

The ultimate

APD

Notebook

(for noobs...)

Exam practice

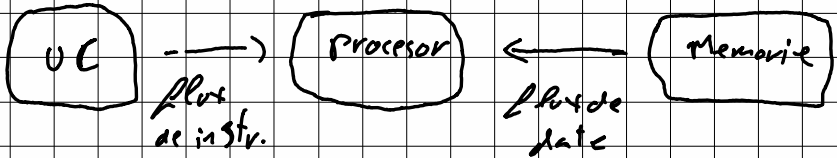
serial :

- algoritmi mai buni
- hardware mai bun

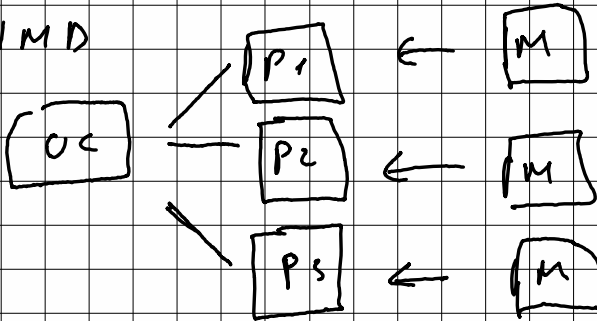
parallel :

- mai multe procesoare in acelasi timp

- SISD (modelul clasic de executie)
single instruction / single data

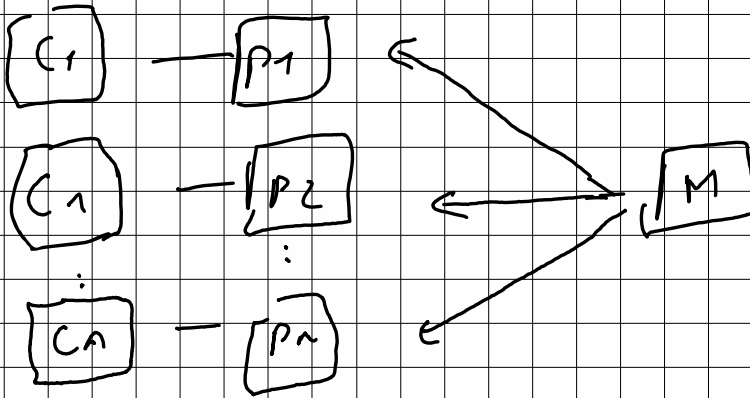


- SIMD



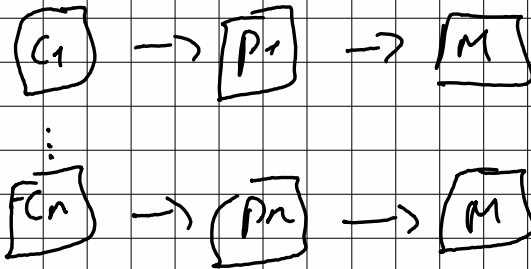
UC dă o comandă (toți execută load A),
așteaptă să se termine toate. Repeat,

• MISD



Decriptare: aplici mai multe soluții de decriptare pt același text.

• SIMD



sisteme distribuite >

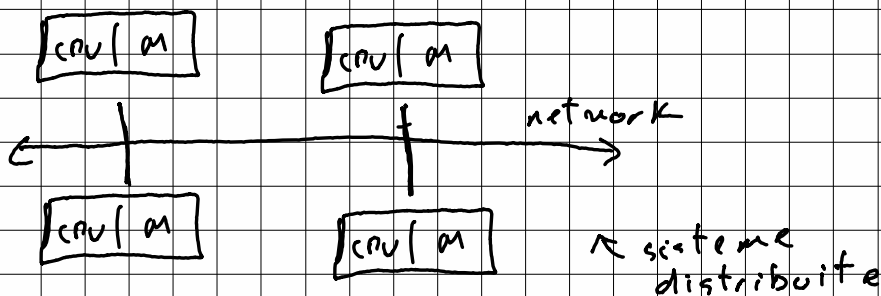
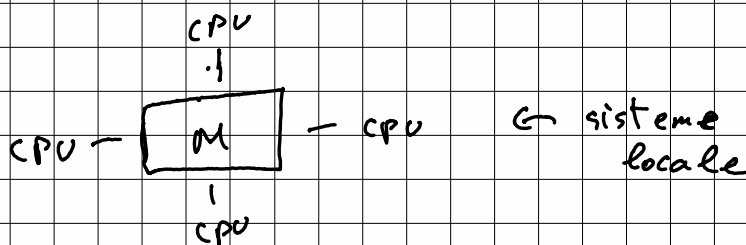
Shared memory.

