Introducción a Paralelismo

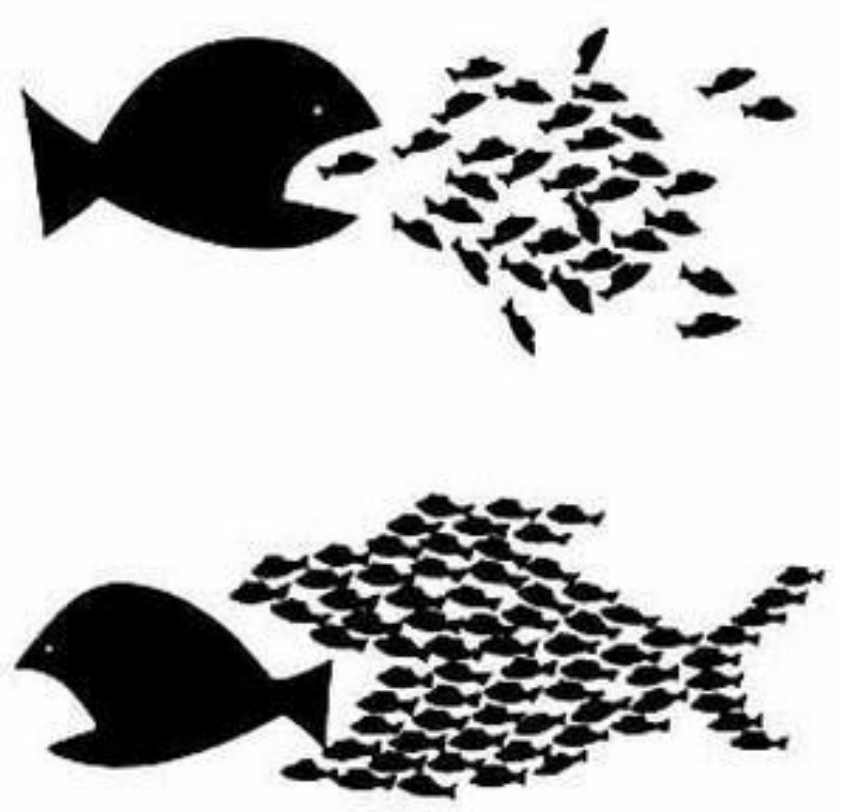
Conceptos básicos

Oscar A. Esquivel-Flores

LCD-UNAM

23 de agosto de 2019

¿Divide y vencerás?



Paralelismo I

- La velocidad de los procesadores depende del número de circuitos que posee.
- El aumento de la velocidad conduce al calentamiento excesivo de los circuitos con lo que el consumo de energía del procesador también se incrementa.
- Debido a la ineficiencia en el control de energía y el incremento de la densidad de circuitos integrados se ha considerado como alternativa el paralelismo.
- La alternativa es colocar muchos procesadores, relativamente simples, en un solo chip (arquitectura multicore) en lugar de crear un procesador único muy complejo (single-core).

Paralelismo II

- El número de los problemas que requieren más recursos computacionales se incrementa al mismo nivel que el avance de la ciencia y la tecnología.
  - Modelado del clima
  - Dinámica de proteínas
  - Diseño de fármacos
  - Exploración y explotación energética
  - Análisis de datos

Programas Paralelos I

- La mayoría de los programas son diseñados para un sistema single-core. Correr multiples instancias de estos programas no utilizan las ventajas de un sistema multi-core.
- Sin embargo, aunque puedan identificarse acciones en un programa serial que son posibles de ser paralelizadas, una eficiente implementación paralela de un programa serial no se obtiene de una secuencia de paralelizaciones.

Programas Paralelos II

- Podría esperarse que el diseño de programas paralelos identifique tareas seriales comunes y sean paralelizadas eficientemente.
- Sin embargo, con base en lo anterior, si el diseño serial es más complicado entonces se complicaría también reconocer qué paralelizar y por tanto hacerlo de manera eficiente.
- No es suficiente generar algoritmos seriales y paralelizar sus partes, es necesario vislumbrar una *forma de concebir completamente un nuevo algoritmo que explote el poder de múltiples procesadores*.

