Paradigmes de programació concurrent

Sistemes Operatius 2

Grau d'Enginyeria Informàtica

Paradigmes o models de programació concurrent

Paradigmes de programació concurrent[1]

1. Paral·lelisme (descomposició en les dades i en la tasca)
2. Productors i consumidors
3. Lectors i escriptors
4. Clients i servidors
5. Parells (Peer to peer, P2P)

A l'hora de dissenyar la nostra aplicació, cal identificar quin tipus de paradigma (o model) hem de fer servir.

[1] Els dos darrers paradigmes es veuran a Software Distribuït.

Paral·lelisme

L'objectiu a la programació paral·lela és aprofitar els múltiples processadors d'una (o més) màquina(es) per resoldre un problema de forma més ràpida. A la programació paral·lela concurrent

- S'executa un fil (o procés) per cada processador de l'ordinador.
- Cal reorganitzar el codi per implementar el codi i que pugui aprofitar els múltiples processadors.
- Tipus: descomposició en les dades i en la tasca

El paral·lelisme distribuït és aquell en què es fan servir múltiples màquines per implementar-lo. Una forma d'implementar el paral·lelisme distribuït es mitjançant el "Peer to peer".

Paradigma: paral·lelisme per descomposició en les dades

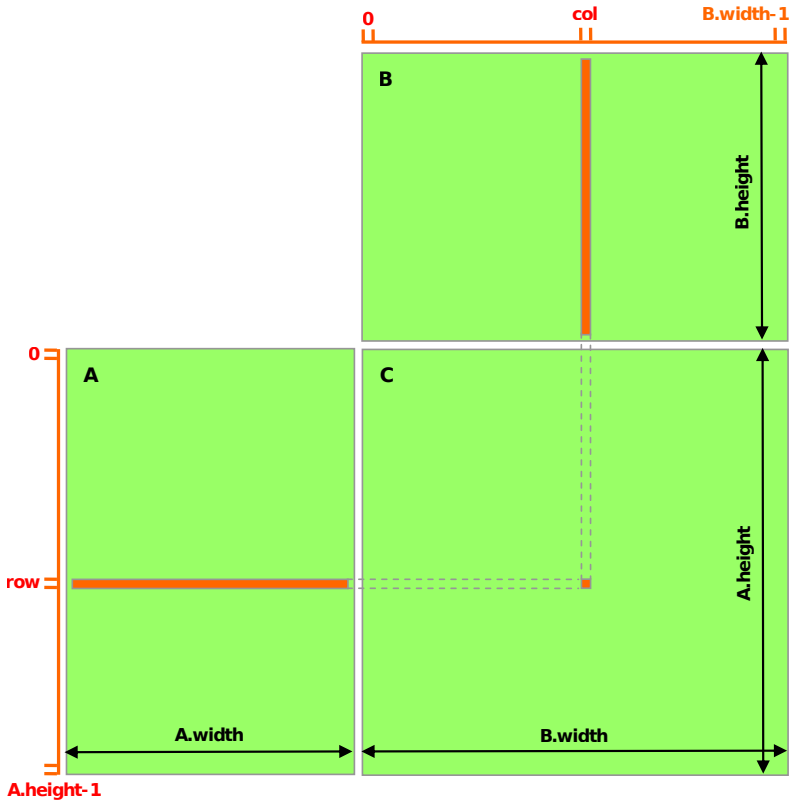
- L'objectiu és dividir les dades en conjunts més petits de forma que puguin ser processades de forma independent per cada fil. Els resultats de cada fil és "combinen" per obtenir el resultat final. És típic en codis amb bucles for i while.
- Exemple: producte de matrius

```
/* Codi */

double A[N][N], B[N][N], C[N][N];

for(col = 0; col < N; col++) {
  for(row = 0; row < N; row++) {
     C[row][col] = 0.0;
     for(k = 0; k < N; k++)
        C[row][col] +=
            A[row][k] * B[col][k];
  }
}


/* Per paral·lelitzar cal identificar
 * aquelles parts del codi que es
 * poden realitzar de forma indepen-
 * den entre si. */
```



Paradigma: paral·lelisme per descomposició en les dades

Codi de producte de matrius amb múltiples fils

- Codi `matrix_product.c`.

  Fa servir els pthreads per gestionar fils. És una API de "baix nivell".

- Codi `matrix_product_openmp.c`.