- exclusiv \rightarrow se așteaptă procesarele pe rând.

- concurent \rightarrow se citește fiecare simultan.

citire CR - 1 pas

citire ER - $\log(N)$



Tipuri de paralelism

- bit level
- instruction level
- task level.

Proces == aplicație secvențială.

comunicare intra-procese:

- prin sockets
- prin scriere / citire din fișiere

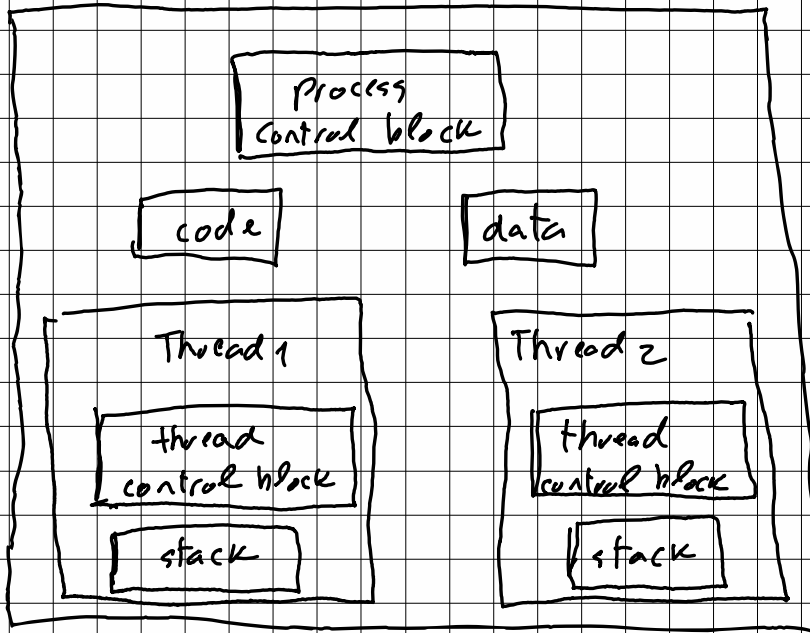
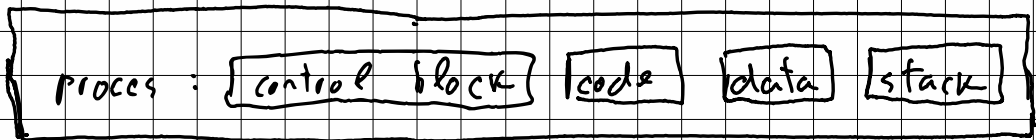
Notatii pseudocod:

• $co \ S_1 \parallel S_2 \parallel \dots \parallel S_n \ oc$

ex: $\left\{ \begin{array}{l} X=0; Y=0; \\ co \ X = X+1 \parallel Y = Y+1 \ oc \\ Z = X+Y; \end{array} \right.$

• $co \ [\text{cuantificator}] \{ S_j \}$

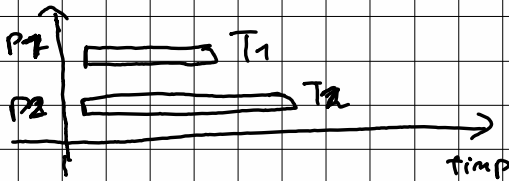
ex: $co \ [j=1 \text{ to } n] \{ a[j]=0; b[j]=0; \}$



