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

Códigos fuente

Índice de figuras

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 ssitema 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

[1] Código 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*.

```
...
a=b+c //[B1 escribe a]
d=a+c //[B2 lee a]
...
```

Listing 1: Dependencia RAW

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

```
...
a=b+c //[B1 escribe a]
a=d+e //[B2 escribe a]
...
```

Listing 2: Dependencia WAW

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

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

Listing 3: Dependencia WAR

Paralelismo funcional Podemos considerar que un programa está compuesto por funciones (nivel de funciones), éstas estñan 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 vasrios 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.

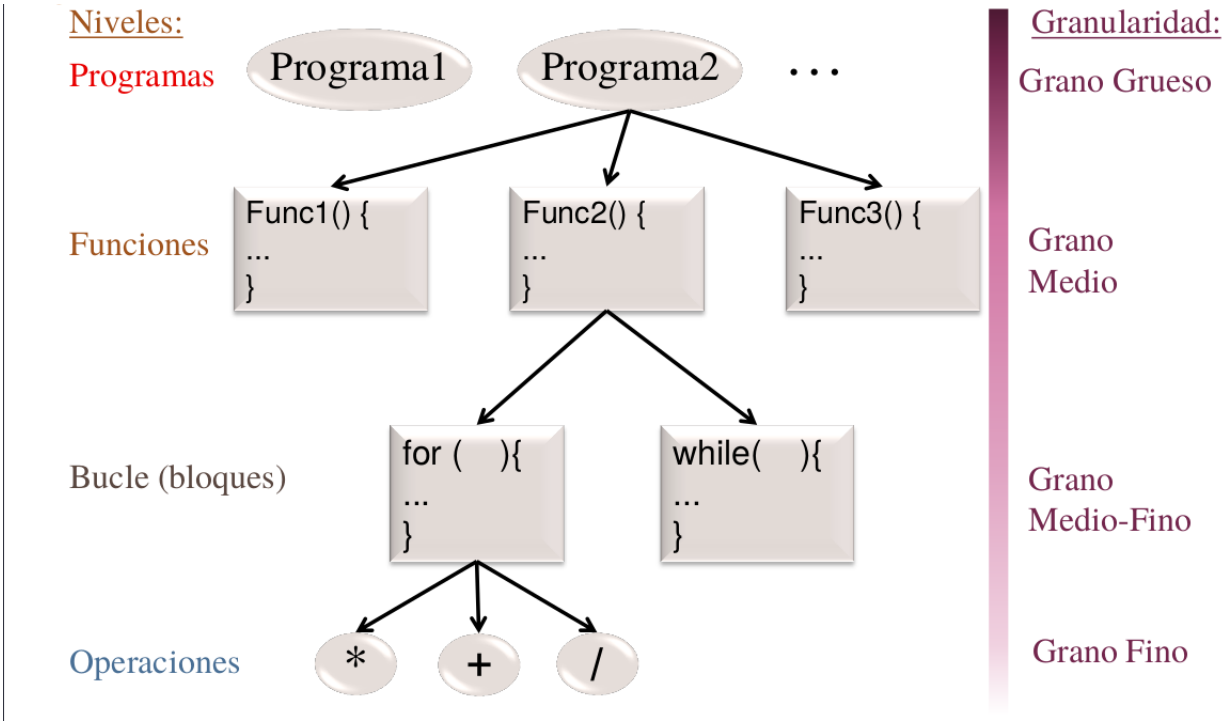


Figura 1: Niveles de paralelismo y granularidad

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

Por ejemplo:

```
#include <stdio.h>
using namespace std;
int func1();
int func2();
int func3();
int main(){
        #pragma omp pararell sections{
                #pragma omp section
                        func1();
                #pragma omp section
                        func2();
                #pragma omp section
                        func3();
                }
}
```

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 estruyctura, 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**.

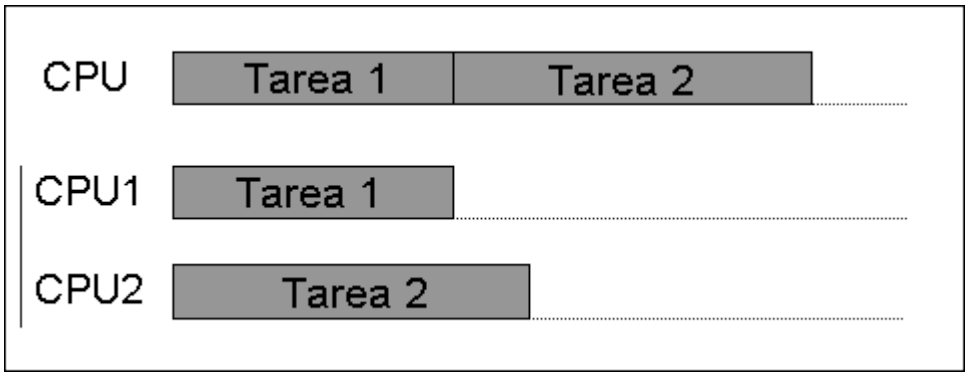


Figura 2: Paralelismo de tareas

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]

[2]Las 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 nuvel 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.

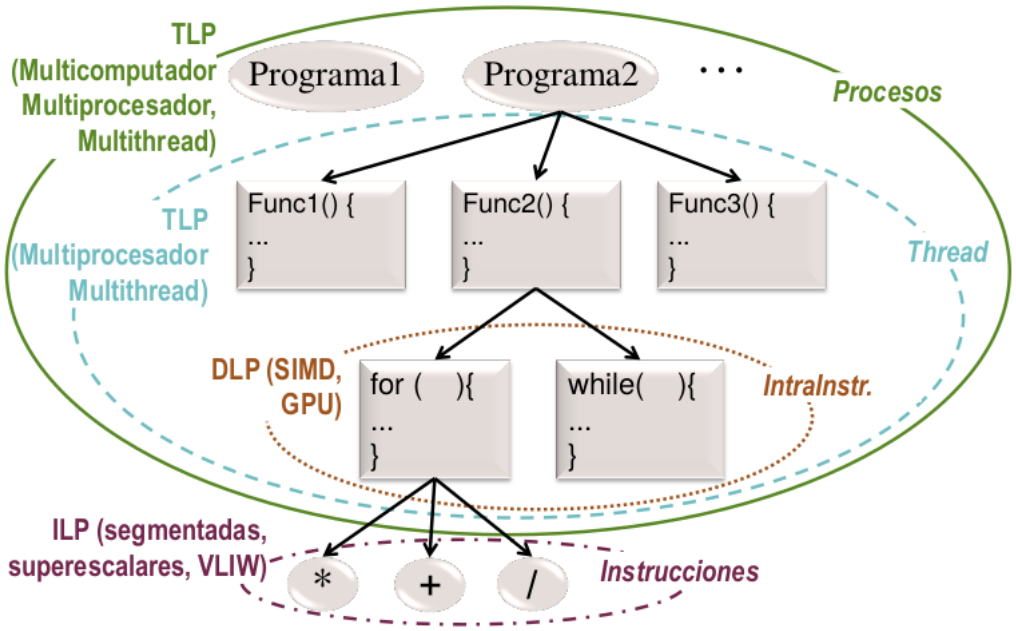


Figura 3: Paralelismo implícito, explícito y arquitecturas paralelas

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 prograamador 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.

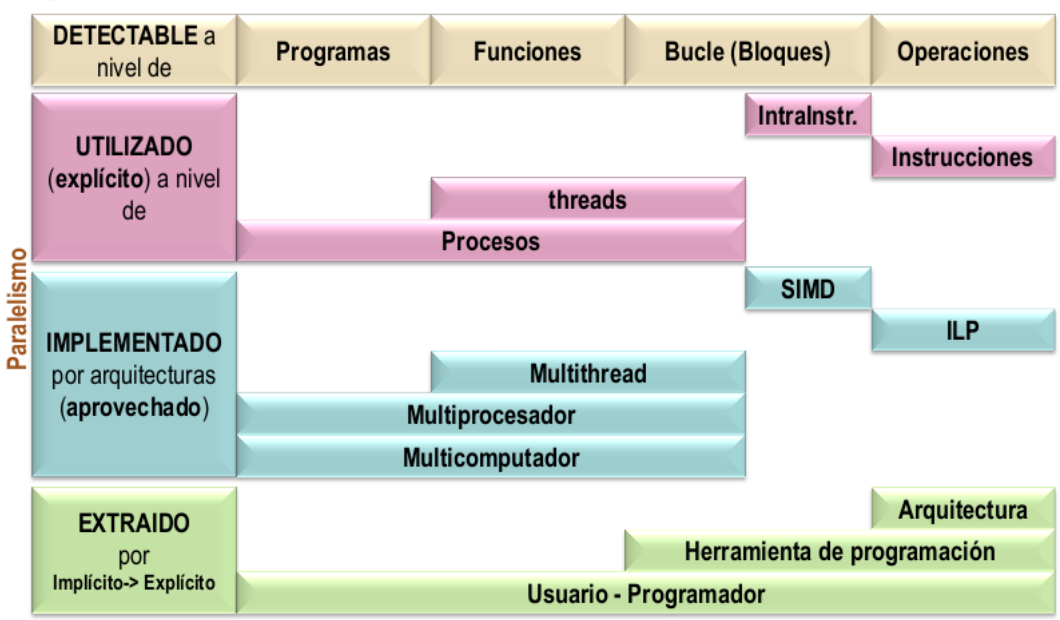


Figura 4: Detección, utilización, implementación y extracción del paralelismo

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.

[3] Instruction 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, defiendo 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.

