Arquitectura de Computadores

Parte 2

Carmen Azorín Martí

Lección 4: Herramientas, estilos y estructuras en programación paralela

Problemas que plantea la programación paralela al programador. Punto de partida

Los problemas respecto a la programación secuencial:

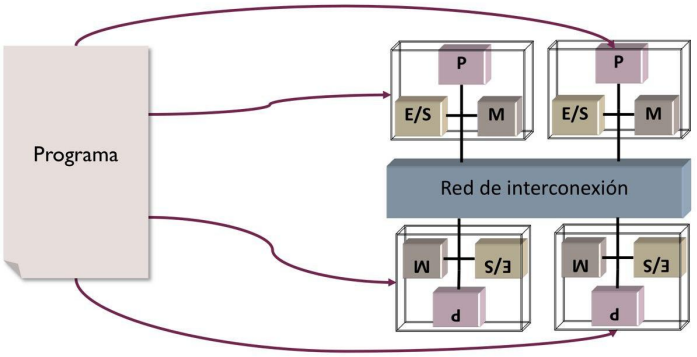
1. División en tareas
2. Asignación de tareas en procesos
3. Asignación a procesadores
4. Sincronización y comunicación

Para desarrollar programas paralelos el programador puede usar compiladores que extraen paralelismo automáticamente (aunque no son lo suficientemente eficientes) o pueden usar métodos por su cuenta.

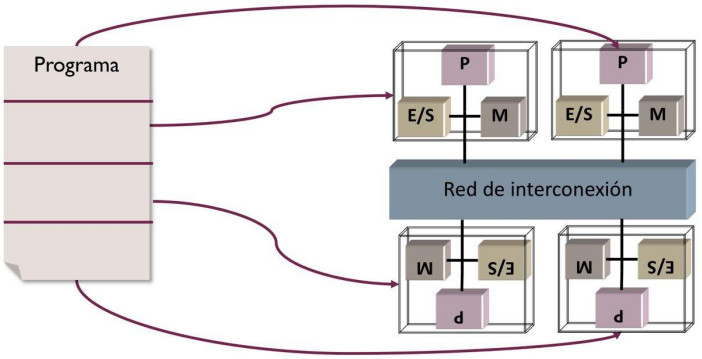
Cuando se plantea obtener una versión paralela de una aplicación, se puede utilizar como punto de partida un código secuencial que resuelva el problema.
Otra posibilidad, sería partir de la definición de la aplicación. Habría que buscar una descripción para el problema que admita paralelización.
Y finalmente, podemos apoyarnos en programas paralelos ya hechos que aprovechen las características de la arquitectura y en bibliotecas de funciones paralelas.

Existen dos modos de programación:

- SPMD (paralelismo de datos): todos los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa. Es decir, único código ejecutado por todos los núcleos pero cada uno lo hace con un conjunto de datos distinto.



- MPMD (paralelismo de funciones): los códigos que se ejecutan en paralelo se obtienen compilando programas independientes. Es decir, un código en secuencial donde distintos trozos se pueden ejecutar en paralelo con sus propios datos.



Herramientas para obtener código paralelo

Las herramientas encargados de obtener programas paralelos deben:

1. Localizar paralelismo
2. Distribuir la carga de trabajo entre procesos (trozos de código)
3. Crear y terminar procesos
4. Comunicar y sincronizar procesos
5. Opcional: asignar procesos a procesadores (se encarga el programador, la herramienta o el SO)

Niveles de abstracción (de mayor a menor):

1. Compiladores paralelos (paralelización automática): sólo se puede hacer a nivel de bucle. Al ser automático, el código es peor
2. Lenguajes paralelos (Java): construcciones del lenguaje + funciones. Nos quita el trabajo de comunicación y sincronización
   API funciones + Directivas (OpenMP): lenguaje secuencial + directivas + funciones. Sin las directivas tendríamos menor nivel de abstracción
3. API funciones (MPI, Pthreads): lenguaje secuencial + funciones

Ejemplo: cálculo de π con OpenMP

```
#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);
#pragma omp parallel
{
  #pragma omp for reduction(+:sum) private(x) \
                                  schedule(dynamic)
  for (i=0;i< intervalos; i++) {
      x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
  }
}
  sum* = ancho;
}
```

Crear y Terminar

Comunicar y sincronizar

Localizar y Asignar

Asignar

Ejemplo: cálculo de π con MPI/C

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

Enrolar

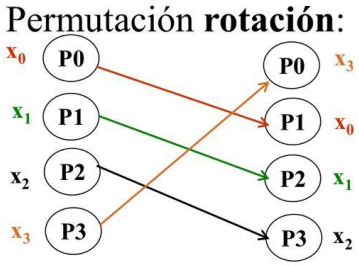
Localizar y Asignar

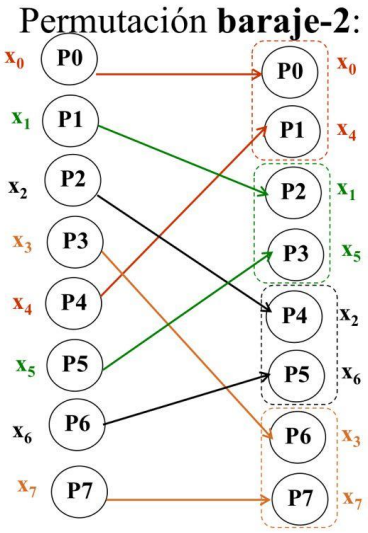
Comunicar/sincronizar

Desenrolar

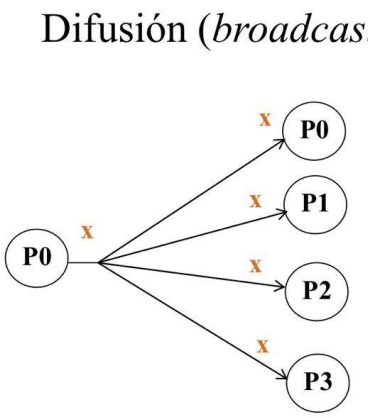
Clasificación de las funciones colectivas en cinco grupos:

