Centro Nacional de Cálculo Científico
Universidad de Los Andes
CeCalCULA
Mérida - Venezuela

Introducción a la Computación Paralela

Herbert Hoeger
hhoeger@ing.ula.ve

CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

1. INTRODUCCION

Tradicionalmente, la simulación numérica de sistemas complejos como dinámica de fluidos, clima, circuitos electrónicos, reacciones químicas, modelos ambientales y procesos de manufacturación, han impulsado el desarrollo de computadores cada vez más potentes. Hoy en día estas máquinas están siendo promovidas por aplicaciones comerciales que requieren procesar grandes cantidades de datos. Entre ellas encontramos realidad virtual, vídeo conferencias, bases de datos paralelas y diagnóstico médico asistido por computadoras.

La eficiencia de un computador depende directamente del tiempo requerido para ejecutar una instrucción básica y del número de instrucciones básicas que pueden ser ejecutadas concurrentemente. Esta eficiencia puede ser incrementada por avances en la arquitectura y por avances tecnológicos. Avances en la arquitectura incrementan la cantidad de trabajo que se puede realizar por ciclo de instrucción: memoria bit-paralela[1], aritmética bit-paralela, memoria cache[2], canales, memoria intercalada, múltiples unidades funcionales, *lookahead*[3] de instrucciones, *pipelining*[4] de instrucciones, unidades funcionales *pipelined* y *pipelining* de datos. Una vez incorporados estos avances, mejorar la eficiencia de un procesador implica reducir el tiempo de los ciclos: avances tecnológicos.

Hace un par de décadas, los microprocesadores no incluían la mayoría de los avances de arquitectura que ya estaban presentes en los supercomputadores. Esto ha causado que en el último tiempo el adelanto visto en los microprocesadores haya sido significativamente más notable que el de otros tipos de procesadores: supercomputadores, mainframes y minicomputadores. En la Figura 1 se puede apreciar que el crecimiento en la eficiencia para los

---

[1] *n* bits ($n>1$) son procesados simultáneamente en oposición con bit-serial en donde solo un bit es procesado en un momento dado.

[2] La memoria cache es un buffer de alta velocidad que reduce el tiempo efectivo de acceso a un sistema de almacenamiento (memoria, disco, CD, etc.). El cache mantiene copia de algunos bloques de datos: los que tengan más alta probabilidad de ser accesados. Cuando hay una solicitud de un dato que esta presente en el cache, se dice que hay un *hit* y el cache retorna el dato requerido. Si el dato no esta presente en el cache, la solicitud es pasada al sistema de almacenamiento y la obtención del dato se hace más lenta.

[3] Consiste en buscar, decodificar y buscar los operadores de la siguiente instrucción mientras se está ejecutando la instrucción actual.

[4] *Pipelining* se puede ver como la división de una tarea en varias subtareas cada una de las cuales puede ser ejecutada independientemente como en una línea de producción.

minicomputadores, mainframes y supercomputadores ha estado por debajo del 20% por año, mientras que para los microprocesadores ha sido de un 35% anual en promedio.

El tiempo para ejecutar una operación básica definitivamente depende del tiempo de los ciclos del procesador, es decir; el tiempo para ejecutar la operación más básica. Sin embargo, estos tiempos están decreciendo lentamente y parece que están alcanzando limites físicos como la velocidad de la luz (Figura 2). Por lo tanto no podemos depender de procesadores más rápidos para obtener más eficiencia.

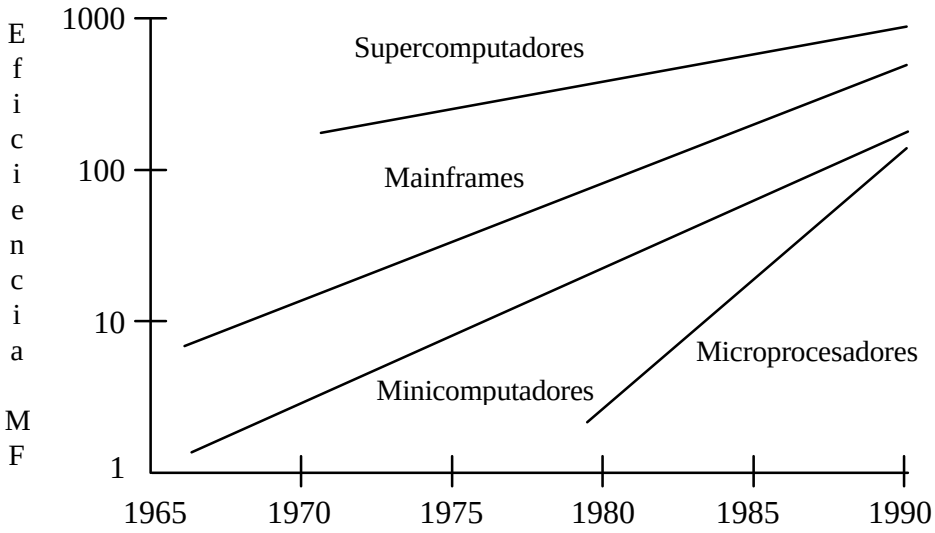

Figura 1. Creciemiento en la eficiencia de los computadores en megaflops.

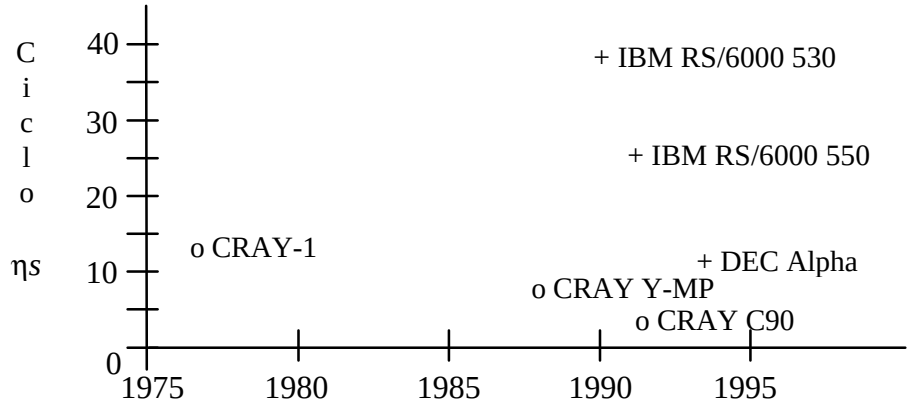

Figura 2. Tendencias en el tiempo de los ciclos (o = supercomputadores, + = microprocesadores RISC).

Dadas estas dificultades en mejorar la eficiencia de un procesador, la convergencia relativa en eficiencia entre microprocesadores y los supercomputadores tradicionales, y el relativo bajo costo de los microprocesadores[5], ha permitido el desarrollo de computadores paralelos viables comercialmente con decenas, cientos y hasta miles de microprosesadores.

Un **computador paralelo** es un conjunto de procesadores capaces de cooperar en la solución de un problema. Esta definición incluye supercomputadores con cientos de procesadores, redes de estaciones de trabajo y máquinas con múltiples procesadores.

2. APLICACIONES DEMANDANTES DE RECURSOS

En grosso modo podemos dividir las aplicaciones que requieren de recursos computacionales y de comunicación importantes de la siguiente forma:

2.1. APLICACIONES INTENSIVAS EN CALCULO

Estas son aplicaciones que requieren muchos ciclos de máquinas y son las que han impulsado el desarrollo de *supercomputadores*. Son típicas en ciencias e ingeniería, aunque recientemente han aparecido en otras áreas como simulación financiera y económica. Dependen grandemente de la velocidad y el procesamiento de punto flotantes de los supercomputadores. Entre ellas tenemos:

1. Dinámica de fluidos computacional.
2. Simulaciones electromagnéticas.
3. Modelado ambiental.
4. Dinámica estructural.
5. Modelado biológico.
6. Dinámica molecular.
7. Simulación de redes.
8. Modelado financiero y económico.

2.2. APLICACIONES DE ALMACENAMIENTO MASIVO

Estas aplicaciones dependen de la capacidad para almacenar y procesar grandes cantidades de

[5] Tienen una demanda sustancialmente mayor que la de otros procesadores que permite dividir los costos de diseño, producción y comercialización entre más unidades.

información. Requieren de un acceso rápido y seguro a una masa considerable de datos almacenados. Por ejemplo:

1. Análisis de data sísmica.
2. Procesamiento de imágenes.
3. Minería de datos.
4. Análisis estadístico de datos.
5. Análisis de mercados.

2.3. APLICACIONES EXIGENTES COMUNICACIONALMENTE

Estas son relativamente nuevas y pueden ser llamadas servicios por demanda. Requieren de recursos computacionales conectados por redes con anchos de banda considerables. Ejemplos:

1. Procesamiento de transacciones en línea.
2. Sistemas colaborativos.
3. Texto por demanda.
4. Vídeo por demanda.
5. Imágenes por demanda.
6. Simulación por demanda.

Obviamente que todas las aplicaciones anteriores dependen en cierto grado de cada uno de los aspectos computacionales mencionados: poder de computo, capacidades de almacenamiento y eficientes canales de comunicación, sin embargo las podemos agrupar por su característica dominante.

2.4. SISTEMAS DE SISTEMAS

Las aplicaciones en este grupo combinan en forma más compleja las características anteriores y dependen, en muchos casos, de sistemas computacionales integrados diseñados primordialmente para ellas. En este grupo tenemos aplicaciones como:

1. Soporte a decisiones corporativas y gubernamentales.
2. Control de sistemas a tiempo real.
3. Banca electrónica.
4. Compras electrónicas.
5. Educación.

Hay una alta correspondencia entre la evolución de las tecnologías computacionales y el desarrollo de aplicaciones; en particular el *hardware* tiene gran influencia en el éxito de ciertas áreas. La aplicaciones intensivas en cálculo fueron estimuladas principalmente por *máquinas vectoriales* y *procesadores masivamente paralelos*. Las aplicaciones de almacenamiento masivo han sido guiadas por dispositivos de almacenamiento como *RAID* y *robots de cintas*. Las aplicaciones exigentes comunicacionales como *herramientas colaborativas basadas en WWW* y *servicios por demanda en línea* originalmente surgieron con las *LAN* y estan creciendo drásticamente con *Internet*.

Definitivamente, las nuevas aplicaciones hacen uso de combinaciones complejas de poder de computo, almacenamiento y comunicación.

3. ORGANIZACION DE PROCESADORES

La organización de los procesadores o red se refiere a como se **conectan** o **enlazan** los procesadores o **nodos** en un computador paralelo. Entre los criterios que existen para evaluar los distintos diseños tenemos:

1) El **diámetro** de la red viene dado por la mayor distancia entre dos nodos. Mientras menor sea el diámetro menor será el tiempo de comunicación entre nodos arbitrarios.
2) El **ancho de bisección** de la red es el menor número de enlaces que deben ser removidos para dividir la red por la mitad. Un ancho de bisección alto es preferible porque puede reducir el tiempo de comunicación cuando el movimiento de datos es sustancial, ya que la información puede viajar por caminos alternos y así evitar o reducir la congestión entre ciertos nodos de la red. Igualmente un ancho de bisección alto hace el sistema más tolerante a fallas debido a que defectos en un nodo no hacen inoperable a todo el sistema.
3) Es preferible que el número de enlaces por nodo sea una constante independiente del tamaño de la red, ya que hace más fácil incrementar el número de nodos.
4) Es preferible que la longitud máxima de los enlaces sea una constante independiente del tamaño de la red, ya que hace más fácil añadir nodos. Por lo tanto, es recomendable que la red se pueda representar tridimensionalmente.
5) Redes **estáticas** y **dinámicas**. En las redes estáticas la topología de interconexión se define cuando se construye la máquina. Si la red es dinámica, la interconexión puede variar durante la ejecución de un programa o entre la ejecución de programas. Entre dos redes, una estática y la otra dinámica, que ofrezcan el mismo grado de conectividad entre los nodos, la dinámica es menos costosa de implementar (requieren menos puertos y

cableado) pero incrementa el tiempo promedio que un nodo ha de esperar por una vía de comunicación libre.

A continuación se presentan algunos tipos de organizaciones de redes de procesadores. Estas organizaciones también son aplicables a interconexiones de módulos de memoria.

3.1. BUS Y ETHERNET

En una red basada en un bus, los procesadores comparten el mismo recurso de comunicación: el **bus**. Esta arquitectura es fácil y económica de implementar, pero es altamente no escalable ya que solo un procesador puede usar el bus en un momento dado; a medida que se incrementa el número de procesadores, el bus se convierte en un cuello de botella debido a la congestión. Esta topología en muy popular en multiprocesadores de memoria compartida como el Encore Multimax y el Sequent Symetry, en donde el bus (Figura 3) es usado para leer y escribir en la memoria global (compartida). En principio, la memoria global simplifica la programación paralela ya que no hay que tomar en cuenta la *localidad*[6]. Sin embargo, la mayoría de las máquinas paralelas de memoria compartida usan memorias cache para reducir el trafico en el bus; por lo tanto, la localidad continua siendo importante ya el acceso al cache es mucho más rápido que a la memoria compartida.

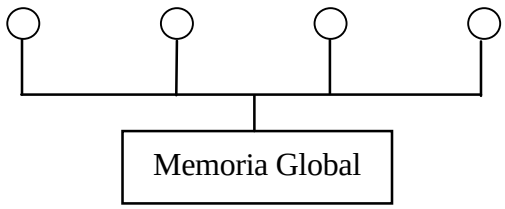

Figura 3. Bus.

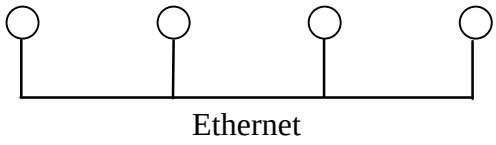

Figura 4. Ethernet.

Un conjunto de máquinas conectadas por **Ethernet** es otro ejemplo de una interconección por bus. Es muy popular para interconectar estaciones de trabajo y computadores personales. Todos los computadores conectados via Ethernet comparten el mismo canal de comunicación (Figura

[6] El acceso a la memoria local (en el mismo nodo) es menos costoso que el acceso a memoria remota (en otro nodo). Por lo tanto es deseable que el acceso a datos locales sea más frecuente que a datos remotos.

4). Una máquina que requiere enviar un mensaje tiene que esperar hasta que el canal este libre; si detecta una colisión, espera cierto tiempo y retrasmite.

