


Paradigmas y Lenguajes

Paralelismo y Concurrency - 2



Lic. Ricardo Monzón



PROGRAMACION CONCURRENTE Y PARALELA. Descomposición de Problemas y Especificación Formal

DESCOMPOSICION DE PROBLEMAS

La descomposición de los problemas a resolver plantea diferentes retos a nivel distribuido y paralelo, las aplicaciones distribuidas y/o paralelas consisten en una o más tareas que pueden comunicarse y cooperar para resolver un problema.

Por *descomposición* entendemos la división de las estructuras de datos en subestructuras que pueden distribuirse separadamente, o bien una técnica para dividir la computación en computaciones menores, que pueden ser ejecutadas separadamente.

Las estrategias más comúnmente usadas incluyen:

- ***Descomposición funcional***
- ***Descomposición geométrica***
- ***Descomposición iterativa***

DESCOMPOSICION DE PROBLEMAS

Descomposición funcional: se rompe el cómputo en diferentes subcálculos, que pueden: a) realizarse de forma independiente; b) en fases separadas (implementándose en *pipeline*); c) con un determinado patrón jerárquico o de dependencias de principio o final entre ellos.

Descomposición geométrica: el cálculo se descompone en secciones que corresponden a divisiones físicas o lógicas del sistema que se está modelando. Para conseguir un buen balanceo, estas secciones deben ser distribuidas de acuerdo a alguna regla regular o repartidas aleatoriamente. Normalmente, es necesario tener en cuenta cierta ratio de cómputo-comunicación, para realizar un balanceo más o menos uniforme.

DESCOMPOSICION DE PROBLEMAS

Descomposición iterativa: romper un cómputo en el cual una o más operaciones son repetidamente aplicadas a uno o más datos, ejecutando estas operaciones de forma simultánea sobre los datos. En una forma determinística, los datos a procesar son fijos, y las mismas operaciones se aplican a cada uno. En la forma especulativa, diferentes operaciones son aplicadas simultáneamente a la misma entrada hasta que alguna se complete.

En cuanto a la creación de las aplicaciones distribuidas o paralelas, basándose en las posibles descomposiciones, no hay una metodología claramente establecida ni fija, debido a la fuerte dependencia de las arquitecturas de las máquinas que se usen, y los paradigmas de programación usados en su implementación.

DESCOMPOSICION DE PROBLEMAS

Una metodología básica

Una metodología simple de creación de aplicaciones paralelas y/o distribuidas podría ser la que estructura el proceso de diseño en cuatro etapas diferentes: ***partición, comunicación, aglomeración y mapping*** (a veces a esta metodología se la denomina con el acrónimo ***PCAM***).

Las dos primeras etapas se enfocan en la concurrencia y la escalabilidad, y se pretende desarrollar algoritmos que primen estas características.

En las dos últimas etapas, la atención se desplaza a la localidad y las prestaciones ofrecidas.

DESCOMPOSICION DE PROBLEMAS

Partición: el cómputo a realizar y los datos a operar son descompuestos en pequeñas tareas. El objetivo se centra en detectar oportunidades de ejecución concurrente. Para diseñar una partición, observamos los datos del problema, determinamos particiones de estos datos y, finalmente, se asocia el cómputo con los datos. A esto se denomina *descomposición del dominio*.

Una alternativa consiste en la descomposición funcional, asignando diferentes cálculos o fases funcionales a las diferentes tareas. Las dos son complementarias que pueden ser aplicadas a diversos componentes o fases del problema, o al mismo problema para obtener algoritmos distribuidos o paralelos alternativos.

Comunicación: se determinan las comunicaciones necesarias (en forma de estructuras de datos necesarias, protocolos, y algoritmos), para coordinar la ejecución de las tareas.