- Aplicații care vor să facă multe task-uri simultan = mai multe thread-uri
- Multe procesări pe cantitate mare de date = mai multe procese

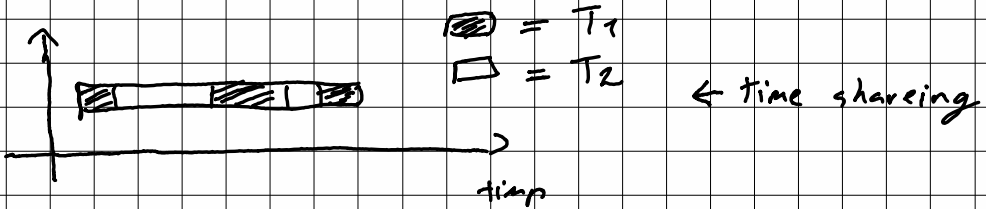
nr. procese \gg nr. cores.

???



dacă am mai multe thread-uri decât procese??

O.S. alocă un singur core pt mai multe thread-uri ale mai multor procese (prin scheduler).



Hyperthreading (Imm intel) = core-urile pot executa mai multe thread-uri.

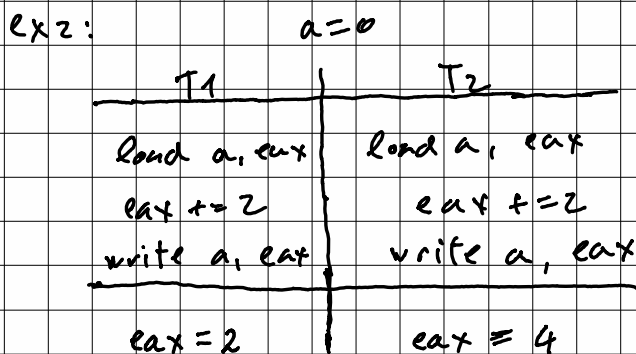
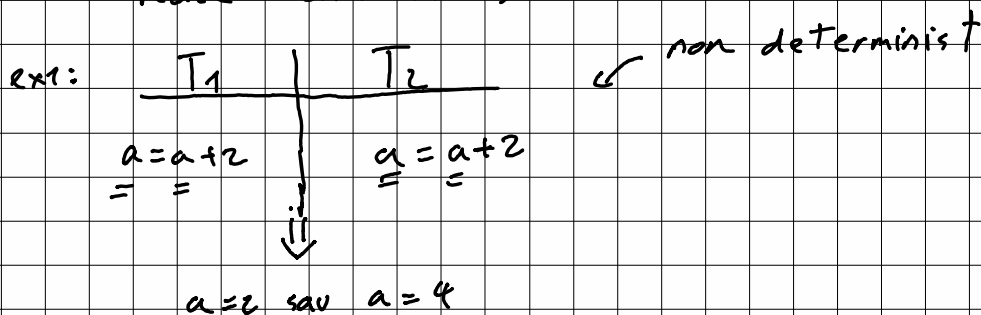
multi tasking : 1 task pe rând / core.

multi threading : 1 / mai multe thread-uri simultan

Poți avea mai multe task-uri pe mai multe core-uri.

Poți avea mai multe thread-uri pe un singur core.

Race conditions:



Cum rezolvăm race conditions??

Primitive de sincronizare:

- instrucțiuni atomice:

- actualizarea se execută complet, neîntrerupt.

- semafoare:

- mutex == semafor binar.

⇓

secțiuni critice == zona de cod se execută de un thread o dată.

- bariera:

- porțezi procesele să se aștepte între ele.

