Sistemas Paralelos

FUNDAMENTOS TECNOLÓGICOS PARA BIG DATA

Isaac Esau Rubio Torres

2018

Contenido

- Introducción al paralelismo
- Evaluación del rendimiento: métricas y análisis
- Metodologías del desarrollo de programas paralelos
- Esquemas de algoritmos paralelos

Paralelismo

- El paralelismo es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente
- Basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo.
- Hay varios tipos diferentes de paralelismo:
- nivel de bit,
- nivel de instrucción, ▪ de datos y de tarea.

- El paralelismo ha sido empleado durante muchos años, sobre todo para la Computación de alto rendimiento.

- Los problemas que pueden resolverse mediante un algoritmo paralelo son, obviamente, muy heterogéneos.

  Suelen ser problemas de **complejidad elevada**, aún no perteneciendo al grupo de problemas intratables (el número de operaciones crece de forma rápida –p.e. exponencial– con el tamaño del problema).

- Dentro del conjunto de problemas tratables (el número de operaciones crece polinómicamente con el tamaño del problema) se suelen dar dos situaciones que hacen necesaria la programación paralela:

  - Problemas de **gran dimensión**
  - Problemas de **tiempo real**

Otro tipo de problemas: problemas de **gran desafío**, por su relevancia social (genoma humano, meteorología, clima, fenómenos sísmicos…).

- Diferentes modelos sobre distintos aspectos de la programación paralela:

  - Modelo **arquitectónico**: arquitectura de la máquina
    -- multiprocesadores: memoria compartida
    -- multicomputadores: paso de mensajes
    -- modelos mixtos

  - Modelo de **programación**: herramientas de alto nivel (OpenMP, MPI, MapReduce).

  - Modelo de **coste**: permite evaluar el coste del algoritmo.

- En la programación paralela (al igual que en la secuencial) son necesarias herramientas que permitan estimar el **tiempo de ejecución** y la **memoria** consumidos por un algoritmo, para determinar si es adecuado o no para la resolución del problema.

  El objetivo es desarrollar algoritmos eficientes (**eficiencia**: relación entre los recursos que consume y los resultados que obtiene).

Contenido

- Introducción al paralelismo
- **Evaluación del rendimiento: métricas y análisis**

- Metodologías del desarrollo de programas paralelos
- Esquemas de algoritmos paralelos

Rendimiento

- El rendimiento R(x) es una métrica inversa del tiempo de ejecución T(x).
- R(x) = 1 / T(x)
- Alto rendimiento → Bajo tiempo de ejecución
- X se ejecuta n veces más rápido que Y.
- T(y) / T(x) = n

$$\frac{T(x)}{R(x)} = \frac{R(y)}{1} = n$$

$$T(y) \quad \frac{1}{R(y)} \quad R(x)$$

Métricas

- La **única** métrica fiable para comparar el rendimiento de computadores es la **ejecución de programas reales**.
- Cualquier otra métrica conduce a errores.
- Cualquier alternativa a programas reales conduce a errores.
- Tiempo de ejecución.
- Tiempo de respuesta: Tiempo total transcurrido.
- Percibido por el usuario.
- Tiempo de CPU: Tiempo que la CPU ha estado ocupada.

Benchmark

- Aplicación o conjunto de aplicaciones usadas para evaluar el rendimiento.

- Aproximaciones:
- Kernels: Partes pequeñas de aplicaciones reales.
- Ejemplo: FFT.
- Programas de juguete: Programas cortos.
- Ejemplo: quicksort.
- Benchmarks sintéticos: Inventados para representar aplicaciones reales:
- Ejemplo: Dhrystone.

- Todas malas aproximaciones:
- ¡El arquitecto y el compilador pueden *engañar*!

Benchmarks

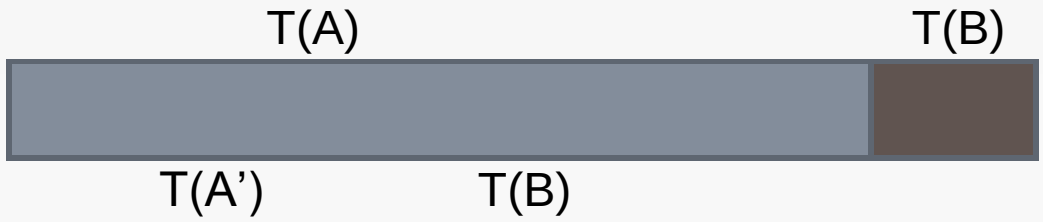
- Empotrados:
- Dhrystone (relevancia discutible).
- EEMBC (kernels).
- Desktop:
- SPEC2006 (mezcla de programas enteros y coma flotante).
- Servidores:
- SPECWeb, SPECSFS, SPECjbb, SPECvirt_Sc2010.
- TPC

Ley de Amdahl (1967)

- El incremento de rendimiento obtenido usando un modo de ejecución más rápido está limitado por la fracción de tiempo que se puede usar dicho modo.

- Sepeedup o aceleración:
- Ratio entre el rendimiento mejorado y rendimiento original.
- S = R(M) / R(O)
- S = T(O) / T(M)
- Factores:
- Fracción susceptible de mejora. [F]
- Speedup de la mejora. [S(m)]

Ley de Amdahl



Ley de Amdahl

$$S = \frac{T}{T'} = \frac{T}{T \times \left((1 - F) + \frac{F}{S(m)} \right)} = \frac{1}{(1 - F) + \frac{F}{S(m)}}$$

El *speedup* depende exclusivamente de la fracción de mejora y el *speedup* de la mejora

Caso 1

- Un servidor Web distribuye su tiempo en:
- Cómputo: 40%

- E/S: 60%
- Si se sustituye por otra máquina que puede hacer el cómputo 10 veces más rápido, ¿Cuál es el speedup global?

$$S = \frac{1}{\frac{0.4}{0.6 + \frac{0.64}{10}}} = \frac{1}{1} = 1.5625$$

Caso 2

- Una aplicación tiene una parte paralelizable que consume el 50% del tiempo.
- Si se asume que se puede paralelizar esta parte completamente con 32 procesadores, ¿cuál será el máximo *speedup*?

$$S = \frac{1}{0.5} = \frac{1}{0.515625} = 1.9393$$

$$0.5 + \frac{0.5}{32}$$

El speedup de esta aplicación nunca será superior a 2

Reflexiones sobre la Ley de Amdahl

- Una mejora es más efectiva cuanto más grande es la fracción de tiempo en que ésta se aplica
- Para mejorar un sistema complejo hay que optimizar los elementos que se utilicen durante la mayor parte del tiempo (caso más común)
- Campos de aplicación de las optimizaciones
- Dentro del procesador: la ruta de datos (*data path*)
- En el juego de instrucciones: la ejecución de las instrucciones más frecuentes
- En el diseño de la jerarquía de memoria, la programación y la compilación: hay que explotar la localidad de las referencias
  - El 90% del tiempo se está ejecutando el 10% del código

Planteamiento de Gustafson

- La Ley de Amdahl enfatiza el aspecto más negativo del procesamiento paralelo.
- Sin embargo:

- Las máquinas paralelas se usan para resolver grandes problemas (meteorología, biología molecular…).
- Un computador secuencial nunca podría ejecutar un gran programa paralelo.
- No tendría capacidad para ello.

