

Írta: Vámossy Zoltán, Miklós Árpád, Szénási Sándor
Lektorálta: oktatói munkaközösségek

TÖBBSZÁLÚ/TÖBBMAGOS PROCESSZORARCHITEKTÚRÁK PROGRAMOZÁSA

PÁRHUZAMOS SZÁMÍTÁSTECHNIKA MODUL

PROAKTÍV INFORMATIKAI MODULFEJLESZTÉS

COPYRIGHT:

© 2011-2016, Vámossy Zoltán, Miklós Árpád, Szénási Sándor, Óbudai Egyetem,
Neumann János Informatikai Kar

LEKTORÁLTA: oktatói munkaközösségek

Creative Commons NonCommercial-NoDerivs 3.0 (CC BY-NC-ND 3.0)

A szerző nevének feltüntetése mellett nem kereskedelmi céllal szabadon másolható,
terjeszthető, megjelentethető és eladható, de nem módosítható.

TÁMOGATÁS:

Készült a TÁMOP-4.1.2-08/2/A/KMR-2009-0053 számú, "Proaktív informatikai
modulfejlesztés (PRIM1): IT Szolgáltatásmenedzsment modul és Többszálas
processzorok és programozásuk modul" című pályázat keretében



KÉSZÜLT: a [Typotex Kiadó](#) gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

ISBN 978-963-279-563-8

KULCSSZAVAK:

párhuzamos számítások, párhuzamos platformok és modellek, párhuzamos algoritmusok tervezése, felosztási technikák (dekompozíció), leképzési módszerek, diszkrét optimalizálás párhuzamosítása, dinamikus programozás párhuzamosítással, képfeldolgozás alapjainak párhuzamosítási lehetősége, folyamatok, szálak, szinkronizáció, párhuzamos programozás .NET környezetben

ÖSSZEFOGLALÓ:

A tárgy keretében a hallgatók elmélyítik – az alapképzésben szerzett – a párhuzamos rendszerekkel kapcsolatos tervezési és programozási ismereteiket. Az előadások keretében ismertetésre kerülnek a párhuzamos számítási modellek, részletesen taglalásra kerül a párhuzamos algoritmusok tervezési lehetőségei és módszerei (a dekompozíciótól a leképzésekig), majd olyan nagyszámítási igényű területek párhuzamosítása kerül bemutatásra, mint a diszkrét optimalizálás, dinamikus programozás és a képfeldolgozás alaptechnikái. A gyakorlatok során a hallgatók megismerik és elsajátítják a párhuzamos programozás technikáit, a folyamat- és szálkezelést, a szálak közti kommunikáció módoszatait és a szinkronizáció módszereit.

Tartalomjegyzék

- Bevezetés
- Párhuzamos programozási modellek
- Párhuzamos algoritmusok tervezési lehetőségei és módszerei
- Párhuzamosság a modern operációs rendszerekben
- Párhuzamos algoritmusok tervezésének alapjai
- Kereső algoritmusok a diszkrét optimalizálás problémájához
- Dinamikus programozás párhuzamosítási lehetőségekkel
- Képfeldolgozás és párhuzamosíthatóság
- Párhuzamos programozás .NET környezetben
- Párhuzamos programozási feladatok

Bevezetés

Motiváció

Soros és párhuzamos végrehajtás, soros és párhuzamos programozás
Miért?

Alapfogalmak

Párhuzamosság, egyidejűség, kvázipárhuzamosság, kvázi-egyidejűség

Implicit és explicit párhuzamosság

Program, feladat, folyamat, szál, végrehajtási egység, feldolgozóegység

Korlátok

Függőségek és típusai

Szinkronizáció

Hallgatói tájékoztató

A jelen bemutatóban található adatok, tudnivalók és információk a számonkérő anyag vázlatát képezik. Ismeretük *szükséges, de nem elégséges* feltétele a sikeres zárthelyinek, illetve vizsgának.

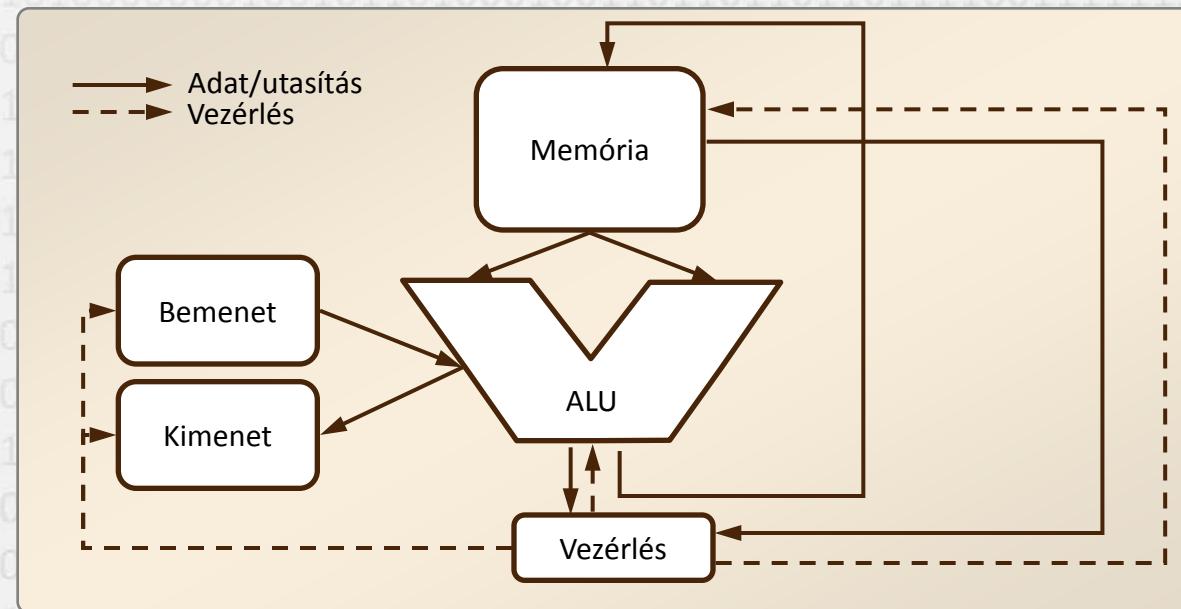
Sikeres zárthelyihez, illetve vizsgához a jelen bemutató tartalmán felül a kötelező irodalomként megjelölt anyag, a gyakorlatokon szóban, illetve a táblán átadott tudnivalók ismerete, valamint a gyakorlatokon megoldott példák és az otthoni feldolgozás céljából kiadott feladatok önálló megoldásának képessége is szükséges.

Soros végrehajtás és programozás

- A Neumann-architektúrára épülő számítógépek a programokat **sorosan hajtják végre.**

Ezen az architektúrán a gépi kódú programok tényleges futtatása az órajel ütemére sorban egymás után, utasításról utasításra történik. A programokon belül bekövetkező egyes események szigorú időbeli sorrendisége következtében az állapotátmenetek elvben előre kiszámíthatóak.

Minden mai, általánosan elérhető számítógép-architektúra a Neumann-architektúrára épül.



Párhuzamos végrehajtás és programozás

- A párhuzamos végrehajtást megvalósító architektúrákkal a jelen tárgy csak érintőlegesen foglalkozik.

Ezt a területet teljes részletességében a *Párhuzamos rendszerek architektúrája* c. tárgy fedi le.

- **Párhuzamos programozás fogalma**

A párhuzamos programozás azt a lehetőséget takarja, hogy szoftvereinket olyan programozási nyelven fejlesszük ki, amely explicit (kifejezett) lehetőséget ad a számítási feladatok egymással egy időben végrehajtandó részfeladatainak és a számítás módjának megadására.

Mint később látni fogjuk, ez a módszer új lehetőségek és problémák egész tárházát nyitja fel, és nagyságrenddel nehezebbé teszi jól működő szoftverek fejlesztését.

...de miért is kellene változtatnunk?

- **A soros (egyprocesszoros) végrehajtás megközelítette határait.**

A mai technológiákkal az egy processzoron elérhető végrehajtási teljesítmény növelésének a fizika határokat szab, melyeket már az implicit párhuzamos végrehajtás lehetőségeinek teljes kimerítése ellenére is megközelítettünk.

- A komplex vezérlőegységek órajele reálisan legfeljebb kb. $5 \cdot 10^9$ Hz (5 GHz).

- A processzorok belsejében lévő adatutak maximális hossza elég nagy ahhoz, hogy az órajel további lényegi növelése esetén még fénysebességgel terjedő impulzusok se „érjenek át” időben a processzor egyik oldaláról a másikra.
- A csíkszélesség további csökkentésével drasztikusan nő a hődisszipáció.

A mai (2010) processzorok például néhány mm² felületen 125-175 W teljesítményt disszipálnak, így hűtésük egyre nagyobb problémát okoz. Sőt a processzorok egyes területei a változó terheléstől függően nem egyformán melegszenek, ami nagy hőingadozások esetén mechanikai sérüléseket is okozhat.

Sajnos a fizikai határok mellett számításelméleti korlátokkal is számolni kell.

- A hagyományos programokban rejlő belső párhuzamosítási lehetőségeket a mai processzorok már kihasználták.

- Egy általános programban elméletileg legfeljebb kb. 4-6, a gyakorlatban kb. 2-3 utasítás hajtható végre egyidejűleg. Ezt a lehetőséget a három-, illetve négyutas szuperskalár processzorok (pl. IBM Power 3, Intel Pentium III, AMD Athlon) és közvetlen utódaik már a 2000-es évek elejére szinte teljes mértékben kiaknázták.

...de miért kellene változtatnunk?

- **A teljesítmény növelésének igénye nagyobb, mint valaha.**
 - Végrehajtási idő minimalizálása = költségcsökkentés.
 - A teraflop (10^{12} lebegőpontos utasítás/másodperc) sebességkategóriába soros végrehajtással már nem lehet belépni.
 - A memóriák sebességének növekedése sokkal lassabb, mint a processzoroké, párhuzamos architektúrák segítségével viszont jobban kiaknázható a teljesítményük (csökkenthetők a szűk keresztmetszetek).
- **A hálózatok döntő jelentőségűvé váltak.**
 - Az elosztott számítási kapacitás kihasználásával új típusú alkalmazások válnak lehetővé:
 - SETI@home, Folding@home
 - A hálózatokon és az interneten tárolt adatok mennyisége olyan nagy, hogy nagyobb feladatoknál az adatok mozgatása (pl. egy, a számításokat elvégző központi számítógép felé) lehetetlen.
 - A számítási műveleteket a hálózaton keresztül kell megvalósítani párhuzamos programozási technikákkal.

...de miért kellene változtatnunk?

- **A soros programozás korszaka tehát fokozatosan letűnik.**

A Neumann-architektúra logikai alapjaiba ágyazott soros programozási megoldások sorra elveszítik előnyeiket (egyszerű fejlesztés, hatékony hibakeresés, kiszámíthatóság), hátrányaik (teljesítmény skálázódásának hiánya, összetett és elosztott programok fejlesztésének nehézsége) pedig egyre kínzóbbak.

A párhuzamos programozás alpfogalmai

- **Soros (vagy szekvenciális) végrehajtás**

Fizikailag egy időben kizárolag egy művelet végrehajtásának lehetősége.

- **Párhuzamos (valódi párhuzamos) végrehajtás**

Fizikailag egy időben egynél több művelet végrehajtásának lehetősége.

- **Kvázipárhuzamos végrehajtás**

Látszólagos párhuzamosság időosztásos technika felhasználásával.

- **Egyidejűség (valódi egyidejűség)**

Valódi párhuzamos végrehajtás esetén kettő vagy több művelet fizikailag időben egymást átfedő végrehajtása.

- **Kvázi-egyidejűség**

Kvázipárhuzamos végrehajtás esetén kettő vagy több művelet logikailag időben egymást átfedő végrehajtása.

A párhuzamos programozás alpfogalmai

- **Program**

Utasítások és adatok önálló egységet képező halmaza.

- **Feladat**

A program elemeinek önhordó és önállóan végrehajtható halmaza vagy részhalmaza. A programok feladataikra bonthatók (de fordítva nem).

- **Folyamat**

A program végrehajtása, illetve a végrehajtás alatt álló program (adataival, erőforrásaival és belső állapotával együtt).

- **Szál**

A folyamaton belüli kisebb egység, amely végrehajtható műveletek sorát tartalmazza. Egy folyamaton belül több szál is létezhet (de fordítva nem).

- **Végrehajtási egység**

Egy adott feladat (folyamat vagy szál) legkisebb végrehajtható részhalmaza.

- **Feldolgozóegység, végrehajtóegység**

A végrehajtási egységet feldolgozó hardverelem (pl. processzormag, ALU).

A párhuzamos programozás korlátai

• Függőségek

A függőség olyan kapcsolatot vagy egymásrautaltsági viszonyt jelöl két művelet (feladat) között, amely megakadályozza ezek egyidejű (párhuzamos) végrehajtását. A függőségek alaptípusai:

- **adatfüggőség** (flow dependency): egy művelet eredményét egy másik művelet bemeneti adatként kívánja felhasználni (ez az ún. *valódi függőség* egy példája).
Léteznek ún. „áladatfüggőségek” is, amelyek nem szükségszerű, ok-okozati jellegűek, hanem az alkalmazott architektúra jellemzői miatt lépnek fel. Ezek megfelelő áttervezéssel, új technikai megoldásokkal kiküszöbölhetők, tehát jelentőségük csekély és időleges.
- **elágazási függőség** (branch dependency): egy művelet eredménye dönti el, hogy egy másik műveletet végre kell-e hajtani, illetve hogy mely művelet kerüljön sorra.
- **erőforrás-függőség** (resource dependency): egy művelet végrehajtásához olyan erőforrásra (pl. végrehajtóegységre) van szükség, amely éppen egy másik művelet végrehajtásával foglalkozik, így meg kell várni a másik művelet befejeződését (ez az ún. *álfüggőség* egy példája).

A párhuzamos programozás korlátai

• Szinkronizáció

A szinkronizáció olyan tetszőleges mechanizmus, amely két párhuzamos feladat végrehajtása között megbízható, előre megjósolható, determinisztikus kapcsolatot teremt.

Szinkronizáció nélkül előfordulhat, hogy bizonyos műveletek párhuzamos végrehajtás esetén bizonyos konkrét körülmények között helytelen eredményeket szolgáltatnak, pedig maguk a műveletek algoritmikus szempontból helyesek, illetve akár a program látszólagos leállását vagy két állapot közötti végtelen váltakozását eredményezhetik.

Amennyiben két feladat között nincs szükség szinkronizációra, ezeket aszinkron feladatoknak nevezzük, ezzel szemben az egymással szorosan összefüggő műveleteket szinkron feladatoknak nevezzük. (A valóságban a feladatok e két véglet között egy folytonos spektrumon helyezkednek el.)

Párhuzamos programozási modellek

Osztályozás

Párhuzamos rendszerek Flynn-féle osztályozása

Párhuzamos rendszerek modern osztályozása

Elméleti (idealizált) modellek áttekintése

A PRAM-modell

Az adatfolyam-modell

A feladat/csatorna-modell

Gyakorlati (hardverben is megvalósított) modellek áttekintése

Végrehajtási modellek

Memóriamodellek

Kommunikáció

Kommunikációs modellek és hálózati topológiák

A kommunikáció időbeli modellezése

Párhuzamos rendszerek osztályozása

- **Az osztályozás elősegíti a rendkívül sokféle megoldás áttekintését.**

Lényegében minden osztályozás egységes rendszert kínál a különböző megoldások jellemzőinek tárgyalásához.

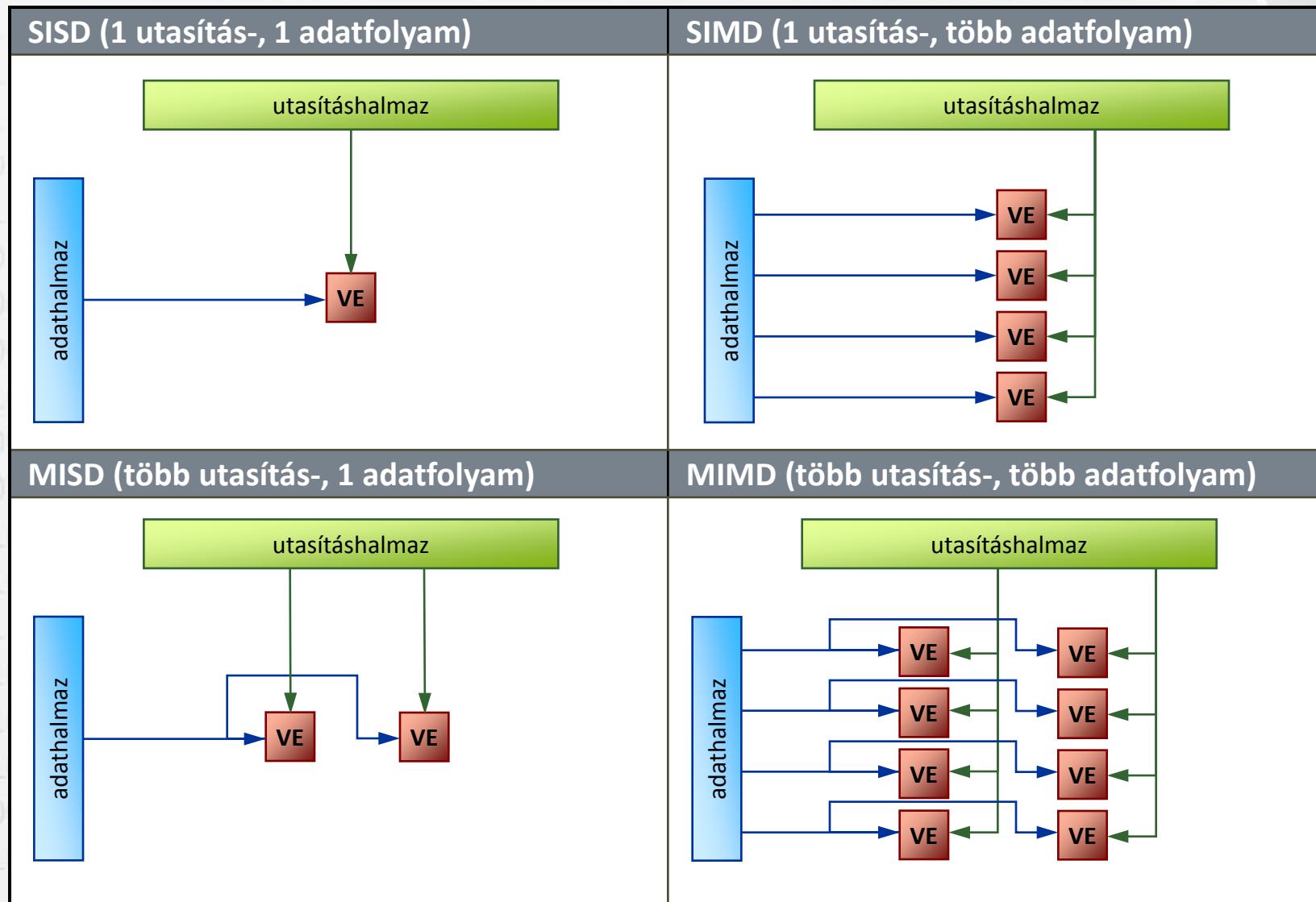
- **A párhuzamos rendszerek osztályozása Michael J. Flynn szerint**

Az 1966-ból származó Flynn-féle osztályozás a párhuzamos programozás legismertebb és legnépszerűbb tárgyalási kerete. Alapfeltevése, hogy minden („átlagos”) számítógép utasításfolyamokat hajt végre, melyek segítségével adatfolyamokon végez műveleteket. Az utasítás- és adatfolyamok száma

SISD (1 utasítás-, 1 adatfolyam)	SIMD (1 utasítás-, több adatfolyam)
Soros működésű, „hagyományos” számítógépek.	Vektorszámítógépek. Több formában létezett már; ma SSEx kiterjesztések, ill. DSP és GPGPU architektúrák formájában élő reneszánszát.
MISD (több utasítás-, 1 adatfolyam)	MIMD (több utasítás-, több adatfolyam)
Hibatűrő architektúrák (pl. Űrrepülőgép), melyeknél több VE is elvégzi ugyanazt a műveletet, és az eredményeknek egyezniük kell.	Teljesen párhuzamos számítógép, melynél minden végrehajtóegység külön-külön programozható fel.

A Flynn-féle osztályozás

- Illusztráció



A Flynn-féle osztályozás fogyatékosságai

- A MISD kategória nehezen kezelhető.
- Az utasítások és az adatok áramlása a valóságban nem folytonos.
 - Általában a memóriából származnak az adatok, és bizonyos méretű „löketekben” (soronként) kerülnek a gyorsítótárakba.
- Eltúlozza az egy, illetve a több utasításfolyamot kezelő modellek közötti megkülönböztetés fontosságát.
- Nehezen finomítható, nem ismeri el köztes osztályok létezését.

Az SPMD (1 program, több adatfolyam) modell esetében például minden végrehajtóegység egyazon programot hajt végre, de a programon belül egy-egy időpillanatban más-más pozícióban is tartózkodhatnak.

Az MPMD (több program, több adatfolyam) modellnél egy végrehajtóegység egy „mesterprogramot” futtat, amely a többi feldolgozóegység számára egy közös „alprogramot” oszt szét, és ez utóbbi valósítja meg a több adatfolyam feldolgozását.

- **Erősen hardverközpontú**

Ez az osztályozási szempontrendszer kevéssé segíti párhuzamos algoritmusok kialakítását.

Párhuzamos rendszerek modern osztályozása

- Ma már a Flynn-féle osztályozáshoz viszonyítva sokkal árnyaltabb szempontrendszer alakítható ki.

- Programozási nyelv és környezet dimenziója: hagyományos, bővített hagyományos, explicit párhuzamos, agnosztikus stb.
- Végrehajtási elemek együttműködésének dimenziója: osztott adatszerkezetek, üzenetváltás stb.
- Szemcsézettség dimenziója: implicit párhuzamosság, fordítóprogram által vezérelt párhuzamosság, többpéldányos hardverelemek, osztott memória kontra hálózati kommunikáció stb.
 - Ez a szempont inkább folytonos spektrumnak tekinthető, mintsem diszkrét értékhalmaznak.

Elméleti (idealizált) modellek 1

• PRAM (Parallel Random Access Machine)-modell

Az elméleti RAM-gép (Random Access Machine; gyakorlatilag az összes Neumann-elvű számítógép) továbbgondolása párhuzamos formában.

A PRAM lényegében egy osztott memóriájú absztrakt számítógép, amely párhuzamos algoritmusok bonyolultságának elemzését teszi lehetővé elvben tetszőleges számú feldolgozóegységgel. Mivel elhanyagolja a kommunikáció és a szinkronizáció problémáit, a rá készített algoritmusok becsült „költsége”:

$$O(időigény * processzorszám)$$

A memóriaírási és -olvasási műveletek ütközésének (egyazon memóriahelyre való egyidejű írási és olvasási igények) kezelésére négy lehetőséget ad:

- EREW: kizárasos (exkluzív) olvasás, kizárasos (exkluzív) írás
- CREW: egyidejű olvasás, kizárasos írás
- ERCW: kizárasos olvasás, egyidejű írás
- CRCW: egyidejű olvasás, egyidejű írás

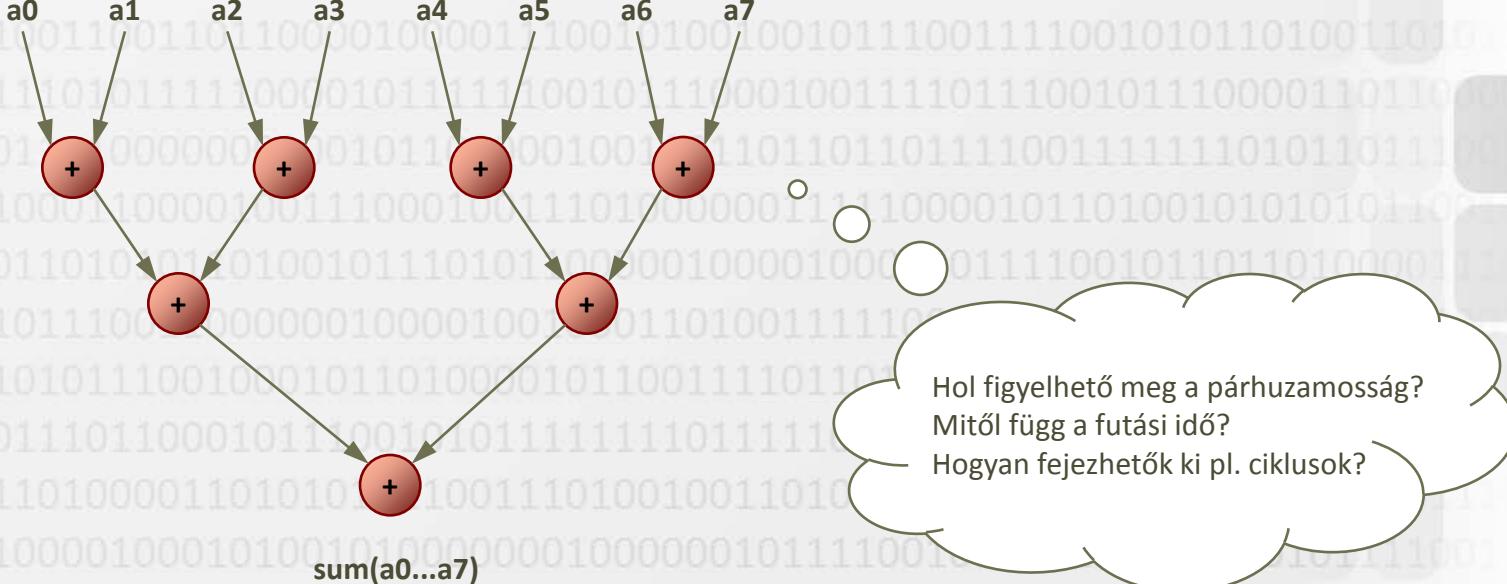
Az egyidejű olvasás nem probléma, az egyidejű írás kezelési lehetőségei:

- közös: minden írási művelet ugyanazt az adatot írja, ellenkező esetben hiba történt
- önkényes: az egyik írás sikeres, a többi eredménytelen (nem determinisztikus)
- prioritásos: a feldolgozóegységek fontossági mutatója dönti el, melyik írás lesz sikeres
- egyéb lehetőségek (AND, OR, SUM stb. műveletekkel kombinált eredmény beírása)

Elméleti (idealizált) modellek 2

• Adatfolyam-gráf modell

Ennél a modellnél a bemenő adatfolyamokból a modellstruktúrát felépítő feldolgozóegységek kimenő adatfolyamo(ka)t állítanak elő. Az adatok közötti függőségek és a párhuzamos véghajtás módja adatfolyam-ábra segítségével illusztrálható.





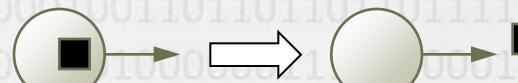
Elméleti (idealizált) modellek 3

- **Feladat/csatorna (Task/Channel)-modell (Ian T. Foster, 1995)**

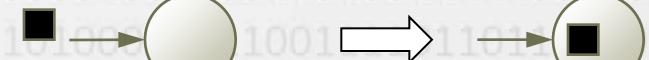
Ez a modell a párhuzamosan elvégzendő számításokat tetszőleges számú, egyidejűleg végzett feladatok halmazaként fogja fel, amelyeket egymással kommunikációs csatornák kapcsolnak össze. Egy feladat önmagában egy hagyományos soros végrehajtású program, melynek helyi memória, valamint be- és kimeneti portok állnak rendelkezésre.

A feladatok végrehajtása során a helyi memória tartalmának olvasása és írása mellett négy alapműveletre kerülhet sor:

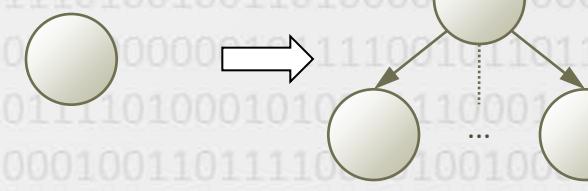
- Üzenet küldése



- Üzenet fogadása



- új feladat létrehozása



- feladat befejeződése



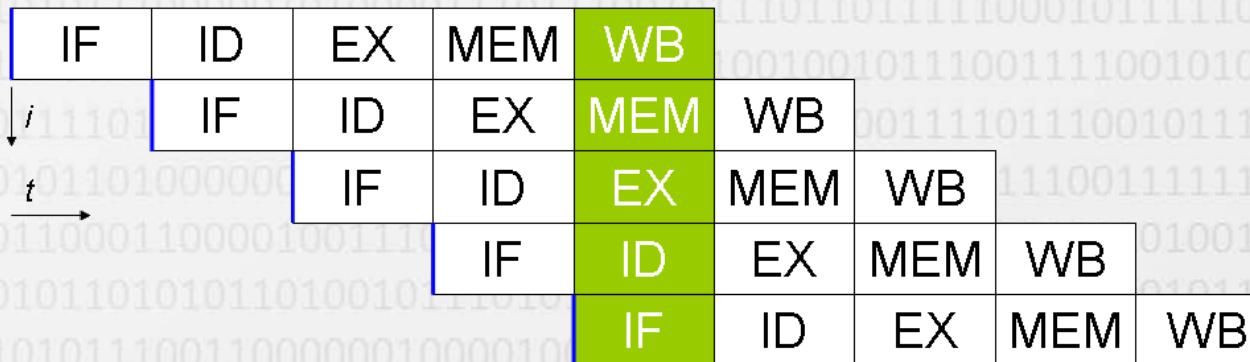


Gyakorlati modellek 1: végrehajtási modellek

• Futószalagelvű végrehajtás

Az elv lényege, hogy a végrehajtást ún. lépcsőkre osztjuk, és az egyes utasításokat ezen a többlépcsős „futószalagon” folyamatosan, azaz órajelenként egyesével „léptetjük végig”.

Az alábbi példa egy 5-lépcsős klasszikus RISC futószalagot ábrázol:



Ábramagyarázat: IF = utasításlehívás (Instruction Fetch), ID = dekódolás (Instruction Decode), EX = végrehajtás (Execute), MEM = memória-hozzáférés (Memory Access), WB = visszaírás (Register Write Back). A vízszintes tengely (t) az időt, míg a függőleges tengely (i) az utasítások sorozatát jelöli. A zöld színű oszlopból látható, hogy az első utasítás már a WB lépcsőnél tart, miközben ezzel egy időben pl. az utolsó (azaz ötödik) utasítás lehívása van folyamatban.

A sebességnövelés korlátját itt elsősorban a lépcsők száma, a leglassabb lépcső végrehajtási ideje, valamint a váratlan események (gyorsítótárban nem található meg az adat, elágazás következett be stb.) szabják meg.



Gyakorlati modellek 1: végrehajtási modellek

- **Szuperskalár végrehajtás**

Ennél a megoldásnál egyszerre több futószalag működik párhuzamosan.

Az alábbi példa egy 2-utas (két futószalagos) szuperskalár megoldást ábrázol:

IF	ID	EX	MEM	WB	
IF	ID	EX	MEM	WB	
i	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
t	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB

A szuperskalár végrehajtás szélesítésének elvi gátat szabnak a valódi adatfüggőségek, amelyek mai programokban csak legfeljebb 4–6 utasítás egyidejű végrehajtását teszik lehetővé.

Gyakorlati modellek 1: végrehajtási modellek

• Szuperskalár végrehajtás: az optimalizálás szerepe

Az alábbi három kódpélda szemantikailag (értelmét és eredményét tekintve) egyenértékű, mégis egészen eltérő teljesítményjellemzőkkel bír:

- A**
1. load R1, @1000
 2. load R2, @1008
 3. add R1, @1004
 4. add R2, @100C
 5. add R1, R2
 6. store R1, @2000

- B**
1. load R1, @1000
 2. add R1, @1004
 3. add R1, @1008
 4. add R1, @100C
 5. store R1, @2000

- C**
1. load R1, @1000
 2. add R1, @1004
 3. load R2, @1008
 4. add R2, @100C
 5. add R1, R2
 6. store R1, @2000

0 1 2 3 4 5 6
—→ t (órajelciklus)

IF	ID	OF
IF	ID	OF

1. load R1, @1000
2. load R2, @1008

IF	ID	OF	E
IF	ID	OF	E

3. add R1, @1004
4. add R2, @100C

IF	ID	NA	E

5. add R1, R2
6. store R1, @2000

IF = utasításlehívás
ID = dekódolás

OF = operanduslehívás

EX = végrehajtás

WB = visszaírás

IF	ID	NA	WB

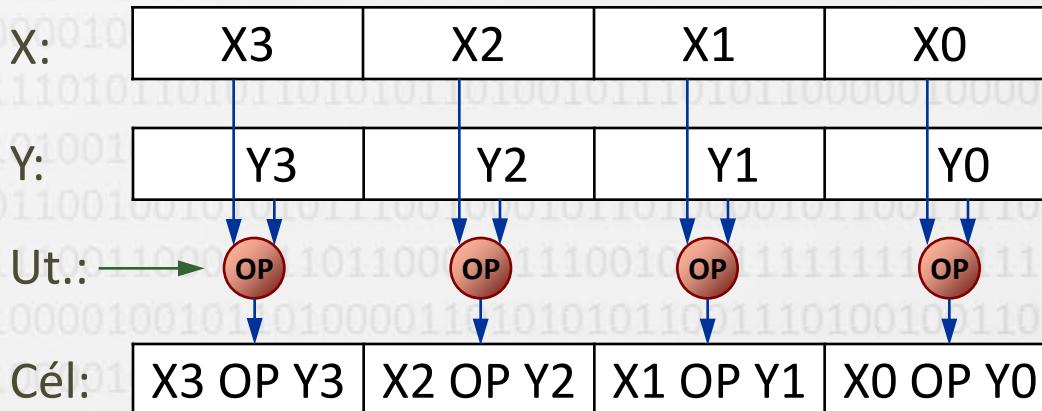
Milyen feladatot old meg a fenti 3 példa?
Hogyan hajtódik végre a B és a C megoldás?
Melyik megoldás lesz a leggyorsabb,
illetve a leglassabb? Miért?

Gyakorlati modellek 1: végrehajtási modellek

- **Vektoralapú (SIMD) végrehajtás**

A vektoralapú megoldások lényege, hogy a fordítóprogram automatikusan (vagy programozói segítséggel) észleli a függőségek nélküli azonos kódrészleteket. SIMD végrehajtás esetén a program más-más adatokon végzi el ugyanazt a műveletsort, ezt pedig párhuzamosan is végre lehet hajtani.

A SIMD műveletek végrehajtási elvét illusztrálja az alábbi ábra és kódpélda*:



```
void Quarter(int[] array)
{
    for (int i = 0; i < array.Length; i++)
        array[i] = array[i] >> 2;
}

void QuarterSSE2(int[] array)
{
    ...
$B1$10:
    movdqa    xmm0, XMMWORD PTR [edi+edx*4]
    psrad
    movdqa    XMMWORD PTR [edi+edx*4], xmm0
    add      edx, 4
    cmp      edx, ecx
    jb       $B1$10
    ...
}
```

* Intel Corporation, „The Software Optimization Cookbook”, 2006, ISBN 0-9764832-1, 192. és 195-196. oldal

Gyakorlati modellek 1: végrehajtási modellek

• Vektoralapú (VLIW) végrehajtás

A VLIW végrehajtásnál egy igen hosszú („very long”) gépi kódú utasításszóba („instruction word”) a fordítóprogram a megfelelő elemzést követően számos elemi utasítást egymás mellé „csomagol”, és az egy „csomagban” szereplő utasítások párhuzamosan hajtódnak végre.

Sajnos a párhuzamosítási lehetőségeket előre felismerni és megfelelően kiaknázni képes fordítóprogramok készítése egyelőre a lehetetlenséggel határosan nehéz*.

Példa VLIW műveletekre:

Utasítás

$n-1$
n	BRANCH	ADD	ADD	MUL
$n+1$

* Donald E. Knuth, „Interview with Donald Knuth”, 2008. április 25., <http://www.informit.com/articles/article.aspx?p=1193856>

Gyakorlati modellek 1: végrehajtási modellek

• Szál szinten párhuzamos végrehajtás

Ennél a modellnél azt használjuk ki, hogy egy végrehajtási szálon belül biztosan lesznek feloldhatatlan adatfüggőségek, amelyek korlátozzák a párhuzamosítást. A várakozások miatt felszabaduló számítási kapacitást a modell úgy igyekszik növelni, hogy egy-egy végrehajtóegységet egynél több szálból is „ellát” feladatakkal. Általában az operációs rendszer ezt a beépített szál szintű párhuzamosságot több „logikai” feldolgozóegységgel kezeli.

A szál szintű párhuzamosság főbb megvalósítási lehetőségei:

- szimmetrikus többszálú feldolgozás (SMT)
 - Egy processzoron belül bizonyos részegységek (de nem a teljes processzor) többszörözése.
 - Az Intel ezt „Hyperthreading” néven valósította meg.
- többmagos processzor
 - Egy fizikai egységen (tokon) belül több azonos, teljes értékű processzor, melyek egymáshoz speciális adatutakkal is kapcsolódhatnak.
- többmagos processzor SMT-vel
 - Az első két megoldás kombinációja.

Gyakorlati modellek 2: memóriamodellek

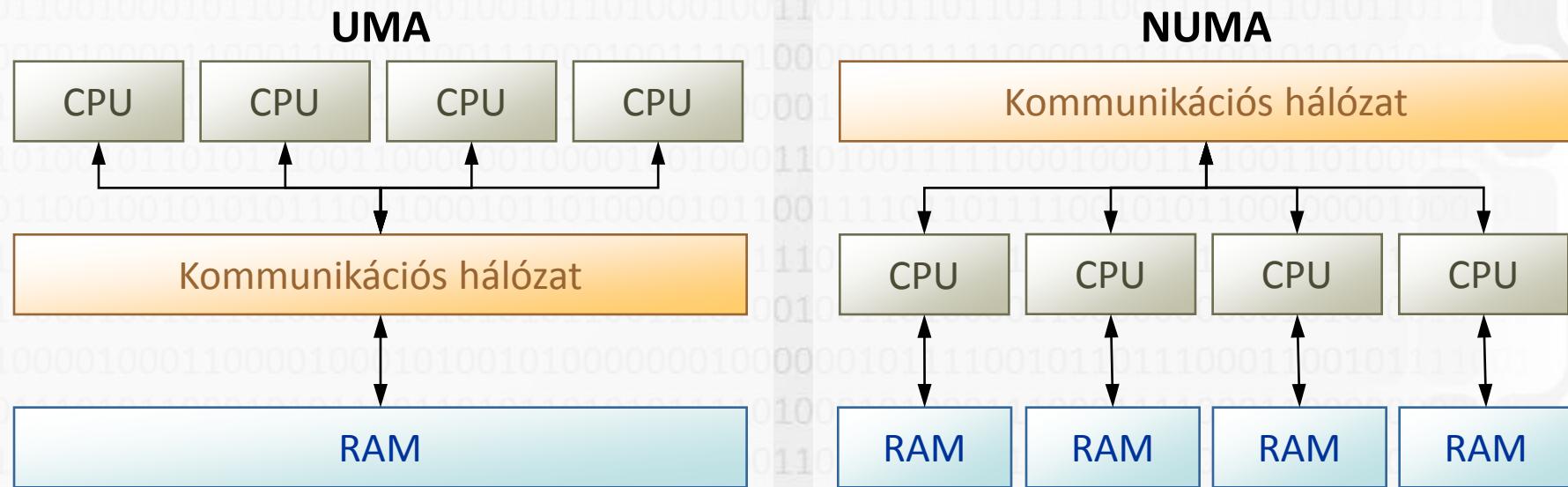
- Az UMA- és a NUMA-modell

- UMA: Uniform Memory Access (közös memória)

- minden processzor egyetlen közös memóriában tárolja az adatokat.
 - más néven SMP (symmetric multiprocessor)-architektúra.

- NUMA: Non-Uniform Memory Access (elosztott memória)

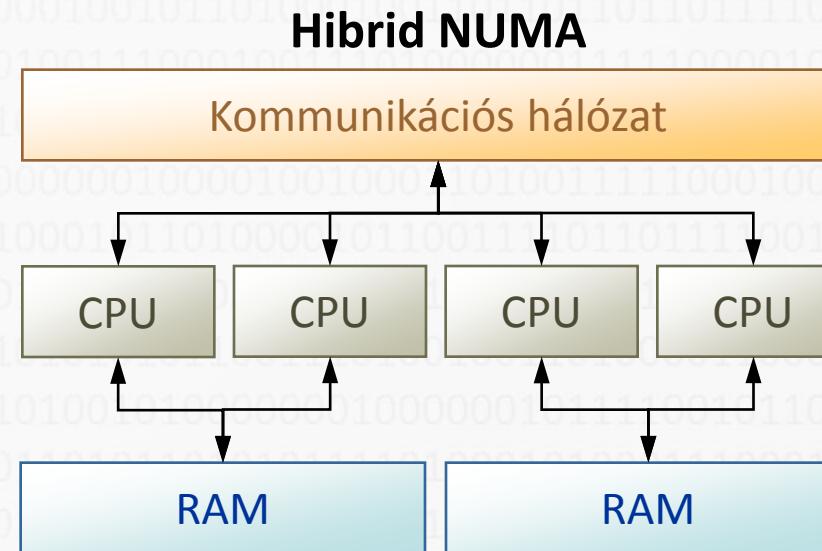
- minden processzor saját, független memóriával rendelkezik.
 - más néven DM (distributed memory)-architektúra.



Gyakorlati modellek 2: memóriamodellek

- **A Hibrid (N)UMA-modell**

Ez a rendszer egynél több memóriablokkot tartalmaz (tehát nem tisztán UMA), de ezeket a processzorok közösen is használják (tehát nem is tisztán NUMA). A nagy teljesítményű párhuzamos számítógépek (más néven „szuperszámítógépek”) körében egyértelmű trend a hibrid NUMA megoldás terjedése.

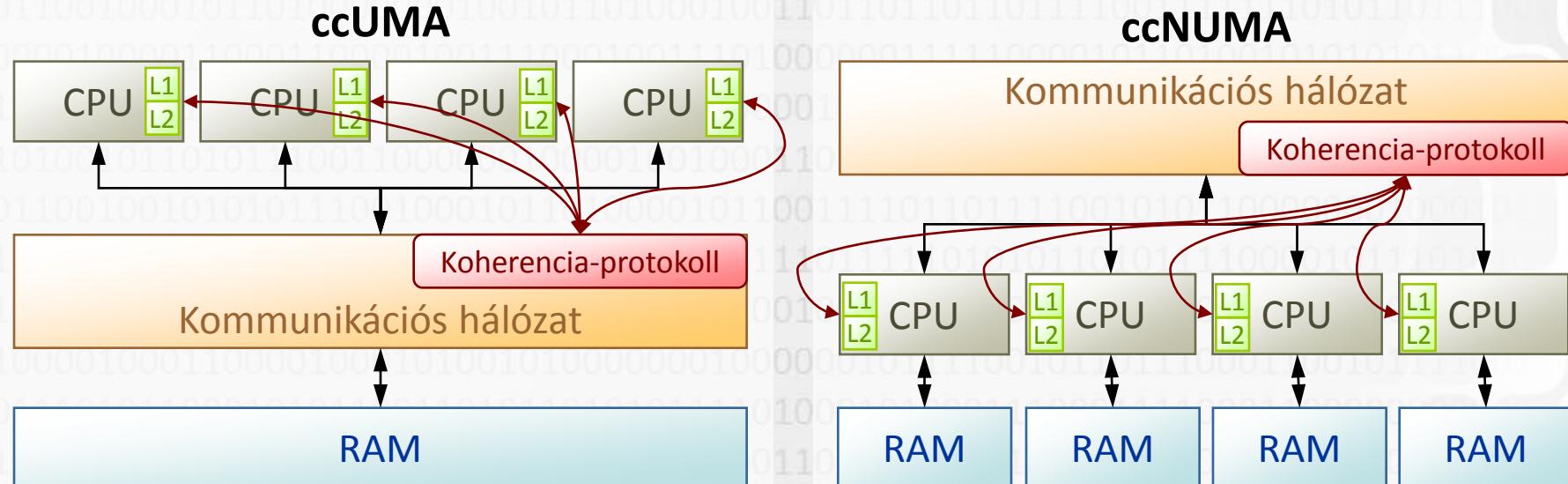


Gyakorlati modellek 2: memóriamodellek

- A ccUMA- és a ccNUMA-modell

A processzorok minden rendelkeznek gyorsítótárrakkal, melyek a memóriából betöltött, illetve oda kiírandó adatok egy részét tárolják. Ezek tartalmának összehangolása külön feladatként merül fel, mely az UMA és NUMA modellekben igen bonyolultan oldható meg.

A ccUMA (cache-coherent UMA) és ccNUMA modellekben speciális, egy ún. koherencia-protokollt megvalósító egység gondoskodik az összehangolásról.



Kommunikáció és kommunikációs modellek

- A párhuzamos rendszerek elemei közötti kommunikáció tervezése a teljesítménynövelés szempontjából döntő tényező.

A feldolgozás részeredményeinek továbbítása az egyes feldolgozóegységek között (azaz a feldolgozóegységek kommunikációja) a párhuzamos programozás „szükséges rossz” jellegű tényezője.

Az algoritmusok kialakításánál ezért szintén lényeges feladat a kommunikációs igények minimalizálása, a megfelelő szoftveres kommunikációs struktúra megtervezése.

Kommunikációs hálózati topológiák

- **Egyeszerű topológiák:**

- teljes
- csillag
- lineáris (1D) tömb
- 2D és 3D tömb (átfordulással vagy anélkül)

Kommunikációs hálózati topológiák

- **Hiperkocka:**

- 0D, 1D, 2D, 3D, 4D hiperkocka

Kommunikációs hálózati topológiák

- **Fák:**

- statikus bináris fa
- dinamikus bináris fa

Hálózati topológiák jellemzési szempontjai

- **Átmérő**

Bármely két csomópont között mérhető legnagyobb távolság.

- **Szomszédos élek száma**

Bármely két csomópont közötti összeköttetések száma.

- **Kapcsolati számosság**

Az a minimális élszám, melynek eltávolítása esetén a hálózat két, egymástól független hálózatra válik szét.

- **Felezési szélesség**

Az a minimális élszám, melynek eltávolítása esetén a hálózat két, egymástól független, **egyforma** hálózatra válik szét.

- **Felezési sávszélesség**

A hálózat bármely két része között minimálisan meglévő átviteli sávszélesség.

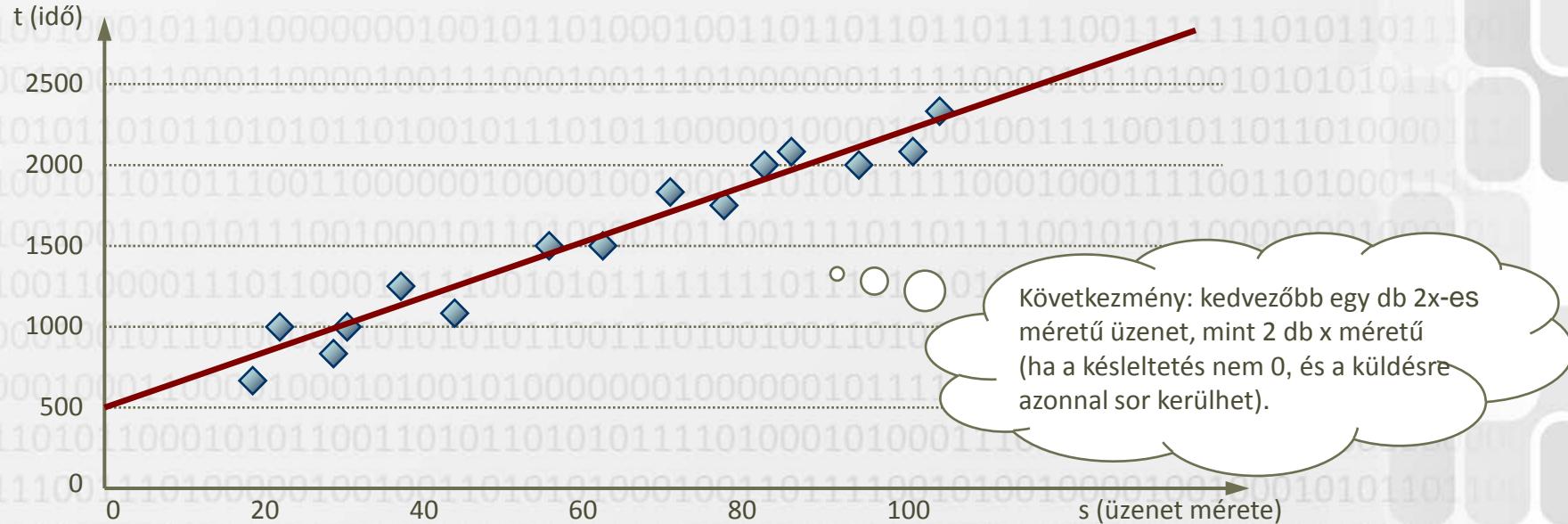
- **Költség**

A kapcsolatok számával, átviteli paraméterekkel stb. arányos jellemző.

A kommunikáció időbeli modellezése

- A kommunikáció időbeli lefolyásának modelljét két fő tényező jellemzi:
 - késleltetés (latency): egy-egy üzenet indítása, illetve megérkezése között eltelt idő;
 - sávszélesség (bandwidth): időegységenként átvihető maximális adatmennyiség.

Általánosságban a kommunikáció idődiagramja az alábbihoz hasonló:



A kommunikáció időbeli modellezése

- A kommunikáció hatékonyságának növeléséhez elengedhetetlen a fenti két fő tényező kedvezőbbé tétele.

A sávszélesség növelését általában a kommunikációs hálózat közegének fizikai adottságai korlátozzák, a késleltetés azonban a struktúra kialakításának változtatásával, illetve jobb algoritmusokkal csökkenthető.

A késleltetés csökkentésének lehetőségei (példák):

- nem blokkoló küldési és fogadási műveletek

A küldő és a fogadó oldalon pufferek segítségével biztosítjuk, hogy pusztán a kommunikáció tényle miatt ne kelljen várakoznia a feldolgozóegységeknek.

- többszálú programozás alkalmazása

Amikor egy-egy feldolgozási szál várakozásra kényszerül (például egy olvasási vagy írási művelet befejezése érdekében), más szálak ezzel egy időben más feladatokat végezhetnek, így a rendszer átbocsátó képessége javul.

Párhuzamos algoritmusok tervezési lehetőségei és módszerei

A párhuzamos működés célkitűzései

Teljesítménynövelés

Rendszerelemek teljesítménye közötti „sebességolló” kompenzálása

Óriási számítási kapacitást igénylő feladatok megoldása

Párhuzamos rendszerek programozásának elvi lehetőségei

Párhuzamos algoritmusok tervezése

Tervezési módszerek

Felosztás, kommunikáció, összevonás, leképezés

A párhuzamos teljesítménynövelés határai

Amdahl törvénye, Gustafson törvénye

Programhelyességi szempontok és fogalmak

Kritikus szakasz, versenyhelyzet, holtpont, „élőpont”

A párhuzamos működés célkitűzései

1. A Moore-törvény adta lehetőség kiaknázása: teljesítménynövelés

Gordon Moore 1975-ben tette közzé (kissé pontosítva tíz évvel korábbi megfigyelését), hogy a lapkákon lévő áramkörök bonyolultsága 18 havonta megduplázódik.

Ezt az empirikus „törvényt” azóta kiterjesztett értelemben a rendelkezésre álló számítási kapacitás növekedési ütemére is használják.

A kapacitás növekedésének kihasználása azonban ma már nem egyszerű, a többletként jelentkező tranzisztorszámot további teljesítménynövekedéssé alakítani igen komplex feladat.

A feladat legkézenfekvőbb, logikus megoldása az implicit és explicit párhuzamosság kiaknázása. Míg az előbbi a korábban tárgyalt gyakorlati (hardverben is megvalósított) párhuzamos programozási modellek teszik lehetővé, a tárgy fő témaköre az utóbbival foglalkozó, párhuzamos algoritmusokra épülő programozási módszerek.

A párhuzamos működés célkitűzései

2. A memóriák és háttértárak alacsony sebességének kompenzációja

A processzorok sebessége és a memória, illetve a háttértárak sebessége között több nagyságrend különbség van, és az eszközök sebességének növekedése során ez a „sebességolló” egyelőre tovább tágul.

Áthidalására általában többszintű gyorsítótárazási technikákat alakítanak ki:

Tártípus	Méret* (2010)	Sebesség* (2010)
Regiszter	~256 B	~100 GB/s
Belső (L1) gyorsítótár	~128 KB	~20 GB/s
Belső (L2) gyorsítótár	~2 MB	~15 GB/s
Belső/külső (L3) gyorsítótár	~4 MB	~12 GB/s
Központi memória	~4 GB	~8 GB/s
Helyi háttértár	~512 GB	~0,2 GB/s
Hálózati háttértár	~4 TB	~0,05 GB/s

A párhuzamos megoldások azért is képesek sokkal nagyobb teljesítményre, mert összességében jóval nagyobb gyorsítótár-terület és memória-sávszélesség áll rendelkezésükre.

* Az Intel Core i7 processzor egy magjára vonatkoztatott hozzávetőleges tervezési adatok.

A párhuzamos működés célkitűzései

3. Óriási számítási kapacitást igénylő feladatok megoldása

– Tudományos és kutatási célú alkalmazások:

- bioinformatika: genomfeltérképezés, fehérjeszintézis vizsgálata;
- kvantumfizika: kvantummechanikai folyamatok modellezése és vizsgálata;
- asztrofizika: galaxisok kutatása, termonukleáris folyamatok szimulációja;
- kémia: makromolekuláris szerkezetek kutatása;
- időjárási folyamatok modellezése, árvizek és földrengések előrejelzése.

– Mérnöki és tervezési célú alkalmazások:

- mikro- és nano-elektromechanikai rendszerek („nanorobotok”) tervezése;
- nagysebességű áramkörök tervezése;
- diszkrét és folytonos optimalizációs feladatok megoldása;
- genetikus algoritmusok.

– Kereskedelmi célú alkalmazások

- tőzsdei tranzakciók kezelése;
- adatbányászat, adatelemzés, statisztika;
- skálázható webes közösségi programok.



Párhuzamos rendszerek programozása

- A párhuzamos rendszereken végzett fejlesztés kulcsfeladata a párhuzamosítási lehetőségek implicit és explicit kiaknázása.**

Az egyes párhuzamos algoritmusokat általánosságban a folyamatok tulajdonságai (struktúra, topológia, végrehajtás módja), a folyamatok közötti interakció jellemzői, illetve az adatkezelés módja (particionálás, elhelyezés) különböztetik meg egymástól.

- A fenti rendszereken az explicit párhuzamosság kiaknázására James R. McGraw és Timothy S. Axelrod szerint (1987) az alábbi érdemi lehetőségek állnak rendelkezésre:**

1. fordítóprogramok funkcionális bővítése;
2. programozási nyelvek bővítése;
3. párhuzamos programozási rétegek, illetve megoldási minták kialakítása;
4. új párhuzamos programozási nyelvek tervezése.



A fejlesztés 1. lehetséges iránya

- **Fordítóprogramok funkcionális bővítése**

Ennek során olyan, párhuzamosításra képes fordítóprogramok készülnek, amelyek képesek egy soros programozási nyelven korábban megírt programokban rejlő párhuzamosítási lehetőségek automatikus kiaknázására.

Előnyök:

- a már elkészült szoftverek hatalmas összegek és több ezer emberévnyi munka árán születtek meg;
- az automatikus párhuzamosítás pénzt és munkaidőt takarít meg ;
- több, mint húsz éve folynak a kutatások e témaban.

Hátrányok:

- a programozók és a fordítóprogramok gyakorlatilag egymás ellen dolgoznak, mert a soros vezérlési szerkezetek és ciklusok elrejtik a párhuzamosítási lehetőségeket, a fordítóprogram pedig ezeket nem képes maradéktalanul visszanyerni.



A fejlesztés 2. lehetséges iránya

• Programozási nyelvek bővítése

Ennek során egy soros programozási nyelvet bővítenek olyan funkciókkal, amelyek segítségével a programozók párhuzamos folyamatokat tudnak létrehozni, leállítani, szinkronizálni és egymással kommunikáltatni.

Előnyök:

- a legegyszerűbb és leggyorsabb módszer, mivel „csak” néhány függvénykönyvtárat kell kialakítani;
- szinte minden létező párhuzamos számítógépfajtához léteznek már bővítőkönyvtárak;
- rugalmas lehetőséget ad a programozóknak a továbbfejlesztésre.

Hátrányok:

- a fordítóprogram nem vesz részt a párhuzamos kód kialakításában, így az esetleges hibákat sem tudja jelezni;
- nagyon könnyen lehet olyan párhuzamos programokat írni, amelyekben a hibakeresés a lehetetlenség határáig bonyolult.



A fejlesztés 3. lehetséges iránya

- **Párhuzamos programozási rétegek kialakítása**

Ennek során két vagy több szoftverréteget alakítanak ki. Az alsó rétegek tartalmazzák a lényegi számítási műveleteket, amelyek a forrásadatokból eredményeket állítanak elő. A felső réteg szabályozza a feldolgozási folyamatok létrehozását, szinkronizációját és befejezését.

Ezeket a szinteket egy fordítóprogram fésüli össze a párhuzamos számítógéprendszeren futtatható kóddá.

Előnyök:

- a párhuzamos programok irányított gráfként jeleníthetők meg, melyek csomópontjai önmagukban soros műveleteket, élei pedig a műveletek közötti adatfüggőségeket jelentik.

Hátrányok:

- a programozóknak egy új, párhuzamos programozási rendszert kell megismerniük és használniuk.



A fejlesztés 4. lehetséges iránya

• Új, párhuzamos programozási nyelvek készítése

Ennek során teljesen új, előd nélküli párhuzamos programozási nyelvek születnek, amelyeken a programozó explicit módon fejezhet ki párhuzamos műveleteket.

- Egy közismert példa az Occam-nyelv (<http://wotug.ukc.ac.uk/parallel/occam/>)

Előnyök:

- az explicit párhuzamosság következtében a programozó és a fordítóprogram egymást segíti.

Hátrányok:

- új fordítóprogramokat kell kifejleszteni, ami jó minőségben általában több évet igényel.
- a párhuzamos nyelvek szabvánnyá válása, és így a bennük megírt kód hordozhatósága nem garantált (pl. C*);
- programozói ellenállás („ki akar még egy új nyelvet megtanulni?”).



Párhuzamos algoritmusok tervezése

- A feladat/csatorna-modellre építve Ian Foster négylépéses tervezési módszert (**PCAM**) alakított ki:

- Felosztás (partitioning)

- A számítás és/vagy az adatok felbontása kisebb részekre a gyakorlati szempontok (pl. a célszámítógép feldolgozóegységeinek száma) figyelmen kívül hagyásával, kizárálag a párhuzamosítási lehetőségek maximális kiaknázására törekedve.

- Kommunikáció (communication)

- A feladatok végrehajtásához szükséges kommunikációs igények megállapítása, a megfelelő struktúrák és algoritmusok kialakítása.

- Összevonás (agglomeration)

- Az első két lépésben kialakult feladatok és kommunikációs struktúrák felülvizsgálata a teljesítményigény és a megvalósítási költségek figyelembe vételevel: ha szükséges, feladatok összevonása a teljesítmény javítása és/vagy költségcsökkentés céljából.

- Leképezés (mapping)

- Feladatok hozzárendelése az egyes feldolgozóegységekhez a processzorkihasználás maximalizálása és a kommunikáció minimalizálása céljából (e két cél gyakran egymással ellentétes, ezért egyensúlyra kell törekedni). A hozzárendelés lehet statikus vagy dinamikus (azaz futás közben változtatható) jellegű.



Párhuzamos algoritmusok tervezése

- Illusztráció: a Foster-féle négylépéses tervezési módszer

1. Felosztás

Adatok és számítások szétbontása kisebb, jól kezelhető részekre.

2. Kommunikáció

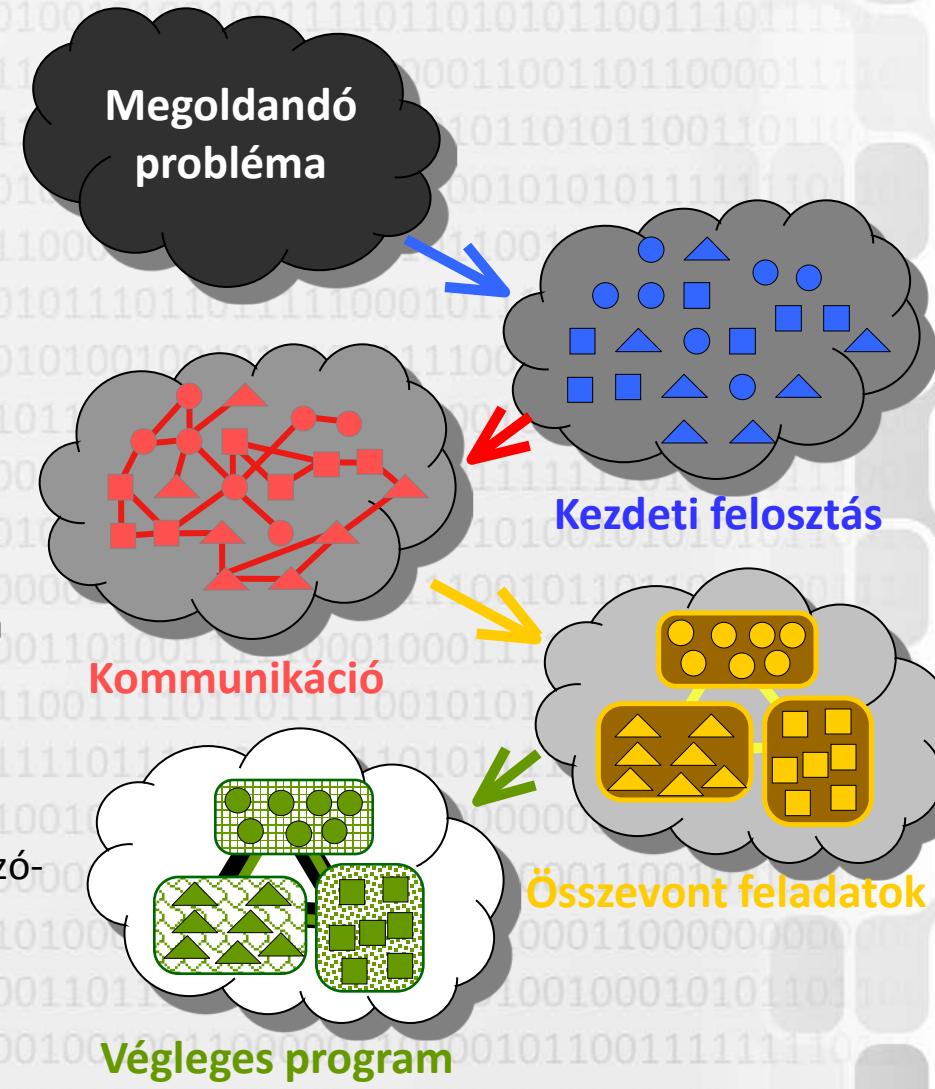
Adatok megosztása a számítások között, a megfelelő struktúrák kialakítása.

3. Összevonás

A feladatok részleges csoportosítása a nagyobb teljesítmény, illetve költséghatékonyúság érdekében.

4. Leképezés

A kapott feladatok konkrét feldolgozó-egységekhez rendelése.



1. lépés: felosztás (dekompozíció)

- A lépés célja a párhuzamosság összes lehetséges forrásának felfedezése.
- A tervezés során itt kerül sor a számítások és az adatok kisebb egységekre bontására.
- A dekompozíció két alaptípusa:
 - adatfelosztás (adatok felosztása kisebb egységekre)
 - Ennek során először az adatokat bontjuk fel kisebb egységekre, majd meghatározzuk, hogyan rendelhetők az adatokhoz az elvégzendő számítások.
 - Példa: egy 3D mátrix felosztható (1) 2D szeletekre, amelyeket egy 1D feladatlista segítségével dolgozunk fel, (2) 1D szeletekre, amelyeket egy 2D feladatlista segítségével dolgozunk fel, vagy (3) egyedi elemekre, amelyeket egy 3D feladatlista segítségével dolgozunk fel.
 - funkcionális felosztás (műveletek felosztása kisebb egységekre)
 - Ennek során először a feladatot bontjuk fel részfeladatokra, majd minden részszámításhoz hozzárendeljük a szükséges adatokat.
 - Példa: számozott színmezők kifestése, ahol minden egyes színnel való festés egy-egy alfeladat, melyekhez adatként a szín és az adott színszámú mezők halmaza tartozik.



A dekompozíció hatékonyságának vizsgálata

- Mivel egy-egy feladat dekompozíójára számos lehetőség nyílik, szükség van a dekompozíció jóságának ellenőrzésére is.
- Foster az alábbi fő szempontokat nevezte meg:
 - legalább egy nagyságrenddel több részfeladat legyen, mint végrehajtóegység.
 - Ellenkező esetben a későbbiekben gyorsan leszűkülnek a bővítési lehetőségek.
 - kerülendők a felesleges (redundáns) számítások és tárigények.
 - Amikor később a probléma mérete növekszik, ezek a felesleges igények fokozatosan lerontják a megoldás alkalmazhatóságát.
 - a részfeladatok mérete (időigénye) lehetőleg közel azonos legyen.
 - Ha ezt nem sikerül teljesíteni, akkor nehéz lesz minden feldolgozóegységnek azonos mennyiségű munkát biztosítani, így egyes feldolgozóegységek kihasználatlanul maradnak.
 - a részfeladatok száma a megoldandó feladattal együtt növekedjen.
 - Erre azért van szükség, hogy a végrehajtóegységek számának növekedésével (pl. új számítógépmodell) egyre nagyobb méretű feladatok legyenek megoldhatók.
- A rugalmasság érdekében érdemes több alternatívát is vizsgálni.