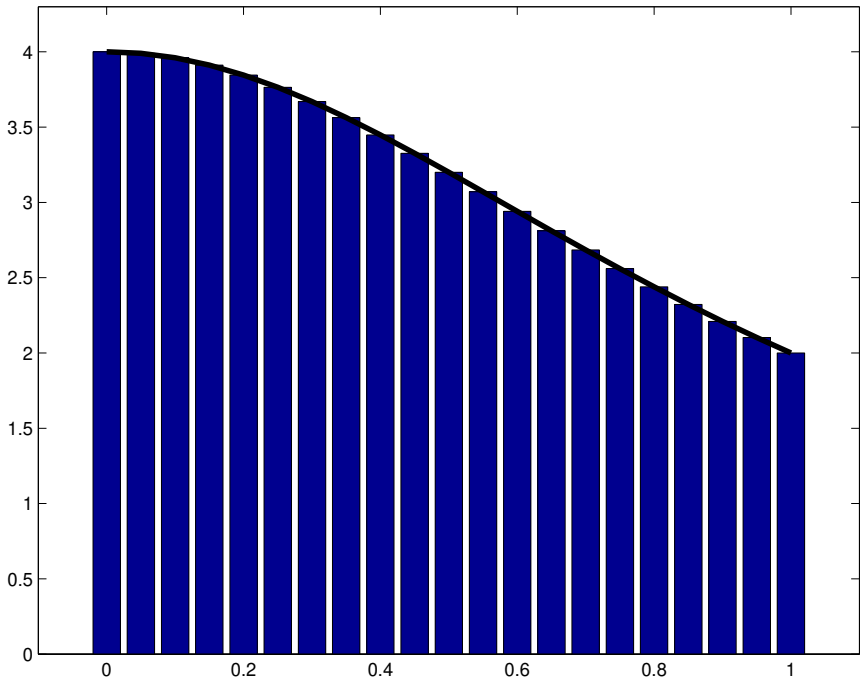
  Utilitza OpenMP, una API d'alt nivell que permet implementar de forma senzilla programes paral·lels[2].

[2] L'API d'OpenMP es veurà amb més detall a l'assignatura de Programació Paral·lela.

Paradigma: paral·lelisme per descomposició en les dades

Veiem un altre exemple de programació paral·lela: càlcul del valor de π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$




```c
#define NUM_RECTS 1E09
int main()
{
  int i;
  double mid, height, width, sum = 0.0;
  double area;

  width = 1.0 / (double) NUM_RECTS;
  for(i = 0; i < NUM_RECTS; i++) {
    mid = (i + 0.5) * width;
    height = 4.0 / (1.0 + mid * mid);
    sum += height;
  }
  area  = width * sum;
  printf("pi = %e\n", area);

  return 0;
}
```

El codi `calcul_pi_1proces.c` només té un fil d'execució.

Paradigma: paral·lelisme per descomposició en les dades

El càlcul de l'àrea de cada rectangle és independent de la resta. Aprofitem-ho! Es proposa una solució

Exemples

- Amb semàfors: codi `calcul_pi_fils_semafors.c`
- Amb monitors: codi `calcul_pi_fils_mutex.c`

Analitzeu el codi i observeu

- Quina és la secció crítica? Per què?
- L'eficiència del codi es molt baixa. Per què?

Paradigma: paral·lelisme per descomposició en les dades

Per aprofitar el paral·lelisme cal analitzar el codi i reduir l'entrada i sortida de seccions crítiques

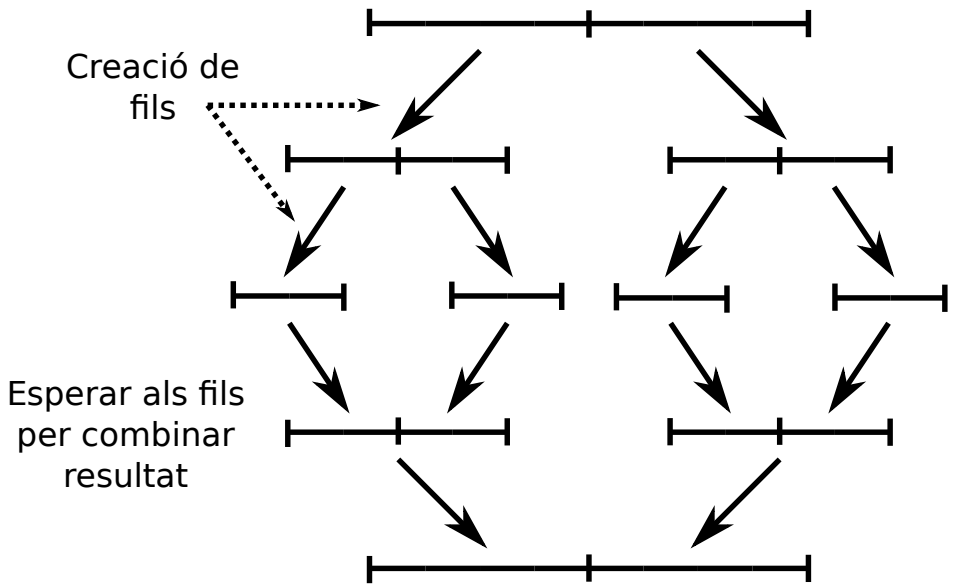
- El codi `calcul_pi_fils_mutex_millorat.c` és molt més eficient. Per què?
- El codi `calcul_pi_fils.c` no fa servir seccions crítiques. Com s'ha aconseguit?

