



ugr

Universidad
de Granada

ARQUITECTURA DE COMPUTADORES
GRADO EN INGENIERÍA INFORMÁTICA

Resumen del temario

Autor

Carlos Sánchez Páez



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2017-2018

Índice

1. Tema 1	2
1.1. Arquitecturas paralelas y niveles de paralelismo	2
1.1.1. Niveles y tipos de paralelismo implementados en la arquitectura . .	2
1.1.2. Niveles y tipos de paralelismo implícito en una aplicación	2
1.1.3. Unidades de ejecución: instrucciones, hebras y procesos	6
1.1.4. Relación entre paralelismo implícito, explícito y arquitecturas pa- ralemas	7
1.1.5. Detección, utilización, implementación y extracción de paralelismo .	7
1.2. El paralelismo en las arquitecturas	8
1.2.1. Clasificaciones de las arquitecturas paralelas	8
1.3. Espacio de diseño. Clasificación y estructura general	11
1.3.1. Clasificación	11
1.3.2. Propuesta de clasificación de arquitecturas con múltiples threads . .	14
1.4. Evolución y prestaciones de las arquitecturas	14
1.4.1. Tiempo de CPU de un programa	14
1.4.2. Medidas de productividad: MIPS y MFLOPS	16
1.4.3. Conjuntos de programas de prueba (<i>benchmarks</i>)	16
1.4.4. Ganancia en prestaciones	17

Códigos fuente

1. Dependencia RAW	3
2. Dependencia WAW	3
3. Dependencia WAR	3
4. Paralelismo funcional	5
5. Paralelismo de datos	6

Índice de figuras

1. Niveles de paralelismo y granularidad	4
2. Paralelismo de tareas	5
3. Paralelismo implícito, explícito y arquitecturas paralelas	7
4. Detección, utilización, implementación y extracción del paralelismo	8
5. Taxonomía de Flynn	9
6. Arquitectura SISD	10
7. Arquitectura SIMD	10
8. Arquitectura MIMD	10
9. Arquitectura MISD	11
10. Jerarquía de buses	13
11. Multietapa	13
12. Barras cruzadas	13
13. Optimizaciones del tiempo de CPU y sus causantes	15

1. Tema 1

1.1. Arquitecturas paralelas y niveles de paralelismo

1.1.1. Niveles y tipos de paralelismo implementados en la arquitectura

Un computador es un sistema complejo tanto desde el punto de vista del hardware como desde el del software. Para que su estudio sea más sencillo, éste se divide en diferentes niveles de abstracción.

Por ejemplo, a nivel hardware, los niveles podrían ser: de componentes, de circuito electrónico, nivel de lógica digital, RT y nivel de sistema computador. En cada uno de estos niveles se implementa el paralelismo.

Para incrementar las prestaciones de un sistema se aprovecha el paralelismo (explícito o implícito) en las entradas. Hay dos alternativas para implementar paralelismo en un sistema aprovechando las entradas:

1. *Replicar* componentes del sistema.
2. *Segmentar* el uso de los componentes.

Por ejemplo, el paralelismo en un procesador se implementa replicando unidades funcionales y segmentando en uso de sus componentes. Los computadores paralelos son arquitecturas paralelas que implementan paralelismo *a nivel de sistema computador*. Para ello, replican computadores.

1.1.2. Niveles y tipos de paralelismo implícito en una aplicación

En el mercado hay computadores que implementan paralelismo en varios niveles de la arquitectura. En una aplicación se pueden distinguir distintos niveles de paralelismo, que se aprovechan en diferentes niveles del computador. Podemos clasificar estos niveles según el nivel de abstracción dentro del código secuencial ¹ de un programa en el que podamos encontrar el paralelismo.

Dependencia de datos Antes de ver los tipos de paralelismo, veamos lo que son las dependencias de datos. Para que el bloque de código B_2 presente una dependencia de datos con respecto a B_1 :

1. Ambos bloques deben referenciar a una misma posición de memoria M (variable).
2. B_1 debe aparecer en el código antes que B_2

¹Código escrito en lenguaje imperativo, próximo a las arquitecturas de flujo de control.

Tipos de dependencias de datos

1. **RAW**. Read After Write o *dependencia verdadera*.

```
1  ...
2  a=b+c //[B1 escribe a]
3  d=a+c //[B2 lee a]
4  ...
```

Listing 1: Dependencia RAW

2. **WAW**. Write After Write o *dependencia de salida*.

```
1  ...
2  a=b+c //[B1 escribe a]
3  a=d+e //[B2 escribe a]
4  ...
```

Listing 2: Dependencia WAW

3. **WAR**. Write After Read o *antidependencia*.

```
1  ...
2  b=a+1 //[B1 lee a]
3  a=d+e //[B2 escribe a]
4  ...
```

Listing 3: Dependencia WAR

Paralelismo funcional Podemos considerar que un programa está compuesto por funciones (nivel de funciones), éstas están compuestas por bucles (nivel de bucle) y éstos se basan en operaciones (nivel de operaciones). Como nivel superior encontramos al de programas, que pueden formar parte de la misma aplicación y usuario o no.

