Bigbounze
www.wuolah.com/student/Bigbounze

Tema 2.pdf

Resumenes Por Temas

2° Arquitectura de Computadores

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
UGR - Universidad de Granada

Tema 2

- Lección 4

. Problemas que plantea la programación paralela

La programación paralela presenta una serie de problemas con respecto a la secuencia:

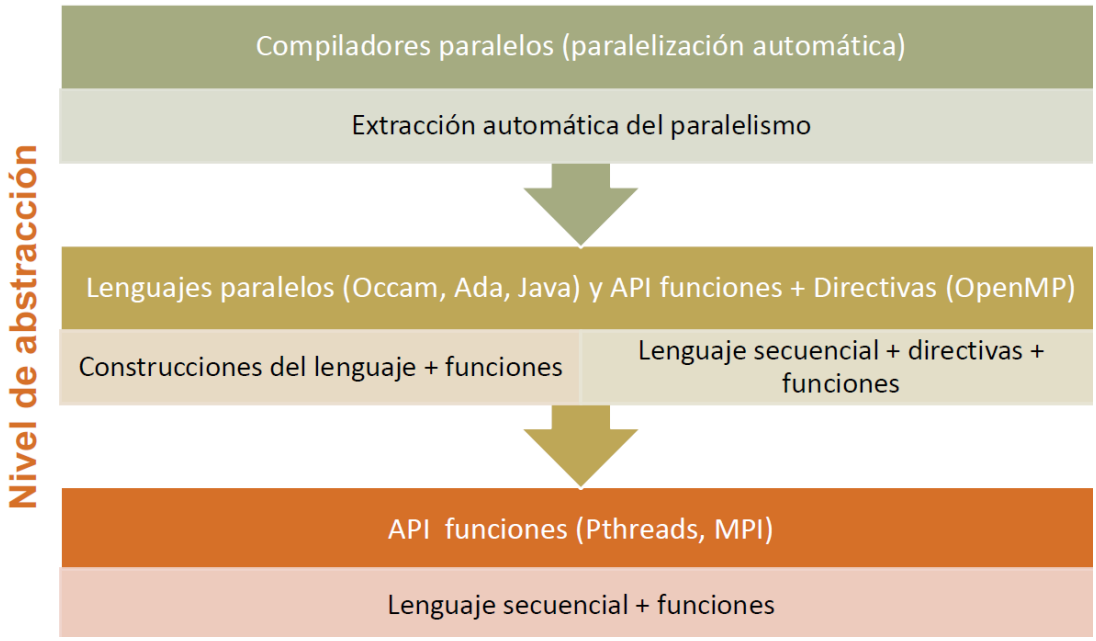
- División en unidades de cómputo independientes.
- Agrupación/asignación de tareas o carga de trabajo (código, datos) en procesos/threads.
- Asignación a procesadores.
- Sincronización y comunicación.

Estos problemas los debe abordar tanto la herramienta de programación como el programador.

. Modos de programación MIMD

- SPMD: 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. En este caso la aplicación a ejecutar se divide en unidades independientes. Cada unidad trabaja con un conjunto de datos y se asigna a un procesador distinto.

. Herramientas de programación paralela



2x1

carné universitario

FOSTER'S HOLLYWOOD

*Consulta las condiciones de la promoción en fostershollywood2x1universitario.com

Las herramientas permiten de forma implícita o explícita:

- Localizar el paralelismo o descomponer en tareas independientes (descomposition).
- Asignar las tareas, es decir, la carga de trabajo, a procesos/threads (scheduling).
- Crear y terminar procesos/threads.
- Comunicar y sincronizar procesos/threads.

