

AC 21/22 - Tema 2: Programación paralela

1. Introducción

En este capítulo abordaremos los siguientes temas relacionados con la programación paralela:

1. Las herramientas existentes para escribir código paralelo y el trabajo extra que implica para las herramientas y el programador la paralelización
2. Las estructuras típicas de flujos de instr en códigos paralelos
3. Cómo evaluar las prestaciones del código paralelo implementado y qué prestaciones se pueden esperar

2. Herramientas de programación paralela

2.1. Trabajo a realizar por las herramientas de programación paralela o por el programador

La programación paralela requiere que la herramienta de programación utilizada, el programador o ambos realicen el siguiente trabajo, no requerido en la programación secuencial:

- **Localización del paralelismo implícito** en una aplicación, es decir, descomponer la aplicación en unidades de cómputo independientes, o **tareas**.
Es recomendable terminar la descomposición con un grafo dirigido en el que los vértices sean tareas y los arcos dependencias entre ellas. Nos permitirá ver fácilmente el número máximo de tareas que se pueden ejecutar en paralelo, o **grado de paralelismo** (número de flujos de instr o procesadores máximo necesario para la parallelización).
- **Asignación de las tareas o carga de trabajo** a flujos de instrucciones, y asignación de flujos de instr a procesadores. No suele ser rentable usar más flujos que procesadores. Los flujos pueden estar asociados a los procesadores previa asignación. La asignación puede ser:

- Estática o dinámica
 - Con una asignación estática el reparto de tareas es siempre el mismo; mientras que con la dinámica el reparto puede variar y solo se conoce al final de la ejecución.

La asignación dinámica implica código extra y sincronización/comunicación entre hebras, lo cual introduce retardos adicionales. Pero es la única opción cuando no conocemos el número de tareas previamente.

- Implícita (herramienta) o explícita (programador)
- **Comunicación/sincronización** entre los flujos de instr. Los flujos tienen que colaborar, enviarse los resultados que producen...

(a) Cálculo de Pi con C

```

main(int argc, char **argv) {
    double ancho, sum=0;
    int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    for (i=0;i< intervalos; i++){
        x = (i+0.5)*ancho;
        sum = sum + 4.0/(1.0+x*x);
    }
    sum* = ancho; }

```

Grafo de dependencias entre tareas

(b) Cálculo de Pi con OpenMP/C

```

#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    double ancho,x, sum=0; int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); →Crear y Terminar
#pragma omp parallel →Comunicar y sincronizar
    Localizar{
        #pragma omp for reduction(+:sum) private(x)
        for (i=0;i< intervalos; i++) {
            x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
        }
        sum* = ancho;
    }
}

```

(c) Cálculo de Pi con MPI/C

```

#include <mpi.h>
main(int argc, char **argv) {
    double ancho,x,sum,lsum=0; int intervalos,i,nproc,iproc;
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1); →Enrolar
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]); →Localizar y Asignar
    ancho=1.0/(double) intervalos;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho; →Comunicar/sincronizar
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
    MPI_Finalize(); →Desenrolar
}

```

Figura 1. (a) Código secuencial C para el cálculo de Pi y códigos paralelos en C usando (b) OpenMP (API+Directivas+Funciones), estándar de-facto para programación con el estilo de variables compartidas y (c) MPI (API+funciones), estándar de-facto para el estilo de paso de mensajes. Las modificaciones realizadas al código secuencial (a) para obtener código paralelo se han destacado en negrita en (b) y (c)

Las herramientas permiten, de forma implícita (lo hace la herramienta) o explícita (lo hace el programador), las siguientes **LABORES**:

1. Localizar/Detectar paralelismo, o sea, descomponer la aplicación en tareas independientes (*decomposition*).
2. Asignar las tareas, la carga de trabajo (instr+datos) a flujos (*scheduling*).
3. Crear y terminar flujos de instr, o enrolar y desenrolar en un grupo flujos previamente creados.
4. Comunicar/Sincronizar flujos de instr.
5. Asignar flujos de instr a unidades de procesamiento (*mapping*).