Függőségi gráf – dekompozíciós leírás

Gráf = csomópontok, irányított élek

Csomópont lehet:

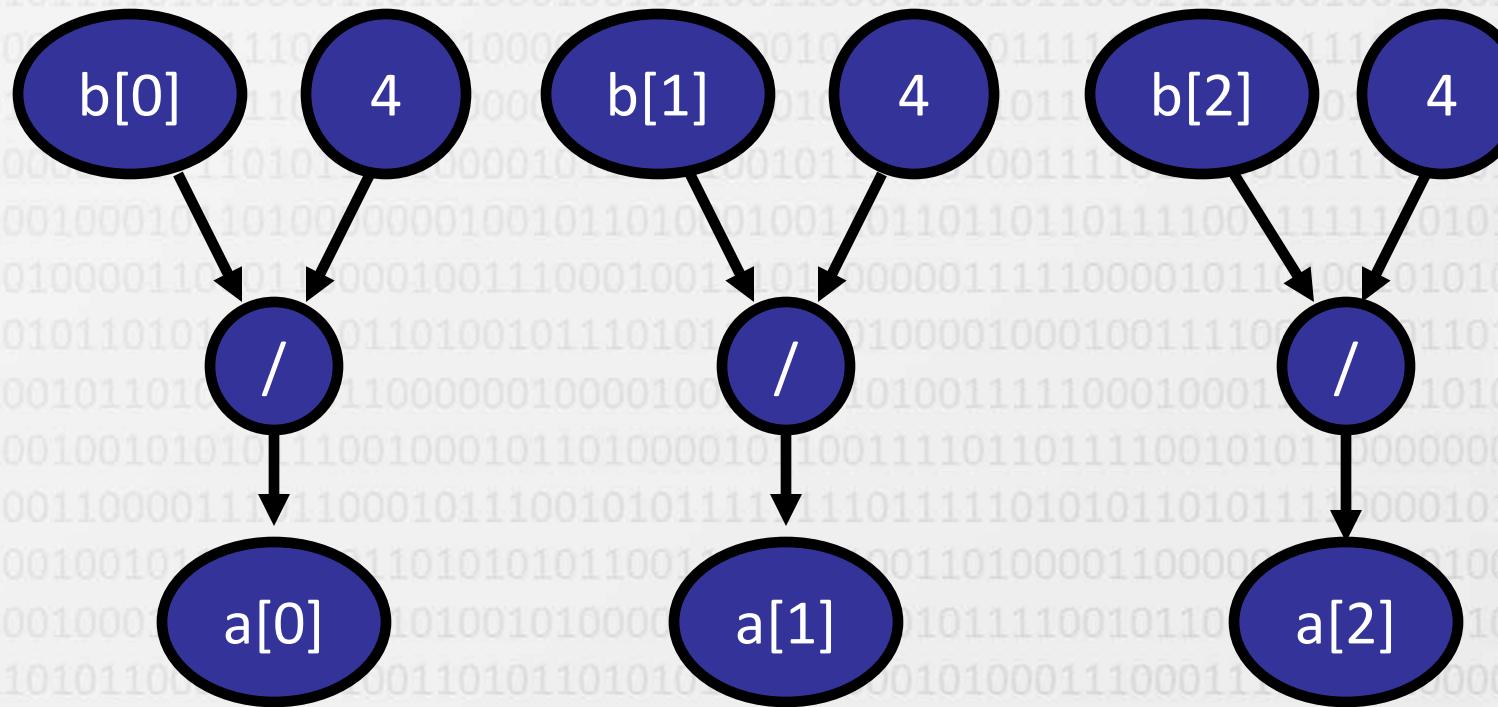
- változó hozzárendelés (**kivétel index**)
- konstans
- Operátor- vagy függvényhívás

Az élek a változók vagy a konstansok használatát jelölik:

- adatfolyamokban (**adatfüggőség**: egyik adat új értéke függ a másik adattól)
- vezérlési folyamatban (**vezérlésfüggőség**: „if (a < b) a = b;”)

Függőségi gráf 1. példa

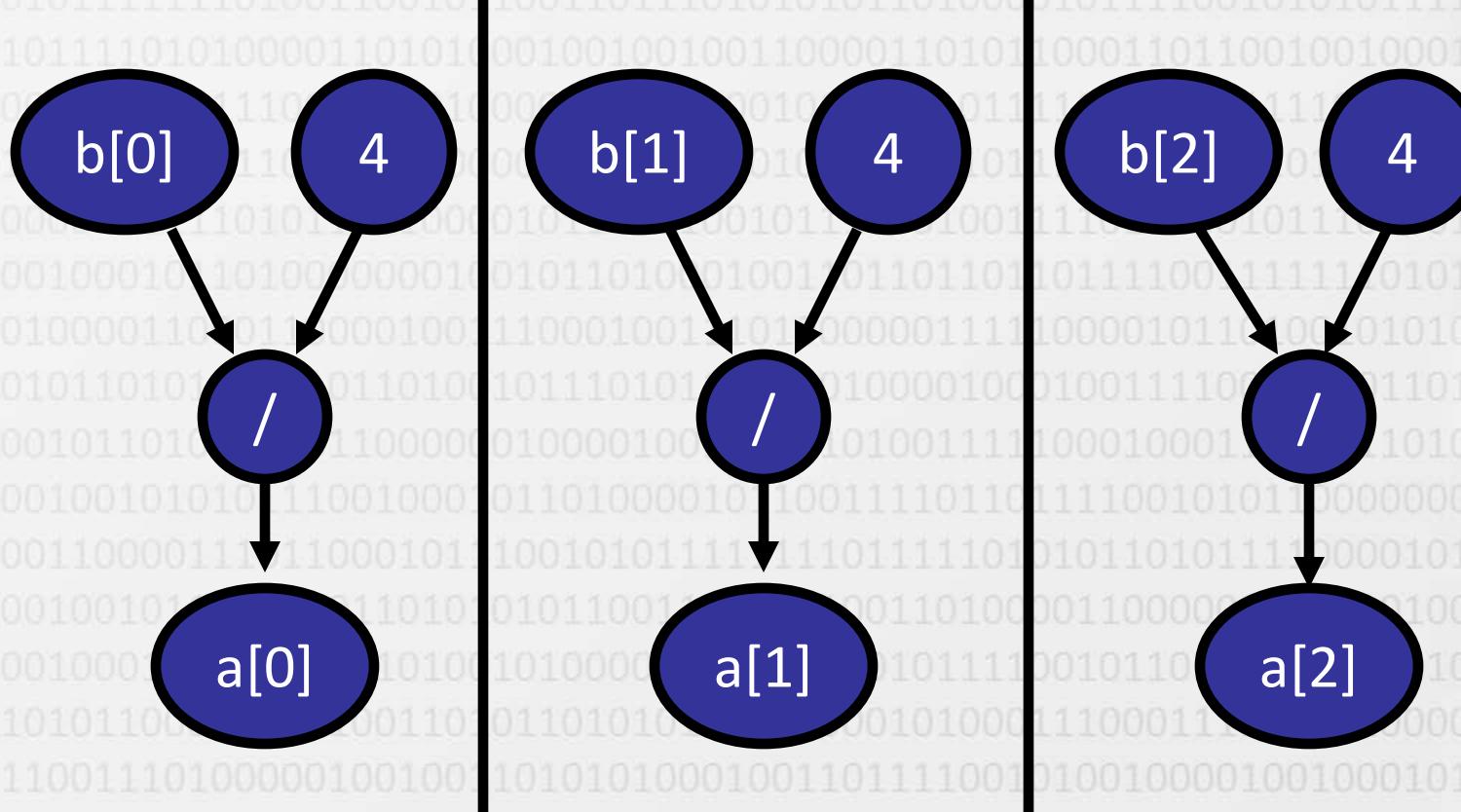
```
for (i = 0; i < 3; i++)  
    a[i] = b[i] / 4;
```



Függőségi gráf 1. példa

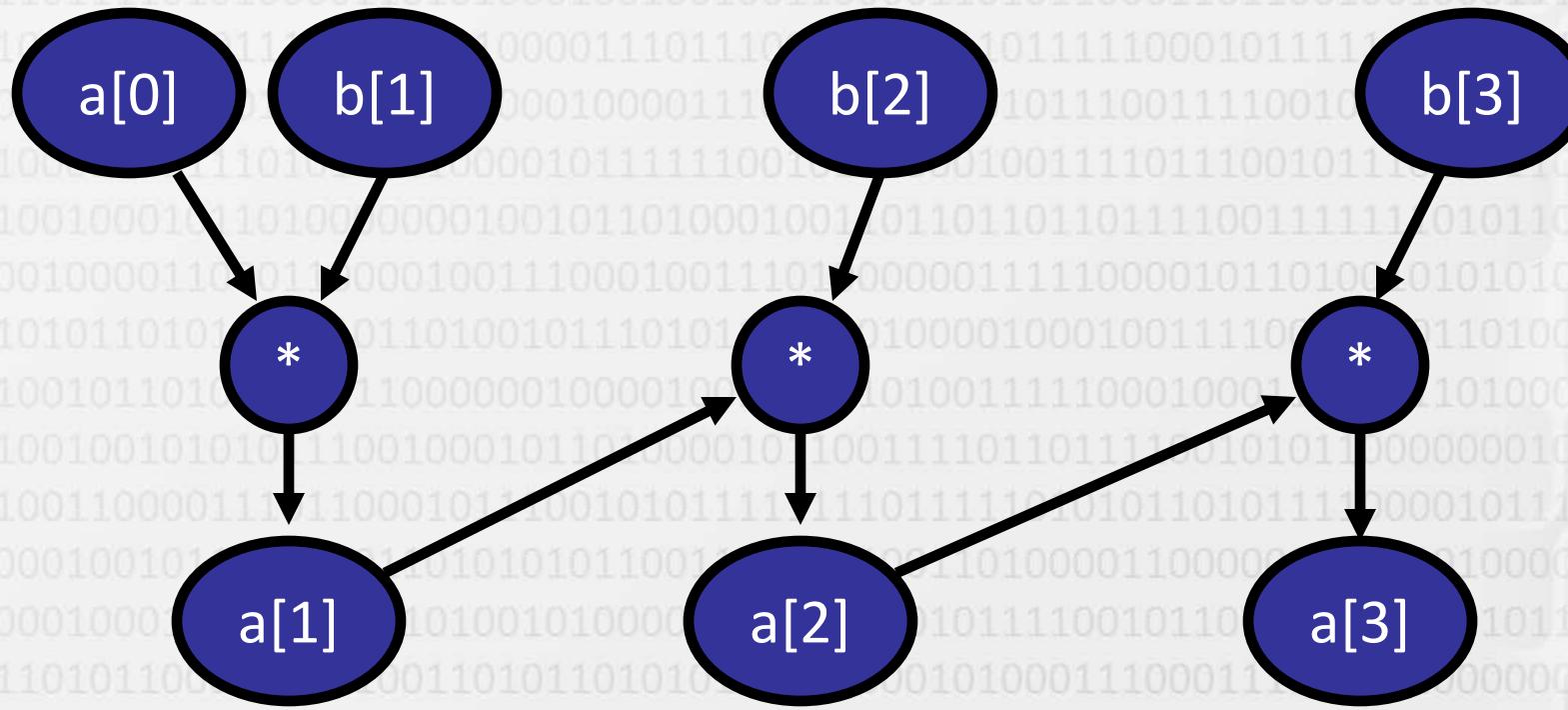
```
for (i = 0; i < 3; i++)  
    a[i] = b[i] / 4;
```

Lehetséges
adatdekompozíció



Függőségi gráf 2. példa

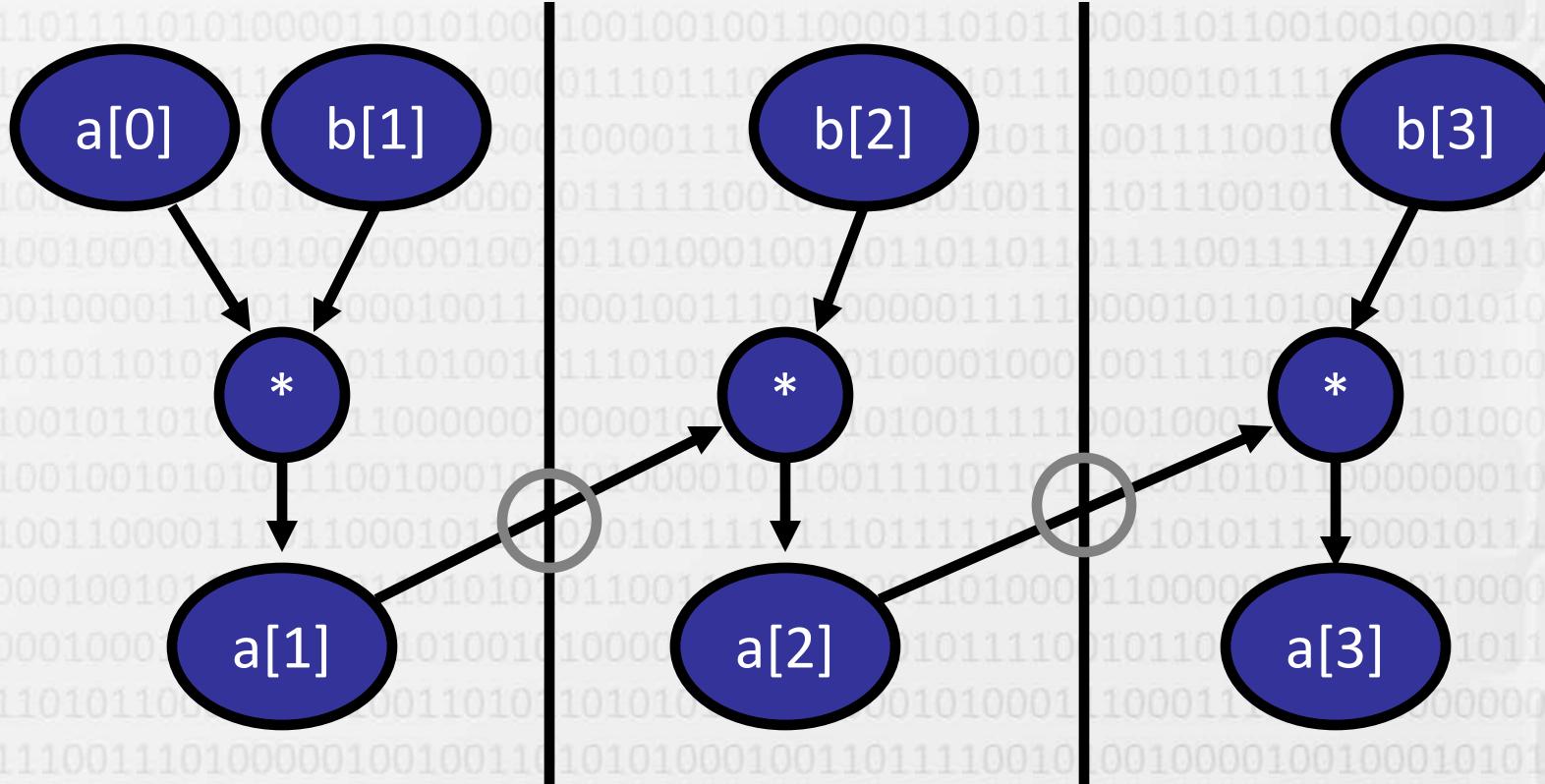
```
for (i = 1; i < 4; i++)  
    a[i] = a[i-1] * b[i];
```



Függőségi gráf 2. példa

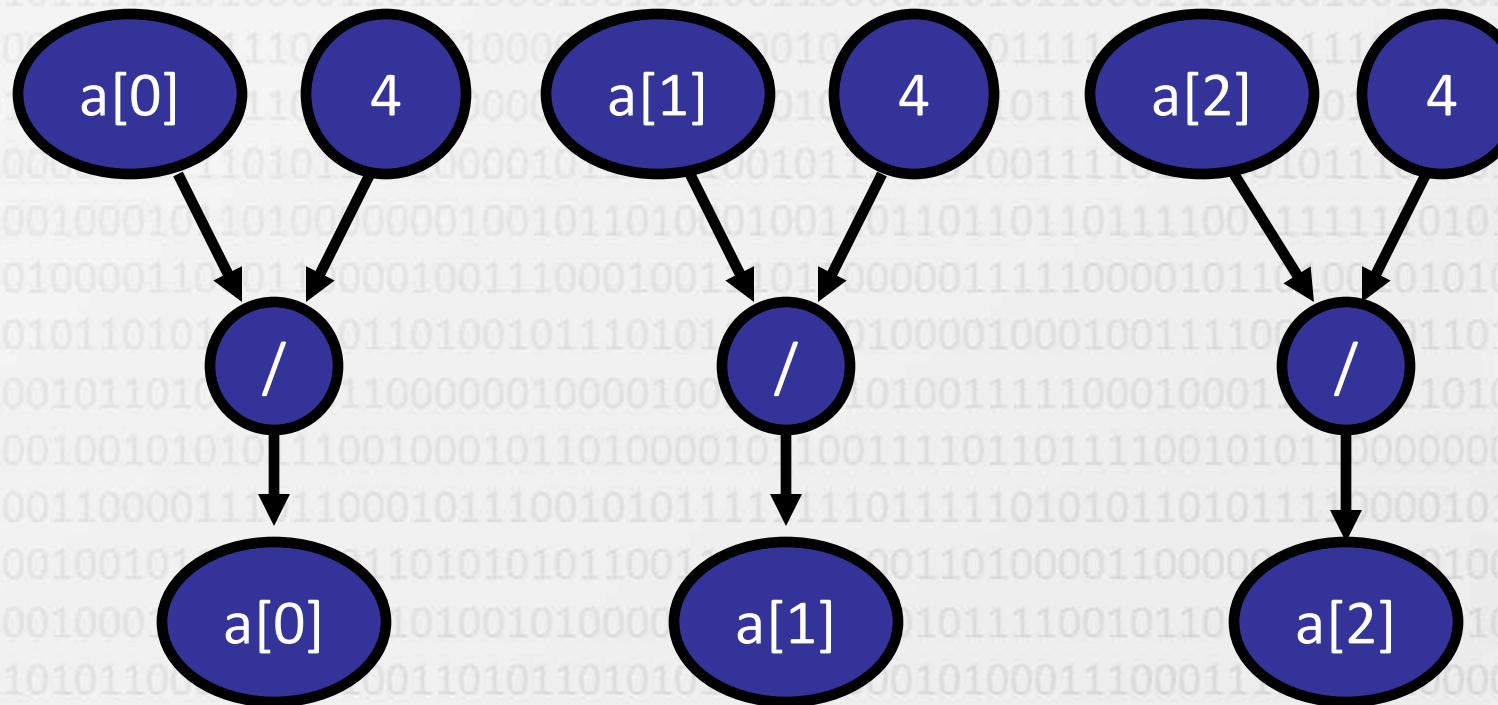
```
for (i = 1; i < 4; i++)  
    a[i] = a[i-1] * b[i];
```

Nincs adatdekompozíció



Függőségi gráf 3. példa

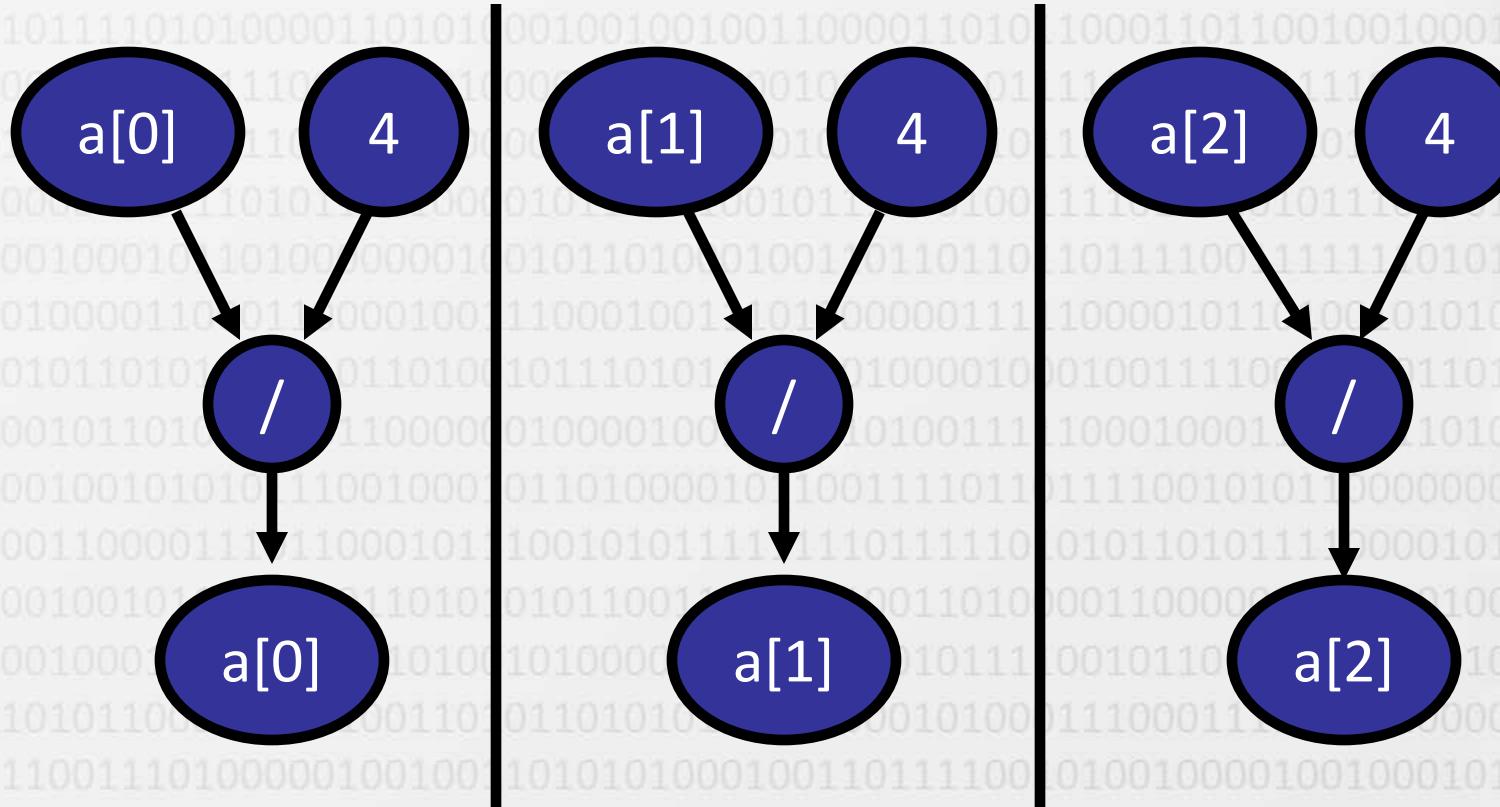
```
for (i = 0; i < 3; i++)  
    a[i] = a[i] / 4;
```



Függőségi gráf 3. példa

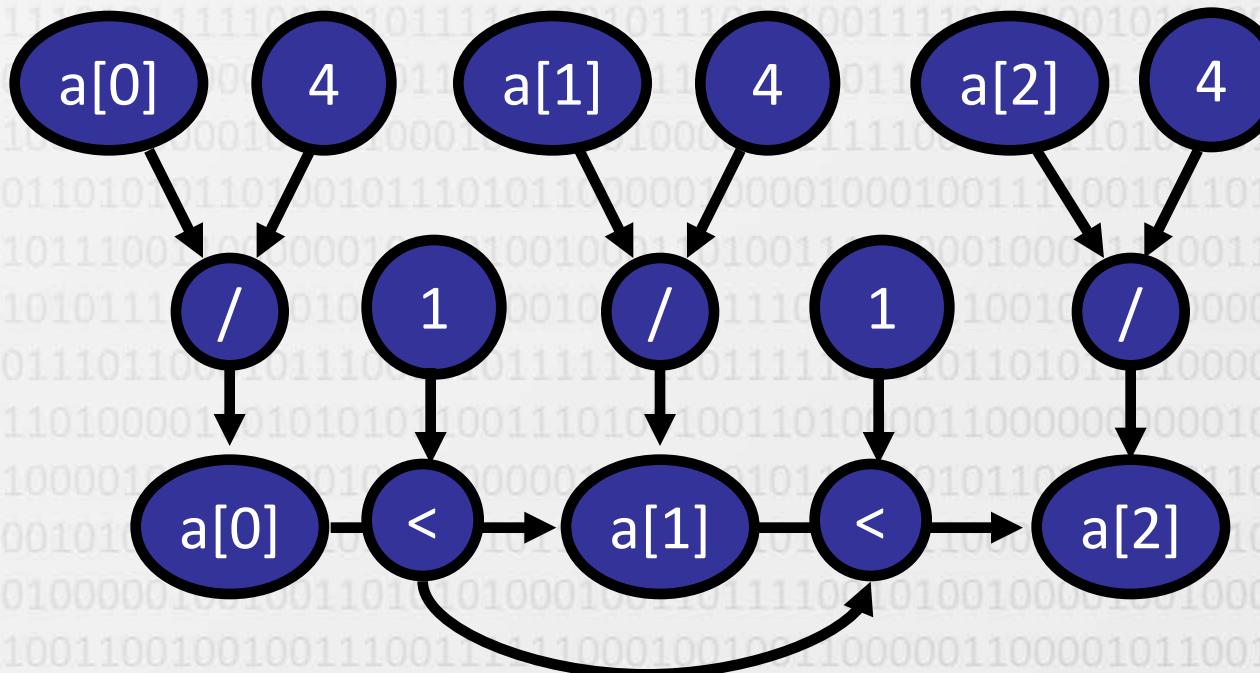
```
for (i = 0; i < 3; i++)  
    a[i] = a[i] / 4;
```

Adatdekompozíció



Függőségi gráf 4. példa

```
for (i = 0; i < 3; i++) {  
    a[i] = a[i] / 4;  
    if (a[i] < 1) break;  
}
```





2. lépés: kommunikáció

- A részfeladatok meghatározását követően meg kell vizsgálni a szükséges kommunikáció típusát és módját.
 - A kommunikációt a párhuzamos algoritmusok „rezsiköltségének” szokás tartani, hiszen egy soros algoritmusnál erre nem lenne szükség. Lényeges célkitűzés tehát ennek a költségnek a minimalizálása.
 - A kommunikációs műveletek sok szempontból csoportosíthatók, leggyakrabban **lokális** és **globális** kommunikációra szokás osztani őket:
 - lokális kommunikáció esetén a feldolgozóegységek csak kisszámú, „szomszédos” feldolgozóegységekkel továbbítanak adatokat egymás között (pl. képfeldolgozás);
 - globális kommunikáció esetén a számításhoz a feldolgozóegységek nagy többségének (vagy mindegyikének) részeredményt kell szolgáltatnia (pl. felösszegzés N részfeladattal).
 - Néhány más felosztási lehetőség:
 - strukturált és nem strukturált kommunikáció;
 - statikus és dinamikus kommunikáció;
 - szinkron és aszinkron kommunikáció.



A kommunikáció hatékonyságának vizsgálata

- **Nincs általánosan elfogadott, minden tökéletes vizsgálati módszer.**
- **Az alábbi ellenőrző lépések iránymutatóként szolgálnak, és nem szigorú szabályként értelmezendők:**
 - a részfeladatok egyidejű végrehajtása valóban lehetséges lesz a választott kommunikációs módszerrel?
 - kiegyensúlyozott-e a kommunikáció mennyisége a részfeladatok között?

Amennyiben nem, az algoritmus várhatóan nem, vagy csak kevéssé lesz skálázható.

- minden feldolgozóegység a lehető legkevesebb másikkal kommunikál-e?
- a feldolgozóegységek képesek lesznek egymással egy időben kommunikálni?

Amennyiben nem, az algoritmus rossz hatékonyságú lesz, és nem, vagy csak kevéssé lesz skálázható.

3. lépés: összevonás

- A tervezés első két lépése során igyekeztünk a lehető legnagyobb mértékben kiaknázni az elérhető párhuzamosságot.
 - Ezen a ponton az algoritmus nagy valószínűséggel igen rossz hatékonysággal hajtódna végre bármely párhuzamos számítógép-architektúrán. Ha például nagyságrendekkel több feladatot határoztunk meg, mint ahány feldolgozóegység rendelkezésre áll, a kommunikáció többletköltsége igen számottevő lesz.
- A tervezés következő lépéseinél célja a feladatok részleges összevonása nagyobb részfeladatokká.
 - A nagyobb részfeladatok kialakítása során a teljesítmény növelése és/vagy a program létrehozásának egyszerűsítése a lényegi cél.



Az összevonás célkitűzései

- **Kommunikációs veszteségek minimalizálása**

- Az egymással gyakran kommunikáló részfeladatok összevonása megszünteti a kommunikációt, hiszen a szükséges adatok az összevonás után a keletkező részfeladat rendelkezésére állnak („lokalitás növelése”).
- Az üzeneteket küldő és fogadó részfeladatok csoportjainak összevonásával csökkenhető az elküldendő üzenetek száma.
 - Kevés, de nagyobb méretű üzenet elküldése rövidebb időt igényel, mint sok, de kisebb méretű üzenet elküldése, mivel minden üzenet a hosszától független fix késleltetéssel is jár.

- **A párhuzamos megoldás skálázhatóságának megtartása**

- Kerüljük a túl intenzív összevonásokat, hogy az algoritmus nagyobb processzorszámú rendszeren is párhuzamos maradjon.
 - Példa: egy $16*256*512$ méretű 3D mátrixot 8 processzoros gépen dolgozunk fel. Ha a 2. és a 3. dimenziót összevonjuk, minden processzor egy $2*256*512$ méretű részt dolgoz fel, s az algoritmus csak 16 processzorig marad hatékony, efölött jelentős átdolgozást igényel.

- **A szoftverfejlesztési költségek csökkentése**

- Egy már meglévő soros algoritmus párhuzamosítása általában kisebb időigényű és költségű, mint egy teljesen új, párhuzamos algoritmus kialakítása.



Az összevonás hatékonyságának vizsgálata

- **Az alábbi (részben mennyiségi jellegű, a konkrét megvalósítás szempontjából fontos) kérdések feltevésére lehet szükség:**
 - Javította-e az összevonás a párhuzamos algoritmus lokalitását?
 - Az összevont részfeladatok végrehajtási ideje kisebb, mint az általuk felváltott részfeladatok összesített végrehajtási ideje?
 - Az összevonáshoz szükséges adatmásolás mértéke elég kicsi ahhoz, hogy az algoritmus nagy processzorszám mellett is skálázódni tudjon?
 - Az összevont részfeladatok hasonló számítási és kommunikációs igényűek?
 - A részfeladatok száma a megoldandó feladat méretében együtt növekszik?
 - A részfeladatok száma a lehető legkisebb, de legalább annyi-e, mint a párhuzamos célszámítógép processzorainak száma?
 - Az összevonás, illetve az esetlegesen meglévő soros algoritmus párhuzamosításának költsége közötti kompromisszum ésszerű-e?

4. lépés: leképezés

- A leképezés célja a processzorok kihasználtságának maximalizálása és a processzorok közötti kommunikáció minimalizálása (ezek sajnos egymásnak ellentmondó célok).
 - A processzorok kihasználtsága akkor maximális, amikor a számítások elosztása tökéletesen egyenletes. Ebből következik, hogy ellenkező esetben egy vagy több processzor időnként nem fog hasznos munkát végezni.
 - Ha a részfeladatok elosztása tökéletesen egyenletes processzorterheléseket eredményez, akkor a processzorok kihasználtsága maximális lesz, de óriási mennyiséggű kommunikációra lesz szükség, ami rontja a „hasznos munka” és a „rezsiköltség” arányát.
 - Processzorok közötti kommunikációra akkor van szükség, amikor két, csatornával összekapcsolt részfeladat különböző processzorokra képeződött le. Ebből következik, hogy ha a két részfeladatot azonos processzorra képezzük le, a kommunikációs igény csökken.
 - Ha minden részfeladatot azonos processzor hajt végre, akkor nincs kommunikáció, de a processzorok kihasználtsága gyenge.
- **A lényeg a lehető legjobb egyensúlyt képviselő leképezés megtalálása az egymásnak ellentmondó célkitűzések fényében.**



A leképezés jellemzői

- A dekompozíció és az összevonás után a részfeladatok száma általában meghaladja az elérhető processzorok számát*.
- A megfelelő leképezés kritikus jelentőségű a párhuzamos teljesítmény és a szoftver elkészíthetősége szempontjából.
 - A minimális futási idő eléréséhez a következő stratégiát érdemes követni:
 - a független részfeladatokat különböző OS-folyamatokra képezzük le;
 - a kritikus úton lévő részfeladatokat a lehető leggyorsabban OS-folyamatokhoz rendeljük;
 - a processzek közötti kommunikáció minimalizálása úgy, hogy a sokat kommunikáló részfeladatokat ugyanahhoz az OS-folyamathoz rendeljük.

• A leképezési feladat NP-teljes probléma.

Ez azt jelenti, hogy a problémát általános esetben csak heurisztikus megoldással tudjuk ésszerű időn belül megoldani.

* Megjegyzés: a gyakorlatban nem konkrét processzorokra, hanem folyamatokra történő leképezésről beszélünk, mert az operációs rendszerek programozási felületei tipikusan ezt biztosítják. Az összevont részfeladatokat egy folyamatként kezeljük, és ezeket az operációs rendszer fogja a fizikai processzorokra terhelni.



A leképezés hatékonyságának vizsgálata

- **A leképezés során érdemes több tervezési alternatívát is figyelembe venni:**

- A megoldás processzoronként egy vagy több részfeladatot irányoz elő?
- A részfeladatok processzorokhoz rendelésénél statikus és dinamikus megoldásokat is figyelembe vettünk?
- Ha dinamikus leképezési megoldást választottunk, akkor a vezérlést (a részfeladatokat elosztó) végző szoftverkomponens teljesítmény szempontjából lehet-e szűk keresztmetszet?
- Ha valószínűségi alapon vagy ciklizálási módszerrel oldottuk meg a leképezést, van-e elegendően sok részfeladat ahhoz, hogy a processzorok terhelése egyenletes legyen (ehhez általában a processzorok számához viszonyítva tízszer annyi részfeladatra van szükség)?



A párhuzamos teljesítménynövelés határai

• Amdahl törvénye

Ez a törvény Gene Amdahl megfigyelését írja le azzal kapcsolatban, hogy egy rendszer sebessége legfeljebb milyen mértékben növelhető, ha csupán részben (azaz csak egyes elemeit) gyorsítják fel.

A párhuzamos programozásban ezt a törvényt a több feldolgozóegységgel végzett feldolgozás várható gyorsulásának előrejelzésére használják.

Amdahl törvénye (részszámítás felgyorsításának eredő gyorsító hatása):

$$\frac{1}{(1-P) + \frac{P}{S}}$$

ahol
P: a számítás felgyorsított szakasza
S: P gyorsulásának mértéke

Speciális eset: számítások párhuzamosítása

$$\frac{1}{F + \frac{1-F}{N}}$$

ahol
F: a számítás soros szakasza
N: feldolgozóegységek száma



A párhuzamos teljesítménynövelés határai

• Gustafson törvénye

Ez a John L. Gustafson nevéhez fűződő törvény azt mondja ki, hogy a nagyméretű, könnyen párhuzamosítható (ismétlődő elemekből álló) adathalmazok feldolgozása hatékonyan párhuzamosítható.

Gustafson törvénye (számítás párhuzamosításának eredő gyorsító hatása):

ahol

P: feldolgozóegységek száma

S: eredő gyorsulás mértéke

α : a probléma nem párhuzamosítható része

$$S(P) = P - \alpha \cdot (P - 1)$$

Amdahl rögzített méretű adathalmazokra, illetve problémákra vonatkozó törvényével szemben Gustafson megállapítja, hogy a feladatok mérete az elérhető számítási kapacitáshoz igazítható, és azonos idő alatt így sokkal nagyobb feladatok oldhatók meg, mint akkor, ha a feladatok mérete a számítási kapacitástól független.



Programhelyességi szempontok és fogalmak

- A párhuzamos algoritmusok tervezése során folyamatosan ügyelni kell a programok helyességének biztosítására vagy megőrzésére.
 - Soros algoritmus párhuzamosításakor nem engedhető meg, hogy a teljesítménynövelés hibás működést eredményezzen.
- A párhuzamos futtatás sajátja, hogy a részfeladatok folyamatosan versenyeznek az erőforrásokért.
 - A versengés mértéke függ a szükséges kommunikáció mennyiségétől is.
 - Kommunikáció nélküli (masszív párhuzamos) feladatoknál a versengés el is maradhat.
- Ez a **versenyhelyzet** bármely erőforrást érintheti
 - Processzor
 - Verseny a végrehajtási lehetőségért (általában az operációs rendszer szabályozza).
 - Memória
 - Egyszerre többen kívánnak a közös memória azonos eleméhez hozzáférni (ezek közül legalább egy részfeladat írási céllal).
 - Portok



Kölcsönös kizárási mechanizmusok

- A versenyhelyzet kezelésének egy lehetősége a **kölcsönös kizárást**.
 - Kritikus szakasz: olyan kód részlet, amely megosztva használt (közös) erőforrásokhoz fér hozzá olvasási és/vagy írási céllal.
 - Kölcsönös kizárás: biztosítja, hogy a kritikus szakaszba egyszerre csak egy részfeladat léphessen be.
 - Szemantikailag helyes programozási struktúrát biztosít a versenyhelyzetek elkerüléséhez.
 - Lényegében bizonyos pontokon sorossá teszi a végrehajtást, ezzel viszont csökkenti a párhuzamos teljesítménynövekedés mértékét.
- A kölcsönös kizárást biztosítására a gyakorlatban szinkronizációs objektumok szolgálnak.
 - Leggyakoribb típusai: zár (lock), szemafor (semaphore), kritikus szakasz (critical section), atomi végrehajtás (atomic/interlocked execution).
 - Működésük: egy részfeladat „magánál tartja” a szinkronizációs objektumot, a többi várakozik, majd amikor elkészült a feladatával, a fenti részfeladat „elengedi” az objektumot, amelyet ezt követően egy másik, addig várakozó részfeladat kaphat meg.



Randevú

- **A versenyhelyzet kezelésének másik lehetősége a *randevú*.**
 - A közös erőforráshoz hozzáférő részfeladatok a végrehajtás adott pontján „bevárják” egymást, azaz amikor mindenkiük „megérkezett” a „randevú helyszínére”, egyszerre továbblépnek.
 - Az éppen várakozó részfeladatok nem végeznek hasznos munkát.
- **A randevú megvalósítására a gyakorlatban speciális objektumok szolgálnak.**
 - Leggyakoribb típusaiak: feltételváltozó (condition variable), esemény (event).
 - Működésük: amint egy részfeladat végrehajtása során eléri az adott objektumot, automatikusan felfüggesztődik, amíg az összes érintett részfeladat szintén el nem ért erre a pontra, majd mindenkorán ismét működésbe lépnek.



Holtpont

- **Akkor beszélünk holpontról, ha a párhuzamos program feldolgozást végző részfeladatai a végrehajtás egy bizonyos pontján nem tudnak továbblépni.**

Holtpont akkor keletkezhet, ha a feldolgozás során egynél több részfeladat egynél több erőforráshoz kíván hozzáérni, miközben maga is lefoglalva tart bizonyos erőforrásokat. Ebben az esetben „egymásra várnak” a részfeladatok, miközben nem végeznek semmilyen hasznos munkát, és a várakozó állapotot semmilyen belső módszerrel nem lehet megszüntetni.

Példa holpontra: forgalmi dugó útkereszteződésben mind a négy irányból, amikor egyetlen érintett autó sem tud sem továbbmenni, sem visszafordulni.



Élőpont

- **Akkor beszélünk élőpontról, ha a párhuzamos program egyes részfeladatai ugyanazokat a (rövid) végrehajtási lépéssorozatokat ismétlik, és ezeken nem képesek túljutni.**

Előpont akkor keletkezhet, amikor az egymásba ágyazott erőforrás-lefoglalások miatt a feldolgozás ugyan nem áll le, de egy rövid szakaszban a végtelenséggel ismétlődik minden érintett részfeladatnál.

A holtpontnál sokkal ritkábban fordul elő, kezelése azonban még nehezebb.

Példák élőpontra: malomjátékban a „csiki-csuki” lépéssorozat, sakkjátékban pedig az „örökös sakk”.

Párhuzamosság a modern operációs rendszerekben

Bevezetés

Folyamatok nyilvántartása

Folyamatok életciklusa

Folyamatok állapotátmenetei

Folyamatmodellek és megvalósításuk elvei

Egyszerű folyamatalapú modell

Folyamatszál alapú modell

Ütemezés típusai és algoritmusai

Folyamatkezelési példa: Microsoft Windows

Bevezetés

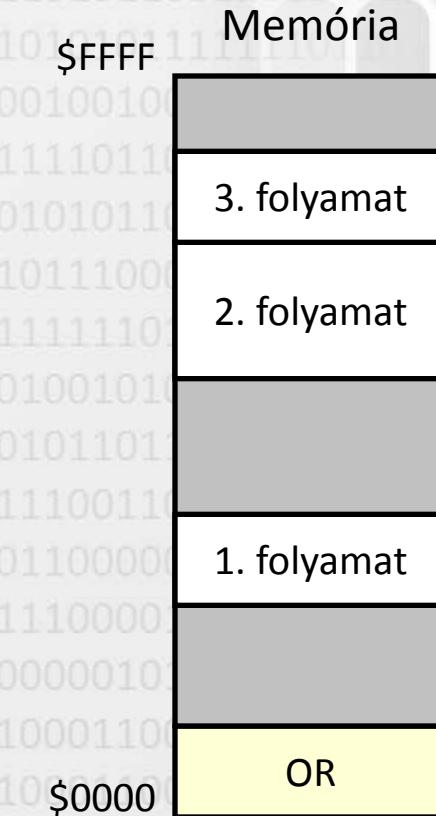
- Az operációs rendszerek fontos feladata lehetővé tenni a számítógép teljesítményének lehető legjobb kihasználását.
- Egy számítógép – egy vagy több futó program

Ha az operációs rendszer egy időben több programot futtat, *multiprogramozásról* beszélünk, melynek célja az erőforrások jobb kihasználása és a kényelem.

- A Neumann-architektúrára épülő számítógépek a programokat sorosan hajtják végre.

A gépi kódú programok tényleges futtatása utasításról utasításra történik. Ahhoz, hogy egy számítógépen egyszerre több program is futtatható legyen, olyan megoldásra van szükség, amely biztosítja:

- a végrehajtás alatt álló programok egymástól való elszigetelését, valamint
- a végrehajtás alatt álló programok (látszólag) egyidejű futását.



Bevezetés

- **Az operációs rendszerek programokat hajtanak végre.**
 - Kötegelt rendszerek: feladatok („job”).
 - Interaktív rendszerek: felhasználói programok („task”).
- **A programok elszigetelése és párhuzamosítása a folyamatok koncepciójának segítségével megoldható.**
 - Folyamat (definíció): a folyamat egy végrehajtás alatt álló, a számítógép virtuális memoriájába betöltött, környezetével (azaz az operációs rendszerrel és/vagy a többi folyamattal) kölcsönhatásban lévő programpéldány.
 - Az elszigetelés érdekében minden folyamat saját memóriaterülettel rendelkezik, amelyet más folyamatok nem érhetnek el, így hiba esetén csak a hibázó folyamat sérül, a rendszer többi eleme működőképes marad (viszont a folyamatok közötti közvetlen kommunikációra sincs egyszerű lehetőség).
 - A párhuzamosítás tipikus megoldása az *időosztás*, amikor minden folyamat kap egy-egy ún. időszeletet, melynek letelejt követően egy másik folyamat kapja meg a vezérlést. Ez a megoldás gyakorlatilag függetleníti egymástól a processzorok és a rajtuk egy időben futtatható programok számát.

Folyamatok nyilvántartása

- **Folyamatleíró blokk (Process Control Block)**

- Azonosító (ID)
- Állapotjelző
- CPU regiszterek tartalma (programszámláló is)
- Memória- és erőforrásadatok
- Prioritási és elszámolási információk
- Egyéb mutatók

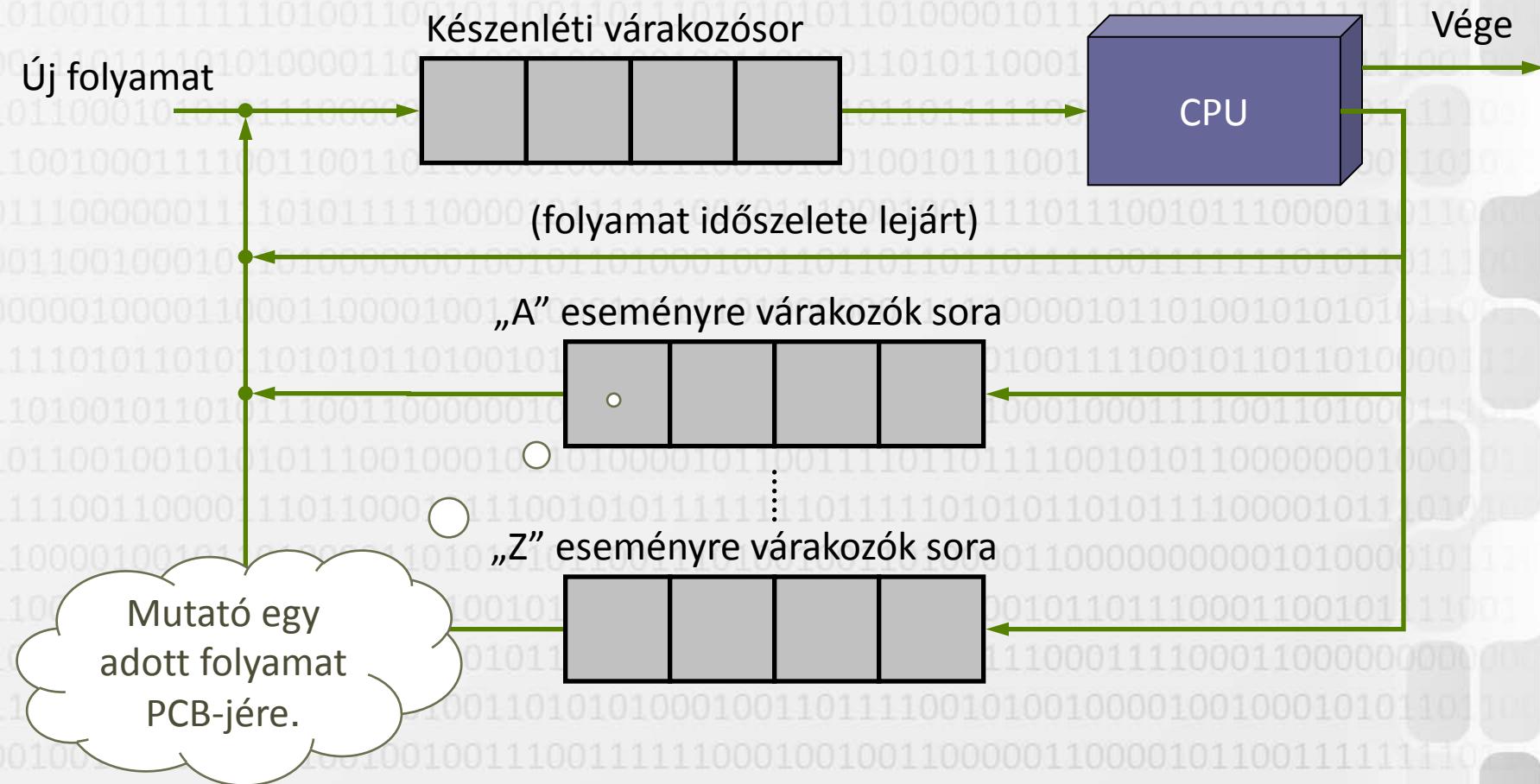
- **Folyamattáblázat (Process Table)**

- Logikailag tartalmaz minden PCB-t.
 - Táblázat vagy lista belső mutatókkal.

Folyamatok nyilvántartása

- **Várakozósorok**

Minden állapothoz egy vagy több várakozósor tartozhat.



Folyamatok életciklusa

• Létrehozás

- Betöltés háttértárról vagy közvetlenül a központi memóriából.
- Címtér hozzárendelése:
 - a „gyermek” (új) folyamat új programot tartalmaz
 - a „gyermek” lemásolja a „szülő” (létrehozó) folyamat címterét
- Végrehajtás megkezdése:
 - a „szülő” folyamattal párhuzamosan
 - a „szülő” megvárja a „gyermek” befejeződését

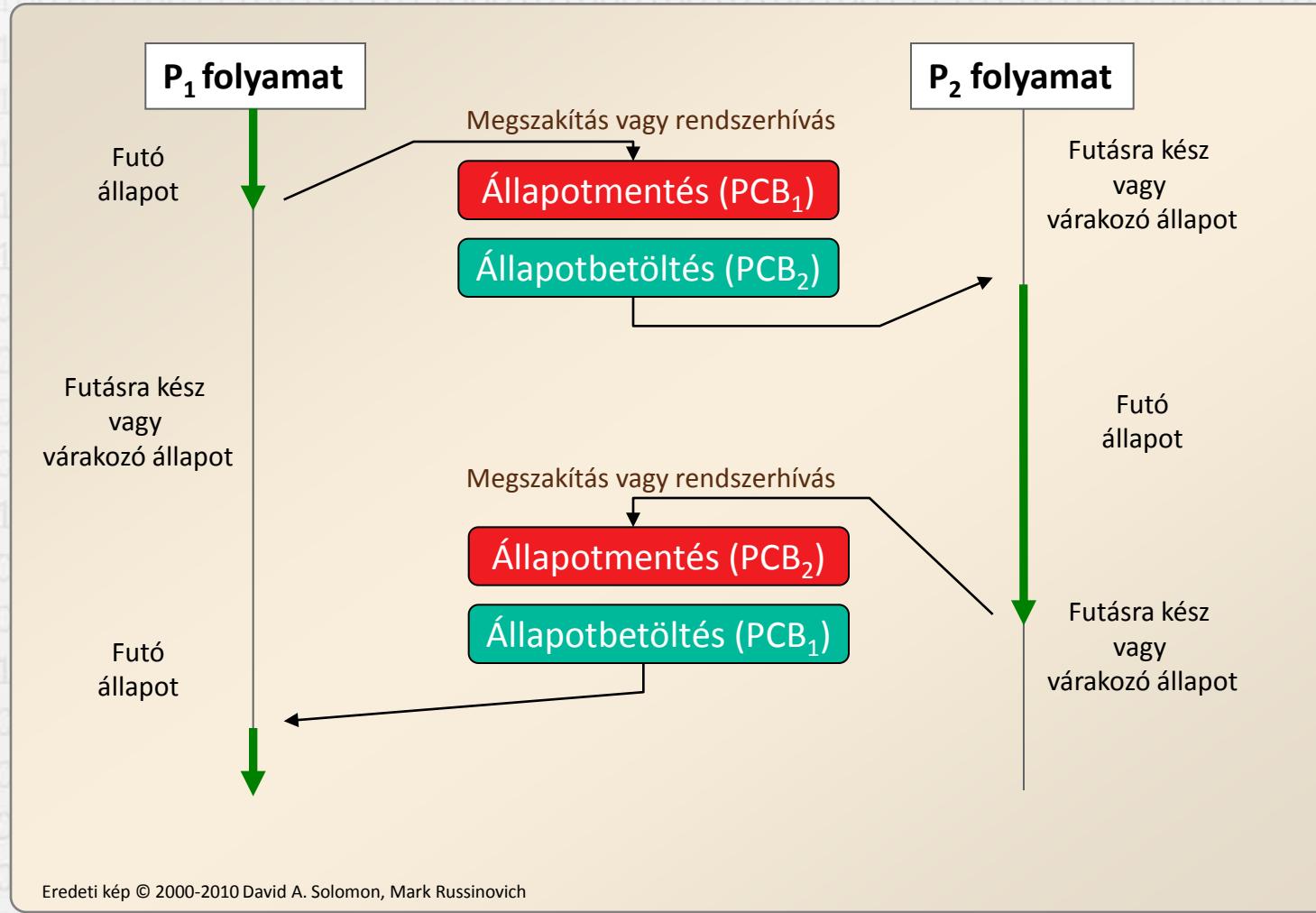
• Futás és kontextusváltások

- A futó folyamat elveszti, egy másik (várakozó) folyamat pedig megkapja a processzort.
 - Az operációs rendszer gondoskodik a „régi” és az „új” folyamat kontextusának cseréjéről.

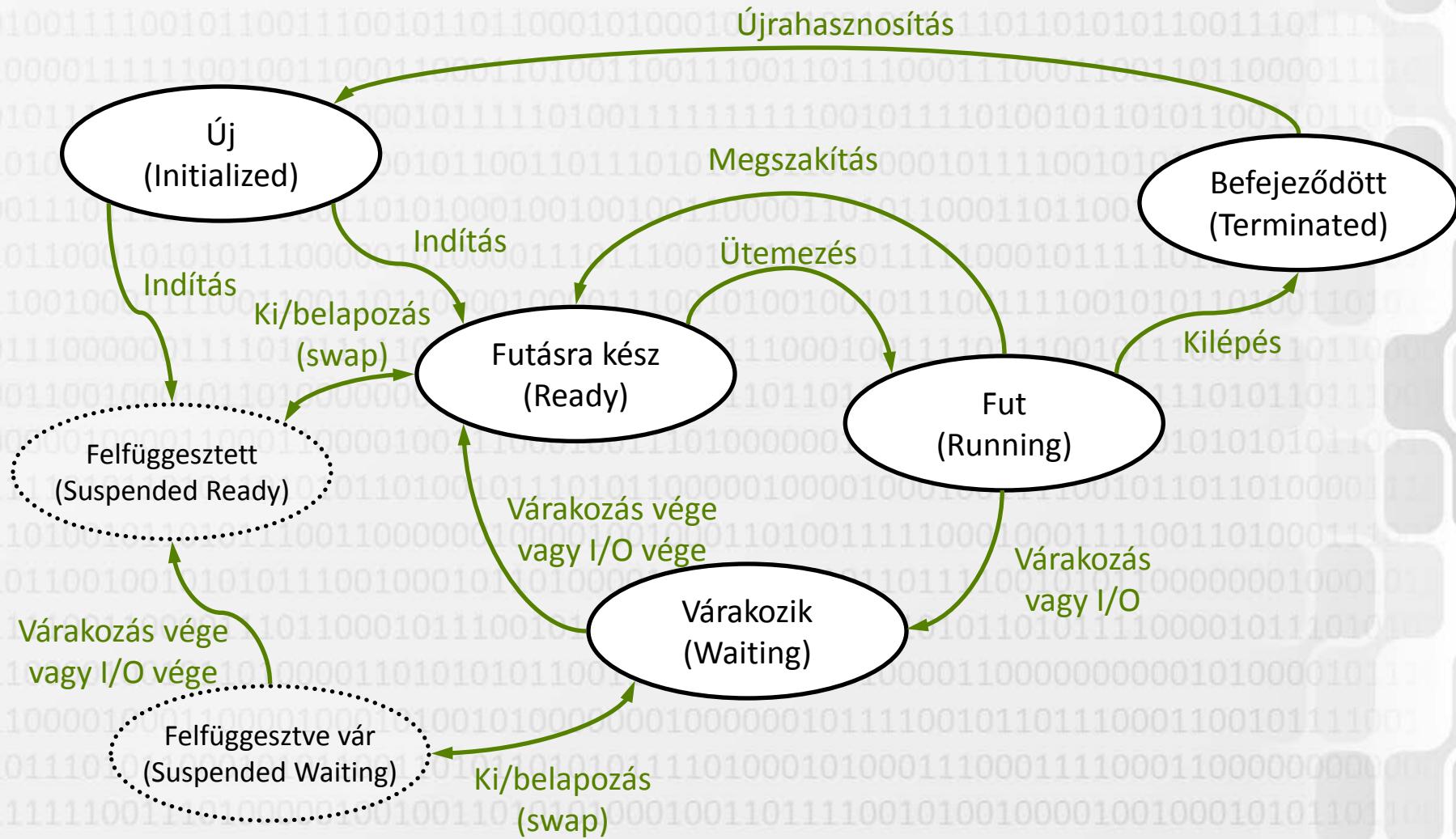
• Befejeződés

- Lehetséges okai:
 - a folyamat befejezte feladatát;
 - a folyamat „szülője” befejeződött és ilyen helyzetben futhatnak tovább a „gyermekek”;
 - a folyamat erőforráskorlátba ütközött vagy meg nem engedett műveletet hajtott végre.

Illusztráció: időosztás és kontextusváltások



Folyamatok állapotátmenetei



Folyamatmodellek (1)

- **Egyszerű folyamatállapú modell**

- Csak folyamatok léteznek benne.
- minden feladat megoldható ezzel az egyszerű modellel is.
 - Ütemezés, kvázi- és valódi párhuzamos futtatás.
 - Kommunikáció (folyamatok közötti kommunikáció [IPC], memória megosztása).
 - Aszinkron (párhuzamos) I/O műveletek.
- Teljes kontextusváltások.
- Kiváló belső védelem.
 - Külön címtér, külön kontextus.

Folyamatmodellek (2)

- **Folyamat-szál alapú modell**

- Szál: „pehelysúlyú folyamat”
 - Programfuttatás egysége.
 - Saját programszámláló, regiszterek, verem.
- A szálak a folyamatokon belül léteznek.
 - A szálak osztoznak a folyamat címterén.
 - Segítségükkel hatékonyabban valósítható meg kvázi- és valódi párhuzamosság is.
- Részleges kontextusváltások
- Előnyei:
 - jelentős teljesítménynövekedés (csak jól megírt programok esetén);
 - kényelmesen kezelhető.

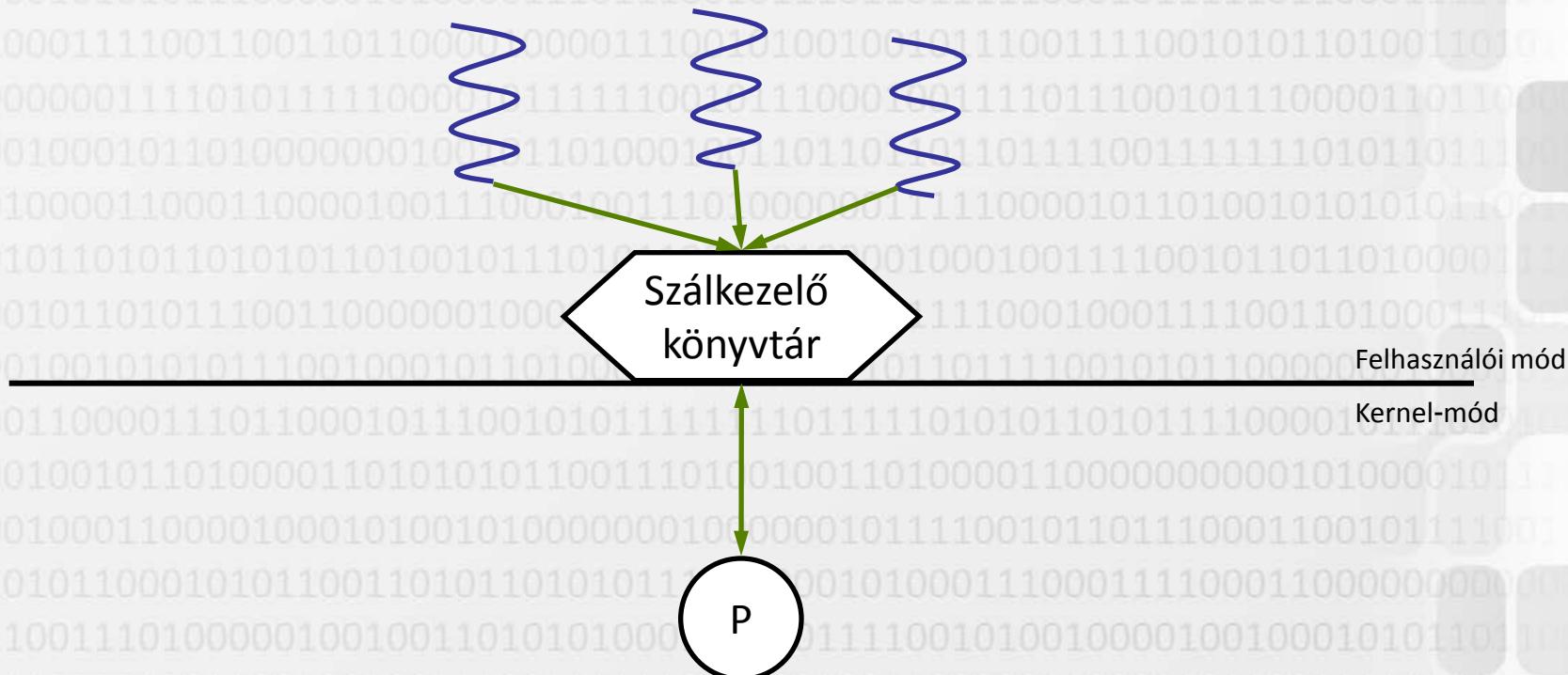
- **Léteznek további, összetettebb modellek is.**



Folyamat-szál alapú modell megvalósítása

- **Felhasználói módú szálkezelés**

- Megvalósítás függvénykönyvtárként.
- Az operációs rendszer nem foglalkozik a szálakkal.

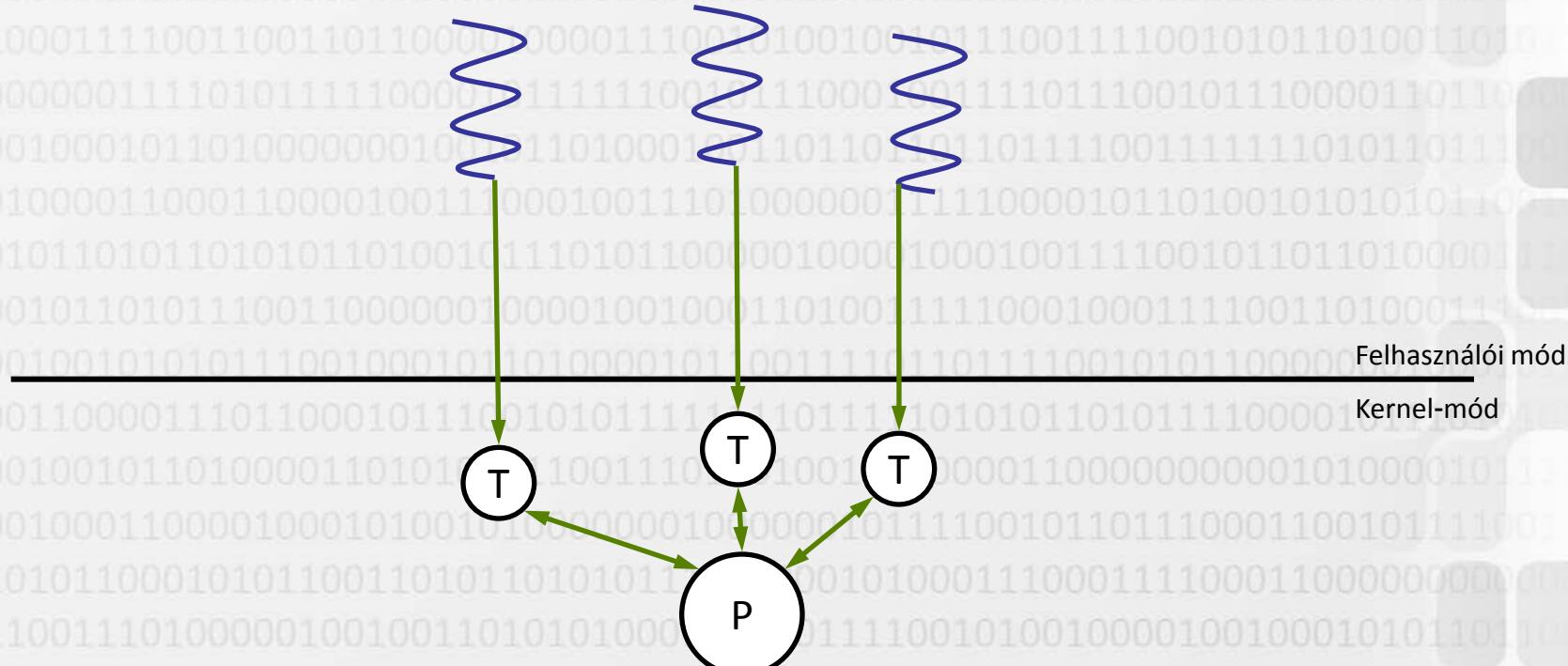




Folyamat-szál alapú modell megvalósítása

- **Kernel-módú szálkezelés**

- Hasonló programozói felület (API) folyamatokhoz és szálakhoz.
- Szálkezelés az operációs rendszer szintjén.



Ütemezés

• Rövid távú ütemezés

A rövid távú ütemezés lényege, hogy előre meghatározott időintervallumok (ún. „kvantum”) elteltével kontextusváltásra kerül sor, és ekkor a rendszer dönt arról, mely folyamat kaphat meg egy-egy processzort (végrehajtási erőforrást).

- Itt rendkívül gyors algoritmusra van szükség, ezért a gyakorlatban általában kombinált algoritmusokat használnak.

• Középtávú ütemezés

A középtávú ütemezés során a rendszer a rendelkezésre álló szabad memória mennyisége alapján dönt a folyamatválasztásról (memóriagazdálkodási szempont).

• Hosszú távú ütemezés

A hosszú távú ütemezés azt a szabályozási mechanizmust jelenti, amely segítségével az operációs rendszer a rendszerbe az aktuális állapot függvényében enged új folyamatokat működésbe lépni.

- Felhasználónkénti maximális folyamatszám.
- Rendszerszintű maximális folyamatszám.

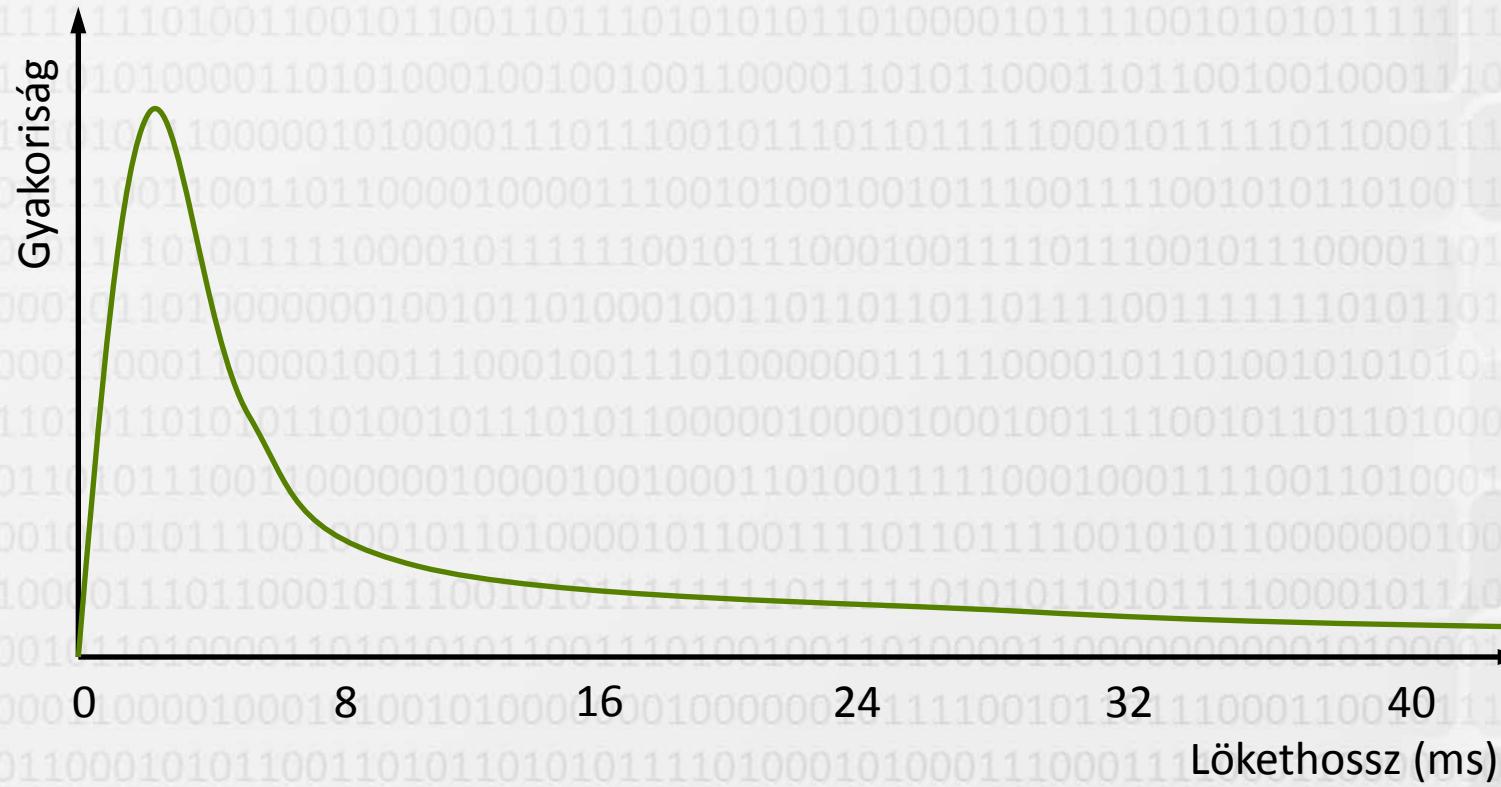
Rövid távú ütemezés

- **Célfüggvények:**

- legjobb processzorkihasználás
 - A rendelkezésre álló processzorteljesítmény lehető legjobb kiaknázása.
- legnagyobb teljesítmény
 - Időegységenként befejezett folyamatok számának maximalizálása.
- legkisebb teljes végrehajtási idő
 - Átlagos lassulás mértékének minimalizálása.
- legrövidebb várakozási idő
 - Átlagos várakozási idő minimalizálása a készenléti állapot sorában (soraiban).
- legrövidebb válaszidő
 - Eseményekre való reagálás átfutási idejének minimalizálása.

CPU löket

- Egy adott folyamat két, egymást követő várakozó állapota között eltelt idő.**



Rövid távú ütemezés algoritmusai

- „First Come, First Served” (FCFS) algoritmus
 - Más néven FIFO algoritmus.
 - Végrehajtás érkezési sorrendben.
- „Shortest Job First” (SJF) algoritmus
 - Legrövidebb CPU löketre készülő folyamat végrehajtása elsőként.
 - Típusok:
 - nem preemptív (nem szakítható meg az éppen futó folyamat);
 - preemptív (ha rövidebb CPU löketű folyamat érkezik, ó futhat).

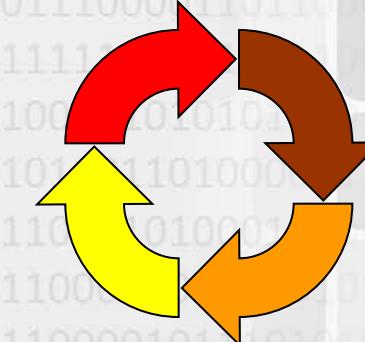
Rövid távú ütemezés algoritmusai

- **Prioritásos ütemező algoritmus**

- minden folyamat rendelkezik egy prioritással.
- mindig a legnagyobb prioritású folyamat fut.
- típusok:
 - nem preemptív (nem szakítható meg az éppen futó folyamat);
 - preemptív (ha magasabb prioritású folyamat érkezik, ő futhat).
- súlyos probléma: kiéheztetés
 - alacsony prioritású folyamatok nem jutnak szóhoz.
 - megoldás: pl. öregítés (régen futott folyamatok prioritásának fokozatos növelése).

Rövid távú ütemezés algoritmusai

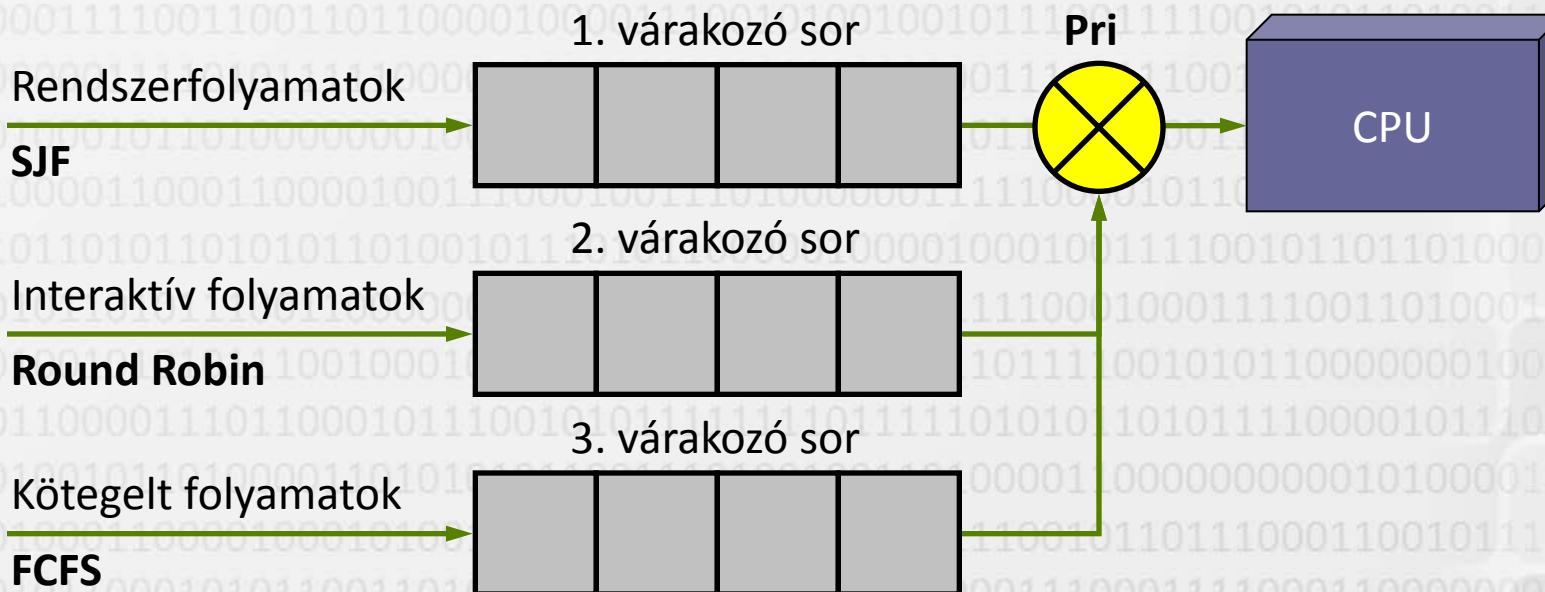
- „Round Robin” (körbenforgó) algoritmus
 - minden folyamat kap egy időszeletet (q).
 - Általában 10–100 ms közötti időtartam.
 - Az időszelet lejártával lekerül a processzorról és a készenléti sor végére kerül.
 - n folyamat esetén minden folyamat a processzoridő $1/n$ -ed részét kapja.
 - A maximális várakozási idő $(n-1)*q$.
 - Teljesítmény:
 - Nagy időszelet \approx FCFS.
 - Kis időszeletnél nagyobb a holtidő.



Többszintű rövid távú ütemezés

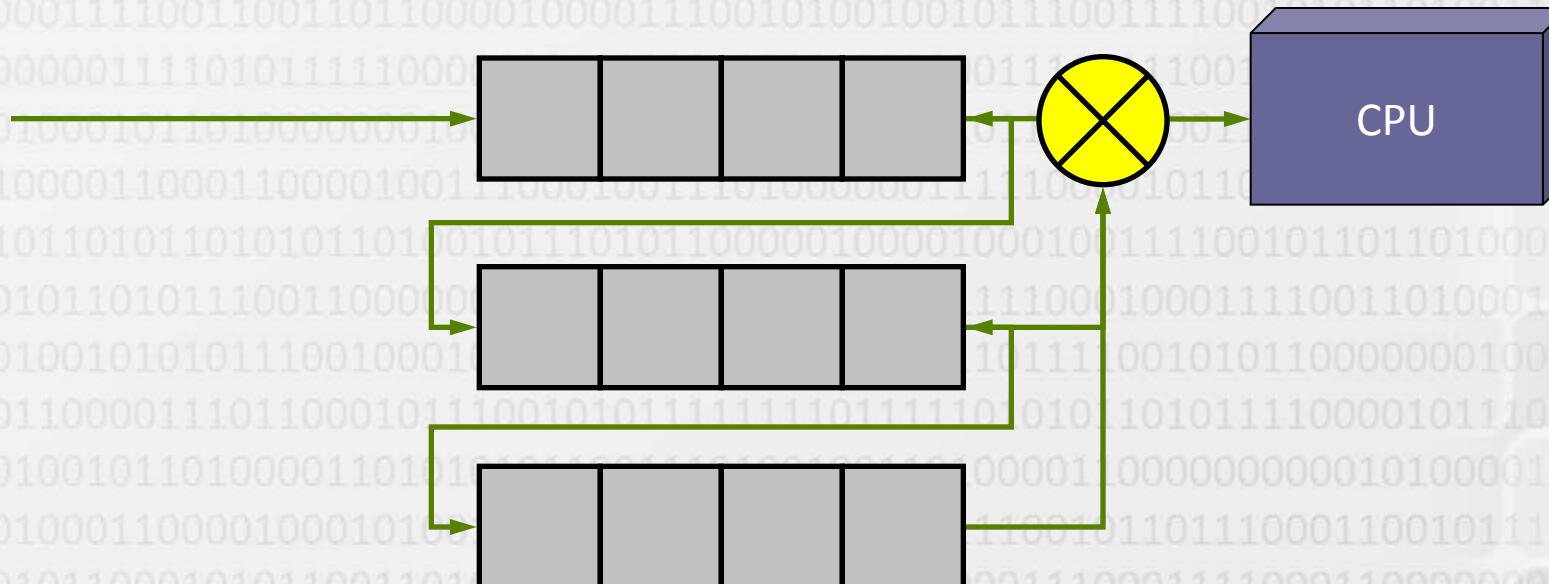
- „Multilevel Queue” algoritmus

- Több készenléti sor
 - Saját ütemezési algoritmus.
- Sorközi ütemezés (más algoritmussal)
 - Pl. fix prioritásos, soronkénti időszeletes.



Többszintű rövid távú ütemezés

- „Multilevel Feedback Queue” algoritmus
 - A folyamatok mozoghatnak a sorok között.
 - Számos független paraméter.
 - Sorok száma, sorok ütemezési politikája, folyamatok mozgatási feltételei, folyamatok kezdősora, sorok közötti ütemezés algoritmusa stb.



Folyamatkezelési példa: Microsoft Windows

- **Általános jellemzők (összefoglalás)**

- Négyszintű, bővített folyamat-szál alapú modell
 - A modell szintjei: feladat (job), folyamat (process), szál (thread), vékonyított szál (fiber).
- „Multilevel Feedback Queue” ütemezési algoritmus
 - 32 különböző prioritású sor.
 - Preemptív, prioritásos, körbenforgó ütemező.

Folyamatmodell elemei (Windows)

- **1. szint: feladat („job”)**

- A feladat egy adott szempontból közösnek tekintett folyamatok csoportja (folyamatkészlet vagy folyamathalmaz).
- A koncepció segítségével központilag szabályozhatók a feladathoz társított folyamatok paraméterei.
 - CPU idő, munkakészlet, biztonsági beállítások stb.
- Kötegelt feldolgozáshoz (interaktív bejelentkezés nélküli rendszerfolyamatok szabályozásához) ideális megoldás.

Folyamatmodell elemei (Windows)

- **2. szint: folyamat („process”)**

- A folyamat egy adott program dinamikus, a virtuális memóriában elhelyezkedő, „élő” példánya.
- Fő adminisztrációs egység (állapotinformáció-tár).
 - Erőforrások aktuális állapota (memória, fájlok, objektumkezelők, hálózati kapcsolatok stb.).
 - Processzoridő-alapú elszámolást is lehetővé tesz.
- Korábbi operációs rendszereknél a folyamat volt az ütemezés alapegysége, a Windows-nál azonban a szál („thread”) tölti be ezt a szerepet .

Folyamatmodell elemei (Windows)

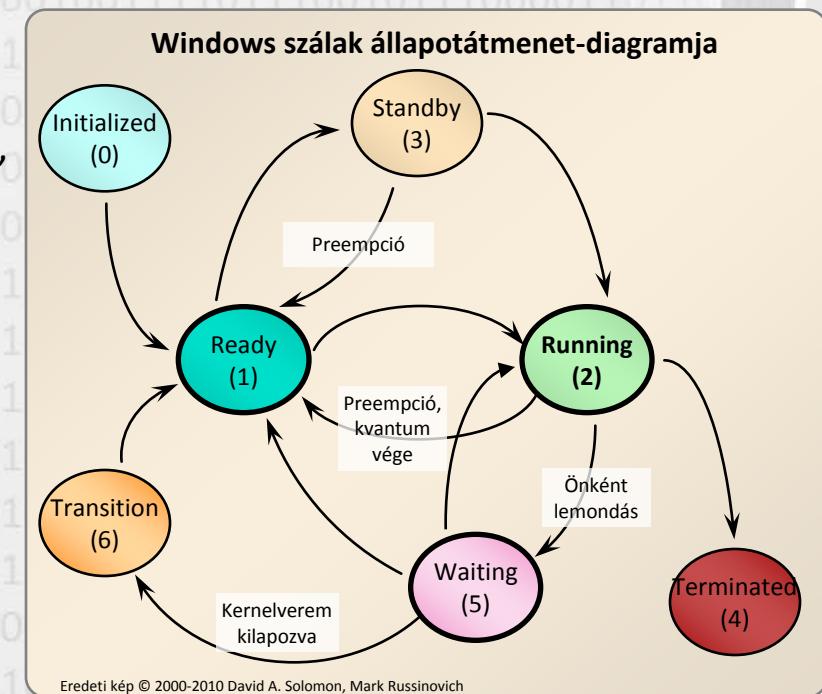
• 3. szint: szál („thread”)

- Az ütemezés alapegysége (a processzorokon ténylegesen csak szálak futnak).
- Folyamaton belüli állapotadminisztrációs egység.
 - Processzorok, ütemezés, I/O műveletek állapotai.
- Nincs saját címtere.
 - Az egy folyamathoz tartozó szálak egymással osztognak a folyamat címterén.
- A többszálú működés felveti a párhuzamos programozás alapproblémáit.

- Összehangolási pontok (randevú, kilépés).
- Adatkezelés (szálspecifikus adatok kezelése, szál szempontjából globális adatok kezelése, reentráns/nem reentráns kód stb.).

– Lehetséges állapotok:

- 0: Indítható („initialized”)
- 1: Futásra kész („ready”)
- 2: Futó („running”)
- 3: Készenléti („standby”)
- 4: Befejezett („terminated”)
- 5: Várakozó („waiting”)
- 6: Átmeneti („transition”)



Eredeti kép © 2000-2010 David A. Solomon, Mark Russinovich

Folyamatmodell elemei (Windows)

- **4. szint: vékonyított szál („fiber”)**

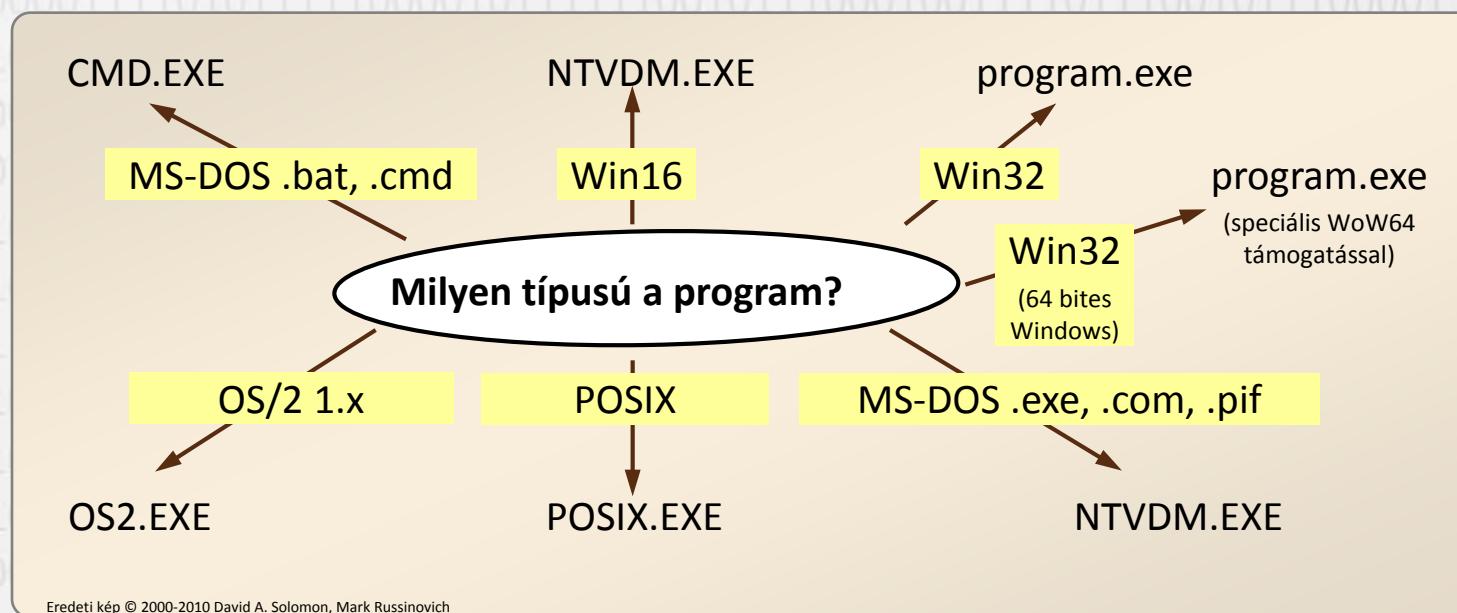
A vékonyított szálak egyetlen szempontból térnek el a „normál” szálaktól: ütemezésük nem automatikus, hanem a program maga végzi azt el a létrehozó szál kontextusán belül.