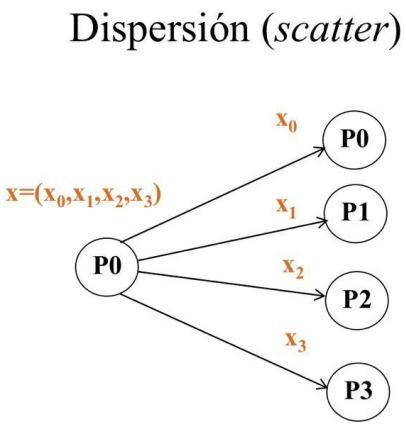
1. Múltiple uno-a-uno: hay componentes del grupo envían un único mensaje y componentes del grupo que reciben un único mensaje.
    a. Permutación: todos los componentes del grupo envían y reciben. Algunos ejemplos son la rotación, el intercambio o el hipercubo.



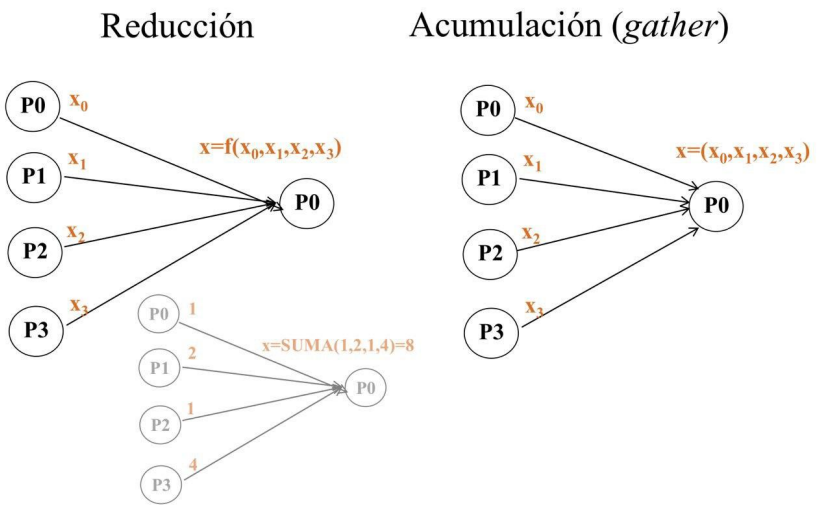


2. Uno-a-todos: un proceso envía y todos los procesos reciben. Es útil para llevar datos iniciales a todos los flujos. Las variantes son:
    a. Difusión: todos los procesos reciben el mismo mensaje
    b. Dispersión (scatter): cada proceso receptor recibe un mensaje diferente.

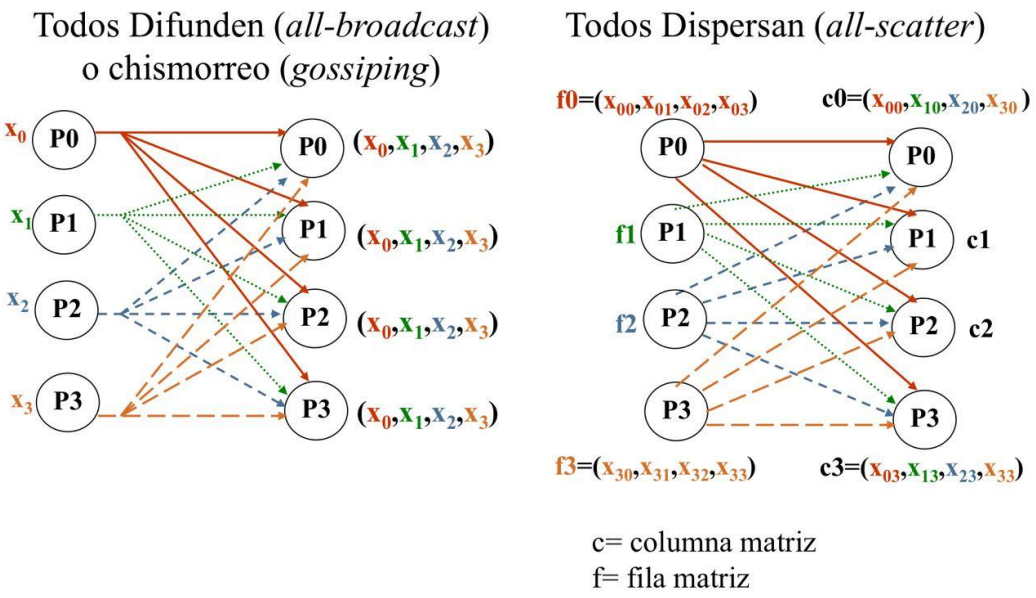




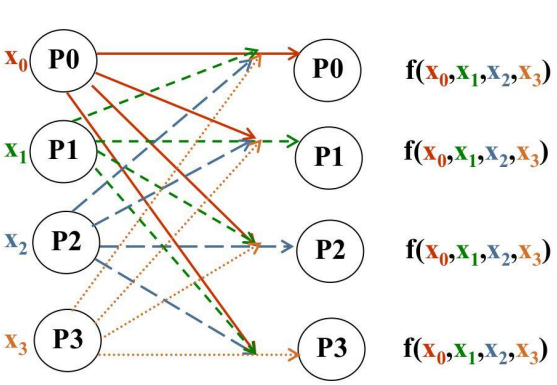
3. Todos a uno: todos los procesos del grupo envían un mensaje a un único proceso. Las variantes son:
    a. Reducción: los mensajes enviados por los procesos se combinan en un solo mensaje mediante algún operador. por ejemplo, se le van mandando datos que tiene que sumar, por tanto, el valor que recibe el proceso receptor es la suma de los mensajes.
    b. Acumulación (gather): los mensajes se reciben de forma concatenada por el receptor.