Podem veure el codi escrit amb OpenMP

- Codi `calcul_pi_openmp.c`.

Paradigma: paral·lelisme per descomposició en la tasca

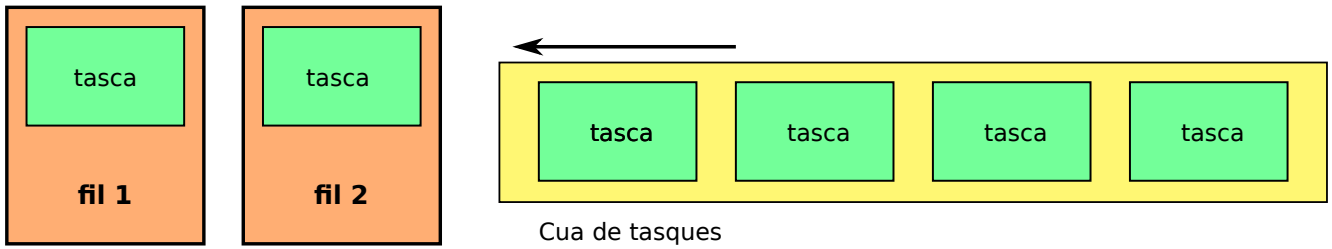
El paral·lelisme en la tasca s'utilitza quan un programa té una o més tasques (que poden ser diferents) que es poden executar de forma independent.
Es pot aplicar, per exemple, en algorismes de tipus divisió-i-vèncer.
Exemple: algorisme quicksort.



Paradigma: paral·lelisme per descomposició en la tasca

Crear fils té un cost considerable. No és bona idea crear fils a cada recursió! Una solució és utilitzar una cua de tasques:

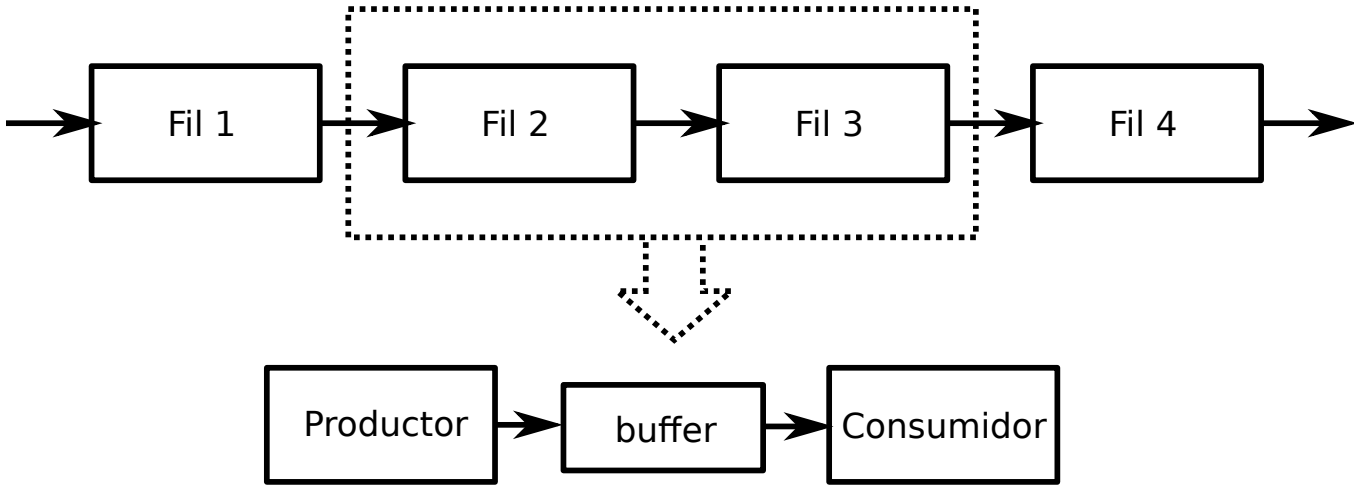
- Es crea un determinat nombre de fils a l'inici i les tasques a realitzar s'insereixen en una cua.Els fils agafen les tasques, les executen i poden crear noves tasques que s'insereixen a la cua.