En general el paralelismo está implícito en mayor o menor grado en la descripción de una aplicación. Dentro del código secuencial podemos encontrar paralelismo implícito en los siguientes niveles de abstracción:

1. *Nivel de programas*. Los diferentes programas que intervienen en una o varias aplicaciones se pueden ejecutar en paralelo. Es poco probable que exista dependencia entre ellos.
2. *Nivel de funciones*. Un programa puede considerarse constituido por funciones. Se pueden ejecutar en paralelo, siempre que no haya dependencias (riesgos) inevitables entre ellas, como dependencias de datos verdaderas (lectura después de escritura, RAW, *Read After Write*). *Nivel de bucle (bloques)*. Una función puede estar basada en la ejecución de uno o varios bucles. El código dentro de cada uno se ejecuta múltiples veces, completando una tarea en cada iteración. Se pueden ejecutar en paralelo las iteraciones de un bucle siempre que resolvamos los riesgos derivados de las dependencias verdaderas. Para detectarlas, tendremos que analizar las entradas y las salidas de las iteraciones del bucle.

3. *Nivel de operaciones.* Se extrae el paralelismo disponible entre operaciones. Aquellas que son independientes se podrán ejecutar en paralelo. Por otra parte, en los procesadores podemos encontrar instrucciones compuestas de varias operaciones que se aplican en secuencia al mismo flujo de datos de entrada. Por tanto, podremos utilizar estas instrucciones compuestas para evitar penalizaciones por dependencias verdaderas.

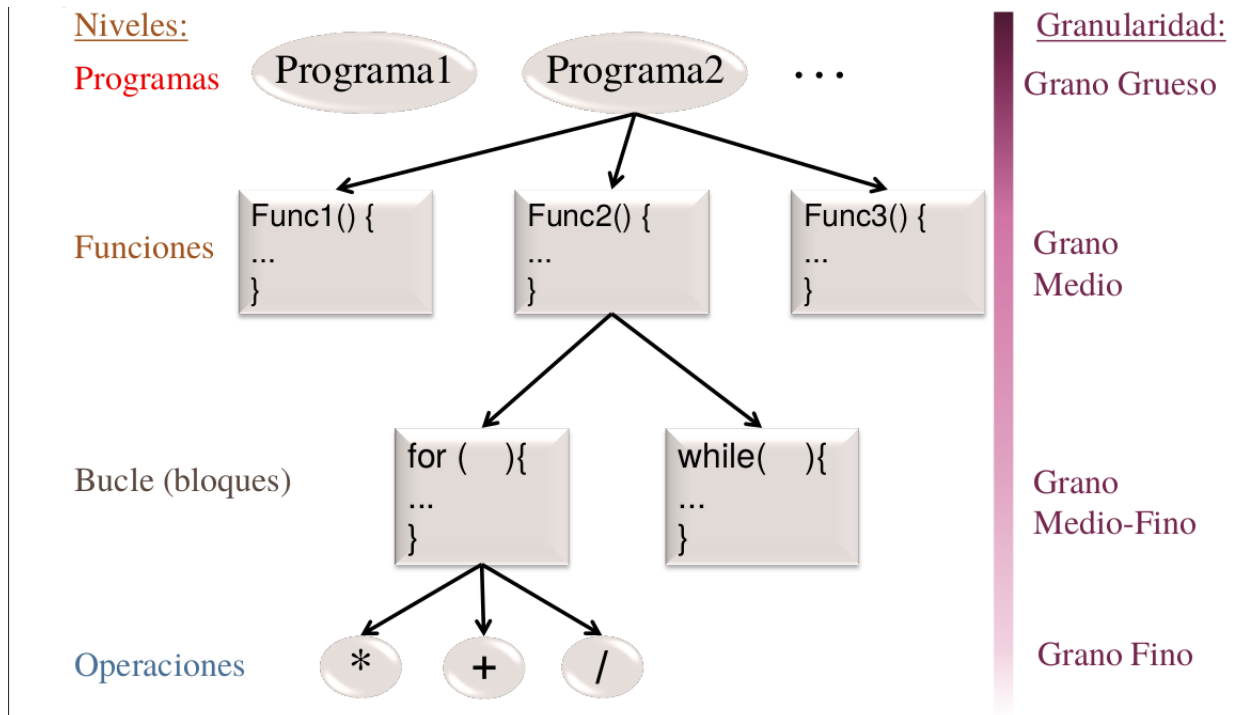


Figura 1: Niveles de paralelismo y granularidad

A este paralelismo que se puede detectar en distintos niveles de un código secuencial se le denomina *paralelismo funcional*.

Por ejemplo:

```
1  #include <stdio.h>
2  using namespace std;
3  int func1();
4  int func2();
5  int func3();
6  int main(){
7      #pragma omp pararell sections{
8          #pragma omp section
9              func1();
10         #pragma omp section
11             func2();
12         #pragma omp section
13             func3();
14     }
15 }
```

Listing 4: Paralelismo funcional

Paralelismo de tareas El *paralelismo de tareas* se encuentra extrayendo de la definición de la aplicación la estructura lógica de funciones de la aplicación. En esta estruycutura, los bloques son funciones y las conexiones entre ellos reflejan el flujo de datos entre funciones. Analizando esta estructura podemos encotnrar paralelismo entre las funciones. Está relacionado con el **paralelismo a nivel de función**.

