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.

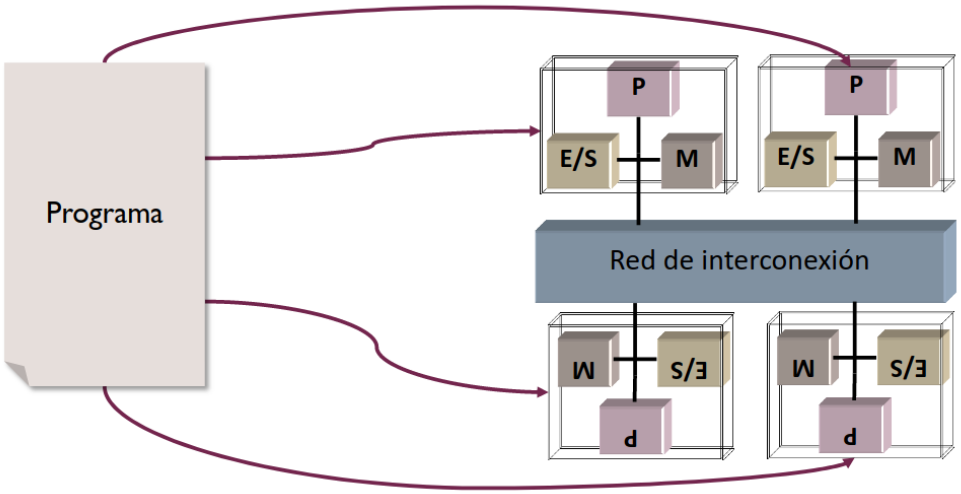

Figura 14: SPMD (Single Program Multiple Data).

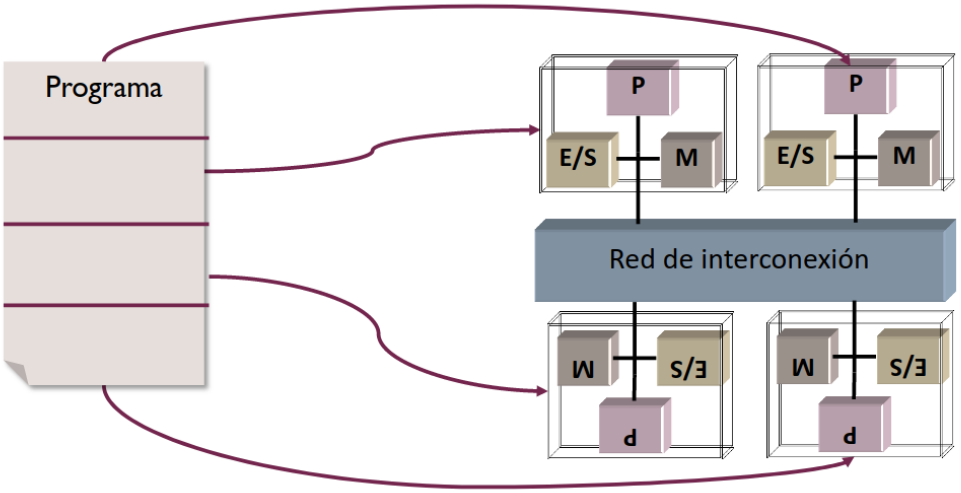

Figura 15: MPMD (Multiple Program Multiple Data).

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.

