Capítulo 4.
Introducción al paralelismo y al rendimiento.

4.1. Magnitudes y medidas del rendimiento.

En esta sección se definirán algunas de las medidas más utilizadas a la hora de determinar el rendimiento de una arquitectura paralela. Aunque lo más común es medir la eficiencia de un sistema paralelo atendiendo a una de estas dos medidas:

- **Tiempo de ejecución** (o respuesta) de una aplicación.
- **Productividad** (*throughput*) o número de aplicaciones que es capaz de procesar por unidad de tiempo.

Pero no es tan sencillo como todo esto, por ello se introducen también otras medidas como: *speed-up*, eficiencia de un sistema, utilización, redundancia, etc. Este tema se encuentra completo en [Hwa93] y parte en [Ort05].

4.1.1. Eficiencia, redundancia, utilización y calidad

Ruby Lee (1980) definió varios parámetros para evaluar el cálculo paralelo. A continuación se muestra la definición de dichos parámetros.

Rapidez y Eficiencia del sistema. Sea $O(n)$ el número total de operaciones elementales realizadas por un sistema con n elementos de proceso, y $T(n)$ el tiempo de ejecución en pasos unitarios de tiempo. En general, $T(n) < O(n)$ si los n procesadores realizan más de una operación por unidad de tiempo, donde $n \geq 2$. Supongamos que $T(1) = O(1)$ en un sistema mono-procesador, lo cual supone que IPC = 1 (IPC= Instrucciones Por Ciclo). El *factor de mejora del rendimiento* (*speed-up*, aceleración o rapidez) se define como

$$S(n) = T(1)/T(n)$$

La *eficiencia del sistema* para un sistema con n procesadores se define como

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n * T(n)}$$

La eficiencia es una comparación del grado de *speed-up* conseguido frente al valor máximo. Dado que $1 \leq S(n) \leq n$, tenemos $1/n \leq E(n) \leq 1$.

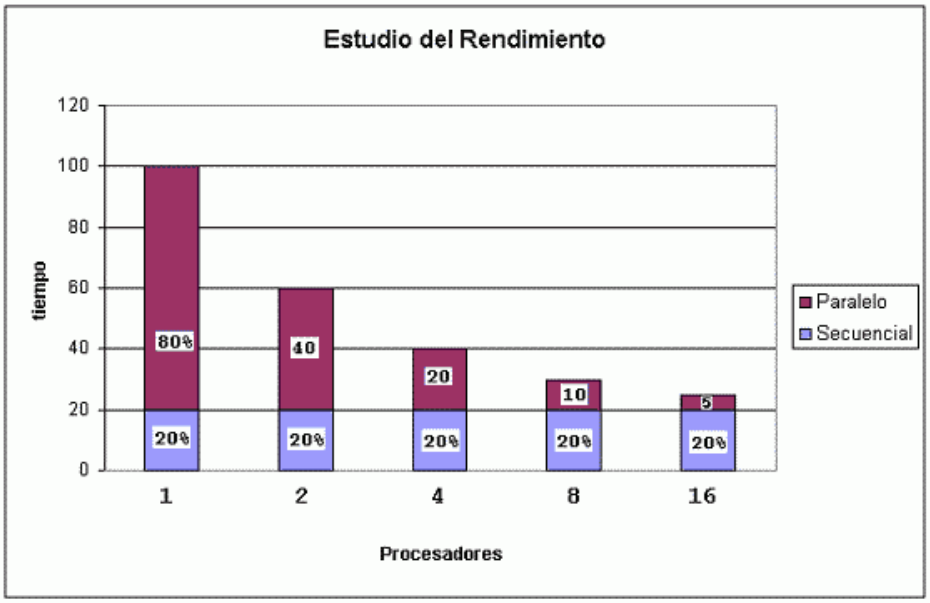
La eficiencia más baja $E(n) \rightarrow 0$ corresponde al caso en que todo el programa se ejecuta en un único procesador de forma serie. La eficiencia máxima $E(n) = 1$, se obtiene cuando todos los procesadores están siendo completamente utilizados durante todo el periodo de ejecución.

Escalabilidad. Un sistema se dice que es *escalable* para un determinado rango de procesadores [1... *n*], si la eficiencia *E*(*n*) del sistema se mantiene constante y en todo momento por encima de un factor *0.5*. Normalmente todos los sistemas tienen un determinado número de procesadores a partir del cual la eficiencia empieza a disminuir de forma más o menos brusca. Un sistema es más escalable que otro si este número de procesadores, a partir del cual la eficiencia disminuye, es menor que el otro.

No hay que confundir escalabilidad con ampliable. Un sistema es ampliable si físicamente se le pueden poner más módulos (más memorias, más procesadores, más tarjetas de entrada/salida, etc.). Que un sistema sea ampliable no significa que sea escalable, pues la eficiencia no tiene por qué mantenerse constante al ampliar el sistema y por tanto podría no ser escalable.

Veamos ahora el problema típico que nos mostrará lo importante que es que la parte paralelizable de un programa sea lo mayor posible si queremos obtener el máximo rendimiento a un sistema paralelo.

Problema 1: Sea un programa que posee un tiempo de ejecución de 100 unidades de tiempo (por ejemplo segundos). El 80% de su código es perfecta y absolutamente paralelizable. Se pide calcular el Rendimiento y la Eficiencia de este programa cuando se está ejecutando sobre {1, 2, 4, 8 ,16} procesadores.



$$S(n) = T(1)/T(n) \qquad E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

N =	**2**	**4**	**8**	**16**
S =	100/(40+20) = 1.666	2.5	3.333	4
E =	100/(2*60) = 0.83 = 1.666/2	0.625	0.416	0.25

Problema 2: Sea el mismo caso anterior, pero ahora el 90% de su código es perfectamente paralelizable. Calcular el Rendimiento y la Eficiencia. Estudia su escalabilidad.

Realizar un estudio sobre la rentabilidad de este sistema paralelo si el equipo básico con un procesador vale 1000€, y cada procesador añadido al sistema cuesta 100€.

¿Y si cada procesador de más cuesta 300€?

Redundancia. La *redundancia* en un cálculo paralelo se define como la relación $O(n)/O(1)$, que son respectivamente el número total de operaciones ejecutadas por el sistema con n procesadores dividido por el número de operaciones ejecutadas cuando utilizamos un solo procesador en la ejecución.

$$R(n) = O(n)/O(1)$$

Indica la relación entre el paralelismo software y hardware. Obviamente, $1 \leq R(n) \leq n$.

Utilización. La *utilización del sistema* en un cálculo paralelo se define como

$$U(n) = R(n)E(n) = O(n)/nT(n)$$

La utilización del sistema indica el porcentaje de recursos (procesadores, memoria, recursos, etc.) que se utilizan durante la ejecución de un programa paralelo. Es interesante observar la siguiente relación:

$$1/n \leq E(n) \leq U(n) \leq 1 \quad \text{y} \quad 1 \leq R(n) \leq 1/E(n) \leq n.$$

Calidad del paralelismo. La *calidad* de un cálculo paralelo es directamente proporcional al *speed-up* y la eficiencia, inversamente proporcional a la redundancia. Así, tenemos

$$Q(n) = \frac{S(n) * E(n)}{R(n)} = \frac{T^3(1)}{n * T^2(n) * O(n)}$$

Dado que $E(n)$ es siempre una fracción y $R(n)$ es un número entre 1 y n, la calidad $Q(n)$ está siempre limitada por el *speed-up* $S(n)$.

En resumen, usamos el *speed-up* $S(n)$ para indicar el grado de ganancia de velocidad de una computación paralela. La eficiencia $E(n)$ mide la porción útil del trabajo total realizado por n procesadores. La redundancia $R(n)$ mide el grado del incremento de la carga. La utilización $U(n)$ indica el grado de utilización de recursos durante un cálculo paralelo. Finalmente, la calidad $Q(n)$ combina el efecto del *speed-up*, eficiencia y redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema.

Veamos ahora algunos problemas que nos ayudarán a fijar mejor los conceptos y definiciones que acabamos de introducir. Sobre todo insistiremos en lo importante que es que nuestras aplicaciones paralelas posean un alto grado de paralelismo en el código.

Problema 3: Sabemos que para que un programa se escalable, debe cumplir que el rendimiento alcanzado debe ser mayor o igual al 50%. La pregunta es ¿Qué porcentaje de su código debe ser perfectamente paralelizable si queremos que sea escalable hasta 16 procesadores? ¿Y hasta 32?