3.2. MALLAS

Al igual que el bus (o ethernet), las mallas son fáciles y económicas de implementar, sin embargo el diámetro se incrementa al añadir nodos. En las mallas de dimensión 1, si los dos nodos extremos también son conectados, entonces se tiene un anillo (Figura 5).

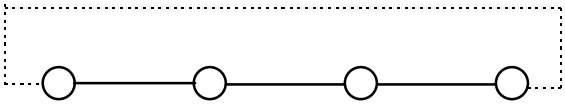

Figura 5. Malla de dimensión 1 (lineal) o anillo.

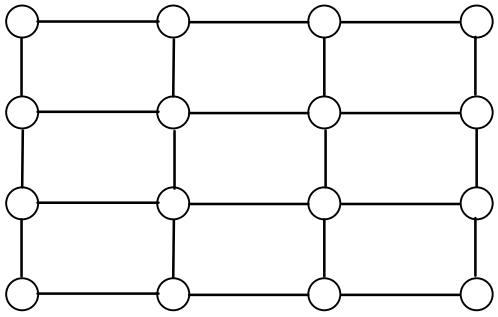

Figura 6. Malla bidimensional.

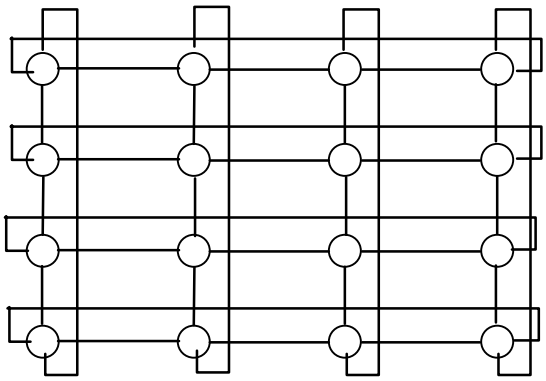

Figura 7. Torus bidimensional.

Mallas de 2 dimensiones (Figura 6) y de 3 dimensiones son comunes en computación paralela y tiene la ventaja de que pueden ser construidas sin conexiones largas. El diámetro de las mallas puede ser reducido a la mitad si se extiende la malla con conexiones toroidales de forma que los procesadores en los bordes también estén conectados con vecinos (Figura 7). Esto sin embargo presenta dos desventajas: a) conexiones más largas son requeridas y b) un subconjunto de un torus no es un torus y los beneficios de esta interconexión se pierden si la máquina es particionada entre varios usuarios. Los siguientes arreglos de procesadores[7] usan mallas

[7] Posteriormente se definirá lo que son arreglos de procesadores y multicomputadores.

bidimensionales: Goodyear Aerospace MPP, AMT, DAP y MasPar MP-1. El multicomputador[7] Intel Paragon XP/S también conecta sus procesadores mediante mallas bidimensionales.

3.3. MARIPOSA (BUTTERFLY)

Comparada con las mallas, esta topología presenta menor diámetro. La máquina BBN TC2000 usa una red en mariposa para enrutar datos entre procesadores.

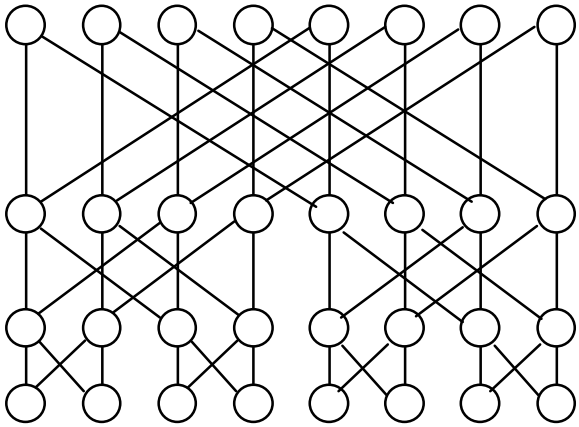

Figura 8. Mariposa o Butterfly.

3.4. ARBOLES BINARIOS

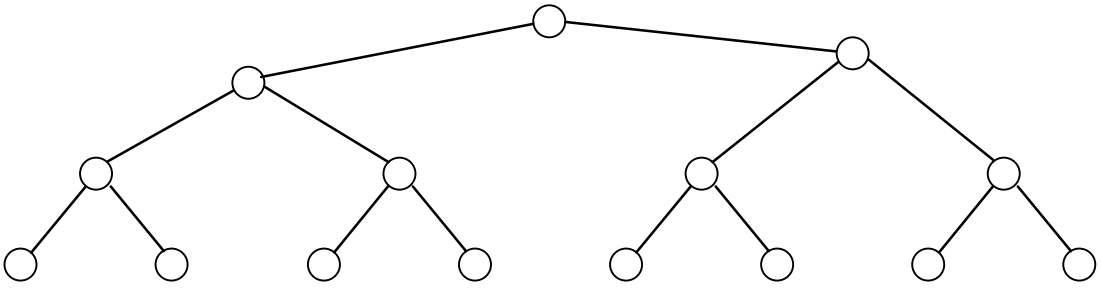

Figura 9. Arboles binarios.

Estas organizaciones son particularmente útiles en problemas de ordenamiento, multiplicación de matrices, y algunos problemas en los que su tiempo solución crece exponencialmente con el tamaño del problema (NP-complejos). El esquema básico de solución consiste en técnicas de

división-y-recolección[8].

3.5. PIRAMIDES

Estas redes intentan combinar las ventajas de las mallas y los arboles. Notese que se ha incrementado la tolerancia a fallas y el número de vias de comunicación sustancialmente.

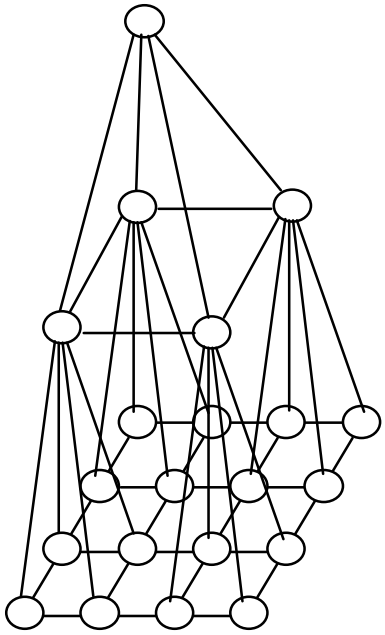

Figura 10. Piramide.

3.6. HIPERCUBO

Un hipercubo puede ser considerado como una malla con conexiones largas adicionales, las cuales reducen el diámetro e incrementan el ancho de bisección. Un hipercubo puede ser definido recursivamente como sigue. Un hipercubo de dimensión-cero es un único procesador (Figura 11) y un hipercubo de dimensión-uno conecta dos hipercubos de dimensión-cero. En general, un hipercubo de dimensión d+1 con 2^{d+1} nodos, se construye conectando los procesadores respectivos de dos hipercubos de dimensión d. Está organización está presente en máquinas

[8] El problema es dividido y cada parte es resulta independientemente. Después se recolectan las soluciones parciales y se ensambla la solución del problema. La división puede ser recursiva.

construidas por nCUBE y los grupos de elementos de procesamiento del arreglo de procesadores Connection Machine CM-200 están conectados por un hipercubo.

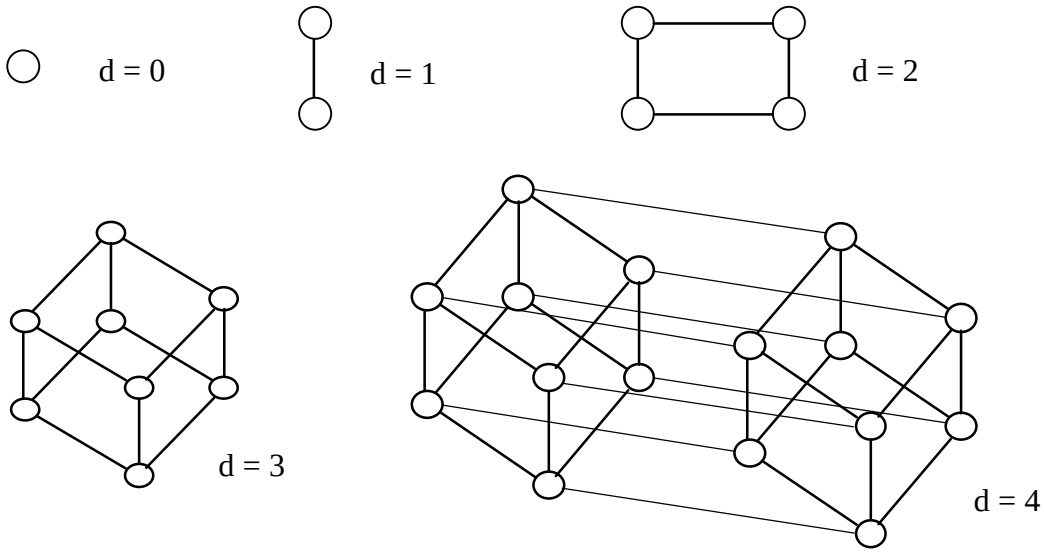

Figura 11. Hipercubos de dimensión 0 a 4.

3.7. OMEGA

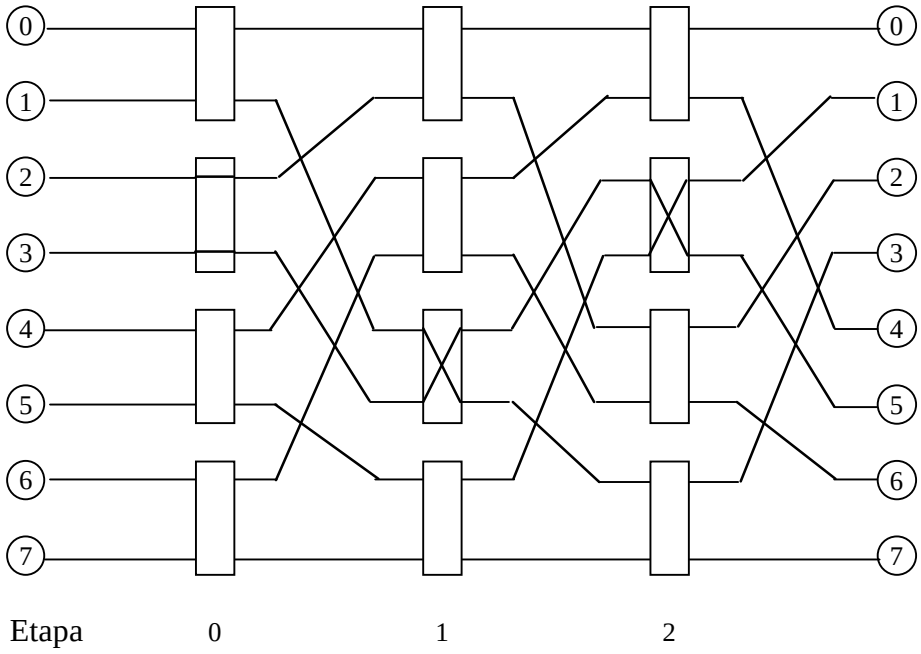

Figura 12. Red omega de 3 etapas que conecta 8 procesadores.

La Figura 12 muestra una red omega de *3 etapas* que conecta 8 procesadores). La red está formada por *crossbar switches 2x2*. Los *switches* tiene cuatro estados posibles: recto, cruzado, *broadcast* superior y *broadcast* inferior; que pueden ser configurados dependiendo de la conexión que se desea (Figura 13). Debido a esto, estas topologías se llaman dinámicas o reconfigurables. Los *switches* son unidireccionales y debemos ver la red plegada en donde los procesadores de la izquierda y la derecha son los mismos. Estas redes reducen considerablemente la competencia por ancho de banda, pero son altamente no escalables y costosas. El *high-performance-switch* de la SP2 es una red omega.

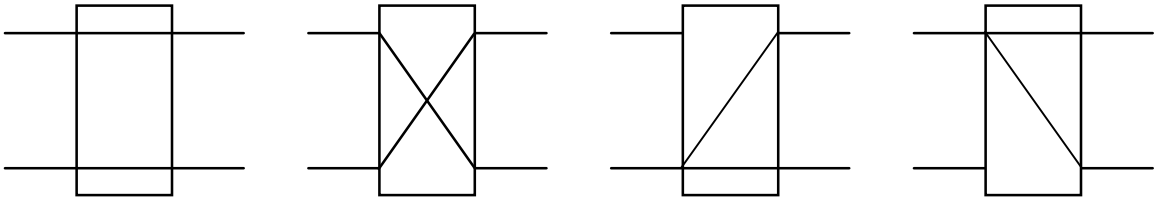

Figura 13. Configuraciones posibles de los *crossbar switches 2x2*.

3.8. FIBRAS DE INTERCONEXION

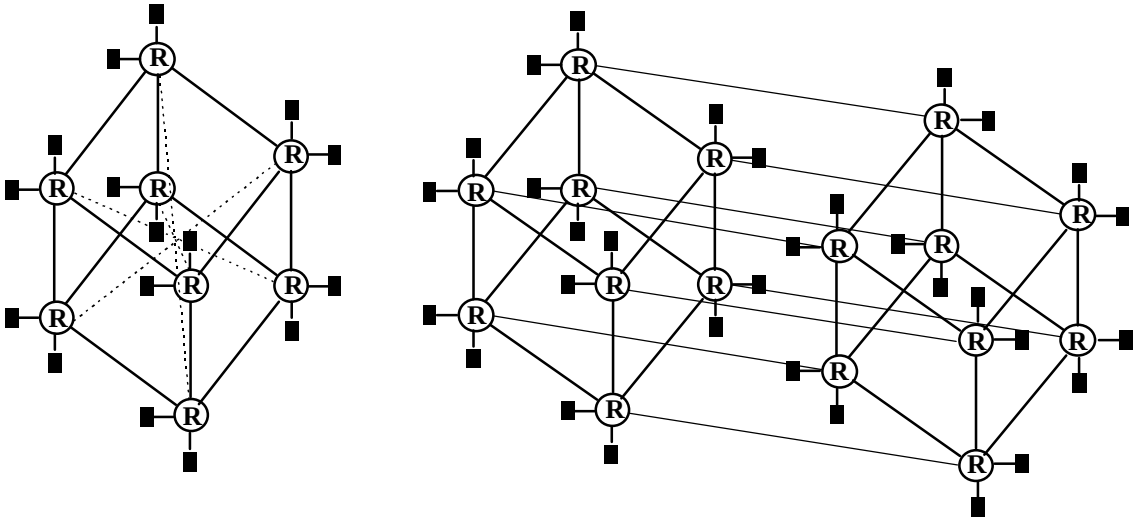

Figura 14. Fibras de interconexión.