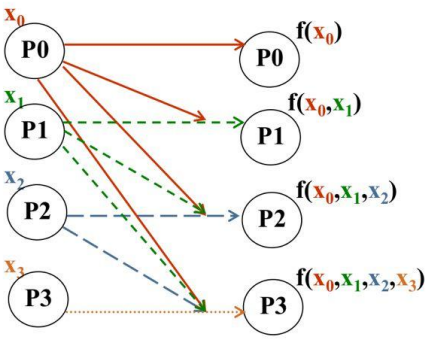
4. Todos-a-todos: todos los procesos del grupo ejecutan una comunicación uno-a-todos. Las variantes son:
    a. Todos-difunden (all-broadcast): todos los procesos realizan una difusión.
    b. Todos-dispersan (all-scatter): los procesos concatenan diferentes transferencias.



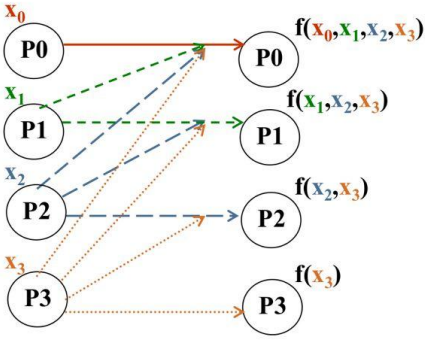
5. Servicios compuestos: combinaciones de los anteriores
    a. Todos combinan: el resultado de aplicar una reducción se obtiene en todos los procesos.
    b. Recorrido (scan): todos los procesos envían un mensaje, recibiendo cada uno de ellos el resultado de reducir un conjunto de estos mensajes.



Recorrido (scan) prefijo paralelo



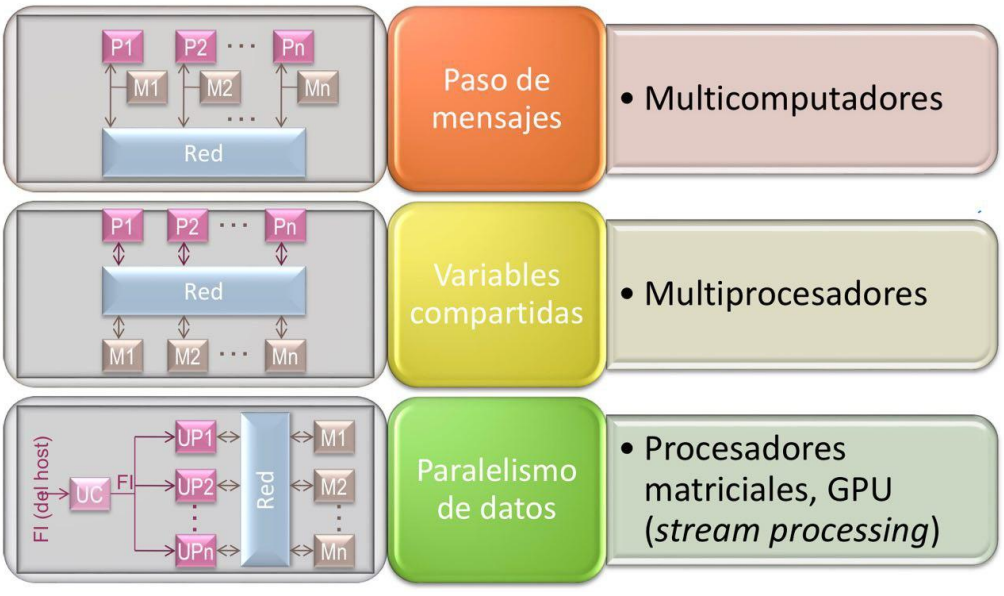
Recorrido sufijo paralelo



Estilos/paradigmas de programación paralela

Existen tres estilos de programación que se utilizan en procesamiento paralelo: paso de mensajes para multicomputadores, variables compartidas para multiprocesadores y paralelismo de datos para procesadores matriciales (SIMD).

- Con paso de mensajes se supone que cada procesador en el sistema tiene su espacio de direcciones propio. Los mensajes llevan datos de uno a otro espacio de direcciones y además se pueden aprovechar para sincronizar procesos.
    - Bibliotecas de funciones (API): MPI
    - Lenguajes de programación: Ada
- Con variables compartidas, los procesadores en el sistema comparten el mismo espacio de direcciones. Para sincronizar, el programador utiliza primitivas que ofrece el software.
    - Lenguajes de programación: Java
    - API (directivas + funciones): OpenMP
    - API (bibliotecas de funciones): C++ con clases thread
- Con paralelismo de datos, las mismas operaciones se ejecutan en paralelo e múltiples unidades de procesamiento de forma que cada unidad aplica la operación a un conjunto de datos distinto,
    - Lenguajes de programación + funciones: Nvidia CUDA
    - API (directivas +funciones - stream processing): OpenACC, OpenMP 4.0