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

Datos : E(16) = 50% = 0.5
T(1) = 100 = Ts + Tp(1)
T(n) = Ts + Tp(n) = Ts + [Tp(1)/n] = Ts + [(100-Ts)]/ n

Por lo tanto:

0.50 = 100 / {16 * [Ts + (100-Ts) / 16] }
0.50 = 100 / {16Ts + 100 - Ts }
0.50 = 100 / {15Ts + 100 }
15Ts + 100 = 100 / 0.50 = 200
Ts = (200 – 100) /15
Ts = 6,6666…

Tp = 93,3333… ➔ **Porcentaje paralelo = 93,3%**

Para N= 32 tenemos:

0.50 = 100 / {32 * [Ts + (100-Ts) / 32] }
. . .
Ts = (200 – 100) / 31 = 3,226 ➔ **Tp = 96.774**

➔ **Porcentaje paralelo = 96,8%**

Problema 4: Volvemos a los datos del problema 1. Se pide calcular el valor de la Redundancia *R(n)*, de la Utilización del sistema *U(n)* y de la Calidad del Paralelismo *Q(n)* del sistema con la siguiente premisa: la paralelización del código implica un incremento de 5 unidades de tiempo por proceso creado. *(Recordar que el total de instrucciones del código secuencial es 100)*

A) Realizar los cálculos para *n = {2, 4}* procesadores.
B) Realizarlos también para *n = 16* procesadores.

A)

```
O(n=2) = 20 + (40+5) * 2    = 110 instrucciones
T(n=2) = 20 + 45            =  65 ciclos
S(n=2) = 100 / (20+45)      = 1,538
E(n=2) = 1,538/2            = 0.769
R(n=2) = 110 / 100          = 1.1
U(n=2) = 1.1 *0.769         = 0.846
Q(n=2) = [1.538*0.769]/1.1 = 1.07

O(n=4) = 20 + (20+5) * 4    = 120 instrucciones
T(n=4) = 20 + 25            =  45 ciclos
S(n=4) = 100 / (20+25)      = 2.223
E(n=4) = 2.223/4            = 0.556
R(n=4) = 120 / 100          = 1.2
U(n=4) = 1.2 *0.556         = 0.667
Q(n=4) = [2.223 * 0.556]/1.2 = 1.0288
```

B) . . .

Problema 5: Sean los datos del problema 1. Se pide calcular el valor de la Redundancia *R(n)*, de la Utilización del sistema *U(n)* y de la Calidad del Paralelismo *Q(n)* del sistema con la siguiente premisa: la paralelización del código implica un incremento de 2 unidades de tiempo por proceso creado. Realizar los cálculos para $n = \{ 4, 16 , 32\}$ procesadores.

4.2. Modelos del rendimiento del *speed-up.*

En esta sección se describen tres modelos de medición del *speed-up*. La ley de Amdahl (1967) se basa una carga de trabajo fija o en un problema de tamaño fijo. La ley de Gustafson (1987) se aplica a problemas escalables, donde el tamaño del problema se incrementa al aumentar el tamaño de la máquina o se dispone de un tiempo fijo para realizar una determinada tarea. El modelo de *speed-up* de Sun y Ni (1993) se aplica a problemas escalables limitados por la capacidad de la memoria.

4.2.1. Ley de Amdahl, limitación por carga de trabajo fija.

En muchas aplicaciones prácticas, donde es importante la respuesta más rápida posible, la carga de trabajo se mantiene fija y es el tiempo de ejecución lo que se debe intentar reducir. Al incrementarse el número de procesadores en el sistema paralelo, la carga fija se distribuye entre más procesadores para la ejecución paralela. Por lo tanto, el objetivo principal es obtener los resultados lo más pronto posible. En otras palabras, disminuir el tiempo de respuesta es nuestra principal meta. A la ganancia de tiempo obtenida para este tipo de aplicaciones donde el tiempo de ejecución es crítico se le denomina *speed-up bajo carga fija*.

Retomemos el primer ejemplo visto en este tema e intentemos formular la ley de Amdahl en función de la parte del código que no es paralelizable (llamémosla α).

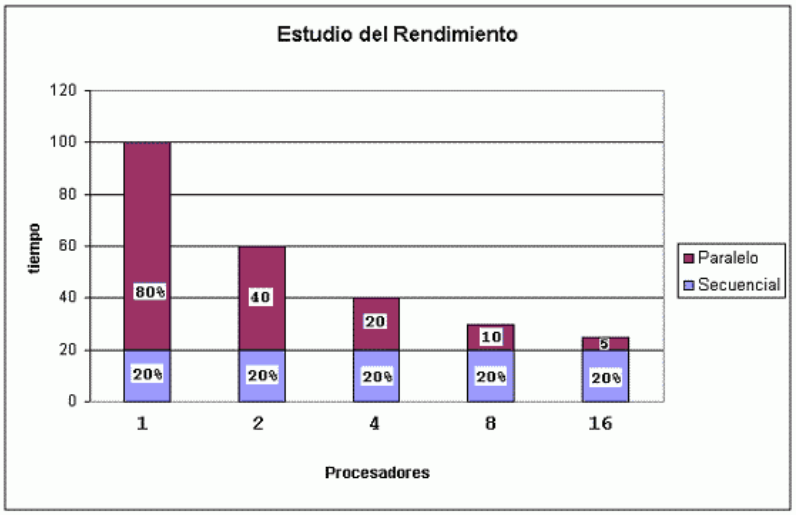


Figura 4.3. Ejemplo de la ley de Amdahl.

Tenemos un sistema con n procesadores y un código es tiene un porcentaje de código no paralelizable α y el resto perfectamente paralelizable (β = 1- α).

El rendimiento del sistema se calcula como

$$S(n) = T(1)/T(n)$$

La *eficiencia del sistema* para un sistema con *n* procesadores se define como

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n * T(n)}$$

Para calcular el rendimiento del sistema, tenemos que escribir los tiempos de ejecución para n=1 procesador y para cuando tenemos n procesadores. Es decir:

$$T(n=1) = T_s(1) + T_p(1) = (\alpha + \beta) * T$$

$$T(n) = T_s(n) + T_p(n) = T_s(1) + [T_p(1)/n] = \alpha T + [\beta T] / n$$

Y si tenemos en cuenta que $\beta = 1 - \alpha$ tenemos que el rendimiento del sistema o *speed-up* se puede expresar como:

$$S_n = \frac{T(1)}{T(n)} = \frac{T}{[T_s(n) + T_p(n)]} = \frac{T}{\alpha T + \frac{\beta T}{n}} = \frac{1}{\alpha + \frac{\beta}{n}} = \frac{1}{\frac{\alpha n + \beta}{n}} = \frac{n}{\alpha n + (1 - \alpha)}$$

$$S_n = \frac{n}{(1 + (n-1) \propto} \qquad (4.1)$$

A esta expresión se le conoce como la *ley de Amdahl*[1]. La implicación es que $S \rightarrow 1/\alpha$ cuando $n \rightarrow \infty$. En otras palabras, independientemente del número de procesadores que se emplee, existe un límite superior del *speed-up* debido a la parte serie de todo programa. En la figura 4.4 se han trazado las curvas correspondientes a la ecuación (4.1) para 4 valores de α. El *speed-up* ideal se

[1] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.

obtiene para α =0, es decir, el caso en que no hay parte serie a ejecutar y todo el código es paralelizable. A poco que el valor de α sea no nulo, el *speed-up* máximo empieza a decaer muy deprisa.