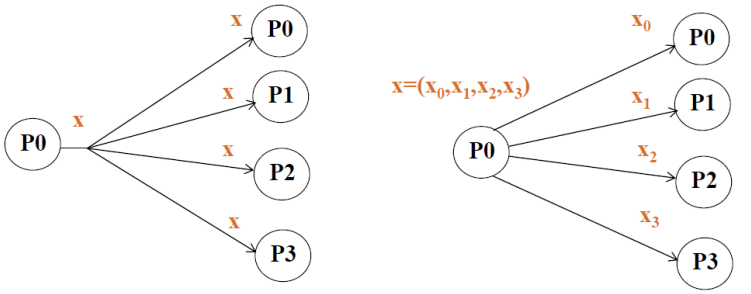
El programador, la herramienta o el SO se encarga de:

- Asignar procesos/threads a unidades de procesamiento (mapping)

. Comunicación

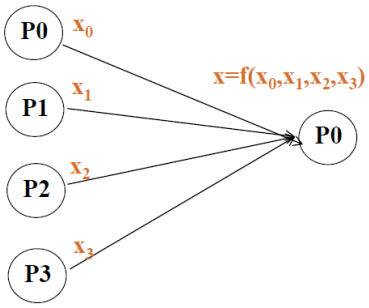
- Comunicación uno-a-todos: Un proceso envía y todos los procesos reciben. Hay dos variantes de este sistema:
  - Difusión: todos los procesos reciben el mismo mensaje.
  - Dispersión: cada proceso receptor recibe un mensaje diferente.

Difusión (*broadcast*) Dispersión (*scatter*)

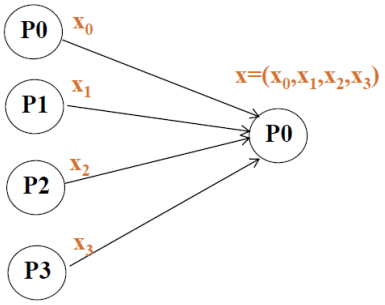


- Comunicación todos-a-uno: en este caso todos los procesos envían un mensaje a un único proceso:
  - Reducción: los mensajes enviados por los procesos se combinan en un solo mensaje mediante algún operador (+, *, max, min, AND, OR…)
  - Acumulación: los mensajes se reciben de forma concatenada en el receptor. El orden en la que se concatenan depende generalmente del identificador del proceso.

Reducción

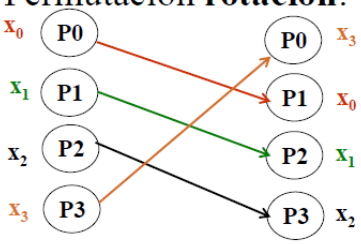


Acumulación (*gather*)

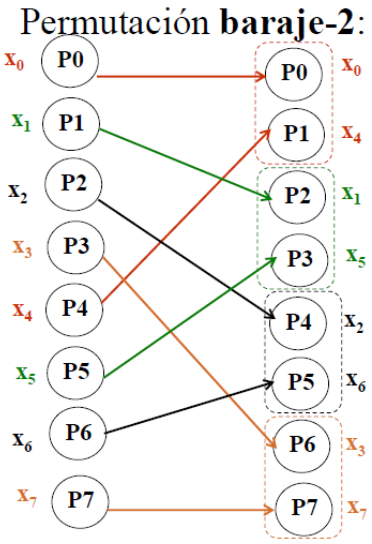


- Comunicación múltiple uno-a-uno: se caracteriza porque hay componentes del grupo que envían un único mensaje y componentes que reciben un único mensaje.

Permutación **rotación**:

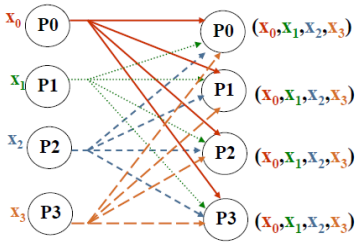


Permutación **baraje-2**:

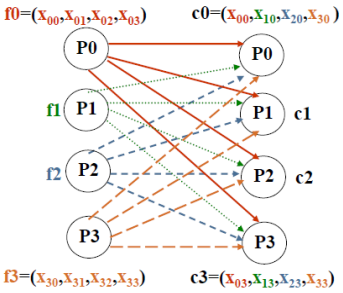


- Comunicación todos-a-todos: todos los procesos del grupo ejecutan una comunicación uno-a-todos. Todos los procesos reciben n mensajes cada uno de un proceso diferente:
  - Todos difunden: todos los procesos realizan una difusión. Usualmente las n transferencias recibidas por un proceso se concatenan en función del identificador del proceso.
  - Todos dispersan: en este caso los procesos concatenan diferentes transferencias.