- Erre speciálisan kialakított API hívások (ConvertThreadToFiber(), SwitchToFiber()) szolgálnak.
 - Alkalmazásukra akkor lehet szükség, ha a beépített ütemezési algoritmusok működése nem felel meg az adott feladat kíváncsalainak.

Folyamatok és szálak kezelése (Windows)

• Folyamatok létrehozása

- Folyamatok háttértáron lévő programok betöltésekor vagy közvetlenül a memóriában elhelyezkedő programkód indításához jöhetnek létre.
 - A leggyakoribb eset az előbbi, melynek során a háttértáron lévő adatok elemzésével dönti el a rendszer, hogy milyen típusú folyamat jön létre.
- A folyamatokat létrehozó mechanizmus 3 részben fut le 3 különböző kontextusban (létrehozó folyamat, Win32 alrendszer, létrejövő folyamat).



Ütemezés: általános politika (Windows)

- **Eseményalapú ütemezés**
 - Nincs központi ütemező modul a kernelben.
- **Az ütemezés alapvetően szálszinten történik**
 - minden folyamathoz tartozik legalább egy szál.
- **Preemptív, prioritásos, körbenforgó algoritmus**
 - Időszak-alapú kötelező preempció.
 - Mindig a legmagasabb prioritású futásképes szál fut.
 - Külön várakozó sor minden prioritási szinthez.

Ütemezés: általános politika (Windows)

- **Preempció (futásmegszakítás) esetei:**

1. Lejár a szál időszelete.

Újraütemezéskor az addig futó szál időszelete 3-mal csökken.

2. Elindul egy nagyobb prioritású szál.

Az előző szál az időszelet megmaradt részét később visszakapja.

3. A szál eseményre kezd várni.

4. A szál önként feladja a futás jogát.

Ez az eset szigorú értelemben véve nem minősül preempciónak.

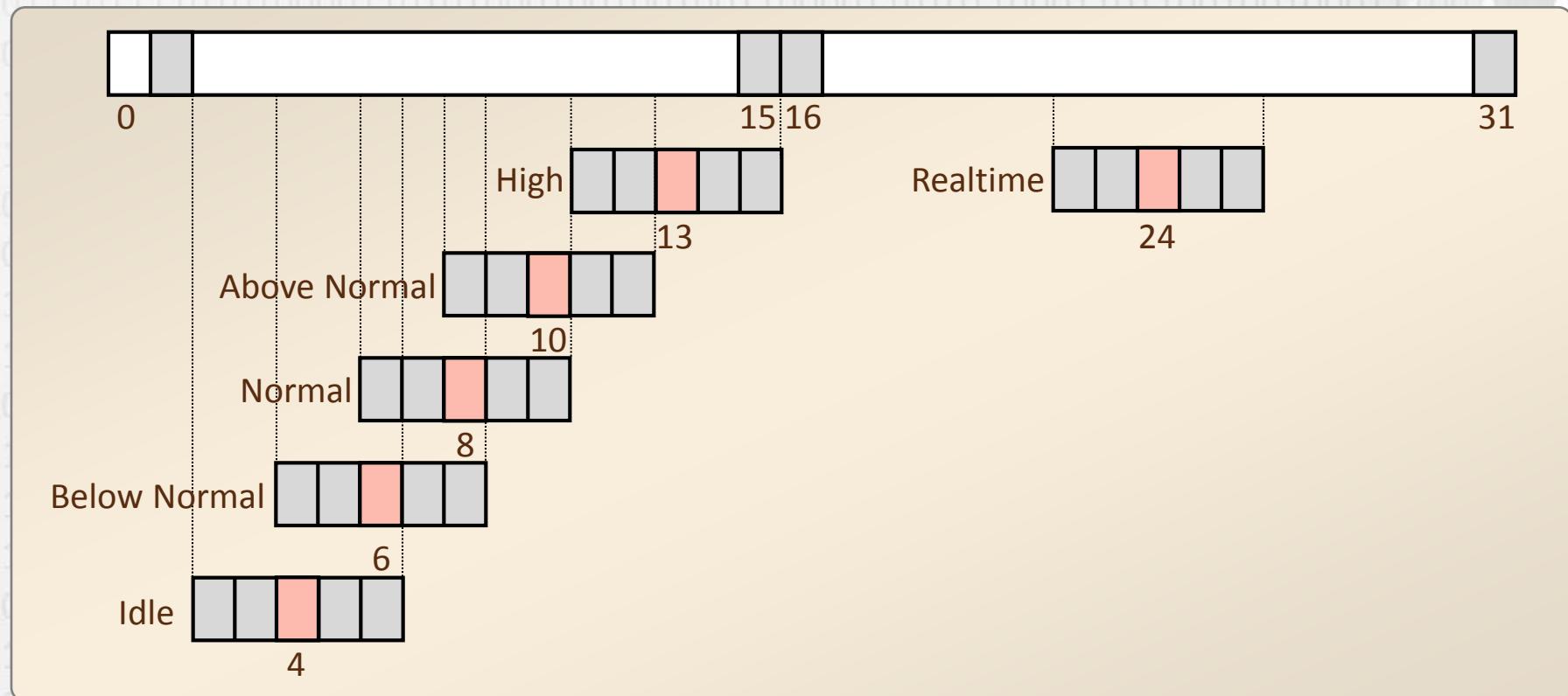
5. Külső megszakítás következik be.

6. Megszűnik a szál.

Ütemezés: prioritások (Windows)

- **Prioritások kezelése (grafikus illusztrációval)**

A „nyers” prioritásértékek 0–31-ig terjednek, de ezek közvetlenül nem állíthatók be, hanem egy kétszintű, ún. prioritási osztályokból (folyamatok) és prioritási szintekből (szálak) álló rendszeren keresztül kezelhetők.



Ütemezés: prioritások (Windows)

- **Prioritások kezelése (táblázatos illusztrációval)**

A „nyers” prioritásértékek 0–31-ig terjednek, de ezek közvetlenül nem állíthatók be, hanem egy kétszintű, ún. prioritási osztályokból (folyamatok) és prioritási szintekből (szálak) álló rendszeren keresztül kezelhetők.

Folyamatok prioritási osztályai

Szálak prioritási szintjei

	Realtime	High	Above Normal	Normal	Below Normal	Idle
Time Critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above Normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below Normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

- **Egyéb speciális jellegzetességek:**

- lapnullázó szál: 0
- üresjárat szál: „-1”
- Windows Vista, Windows 7: multimédia ütemezőszolgáltatás (MMCSS).

Ütemezés: az időszelet (Windows)

• A körbeforgó ütemezés következményei

Ez az ütemezési politika a processzorintenzív alkalmazásoknak kedvez. Az időszelet hossza emellett erősen befolyásolja az észlelt teljesítményt, így az operációs rendszer különböző (szerver- és klienscélú) változatai más-más alapértelmezett időszelet-adatokat használnak.

• Az időszelet („quantum”) hossza

- Az időszelet adatai a „Normal” prioritási osztálynál módosíthatók.
 - Módosítási lehetőségek: hossz, típus, előtérben lévő alkalmazás kiemelése;

	rövid			hosszú		
változó	6	12	18	12	24	36
fix	18	18	18	18	18	18

- 1 időszelet = $3 \times x$ rendszerórajel-ütem ;
- Újraütemezésnél 3-mal csökken az időszelet hossza;
- Windows Vista, Windows 7: a megszakításokban töltött idő (igazságos módon) már nem csökkenti az időszelet hosszát.

Ütemezés: kompenzáció (Windows)

- **Prioritás növelése („priority boosting”)**

- I/O művelet befejezésekor:

- képernyőmeghajtó, lemezmeghajtók: 1;
 - soros port, hálózat: 2;
 - bemeneti eszközök (billentyűzet, egér): 6;
 - hangszerközök: 8.

- Várakozó állapotból kilépéskor:

- eseményre, szemaforra stb. várakozás: 1;
 - előtérben lévő folyamat befejezte a várakozást: 2.

- A szál által várt GUI (ablak) esemény beérkezésekor: 2.

- Régóta készenlétben álló szálaknál:

- a rendszer ebben az esetben 2 időszakos időre 15-re emeli a szál prioritását.

- **Időszak módosítása („quantum stretching”)**

A prioritásnövelés idejére megkétszereződik az időszak hossza, ami még jobb esélyt biztosít az ideiglenes kiemelt alkalmazásnak a rövid CPU löketű műveletek gyors befejezésére.

Több processzor(mag) kezelése (Windows)

• Négyféle kernelmegvalósítás

A rendszer telepítéskor választja ki, hogy az egyes kernelmegvalósítások közül melyiket fogja alkalmazni. A megvalósítások funkcionálisan azonosak, de alkalmazkodnak az eltérő architektúrához.

– NTOSKRNL.EXE – egy processzor

- Egyprocesszoros rendszereken régi 32 bites CPU (Pentium I/II/III, AMD K5/K6/Athlon) esetén.

– NTKRNLPA.EXE – egy processzor/egymagos processzor, fizikai címkiterjesztés

- Egyprocesszoros rendszereken 32 bites CPU esetén
(Pentium IV/M, Core/Core2 Solo, AMD Athlon XP/64/Opteron).

– NTKRNLMP.EXE – több processzor/többmagos processzor

- Többmagos/többprocesszoros rendszereken régebbi 32 bites, illetve bármilyen x64-kompatibilis 64 bites CPU-knál.

– NTKRPAMP.EXE – több processzor/többmagos processzor, fizikai címkiterjesztés

- Többmagos/többprocesszoros rendszereken újabb 32 bites CPU-knál.

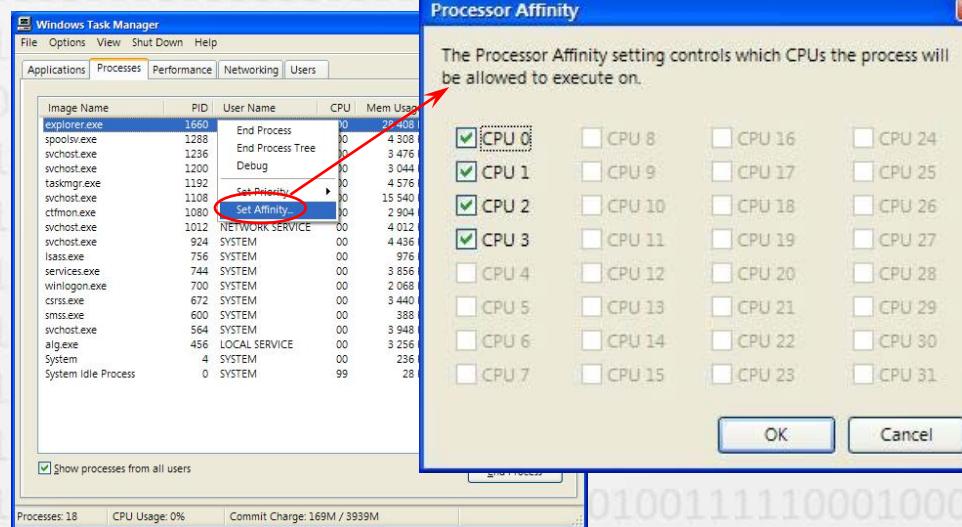
Több processzor(mag) kezelése (Windows)

- **Eltérések az egyprocesszoros ütemezéshez képest**
 - Az ütemezés alaplogikája változatlan: minden legmagasabb prioritású szálak egyike kerül futó állapotba valamelyik processzor(mag)on.
Automatikus terheléskiegyenlítés csak szálszinten történik.
 - A szálak elosztása teljesen szimmetrikus (azaz nincs „főprocesszor” és „alárendelt processzorok”), még abban az esetben is, ha a processzorok (magok) teljesítménye eltérő
 - Windows 2003/2008 Server: minden processzorhoz (maghoz) külön „ready” állapotú várakozósorok

Több processzor(mag) kezelése (Windows)

- **Processzor kiválasztása futtatandó szálhoz**

- minden folyamat rendelkezik egy „processzoraffinitás” nevű jellemzővel, amely meghatározza, hogy a folyamat szálai mely processzorokon futhatnak.



- A rendszer minden folyamathoz előre hozzárendel:
 - egy ún. „ideális” processzort (a szál indulásakor véletlenszerűen kijelölt processzor) és
 - egy ún. „utóbbi processzorok” listát, mely azon processzorokat tartalmazza, amelyeken a szál korábban már futott.
- **Futtatandó szál kiválasztása processzorhoz**
 - Az ütemező igyekszik azonos processzoron tartani a szálakat.

Párhuzamos algoritmusok tervezésének alapjai

Bevezetés

Részfeladatok és dekompozíció

Processzek és leképzés

Dekompozíciós technikák

Adatdekompozíció

Rekurzív dekompozíció

Felfedező dekompozíció

Spekulatív dekompozíció

Leképzési technikák és terhelés-kiegyensúlyozás

Statikus és dinamikus leképzés

A. Grama, A. Gupta, G. Karypis és . Kumar: „Introduction to Parallel Computing”, Addison Wesley, 2003. könyv anyaga alapján

Cél

- A párhuzamosítás maximalizálása.
- A párhuzamosításból következő extra tevékenységek (overhead) redukálása.
- A potenciális sebesség- és teljesítmény-növelés maximalizálása.

Fő lépések a cél irányában

- **A párhuzamosan végezhető munkarészek meghatározása**
 - részfeladatok
- **A részfeladatok processzorokhoz rendelése, leképezés**
 - processzek vs. processzorok
- **Az input/output és közbenső adatok különböző processzorok közötti szétosztása**
- **A megosztott adatok menedzselése**
 - input vagy közbenső adatok
- **A processzorok szinkronizálása a párhuzamos futás különböző pontjain**

Többszálú program tervezése

1. Felosztás

Adatok és számítások szétbontása kisebb, jól kezelhető részekre.

2. Kommunikáció

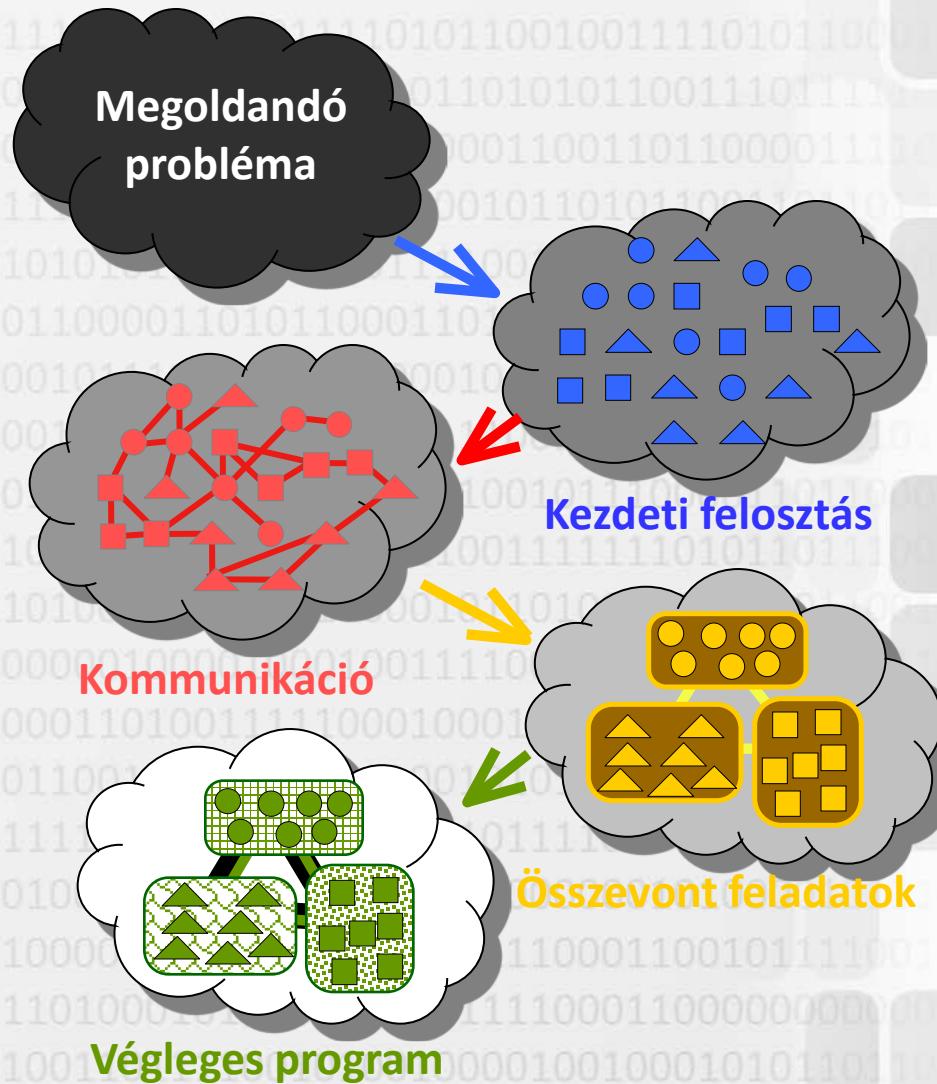
Adatok megosztása a számítások között, a megfelelő struktúrák kialakítása.

3. Összevonás

A feladatok részleges csoportosítása a nagyobb teljesítmény, illetve költséghatékonyúság érdekében.

4. Leképezés

A kapott feladatok konkrét feldolgozóegységekhez rendelése.



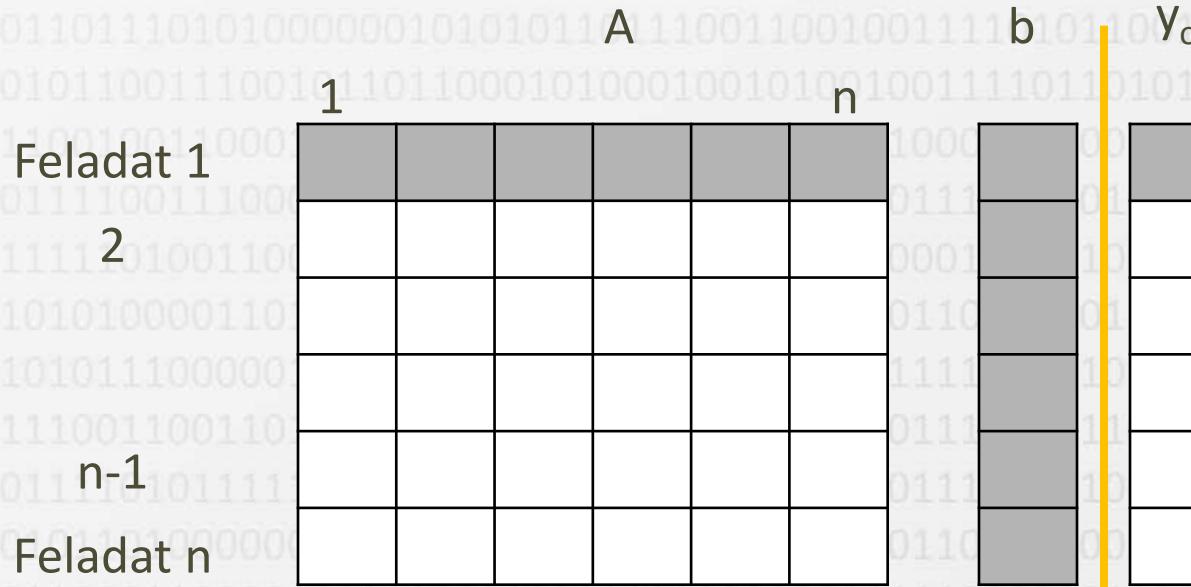
Bevezetés

Dekompozíció, részfeladatok, függőségi gráfok, processzek

Dekompozíció, részfeladatok és függőségi gráfok

- **Párhuzamos algoritmusok fejlesztésekor az első lépés a probléma olyan részfeladatokra bontása, amelyeket majd párhuzamosan végrehajthatunk.**
 - Az adott probléma többféleképpen bontható szét.
- **A részfeladatok lehetnek azonos vagy eltérő méretűek, vagy akár szakaszosak is.**
- **A dekompozíciót irányított gráffal lehet reprezentálni, ahol a csomópontok részfeladatokat jelentenek, az élek pedig azt mutatják, hogy a részfeladat eredménye szükséges-e a következő rész feldolgozásához.**
 - *Ezt feladatfüggőségi gráfnak nevezzük.*

Sűrű mátrix és vektor szorzása - példa



- Az y_{output} vektor minden elemének számítása független a többi elemétől, tehát a sűrű mátrix-vektor szorzatot n részfeladatra lehet bontani.

Sűrű mátrix és vektor szorzása - példa

- **Megfigyelések:**

- a részfeladatok megosztanak adatot (b vektor), de nincs közöttük semmilyen vezérlési kapcsolat;
- a végrehajtáskor semmilyen részfeladatnak sem kell várnia másikra;
- minden részfeladat ugyanolyan méretű (műveletek száma);
- ez a maximális számú részfeladat, amire szét tudtuk bontani a problémát?

Adatlekérdezés feldolgozása - példa

- A következő lekérdezést dolgozzuk fel:

Modell = "Civic" AND Évjárat = 2001 AND (Szín = "Zöld" OR Szín = "Fehér")

- Az adatbázis:

ID#	Modell	Évjárat	Szín	Eladó	Ár
4523	Civic	2002	Kék	MN	\$18,000
3476	Corolla	1999	Fehér	IL	\$15,000
7623	Camry	2001	Zöld	NY	\$21,000
9834	Prius	2001	Zöld	CA	\$18,000
6734	Civic	2001	Fehér	OR	\$17,000
5342	Altima	2001	Zöld	FL	\$19,000
3845	Maxima	2001	Kék	NY	\$22,000
8354	Accord	2000	Zöld	VT	\$18,000
4395	Civic	2001	Piros	CA	\$17,000
7352	Civic	2002	Piros	WA	\$18,000

Adatlekérdezés feldolgozása - példa

- A lekérdezés megvalósítását különböző módon részfeladatokra lehet bontani.
- minden részfeladat egy közbenső táblát generálhat, amely bizonyos kikötésnek tesz eleget.
 - Relációs adatbázis jellemzője.
- A gráf élei azt jelentik, hogy a részfeladat outputja szükséges a következő lépésben.

Adatlekérdezés feldolgozása – köztes táblák

ID#	Évjárat	ID#	Modell	ID#	Szín	ID#	Szín
7623	2001	4523	Civic	3476	Fehér	7623	Zöld
9834	2001	6734	Civic	6734	Fehér	9834	Zöld
6734	2001	4395	Civic			5342	Zöld
5342	2001	7352	Civic			8354	Zöld
3845	2001						
4395	2001						

2001

Civic

Fehér

zöld

Civic AND 2001

Fehér OR Zöld

ID#	Modell	Évjárat
-----	--------	---------

6734	Civic	2001
4395	Civic	2001

Civic AND 2001 AND (Fehér OR Zöld)

ID#	Szín
3476	Fehér
7623	Zöld
9834	Zöld
6734	Fehér
5342	Zöld
8354	Zöld

ID#	Modell	Évjárat	Szín
6734	Civic	2001	Fehér

Adatlekérdezés feldolgozása - példa

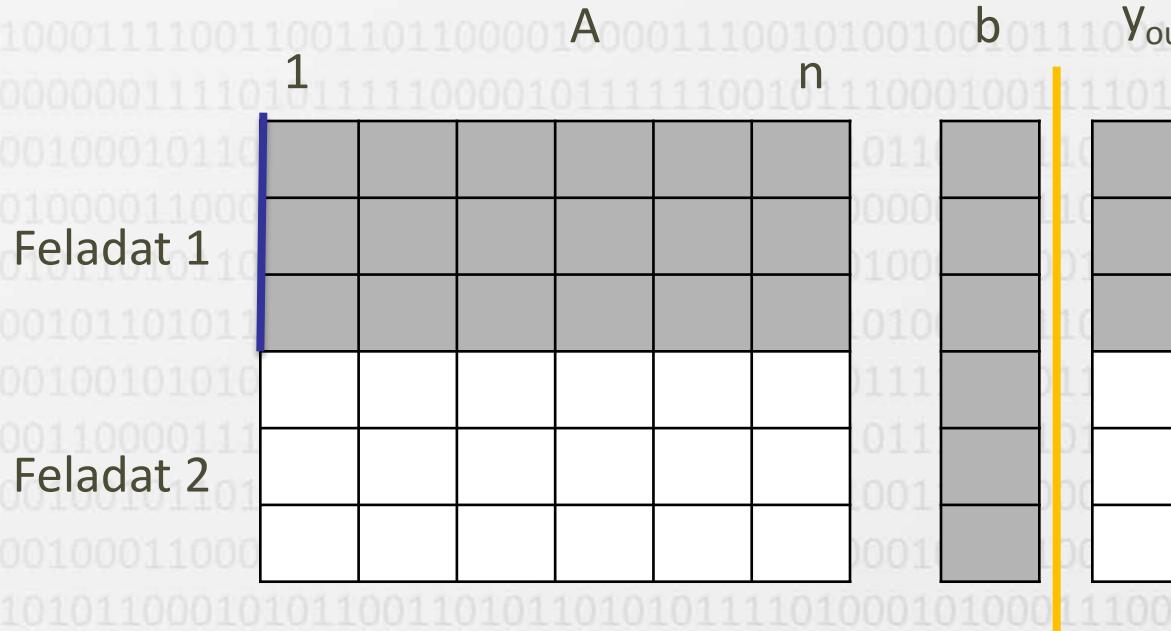
- Ugyanaz a probléma más úton is részfeladatokra bontható (jelen esetben az adatfüggőség szerint).
- Az különböző részfeladatokra bontás szignifikáns teljesítményeltérésekhez vezethet a párhuzamos működés során.

Adatlekérdezés feldolgozása – köztes táblák

ID#	Modell	ID#	Évjárat	ID#	Szín	ID#	Szín
4523	Civic	7623	2001	3476	Fehér	7623	Zöld
6734	Civic	9834	2001	6734	Fehér	9834	Zöld
4395	Civic	6734	2001			5342	Zöld
7352	Civic	5342	2001			8354	Zöld
		3845	2001				
		4395	2001				
Civic				2001		Fehér	
						Zöld	
						3476 Fehér	
						7623 Zöld	
						9834 Zöld	
						6734 Fehér	
						5342 Zöld	
						8354 Zöld	
2001 AND (Fehér OR Zöld)				ID#		Évjárat	
				7623		2001	
				9834		2001	
				6734		2001	
				5342		2001	
ID#	Modell	Évjárat	Szín				
6734	Civic	2001	Fehér	5342		Zöld	

A dekompozíció szemcsézettsége

- A szemcsézettséget az határozza meg, hogy hány részfeladatra osztjuk a problémát
- Sok részfeladatra bontás *finom szemcsézettséget*, kevés számú részfeladatra történő dekompozíció *durva szemcsézettséget* eredményez



- Példa: minden részfeladat az eredményvektor három elemét határozza meg.

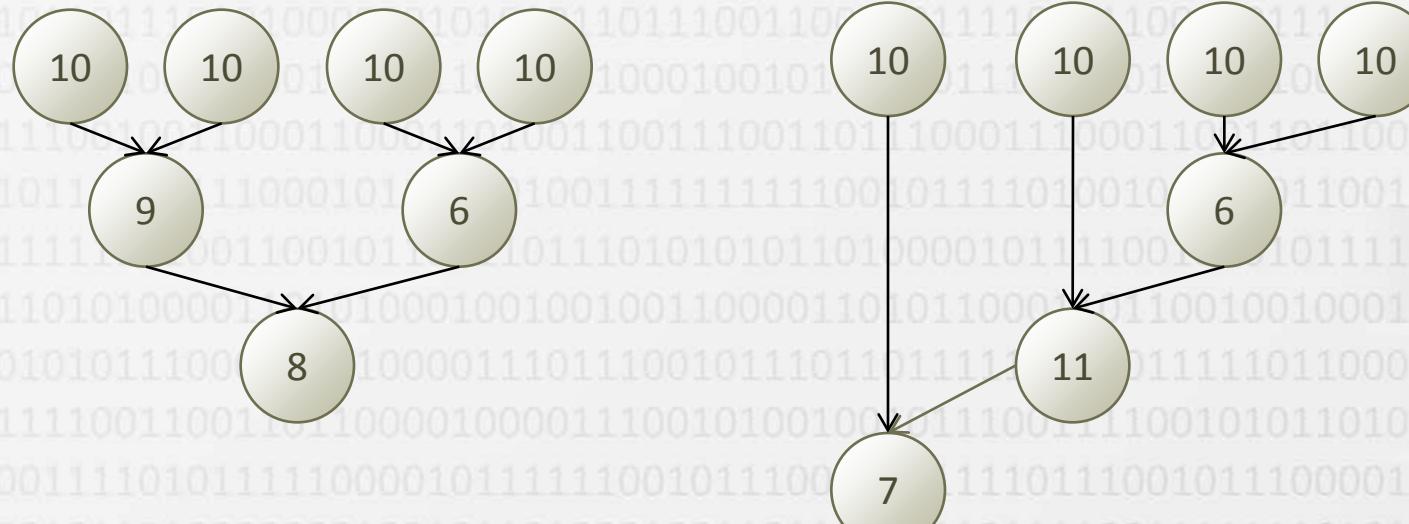
Párhuzamossági fok

- **Párhuzamossági fok, konkurencia fok**
 - A párhuzamosan futtatható részfeladatok száma határozza meg a dekompozíció *párhuzamossági fokát*, vagy *konkurencia fokát*.
- **Maximális párhuzamossági fok**
 - A *maximális párhuzamossági fok* legyen bármely pillanatot tekintve a legtöbb ilyen részfeladat. (A program futása során a párhuzamosan futó feladatok száma változhat.)
- **Átlagos párhuzamossági fok**
 - Az *átlagos párhuzamossági fok* az átlaga azoknak a részfeladatoknak, amelyeket párhuzamosan lehet végrehajtani a program futása során (teljes idő/kritikus út hossza).
- **A párhuzamossági fok növekszik, ha a dekompozíció szemcsézettsége finomodik.**

Kritikus út hossza

- A feladatfüggőségi-gráfban egy (irányított) út egymás után végrehajtandó részfeladatokat reprezentál.
- A részfeladatoknak költsége (ideje) van.
- **Kritikus út**
 - A leghosszabb ilyen út határozza meg a program legrövidebb futásidéjét.
 - A feladatfüggőségi-gráfban a leghosszabb út hossza a kritikus út hossza.

Kritikus út hossza



- **Ha minden részfeladat (kör) a jelölt időegységig tart, akkor mennyi a legrövidebb futási idő a két dekompozíció esetén?**
- **Mennyi a maximális párhuzamossági fok?**
- **Mennyi az átlagos párhuzamossági fok?**
- **Hány processzor szükséges a két esetben a minimális idejű futáshoz?**

A párhuzamos teljesítmény korlátai

- **Úgy tűnhet, hogy a párhuzamosítás következtében tetszőlegesen kicsi lehet a futási idő a dekompozíció finomításával.**
- De a finomításnak természetes korlátai vannak (a sűrű mátrixos feladatban pl. (n^2) párhuzamos részfeladat lehet maximum).
- A párhuzamos részfeladatok adatot cserélhetnek egymással, amely szintén extra kommunikációs ráfordítást jelent.
- A dekompozíció szemcsézettsége és a párhuzamossá alakítás extra ráfordítása közötti kompromisszum szintén meghatározhatja a teljesítménykorlátokat.

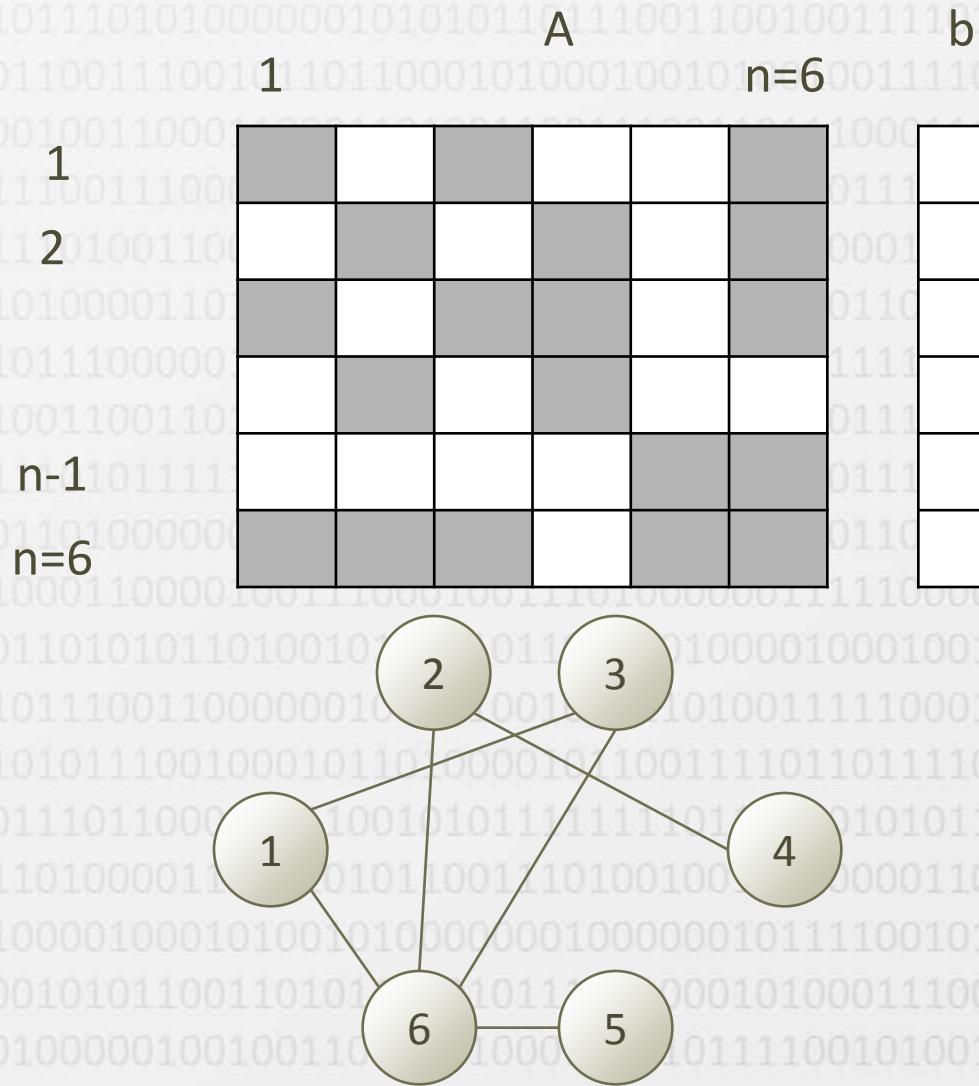
Feladatkölcsönhatási-gráf

- A dekompozíció során meghatározott részfeladatok általában adatot cserélnek egymással (pl. a sűrű mátrixvektor szorzásnál, ha a vektort nem másoljuk le minden részfeladathoz, a feladatoknak kommunikálniuk kell egymással).
- **Feladatkölcsönhatási-gráf**
 - A részfeladatok mint csomópontok és a feladatok kapcsolata/az adatcseré mint élek gráfot határoznak meg.
 - A *feladatkölcsönhatási-gráf* adatfüggőséget reprezentál, míg a *feladatfüggőségi-gráf* vezérlési kapcsolatot,
 - Feladatkölcsönhatási-gráf része a feladatkölcsönhatási-gráfnak,

Feladatkölcsönhatási-gráf - példa

- Az A ritka mátrix és a b vektor szorzása a feladat.
- Mint korábban, az eredményvektor minden eleme függetlenül számolható.
- A korábbi sűrű mátrix-vektor szorzattal ellentétben csak A nem nulla elemei vesznek részt a számításban.
- Ha memóriaoptimalizási szempontból a b vektort megosztjuk a részfeladatokat között, ekkor a feladatkölcsönhatási-gráf azonos A mátrix gráfjával (azzal a mátrixszal, amely A szomszédsági struktúráját reprezentálja).

Feladatkölcsönhatási-gráf - ábra



Dekompozíciós módszerek



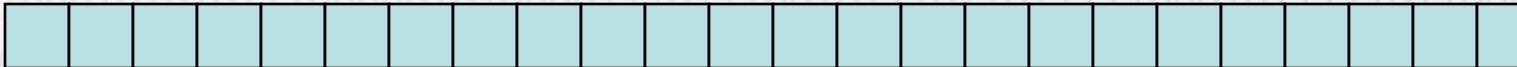
Dekompozíciós módszerek

Nincs általános recept, de gyakran a következőket használják:

- **adatdekompozíció**
 - **rekurzív dekompozíció**
 - **felderítő dekompozíció**
 - **spekulatív dekompozíció**
 - **futószalag-dekompozíció**
 - **hibrid dekompozíció**
- feladatdekompozíciós módszerek**

Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



Adatdekompozíció - példa

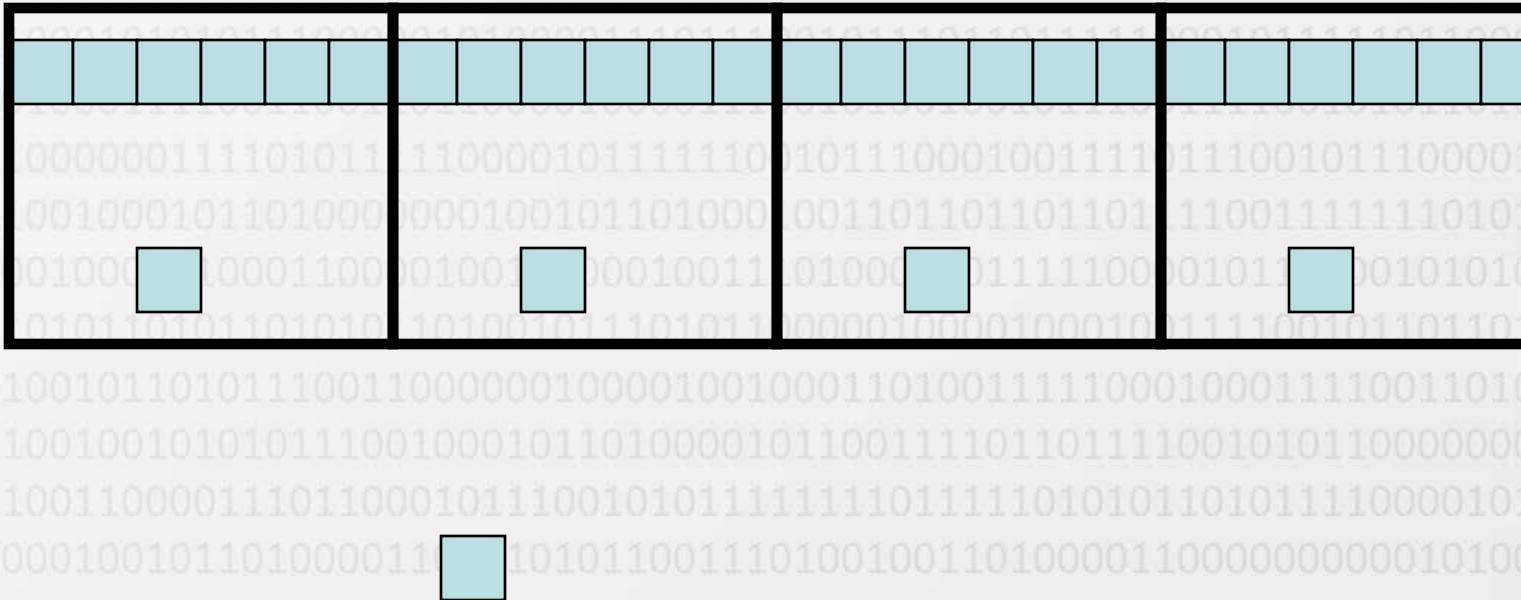
Tömb legnagyobb elemének megkeresése 4 folyamattal:

Szál 0

Szál 1

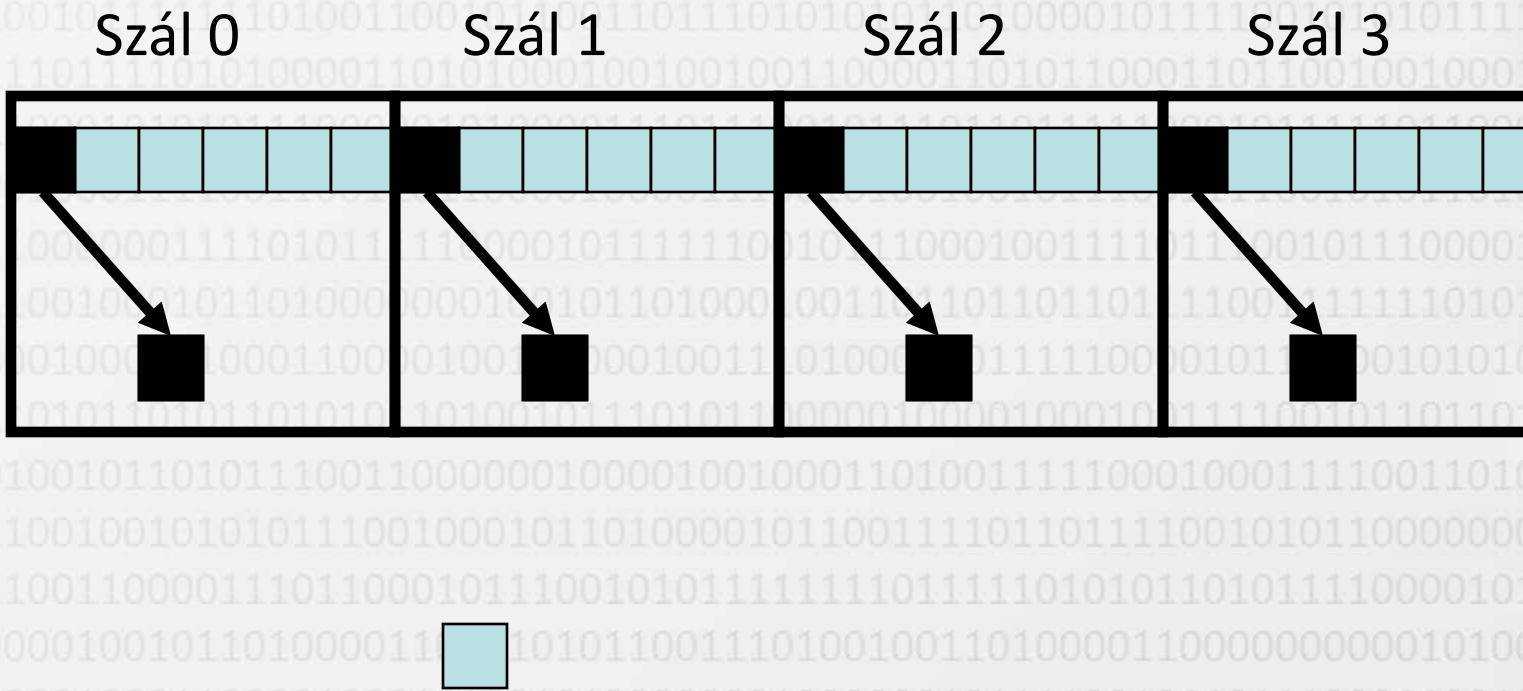
Szál 2

Szál 3



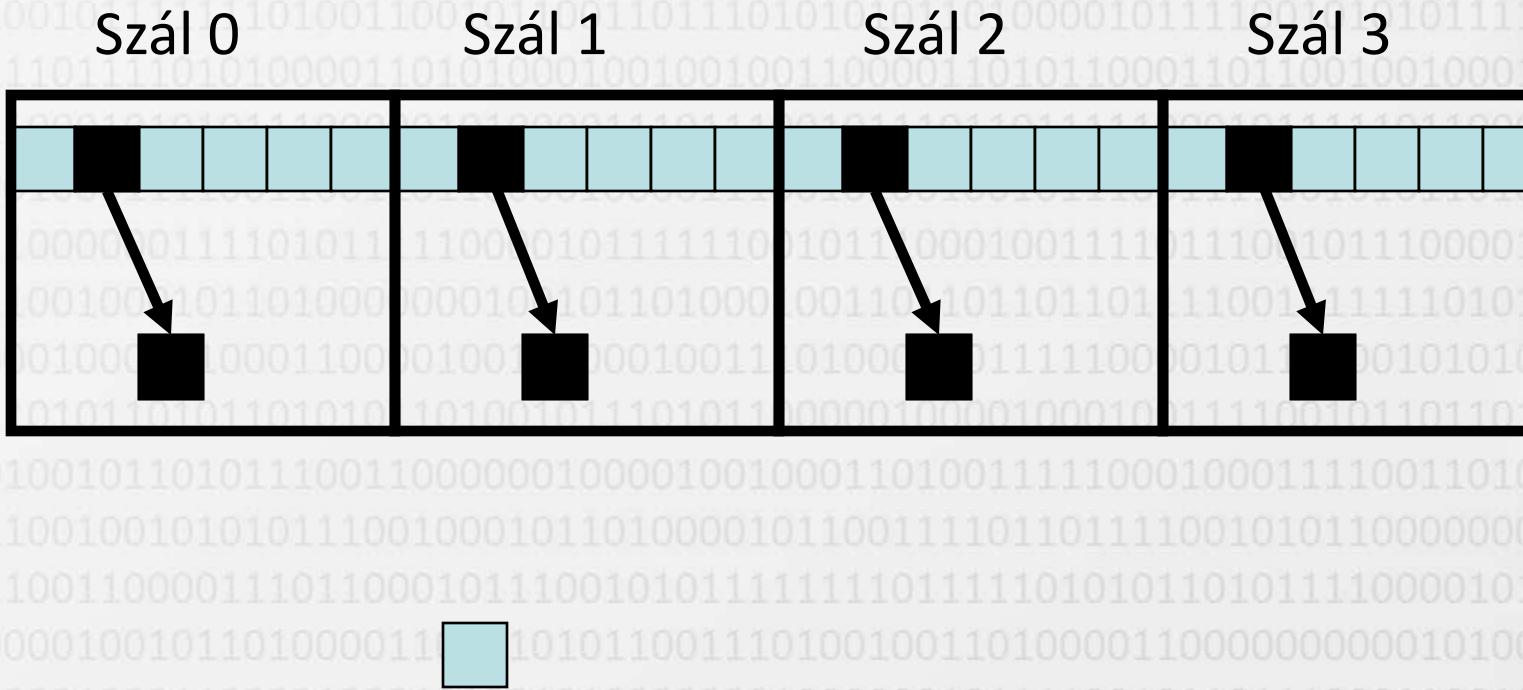
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



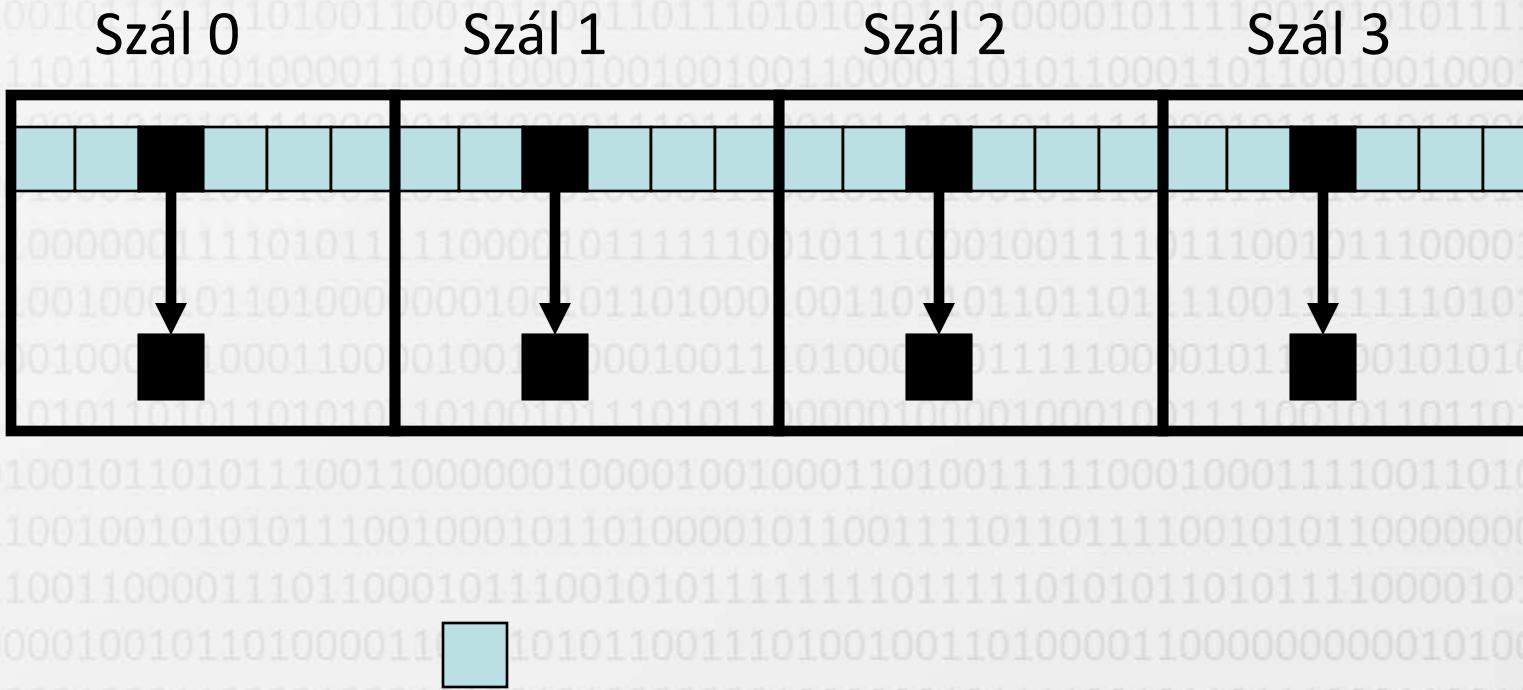
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



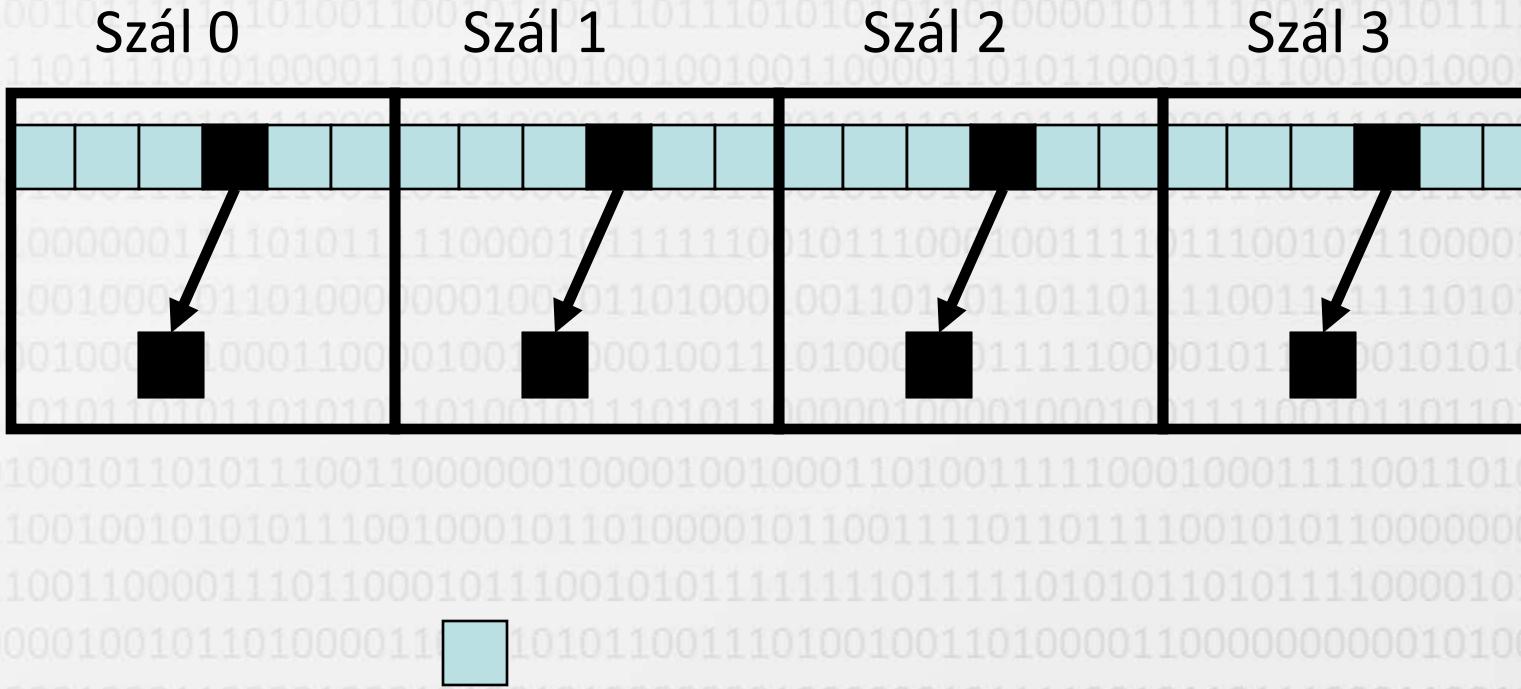
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



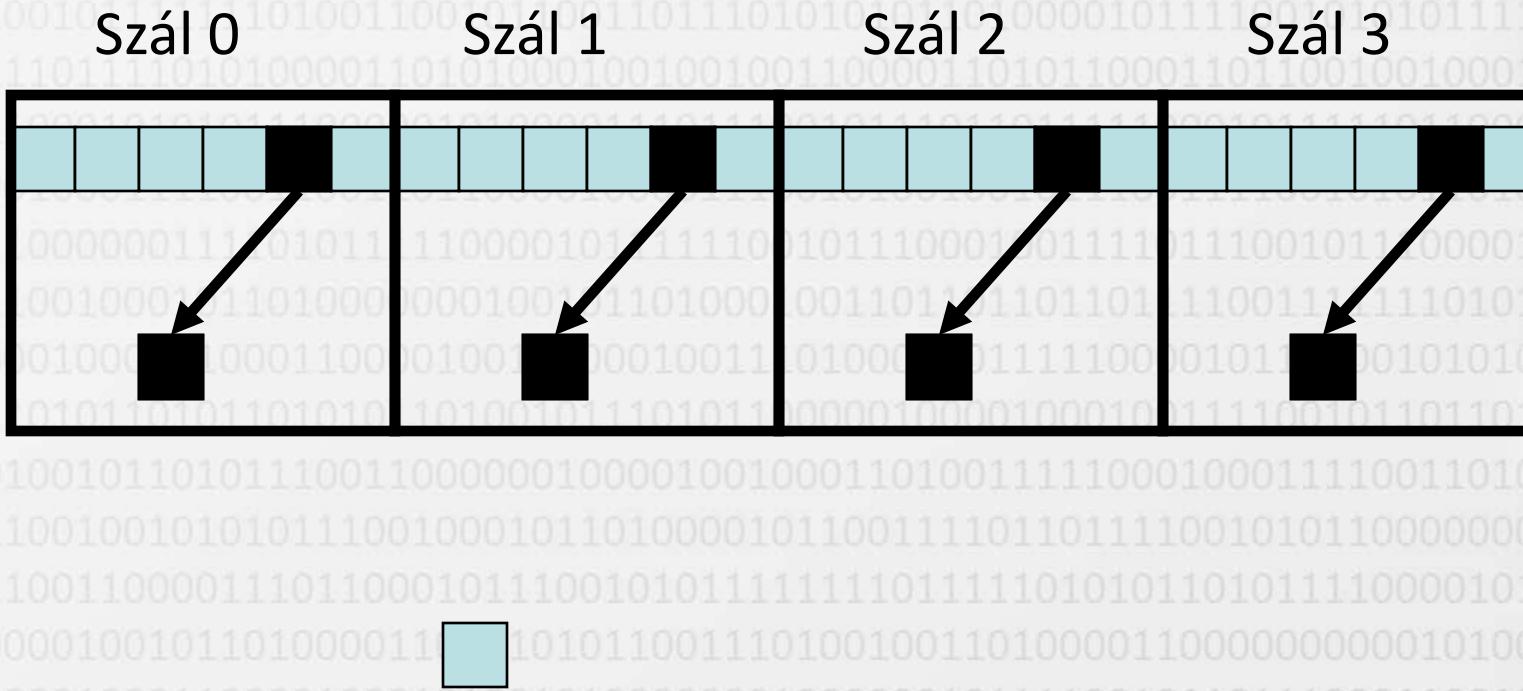
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



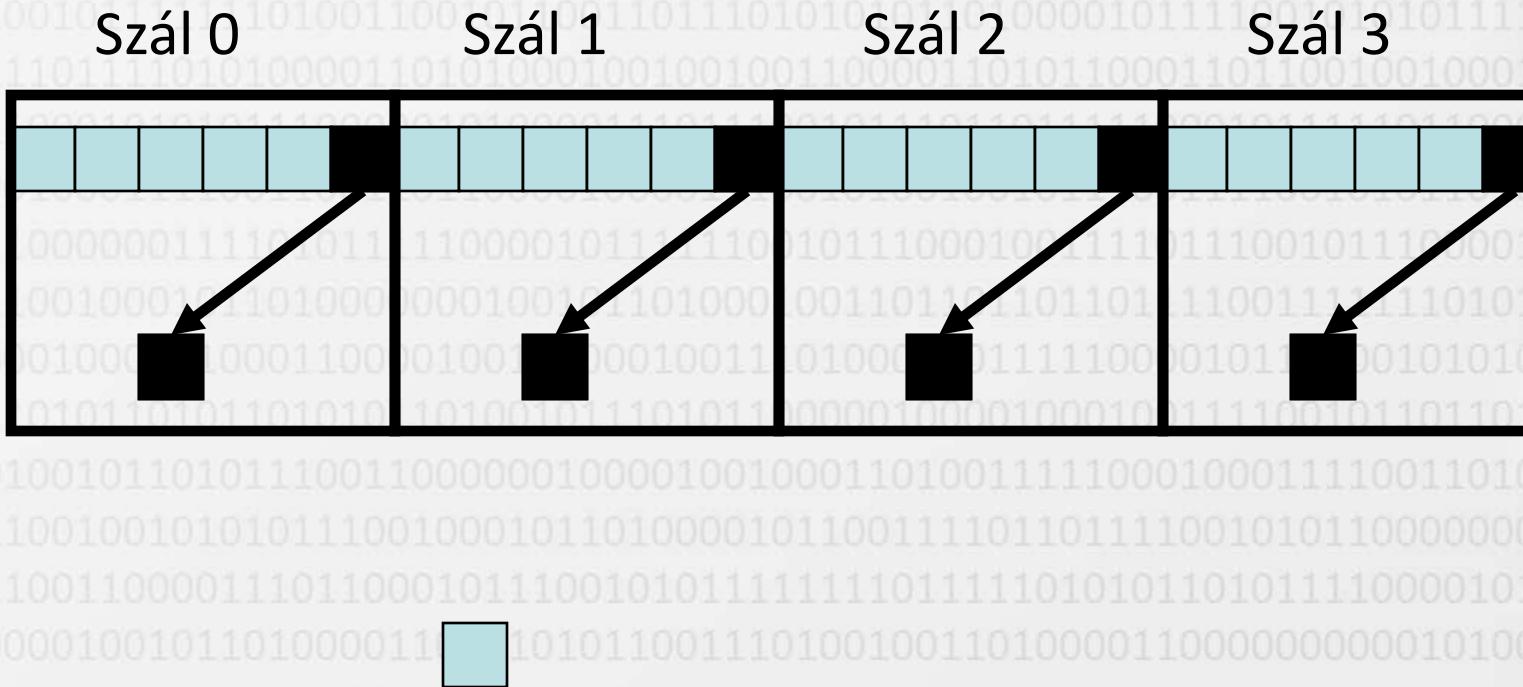
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



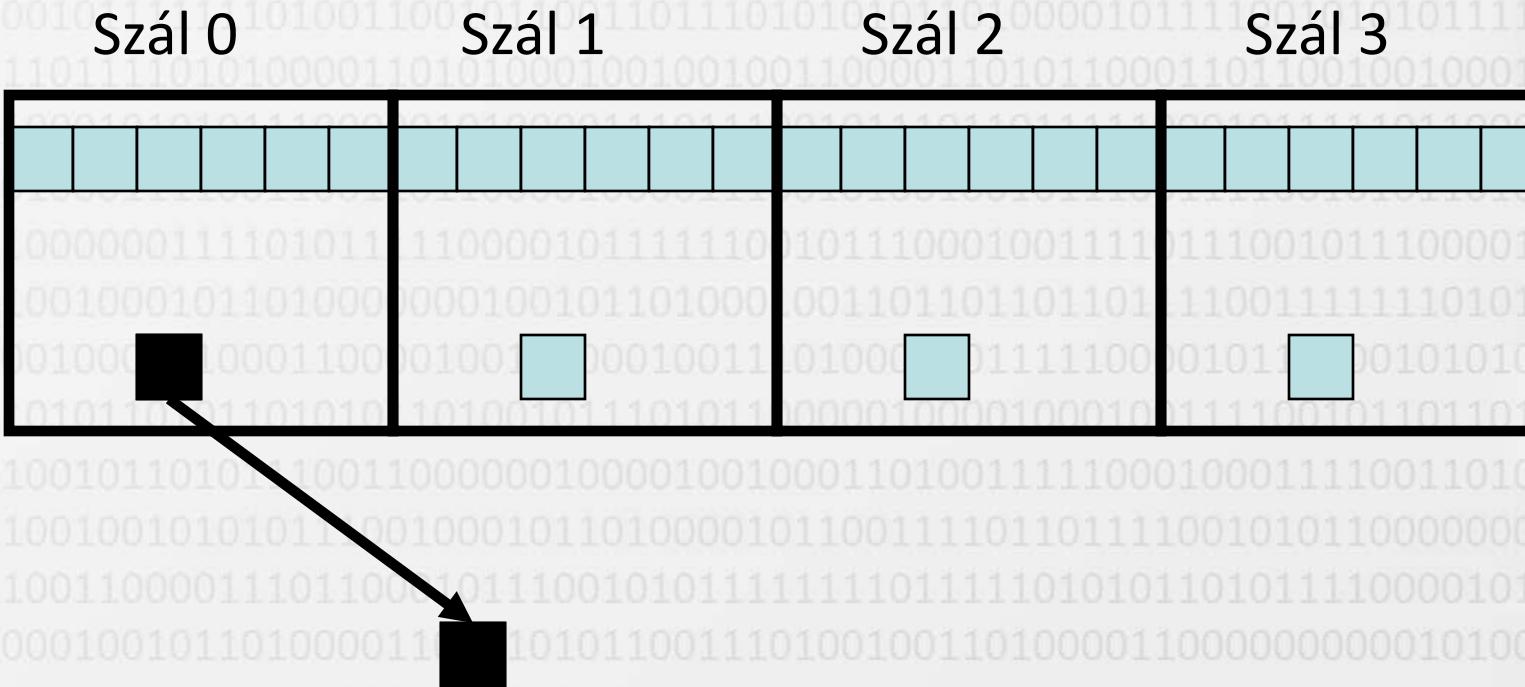
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



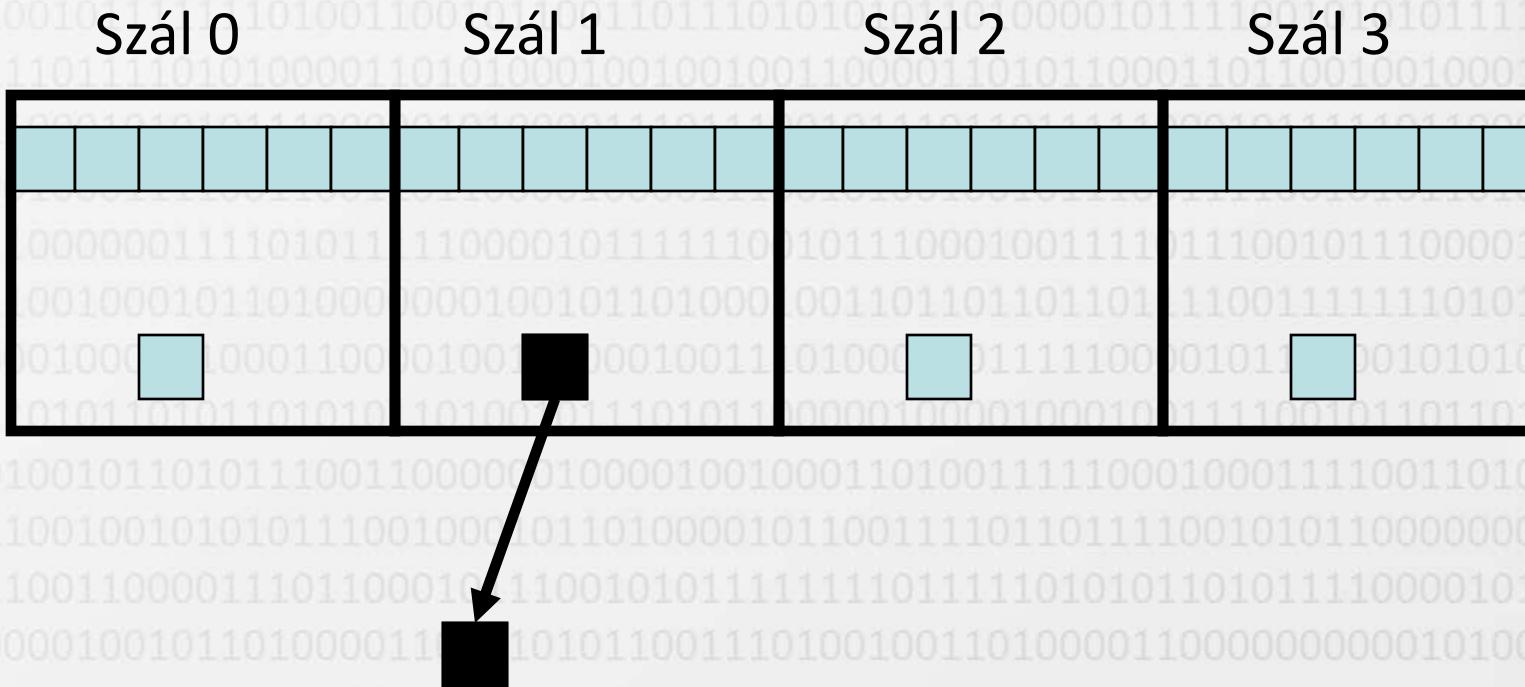
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



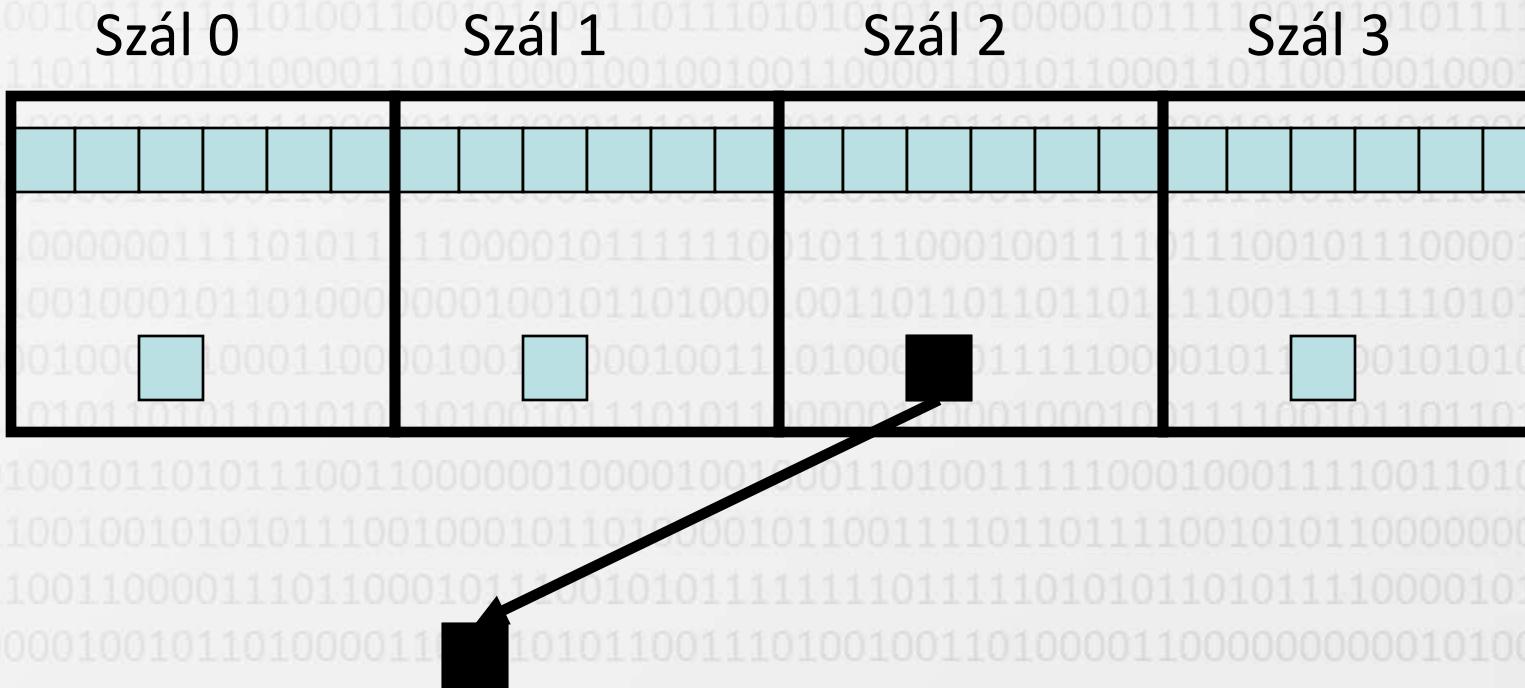
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



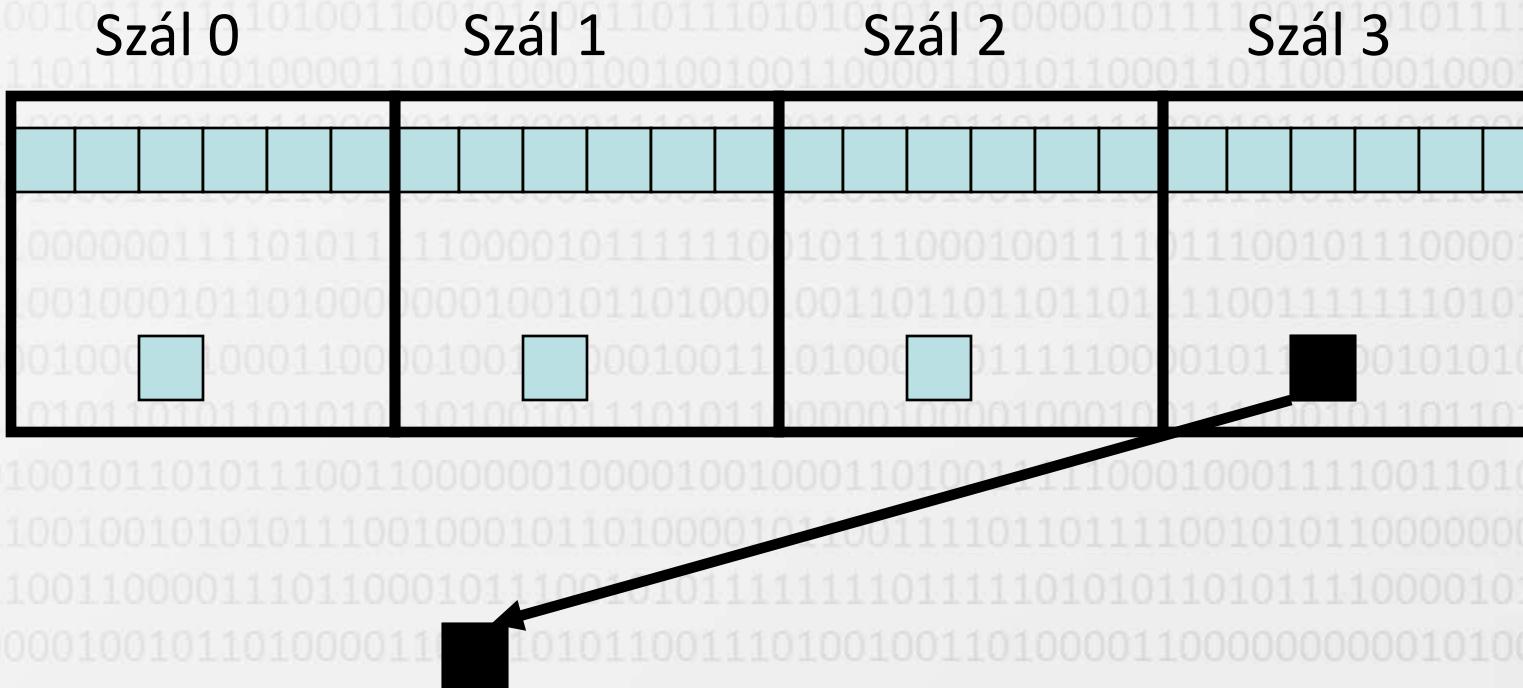
Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



Adatdekompozíció - példa

Tömb legnagyobb elemének megkeresése 4 folyamattal:



Adatdekompozíció

- Nagy mennyiségű adaton dolgozó problémák esetén használatos.
- Alapelv: a részfeladatokat úgy kapjuk meg, hogy a nagyszámú adatból indulunk ki.
- Gyakran két lépésben valósítják meg az adatdekompozíciót:
 - 1: adatok felosztása;
 - 2: az adatparticionálásból indukált számításifeladat-particionálás.
- Milyen adatparticionálásból induljunk ki?
 - Input/Output/Közbenső
- Az indukált számításokat miként hajtsuk végre?
 - „Tulajdonos számol” szabály: amihez rendeljük az adatot, az végzi a számítást.

Adatdekompozíció: output adatból

- Ha az output részei egymástól függetlenek.
- Az input egyszerű függvénye az output.
- A partíciók részfeladatokhoz rendelése a probléma természetes megközelítése.

Output adatdekompozíció - példa

- Két $n \times n$ mátrix, A és B szorzatának eredménye C . Az eredmény mátrix négy részre (is) osztható

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Feladat1: $A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Feladat2: $A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Feladat3: $A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Feladat4: $A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Output adatdekompozíció - példa

- A particionálás nem biztos, hogy egyértelmű részfeladatra bontást eredményez. Az előző feladat egy másik megoldása:

$$\textbf{Feladat1: } C_{1,1} = A_{1,1}B_{1,1}$$

$$\textbf{Feladat2: } C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$$

$$\textbf{Feladat3: } C_{1,2} = A_{1,1}B_{1,2}$$

$$\textbf{Feladat4: } C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$$

$$\textbf{Feladat5: } C_{2,1} = A_{2,1}B_{1,1}$$

$$\textbf{Feladat6: } C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$$

$$\textbf{Feladat7: } C_{2,2} = A_{2,1}B_{1,2}$$

$$\textbf{Feladat8: } C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$$

Output adatdekompozíció - példa

Tranzakciós adatbázis (input)
A, B, C, E, G,
H
B, D, E, F, K, L
A, B, F, H, L
D, B, F, H
F, G, H, K
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Cikkhalmazok (input)
A, B, C
D, E
C, F, G
A, E
C, D
D, K
B, C, F
C, D, K

Cikkhalmaz gyakoriságok (output)
1
3
0
2
1
2
2
0
0

- Tranzakciós adatbázisban cikkhalmazok gyakoriságának meghatározása output szerinti particionálással.
- Másolat az adatbázisról taszkonként?
- Adatbázis-particionálás a taszkok között, majd részeredmények összegzése.