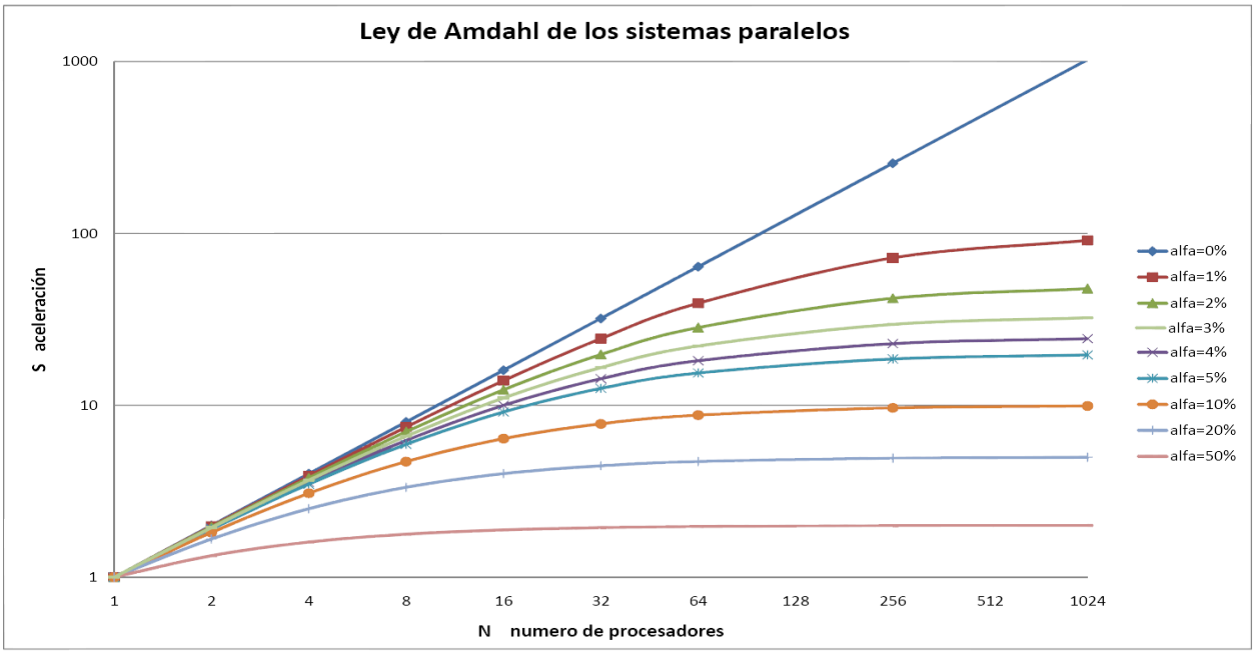


Figura 4.4: Mejora del rendimiento para diferentes valores de α, donde α es la fracción del cuello de botella secuencial (y 100- α es la porción paralela).

Esta ley de Amdahl se puede generalizar y es aplicable a cualquier problema que tenga una parte *mejorable* y otra que no se pueda mejorar.

Problema 8: Sea un sistema similar al del problema 1 que posee 20% del código secuencial y el 80% perfectamente paralelizable. Calcular el ***speed-up medio armónico*** para n = 16 procesadores.

$$S_{up} = \frac{1}{\alpha + \frac{1-\alpha}{n}} = \frac{n}{1 + n\alpha - \alpha} = \frac{n}{1 + (n-1)\alpha}$$

Con n=16 y α=0.20 tenemos:

Sup = 16 / (1 + (n-1) 0.20 = **4**

Problema 9: Sea un sistema que 5% del código secuencial y el 95% perfectamente paralelizable. Calcular el ***speed-up medio armónico*** para n = 8 procesadores y n = 16 procesadores.

4.3. Introducción a las arquitecturas paralelas.

Como bien nos cuenta David E. Culler en [Cul99] la contribución de la arquitectura al aumento de las prestaciones de los computadores ha venido de la mano del paralelismo y del aprovechamiento de la localidad de acceso a los datos. El uso de memorias caches en multiprocesadores ha contribuido a reducir los tiempos de accesos a los datos, pero por otra parte ocasiona problemas de coherencia que deben resolverse correctamente.

Hasta este momento se ha estudiado el paralelismo de ejecución de instrucciones. Se ha visto ya que la segmentación es un primer mecanismo de paralelismo, ya que varias instrucciones consecutivas son ejecutadas de forma solapada casi en paralelo. También se vio que los procesadores superescalares realizan también procesamiento paralelo al lanzar dos o más instrucciones al mismo tiempo gracias a la presencia de varios cauces de ejecución paralelos y se ha estudiado los procesadores VLIW.

Sin embargo, todos estos sistemas están basados en la arquitectura *Von Neumann* con un procesador y memoria donde se guardan datos y programa, es decir, una máquina secuencial que procesa datos escalares. Esta arquitectura se ha ido perfeccionando incluyendo el paralelismo de las unidades de control, de cálculo, etc., pero sigue siendo una máquina de ejecución con un único flujo de instrucciones.

No hay una frontera definida entre la arquitectura monoprocesador y las masivamente paralelas. De hecho, las actuales arquitecturas monoprocesador son realmente máquinas paralelas a nivel de instrucción. La evolución de la arquitectura basada en monoprocesador ha venido ligada con la creación de más y mejores supercomputadores que tenían que librarse del concepto de monoprocesador para poder hacer frente a las demandas de computación.

El primer paso hacia la paralelización de las arquitecturas de los computadores, se da con la aparición de los procesadores o sistemas vectoriales. Los procesadores vectoriales utilizan unidades aritméticas completamente segmentadas, extendiendo el concepto de paralelismo al tratamiento de grandes vectores de datos. En realidad son más una extensión del concepto de segmentación que un computador realmente paralelo.

Pero ¿necesitamos realmente sistemas paralelos? En [ORT05] podemos obtener una buena reflexión sobre este tema. Pero es mejor fijarnos en la evidencia, en estos momentos los grandes supercomputadores son utilizados en innumerables campos de la investigación civil, además de con fines militares. También se utilizan cada vez más en la industria, como el cine, en el diseño industrial, en la arquitectura, los simuladores para la F1, etc.

En este capítulo se repasarán los conceptos básicos sobre sistemas paralelos, supercomputadores y su clasificación, abordándose también las fuentes de paralelismo que utilizaremos para alimentar nuestros sistemas paralelos.

4.3.1 Clasificación de Flynn.

Probablemente la clasificación más popular de computadores sea la clasificación de Flynn. Esta taxonomía de las arquitecturas está basada en la clasificación atendiendo al flujo de datos e instrucciones en un sistema. Un flujo de instrucciones es el conjunto de instrucciones secuenciales que son ejecutadas por un único procesador, y un flujo de datos es el flujo secuencial de datos requeridos por el flujo de instrucciones. Con estas consideraciones, Flynn clasifica los sistemas en cuatro categorías:

- **SISD** (*Single Instruction stream, Single Data stream*): Flujo único de instrucciones y flujo único de datos. Este el concepto de arquitectura serie de Von Neumann donde, en cualquier momento, sólo se está ejecutando una única instrucción. A menudo a los SISD se les conoce como computadores serie escalares. Todas las máquinas SISD poseen un registro simple que se llama *contador de programa* que asegura la ejecución en serie del programa. Conforme se van leyendo las instrucciones de la memoria, el contador de programa se actualiza para que apunte a la siguiente instrucción a procesar en serie. Prácticamente ningún computador puramente SISD se fabrica hoy en día ya que la mayoría de procesadores modernos incorporan algún grado de paralelización como es la segmentación de instrucciones o la posibilidad de lanzar dos o más instrucciones a un tiempo (superescalares).
- **MISD** (*Multiple Instruction stream, Single Data stream*): Flujo múltiple de instrucciones y único flujo de datos. Esto significa que varias instrucciones actúan sobre el mismo y único trozo de datos. Este tipo de máquinas se pueden interpretar de dos maneras. Una es considerar la clase de maquinas que requerirían que unidades de procesamiento diferentes recibieran instrucciones distintas operando sobre los mismos datos. Esta clase de arquitectura ha sido clasificada por numerosos arquitectos de computadores como impracticable o imposible, y en estos momentos no existen ejemplos que funcionen siguiendo este modelo. Otra forma de interpretar los MISD es como una clase de máquinas donde un mismo flujo de datos fluye a través de numerosas unidades procesadoras. Arquitecturas altamente segmentadas, como los *arrays sistólicos* o los *procesadores vectoriales*, son clasificados a menudo bajo este tipo de máquinas. Las arquitecturas segmentadas, o encauzadas, realizan el procesamiento vectorial a través de una serie de etapas, cada una ejecutando una función particular produciendo un resultado intermedio. La razón por la cual dichas arquitecturas son clasificadas como MISD es que los elementos de un vector pueden ser considerados como pertenecientes al mismo dato, y todas las etapas del cauce representan múltiples instrucciones que son aplicadas sobre ese vector.
- **SIMD** *(Single Instruction stream, Multiple Data stream):* Flujo de instrucción simple y flujo de datos múltiple. Esto significa que una única instrucción es aplicada sobre diferentes datos al mismo tiempo. En las máquinas de este tipo, varias unidades de procesado diferentes son invocadas por una única unidad de control. Al igual que las

MISD, las SIMD soportan procesamiento vectorial (matricial) asignando cada elemento del vector a una unidad funcional diferente para procesamiento concurrente. Por ejemplo, el cálculo de la paga para cada trabajador en una empresa, es repetir la misma operación sencilla para cada trabajador; si se dispone de una arquitectura SIMD esto se puede calcular en paralelo para cada trabajador. Por esta facilidad en la paralelización de vectores de datos (los trabajadores formarían un vector) se les llama también *procesadores matriciales*.