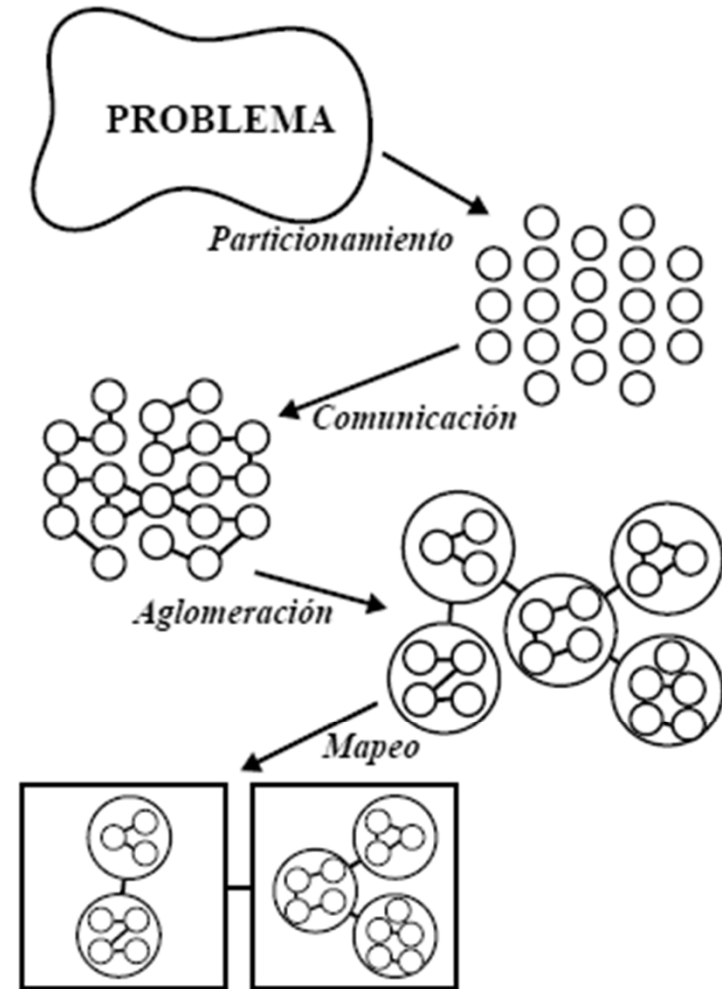
DESCOMPOSICION DE PROBLEMAS

Aglomeración: las tareas y estructuras de comunicación de las dos primeras fases son analizadas respecto de las prestaciones deseadas y los costes de implementación. Si es necesario, las tareas son combinadas en tareas mayores, si con esto se consigue reducir los costes de comunicación y aumentar las prestaciones.

Mapping o mapeo: cada tarea es asignada a un procesador/nodo, de manera que se intentan satisfacer los objetivos de maximizar la utilización del procesador/nodo, y minimizar los costes de comunicación. El *mapping* puede especificarse de forma estática, o determinarlo en ejecución mediante métodos de balanceo de carga.

DESCOMPOSICION DE PROBLEMAS

- ▶ **Particionamiento:** descomposición en tareas.
- ▶ **Comunicación:** estructura para coordinar la ejecución.
- ▶ **Aglomeración:** evaluación de tareas y estructura con respecto a rendimiento y costo, combinando tareas para mejorar.
- ▶ **Mapeo:** asignación de tareas a procesadores.



DESCOMPOSICION DE PROBLEMAS

Mapeo de procesos

- Normalmente tendremos más tareas (procesos) que procesadores físicos.
- Los algoritmos paralelos (o el scheduler de ejecución) deben proveer un mecanismo de “mapping” entre tareas y procesadores físicos, esto significa *especificar dónde se ejecuta cada tarea*.
- Este problema no existe en uniprosesadores o máquinas de memoria compartida con scheduling de tareas automático.
- ***El lenguaje de especificación de algoritmos paralelos debe poder indicar claramente las tareas que pueden ejecutarse concurrentemente y su precedencia/prioridad para el caso que no haya suficientes procesadores para atenderlas.***

DESCOMPOSICION DE PROBLEMAS

Mapecto de procesos

Decidir el mapping es complicado porque:

- *Ubicar tareas concurrentes en distintos procesadores \Rightarrow mejora la concurrencia pero...*
- *Ubicar tareas que se comunican con frecuencia en el mismo procesador \Rightarrow mejora la localidad de los datos*

Un buen mapping es crítico para la performance de los algoritmos paralelos. Debe encontrarse un equilibrio que optimice el rendimiento paralelo.