La fibra de interconexión es un conjunto de *switches*, llamados *routers*, que están enlazados por distintas configuraciones o topologías. En la Figura 14 se muestran dos fibras de interconexión: una con 8 *routers* y la otra con 16 *routers*. Los procesadores están conectados a los *routers*, los cuales se reconfiguran de acuerdo a la interconexión deseada. Los procesadores de la SGI Origin 2000 están conectados usando una fibra de interconexión en donde los *routers* están compuestos por *crossbar switches 6x6*.

3.9. RESUMEN SOBRE LA ORGANIZACION DE PROCESADORES

A continuación se presenta una tabla que resume las características de las distintas organizaciones vistas. Nótese que sólo las mallas tienen la longitud de los enlaces constante y que en el hipercubo el número de enlaces es una función creciente del tamaño de la red.

Organización	Nodos	Diámetro	Ancho de bisección	Número de enlaces constante	Longitud de enlaces constante	¿Dinámica?
Bus o Ethernet	k	1	1	Si	No	No
Malla Dimensión 1	k	$k-1$	1	Si	Si	No
Malla Dimensión 2	k^2	$2(k-1)$	k	Si	Si	No
Malla Dimensión 3	k^3	$3(k-1)$	k^2	Si	Si	No
Mariposa	$(k+1)2^k$	$2k$	2^k	Si	No	No
Arboles Binarios	2^k-1	$2(k-1)$	1	Si	No	No
Piramide	$(4k^2-1)/3$	$2\log k$	$2k$	Si	No	No
Hipercubo	2^k	k	2^{k-1}	No	No	No
Omega	2^k	①	②	Si	No	Si
Fibra de Interconexión	2^{k+1}	$< k$	$> 2^{k-1}$	③	No	Si

① Se podría decir que este valor es k, sin embargo la interconexión es a través de *switches* y no de otros procesadores.
② No se pueden quitar enlaces y dividir la red en dos en el sentido original de la definición.
③ El número de enlaces por procesador es constante, pero no el número de enlaces por *router*.

Tabla 1. Comparación entre los distintos tipos de organizaciones de procesadores.

4. MODELOS DE COMPUTACION PARALELA

4.1. ARREGLOS DE PROCESADORES

Un **computador vectorial** es una máquina cuyo conjunto de instrucciones permite operaciones tanto sobre vectores como escalares.

4.1.1. Procesador Vectorial *Pipelined*

En estas máquinas los vectores fluyen a través de las unidades aritméticas *pipelined* (Figura 15). Las unidades consisten de una cascada de etapas de procesamiento compuestas de circuitos que efectúan operaciones aritméticas o lógicas sobre el flujo de datos que pasan a través de ellas. La etapas están separadas por registros de alta velocidad usados para guardar resultados intermedios. La información que fluye entre las etapas adyacentes esta bajo el control de un reloj *R* que se aplica a todos los registros simultáneamente. La Cray-1 y la Cyber-205 son máquinas en esta categoría.

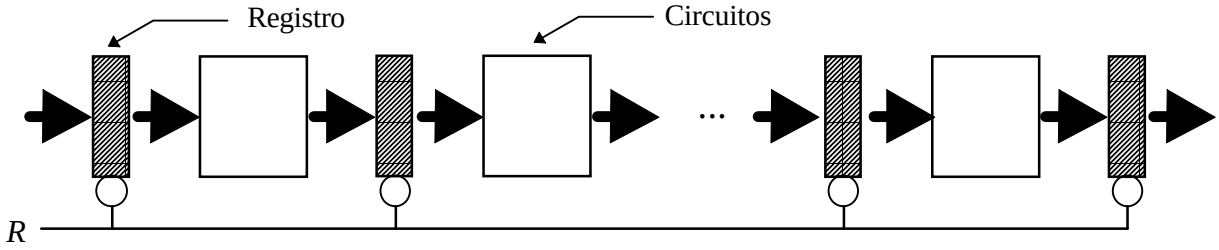

Figura 15. Unidad aritmética *pipelined*.

Para entender mejor este concepto, a continuación se ilustra la construcción de una unidad de multiplicación *pipelined*. Considere la siguiente multiplicación binaria:

```
      10111011     multiplicando
    x  1101001     multiplicador
      10111011
     00000000
    00000000
   10111011
  00000000
 10111011
+ 10111011       .
 100110010110011
```

Figura 16. Multiplicación binaria.

Esta multiplicación se puede descomponer en una serie de sumas y desplazamientos como se muestra a continuación. Los puntos representan ceros y las negritas son los valores que aparecen en el producto final.

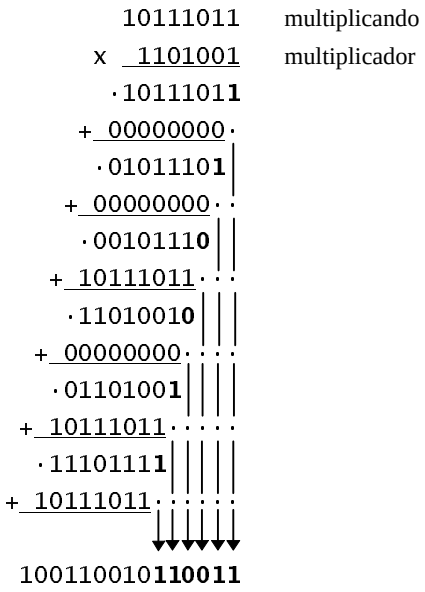

Figura 17. Multiplicación binaria descompuesta como una serie de sumas y desplazamientos.

Una vez descompuesta la multiplicación en etapas, se puede implementar el *pipeline* usando sumadores de 8-bits y compuertas AND. La Figura 18 muestra las primeras 2 etapas (y parte de la tercera) de una unidad de multiplicación *pipelined*. Los registros de una etapa almacenan los resultados de la etapa previa; además todos los valores requeridos en cada etapa son almacenados en registros. Un reloj sincroniza las etapas indicándole a los registros cuando deben leer los resultados de una etapa y hacerlos disponibles a la siguiente etapa. La diferencia entre una unidad *pipelined* y una unidad ordinaria son básicamente los registros y el reloj, lo que permite que vayan efectuando simultáneamente varias operaciones. El tiempo que tarda la unidad en producir el primer resultado es el **tiempo de travesía**, mientras el tiempo que le toma en producir el próximo resultado es **tiempo del ciclo del reloj**.

Pueden haber diferentes formas de descomponer una misma operación y la **granularidad** (del *pipeline*) se refiere a que tan tosca sea esta descomposición. La **variabilidad** se refiere al número de formas en que el mismo *pipeline* se puede configurar para operaciones diferentes.

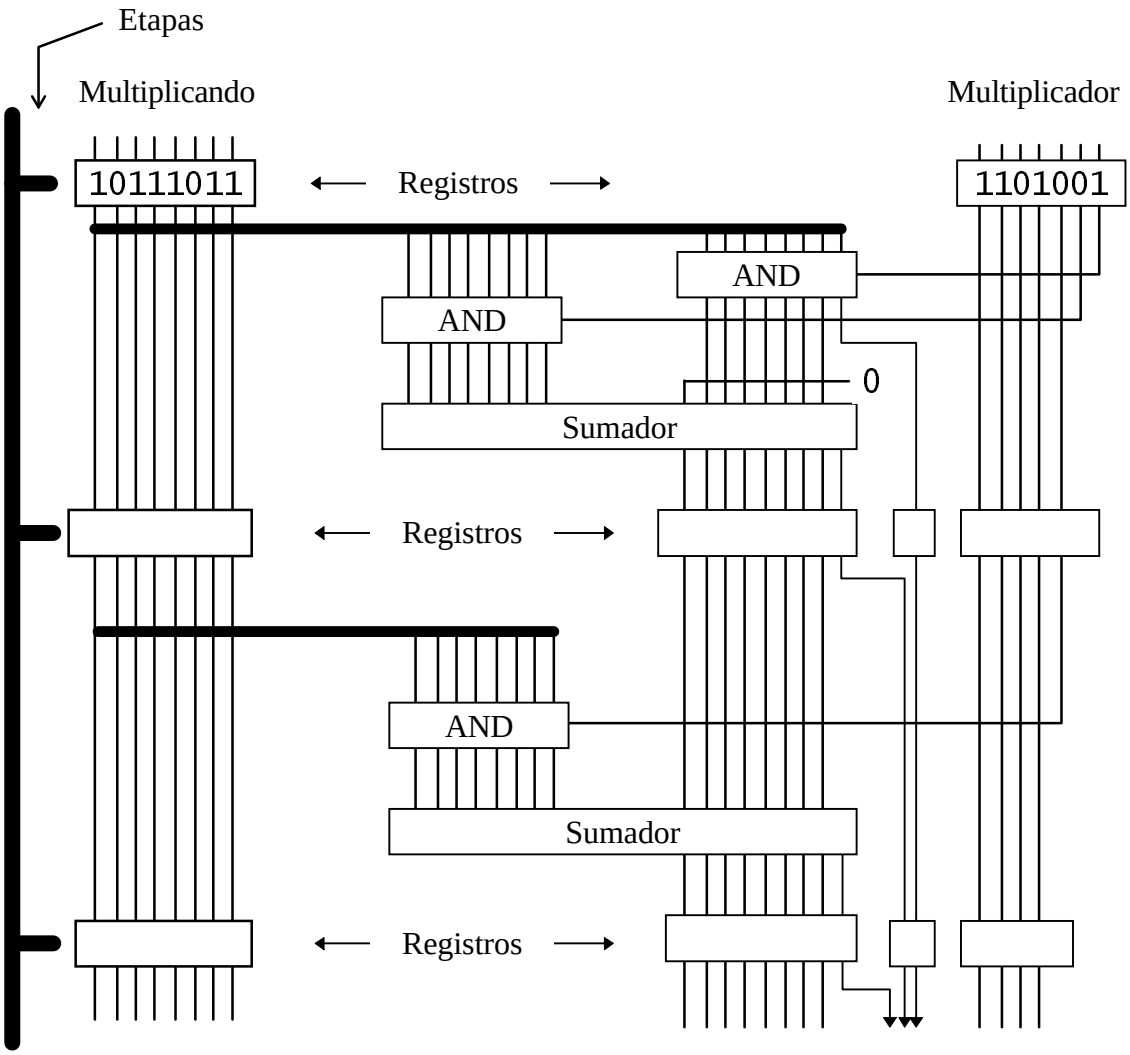

Figura 18. Las primeras dos etapas de una unidad de multiplicación *pipelined*.

4.1.2. Arreglos de Procesadores

Estas son máquinas que constan de un computador secuencial conectado a un arreglo de elementos de procesamiento sincronizados e idénticos capaces de ejecutar las mismas operaciones sobre datos diferentes. (Figura 19) El computador secuencial generalmente es un CPU de propósito general que almacena el programa y los datos que serán operados en paralelo, además de ejecutar la porción del programa que es secuencial. Los elementos de procesamiento se asemejan a CPUs pero no tienen unidades de control propias; el computador secuencial genera todas las señales de control para la unidades de procesamiento en el computador.

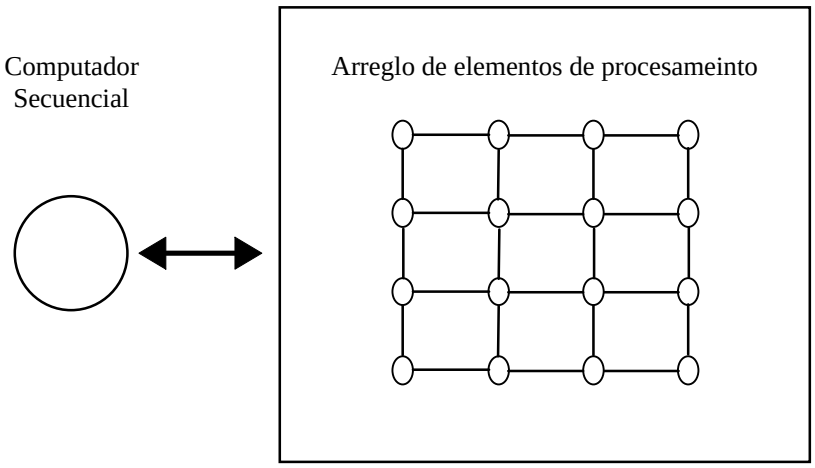

Figura 19. Arreglo de procesadores (los elementos de procesamiento están interconectados por una malla bidimensional).

Los arreglos de procesamiento difieren fundamentalmente en la complejidad y la topología de interconexión de sus elementos de procesamiento. Ejemplos de estas máquinas son: IILIAC IV, Goodyear MPP y Connection Machine CM-200.

4.2. MULTIPROCESADORES

Estos equipos están formados por un número de procesadores completamente programables capaces de ejecutar su propio programa.

4.2.1. Multiprocesadores de Acceso Uniforme a Memoria (UMA: Uniform Memory Access)

Estos computadores tienen todos sus procesadores interconectados a través de un mecanismo de *switches* a una memoria compartida centralizada (Figura 20). Entre estos mecanismos estan: un bus común, *crossbar switches*, *packet-switched networks*[9].

Sistemas que usan buses son de tamaño limitado ya que después de cierto número de procesadores el bus se satura. En el caso de *crossbar switches*, el costo del *switch* se convierte en el factor dominante y limita el número de procesadores.

[9] En las *packet-switched networks*, cada mensaje contiene una etiqueta con información de enrutamiento y cada elemento del *switch* interpreta esta información seleccionando el puerto de salida apropiado. A diferencia, las *circuit-switched networks* establecen una conexión punto-a-punto, reservando toda la red por la duración del mensaje.

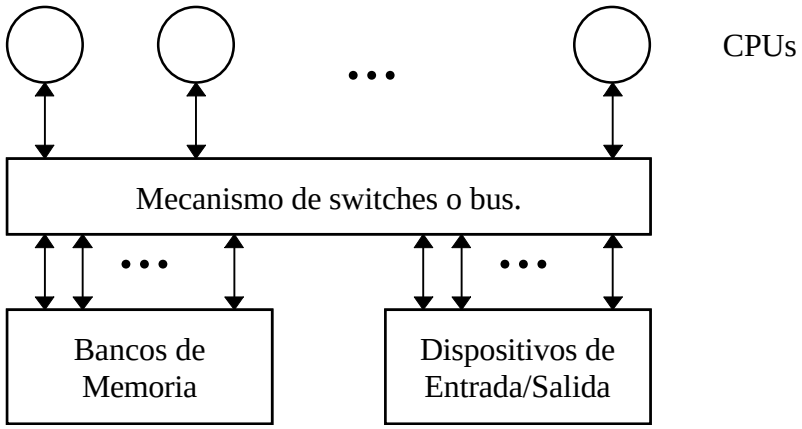

Figura 20. Multiprocesadores de Acceso Uniforme a Memoria.