• Gyakoriság szerinti szétosztás két feladatra

Tranzakciós adatbázis (input)
A, B, C, E, G,
H
B, D, E, F, K, L
A, B, F, H, L
D, B, F, H
F, G, H, K
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Cikkhalmazok (input)
A, B, C
D, E
C, F, G
A, E

Cikkhalmaz gyakoriságok (output)
1
3
0
2

1. taszk

Tranzakciós adatbázis (input)
A, B, C, E, G,
H
B, D, E, F, K, L
A, B, F, H, L
D, B, F, H
F, G, H, K
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Cikkhalmazok (input)
C, D
D, K
B, C, F
C, D, K

Cikkhalmaz gyakoriságok (output)
1
2
0
0

2. taszk

Input adatparticionálás

- **Alkalmazható, ha minden output az input függvényeként természetesen számítható.**
- **Sok esetben ez a természetes út, mert az output előre nem ismert (pl. rendezés, minimum meghatározás).**
- **A részfeladat minden inphothoz kapcsolható. Olyan mennyiségű számítást végez el a részfeladat, amennyit csak lehet az adataiból. Rákövetkező feldolgozás kombinálja a részeredményeket.**

Input adatparticionálás - példa

- A tranzakciós adatbázisban az inputot partcionáljuk. Ez meghatározza a részfeladatokat, minden taszk részeredményeket számol minden cikkhalmazra. Ezeket összegezzük a második lépésben.
- Gyakoriság szerinti szétosztás két feladatra

Tranzakciós adatbázis (input)

A, B, C, E, G,
H
B, D, E, F, K, L

Cikkhalmazok (input)
A, B, C
D, E
C, F, G
A, E
C, D
D, K
B, C, F
C, D, K

Cikkhalmaz gyakoriságok (output)

1
2
0
1
0
1
0
0

1. taszk

Tranzakciós adatbázis (input)

A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Cikkhalmazok (input)
A, B, C
D, E
C, F, G
A, E
C, D
D, K
B, C, F
C, D, K

Cikkhalmaz gyakoriságok (output)

0
1
0
1
1
1
0
0

2. taszk

Input és output particionálás - példa

- Gyakran alkalmazható magasabb fokú párhuzamosítás céljából.
- A tranzakciós adatbázist és a cikkhalmazokat is szétoztuk.

Tranzakciós adatbázis (input)

A, B, C, E, G,
H
B, D, E, F, K, L
A, B, F, H, L
D, B, F, H
F, G, H, K

Cíkkhalmazok (input)

A, B, C
D, E
C, F, G
A, E

1. taszk

Cíkkhalmaz gyakoriságok (output)

1
2
0
1

Tranzakciós adatbázis (input)

A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Cíkkhalmazok (input)

A, B, C
D, E
C, F, G
A, E

Cíkkhalmaz gyakoriságok (output)

0
1
0
1

3. taszk

Tranzakciós adatbázis (input)

A, B, C, E, G,
H
B, D, E, F, K, L
A, B, F, H, L
D, B, F, H
F, G, H, K

Cíkkhalmazok (input)

C, D
D, K
B, C, F
C, D, K

Tranzakciós adatbázis (input)

A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Cíkkhalmazok (input)

C, D
D, K
B, C, F
C, D, K

Cíkkhalmaz gyakoriságok (output)

1
1
0
0

Tranzakciós adatbázis (input)

1
1
0
0

Cíkkhalmaz gyakoriságok (output)

0
1
1
0
0

4. taszk

Közbenső adat particionálása

A _{1,1}
A _{2,1}

•

B _{1,1}	B _{1,2}
------------------	------------------



D _{1,1,1}	D _{1,1,2}
D _{1,2,1}	D _{1,2,2}

+

A _{1,2}
A _{2,2}

•

B _{2,1}	B _{2,2}
------------------	------------------



D _{2,1,1}	D _{2,1,2}
D _{2,2,1}	D _{2,2,2}



C _{1,1}	C _{1,2}
C _{2,1}	C _{2,2}

Közbenső adat particionálása

Első szakasz

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \\ D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Második szakasz

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A dekompozíció tartalmazza D particionálását is.

Taszk 01: $D_{1,1,1} = A_{1,1}B_{1,1}$

Taszk 02: $D_{2,1,1} = A_{1,2}B_{2,1}$

Taszk 03: $D_{1,1,2} = A_{1,1}B_{1,2}$

Taszk 04: $D_{2,1,2} = A_{1,2}B_{2,2}$

Taszk 05: $D_{1,2,1} = A_{2,1}B_{1,1}$

Taszk 06: $D_{2,2,1} = A_{2,2}B_{2,1}$

Taszk 07: $D_{1,2,2} = A_{2,1}B_{1,2}$

Taszk 08: $D_{2,2,2} = A_{2,2}B_{2,2}$

Taszk 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

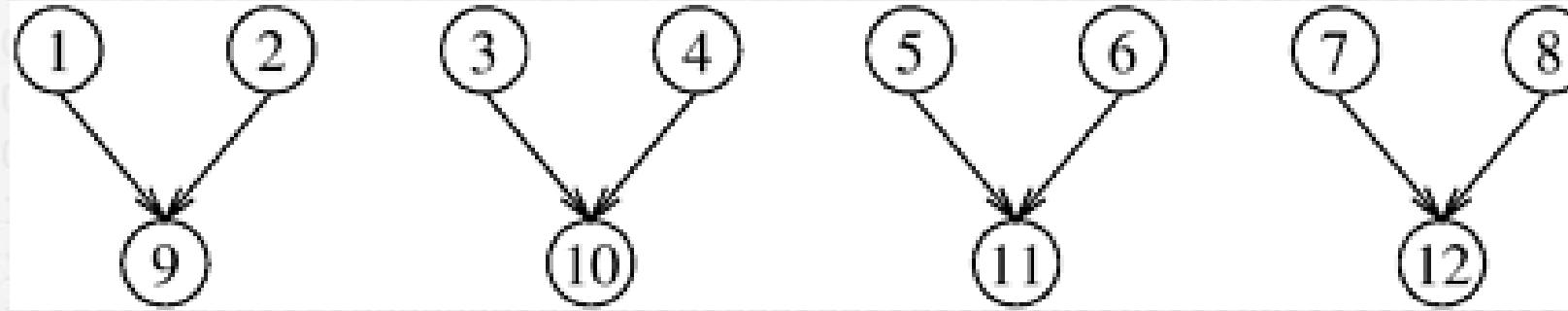
Taszk 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Taszk 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Taszk 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Közbenső adat particionálása

- A dekompozíció taszk-függőségi gráfja

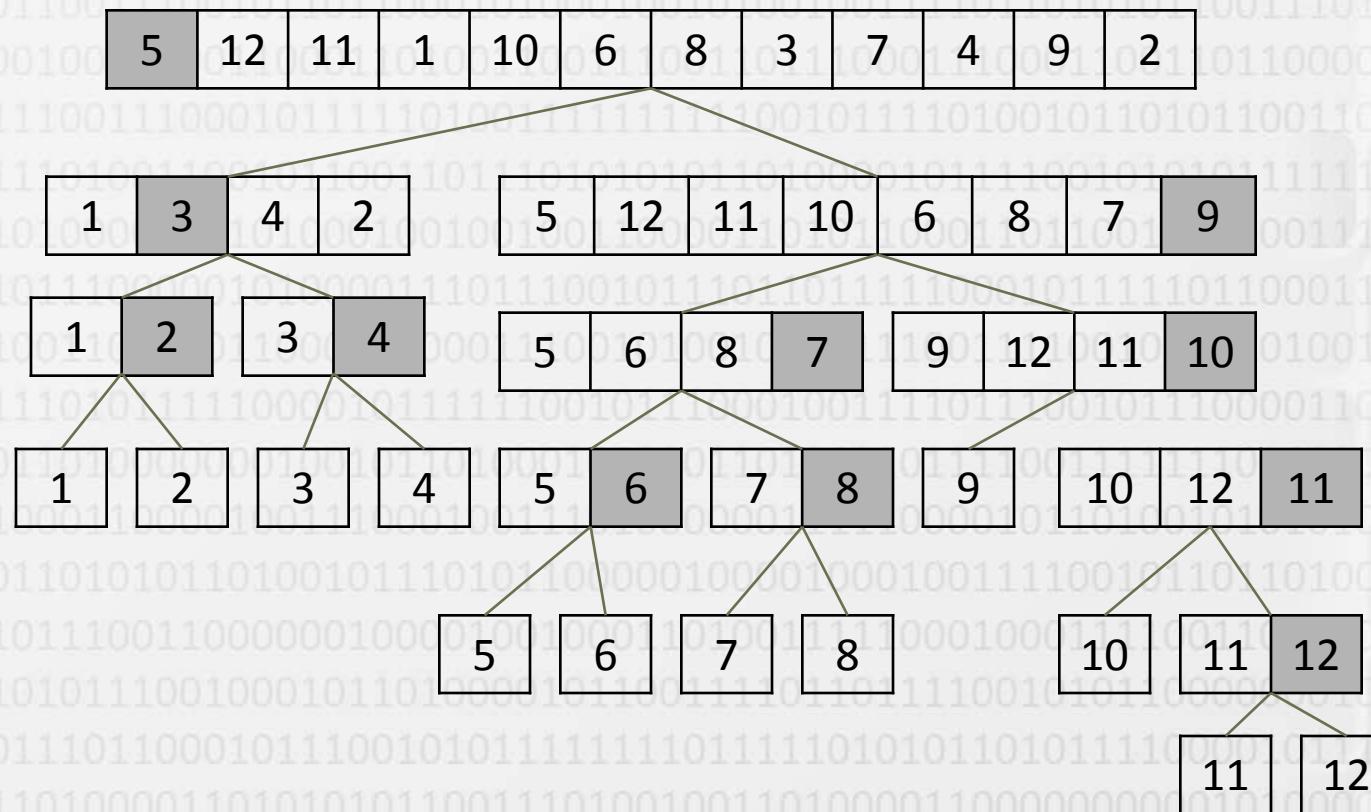


Rekurzív dekompozíció

- Általában minden olyan esetben használható, amikor az „oszd meg és uralkodj” stratégiát alkalmazhatjuk.
- Az adott feladatot először részekre bontjuk.
- Ezeket a részproblémákat rekurzívan tovább bontjuk a kívánt szemcsézettség eléréséig.

Rekurzív dekompozíció - példa

- Egy klasszikus példa a Quicksort algoritmus



- A vezérlőelem segítségével két részre bontjuk a listát, és a részlistákat párhuzamosan dolgozhatjuk fel (részlisták feldolgozása független részfeladat). Rekurzívan végezhető.

Rekurzív dekompozíció - példa

- Lista minimális elemének keresése (vagy más asszociatív operáció) esetén is alkalmazható az „oszd meg és uralkodj” elv.

- **Kiindulás: soros megvalósítás**

1. **function** Soros_Min(A, n)

2. **begin**

3. $min = A[0];$

4. **for** $i := 1$ **to** $n - 1$ **do**

5. **if** ($A[i] < min$) $min := A[i];$

6. **endfor;**

7. **return** $min;$

8. **end** Soros_Min

Ahol:

A – input tömb

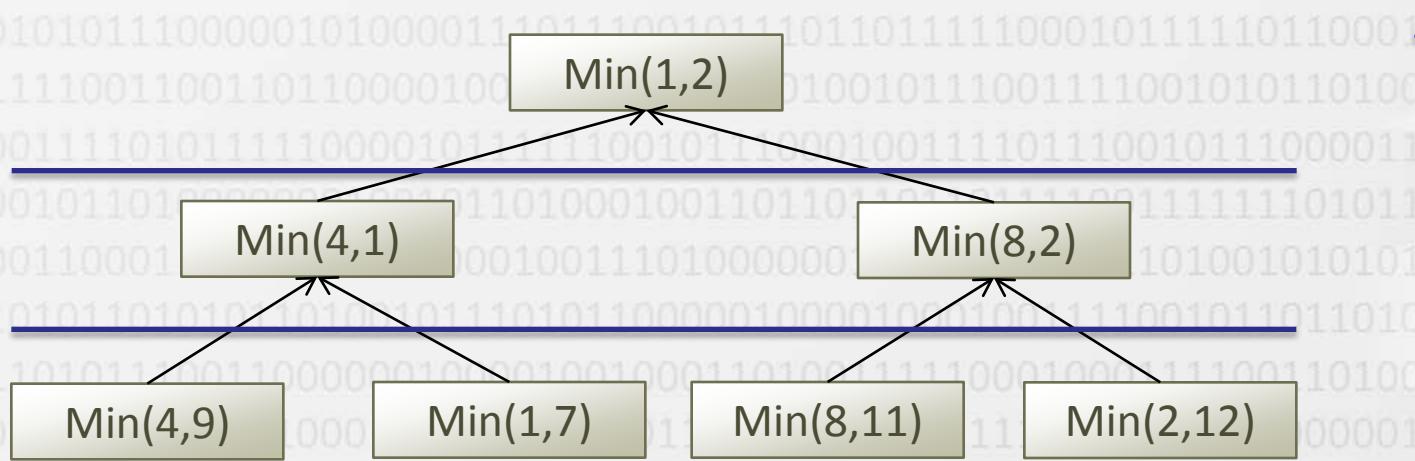
n – az A tömb hossza

Rekurzív dekompozíció - példa

```
1. function Rekurzív_Min (A, n)
2. begin
3.   if (n = 1) then
4.     min := A [0];
5.   else
6.     lmin := RECURSIVE_MIN (A, n/2);
7.     rmin := RECURSIVE_MIN ( &(A[n/2]), n - n/2 );
8.     if (lmin < rmin) then
9.       min := lmin;
10.    else
11.      min := rmin;
12.    endelse;
13.  endelse;
14.  return min;
15. end Rekurzív_Min
```

Rekurzív dekompozíció - példa

- A feladatfüggőségi-gráfban minden csomópont két szám közül a kisebbet adja vissza. Az eredeti halmaz, amelyben a minimumot keressük: {4, 9, 1, 7, 8, 11, 2, 12}



Felderítő dekompozíció

- Olyan számítások dekompozíciója, ahol a megoldás állapottérben történő keresésekhez kapcsolódik.

Példa: tili-toli

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

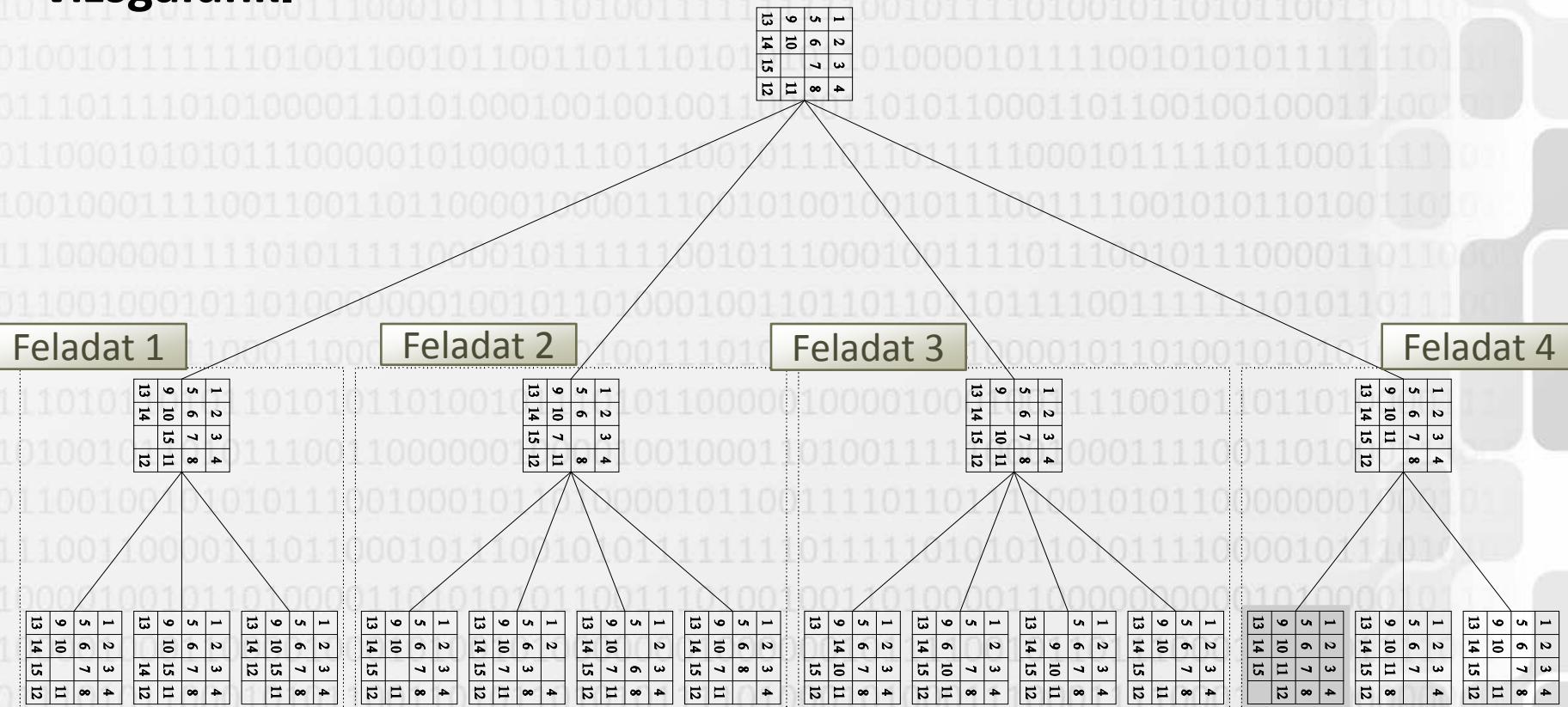
1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	

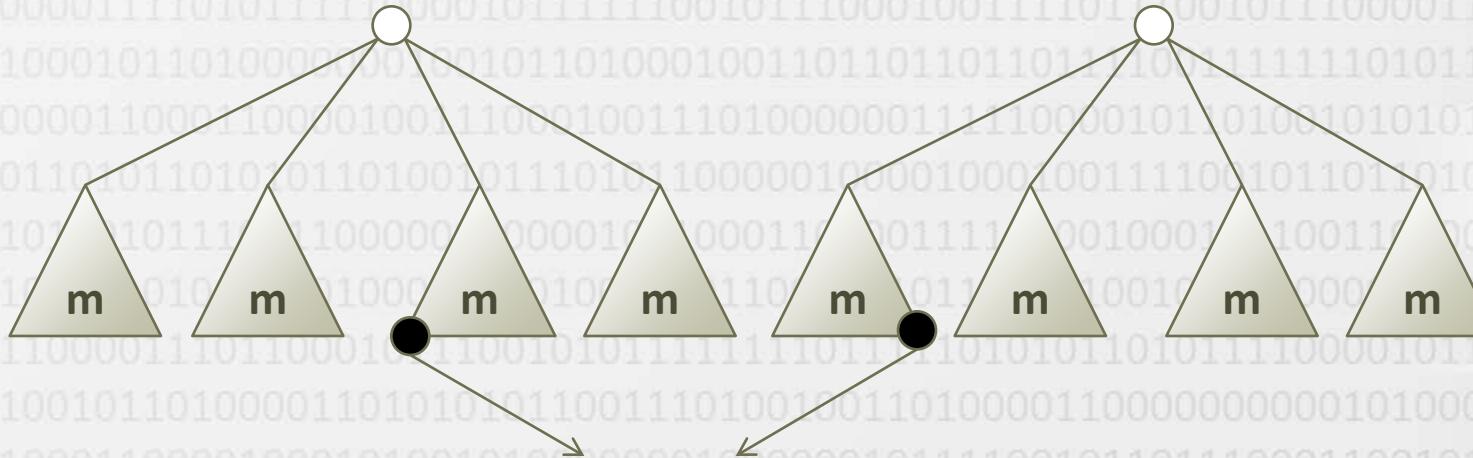
Felderítő dekompozíció - példa

- A állapottér felfedezése oly módon, hogy különböző lehetséges követő lépéseket mint önálló feladatokat vizsgálunk.



Felderítő dekompozíció: anomáliák a számításban

- A **felderítő dekompozíció esetében** a **dekompozíciós technika megváltoztathatja a szükséges munkamennyiséget;** akár megnövelheti, akár csökkentheti azt.
- **Nem biztos, hogy mind hasznos munka.**



Összes soros munka: $2m+1$

Összes párhuzamos: 1

Megoldás

Összes soros munka: m

Összes párhuzamos: $4m$

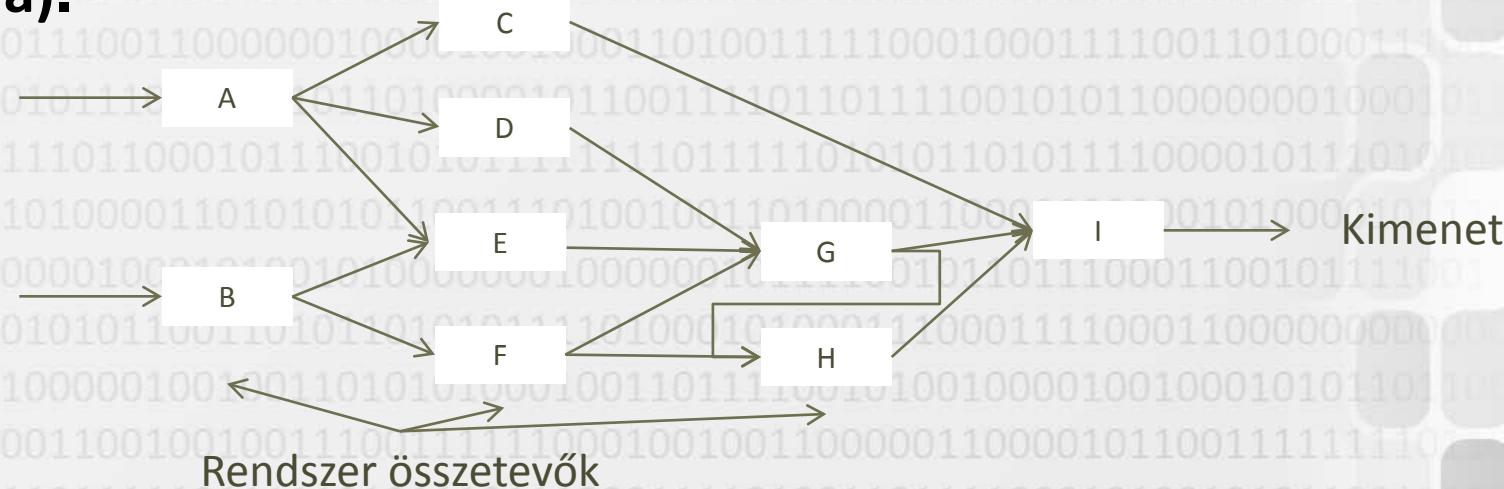
Spekulatív dekompozíció

- Akkor használandó, amikor a következő lépés sok közül az egyik lesz. Hogy melyik, csak akkor határozható meg, amikor az aktuális részfeladat lefutott.
- Feltételezi az aktuális feladat valamilyen kimenetelét, és előre futtat néhány rákövetkező lépést.
 - Mint a mikroprocesszor szinten a spekulatív futtatás.
- Két megközelítés:
 - konzervatív: csak akkor határoz meg feladatokat, ha azok már biztosan függetlenek;
 - optimista: akkor is ütemez feladatot, ha potenciálisan téves lehet.
- A konzervatív megközelítés kisebb párhuzamosságot eredményez; az optimista megközelítés hiba esetén roll-back mechanizmust igényel.

Spekulatív dekompozíció - példa

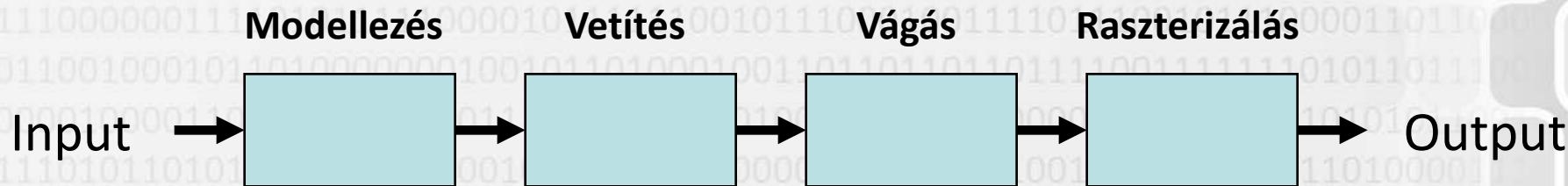
Diszkrét események szimulációja

- Idő szerint rendezett eseménylista a központi adatstruktúra.
- Az események idő szerinti sorrendben játszódnak, feldolgozásra kerülnek, és ha szükséges, akkor az eredmény események beillesztésre kerülnek az eseménylistába.
- Csak a spekuláció révén párhuzamosítható.
- Állapot-visszaállítási extra feladatot követel (számítás és memória).

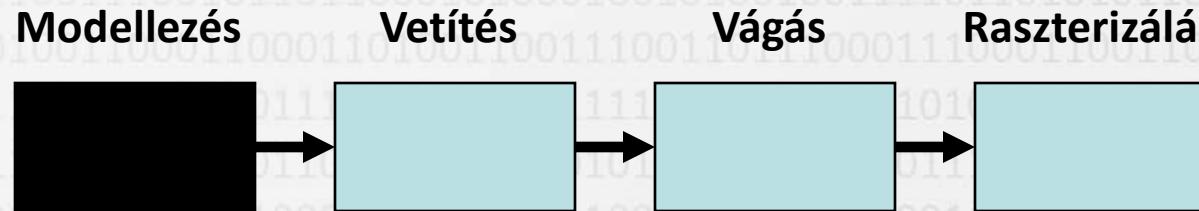


Futószalag dekompozíció

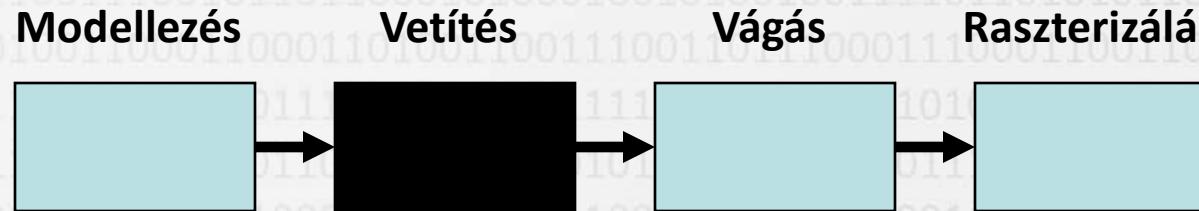
- Speciális feladatdekompozíció
- „Futószalag” párhuzamosság
- Példa: 3D renderelés grafikában



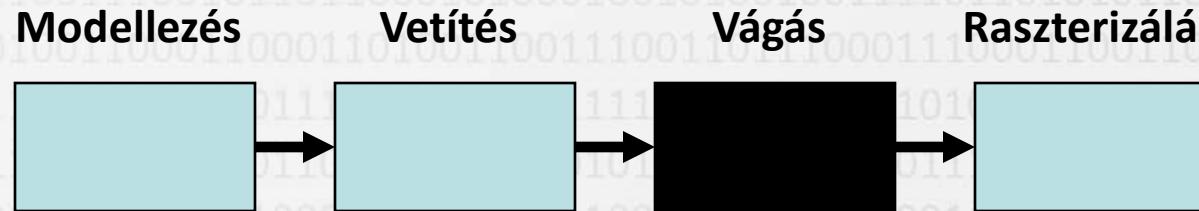
Egy adathalmaz feldolgozása (1. lépés)



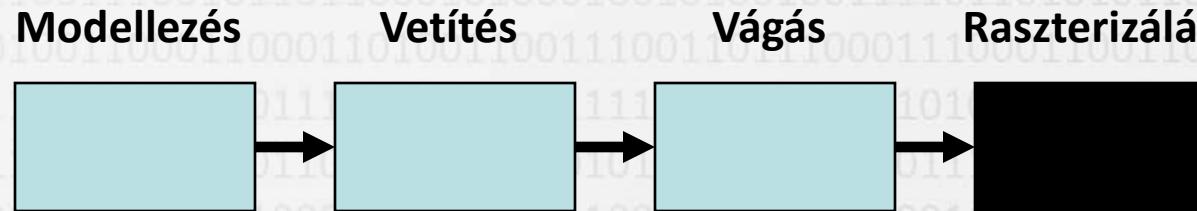
Egy adathalmaz feldolgozása (2. lépés)



Egy adathalmaz feldolgozása (3. lépés)

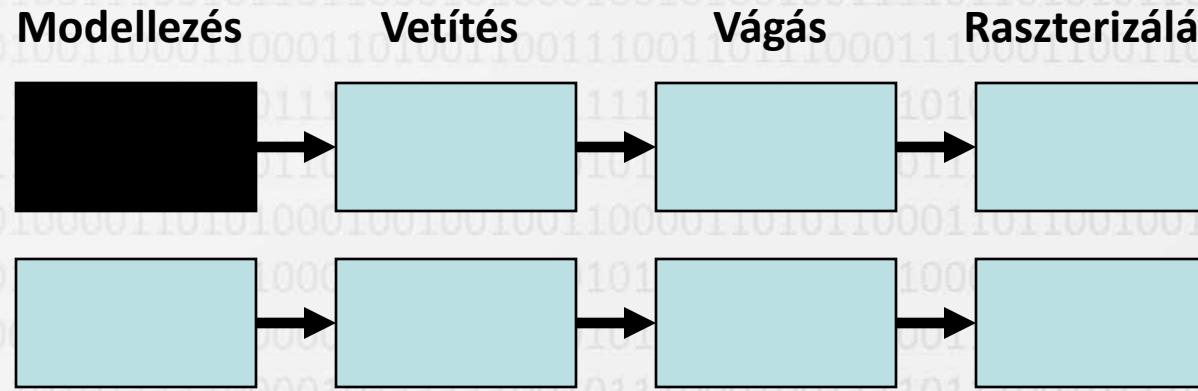


Egy adathalmaz feldolgozása (4. lépés)

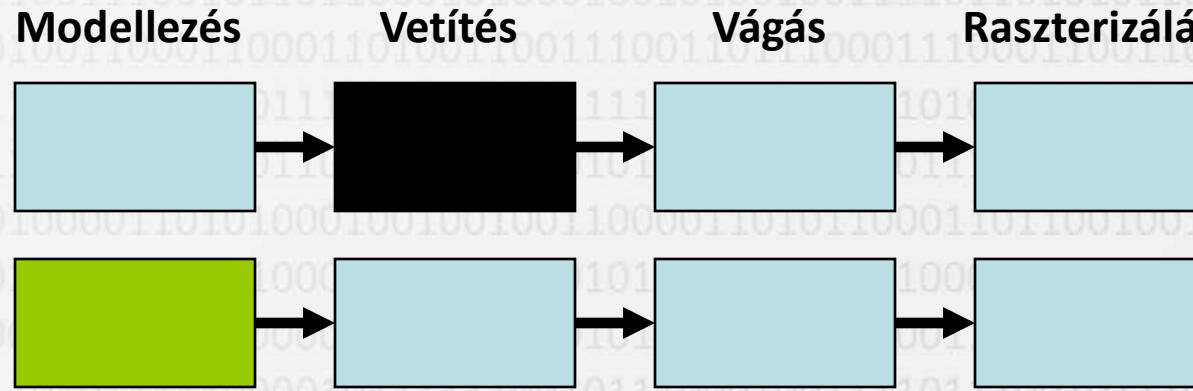


A futószalag 1 adathalmazt 4 lépésekben dolgoz fel

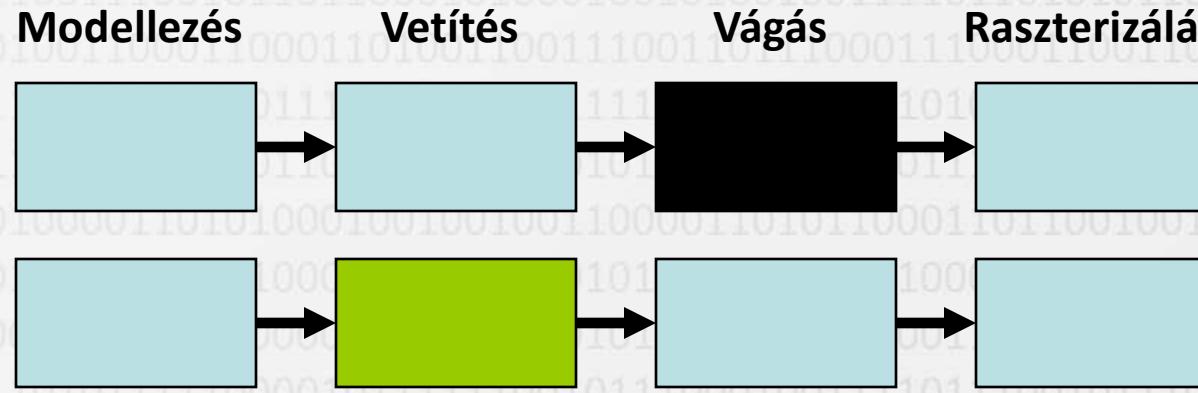
Két adathalmaz feldolgozása (1. lépés)



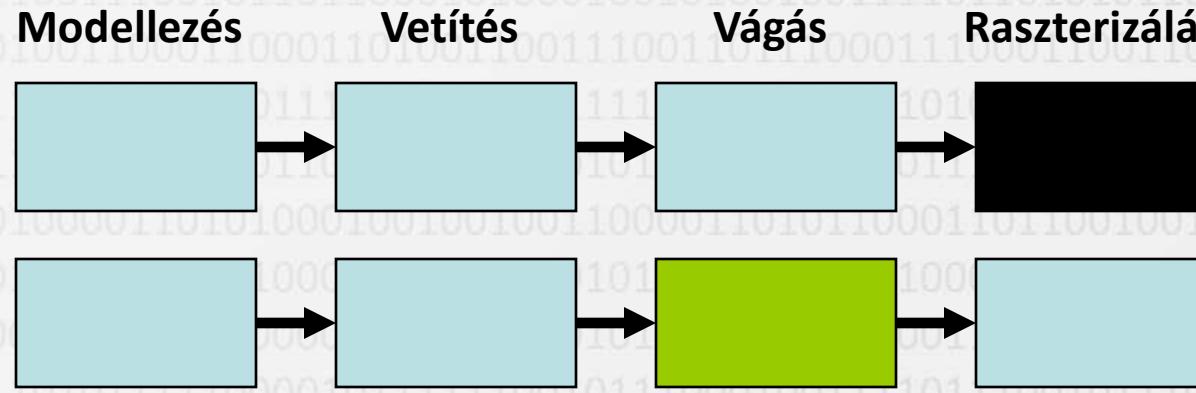
Két adathalmaz feldolgozása (2. időpont)



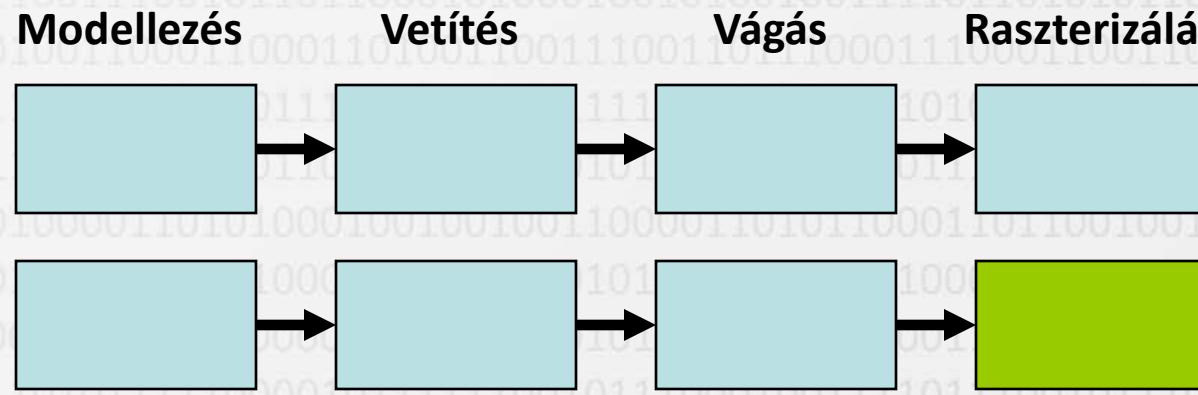
Két adathalmaz feldolgozása (3. lépés)



Két adathalmaz feldolgozása (4. lépés)



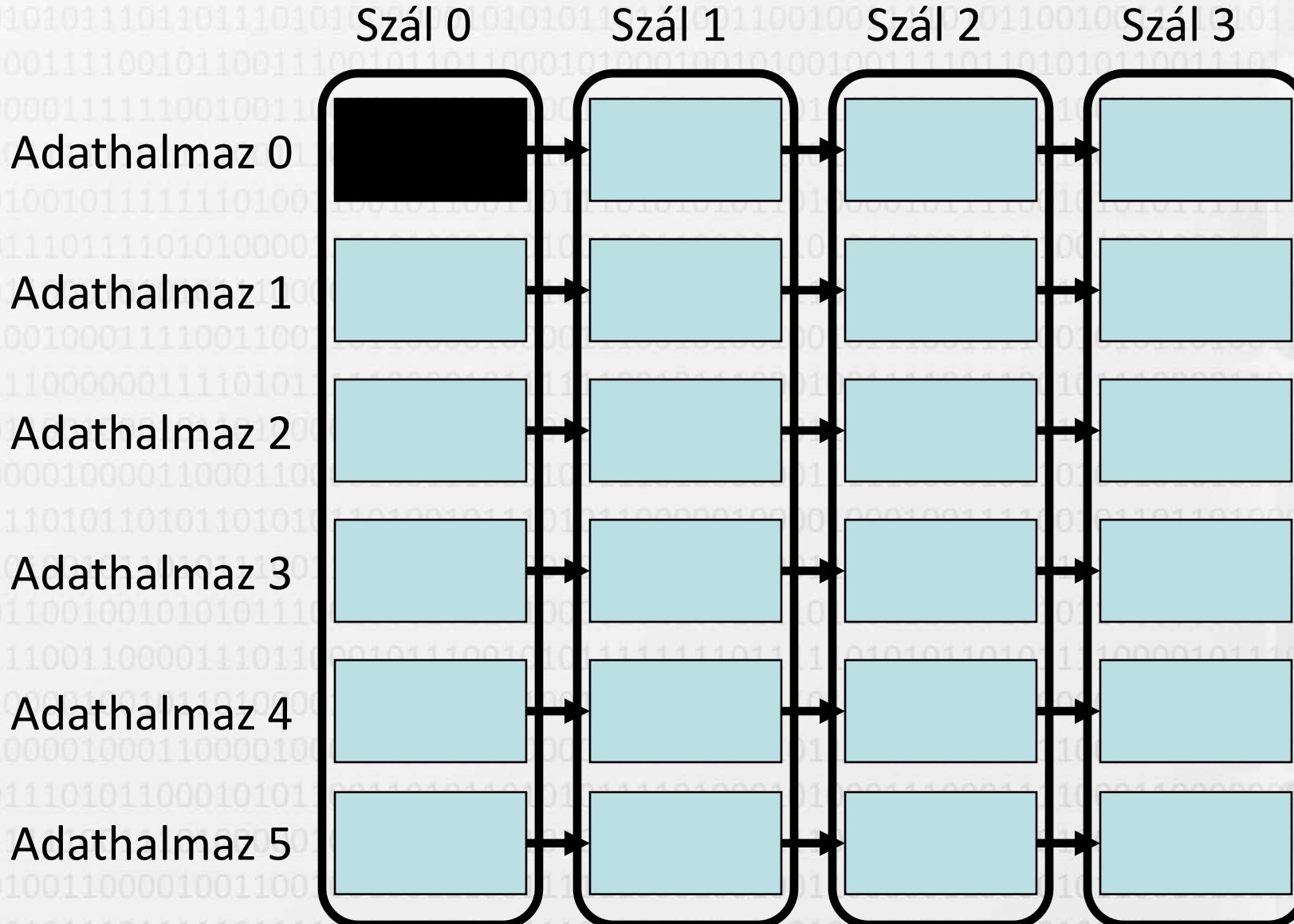
Két adathalmaz feldolgozása (5. lépés)



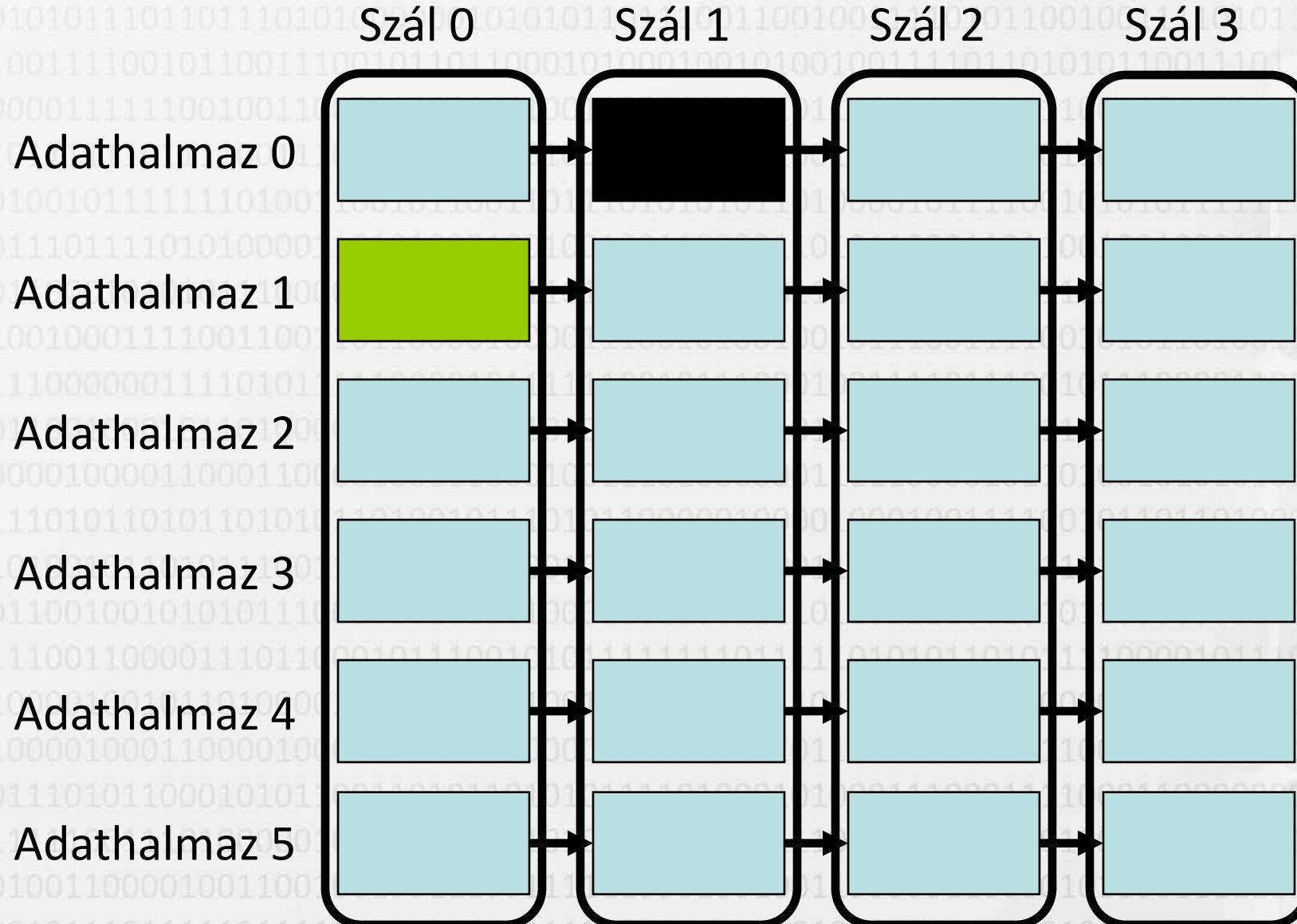
A futószalag 2 adathalmazt 5 lépésekben dolgoz fel



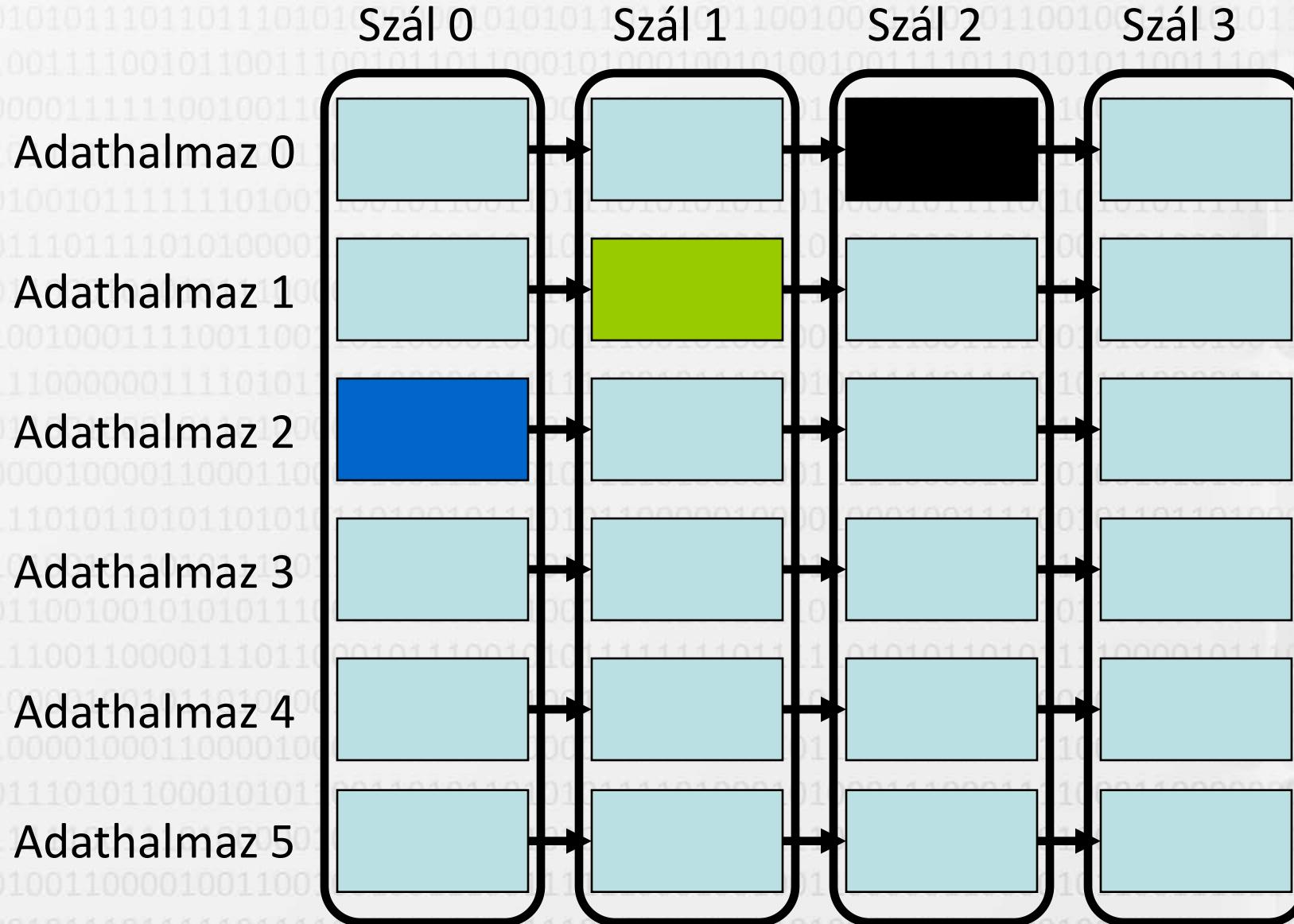
Hat adathalmaz feldolgozása (1. lépés)



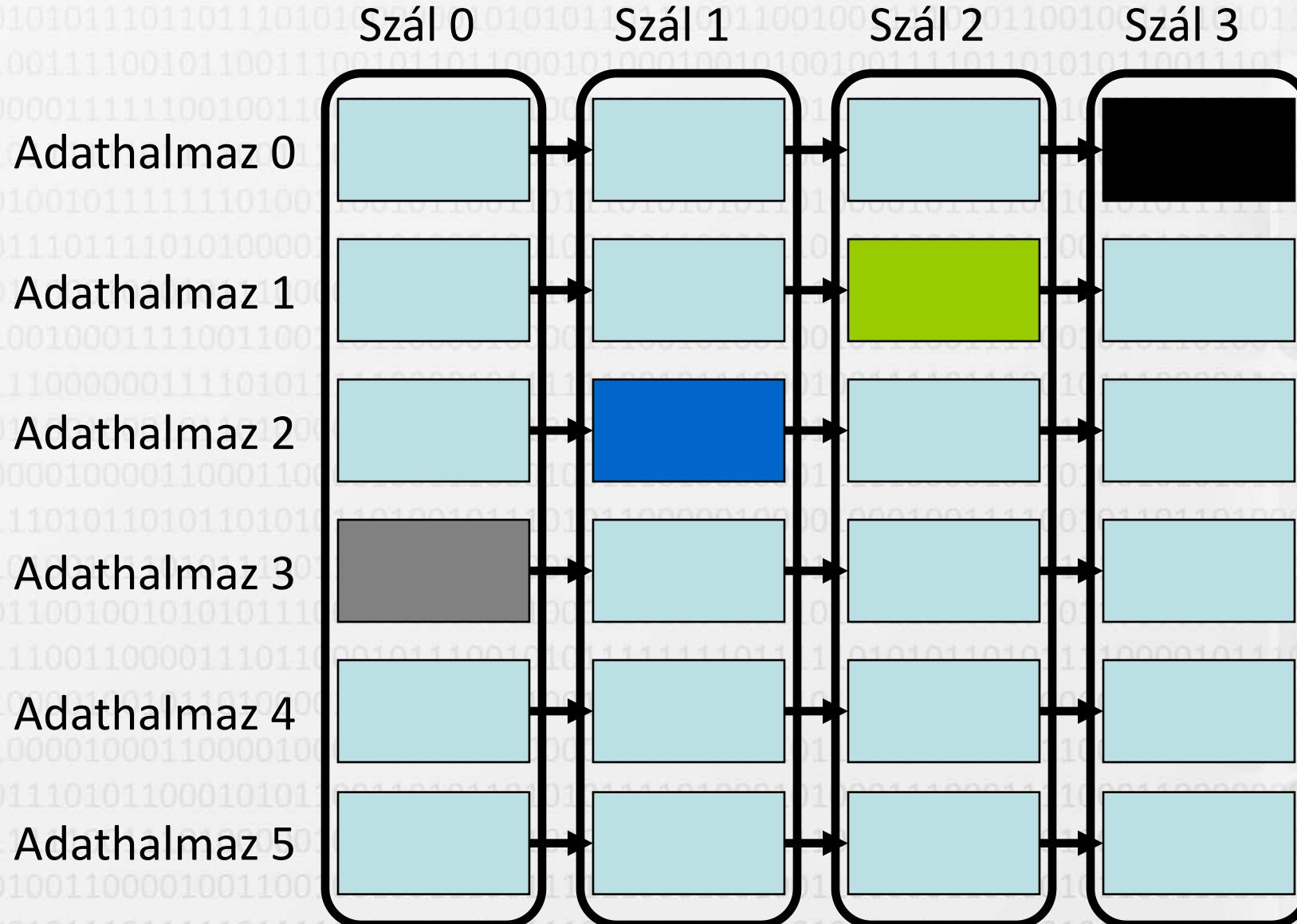
Hat adathalmaz feldolgozása (2. lépés)



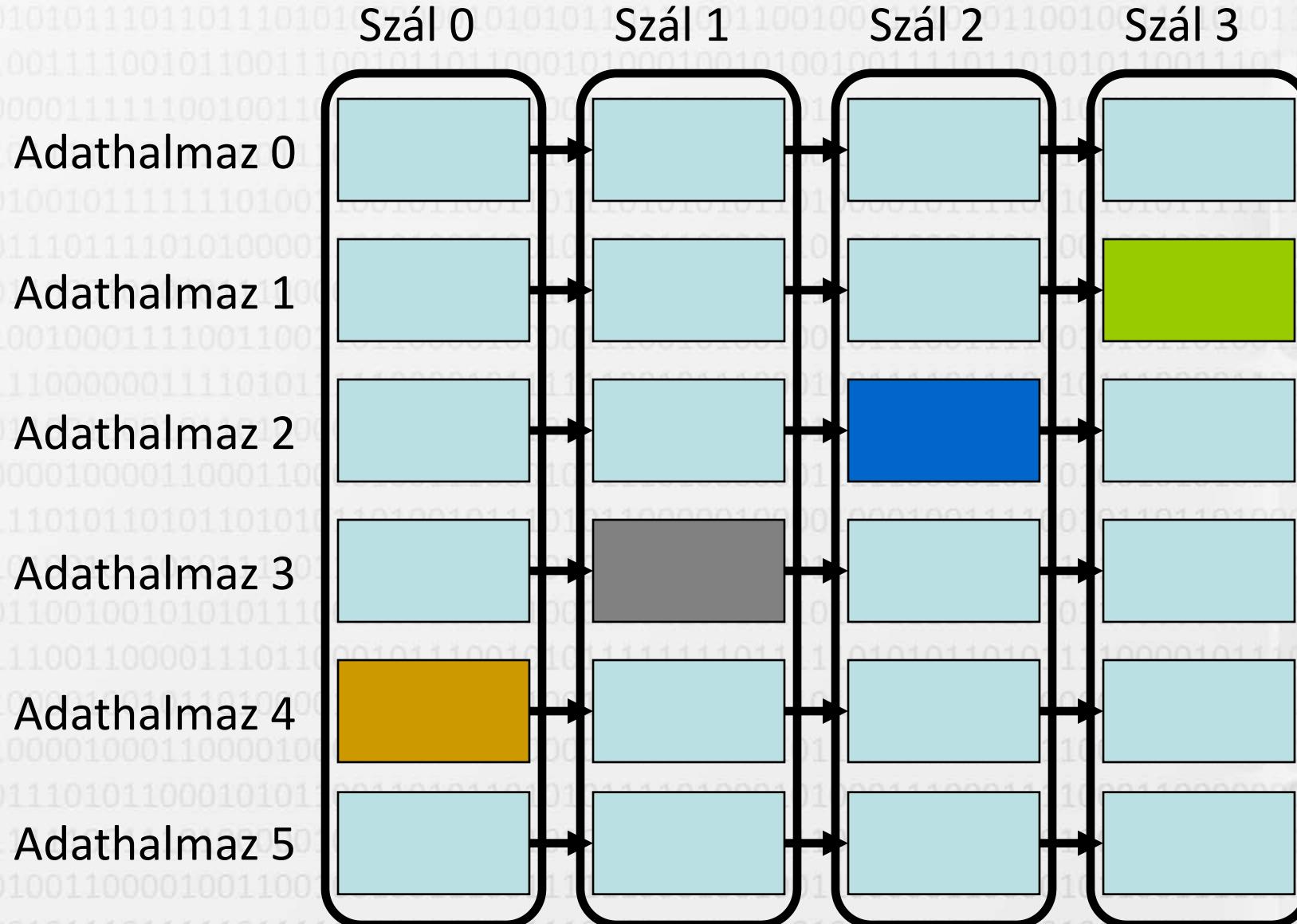
Hat adathalmaz feldolgozása (3. lépés)



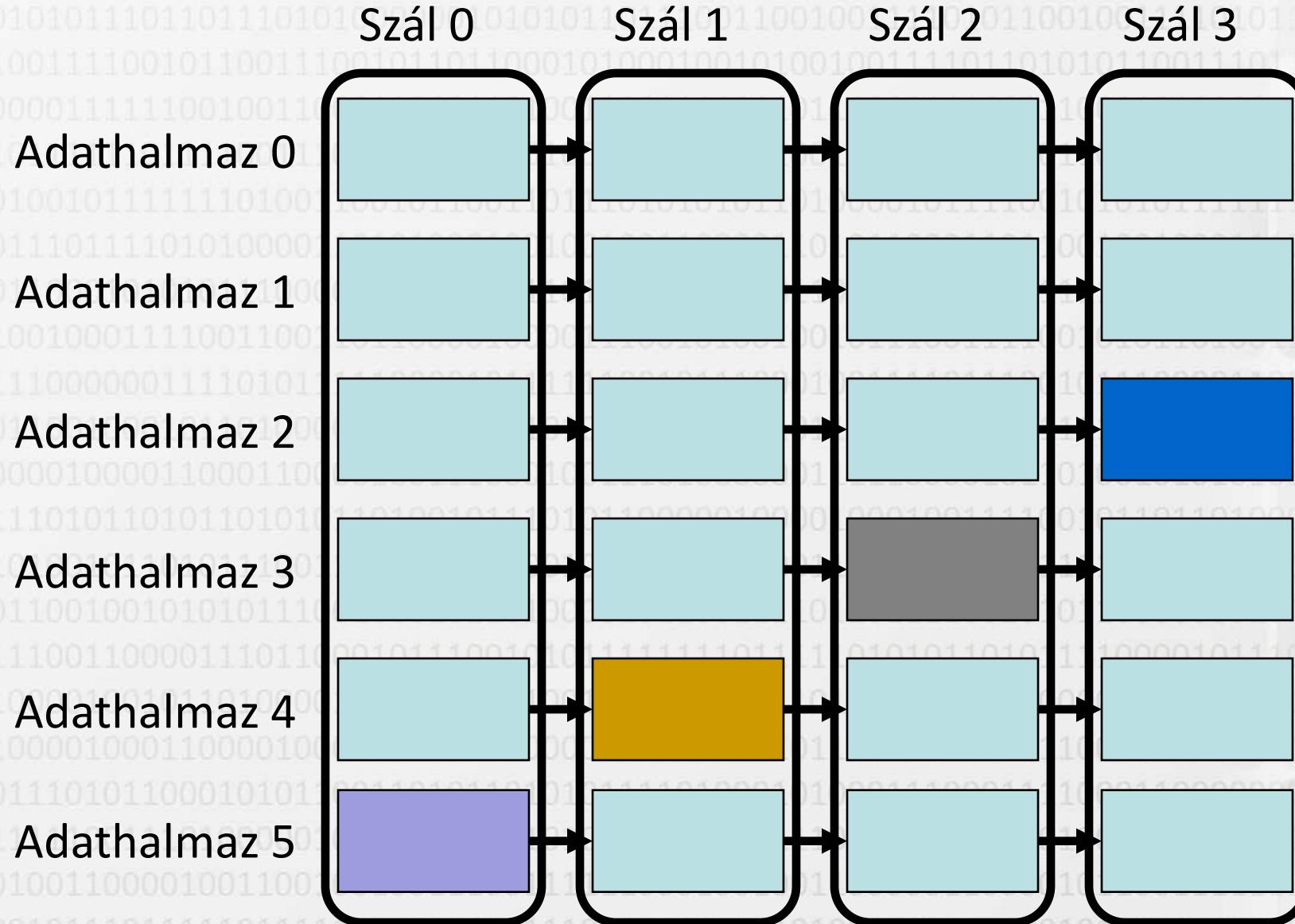
Hat adathalmaz feldolgozása (4. lépés)



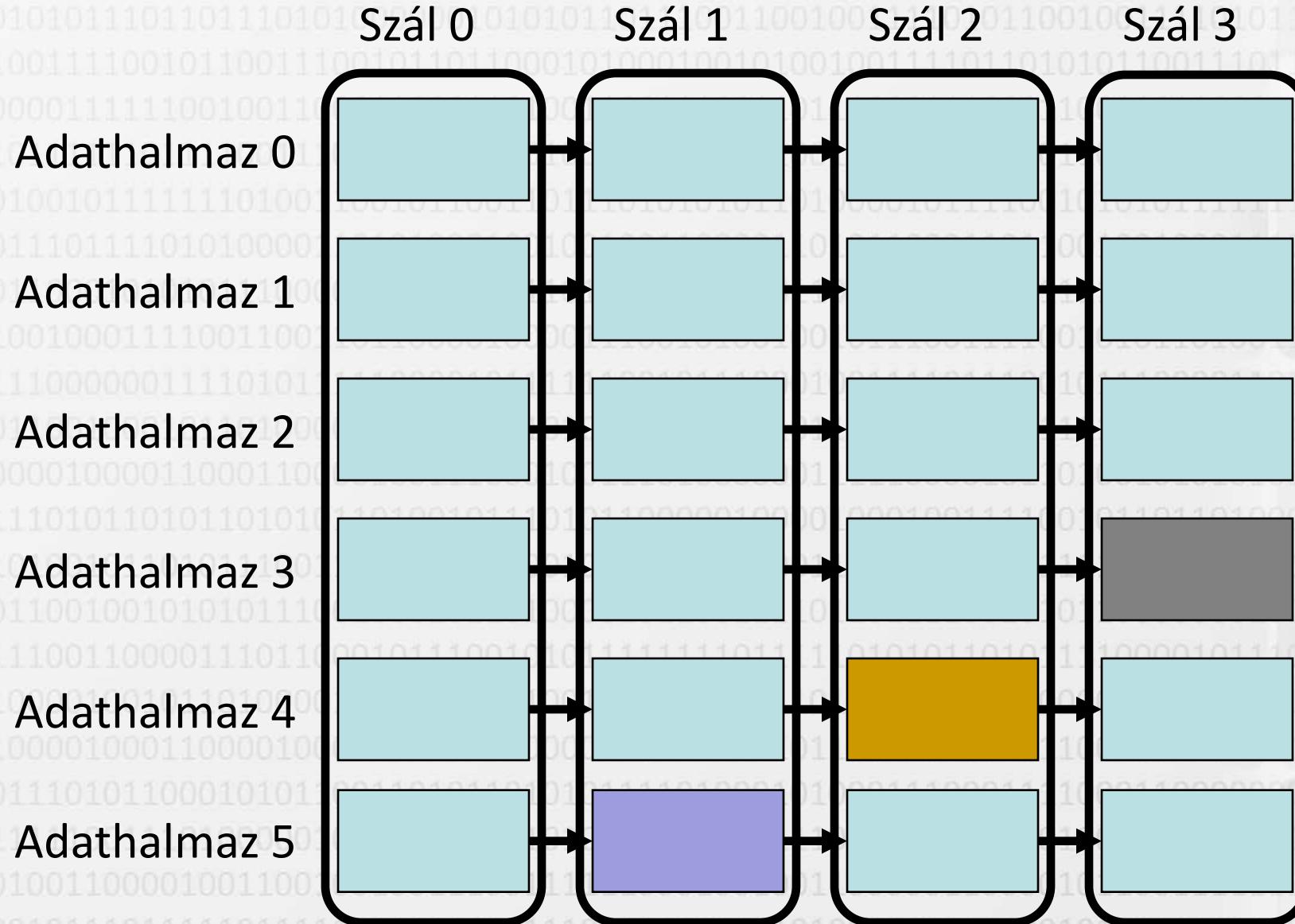
Hat adathalmaz feldolgozása (5. lépés)



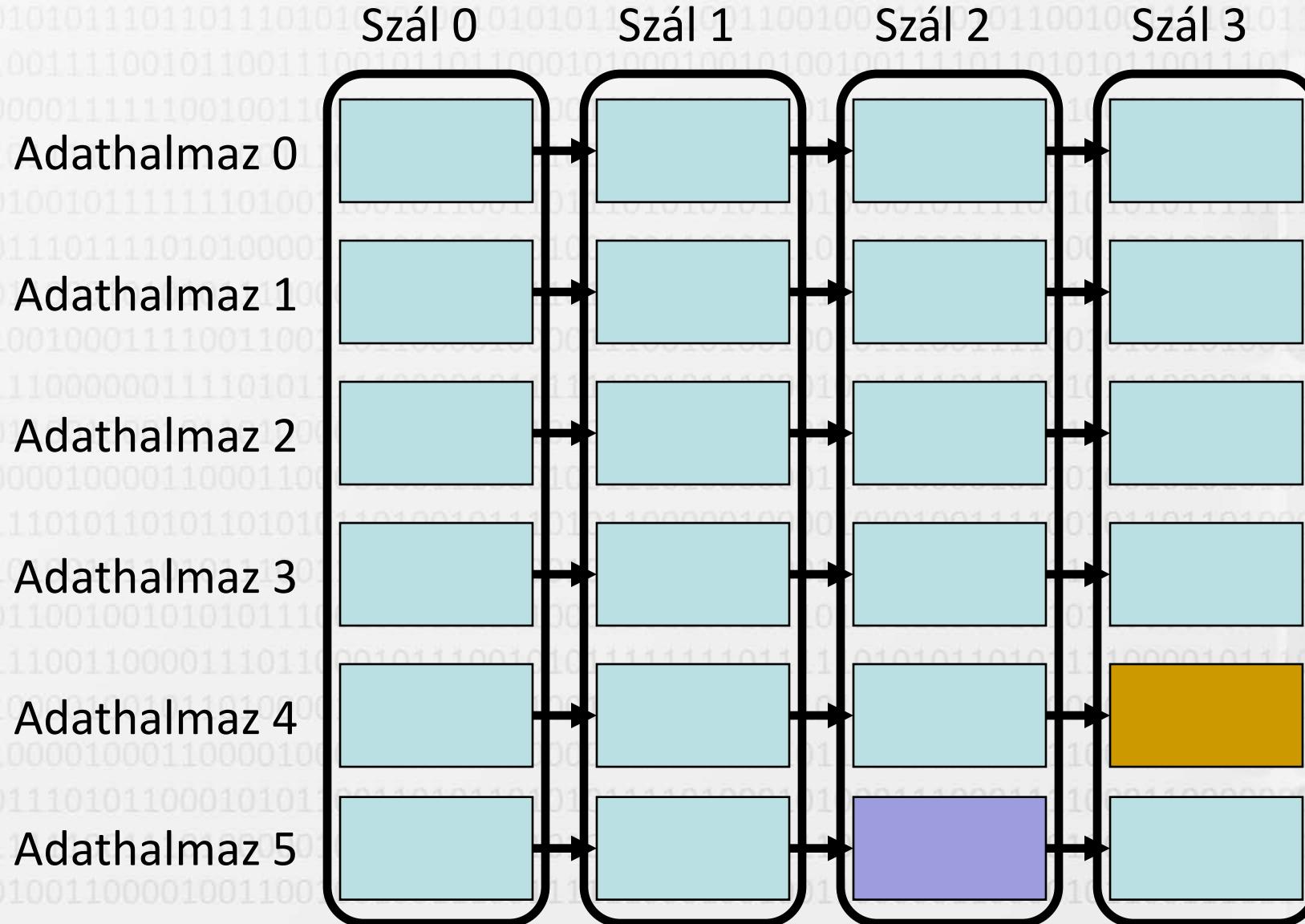
Hat adathalmaz feldolgozása (6. lépés)



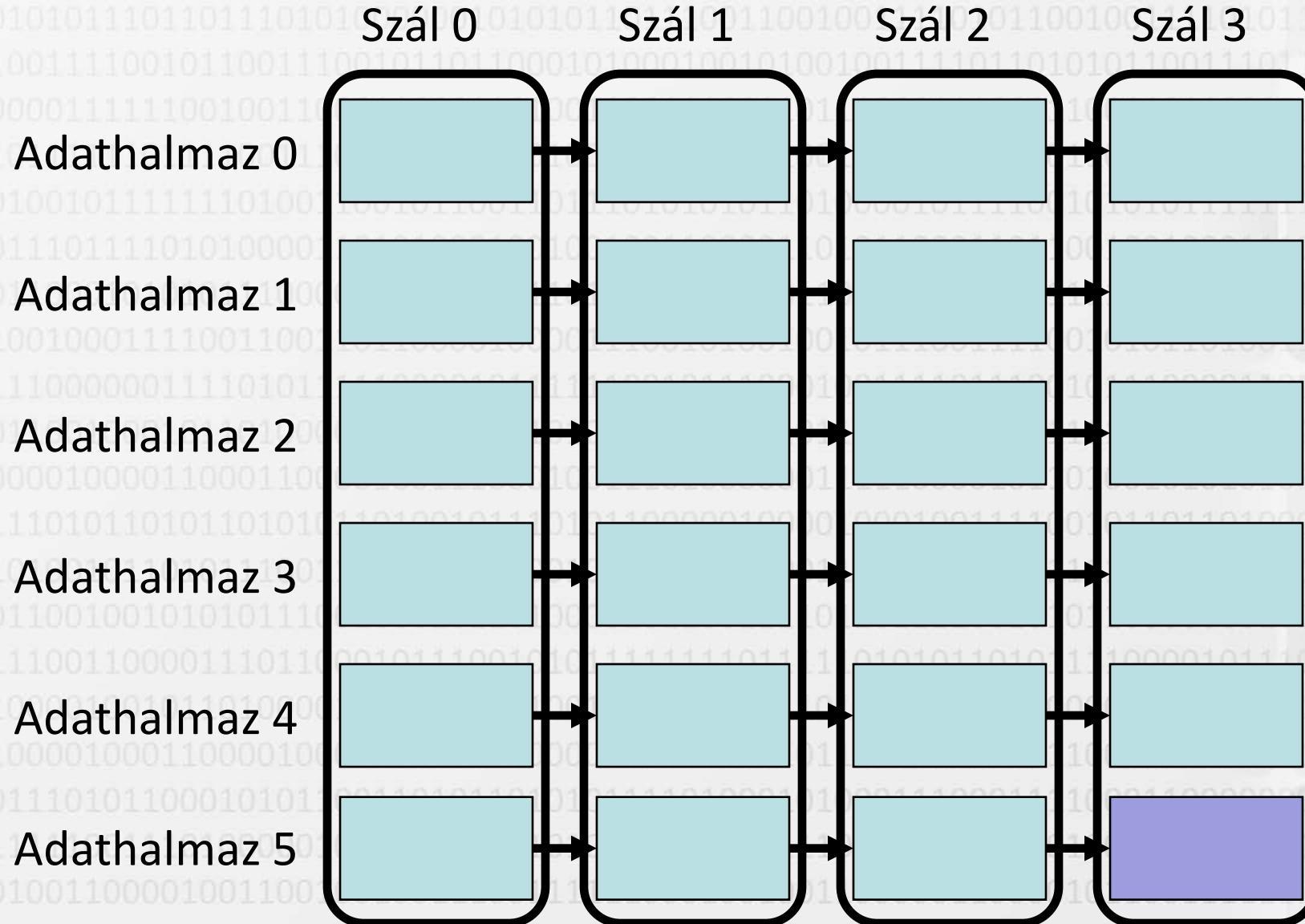
Hat adathalmaz feldolgozása (7. lépés)



Hat adathalmaz feldolgozása (8. lépés)



Hat adathalmaz feldolgozása (9. lépés)

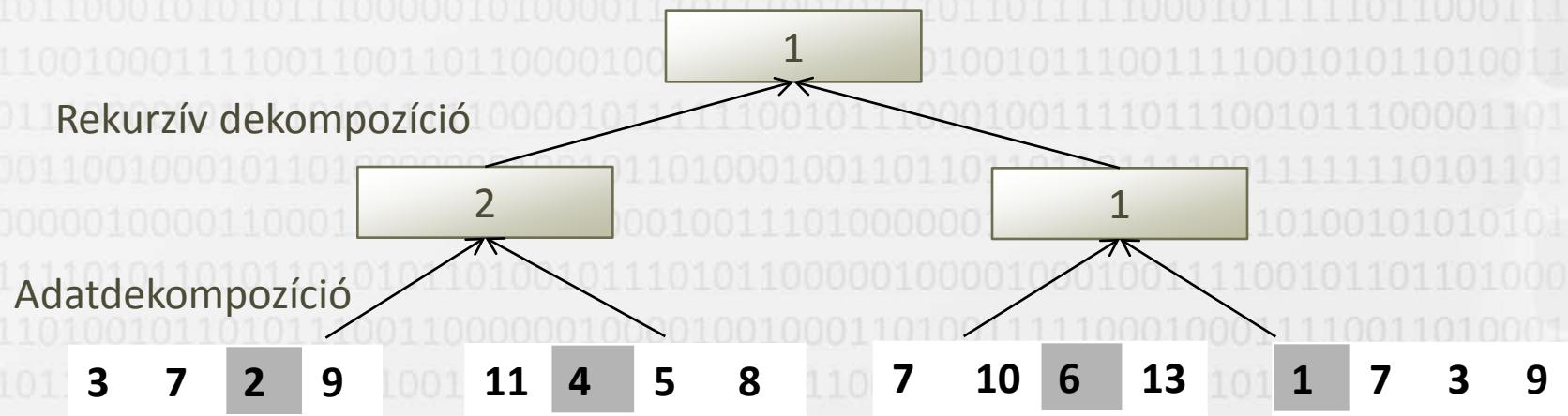


Futószalag feladat

- **Hányszorosára gyorsult a feladatmegoldás 6 adathalmaz esetén?**
- **Hányszorosára gyorsul a feladatmegoldás végtelen sok adathalmaz esetén?**

Hibrid dekompozíció

- Gyakran előnyös a dekompozíciós technikák kombinált használata.
 - Példa: minimumkeresés.



Leképzés

Leképzési technikák

Processzek és leképezés

- Általában a dekompozíció során meghatározott részfeladatok száma meghaladja az elérhető számítási egységek számát.
- A részfeladatokat processzekre képezzük le.

Megjegyzés: nem processzorokra történő leképzésről beszélünk, hanem processzekre, mert az API-k tipikusan ezt biztosítják. A részfeladatokat összegyűjtjük (aggregáció) processzekbe, és a rendszer fogja a fizikai processzorokra terhelni. A processzt feldolgozó entitásként (összegyűjtött részfeladatok és adatok összessége) használjuk, és nem folyamatként.

Processzek és leképezés

- A párhuzamos teljesítmény szempontjából kritikus a részfeladatok megfelelő leképzése processzekre.
- A leképzést a feladatfüggőségi-gráf és a feladatkapcsolati-gráf határozza meg.
- A függőségi-gráf használható a processzek közötti munka – bármely időpontban – egyenletes terítésére (minimális várakozás és optimális terheléskiegyenlítés).
- A kölcsönhatási-gráf biztosíthatja, hogy a processzek minimális kapcsolatban legyenek egymással (kommunikáció minimalizálás).

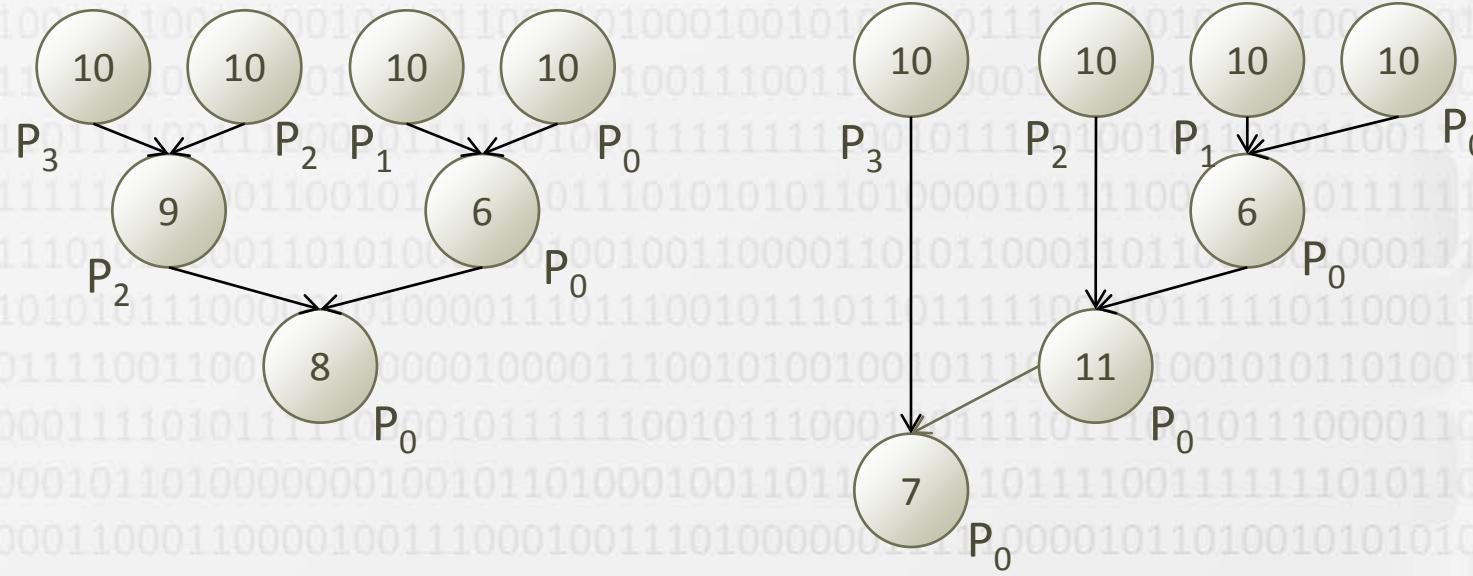
Processzek és leképezés

Cél: minimális párhuzamos futási idő megfelelő leképzéssel:

- független részfeladatok különböző processzekre leképzése;
- a kritikus úton lévő részfeladatok processzekhez rendelése ASAP;
- a processzek közötti interaktivitás minimalizálása úgy, hogy a sokat kommunikáló részfeladatokat ugyanahhoz a processzhez rendeljük.