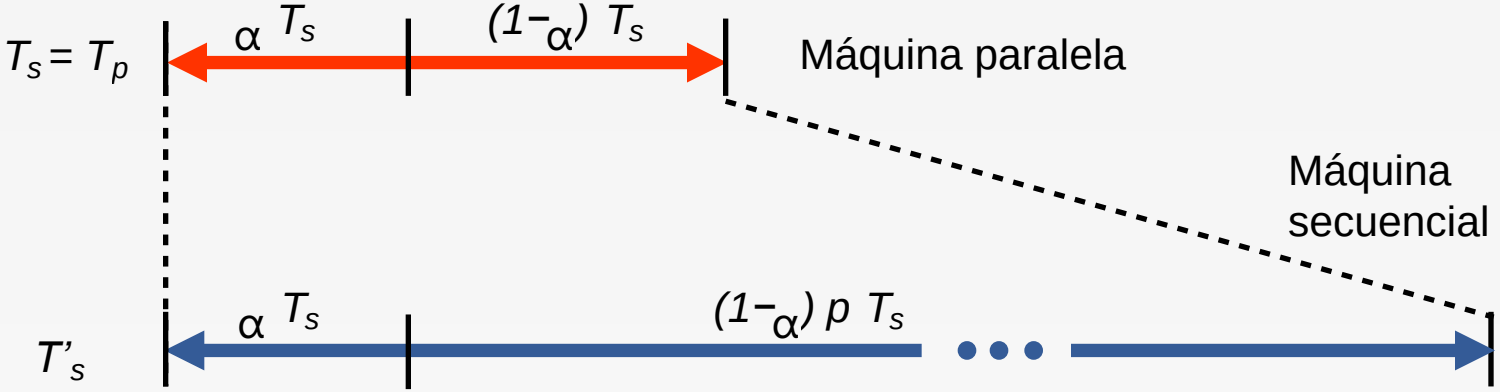
$$S = \frac{T_{sp}}{T_{sp}} = \frac{\text{Tiempo en máquina secuencial}}{\text{Tiempo en máquina paralela}}$$

La aceleración proporcional

- La cantidad de trabajo que se puede hacer en paralelo varía linealmente con el número de procesadores

- Con más procesadores se pueden acometer problemas de mayor coste computacional

$$S_p = \frac{T_s'}{T_p} = \frac{\alpha T_s + (1-\alpha)pT_s}{T_s} = p + \alpha(1-p)$$



Efectos de la Ley de Gustafson

- La ley de Gustafsonasume que la parte secuencial (no paralelizable) disminuye con el tamaño del problema.
  - Cuando el problema crece se puede aproximar el paralelismo lineal (S≈p).
- El paralelismo permite atacar problemas mayores.

Factores

- Factores que influyen en el tiempo de ejecución de un programa:

- el **lenguaje** de programación (el programador)
- la **máquina**
- el **compilador** (opciones)
- los **tipos de datos**
- los **usuarios** que estén trabajando en el sistema

Análisis de algoritmos

- Estudio del algoritmo **a priori**

Antes de hacer el programa correspondiente.

Sirve para identificar si el algoritmo es adecuado para el problema, o para seleccionar entre varios algoritmos.

También sirve para determinar el tamaño de los problemas a resolver en función de las limitaciones de tiempo y memoria.

Análisis de algoritmos

- Estudio **a posteriori**

  Tras haber hecho el programa.

  Sirve para comparar entre dos programas según el tamaño de entrada.

  También para encontrar fallos o posibles mejoras de un programa.

Análisis de algoritmos

- Estudio **teórico** (a priori o posteriori) y estudio **experimental** (a posteriori).

- Tipos de estudios teóricos:

  Tiempo de ejecución (ej. ordenación)

  - caso más favorable, cota inferior: $t_m(n)$
  - caso más desfavorable, cota superior: $t_M(n)$
  - caso promedio: $t_p(n)$

$$t_p(n) = \sum_{\sigma \in S} t(n,\sigma)p(\sigma)$$ donde:

n es el tamaño de la entrada

σ es una entrada de las S posibles entradas

Análisis de algoritmos

- Tipos de estudios teóricos:

Ocupación de memoria

- caso más favorable, cota inferior: $m_m(n)$
- caso más desfavorable, cota superior: $m_M(n)$
- caso promedio: $m_p(n)$

$$m_p(n) = \sum_{\sigma \in S} m(n,\sigma) p(\sigma)$$

donde:

n es el tamaño de la entrada

σ es una entrada de las S posibles entradas

Análisis de algoritmos

- **Conteo de instrucciones**
  - decidir **qué instrucciones/operaciones** (flop) se quieren contar.
  - asignar **costes** a instrucciones de cada **tipo.**
  - una **función**: coste de las instrucciones que la componen.

- **bucles**: mediante sumatorios o cotas superior e inferior si no se conoce el número de veces que se ejecutará.
- **bifurcaciones**: contar el número de veces que pasa por cada rama, o establecer una cota superior (rama más costosa) o una inferior (rama menos costosa).

Análisis de algoritmos

- A nivel práctico, a veces interesa no perder la información de las **constantes del término de mayor** orden:

$$o(f)=\{\lfloor te: N \to \Re_+ / \lim_{n\to\infty} \overline{tef}((nn))=1\rfloor\}$$

- Algunas relaciones entre órdenes:

$$O(1)\subset O(\log n)\subset O(\sqrt{n})\subset O(n)\subset O(n\log n)\subset$$

$$\subset O(n\log n\log n)\subset O(n^2)\subset O(n^3)\subset \ldots \subset O(2^n)\subset O(n!)\subset O(n^n)$$

Análisis de algoritmos

- Factores que afectan al **tiempo de ejecución** de un programa paralelo:

$$t(n,p) = t_{calc}(n,p) + t_{com}(n,p) + t_{overhead}(n,p) - t_{overlapping}(n,p)$$

Estimación del tiempo de ejecución real

$t_{calc}(n, p)$ ⟶ Conteo de instrucciones

$t_{com}(n, p)$ ⟶ **¿?**

Análisis de algoritmos

- Tiempo de comunicación punto a punto entre dos procesadores:

$$t_{com} = t_s + t_w n$$

- Tiempo de comunicación de un mensaje dividido en paquetes a distancia d:

$$t_{com} = d\left(t_s + t_w L_{paq}\right) + \left\lceil \frac{L}{L_{menspaq}} - 1 \right\rceil \left(t_s + t_w L_{paq}\right)$$

- En general, conviene agrupar mensajes (full duplex?, red conmutada?, Ethernet…)