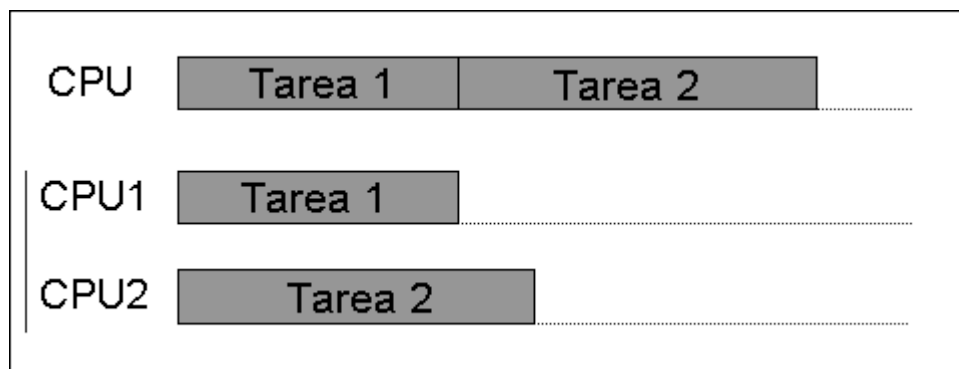


Figura 2: Paralelismo de tareas

Paralelismo de datos El *paralelismo de datos* está relacionado con el **paralelismo a nivel de bucle**. Se encuentra implícito en las operaciones con estructuras de datos (vectores y matrices) y se puede extraer de una representación matemática de las operaciones de la aplicación. Por ejemplo, como vectores y matrices se operan mediante bucles, podremos implementarlas mediante paralelismo a nivel de bucle. Así aparece el paralelismo de datos al analizar las operaciones realizadas con la misma estructura de datos pero en distinta iteración.²

²Las instrucciones multimedia de los procesadores suelen acelerar el procesamiento vectorial ya que aplican la misma operación en paralelo a múltiples datos dentro de un registro.

El paralelismo también se puede clasificar en función de la *granularidad* o *magnitud de la tarea* candidata a la paralelización. El grano más pequeño (*grano fino*) se asocia generalmente al paralelismo entre operaciones o instrucciones y el *grueso* al paralelismo entre programas. Entre ambos existe el *grano medio*, asociado a los bloques funcionales lógicos de la aplicación.

Por ejemplo:

```
1  #include <stdio.h>
2  using namespace std;
3  const int SIZE=100;
4  int main(){
5  int a[SIZE];
6  int b[SIZE];
7  int c[SIZE];
8  for(int i=0;i<SIZE;i++)
9      a[i]=b[i]+c[i]
10 return 0
11 }
```

Listing 5: Paralelismo de datos

1.1.3. Unidades de ejecución: instrucciones, hebras y procesos

En el nivel superior, el sistema operativo se encarga de gestionar la ejecución de unidades de mayor granularidad (*procesos y hebras*). Cada proceso en ejecución tiene su propia región de memoria. Los sistemas operativos multihebra permiten que un proceso se componga de una o varias hebras.

La diferencia principal entre hebra y proceso es que una hebra tiene su propia pila y contenido de registros, entre ellos IP (*Instruction Pointer*) que almacena la dirección de la siguiente instrucción a ejecutar por la hebra, pero comparte código, variables globales y otros recursos como archivos abiertos con el resto de hebras del mismo proceso. Estas características hacen que las hebras se puedan crear y destruir en menor tiempo que los procesos y que la comunicación, sincronización y conmutación entre hebras de un proceso sea más rápida que entre procesos. Esto permite que las hebras tengan una **granularidad menor que los procesos**.

El paralelismo implícito en el código de una aplicación se puede hacer *explícito* a nivel de instrucciones, hebras, procesos o dentro de una instrucción.

1.1.4. Relación entre paralelismo implícito, explícito y arquitecturas paralelas

El paralelismo se puede hacer explícito de varias formas:

1. El paralelismo *entre programas* se utiliza a través de procesos. Cuando se ejecuta un programa, se crea su proceso asociado.
2. El paralelismo *entre funciones* se puede utilizar a nivel de procesos o de hebras.
3. El paralelismo *dentro de un bucle* también se puede utilizar a nivel de procesos o hebras. Además, podemos aumentar la granularidad asociando un mayor número de iteraciones a cada unidad de ejecución paralela. También se puede hacer explícito dentro de una instrucción vectorial.
4. El paralelismo *entre operaciones* se puede aprovechar en arquitecturas con paralelismo a nivel de instrucción (*ILP*), ejecutando en paralelo las instrucciones asociadas a las operaciones independientes.