I Atomics :

ex. add 64 biți pe 32 bit registers.

```
load A0, eax
load B0, ebx
eax += ebx
write C0, eax
```

```
load A1, eax
load B1, ebx
eax += ebx
write C1, eax
```

← pot fi întrerupt și să am două jumătăți de C modificate

⇓

când un thread începe să scrie C0 nu e întrerupt până nu scrie și C1

II: Mutex

- Soluția lui Dekker

wants_to_enter[0] = true;

while (wants_to_enter[1])

```

    if (turn != 0)
        wants_to_enter[0] = false;
        while (turn != 0)
            wait;
        wants_to_enter[0] = true;

```

// CRITICAL SECTION

turn = 1

wants_to_enter[0] = false;

T₀

T₁

{ want == ∃ un thread care
vrea să intre.
turn == e rândul threadului x

turn = 0

want₀

!want₀

want₀

!want₁ → T₀ execută instrucțiunile; schimbă turn.

want₁ → ∅

want₁ → T₁ want = false; wait; T₀ execută;
T₀ schimbă turn; T₁ want = true;

turn = 1

want₀

!want₀

want₀

!want₁ → ∅

want₁ → T₁ execută instrucțiunile; schimbă turn.

want₁ → T₀ want = false; wait; T₁ execută;
schimbă turn.

● Soluția lui Dijkstra.

```

L1:
    b[i] = false;
    while (sw[i])
        sw[i] = false;
        if (k == i)
            c[i] = true;
            if (b[k])
                k = i;
            sw[i] = true; } goto L2
        else
            c[i] = false;
            for (j = 0...n)
                if (j != i && !c[j])
                    sw[i] = true;
                    } else
                    goto L1
    
```

- i vrea să facă ceva.
- sunt pe semaforul lui i

} preia controlul

} dacă
} altfel
} vrea să
} intre,
} mă întorc
} la început

lock - P

L2: // CRITICAL AREA

```

    b[i] = true;
    c[i] = true;
    
```

} i nu mai vrea să facă ceva.

unlock - V

sw[i] == semaforul lui i
b[i] == i vrea să intre dacă b[i] = false.
c[i] == a rată vândul lui i
i == thread id
j == thread id
k == ???

● Soluția lui Peterson

flag[0] = true;

PO-gate: turn = 1;

while (flag[1] && turn == 1) - thread vrea să
| wait; intre și e rândul
| lui.

// CRITICAL SECTION

flag[0] = false;

- Ca să celălalt thread
să lucreze

flag[0] == vreau acces.

turn == rândul cui este.

● Soluția cu asistare Hardware.

while (test-and-set(lock));

// critical section

lock = 0

Concluzie:

Dekker
Dijkstra
Peterson

}
{
}

soluții prin busy
waiting

Hardware

}

fără busy waiting

Probleme care nu au soluție paralelă

Huffman decoding

DFS

hash of hash of hash... of string
outer loops of simulations.

Complete problems

Pt calculul start / stop.

ceil vs floor

⇓

folosim ptc oferă garanția
că trece de pragul limită.

Performanța programelor:

T timp de execuție

P nr de nuclee

G timpul celui mai bun algoritm
secvențial

I: S speedup

$$S = \frac{T}{G} = P \text{ (ideal)}$$

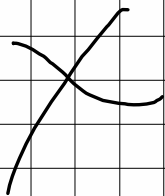
$\neq P \text{ (real)} - \text{crește / joia costă}$

II: E eficiența

$$E = \frac{G}{\underset{\text{costul}}{c}} = \frac{G}{TP} = \frac{S}{P}$$

↖ La nivel de threaduri ↗

↙ La nivel de procese ↘



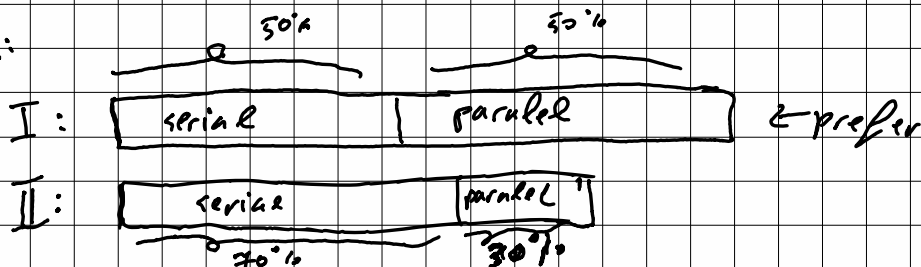
Legea lui Amdahl

În fiecare algoritm este un procent de operații care nu pot fi paralelizate (P).