Programas Paralelos III

- Escribir un programa paralelo se basa en la idea de segmentar o **particionar** el trabajo entre muchos cores.
- Existen dos tipos de diseños básicos: **paralelismo de tareas** y **paralelismo de datos**.
- Repartir el trabajo entre cores requiere coordinar tareas, lo cual involucran la **comunicación** y **balanceo de carga** de trabajo entre los cores.
- Otro tipo de coordinación, es la **sincronización.**

Programas Paralelos IV

- Actualmente la mayoría de programas paralelos utilizan paralelismo explícito, es decir, son extensiones de lenguaje C, C++ que incluyen **instrucciones explícitas** para paralelizar.
- Existen lenguajes de alto nivel que permiten hacer paralelismo, pero tienden a sacrificar el rendimiento por facilitar el desarrollo (*are you shure?*).

Multitareas

- **Multitasking**: El Sistema Operativo ofrece facilidades para ejecutar múltiples programas aparentemente al mismo tiempo (simultáneamente), aún en un sistema mono-core.
  - Esto es posible pues cada proceso se ejecuta en un pequeño intervalo de tiempo (*ms*).
  - Distintos programas son ejecutados en una ventana de tiempo (*time slicing*).
- El S.O. puede cambiar la ejecución de los procesos varias veces, administrar recursos y pausar procesos (blocking).

Parelismo a nivel instrucción

- **Paralelismo a nivel instrucción** (ILP): Intenta mejorar el rendimiento del procesador al tener múltiples unidades funcionales (componentes del procesador) trabajando simultáneamente. Existen dos tipo:
  - **pipelining**
  - **Inicialización múltiple** (*multiple issue*): Optimización del Pipelining, conecta en secuencia pedazos de hardware y unidades funcionales. Puede ser:
    - estático (planificación al compilarse),
    - dinámico (planificación al tiempo de ejecución).
- Procesadores que soportan inicialización múltiple dinámica son conocidos como superescalares.
- Los superescalares anticipan (especulan) la instrucción previa que se va ejecutar.

Parelismo a nivel thread

- **Paralelismo a nivel thread** (TLP): Intenta proveer paralelismo al hacer ejecuciones simultáneas de diferentes threads:
  - *Coarser-grained:* Satura la ejecución simultánea
  - *Finer-grained:* Ejecución de instrucciones individuales

Hardware Paralelo I

- **Single Instruction Stream-Single Data Stream** (SISD). Ejecutan una instrucción cada vez y almacenan un dato cada vez, es utilizado en arquitecturas Von Neumann.
- **SIMD**. Single instruction,-multiple Data (SIMD). Aplican la misma instrucción a múltiples datos.
  - **Procesadores vectoriales:** Pueden operar sobre arreglos o vectores de datos. Compiladores vectorizados identifican código que puede ser vectorizado y ciclos que no pueden ser vectorizados. Son limitados en cuanto a escalabilidad.
  - **GPU:** Utilizan paralelismo SIMD por medio de *shader functions*.

Hardware Paralelo II

- **MIMD**: *Multiple Instructions-Multiple Data*. Son sistemas que soportan instrucciones simultáneas múltiples que operan sobre flujos de datos multiples. Son sistemas asíncronos.
  - **Sistemas de memoria compartida**
  - **Sistemas de memoria distribuida**.

Software Paralelo I

- Desarrollar programas explícitamente paralelos sería un objetivo a alcanzar. Entonces, el propósito es usar lenguaje C junto con tres diferentes extensiones:
  - **Message-Passsing Interface**, MPI.
  - **POSIX threads**, Pthreads.
  - **Multiproceso**, OpenMP
- Pthreads y MPI son librearías con definiciones de tipos, funciones y macros que pueden ser llamadas desde C.
- OpenMP consiste de una biblioteca y algunas modificaciones al compilador de C.

Software Paralelo II

- La necesidad de tres tipos diferentes de lenguajes atiende a que existen dos tipos diferentes de sistemas paralelos: **memoria compartida** (Pthreads, OpenMP), **memoria distribuida**.

