hermes, Francia, 1993.

[27] V. Kumar, A Grama, A. Gupta, G. Karypis, "Introduction to Parallel Computing", Benjamin/Cummings, 1993.

[28] D. Kuck, "High Performance Computing", Oxford University, 1996.

[29] E. Leiss, "Parallel and Vector Computing: A Practical Introduction", McGraw-Hill, 1995.

[30] T. Lewis, H. El-Rewini, "Introduction to Parallel Programming", Prentice-Hall, 1992.

[31] J. May, "Parallel I/O for High Performance Computing", Morgan Kaufmann, 2001.

[32] M. Quinn, "Parallel Computing: Theory and Practice", McGraw-Hill, 1994.

[33] J. Smith, "The Design and Analysis of Parallel Algorithms", Oxford University, 1993.

[34] H. Stone, "High Performance Computer Architectures", Addison Wesley, 1993.

[35] A. Xavier, P. Iyengar, "Introduction to Parallel Algorithms", Wiley Interscience, 1998.

[36] C. Xu, F. Lau, "Load Balancing in Parallel Computers: Theory and Practice", Kluwer Academic, 1998.

[37] B. Wilkinson, M. Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Prentice-Hall, 1999

[38] M. Wolfe, "High Performance Compilers for Parallel Computing", Addison Wesley, 1996.

[39] H. Zima, B. Chapman, "Supercompilers for Parallel and Vector Computers", Addison Wesley, 1991.

[40] A. Zomaya, "Parallel and Distributed Computing Handbook", McGraw-Hill, 1996.

Glosario

Escalabilidad: es la posibilidad de extender, o hacer crecer, a un sistema.

Escalar: en este libro es usado de dos formas: i) un tipo de número ii) para indicar el tipo de extensión que se le puede hacer a un sistema, de tal forma de aumentar su potencial.

Encapsular: agrupar en algún tipo de abstracción a un grupo de objetos.

Encauzamiento (pipelining): es un tipo de procesamiento en cadena, lo que implica que ciertas operaciones se traslapan para lograr una mayor velocidad de procesamiento.

Enrutamiento: mecanismos en los sistemas de comunicación para definir una ruta que se debe seguir entre dos sitios en el momento del envío de mensajes.

Incrustar/fijar/empotrar (embedding): sistema controlado por un software especializado, desarrollado de acuerdo a unas necesidades especificas y adaptado al problema en particular.

Latencia: lapso de tiempo para realizar una operación sobre una plataforma dada.

Memoria Escondida (cache memory): memoria veloz usada por los procesadores para ejecutar algunas funciones rápidamente.

Banco de Memorias (memory bank): descomposición de una memoria para permitir transferencias de información simultanea.

Difusión (Broadcast): envío de un mensaje a todos los componentes de un sistema.

Reenrollar: descomponer un lazo siguiendo los valores de los índices que lo caracterizan.

Sobrecosto (overhead): es introducido por alguna función, normalmente, del sistema operativo.

Puntos de referencia o programas de comparación del rendimiento (benchmark): tipos de programas para evaluar el rendimiento de un sistema computacional.