- OpenMP proporciona una API senzilla per paral·lelisme en la tasca, compareu `quick_sort_serial.c` i `quick_sort_openmp.c`.

Paradigma: productors i consumidors

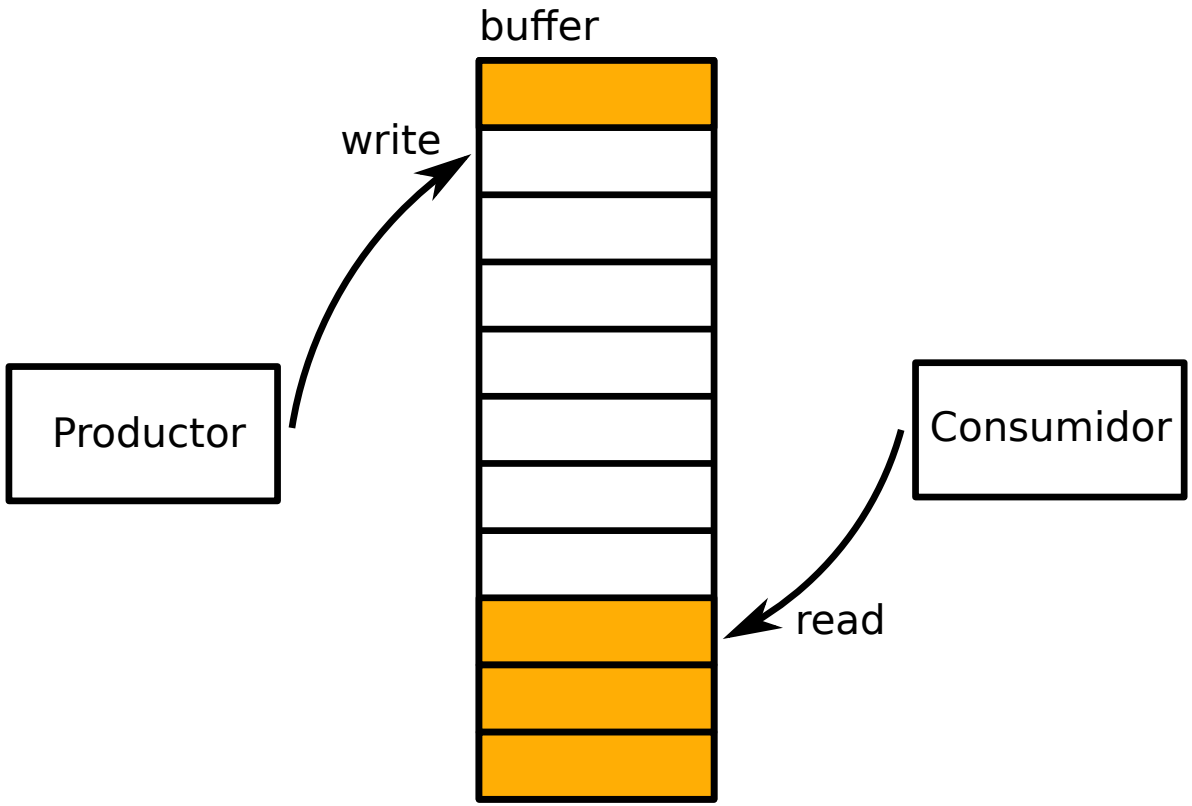
- Un productor processa i produeix una sortida de dades. Un consumidor processa l'entrada de dades.
- Moltes aplicacions són de tipus productor-consumidor, en què els productors i consumidors estan organitzats al llarg d'una cadena de processament.



- El productor pot produir les dades a ràfegues. El *buffer* ha d'assegurar que el productor hi pugui escriure tot i que el consumidor no les pugui processar tan ràpid.