- **MIMD** (*Multiple Instruction stream, Multiple Data stream*): Flujo de instrucciones múltiple y flujo de datos múltiple. Son máquinas que poseen varias unidades procesadoras en las cuales se pueden realizar múltiples instrucciones sobre datos diferentes de forma simultánea. Las MIMD son las más complejas, pero son también las que potencialmente ofrecen una mayor eficiencia en la ejecución concurrente o paralela. Aquí la concurrencia implica que no sólo hay varios procesadores operando simultáneamente, sino que además hay varios programas (procesos) ejecutándose también al mismo tiempo.

La figura 4.5 muestra los esquemas de estos cuatro tipos de máquinas clasificadas por los flujos de instrucciones y datos.

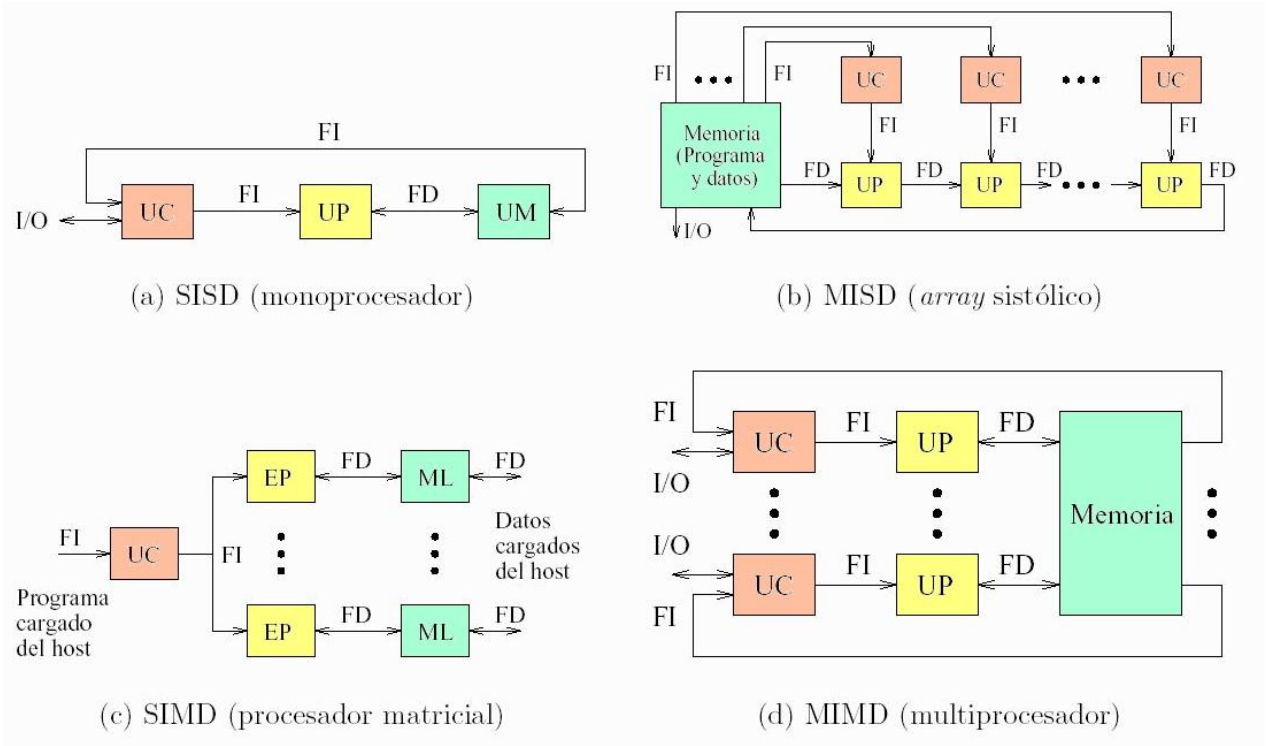


Figura 4.5: Clasificación de Flynn de las arquitecturas de computadores (UC=Unidad de Control, UP=Unidad de Procesamiento, UM=Unidad de Memoria, EP=Elemento de Proceso, ML=Memoria Local, FI=Flujo de Instrucciones, FD=Flujo de datos.)

4.3.2 Otras clasificaciones.

La clasificación de Flynn ha demostrado funcionar bastante bien para la tipificación de sistemas, y se ha venido usando desde décadas por la mayoría de los arquitectos de computadores. Sin embargo, los avances en tecnología y diferentes topologías, han llevado a sistemas que no son tan fáciles de clasificar dentro de los 4 tipos de Flynn. Por ejemplo, los procesadores vectoriales no encajan adecuadamente en esta clasificación, ni tampoco las arquitecturas híbridas. Para solucionar esto se han propuesto otras clasificaciones, donde los tipos SIMD y MIMD de Flynn se suelen conservar, pero que sin duda no han tenido el éxito de la de Flynn.

La figura 4.6 muestra una taxonomía ampliada que incluye alguno de los avances en arquitecturas de computadores en los últimos años. No obstante, tampoco pretende ser una caracterización completa de todas las arquitecturas paralelas existentes.

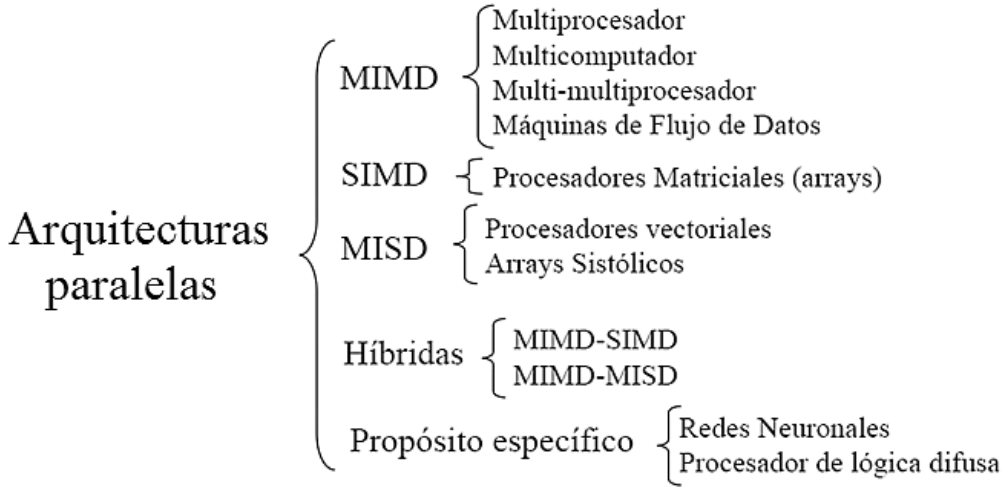


Figura 4.6: Clasificación de las arquitecturas paralelas.

Tal y como se ve en la figura, los de tipo MIMD pueden a su vez ser subdivididos en multiprocesadores, multicomputadores, multi-multiprocesadores y máquinas de flujo de datos. Incluso los multiprocesadores pueden ser subdivididos en NUMA, UMA y COMA según el modelo de memoria compartida. El tipo SIMD quedaría con los procesadores matriciales y el MISD se subdividiría en procesadores vectoriales y en *arrays* sistólicos. Se han añadido dos tipos más que son el híbrido y los de aplicación específica.

Multiprocesadores.

Un *multiprocesador* se puede ver como un computador paralelo compuesto por varios procesadores interconectados que pueden compartir un mismo sistema de memoria. Los procesadores se pueden configurar para que ejecute cada uno una parte de un programa o varios programas al mismo tiempo. Un diagrama de bloques de esta arquitectura se muestra en la figura 4.7. Tal y como se muestra en la figura, que corresponde a un tipo particular de multiprocesador

que se verá más adelante, un multiprocesador está generalmente formado por *n* procesadores y *m* módulos de memoria. A los procesadores los llamamos P_1, P_2,... , P_n y a las memorias M_1,M_2,... ,M_n. La red de interconexión conecta cada procesador a un subconjunto de los módulos de memoria.