Análisis de algoritmos

- **Ejemplo**: **suma de n números**

```
s = a[0];
for(i=1; i<n;
i++) s = s +
a[i];
```

- Tiempos de la versión secuencial:

- conteo de instrucciones: $t(n) = t_{calc}(n) = 2n - 1$
- conteo de **operaciones**: $t(n) = t_{calc}(n) = n - 1$

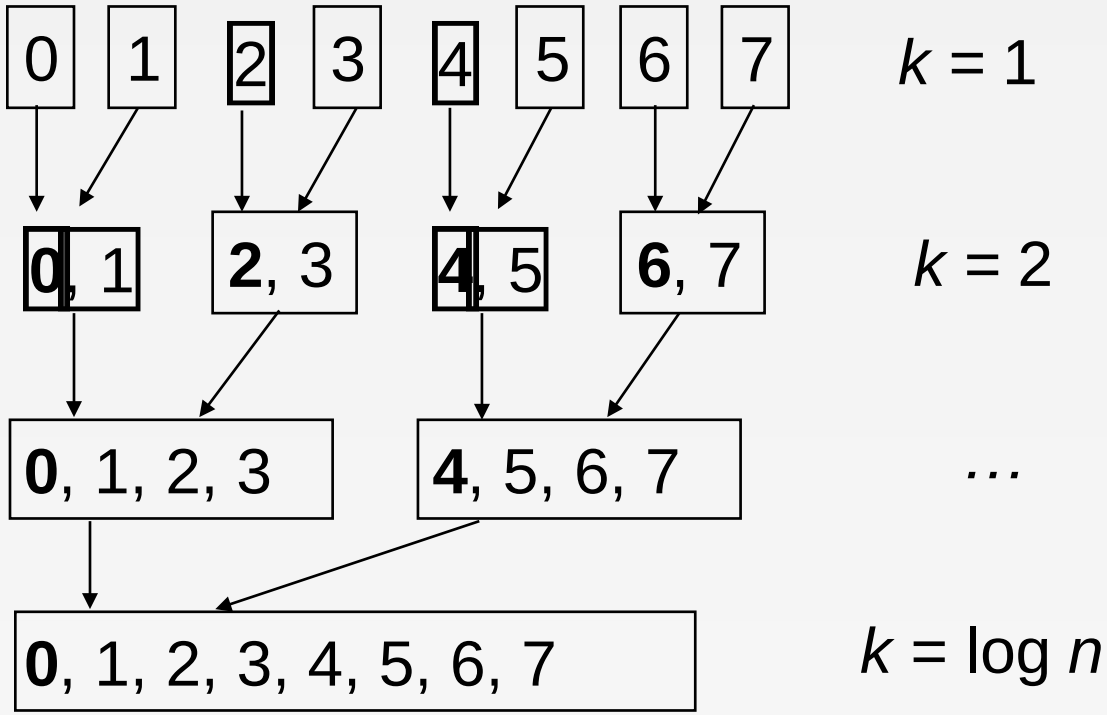
Análisis de algoritmos

- Ejemplo: suma de n números (memoria compartida)
  - una versión paralela con n/2 procesos

```
doall pid = 0, n/2-1
{ ini = 2 * pid;
    des = 1; act
    = true;
    for (k=1; k++; k <= log n)
    { if (act) { a[ini] = a[ini] +
        a[ini+des]; des = des * 2;
        }
        if ((i mod des)!=0) act = false;
    }
}
```

Análisis de algoritmos

- Ejemplo: suma de n números (memoria compartida)
  - una versión paralela con n/2 procesos (memoria compartida):



Análisis de algoritmos

- Ejemplo: suma de n números (memoria compartida)

- Tiempos de la versión paralela (mem. compartida):
  - conteo de instrucciones: $t_{calc}(n, n/2) = 3 + 6 \log n$
  - conteo de **operaciones**: $t_{calc}(n, n/2) = \log n$

  (+ sincronización tras cada iteración)

- Problemas:
  - distribución del trabajo entre procesos
  - *overheads* = variables auxiliares, comprobaciones… - ley de Amdahl

Análisis de algoritmos

- Ejemplo: suma de n números (memoria distribuida)

- una versión paralela con n/2 procesos

```
doall Pi, i = 0, n/2-1
    { des = 1; act =
    true;
    for (k=1; k++; k<=log n -1) {
        if (act) { a = a + b; des =
        des  *  2;  if  ((i  mod
        des)!=0) { act = false;
                Envia (a, Pi-des/2);
        } else Recibe (b, Pi+des/2);
        }
    }
    if (i = 0) a = a + b;
}
```

Análisis de algoritmos

- Ejemplo: suma de n números (mem. distribuida)

- Tiempos de la versión paralela (mem. distribuida):
  - instrucciones: $t_{calc}(n, n/2) = 4+6\ (\log n\ -1)$ -
    operaciones: $t_{calc}(n, n/2) = \log n$
  - comunicación: $t_{com}(n, n/2) = (\log n\ -1)\ (t_s + t_w)$
    (suponiendo comunicaciones directas y en paralelo)
- Problemas:
  - añade la comunicación y su gestión, cuyo coste puede influir significativamente

Análisis de algoritmos

- Algunas conclusiones:
  - No tiene sentido suponer p ilimitado para una entrada constante (eliminar la restricción n = 2p), **n y p deben ser independientes.**
  - No tiene sentido utilizar programación paralela para resolver **problemas pequeños**. Mejor resolverlos secuencialmente. En el ejemplo, el coste es lineal, y, por tanto, no es adecuado.
  - Dependiendo de la **plataforma,** un programa derivado de un algoritmo puede proporcionar unas prestaciones muy diferentes.

- Introducción al paralelismo
- Evaluación del rendimiento: métricas y análisis
- **Metodologías del desarrollo de programas paralelos**
- Esquemas de algoritmos paralelos

Metodología de desarrollo

- Es diferente **paralelizar** un algoritmo o programa secuencial, que **programar en paralelo** una aplicación desde el comienzo.

- En el primer caso, interesa detectar aquellas partes del código con un mayor coste computacional.

Lo más habitual es utilizar trazas, *timers*, *profiling*, etc., y ejecutar en paralelo aquellas partes que ofrecen un buen rendimiento (por ejemplo, paralelismo incremental de OpenMP).

Metodología de desarrollo

- En el segundo caso, se empieza analizando las características de la propia aplicación, para determinar el/los **algoritmos paralelos** más adecuados.

  OJO: conviene partir de un buen algoritmo ya optimizado (¡no hay que reinventar la rueda!).

- Aunque no hay un “camino único”, se suele recomendar utilizar un determinado procedimiento o metodología.