La última labor la puede hacer el SO o el hardware, depende del sistema

2.2. Nivel de abstracción en que sitúan al programador las herramientas

Cuanto mayor sea el nivel de abstracción, menos labores de las enunciadas anteriormente dependerán del programador. Hay herramientas que permiten escoger el nivel de abstracción. La labor más difícil para la herramienta es la primera, localizar el paralelismo. Cuantas más labores haga el programador, mayores prestaciones se pueden llegar a conseguir si el programador conoce la arquitectura del sistema de cómputo.

Las herramientas de paralelismo se pueden clasificar según su nivel de abstracción, de menos a más:

- **Compiladores paralelos:** Ej. compiladores de Intel. Generan código paralelo a partir de código secuencial. El programador sólo tiene que implementar el código secuencial, no realiza ninguna de las labores previas.
- **Lenguajes paralelos y APIs de funciones y directivas:** en este nivel, el programador debe al menos detectar el paralelismo (labor 1). El programador no hace el reparto (labor 2), y pueden librarse al programador de asignar flujos a procesadores (labor 5), saber como se crean/terminan flujos (labor 3) o detalles de comunicación y sincronización (labor 4).
 - Lenguajes paralelos, como Occam, Ada, Java; tienen construcciones específicas y bibliotecas de funciones. Requieren un compilador exclusivo.
 - APIs de func y directivas, como OpenMP, OpenACC; constan de directivas y una biblioteca de funciones que se añaden a un compilador de un lenguaje secuencial normal, como C/C++ o Fortran.
- **APIs de funciones:** Ej. pthreads, MPI. Consisten en una biblioteca de funciones que se añaden a un compilador de lenguaje secuencial. El programador debe hacer explícitamente la asignación de tareas (labor 1), y participar en todas las labores menos, probablemente, la última.
- **Lenguajes paralelos para arquitecturas de propósito específico:** ej. CUDA. Consisten en construcciones de lenguaje y bibliotecas de funciones. Requieren un compilador exclusivo. El programador debe participar en todas las labores excepto, probablemente, la última; y debe tener conocimientos mayores sobre la arquitectura de destino para poder escribir el código paralelo.

2.3. Estilos de programación paralela

Cada herramienta ofrece al programador un modelo de programación particular. Las arquitecturas paralelas se caracterizan por el estilo de programación más cercano a su implementación hardware: paso de mensajes para multicomputadores (Ej. MPI); variables compartidas para multiprocesadores y núcleos multithread (Ej. OpenMP); y paralelismo de datos para procesadores que ejecutan una instrucción en paralelo en múltiples unidades de procesamiento (SIMD, Single Instruction Multiple Data) (Ej. HPF). Teniendo eso en cuenta, es fácil ver qué caracteriza a cada uno de estos modelos de programación:

- **Paso de mensajes:** el programador debe tener en cuenta que los flujos de instrucción no comparten memoria, cada uno de ellos tiene su espacio de direcciones en particular. La herramienta debe ofrecer medios para copiar datos de un espacio de direcciones a otro.
- **Variables compartidas:** como los flujos comparten memoria, la información puede pasarse a través de variables. La herramienta debe ofrecer (si no lo hace ya implícitamente) medios para sincronizar los flujos con el fin de que las comunicaciones a través de variables no tengan problemas: la sincronización debe conseguir que el consumidor de un dato lo lea después de que el productor lo escriba en la variable compartida, no antes.
- **Paralelismo de datos:** el programador debe tener en cuenta que todos los flujos de datos ejecutan a la vez la misma instrucción con datos distintos.

2.4. Comunicación/Sincronización en herramientas de programación paralela

Las herramientas de comunicación implementan comunicación uno-a-uno, es decir, envío de datos desde un emisor hasta un receptor. También pueden ofrecer comunicaciones colectivas de distinto tipo, que evitan tener que implementar estas comunicaciones con múltiples envíos uno-a-uno.