Megjegyzés: egymásnak ellentmondó kritériumok.

Processzek és leképzés - példa



- A függőségi gráf „szintjei” alapján lehet meghatározni a leképzést: egy szinten lévő részfeladatokat különböző processzekhez kell rendelni.

A részfeladatok leképzése

- **Miért kell körültekintően leképezni a részfeladatokat?**
 - Véletlenszerűen hozzárendelhetjük a processzorokhoz?
- **Helyes leképzés kritikus lehet, mivel minimalizálni kell a párhuzamos feldolgozás miatti extra ráfordítást.**
 - Ha T_p a párhuzamos futásideje p processzoron és T_s a sorosé, akkor a teljes extra ráfordítási idő (*total overhead*) $T_o = p * T_p - T_s$
 - A párhuzamos rendszer munkája több, mint a sorosé.
 - Az extra ráfordítás forrásai:
 - terhelés egyenetlenség;
 - processzek közötti kommunikáció (Inter-process communication: IPC)
 - Koordináció / szinkronizáció / adatmegosztás.

Miért lehet összetett a leképzés?

A feladatfüggőségi és a feladatkölcsönhatási-gráfot figyelembe kell venni a leképzésnél.

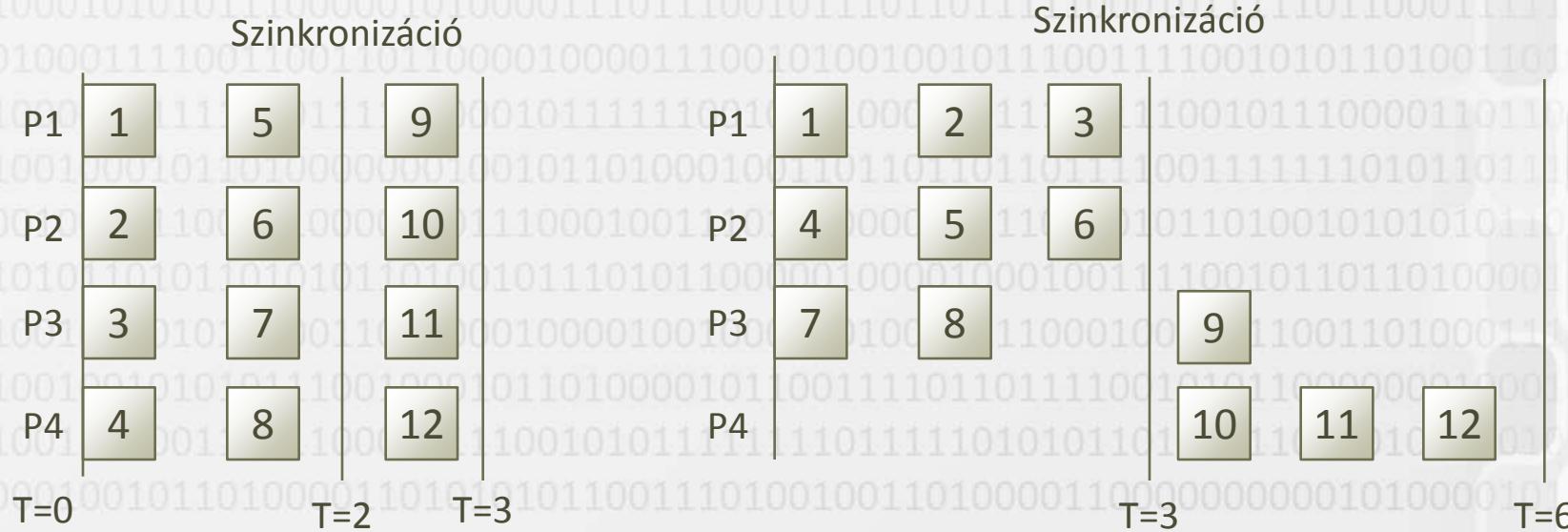
- **Ismertek a részfeladatok előre?**
 - Statikus vs. dinamikus részfeladat generálás.
- **Milyenek a számítási követelmények?**
 - Egyformák vagy nem egyformák?
 - Ismerjük előre?
- **Mennyi adat kapcsolódik az egyes részfeladatokhoz?**
- **Milyen a kapcsolat a részfeladatok között?**
 - Statikus vagy dinamikus?
 - Ismerjük előre?
 - Adatfüggők?
 - Szabályosak vagy szabálytalanok?
 - Csak írnak, vagy írnak és olvasnak?
- **A fenti jellemzőktől függően különböző leképzési technikák szükségesek, és ezek eltérő komplexitásúak és költségűek.**

Feladatfüggőségi gráf

Feladat-kölcsönhatási gráf

Leképzési esetek terheléskiegyenlítés céljal - példa

- **1-8 után szinkronizáció szükséges.**
- **Azonos terhelés, de nem azonos várakozás.**



Terheléskiegyenlítéses technikák

Leképzési technikák lehetnek statikusak vagy dinamikusak.

- **Statikus leképzés:** a részfeladatok processzekre történő leképzésre előre meghatározott.
 - Ehhez pontos ismeret szükséges minden részfeladat méretéről. De ekkor is NP-teljes feladat a leképzés.
- **Dinamikus leképzés:** futási időben történik a részfeladatok processzekhez rendelése.
 - Futási időben generálódó részfeladatok esetén.
 - Előre nem ismert számítási igény.

Statikus leképzési technikák

- **Adatparticionáláson alapuló módszerek**
- **Gráf particionáláson alapuló leképzések**
- **Hibrid módszerek**

Statikus leképzés – tömbszétosztás

A feladatok és processzek növekvően címkézettek.

- **Blokkleképzés:** n/p darab egymás utáni részfeladat képződik le az egymás utáni processzekre.
- **Ciklikus leképezés:** az i feladat az $(i \bmod p)$ processzre kerül.
- **Tükrözött leképzés:** mint a ciklikus, de a részfeladatok fordított sorrendben kerülnek ki.
- **Blokk-ciklikus és blokk-tükrözött leképzés:** a részfeladatok blokkjai kerülnek hozzárendelésre.

Magasabb dimenzióban is alkalmazható módszerek: külön minden dimenzióban.

Statikus leképzés – tömbszétosztás

Blokk



Ciklikus



Tükörzés



Blokk-ciklikus



Tömbszétosztás - példa

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

(a)

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

(b)

Sűrű mátrix - példa

$$\begin{matrix} A & & B \\ \begin{array}{|c|c|c|}\hline & & \\ \hline \end{array} & \times & \begin{array}{|c|c|c|}\hline & & \\ \hline \end{array} & = & \begin{matrix} C \\ P_0 \\ P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \\ P_7 \\ P_8 \\ P_9 \\ P_{10} \\ P_{11} \\ P_{12} \\ P_{13} \\ P_{14} \\ P_{15} \end{matrix} \end{matrix}$$

(a)

$$\begin{matrix} A & & B & & C \\ \begin{array}{|c|c|c|}\hline & & \\ \hline \end{array} & \times & \begin{array}{|c|c|c|}\hline & & \\ \hline & | & | \\ \hline & & \\ \hline & | & | \\ \hline & & \\ \hline & | & | \\ \hline & & \\ \hline \end{array} & = & \begin{matrix} P_0 & P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} \\ P_{12} & P_{13} & P_{14} & P_{15} \end{matrix} \end{matrix}$$

(b)

Ciklikus és blokk-ciklikus szétosztás

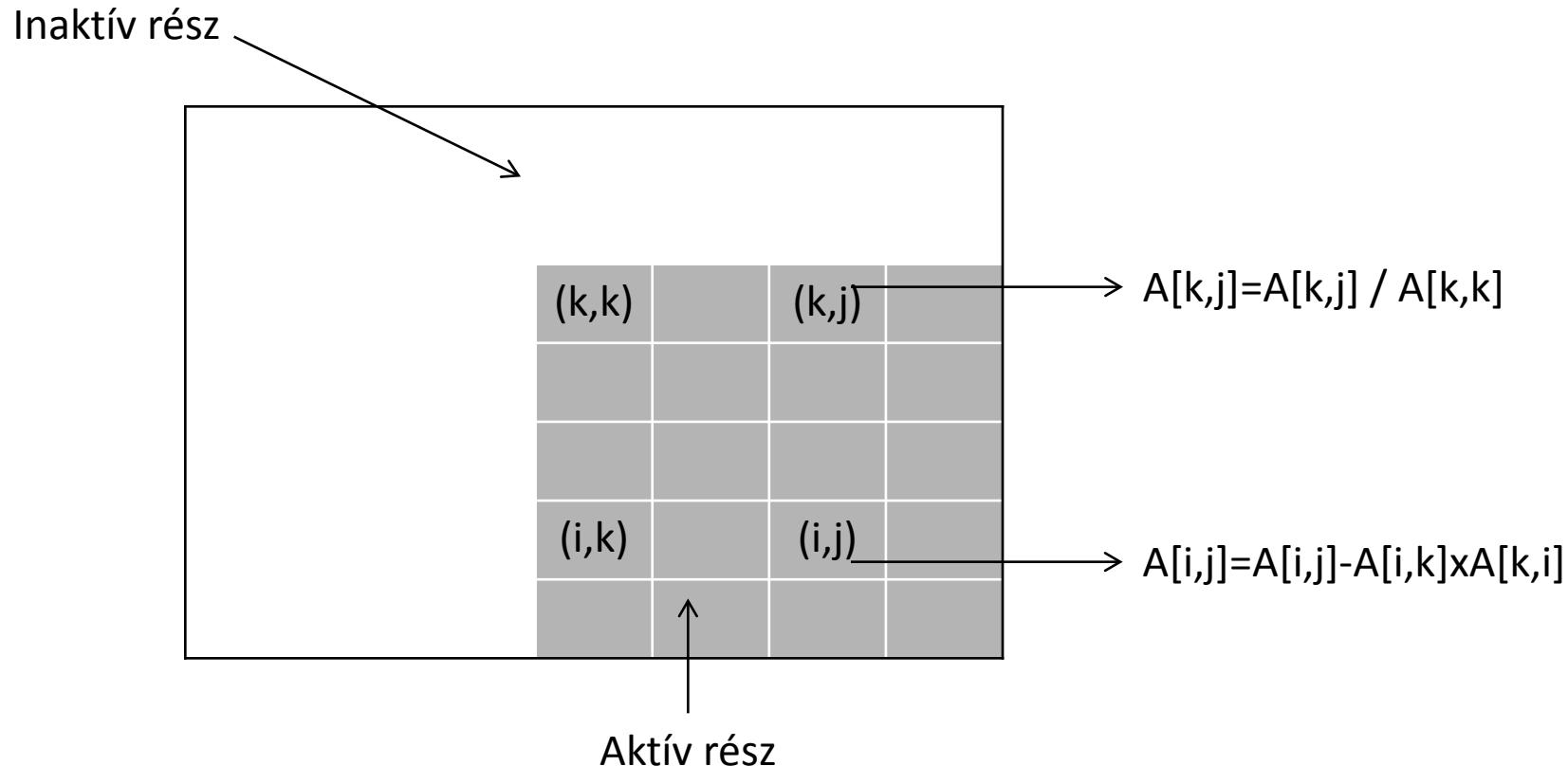
- Ha az adatelemekhez kapcsolódó számítási igény változik, a blokkleképzéses módszer terhelésegyenlenséghez vezet.
- Példa: sűrű mátrix Gauss-elimináció.

Blokk-ciklikus szétosztás

- **Blokk-ciklikus szétosztás**
 - A blokkszétosztásos módszer variációja, amellyel csökkenteni lehet a terhelésegyenetlenséget és az üresjárat problémáját.
- **Az elérhető processzek számánál sokkal több részre osztjuk a tömböt.**
- **A blokkokat körbeforgó módon rendeljük a processzekhez úgy, hogy minden processz különböző, nem szomszédos blokkot kap.**

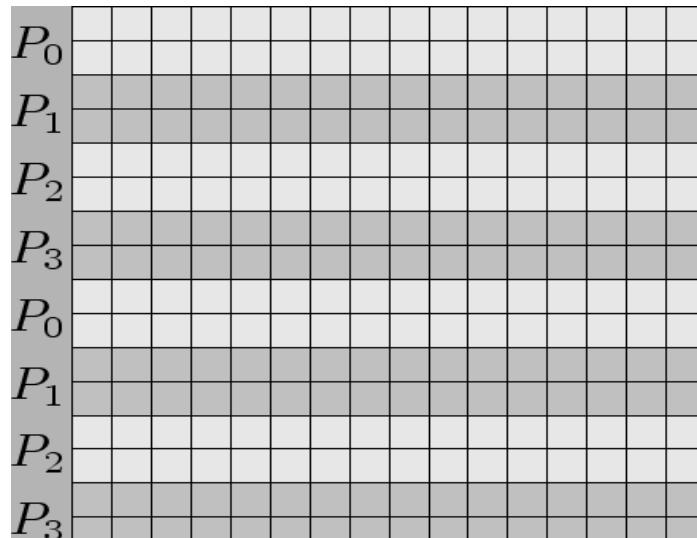
Blokk-ciklikus leképzés – Gauss-elimináció - példa

Minden processz a mátrix különböző részeiből is kap részfeladatokat.

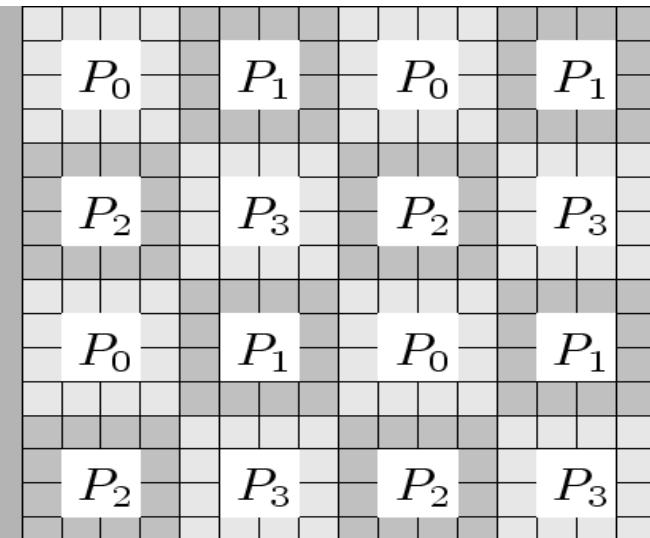


Blokk-ciklikus leképzés – Gauss-elimináció - példa

- A blokkméret n/p , ahol n a mátrix mérete és p a processzek száma.



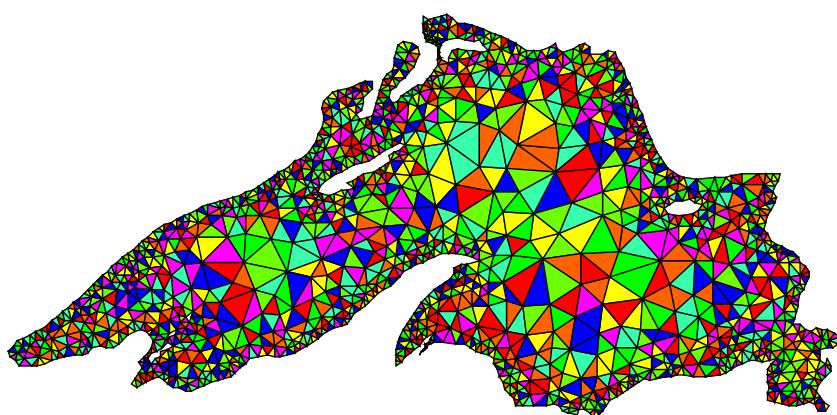
(a)



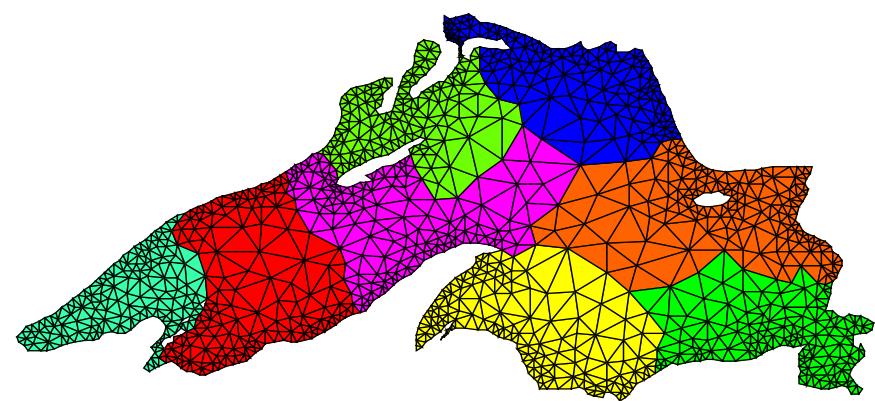
(b)

Gráf-particionáláson alapuló leképzés

- A feladatkapcsolati-gráf partitionálásával
 - Azonos mennyiségű feladat
 - Minimális számú élvágás
 - NP-teljes probléma, heurisztika



Véletlen particionálás



Minimális élvágás szerinti particionálás

Dinamikus terheléskiegyenlítés

- **Dinamikus leképzés esetén az elsődleges cél a terheléskiegyenlítés.**
- **Centralizált sémák**
 - Egy processz felelős a feladatok kiadásáért:
 - „mester-szolga” paradigma.
 - Kérdés:
 - részfeladatok szemcsézettsége.
- **Szétosztott sémák**
 - A munka bármely processz-pár között szétosztható (küldő-fogadó szerep).
 - Kérdés
 - Hogyan párosíthatók a processzek?
 - Ki inicializálja a munka kiosztást?
 - Mennyi munka kerül kiosztásra?

Kereső algoritmusok a diszkrét optimalizálás problémájához

A diszkrét optimalizálási probléma

Soros megoldás

Párhuzamos megoldási lehetőségek és problémák

www.tankonyvtar.hu

A. Grama, A. Gupta, G. Karypis és
V. Kumar: „Introduction to Parallel Computing”, Addison Wesley, 2003.
könyv anyaga alapján

A kereső eljárások pontosabb tárgyalása Futó Iván (szerk.):
Mesterséges Intelligencia, Aula, 1999. fejezetéből származnak

Diszkrét optimalizálás

- A diszkrét optimalizálási problémák komoly számítási igényűek. Jelentős elmélettel és gyakorlati fontossággal rendelkeznek (ütemezési feladatok, pályatervezés, digitális áramkörökhoz tesztpéldák generálása, logisztikai feladatok stb.).
- A kereső algoritmusok a feltételeknek megfelelő megoldások terét szisztematikusan vizsgálják.

Definíció

- A **diszkrét optimalizációs probléma (DOP)** a következő módon fejezhető ki: (S, f) .
 - S bizonyos feltételeknek eleget tévő megoldások véges vagy megszámlálhatóan végtelen halmaza;
 - f költségfüggvény S minden eleméhez valós R értéket rendel.
- A DOP megoldása olyan x_{opt} megoldás keresése, hogy $f(x_{opt}) \leq f(x)$ minden $x \in S$ esetén.
- **Gyakran NP-teljes probléma.**

Diszkrét optimalizáció - példa

- *Egyértékű lineáris programozási feladat* (0/1 integer-linear-programming problem):
adott A $m \times n$ mátrix, b $m \times 1$ vektor és c $n \times 1$ vektor.
- A cél egy olyan x $n \times 1$ vektor meghatározása, amelynek elemei csak 0 és 1 értékekből állhatnak,
a vektor teljesíti a következő feltételt: $Ax \geq b$,
és az f függvénynek minimuma kell legyen: $f(x) = c^T x$

Duális megfogalmazás

DOP és a gráfkeresés

- Az S lehetséges elemeinek száma nagyon nagy.
- A DOP gyakran úgy is megfogalmazható, hogy egy gráfban minimum költségű utat keressünk egy kezdőponttól egy vagy több lehetséges célcsomópontig.
- Az S minden x elemét tekinthetjük egy útnak a kezdőponttól valamely célig.
 - A gráf csomópontjai állapotok.
 - Az állapotok vagy végpontok, vagy nem végpontok.
 - Bizonyos állapotok lehetséges megoldáshoz tartoznak, mások nem.
 - Az élekhez költségek tartoznak, amelyek az egyik állapotból a másikba történő átmenet ráfordításai.
- A gráfot állapottérnek nevezük.

Állapottér gráf - példa

- Egyértékű lineáris programozási feladat

$$A = \begin{bmatrix} 5 & 2 & 1 & 2 \\ 1 & -1 & -1 & 2 \\ 3 & 1 & 1 & 3 \end{bmatrix} \quad b = \begin{bmatrix} 8 \\ 2 \\ 5 \end{bmatrix} \quad c = \begin{bmatrix} 2 \\ 1 \\ -1 \\ -2 \end{bmatrix}$$

- Az A , b és c értékeihez kapcsolódó feltételek:

$$5x_1 + 2x_2 + x_3 + 2x_4 \geq 8$$

$$x_1 - x_2 - x_3 + 2x_4 \geq 2$$

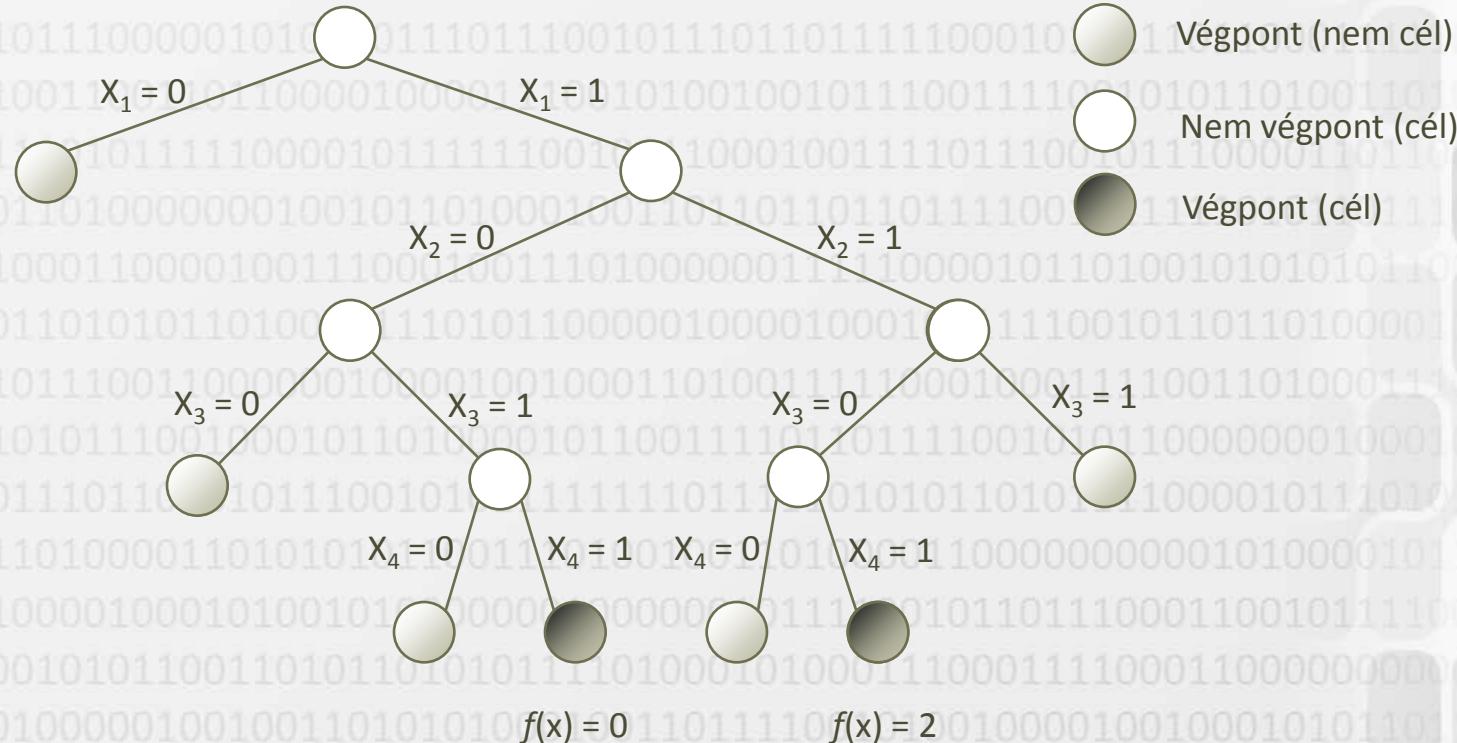
$$3x_1 + x_2 + x_3 + 3x_4 \geq 5$$

- A minimalizálandó f függvény pedig:

$$f(x) = 2x_1 + x_2 - x_3 - 2x_4$$

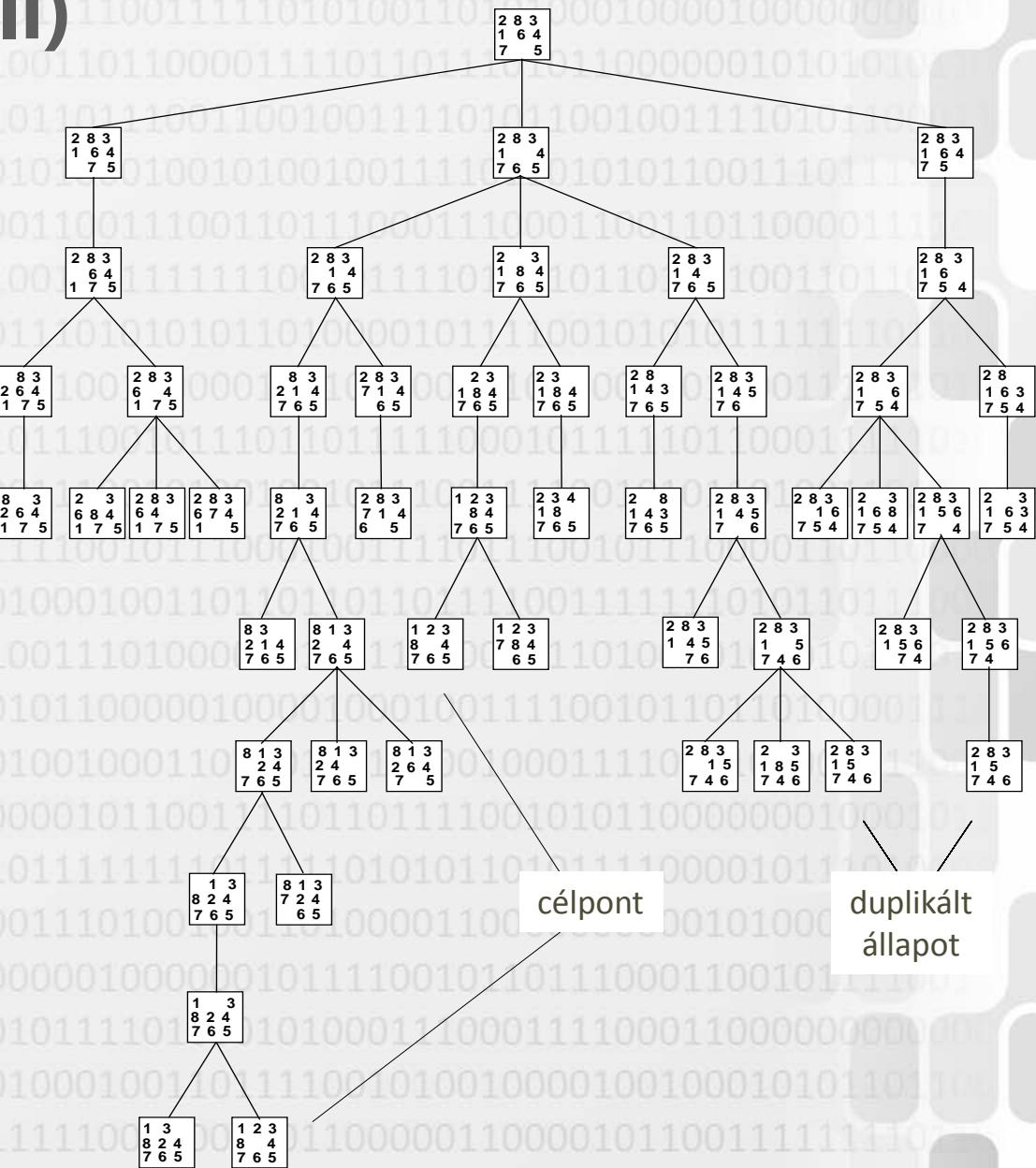
Állapottér gráf - példa

- Az állapotok az x vektor egyes koordinátáihoz rendelt lehetséges értékek (0 vagy 1) \Rightarrow faként reprezentálható.
- A problémának megfelelő gráf:



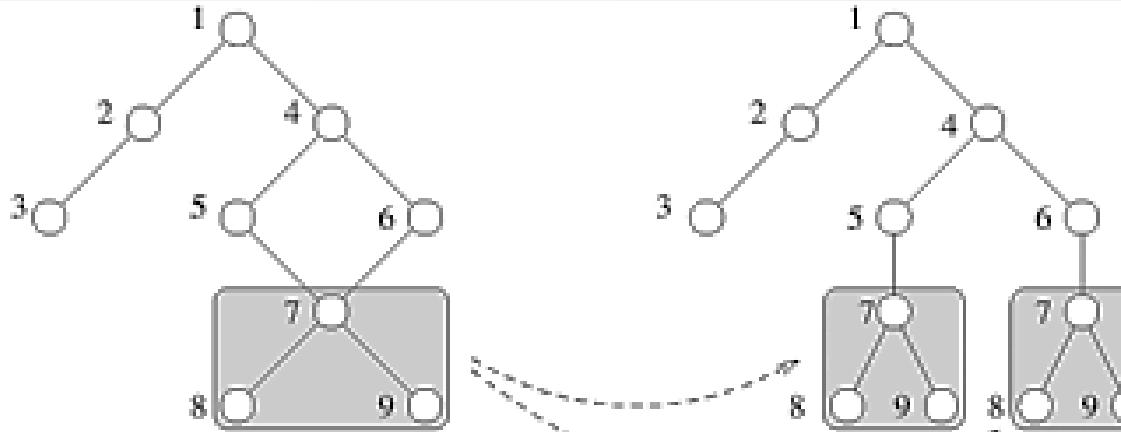
8-as kirakó (tili-toli)

- Egy kezdeti állapotból találjuk meg azt a legrövidebb lépéssorozatot, amely a célkonfigurációba vezet.
- Állapottér: gráf



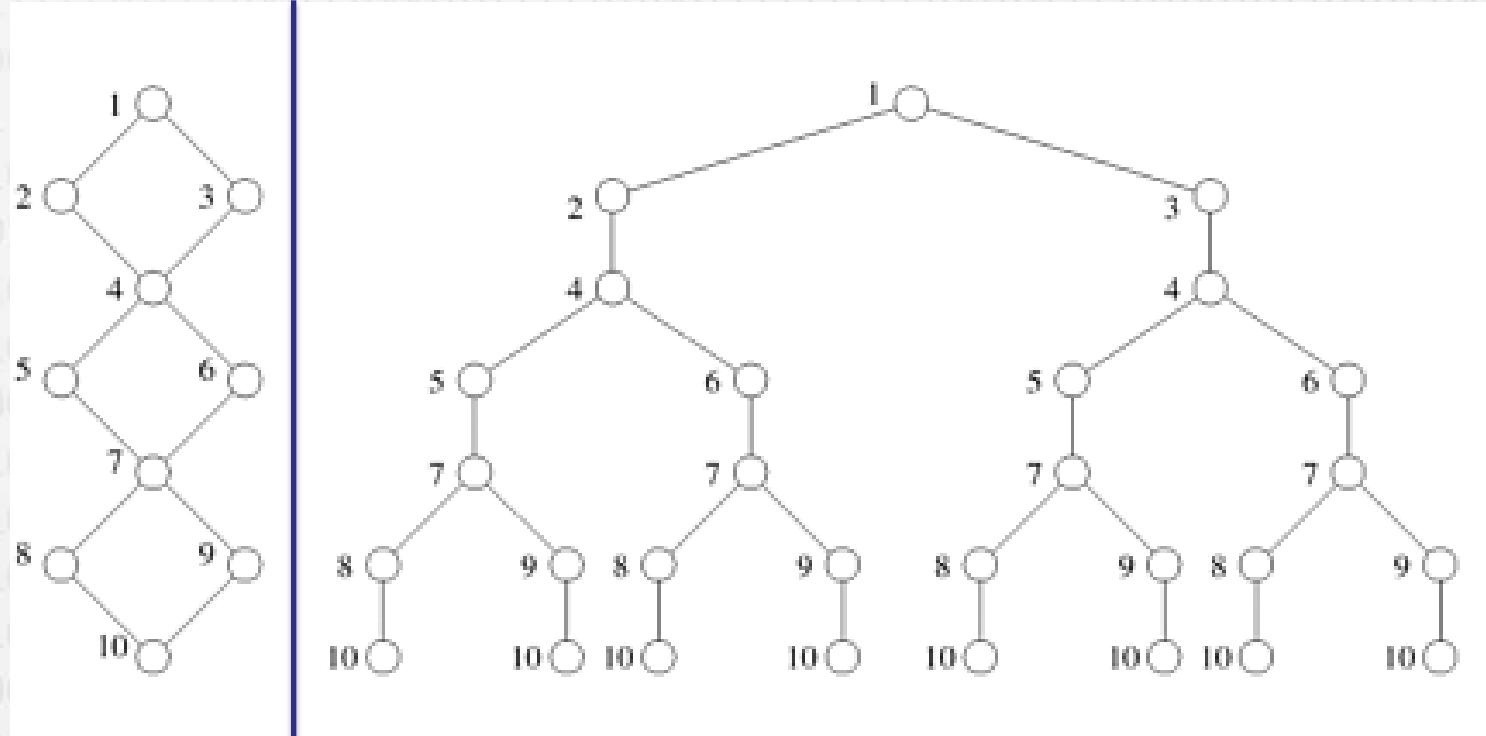
Kereső algoritmusok tere

- A keresési tér gráf vagy fa?
- A gráf fává történő kiterítése gyakran nagyméretű feladathoz vezet. Első példa:



Kereső algoritmusok tere

- A gráf favá történő kiterítése gyakran nagyméretű feladathoz vezet. Második példa:



Kereső stratégiák I.

- **Nem módosítható keresés**

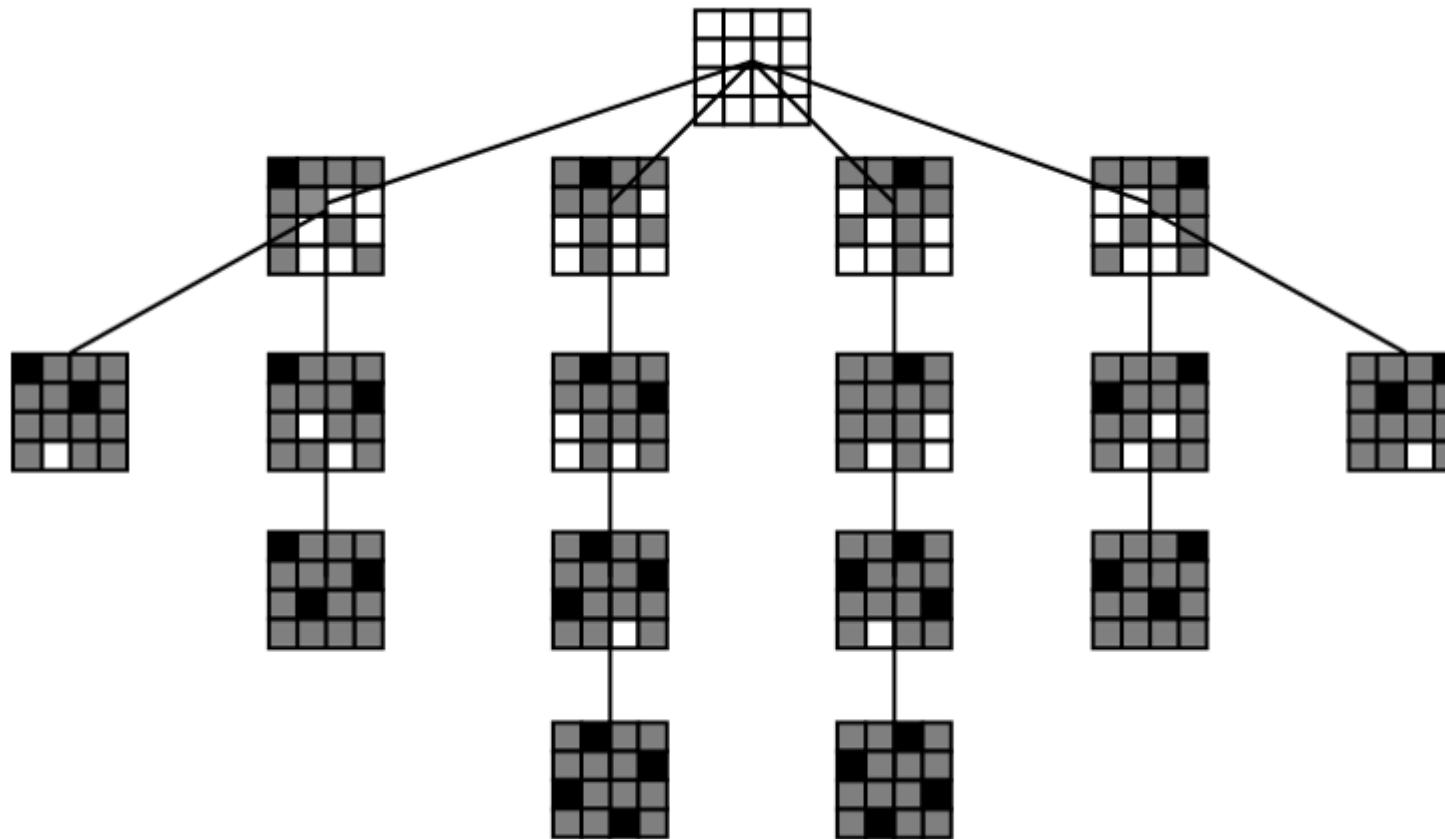
- Nincs visszalépés, pl. hegymászó-algoritmus (gradiens keresés).

- **Visszalépéses keresés**

- Törli a zsákutcát.
 - Ha körre futunk, akkor visszalépés.
 - Mélységi korlátot használhatunk (heurisztika).
 - Azonos szinten célszerű sorrendi heurisztikát használni.
 - Kis memóriaigényű.
 - Nem garantálja az optimális megoldást (csak növekvő mélységi korláttal és iterálva).

„Négy-királynő” probléma

- Visszalépéses keresés + azonos sorban haladva, minden balról jobbra helyezzük el a királynőket.



Feketével jelöltük a már lerakott királynőket, fehérrel pedig azokat a helyeket, ahová szabad elhelyezni királynőt a következő lépésben.

Kereső stratégiák II. Nem informált gráfkeresés

- **Nem informált gráfkeresések** – a gráfkiértékelő függvényben ($f(n) = g(n) + h(n)$) beépülő heurisztikát nem használunk.
- Csúcsok NYÍLT és ZÁRT lista.
- **Mélységi keresés (Depth-First Search: DFS)**
 - Egységes élköltséget tekintünk.
 - $f(n) := -g(n)$ minden n NYÍLT csúcsra.
 - Mélységi korlát használható, iterációs változatban is.
 - Fa állapottérnél előnyös.
- **Szélességi keresés**
 - Egységes élköltséget tekintünk.
 - $f(n) := g(n)$ minden n NYÍLT csúcsra.

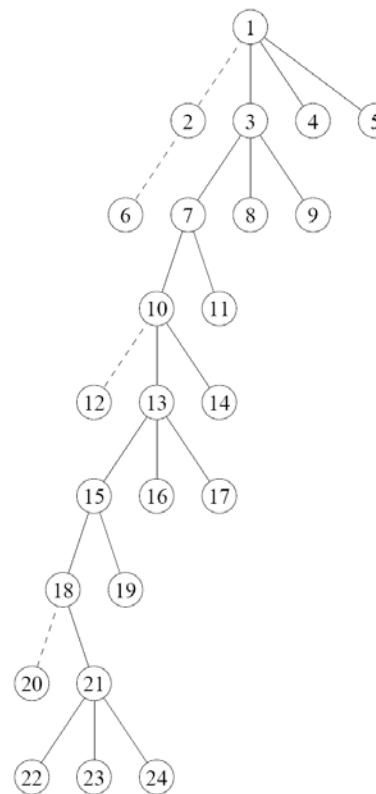
Iteratív mélységi keresés

- Gyakran a megoldás a gyökérhez közel, de másik ágon van.
- Az egyszerű mélységi keresés nagy részt dolgozna fel, mielőtt erre az ágra jut.
- Iteratív módon növeljük a mélységi határt, ameddig keresünk.
- Ha nem találunk megoldást, akkor a határt növeljük, és ismételjük a folyamatot.

Mélységi keresés: tárolási követelmény és adatstruktúra

- A mélységi keresés minden lépésénél a még ki nem próbált alternatívákat el kell tárolni.
- Ha m egy állapot tárolásához szükséges adatterület, és d a maximális mélység, akkor a mélységi kereséshez szükséges teljes tér $O(md)$ nagyságrendű.
- Ha a keresendő állapottér fa, akkor a mélységi keresést veremmel hatékonyan lehet reprezentálni.
- A verem memóriaszükséglete egyenesen arányos a fa mélységével.

Mélységi keresés: tárolási követelmény és adatstruktúra



Stack alja

5
4
9
8
11
14
17
16
19
24
23

1	4	5
3	8	9
7	11	
10	14	
13	16	17
15	19	
18		
21	23	24

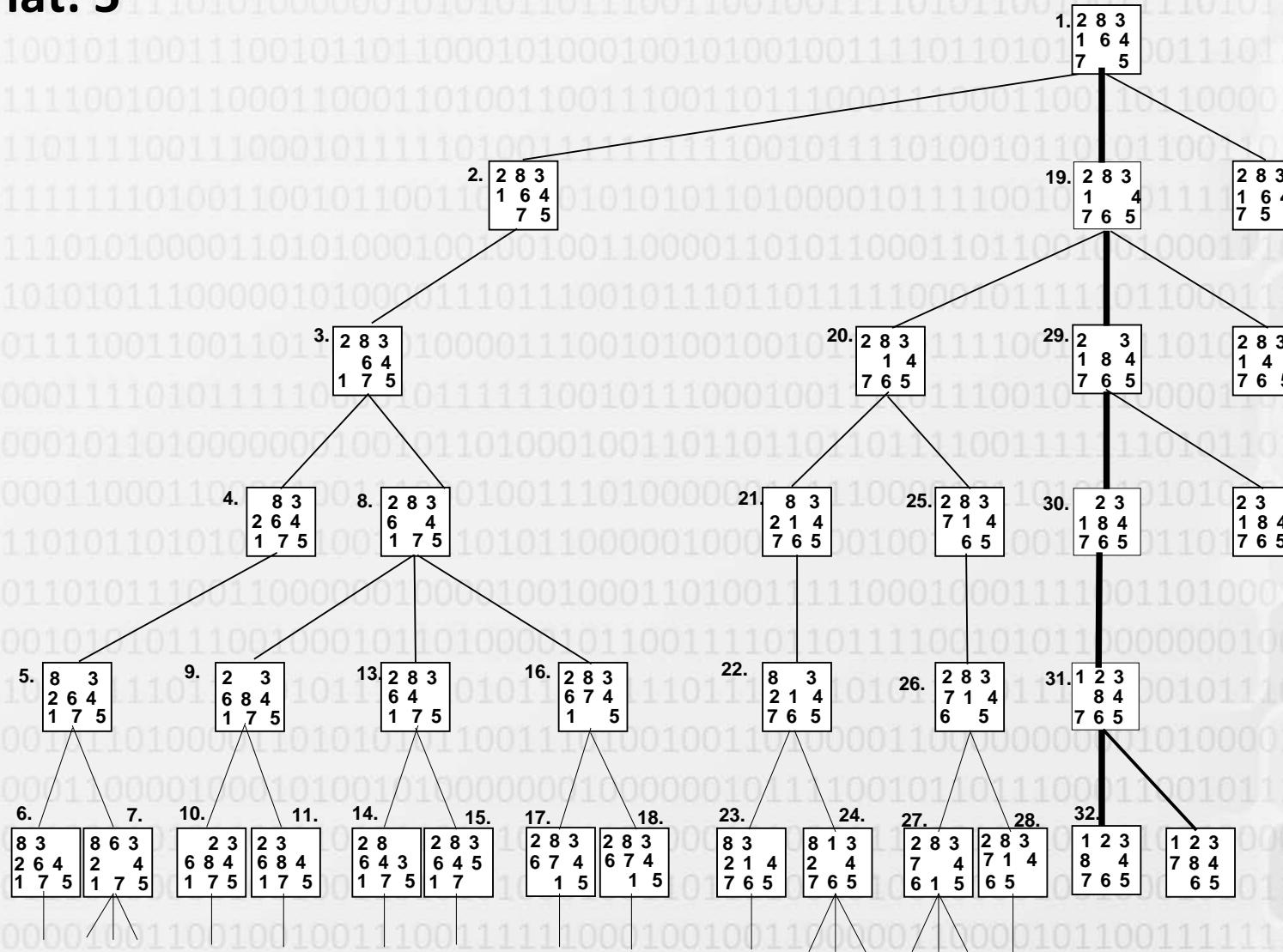
Stack teteje
a,
b,

Aktuális állapot
a,

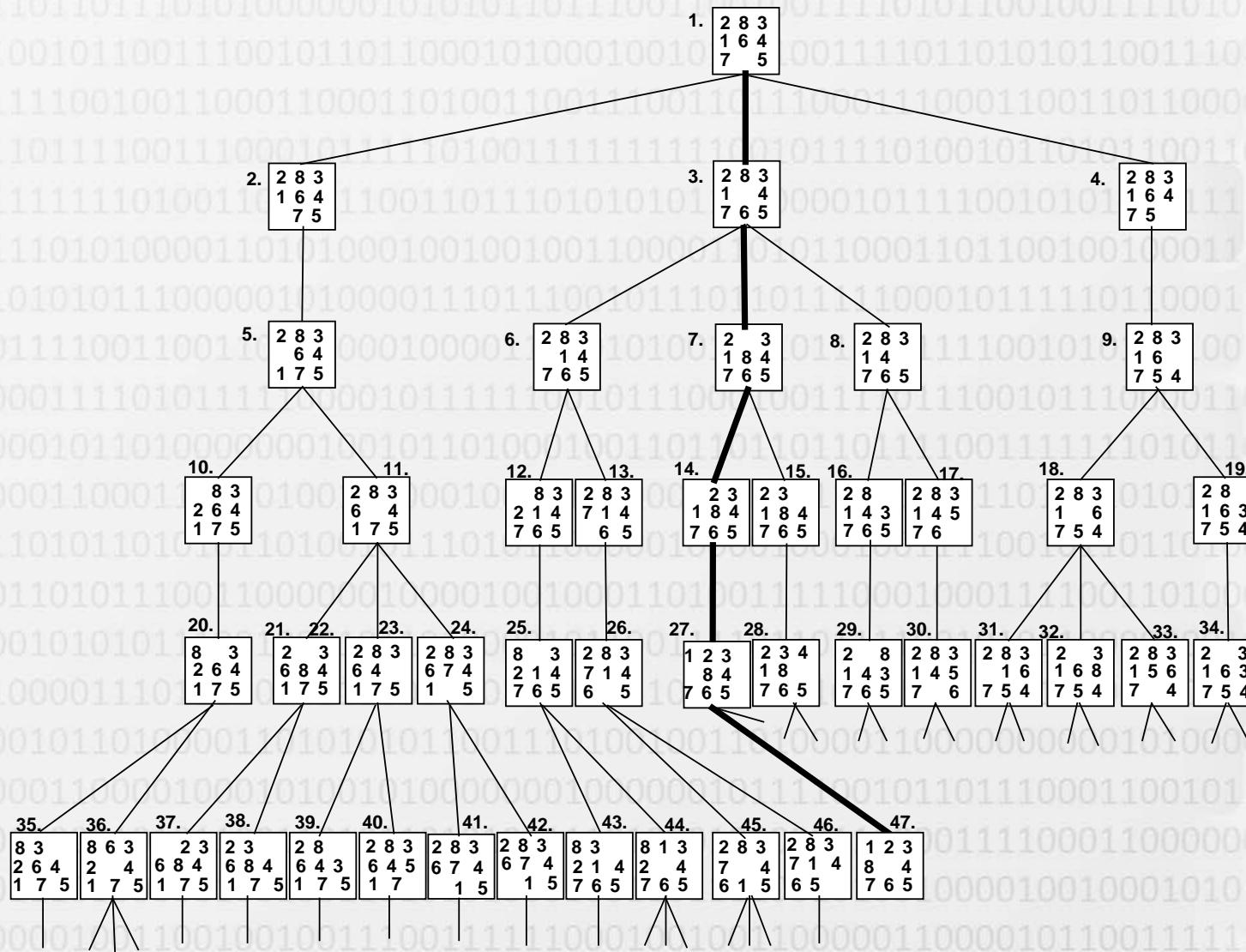
(a) Mélységi keresés fa esetén; pontozottan jelennek meg a már felkeresett csúcsok; a verem a még ki nem próbált alternatívákat tárolja csak (a); a verem a ki nem próbált alternatívákat és szüleiket (szürkített) tárolja (b).

Mélységi keresés korláttal - példa

- **Korlát: 5**



Szélességi keresés - példa



Heurisztikus gráfkereső stratégiák

- Az $f(x)$ kiértékelő függvénybe valamelyen heurisztikát építünk be ($f(x) = g(x) + h(x)$). Gyakran például becsülhető a cél elérésének költsége az aktuális csúcsból.
- A becslést *heurisztikus becslésnek* nevezik.
- Ha a becslés garantáltan alábecsül, akkor *megengedő heurisztikáról* beszélünk.
- Elképzelés: ha heurisztikát használunk, akkor a „rossz” vagy „nem sokat ígérő” utakra nem költünk.

Diszkrét optimalizálás példához heurisztika

A 8-as kirakóhoz (megengedő) heurisztika

- $W(n)$ (n NYÍLT), ahol W a rossz helyek száma.

A 8-as kirakóhoz megengedő heurisztika

- A rács minden pontja a koordinátáival azonosítható.
- Az (i, j) és a (k, l) pozíciók távolsága: $|i - k| + |j - l|$. Ez a Manhattan-távolság.
- A kezdeti- és célpozíciók közötti Manhattan-távolságok összege (P -vel fogjuk jelölni) megengedő heurisztika.

Kereső stratégiák III. Heurisztikus gráfkereső stratégiák

- **Előretekintő keresés**

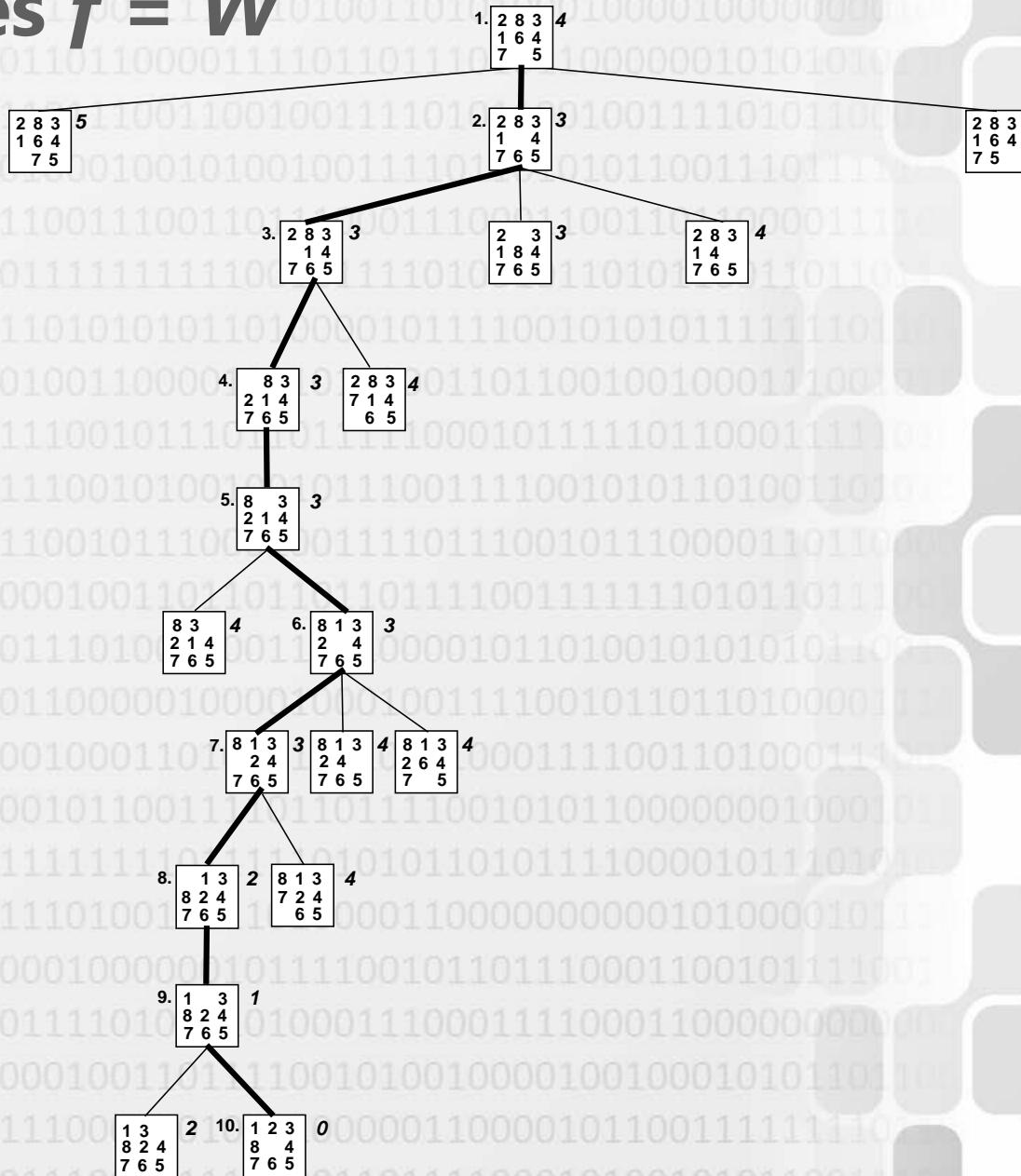
- $f(n) = h(n)$ minden n NYÍLT csúcsra.

- **A***

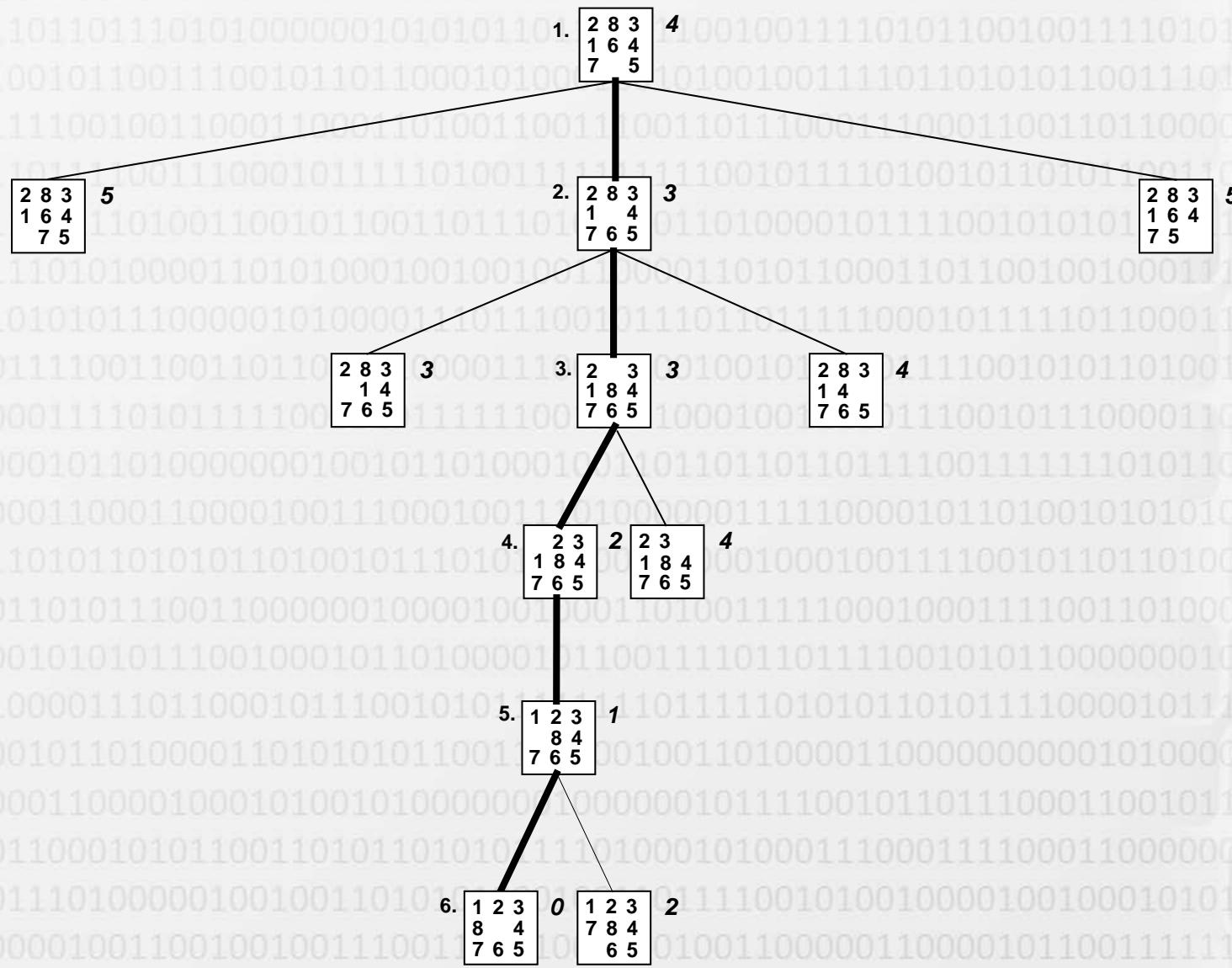
- Kiértékelő fgv. $f(n) = g(n) + h(n)$ minden n NYÍLT csúcsra, ahol $h(n)$ megengedő heurisztika.
 - Garantálja az optimális megoldást.
 - Nagy a memóriaigény.
 - Meglátogatott csúcsokkal (állapotokkal) arányos.
 - Mind fák, mind gráfok estében hatékony.

Előretekintő keresés $f = W$

- W hibásak száma,
hátralévő mozgások
alsó becslését adja.



A* példa P heurisztikával



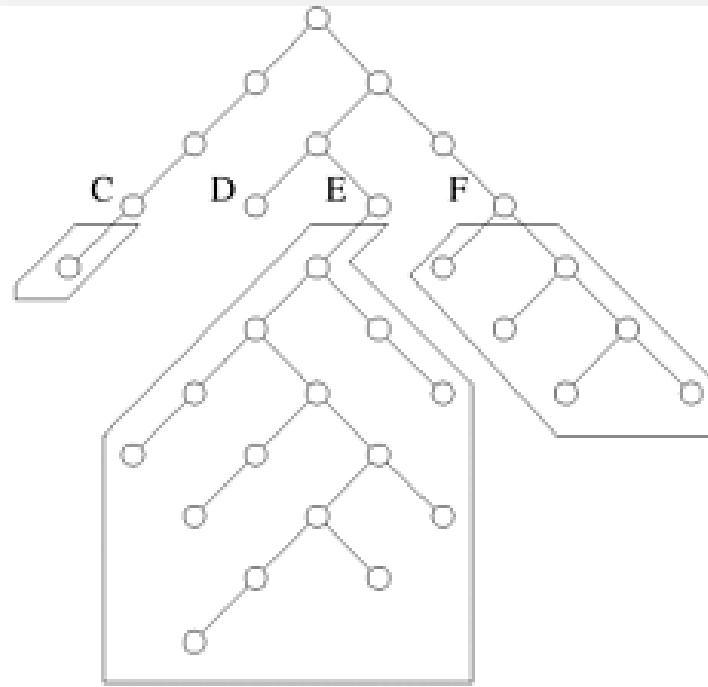
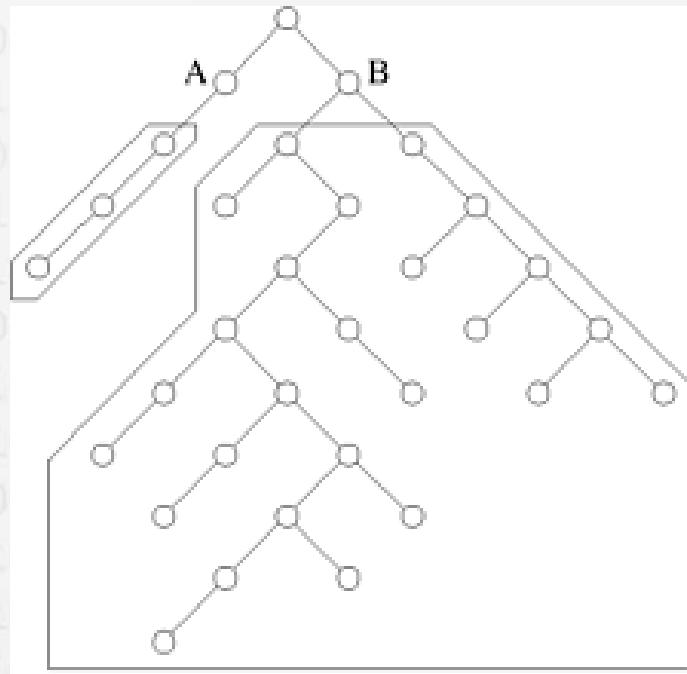
Párhuzamos diszkrét optimalizáció: motiváció

- DOP általában NP-teljes probléma. Segít-e a párhuzamosítás?
- Sok problémánál az átlagos eset polinomiális idejű.
- Sokszor viszonylag kis állapotterünk van, de valós idejű megoldásra van szükségünk.
- Többleträfordítás
 - A soros és párhuzamos keresés közötti munkamennyiség gyakran különböző.
 - Ha W a soros munka és W_P a párhuzamos, a többleträfordítás $s = W_P/W$.
 - A sebességnövekedés határa $p \times (W/W_P)$.

Párhuzamos mélységi keresés

- A keresési teret miként osszuk fel processzorok között?
- A különböző ágakat lehet párhuzamosan keresni.
- De az ágak nagyon eltérő méretűek lehetnek.
- Nehéz becsülni az ágak méretét a gyökerükből.
- Dinamikus terheléskiegyenlítés szükséges.

Párhuzamos mélységi keresés

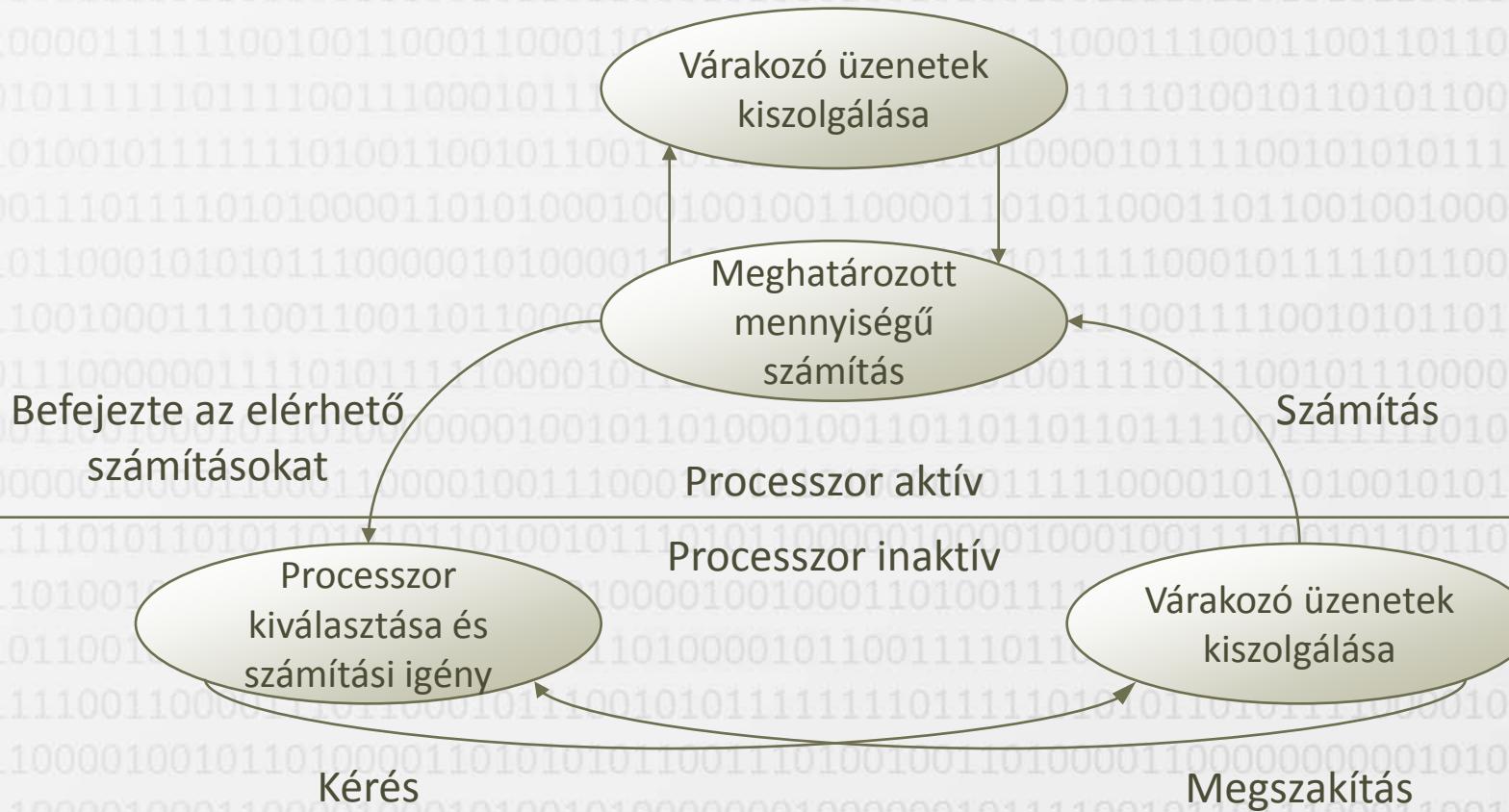


- A statikus dekompozíció strukturálatlan fakereséshez és nem kiegyensúlyozott terheléshez vezet.

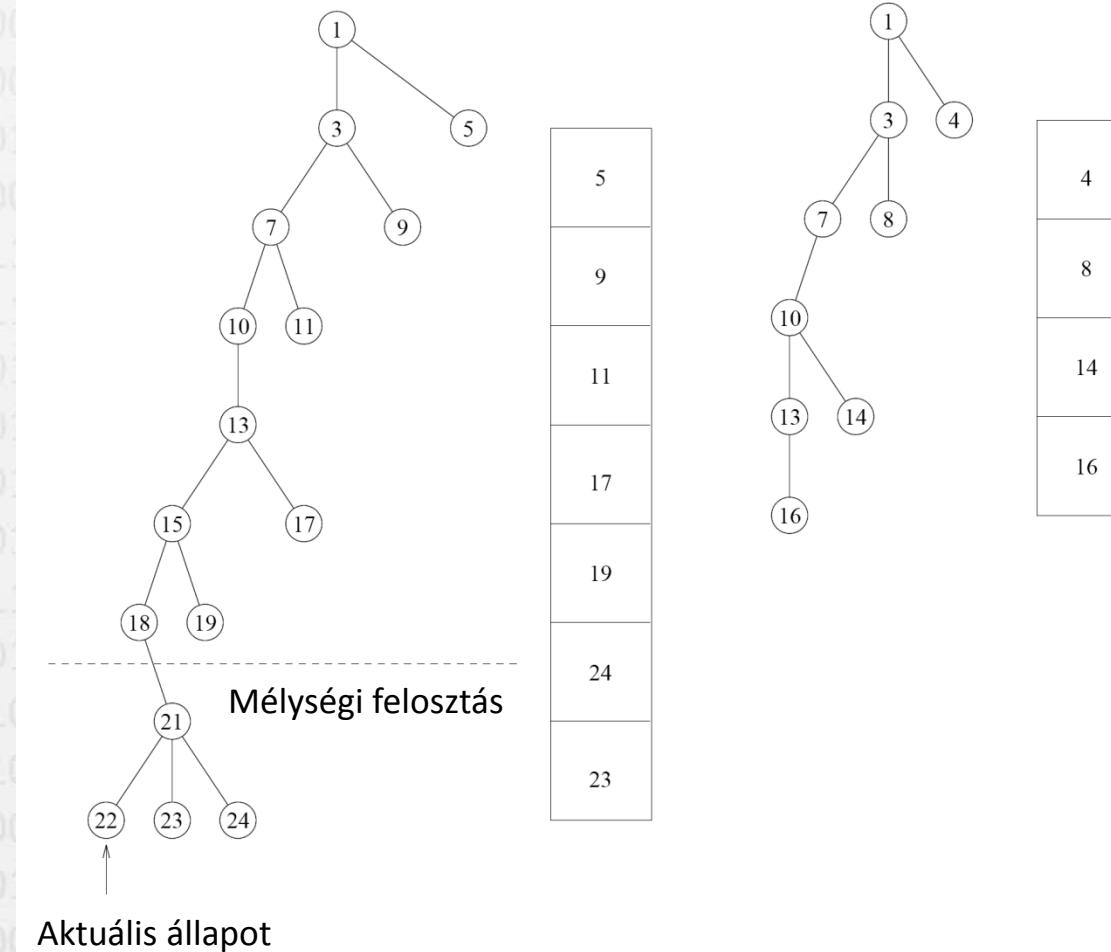
Párhuzamos mélységi keresés: dinamikus terheléskiegyenlítés

- **Amikor egy processzor befejezi munkáját, akkor egy másiktól kér.**
- **Osztott memóriás modellnél ez lockolással és a munka szétvágásával történik (lásd később).**
- **Ha valamely processzor megoldáshoz ér, akkor a többi terminál.**
- **Az eddig fel nem dolgozott részeket saját veremben tárolhatja minden processzor.**
- **Kezdetben az egész teret egy processzorhoz rendeljük.**

Párhuzamos mélységi keresés: dinamikus terheléskiegyensúlyozás (load balancing)



Párhuzamos mélységi keresés: munkafelosztás



Mélységi keresés: a két részfa a verem reprezentációval.

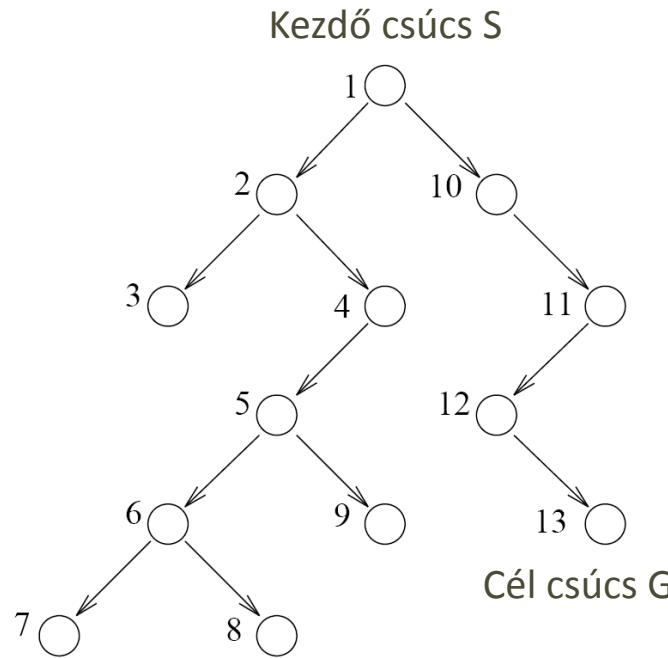
Terheléskiegyenlítés

A munka kiosztásának sémája:

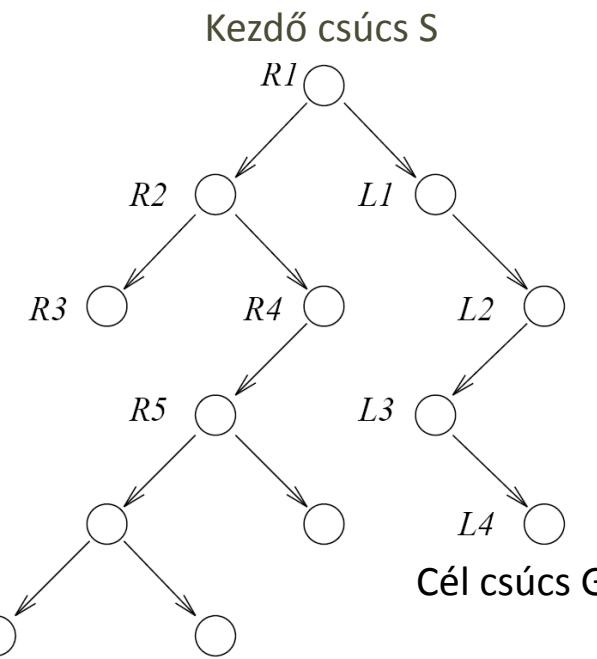
- **Aszinkron (lokális) körbeforgó:** minden processzor egy számlálót tart karban ($(target + 1) \bmod p$), és az igényét a felé közvetíti.
 - Nagyszámú munkaigényt generál.
- **Globális körbeforgó:** a rendszer tart karban egy számlálót és az igények körbeforgó módon kerülnek kezelésre.
 - Legkevesebb munkaigényt generálja.
- **Véletlen ciklikus:** véletlenszerűen kerül kiválasztásra egy processzor.
 - Megfelelő kompromisszum.

Sebesség anomáliák párhuzamos keresésnél

Soros mélységi keresés



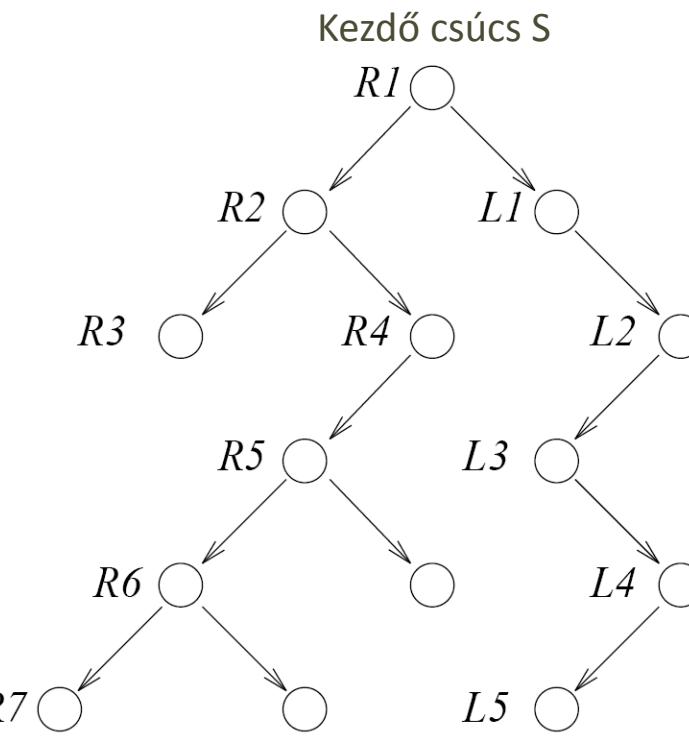
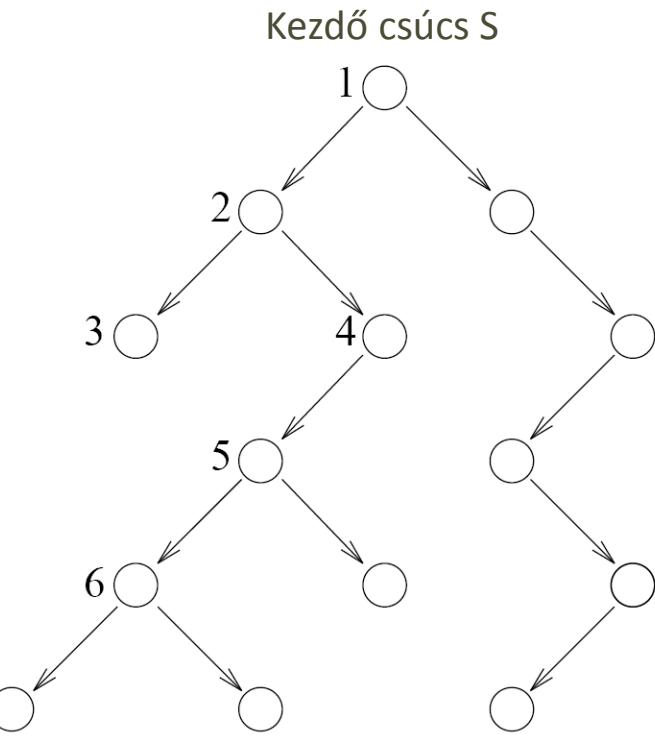
Párhuzamos mélységi keresés



A soros mélységi keresés 13 csúcsot generált, míg a párhuzamos mélységi keresés 2 processzorral (R és L) csak 9 csúcsot.