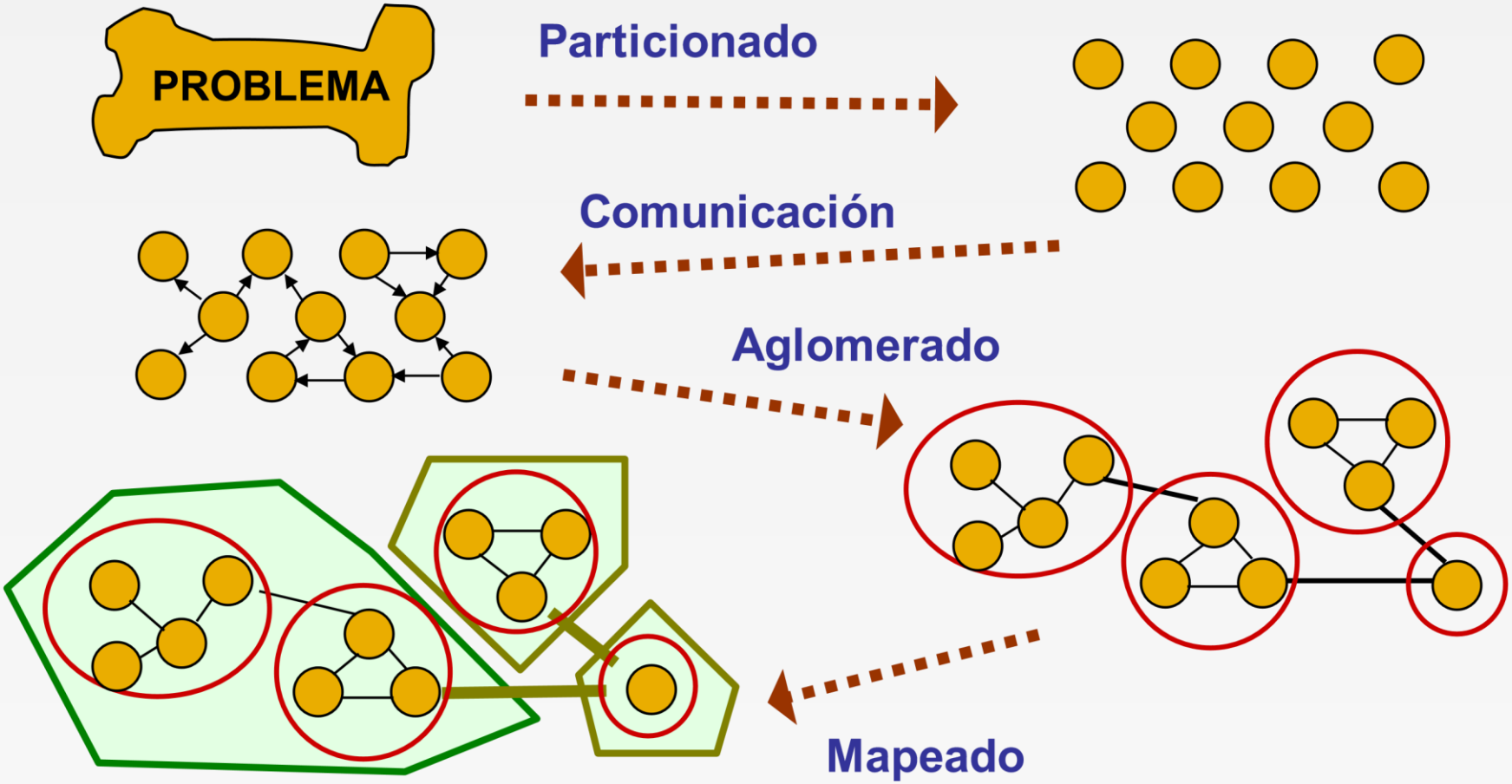
Metodología de desarrollo

- Otra diferencia entre la programación secuencial y la paralela es la forma en que los módulos que componen una aplicación se pueden ensamblar: - Composición **secuencial**: los módulos se ejecutan secuencialmente.

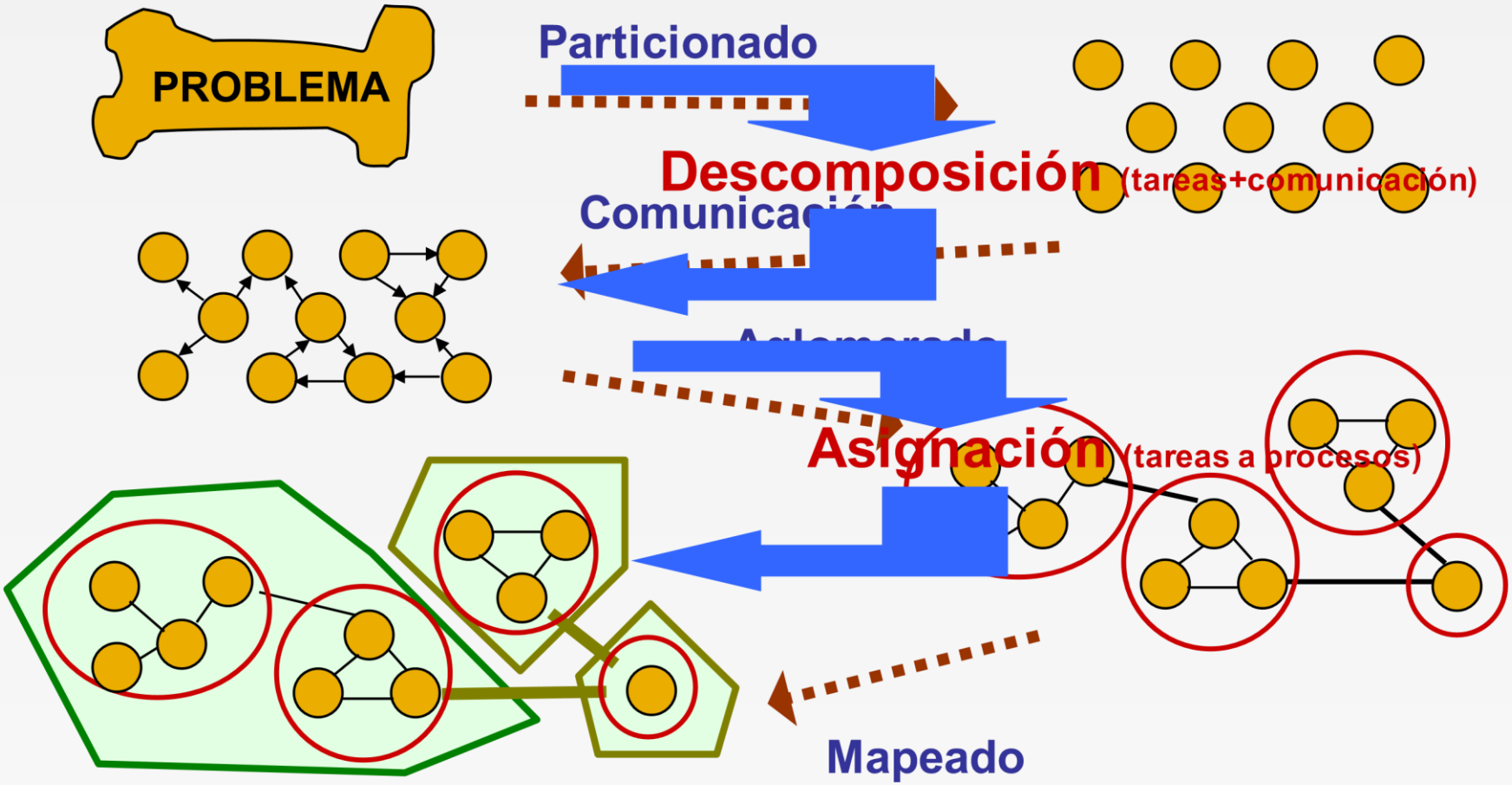
  - Composición **paralela**: diferentes módulos se ejecutan simultáneamente sobre conjuntos disjuntos de procesos (escalabilidad y localidad).