Las comunicaciones colectivas se pueden clasificar así:

- **Comm uno-a-todos:** uno envía y todos reciben (el que envía puede estar incluido con los que reciben). Difusión o dispersión (*Broadcast, Scatter*).
- **Comm todos-a-uno:** todos envían y uno recibe (el que recibe puede estar incluido con los que envían). Acumulación o reducción (*Gather, Reduction*).
- **Comm todos-a-todos:** todos difunden (*all-broadcast*), todos dispersan (*all-scatter*)
- **Comm múltiples uno-a-uno:** permutaciones (rotación, barajar...), todos los flujos envían a uno y todos reciben de uno, desplazamientos...
- **Comm colectivas compuestas de varias de las anteriores:**
 - Todos combinan (*all reduction*)
 - Recorrido (*scan*)
 - Barreras (*barrier*)
 - Punto en el código en el que todos los flujos se esperan entre sí. Cuando todos llegan a la barrera, continúan a la vez con la ejecución. Se compone de una reducción y una difusión.

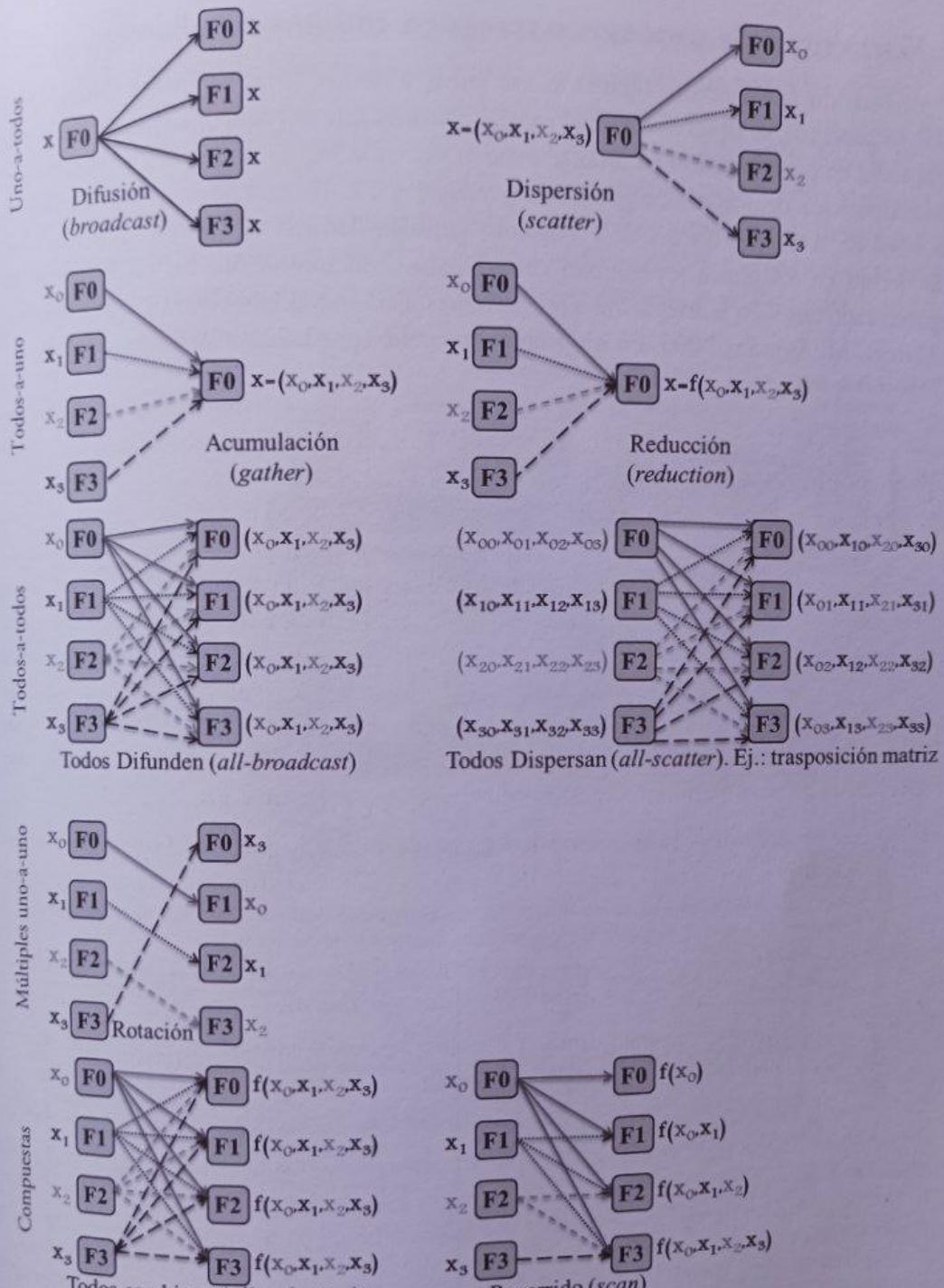
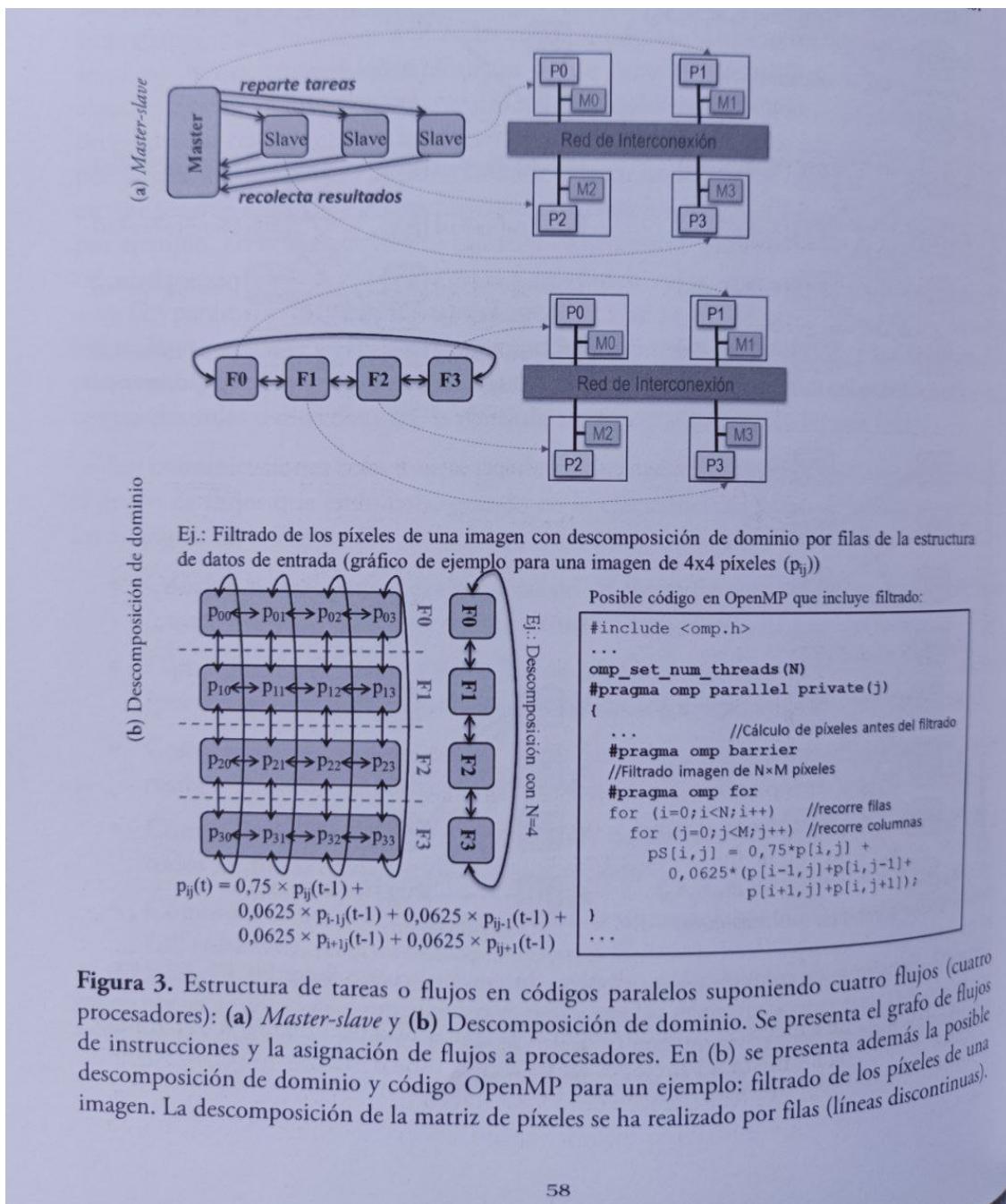


Figura 2. Funciones de comunicación colectivas. Intervienen cuatro flujos F0, F1, F2 y F3. La operación $f()$ que reduce múltiples valores a uno es commutativa y asociativa. Cuando en los ejemplos un flujo recibe o envía un dato (x_i o x_{ij}) se podría también recibir o enviar múltiples datos (x_i o x_{ij} podrían ser vectores).