Todos Difunden (*all-broadcast*) o chismorreo (*gossiping*)



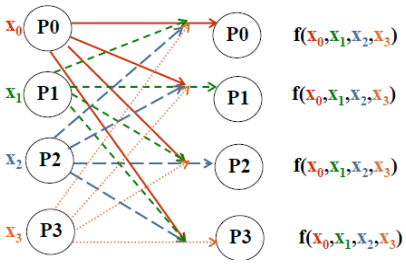
Todos Dispersan (*all-scatter*)



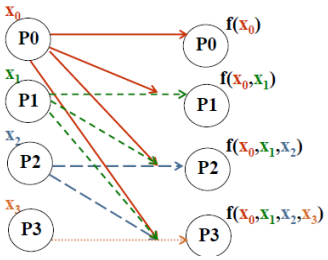
c= columna matriz
f= fila matriz

- Servicios compuestos: hay servicios que resultan de una combinación de algunos de los anteriores:
  - Todos combinan: el resultado de aplicar una reducción se obtiene en todos los procesos.
  - Recorrido (scan): todos los procesos envían un mensaje, recibiendo cada uno de ellos el resultado de reducir un conjunto de estos mensajes.

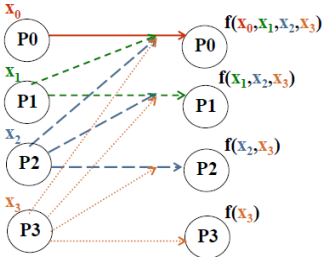
Todos combinan



Recorrido (scan) prefijo paralelo



Recorrido sufijo paralelo



. Estilos de programación, arquitecturas paralelas y herramientas de programación

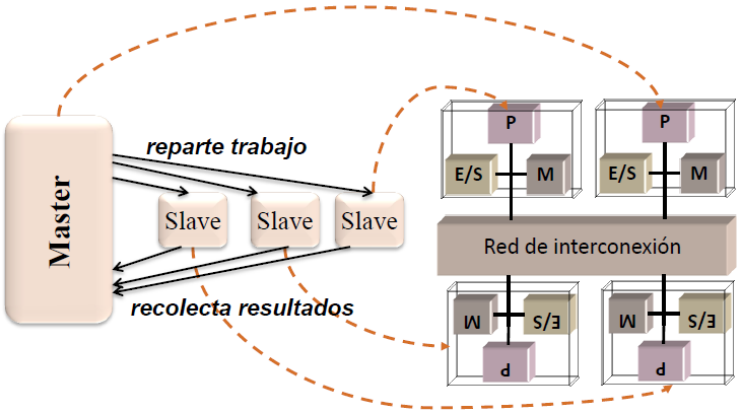
Los multicomputadores emplean paso de mensajes, los multiprocesadores usan variables compartidas, mientras que los procesadores matriciales usan el paralelismo de datos.

Entre las herramientas que podemos emplear para este tipo de programación se encuentran:

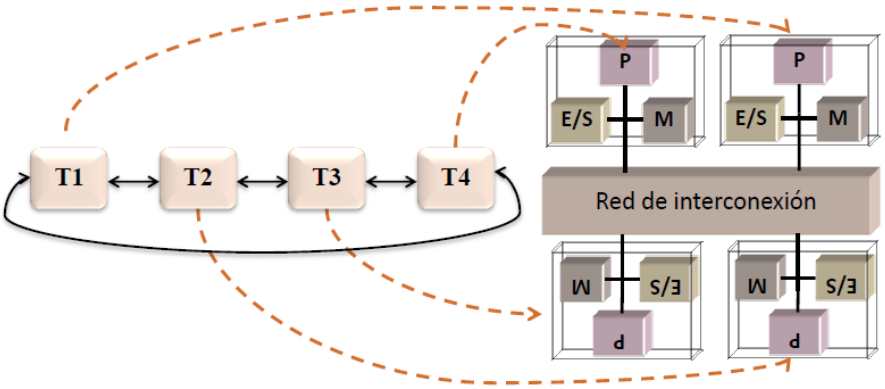
- Paso de mensajes:
  - Lenguajes de programación: Ada, Occam.
  - API (biblioteca de funciones): MPI, PVM
- Variables compartidas:
  - Lenguajes de programación: Ada, Java
  - API (directivas del compilador + funciones): OpenMP
  - API (biblioteca de funciones): POSIX Threads, shmem, Intel TBB.
- Paralelismo de datos:
  - Lenguajes de programación: HPF, Fortran 95.
  - API (funciones): Nvidia CUDA