$$\begin{cases} T = P \cdot G + (1-P) \cdot \frac{G}{p} \\ S = \frac{G}{T} = \frac{G}{P \cdot G + (1-P) \frac{G}{p}} = \frac{1}{P + \frac{1-P}{p}} \leq \frac{1}{P} \end{cases}$$

$$S \leq \frac{1}{P} \Leftrightarrow T \geq P \cdot G$$

Ex:



prefer I ptc are mai multe operații care pot fi paralelizate, chiar dacă instrucțiunile serial se execută mai rapid la II.

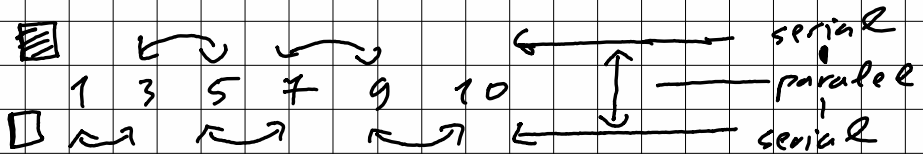
De ce ???



Speedup-ul este limitat de procentul de operații care nu pot fi paralelizate:

$$\begin{aligned} \text{I: } S &\leq \frac{1}{\frac{1}{2}} \\ \text{II: } S &\leq \frac{1}{\frac{1}{3}} \end{aligned} \Rightarrow \text{prefer I}$$

OETS

Operații care pot fi executate în paralel:



execut operațiile 
 BARIERĂ
 execut operațiile 
 BARIERĂ

Complexitate : N operații (comparații) \Rightarrow
 $\frac{N}{p}$ iterații

$$\Rightarrow O\left(\frac{N^2}{p}\right)$$

~~$$S = \frac{N^2}{\frac{N^2}{p}} = p$$~~

Dar $G \neq N^2$; $G = N \log N$
 (G = timpul celui mai bun - asort).

$$\Rightarrow S = \frac{N \log N}{\frac{N^2}{p}} = \frac{p \log N}{N}$$

Shear Sort

vectorii cu n^2 el \rightarrow matrici $n \times n$

iterativ : fiecare linie pară sortată crescător
 fiecare linie impară sortată descrescător

Paralelizare :

- sortările liniilor nu depind una de alta.
- sortările coloanelor nu depind una de alta.

Complexitate:

$\log N$ operații

2 etape

\sqrt{N} linii/coloane

$N \log N$ sortarea liniilor/coloanelor

$$O(\log N \cdot N \cdot \log \sqrt{N}) < O(N \log N)$$

decă $G = O(N \log N)$

Sortez liniile paralel
BARIERĂ
sortez coloanele paralel
BARIERĂ

Complexitate: $O\left(\frac{\log N \cdot N \cdot \log \sqrt{N}}{P}\right)$

$$S = \frac{P}{\log \sqrt{N}}$$

OETS vs SHEAR

$$\frac{2P}{\log N} \text{ cu } \frac{P \log N}{N}$$

$$N \text{ cu } \log N^2$$

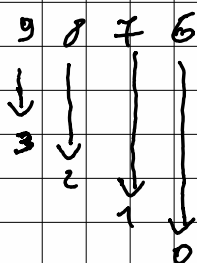
$$2^N \text{ cu } N^2$$

$$2^N < N^2 \Rightarrow$$

Shear Sort \gg OETS

Rank Sort

reții într-un vector câte elemente din vectorul inițial sunt mai mici decât elementul corespunzător indexului.



\Rightarrow pozițiile pe care trebuie
puse elementele din vectorul
original.