- Composición **concurrente**: diferentes módulos se ejecutan concurrentemente sobre los mismos procesos (solapamiento computación y comunicación).

Modelo PCAM



Modelo PCAM



Descomposición

Descomposición

- La **descomposición** consiste en dividir el cálculo en partes de menor tamaño que vamos a denominar tareas, con el objetivo de ejecutarlas en paralelo.

- Según el tamaño (coste computacional) de las tareas se habla de:
  - granularidad fina (muchas tareas pequeñas).
  - granularidad gruesa (pocas tareas grandes).

- Es deseable obtener un número suficientemente alto de tareas (grano fino) para tener más flexibilidad en la fase de asignación.

- En esta fase es fundamental tener en cuenta las **dependencias** entre las tareas y reflejarlas en un **grafo** de dependencias para poder estimar las necesidades de sincronización y estructura de comunicación que hay entre las tareas.

Descomposición

- Algunas **técnicas de descomposición**

  - Descomposición de dominio (salida/entrada/bloques)
  - Descomposición funcional (flujo de datos)
  - Descomposición recursiva

  - Descomposición exploratoria

- Descomposición especulativa

Descomposición

- **Factores** a considerar en la descomposición
  - Características de las **tareas**

    -- **Replicación de datos/cálculo**

    Conviene evitar que las tareas compartan mucho cálculo o datos del problema.

    -- **Tamaño y localización de los datos asociados a cada tarea**

    Deben ser accesibles por el proceso que ejecuta esa tarea (fase de asignación) y hay que evitar una sobrecarga por cálculo y/o comunicación.

Descomposición

- **Factores** a considerar en la descomposición

- Patrones de comunicación entre tareas

-- Patrones locales/globales

Un patrón de comunicación se dice que es local cuando cada tarea interacciona sólo con un subconjunto pequeño de tareas (vecinas).

Se utiliza el grafo de dependencias para determinar las necesidades de comunicación o sincronización.

Descomposición

- **Factores** a considerar en la descomposición

- Patrones de comunicación entre tareas

-- Patrones regulares/irregulares

Se habla de patrón regular cuando la comunicación presenta una **topología espacial**. Los patrones regulares simplifican la etapa de asignación y programación en el modelo de paso de mensajes.
–jacobi– la comunicación se realiza entre las cuatro tareas vecinas en la malla.
–integración numérica– presenta un patrón de comunicación irregular.

Descomposición

- **Factores** a considerar en la descomposición

- Patrones de comunicación entre tareas

-- Datos compartidos de lectura/lectura+escritura

-- Comunicación unilateral/bilateral

En la comunicación unilateral la comunicación de una tarea con otra tarea (productora) se hace **sin interrumpirla**; sin embargo, en la bilateral la comunicación se hace de forma explícita entre la tarea productora y la tarea que precisa de los datos. En el modelo de paso de mensajes la comunicación unilateral se convierte en bilateral, y con patrón irregular, la dificultad aumenta (–suma de un subconjunto – con poda).

Asignación

- Tras la fase de descomposición se obtiene un algoritmo paralelo de grano fino, independiente de la plataforma paralela que se vaya a usar para su ejecución.
  La fase de **asignación** es en la que se decide qué tareas **agrupar** y en qué unidades de procesamiento se va a ejecutar cada tarea y en qué orden. Por ello, es en esta fase en la que se tienen en cuenta aspectos de la **plataforma paralela** como: número de unidades de procesamiento, modelo de memoria compartida o paso de mensajes, costes de sincronización y comunicación, etc.

Asignación

- En esta fase se asocian las tareas a los **procesos** (entidades abstractas capaz de ejecutar cálculo) y no a los **procesadores** (entidad física, hardware) para mantener un mayor nivel de abstracción que aumente la flexibilidad del diseño.

  Será en las últimas fases de implementación del algoritmo cuando se asignen los procesos a procesadores (normalmente, uno a uno).

Asignación

- **Objetivos** de la asignación:
  - Reducir el tiempo de **computación**.
  - Minimizar el tiempo de **comunicación**.

- Evitar el tiempo de **ocio** (por mal reparto de carga o por esperas en sincronizaciones/comunicaciones).

- Dos **tipos** generales de esquemas de asignación:
  - Asignación **estática** (técnicas de planificación deterministas).
  - Asignación **dinámica**.

Asignación

- **Asignación estática**

  Cuando se conoce o se puede estimar a priori el coste computacional de las tareas y las relaciones

entre ellas, y, por tanto, se decide a priori qué unidad de proceso ejecutará cada tarea.

El problema de asignación estática óptima es NPcompleto → técnicas heurísticas.

La gran ventaja frente a los dinámicos es que no añaden ninguna sobrecarga en tiempo de ejecución.

Asignación

- **Asignación dinámica**

  Se utiliza cuando las tareas se generan dinámicamente (–integración numérica–), o cuando

no se conoce a priori el tamaño de las tareas (–suma del subconjunto– con poda).

Es más compleja que la estática: ¿cómo redistribuir el trabajo en tiempo de ejecución?

Asignación

- **Esquemas de asignación estática**

  Algunos se centran en métodos para **descomposición de dominio**(distribuciones de matrices por bloques, diferencias finitas en una malla bidimensional…).

Otros se centran en métodos sobre **grafos** de dependencias estáticas (normalmente obtenidas mediante descomposición funcional o recursiva).