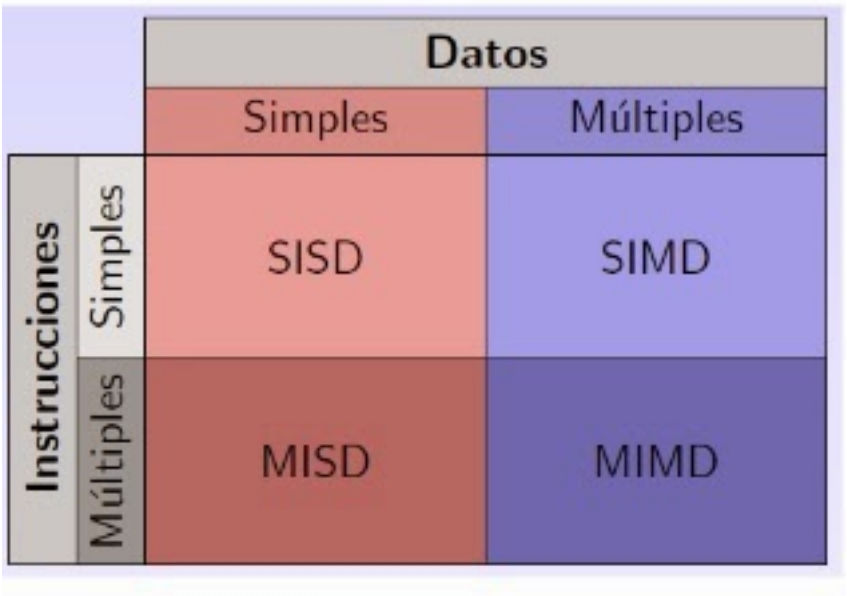


Figura 5: Taxonomía de Flynn

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).

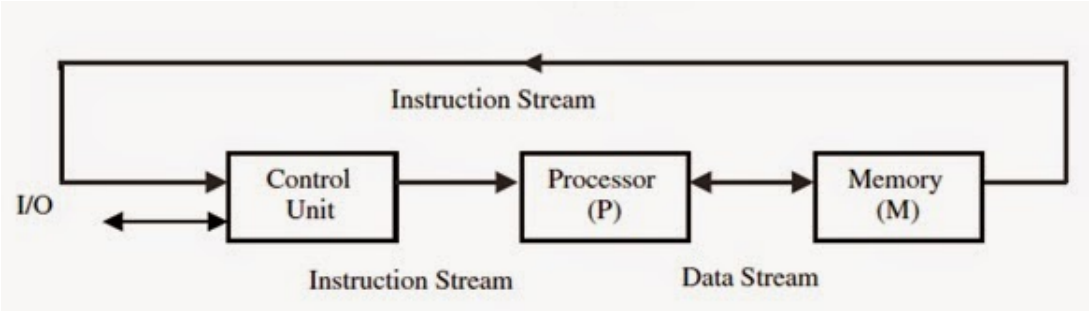


Figura 6: Arquitectura SISD

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.

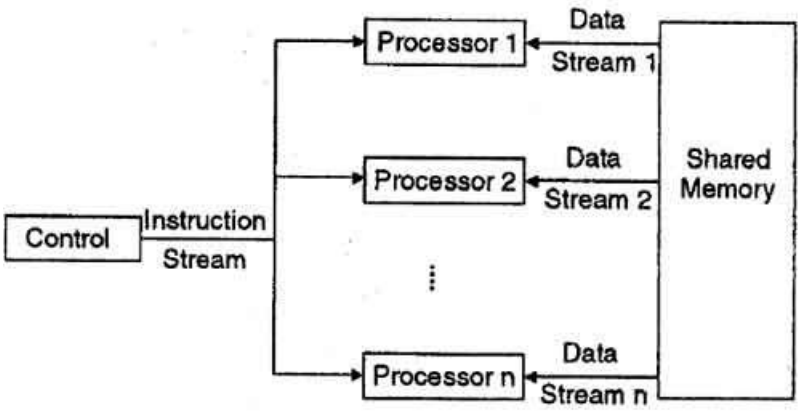


Figura 7: Arquitectura SIMD

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.

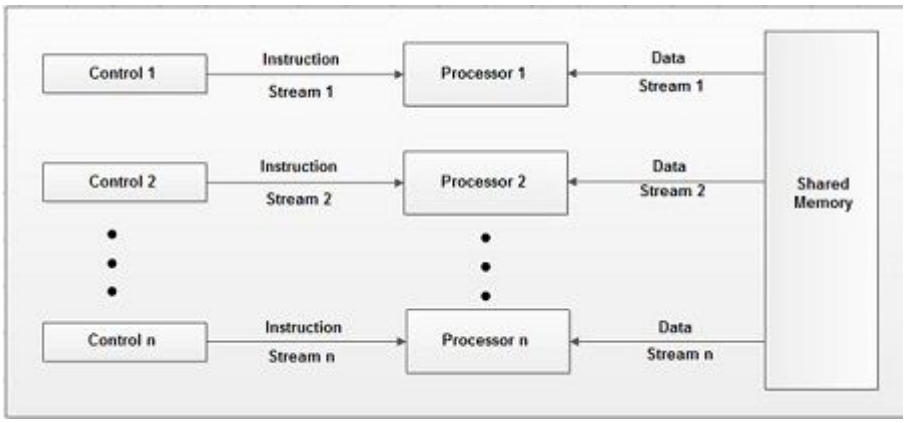


Figura 8: Arquitectura MIMD

MISD Los computadores **MISD** forman una clase de computadores cuyo comportamiento se puede implementear 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.

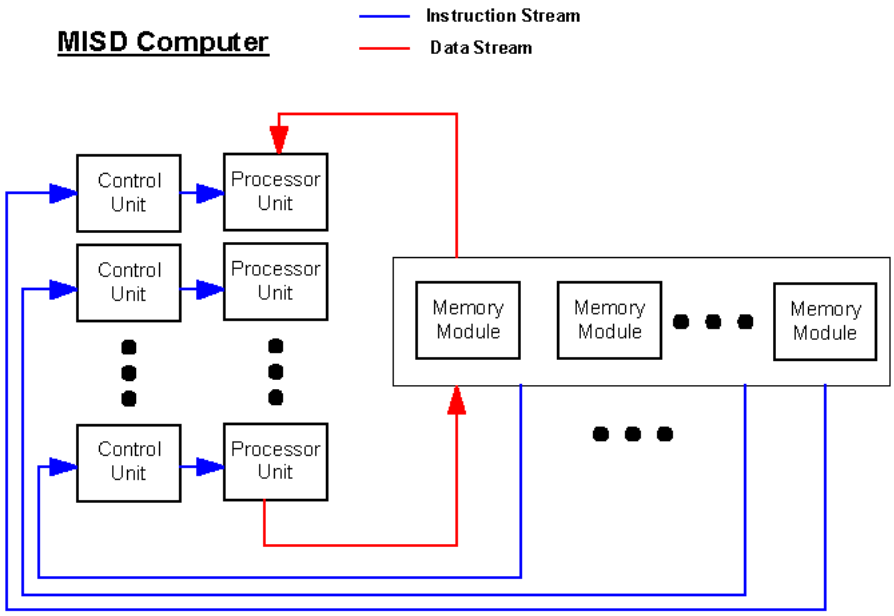


Figura 9: Arquitectura MISD

La taxonomía de Flynn pone de manifiesto dos tipos de paralelismo que pueden utilizarse: paralelismo de *datos* y paralelismo de *instrucciones*. El de *datos* ese 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

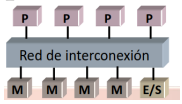
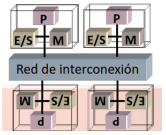
1.3.1. Clasificación

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.

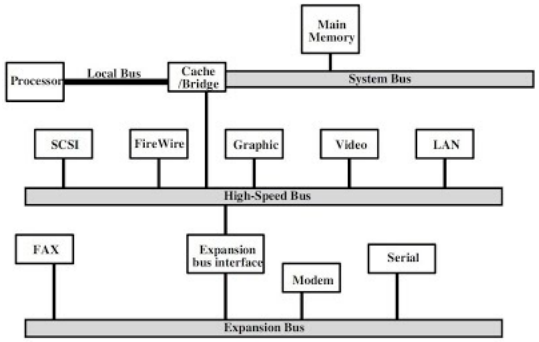


Figura 10: Jerarquía de buses

   b) Multietapa

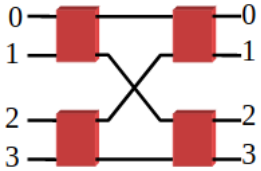


Figura 11: Multietapa

   c) Barras cruzadas. Proporciona el mejor rendimiento.

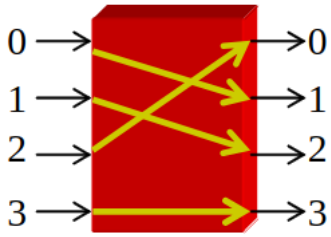


Figura 12: Barras cruzadas

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:

$$T_{tarea} = NI \cdot CPI \cdot T_{ciclo} = NI \cdot \frac{CPI}{f} \tag{1}$$

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

donde T_{tarea} es el tiempo de CPU de un programa (también se puede notar como T_{CPU}), NI es el número de instrucciones mñaquina 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:

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

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

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:

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

$$T_{CPU} = NI \cdot \frac{CPE}{IPE} \cdot T_{ciclo} \tag{3}$$

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:

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

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

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

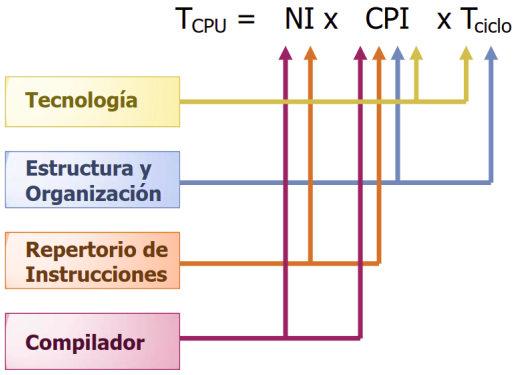


Figura 13: Optimizaciones del tiempo de CPU y sus causantes

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:

$$T_{CPU} = NI \cdot CPI \cdot T_{ciclo}$$

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

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{f}{CPI \cdot 10^6} \tag{5}$$

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.