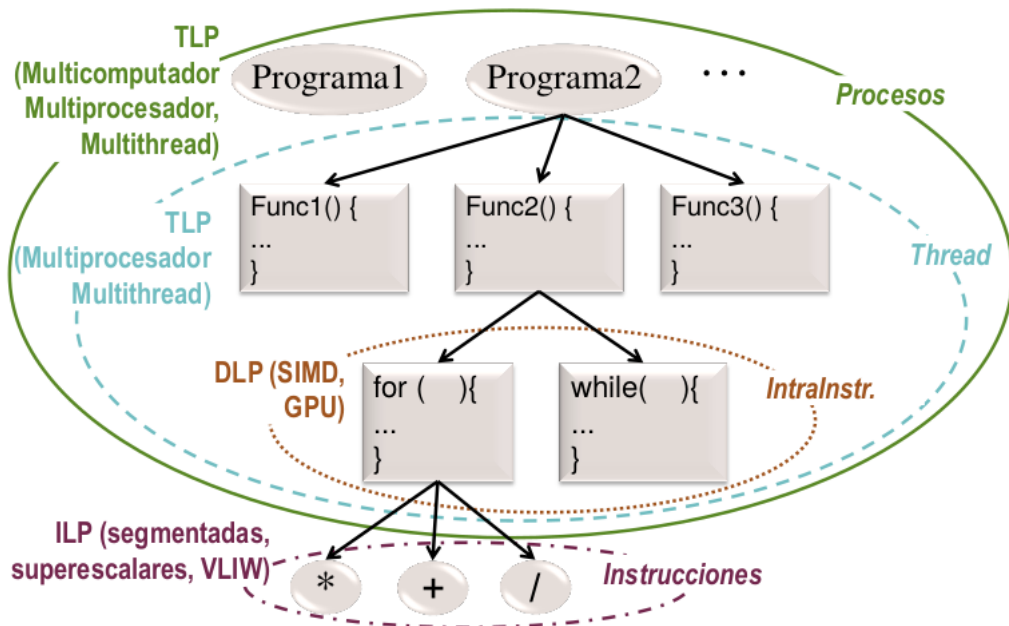


Figura 3: Paralelismo implícito, explícito y arquitecturas paralelas

1.1.5. Detección, utilización, implementación y extracción de paralelismo

En los procesadores *ILP* superescalares o segmentados la arquitectura extrae paralelismo. Para ello, se eliminan dependencias de datos falsas entre instrucciones. En estos procesadores, la arquitectura extrae paralelismo *implícito* en las entradas en tiempo de ejecución (*dinámicamente*). El **grado de paralelismo** de las instrucciones aprovechado se puede incrementar con ayuda de *compilador* y *programador*. En general,

Definición. El grado de paralelismo de un conjunto de entradas a un sistema es el máximo número de entradas del conjunto que se pueden ejecutar en paralelo.

Aclaración. En el caso de los procesadores, las entradas son instrucciones.

Debido a las dependencias entre entradas, este grado máximo será generalmente inferior al de entradas del conjunto. En las arquitecturas ILP VLIW ³ el paralelismo está ya *explícito* en las entradas, ya que el paralelismo se determina fuera del hardware. En este caso, el análisis de dependencias es estático; el compilador es el principal responsable de extraer el paralelismo. No obstante, la ayuda de un programador puede incrementar el grado de concurrencia aprovechado finalmente por la arquitectura. Para el compilador es difícil extraer el paralelismo, por lo que si el programador lo hace definiendo hebras y/o procesos se conseguirá mayor concurrencia.

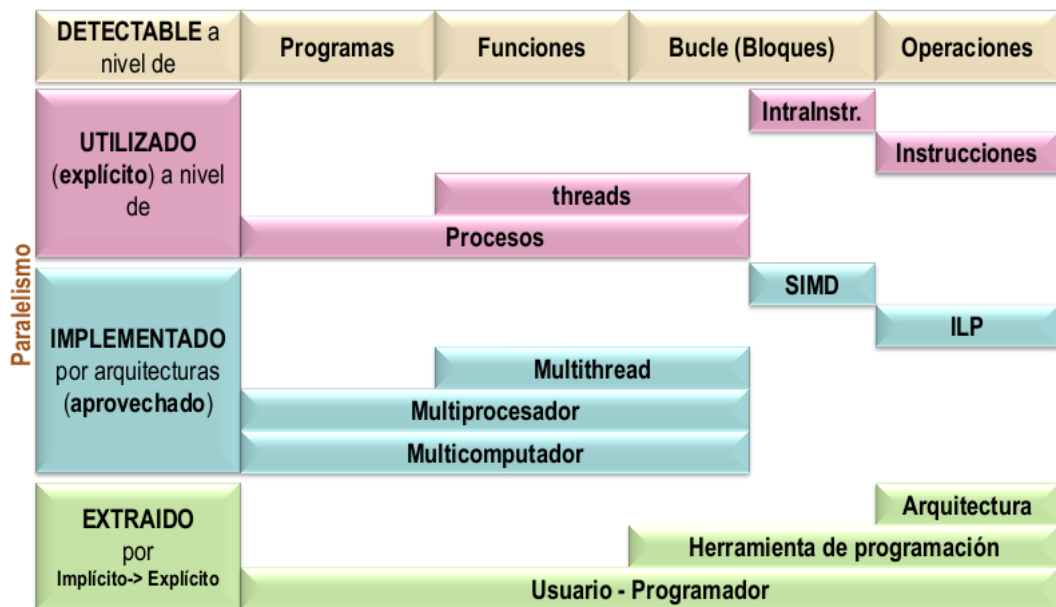


Figura 4: Detección, utilización, implementación y extracción del paralelismo

1.2. El paralelismo en las arquitecturas

1.2.1. Clasificaciones de las arquitecturas paralelas

El paralelismo se ha implementado en las arquitecturas siguiendo dos líneas fundamentales.

1. **Replicación de elementos.** Se incluyen unidades funcionales, procesadores, módulos de memoria, etc. entre los que se distribuye el trabajo. Ejemplos: multiprocesadores, procesadores de E/S, etc.
2. **Segmentación de cauce.** Consiste en dividir un elemento (unidad funcional, procesador, etc.) en una serie de etapas que funcionan de forma independiente y por las que pasan los operandos, instrucciones, etc. procesados por el elemento. Así, el elemento en cuestión realiza simultáneamente distintas etapas de su procesamiento.

La **clasificación (o taxonomía) de Flynn** divide los computadores en cuatro clases según el número de secuencias o flujos *de instrucciones* y flujos o secuencias *de datos* que se pueden procesar simultáneamente en el computador.

³Instruction Level Parallelism Very Long Instruction Word

1. **Computadores SISD** (*Single Instruction Single Data*). Un único flujo de instrucciones genera resultados, definiendo un único flujo de datos.
2. **Computadores SIMD** (*Single Instruction Multiple Data*). Un único flujo de instrucciones procesa operandos y genera resultados, definiendo varios flujos de datos, ya que cada instrucción codifica realmente varias operaciones iguales que actúan sobre operandos distintos.
3. **Computadores MIMD** (*Multiple Instruction Multiple Data*). El computador ejecuta varias secuencias o flujos distintos de instrucciones y cada uno de ellos procesa operandos y genera resultados definiendo un único flujo de instrucciones, de forma que también se generan varios flujos de datos, uno por cada flujo de instrucciones.
4. **Computadores MISD** (*Multiple Instruction Single Data*). Se ejecutan varios flujos distintos de instrucciones aunque todos actúan sobre el mismo flujo de datos.