Las máquinas Encore Multimax y Sequent Symetry S81 son ejemplos comerciales de este tipo de multiprocesadores.

4.2.2. Multiprocesadores de Acceso No-Uniforme a Memoria (NUMA: Non-Uniform Memory Access)

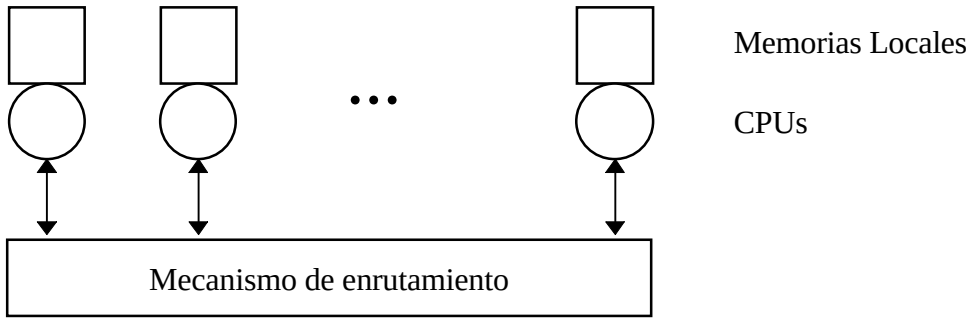

Figura 21. Multiprocesadores de Acceso No-Uniforme a Memoria.

En estos multiprocesadores el espacio de direccionamiento es compartido a pesar de que la memoria es distribuida (Figura 21). Cada procesador tiene cierta memoria local que combinadas forman la memoria compartida. El tiempo de acceso a memoria depende de si el acceso es local al procesador o no. La BBN TC2000 y la SGI Origin 2000 son ejemplos de este modelo de computación paralela.

El principal problema que presentan los multiprocesadores (máquinas de memoria compartida) radica en que no se pueden agregar procesadores indefinidamente ya que a partir de cierto

número y dependiendo de la aplicación, el mecanismo de *switches* o enrutamiento se satura (congestiona), en otras palabras, tienen poca extensibilidad.

4.3. MULTICOMPUTADORES

En este modelo, los procesadores no comparten memoria. Cada procesador tiene su propia memoria privada (máquinas de memoria distribuida) y la interacción entre ellos es a través de pase de mensajes. Ejemplos son: Intel ParagonXP/S, Meikos Computing Surface, nCUBE 2, Parsytec SuperCluster, Thinking Machine CM-5 y la IBM SP2.

4.4. MAQUINAS DE MEMORIA COMPARTIDA DISTRIBUIDA

La tendencia actual en las máquinas paralelas es de aprovechar las facilidades de programación que ofrecen los ambientes de memoria compartida y la escalabilidad de los ambientes de memoria distribuida. En este modelo se conectan entre si módulos de multiprocesadores, pero se mantiene la visión global de la memoria a pesar de que se esta distribuyendo (Figura 22). Este tipo de máquinas en realidad caen dentro de la clasificación NUMA, pero dada su importancia actual las consideramos separadamente. La SGI Origin 2000 es un ejemplo de este modelo, y la SP2 esta tendiendo a él.

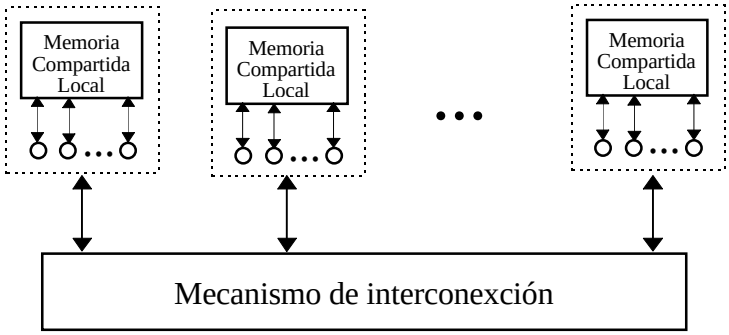

Figura 22. Máquinas de memoria compartida distribuida.

4.5. MULTIPROCESADORES MULTI-HEBRADOS

En estas máquinas cada procesador tiene cierto número de flujos de instrucciones implementados en *hardware*, incluyendo el contador de programa y registros, cada uno destinado a ejecutar una hebra. En cada ciclo el procesador ejecuta instrucciones de una de la hebras. En el ciclo siguiente, el procesador hace un cambio de contexto y ejecuta instrucciones de otra hebra.

La Tera MTA (multithreaded architecture), es la primera de estas máquinas y esta siendo probada en el Centro de Supercomputación de San Diego. Cada procesador tiene 128 flujos de instrucciones. Dado que un acceso a memoria dura aproximadamente 100 ciclos, en la próxima ejecución de la hebra se tendrán los datos requeridos. Este es el mecanismo que le permite a la MTA tolerar la latencia a memoria y por lo tanto no requiere de memorias cache. Cada instrucción de 64-bits codifica 3 operaciones: una de memoria y dos que pueden ser aritméticas o de control. El sistema operativo es una versión distribuida paralela completamente simétrica de UNIX.El sistema tiene de 1 a 256 procesadores que comparten una enorme memoria. Cada procesador tiene 1 o 2 Gb de memoria, pero un mapeo aleatorizado de la memoria y una red altamente interconectada proveen acceso casi uniforme de cualquier procesador a cualquier memoria.

La pruebas realizadas para el momento muestran que el costo del multi-hebrado es pequeño, su rendimiento es comparable con el de la T90 y los códigos de la MTA son significativamente más fáciles de optimizar que en máquinas masivamente paralelas o estaciones de trabajo de alto rendimiento.

TERA afirma que el impacto en el mercado de la MTA será más profundo que el de la Cray 1 y que este tipo de computadores son necesarios antes de que nuevos crecimientos reales se puedan dar en la computación de alto rendimiento.

4.6. CLUSTERS DE PC'S

El desarrollo de sistemas operativos y compiladores del dominio publico (Linux y GNU software), estándares para el pase de mensajes (MPI) y conexiones universales a periféricos (PCI), han hecho posible tomar ventaja de los económicos recursos computacionales de producción masiva (CPU's, discos, redes).

La principal desventaja que presentan los proveedores de multiprocesadores es que deben satisfacer una amplia gama de usuarios, es decir, deben ser generales. Esto aumenta los costos de diseño y producción de equipos, así como los costos de desarrollo del *software* que va con ellos: sistema operativo, compiladores y aplicaciones. Todos estos costos deben ser añadidos cuando se hace una venta. Por supuesto, alguien que solo necesita procesadores y un mecanismo de pase de mensajes no debería pagar por todos estos añadidos que nunca usara. Estos usuarios son los que están impulsando el uso de clustres de PC's.

4.6.1. Ventajas del usos de clusters de PC's para procesamiento paralelo:

- La reciente explosión en redes implica que la mayoría de los componentes necesarios para

construir un cluster son vendidos en altos volúmenes y por lo tanto son económicos. Ahorros adicionales se pueden obtener debido a que solo se necesitaría una tarjeta de vídeo, un monitor y un teclado por cluster. El mercado de los multiprocesadores es más reducido y más costosos.

- Remplazar un componente defectuoso en un cluster es relativamente trivial comparado con hacerlo en un multiprocesador, permitiendo una mayor disponibilidad en clusters cuidadosamente diseñados.

4.6.2. Desventajas del usos de clusters de PC's para procesamiento paralelo:

- Con raras excepciones, los equipos de redes generales producidos masivamente no están diseñados para procesamiento paralelo y típicamente su latencia es alta y los anchos de banda pequeños comparados con multiprocesadores. Dado que los clusters explotan tecnología que sea económica, los enlaces en el sistema no son veloces implicando que la comunicación entre componentes debe pasar por un proceso de protocolos de negociación lentos, incrementando seriamente la latencia, y recurrir en muchos y en el mejor de los casos (debido a costos) a Fast Ethernet restringiendo la escalabilidad del cluster.
- Hay poco soporte de software para manejar un cluster como un sistema integrado. La administración, por lo general, se debe hacer máquina por máquina y no es centralizada como en los multiprocesadores. Esto también los hace más susceptibles a problemas de seguridad.
- Los procesadores para los PC's no son tan eficientes como los procesadores de los multiprocesadores para manejar múltiples usuarios y/o procesos. Esto hace que el rendimiento de los clusters se degrade con relativamente pocos usuarios y/o procesos.
- Muchas aplicaciones importantes disponibles en multiprocesadores y optimazadas para ciertas arquitecturas, no lo estan en clusters de PC's.

Sin lugar a duda los clusters presentan una alternativa importante para varios problemas particulares, no solo por su economía, sino también porque pueden ser diseñados y ajustados para ciertas aplicaciones. La aplicaciones que pueden sacar provecho de estos clusters estan actualmente limitadas a las que se pueden descomponer en componentes altamente independientes o que requieren poca comunicación entre ellos. Los clusters de PC's no son propicios para aplicaciones en donde el grado de comunicación entre los componetes es de mediano a alto o descompuestas en etapas dependientes unas de las otras. Por los momentos tampoco no son adecuados para prestar servicios de computación de alto rendimiento a un gama variada de usuarios.

4.7. TAXONOMIA DE FLYNN

La taxonomía de Flynn es otra manera de clasificar computadores seriales y paralelos y es una de las más conocidas. Se basa en la multiplicidad del flujo de instrucciones y del flujo de datos en un computador. El flujo de instrucciones es la secuencia de instrucciones ejecutadas por el computador y el flujo de datos es la secuencia de datos sobre los cuales operan las instrucciones. La clasificacion de Flynn incluye las siguientes categorías:

4.7.1. Single instruction stream, single data stream (SISD)

Computadores SISD representan la mayoría de los máquinas seriales. Tienen un CPU que ejecuta una instruccion en un momento dado y busca o guarda un dato en un momento dado (Figura 23).

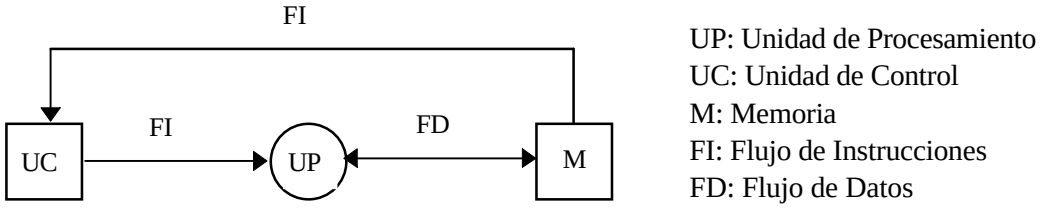

Figura 23. Single instruction stream, single data stream.

4.7.2. Single instruction stream, multiple data stream (SIMD)

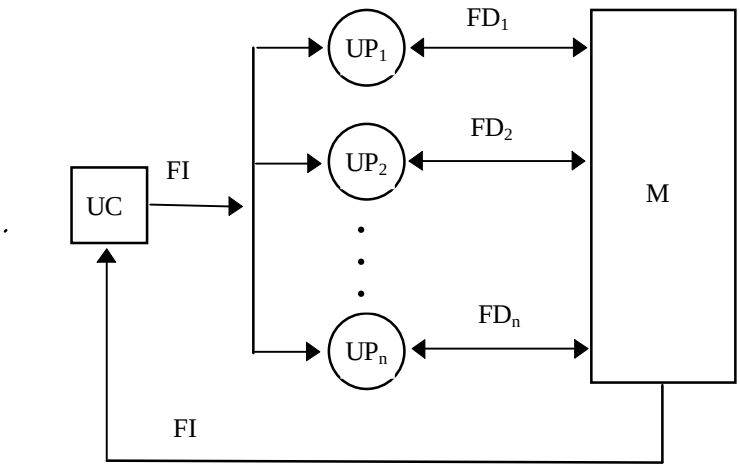

Figura 24. Single instruction stream, multiple data stream.

Los arreglos de procesadores están en esta categoría. Hay un conjunto de unidades de procesamiento cada una ejecutando la misma operación sobre datos distintos (Figura 24).

4.7.3. Multiple instruction stream, single data stream (MISD)

Esta categorización es poco intuitiva. Hay *n* procesadores cada uno recibiendo una instrucción diferente y operando sobre el mismo flujo de datos. En la Figura 25 hay dos formas de ver el flujo de datos. En el FD^1 el flujo va pasando de UP a UP (esto seria como un *pipeline*) y en el FD^2 cada UP recibe una copia del FD. Esta estructura ha sido catalogada como impráctica por algunos arquitectos de computadoras y actualmente no hay máquinas de este tipo, a pesar de que ciertas MIMD puedan ser usadas de esta forma.

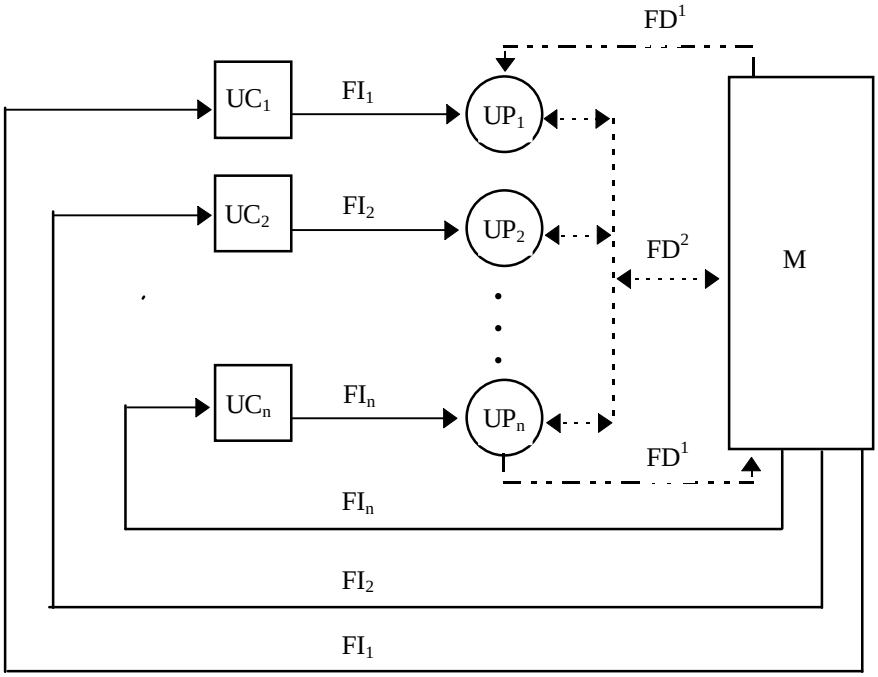

Figura 25. Multiple instruction stream, single data stream.