Asignación

- **Esquemas de asignación dinámica**

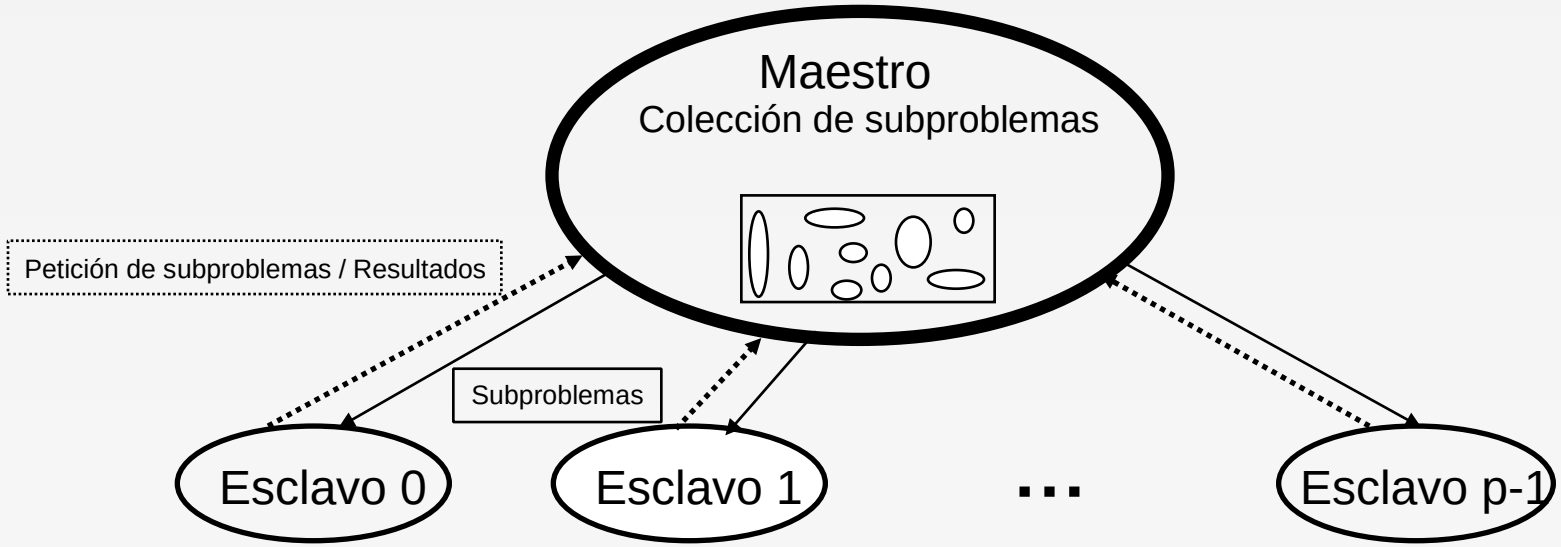
  Cuando se aplican descomposiciones recursivas o exploratorias suele ser más adecuado utilizar esquemas de asignación dinámica.

  Uno de los problemas a afrontar es la necesidad de un mecanismo de **detección de fin** de trabajo.

Asignación

Esquemas de asignación dinámica

- Esquemas centralizados: **maestro – esclavos**



Asignación

Esquemas de asignación dinámica

- Esquemas centralizados: **maestro – esclavos**

Adecuado con un **número moderado** de esclavos y cuando el coste de ejecutar los subproblemas es alto comparado con el coste de obtenerlos.

Algunas estrategias para mejorar la eficiencia (reduciendo la interacción entre maestro y esclavos):

- Planificación por bloques.
- Colecciones locales de subproblemas.
- Captación anticipada (solapar cálculo y comunic.).

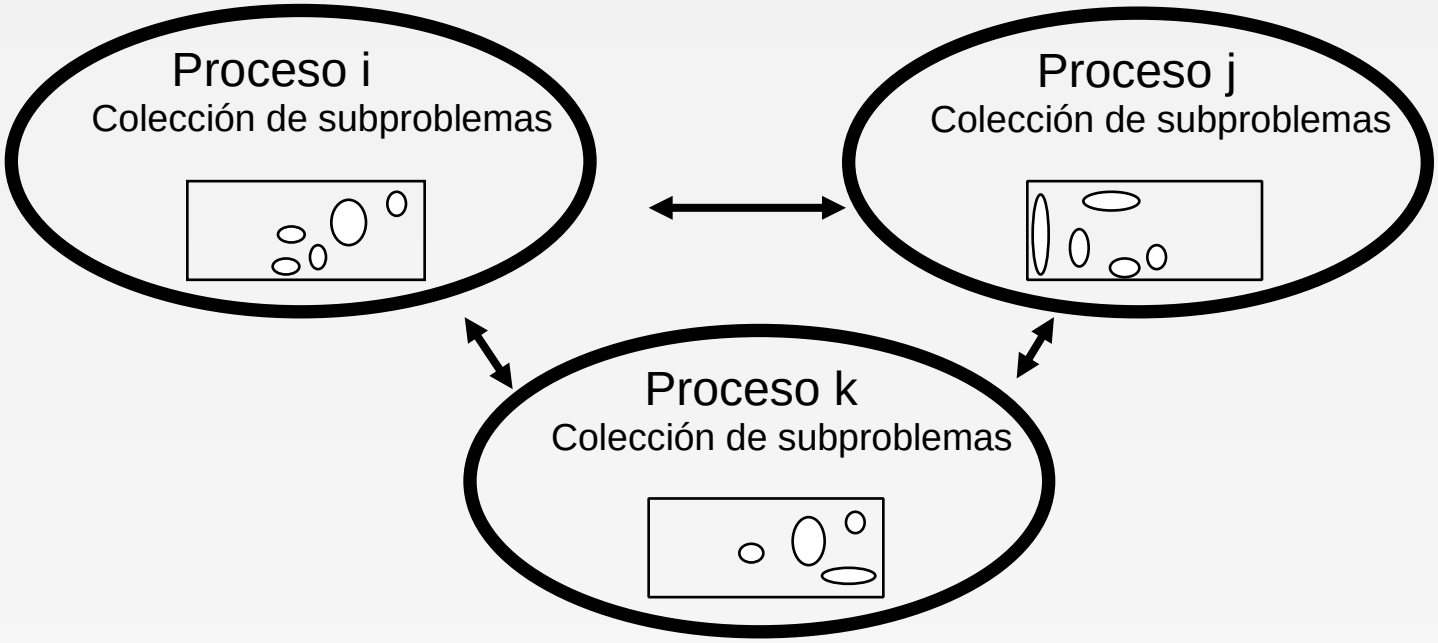
Fácil determinación de fin (en el maestro).

Asignación

- **Esquemas de asignación dinámica**

  - Esquemas completamente **descentralizados**

    No existe un proceso maestro; los subproblemas se encuentran distribuidos por todos los procesos.



Asignación

- **Esquemas de asignación dinámica** -

  Esquemas completamente **descentralizados**

El equilibrio de carga es más difícil y requiere poder transferir subproblemas a procesos ociosos (cuántos?):

- Transferencia iniciada por el receptor
  (sondeo aleatorio, sondeo cíclico…)
- Transferencia iniciada por el emisor (cargas bajas)

La detección de fin es más compleja (algoritmo de terminación de Dijkstra…)

Contenido

- Introducción al paralelismo
- Evaluación del rendimiento: métricas y análisis
- Metodologías del desarrollo de programas paralelos
- **Esquemas de algoritmos paralelos**

- Un **esquema algorítmico** es un patrón común a la resolución de distintos problemas.