A párhuzamos mélységi keresés kevesebb csomópontot jár be, mint a soros.

Sebesség anomáliák párhuzamos keresésnél



A soros mélységi keresés 7csúcsot generált, míg a párhuzamos mélységi keresés 12 csúcsot.

A párhuzamos mélységi keresés több csomópontot jár be, mint a soros.

Dinamikus programozás párhuzamosítási lehetőségekkel

A dinamikus programozás (DP) áttekintése

Soros, egyszerű-argumentumú (Monadic) DP

Nem soros, egyszerű-argumentumú DP

Soros, összetett-argumentumú (Polyadic) DP

Nem soros, összetett-argumentumú DP

A dinamikus programozás áttekintése

- A *dinamikus programozás (DP)* széles körben használt optimalizációs problémák megoldására: ütemezés, stringszerkesztés, csomagolási probléma stb.
- A problémákat részproblémáakra bontjuk, és ezek megoldásait kombináljuk, hogy a nagyobb probléma megoldását megtaláljuk.
- Az „oszd meg és uralkodj” típusú megoldási módszerrel szemben itt a részproblémák között kapcsolatok, átfedések állhatnak fent (*overlapping subproblems*).
- Az elnevezés matematikai optimálásra utal.

A dinamikus programozás áttekintése

- **A problémamegoldás menete:**

- részproblémákra osztás;
- a részproblémák optimális megoldása rekurzívan;
- az optimális megoldások felhasználása az eredeti feladat optimális megoldásának megtalálásához: egy részprobléma megoldását gyakran egy vagy több az előző szintű részproblémák függvényeként adják meg.

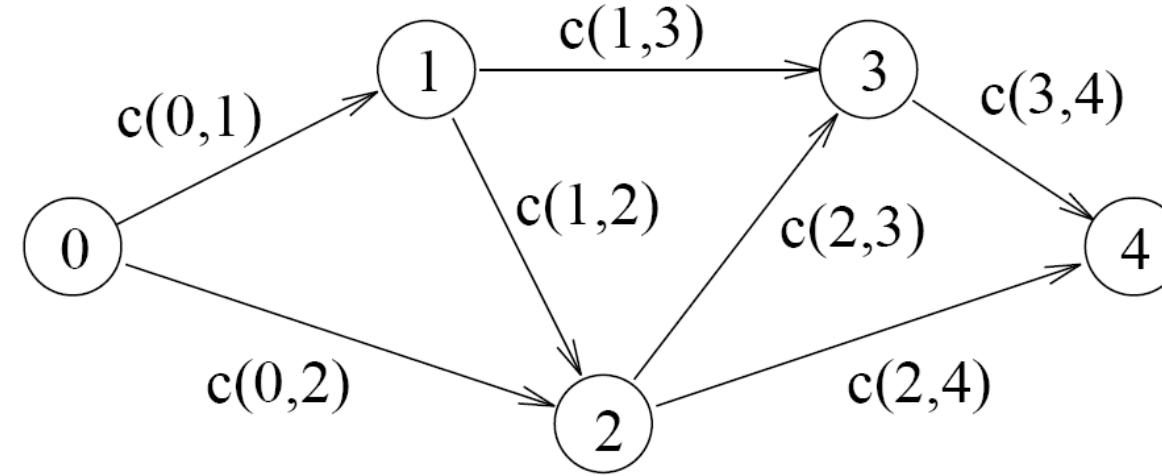
- **A dinamikus programozás (DP) során a részmegoldások eredményeit gyakran táblázatokban tároljuk, hogy azokat – szemben a rekurzív megközelítésekkel – ne kelljen újra és újra kiszámolni (*memorization*).**

Dinamikus programozás: példa

- Tekintsük a hurokmentes irányított gráfokban két csúcs (csomópont) között a legrövidebb út problémáját.
- Az i és j csomópontokat összekötő él költsége $c(i, j)$.
- A gráf n csomópontot tartalmaz: $0, 1, \dots, n-1$, és az i csomópontból a j csomópontba akkor vezethet él, ha $i < j$. A 0 csomópont a forrás és az $n-1$ csomópont a cél.
- Jelölje $f(x)$ a legrövidebb utat a 0 csomóponttól x -ig.

$$f(x) = \begin{cases} 0 & x = 0 \\ \min_{0 \leq j < x} \{ f(j) + c(j, x) \} & 1 \leq x \leq n - 1 \end{cases}$$

Dinamikus programozás: példa



- Irányított gráf, melyben a legrövidebb utat számolhatjuk a 0 és 4 csomópontok között.

$$f(4) = \min\{f(3) + c(3,4), f(2) + c(2,4)\}.$$

Dinamikus programozás

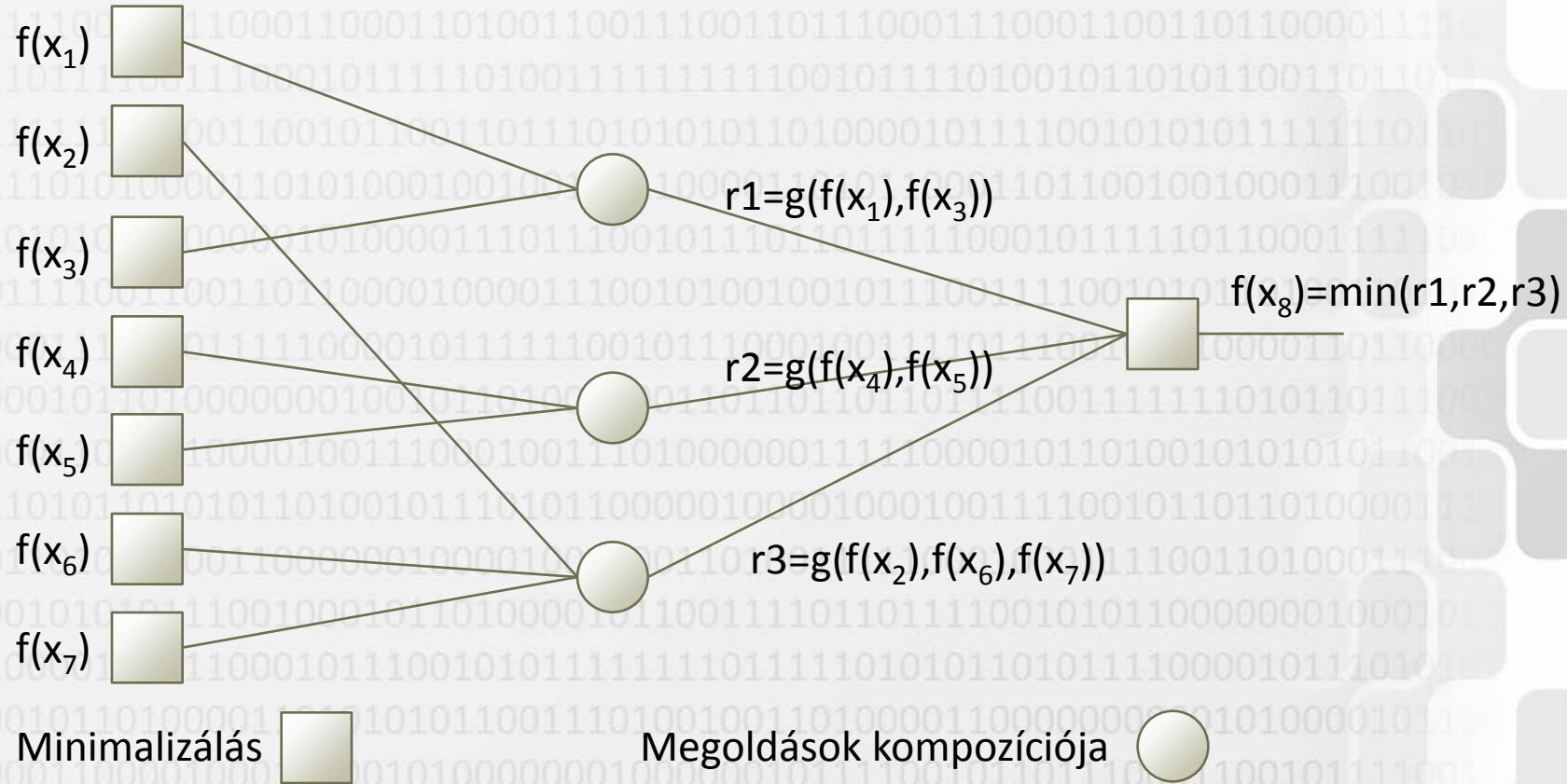
- A DP formátumú probléma megoldását tipikusan a lehetséges megoldások minimumaként vagy maximumaként fejezzük ki.
- Ha r az x_1, x_2, \dots, x_l részproblémák kompozíciójából meghatározott megoldás költségét jelenti, akkor r a következő alakban írható fel:

$$r = g(f(x_1), f(x_2), \dots, f(x_l)).$$

Itt g az ún. kompozíciós függvény.

- Ha minden probléma optimális megoldása a részfeladatok optimális megoldásának optimális kompozíciójaként kerül meghatározásra, és a minimum (vagy maximum) érték kiválasztásra kerül, akkor DP formátumú megoldásról beszélünk. (Metamódszer, nem konkrét algoritmus.)

Dinamikus programozás: példa



Az $f(x_8)$ megoldás meghatározásának kompozíciója és számítása részfeladatok megoldásából

Dinamikus programozás

- A rekurzív DP egyenletet *funkcionális egyenletnek* vagy *optimalizációs egyenletnek* is nevezzük.
- A legrövidebb út problémánál a kompozíciós függvény $f(j) + c(j, x)$. Ez egyszerű rekurzív tagként tartalmazza ($f(j)$). Az ilyen DP megfogalmazású feladatot monadic típusúnak (egyszerű-argumentumúnak) nevezzük.
- Ha a rekurzív kompozíciós függvény több tagú, akkor a DP feladat megszövegezése polyadic (összetett- argumentumú) típusú.

Dinamikus programozás

- A részproblémák közötti kapcsolatokat gráf segítségével fejezhetjük ki.
- Ha a gráf szintekre bontható, és a probléma megoldása egy bizonyos szinten csak az előző szinten megjelenő problémamegoldástól függ, ekkor a megfogalmazást *sorosnak* nevezzük, egyébként pedig *nem Sorosnak*.
- A két kritérium alapján a DP feladatmegfogalmazásokat osztályozhatjuk: serial-monadic, serial-polyadic, non-serial-monadic, non-serial-polyadic vagy egyéb.
- Az osztályozás azért fontos, mert meghatározza a konkurenciát és a függőségeket, amelyek a párhuzamos megoldásokhoz vezethetnek.

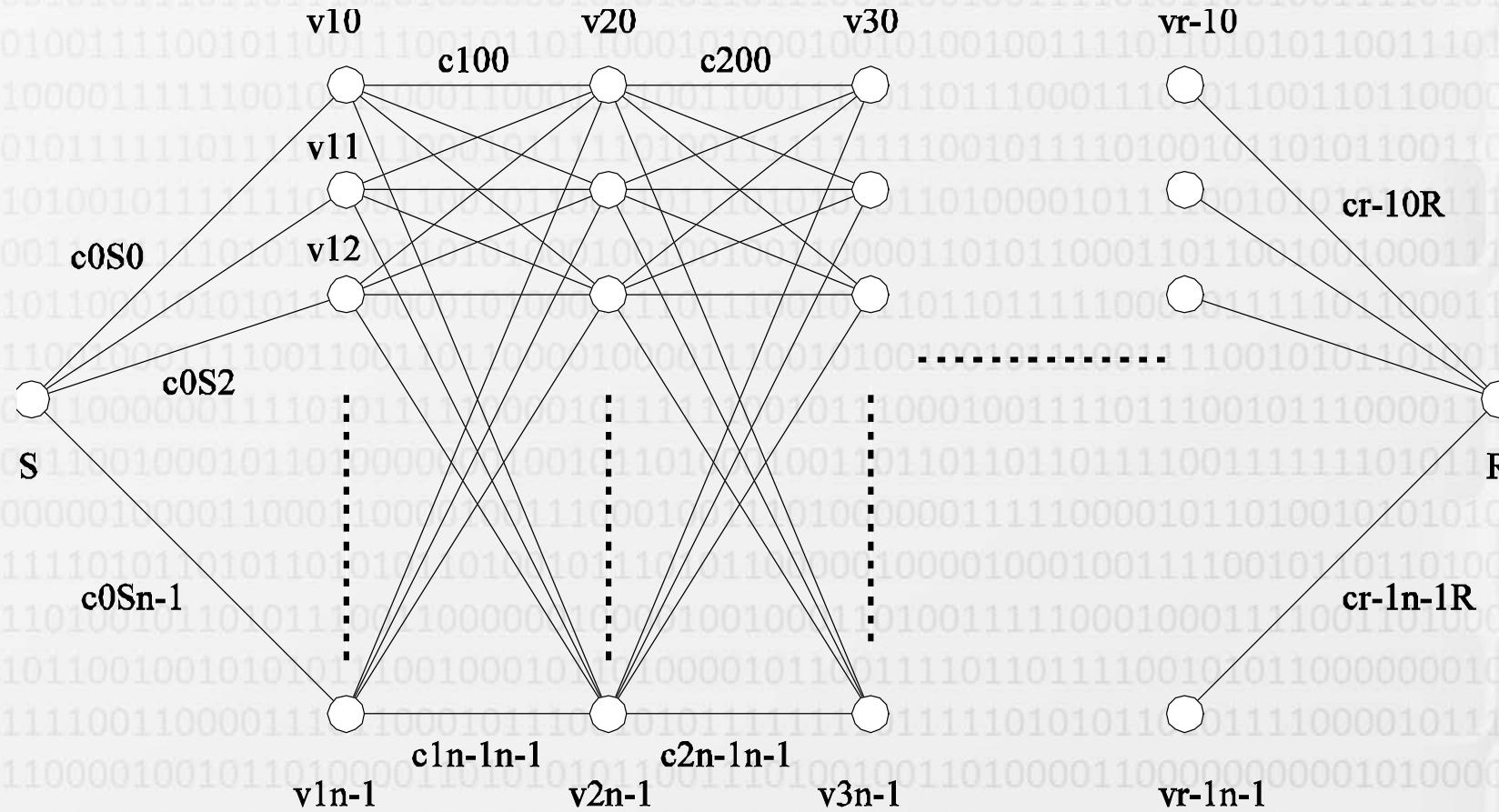
Soros, egyszerű-argumentumú DP

- Az egész osztályra vonatkozó általános megfogalmazást, mintát nehéz megadni.
- Két reprezentatív feladat kerül bemutatásra:
 - A legrövidebb út probléma többszintű gráfban (*shortest-path problem for a multistage graph*),
 - 0/1 típusú hátizsák probléma (*0/1 knapsack problem*).
- Célunk a feladatok párhuzamos megfogalmazása és ezek alapján az osztályon belüli közös elvek lefektetése.

Legrövidebb út probléma

- A legrövidebb út problémának speciális osztálya, amikor a gráf súlyozott és többszintű. Jelen esetben a szintek száma $r + 1$.
- minden szinten n csomópont van, és minden csomópont az i . szintről össze van kötve az $i + 1$ szint minden egyes csomópontjával.
- A 0. szint és az r . csak egy csomópontot tartalmaz, ezeket forrás- (S) és cél- (R) csomópontoknak nevezzük.
- Célunk legrövidebb utat találni S -től R -ig.

Legrövidebb út probléma



Soros, egyszerű-argumentumú (monadic) DP megfogalmazása a legrövidebb út problémának olyan gráfban, ahol a csomópontokat szintekbe szervezhetjük.

Legrövidebb út probléma

- A gráf / szintjének *i*. csomópontja: v_i^l és a v_i^l csomópontot v_j^{l+1} csomóponttal összekötött él súlya $c_{i,j}^l$.
- Bármely v_i^l csomópont és az cél csomópont R költsége: C_i^l .
- Ha n csomópont van egy szinten (*l*), akkor a költségek (legrövidebb utak R -ig) egy vektorban foglalhatók össze: $[C_0^l, C_1^l, \dots, C_{n-1}^l]^T$. Ez a vektor C^l . Megjegyzés: $C^0 = [C_0^0]$.
- Az *l* szint *i* eleme a céltól $C_i^l = \min \{(c_{i,j}^l + C_j^{l+1}) \mid j \text{ az } l+1 \text{ szint egy csomópontjának indexe}\}$

Legrövidebb út probléma

- Mivel a v_j^{r-1} csomópontokat csak egy-egy él köti össze a cél, R csomóponttal az r szinten, a C_j^{r-1} költsége: c_{j-R}^{r-1} .
- Ezért:

$$C^{r-1} = [c_{0,R}^{r-1}, c_{1,R}^{r-1}, \dots, c_{n-1,R}^{r-1}].$$

A feladat soros és egyszerű-argumentumú (monadic) típusú.

Legrövidebb út probléma

- Az I szint valamely csomópontjától a célcsomópont, R elérésének költsége, ahol ($0 < I < r - 1$):

$$C_0^l = \min\{(c_{0,0}^l + C_0^{l+1}), (c_{0,1}^l + C_1^{l+1}), \dots, (c_{0,n-1}^l + C_{n-1}^{l+1})\},$$

$$C_1^l = \min\{(c_{1,0}^l + C_0^{l+1}), (c_{1,1}^l + C_1^{l+1}), \dots, (c_{1,n-1}^l + C_{n-1}^{l+1})\},$$

⋮

$$C_{n-1}^l = \min\{(c_{n-1,0}^l + C_0^{l+1}), (c_{n-1,1}^l + C_1^{l+1}), \dots, (c_{n-1,n-1}^l + C_{n-1}^{l+1})\}.$$

Legrövidebb út probléma

- A probléma megoldását egy módosított mátrix-vektor szorzat formájában fejezhetjük ki.
- Helyettesítsük az összeadás operációt minimalizálással és a szorzás műveletet összeadással, az előző egyenletek a következő alakúak lesznek:

$$C^l = M_{l,l+1} \times C^{l+1},$$

ahol C' és C'^{+1} $n \times 1$ méretű vektorok a célcsomópont elérésének költségeit reprezentálják az l és $l + 1$ szint minden egyes csomópontjától mérve.

Legrövidebb út probléma

- Az $M_{l,l+1}$ mátrix $n \times n$ méretű, és minden (i, j) eleme az l szint i csomópontjának az $l + 1$ szint j csomópontjával történő összekötés költségét tartalmazza:

$$M_{l,l+1} = \begin{bmatrix} c_{0,0}^l & c_{0,1}^l & \dots & c_{0,n-1}^l \\ c_{1,0}^l & c_{1,1}^l & \dots & c_{1,n-1}^l \\ \vdots & \vdots & & \vdots \\ c_{n-1,0}^l & c_{n-1,1}^l & \dots & c_{n-1,n-1}^l \end{bmatrix}.$$

- A legrövidebb út problémát r mátrix-vektor szorzatok sorozataként definiáltuk: C^{r-1}, C^{r-k-1} ($k = 1, 2, \dots, r-2$), C^0 .

Párhuzamos legrövidebb út probléma

- Az algoritmus úgy párhuzamosítható, mint bármelyik mátrixvektor szorzást megvalósító algoritmus.
- n processzor elem számolhat minden C' vektort.
- Sok gyakorlati esetben az M mátrix elemei ritkák lehetnek.
Ilyenkor ritkamátrixokhoz kapcsolódó módszert alkalmazzunk.

0/1 „hátizsák” probléma

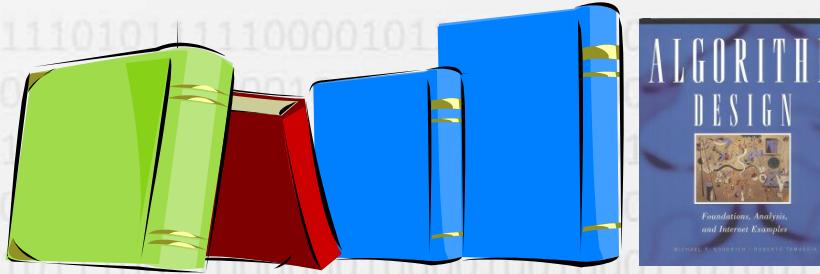
0/1 knapsack problem

- „A thief is robbing the King’s treasury, but he can only carry a load weighing at most W ...”
- Adott: S halmaz n elemmel és egy súlyhatár c , minden i elemnek
 - p_i – pozitív egész értéke van (*profit*)
 - w_i – és pozitív egész súlya.
- Cél: úgy válasszuk ki az elemeket, hogy az összérték maximális legyen, de az összsúly kisebb legyen, mint c .
 - T jelölje azokat az elemeket, amit kiválasztunk, $T \subseteq S$
 - Cél: összérték maximalizálása. $\sum_{i \in T} p_i$
 - Feltétel: a súlyhatáron belül kell maradni. $\sum_{i \in T} w_i \leq C$

0/1 hátizsák probléma: példa

- Adott: n elemű S halmaz, minden elem
 - p_i – pozitív értékű és w_i – pozitív súlyú.
- Cél: válasszunk ki elemeket, hogy az összérték maximális legyen, de az összsúly ne haladja meg c -t.
- Összesen 2^n eset lehet!

Elemek:



1 2 3 4 5

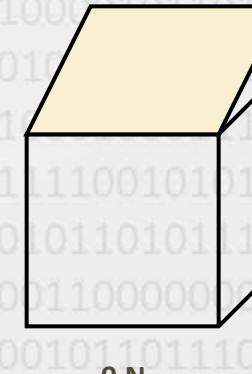
Súly:

4 N 2 N 2 N 6 N 2 N

Érték:

20 3 6 25 80

“zsák”



9 N

Megoldás:

- 5 (2 N)
- 3 (2 N)
- 1 (4 N)

Összérték:

- $80 + 6 + 20 = 106$

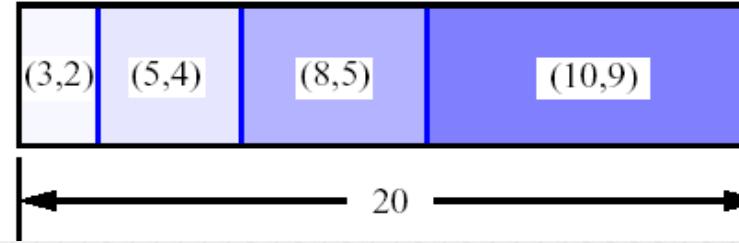
0/1 hátizsák probléma, első próbálkozás

- S_i : a halmaz elemeinek azonosítója 1-től i -ig ($i < n$).
- Definiálja $F[i]$ = a legjobb kiválasztást S_i -ből.
- Legyen pl. $S = \{(3,2), (5,4), (8,5), (4,3), (10,9)\}$ érték-súly párok, $c = 20$

A legjobb S_4 ,
ha csak négyet veszünk ki:
Összsúly: 14, érték: 20.



A legjobb S_5 :
az 5. benne van, de a 4. nem.



- Rossz hír: S_4 nem része az S_5 optimális megoldásnak.

0/1 hátizsák probléma, más megközelítés

- S_i : az elemek halmaza 1-től i -ig, $i < n$.
- Definiálja $F[i, x] =$ az S_i halmazból a legjobb kiválasztást, ahol a súly legfeljebb x – további paraméter.
- Ez optimális részprobléma-megoldáshoz vezet.
- Az S_i legjobb kiválasztás legfeljebb x súlyal két esetet jelenthet:
 - ha $w_i > x$, akkor az i . elemet nem lehet hozzávenni, mert súlya nagyobb, mint az aktuális határ;
 - egyébként: ha a legjobb S_{i-1} részhalmaz súlya $x - w_i$ és ehhez vagy jön i , vagy nem eredményez nagyobb értéket

$$F[i, x] = \begin{cases} F[i-1, x] & \text{if } w_i > x \\ \max\{F[i-1, x], F[i-1, x - w_i] + p_i\} & \text{egyébként} \end{cases}$$

0/1 hátizsák algoritmus

- $F[i, x]$ rekurzív formula:

$$F[i, x] = \begin{cases} F[i-1, x] & \text{if } w_i > x \\ \max\{F[i-1, x], F[i-1, x - w_i] + p_i\} & \text{egyébként} \end{cases}$$

- $F[i, x]$ = az 1-től i -ig tartó elemekből a legjobb kiválasztás, ahol az összsúly legfeljebb x .
- Alapeset: $i = 0$, nem került elem kiválasztásra. Összérték 0.
- A feladat megoldása: a legnagyobb érték az n . sorban, utolsó (c) oszlopban.
- Futási idő: $O(nc)$.
Nem 2^n ideig tart.

0/1 hátizsák algoritmus

Algorithm 0-1Knapsack(S, c):

Input: S halmaz elemei p_i értékkal, w_i súllyal; valamint a max. súly c

Output: a legjobb részhalmaz értéke ($F[n, c]$), hogy teljesül: összsúly $\leq c$

for $x \leftarrow 0$ **to** c **do**

$F[0, x] \leftarrow 0$

for $i \leftarrow 0$ **to** n **do**

$F[i, 0] \leftarrow 0$

for $i \leftarrow 0$ **to** n **do**

for $x \leftarrow 0$ **to** c **do**

if $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

- **Mivel $F[i, x]$ csak $F[i - 1, *]$ értékeitől függ, elég lehet vektorokat használni.**

0/1 hátizsák példa

- **n = 4 az adatok száma**
- **c = 5 kapacitás (maximális összsúly)**

• Elemek:	i	súly (w_i)	profit (p_i)
	1	2	3
	2	3	4
	3	4	5
	4	5	6

Példa

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for $x \leftarrow 0$ **to** c **do**

$$F[0, x] \leftarrow 0$$

Példa

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for i ← 0 to n do

$F[i, 0] \leftarrow 0$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0			
2	0					
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 1$
 $p_i = 3$
 $w_i = 2$
 $x = 1$
 $x - w_i = -1$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 1$
 $p_i = 3$
 $w_i = 2$
 $x = 2$
 $x - w_i = 0$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 1$
 $p_i = 3$
 $w_i = 2$
 $x = 3$
 $x - w_i = 1$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 1$
 $p_i = 3$
 $w_i = 2$
 $x = 4$
 $x - w_i = 2$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 1$
 $p_i = 3$
 $w_i = 2$
 $x = 5$
 $x - w_i = 3$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 2$
 $p_i = 4$
 $w_i = 3$
 $x = 1$
 $x - w_i = -2$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 2$
 $p_i = 4$
 $w_i = 3$
 $x = 2$
 $x - w_i = -1$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 2$
 $p_i = 4$
 $w_i = 3$
 $x = 3$
 $x - w_i = 0$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 2$
 $p_i = 4$
 $w_i = 3$
 $x = 4$
 $x - w_i = 1$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 2$
 $p_i = 4$
 $w_i = 3$
 $x = 5$
 $x - w_i = 2$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 3$
 $p_i = 5$
 $w_i = 4$
 $x = 1..3$
 $x - w_i = -3..-1$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 3$
 $p_i = 5$
 $w_i = 4$
 $x = 4$
 $x - w_i = 0$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5	
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	
2	0	0	3	4	4	7	↓
3	0	0	3	4	5	7	↓
4	0						

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 3$
 $p_i = 5$
 $w_i = 4$
 $x = 5$
 $x - w_i = 1$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 4$
 $p_i = 6$
 $w_i = 5$
 $x = 1..4$
 $x - w_i = -4..-1$

Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

i \ x	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

if $w_i \leq x$

$$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$$

else

$$F[i, x] \leftarrow F[i - 1, x]$$

$i = 4$
 $p_i = 6$
 $w_i = 5$
 $x = 5$
 $x - w_i = 0$

Példa

- Az algoritmus a maximális összsúlyt vette figyelembe úgy, hogy a zsákba tehető $F[n, c]$ érték a lehető legnagyobb legyen.
- Az elemek olvasásához egy visszafele haladó algoritmus szükséges, amely a táblázatot használja:

```
i ← n, x ← c  
while (i, x > 0)
```

```
    if F[i, x] <> F[i - 1, x]
```

Jelöljük meg az i. elemet, hogy a zsákban van

```
    i = i - 1, x = x - wi
```

```
else
```

```
    i = i - 1
```

0/1 hátizsák probléma (könyv szerint)

- Adott a c kapacitású hátizsák és n objektumból álló halmaz $1, 2, \dots, n$. minden objektumot w_i , súly és p_i , profit jellemzi.
- Legyen $v = [v_1, v_2, \dots, v_n]$ egy megoldásvektor, amelyben $v_i = 0$ ha az i . objektum nincs a hátizsákból $v_i = 1$, ha a hátizsákból van.
- A cél az objektumok olyan részhalmazát megtalálni, amit a hátizsákba helyezhetünk, azaz

$$\sum_{i=1}^n w_i v_i \leq c$$

(az összsúly nem nagyobb, mint a kapacitás) és

$$\sum_{i=1}^n p_i v_i$$

a profit maximalizált.

0/1 hátizsák probléma

- Legyen a maximális profit $F[i, x]$ az x kapacitású hátizsákhöz kapcsolódóan, ha a következő objektum-részhalmazt használjuk $\{1, 2, \dots, i\}$. A DP megfogalmazása:

$$F[i, x] = \begin{cases} 0 & x \geq 0, i = 0 \\ -\infty & x < 0, i = 0 \\ \max\{F[i - 1, x], (F[i - 1, x - w_i] + p_i)\} & 1 \leq i \leq n \end{cases}$$

0/1 hátízsák probléma

- Konstruáljuk meg az $n \times c$ méretű F táblázatot soronként haladva.
- Egy elem kitöltése két elem ismeretét igényli az előző sorból: egy ugyanabból az oszlopból és egy olyan oszloptávolságnyira, mint amilyen súllyal az objektum a sorhoz tartozik.
- minden elem kiszámolása konstans idejű; a soros futási komplexitás $\Theta(nc)$.
- A megfogalmazás soros, egyszerű-argumentumú.

0/1 hátizsák probléma

F tábla

		F tábla					
		F tábla					
		F tábla					
n							
i							
2							
1							
Súlyok		1	$j - w_i$	j		$c - 1$	c
Processzorok		P_0	P_{j-w_i-1}	P_{j-1}		P_{c-2}	P_{c-1}

A 0/1 hátizsák probléma F táblájának számítása. Az $F[i,j]$ meghatározásához szükséges kommunikáció processzorelemekkel, amelyek tartalmazzák $F[i-1,j]$ és $F[i-1,j-w_i]$ elemeket.

0/1 hátízsák probléma

- Ha c processzorelemet használunk (CREW PRAM ideális, közös memóriás modellen), párhuzamos algoritmust készíthetünk az oszlopok processzorokhoz particionálásával és $O(n)$ futási idővel.
- Ha osztott memóriás modellen minden processzorelem lokálisan tárolja az objektumok súlyait és profitjait a j . iteráció során, $F[j, r]$ számításához a P_{r-1} processzoron lokálisan rendelkezésre áll $F[j-1, r]$, de $F[j-1, r-w_j]$ -t másik processzorelemtől kell megkapni. n iteráció ekkor $O(n \log c)$ ideig tart.

Nem soros, egyszerű-argumentumú DP megfogalmazás: leghosszabb közös részsorozat

- Adott egy $X = \langle x_1, x_2, \dots, x_n \rangle$ sorozat; az X részsorozatát kapjuk, ha valahány elemet törlünk belőle.
- Cél: adott két részsorozat: $X = \langle x_1, x_2, \dots, x_n \rangle$ és $Y = \langle y_1, y_2, \dots, y_m \rangle$, keressük meg a leghosszabb sorozatot, ami részsorozata egyaránt X -nek és Y -nak.
- Példa: ha $X = \langle c, a, d, b, r, z \rangle$ és $Y = \langle a, s, b, z \rangle$, a leghosszabb közös részsorozata X -nek és Y -nak $\langle a, b, z \rangle$.

Leghosszabb közös részsorozat (Longest-Common-Subsequence: LCS)

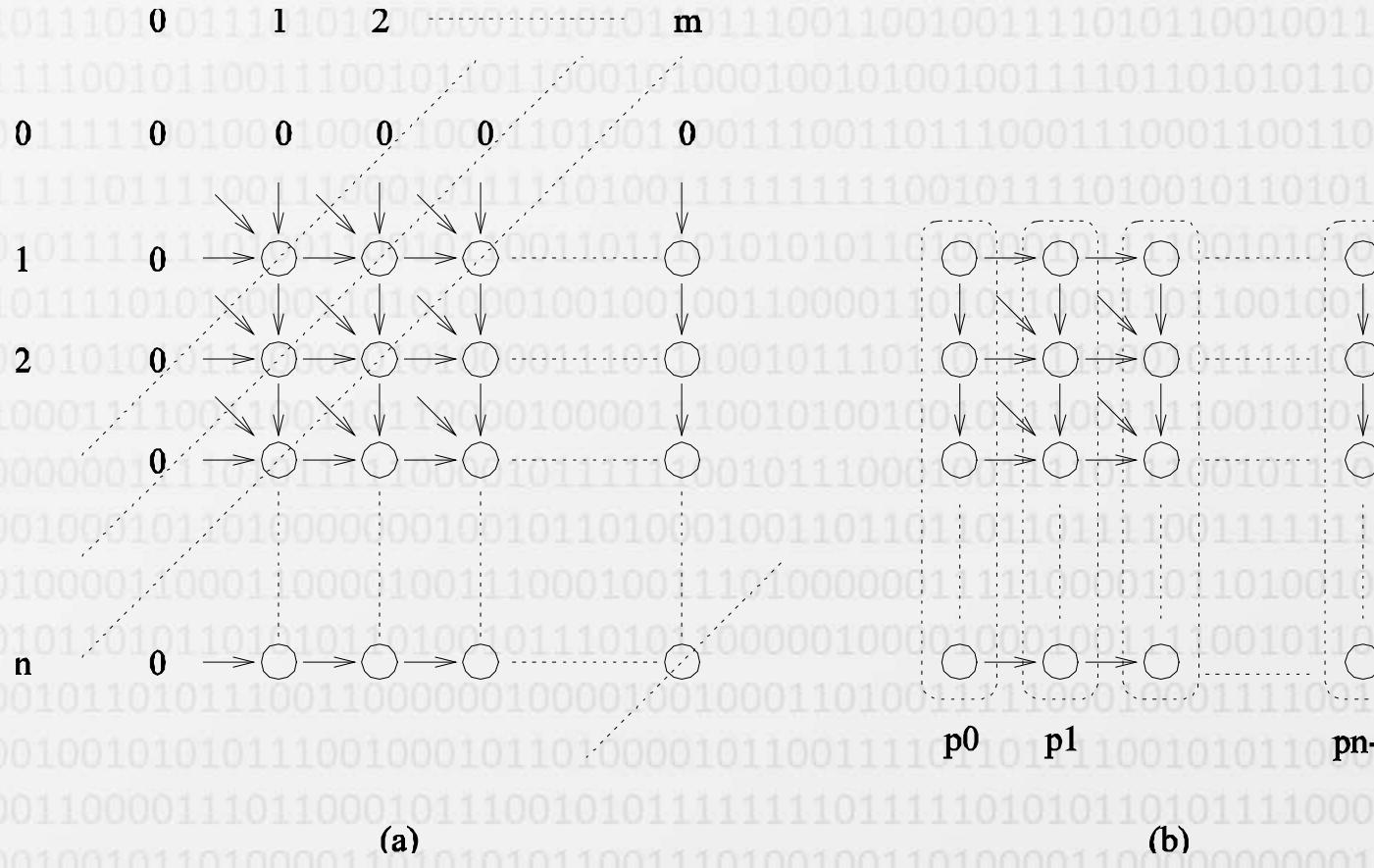
- Jelölje $F[i, j]$ az X első i elemének és Y első j elemének leghosszabb közös részsorozat hosszát. Az LCS célja, hogy megtaláljuk $F[n, m]$ értékét (és a sorozat elemeit).
- Ekkor igaz, hogy:

$$F[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ F[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max \{F[i, j - 1], F[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Leghosszabb közös részsorozat

- Az algoritmus kiszámolja a kétdimenziós F táblát sor-oszlop sorrendben.
- Az átlós csomópontok mindegyike két részproblémához kapcsolódik, az előző szinthez és az azt megelőző szinthez. Ezért ez a DP megfogalmazás nem soros, egyszerű-argumentumú típusú.

Leghosszabb közös részsorozat



(a) Az LCS táblázat számítási elemei. A számítás a jelzett átlós irányban halad. (b) A táblázat elemeinek leképzése a processzorelemekre.

Leghosszabb közös részsorozat

- Legyen két aminosavnak a szekvenciája **H E A G A W G H E E** és **P A W H E A E**, ahol A: Alanine, E: Glutamic acid, G: Glycine, H: Histidine, P: Proline és W: Tryptophan.

	H	E	A	G	A	W	G	H	E	E
H	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1	1	1
G	0	0	0	1	1	1	2	2	2	2
A	0	1	1	1	1	1	2	2	3	3
W	0	1	2	2	2	2	2	3	3	3
H	0	2	3	3	3	3	3	4	4	4
E	0	2	3	3	3	3	3	4	4	5
E	0	1	2	3	3	3	3	3	4	5

- Az LCS: **A W H E E**.

Parallel LCS

- A táblázat elemeinek kiszámolása átlós irányban történik a bal felső saroktól a jobb alsó irányban.
- Ha n processzort használunk (CREW PRAM), minden elem kiszámolása átlósan konstans ideig tart.
- A két n hosszú szekvenciával az algoritmus $2n-1$ átlós lépést tesz összesen.

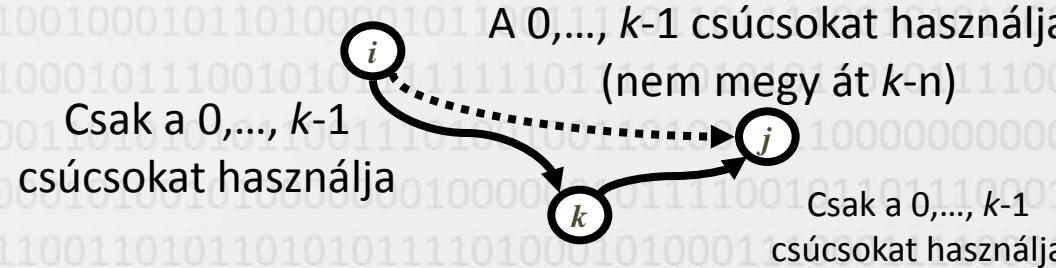
Parallel LCS

- Az algoritmus processzorok lineáris tömbjére is adaptálható: a P_i processzor elem az F tábla $(i+1)$. oszlopának meghatározásáért felelős.
- Az $F[i,j]$ kiszámolásához a P_{j-1} processzorelemnek szüksége van $F[i-1,j-1]$ vagy $F[i,j-1]$ értékre a tőle balra lévő processzorelemtől, amely kommunikációt jelent.
- A számítás konstans idejű (t_c) minden elemre.
- A teljes parallel idő:
$$T_P = (2n - 1)(t_s + t_w + t_c).$$
- A hatékonyság felső korlátja 0.5!

Soros, összetett-argumentumú DP: Floyd legrövidebb út algoritmusa, minden párra

- Adott egy súlyozott gráf $G(V,E)$, Floyd algoritmusa a csúcspárok közötti $d_{i,j}$ költségeket határozza meg, hogy a legrövidebb utat célozza meg.
- Legyen $d_{i,j}^k$ a minimum költségű út az i csomóponttól a j csomópontig, és csak a következő csúcsokat használja v_0, v_1, \dots, v_{k-1} .
- Így:

$$d_{i,j}^k = \begin{cases} c_{i,j} & k = 0 \\ \min \{d_{i,j}^{k-1}, (d_{i,k}^{k-1} + d_{k,j}^{k-1})\} & 0 \leq k \leq n - 1 \end{cases} .$$



Nem soros összetett-argumentumú DP megfogalmazás: optimális mátrix zárójelezési probléma

- Amikor mátrixok sorozatát kell összeszorozni, akkor a szorzás sorrendje alapvetően meghatározza a műveletek (szorzások) számát.
- Legyen $C[i,j]$ (egy résznél $N[i,j]$ a jelölés!) az $A_i \dots A_j$ mátrixok optimális szorzásának költsége.
- A láncszorzást két kisebb lánc szorzására lehet bontani, A_i, A_{i+1}, \dots, A_k és A_{k+1}, \dots, A_j .
- Az A_i, A_{i+1}, \dots, A_k eredménye $d_i \times d_{k+1}$ méretű mátrix, az A_{k+1}, \dots, A_j lánc eredménye $d_{k+1} \times d_{j+1}$ méretű mátrix.
- A két mátrix összeszorzásának költsége (a szorzások száma) $d_i d_{k+1} d_{j+1}$.

Mátrixok láncszorzása

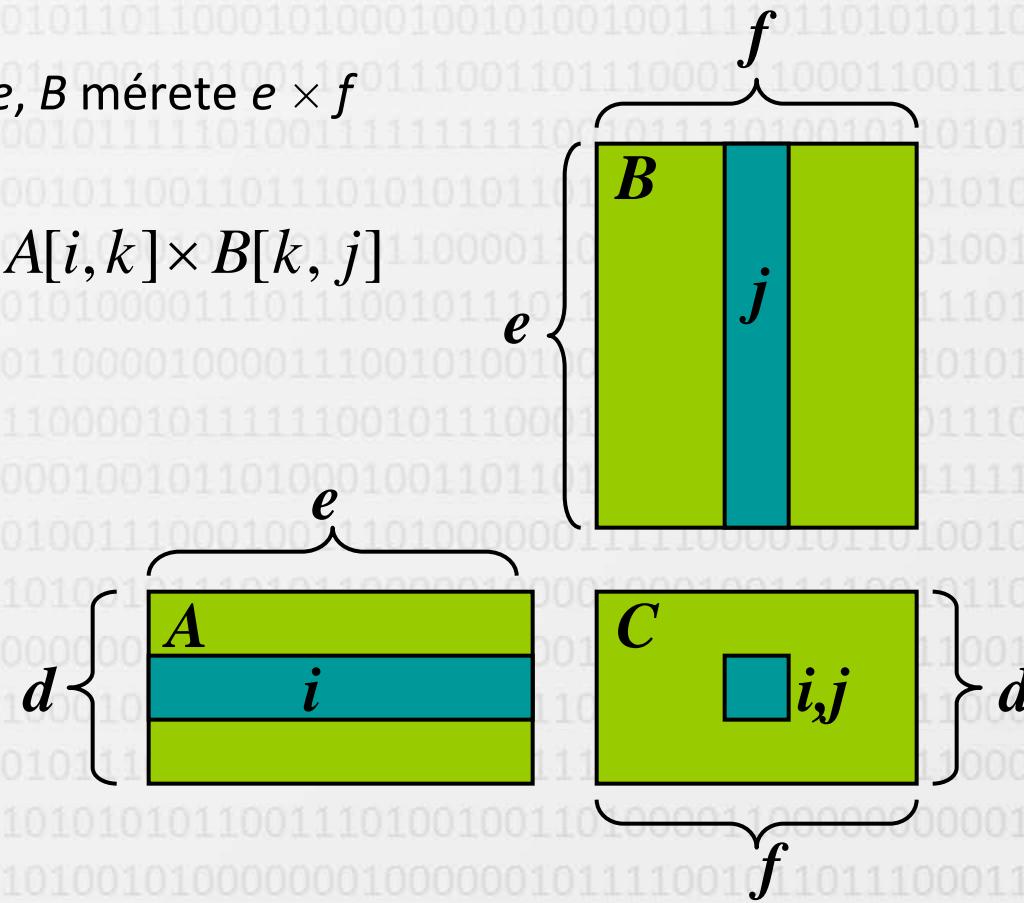
• Emlékeztető: mátrix-szorzás

- $C = AB$

- A mérete $d \times e$, B mérete $e \times f$

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] \times B[k, j]$$

- $O(def)$ idő



Mátrixok láncszorzása

- **Mátrixok láncszorzása:**

- számítandó $A = A_0 A_1 \dots A_{n-1}$
- A_i mérete $d_i \times d_{i+1}$
- feladat: hogyan zárójelezzük?

- **Példa:**

- B: 3×100
- C: 100×7
- D: 7×5
- $(BC)D$ $3 \times 100 \times 7 + 3 \times 7 \times 5 = 2305$ szorzás
- $B(CD)$ $3 \times 100 \times 5 + 100 \times 7 \times 5 = 5000$ szorzás

Felsorolásos megközelítés

- **Mátrixok láncszorzása algoritmus:**

- próbáljuk minden lehetséges módon zárójelezni
 $A = A_0 A_1 \dots A_{n-1}$
- számoljuk ki minden esetre a szorzások számát
- a legjobbat vegyük

- **Futási idő:**

- a zárójelezések száma = az n csomópontú bináris fáki számával azonos
 - Exponenciálisan nő
- gyakorlatban nem megvalósítható

Egy mohó algoritmus

Ötlet: a legkevesebb műveletet használó szorzatokat válasszuk ki elsőként.

Példa:

- A: 101×11
- B: 11×9
- C: 9×100
- D: 100×99
- Ötlet szerint: A((BC)D) $109989+9900+108900=228789$ szorzás
- A legjobb (AB)(CD) $9999+89991+89100=189090$ szorzás

Másik mohó algoritmus

Ötlet: a legtöbb műveletet használó szorzatokat válasszuk ki elsőként.

Példa:

- A: 10×5
- B: 5×10
- C: 10×5
- D: 5×10
- Ötlet szerint $(AB)(CD)$ $500+1000+500 = 2000$ szorzás
- Legjobb $A((BC)D)$ $500+250+250 = 1000$ szorzás

Egy rekurzív megközelítés

- **Definiálunk részproblémákat:**

- Keressük meg az $A_i A_{i+1} \dots A_j$ legjobb zárójelezését.
- Legyen $N_{i,j}$ = a műveletek száma ennél a részfeladatnál.
- Az egész feladatra optimális megoldás $N_{0,n-1}$.

- **A részfeladat-optimalizálás: az optimális megoldást optimális részproblémákként lehet definiálni.**

- Kell, hogy legyen néhány legvégül futó szorzás (a kifejezésfa gyökere) az optimális megoldásnál.
- Mondjuk, a végső szorzás az i indexnél van:
 $(A_0 \dots A_i)(A_{i+1} \dots A_{n-1})$.
- Ekkor az optimális megoldás $N_{0,n-1}$ két optimális részfeladat megoldása: $N_{0,i}$ és $N_{i+1,n-1}$, valamint az utolsó szorzás költsége.
- Ha a globális optimumnak nem ezek az optimális részproblémái, akkor tudunk definiálni még jobb „optimális” megoldást.

Karakterisztikus egyenlet

- A globális optimális megoldást optimális részproblémákként kell definiálni, attól függően, hogy hol van az utolsó szorzás.
- Tételezzük fel az összes lehetséges helyet a végső szoratra:
 - az A_i mérete $d_i \times d_{i+1}$
 - így a karakterisztikus egyenlet $N_{i,j}$ -re a következő:

$$N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- Ezek a részfeladatok nem függetlenek, hanem átfedésben vannak egymással.

Dinamikus programozási algoritmus

- Mivel a részfeladatok átfedik egymást, nem használunk rekurziót.
- Helyette „bottom-up” módon építkezünk.
- $N_{i,i}$ -k meghatározása egyszerű, ezzel kezdünk.
- Ezután 2, 3,... részfeladat.
- Futási idő: $O(n^3)$.

Algorithm $\text{matrixChain}(S)$:

Input: a szorzandó mátrixok S sorozata

Output: az S optimális zárójelezésének a száma

for $i \leftarrow 1$ to $n-1$ **do**

$N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ to $n-1$ **do**

for $i \leftarrow 0$ to $n-b-1$ **do**

$j \leftarrow i+b$

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ to $j-1$ **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

Dinamikus programozás

- A bottom-up konstrukció átlósan tölti ki az N tömböt.
- $N_{i,j}$ értékeket kap az előző i . sor és j . oszlop elemeiből.
- A tábla elemeinek kitöltése $O(n)$ idejű.
- Teljes futási idő: $O(n^3)$.
- A zárójelezést úgy kaphatjuk meg, ha eltároljuk „ k ”-t N minden elemére.

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

N	0	1	2	\dots	j	\dots	$n-1$	válasz
0								
1								
\dots								
i								
$n-1$								

Láncszorzás algoritmus

Algorithm *matrixChain*(*S*):

Input: a szorzandó mátrixok *S* sorozata

Output: az *S* optimális zárójelezésének száma

for *i* $\leftarrow 0$ **to** *n-1* **do**

N_{i,i} $\leftarrow 0$

for *b* $\leftarrow 1$ **to** *n-1* **do** //b a műveletek száma

for *i* $\leftarrow 0$ **to** *n-b-1* **do**

j $\leftarrow i+b$

N_{i,j} $\leftarrow +\text{infinity}$

for *k* $\leftarrow i$ **to** *j-1* **do**

 sum = *N_{i,k}* + *N_{k+1,j}* + *d_i d_{k+1} d_{j+1}*

if (sum < *N_{i,j}*) **then**

N_{i,j} \leftarrow sum

O_{ij} $\leftarrow k$

return *N_{0,n-1}*

- **Példa: ABCD**

- A: 10×5

- B: 5×10

- C: 10×5

- D: 5×10

<i>N</i>	0	1	2	3
0	0	500	500	1000
1	A	AB	A(BC)	(A(BC))D
2	0	250	500	1
3	B	BC	(BC)D	
4	0	0	500	0
5	C		CD	
6	0		D	

A műveletek visszanyerése

- Példa: ABCD

- A: 10×5
- B: 5×10
- C: 10×5
- D: 5×10

N	0	1	2	3
0	0	500	500	1000
1	A	AB	A(BC)	(A(BC))D
2	B	250	0	500
3	C	BC	(BC)D	0
4	D			0

// $A_i \dots A_j$ mátrixlánc szorzásából

// a visszatérő érték

exp(i,j)

if (i=j) then // alapeset, 1 mátrix

return 'A_i'

else

k = O[i,j] // vörös értékek

S1 = exp(i,k) // 2 rekurzív hívás

S2 = exp(k+1,j)

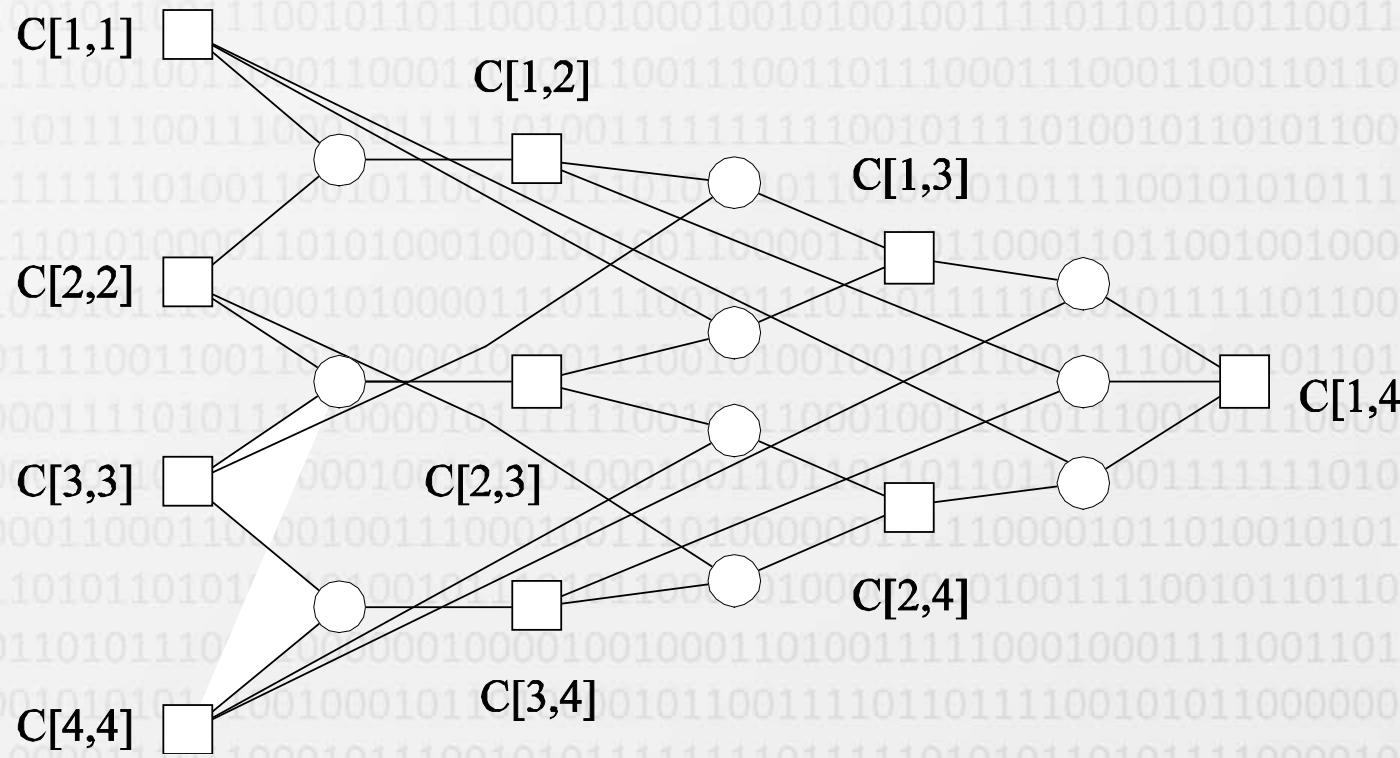
return ('(S1 S2)')

Optimális mátrix láncszorzás

- Könyv szerint (ami eddig N volt mostantól C):

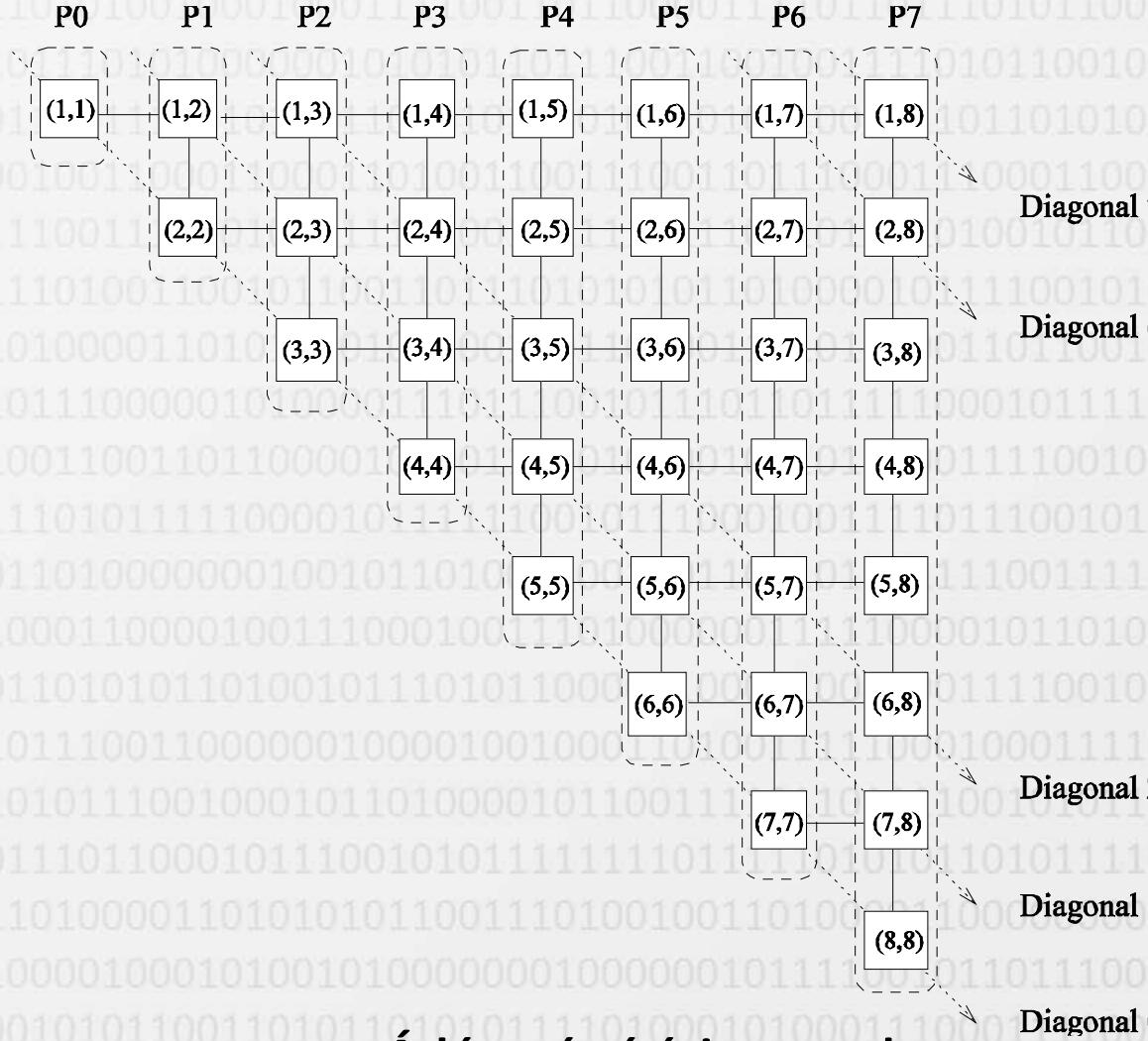
$$C[i, j] = \begin{cases} \min_{i \leq k < j} \{C[i, k] + C[k + 1, j] + r_{i-1} r_k r_j\} & 1 \leq i < j \leq n \\ 0 & j = i, 0 < i \leq n \end{cases}$$

Optimális mátrix láncszorzás



Nem soros, összetett-argumentumú DP megfogalmazás mátrixok láncszorzásának optimális meghatározására. A négyzetek a mátrixok láncszorzatának optimális költségét jelentik. A körök lehetséges zárójelezést. $C[i, j]$ jelölés azonos a korábbi $N_{i,j}$ -vel

Optimális mátrix láncszorzás



Átlós számítási sorrend

Párhuzamosítási lehetőség

- **Tételezzünk fel egy gyűrűs logikai processzortopológiát. Az 1. lépésben minden processzorelem egyetlen elemet számol ki, amely az 1. átlóhoz tartozik.**
- **A C tábla kiszámítása során minden processzor elküldi broadcasting módon az általa kiszámolt elemeket az összes többi processzornak.**
- **A következő érték lokálisan számítható.**

Az általános dinamikus programozási technika

- Általában olyan feladatoknál alkalmazzuk, amelyek első ránézésre rengeteg időt vesznek igénybe.

Részei:

- **egyszerű részproblémák:** a részproblémákat néhány változó függvényeként kell definiálni.
- **részprobléma optimalitás:** a globális optimum a részproblémák optimumaként definiálható.
- **átfedő, kapcsolódó részproblémák:** a részfeladatok nem függetlenek, hanem átfedőek (ezért bottom-up konstrukcióban kell dolgozni).

Párhuzamos dinamikus programozási algoritmusok

- A számítási menetet gráfként reprezentálva háromféle párhuzamosítási lehetőséget azonosíthatunk:
csomópontokon belüli párhuzamosság; egy szinten a csomópontok közötti párhuzamosítás és futószalagosított csomópontok (pipelining) eltérő szintű csomópontok között.
Az első kettő soros megfogalmazású, a harmadik nem soros DP.
- **Az adatok helyzete (hová rendeljük) a teljesítmény szempontjából kritikus.**

Képfeldolgozás és párhuzamosíthatóság

A képfeldolgozás olyan alkalmazási terület, amely számos lehetőséget biztosít párhuzamosításra.

Témakörök:

Alacsonyszintű műveletek: simítás, hisztogram, éldetektálás

Hough-transzformáció egyenes detektáláshoz

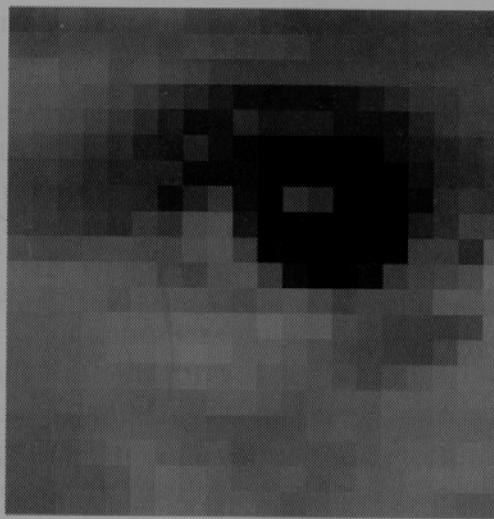
Leképzési technikák és terheléskiegysúlyozás

B. Wilkinson, M. Allen: „Parallel Programming”, Pearson Education
Prentice Hall, 2nd ed., 2005, a könyv 12. fejezete alapján

Mi a képfeldolgozás?

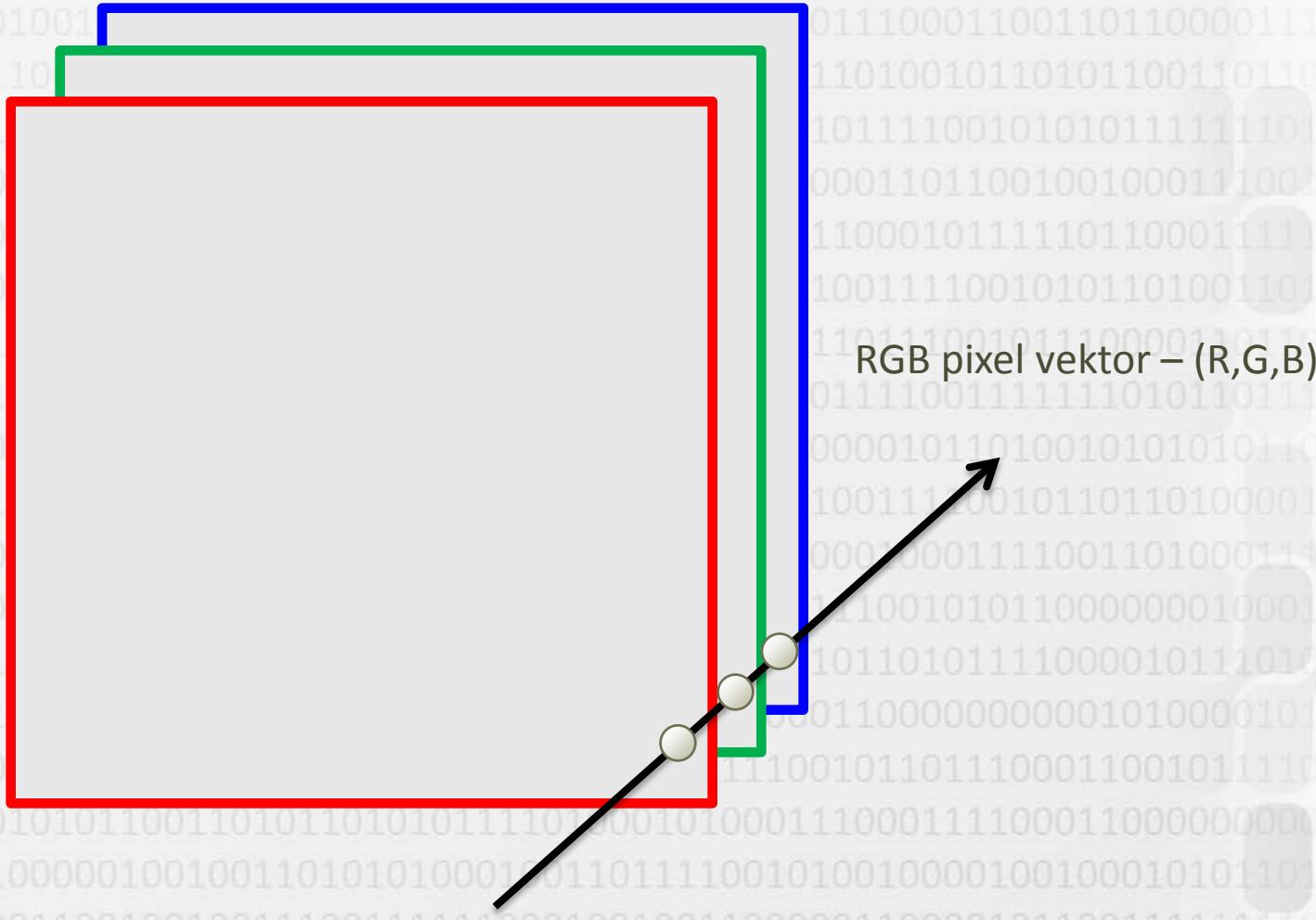
- **A képfeldolgozás a jelfeldolgozás része, amely képekkel foglalkozik.**
- **Célja: a kép minőségének javítása az ember vagy további számítógépes feldolgozás számára.**
- **Kép – képfeldolgozás – „jobb” kép**

Képek ábrázolása



117	125	133	127	130	130	133	121	116	115	100	91	93	94	99	103	112	105	109	106
134	133	138	138	132	134	130	133	128	123	121	113	106	102	99	106	113	109	109	113
146	147	138	140	125	134	124	115	102	96	93	94	99	96	99	100	103	110	109	110
144	141	136	130	120	108	88	74	53	37	31	37	35	39	53	79	93	100	109	116
139	136	129	119	102	85	58	31	41	77	51	53	53	33	37	41	69	94	105	108
132	127	117	102	87	57	49	77	42	28	17	15	13	13	17	41	53	69	88	100
124	120	108	94	72	74	72	31	35	31	15	13	15	11	15	13	46	75	83	96
125	115	102	93	88	82	42	79	113	41	19	100	82	11	11	17	31	91	99	100
124	116	109	99	91	113	99	140	144	57	20	20	15	11	15	17	63	87	119	124
136	133	133	135	138	133	132	144	150	120	24	17	15	15	17	20	115	113	88	150
158	157	157	154	149	145	133	127	146	150	116	35	20	19	28	105	124	128	141	171
155	154	156	155	146	155	154	154	147	139	148	150	138	120	128	129	130	151	156	165
150	151	154	162	166	167	169	174	172	167	177	166	164	140	134	120	121	120	127	172
145	149	151	157	165	169	173	179	176	166	166	157	145	136	129	124	120	136	163	168
144	148	153	160	159	158	165	172	165	169	157	151	149	141	130	140	151	162	169	167
144	141	147	155	154	149	156	151	157	157	151	144	147	147	149	159	158	159	166	165
139	140	140	150	153	151	150	146	140	139	138	140	145	151	149	156	156	162	162	161
136	134	138	146	156	164	153	146	145	136	139	139	140	141	149	157	159	161	169	166
136	133	136	135	144	159	168	159	151	142	141	145	139	146	153	156	164	167	172	168
133	129	140	142	146	159	167	165	154	151	146	141	147	154	156	160	161	157	153	154

Színes képek



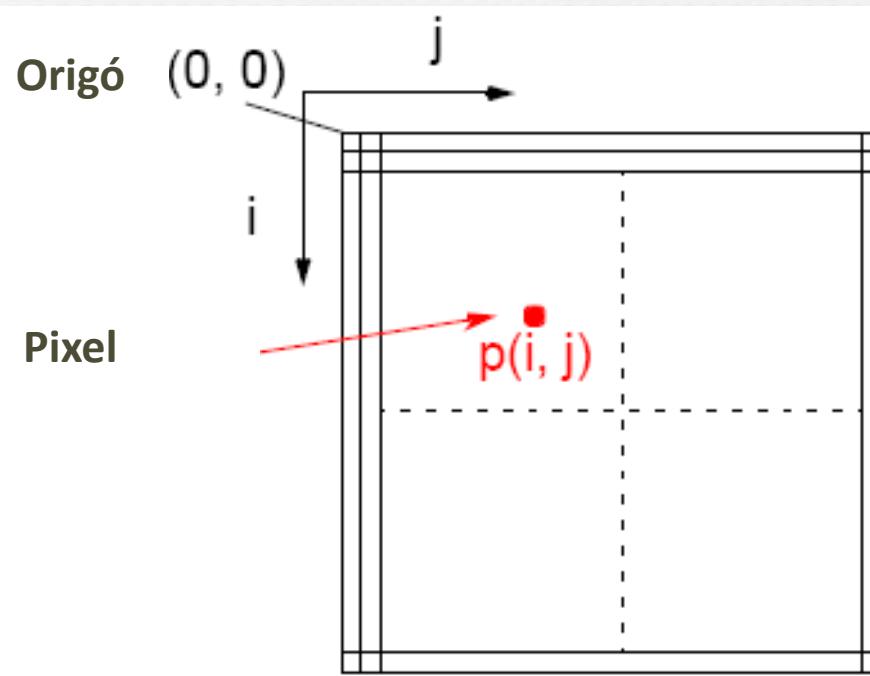
A digitális képfeldolgozás szintjei

A képek számítógépes feldolgozását három szintre lehet osztani: alacsony, közép- és magas szintű feladatok (low-level, middle-level, high-level):

- **low-level:** mind az input, mind az output kép;
- **middle-level:** az inputok általában képek, de az outputok a képekből nyert attribútumok (pl. egy objektum azonosítói a képen);
- **high-level:** a felismert objektumok együttesének érzékelése.

Alacsonyszintű képfeldolgozás

- A tárolt képen operál, hogy javítsa/módosítsa azt.
- A kép képelemek (pixel = picture element) kétdimenziós tömbje.



- Számos alacsonyszintű képművelet az intenzitásokat (szürkeségi értékeket) manipulálja.

Számítási igény

- Tételezzünk fel egy 1024×1024 pixelből álló képet, ahol az intenzitást 8 biten tároljuk.
- Tárolási igény 2^{20} byte (1 Mbytes).
- Tegyük fel, hogy minden pixelen csak egy műveletet végzünk, ekkor 2^{20} operációt kell végeznünk a képen. Ez kb. 10^{-8} mp/operáció, ami hozzávetőlegesen 10 ms.
- Valós idejű képfeldolgozás esetén tipikusan 25-30 képet kell másodpercenként feldolgozni (fps).
- Tipikusan nem csak egyetlen operációt kell végezni a pixeleken, hanem több és összetettebb funkciókat.

Pont alapú műveletek

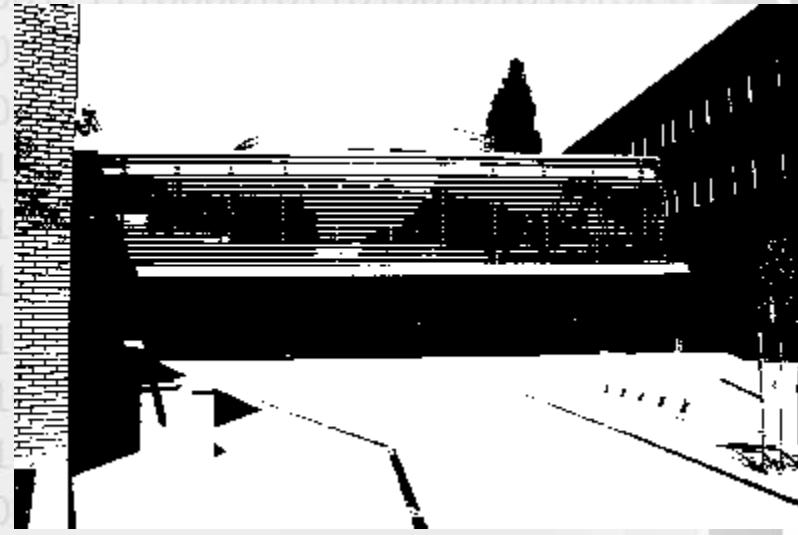
Olyan műveletek, ahol az output csak egy képponttól függ.

Küsztöölés:

- egy bizonyos határnál (threshold) nagyobb intenzitású képpont értékéhez az egyik szélsőértéket, ellenkező esetben a másik szélsőértéket rendeljük hozzá.