Paradigma: productors i consumidors

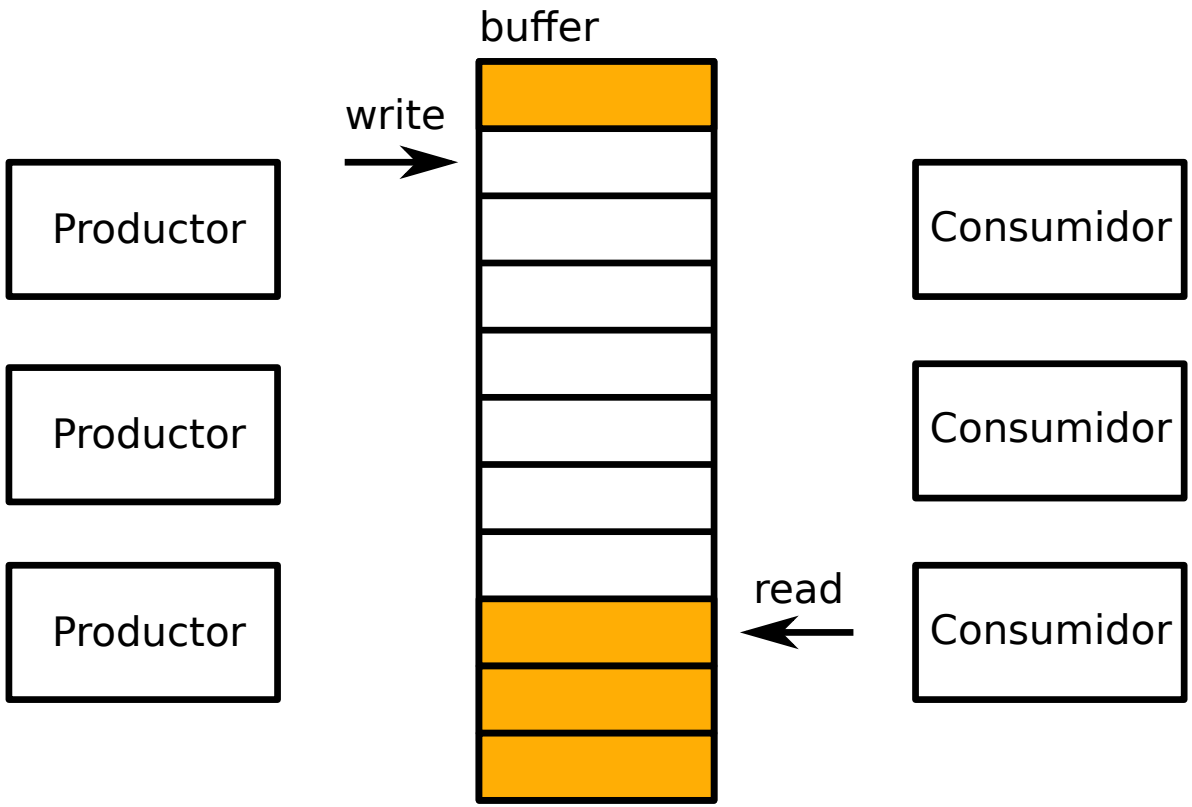
- Moltes vegades s'utilitza un *buffer* circular.
- Ens hem d'assegurar que el punter `write` mai superi a `read` (el productor ha d'esperar si el consumidor és lent), i que `read` mai superi a `write` (el consumidor haurà d'esperar si el productor és lent).



Paradigma: productors i consumidors

Hi pot haver múltiples productors i múltiples consumidors.

- Cada productor i cada consumidor és un fil.
- Els productors comparteixen l'apuntador a `write`. Els consumidors l'apuntador a `read`.



Paradigma: lectors i escriptors

En una base de dades

- Els lectors accedeixen a la base de dades i la llegeixen.
- Els escriptors hi accedeixen però per escriptura.

Per evitar interferències entre fils

- Un escriptor necessita accés exclusiu a la base de dades (o el registre de la BBDD) en escriure-hi.
- Si no hi ha cap escriptor a la base de dades (o el registre de la BBDD), múltiples lectors poden accedir a la vegada per realitzar transaccions.

Paradigma: lectors i escriptors

En programació orientada a objectes es dóna sovint aquest paradigma:

- Les funcions tipus "get" no modifiquen l'objecte (lector).
- Les funcions tipus "set" modifiquen l'objecte (escriptor).

Recordeu

- Hi pot haver múltiples lectors accedint de forma concurrent a un objecte.
- Quan hi accedeixi un escriptor hem d'assegurar que cap lector i ni escriptor hi accedeixi al mateix temps.

Conclusions

En realitzar programació concurrent

- En programar, identifiqueu quin(s) paradigma és l'adequat per implementar la vostra aplicació i utilitzeu-lo. A vegades cal combinar paradigmes...
- Existeixen APIs (o llibreries) que faciliten la implementació de determinats paradigmes. Utilitzeu-les sempre que sigui possible. OpenMP està centrat en implementar algorismes paral·lels i fa que el codi sigui portable.