Tema 1: Introducción Programación Concurrente

Elvira Albert

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

elvira@sip.ucm.es

Madrid, 2022

Introducción

- **Programa concurrente**: tiene 2 o más procesos que trabajan juntos para realizar una tarea
- **Proceso**: programa secuencial que ejecuta en un "thread" vs programa concurrente que tiene múltiples threads de control
- **Comunicación**: los procesos cooperan comunicándose
  - **memoria compartida**: un proceso escribe en una variable que es leída por otro
  - **paso de mensajes**: proceso envía un mensaje que es leído por otro
- **Sincronización**: los procesos necesitan sincronizarse para llevar a cabo las tareas
  - **exclusión mutua**: las secciones críticas de instrucciones no se ejecutan a la vez
  - **sincronización condicional**: retrasar la ejecución de un proceso hasta que se cumpla cierta condición

Arquitecturas Hardware

- **Multi-procesadores de memoria compartida**
  - problema de consistencia de la cache
  - problema de consistencia de la memoria
  - $\rightarrow$ el programador debe conocer las características del sistema de memoria y escribir programas considerándolo
- **Multi-computadores con memoria distribuida**
  - Multi-procesador con memoria distribuida (no tienen problemas de consistencia)
  - Multi-computador: los procesadores y la red están físicamente cerca (gran velocidad de comunicación)
  - Sistema de red: los nodos están conectados por una red local (los mensajes tardan más en llegar que en multi-computador)
- **Combinaciones multi-procesadores de memoria compartida y distribuidos**
  - los nodos de una máquina distribuida pueden ser multi-procesadores de memoria compartida
  - la red de interconexión puede soportar paso de mensajes y acceso directo a la memoria remota

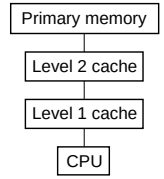


Figure 1.1 Processors, cache, and memory in a modern machine.

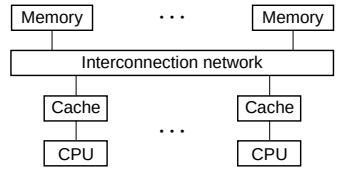


Figure 1.2 Structure of Shared-Memory Multiprocessors.

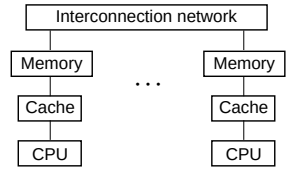


Figure 1.3 Structure of distributed-memory machines.

Aplicaciones y estilos de programación

- **Aplicaciones multi-hilo concurrentes**
  - programa que contiene múltiples procesos
  - cada proceso se planifica y ejecuta independientemente
  - #procesos>#procesadores (tienen que pedir turno)
  - interacción típicamente a través de memoria compartida
  - ejemplos: sistemas de ventana, sistemas de tiempo real, sistemas operativos
- **Aplicaciones distribuidas**
  - procesos que se comunican intercambiando mensajes
  - a menudo se organizan como cliente-servidor
  - sus componentes pueden ser aplicaciones multi-hilo
  - interacción típicamente a través de paso de mensajes
  - ejemplos: servidores de ficheros, bases de datos, servidores web
- **Computación paralela**
  - resolver computaciones grandes de manera muy eficiente
  - se ejecutan en procesadores paralelos (#procesos $\equiv$ #procesadores)
  - ejemplos: computaciones científicas, procesamiento de imágenes, optimización

Objetivo general PC

Nuestro objetivo

- diseño de programas concurrentes:
  - tomar decisiones sobre que procesos emplear,
  - cuántos usar y,
  - como deberían interactuar (estas decisiones dependen del tipo de aplicación y del hardware)
- asegurar que la interacción de procesos está correctamente sincronizada

Estudiar modelos de programación concurrente:

- programas imperativos con concurrencia explícita, comunicación y sincronización
- programación asíncrona a través de paso de mensajes

Temario

1. Introducción
2. Procesos y sincronización
3. Cerrojos y barreras
4. Semáforos
5. Monitores
6. Implementación
7. Paso de mensajes

Planificación y evaluación

- Planificación: 2 horas teoría y 2 horas de problemas o laboratorio
- Prácticas de laboratorio: semanas alternas, salvo previo aviso (20 % nota)
- 1 examen escrito
  - Práctica final (mes de mayo): 20 % nota
  - Examen final: 80 % nota

Bibliografía

- Bibliografía básica:
  - Gregory R. Andrews. Foundations of Multithreaded, Parallel and Distributed Programming, Addison Wesley.
  - D. Lea, "Programación concurrente en Java. Principios y patrones de diseño". 2ª edición, Addison Wesley, 2001.
- Bibliografía complementaria:
  - M. Ben-Ari, "Principles of Concurrent and Distributed Programming". 2ª edición, Addison - Wesley, 2006.
  - J. Magee y J. Kramer, Çoncurrency. State Models and Java Programmming". Wiley 2006.
  - M. Herlihy y N. Shavit, "The Art of Multiprocessor Programmin g". Elsevier, 2008.
  - T. Rauber y G. Rünger, "Parallel Programming: for Multicore and Cluster Systems". Springer 2010.