4.7.4. Multiple instruction stream, multiple data stream (MIMD)

La mayoria de los multiprocesadores y multicomputadores pueden ser clasificados bajo esta categoría. Tienen más de un procesador independiente y cada uno puede ejecutar un programa diferente sobre sus propios datos (Figura 26). Podemos hacer otra subdivisión de los

multiprocesadores dependiendo de como esté organizada su memoria: memoria compartida o memoria distribuida.

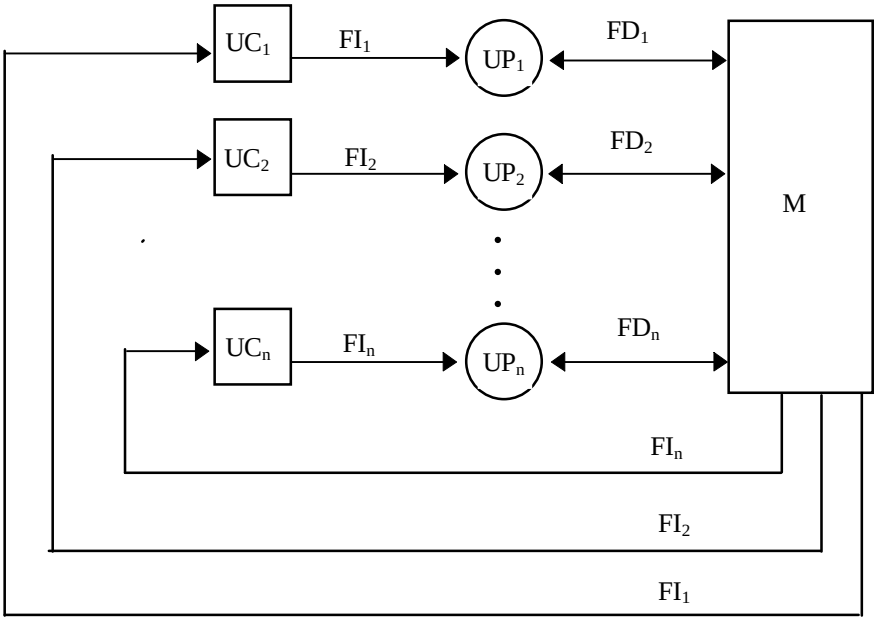

Figura 26. Multiple instruction stream, multiple data stream.

4.7.4.1. Single program, multiple data (SPMD)

Esta categoría no fue definida por Flynn, sin embargo es muy usada actualmente y pueder ser considerada como un caso particular de MIMD. Este modelo se dá cuando cada procesador ejecuta una copia exacta del mismo programa, pero opera sobre datos diferentes.

4.8. EL IBM SCALABLE POWER PARALLEL SYSTEM (SP2)

Dentro de los modelos vistos y la taxonomía de Flynn, esta máquina es básicamente un multicomputador (memoria distribuida) MIMD. Cada nodo puede ser usado como un computador independiente, pero también pueden cooperar en la solución del mismo problema, es decir, ser usados en paralelo. La Figura 27 muestra una IBM SP2.

La SP2 puede tener tres tipos de nodos: finos, anchos y altos. Los anchos generalmente se configuran como servidores para proveer servicios necesarios para la ejecución de tareas y permiten conectar directamente dispositivos de almacenamiento a la SP2. Los nodos finos son preferibles para ejecutar tareas. Los nodos altos constan de hasta 8 procesadores que comparten

la misma memoria. Cada módulo de una SP2 puede tener hasta 16 nodos finos, u 8 anchos o 4 altos. Los nodos altos son los que aproximan la SP2 al modelo de memoria compartida distribuida; la memoria de un nodo alto es compartida entre sus procesadores, pero entre nodos altos es distribuida y deben comunicarse mediante pase de mensajes. Una misma máquina puede estar compuesta por distintos tipos de nodos.

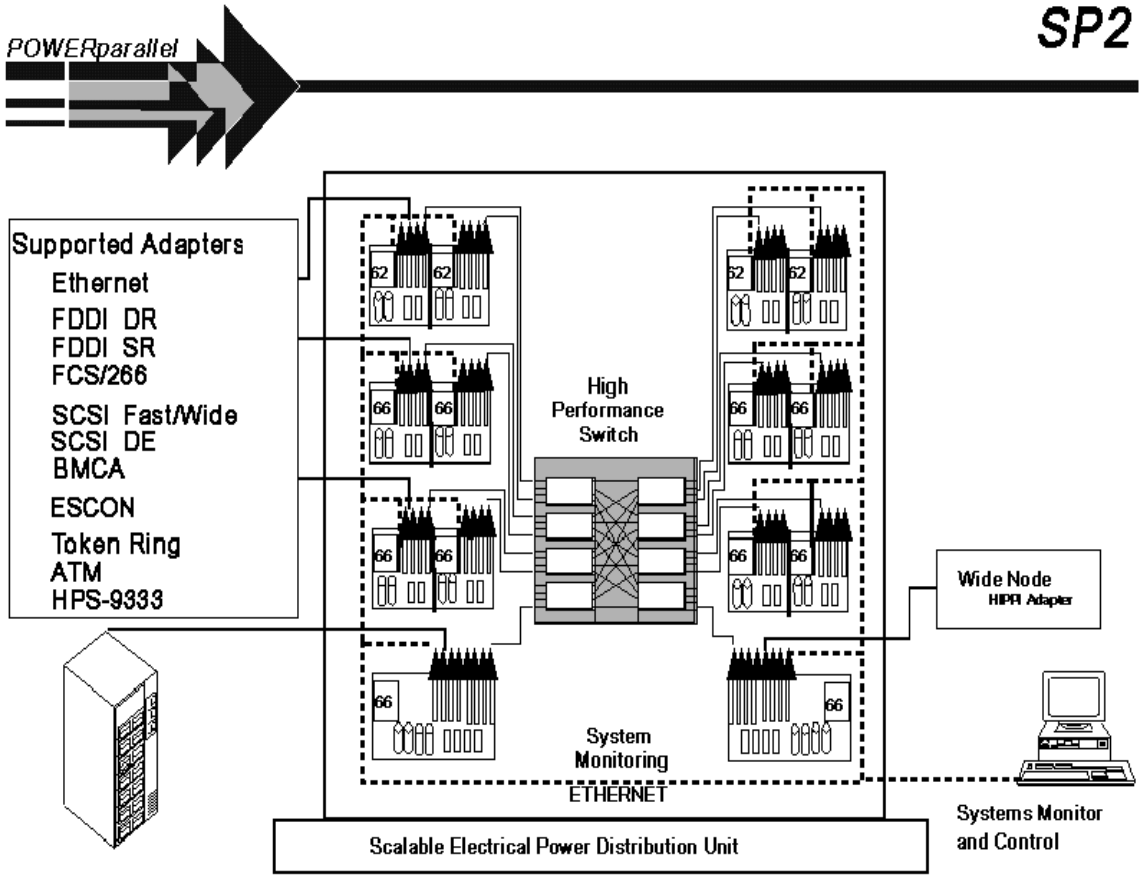

Figura 27. Una IBM SP2 con dos nodos anchos y doce nodos finos.

Un *switch* de alta eficiencia (*high-performance-switch*) conecta los procesadores para proveer altas velocidades de comunicación. En particular consiste de una red omega, de mutiples etapas, *buffers* y un mecanismo de enrutamiento *packet-switched*. Los *buffers* permiten el almacenamiento temporal de paquetes entre etapas en caso de que la etapa siguiente esté ocupada. En los protocolos *packet-switched*, cada mensaje contiene información de enrutamiento que es interpretada por cada elemento del *switch*.

La SP2 esta basada en el sistema operativo AIX/6000 y en la tecnología RISC System/6000 POWER2, lo que implica que miles de aplicaciones bajo RS/6000 correren sin cambios en la SP2.

4.9. LA SGI ORIGIN 2000

La SGI Origin 2000 es un MIMD con memoria distribuida compartida..Cada nodo tiene uno o dos procesadores R10000 y la memoria esta físicamente dispersa a través del sistema. Hardware para la migración de paginas mueve los datos que son frecuentemente usados a la memoria más cercana al procesador para reducir latencia de memoria. A pesar de que la memoria esta distribuida, esta es universalmente accesible y compartida entre todos los procesadores. Similarmente, los dispositivos de I/O están distribuidos entre los nodos, pero cada uno esta disponible a todos los procesadores.

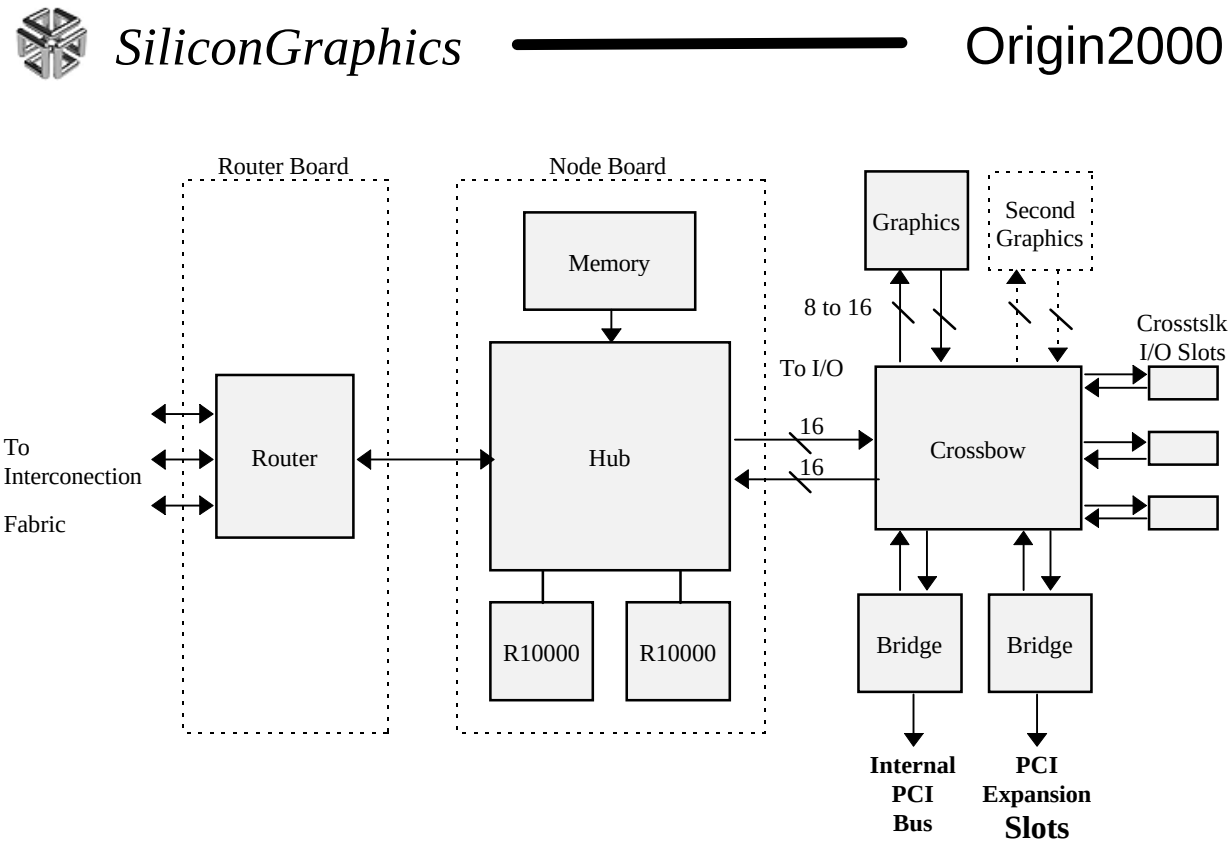

Figura 28. Un nodo de la SGI Origin 2000.

La Figura 28 muestra un bloque de la Origin2000. El *Hub* es el controlador de la memoria distribuida compartida y esta encargado de que todos los procesadores y los dispositivos de I/O tengan acceso transparente a toda la memoria distribuida compartida. El *Crossbow* es un *crossbar* chip encargado de conectar dos nodos con hasta seis controladores de I/O. La fibra de interconexión es una malla de múltiples enlaces punto-a-punto conectados por los enrutadores y provee a cada par de nodos con un mínimo de dos trayectorias distintas. Esta redundancia le permite al sistema eludir enrutadores o enlaces de la fibra que estén fallando. La Origin 2000 puede tener hasta 128 procesadores, 4GB de memoria por nodo (256GB en total) y 64 interfaces de I/O con 192 controladores de I/O. A medida que se agregan nodos a la fibra de interconexión, el ancho de banda y la eficiencia escalan linealmente sin afectar significativamente las latencias del sistema. Esto es debido a que se reemplazo el tradicional bus por la fibra de interconexión, y porque la memoria centralizada fue reemplazada por la memoria compartida distribuida pero que esta fuertemente integrada

5. DISEÑO DE ALGORITMOS PARALELOS

Diseñar algoritmos paralelos no es tarea fácil y es un proceso altamente creativo. Inicialmente se deben explorar los aspectos independientes de la máquina y los específicos a la máquina deben ser dejados para más tarde. El diseño involucra cuatro etapas las cuales se presentan como secuenciales pero que en la práctica no lo son (Figura 29).

1) **Partición**: El cómputo y los datos sobre los cuales se opera se descomponen en tareas. Se ignoran aspectos como el número de procesadores de la máquina a usar y se concentra la atención en explotar oportunidades de paralelismo.
2) **Comunicación**: Se determina la comunicación requerida para coordinar las tareas. Se definen estructuras y algoritmos de comunicación.
3) **Agrupación**: El resultado de las dos etapas anteriores es evaluado en términos de eficiencia y costos de implementación. De ser necesario, se agrupan tareas pequeñas en tareas más grandes.
4) **Asignación**: Cada tarea es asignada a un procesador tratando de maximizar la utilización de los procesadores y de reducir el costo de comunicación. La asignación puede ser estática (se establece antes de la ejecución del programa) o a tiempo de ejecución mediante algoritmos de balanceo de carga.