Dado que los multiprocesadores comparten los diferentes módulos de memoria, pudiendo acceder varios procesadores a un mismo módulo, a los multiprocesadores también se les llama *sistemas de memoria compartida.* Dependiendo de la forma en que los procesadores comparten la memoria, podemos hacer una subdivisión de los multiprocesadores:

- **UMA** (*Uniform Memory Access*). En un modelo de *Memoria de Acceso Uniforme*, la memoria física está uniformemente compartida por todos los procesadores. Esto quiere decir que todos los procesadores tienen el mismo tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener su cache privada, y los periféricos son también compartidos de alguna manera. A estos computadores se les suele llamar *sistemas fuertemente acoplados* dado el alto grado de compartición de los recursos. La red de interconexión toma la forma de bus común, conmutador cruzado, o una red multietapa como se verá en próximos capítulos.

  Cuando todos los procesadores tienen el mismo acceso a todos los periféricos, el sistema se llama multiprocesador *simétrico*. En este caso, todos los procesadores tienen la misma capacidad para ejecutar programas, tal como el Kernel o las rutinas de servicio de I/O. En un multiprocesador *asimétrico*, sólo un subconjunto de los procesadores puede ejecutar programas. A los que pueden, o al que puede ya que muchas veces es sólo uno, se le llama *maestro.* Al resto de procesadores se les llama *procesadores adheridos* (*attached processors*). La figura 4.7 muestra el modelo UMA de un multiprocesador.

  Es frecuente encontrar arquitecturas de acceso uniforme que además tienen coherencia de caché, a estos sistemas se les suele llamar CC-UMA (*Cache-Coherent Uniform Memory Access*).

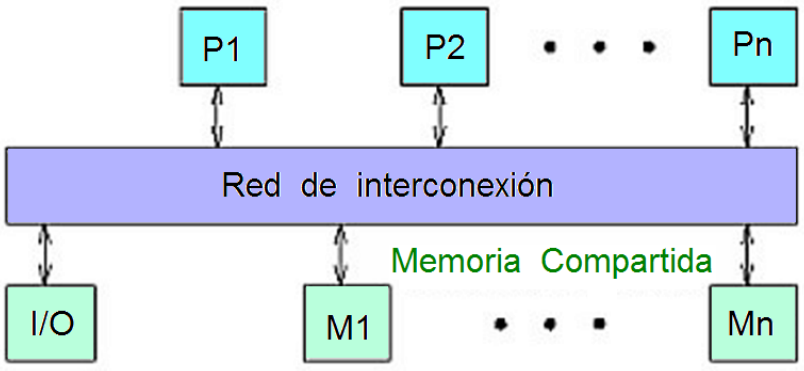


Figura 4.7: El modelo UMA de multiprocesador.

- **NUMA** (*Non Uniform Memory Access*). Un multiprocesador de tipo NUMA es un sistema de memoria compartida donde el tiempo de acceso varía según el lugar donde se encuentre localizado el acceso. La figura 4.8 muestra una posible configuración de tipo NUMA, donde toda la memoria es compartida pero local a cada módulo procesador. Otras posibles configuraciones incluyen los sistemas basados en agrupaciones (*clusters*) de sistemas como el de la figura que se comunican a través de otra red de comunicación que puede incluir una memoria compartida global.

  La ventaja de estos sistemas es que el acceso a la memoria local es más rápido que en los UMA aunque un acceso a memoria no local es más lento. Lo que se intenta es que la memoria utilizada por los procesos que ejecuta cada procesador, se encuentre en la memoria de dicho procesador para que los accesos sean lo más locales posible.

  Aparte de esto, se puede añadir al sistema una memoria de acceso global. En este caso se dan tres posibles patrones de acceso. El más rápido es el acceso a memoria local. Le sigue el acceso a memoria global. El más lento es el acceso a la memoria del resto de módulos. Al igual que hay sistemas de tipo CC-UMA, también existe el modelo de acceso a memoria no uniforme con coherencia de caché CC-NUMA (*Cache-Coherent Non-Uniform Memory Access*) que consiste en memoria compartida distribuida y directorios de cache.

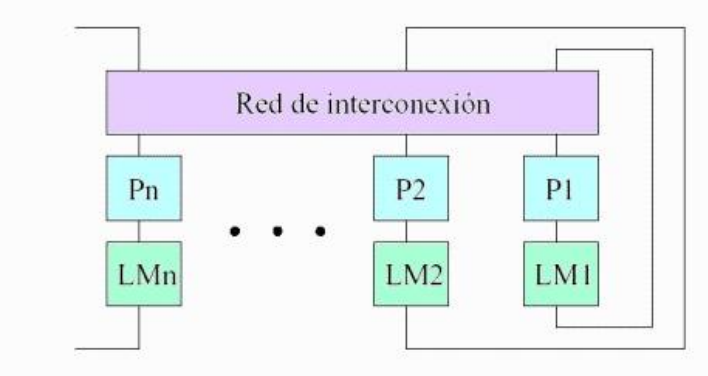


Figura 4.8: El modelo NUMA de multiprocesador.

- **COMA** (*Cache Only Memory Access*). Un multiprocesador que sólo use caché como memoria es considerado de tipo COMA. La figura 4.9 muestra el modelo COMA de multiprocesador. En realidad, el modelo COMA es un caso especial del NUMA donde las memorias distribuidas se convierten en cachés. No hay jerarquía de memoria en cada módulo procesador. Todas las cachés forman un mismo espacio global de direcciones. El acceso a las cachés remotas se realiza a través de los directorios distribuidos de las cachés. Dependiendo de la red de interconexión empleada, se pueden utilizar jerarquías en los directorios para ayudar en la localización de copias de bloques de caché. El emplazamiento inicial de datos no es crítico puesto que el dato acabará estando en el lugar en que se use más.

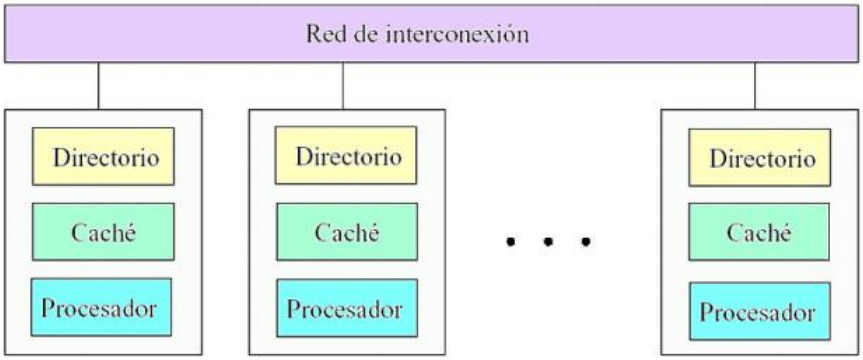


Figura 4.9: El modelo COMA de multiprocesador.

Multicomputadores.

Un multicomputador se puede ver como un computador paralelo en el cual cada procesador tiene su propia memoria local. La memoria del sistema se encuentra distribuida entre todos los procesadores y cada procesador sólo puede direccionar su memoria local; para acceder a las memorias de los demás procesadores debe hacerlo por *paso de mensajes.* Esto significa que un procesador tiene acceso directo sólo a su memoria local, siendo indirecto el acceso al resto de memorias del resto de procesadores. Este acceso local y privado a la memoria es lo que diferencia los multicomputadores de los multiprocesadores.

El diagrama de bloques de un multicomputador coincide con el visto en la figura 4.8 que corresponde a un modelo NUMA de procesador, la diferencia viene dada porque la red de interconexión no permite un acceso directo entre memorias, sino que la comunicación se realiza por paso de mensajes.

La transferencia de datos se realiza a través de la red de interconexión que conecta un subconjunto de procesadores con otro subconjunto. La transferencia de unos procesadores a otros se realiza por tanto por múltiples transferencias entre procesadores conectados dependiendo de cómo esté establecida la red.

Dado que la memoria está distribuida entre los diferentes elementos de proceso, a estos sistemas se les llama *distribuidos* aunque no hay que olvidar que pueden haber sistemas que tengan la memoria distribuida pero compartida y por lo tanto no ser multicomputadores. Además, y dado que se explota mucho la localidad, a estos sistemas se les llama *débilmente acoplados*, ya que los módulos funcionan de forma casi independiente unos de otros.