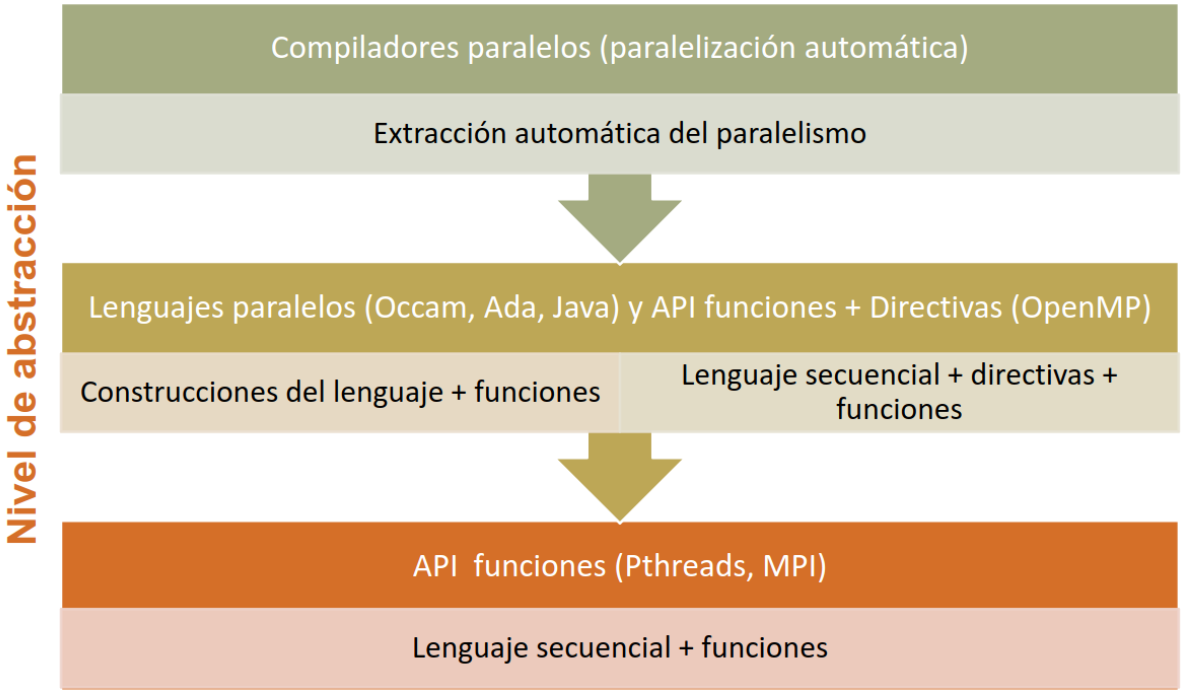

Figura 16: Principales herramientas de programación paralela.

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.

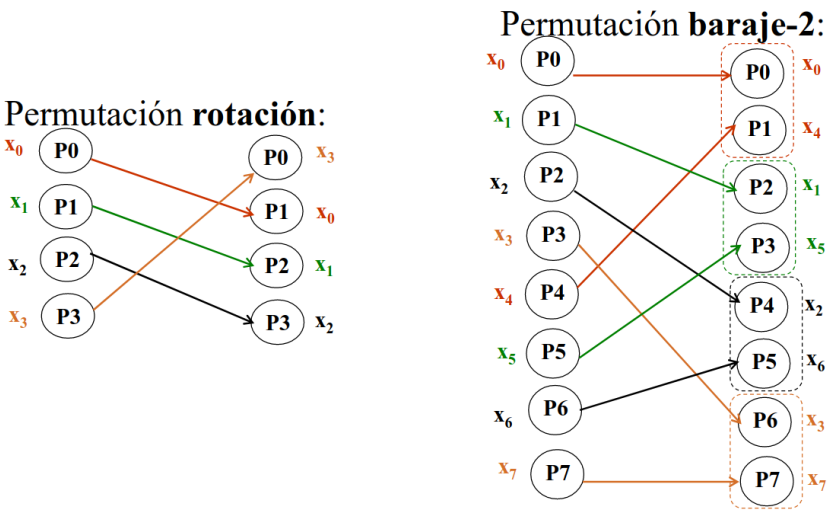

Figura 17: Comunicación uno a uno.

Comunicación uno a todos Un proceso envía y 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.

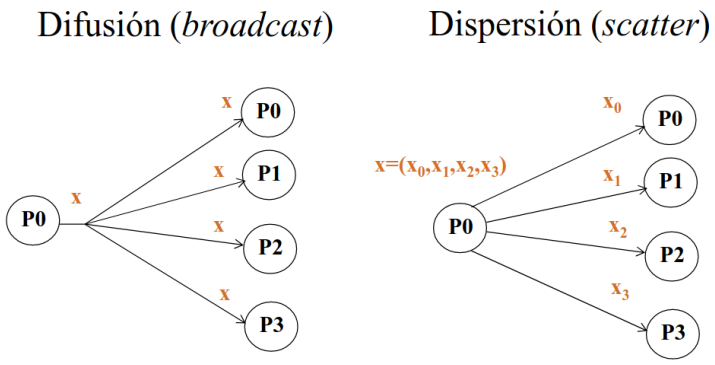

Figura 18: Comunicación uno a todos.

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.