		Datos	
		Simples	Múltiples
Instrucciones	Simples	SISD	SIMD
	Múltiples	MISD	MIMD

Figura 5: Taxonomía de Flynn

SISD En un computador **SISD** existe una única unidad de control que recibe las instrucciones de memoria, las decodifica y genera los códigos que definen la operación correspondiente a cada instrucción que debe realizar la unidad de procesamiento. El flujo de datos se establece a partir de los operandos necesarios para realizar la operación correspondiente (se traen de memoria) y de los resultados generados por las instrucciones (se almacenan en memoria).

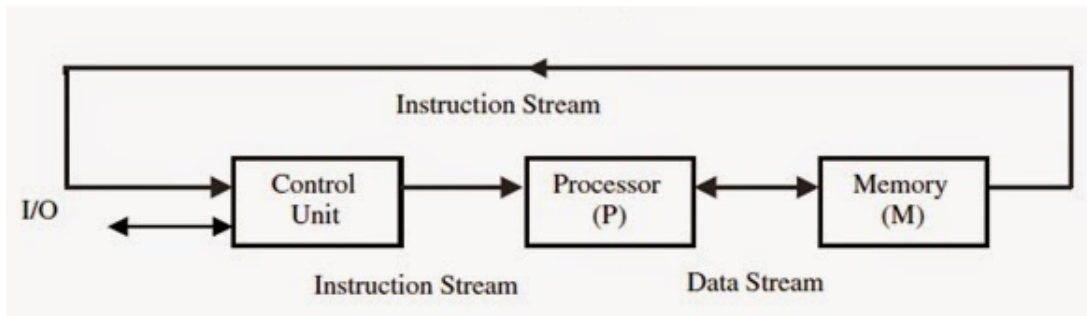


Figura 6: Arquitectura SISD

SIMD En un computador **SIMD** los códigos que genera la única unidad de control a partir de cada instrucción actúan sobre varias unidades de procesamiento distintas. De esta forma, un computador SIMD puede realizar varias operaciones similares simultáneas con operandos distintos. Cada una de las secuencias de operandos y resultados utilizados por las distintas unidades de proceso definen un flujo de datos diferente.

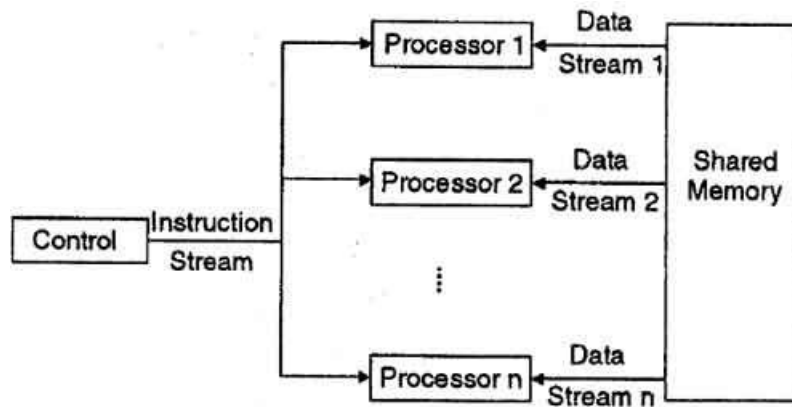


Figura 7: Arquitectura SIMD

MIMD En un computador **MIMD** existen varias unidades de control que decodifican las instrucciones correspondientes a distintos programas. Cada uno de estos programas procesa conjuntos de datos diferentes, que constituyen distintos flujos de datos.

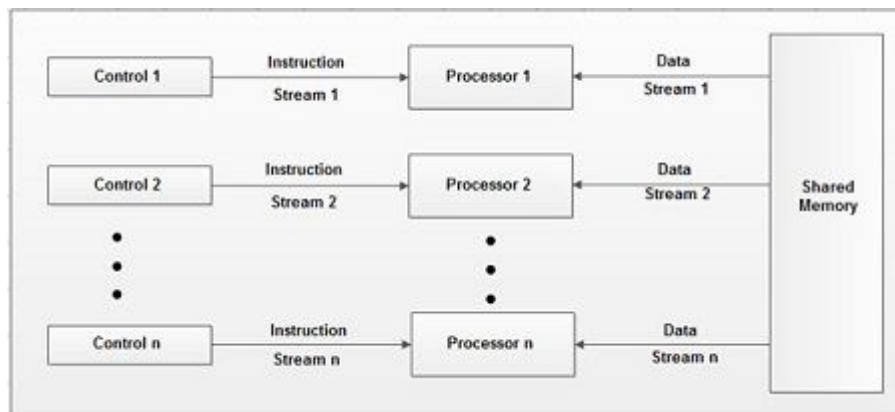


Figura 8: Arquitectura MIMD

MISD Los computadores **MISD** forman una clase de computadores cuyo comportamiento se puede implementar con iguales prestaciones que en un computador MIMD en el que sus procesadores se sincronizan para que los datos vayan pasando desde un procesador a otro. Por tanto, si bien existen computadores SISD (monoprocesador), SIMD (procesadores matriciales y vectoriales) y MIMD (multiprocesadores y multicomputadores), los computadores MISD específicos no existen, ya que su forma de procesamiento se puede implementar en un MIMD.