***MIPS*$_{pico}$** Es el máximo valor téórico de MIPS para una arquitectura. Se calcula mediante la siguiente fórmula:

$$MIPS_{pico} = \frac{f \cdot IPC_{max}}{10^6} \tag{6}$$

donde IPC_{max} representa el número máximo de operaciones por ciclo que puede realizar la arquitectura.

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

$$MFLOPS = \frac{Operaciones_{comaflotante}}{T_{CPU} \cdot 10^6} \tag{7}$$

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.

***MFLOPS*$_{pico}$** Se calculan mediante la siguiente expresión:

$$MFLOPS_{pico} = \frac{f \cdot \text{Operaciones en coma flotante/ciclo}_{max}}{10^6} \tag{8}$$

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 posinlrd 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*)

$$S_p = \frac{T_1}{T_p} = \frac{V_p}{V_1} \tag{9}$$

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 (por supuesto, $0 \leq f \leq 1$). Así, solo si $f = 0$ (la mejora se utiliza siempre) una mejora de factor p en un elemento se observa n 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 %.

1.5. Ejercicios

1. Calcule los $MIPS_{pico}$ de una arquitectura con una CPU superescalar con un IPC de 3 instrucciones por ciclo y una frecuencia de 2Ghz.

   Solución

$$MIPS_{pico} = \frac{2 \cdot 10^9 ciclos/seg \cdot 3 instrucciones/ciclo}{10^6} = 6 \cdot 10^3 \ MIPS \simeq 6 \ GIPS$$

2. En el código de prueba (benchmark) que ejecuta un procesador no segmentado que funciona a 300 MHz, hay un 20 % de instrucciones LOAD que necesitan 4 ciclos, un 10 % de instrucciones STORE que necesitan 3 ciclos, un 25 % de instrucciones con operaciones de enteros que necesitan 6 ciclos, un 15 % de instrucciones con operandos en coma flotante que necesitan 8 ciclos por instrucción, y un 30 % de instrucciones de salto que necesitan 3 ciclos.

   a) ¿Cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones con enteros?

   b) ¿Cuál es la ganancia que se puede obtener por reducción a 3 ciclos de las instrucciones en coma flotante?

   Solución

Instrucción	**Porcentaje de uso**	**CPI**
LOAD	20 %	4
STORE	10 %	3
Operaciones con enteros	25 %	6
Operaciones en coma flotante	15 %	8
Salto	30 %	3

Antes de la optimización

- Apartado a)

Instrucción	Porcentaje de uso	CPI
LOAD	20 %	4
STORE	10 %	3
Operaciones con enteros	25 %	~~6~~ 3
Operaciones en coma flotante	15 %	8
Salto	30 %	3

Tras la optimización

Como el CPI de las operaciones con enteros se reduce a la mitad, $p = 2$. Aplicamos la ley de Amdahl:

$$S_p = \frac{p}{1 + f \cdot (p-1)} \rightarrow S_p = \frac{2}{1 + 0{,}75 \cdot (2-1)} = \frac{2}{1{,}75} \simeq 1{,}143$$

Por tanto la mejora será de aproximadamente el $14,3\,\%$

- Apartado b)

Instrucción	Porcentaje de uso	CPI
LOAD	20 %	4
STORE	10 %	3
Operaciones con enteros	25 %	6
Operaciones en coma flotante	15 %	~~8~~ 3
Salto	30 %	3

Tras la optimización

Como el CPI de las operaciones en coma flotante pasa de 8 a 3, $p = \frac{8}{3} = 2{,}67$. Aplicamos la Ley de Amdahl

$$S_p = \frac{p}{1 + f \cdot (p-1)} \rightarrow S_p = \frac{2{,}67}{1 + 0{,}85 \cdot (2{,}67-1)} = \frac{2,67}{1{,}4195} \simeq 1{,}88$$

Por tanto, la mejora es aproximadamente del 88 %

3. En un procesador sin segmentación de cauce, determine cuál de estas dos alternativas para realizar un salto condicionales mejor:

   a) Una instrucción COMPARE actualiza un código de condición y es seguida por una instrucción BRANCH que comprueba esa condición.Se usan dos intrucciones.

Instrucción	NI	CPI
COMPARE	$0{,}2NI_1$	4
Resto	$0{,}8NI_1$	3
	NI_1	

b) Una sola instrucción incluye la funcionalidad de las instrucciones COMPARE y BRANCH.Se usa una única instrucción.

Instrucción	**NI**	**CPI**
COMPARE + BRANCH	$0{,}2NI_1$	4
Resto	$0{,}8NI_1 - 0{,}2NI_1 = 0{,}6NI_1$	3
	$0{,}8NI_1$	

Hay que tener en cuenta que hay un 20 % de instrucciones BRANCH para a) en el conjunto de programas de prueba; que las instrucciones BRANCH en a) y COMPARE+BRANCH en b) necesitan 4 ciclos mientras que todas las demás necesitan sólo 3; y que el ciclo de reloj de la a) es un 25 % menor que el de la b), dado que en éste caso la mayor funcionalidad de la instrucción COMPARE+BRANCH ocasiona una mayor complejidad en el procesado.

Solución

Considero que en el apartado a) la instrucción BRANCH entra en el resto, por lo que consume 3 CPI.

Sabemos que $T_{ciclo1} = 0{,}75 \cdot T_{ciclo2} \rightarrow T_{ciclo2} = 1{,}33 \cdot T_{ciclo1}$ Ahora calculamos los T_{CPU}. Para ello, comenzamos por los CPI_i

- $CPI_1 = \frac{0{,}2 \cdot NI_1}{NI_1} \cdot 4 + \frac{0{,}8 \cdot NI_1}{NI_1} \cdot 3 = 3{,}2$
- $T_{CPU1} = NI_1 \cdot CPI_1 \cdot T_{ciclo1} = 3{,}2 \cdot NI_1 \cdot T_{ciclo1}$
- $CPI_2 = \frac{0{,}2 \cdot NI_1}{0{,}8 \cdot NI_1} \cdot 4 + \frac{0{,}6 \cdot NI_1}{0{,}8 \cdot NI_1} \cdot 3 = 3{,}25$
- $T_{CPU2} = 0{,}8 \cdot NI_1 \cdot 3{,}25 \cdot 1{,}33 \cdot T_{ciclo1} = 3{,}45 \cdot NI_1 \cdot T_{ciclo1}$

Como $T_{CPU1} < T_{CPU2}$, la mejor opción es la **a)**.

4. ¿Qué ocurriría en el ejercicio anterior si el ciclo de reloj fuese únicamente un 10 % mayor para b) ?

Solución

El problema nos indica que $T_{ciclo2} = 1{,}1T_{ciclo1}$

$$T_{CPU1} = 3{,}2 \cdot NI_1 \cdot T_{ciclo1}$$

$$T_{CPU2} = (0{,}8 \cdot 3{,}25 \cdot 1{,}1)NI_1 \cdot T_{ciclo1} = 2{,}86 \cdot NI_1 \cdot T_{ciclo1}$$

En este caso, $T_{ciclo2} < T_{ciclo1}$, por lo que la opción **b)** se convierte en la mejor.

5. Considere un procesador no segmentado con una arquitectura de tipo LOAD/STORE en la que las operaciones sólo utilizan como operandos registros de la CPU. Paraun conjunto de programas representativos de su actividad se tiene que el 43 % de las instrucciones son operaciones con la ALU (3 CPI), el 21 % LOADs (4 CPI), el 12 % STOREs (4 CPI) y el 24 % BRANCHs (4 CPI).Se ha podido comprobar que un 25 % de las operaciones con la ALU utilizan operandos en registros que no se vuelven a utilizar. Compruebe si mejorarían las prestaciones si, para sustituir ese

25 % de operaciones,se añaden instrucciones con un dato en un registro y otro en memoria. Tengan en cuenta en la comprobación que para estas nuevas instrucciones el valor de CPI es 4 y que añadirlas ocasiona un incremento de un ciclo en el CPI de los BRANCHs, pero no afectan al ciclo de reloj.

Instrucción	**NI**	**CPI**
ALU (R-R)	$0{,}43NI_1$	3
LOAD	$0{,}21NI_1$	4
STORE	$0{,}12NI_1$	4
BRANCH	$0{,}24NI_1$	4
	NI_1	

Antes de la mejora

Solución

Instrucción	**NI**	**CPI**
ALU (R-R)	$0{,}75 \cdot 0{,}43NI_1 = 0{,}3225NI_1$	3
ALU (R-M)	$0{,}25 \cdot 0{,}43NI_1 = 0{,}1075NI_1$	4
LOAD	$0{,}21NI_1 - 0{,}25 \cdot 0{,}43NI_1 = 0{,}1025NI_1$	4
STORE	$0{,}12NI_1$	4
BRANCH	$0{,}24NI_1$	4 5
	$0{,}8925NI_1$	

Tras la mejora

Calulamos el tiempo de CPU de ambas situaciones:

$$CPI_1 = \frac{0{,}43NI_1}{NI_1} \cdot 3 + \frac{0{,}21NI_1}{NI_1} \cdot 4 + \frac{0{,}12NI_1}{NI_1} \cdot 4 + \frac{0{,}24NI_1}{NI_1} \cdot 4 = 3{,}57$$

$$T_{CPU1} = CPI_1 \cdot NI_1 \cdot T_{ciclo1} = 3{,}57NI_1 \cdot T_{ciclo1}$$

$$CPI_2 = \frac{0{,}3225NI_1}{0{,}8925NI_1} \cdot 3 + \frac{0{,}1075NI_1}{0{,}8925NI_1} \cdot 4 + \frac{0{,}21NI_1}{0{,}8925NI_1} \cdot 4 + \frac{0{,}12NI_1}{0{,}8925NI_1} \cdot 4 + \frac{0{,}24NI_1}{0{,}8925NI_1} \cdot 4 = 3{,}91$$

$$T_{CPU2} = 3{,}91 \cdot 0{,}8925NI_1 \cdot T_{ciclo2} = 3{,}49NI_1 \cdot T_{ciclo1}$$

Como $T_{CPU2} < T_{CPU1}$; la mejora **es efectiva**.