Paralelizare: fiecare element va număra elem.
mai mici decât el în paralel.

Counting
BARIERĂ
schimbarea pozițiilor

Complexitate: N numere
 $N-1$ comparații $\Rightarrow O(N^2)$
 N schimbări

paralel: $O\left(\frac{N^2}{P}\right)$

$$S = \frac{N \log N}{\frac{N^2}{P}} = \frac{P}{N} \log N$$

Sheer Sort \Rightarrow
OETS =
Rank Sort

Înmulțirea de matrici $N \times N$

Secvențial : $O(N^3)$

paralel : pot paraleliza oricare dintre cele n por-uri superioare (care merg pe linii / coloane). Le impart egal între thread-uri.

Complexitate : $O\left(\frac{N^3}{P}\right)$

$$S = \frac{\cancel{N^3}}{\frac{\cancel{N^3}}{P}} = P$$

Super-linear speed-up?

$S > P$? \rightarrow NU în teorie; P = limită superioară \sim

În practică : timpul de execuție paralel poate să depășească P , când e raportat la G .

De ce? cacheing!

Programare concurentă în Java :

- Concurență
- Cooperare

Sincronizare:

orice caching se
sincronizează la memoria
principală.

- Fiecare obiect are asociat un lock.
- `synchronized` == lock pe metodă/secvență de cod.
== zona de cod din metodă.
devine regiune critică.
- `wait` = manevrarea lock-ului asociate unui obiect

`wait` pe obiectul `m` din thread-ul `t`.

- se face lock pe `m`
- `t` se adaugă în lista de threaduri
blocate a lui `m`.

★ `t` trebuie să dețină lock-ul pt `m`.

- `t` continuă execuția când va fi
scos din wait-set-ul lui `m`:

`notify`:

- thread u din wait-list-ul `m` e scos
și repus în execuție.

`notifyAll`:

- toate threads scoase. Numai `t` are lock-ul.

★ `t` trebuie să dețină lock pt `m`.

How it works!



din thread-ul T:

t este blocat

←

apeler sincronizată pe m

apeler wait pe m

se pus într-o
multime de thread-uri
blocate asociate
din thread-ul U:

U este deblocat
(scos din listă)

←

apeler sincronizată pe m

apeler wait pe m

(⇒) un thread se pune singur pe wait, până
când primește notify (semnal că poate să
continue) de la alt thread.

volatile int i;

↓
se actualizează pentru
toate thread-urile. Nu
ajunge în cache.

Crearea / distrugerea threads == costisitor.



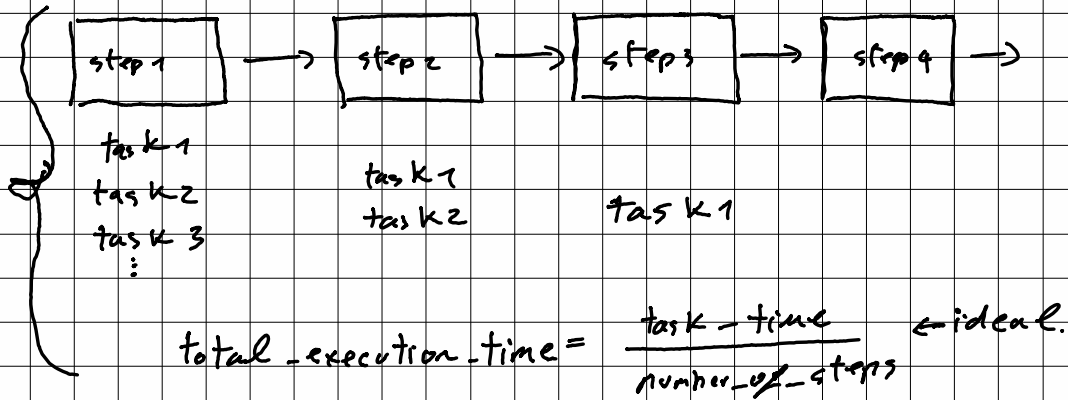
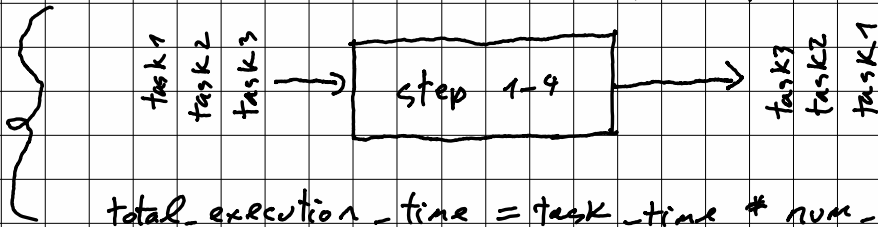
folosesc Thread Pooling. == replicated