  En el caso paralelo se encuentran **versiones paralelas** de esquemas secuenciales, esquemas que son **propiamente paralelos** (maestro-esclavo, granja de procesos…) o esquemas que son adecuados para ciertos sistemas o topologías paralelas o para esquemas concretos de descomposición/asignación vistos en el apartado anterior.

Esquemas de algoritmos paralelos

- **Paralelismo de datos** (bucles)

Normalmente en memoria compartida trabajando distintos *threads* o procesos sobre una estructura de datos común pero en zonas diferentes. Algunos ejemplos:

- Suma de n números
- Ordenación por rango
- Evaluación polinomial
- Multiplicación matriz-vector
- Integración numérica

Esquemas de algoritmos paralelos

- **Particionado de datos**

Es una especie de paralelismo de datos con paso de mensajes.

La diferencia está en que no sólo se reparte el trabajo sino que hay que distribuir los datos entre los procesos. Por tanto, hay que tener en cuenta las necesidades de comunicación entre procesos (y la topología sobre la que se trabaja).

Los mismos ejemplos que en el paralelismo de datos se pueden programar mediante el particionado de datos.

Esquemas de algoritmos paralelos

- **Algoritmos relajados o paralelismo “obvio”**

(embarrassingly parallel)

El cálculo de cada proceso es independiente por lo que no hay sincronización ni comunicación, excepto, quizás, al comienzo y al final.
Algunos ejemplos: suma de n números, ordenación por rango, multiplicación matrices, etc.

Otros ejemplos

Esquemas de algoritmos paralelos

- **Algoritmos relajados o paralelismo "obvio"**

  Procesado de imagen

  > desplazamiento: $(x, y) \rightarrow (x+\Delta x, y+\Delta y)$

> escalado: $(x, y) \rightarrow (x*Sx, y*Sy)$

> rotación: $x \rightarrow x \cos \theta + y \operatorname{sen} \theta$

$y \rightarrow -x \operatorname{sen} \theta + y \cos \theta$

> recorte (*clipping*): borrar fuera de un área

Reparto de tareas: por filas, columnas, bloques...

Esquemas de algoritmos paralelos

Algoritmos relajados o paralelismo "obvio"

Cálculo de funciones tipo "Mandelbrot"

> Iteración de una función con los puntos de una determinada área hasta que se cumpla cierta condición.
>
> Por ejemplo: $Z_{k+1} = Z_k^2 + C$ (complejos)

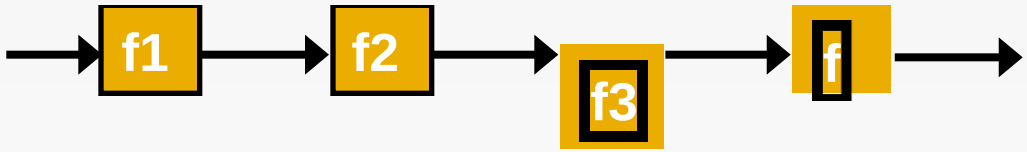
hasta que $|Z_k| > 2$

Reparto de puntos independientes; **la asignación de tareas podría ser dinámica**.

Esquemas de algoritmos paralelos

- **Computación *pipelineo* segmentada**

El problema se divide en una serie de etapas. Cada etapa se ejecuta, por ejemplo, en un procesador, y pasa resultados a la siguiente.



Es un tipo de descomposición funcional, relacionado con la repetición del mismo proceso sobre una serie larga de datos (p.e., en tiempo real: procesado de vídeo).

Esquemas de algoritmos paralelos

Computación *pipelineo* segmentada

Condiciones:
-- que se ejecute más de una vez el mismo problema.
-- que se procese una serie larga de datos.
-- que se pueda pasar datos a la siguiente fase mucho antes del final del cálculo de cada fase.
-- que el tiempo de proceso asociado a cada fase sea similar (*load balancing*).

Topología ideal: cadena / anillo.

Esquemas de algoritmos paralelos

- **Computación *pipelineo* segmentada**

  Normalmente no se genera un número muy elevado de procesos.

  Algunos ejemplos:
  -- procesado de señal (sonido, vídeo...)
  -- simulaciones de procesos segmentados (computación)

Esquemas de algoritmos paralelos

- **Paradigma maestro-esclavo**

Thread/proceso maestro que se encarga de poner en marcha los esclavos dándoles trabajo y recopilando las soluciones que van calculando.

- **Granja de procesos**

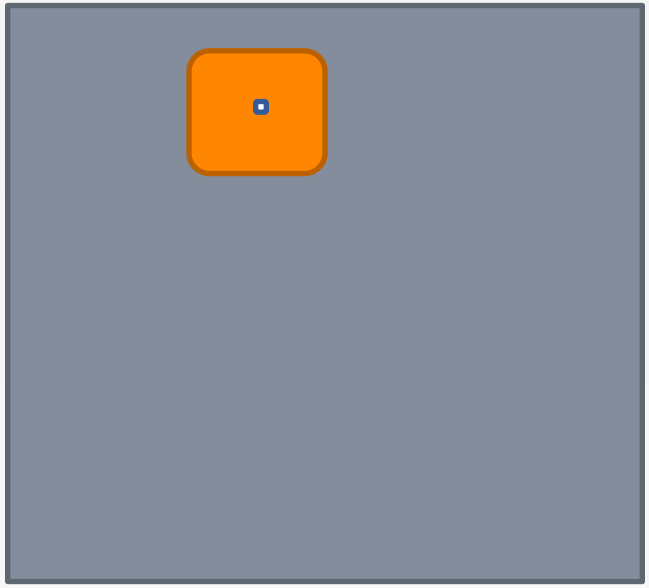
  Conjunto de procesos que trabaja de manera conjunta pero independiente en la resolución de un problema (similar al de los algoritmos relajados). Similar a maestro-esclavo, pero los procesos hacen un trabajo idéntico.

- **Trabajadores replicados**

  Los workers son capaces de generar nuevas tareas que pueden ejecutarlas ellos mismos y otros workers. Es una versión descentralizada (control de finalización!).

Ejemplo práctico: procesamiento de imágenes

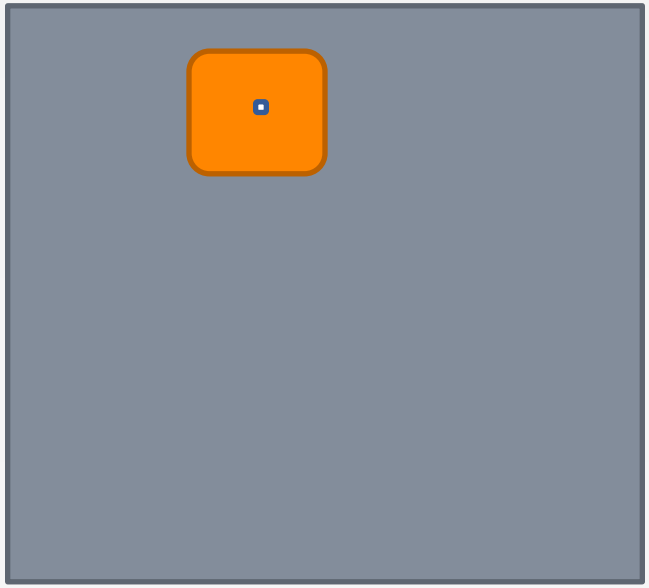
- Se quiere procesar un algoritmo de filtrado sobre una imagen.
- El filtro consiste en calcular la mediana sobre la vecindad de cada pixel.
  - El pixel almacena tres componentes del color
  - ¿Versión paralela?