6. Se ha diseñado un compilador para la máquina LOAD/STORE del problema anterior. Ese compilador puede reducir en un 50 % el número de operaciones con la ALU, pero no reduce el número de LOADs, STOREs y BRANCHs. Suponiendo que la frecuencia de reloj es de 50 Mhz. ¿Cuál es el número de MIPS y el tiempo de ejecución que se consigue con el código optimizado? Compárelos con los correspondientes del código no optimizado.

Instrucción	**NI**	**CPI**
ALU (R-R)	$0{,}43NI_1$	3
LOAD	$0{,}21NI_1$	4
STORE	$0{,}12NI_1$	4
BRANCH	$0{,}24NI_1$	4
	NI_1	

Antes de la mejora

Solución

Instrucción	**NI**	**CPI**
ALU (R-R)	$\frac{0{,}43NI_1}{2}$	3
LOAD	$0{,}21NI_1$	4
STORE	$0{,}12NI_1$	4
BRANCH	$0{,}24NI_1$	4
	$0{,}785NI_1$	

Tras la mejora

Comenzamos calculando el Tiempo de CPU

$$CPI_1 = \frac{0{,}43NI_1}{NI_1} \cdot 3 + \frac{0{,}21NI_1}{NI_1} \cdot 4 + \frac{0{,}12NI_1}{NI_1} \cdot 4 + \frac{0{,}24NI_1}{NI_1} \cdot 4 = 3{,}57$$

$$CPI_2 = \frac{\frac{0{,}43NI_1}{2}}{0{,}785NI_1} \cdot 3 + \frac{0{,}21NI_1}{0{,}785NI_1} \cdot 4 + \frac{0{,}12NI_1}{0{,}785NI_1} \cdot 4 + \frac{0{,}24NI_1}{0{,}785NI_1} \cdot 4 = 3{,}726$$

$$T_{CPU1} = NI_1 \cdot T_{ciclo} \cdot CPI_1 = 3{,}57 \cdot NI_1 \cdot T_{ciclo}$$

$$T_{CPU2} = 0{,}785NI_1 \cdot CPI_2 \cdot T_{ciclo} = 0{,}785NI_1 \cdot 3{,}726 \cdot T_{ciclo} = 2{,}925 \cdot T_{ciclo}$$

Ahora, calculamos MIPS

$$MIPS_1 = \frac{NI_1}{T_{CPU1} \cdot 10^6} = \frac{NI_1}{NI_1 \cdot CPI_1 \cdot T_{ciclo} \cdot 10^6} = \frac{f}{CPI_1 \cdot 10^6} = \frac{50 ciclos/segundo \cdot 10^6}{3{,}57 ciclos/instruccion \cdot 10^6} = 14{,}005 MIPS$$

$$MIPS_2 = \frac{0{,}785NI_1}{T_{CPU2} \cdot 10^6} = \frac{NI_1}{0{,}785NI_1 \cdot CPI_2 \cdot T_{ciclo} \cdot 10^6} = \frac{f}{CPI_2 \cdot 10^6} = \frac{50 ciclos/segundo \cdot 10^6}{3{,}726 ciclos/instruccion \cdot 10^6} = 13{,}42 MIPS$$

La configuración 2 tiene un tiempo de CPU **mejor**. Sin embargo, si miramos la productividad, vemos como es **peor**. Esto se debe a que el sistema es más rápido, pero menos productivo, ya que la mejora se ha implementado en la parte que menos se usa.

2. Tema 2

2.1. Programación paralela

La programación paralela introduce un conjunto de problemas para el programador: división del trabajo en unidades independientes (tareas), sincronización, comunicación, etc.

Actualmente, las herramientas y métodos para facilitar el desarrollo de aplicaciones paralelas existentes son un activo campo de investigación. Para el programador lo más sencillo es utilizar compiladores capaces de extraer paralelismo automáticamente. Sin embargo, estos compiladores no generan código eficiente para todos los programas. Por tanto, no sirven de mucho.

2.1.1. Punto de partida

Cuando se plantea obtener una versión paralela de una aplicación, podemos utilizar como punto inicial un código secuencial que resuelva el problema para implantar el paralelismo sobre él. La principal ventaja de esta opción es la posibilidad de medir el tiempo en cada sección, permitiendo distribuir el trabajo de forma equilibrada.

Otra posibilidad es partir de la definición de una aplicación y buscar a partir de ella una descripción que admita paralelización.

Para facilitar el trabajo podemos apoyarnos en programas paralelos que aprovechen las características de la arquitectura. Si disponemos de un programa paralelo que resuelve un problema parecido en una arquitectura, podemos fijarnos en él para diseñar el nuestro.

2.1.2. Modos de programación

SPMD (paralelismo de datos) Todos los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa. Cada copia trabaja con un conjunto de datos distintos y se ejecuta en un procesador diferente.

MPMD Los códigos que se ejecutan en paralelo se obtienen compilando programas independientes, es decir, la aplicación principal se divide e unidades independientes. Cada unidad trabaja con un conjunto de ddatos y es asignada a un procesador.

SPMD es recomendable en sistemas masivamente paralelos, ya que es muy difícil encontrar cientos de unidades de código diferentes dentro de una aplicación, siendo más fácil escribir un sólo programa. En la práctica es el más utilizado en multiprocesadores y multicomputadores.

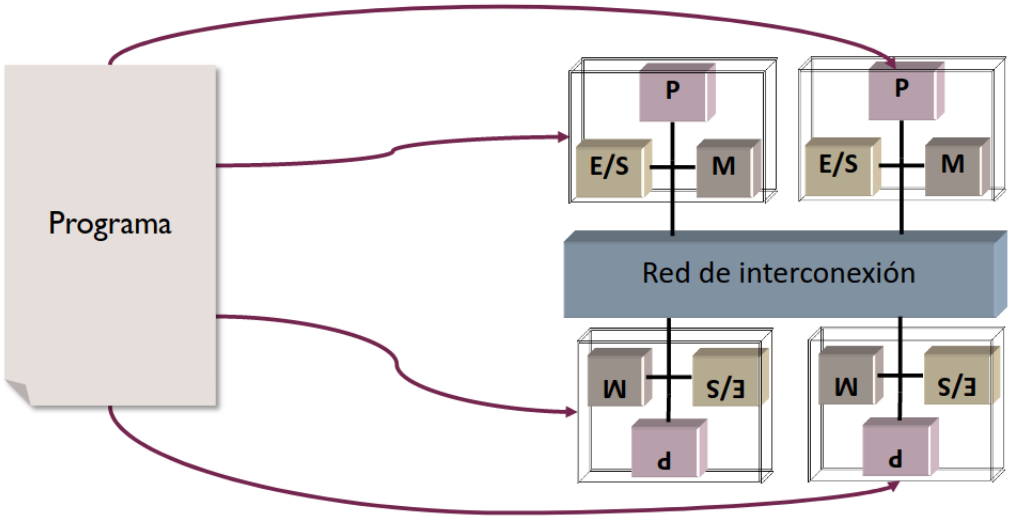


Figura 14: SPMD (Single Program Multiple Data).

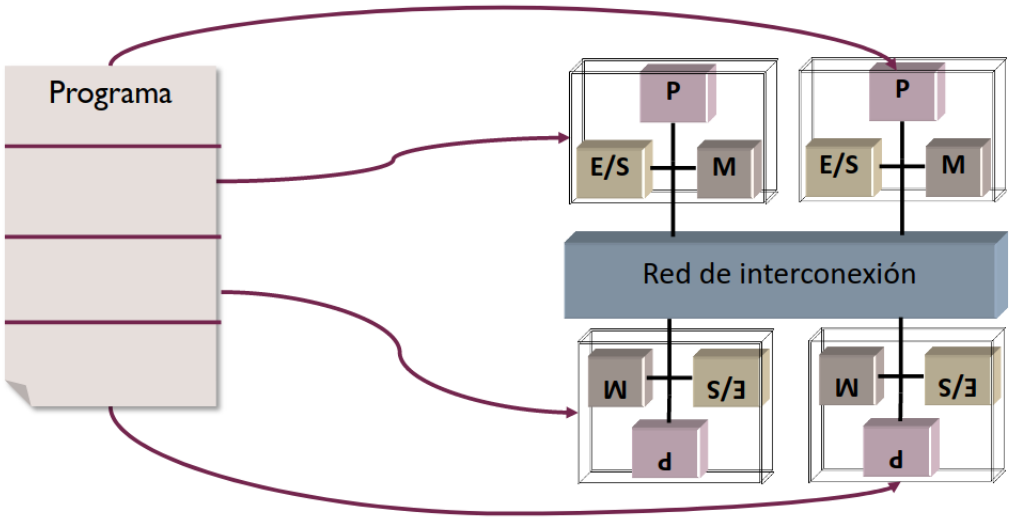


Figura 15: MPMD (Multiple Program Multiple Data).

2.1.3. Herramientas para obtener programas paralelos

Las herramientas para obtener programas paralelos deben permitir de forma explícita (el trabajo lo hace el programador) o implícita (el trabajo lo hace la propia herramienta) las siguientes tareas:

- Localizar paralelismo.
- Crear y terminar procesos.
- Distribuir trabajo entre procesos.
- Comunicación y sincronización entre procesos.

- Asignación de procesos a procesadores.