Atomic (⇒) same as C.

Abordare a soluțiilor de paralelizare

I. Modelul Pipeline

paralelism pt seturi distincte de date.
ex. calcul polinomial.

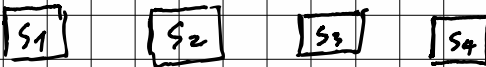


Ex.: Sorting

trebuie să fie atâta pași câte el. în vector.

time: $2N-1$

→ N pași introduce
→ N-1 propagare până la ultima poziție



← n unități de comparație (pași din pipeline)

preia valoare din stânga.
compară cu valoarea la care
reține val mai mică,
aruncă în dreapta val mai mare

Valorile sunt ordonate, dar în memoria locală a pașilor.

1. s_4 pasează la s_3
când s_3 primește, pasează val locală la s_2
.....
 s_1 le scoate.

2. În mom. în care s_1 nu mai are ce să primească, scoate spre stânga.
- || - s_2 - 1 -
.....

1. durează $2N - 2$ pași
un procesor întâi primește val locală
apoi o aruncă în stânga $\Rightarrow 2(N - 1)$

2. durează $N - 1$

\Rightarrow Total =

$$1. \quad 2N - 1 + 2N - 2 = 4N - 3$$

$$2. \quad 2N - 1 + N - 1 = 3N - 2$$

la primire la scoatere

N Queens Problem

Iterativ: Back Tracking

0	1	2	3	←	liniile din matrice
1	3	0	2	←	coloanele pe care pun damele

} încercăm toate posibilitățile cu BT.

Paralel:

1. Replicated Workers

task 1-n : poziționarea primei dame

workeri iau task-uri 1-n

încearcă poziționarea damei pe următoarea col.
dacă este valid, pune noul task în roadă.

Binary Search

Iterativ : LAM E - \bar{O}

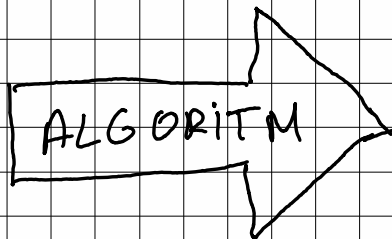
Paralel :

1. spargi vectorul în subintervale
compari cu capetele fiecărui subinterval
când găsești subintervalul bun,
te împart pe el în N/P threads, BARIERĂ.
Nu las celelalte threads să caute inutile,
ci le dau noile subintervale.

$$O(\log_p N)$$

Obs: dacă împart în $P+1$ intervale o să
am doar o singură comparație, în
loc de 2 (capetele)

$$O(\log_{P+1} N)$$



$l = 1$ // capăt stânga interval curent
 $r = n$ // capăt dreapta interval curent

$x, k = 0$ // $x =$ căutat ; $k =$ poziția lui

$g = \lceil \log(n+1) / \log(p+1) \rceil$ // nr pași

enum sens {stânga, dreapta} // sens de căutare: st \rightarrow dr

sens $c[0:p+1]$ // $c[i] = +$ e la st/dr procesului i

$c[0] =$ dreapta ; $c[p+1] =$ stânga

$j[1:p]$ // poziția din sir inspectată de procesul i

To Be Continued ...