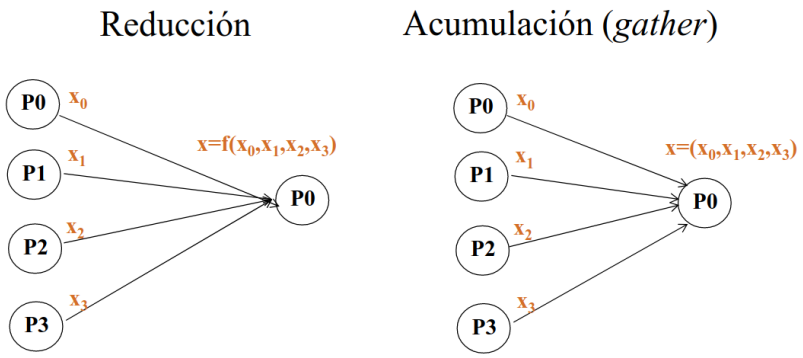

Figura 19: Comunicación todos a uno.

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 $i(x_{0i}, x_{1i}, ...)$.

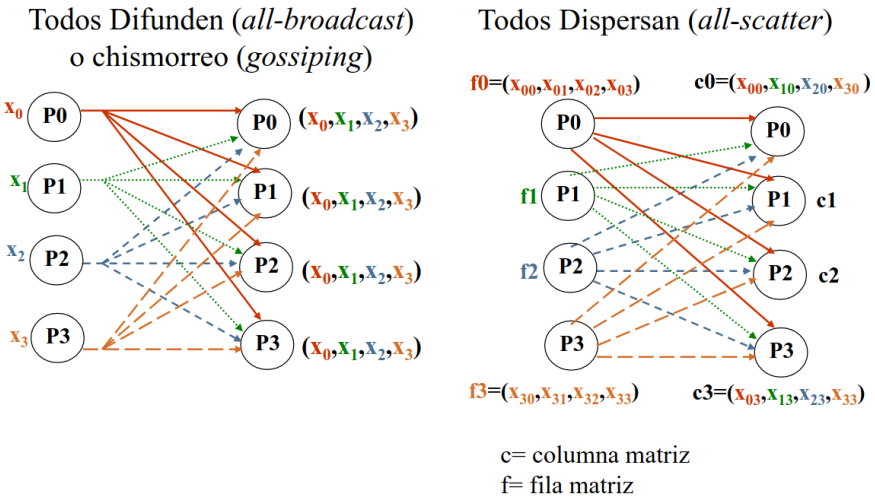

Figura 20: Comunicación todos a todos.

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).

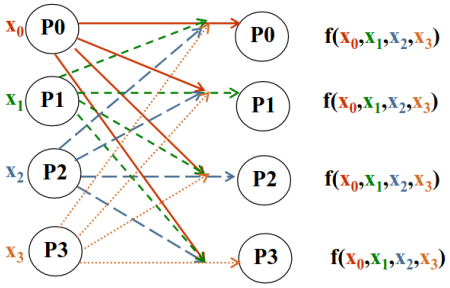

Figura 21: Todos combinan.

- **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.

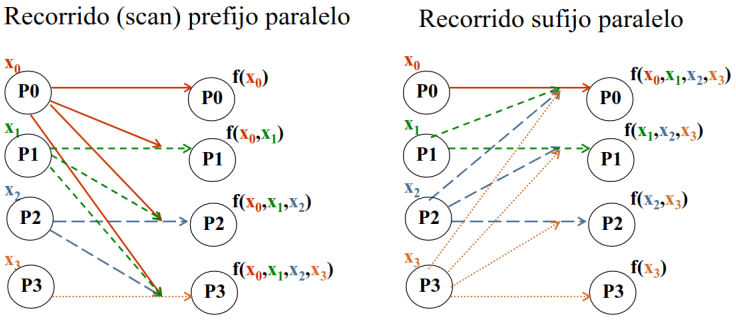

Figura 22: Recorrido (*scan*).

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*.