Ha egy pixel intenzitása x_i , akkor minden pixelre:

$\text{if } (x_i < \text{threshold}) x_i = 0; \text{ else } x_i = 255;$



Pont alapú műveletek

Kontrasztnyújtás:

- **az intenzitásértékek tartományát széthúzzuk, hogy a részletek jobban érzékelhetőek legyenek.** Adott egy pixel intenzitása x , az $x_h - x_l$ tartományból, a kontrasztnyújtás során az $x_H - x_L$ tartományba transzformáljuk az intenzitást:

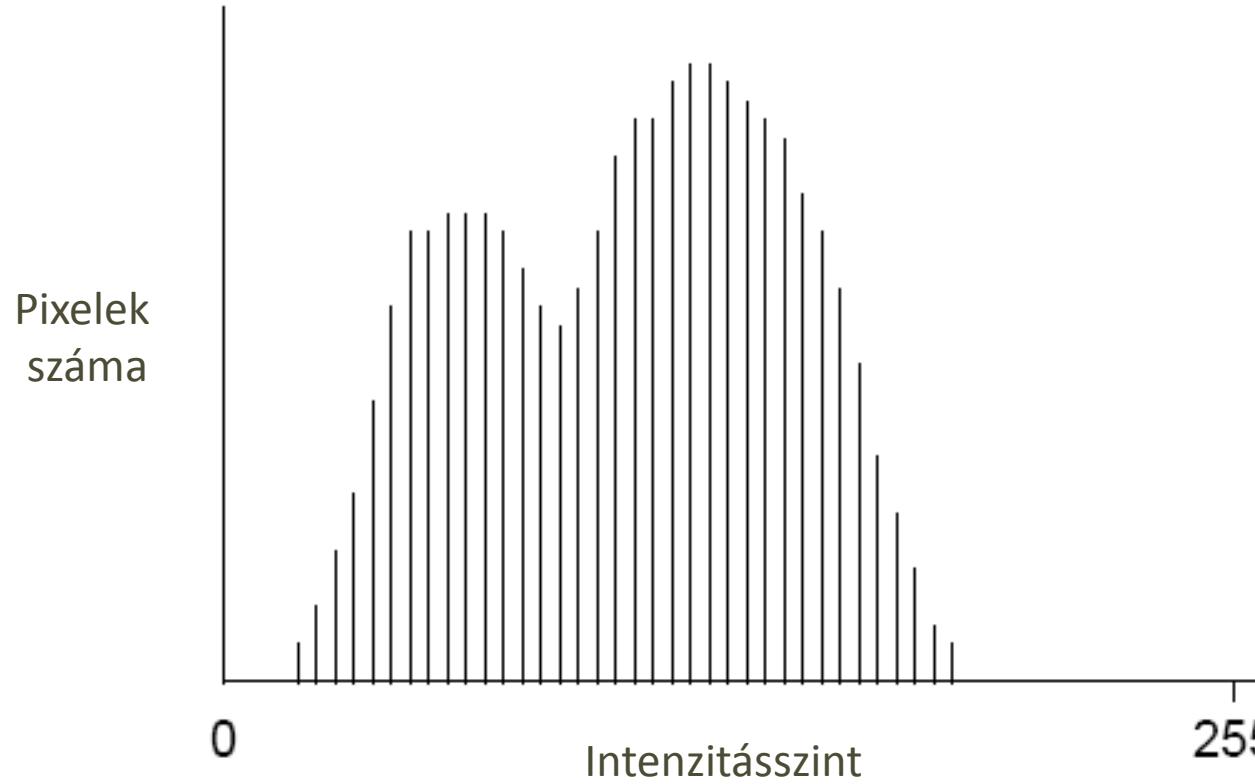
$$x_I = \frac{(x_H - x_L)}{(x_h - x_l)} x_i + x_L$$

Szürkeségi szint csökkentés:

- **a kevésbé szignifikáns biteket elhagyjuk.**

Hisztogram

- Az egyes intenzitásokból hány darab van a képen



Hisztogram soros kód

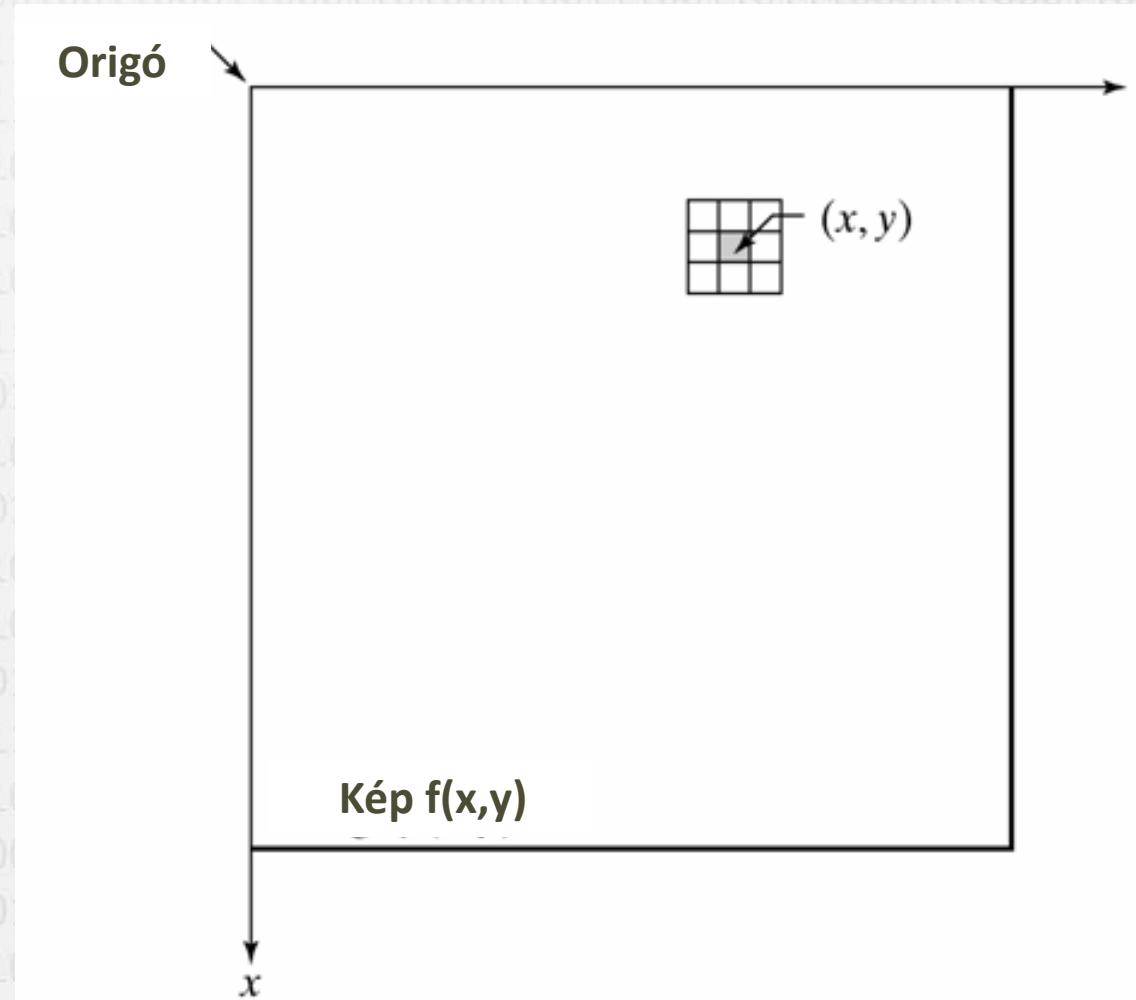
```
for(i = 0; i < height_max; x++)
    for(j = 0; j < width_max; y++)
        hist[p[i][j]] = hist[p[i][j]] + 1;
```

ahol a pixeleket a p[][] tömb tárolja, és a hist[k] vektor megmondja, hogy a k-ik intenzitásból hány darab van a képen.

- Egyszerű összegző tömb.
- Könnyen párhuzamosítható dekompozícióval.

Maszk-alapú műveletek

- $g(x, y) = T[f(x, y)]$, T a szomszédos pixeleken operál (lokális művelet)



Maszk-alapú műveletek

Simítás (zajszűrés):

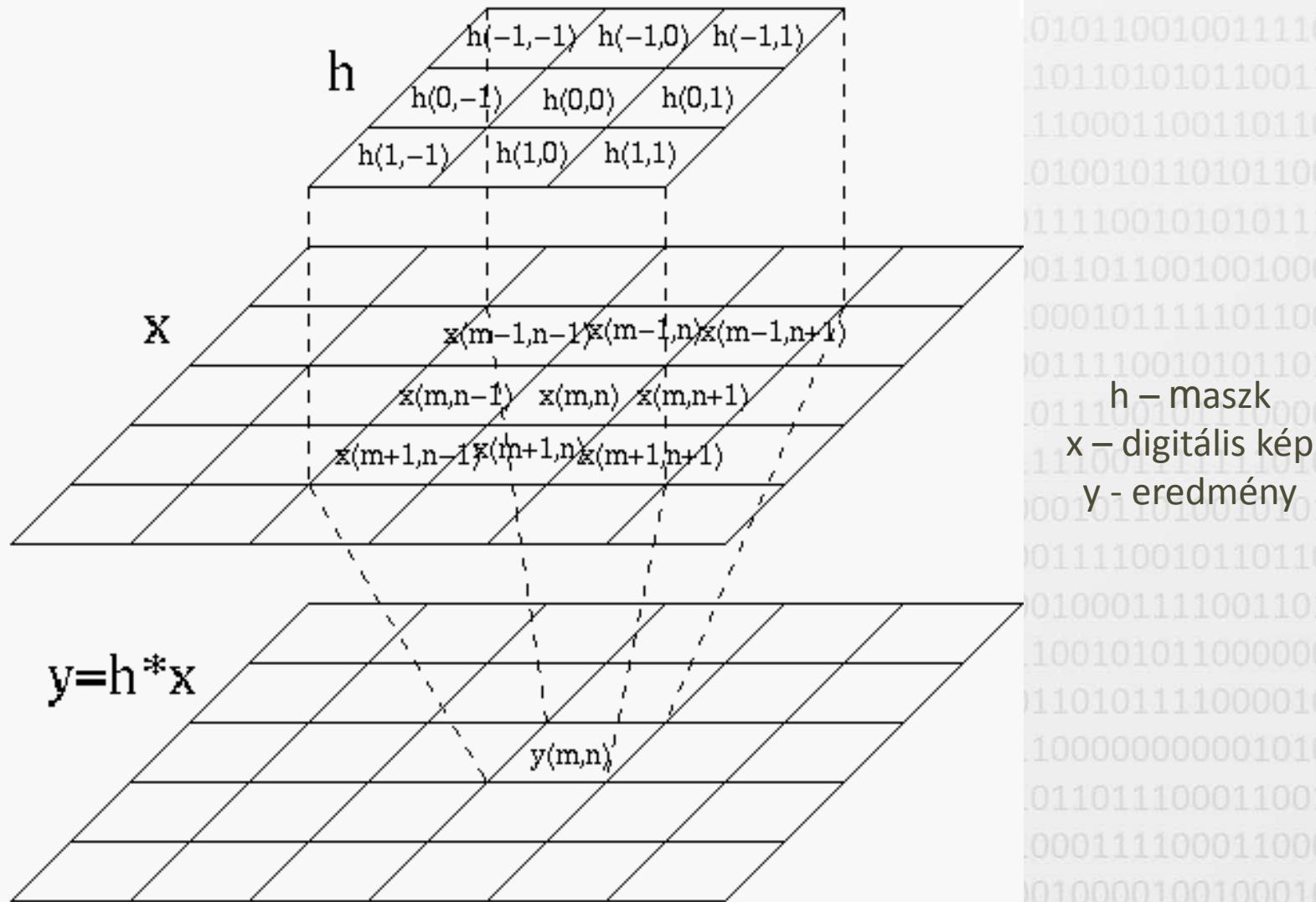
- az intenzitás nagy változásait simítjuk el, a magas-frekvenciás tartalom csökkentése.**



Élesítés:

- a részletek kiemelése.**

Maszk használata



Maszk

- Ablak alapú műveletekhez gyakran alkalmaznak $n \times n$ méretű maszkot ($n = 3, 5, 7, \dots$).

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

Átlagoló szűrő

- **Egyeszerű simítási technika, ahol az ablakban lévő intenzitások átlaga az új intenzitásérték:**

$$x_4 = \frac{x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{9}$$

- **Soros kód:**

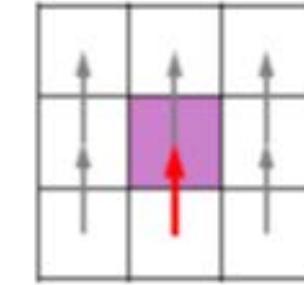
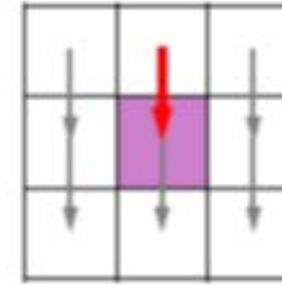
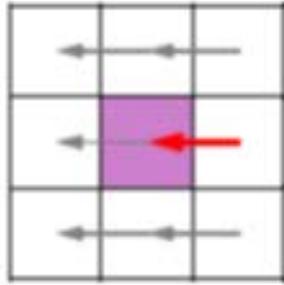
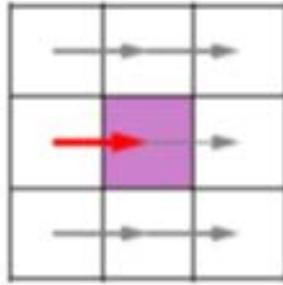
kilenc (+1) lépés kell az átlag kiszámításához, n pixelre $9n$.

Komplexitás: $O(n)$.

Párhuzamos átlagoló szűrő

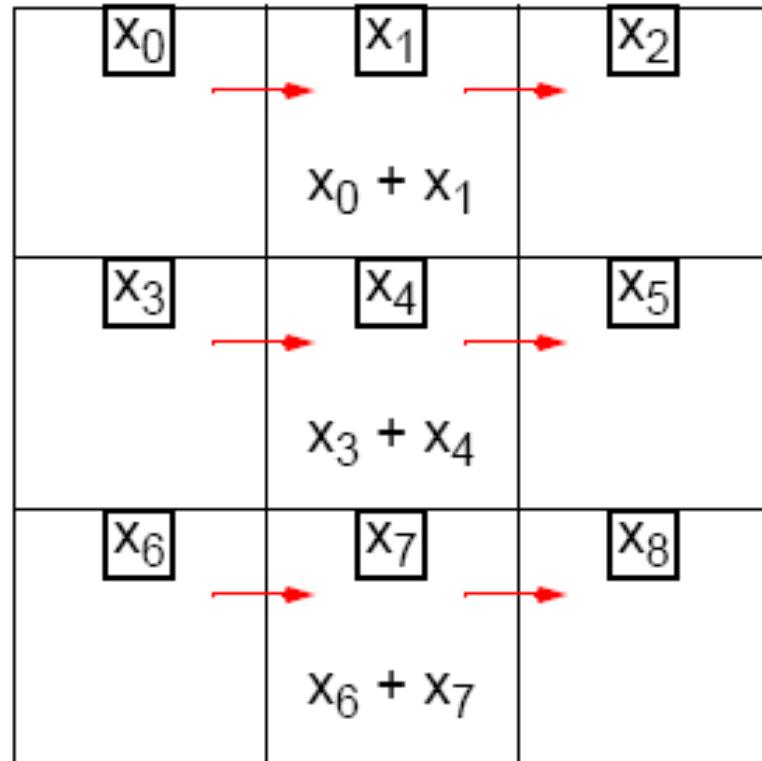
- A műveletek száma redukálható négyre.

– Elv, amely pontosításra kerül minden általánosított pixelről.

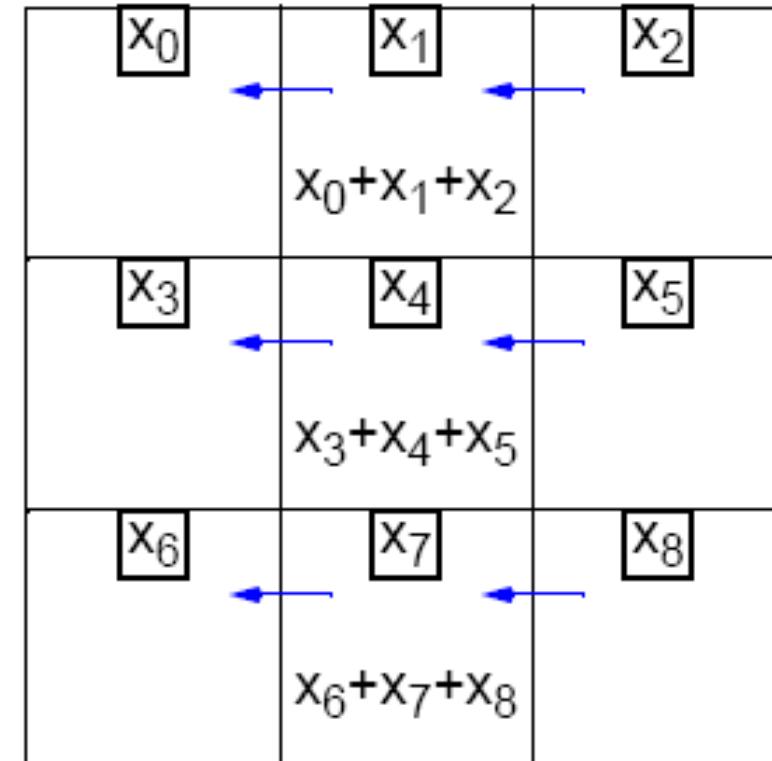


– minden pixelt összeadunk balról, jobbról, felülről és alulról.

Párhuzamos átlagoló szűrő I.

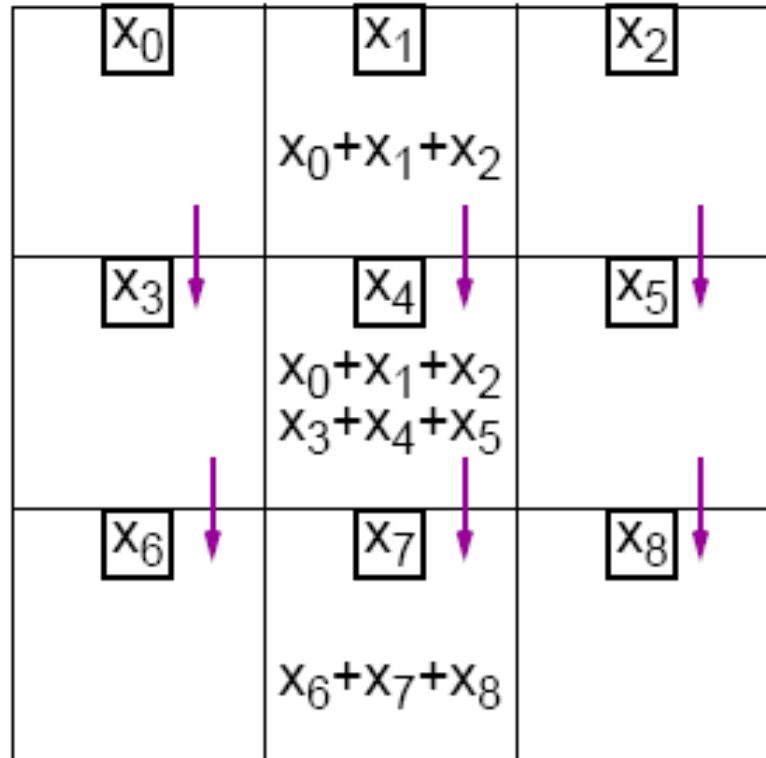


(a) Első lépés

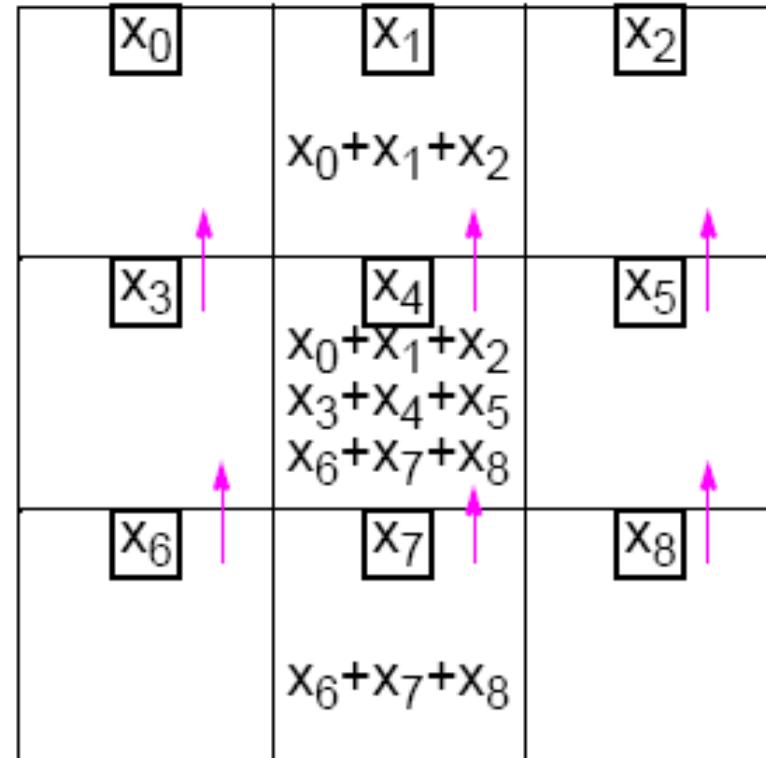


(b) Második lépés

Párhuzamos átlagoló szűrő II.



(c) Harmadik lépés



(d) Negyedik lépés

Medián szűrő

Soros megvalósítás:

- **a medián meghatározása érdekében rendezni kell a pixelértékeket és a középsőt kell kiválasztani.**
 - Például 3 x 3-as esetben a rendezett értékek: $y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7$, és y_8 . A medián y_4 .
 - Az ötödik elemet kell kivenni a rendezés után.
 - Pl. buborékos rendezésnél a műveletek (összehasonlítás, és ha kell, csere) száma: $8 + 7 + 6 + 5 + 4 = 30$ lépés, azaz n pixelre $30n$ művelet.

Közelítő medián szűrő

Párhuzamos megvalósítás:

- elsőként a soron belül hajtsunk végre három összehasonlítást és cserét:

$$p_{i,j-1} \leftrightarrow p_{i,j}$$

$$p_{i,j} \leftrightarrow p_{i,j+1}$$

$$p_{i,j-1} \leftrightarrow p_{i,j}$$

ahol \leftrightarrow jelenti, hogy hasonlítsd össze és cseréld fel, ha a baloldali érték nagyobb, mint a jobboldali.

- ezután oszlopokra vonatkozóan három lépés:

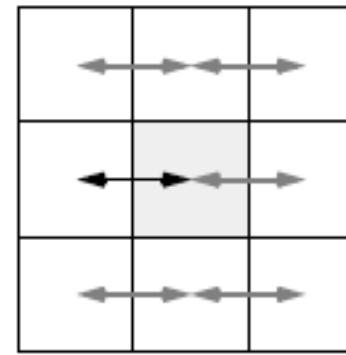
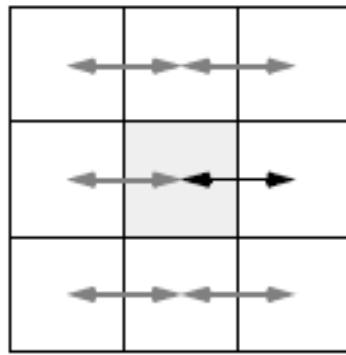
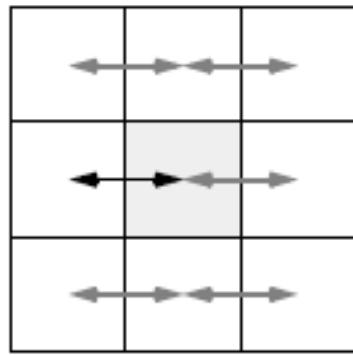
$$p_{i-1,j} \leftrightarrow p_{i,j}$$

$$p_{i,j} \leftrightarrow p_{i+1,j}$$

$$p_{i-1,j} \leftrightarrow p_{i,j}$$

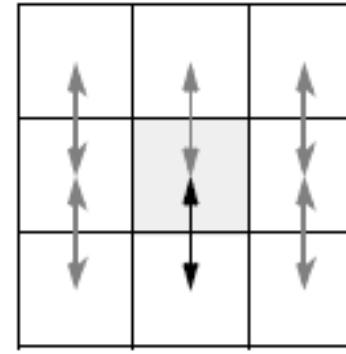
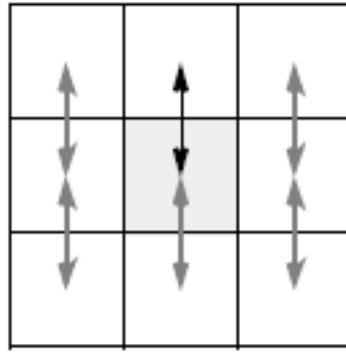
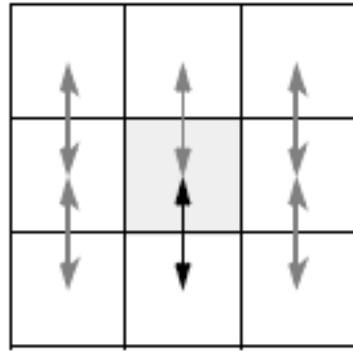
- Összesen hat lépés.
- Mikor nem pontos?

Közelítő medián szűrő



A
legnagyobb
a sorban

A következő
legnagyobb a
sorban



A következő
legnagyobb az
oszlopban

Hibadiffúziós algoritmus (Floyd, 1975)

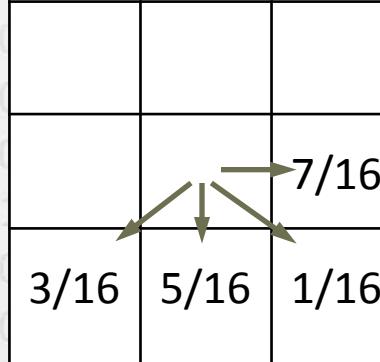
- Folytonos tónusú kép megjelenítése korlátosztott színtartományú eszközön (pl. 8-bites szürke kép fekete-fehér nyomtatón).
- Közelítő technikával lehet szimulálni az árnyalatokat.



- Bal oldali 8 bit felbontású, jobb oldali 2 bit felbontású kép.

Algoritmus

- Az aktuális képpont értéke alapján az output meghatározása: ha $[0, 127] \Rightarrow 0$; ha $[128, 255] \Rightarrow 1$.**
- Hibameghatározás: pl. ha az input 168 volt, akkor az output: 1, a hiba pedig az 1 reprezentációjának megfelelő 255 és az eredeti érték különbsége, azaz $168 - 255 = -87$.**
- A hibaérték terjesztése a szomszédos képpontokra:**



Algoritmus (pszeudó kód)

```
1 for each y fentről lefele
2   for each x balról jobbra
3     korábbi_érték := pixel[x][y]
4     új_érték := A_legközelebbi_érték_a_cél_színskáláján (korábbi_érték)
5     pixel[x][y] := új_érték
6     hiba := korábbi_érték - új_érték
7     pixel[x+1][y] := pixel[x+1][y] + 7/16 * hiba
8     pixel[x-1][y+1] := pixel[x-1][y+1] + 3/16 * hiba
9     pixel[x][y+1] := pixel[x][y+1] + 5/16 * hiba
10    pixel[x+1][y+1] := pixel[x+1][y+1] + 1/16 * hiba
```

Párhuzamosított hibadiffúzió

- Fordított jellegű megfogalmazás, mivel az előző pixelek intenzitásából generált hibát (a, b, c, d) kell ismerni:



Párhuzamosított hibadiffúzió 3.

- A hibaterjesztéshez három elem kell az előző sorból, és egy a megelőző oszloból => két képpontnyi késés!



Feldolgozott pixel

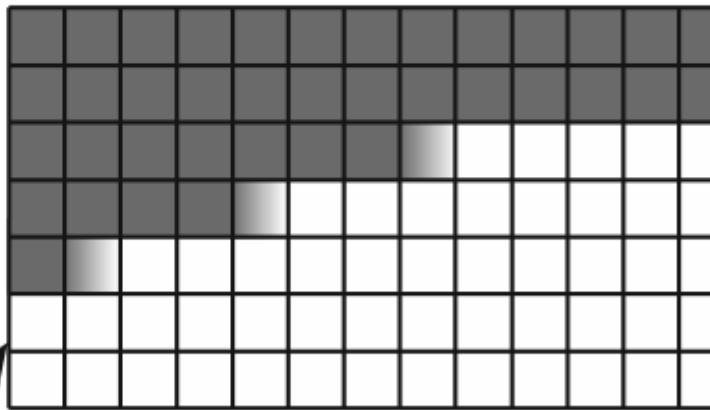
Feldolgozás alatt álló pixel

Még nem feldolgozott pixel

Az első szál által feldolgozott sor
A második szál által feldolgozott sor
A harmadik szál által feldolgozott sor

Az oldal bal széle

Több szál több sort is fel tud dolgozni egymástól függetlenül.



Súlyozó maszkok

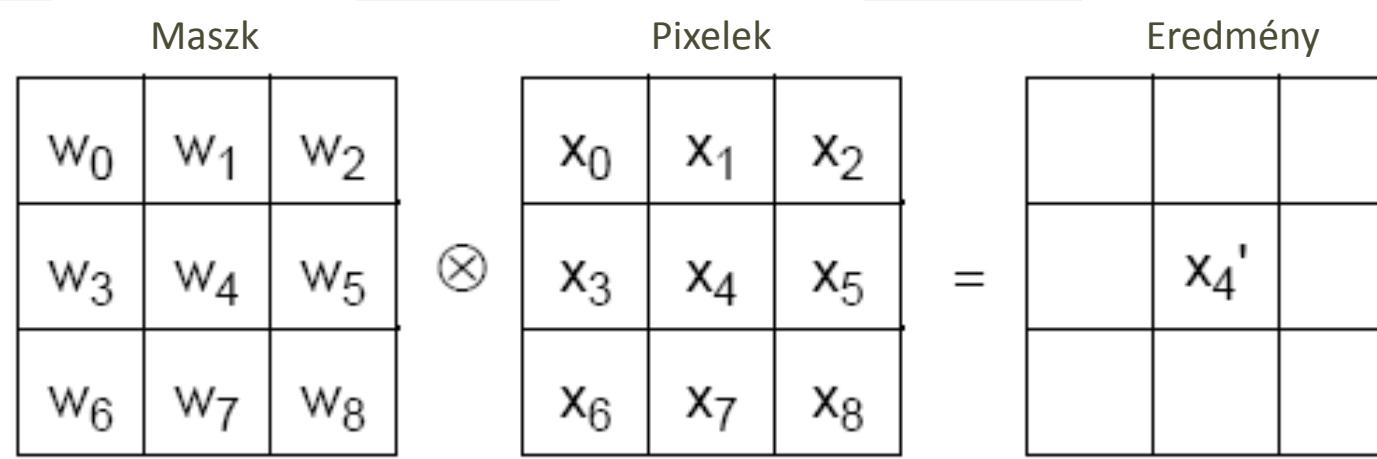
- Gyakran nem egységnyi súlyúak a maszkelemek: $w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7$, és w_8 .
Az új intenzitásérték x'_4 :

$$x'_4 = \frac{w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6 + w_7 x_7 + w_8 x_8}{k}$$

ahol $1/k$ skálázási tényező az intenzitás értékét állítja be, és k nagysága gyakran a súlyok összegével egyezik meg.

Kereszt-korreláció

- 3 x 3-as súlyozó maszk használatával



azaz a súlyozott összeg ($w_i x_i$) két függvénynek (f és w) a (diszkrét) kereszt-korrelációja: $f \otimes w$.

Maszkok

- **Átlagoló**

1	1	1
1	1	1
1	1	1

$k = 9$

- **Zajszűrő**

1	1	1
1	8	1
1	1	1

$k = 16$

- **Felülátereszítő-élesítő**

-1	-1	-1
-1	8	-1
-1	-1	-1

$k = 9$

Éldetektálás

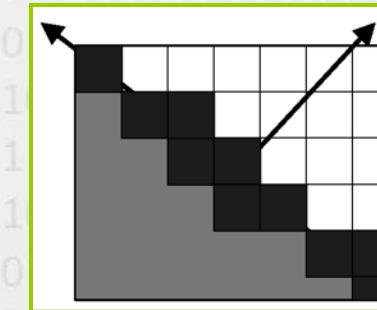
- Ahol az intenzitásban jelentős változás van, ott található él.
- Kép deriváltja:

$$\frac{\partial f(x, y)}{\partial x} = \lim_{\varepsilon \rightarrow 0} \left(\frac{f(x + \varepsilon, y) - f(x, y)}{\varepsilon} \right) \quad \frac{\partial f(x, y)}{\partial y} = \lim_{\varepsilon \rightarrow 0} \left(\frac{f(x, y + \varepsilon) - f(x, y)}{\varepsilon} \right)$$

$$\frac{\partial f(x, y)}{\partial x} \approx \frac{f(x_{n+1}, y_m) - f(x_n, y_m)}{\Delta x} \quad \frac{\partial f(x, y)}{\partial y} \approx \frac{f(x_n, y_{m+1}) - f(x_n, y_m)}{\Delta x}$$

- Gradiens:

$$grad(f) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$



Éldetektálás differenciálással

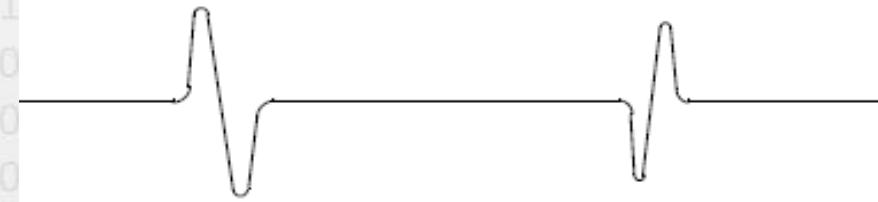
- **Intenzitás**



- **Első derivált**



- **Második derivált**



Gradiens nagyság és irány

- Gradiens nagyság:

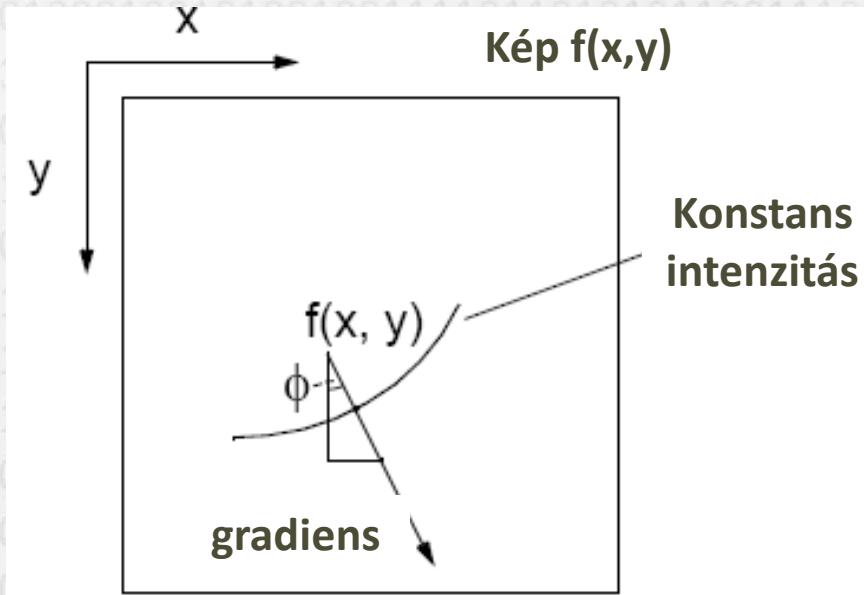
$$|\nabla f| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

- Gradiens irány:

$$\varphi(x, y) = \arctan \left\{ \frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}} \right\}$$

- Gradiens nagyságra más mérték:

$$|\nabla f| = \left| \frac{\partial f}{\partial x} \right| + \left| \frac{\partial f}{\partial y} \right|$$



Éldetektáló maszkok

- A gradiens összetevőinek közelítő számolása:

$$\left| \frac{\partial f}{\partial x} \right| = (x_5 - x_3)$$

$$\left| \frac{\partial f}{\partial y} \right| = (x_7 - x_1)$$

- A gradiens nagyság:

$$|\nabla f| = \left| \frac{\partial f}{\partial x} \right| + \left| \frac{\partial f}{\partial y} \right| = |x_7 - x_1| + |x_5 - x_3|$$

0	-1	0
0	0	0
0	1	0

0	0	0
-1	0	1
0	0	0

Prewitt operátor

- A gradiens összetevőinek közelítő számolása:

$$\left| \frac{\partial f}{\partial x} \right| = (x_2 - x_0) + (x_5 - x_3) + (x_8 - x_6)$$

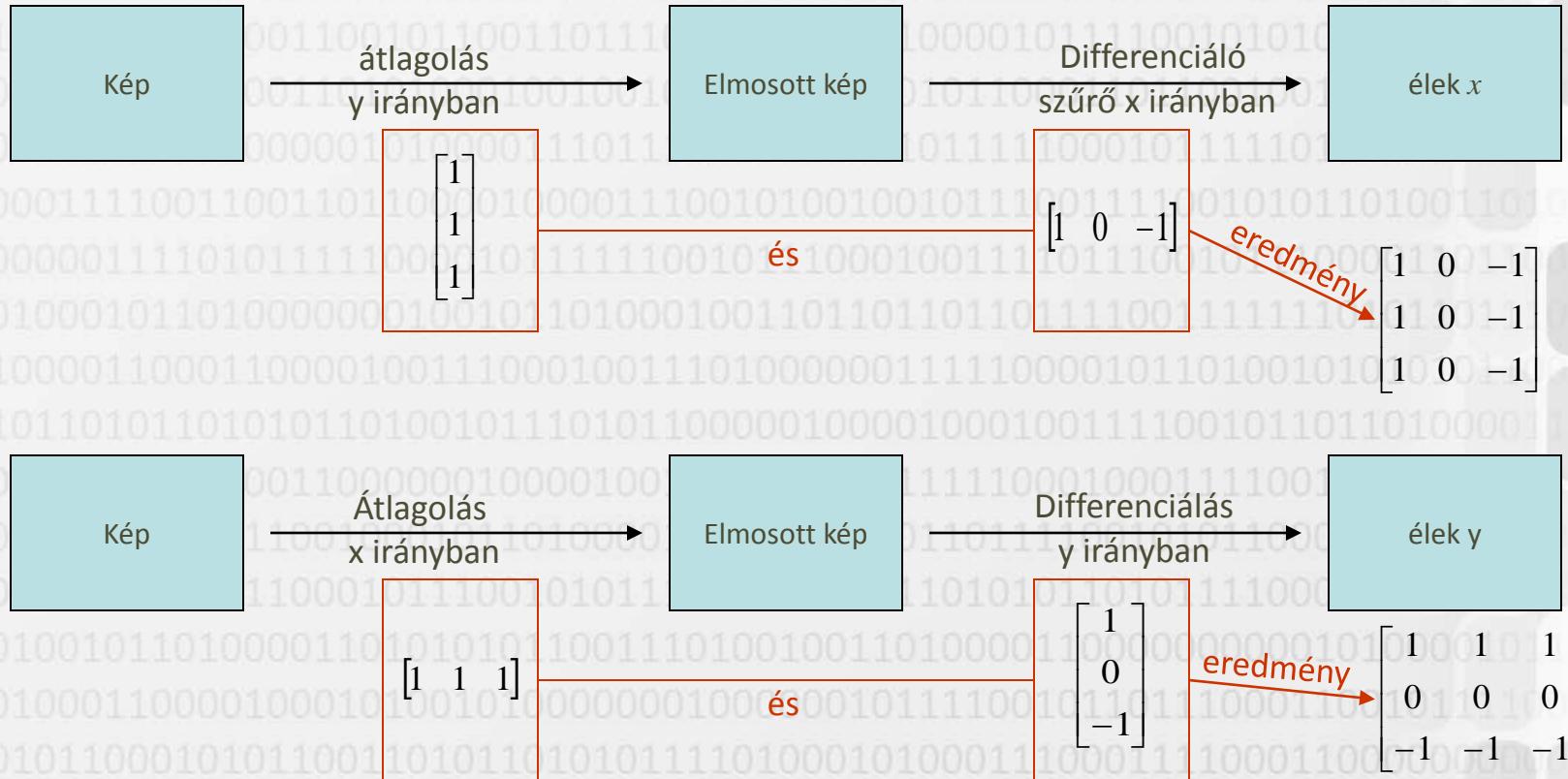
$$\left| \frac{\partial f}{\partial y} \right| = (x_6 - x_0) + (x_7 - x_1) + (x_8 - x_2)$$

- A Prewitt-maszk:

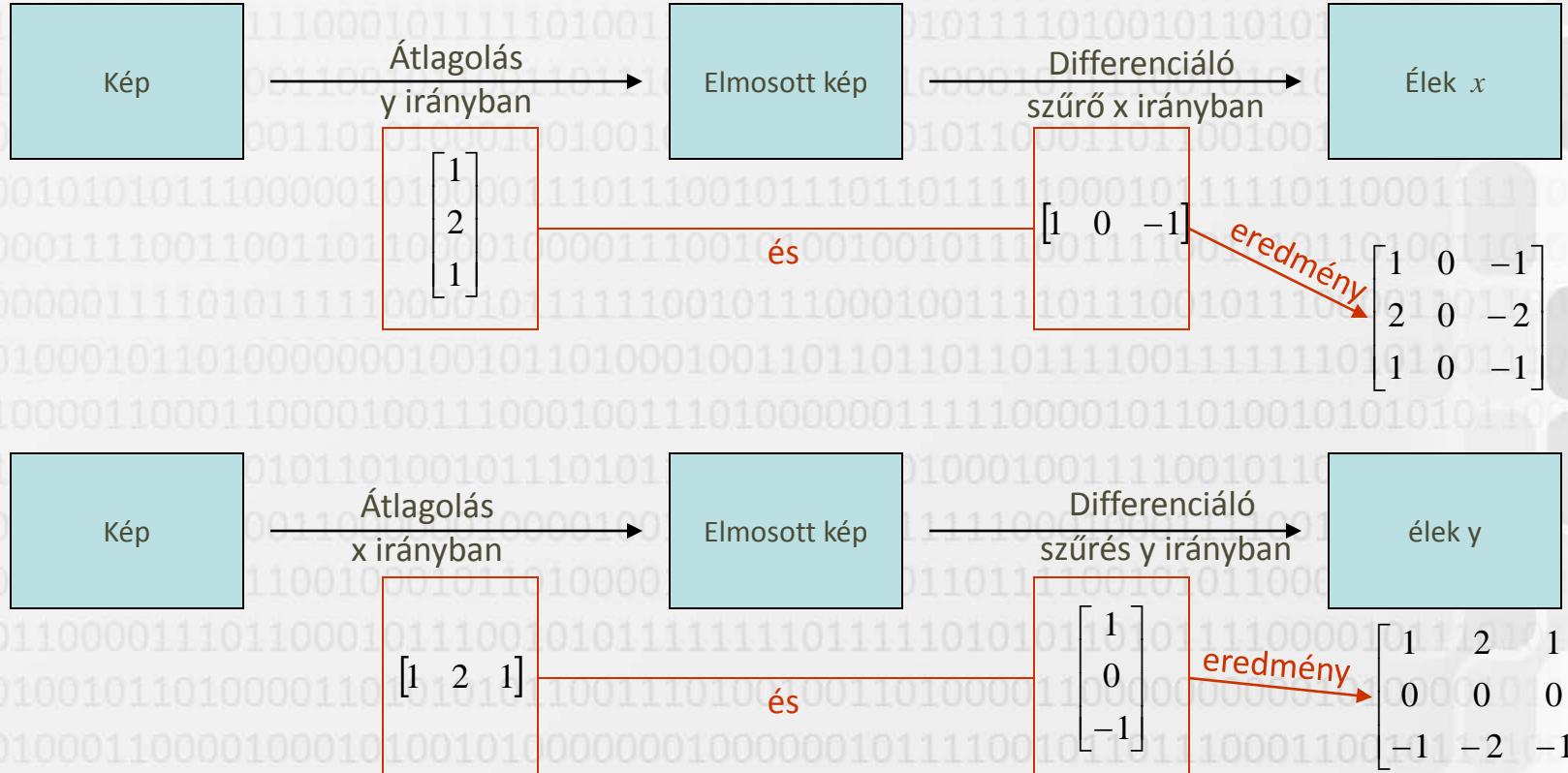
-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

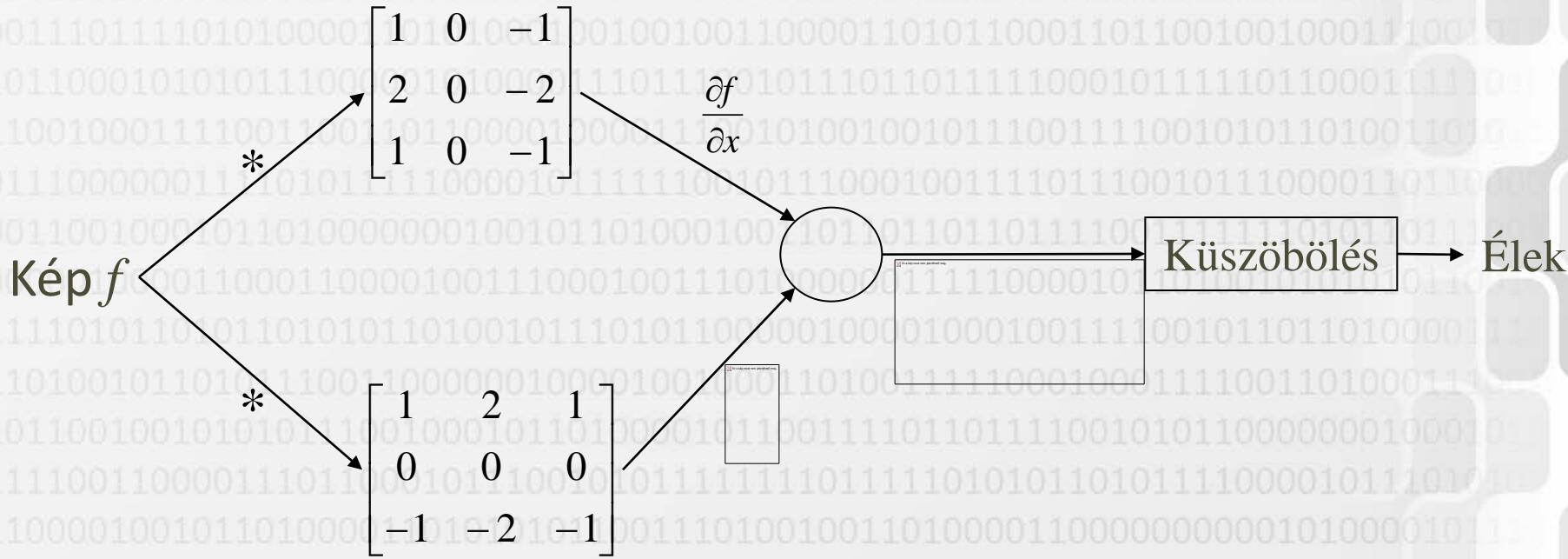
Prewitt éldetektor



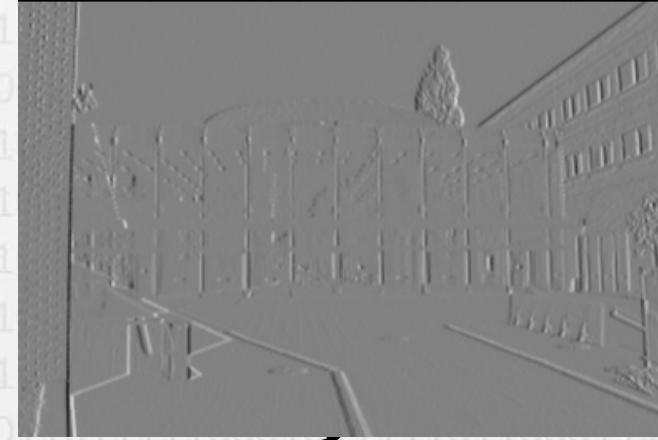
Sobel éldetektáló



Sobel éldetektáló

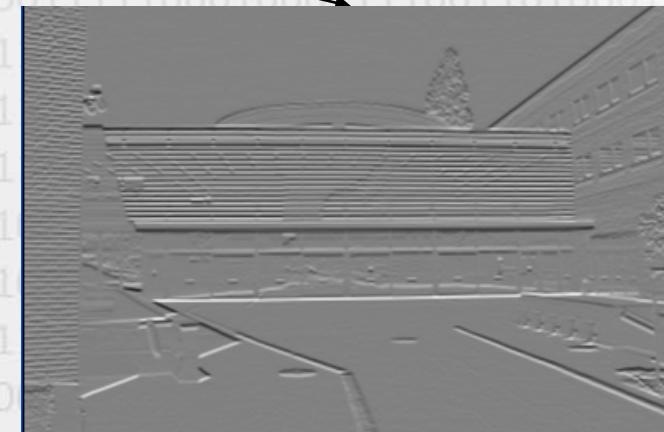


Sobel éldetektáló



$$\frac{\partial f}{\partial x}$$

$$\frac{\partial f}{\partial y}$$

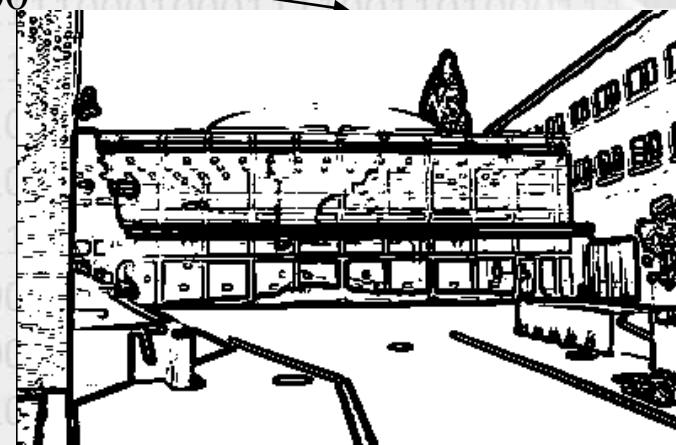


Sobel éldetektáló

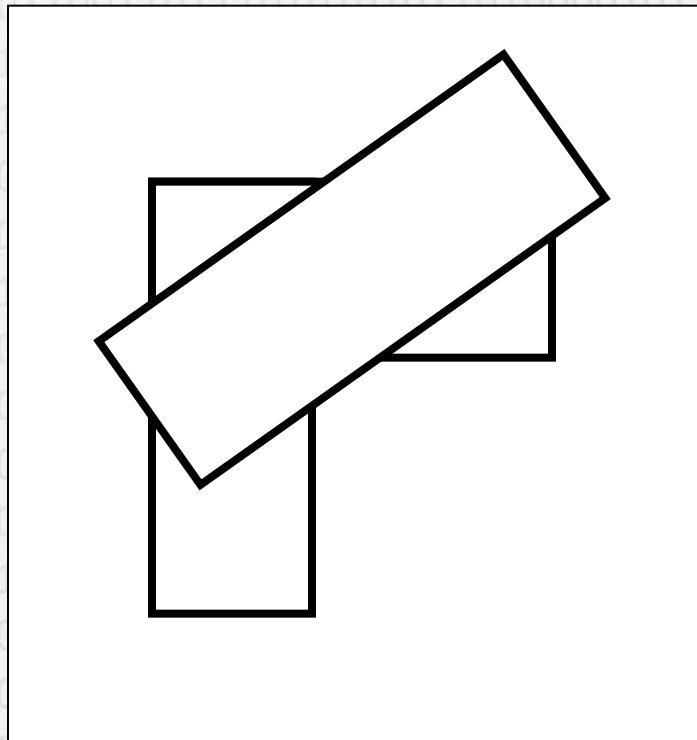


$$\nabla = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

$$\nabla \geq \text{Threshold} = 100$$

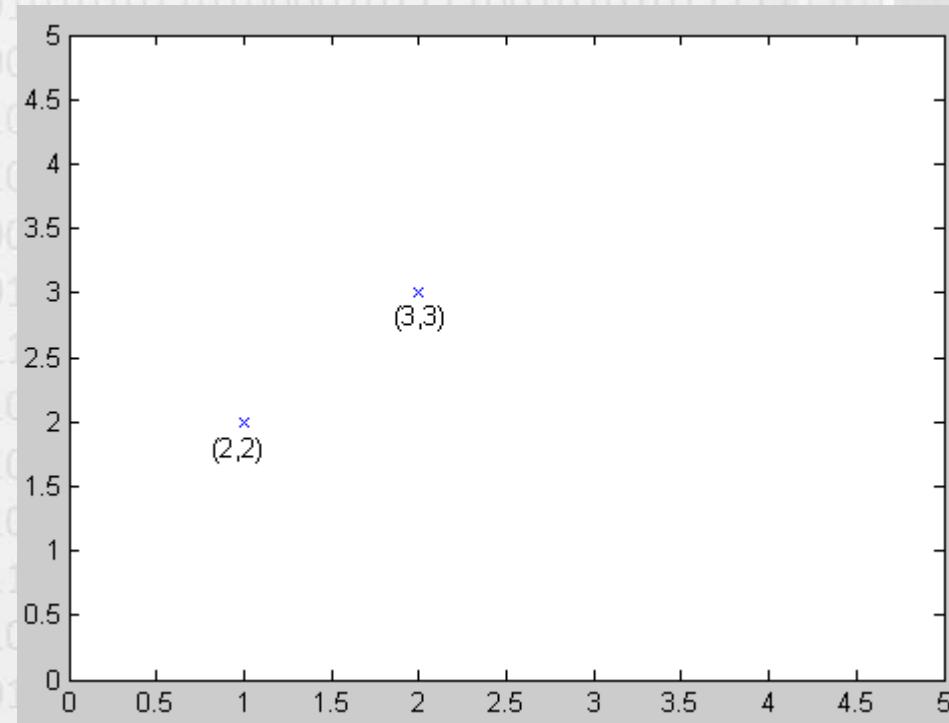


Vonaldetektálás



Vonaldetektálás Hough-transzformációval

- Cél: az $f(x, y)$ képen találjuk meg a vonalakat és határozzuk meg azok egyenletét.
- $O(NNMM)$ nagyságrendű számítást kell elvégezni.



Vonaldetektálás Hough-transzformációval

Főbb pontok:

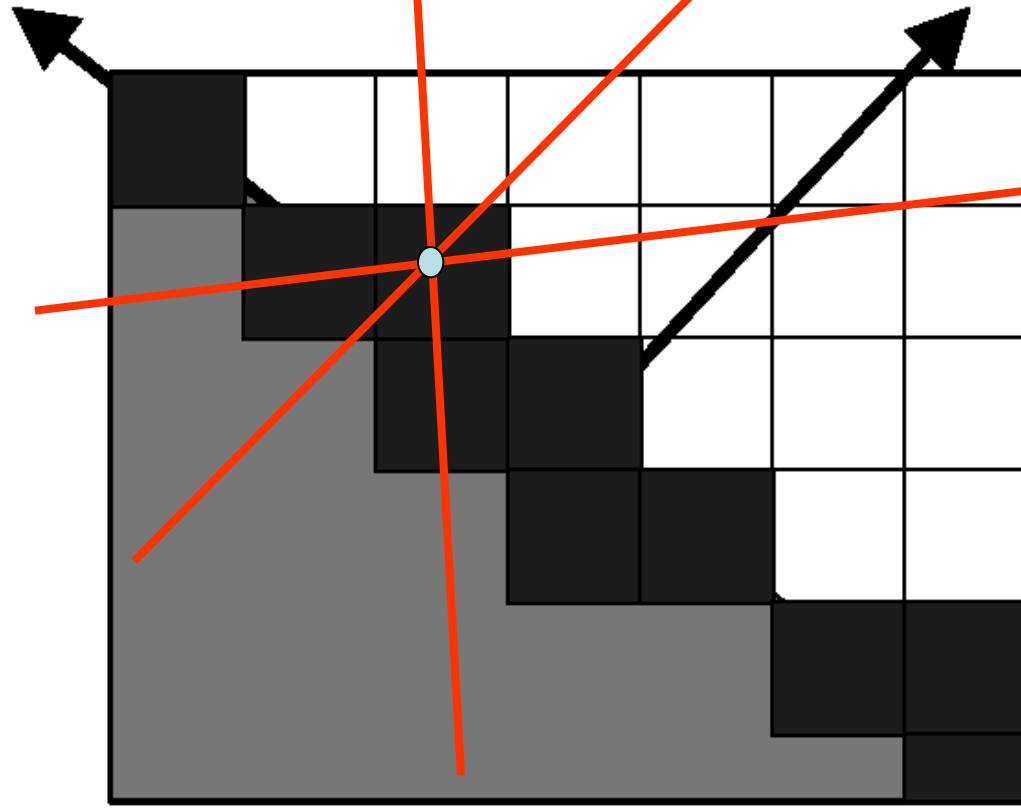
- **konverzió paraméter térbe;**
- **az m és n paraméterek megtalálása;**
- **visszakonvertálás derékszögű koordinátákba.**

Vonaldetektálás Hough-transzformációval

- **Kulcs:** használjuk a paraméterteret, ahol a bonyolult probléma az egyszerűbb lokális maximumok megtalálását jelenti.
- **Input:**
 - bináris kép élpontokkal;
 - küszöb.

Hough-transzformáció

$$b = -mx + y \quad b' = -m'x + y$$



$$b'' = -m''x + y$$

Vonalillesztés

- **Vonalegyenlet**

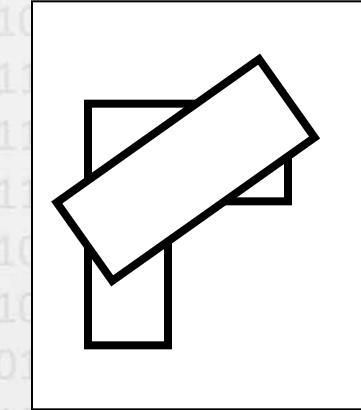
$y = mx + b$ m meredekség, b az y -metszéspont

- **Az (m, b) teret osszuk fel egy ráccsal, és minden cellához rendeljünk egy számlálót: $c(m, b)$ kezdetben 0 értékkel.**

- **Minden élpixel ismert koordinátáival**

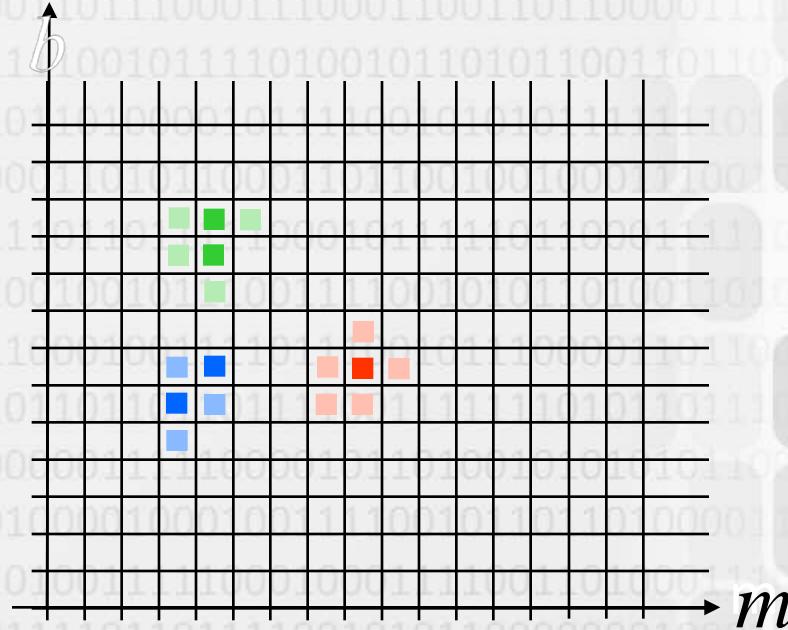
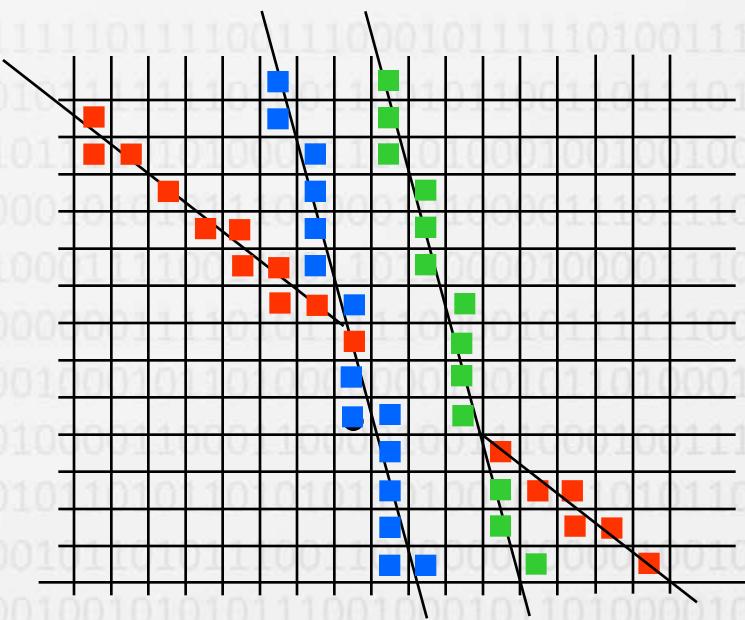
- számoljuk ki b értékét minden lehetséges m mellett; $b_i = y - m_j x$
 - növeljük meg a $c(m_i, b_i)$ -t eggyel!

- **Keressük meg a lokális maximumokat a paramétertérből!**



Hough-trafó: kvantálás

- Vonaldetektálás maximum/klaszter keresésével a paramétertérben.



- Függőleges vonalak esetén problema:
 - m és b végtelen.

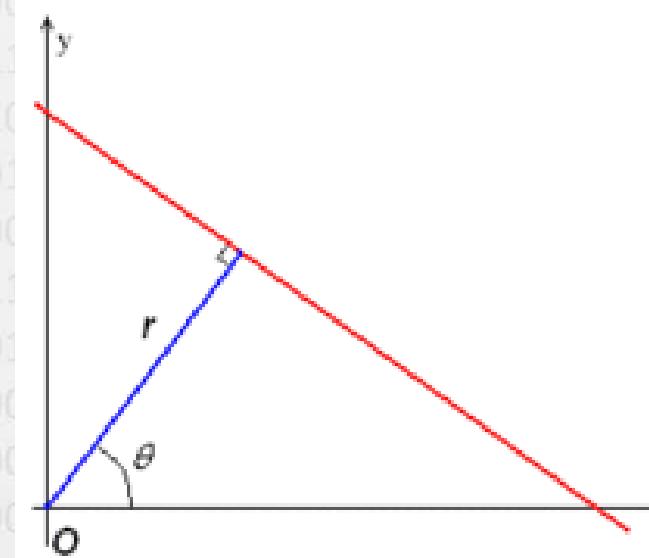
Hough-transzformáció

- **Polárkoordinátás reprezentáció**

- Egy egyenes minden pontjára θ és r állandó.
- Bármely irányban numerikusan stabil leírás.

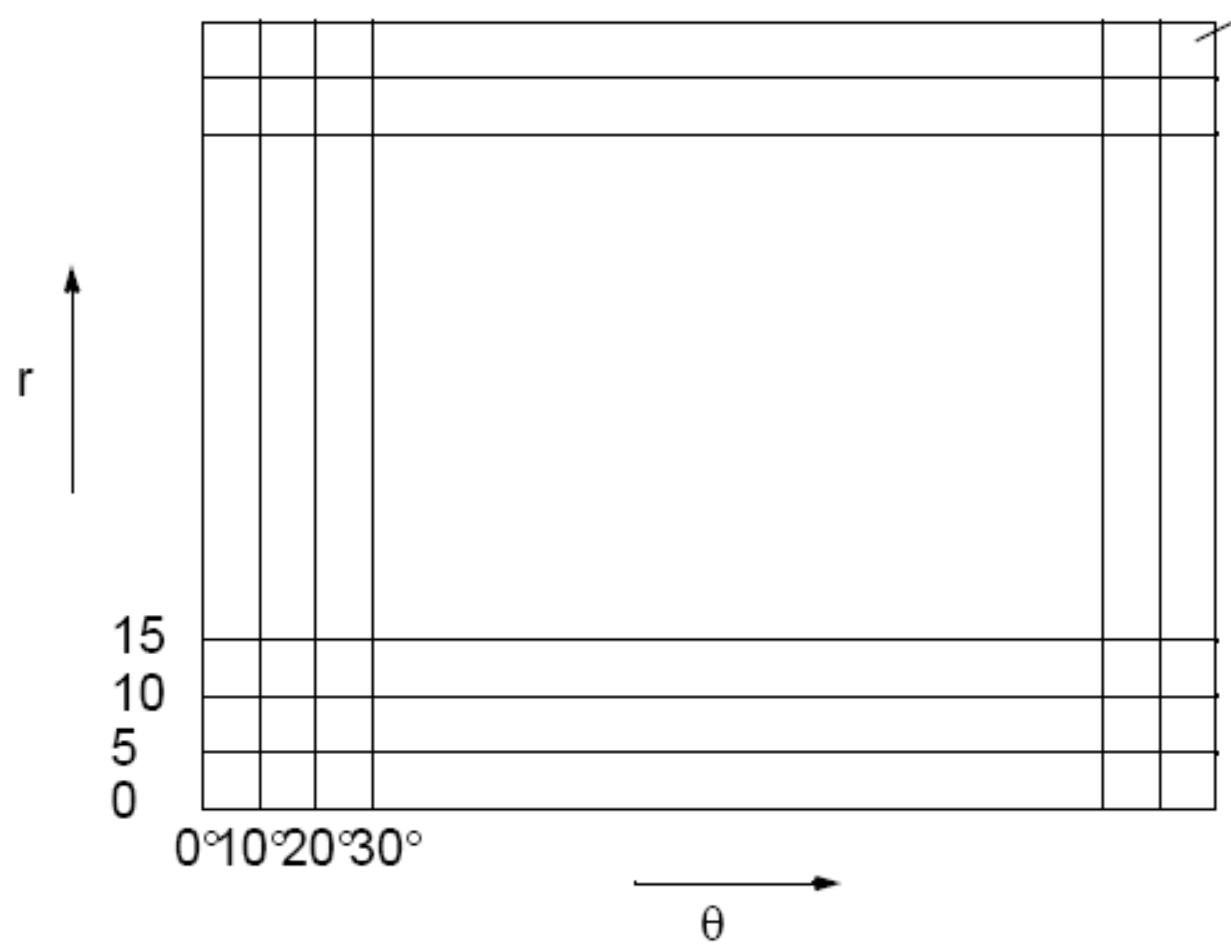
- **Különböző θ konstans értékekre ρ fix értékeinél különböző vonalakat szolgáltatnak.**

$$x \cos \theta + y \sin \theta = r$$



Akkumulátor tömb

Gyűjtő (Akkumulátor)

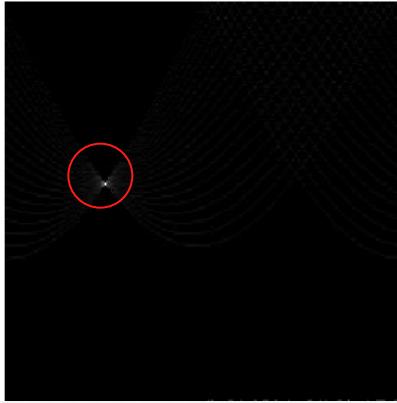
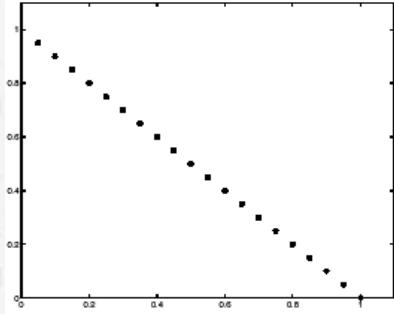


Algoritmus

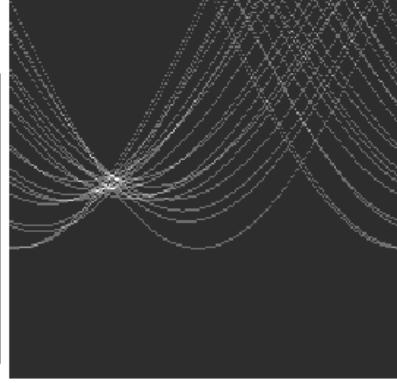
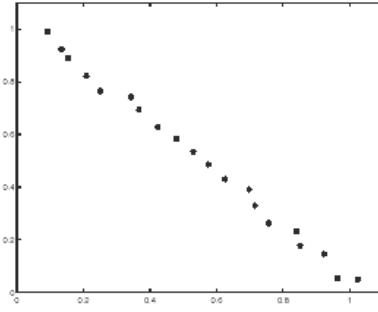
- Készítsünk egy $2D(\theta, r)$ számláló tömböt, a szög 0 és 180 fok között változik, a távolság maximum a kép átlója!
 - Nullázzuk ki az akkumulátort!
- A θ szög lehetséges értékeit vegyük fel!
 - Például 10° -os növekmények.
- minden (x, y) élpontra és minden lehetséges szögre
 - számoljuk ki r értékét! $x \cos \theta + y \sin \theta = r$
 - minden kiszámolt (θ, r) párra növeljük meg a számlálótömb értékét!
- Keressük meg a lokális maximumokat!

Vonaldetektálás - példa

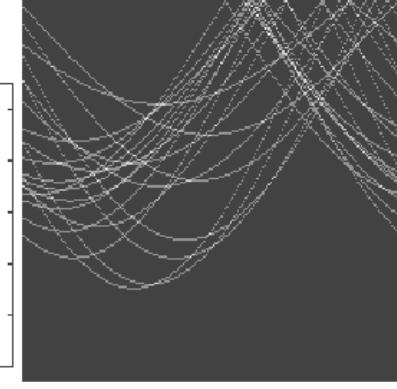
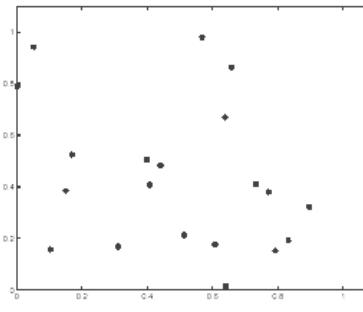
ideális



zajos



nagyon zajos



Nehézségek

- **Hogyan osszuk fel a paraméterteret (θ, r)?**
 - Nagy? Nem tudunk különbséget tenni vonalak között.
 - Kicsi? A zaj hibákat eredményez.
- **Hány vonalat találunk?**
- **Melyik él pont melyik vonalhoz tartozik?**
- **A zaj miatt nehéz kielégítő megoldást találni.**

Soros kód

```
1 for each y fentről lefele
2   for each x balról jobbra
3     sobel ([x], [y], dx, dy)           // éldetektálás
4     grad_nagyság := grad_mag (dx, dy)  // gradiens nagyság
5     if (grad_nagyság > threshold) {
6       theta := grad_dir(dx, dy)
7       theta := theta_qunzite(theta)
8       r := x * cos(theta) + y* sin(theta)
9       r := r_quantize(r)
10      acc[r, theta] ++
11      pixel[x+1][y+1] := pixel[x+1][y+1] + 1/16 * hiba
12    }
```

Párhuzamosítás

- **Mivel az akkumulátor tömb számítása független más összegzések től, ezért párhuzamosítható:**
 - az egész képre olvasási jog szükséges.

Felhasznált és javasolt irodalom

- [1] A. Grama, A. Gupta, G. Karypis, V. Kumar

Introduction to Parallel Computing

Addison-Wesley, ISBN 0-201-64865-2, 2003, 2nd ed., angol, 636 o.

- [2] B. Wilkinson, M. Allen

Parallel Programming

Prentice Hall, ISBN 978-0131405639, 2004, 2nd ed., angol, 496 o.

- [3] S. Akhter, J. Roberts

Multi-Core Programming (Increasing Performance through Software Multi-threading)

Intel Press, ISBN 0-9764832-4-6, 2006, angol, 340 o.

- [4] Iványi A.

Párhuzamos algoritmusok

ELTE Eötvös Kiadó, ISBN 963-463-590-3, Budapest, 2003, magyar, 334 o.

- [5] R. Gonzales, R. Woods

Digital Image Processing

Addison-Wesley, ISBN-13: 978-0201508031, 2nd ed. 2005, angol 730 o.

Párhuzamos programozás .NET környezetben

A párhuzamos programozás alapjai

A végrehajtás szétválasztása: elszigetelés és párhuzamosítás
Párhuzamosság és ütemezés az operációs rendszer szintjén

Folyamatok fogalma és kezelése

Új folyamat indítása, létező folyamatok leállítása és bevárása
Folyamatok tulajdonságainak és eseményeinek kezelése

Szálak fogalma és kezelése

Új szálak indítása, szálak felfüggesztése, állapotvezérlése és leállítása, szálak prioritási szintjei
Előtér- és háttérszálak, ThreadPool szálak, aszinkron metódushívás

Szinkronizáció

A szinkronizáció alapfogalmai, versenyhelyzet és holtpont fogalma
Kölcsönös kizárási biztosítása, a „lock” utasítás, szálak bevárása és randevúja

A párhuzamos végrehajtás alapjai

- **A Neumann-architektúrára épülő számítógépek a programokat sorosan hajtják végre**

A gépi kódú programok tényleges futtatása utasításról utasításra történik.

Ahhoz, hogy egy számítógépen egyszerre több program is futatható legyen, olyan megoldásra van szükség, amely biztosítja

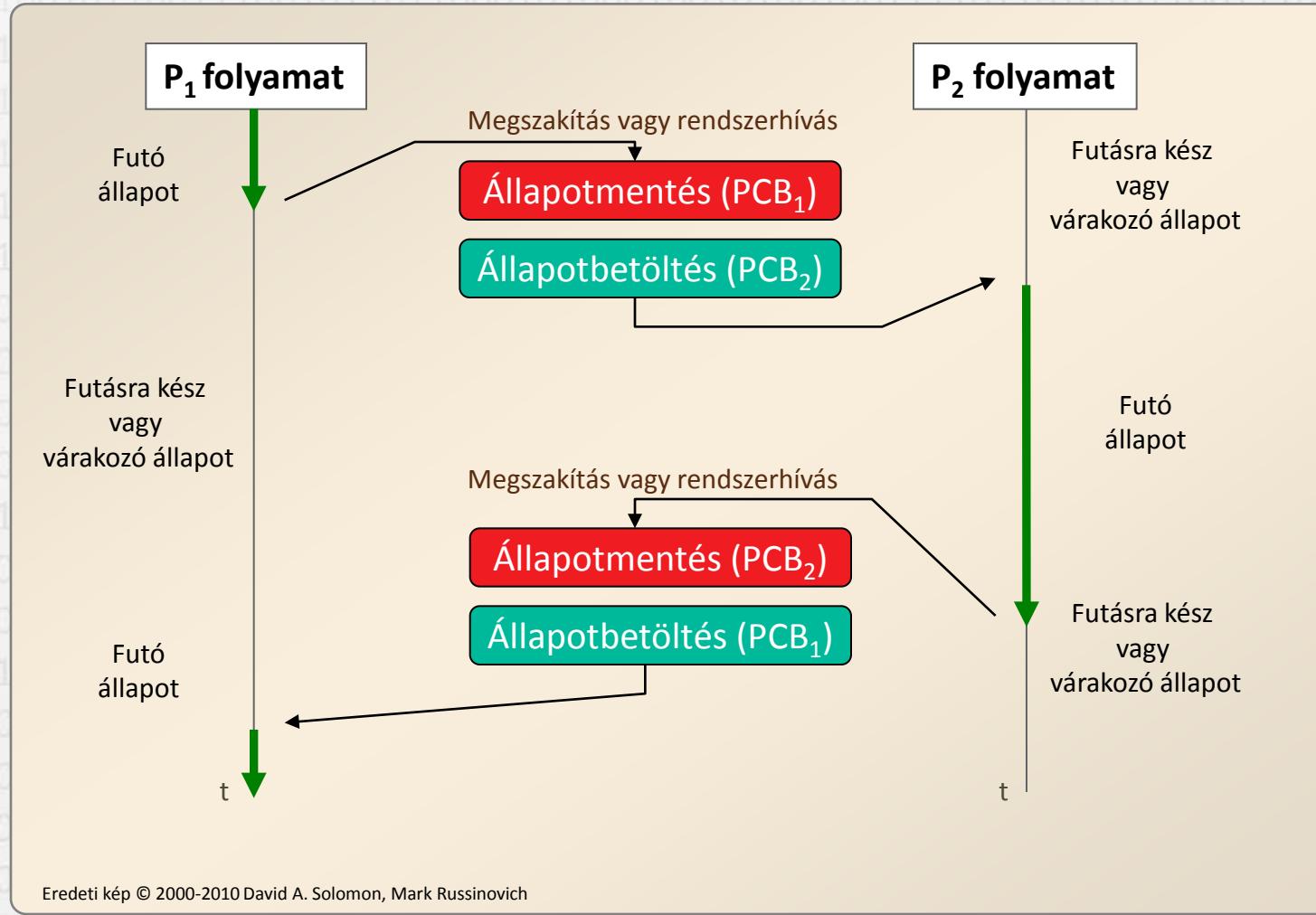
- a végrehajtás alatt álló programok egymástól való *elszigetelését*, valamint
- a végrehajtás alatt álló programok (látszólag) *egyidejű futását*.

- **A programok elszigetelése és párhuzamosítása a folyamatok koncepciójának segítségével megoldható**

Az elszigetelés érdekében minden folyamat saját memóriaterülettel rendelkezik, amelyet más folyamatok nem érhetnek el, így hiba esetén csak a hibázó folyamat sérül, a rendszer többi eleme működőképes marad (viszont a folyamatok közötti közvetlen kommunikációra sincs egyszerű lehetőség).

A párhuzamosítás tipikus megoldása az *időosztás*, amikor minden folyamat kap egy-egy ún. időszeletet, melynek leteltét követően egy másik folyamat kapja meg a vezérlést. Ez a megoldás gyakorilag függetleníti egymástól a processzorok és a rajtuk egyidőben futtatható programok számát.

Illusztráció: időosztás és kontextusváltások



Rövidtávú ütemezés operációs rendszerekben

- A rövidtávú ütemezés fogalma a végrehajtás alatt álló programok közötti rendszeres váltás módját, időzítését és szabályait takarja

A Microsoft Windows preemptív, prioritásos, körbenforgó, kétszintű* rövidtávú ütemezési politikát alkalmaz.

- Az ütemezés *preemptív*: az operációs rendszer kívülről bármikor képes megszakítani a programok futását
- Az ütemezés *prioritásos*: minden programnak van egy fontossági szintje (prioritása), amely meghatározza, hogy egy-egy időszelet lejártakor melyik program következhet sorra
- Az ütemezés *körbenforgó*: az egyforma prioritású programok között a rendszer egyenlően osztja el a rendelkezésre álló időt, és a programok sorban egymás után kapnak egy-egy időszeletet (az utolsó program időszelete után ismét az első következik)
- Az ütemezés *kétszintű**: az elszigetelt folyamatok mellett léteznek ún. szálak is, amelyek a futtatni kívánt kód egy folyamaton belüli további bontását teszik lehetővé

A Unix/Linux rendszerek a legutóbbi évekig félig preemptív, prioritásos, körbenforgó, egyszintű ütemezési politikát alkalmaztak.