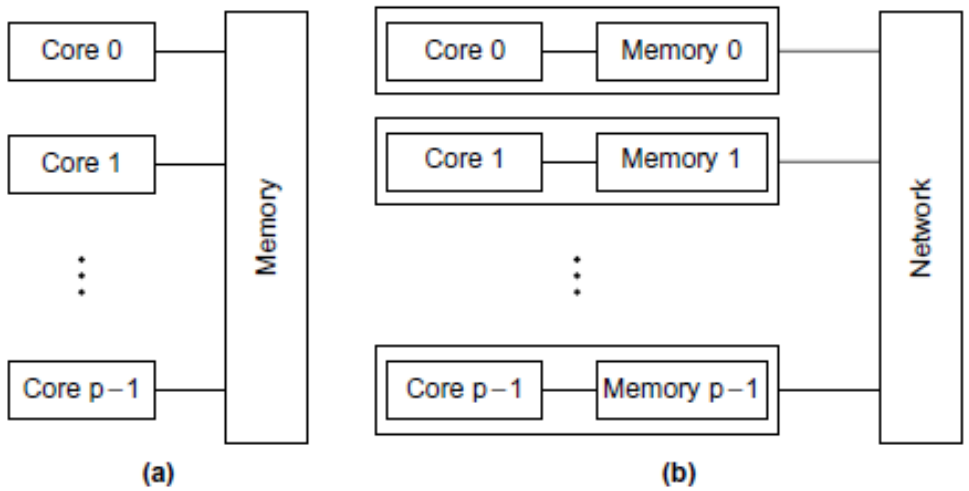


FIGURE 1.2

(a) A shared-memory system and (b) a distributed-memory system

Software Paralelo III

- La mayoría del software paralelo aplican el modelo MIMD utilizando un solo programa que se paraleliza por medio de ramas. Esta técnica es llamada SPMD (*Single Program Multiple Data*).
- Considera balanceo de carga, comunicación y sincronización entre los procesos o threads.
- En sistemas con memoria compartida se consideran procesos asíncronos, no derminismo y condiciones de carrera como la sección crítica (bloque de código que puede ser ejecutado únicamente por un thread a la vez) y candado de exclusión mutua (**mutex**).
- Para los sistemas con memoria distribuida el paso de mensajes es lo más común a considerar lo que representa tener precaución en la comunicación y sincronización.

Performance

- **Speedup**
- **Eficiencia**
- **Ley de Amdahal**
- **Escalabilidad**

Speedup

- Asumimos que es posible dividir el trabajo equitativamente entre p procesadores con ejecución de un proceso/hilo en cada uno.
- Un programa con una ejecución serial T_s y una ejecución paralela T_p
- Si $T_p = T_s/p$ se tiene un *speedup lineal*
- El **speedup real** corresponde a

$$S = \frac{T_s}{T_p}$$

- Es inusual que se tenga $S = p$, frecuentemente cuando p se incrementa S diminuye.

Eficiencia I

- La relación S/p es común que decrezca en medida que p aumenta.
- La **eficiencia** es la relación:

$$E = \frac{S}{p} = \frac{\frac{T_s}{T_p}}{p}$$

Eficiencia II

- Es importante subrayar que T_s, T_p, S, E dependen del tamaño del problema.
- Es común que el

Table 2.5 Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Ley de Amdahl

- Asúmase que existe una fracción r del programa serial que no es posible paralelizar.
- Esta ley mencina que no es posible obtener un *speedup* mejor a $1/r$.
- Sea un programa serial con el 90 % paralelizado entonces $r = 1 - 0{,}9 = 1/10$
- El mejor *speedup* que puede lograrse es de 10

Escalabilidad

- Supóngase que se ejecuta un porgrama paralelo con un número fijo de procesos/threads y tamaño de entrada también fijo.
- Es posible calcular la eficiencia E.
- Supóngase que se aumenta el numero de procesos/threads.
- Si es posible encontrar una tasa de crecimiento en el tamaño del problema tal que el programa tenga la misma E entonces el programa es escalable.