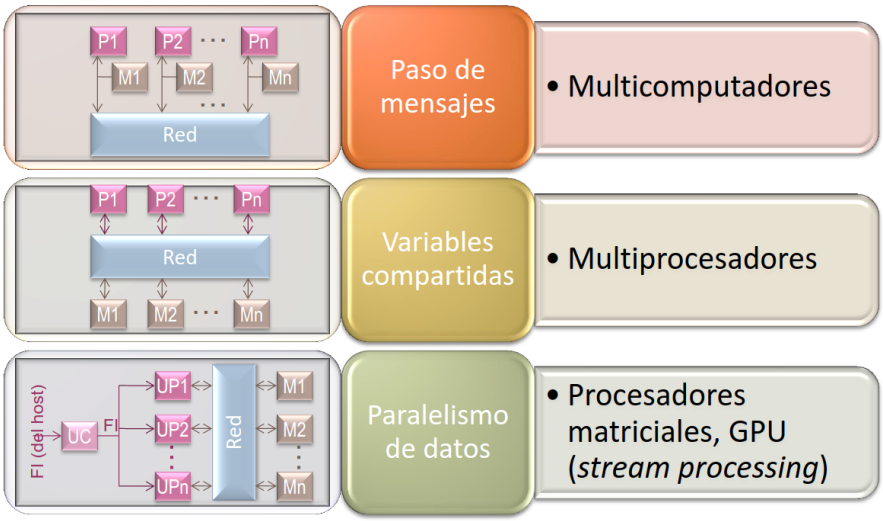

Figura 23: Estilos de programación paralela.

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.

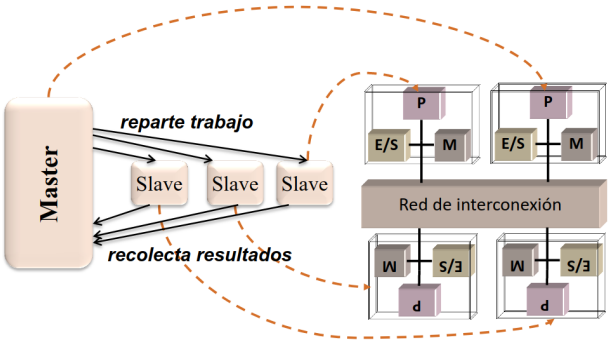

Figura 24: Dueño-esclavo.

```
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.

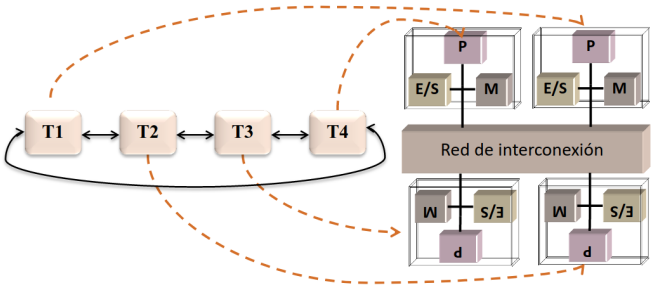

Figura 26: Descomposición de datos.

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.

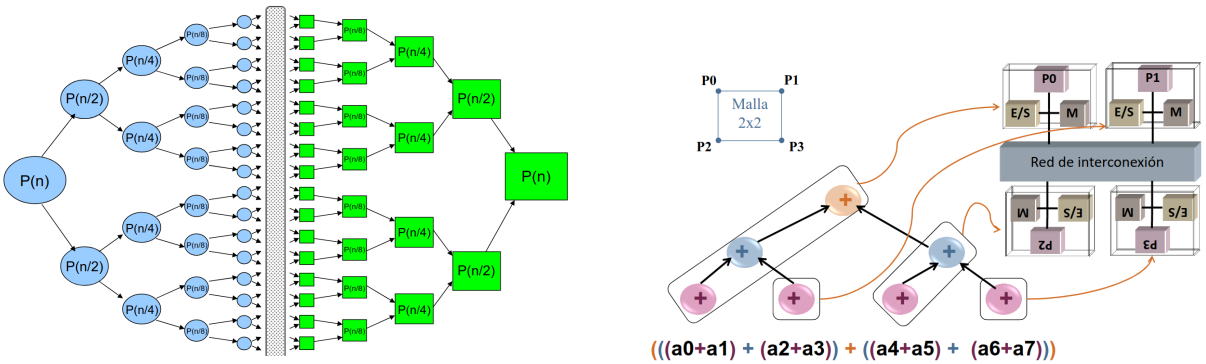

Figura 27: Divide y Vencerás.

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

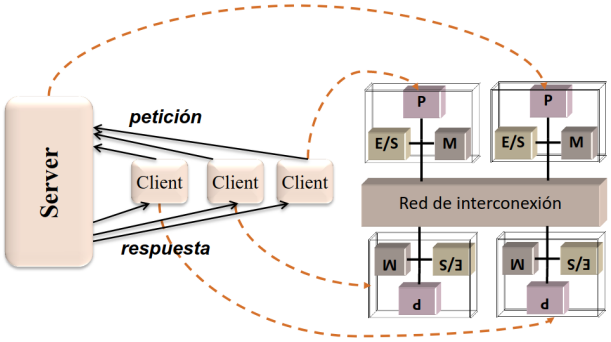

Figura 28: Cliente-servidor.

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();
   } ...
 }
...
} ...
```