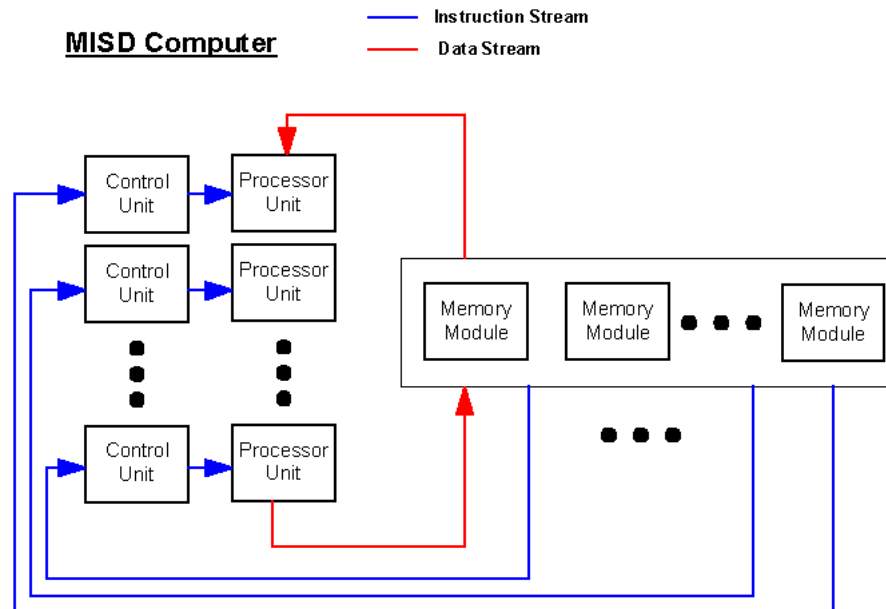


Figura 9: Arquitectura MISD

La taxonomía de Flynn pone de manifiesto dos tipos de paralelismo que pueden utilizarse: paralelismo de *datos* y paralelismo de *instrucciones*. El de *datos* se explota cuando una misma función, instrucción, etc. se ejecuta repetidas veces en paralelo con diferentes datos. Se explota fundamentalmente en las arquitecturas **MIMD**.

El paralelismo *funcional* se aprovecha cuando las funciones, bloques, instrucciones, etc. (iguales o distintas) que intervienen en la aplicación se ejecutan en paralelo.

1.3. Espacio de diseño. Clasificación y estructura general

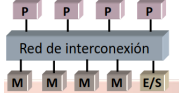
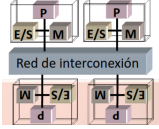
1.3.1. Clasificación

Los sistemas de paralelismo de alto nivel se han clasificado en dos grupos en función de la organización de su espacio de direcciones:

1. **Sistemas con memoria compartida** (SM, *Shared Memory*) o simplemente **multiprocesadores**. Son sistemas en los que todos los procesadores comparten el mismo espacio de direcciones. El programador no necesita saber dónde están almacenados los datos.
2. **Sistemas con memoria distribuida** (DM, *Distributed Memory*) o **multicomputadores**. Cada procesador tiene su propio espacio de direcciones. El programador necesita saber dónde están almacenados los datos.

En un procesador SMP (*Symmetric MultiProcessor*) el tiempo de acceso de los procesadores a memoria o dispositivos de E/S será igual sea cual sea la posición a la que accedan. En estos procesadores, el acceso a memoria se realiza a través de la red de interconexión.

En los multicomputadores, cada procesador tiene su propio módulo de memoria local al que puede acceder directamente. En esta configuración la red de interconexión se utiliza para transferir mensajes entre nodos de la red.

Atributo	Multiprocesador SMP 	Multicomputador 
Espacio de direcciones	Compartido por todos los procesadores	Cada procesador tiene el suyo
Programador	NO necesita saber dónde están almacenados los datos	Necesita saber dónde están almacenados los datos
Latencia en el acceso a memoria	Grande	Pequeña
Comunicación	Implícita mediante variables compartidas. Datos no duplicados	Explícita mediante software para paso de mensajes. Datos duplicados.
Sincronización	Necesita implantar primitivas	Mediante software de comunicación
Distribución de código y datos entre procesadores	No necesaria	Necesaria
Programación	Sencilla	Complicada

Cuadro 1: Diferencias entre multicomputadores y multiprocesadores

Incremento de escalabilidad en multiprocesadores y red de interconexión Se han seguido varios caminos para lograrlo:

1. Incorporar cachés para que cada procesador disponga de una caché local, reduciendo el número de accesos a memoria.

2. Usar redes con menor latencia y mayor ancho de banda que un bus.

a) Jerarquía de buses.

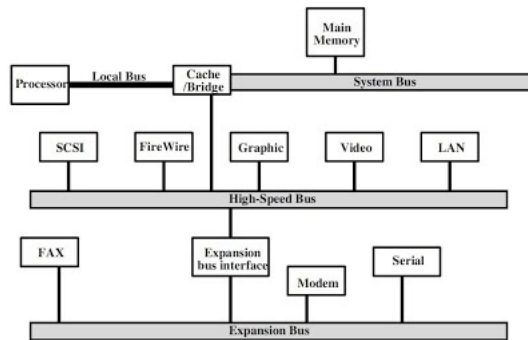


Figura 10: Jerarquía de buses

b) Multietapa

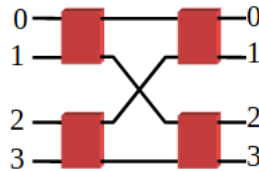


Figura 11: Multietapa

c) Barras cruzadas. Proporciona el mejor rendimiento.