5.1. PARTICION

En la etapa de partición se buscan oportunidades de paralelismo y se trata de subdividir el problema lo más finamente posible, es decir; que la **granuralidad** sea **fina**[10]**.** Evaluaciones futuras podrán llevar a aglomerar tareas y descartar ciertas posibilidades de paralelismo. Una buena partición divide tanto los cómputos como los datos. Hay dos formas de proceder con la descomposición.

Descomposición del dominio: el centro de atención son los datos. Se determina la partición apropiada de los datos y luego se trabaja en los cómputos asociados con los datos.

Descomposición funcional: es el enfoque alternativo al anterior. Primero se descomponen los cómputos y luego se ocupa de los datos.

Estas técnicas son complementarias y pueden ser aplicadas a diferentes componentes de un problema e inclusive al mismo problema para obtener algoritmos paralelos alternativos.

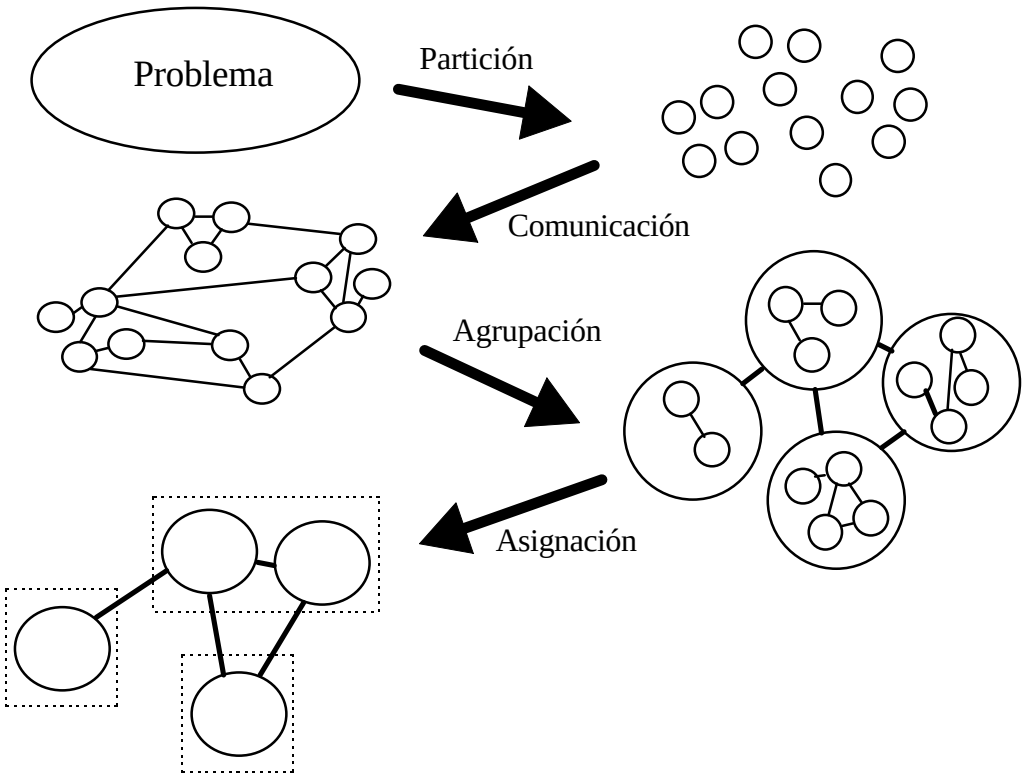

Figura 29. Etapas en el diseño de algoritmos paralelos.

[10] Un grano es una medidad del trabajo computacional a realizar.

Al particionar se deben tener en cuenta los siguientes aspectos:

1) El número de tareas debe ser por lo menos un orden de magnitud superior al número de procesadores disponibles para tener flexibilidad en las etapas siguientes.
2) Hay que evitar cómputos y almacenamientos redundantes; de lo contrario el algoritmo puede ser no extensible a problemas más grandes.
3) Hay que tratar de que las tareas sean de tamaños equivalentes ya que facilita el balanceo de la carga de los procesadores.
4) El número de tareas debe ser proporcional al tamaño del problema. De esta forma el algoritmo será capaz de resolver problemas más grandes cuando se tenga más disponibilidad de procesadores. En otras palabras, se debe tratar de que el algoritmo sea **escalable**; que el grado de paralelismo aumente al menos linealmente con el tamaño del problema.
5) Considere alternativas de paralelismo en esta etapa ya que pueden flexibilizar etapas subsecuentes.

5.2. COMUNICACION

Las tareas definidas en la etapa anterior, en general, pueden correr concurrentemente pero no independientemente. Datos deben ser transferidos o compartidos entre tareas y esto es lo que se denomina la fase de comunicación.

La comunicación requerida por un algoritmo puede ser definida en dos fases. Primero se definen los canales que conectan las tareas que requieren datos con las que los poseen. Segundo se especifica la información o mensajes que deben ser enviado y recibidos en estos canales.

Dependiendo del tipo de máquina en que se implementará el algoritmo, memoria distribuida o memoria compartida, la forma de atacar la comunicación entre tareas varía.

En ambientes de memoria distribuida, cada tarea tiene una identificación única y las tareas interactuan enviando y recibiendo mensajes hacia y desde tareas específicas. Las librerías más conocidas para implementar el pase de mensajes en ambientes de memoria distribuida son: MPI (*Message Passing Interface*) y PVM (*Parallel Virtual Machine*).

En ambientes de memoria compartida no existe la noción de pertenencia y el envío de datos no se dá como tal. Todas las tareas comparten la misma memoria. *Semáforos, semáforos binarios, barreras* y otros mecanismos de sincronizacion son usados para controlar el acceso a la memoria compartida y coordinar las tareas.

Un **semáforo** es un tipo de variables que puede tomar valores enteros ≥ cero[11]. El valor del semáforo es cambiado mediante dos operaciones: **UP** (incrementa) y **DOWN** (decrementa). La operación DOWN cheque si el valor es mayor que 0, y de ser así, decrementa el semáforo en uno y continua. Si el valor es 0, el proceso es bloqueado. Chequear el valor del semáforo, cambiarlo y posiblemente ser bloqueado es hecho en una **acción atómica** única e indivisible. Se garantiza que una vez que una operación haya comenzado, ningún otro proceso puede accesar el semáforo hasta que la operación haya sido completada. Esta atomicidad es esencial para lograr la sincronización entre procesos. La operación UP incrementa en uno el semáforo y en caso de haber procesos bloqueados por no haber podido completar un DOWN en ese semáforo, uno de ellos es seleccionado por el sistema y se le permite completar el DOWN. Nunca un proceso es bloqueado efectuando una operación UP.

En esta etapa hay que tener en cuenta los siguientes aspectos:

1) Todas las tareas deben efectuar aproximadamente el mismo número de operaciones de comunicación. Si esto no se da, es muy probable que el algoritmo no sea extensible a problemas mayores ya que habrán cuellos de botella.
2) La comunicación entre tareas debe ser tan pequeña como sea posible.
3) Las operaciones de comunicación deben poder proceder concurrentemente.
4) Los cómputos de diferentes tareas deben poder proceder concurrentemente.

5.3. AGRUPACION

En las dos etapas anteriores se particionó el problema y se determinaron los requerimientos de comunicación. El algoritmo resultante es aún abstracto en el sentido de que no se tomó en cuenta la máquina sobre el cual correrá. En esta etapa se va de lo abstracto a lo concreto y se revisa el algoritmo obtenido tratando de producir un algoritmo que corra eficientemente sobre cierta clase de computadores. En particular se considera si es útil agrupar tareas y si vale la pena replicar datos y/o cómputos.

En la fase de partición se trató de establecer el mayor número posible de tareas con la intensión de explorar al máximo las oportunidades de paralelismo. Esto no necesariamente produce un algoritmo eficiente ya que el costo de comunicación puede ser significativo. En la mayoría de los computadores paralelos la comunicación es mediante el pase de mensajes y frecuentemente hay que parar los cómputos para enviar o recibir mensajes. Mediante la agrupación de tareas se puede reducir la cantidad de datos a enviar y así reducir el número de

[11] Los semáforos binarios solo toman los valores 0 o 1.

mensajes y el costo de comunicación.

La comunicación no solo depende de la cantidad de información enviada. Cada comunicación tiene un costo fijo de arranque. Reduciendo el número de mensajes, a pesar de que se envíe la misma cantidad de información, puede ser útil. Así mismo se puede intentar replicar cómputos y/o datos para reducir los requerimientos de comunicación. Por otro lado, también se debe considerar el costo de creación de tareas y el costo de cambio de contexto[12] (*context switch*) en caso de que se asignen varias tareas a un mismo procesador.

En caso de tener distintas tareas corriendo en diferentes computadores con memorias privadas, se debe tratar de que la **granularidad** sea **gruesa**: exista una cantidad de computo significativa antes de tener necesidades de comunicación. Esto se aplica a máquinas como la SP2 y redes de estaciones de trabajo. Se puede tener **granularidad media** si la aplicación correrá sobre una máquina de memoria compartida. En estas máquinas el costo de comunicación es menor que en las anteriores siempre y cuando en número de tareas y procesadores se mantenga dentro de cierto rango. A medida que la granularidad decrece y el número de procesadores se incrementa, se intensifican la necesidades de altas velocidades de comunicaciones entre los nodos. Esto hace que los sistemas de grano fino por lo general requieran máquinas de propósito especifico. Se pueden usar máquinas de memoria compartida si el número de tareas es reducido, pero por lo general se requieren máquinas masivamente paralelas conectadas mediante una red de alta velocidad (arreglos de procesadores o máquinas vectoriales).

Los puntos resaltantes que se deben considerar en esta etapa son:

1) Chequear si la agrupación redujo los costos de comunicación.
2) Si se han replicado cómputos y/o datos, se debe verificar que los beneficios son superiores a los costos.
3) Se debe verificar que las tareas resultantes tengan costos de computo y comunicación similares.
4) Hay que revisar si el número de tareas es extensible con el tamaño del problema.
5) Si el agrupamiento ha reducido las oportunidades de ejecución concurrente, se debe verificar que aun hay suficiente concurrencia y posiblemente considerar diseños alternativos.
6) Analizar si es posible reducir aun más el número de tareas sin introducir desbalances de cargas o reducir la extensibilidad.

[12] Cada vez que el procesador cambia de tarea, se deben cargar registros y otros datos de la tarea que correrá.

5.4. ASIGNACION

En esta última etapa se determina en que procesador se ejecutará cada tarea. Este problema no se presenta en máquinas de memoria compartida tipo UMA. Estas proveen asignación dinámica de procesos y los procesos que necesitan de un CPU están en una *cola de procesos listos*. Cada procesador tiene acceso a esta cola y puede correr el próximo proceso. No consideraremos más este caso.

Para el momento no hay mecanismos generales de asignación de tareas para máquinas distribuidas. Esto continúa siendo un problema difícil y que debe ser atacado explícitamente a la hora de diseñar algoritmos paralelos.

La asignación de tareas puede ser estática o dinámica. En la **asignación estática**, las tareas son asignadas a un procesador al comienzo de la ejecución del algoritmo paralelo y corren ahí hasta el final. La asignación estática en ciertos casos puede resultar en un tiempo de ejecución menor respecto a asignaciones dinámicas y también puede reducir el costo de creación de procesos, sincronización y terminación.

En la **asignación dinámica** se hacen cambios en la distribución de las tareas entre los procesadores a tiempo de ejecución, o sea, hay **migración** de tareas a tiempo de ejecución. Esto es con el fin de balancear la carga del sistema y reducir el tiempo de ejecución. Sin embargo, el costo de balanceo puede ser significativo y por ende incrementar el tiempo de ejecución. Entre los algoritmos de balanceo de carga están los siguientes:

Balanceo centralizado: un nodo ejecuta el algoritmo y mantiene el estado global del sistema. Este método no es extensible a problemas más grandes ya que el nodo encargado del balanceo se convierte en un cuello de botella.

Balanceo completamente distribuido: cada procesador mantiene su propia visión del sistema intercambiando información con sus vecinos y así hacer cambios locales. El costo de balanceo se reduce pero no es óptimo debido a que solo se dispone de información parcial.

Balanceo semi-distribuido: divide los procesadores en regiones, cada una con un algoritmo centralizado local. Otro algoritmo balancea la carga entre las regiones.

El balanceo puede ser iniciado por envío o recibimiento. Si es **balanceo iniciado por envío**, un procesador con mucha carga envía trabajo a otros. Si es **balanceo iniciado por recibimiento**, un procesador con poca carga solicita trabajo de otros. Si la carga por procesador es baja o mediana, es mejor el balanceo iniciado por envío. Si la carga es alta se debe usar balanceo iniciado por recibimiento. De lo contrario, en ambos casos, se puede producir una fuerte migración innecesaria de tareas.

Entre los puntos que hay que revisar en esta etapa encontramos:

1) ¿Se han considerado algoritmos con un número estático de tareas y algoritmos de creación dinámica de tareas?
2) Si se usan algoritmos centralizados de balanceo, hay que asegurarse de que no sea un cuello de botella.
3) Hay que evaluar los costos de las diferentes alternativas de balanceo dinámico, en caso de que se usen, y que su costo no sea mayor que los beneficios.

6. ACELERACION Y EFICIENCIA

Dos de las medidas más importantes para apreciar la calidad de la implementación de un algoritmo paralelo en multicomputadores y multiprocesadores son **aceleración** y **eficiencia**. La **aceleración** de un algoritmo paralelo ejecutado usando *p* procesadores es la razón entre el tiempo que tarda el mejor algoritmo secuencial en ser ejecutado usando un procesador en un computador y el tiempo que tarda el correspondiente algoritmo paralelo en ser ejecutado en el mismo computador usando *p* procesadores. La **eficiencia** de un algoritmo paralelo ejecutado en *p* procesadores es la aceleración dividida por *p*. La ley de *Amdahl* permite expresar la aceleración máxima obtenible como una función de la fracción del código del algoritmo que sea paralelizable.