Multicomputadores con memoria virtual compartida

En un multicomputador, un proceso de usuario puede construir un espacio global de direccionamiento virtual. El acceso a dicho espacio global de direccionamiento se puede realizar por software mediante un paso de mensajes explícito. En las bibliotecas de paso de mensajes hay siempre rutinas que permiten a los procesos aceptar mensajes de otros procesos, con lo que cada proceso puede servir datos de su espacio virtual a otros procesos. Una lectura se realiza mediante

el envío de una petición al proceso que contiene el objeto. La petición por medio del paso de mensajes puede quedar oculta al usuario, ya que puede haber sido generada por el compilador que tradujo el código de acceso a una variable compartida.

De esta manera el usuario se encuentra programando un sistema aparentemente basado en memoria compartida cuando en realidad se trata de un sistema basado en el paso de mensajes. A este tipo de sistemas se les llama multicomputadores con memoria virtual compartida.

Otra forma de tener un espacio de memoria virtual compartido es mediante el uso de páginas. En estos sistemas una colección de procesos tiene una región de direcciones compartidas pero, para cada proceso, sólo las páginas que son locales son accesibles de forma directa. Si se produce un acceso a una página remota, entonces se genera un fallo de página y el sistema operativo inicia una secuencia de pasos de mensaje para transferir la página y ponerla en el espacio de direcciones del usuario.

Máquinas de flujo de datos

Hay dos formas de procesar la información, una es mediante la ejecución en serie de una lista de comandos y la otra es la ejecución de un comando demandado por los datos disponibles. La primera forma empezó con la arquitectura de Von Neumann donde un programa almacenaba las órdenes a ejecutar, sucesivas modificaciones, etc., han convertido esta sencilla arquitectura en los multiprocesadores para permitir paralelismo.

La segunda forma de ver el procesamiento de datos quizá es algo menos directa, pero desde el punto de vista de la paralelización resulta mucho más interesante puesto que las instrucciones se ejecutan en el momento tienen los datos necesarios para ello, y naturalmente se debería poder ejecutar todas las instrucciones demandadas en un mismo tiempo. Hay algunos lenguajes que se adaptan a este tipo de arquitectura comandada por datos como son el Prolog, el ADA, etc., es decir, lenguajes que exploten de una u otra manera la concurrencia de instrucciones.

En una arquitectura de flujo de datos una instrucción está lista para su ejecución cuando los datos que necesita están disponibles. La disponibilidad de los datos se consigue por la canalización de los resultados de las instrucciones ejecutadas con anterioridad a los operandos de las instrucciones que esperan. Esta canalización forma un flujo de datos que van disparando las instrucciones a ejecutar. Por esto se evita la ejecución de instrucciones basada en *contador de programa* que es la base de la arquitectura Von Neumann.

Las instrucciones en un flujo de datos son puramente autocontenidas; es decir, no direccionan variables en una memoria compartida global, sino que llevan los valores de las variables en ellas mismas. En una maquina de este tipo, la ejecución de una instrucción no afecta a otras que estén listas para su ejecución. De esta manera, varias instrucciones pueden ser ejecutadas simultáneamente lo que lleva a la posibilidad de un alto grado de concurrencia y paralelización.

La figura 4.10 muestra el diagrama de bloques de una máquina de flujo de datos. Las instrucciones, junto con sus operandos, se encuentran almacenados en la memoria de datos e instrucciones (D/I). Cuando una instrucción está lista para ser ejecutada, se envía a uno de los elementos de proceso (EP) a través de la red de arbitraje. Cada EP es un procesador simple con memoria local limitada. El EP, después de procesar la instrucción, envía el resultado a su destino a través de la red de distribución.

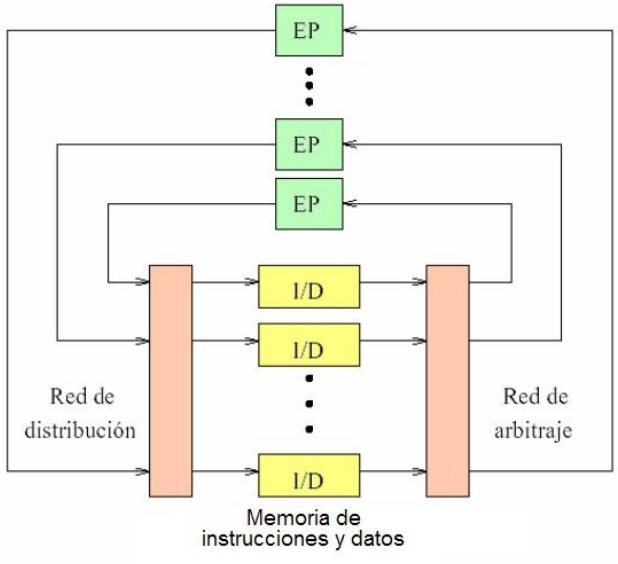


Figura 4.10: Diagrama de bloques de una máquina de flujo de datos.

Procesadores matriciales

Esta arquitectura es la representativa del tipo SIMD, es decir, hay una sola instrucción que opera concurrentemente sobre múltiples datos.

Un procesador matricial consiste en un conjunto de elementos de proceso y un procesador escalar que operan bajo una unidad de control. La unidad de control busca y decodifica las instrucciones de la memoria central y las manda bien al procesador escalar o bien a los nodos procesadores dependiendo del tipo de instrucción. La instrucción que ejecutan los nodos procesadores es la misma simultáneamente, los datos serán los de cada memoria de procesador y por tanto serán diferentes. Por todo esto, un procesador matricial sólo requiere un único programa para controlar todas las unidades de proceso.

La idea de utilización de los procesadores matriciales es explotar el paralelismo en los datos de un problema más que paralelizar la secuencia de ejecución de las instrucciones. El problema se paraleliza dividiendo los datos en particiones sobre las que se pueden realizar las mismas operaciones. Un tipo de datos altamente particionable es el formado por vectores y matrices, por eso a estos procesadores se les llama matriciales.

Procesadores vectoriales

Un procesador vectorial ejecuta de forma segmentada instrucciones sobre vectores de datos. La diferencia con los matriciales es que mientras los matriciales son comandados por las instrucciones, los vectoriales son comandados por flujos de datos continuos. A este tipo se le considera MISD puesto que varias instrucciones son ejecutadas sobre un mismo dato (el vector), si bien es una consideración algo confusa aunque aceptada de forma mayoritaria. En la Figura 4.11 se puede observar las dos unidades aritméticas que forman un procesador vectorial, una escalar y otra vectorial completamente segmentada.

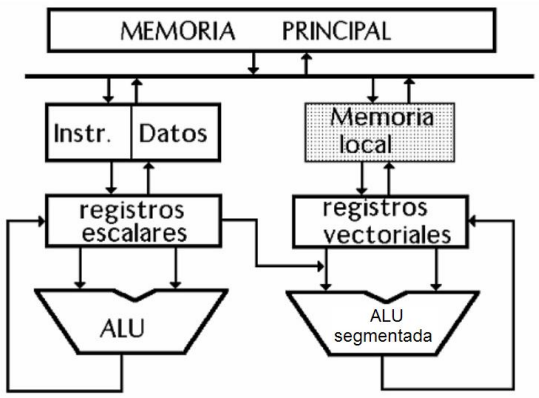


Figura 4.11: Diagrama de bloques de una procesador vectorial.

Este tipo de arquitectura funciona especialmente bien cuando nos manejamos con vectores o matrices de datos. En la figura 4.12 se puede ver como se reduce drásticamente el número de instrucciones emitidas cuando se quiere realizar un bucle que calcule los valores de un vector.

Esta reducción vienen dada por el hecho de que el bucle se elimina y se convierte en solo 6 instrucciones que son emitidas y ejecutadas utilizando la Alu convencional y la Alu completamente segmentada que utiliza como entrada vectores de datos.

```
      LD      F0,a
      ADDI    R4,Rx,#512   ;última dirección a cargar
loop:
      LD      F2,0(Rx)     ;carga X(i)
      MULTD   F2,F0,F2     ;a·X(i)
      LD      F4,0(Ry)     ;carga Y(i)
      ADDD    F4,F2,F4     ;a·X(i) + Y(i)
      SD      F4,0(Ry)     ;almacena en Y(i)
      ADDI    Rx,Rx,#8     ;incrementa índice a X
      ADDI    Ry,Ry,#8     ;incrementa índice a Y
      SUB     R20,R4,Rx    ;calcula límite
      BNZ     R20,loop     ;comprobación si se ha terminado
```

Aquí está el código DLXV para DAXPY.

```
      LD      F0,a         ;carga escalar a
      LV      V1,Rx        ;carga vector X
      MULTSV  V2,F0,V1     ;multiplicación vector-escalar
      LV      V3,Ry        ;carga vector Y
      ADDV    V4,V2,V3     ;suma
      SV      Ry,V4        ;almacena el resultado
```

Figura 4.12: Realización en una máquina vectorial de un bucle.