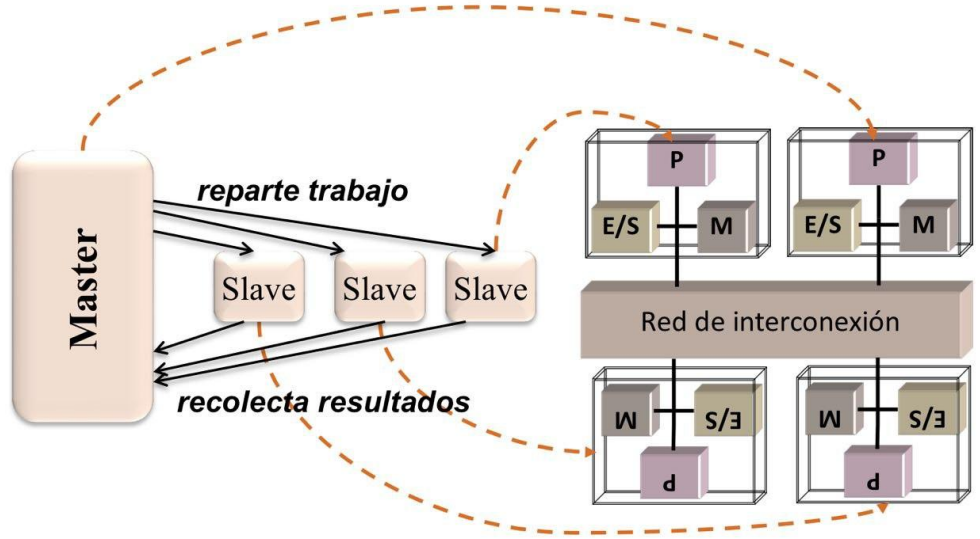
Estructuras típicas de códigos paralelos

Estructuras típicas de procesos en código paralelo:

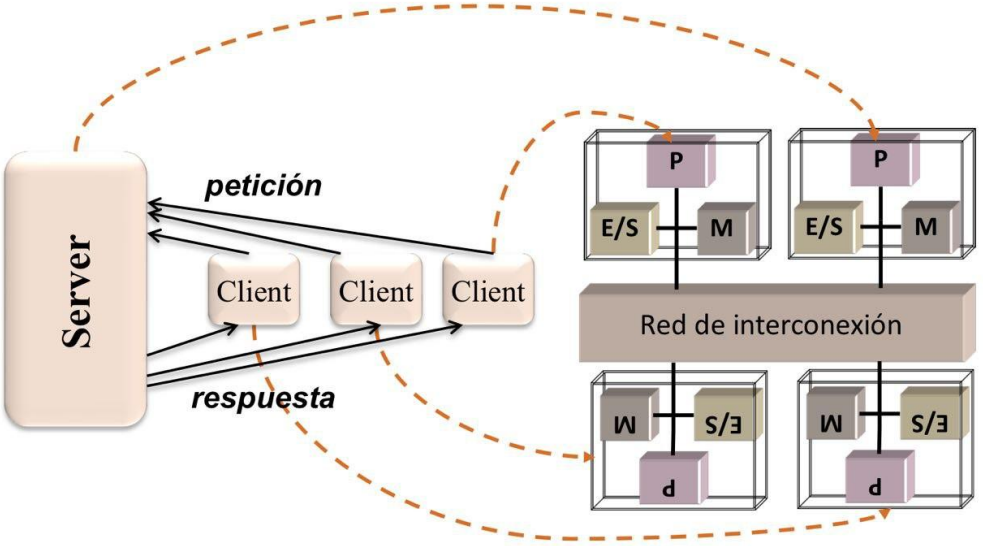
1. Descomposición de dominio o de datos
2. Cliente/servidor
3. Divide y vencerás o descomposición recursiva
4. Segmentación o flujo de datos
5. Master-slave o granja de tareas

Explicación (los rectángulos representan el flujo de instrucciones; los círculos, las tareas o procesos; y las flechas discontinuas representan el paso de datos):

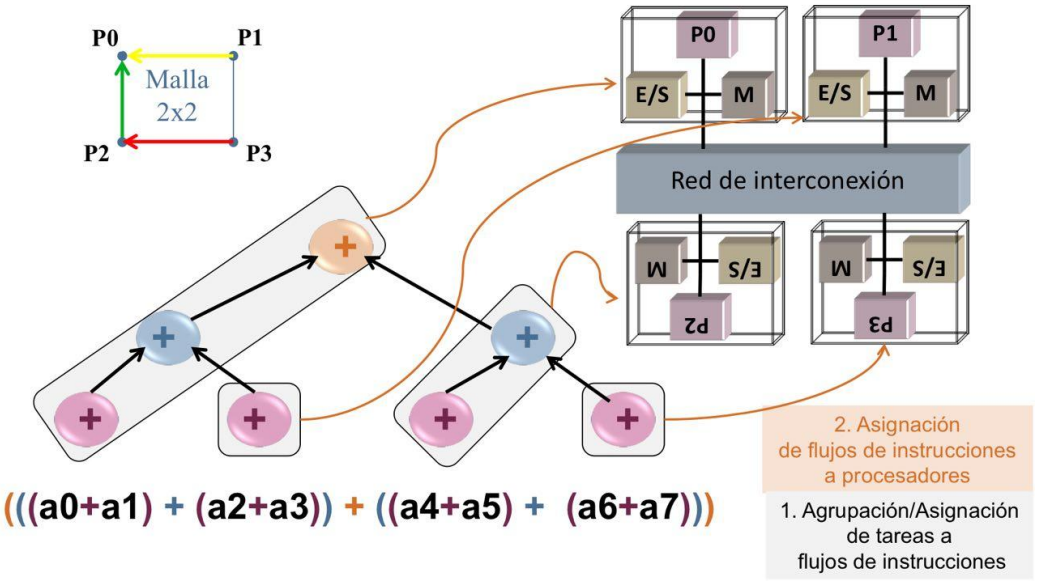
1. Dueño-esclavo:El proceso dueño se encarga de distribuir las tareas de un conjunto entre el grupo de esclavos, y de ir recolectando los resultados parciales que van calculando los esclavos, con los que el dueño obtiene el resultado final.