. Estructuras de programas paralelos

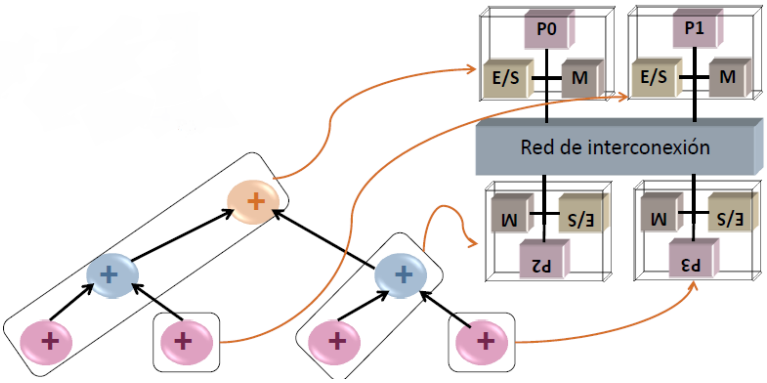
- Maestro-Esclavo o granja de tareas: en la estructura de procesos se distingue un proceso dueño y múltiples esclavos. El proceso dueño se encarga de distribuir un conjunto de tareas entre los esclavos y de ir recolectando los resultados parciales que estos que van calculando estos. Al final es el proceso maestro el que obtiene el resultado completo.



- Descomposición de datos: la estructura de datos de entrada o la de salida, o ambas, se dividen en partes. A partir de esta división se derivan las tareas paralelas. Puede haber comunicación entre los procesos.



- Divide y vencerás: esta estructura 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.



- Lección 5: proceso de paralelización

. Proceso de paralelización

1- Descomponer en tareas independientes o localizar paralelismo
   a. Es conveniente en esta fase representar la estructura mediante un grafo dirigido. Los arcos representar flujo de datos y de control y los vértices tareas. De este modo se puede comprobar gráficamente las tareas que se pueden paralelizar y las dependencias entre ellas.
   b. Nos podemos situar en dos niveles de abstracción:
      i. Nivel de función: Analizando las dependencias entre las funciones del código, encontramos las que son independientes o las que se pueden hacer independientes y por tanto se pueden ejecutar en paralelo.
      ii. Nivel de bucle: analizando las iteraciones de los bucles dentro de las funciones podemos encontrar si son o se pueden hacer independientes. Con ello podemos detectar paralelismo de datos.
2- Asignar tareas a procesos y/o threads

a. Incluimos agrupaciones de tareas en procesos/threads y mapeo a procesadores/cores.
    i. El mapeo normalmente se deja al SO
    ii. Lo puede hacer el entorno o sistema en tiempo de ejecución
    iii. El programador puede influir
b. La granularidad de la carga asignada a los procesos depende de:
    i. Numero de cores
    ii. Tiempo de comunicación frente a tiempo de calculo
c. Se debe lograr que unos procesos no deban esperar a otros. Esto depende de:
    i. La arquitectura:
        1. Homogénea frente a heterogénea
        2. Uniforme frente a no uniforme
    ii. La aplicación/descomposición
d. Los tipos de asignación pueden ser:
    i. Estática: Está determinado qué tarea va a realizar cada procesador.
    ii. Dinámica: Distintas ejecuciones pueden asignar distintas tareas a un procesador.