6.1. LEY DE AMDAHL (LIMITE A LA ACELERACION OBTENIBLE)

Sea T el tiempo que tarda el mejor algoritmo secuencial en correr en un computador. Si f es la fracción de operaciones del algoritmo que hay que ejecutar secuencialmente ($0 \leq f \leq 1$), entonces la aceleración máxima obtenible ejecutando el algoritmo en p procesadores es:

$$A_p = \frac{T}{T(f + (1-f)/p)} = \frac{1}{f + (1-f)/p}$$

En el limite, cuando $f \rightarrow 0$, $A = p$. Excepto casos muy particulares, este limite nunca es obtenible debido a las siguientes razones:

- inevitablemente parte del algoritmo es secuencial,
- hay conflictos de datos y memoria,
- la comunicación y sincronización de procesos consume tiempo, y
- es muy difícil lograr un balanceo de carga perfecto entre los procesadores.

En la Tabla 2 se muestra la aceleración máxima obtenible en términos del porcentaje de código paralelizable y en la Figura 30 el gráfico respectivo. Se puede observar que la aceleración

no es significativa a menos que un gran porcentaje del código sea paralelizable; al menos 90% y preferiblemente más del 95%. Si el 100% del código es paralelizable, tendremos aceleración lineal.

(1-f) %	*p=1*	*p=2*	*p=4*	*p=8*	*p=16*	*p=32*	*p=64*	*p=128*	*p=∞*
10	1.00	1.05	1.08	1.10	1.10	1.11	1.11	1.11	1.11
30	1.00	1.18	1.29	1.36	1.39	1.41	1.42	1.42	1.43
50	1.00	1.33	1.60	1.78	1.88	1.94	1.97	1.98	2.00
70	1.00	1.54	2.11	2.58	2.91	3.11	3.22	3.27	3.33
80	1.00	1.67	2.50	3.33	4.00	4.44	4.71	4.85	5.00
90	1.00	1.82	3.08	4.71	6.40	7.80	8.77	9.34	10.00
100	1.00	2.00	4.00	8.00	16.00	32.00	64.00	128.00	∞

Tabla 2. Aceleración en términos del porcentaje de código paralelizable.

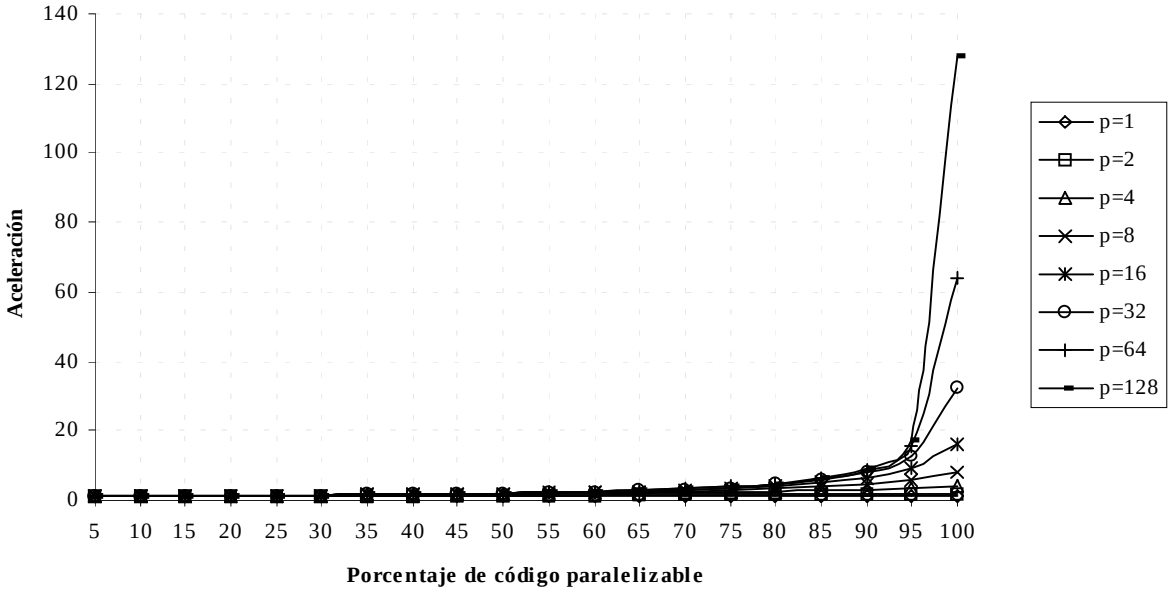

Figura 30. Aceleración obtenible de acuerdo al porcentaje de código paralelizable y al núnmero de procesadores empleados.

6.2. ACELERACION SUPERLINEAL.

De acuerdo a la ley de Amdahl, no es posible obtener una aceleración superior a la lineal. Esto esta basado en el supuesto de que un único procesador siempre puede emular procesadores

paralelos. Suponga que un algoritmo paralelo *A* resuelve una instancia del problema Π en T_p unidades de tiempo en un computador paralelo con *p* procesadores. Entonces el algoritmo *A* puede resolver la misma instancia del problema en pT_p unidades de tiempo en el mismo computador pero usando un solo procesador. Por lo tanto la aceleración no puede ser superior a *p*. Dado que usualmente los algoritmos paralelos tienen asociados costos adicionales de sincronización y comunicación, es muy probable que exista un algoritmo secuencial que resuelva el problema en menos de pT_p unidades de tiempo, haciendo que la aceleración sea menos que lineal.

Sin embargo, hay circunstancias algoritmicas especiales en las que una aceleración mayor que la lineal es posible. Por ejemplo, cuando se esta resolviendo un problema de búsqueda, un algoritmo secuencial puede perder una cantidad de tiempo considerable examinando estrategias que no llevan a la solución. Un algoritmo paralelo puede revisar muchas estrategias simultáneamente y se puede dar el caso de que una de ellas de con la solución rápidamente. En este caso el tiempo del algoritmo secuencial comparado con el paralelo es mayor que el número de procesadores empleados *p*.

También existen circunstancias de arquitectura que pueden producir aceleraciones superlineales. Por ejemplo, considere que cada CPU en un multiprocesador tiene cierta cantidad de memoria cache. Comparando con la ejecución en un solo procesador, un grupo de *p* procesadores ejecutando un algoritmo paralelo tiene *p* veces la cantidad de memoria cache. Es fácil construir ejemplos en los que la tasa colectiva de *hits* de cache para los *p* procesadores sea significativamente mayor que los *hits* de cache del mejor algoritmo secuencial corriendo en un solo procesador. En estas condiciones el algoritmo paralelo puede correr más de *p* veces más rápido. Estas circunstancias son especiales y se dan raras veces. Lo que aplica en la mayoria de los casos es la ley de Amdahl.

6.3. EFICIENCIA

Si normalizamos la aceleración dividiéndola por el número de procesadores obtenemos la eficiencia $\eta = A_p / p$. En la Tabla 3 se puede observar que cuando se incrementa el número de procesadores *p*, la eficiencia η decrementa. La Figura 31 muestra el gráfico respectivo. Esto refleja el hecho de que a medida que el código es dividido en trozos más y más pequeños, eventualmente algunos procesadores deben permanecer ociosos esperando que otros terminen ciertas tareas. En la practica esto no es así, ya que solo a medida que los problemas se hacen más grandes es que se asignan a más procesadores.

(1-f) %	*p=1*	*p=2*	*p=4*	*p=8*	*p=16*	*p=32*	*p=64*	*p=128*
0	1.00	0.50	0.25	0.13	0.06	0.03	0.02	0.01
10	1.00	0.53	0.27	0.14	0.07	0.03	0.02	0.01
30	1.00	0.59	0.32	0.17	0.09	0.04	0.02	0.01
50	1.00	0.67	0.40	0.22	0.12	0.06	0.03	0.02
70	1.00	0.77	0.53	0.32	0.18	0.10	0.05	0.03
80	1.00	0.83	0.63	0.42	0.25	0.14	0.07	0.04
90	1.00	0.91	0.77	0.59	0.40	0.24	0.14	0.07
100	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Tabla 3. Eficiencia en términos del porcentaje del código paralelizable y el número de procesadores usados.

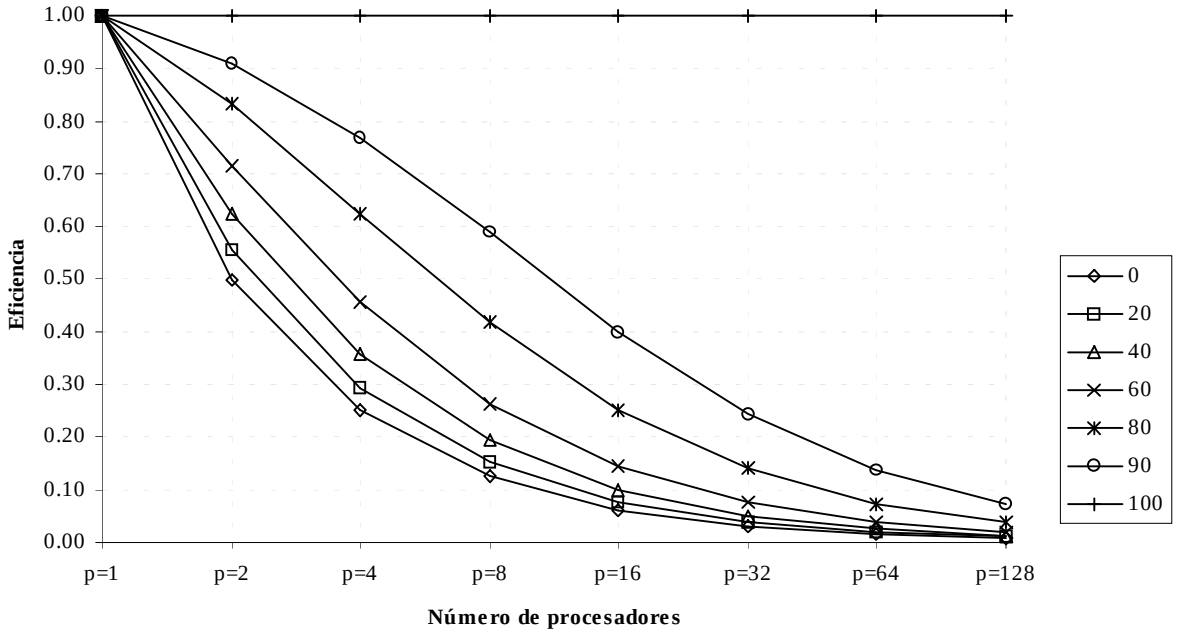

Figura 31. Eficiencia η para *p* procesadores y los distintos porcentajes de código paralelizable.

6.4. COSTO DE COMUNICACION

Asumamos que el código sea 100% paralelizable y por lo tanto la aceleración es $A_p = p$; despreciando el tiempo de comunicación entre procesadores. Considerando el *tiempo de comunicación* o *latencia* T_c , la aceleración decrece aproximadamente a

$$A_p = \frac{T}{T / p + T_c} < p$$

y para que la aceleración no sea afectada por la latencia de comunicación necesitamos que

$$\frac{T}{p} >> T_c \Rightarrow p << \frac{T}{T_c}$$

Esto significa que a medida que se divide el problema en partes más y más pequeñas para poder correr el problema en más procesadores, llega un momento en que el costo de comunicación T_c se hace muy significativo y desacelera el computo.

7. HERRAMIENTAS

Tal como en programación secuencial, en programación paralela hay diferentes lenguajes y herramientas de programación. Cada una adecuada para diferentes clases de problema. A continuación se presentan muy brevemente algunas de ellas.

C++ Composicional (C++): Es una extensión de C++ que provee al programador con facilidades para controlar localidad, concurrencia, comunicaciones, y asignación de tareas. Puede ser usado para construir librerías que implementen tareas, canales, y otras abstracciones básicas de la programación paralela.

High Performance Fortran (HPF): Es un ejemplo de lenguajes datos-paralelos y se ha convertido en un estándar para aplicaciones científicas e ingenieriles. El paralelismo es expresado en términos de operaciones sobre matrices y la comunicación es inferida por el compilador.

Parallel Virtual Machine (PVM): Es una librería de subrutinas para enviar y recibir mensajes.

Message Passing Interface (MPI): Similar a PVM pero, tal como HPF, MPI se ha convertido un estándar.

8. EJEMPLOS DE PROGRAMAS PARALELOS

A continuación se dan unos ejemplos muy sencillos de programas paralelos en ambientes de memoria compartida (usando semáforos y otros mecanismos de sincronización) y en ambientes de memoria distribuida (usando PVM). Estos ejemplos son solo con fines ilustrativos y no pretenden enseñar programación paralela.

8.1. *BUFFER* LIMITADO (MEMORIA COMPARTIDA)

Este ejemplo describe una situación clásica de productor/consumidor. Se tiene un proceso encargado de producir información que se guarda en un *buffer* de tamaño limitado, paralelamente a esto, existe un proceso que remueve la información del *buffer* y la consume. El *buffer* consta de *n* particiones de igual tamaño cada una con capacidad para almacenar un registro. En la implementación dada a continuación, el *buffer* es un arreglo es circular ya que al llegar a la ultima partición se regresa a la primera. El productor y el consumidor generan y procesan información indefinidamente.

```
PROGRAM Buffer_Limitado;
CONST n = 100;
TYPE tipo_registro = RECORD
       campo_1 : ...;
             .
             .
             .
       campo_i : ...
     END;
VAR
    k : INTEGER;
    registro : tipo_registro;
  SHARED
    buffer : ARRAY[0..n-1] OF tipo_registro;
    lleno, vacio : SEMAPHORE;
BEGIN
  k := 0;
  vacio := n;
  lleno := 0;
  IF (FORK <> 0) THEN           { yo soy el padre o productor }
    WHILE TRUE DO BEGIN
      { producir registro }
      DOWN(vacio);              { espera por una partición vacía }
      buffer[k] := registro;
      UP(lleno);                { una partición quedo con información }
      k := (k + 1) MOD n;
    END;
  ELSE                          { yo soy el hijo o consumidor }
    WHILE TRUE DO BEGIN
      DOWN(lleno);              { espera por información }
      registro := buffer[k];
      UP(vacio);                { una partición quedo vacía }
      k := (K + 1) MOD n;
      { consumir registro }
    END;
END.
```

Para generar los dos procesos que representan el productor y el consumidor, se usa la función **fork** del sistema operativo *Unix*. Esta función hace una replica del proceso que la invoco[13] y retorna el valor 0 en el *hijo* y la *identificación del hijo* (el *PID* -process ID- en *Unix*) en el padre, lo que permite distinguir ambos procesos en el código. A excepción de la variables declaradas en la sección **shared** que son compartidas, cada proceso tiene una copia privada de los datos y de los registros (*control program*, etc.). Nótese en el programa el uso privado que se le da a las variables *k* y *registro* ya que cada proceso tiene localidades distintas para ellas.