Para obtener un programa paralelo tenemos varias opciones:

Bibliotecas de funciones para programación paralela En esta alternativa el programador utiliza un lenguaje secuencial y una biblioteca de funciones. El cuerpo de los procesos y hebras se escribe con lenguaje secuencial y el programador se encarga explícitamente de dividir las tareas entre los procesos, crear o destruir los procesos, implementar la comunicación, etc. Las principales ventajas de esta alternativa son:

- Los programadores no tienen que aprender un nuevo lenguaje.
- Las bibliotecas están disponibles para todos los sistemas paralelos.
- Las bibliotecas están más cercanas al hardware y dan al programador un control a más bajo nivel.
- Se pueden utilizar a la vez bibliotecas para programar con hebras y bibliotecas para programar con procesos.

Las APIs más famosas son MPI, OpenMP, Pthread, etc.

Lenguajes paralelos y directivas del compilador Sitúan al programador en un nivel de abstracción superior, ahorrando o facilitando el trabajo de paralelización, aunque puede que sólo se aproveche uno de ellos: de datos o de tareas. Los lenguajes paralelos facilitan estas tareas mediante:

- Construcciones propias del lenguaje. Pueden tanto distribuir la carga de trabajo como crear y terminar procesos e incluir sincronización.
- Directivas del compilador.
- Funciones de biblioteca. Implementan en paralelo algunas operaciones usuales.

La ventaja principal de los lenguajes paralelos es que son más fáciles de escribir y entender a la vez que más cortos.

Compiladores paralelos Se pretende que un compilador paralelo extraiga automáticamente el paralelismo tanto a nivel de bucle (paralelismo de datos) como a nivel de función (paralelismo de tareas). Para ello, hacen análisis de dependencias entre bloques de código, iteraciones de un bucle o funciones. Las dependencias que detecta son RAW, WAW y WAR.

Los compiladores paralelos están aún limitados a aplicaciones que exhiben un paralelismo regular, como los cálculos a nivel de bucle.

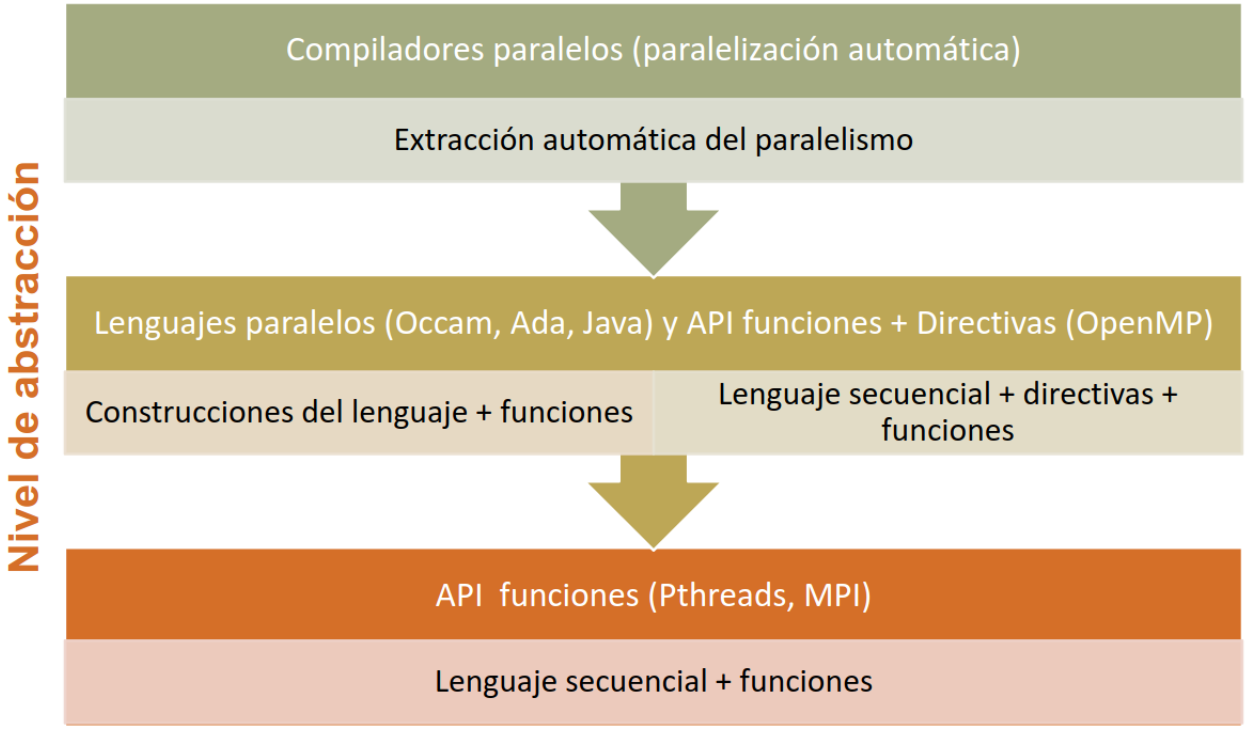


Figura 16: Principales herramientas de programación paralela.

Otras alternativas

Comunicación múltiple uno a uno Hay componentes del grupo que envían un único mensaje (dato o estructura de datos) y componentes que reciben un único lenguaje.

Si todos los componentes envían y reciben, se implementa una *permutación*. Algunos ejemplos de permutaciones son la rotación, el intercambio, los barajes, etc.

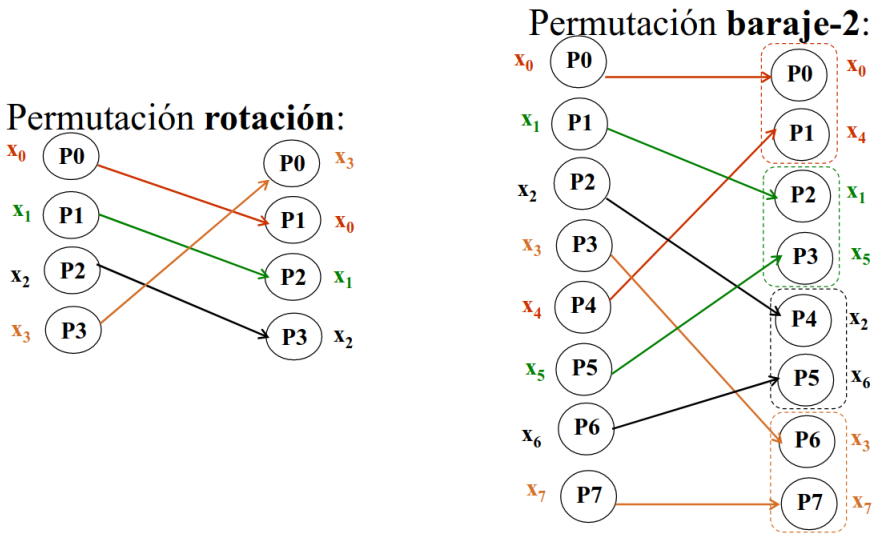


Figura 17: Comunicación uno a uno.

Comunicación uno a todos Un proceso envía u todos los procesos reciben. Hay variantes en las que el proceso que envía no forma parte del grupo y otras en las que reciben todos excepto el que envía. Hay dos subtipos:

- **Difusión**. Todos los procesos reciben el mismo mensaje.
- **Dispersión (*scatter*)**. Cada proceso receptor recibe un mensaje diferente.

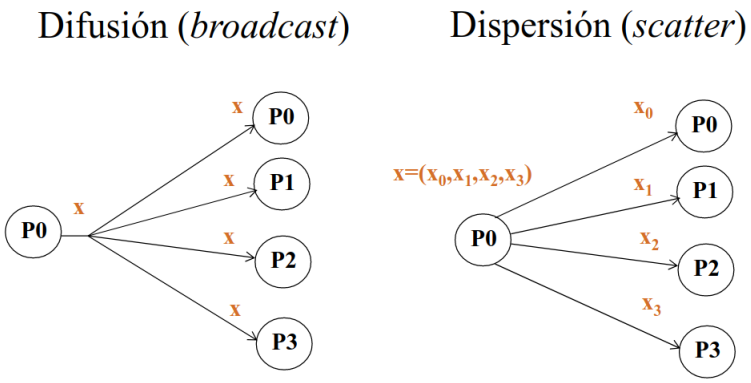


Figura 18: Comunicación uno a todos.

Comunicación todos a uno Todos los procesos del grupo envían un mensaje a un único proceso.

- **Reducción**. Los mensajes enviados por los procesos se combinan en un solo mensaje mediante algún operador. La combinación es normalmente conmutativa y asociativa.
- **Acumulación (*gather*)**. Los mensajes se reciben de forma concatenada en el receptor. El orden en que se concatenan depende normalmente del identificador de proceso.

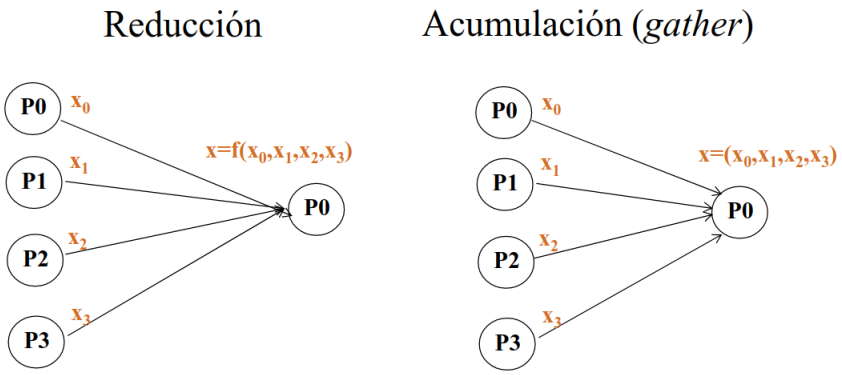


Figura 19: Comunicación todos a uno.

Comunicación todos a todos Todos los procesos del grupo ejecutan una comunicación *uno a todos*. Cada proceso recibe n mensajes, cada uno de un proceso diferente del grupo.