Arrays sistólicos

Otro tipo de máquinas que se suelen considerar MISD son los *arrays* sistólicos. En un *array* sistólico hay un gran número de elementos de proceso (EPs) idénticos con una limitada memoria local. Los EPs están colocados en forma de matriz (*array*) de manera que sólo están permitidas las conexiones con los EPs vecinos. Por lo tanto, todos los procesadores se encuentran organizados en una estructura segmentada de forma lineal o matricial. Los datos fluyen de unos EPs a sus vecinos a cada ciclo de reloj, y durante ese ciclo de reloj, o varios, los elementos de proceso realizan una operación sencilla. El adjetivo *sistólico* viene precisamente del hecho de que todos los procesadores vienen sincronizados por un único reloj que hace de "corazón" que hace moverse a la máquina.

Arquitecturas híbridas

Hemos visto dos formas de explotar el paralelismo. Por un lado estaba la paralelización de código que se consigue con las máquinas de tipo MIMD, y por otro lado estaba la paralelización de los datos conseguida con arquitecturas SIMD y MISD. En la práctica, el mayor beneficio en paralelismo viene de la paralelización de los datos. Esto es debido a que el paralelismo de los datos explota el paralelismo en proporción a la cantidad de los datos que forman el cálculo a realizar. Sin embargo, muchas veces resulta imposible explotar el paralelismo inherente en los datos del problema y se hace necesario utilizar tanto el paralelismo de control como el de datos. Por lo tanto, procesadores que tienen características de MIMD y SIMD (o MISD) a un tiempo, pueden resolver de forma efectiva un elevado rango de problemas.

Arquitecturas específicas

Las arquitecturas específicas son muchas veces conocidas también con el nombre de *arquitecturas VLSI* ya que muchas veces llevan consigo la elaboración de circuitos específicos con una alta escala de integración.

Un ejemplo de arquitectura de propósito específico son las redes neuronales (ANN de *Artificial Neural Network*). Las ANN consisten en un elevado número de elementos de proceso muy simples que operan en paralelo. Estas arquitecturas se pueden utilizar para resolver el tipo de problemas que a un humano le resultan fáciles y a una máquina tan difícil, como el reconocimiento de patrones, comprensión del lenguaje, etc. La diferencia con las arquitecturas clásicas es la forma en que se programa; mientras en una arquitectura *Von Neumann* se aplica un programa o algoritmo para resolver un problema, una red de neuronas aprende a fuerza de aplicarle patrones de comportamiento.

La idea es la misma que en el cerebro humano. Cada elemento de proceso es como una neurona con numerosas entradas provenientes de otros elementos de proceso y una única salida que va a otras neuronas o a la salida del sistema. Dependiendo de los estímulos recibidos por las entradas a la neurona la salida se activará o no dependiendo de una función de activación. Este esquema permite dos cosas, por un lado que la red realice una determinada función según el umbral de

activación interno de cada neurona, y por otro, va a permitir que pueda programarse la red mediante la técnica de ensayo-error.

Otro ejemplo de dispositivo de uso específico son los procesadores basados en *lógica difusa.* Estos procesadores tienen que ver con los principios del razonamiento aproximado. La lógica difusa intenta tratar con la complejidad de los procesos humanos eludiendo los inconvenientes asociados a lógica de dos valores clásica.

4.4. Fuentes del paralelismo

El procesamiento paralelo tiene como principal objetivo explotar el paralelismo inherente a las aplicaciones informáticas. Todas las aplicaciones no presentan el mismo perfil cara al paralelismo: unas se pueden paralelizar mucho y en cambio otras muy poco. Existen distintos niveles en los que se puede encontrar paralelismo. Así, podemos encontrar paralelismo en:

- A Nivel de Instrucciones u Operaciones, como hemos visto en las arquitecturas monoprocesador estudiadas hasta la fecha. Es la granularidad más fina.
- A Nivel de Bucle, que también hemos visto en esta asignatura y que nos ha permitido utilizar múltiples unidades aritméticas en paralelo, mejorando el rendimiento de los programas. Es un caso de granularidad fina-media.
- A Nivel de Funciones, en el que los distintos procedimientos que constituyen un programa se ejecutan simultáneamente. Tenemos granularidad media, en este caso.
- Y finalmente a Nivel de Programas, cuando en nuestro sistema paralelo ejecutamos distintos programas concurrentemente, perteneciendo estos a una misma aplicación o no. Se dice en este caso que nuestro paralelismo es de granularidad gruesa.

Al lado de este factor cuantitativo evidente, es necesario considerar también un factor cualitativo: la manera a través de la cual se explota el paralelismo. Cada técnica de explotación del paralelismo se denomina fuente. Distinguiremos tres fuentes principales:

- El paralelismo de control.
- El paralelismo de datos.
- El paralelismo de flujo.

Y pasaremos a estudiar a continuación cada una de estas fuentes de paralelismo.

4.4.1 El paralelismo de control.

La explotación del paralelismo de control proviene de la constatación natural de que en una aplicación existen acciones que podemos “hacer al mismo tiempo”. Las acciones, llamadas también tareas o procesos pueden ejecutarse de manera más o menos independiente sobre unos

recursos de cálculo llamados también procesadores elementales (o PE). La figura 4.13 muestra el concepto que subyace tras el paralelismo de control

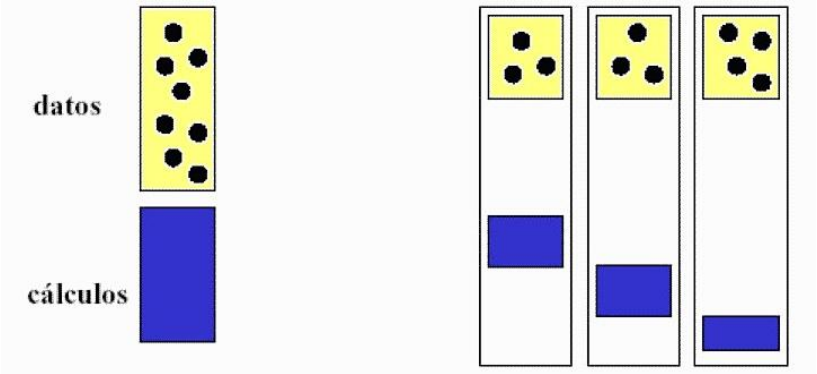


Figura 4.13: Paralelismo de control.

En el caso de que todas las acciones sean independientes es suficiente asociar un recurso de cálculo a cada una de ellas para obtener una ganancia en tiempo de ejecución que será lineal: *N* acciones independientes se ejecutarán *N* veces más rápido sobre *N* Elementos de Proceso (PE) que sobre uno solo. Este es el caso ideal, pero las acciones de un programa real suelen presentar dependencias entre ellas. Distinguiremos dos clases de dependencias que suponen una sobrecarga de trabajo:

- Dependencia de control de secuencia: corresponde a la secuenciación en un algoritmo clásico.
- Dependencia de control de comunicación: una acción envía informaciones a otra acción.

La explotación del paralelismo de control consiste en administrar las dependencias entre las acciones de un programa para obtener así una asignación de recursos de cálculo lo más eficaz posible, minimizando estas dependencias. La figura 4.14 muestra un ejemplo de paralelismo de control aplicado a la ejecución simultánea de instrucciones.