3. Estructuras de procesos o tareas en códigos paralelos

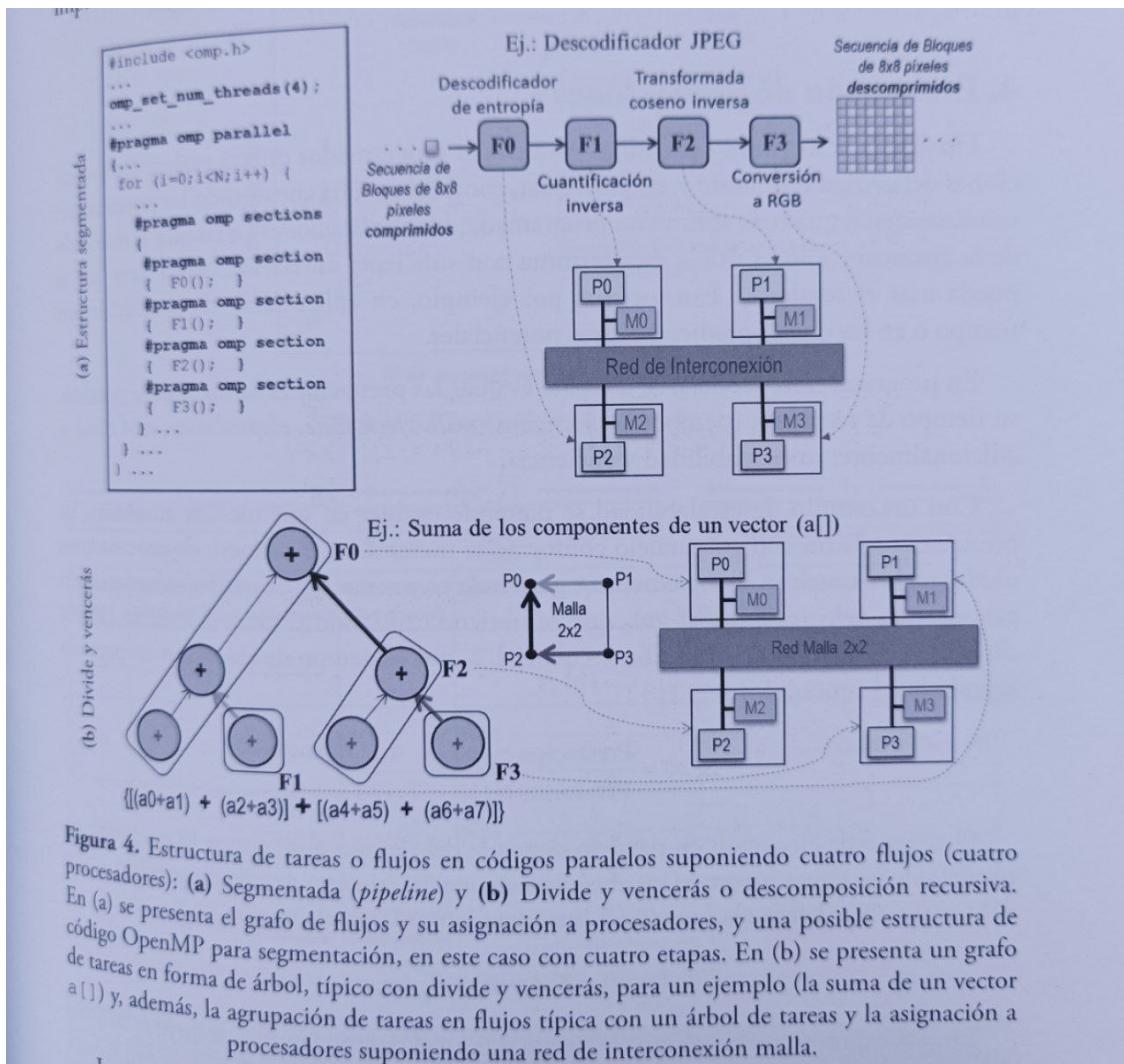
Analizando la estructura o grafo de las tareas y de los flujos de instr de los códigos paralelos, podemos ver ciertos patrones que se repiten. En un programa paralelo puede haber varias estructuras. Las estructuras más comunes son:

- **Descomposición del dominio y Maestro-Esclavo:** el trabajo a realizar por cada proceso se determina dividiendo las estructuras de datos de entrada o las de salida en trozos, tanto como flujos (descomposición del dominio), y luego cada flujo gestiona sus tareas y reporta el resultado a una hebra master.
- Si el flujo se divide en entradas, el trabajo a asignar al flujo i será el que utilice las entradas del flujo i .
- Si el flujo se divide en salidas, el trabajo a asignar al flujo i será el que genere las salidas del trozo i .



- **Cauce segmentado (*pipeline*):** se utiliza cuando se aplica a un flujo de datos de entrada la misma secuencia de instrucciones, una detrás de otra, como una cadena de montaje.

- **Divide y vencerás:** se usa cuando el problema a resolver se puede dividir en subtareas que son instancias más pequeñas del problema original, de forma que combinando los resultados de las subtareas se resuelve el problema original. Las subtareas también se pueden dividir en subtareas, dando lugar a una estructura en forma de árbol.



4. Evaluación de prestaciones

Tras escribir un código paralelo, hay que comprobar que se cumplen los requisitos de tiempo. Para evaluar las prestaciones del código, se evalúa su tiempo de respuesta (desde el inicio de ejecución hasta obtener los resultados).

Un estudio de escalabilidad analiza en qué medida aumentan las prestaciones del código conforme aumenta el número de procesadores. La ganancia en prestaciones (o ganancia en velocidad) se calcula:

$$S(\text{num procesadores}) = \text{Prestaciones}(\text{num procesadores}) / \text{Prestaciones}(1 \text{ procesador})$$

$$S(\text{num procesadores}) = \text{tiempo ej. secuencial} / \text{tiempo ej. paralelo con } p \text{ procesadores} = T_s / T_p(1)$$

El tiempo de ejecución en paralelo se calcula:

$$T_p(p, \text{tam problema } n) = \text{Tiempo calculo paralelo}(p, n) + \text{Tiempo de sobrecarga/overhead }(p, n)$$

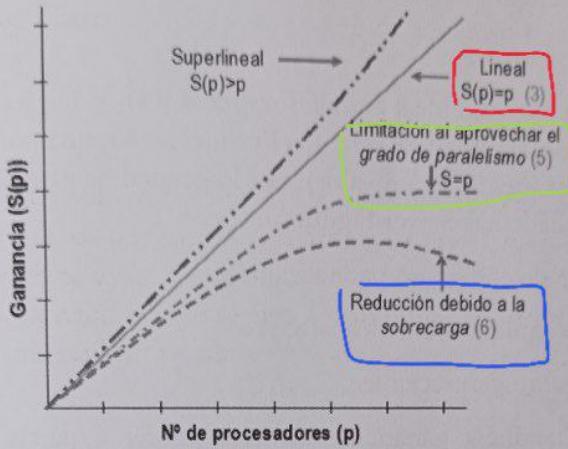
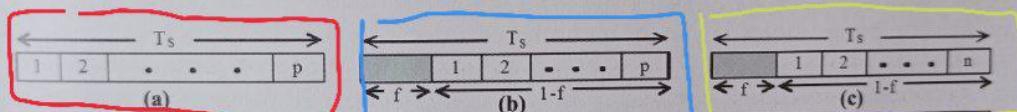


Figura 5. Curvas de escalabilidad



Modelo código	Fracción no paral. en T_s	Grado paralelismo	Sobrecarga (Overhead)	Ganancia en función del número de procesadores p con T_s constante
(a)	0	ilimitado	0	$S(p) = \frac{T_s}{T_p(p)} = \frac{T_s}{T_s/p} = p$ Ganancia lineal (3)
(b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$ Ley de Amdahl (4)
(c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$ (5)
(b)	f	ilimitado	Incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_0(p)}{T_s}} \xrightarrow{p \rightarrow \infty} 0$ (6)