Se usan dos semáforos generales. Uno, *lleno*, indica el número de particiones del *buffer* con información que no ha sido procesada. El otro, *vacío*, da el número de particiones vacías o con información que ya fue procesada. Inicialmente *vacío* tiene el valor *n* ya que todas las particiones están sin información, y por la misma razón *lleno* tiene el valor cero. El hecho de usar un arreglo para implementar el *buffer*, hace que el productor siempre vaya adelante del consumidor y que manipulen particiones diferentes. Esto hace innecesario el uso de secciones criticas[14] a la hora de accesar el *buffer*. Sin embargo, si el *buffer* se implementa como una lista encadenada, o el programa se generaliza a *m* productores y *n* consumidores ($m,n > 1$), las secciones criticas se hacen necesarias ya que no quisiera tener, por ejemplo, más de un proceso manipulando o cambiando el mismo apuntador.

8.2. MULTIPLICACION DE MATRICES (MEMORIA COMPARTIDA)

El siguiente programa muestra una solución paralela a la multiplicación de dos matrices, A y B de tamaño *m.n* y *n.s* respectivamente, que producen como resultado la matriz C de tamaño *m.s*. En la implementación que se da a continuación, se emplea un número de N_PROCESOS procesos para ejecutar en paralelo la multiplicación: N_PROCESOS -1 procesos hijos y el proceso padre. Cada hijo esta encargado de procesar M DIV N_PROCESOS filas, y el padre las restantes: M - (M DIV N_PROCESOS).(N_PROCESOS -1). El padre también esta encargado de leer las matrices A y B, y de imprimir la matriz resultante C.

La implementación ilustra el uso de **barreras**[15]. Una barrera se inicializa a un valor que

[13] A la replica se le llama *proceso hijo* (o simplemente *hijo*) y al proceso que invoco la función *fork* se le llama *padre*.

[14] Una sección critica (*SC*) es un conjunto de instrucciones que pueden ser ejecutadas por uno y solo un proceso en un momento dado. Estas se implementan usando semáforos binarios: DOWN(*s*); *SC*; UP(*s*).

[15] En realidad la barrera no es necesaria en esta implementación ya que cada hijo, una vez creado, puede comenzar la multiplicación. Las matrices A y B ya han sido leídas y en el proceso de multiplicación solo son

representa el número de procesos que deben arribar a la barrera antes de permitírseles proseguir; adicionalmente se especifica que hacer con los procesos que van arribando. Si se usa SPIN_LOCK, como se hace en la implementación, los procesos retenidos revisan constantemente la barrera hasta que el número predefinido de procesos arriben a ella y puedan continuar. Con PROCESS_LOCK los procesos retenidos en la barrera son bloqueados, es decir; voluntariamente ceden el procesador sobre el cual están corriendo[16] y son reactivados al darse las condiciones. Esto no sucede con SPIN_LOCK en donde estos procesos solo son suspendidos por el sistema operativo cundo deben compartir los CPUs con otros procesos.

```
PROGRAM Multiplica_Matrices;
CONST M = 5000; N = 1000; S = 3000;
      N_PROCESOS = 10;  { 10 procesos efectuaran la multiplicación : el padre
                          y 9 hijos }
VAR
    i, j, k : INTEGER;
  SHARED
    A: ARRAY[1..M,1..N] OF REAL;   B: ARRAY[1..N,1..S] OF REAL;
    C: ARRAY[1..M,1..S] OF REAL;
    Barrera: BARRIER_TYPE;

PROCEDURE Crea_Hijos(Num_Hijos : INTEGER);
Var
  i, j, k, l : INTEGER;
BEGIN
  { inicializa la barrera con el número de procesos total }
  BARRIER_INIT(Barrera,Num_Hijos + 1,SPIN_BLOCK);

  { número de filas que procesará cada hijo - el padre procesa las restantes }
  l := M DIV N_PROCESOS;
  { filas que procesará el primer hijo : filas j a la k }
  j := 1;
  k := j + l -1;

  { crea el número de hijos solicitados }
  FOR i := 1 TO Num_Hijos DO BEGIN
    { crea un proceso hijo e indica filas a procesar }
    Crea_Hijo(Multiplica,j,k)
    { filas que procesará el hijo siguiente }
    j := j + l;
    k := j + l - 1;
  END;
```

fuentes de dato pero no de almacenamiento, por lo que se pueden accesar simultáneamente. Con respecto a los resultados, estos son almacenan en la matriz C y todos los procesos lo hacen en localidades distintas.

[16] Le dicen al sistema operativo “yo no tengo que hacer y puedes disponer del CPU que estoy utilizando”.

```
    { padre espera a que los hijos hayan sido creados - solo el padre ejecuta la
      siguiente barrera }
    BARRIER(Barrera);

    { el padre procesa las filas restantes }
    Multiplica(j,M);

  END; { de Crea_Hijos }

  { crea un hijo que corra el proceso P }
  PROCEDURE Crea_Hijo(PROCEDURE P; j, k : INTEGER);
  begin
    { FORK crea el proceso hijo y retorna 0 en el hijo }
    IF FORK = 0 THEN BEGIN
      { espera por todos los hijos a ser creados }
      BARRIER(Barrera);
      { hijo ejecuta el procedimiento P y procesa las filas respectivas }
      P(j,k);
      { termina el proceso hijo }
      EXIT(0)
    END
  END; { de Crea_hijo }

  PROCEDURE Multiplica(j, k : INTEGER);
  VAR
    f, c, l : INTEGER;
    suma : REAL;
  BEGIN
    FOR fil := j TO k DO
      FOR col := 1 TO S DO BEGIN
        C(fil,col):= 0;
        FOR l := 1 to N DO C(fil,col):= C(fil,col) + A[fil,l]*B[l,col];
      END;
  END; { de Multiplica }

  BEGIN
    { el padre lee las matrices A y B }

    { crea los hijos y que efectúen la multiplicación junto con el padre }
    Crea_Hijos(N_PROCESOS - 1);

    { el padre espera a que todos los hijos terminen - ejecuten EXIT }
    FOR i := 2 TO N_PROCESOS DO
      k := WAIT(j);

    { el padre imprime el resultado de la multiplicación - Matriz C }

  END.
```

Para terminar la ejecución de un hijo se usa la función **exit** de *Unix*. Esta función tiene un parámetro que puede ser usado para indicarle a padre bajo que condiciones el hijo termino su

ejecución: normal, error, etc. El padre usa la función **wait** de *Unix* para saber si algún hijo ha terminado. Cuando un proceso ejecuta una instrucción como `k := WAIT(j)`, el proceso es bloqueado hasta que uno de los hijos termine y cuando esto sucede (o haya sucedido antes de ejecutar la instrucción) `k` es la identificación del hijo (*PID*) y `j` el valor usado por dicho hijo al invocar la función exit.

8.3. FORK JOIN (MEMORIA DISTRIBUIDA)

El siguiente programa crea varias tareas o procesos (3 por defecto) que luego sincronizan enviando un mensaje a su padre, el cual recibe los mensajes e imprime información referente a los hijos. El programa contiene el código que ejecutaran tanto los hijos como el padre.

```
/* Ejemplo 1 - Capitulo 6 - Pag 65 - PVM:A Users' Guide... - Al Geist... */

#include <pvm3.h>       /* definiciones y prototipos de la librería PVM */
#define MAXNCHILD  20   /* máximo número de hijos que se pueden crear */
#define JOINTAG    11   /* etiqueta de los mensajes de unión */

int main(argc,argv)
     int argc;
     char* argv[];
{
  int ntask = 3;             /* número de tareas a crear - 3 por defecto */
  int info;                  /* código que retornan las llamadas a pvm */
  int mytid;                 /* mi identificación */
  int myparent;              /* la identificación de mi padre */
  int child[MAXNCHILD];      /* arreglo para la identificación de los hijos */
  int i, mydata, buf, len, tag, tid;

  mytid = pvm_mytid();       /* ¿cual es mi identificación? */

  /* revisa si hay error */
  if (mytid < 0) {
    pvm_perror(argv[0]);     /* imprime el error */
    return -1;               /* finaliza el programa */
  }

  myparent = pvm_parent();   /* ¿cual es la identificación de mi padre? */

  /* finaliza si hay un error distinto a PvmNoParent */
  if ((myparent < 0) && (myparent != PvmNoParent)) {
    pvm_perror(argv[0]);
    pvm_exit();
    return -1;
  }
```

```
  /* si no tengo padre entonces yo soy el padre */
  if (myparent == PvmNoParent) {
    /* revisa el número de tareas a crear */
    if (argc == 2) ntask = atoi(argv[1]);
    /* revisa que el número de tareas sea legal */
    if ((ntask < 1) || (ntask > MAXNCHILD)) { pvm_exit(); return 0; }
    /* crea las tareas hijo */
    info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0,
                     ntask, child);
    /* imprime el código del error o las identificaciones de los hijos */
    for (i = 0; i < ntask; i++)
      if (child[i] < 0) printf(" %d", child[1]); /* código de error */
      else printf("t%x\t", child[i]);            /* identificación del hijo */
    putchar('\n');

    /* ¿la creación de los hijos fue exitosa? */
    if (info == 0) { pvm_exit(); return -1; }

    /* solo se quiere respuesta de los hijos creados correctamente */
    ntask = info;

    for (i = 0; i < ntask; i++) {
      buf = pvm_recv(-1, JOINTAG);      /* recibe mensaje de cualquier hijo */
      if (buf < 0) pvm_perror("llamando recv");
      info = pvm_bufinfo(buf, &len, &tag, &tid);
      if (info < 0) pvm_perror("llamando bufinfo");
      info = pvm_upkint(&mydata, 1, 1);
      if (info < 0) pvm_perror("llamando upkint");
      if (mydata != tid) printf("Esto no debería ocurrir!\n");
      printf("Length %d, Tag %d, Tid t%x\n", len, tag, tid);
    }

    pvm_exit();
    return 0;
  }

  /* código de los hijos */
  info = pvm_initsend(PvmDataDefault);
  if (info < 0) { pvm_perror("llamando initsend"); pvm_exit(); return -1; }
  info = pvm_pkint(&mytid, 1, 1);
  if (info < 0) { pvm_perror("llamando pkint"); pvm_exit(); return -1; }
  info = pvm_send(myparent, JOINTAG);
  if (info < 0) { pvm_perror("llamando send"); pvm_exit(); return -1; }
  pvm_exit();
  return 0;
}
```

Lo primero que el programa debe hacer es llamar la rutina **pvm_mytid** que incorpora el proceso al PVM y devuelve su identificación. Si esta ejecución no es exitosa, se finaliza el programa. Seguidamente se hace una llamada a **pvm_parent** que devuelve la identificación del

padre en caso de que el proceso tenga uno. El primer proceso que comienza ejecutando el código no tiene padre (ya que el es el padre) y recibirá **PvmNoParent** como respuesta. Esto nos permite distinguir al padre de los hijos, el cual debe crearlos.

El código del padre revisa si se ha especificado un número de hijos distinto al definido por defecto y verifica que el número de hijos a crear sea valido. Una vez hecho esto, se llama **pvm_spawn** que le dice a PVM que intente crear *ntask* (quinto parámetro) tareas con nombre *argv[0]*. El segundo paramento es una lista de argumentos para los hijos que en este caso no se usara y por eso es nula. Con el tercer parámetro se le puede indicar a PVM en que máquina especifica o arquitectura en particular se deben crear los hijos. En este ejemplo no importa y se usa PvmTaskDefault, de lo contrario se deben usar valores adecuados en el tercer y cuarto parámetro. Finalmente *child* tendrá las identificaciones de los hijos. Luego se imprimen estas identificaciones o cualquier código de error. De no haber hijos creados exitosamente, el programa finaliza.

Por cada hijo creado el padre espera recibir un mensaje e imprime información sobre dicho mensaje. **pvm_recv** recibe cualquier mensaje de cualquier hijo que tenga la etiqueta JOINTAG y retorna un entero que indica un *buffer* de mensaje. Este entero es usado en **pvm_bufinfo** para obtener información sobre el mensaje. La única información que un hijo le envía a su padre es su identificación (un entero) la cual es desempacada del mensaje mediante **pvm_upkint**.

El ultimo segmento de código del programa es ejecutado solo por los hijos. Antes de poder colocar datos en el *buffer* del mensaje, deben inicializarlo usando **pvm_initsend**. El parámetro **PvmDataDefault** instruye al PVM a efectuar cualquier conversión necesaria para asegurarse de que los datos arriben en el proceso destino en el formato correcto. Finalmente los hijos usan **pvm_pkint** para empaquetar su identificación en el mensaje y **pvm_send** para enviarlo.

BIBLIOGRAFIA

1) Baron, R., y Higbie, L. *Computer Architecture*, Addison-Wesley, New York, 1992.
2) De Balsi, M. *Computer Architecture*, Addison-Wesley, New York, 1990.
3) Foster, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, New York, 1995.
4) Geist, A., Suderam, V., y otros. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Massachusetts, 1994.
5) Hwang, K., y Briggs, F. *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

6) Bic, L., y Shaw, A. *The Logical Design of Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.
7) Quinn, M. *Parallel Computing: Theory and Practice*, McGraw-Hill, New York, 1994.
8) Snir, M., Dongarra, J., y otors. *MPI: The Complete Reference.* MIT Press, Massachusetts, 1996.
9) Tanenbaum, A. *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

SITIOS DE INTERES EN INTERNET

1) *CeCalCULA*:
   http://www.cecalc.ula.ve/
2) Número 3 de la bibliografia:
   http://www.mcs.anl.gov/dbpp/
3) Número 4 de la bibliografia:
   htpp://www.netlib.org/pvm3/book/pvm-book.html/
4) Información sobre *MPI: Message Passing Interfae*:
   http://www.mcs.an.gov/mpi/index.html
5) Servidor Web del *IBM POWER Paralell System* en la *Universidad de Cornell*:
   http://ibm.tc.cornell.edu/
6) Información sobre la *SGI Origin2000*
   http://www.sgi.com/Products/hardware/servers/technology/overwiew.html
7) National HPCC Software Exchange:
   http://nhse.npac.syr.edu/
8) Queen's University SP2 Site:
   http://SP2.CCS.QueensU.CA/
9) Información sobre la TERA.
   http://www.tera.com
10) High Performance Computing and Communications Glossary:
    http://nhse.npac.syr.edu/hpccgloss/hpccgloss.html
11) 3Com Glossary of Networking Terms:
    http://www.3com.com