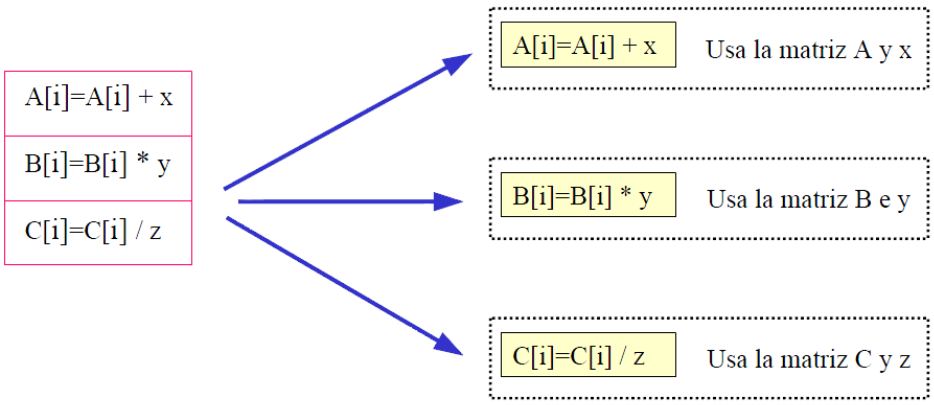


Figura 4.14: Ejemplo de paralelismo de control.

4.4.2 El paralelismo de datos

La explotación del paralelismo de datos proviene de la constatación natural de que ciertas aplicaciones trabajan con estructuras de datos muy regulares (vectores, matrices) repitiendo una misma acción sobre cada elemento de la estructura. Los recursos de cálculo se asocian entonces a los datos. A menudo existe un gran número (millares o incluso millones) de datos idénticos. Si el número de PE es inferior al de datos, éstos se reparten en los PE disponibles. La figura 4.15 muestra de forma gráfica el concepto de paralelismo de datos.

Como las acciones efectuadas en paralelo sobre los PE son idénticas, es posible centralizar el control. Siendo los datos similares, la acción a repetir tomará el mismo tiempo sobre todos los PE y el controlador podrá enviar, de manera síncrona, la acción a ejecutar a todos los PE.

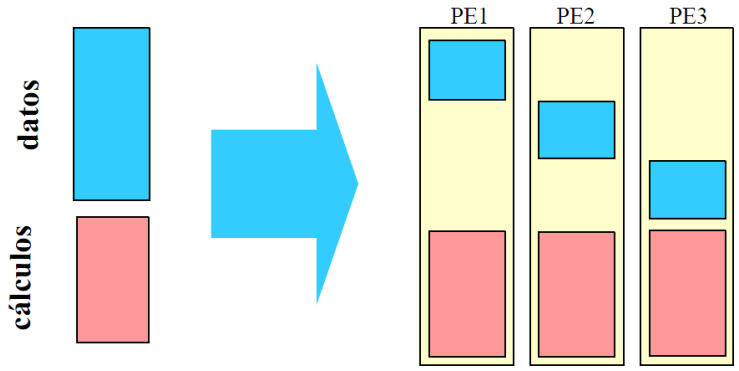


Figura 4.15. Paralelismo de datos.

Las limitaciones de este tipo de paralelismo vienen dadas por la necesidad de dividir los datos vectoriales para adecuarlos al tamaño soportado por la máquina, la existencia de datos escalares que limitan el rendimiento y la existencia de operaciones de difusión (un escalar se reproduce varias veces convirtiéndose en un vector) y reducciones que no son puramente paralelas. En la figura 4.16 se muestra un ejemplo de paralelismo de datos.

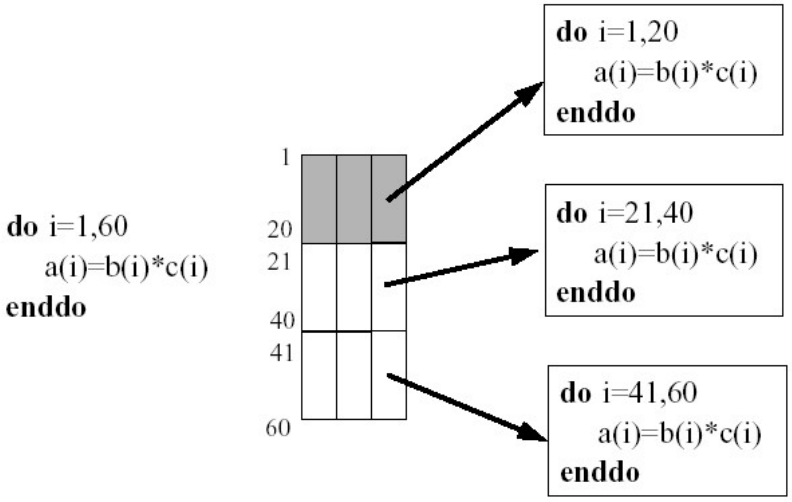


Figura 4.16: Ejemplo de la aplicación del paralelismo de datos a un bucle.

4.4.3. El paralelismo de flujo.

La explotación del paralelismo de flujo proviene de la constatación natural de que ciertas aplicaciones funcionan en modo "secuencia de acciones": disponemos de un flujo de datos, generalmente semejantes, sobre los que debemos efectuar una sucesión de operaciones en cascada. La figura 4.17 muestra de forma gráfica el concepto de paralelismo de flujo.

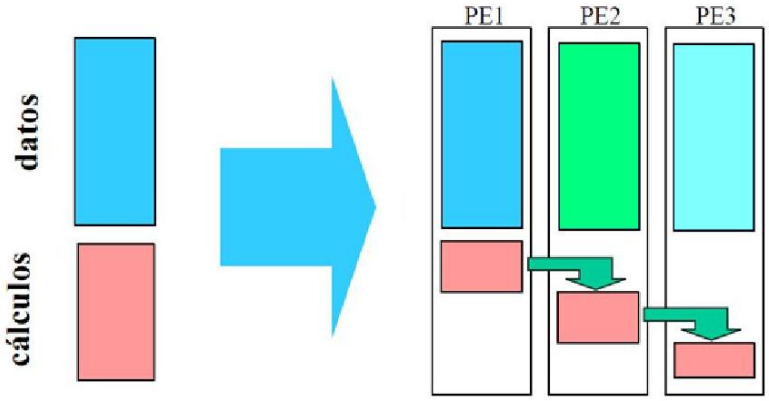


Figura 4.17: Paralelismo de flujo.

Los recursos de cálculo se asocian a las acciones y en cadena, de manera que los resultados de las acciones efectuadas en el instante *t* pasen en el instante *t+1* al PE siguiente. Este modo de funcionamiento se llama también *segmentación* o *pipeline*.

El flujo de datos puede provenir de dos fuentes:

- Datos de tipo vectorial ubicados en memoria. Existe entonces una dualidad fuerte con el caso del paralelismo de datos.
- Datos de tipo escalar provenientes de un dispositivo de entrada. Este dispositivo se asocia a menudo a otro de captura de datos, colocado en un entorno de tiempo real.

En ambos casos, la ganancia obtenida está en relación con el número de etapas (número de PE). Todos los PEs no estarán ocupados mientras el primer dato no haya recorrido todo el cauce, lo mismo ocurrirá al final del flujo. Si el flujo presenta frecuentes discontinuidades, las fases transitorias del principio y del fin pueden degradar seriamente la ganancia. La existencia de bifurcaciones también limita la ganancia obtenida.

4.4.4 Ejemplos de Aplicaciones Paralelas.

En esta sección se pretende que el alumno busque aplicaciones de las cuales se pueda extraer paralelismo. De todo tipo. Algunas aplicaciones pueden tener incluso varias fuentes de paralelismo posible. A continuación se muestran algunas aplicaciones donde el uso de multiprocesadores está indicado si queremos resolverlas con rapidez:

- Análisis y transmisión de video: Transformada Wavelet, codificadores, …
- Algoritmos Genéticos.
- Análisis de Redes de Interacción (grafos en realidad).
- Ajedrez. ¿Por qué ya no hay partidas hombre–máquina?
- Predicciones Meteorológicas, . . .

Los trabajos no tienen porque realizarse de todas estas aplicaciones. Vale con que al menos presentéis dos de ellas. Pero eso sí, contar con detalle en qué consisten estos algoritmos y como extraeríais paralelismo de ellas.

Importante: os animo a que me presentéis aplicaciones que no aparezcan en la lista.

Bibliografía:

[Hwa93]: Kai Hwang. *Advanced computer architecture: Parallelism, scalability, programmability*. McGraw-Hill, 1993. BIBLIOTECA: CI 681.3 HWA (3 copias), CI-IFIC (1 copia), CI-Informática (1 copia).

[Ort05]: *Arquitectura de computadores.* J. Ortega, M. Anguita y A. Prieto. Thomson, 2005.