Ejemplo práctico: procesamiento de imágenes

- Se quiere procesar un algoritmo de filtrado sobre una imagen.

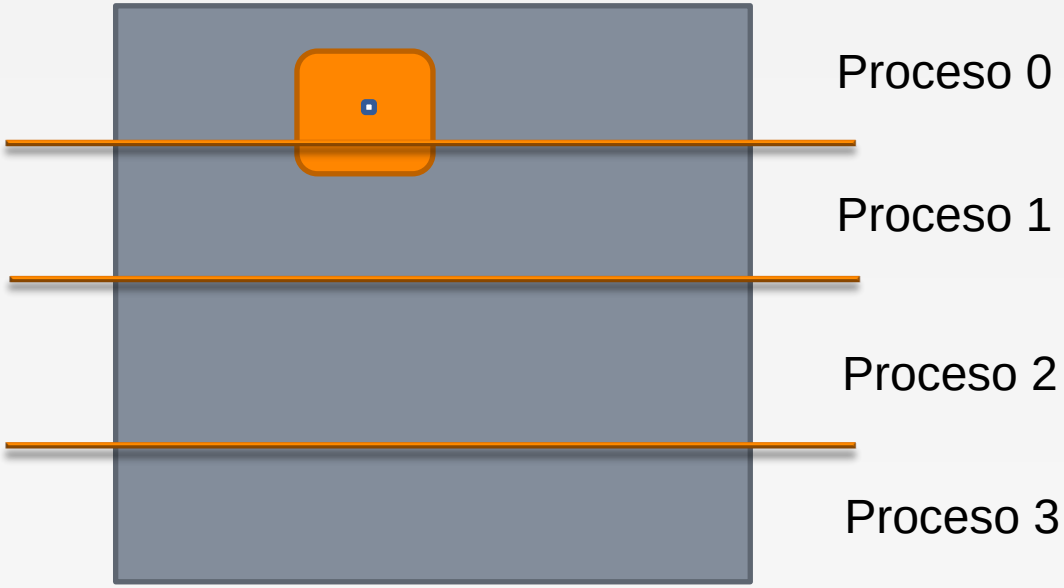
- El filtro consiste en calcular la mediana sobre la vecindad de cada pixel.
  - El pixel almacena tres componentes del color
  - ¿Versión paralela?



Ejemplo práctico: procesamiento de imágenes

- Se quiere procesar un algoritmo de filtrado sobre una imagen.

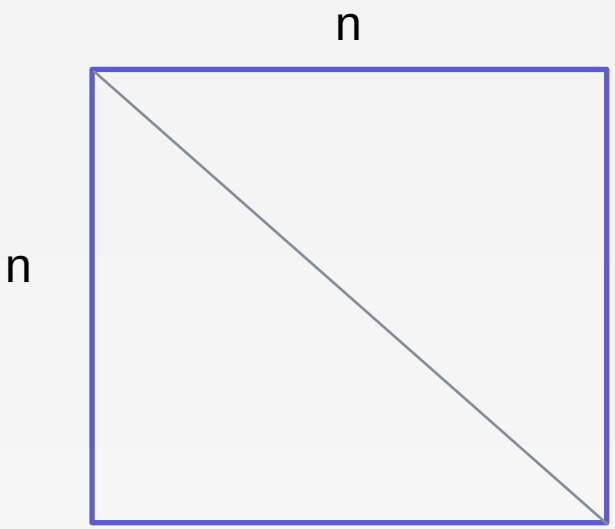
- El filtro consiste en calcular la mediana sobre la vecindad de cada pixel.
  - El pixel almacena tres componentes del color
  - ¿Versión paralela?



Ejemplo práctico: N-cuerpos

- El problema de los n-cuerpos trata de determinar los movimientos individuales de un grupo de partículas materiales (en sus orígenes, un

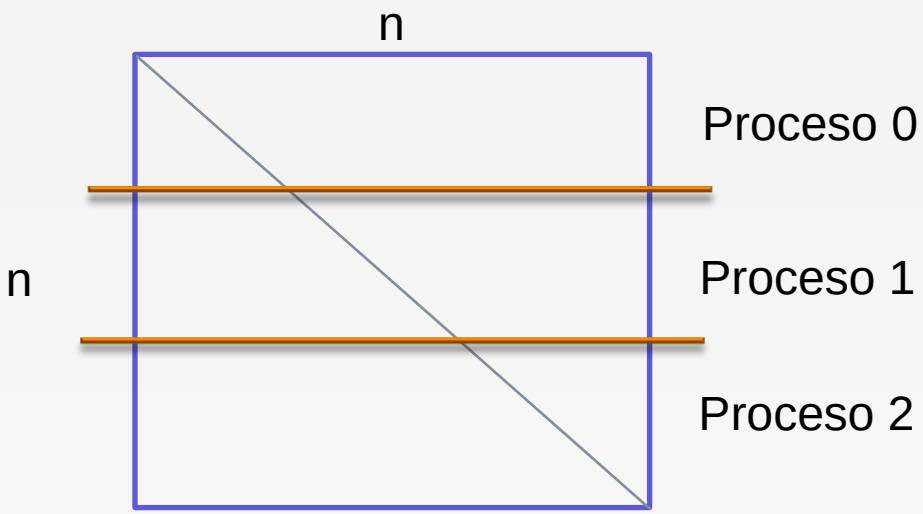
conjunto de objetos astronómicos) que interactúan mutuamente según las leyes de la gravitación universal de Newton..



Ejemplo práctico: N-cuerpos

- El problema de los n-cuerpos trata de determinar los movimientos individuales de un grupo de partículas materiales (en sus orígenes, un

conjunto de objetos astronómicos) que interactúan mutuamente según las leyes de la gravitación universal de Newton..



Referencias

Programación Paralela

- F. Almeida et al: *Introducción a la programación paralela*. Paraninfo, 2008.
- I. Foster: *Designing and Building Parallel Programs*. Addison Wesley (version on-line).
- B. Wilkinson, M. Allen: *Parallel Programming Techniques and Applications…* Pearson, 2005 (2. ed.).

- M.J. Quinn: *Parallel Programming in C with MPI and OpenMP.* McGraw Hill, 2003.
- A. Grama, A. Gupta, G. Karypis, V. Kumar: *Introduction to Parallel Computing* (2. ed.). Pearson, 2003.