La clave es obtener un mapping que logre un balance para buscar el rendimiento de toda la arquitectura paralela.

DESCOMPOSICION DE PROBLEMAS

Efecto de la Granularidad en la Performance

En algunas ocasiones, *disminuir* el número de procesadores puede *mejorar* la performance de un sistema paralelo:

- Se denomina *scaling down* de un sistema paralelo.

Un modo simple de realizar el scaling down es pensar que sobre un procesador físico real asignamos más de un procesador virtual. Si por ejemplo tenemos **P** procesadores reales y **N** procesadores virtuales:

N/P es el factor de aumento de los cálculos a realizar en cada procesador físico

ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

¿Qué se puede ejecutar concurrentemente?

Ahora es necesario, dado un programa concurrente, saber que secciones del código son concurrentes y cuáles no, además es indispensable especificarlo en un lenguaje de programación

No todas las partes se pueden ejecutar en forma concurrente.

Considerando el siguiente fragmento de programa:

$x := x + 1;$ $y := x + 2;$

Está claro que la primera sentencia debe ejecutarse antes que la segunda, sin embargo si consideramos ahora:

$x := 1;$ $y := 2;$ $z := 3;$

Se puede observar que el orden en que se ejecuten no interviene en el resultado final. Si se dispusiera de 3 procesadores, se podría ejecutar cada una de las líneas en uno de ellos, incrementando la velocidad del sistema.

ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

Bernstein, definió unas condiciones para determinar si dos conjuntos de instrucciones S_i y S_j se pueden ejecutar concurrentemente.

Condiciones de Bernstein: Para poder determinar si dos conjuntos de instrucciones se pueden ejecutar de forma concurrente, se define en primer lugar los siguientes conjuntos:

- $L(S_k) = \{a_1, a_2, \dots, a_n\}$, como el **conjunto de lectura** del conjunto de instrucciones S_k y que esta formado por todas las variables cuyos valores son referenciados (se leen) durante la ejecución de las instrucciones en S_k .
- $E(S_k) = \{b_1, b_2, \dots, b_m\}$, como el **conjunto de escritura** del conjunto de instrucciones S_k y que esta formado por todas las variables cuyos valores son actualizados (se escriben) durante la ejecución de las instrucciones en S_k .

ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

Para que dos conjuntos de instrucciones S_i y S_j se puedan ejecutar concurrentemente, se tiene que cumplir que:

- $L(S_i) \cap E(S_j) = \emptyset$
- $E(S_i) \cap L(S_j) = \emptyset$
- $E(S_i) \cap E(S_j) = \emptyset$

(L=Lectura, E=Escritura)

Como ejemplo supongamos que tenemos :

$S_1 \rightarrow a := x + y;$

$S_2 \rightarrow b := z - 1;$

$S_3 \rightarrow c := a - b;$

$S_4 \rightarrow w := c + 1;$

Empleando las condiciones de Bernstein veremos que sentencias pueden ejecutarse de forma concurrente y cuáles no. Para ello, vamos a establecer los conjuntos de lectura y escritura correspondientes:

ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

$$L(S_1) = \{x, y\}$$

$$L(S_2) = \{z\}$$

$$L(S_3) = \{a, b\}$$

$$L(S_4) = \{c\}$$

$$E(S_1) = \{a\}$$

$$E(S_2) = \{b\}$$

$$E(S_3) = \{c\}$$

$$E(S_4) = \{w\}$$

Luego se aplica las condiciones de Bernstein a cada par de sentencias:

Entre $S_1 \cap S_2$:

1. $L(S_1) \cap E(S_2) = \emptyset$
2. $E(S_1) \cap L(S_2) = \emptyset$
3. $E(S_1) \cap E(S_2) = \emptyset$

Entre $S_1 \cap S_3$:

1. $L(S_1) \cap E(S_3) = \emptyset$
2. $E(S_1) \cap L(S_3) = a \neq \emptyset$
3. $E(S_1) \cap E(S_3) = \emptyset$

Entre $S_1 \cap S_4$:

1. $L(S_1) \cap E(S_4) = \emptyset$
2. $E(S_1) \cap L(S_4) = \emptyset$
3. $E(S_1) \cap E(S_4) = \emptyset$

ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

Entre $S_2 \cap S_4$:

1. $L(S_2) \cap E(S_4) = \emptyset$
2. $E(S_2) \cap L(S_4) = \emptyset$
3. $E(S_2) \cap E(S_4) = \emptyset$

Entre $S_2 \cap S_3$:

1. $L(S_2) \cap E(S_4) = \emptyset$
2. $E(S_2) \cap L(S_4) = b \neq \emptyset$
3. $E(S_2) \cap E(S_4) = \emptyset$

Entre $S_3 \cap S_4$:

1. $L(S_3) \cap E(S_4) = \emptyset$
2. $E(S_3) \cap L(S_4) = c \neq \emptyset$
3. $E(S_3) \cap E(S_4) = \emptyset$

De todo se deduce la siguiente tabla en la que puede verse que pares de sentencias pueden ejecutarse en forma concurrente:

	S_1	S_2	S_3	S_4
S_1	-	Si	No	Si
S_2	-	-	No	Si
S_3	-	-	-	No
S_4	-	-	-	-

Una vez identificado que se puede o no ejecutar concurrentemente se hace necesario algún tipo de notación para especificarlas.

ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

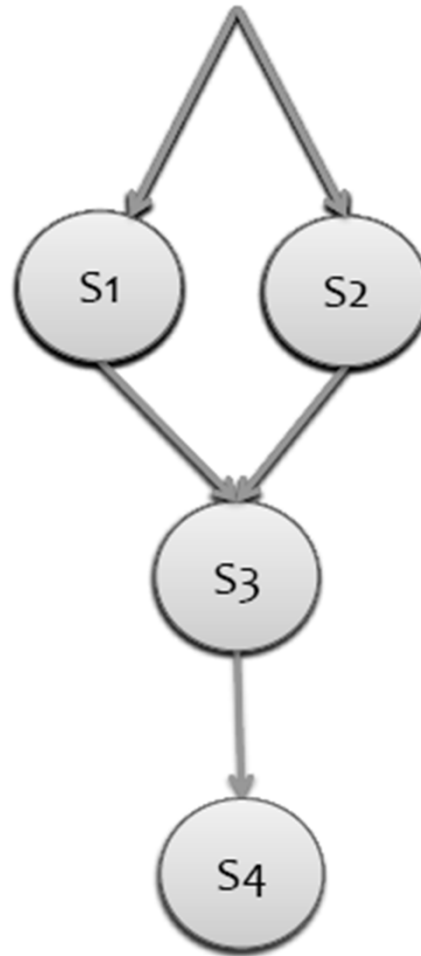
Veremos 2 formas de especificar la ejecución concurrente de instrucciones basadas en una notación gráfica **«grafo de precedencia»** y otra basada en lo que suelen utilizar diversos lenguajes de programación, el **«par cobegin/coend»**.

Grafos de precedencia:

- Se trata de una notación gráfica.
- Es un grafo dirigido acíclico.
- Cada nodo representará una parte (conjunto de instrucciones) de nuestro sistema.
- Una flecha desde A hasta B, representa que B solo puede ejecutarse cuando A haya finalizado.
- Si aparecen dos procesos en paralelo, querrá decir que se pueden ejecutar concurrentemente.

ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

Para el ejemplo anterior, el grafo de precedencia sería la siguiente figura:



ESPECIFICACION FORMAL DE SISTEMAS CONCURRENTES

Sentencias COBEGIN-COEND:

- Todas las acciones que puedan ejecutarse concurrentemente las instrucciones dentro del par cobegin/end.
- Las instrucciones en el bloque pueden ejecutarse en cualquier orden, el resto de manera secuencial.
- El ejemplo anterior quedaría:

- $S_1 \rightarrow a := x + y;$
- $S_2 \rightarrow b := z - 1;$
- $S_3 \rightarrow c := a - b;$
- $S_4 \rightarrow w := c + 1;$

```
Begin
  cobegin
    a:=x+y
    b:=z-1
  coend
  c:=a-b;
  w:=c+1;
end
```