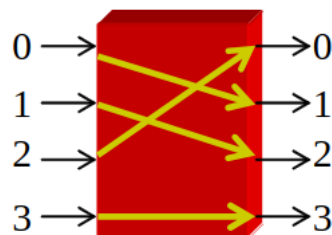


Figura 12: Barras cruzadas

3. Distribuir físicamente los módulos de memoria principal entre los procesadores manteniendo el espacio de direcciones compartido. Se pierde la propiedad de la simetría.

A raíz de la aparición de multiprocesadores con memoria físicamente distribuida surgieron nuevas denominaciones para sistemas con múltiples procesadores. Entonces los multiprocesadores se empezaron a clasificar según la uniformidad en el acceso a memoria:

1. **Multiprocesadores con acceso a memoria uniforme** o UMA (*Uniform Memory Access*). El tiempo de acceso de los procesadores a una determinada posición de memoria principal (o caché) es igual sea cual sea el procesador (SMP).

2. **Multiprocesadores con acceso a memoria no uniforme** o NUMA (*Non-Uniform Memory Access*). El tiempo de acceso depende del procesador.

- a) **NCC-NUMA** (*Non-Cache-Coherent Non-Uniform Memory Access*) o arquitecturas con acceso a memoria no uniforme sin coherencia de caché entre nodos o, simplemente, **NUMA**. En estas arquitecturas no hay hardware para evitar incoherencias entre cachés de distintos nodos. Esto hace que los datos modificables compartidos no se pueden trasladar a caché de nodos remotos, hay que acceder individualmente a ellos a través de la red.
- b) **CC-NUMA** (*Cache-Coherent Non-Uniform Memory Access*) o arquitecturas con acceso a memoria no uniforme y caché coherente. Tienen hardware para mantener la coherencia de caché, pero introduce un retardo que hace que la arquitectura escale en menor grado que un NUMA.
- c) **COMA** (*Cache Only Memory Access*) o arquitecturas con acceso a memoria solo caché. La memoria local se gestiona como caché e incluye un hardware de mantenimiento. El coste y el retardo de estos sistemas es mayor que en CC-Numa, por lo que no existe actualmente ningún sistema comercial COMA.

1.3.2. Propuesta de clasificación de arquitecturas con múltiples threads

- **TLP** (*Thread Level Parallelism*). Múltiples flujos de control concurrentemente o en paralelo.
 - **Implícito**. Flujos de control creados y gestionados por la arquitectura.
 - **Explícito**. Flujos de control creados y gestionados por el Sistema Operativo.
 - **Con una instancia SO**. Multiprocesadores, multicores y cores multithread.
 - **Con múltiples instancias SO**. Multicomputadores.

1.4. Evolución y prestaciones de las arquitecturas

1.4.1. Tiempo de CPU de un programa

Para ilustrar el proceso de evolución de las arquitecturas se utiliza una expresión que relaciona el tiempo de CPU de un programa con tres características a través de las cuales podemos extraer consecuencias importantes relacionadas con la influencia de la tecnología, el compilador y la arquitectura en las prestaciones:

$$T_{tarea} = NI \cdot CPI \cdot T_{ciclo} = NI \cdot \frac{CPI}{f} \quad (1)$$

$$T_{ciclo} = \frac{1}{f}$$

donde T_{tarea} es el tiempo de CPU de un programa (también se puede notar como T_{CPU}), NI es el número de instrucciones máquina del programa, CPI es el número medio de ciclos por instrucción y T_{ciclo} el período de reloj del procesador (inverso de la frecuencia, f). EL valor de CPI se puede expresar como:

$$CPI = \frac{Ciclos_{programa}}{NI} = \frac{\sum_{i=1}^n CPI_i \cdot I_i}{NI} \quad (2)$$

$$Ciclos_{programa} = \sum_i^n CPI_i \cdot I_i$$

donde CPI_i es el número medio de ciclos de las instrucciones de tipo i y NI_i es el número de instrucciones de ese tipo.

Uno de los objetivos fundamentales en el diseño de un computador es reducir el tiempo de ejecución de los programas (T_{tarea}).

La ecuación (1) puede ayudarnos a comprender, por ejemplo, la diferencia entre RISC y CISC:

- En **CISC** se opta por reducir el NI para reducir T_{CPU} , aunque aumenta el CPI_i . Sin embargo, este aumento se puede contrarrestar mejorando la frecuencia del procesador (f).
- En **RISC** se opta por reducir el CPI_i y aumentar el NI_i . Al igual que en el caso anterior, el aumento de NI_i se puede contrarrestar mejorando f .

Hay procesadores que pueden emitir (mandar a ejecutar) varias instrucciones al mismo tiempo. En ese caso:

$$\frac{CPE}{IPE} = CPI$$

$$T_{CPU} = NI \cdot \frac{CPE}{IPE} \cdot T_{ciclo} \quad (3)$$

donde CPE es el número mínimo de ciclos transcurridos entre los instantes en los que el procesador puede emitir instrucciones y IPE es el número de instrucciones que pueden emitirse en un instante.

También hay procesadores que pueden codificar varias operaciones en una instrucción:

$$NI = \frac{N_{operaciones}}{Operaciones_{instruccion}}$$

$$T_{CPU} = \frac{N_{operaciones}}{Operaciones_{instruccion}} \cdot CPI \cdot T_{ciclo} \quad (4)$$

donde $N_{operaciones}$ es el número de operaciones que realiza el programa y $Operaciones_{instruccion}$ el número de operaciones que puede codificar una instrucción.