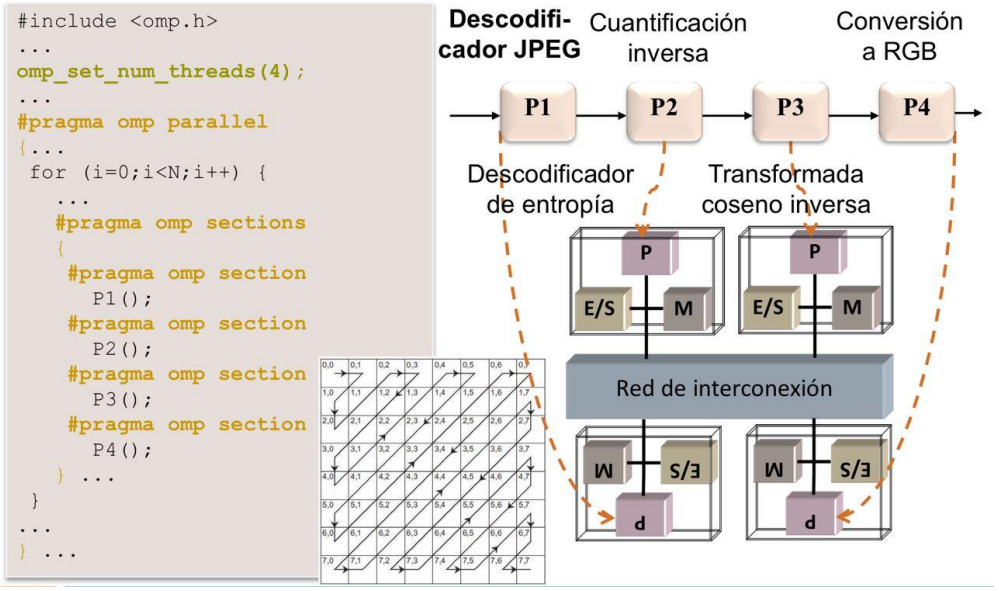
2. Paralelismo de datos: útil para obtener tareas paralelas en problemas en los que se opera con grandes estructuras de datos. La estructura de datos de entrada o la estructura de datos de salida, o ambas, se dividen en partes y se derivan las tareas paralelas.



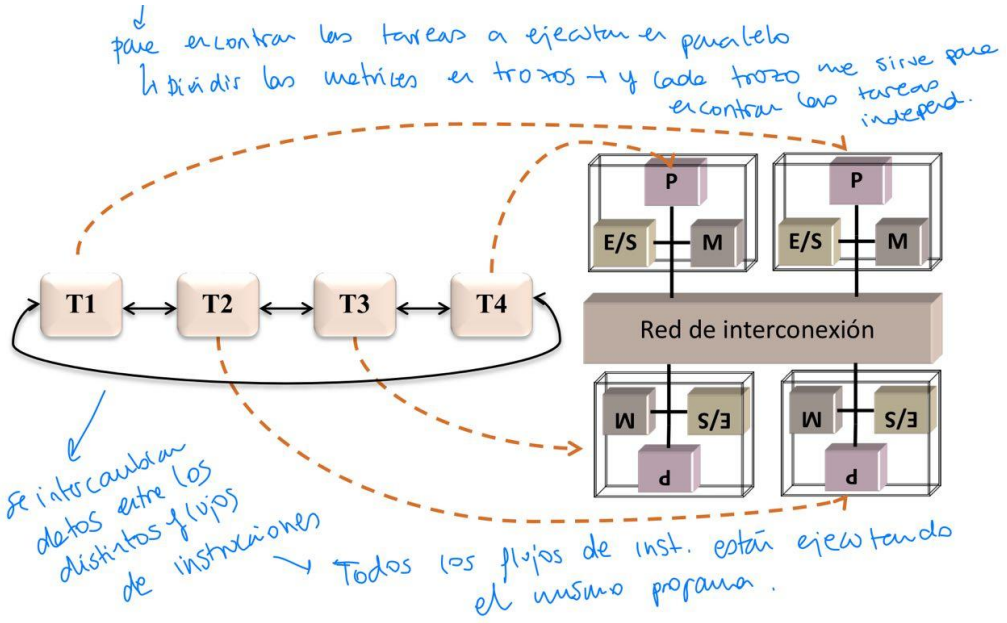
3. Divide y vencerás: se utiliza cuando un problema se puede dividir en dos o más subproblemas, de forma que cada uno se puede resolver independientemente, combinándose los resultados para obtener el resultado final. Por ejemplo, un vector cuyos componentes se van a sumar se puede descomponer recursivamente en dos vectores, hasta llegar a vectores de dos componentes.