- **Todos difunden (*all-broadcast*)**. Todos los procesos realizan una difusión. Normalmente las transferencias recibidas por un proceso se concatenan según el identificador de proceso.
- **Todos dispersan (*all-scatter*)**. Los procesos concatenan diferentes transferencias. En el ejemplo se muestra una trasposición de una matriz 4x4. Cada procesador P_i dispersa la fila $i(x_{i0}, x_{i1}, ...)$. Tras la ejecución, P_i tendrá la columna $i(x_{0i}, x_{1i}, ...)$.

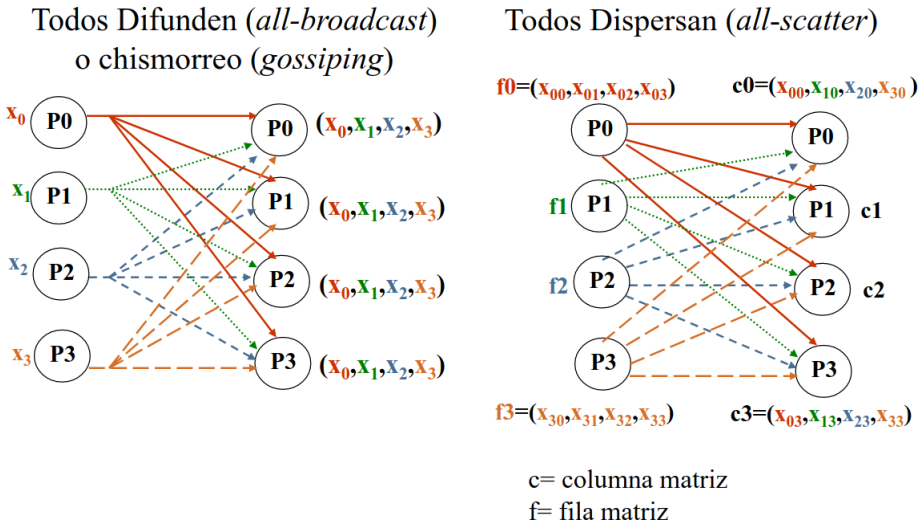


Figura 20: Comunicación todos a todos.

Comunicaciones colectivas compuestas Las comunicaciones anteriores se pueden combinar dando lugar a nuevos servicios:

- **Todos combinan o reducción y extensión**. Se aplica una reducción a todos los procesos, ya sea difundiéndola una vez obtenida (reducción y extensión) o bien realizándola en todos los procesos (todos combinan).

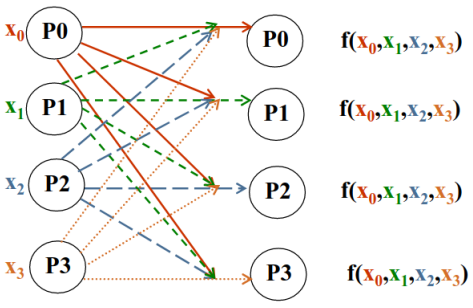


Figura 21: Todos combinan.

- **Barrera**. Es un punto de sincronización que todos los procesos de un grupo deben alcanzar para poder continuar su ejecución. Se puede implementar mediante cerrojos o a nivel software.
- **Recorrido (*scan*)**. Todos los procesos envían un mensaje, recibiendo cada uno el resultado de reducir un conjunto de esos mensajes.

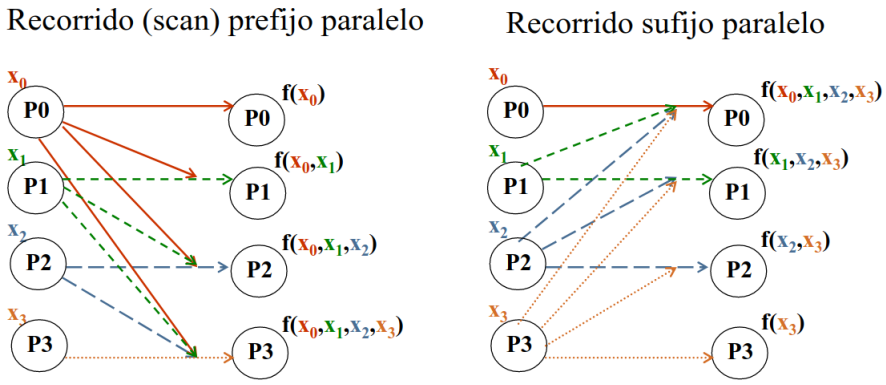


Figura 22: Recorrido (*scan*).

2.1.4. Estilos de programación

Paso de mensajes Disponemos de dos funciones principales:

- **send(destino,datos)**. Envía datos.
- **receive(fuente,datos)**. Recibe datos.

Por lo general podemos encontrar transmisiones *síncronas* (cuando ejecutamos un *send*, el proceso se bloquea hasta que el destino recibe el dato y viceversa con *receive*) o *asíncronas* (*send* no bloquea, por lo que suele hacerlo *receive*).

La interfaz más conocida de paso de mensajes es MPI.

Variables compartidas La comunicación entre procesos se realiza accedeiendo a variables compartidas, es decir, mediante accessos y escrituras en memoria. Las hebras de un proceso creadas por el sistema operativo pueden compartir inmediatamente variables globales, pero los procesos no (tienen diferentes espacios de direcciones). En este caso, hemos de utilizar llamadas al sistema específicas.

La exclusión mutua se puede implementar mediante cerrojos, semáforos, variables condicionales, monitores, etc.

La interfaz más utilizada es OpenMP. Hay lenguajes, como Java, que implementan este paradigma.

Paralelismo de datos En este estilo se aprovecha el paralelismo de datos inherente a aplicaciones en las que los datos se organizan en estructuras (vectores o matrices). El programador escribe un programa con construcciones que permiten aprovechar el paralelismo: construcciones para paralelizar bucles, para distribuir datos, etc. Por tanto, no ha de ocuparse de las sincronizaciones, ya que son implícitas.

El lenguaje con paralelismo de datos más conocido es C* (*C star*). En cuanto a APIs, destaca *Nvidia CUDA*.

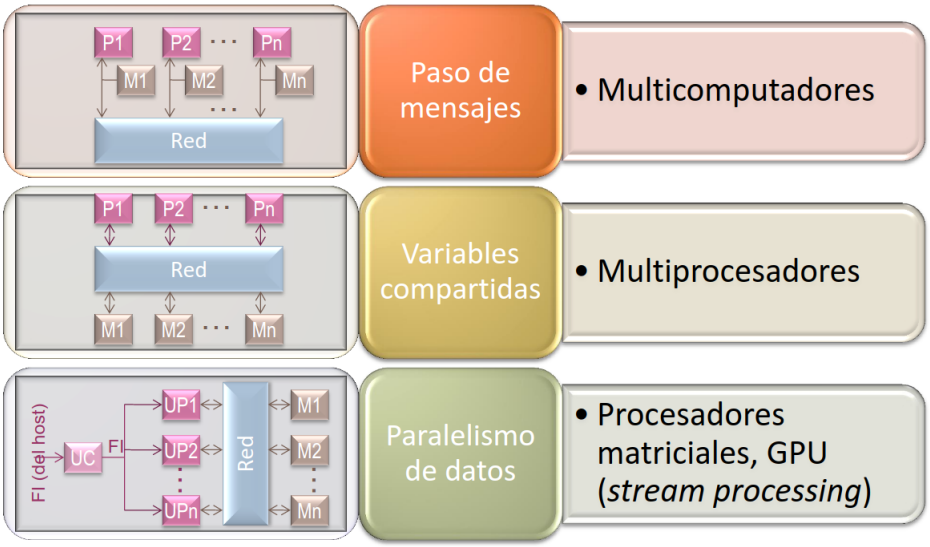


Figura 23: Estilos de programación paralela.

2.1.5. Estructuras de programas paralelos

Dueño-esclavo (*master-slave*) o granja de tareas (*task-farming*) Consta de un dueño y varios esclavos. El dueño se encarga de distribuir las tareas de un conjunto (granja) entre el grupo de esclavos y de ir recogiendo los resultados parciales que van calculando los esclavos. El dueño calcula el resultado final a partir de estos resultados parciales. Normalmente no hay comunicación entre los esclavos.

Se puede implemenmtar de forma mixta MPMD-SPMD con un programa para el dueño y otro para los esclavos o bien mediante SPMD con un sólo programa para ambos.

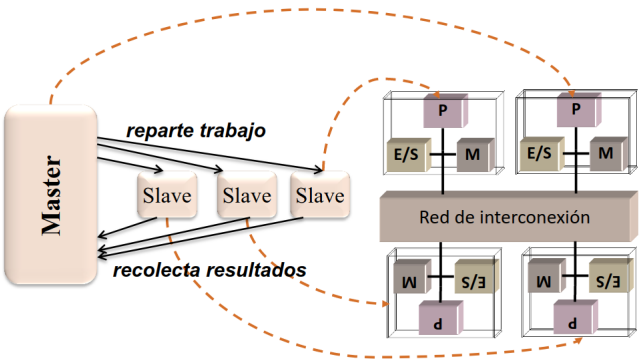


Figura 24: Dueño-esclavo.

```
int main(){
/**Código dueño***/
}
--------------
int main(){
/**Código esclavo***/
}
```

(a) Dueño-esclavo como MPMD-SPMD

```
int main(){
 if(id_proc==id_duenio){
  /**Código dueño**/
 }
 else{
  /**Código esclavo**/
 }
}
```

(b) Dueño-esclavo como MPMD-SPMD

Figura 25: Diferentes implementaciones de dueño-esclavo.

Paralelismo de datos o descomposición de datos Esta alternativa se utiliza para obtener tareas paralelas en problemas en los que se opera con grandes estructuras de datos. La estructura de datos de entrada o la de salida (o ambas) se dividen en partes, de las que derivarán las tareas paralelas.