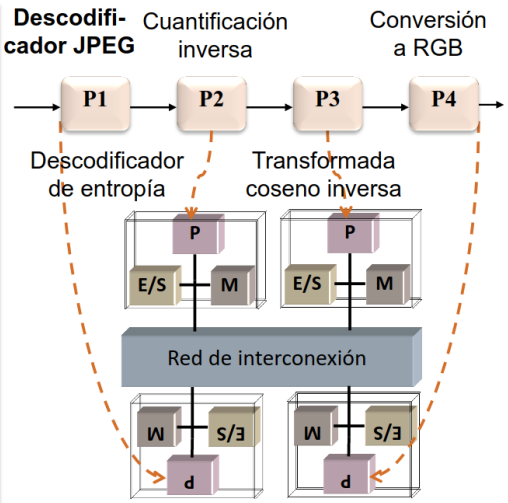

Figura 29: Segmentada (*pipeline*). Decodificación JPEG.

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).

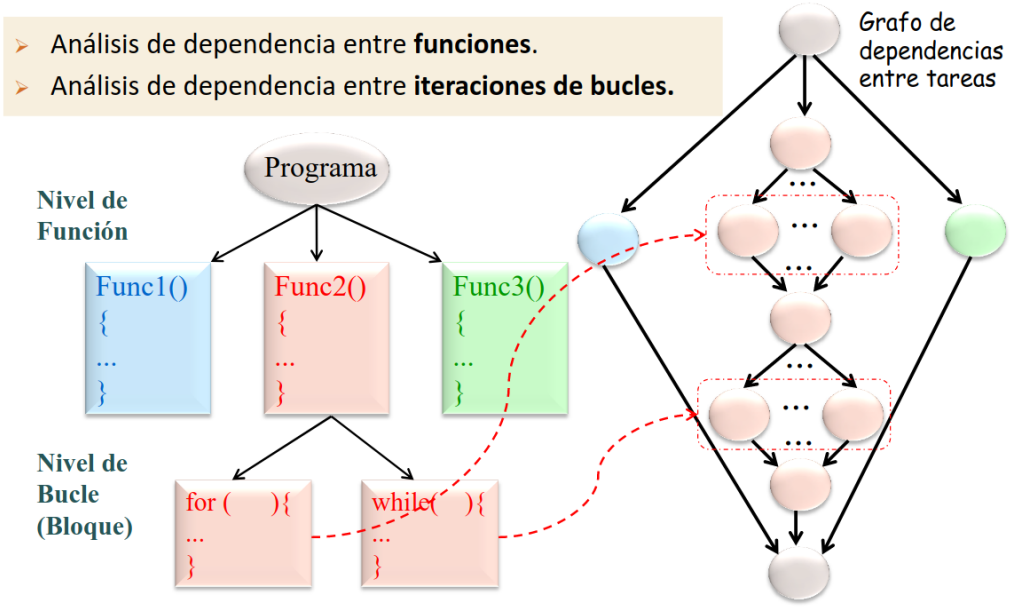

Figura 30: Segmentada (Grafo de dependencias entre tareas).

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 Multi-Processor*) 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.

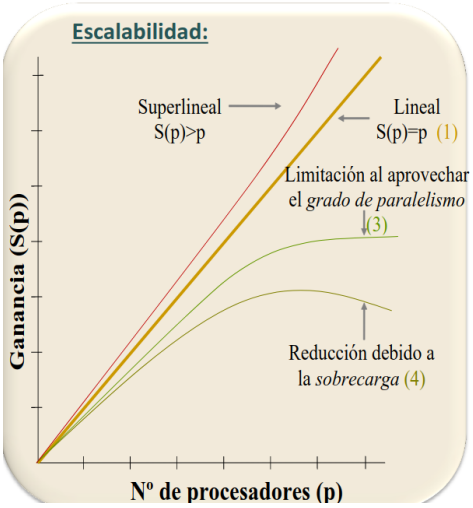

Figura 31: Segmentada (Ganancia frente a número de procesadores (escalabilidad).

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:

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

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, Gustafson 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.

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

donde:

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