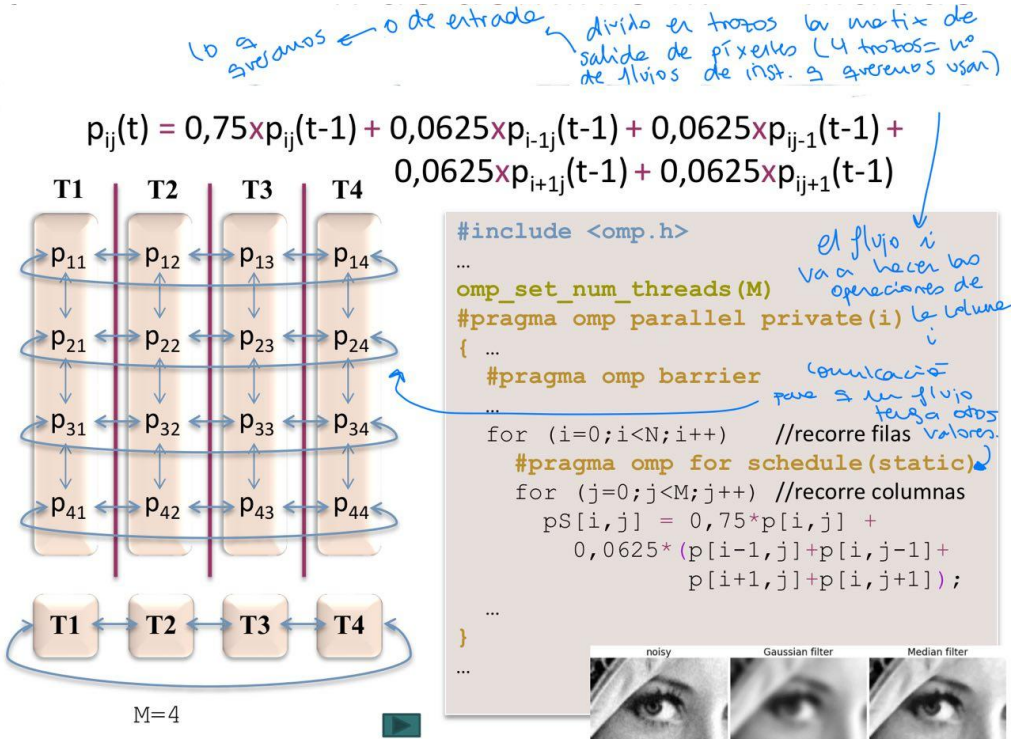


4. Segmentada: utilizada en problemas en los que se aplica a un flujo de datos en secuencia distintas funciones (paralelismo de funciones). La estructura de los procesos y de las tareas es la de un cauce segmentado. Cada proceso ejecuta por tanto distinto código. Caso típico para el modo MPMD.



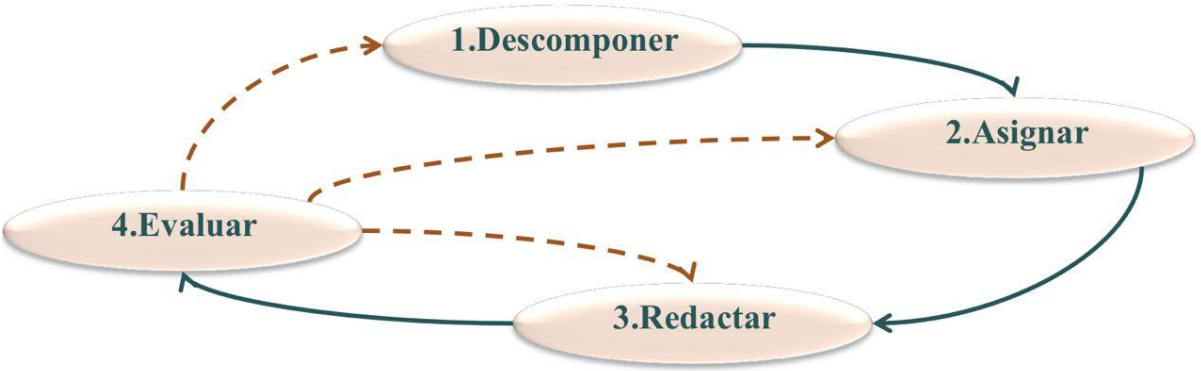
5. Descomposición de dominio: se utiliza para la simulación de fenómenos físicos. Se dividen las matrices en trozos y cada trozo me sirve para encontrar las tareas independientes. (no sé, no lo explica en el libro pero lo explicó en clase)



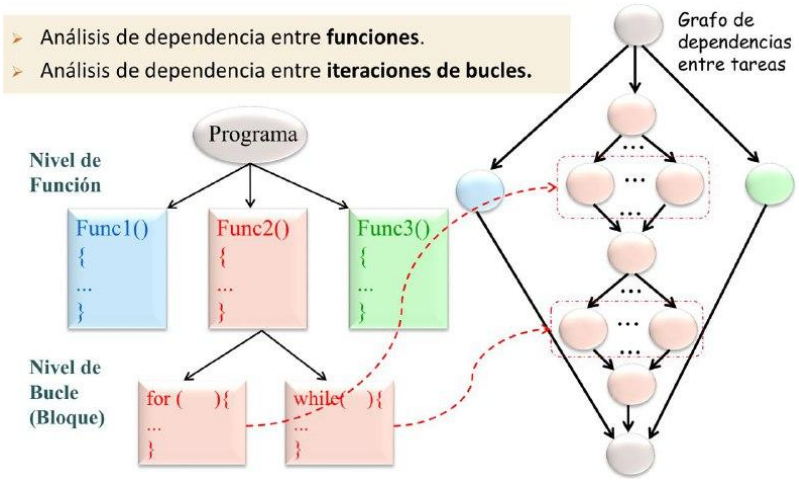


(esos son los apuntes que tengo de clase, suerte entendiendo algo :))

Lección 5: Proceso de paralelización



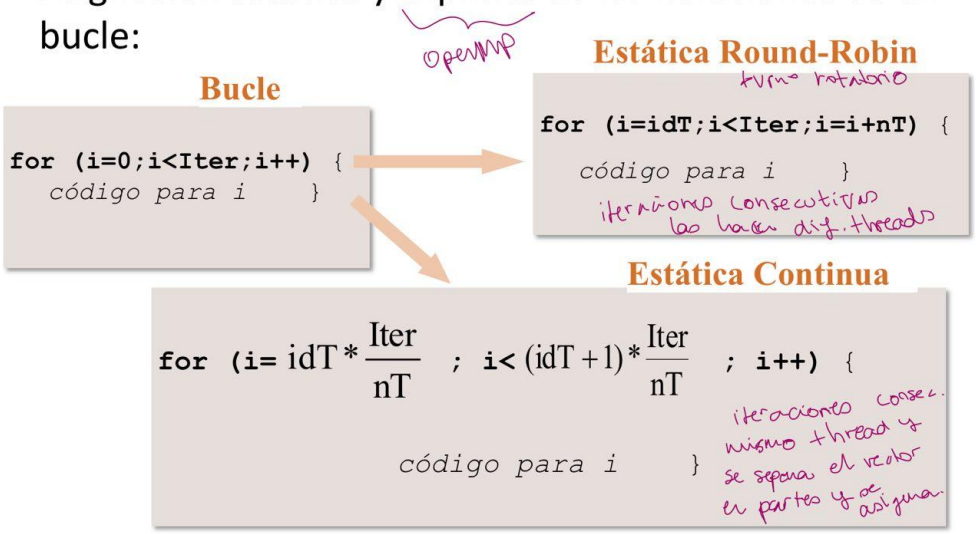
1. *Descomponer en tareas independientes o localizar el paralelismo*
    - El programador busca unidades de trabajo que se puedan ejecutar en paralelo. Estas unidades, junto con los datos que utilizan, formarán las tareas.
    - Es conveniente representar la estructura de las tareas mediante un grafo dirigido. Si partimos del código secuencial, para extraer paralelismo nos podemos situar en dos niveles de abstracción:
        - Nivel de función
        - Nivel de bucle