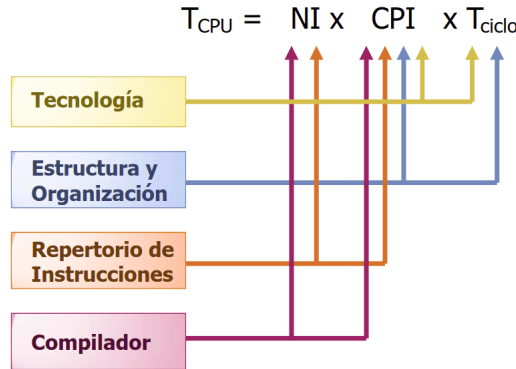


Figura 13: Optimizaciones del tiempo de CPU y sus causantes

1.4.2. Medidas de productividad: MIPS y MFLOPS

El tiempo de respuesta, la productividad y la funcionalidad se pueden englobar en una única medida de prestaciones, conocida como *tiempo de respuesta para una entrada compleja*. Corresponde a una secuencia representativa de entradas elementales del sistema. En esta línea están los *MIPS* y los *MFLOPS*.

MIPS (Millions of Instructions Per Second) Vienen determinados por la siguiente fórmula:

$$\begin{aligned} T_{CPU} &= NI \cdot CPI \cdot T_{ciclo} \\ f &= \frac{1}{T_{ciclo}} \\ MIPS &= \frac{NI}{T_{CPU} \cdot 10^6} = \frac{f}{CPI \cdot 10^6} \end{aligned} \quad (5)$$

Como podemos ver, esta medida puede variar según el programa, por lo que no sirve como medida característica de una máquina. Además, depende del repertorio de instrucciones, por lo que no permite comparar máquinas con repertorios distintos. Por último, puede ser inversamente proporcional a las prestaciones, ya que aquel programa que utilice más instrucciones tendrá más MIPS, indicando erróneamente que es mejor.

MFLOPS (Millions of Floating point Operations Per Second) Vienen definidos por:

$$MFLOPS = \frac{Operaciones_{comaflotante}}{T_{CPU} \cdot 10^6} \quad (6)$$

Por un lado no es una medida adecuada para todos los programas, ya que solo tiene en cuenta las operaciones en coma flotante. Además, el conjunto de operaciones en coma flotante no es el mismo en todas las máquinas, así como el coste por operación. Para resolver este último problema, se utilizan a veces los MFLOPS normalizados, que se obtienen dando un peso relativo a cada instrucción.

1.4.3. Conjuntos de programas de prueba (*benchmarks*)

Para evaluar una arquitectura hay que considerar tanto las medidas de prestaciones que caracterizan a la arquitectura como las medidas de prestaciones que permiten comparar arquitecturas con *igual uso*.

Para ello, se definen conjuntos de programas de prueba (*benchmarks*) representativos de todos los posibles programas, representando la carga de trabajo usual en la máquina que los ejecutará. Hay de varios tipos:

1. **Aplicaciones reales.** Presentan problemas en cuanto a la portabilidad. Ejemplos: SPEC CPU2006
2. **Núcleos o *kernels*.** Programas que evalúan una característica concreta. Resolución de sistemas de ecuaciones, multiplicación de matrices, etc.
3. **Programas de pruebas simples (*toys*).** Son pequeños y fáciles de escribir. Test ping-pong, evaluación de operaciones con enteros y flotantes, etc.
4. **Programas sintéticos.** Dhrystone, Whetstone, etc. No realizan ninguna tarea concreta.

5. Aplicaciones diseñadas. Predicción del tiempo, simulación de terremotos, etc.

1.4.4. Ganancia en prestaciones

Si incrementamos las prestaciones de un computador mejorando alguno de sus recursos o elementos, podremos utilizar la *ganancia* para evaluar hasta qué punto una mejora de prestaciones es p veces mayor o menor que antes.

Este aumento de prestaciones se expresa mediante la ganancia de velocidad (S_p , *Speed Up*)

$$S_p = \frac{T_1}{T_p} = \frac{V_p}{V_1} \quad (7)$$

donde:

- V_1 es la velocidad de la máquina base.
- V_p es la velocidad de la máquina mejorada.
- T_1 es el tiempo de ejecución en la máquina base.
- T_p es el tiempo de ejecución en la máquina mejorada.

Surge la cuestión de hasta qué punto la mejora en un factor p se pone de manifiesto en la mejora final objetiva, surgiendo así la **Ley de Amdahl**.

Ley 1 (de Amdahl). *La mejora de velocidad, S_p que se puede obtener al mejorar un recurso de una máquina en un factor p está limitada por la expresión:*

$$S_p \leq \frac{p}{1 + f \cdot (p - 1)}$$

donde f es el porcentaje de operaciones que realiza la máquina en las que no se pone de manifiesto la mejora. Así, solo si $f = 0$ (la mejora se utiliza siempre) una mejora de factor p en un elemento se observa en la misma medida en la máquina.

Ejemplo Si un programa pasa un 25 % del tiempo de ejecución realizando operaciones de coma flotante y mejoramos la máquina haciendo que esas instrucciones se ejecuten en la mitad de tiempo ($p = 2$);

$$S_p \leq \frac{2}{1 + 0,75 \cdot (2 - 1)} \leq \frac{2}{1,75} \leq 1,14$$

Es decir, la mejora en la arquitectura completa será de sólo un 14 %.