Paradigmas básicos

- Paralelismo iterativo
  - programa tiene varios procesos, a menudo idénticos
  - cada proceso es un programa iterativo
  - trabajan juntos para resolver un problema (pase mensajes o memoria compartida)
  - ejemplo: multiplicación de matrices

```
double a[n,n], b[n,n], c[n,n];

for [i = 0 to n-1] {
  for [j = 0 to n-1] {
    # compute inner product of a[i,*] and b[*,j]
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Sequential Matrix Multiplication

```
co [i = 0 to n-1] {  # compute rows in parallel
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication by Rows

```
co [j = 0 to n-1] {  # compute columns in parallel
  for [i = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication by Columns

```
co [i = 0 to n-1, j = 0 to n-1] { # all rows and
  c[i,j] = 0.0;                   # all columns
  for [k = 0 to n-1]
    c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```

Parallel Matrix Multiplication by Rows and Columns

```
co [i = 0 to n-1] {    # rows in parallel then
  co [j = 0 to n-1] {  # columns in parallel
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication Using Nested co Statements

```
process row[i = 0 to n-1] { # rows in parallel
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[ij] + a[i,k]*b[k,j];
  }
}
```

Parallel Matrix Multiplication Using a Process Declaration

```
process worker[w = 1 to P] {   # strips in parallel
  int first = (w-1) * n/P;     # first row of strip
  int last = first + n/P - 1;  # last row of strip
  for [i = first to last] {
    for [j = 0 to n-1] {
      c[i,j] = 0.0;
      for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
  }
}
```

Parallel Matrix Multiplication by Strips (Blocks)

Paradigmas básicos

- Paralelismo recursivo
  - paralelizar un programa recursivo si tiene múltiples llamadas recursivas independientes
  - llamadas independientes: la intersección de los conjuntos de escritura es vacía
  - ejemplo: integral de una función continua

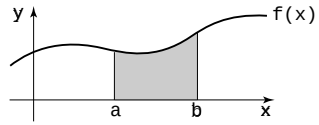


Figure 1.4 The quadrature problem.

```
double fleft = f(a), fright, area = 0.0;
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width] {
  fright = f(x);
  area = area + (fleft + fright) * width / 2;
  fleft = fright;
}
```

Iterative Quadrature Program

```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # recurse to integrate both halves
    larea = quad(left, mid, fleft, fmid, larea);
    rarea = quad(mid, right, fmid, fright, rarea);
  }
  return (larea + rarea);
}
```

Recursive Procedure for Quadrature Problem

```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # recurse to integrate both halves in parallel
    co larea = quad(left, mid, fleft, fmid, larea);
    // rarea = quad(mid, right, fmid, fright, rarea);
    oc
  }
  return (larea + rarea);
}
```

Recursive Parallel Adaptive Quadrature

Paradigmas básicos

- Productores y consumidores
  - secuencia de procesos en el que cada uno consume la salida de su predecesor y produce salida para sucesor
  - ejemplo: Unix pipelines
- Clientes y servidores
  - cliente: pide servicio y espera a ser atendido
  - servidor: repetidamente espera solicitudes, las atiende y devuelve respuesta
  - ejemplo: sistemas operativos, bd...
- Peers
  - típicamente hay un coordinador y una serie de workers que interactúan

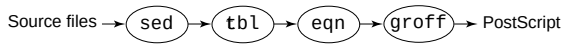


Figure 1.5 A pipeline of processes.

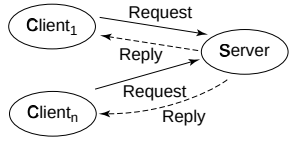
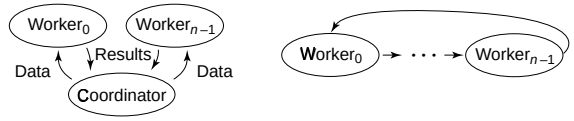


Figure 1.6 Clients and servers.



(a) Coordinator/worker interaction (b) A circular pipeline

Figure 1.7 Matrix multiplications using message passing.