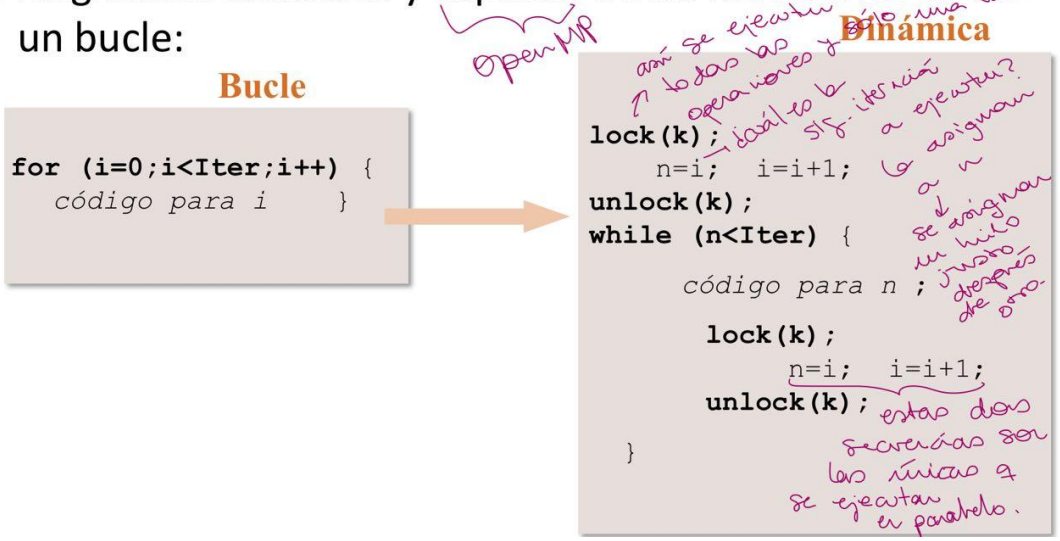
2. *Asignar tareas a procesos y/o threads*
    - Realizamos la asignación de las tareas del grafo de dependencias a procesos y hebras.
    - La granularidad de los procesos y de las hebras dependerá del número de procesadores y del tiempo de comunicación/sincronización frente al de cálculo.
    - El equilibrado (tareas = código + datos) depende de la arquitectura (homogénea frente a heterogénea y uniforme frente a no uniforme) y la aplicación/descomposición.

- Tipos de asignación:
  - Estática: se debe usar siempre que se pueda. Al no necesitar código adicional, menor tiempo. La asignación es la misma en todos los los procesadores antes de la ejecución.
  - Dinámica: cuando un núcleo termina la tarea asignada se le asigna una nueva tarea. Se asignan las tareas durante la ejecución. Necesita más tiempo porque incluye código.
- Como regla básica, se tiende a asignar las iteraciones de un bucle a hebras, y las funciones, a procesos.
- El mapeo de procesos lo realiza normalmente el SO aunque lo puede hacer el entorno o el sistema en tiempo de ejecución. El programador puede influir.

➢ Asignación **estática** y explícita de las iteraciones de un bucle:



➢ Asignación **dinámica** y explícita de las iteraciones de un bucle:



3. *Redactar código paralelo*
    - Habrá que añadir en el programa las funciones, directivas o construcciones del lenguaje que permitan el proceso de paralelización
4. *Evaluar prestaciones*
    - Si las prestaciones alcanzadas no son las que se requieren, se debe volver a etapas anteriores del proceso de implementación.

Lección 6: Evaluación de prestaciones en procesamiento paralelo

Ganancia en prestaciones y escalabilidad

Para estudiar la evaluación de prestaciones se utilizan las siguientes medidas:

1. Tiempo de respuesta: tiempo que supone la ejecución de una entrada
    a. Real (elapsed time)
    b. Usuario, sistema, CPU time = user + sys
2. Productividad
3. $Eficiencia = \frac{prestaciones\ utilizando\ p\ recursos}{p \cdot prestaciones\ utilizando\ 1\ recuso} = \frac{S(p)}{p}$

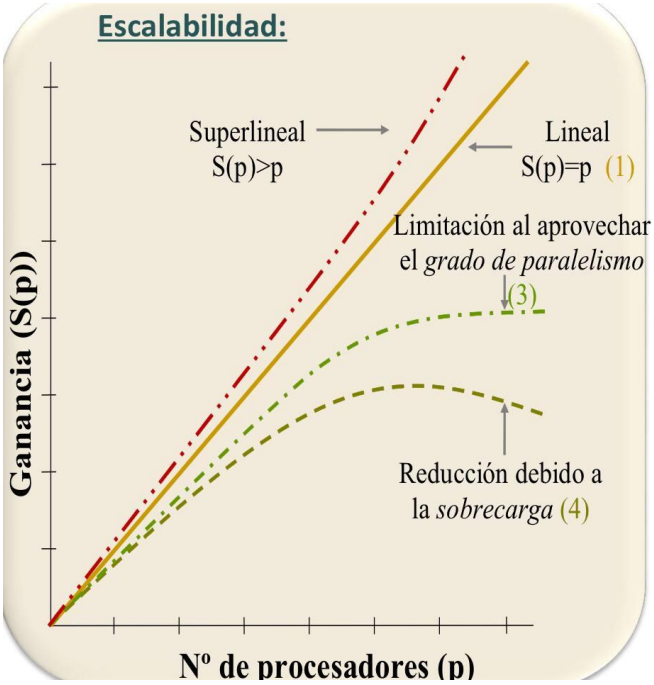
Escalabilidad

Para estudiar en qué medida se incrementan las prestaciones (velocidad) al ejecutar una aplicación en paralelo en un sistema con múltiples procesadores frente a su ejecución en un sistema uniprocesador.