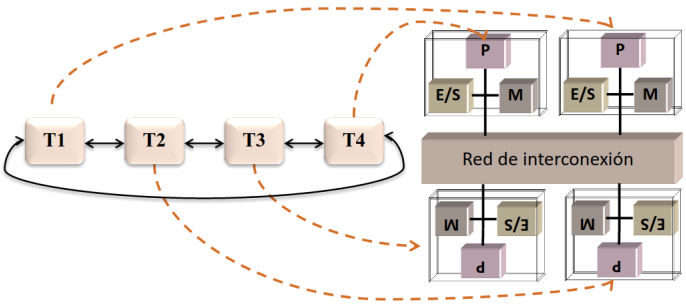


Figura 26: Descomposición de datos.

Divide y Vencerás Consiste en dividir un problema en dos o más subproblemas de forma que cada uno se pueda resolver de forma independiente y combinar los resultados para obtener el resultado final. Si los subproblemas son instancias más pequeñas que el original, podremos implementarlo mediante recursividad.

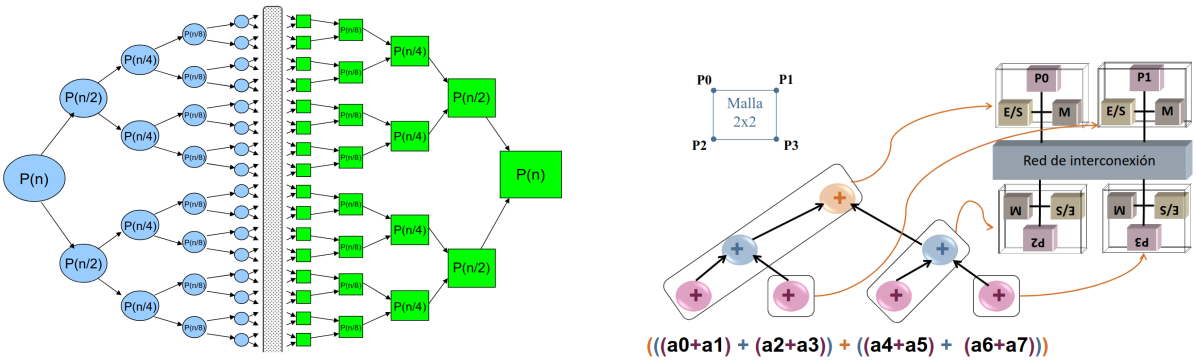


Figura 27: Divide y Vencerás.

Cliente-servidor Los clientes realizan peticiones a un servidor y éste les envía las respuestas.

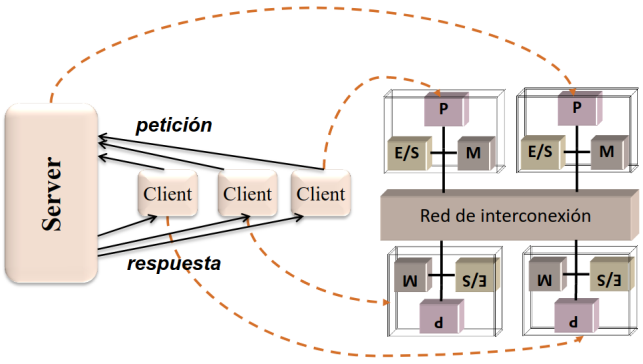


Figura 28: Cliente-servidor.

Segmentada (*pipeline*) o flujo de datos Aparece en probelmas en los que se aplican distintas funciones a un mismo flujo de datos (paralelismo de tareas). La estructura de procesos y de tareas es la de un cauce segmentado, por lo que cada proceso ejecuta distinto código (MPMD).

```
#include <omp.h>
...
omp_set_num_threads(4);
...
#pragma omp parallel
{...
 for (i=0;i<N;i++) {
   ...
   #pragma omp sections
   {
    #pragma omp section
      P1();
    #pragma omp section
      P2();
    #pragma omp section
      P3();
    #pragma omp section
      P4();
   } ...
 }
...
} ...
```

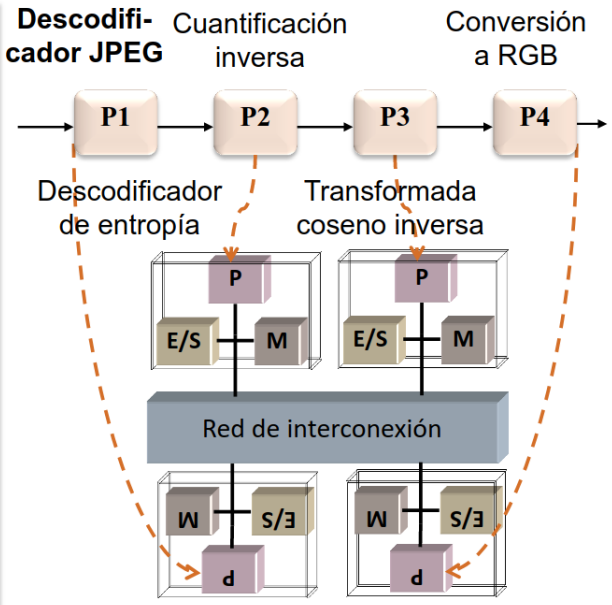


Figura 29: Segmentada (*pipeline*). Decodificación JPEG.

2.2. Proceso de paralelización

Para obtener una versión paralela de una aplicación debemos seguir los siguientes pasos:

- Descomponer la aplicación en tareas.

- Asignar tareas a procesos o hebras.
- Redactar código paralelo
- Evaluar prestaciones

2.2.1. Descomposición de tareas

En esta fase el programador busca unidades de trabajo independientes, es decir, que se podrán ejecutar en paralelo. Estas unidades, junto con los datos que utilizan, formarán las **tareas**. Podemos represetnar la estructura de las tareas (sus dependencias) meddiante un grafo dirigido en el que las aristas representen el flujo de datos y control y los vértices, las tareas.

El paralelismo se puede extraer en varios niveles de abstracción:

- **Nivel de función**. Analizando las dependencias entre las funciones del código (paralelismo de tareas).
- **Nivel de bucle**. Analizando las iteraciones de los bucles dentro de una función (paralelismo de datos).

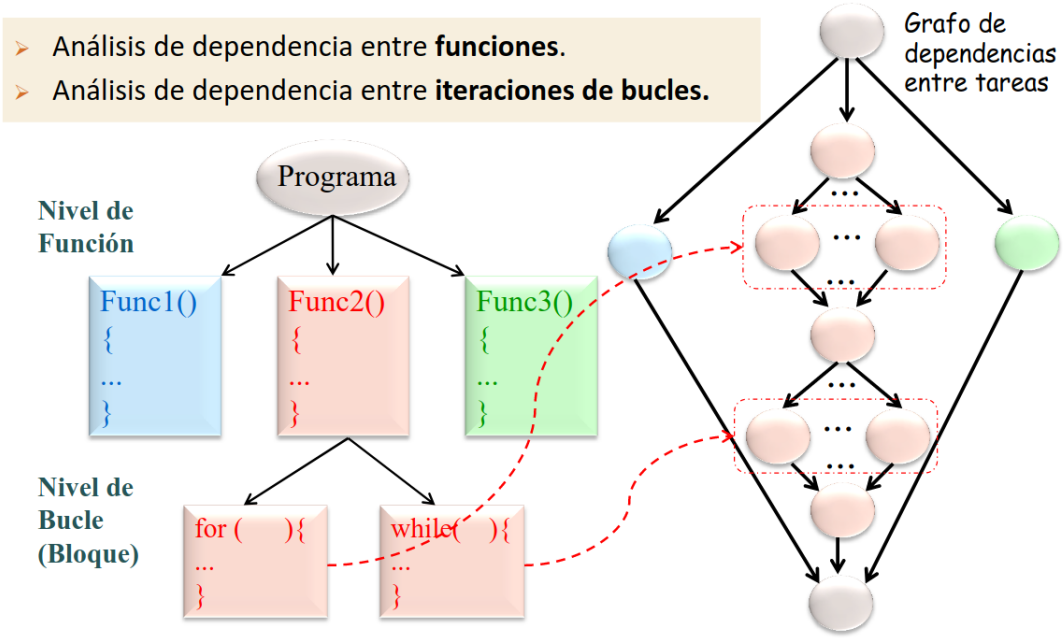


Figura 30: Segmentada (Grafo de dependencias entre tareas.

2.2.2. Asignar tareas a procesos o hebras

Esta etapa consiste en asignar las tareas del grafo de dependencias a procesos y a hebras. Por lo general, en una misma aplicación no resulta conveniente asignar más de un proceso o hebra por procesador, por lo que la asignación a procesos o hebras está ligada con la asignación a procesadores. Es más, se puede incluso realizar la asignación asociando los procesos (hebras) a procesadores concretos.

La posibilidad de utilizar procesos y/o hebras depende de varios factores:

- La **arquitectura** en la que se va a ejecutar el programa. En un SMP (*Symetric MultiProcessor*) y en procesadores multihebra es más eficiente usar hebras, mientras que en arquitecturas mixtas (como clústers de SMPs) es conveniente usar tanto hebras como procesos.
- El **Sistema Operativo** debe ser multihebra.
- La **herramienta de programación** debe permitir el uso de hebras.

Como regla básica, se tiende a asignar las iteraciones de un bucle (paralelismo de datos) a hebras y las funciones a procesos (paralelismo de tareas).

La asignación debe repartir la *carga de trabajo* (tiempo de cálculo) optimizando la *comunicación y sincronización*, de forma que todos los procesadores empiecen y terminen a la vez.

Las arquitecturas pueden ser heterogéneas u homogéneas. Si es **heterogénea**, consta de componentes con diferentes prestaciones, por lo que se deberá asignar más trabajo a nodos más rápidos. Una arquitectura **homogénea** puede ser muy uniforme o no. Si es **uniforme**, la comunicación de los procesadores con memoria (multiprocesadores) o entre sí (multicomputadores) supone el mismo tiempo sean cuales sean los nodos que intervienen.