3- Redactar código paralelo
4- Evaluar prestaciones

- Lección 6: Evaluación de prestaciones en procesamiento paralelo

. Evaluación de prestaciones

- Tiempo de respuesta: tiempo que supone la ejecución de una entrada en el sistema.
    - Real
    - Usuario, sistema, CPU time (usuario + sistema)
- Productividad: número de entradas que el computador es capaz de procesar por unidad de tiempo
- Escalabilidad: evolución del incremento en prestaciones que se consigue en el sistema conforme se añaden recursos. Pretende medir el aprovechamiento efectivo de estos.
- Eficiencia
    - Relación prestaciones/prestaciones máximas
    - Rendimiento = prestaciones/nº de recursos
    - Prestaciones/consumo de potencia

. Ganancia en prestaciones. Escalabilidad

Ganancia en prestaciones:

$$S(p) = \frac{\text{Prestaciones}(p)}{\text{Prestaciones}(1)} = \frac{T_s}{T_p(p)}$$

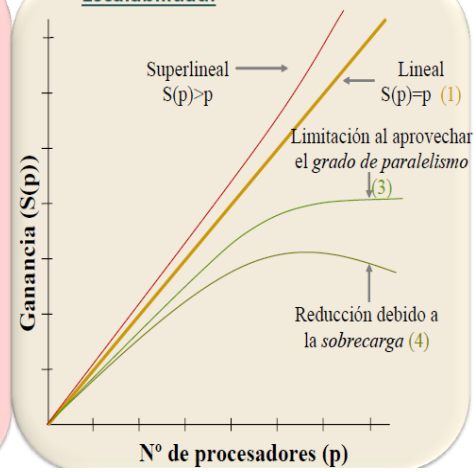
Ganancia en velocidad (Speedup)

$$T_p(p) = T_c(p) + T_o(p)$$

Sobrecarga *(Overhead)*:

- Comunicación/sincronización.
- Crear/terminar procesos/threads.
- Cálculos o funciones no presentes en versión secuencial.
- Falta de equilibrado.

Escalabilidad:



Dividimos las prestaciones logradas con p procesadores entre las prestaciones que nos da un sistema con un único procesador

- Tp(p) -> Tiempo de ejecución en paralelo con p procesadores
- Tc(p) -> Tiempo de cálculo en paralelo
- To(p) -> Tiempo de penalización (overhead)

. Número de procesadores óptimo

$$S(p) = \frac{T_s}{T_p(p)} = \frac{T_s}{T_c(p) + T_o(p)} = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_o(p)}{T_s}}$$

. Ley de Amdahl

La ganancia en prestaciones utilizando p procesadores está limitada por la fracción de código que no se puede paralelizar

$$S(p) = \frac{T_S}{T_P(p)} \leq \frac{T_S}{f \cdot T_S + \frac{(1-f)\cdot T_S}{p}} = \frac{p}{1+f(p-1)} \rightarrow \frac{1}{f}\,(p \rightarrow \infty)$$

S -> Incremento en velocidad que se consigue al aplicar una mejora

P -> Incremento en velocidad máximo que se puede conseguir si se aplica la mejora todo el tiempo

f -> Fracción de tiempo en el que no se puede aplicar la mejora

Este análisis es pesimista y nos dice que la escalabilidad está limitada por f (fracción de tiempo que no se puede paralelizar). Sin embargo, la ganancia se puede incrementar aumentando el tamaño del problema.

. Ganancia escalable o 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.

Cuando llegamos a un nivel aceptable de tiempo de ejecución en paralelo para la aplicación, el siguiente objetivo que podemos tener es aumentar el tamaño del problema para mejorar otros aspectos de las prestaciones de la aplicación.

Amdahl mantiene constante $\mathbf{T_S}$,
Gustafson mantiene constante $\mathbf{T_P}$

$$S(p) = \frac{T_S(n = kp)}{T_P} = \frac{fT_P + p(1-f)T_P}{T_P}$$

$$S(p) = p(1-f) + f$$