Figura 6. Ganancia en prestaciones para diferentes modelo de código: (a) Todo el código se puede repartir por igual entre los p procesadores disponibles, sea cual sea p (i.e. el grado de paralelismo es ilimitado), (b) hay una fracción f del código secuencial (del tiempo de ejecución secuencial T_s) que no se puede paralelizar y el resto ($1-f$) se puede repartir entre los p procesadores disponibles, sea cual sea p , (c) hay una fracción f del código secuencial que no se puede paralelizar y el resto ($1-f$) se puede dividir hasta n trozos (como ocurre en un trozo de código con un bucle de n iteraciones).

61

En el mejor caso se esperaría obtener un tiempo de ejecución del tiempo secuencial, dividido entre el número de procesadores. Para llegar a este tiempo, se debe...

1. Poder repartir todo el código entre los procesadores disponibles, sea cual sea este número (paralelismo ilimitado)
2. Hacer el reparto asignando a cada procesador la misma cantidad de trabajo (carga equilibrada)
3. No añadir sobrecarga

La escalabilidad en tal caso sería lineal (en rojo en la figura anterior). En la práctica, la curva de escalabilidad no será una línea recta, pero puede acercarse para un rango de p . Por lo general, la ganancia está debajo de la ideal por...

1. Una fracción f no es paralelizable
2. El reparto de trabajo no está equilibrado
3. Existe sobrecarga no despreciable

La ganancia además deja de crecer a partir de un cierto número de p (en verde en la figura), de hecho si añadimos demasiados podemos generar mucha sobrecarga y afectar negativamente al rendimiento (azul en la figura).

Pero incluso si el grado de paralelismo es ilimitado y la sobrecarga no existe, **la ganancia está limitada** (caso b, segunda fila de la tabla), porque la limita la fracción no paralelizable f, que se mantendrá siempre constante. El tiempo de ejecución nunca podrá ser menor que el tiempo de ejecución secuencial de ese fragmento no paralelizable.

$$1/\text{fracción no paralelizable} = \text{tiempo secuencial} / (\text{f no paral.} * \text{tiempo secuencial})$$

La ley de Amdahl (fórmula en la segunda fila de la tabla) dice que la ganancia de prestaciones que se puede conseguir aplicando paralelismo está limitada por la fracción no paralelizable del mismo. Ahmdal da una visión pesimista (la ganancia es limitada) porque no creía en la viabilidad del paralelismo masivo pero, en la práctica, la fracción de código secuencial no paralelizable **se puede reducir si aumenta mucho el tamaño del problema** - Aumentando el tamaño se mejora la calidad de los resultados. Esto nos lleva a pensar que la paralelización de un código puede tener dos objetivos:

- Disminuir el tiempo de ejecución (nuestro enfoque hasta ahora)
- Aumentar el tamaño del problema a resolver para mejorar la calidad del resultado

Si el objetivo al aumentar p es mejorar la calidad de los resultados mediante aumento de tamaño y no el tiempo, se puede obtener una ganancia de prestaciones que **no está limitada** y depende linealmente de p:

$$S(p) = Ts / Tp = (f \cdot Tp + p \cdot (1-f) \cdot Ts) / Tp = f + p \cdot (1-f) = Ts / (f * Ts + ((1+f)*Ts) / p)$$

Esta expresión es la **ganancia estable**, de Gustafson. Mientras que Amdahl asume que el Ts se mantendrá constante aun si incrementamos la p, concluyendo pues que la ganancia está limitada por la f, Gustafson mantiene cte el Tp y muestra que la ganancia puede crecer con pendiente constante relativa al número de procesadores p.

Para evaluar en qué medida se aproximan [las prestaciones que ofrece una máquina para un programa paralelo] a las [prestaciones máximas que idealmente ofrecería esa máquina, según su num de procesadores], se usa la siguiente expresión de **eficiencia**:

$$E(p) = \text{prestaciones}(p, n) / (p * \text{prestaciones}(1, n)) = S(p) / p$$

$$S(p) = \text{tiempo ej. secuencial} / \text{tiempo ej. paralelo con } p \text{ procesadores} = Ts / Tp(p)$$