Si la arquitectura es homogénea pero **no uniforme**, es más difícil asignar las tareas de forma que se minimice el tiempo de comunicación y sincronización (aristas en el grafo de tareas) y en general el tiempo de ejecución.

La asignación de tareas a procesadores (procesos, hebras) se puede realizar de forma **estática**, es decir, en tiempo de compilación o al escribir el programa o de forma **dinámica** (en tiempo de ejecución).

2.2.3. Escribir el código paralelo

El código dependerá del estilo de programación utilizado (variables compartidas, paso de mensajes, paralelismo de datos), del modo de programación (SPMD, MPMD, mixto,...), del punto de partida, etc.

En el programa habrá que añadir olas funciones, directivas o construcciones del lenguaje que hagan falta para:

- Crear y terminar procesos.
- Localizar paralelismo.
- Asignar la carga de trabajo.
- Comunicar y sincronizar los diferentes procesos.

2.2.4. Evaluación de prestaciones

Una vez redactado el programa paralelo, se evaluarían sus prestaciones. Si no son las requeridas, se debe volver a etapas anteriores del proceso de implementación. Si volvemos a la etapa de escritura, podemos elegir otra herramienta de programación, ya que no todas ofrecen las mismas prestaciones.

2.3. Prestaciones en computadores paralelos

En computadores paralelos se utilizan principalmente las siguientes medidas de prestaciones:

- **Tiempo de ejecución (respuesta)**. Es el tiempo que supone la ejecución de una entrada en el sistema.
- **Productividad (*throughput*)**. Es el número de entradas (aplicaciones) que el computador es capaz de procesar por unidad de tiempo.

Hay sistemas orientados a la mejora de la productividad (asignan cada entrada a un nodo de cómputo diferente), a la mejora del tiempo de respuesta (dividiendo el trabajo entre procesos) y orientados a ambos propósitos, como los servidores de internet.

También se utilizan otras medidas adicionales de prestaciones:

- **Funcionalidad**. Cargas de trabajo para las que está orientado el diseño de la arquitectura.
- **Alta disponibilidad**. Presencia de recursos y software en el sistema para reducir el tiempo de inactividad y la degradación de prestaciones ante un fallo o por mantenimiento.
- **RAS (*Reliability, Availability, Serviceability*)**. Comprende tres propiedades cruciales: fiabilidad (capacidad del sistema de producir siempre los mismos resultados para las mismas entradas), disponibilidad (grado en el que un sistema sufre degradaciuón de prestaciones o detiene su funcionamiento por fallos o mantenimientos y serviciabilidad (facilidad con la que un técnico de hardware puede realizar el mantenimiento).
- **Tolerancia a fallos**. Capacidad de un sistema de mantenerse en funcionamiento ante un fallo.
- **Expansibilidad**. Posibilidad de expandir el sistema modularmente.
- **Escalabilidad**. Evolución del incremento (ganancia) en prestaciones que se consigue en el sistema conforme se añaden recursos (principalmente procesadorees). Pretende medir el nivel de aprovechamiento efectivo de los recursos.
- **Consumo de potencia**. Afecta a los costos de mantenimiento.

También se suele hablar de **eficiencia**, con la que se evalúa en qué medida las prestaciones que ofrece un sistema para sus entradas se acerca a las prestaciones máximas que idealmente debería ofrecer dados los recursos de los que dispone. Es decir, se emplea para evaluar en qué medida se utilizan los recursos del sistema.

2.3.1. Ganancia en prestaciones. Escalabilidad

Se emplea la ganancia en velocidad para estudiar en qué medida se incrementan las prestaciones al ejecutar una aplicación en paralelo en un sistema con múltiples procesadores frente a su ejecución en un sistema uniprocesador.

$$S(p) = \frac{Prestaciones(p)}{Prestaciones(1)} \tag{10}$$

Es decir, dividiendo las prestaciones conseguidas con un sistema multiprocesador entre las prestaciones obtenidas con la versión secuencial. Utilizando el tiempo de respuesta para evaluar prestaciones, quedaría:

$$S(p) = \frac{T_{secuencial}}{T_{paralelo}(p)} \tag{11}$$

siendo $T_{paralelo}(p) = T_{computo}(p) + T_{overhead}(p)$
$T_{secuencial}$ debe ser el tiempo del mejor programa secuencial para la aplicación. En la penalización (*overhead*) influyen factores como:

- Tiempo de comunicación/sincronización entre procesos.
- Tiempo para crear/terminar procesos.
- Tiempo de ejecución de operaciones añadidas a la versión paralela no presentes en la secuencial.

Tanto el tiempo de cálculo paralelo como la sobrecarga dependen del número de procesos, Cuanto mayor sea, mayor será el *grado de paralelismo* aprovechado. El grado de paralelismo para un programa es el número máximo de tareas independientes que se pueden ejecutar en paralelo. La sobrecarga depende del número de procesos involucrados.

La representación de la ganancia en función del número de procesadores nos permite evaluar la escalabilidad de una implementación paralela o una arquitectura.

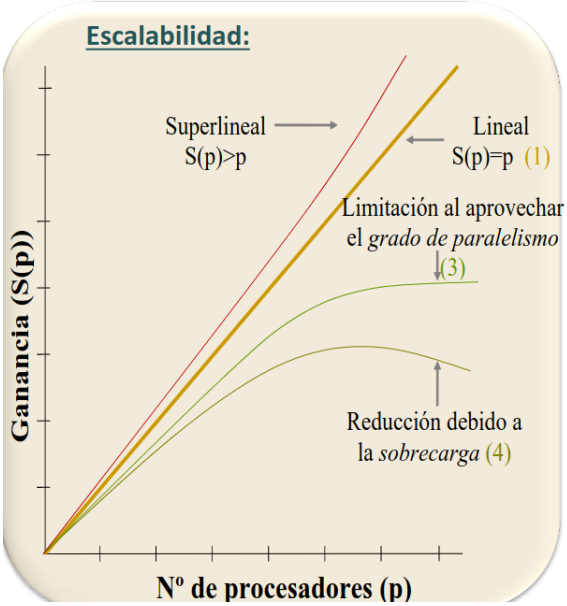


Figura 31: Segmentada (Ganancia frente a número de procesadores (escalabilidad).

Podemos ver que se dan varios casos:

1. **Ganancia lineal**. El grado de paralelismo debe ser ilimitado, es decir, siempre se debe poder dividir el código entre los p procesadores disponibles sea cual sea el valor de p. Además, el *overhead* debe ser despreciable. $S_p = \frac{T_{secuencial}}{T_{paralelo}} = p$.

2. **Ganancia superlineal** ($S_p > p$). Se debe a que al aumentar el número de procesadores aumentamos también el de otros recursos (caché, memoria principal, etc.) o bien a que la aplicación debe explorar varias posibilidades y termina cuando una es solución.

3. **Limitación al aprovechar el grado de paralelismo**. Se produce cuando hay código no paralelizable y la paralelización que se puede realizar no mejora las prestaciones.

4. **Reducción debida a la sobrecarga**. Se produce cuando el incremento del número de procesadores no hace que el programa sea más rápido, pero sí que hace que el *overhead* sea mayor.

Ley de Amdahl Como vemos, la mejora en prestaciones está limitada por la fracción de código que no se puede paralelizar. Este razonamiento se formalizó por Amdahl mediante la siguiente ley:

$$S(p) \leq \frac{p}{1 + f(p-1)} \tag{12}$$

donde:

- S: incremento en velocidad que se consigue al aplicar una mejora.
- p: incremento en velocidad máximo que se puede conseguir si se usa siempre la mejora (número de procesadores).
- f: fracción de tiempo en la que no se utiliza la mejora (fracción de tiempo no paralelizable).

La ley de Amdahl nos dice que la escalabilidad está limitada por la fracción de tiempo no paralelizable. Sin embargo, en muchas aplicaciones podemos disminuirla aumentando el tamaño del problema, lo que conduce a un aumento de la ganancia.

Ganancia escalable. Ley de Gustafson Los objetivos al paralelizar una aplicación pueden ser:

- Disminuir el tiempo de ejecución hasta que sea razonable.
- Aumentar el tamaño del problema a resolver, lo que nos puede llevar a mejoras como el aumento de la precisión.

Cuando llegamos a un nivel aceptable de tiempo de ejecución paralelo, el siguiente objetivo puede ser aumentar el tamaño del problema para poder mejorar otros aspectos de las prestaciones de la aplicación. Si consideramos el tiempo de *overhead* insignificante, podemos mantener constante el tiempo de ejecución paralelo T_p variando el número de procesadores p y el tamaño n de forma que $n = \Bbbk p$ con $\Bbbk \in \mathbb{R}$. Bajo estas condiciones, la ganancia en prestaciones sería:

$$S(p) = \frac{T_{secuencial}(n)}{T_{paralelo}} = \frac{f \cdot T_{paralelo} + p(1-f) \cdot T_{paralelo}}{T_{paralelo}} = p(1-f) + f \qquad (13)$$

donde f representa la fracción de tiempo de la ejecución del programa paralelo que supone la ejecución de la parte no paralelizable. Cuanto mayor sea $1 - f$, mayor será la escalabilidad.

Mientras que la Ley de Amdahl asume que el $T_{secuencial}$ (o tamaño de problema) es constante, Gustaffson mantiene que lo constante es el $T_{paralelo}$. Ambos consideran despreciable el *overhead*.

Eficiencia La eficiencia permite evaluar en qué medida las prestaciones ofrecidas por un sistema para un programa paralelo se acercan a las prestaciones máximas que idealmente debería ofrecer.

$$E(p,n) = \frac{Prestaciones(p,n)}{Prestaciones(1,n)} = \frac{S(p,n)}{p} \qquad (14)$$

donde:

- p representa el número de recursos (procesadores).
- n representa el tamaño del problema.

2.3.2. Ejercicios