La ganancia en prestaciones (ganancia en velocidad) se calcula como:

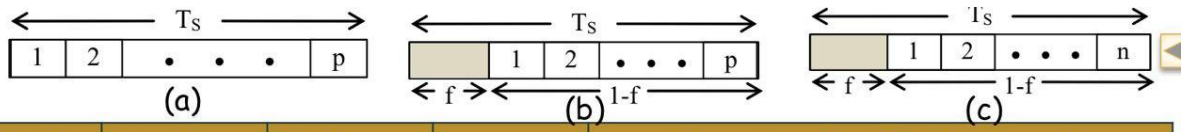
$$S(p) = \frac{Prestaciones\ (p)}{Prestaciones\ (1)} = \frac{T_s}{T_p(p)} \text{ con } T_p(p) = T_c(p) + T_o(p)$$

En la primera fracción se dividen las prestaciones con p procesadores entre las prestaciones obtenidas con la versión secuencial.
En la segunda fracción se usan los tiempos de respuesta donde T_s es el tiempo en secuencial, T_p es el tiempo en paralelo con p procesadores, T_c es el tiempo de cómputo (cálculo) y T_o es el tiempo de sobrecarga (tiempo de sincronización y comunicación entre procesos y de reparto de tareas entre procesos).



Para evaluar la escalabilidad, se representa la ganancia en función del número de procesadores. Se esperaría poder distribuir todo el programa secuencial en partes iguales entre los procesadores y un tiempo de sobrecarga despreciable (función lineal). Aunque podemos

encontrarnos una escalabilidad superlineal ($S(p) > p$), cuando al aumentar el número de procesadores se aumentan la caché y la memoria principal, que benefician a otras aplicaciones.
La ganancia en prestaciones para diferentes modelos de código secuencial y diferente sobrecarga. El modelo c) es el más usual y el a), el modelo ideal.

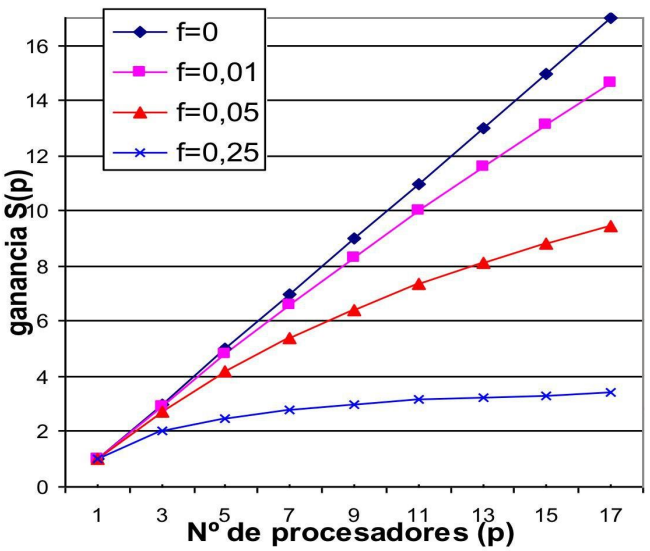


Modelo código	Fracción no paral. en T_S	Grado paralelismo	Overhead	Ganancia en función del número de procesadores p con T_S constante
(a)	0	ilimitado	0	$S(p) = \frac{T_S}{T_P(p)} = p$ Ganancia lineal (1)
(b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p\to\infty} \frac{1}{f}$ (2)
(c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$ (3)
(b)	f	ilimitado	Incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_O(p)}{T_S}} \xrightarrow{p\to\infty} 0$ (4)

Ley de Amdahl
La Ley de Amdahl nos dice que la ganancia en prestaciones que podemos obtener al aplicar una mejora en un sistema está limitada por la parte del sistema que no utiliza la mejora.

$$S(p) = \frac{T_s}{T_p(p)} \leq \frac{T_s}{f \cdot T_s + \frac{(1-f)\cdot T_s}{p}} = \frac{p}{1+f\cdot(p-1)} \to \frac{1}{f}\ (p \to \infty)$$

Esta ley nos da una visión pesimista de las ventajas de la paralelización, ya que nos dice que tenemos limitada la escalabilidad, y que este límite depende de la fracción de código no paralelizable (f) en el código secuencial. La escalabilidad decrece conforme se incrementa f.



Ganancia escalable. Ley de Gustafson

Los objetivos al paralelizar una aplicación pueden ser:

1. Disminuir el tiempo de ejecución hasta que sea razonable
2. Aumentar el tamaño del problema a resolver, lo que nos puede llevar a diversas mejoras, como por ejemplo, a incrementar la precisión en el resultado.

Amdahl mantiene constante T_s y Gustafson mantiene constante T_p. Mientras que la Ley de Amdahl asume que el tiempo de ejecución secuencial se mantiene constante y muestra que la ganancia está limitada por el código no paralelizable, Gustafson mantiene constante el tiempo de ejecución paralelo y muestra que la ganancia en función de p puede crecer con pendiente constante.

Por tanto, para Amdahl la f es la fracción de tiempo del código no paralelizable respecto al tiempo de código secuencial; mientras que para Gustafson, la f es la fracción de tiempo del código no paralelizable respecto al tiempo de código paralelo.

$$S(p) = \frac{T_s(n=kp)}{T_p} = \frac{fT_p + p(1-f)T_p}{T_p} = p(1-f) + f$$