- Az ütemezés *fél preemtív* volt: az operációs rendszer a programok futását kívülről bármikor képes volt megszakítani, saját belső elemeinek futását azonban nem
- Az ütemezés *egyszintű* volt: a rendszerben csak elszigetelt folyamatok léteztek, ezek képezték a párhuzamosítás alapegységét is (a Linux ma már szálkezelést is támogat)

* A modell ténylegesen négyszintű, de a legfelső („job”) és a legalsó („fiber”) szemcsézettségi szint csak speciális módon használható.

Folyamatok

- **A .NET keretrendszerben a folyamatok megfelelnek az operációs rendszer folyamatainak**

A folyamatok kezelését a *System.Diagnostics.Process* és a *System.Diagnostics.ProcessStartInfo* osztályok biztosítják. A Process osztály segítségével új folyamatok hozhatók létre, létező folyamatok szüntethetők meg és a folyamatokról részletes adatok érhetők el. A ProcessStartInfo osztály segítségével számos paraméter és beállítás adható meg a folyamatként elindítani kívánt programokhoz.

- **A .NET a folyamatokon belül egy további szintet, az ún. alkalmazástartományt („application domain”) is meghatároz**

A felügyelt kódú programokat a keretrendszer futtatás közben is ellenőrzi, ezért ezek nem képesek egymást negatívan befolyásolni. Így viszont nem feltétlenül szükséges külön folyamatként futtatni őket, ami sebesség és memóriaigény szempontjából nagy előny, mivel a folyamatok létrehozása, nyilvántartása és a közöttük történő váltás sok időt és memóriát igényel.

Az alkalmazástartományokkal és programozásukkal a jelen tárgy keretében nem foglalkozunk részletesen.

Példa új folyamat indítására

```
1  using System;
2  using System.Diagnostics;
3
4  class Program
5  {
6      static void Main()
7      {
8          Process newProcess = new Process();
9          newProcess.StartInfo = new ProcessStartInfo("hello.exe", "Pistike");
10         newProcess.StartInfo.ErrorDialog = true;
11         newProcess.StartInfo.UseShellExecute = false;
12         newProcess.StartInfo.RedirectStandardOutput = true;
13
14         newProcess.Start();
15
16         Console.WriteLine("Az elindított folyamat üzenetei:");
17         Console.Write(newProcess.StandardOutput.ReadToEnd());
18         newProcess.WaitForExit();
19
20         Console.ReadLine();
21     }
22 }
```

ProcessExamples\RunHello.cs

Feladat (1)

Készítsünk egyszerű konzolos alkalmazást, amely folyamatosan (végtelen ciklusban) számlál 0-tól felfelé, és minden lépés eredményét kiírja a képernyőre! Az operációs rendszer segédeszközeivel vizsgáljuk meg, hogyan reagál a rendszer és a futó program (folyamat) egy, illetve több példánya a prioritási osztály és a processzoraffinitás módosítására!

Ötletek:

- A kiírás gyakoriságát a ciklus belsejében elhelyezett belső modulo számlálóval igény szerint csökkenthetjük
- A megoldáshoz elegendő a Feladatkezelő (Task Manager) használata

Megoldás (1)

```
1 using System;  
2  
3 class Program  
4 {  
5     static void Main()  
6     {  
7         int counter = 0;  
8         while (true)  
9             Console.WriteLine(counter++ % int.MaxValue);  
10    }  
11 }
```

ProcessExamples\Counter.cs

Feladat (2)

Készítsünk konzolos alkalmazást, amely (a számítógépre telepített .NET keretrendszer segítségével) képes a parancssorban megadott C# nyelvű forrásfájl lefordítására és az esetleges hibák megjelenítésére! Amennyiben a forráskód hibátlan volt, ezt a program külön üzenetben jelezze!

Ötletek:

- A .NET keretrendszer része a parancssoros C# fordító (csc.exe)
- A program számára parancssorban megadott adatok kezeléséhez a Main() metódus „args” paraméterét használhatjuk fel, amely karaktersorozatok tömbjeként tartalmazza az átadott adatokat
- Ha a forráskód fordítása sikeres, maga a C# fordító egy néhány soros fejlécen kívül semmilyen üzenetet nem ír ki. A fejléc megjelenítése a „/nologo” parancssori paraméterrel kapcsolható ki.

Megoldás (2)

```
1  using System;
2  using System.Diagnostics;
3
4  class Program
5  {
6      static void Main(string[] args)
7      {
8          if (args.Length > 0)
9          {
10              ProcessStartInfo startinfo = new ProcessStartInfo();
11              startinfo.FileName = String.Format(@ "{0}..\\"Microsoft.NET\Framework\v{1}\csc.exe",
12                  Environment.GetFolderPath(Environment.SpecialFolder.System), Environment.Version.ToString(3));
13              startinfo.Arguments = String.Format(@ "/nologo /t:exe {0}", args[0]);
14              startinfo.RedirectStandardOutput = true;
15              startinfo.UseShellExecute = false;
16
17              Process compilerProcess = Process.Start(startinfo);
18
19              string output = compilerProcess.StandardOutput.ReadToEnd();
20              compilerProcess.WaitForExit();
21
22              if (output == String.Empty)
23                  Console.WriteLine("A forráskód hibátlan, a fordítás sikerült.");
24              else
25                  Console.WriteLine("Hibaüzenetek:" + Environment.NewLine + output);
26      }
27  }
```

Compiler\Program.cs

Feladat (3)

Készítsük el az előző feladat bővített megfelelőjét grafikus Windows alkalmazásként is (az elkészült program adjon lehetőséget a forráskód szerkesztésére, betöltésére, mentésére, fordítására és futtatására)!

Ötletek:

- A program a forráskódot fordítás előtt mentse el egy ideiglenes fájlba
- A futtatáshoz használjuk a Process osztály statikus Start() metódusát

Megoldás (3)

The screenshot shows a Windows application window titled "Vizuális C# fordítóprogram". The main area displays the following C# code:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
            Console.WriteLine(String.Format("Szervusz, {0}!", args[0]));
        else
            Console.WriteLine("Szervusz, világ!");
    }
}
```

Below the code, the "C# fordító üzenetei" section shows the message: "Nincs hiba" (No error).

A secondary window titled "D:\Temp\ProcessesAndThreads\VisualCompiler\..." is displayed, showing the output of the program: "Szervusz, világ!".

VisualCompiler

Folyamatok kezelése (kivonatos referencia)

- **System.Diagnostics.Process** osztály

Metódusok	
Start()	Folyamat indítása
CloseMainWindow()	Folyamat főablakának bezárása*
Kill()	Folyamat leállítása
GetCurrentProcess()	Aktuális folyamatot reprezentáló objektum lekérése
GetProcessById()	Adott azonosítójú folyamatot reprezentáló objektum lekérése
GetProcesses()	Összes folyamat adatainak lekérése a helyi számítógépről
WaitForExit()	Várakozás a folyamat befejeződésére
WaitForInputIdle()	Várakozás a folyamat inaktív állapotának elérésére*

* Ezek a metódusok csak grafikus felhasználói felülettel (és így üzenetkezelő ciklussal) rendelkező alkalmazásoknál használhatók.

Folyamatok kezelése (kivonatos referencia)

• System.Diagnostics.Process osztály

Tulajdonságok	
StartInfo	A folyamathoz tartozó ProcessStartInfo példány
PriorityClass	A folyamat prioritási osztálya
BasePriority	A folyamat alapprioritásának lekérése numerikus formában
EnableRaisingEvents	A folyamat kiválthat-e eseményeket
HasExited	A folyamat kilépett-e
ExitCode, ExitTime	Kilépési kód, illetve a kilépés (vagy leállítás) időpontja
StandardError, StandardInput, StandardOutput	Alapértelmezett hibacsatorna, bemeneti csatorna és kimeneti csatorna (adatfolyam)
ProcessorAffinity	Megadja, hogy a folyamat mely processzorokon futhat
UserProcessorTime	A folyamat felhasználói módban töltött processzorideje
PrivilegedProcessorTime	A folyamat kernel módban töltött processzorideje
Threads	A folyamathoz tartozó Windows szálak gyűjteménye
Események	
ErrorDataReceived	A folyamat adatot írt az alapértelmezett hibacsatornára
Exited	A folyamat kilépett (vagy leállították)
OutputDataReceived	A folyamat adatot írt az alapértelmezett kimeneti csatornára

Folyamatok kezelése (kivonatos referencia)

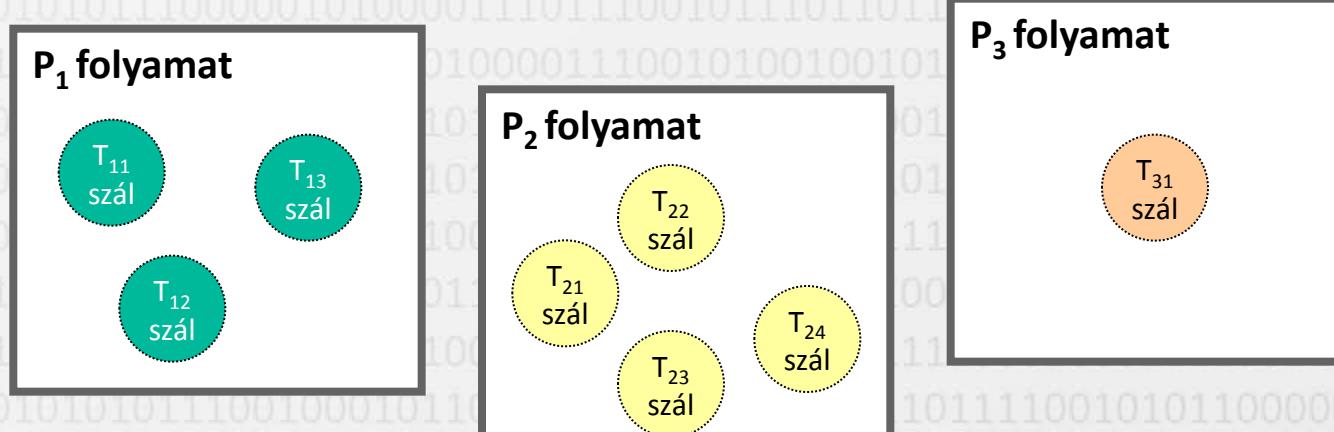
- **System.Diagnostics.ProcessStartInfo osztály**

Tulajdonságok	
FileName	Fájlnév megadása az indítandó folyamathoz (program vagy programmal társított fájltípusba tartozó fájl neve)
Arguments, WorkingDirectory	Parancssori paraméterek és munkakönyvtár megadása az indítandó folyamathoz
Domain, UserName, Password	Folyamat indítása adott felhasználó névében
RedirectStandardError, RedirectStandardInput, RedirectStandardOutput	Alapértelmezett hibacsatorna, bemeneti csatorna és kimeneti csatorna (adatfolyam) átirányítása
ErrorDialog	Hibaüzenet jelenjen-e meg, ha a folyamat indítása sikertelen
UseShellExecute	Operációs rendszerhéj programindító funkciójának használata folyamat indításához
Verb	A társított fájl megnyitásakor végrehajtandó művelet
CreateNoWindow	A folyamathoz automatikusan létrejöjjön-e konzolablak
WindowSize	Kezdeti ablakméret megadása (normál, minimalizált vagy maximalizált méret)

Szálak

- A szálak elsődleges célja a folyamatokon belüli párhuzamosítás

A folyamatok adminisztrációja és váltása igen erőforrásigényes művelet, viszont az általuk nyújtott elszigetelés szintje egy programon belül csaknem mindenkorának nyújtja. Ezt az ellentmondást oldják fel a szálak, amelyek elszigetelést nem nyújtanak, gyors párhuzamos végrehajtást azonban igen.



- A .NET keretrendszer támogatja a szálak kezelését is

A keretrendszer kihassználja az operációs rendszer száltámogatását, de emellett jelentősen (és több absztrakciós szinten) bővíti a felkínált általános szolgáltatásokat.

A többszálúság megvalósítási lehetőségei

- **Közvetlen szálkezelés (System.Threading.Thread osztály)**

Lehetővé teszi szálak egyenkénti létrehozását, azonosítását, állapotvezérlését és megszüntetését. Kezelése egyszerű, viszont sok programozói munkát és pontosságot igényel.

- **Absztrakt szálkezelés (System.Threading.ThreadPool osztály)**

Gyakran ismétlődő, rövid ideig tartó, erősen párhuzamos műveletekhez rendelkezésre álló „szálkészlet”, melynek használatával megtakarítható a szálak egyenkénti létrehozásának és megszüntetésének időigényes munkája. Kezelése egyszerű és hatékony, de a szálak egyéni identitását nem biztosítja.

- **Háttérszál (System.ComponentModel.BackgroundWorker osztály)**

A felhasználói felület kezelésének és a háttérben elvégzendő, esetenként igen sokáig tartó műveletek végrehajtásának szétválasztására szolgál.

Kezelése kényelmes (az állapotváltozásokról események útján értesíti a felhasználóját), ám korlátozott funkcionálitása miatt kevés célra alkalmas.

- **Aszinkron metódushívás**

Képviselők és az aszinkron mintát támogató metódusok aszinkron hívása

A System.Threading névtér

- **Ebben a névtérben találhatók azok a típusok, melyek**
 - megvalósítják a szálak explicit futási idejű vezérlését,
 - szinkronizációs mechanizmusokat szolgáltatnak, illetve
 - bizonyos fokig automatizált absztrakt háttérszálkezelést tesznek lehetővé.

- **A névtér fontosabb elemei:**

Alaposztályok: Thread, ThreadPool, Monitor

Felsorolások: ThreadState, ThreadPriority

Kivételek: ThreadAbortException, ThreadInterruptedException

Képviselők (delegáltak): ThreadStart, WaitCallback, TimerCallback,
IOCompletionCallback...

A Thread osztály

- A .NET-re épülő párhuzamos programozás alaposztálya, amely lehetővé teszi szálak egyenkénti létrehozását, azonosítását, állapotvezérlését és megszüntetését.

Kezelése egyszerű, viszont sok programozói munkát és pontosságot igényel, mivel ez a párhuzamos programozás .NET-ben elérhető legalacsonyabb (és egyben legnagyobb programozói befolyást biztosító) absztrakciós szintje.

A .NET keretrendszer szálai (a Thread osztály példányai) és a Windows szálai (ProcessThread osztály példányai) között a keretrendszer nem garantál egyértelmű (1:1 típusú) megfeleltetést.

A keretrendszer dönthet úgy, hogy egy Windows szálon több .NET szál kódját futtatja, emiatt az általunk használt szálak nem mindenkor azonosíthatók egyértelműen a Windows segédesszközeivel (pl. Feladatkezelő).

ThreadStart, ThreadPriority és ThreadState

```
public sealed class Thread
{
    public Thread(ThreadStart start);
    public ThreadPriority Priority { get; set; }
    public ThreadState State { get; }
}
```

```
public enum ThreadPriority
{
    Lowest = 0,
    BelowNormal = 1,
    Normal = 2,
    AboveNormal = 3,
    Highest = 4,
}
```

```
public enum ThreadState
{
    Running = 0,
    StopRequested = 1,
    SuspendRequested = 2,
    Background = 4,
    Unstarted = 8,
    Stopped = 16,
    WaitSleepJoin = 32,
    Suspended = 64,
    AbortRequested = 128,
    Aborted = 256,
}
```

Példa: szál indítása statikus metódussal

```
1  using System;
2  using System.Threading;
3
4  class Program
5  {
6      static void Main()
7      {
8          Console.WriteLine("Szál közvetlen létrehozása");
9          Console.WriteLine("Főszál (sorszáma: {0})", Thread.CurrentThread.GetHashCode());
10         Thread aThread = new Thread(ThreadMethod); // new Thread(new ThreadStart(ThreadMethod));
11         aThread.Name = "Új szál";
12         aThread.Start();
13     }
14
15     static void ThreadMethod()
16     {
17         Console.WriteLine("{0} (sorszáma: {1})",
18             Thread.CurrentThread.Name,
19             Thread.CurrentThread.GetHashCode());
20     }
21 }
```

ThreadExamples\Static.cs

Példa: két szál indítása példánymetódussal

```
1  using System;
2  using System.Threading;
3
4  class Printer
5  {
6      char ch;
7      int sleepTime;
8
9      public Printer(char ch, int sleepTime)
10     { this.ch = ch; this.sleepTime = sleepTime; }
11
12     public void Print()
13     { for (int i = 0; i < 100; i++) { Console.Write(ch); Thread.Sleep(sleepTime); } }
14 }
15
16 class Test
17 {
18     static void Main()
19     {
20         Printer a = new Printer('.', 10); Printer b = new Printer('*', 100);
21         new Thread(a.Print).Start(); new Thread(b.Print).Start();
22     }
23 }
```

ThreadExamples\Instance.cs

Előtér- és háttérszálak

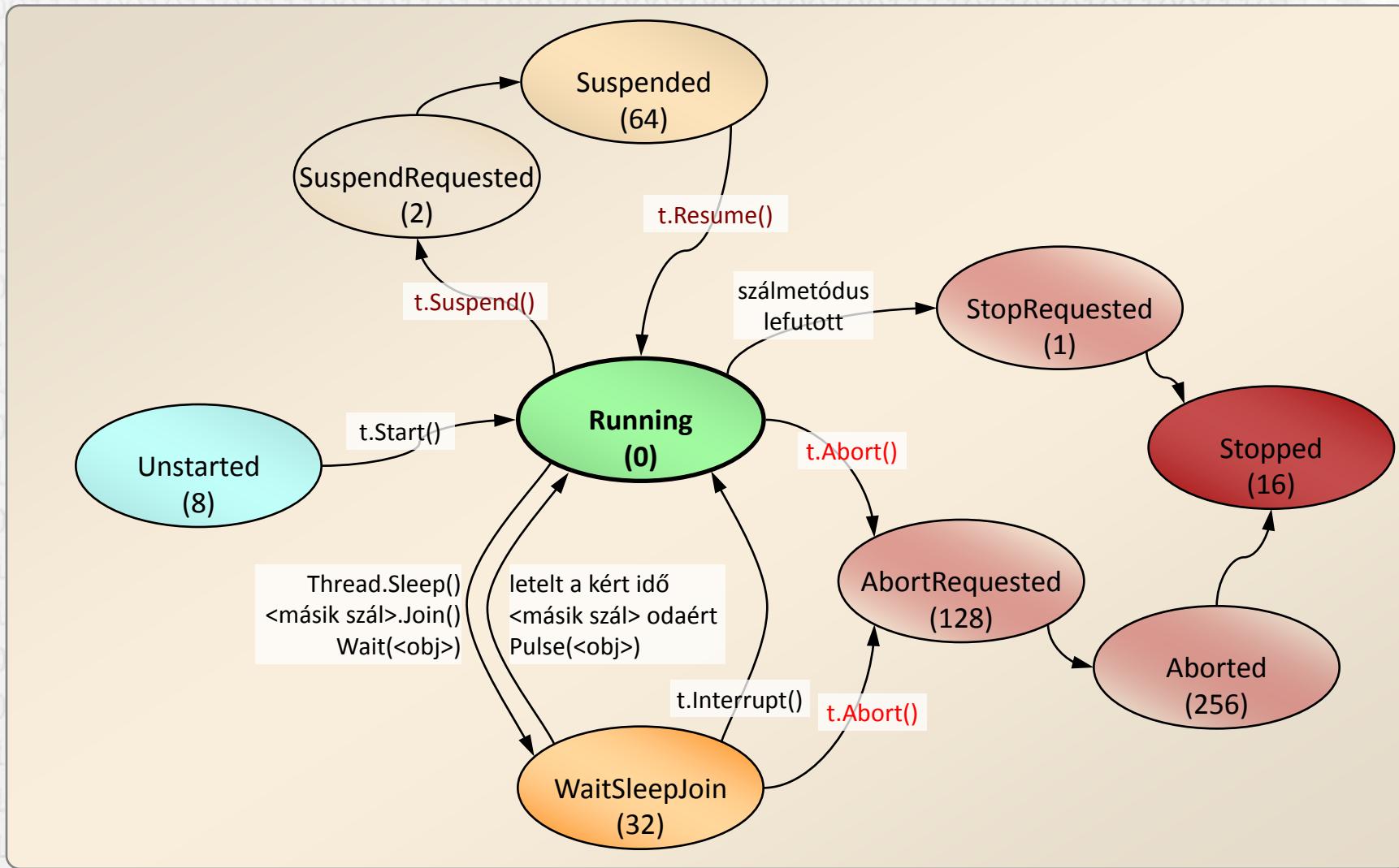
- A .NET keretrendszer szálai előtér- vagy háttérszálak lehetnek
- A két típus közötti különbségek:
 - A program pontosan addig fut, míg legalább egy előtérszála aktív
 - A háttérszálak futása az utolsó előtérszál befejeződésekor automatikusan megáll, maguk a háttérszálak pedig automatikusan megszűnnek
- Az egyes szálak állapota ebből a szempontból az **IsBackground** tulajdonsággal módosítható

Alapértelmezésben minden szál előtérszálként jön létre, majd létrehozás után háttérszállá lehet alakítani.

```
Thread t = new Thread(<metódus>);  
t.IsBackground = true;
```

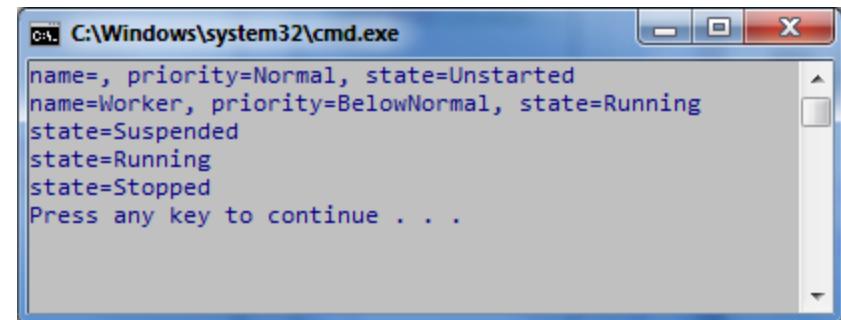
Szálak állapotváltozásai

A .NET szálak állapotátmenet-diagramja



Példa: szál állapotváltozásai

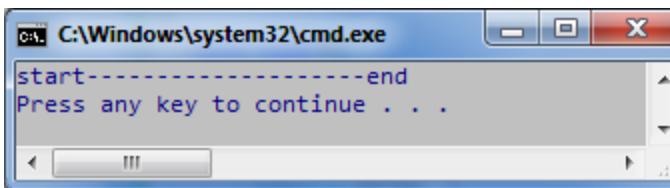
```
1  using System;
2  using System.Threading;
3
4  class Program
5  {
6      static void Main()
7      {
8          Thread t = new Thread(delegate() { while (true); });
9          Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority, t.ThreadState);
10         t.Name = "Worker"; t.Priority = ThreadPriority.BelowNormal;
11         t.Start();
12         Thread.Sleep(1);
13         Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority, t.ThreadState);
14         t.Suspend();
15         Thread.Sleep(1);
16         Console.WriteLine("state={0}", t.ThreadState);
17         t.Resume();
18         Console.WriteLine("state={0}", t.ThreadState);
19         t.Abort();
20         Thread.Sleep(1);
21         Console.WriteLine("state={0}", t.ThreadState);
22     }
23 }
```



ThreadExamples\States.cs

Példa: szál bevárása

```
1 using System;
2 using System.Threading;
3
4 class Program
{
5
6     static void Main()
7     {
8         Thread t = new Thread(P);
9         Console.WriteLine("start");
10        t.Start();
11        t.Join();
12        Console.WriteLine("end");
13    }
14
15    static void P()
16    {
17        for (int i = 0; i < 20; i++)
18        {
19            Console.Write("-");
20            Thread.Sleep(100);
21        }
22    }
23}
```



ThreadExamples\Join.cs

Szálak kezelése (kivonatos referencia)

- **System.Threading.Thread osztály**

Metódusok	
Start()	Szál indítása
Suspend()	Szál futásának felfüggesztése*
Resume()	Szál futásának folytatása*
Abort()	Szál leállítása
GetHashCode()	Szál azonosítójának lekérése
Sleep()	Várakozás a megadott időintervallum elteltéig
Join()	Várakozás az adott szál befejeződésére
Interrupt()	Várakozó állapotban lévő szál reaktiválása
VolatileRead()	Adott mező tartalmából az abszolút legfrissebb érték lekérése (a processzorok számától és azok gyorsítótárainak állapotától függetlenül)
VolatileWrite()	Adott mező tartalmának azonnali kiírása a főmemóriába (az adat a többi processzor számára azonnal láthatóvá válik)

* Elavult metódus, használata a .NET frissebb változatai mellett már nem ajánlott.

Szálak kezelése (kivonatos referencia)

- **System.Threading.Thread osztály**

Tulajdonságok	
CurrentCulture , CurrentUICulture	A szálhoz tartozó aktuális kultúra, illetve a szálhoz tartozó felhasználói felület kiválasztott nyelve
IsBackground	Az adott szál háttérszál vagy előtérszál*
IsThreadPoolThread	Az adott szál a ThreadPool egyik szála-e
IsAlive	Az adott szál él (azaz nem fejeződött be váratlanul)
ManagedThreadId	A szál egyedi azonosítója
Name	A szál megnevezése
Priority	A szál prioritása (fontossági szintje)
ThreadState	A szál aktuális állapota(i)

* A programok futása akkor ér véget, ha az utolsó előtérszáluk is lefutott (a még futó háttérszálak ekkor automatikusan megszűnnek).

A ThreadPool osztály

- **Gyakran ismétlődő, rövid ideig tartó, erősen párhuzamos műveletekhez rendelkezésre álló „szálkészlet”**

Használatával megtakarítható a szálak egyenkénti létrehozásának és megszüntetésének időigényes munkája. Kezelése egyszerű és hatékony, ugyanakkor azonban nem teszi lehetővé a szálak egyenkénti azonosítását és kezelését (pl. prioritás beállítása).

Akkor működik a leghatékonyabban, ha nagy mennyiségű, de egyenként igen rövid ideig tartó párhuzamos művelethez vesszük igénybe.

A ThreadPool osztály által kezelt szálkészlet bizonyos határok között automatikusan igazodik az aktuális terheléshez (nagy terhelés esetén új szálak jönnek létre, kis terhelés esetén pedig a szükségtelen szálak egy idő után megszűnnek). Ezt az adaptív viselkedést az osztály metódusaival befolyásolhatjuk.

Feladat (4)

Készítsünk konzolos alkalmazást, amely a ThreadPool osztály segítségével 4 külön szálban jeleníti meg az egyes szálak által folyamatosan növelt saját belső számláló értékét! A program valamilyen megoldással biztosítsa a szálak által kiírt adatok vizuális elkülönítését!

Ötletek:

- A QueueUserWorkItem() metódus paramétere egy WaitCallback típusú képviselő, amely visszatérési érték nélküli, egyetlen („object” típusú) paraméterrel rendelkező metódusokat képes tárolni
- A ThreadPool szálai is azonosíthatók a ManagedThreadId tulajdonsággal (azonban előfordulhat például, hogy egy szál által megkezdett feladatot egy másik szál folytat és egy harmadik szál fejez be)
- A szálak közötti váltások megfigyeléséhez érdemes sok munkát adni az egyes szálaknak és néha várakoztatni őket (ennek legegyszerűbb módja a Thread osztály statikus Sleep() metódusa)

Megoldás (4)

ProcessesAndThreads - Microsoft Visual Studio (Administrator)

File Edit View Refactor Project Build Debug Data Tools Test Analyze Window Help

Debug Any CPU

Program.cs

```
23 Console.WriteLine("Szálak létrehozása a ThreadPool osztály segítségével");
24 Console.WriteLine("Főszál (sorszáma: {0})", Thread.CurrentThread.GetHashCode());
25 ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadPoolMethod), ConsoleColor.Blue);
26 Thread.Sleep(150);
27 ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadPoolMethod), ConsoleColor.Red);
28 Thread.Sleep(150);
29 ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadPoolMethod), ConsoleColor.DarkGreen);
30 Thread.Sleep(150);
31 ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadPoolMethod), ConsoleColor.White);
32 Console.ReadLine();
33 }
34
35 static void ThreadMethod()
36 {
37     //Thread.Sleep(1000);
38     Console.WriteLine("{0} (sorszáma {1})", Thread.CurrentThread.Name, Thread.CurrentThread.GetHashCode());
39 }
40
41 static void ThreadPoolMethod(object state)
42 {
43     ConsoleColor textcolor = (ConsoleColor) state;
44
45     Console.ForegroundColor = textcolor;
46     Console.WriteLine("Szál sorszáma: " + Thread.CurrentThread.ManagedThreadId);
47
48     DisplayThreadData();
49     DisplayNumbers();
50
51     Console.ForegroundColor = textcolor;
52     Console.WriteLine("Vége");
53 }
```

C:\Windows\system32\cmd.exe

Szálak létrehozása a ThreadPool osztály segítségével

Főszál (sorszáma: 1)

Szál sorszáma: 4

Szál adatai

Prioritás:	Normal
Kultúra:	hu-HU
ThreadPool szál?	True
Állapot:	Background

A számláló értéke 1

Szál sorszáma: 5

Szál adatai

Prioritás:	Normal
Kultúra:	hu-HU
ThreadPool szál?	True
Állapot:	Background

A számláló értéke 1

A számláló értéke 2

Szál sorszáma: 6

Szál adatai

Prioritás:	Normal
Kultúra:	hu-HU
ThreadPool szál?	True
Állapot:	Background

A számláló értéke 1

A számláló értéke 2

Szál sorszáma: 7

Szál adatai

Prioritás:	Normal
Kultúra:	hu-HU
ThreadPool szál?	True
Állapot:	Background

A számláló értéke 1

A számláló értéke 2

A számláló értéke 3

A számláló értéke 4

A számláló értéke 5

A számláló értéke 6

A számláló értéke 7

A számláló értéke 8

A számláló értéke 9

A számláló értéke 10

Vége

A számláló értéke 11

Vége

A számláló értéke 12

Vége

ThreadExamples\ThreadPool.cs

Szálak kezelése (kivonatos referencia)

- **System.Threading.ThreadPool osztály**

Metódusok	
QueueUserWorkItem()	Metódus végrehajtása egy ThreadPool szálón
GetAvailableThreads()	A rendelkezésre álló ThreadPool szálak számának lekérdezése
GetMaxThreads(), GetMinThreads()	Maximálisan rendelkezésre álló, illetve minimálisan életben tartott ThreadPool szálak számának lekérdezése
SetMaxThreads(), SetMinThreads()	Maximálisan rendelkezésre álló, illetve minimálisan életben tartott ThreadPool szálak számának beállítása
RegisterWaitForSingleObject()	Várakozás erőforrásra vagy időzítőre

A BackgroundWorker osztály

- A felhasználói felület kezelésének és a háttérben elvégzendő, esetenként igen sokáig tartó műveletek végrehajtásának szétválasztására szolgál.

Kezelése igen kényelmes (az állapotváltozásokról események útján értesíti a felhasználó osztályt), ám korlátos funkcionálitása miatt kevés célra alkalmas.

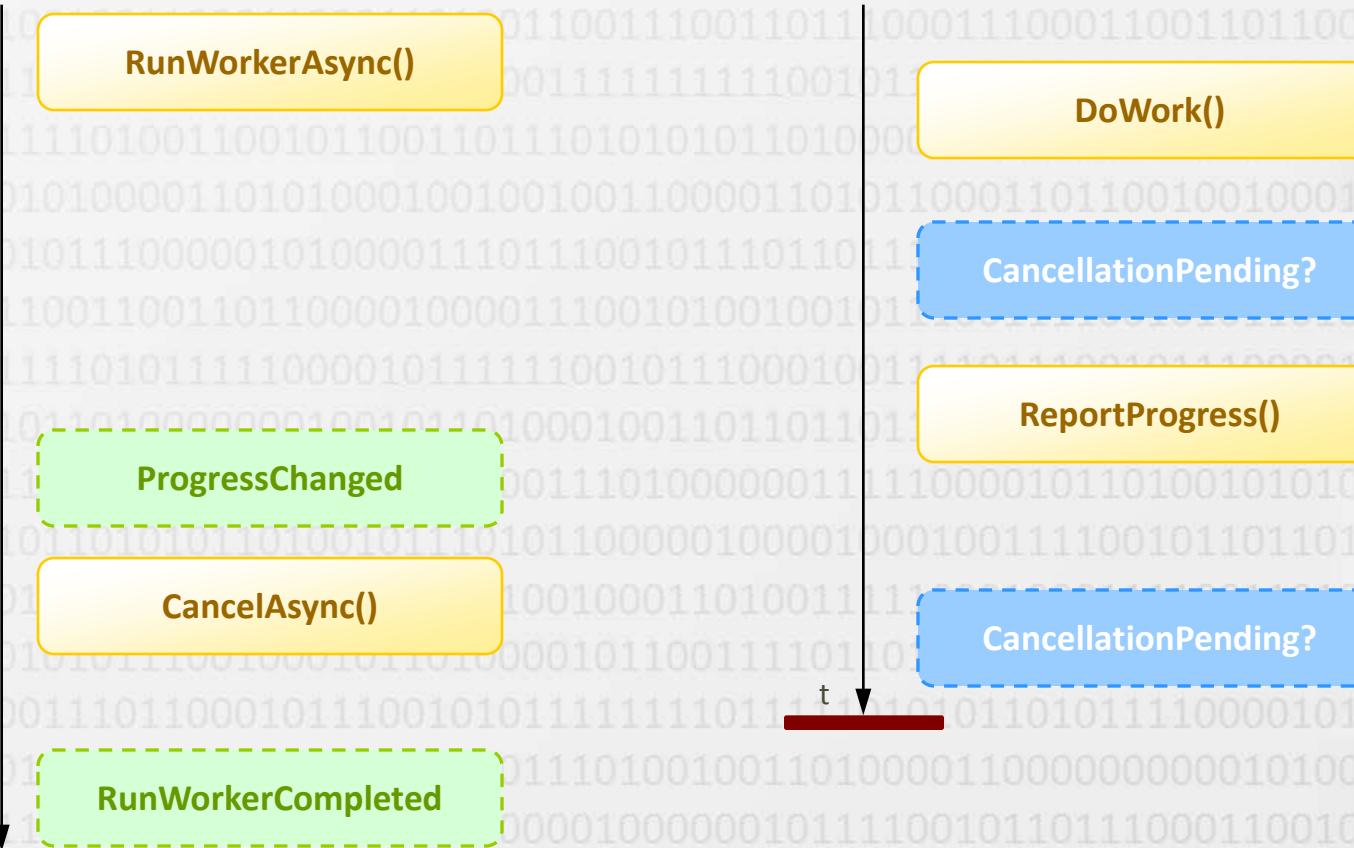
Használatánál ügyelni kell rá, hogy a Windows operációs rendszerben a felhasználói felület elemeihez csak az a szál férhet hozzá, amely az adott elemet létrehozta, tehát a BackgroundWorker háttérszálában sem közvetlenül, sem közvetve nem hivatkozhatunk a felhasználói felület semmilyen elemére.



A BackgroundWorker működése (illusztráció)

UI szál

Háttérszál



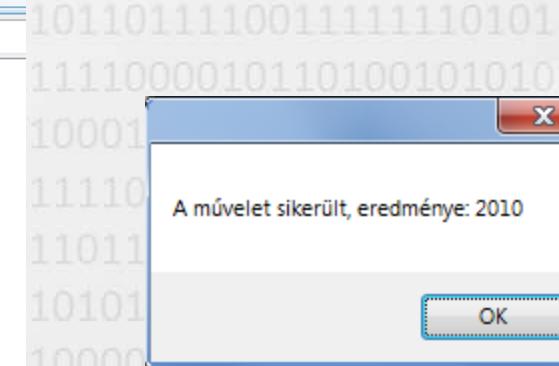
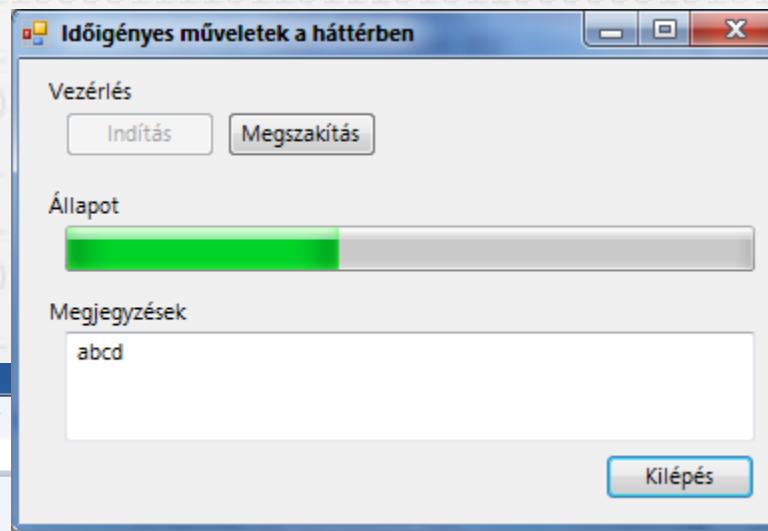
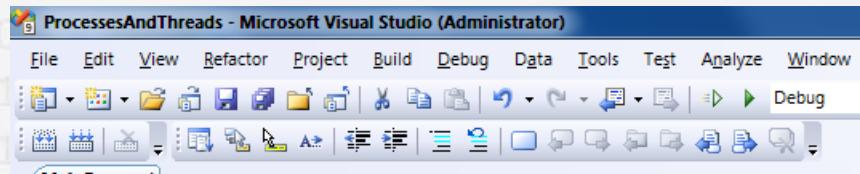
Feladat (5)

Készítsünk Windows alkalmazást, amely időigényes műveletet futtat a háttérben, a felhasználói felülettől független szalon! A program legyen képes a művelet indítására és menet közbeni biztonságos megszakítására, a háttérben futó művelet állapotát pedig folyamatjelzővel jelezze!

Ötletek:

- Célszerű a BackgroundWorker osztály segítségével megoldani a feladatot
- A háttérben futó szál a ReportProgress() metódussal jelezheti az előrehaladást (ezt az adatot a ProgressChanged esemény második paraméterében kapja meg a megfelelő eseménykezelő metódus)
- A művelet megszakítását csak akkor kíséreljük meg, ha valóban fut (ez az IsBusy tulajdonság vizsgálatával állapítható meg)
- A háttérben futó művelet végén a RunWorkerCompleted esemény kezelője a második paraméterben kap információt a műveletről (véget ért vagy megszakítás miatt fejeződött be, mi a végeredmény, történt-e hiba stb.)

Megoldás (5)



```
15:     private void backgroundWorker_DoWork(object sender, DoWorkEventArgs e)
16:     {
17:         for (int i = 10; i <= 100; i += 10)
18:         {
19:             if (backgroundWorker.CancellationPending)
20:             {
21:                 e.Cancel = true;
22:                 return;
23:             }
24:             // Időigényes részművelet
25:             Thread.Sleep(100);
26:
27:             // Ebben a metódusban közvetlenül nem módosíthatjuk a felhasználói felületet,
28:             // csak jelezhetjük az előrehaladást
29:             backgroundWorker.ReportProgress(i);
30:
31:         }
32:         e.Result = 2010;
33:     }
```

BackgroundWorker

Szálak kezelése (kivonatos referencia)

- **System.ComponentModel.BackgroundWorker osztály**

Metódusok	
RunWorkerAsync()	Háttérszál indítása
CancelAsync()	Háttérszál leállítása
ReportProgress	Háttérszál folyamatjelzése
Tulajdonságok	
IsBusy	A háttérszál aktív-e (éppen fut-e)
CancellationPending	Leállítás folyamatban (leállási kérelem érkezett)
WorkerSupportsCancellation	A háttérszál kérés esetén képes idő előtti leállásra
WorkerReportsProgress	A háttérszál képes folyamatjelzésre
Események	
DoWork	Kezelője a háttérben futtatandó metódus*
ProgressChanged	A háttérszál folyamatjelzését fogadó esemény
RunWorkerCompleted	A háttérszál futása befejeződött

* Ez a metódus (a Windows UI megvalósítási modellje következtében) közvetlenül nem érintkezhet a felhasználói felület elemeivel. Ezek szükséges frissítését és állapotmódosításait a ProgressChanged és a RunWorkerCompleted eseménykezelőkben lehet elvégezni.

Aszinkron metódushívás

- **Új képviselő definiálásakor a C# fordító olyan új osztályt hoz létre, amellyel szinkron és aszinkron módon is meghívható(k) a képviselt metódus(ok)**

A képviselő közvetlen (szinkron) hívásakor a képviselő `Invoke()` metódusa hajtódi végre, azaz a fordító a **képviselőPéldány()** szintaxissal megadott „metódushívásokat” automatikusan **képviselőPéldány.Invoke()** hívásra alakítja át – ezért lehet tehát egyszerű szintaxissal „meghívni egy képviselőt”. Aszinkron hívás esetén a képviselt metódus(ok) két „részletben” hívható(k):

delegate «visszatérési típus» **Képviselő**(«paraméterek»);

```
Képviselő k = new Képviselő(«hívandó metódus»);
IAsyncResult<objektum> = k.BeginInvoke(«argumentumok»);
// ...tetszőleges kód, amely a képviselt metódussal párhuzamosan fut majd
«visszatérési típus» éredmény = k.EndInvoke(«objektum»);
```

- **A fenti ún. aszinkron tervezési mintát (Begin.../End...) a képviselőkön kívül számos .NET osztály is megvalósítja**

Fájl- és hálózatkezelés, távoli objektumok, webszolgáltatások, üzenetsorok

Példa aszinkron metódushívásra

```
1  using System;
2  using System.Threading;
3
4  class Program
5  {
6      delegate int TestDelegate(string parameter);
7
8      static void Main(string[] args)
9      {
10         Console.WriteLine("A főszál elindult");
11         TestDelegate d = Method; // .NET 1.1 esetén: TestDelegate d = new TestDelegate(Method);
12         IAsyncResult asyncResult = d.BeginInvoke("Szervusz, világ!", null, null);
13         Console.WriteLine("A főszál kezdeményezte a metódushívást és párhuzamosan tovább fut");
14         int result = d.EndInvoke(asyncResult);
15         Console.WriteLine("A metódus lefutott, visszatérési értéke: {0}", result);
16     }
17
18     static int Method(string argument)
19     {
20         Console.WriteLine("\tA metódus párhuzamosan fut, kapott paramétere: {0}", argument);
21         return 5;
22     }
23 }
```

Szinkronizáció

- A párhuzamos programozás alapproblémája a *versenyhelyzet*: bármely két utasítás végrehajtása között előfordulhat, hogy más szál fér hozzá az előbbi szál által (is) kezelt közös adatokhoz

Egyprocesszoros rendszereknél az operációs rendszer ütemezője (a szálak közötti váltás) ad erre lehetőséget, többprocesszoros rendszereknél pedig a valódi (fizikai) párhuzamosság miatt még gyakrabban merül fel a probléma.

Ennek elkerülését szolgálják elsősorban a különböző szinkronizációs megoldások (másik, ezzel összefüggő céljuk az időzítések összehangolása).

- A szinkronizáció olyan, párhuzamos szálak (vagy folyamatok) együttműködését megvalósító mechanizmus, amely minden körülmények között biztosítja a szálak (vagy folyamatok) által végzett műveletek szemantikai helyességét

A párhuzamosan futó szálak kommunikációjához szinte biztosan szükség van közös erőforrások (memória, portok, I/O eszközök, fájlok) használatára. Ha ezek állapotát egy szál módosítja, de közben más szálak is hozzájuk férnek, akkor az utóbbi szálak könnyen hibás vagy félkész adatokhoz juthatnak.

Példa „egyszerű” C# kód összetettségére

The diagram illustrates the compilation process of a simple C# program. On the left, the `Program.cs` file is shown with its source code. A callout box points to a specific line of code (line 8) and contains the text: "A sötétebb színű vonalak olyan lehetséges megszakítási pontokat jelölnek, amelyek szinkronizációs problémákat okozhatnak". This refers to the darker-colored lines in the assembly code, which represent potential break points for synchronization issues. An orange arrow points from the C# code to the generated assembly code on the right. The assembly code shows the compiled instructions for the `Main()` method.

```
Program.cs
1: using System;
2:
3: class Program
4: {
5:     static void Main()
6:     {
7:         string message = "Hello";
8:         message += " world";
9:         Console.WriteLine(message);
10:    }
11: }
```

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (
        [0] string message)
    L_0000: nop
    L_0001: ldstr "Hello"
    L_0006: stloc.0
    L_0007: ldloc.0
    L_0008: ldstr " world"
    L_000d: call string [mscorlib]System.String::Concat(string, string)
    L_0012: stloc.0
    L_0013: ldloc.0
    L_0014: call void [mscorlib]System.Console::WriteLine(string)
    L_0019: nop
    L_001a: ret
}
```

```
--- D:\Documents\Visual Studio 2005\Projects\Experiments\
1: using System;
2:
3: class Program
4: {
5:     static void Main()
6:     {
7:         string message = "Hello";
8:         message += " world";
9:         Console.WriteLine(message);
10:    }
11: }
```

```
00000000 push    ebp
00000001 mov     ebp,esp
00000003 push    edi
00000004 push    esi
00000005 push    ebx
00000006 sub     esp,30h
00000009 xor     eax,eax
0000000b mov     dword ptr [ebp-10h],eax
0000000e xor     eax,eax
00000010 mov     dword ptr [ebp-1Ch],eax
00000013 cmp     dword ptr ds:[009686F8h],0
0000001a je      00000021
0000001c call    793FF3A9
00000021 xor     esi,esi
00000023 nop
7:          string message = "Hello";
00000024 mov     eax,dword ptr ds:[022B307Ch]
0000002a mov     esi,eax
8:          message += " world";
0000002c mov     edx,dword ptr ds:[022B3080h]
00000032 mov     ecx,esi
00000034 call    786B6E28
00000039 mov     edi,eax
0000003b mov     esi,edi
9:          Console.WriteLine(message);
0000003d mov     ecx,esi
0000003f call    787188D0
00000044 nop
10:         }
00000045 nop
00000046 lea     esp,[ebp-0Ch]
00000049 pop    ebx
0000004a pop    esi
0000004b pop    edi
0000004c pop    ebp
0000004d ret
```

Szinkronizáció kölcsönös kizárással

- **Kritikus szakasz („critical section”)**

A programokon belül megjelölt kritikus kódrészletek soros végrehajtását biztosítja több párhuzamos szál esetén is.

.NET osztályok: *System.Threading.Monitor* (és a C# „lock” utasítása), *System.Threading.Mutex*, *System.Threading.ReaderWriterLock*

- **Szemafor („semaphore”)**

A kritikus szakasz általánosítása (többpéldányos erőforrások esetén egyszerre több szál belépését is lehetővé teszi).

.NET osztály: *System.Threading.Semaphore* (.NET 2.0+)

- **Atomi végrehajtás („atomic/interlocked execution”)**

Egyes egyszerű műveletek oszthatatlan végrehajtását biztosítja (igen gyors).

.NET osztály: *System.Threading.Interlocked*

- **Csővezeték („pipe”)**

Olvasható és írható FIFO puffer, amely szükség szerint várakoztatja az igénylőket (az ún. „termelő-fogyasztó” probléma megoldására készült).

Szinkronizáció bevárással (randevú)

- **Esemény („event”)**

Két kódrészlet soros végrehajtását biztosítja úgy, hogy a „B” kódrészletet végrehajtó szál megvárja, amíg az „A” kódrészletet végrehajtó szál végez feladatával, illetve lehetőséget ad alkalmankénti vagy rendszeres jelzésre is.

.NET osztályok: System.Threading.Thread,
System.Threading.AutoResetEvent, System.Threading.ManualResetEvent

- **Időzítő („timer”)**

Relatív vagy abszolút időhöz való igazodást tesz lehetővé.

.NET osztályok: System.Windows.Forms.Timer, System.Timers.Timer,
System.Threading.Timer

Időzítők jellemzői	Windows.Forms.Timer	Timers.Timer	Threading.Timer
Időmérés pontossága	~10–15 ms	~1 ms	~1 ms
Futtatás saját szálon	–	+	+
Beállítható első aktiválás	–	–	+
Vizuális komponens	+	–	–
UI szálra átlépés	+	+	–
Platformfüggetlen	–	–	+

Szinkronizációs elemek I.

- **Egyeszerű blokkoló metódusok***

Konstrukció	Cél
<code>Thread.Sleep()</code>	Adott szál blokkolása meghatározott ideig
<code>Thread.Join()</code>	Másik szál befejeződésének megvárása

- **Zároló konstrukciók, kritikus szakasz objektumok***

Konstrukció	Cél	Folyamatok között is?	Sebesség
lock (C# kulcsszó)	Csak egy szál férhessen az erőforráshoz vagy kódrészhez	–	+
Monitor (Enter(), Exit())	Csak egy szál férhessen az erőforráshoz vagy kódrészhez	–	+
Mutex	Csak egy szál férhessen az erőforráshoz vagy kódrészhez.	+	–
Semaphore	Meghatározott számú szál férhessen az erőforráshoz vagy kódrészhez	+	–

* Ha a szálat e konstrukciók várakoztatják, akkor *blokkolódik* és WaitSleepJoin állapotba kerül (ilyenkor a rendszer nem ütemezí).

Szinkronizációs elemek II.

- **Jelzőkonstrukciók (szignálok)***

Konstrukció	Cél	Folyamatok között is?	Sebesség
EventWaitHandle	Engedélyezi, hogy a szál várjon, amíg egy másik száltól jelzést nem kap	+	-
Monitor (Wait(), Pulse())**	Engedélyezi, hogy a szál várjon, amíg egy beállított blokkoló feltétel teljesül	-	-

* Ha a szálat e konstrukciók várakoztatják, akkor *blokkolódik* és WaitSleepJoin állapotba kerül (ilyenkor a rendszer nem ütemezí).

** A Pulse() egyirányú kommunikáció, azaz nincs visszajelzés. Sőt, azt sem tudhatjuk pontosan, hogy a hozzá kapcsolódó Wait()-nél mennyi idővel később szűnik meg a blokkolás. Használunk állapotjelzőket!

- **Nem blokkoló jellegű szinkronizációs elemek**

Konstrukció	Cél	Folyamatok között is?	Sebesség
Interlocked	Nem blokkoló jellegű atomi vérehajtást biztosít		++
volatile (C# kulcsszó)	Zároláson kívüli, biztonságos, nem blokkoló jellegű hozzáférést biztosít egyes mezőkhöz		++

Kölcsönös kizáráás a „lock” utasítással

- A „lock” utasítás használata

lock (változó)
utasítás

```
1 class Account
2 {
3     long value = 0;
4     object obj = new object();
5
6     public void Deposit(long amount)
7     {
8         lock (obj) { value += amount; }
9     }
10
11    public void Withdraw(long amount)
12    {
13        lock (obj) { value -= amount; }
14    }
15 }
```

Példa: kölcsönös kizáráás a „lock” utasítással

```
1  using System;
2  using System.Threading;
3
4  class Program
5  {
6      private static int counter = 0;
7      private static object lockObject = new Object();
8
9      static void Main(string[] args)
10     {
11         Thread t1 = new Thread(ThreadMethod);
12         t1.Start();
13         Thread t2 = new Thread(ThreadMethod);
14         t2.Start();
15     }
16
17     private static void ThreadMethod()
18     {
19         lock (lockObject)
20         {
21             counter++;
22             Thread.Sleep(500);
23             Console.WriteLine("A számláló állása: " + counter);
24         }
25     }
26 }
```

Figyelem: **SOHA** ne írunk le az alábbiakra hasonlító kódot:

~~lock (this)~~
~~lock (typeof(Program))~~

A lock utasítás nélkül a metódus sorosan (egy szálon futtatva) helyesen működik, párhuzamosan (több szálon) azonban nem

A kölcsönös kizárási problémái

- **Holtpont („deadlock”)**

Akkor léphet fel holtpont, ha több szál több erőforráshoz kíván hozzáférni, miközben egyes erőforrásokat lefoglalva tartanak (tehát már beléptek egy erőforráshoz tartozó kritikus szakaszba, ott viszont várakozniuk kell, hogy hozzájuthassanak egy másik szükséges erőforráshoz). Egy példa:

```
1 lock (a)
2 {
3     // feldolgozás
4     lock (b)
5     {
6         // feldolgozás
7     }
8 }
```

1. szál

```
1 lock (b)
2 {
3     // feldolgozás
4     lock (a)
5     {
6         // feldolgozás
7     }
8 }
```

2. szál

- **Élőpont („livelock”)**

Akkor beszélünk élőpontról, amikor az egymásba ágyazott zárolások miatt a végrehajtás ugyan nem áll le, de egy rövid szakaszban a végtelenségig ismétlődik minden érintett szalon.

A holtpontnál sokkal ritkábban fordul elő, kezelése azonban még nehezebb.

Kölcsönös kizáráás az Interlocked osztállyal

- A **System.Threading.Interlocked** statikus osztály metódusai atomi műveleteket valósítanak meg

Az osztály metódusai oszthatatlan módon hajtódnak végre (a processzor hardveres támogatását kihasználva). Segítségével egyszerűbb esetekben szükségtelenné válik a zárolás használata, és javul az elérhető teljesítmény.

- **System.Threading.Interlocked** osztály

Metódusok	
Add()	A megadott 32 vagy 64 bites egész érték hozzáadása a hivatkozott változó értékéhez
CompareExchange()	A hivatkozott változó értékének feltételes beállítása összehasonlítás alapján
Decrement(), Increment()	A hivatkozott változó értékének csökkentése, illetve növelése 1-gyel
Exchange()	A hivatkozott változó értékének beállítása és az eredeti érték visszaadása
Read()	A hivatkozott 64 bites változó értékének olvasása (32 bites rendszeren is atomi műveletként)

A Monitor osztály

- A System.Threading.Monitor statikus osztály elsődleges feladata exkluzív hozzáférést biztosítani megadott változókhöz

Ezzel a megoldással valósítható meg a .NET keretrendszerben a kölcsönös kizárás és a feltételváltozókhöz kötött várakoztatás, illetve értesítés.

A C# beépített „lock” utasítása a valóságban a Monitor osztályra épülő speciális programrészlet rövidített, garantáltan hibamentes változatban:

```
1 // kritikus szakasz előtti kód  
2 lock  
3 {  
4     // kritikus szakaszban lévő kód  
5 }  
6 // kritikus szakasz utáni kód
```

```
1 // kritikus szakasz előtti kód  
2 Monitor.Enter(a);  
3 try  
4 {  
5     // kritikus szakaszban lévő kód  
6     goto Label_0001;  
7 }  
8 finally  
9 {  
10    Monitor.Exit(a);  
11 }  
12 Label_0001:  
13 // kritikus szakasz utáni kód
```



Kölcsönös kizáráás a Monitor osztállyal

- Erre a célra az Enter(), TryEnter() és Exit() metódusok szolgálnak
 - Enter() metódus: belépés a kritikus szakaszba (blokkolással)
 - TryEnter() metódus: belépés a kritikus szakaszba (blokkolás nélkül)
 - Exit() metódus: kilépés a kritikus szakaszból

```
1 class MonitorMutualExclusionExample
2 {
3     private object lockObject = new object();
4     private Queue Ipt = new Queue();
5
6     void EnqueueBlocking(object element)
7     {
8         Monitor.Enter(lockObject);
9         try
10        {
11            Ipt.Enqueue(element);
12        }
13        finally
14        {
15            Monitor.Exit(lockObject);
16        }
17    }
```

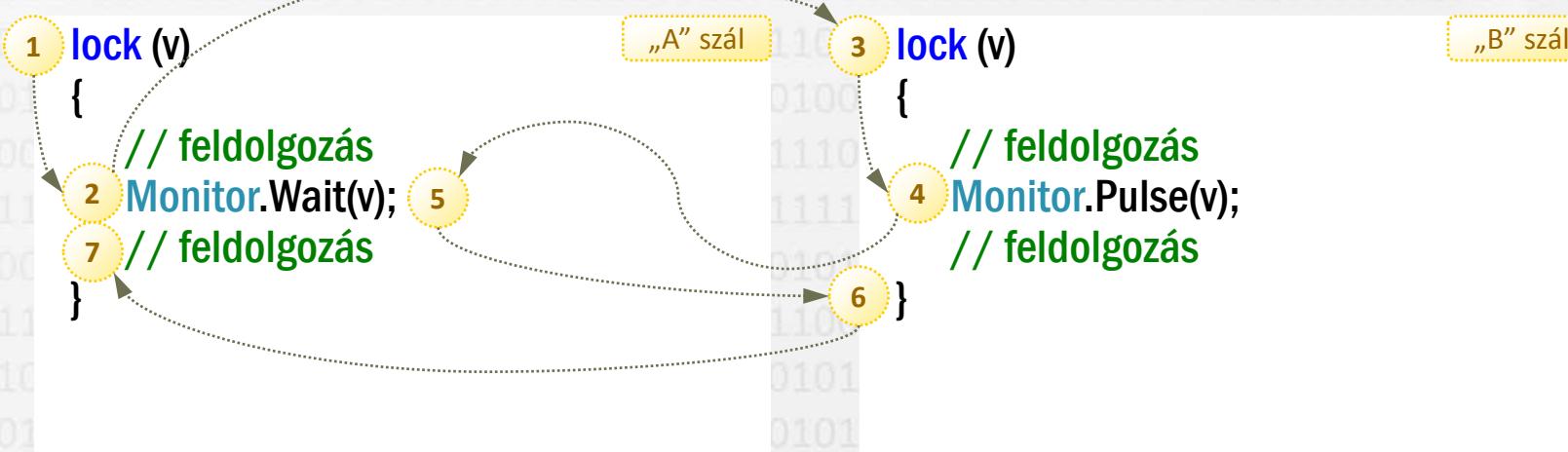
```
18     bool EnqueueNonBlocking(object element)
19     {
20         if (!Monitor.TryEnter(lockObject))
21             return false;
22         try
23         {
24             Ipt.Enqueue(element);
25         }
26         finally
27         {
28             Monitor.Exit(lockObject);
29         }
30         return true;
31     }
32 }
```

Szálak szinkronizációja a Monitor osztállyal

- **Erre a célra a Wait(), Pulse() és PulseAll() metódusok szolgálnak**
 - Wait() metódus: kilépés a kritikus szakaszból és várakozás az újabb belépési lehetőségre (egy másik szál a Pulse()/PulseAll() metódussal értesíti az előzőt)
 - Pulse() metódus: a kritikus szakaszra váró szálak közül az első felébresztése
 - PulseAll() metódus: a kritikus szakaszra váró összes szál egyidejű felébresztése
- **Mindhárom metódus csak akkor hívható, ha a hívó szál előzőleg sikeresen zárolta a kritikus szakaszt**

Amennyiben ezt a feltételt nem tartjuk be, a keretrendszer egy System.Threading.SynchronizationLockException típusú kivételt vált ki.

Elvi példa: Monitor.Wait() és Monitor.Pulse()



- A eléri a lock(v) programsort és belép a(z éppen szabad) kritikus szakaszba
 - A eléri a Monitor.Wait(v) hívást, aludni megy és megszünteti a zárolást
 - B eléri a lock(v) programsort és belép a (már felszabadult) kritikus szakaszba
 - B eléri a Monitor.Pulse(v) hívást és felébreszti A-t
- Lehetséges, hogy ekkor azonnal kontextusváltás történik A és B között, de ez nem garantált
- A szeretne belépni a kritikus szakaszba, de nem tud, mert B még ott van
 - A kritikus szakasz végén B megszünteti a zárolást és tovább fut
 - A visszalép a kritikus szakaszba és tovább fut



Monitor.Wait() és Monitor.Pulse()

- **Megjegyzések:**

- A Pulse(v) és a felébresztett szál folytatása között más szálak is futhatnak, amelyek időközben megpróbálhattak belépni a kritikus szakaszba. Tehát a Pulse(v) által jelzett feltétel már nem biztosan igaz, mire a frissen felébresztett szál a Wait(v) után folytatja futását.

Ezért a Wait(v) metódushívást egy ciklusban kell elhelyezni, amely folyamatosan teszteli a feltételt:

```
1  while /*a feltétel hamis*/
2      Monitor.Wait(v);
3      // feldolgozás
4      // a feltétel igazra állítása
5      Monitor.Pulse(v);
```

- A PulseAll(v) felébreszt minden szálat, amely v-re vár. Ezek közül csak egy olyan szál folytathatja futását, amely ugyanerre a feltételre várt, a többinek várnia kell, amíg a zárolást el nem engedi az előző szál. Ekkor a következő szál léphet a kritikus régióba.

A PulseAll() elsődleges célja tehát lehetőséget adni egyazon kritikus szakaszban különböző feltételekre váró szálak egyidejű felébresztésére.

Wait és Pulse – szinkronizált puffer példa

```
public class Buffer {  
    const int size = 16;  
    char[ ] buf = new char[size];  
    int head = 0, tail = 0, n = 0;
```

Ha a termelő gyorsabb:

```
Put  
Put  
Put  
Get  
Put  
Get
```

```
public void Put(char ch) {  
    lock(obj) {  
        while (n == size) Monitor.Wait(obj);  
        buf[tail] = ch; tail = (tail + 1) % size; n++;  
        Monitor.Pulse(obj);  
    }  
}
```

thread 1

Lock a karakter hozzáadása érdekében
Amíg tele a puffer, fenntartjuk a lockot
A várakozó szálak felébresztése

```
public char Get() {  
    lock(obj) {  
        while (n == 0) Monitor.Wait(obj);  
        char ch = buf[head]; head = (head + 1) % size; n--;  
        Monitor.Pulse(obj);  
        return ch;  
    }  
}
```

thread 2

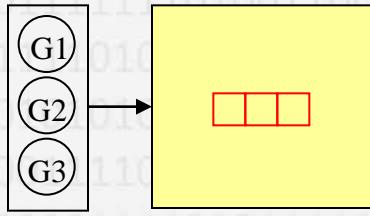
Puffer lockolása a karakter betöltéséhez
Amíg üres a puffer, lock és várakozás
A várakozó szálak felébresztése

Ha a fogyasztó a gyorsabb:

```
Put  
Get  
Put  
Get  
...
```

Szinkronizált puffer

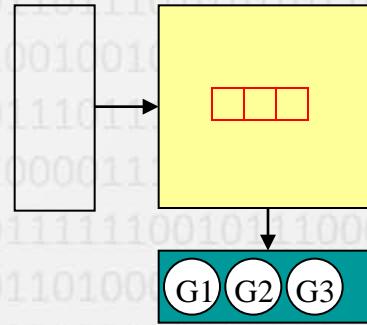
3 Get szál eléri az üres puffer



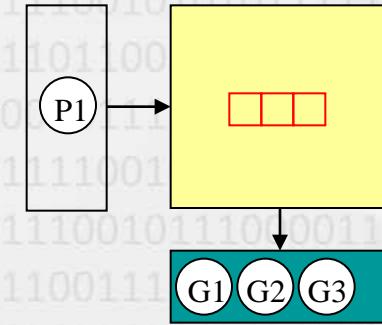
Első belépés

Kritikus
régió

Belépnek a kritikus régióba és aludni mennek, mert üres a puffer



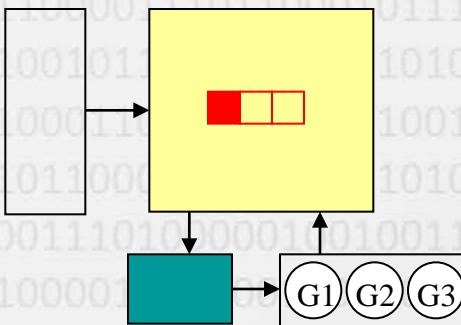
Egy Put szál érkezik, Kritikus régióba lép; Elhelyezi adatát és jelez: PulseAll



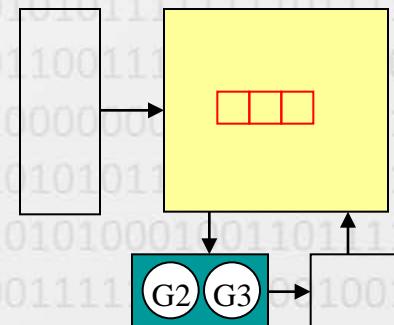
Várakozás

Mindenl Get szál felébred; Az első a kritikus régióba lép, kiolvassa az adatot és kilép

A többiek ismét várakozási állapotba kerülnek, mert üres a puffer



Ismételt belépés



A Monitor osztály (kivonatos referencia)

- **System.Threading.Monitor osztály**

Metódusok	
Enter()	Belépés adott objektum által védett kritikus szakaszba (ha szükséges, blokkolással)
TryEnter()	Belépési kísérlet adott objektum által védett kritikus szakaszba (blokkolás nélkül)
Exit()	Kilépés adott objektum által védett kritikus szakaszból
Wait()	Kilépés adott objektum által védett kritikus szakaszból, és várakozás az ismételt belépési lehetőségre
Pulse(), PulseAll()	Adott objektum által védett kritikus szakaszra váró szálak közül az első vagy az összes felébresztése

Feladat (6)

Készítsünk többszálú konzolos alkalmazást, amely egy időigényes számítási műveletet 2 szállal párhuzamosan végeztet el!

A műveletet most egy közösen használt számláló folyamatos növelése jelentse, és egy szálnak kb. 2-3 másodpercig tartson a művelet elvégzése! Amennyiben szükséges, gondoskodjon a szálak szinkronizációjáról is! A program valamilyen megoldással biztosítsa a szálak által kiírt adatok vizuális elkülönítését!

Ötletek:

- A Stopwatch osztály metódusai segítségével egyszerűen mérhető az eltelt (relatív) idő
- Először egy egyszerű megvalósítással döntsük el, szükség van-e szinkronizációra, majd ha úgy ítéljük meg, hogy igen, akkor használjuk a lock utasítást vagy a Monitor osztály statikus Enter(), illetve és Exit() metódusát
- Szinkronizáció esetén a jobb teljesítmény érdekében igyekezzünk a lehető legrövidebbre venni a kritikus szakaszt

Feladat (7)

Készítsünk konzolos alkalmazást, amelynek csak egy példánya futhat egyszerre. A megvalósításhoz használjuk a Mutex osztályt! (Készítsünk el ugyanilyen funkciójú alkalmazást folyamatok lekérdezésével és vizsgálatával is!)

Ötletek:

- A Mutex „name” paramétere egyedi legyen, pl.: „oenik.hu MUTEXv01”
- A Mutex WaitOne() és ReleaseMutex() metódusait használjuk. Az első metódus várakozási paramétert is kaphat, amit felhasználva például hibaüzenetet is megjeleníthetünk.
- Folyamatok esetében a GetCurrentProcess() és a GetProcesses() metódusok segíthetnek.

Feladat (8)

Készítsünk konzolos alkalmazást, amely a Semaphore osztály segítségével egy garázs működését prezentálja. A garázs kapacitása 10, benne a főnököknek foglalt helyek száma 5. A kocsik (szálak) véletlen ideig állnak a garázsban és összesen 20 tulajdonos jogosult a használatra. Az érkezéskor és induláskor írjuk ki, hogy melyik autó óhajt beállni, illetve elmenne.

Ötletek:

- Használjuk a Semaphore osztály WaitOne(), illetve Release() metódusát egy osztályszintű változón keresztül
- A szálfüggvény paraméterként kapja meg, hogy melyik autó szeretne beállni
- A parkolást a Thread.Sleep() metódussal valósítsuk meg

Feladat (9)

Készítsünk konzolos alkalmazást, amely az Interlocked osztály atomi metódusait teszteli!

Ötletek:

- Egy utasítás atomi, ha egyetlen, oszthatatlan műveletként hajtózik végre. A 32 bites adatok olvasása (32 bites processzoron) magától is atomi. Olvasást és írást kombináló utasítások nem atomiak (pl. az „`x++`” nem atomi).
- Az Interlocked osztály statikus metódusai nem blokkolnak és sosem kell ütemezésre várniuk (szemben pl. a lock utasítással)
- Az Interlocked osztály atomi műveleteket biztosít ref paraméterként átadott int, vagy long típusokra.
- Teszteljük az Increment(), Decrement(), Add(), Read(), Exchange() és CompareExchange() metódusok működését egy-egy long típusú paraméter használatával.

Feladat (10)

Készítsünk konzolos alkalmazást, amelynek elindított minden két szálja egymásnak ötször üzenetet küld. Az üzenet megérkezésekor az egyik szál azt írja ki a konzolra: ThreadPing, a másik ThreadPong!

Ötletek:

- Használjuk a Monitor osztály Pulse() és Wait() metódusait osztályszintű lock objektummal (pl. „private static object ball”)

Feladat (11)

Készítsünk demonstrációs alkalmazást, amely az AutoResetEvent és a ManualResetEvent működése közti különbséget mutatja be! Az elindított szál küldjön jelzést a főszálban adott ideig, többször várakozó ResetEvent-eknek. Írjuk ki, hogy ezek jel hatására milyen állapotba kerültek.

Ötletek:

- Használjuk az osztályok Set(), WaitOne() és Reset() metódusait.
- A szálfüggvényben a Thread.Sleep() használatával „szabályozzunk”

Felhasznált és javasolt irodalom

- [1] J. Albahari

Threading in C#

[online: 2011. 04. 30.] <http://www.albahari.com/threading/>

Párhuzamos programozási feladatok

B. Wilkinson és M. Allen oktatási anyaga alapján feladat javaslatok

Gravitációs N-test probléma

**Fizikai törvények alapján testek helyzetének, mozgásjellemzőinek és
a rájuk ható erőknek a meghatározása.**

Rekurzív szétosztás

Gravitációs N-test probléma - egyenletek

Az m_a és m_b tömegű testek közötti gravitációs erő:

$$F = \frac{m_a * m_b * g}{r^2}$$

g a gravitációs gyorsulás, r a két test távolsága. Az F erő Newton 2. törvénye alapján gyorsítja a testet:

$$F = m * a$$

m a test tömege és a a gyorsulás.

Az időintervallumot jelölje Δt . Az m tömegű testre ható erő:

$$F = m * \frac{(\nu^{t+1} - \nu^t)}{\Delta t}$$

Az új sebesség így:

$$\nu^{t+1} = \nu^t + \frac{F * \Delta t}{m}$$

ahol ν^{t+1} a sebesség a $t + 1$ időpillanatban és ν^t a sebesség t időpontban.

A Δt idő alatt a helyzetváltozás:

$$x^{t+1} - x^t = \nu * \Delta t$$

ahol x^t a pozíció t paraméter szerint.

A test mozgása során az erő változik. A számítást iterálni kell.

N-test probléma soros kódja

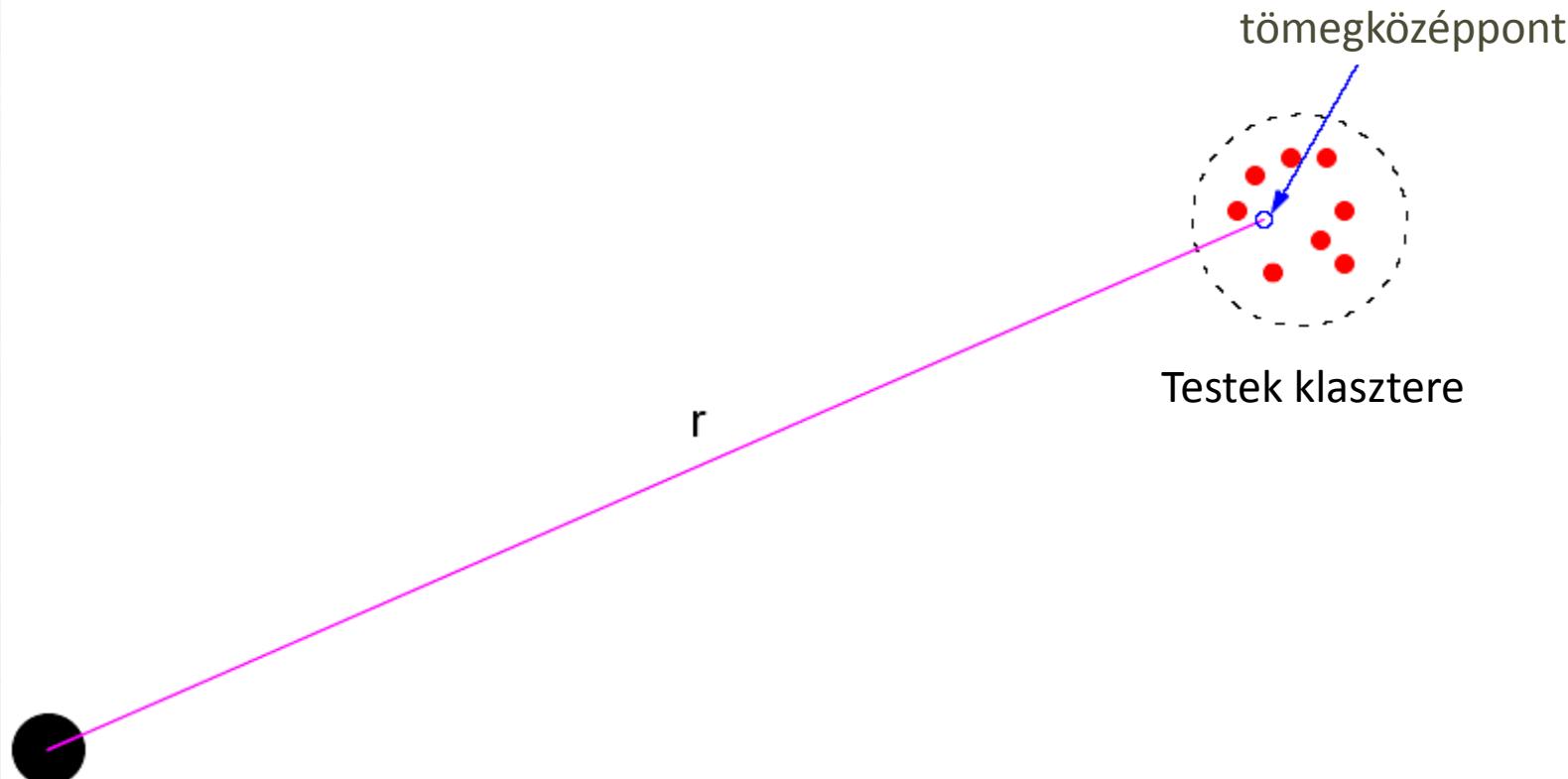
```
for (t = 0; t < tmax; t++)          /* minden időperiódusra */
    for (i = 0; i < N; i++) {        /* minden testre */
        F = Force_routine(i);       /* az i. testre ható erő */
        v[i]new = v[i] + F * dt / m; /* az új sebesség */
        x[i]new = x[i] + v[i]new * dt; /* és pozíció */
    }
    for (i = 0; i < N; i++) {        /* minden testre */
        x[i] = x[i]new;              /* a sebesség és a helyzet */
        v[i] = v[i]new;
    }
```

Párhuzamos kód indoka

A soros program $O(N^2)$ nagyságrendű egy iterációban, mivel minden az N test $N - 1$ testre fejt ki erőt.

Ha N nagy, akkor az N -test problémára nem hatékony a soros kód.

A komplexitás csökkenhető, ha a távoli testek csoportját egy tömegbe redukáljuk úgy, hogy a klasztert alkotó testek tömegközéppontjába transzformáljuk az össztömegeket:



Barnes-Hut algoritmus

Induljunk ki a teljes térből, ahol egy kocka tartalmazza a testeket.

- Elsőként osszuk a kockát nyolc részkockára.
- Ha egy részkocka nem tartalmaz testet, akkor a továbbiakban ne vegyük figyelembe.
- Ha a részkocka egy testet tartalmaz, akkor megőrizzük.
- Ha egy részkocka egynél több testet tartalmaz, akkor rekurzívan addig részkockákra bontjuk, amíg nem egy test lesz benne.

Készítsünk egy nyolcas fát (*octtree*) – ahol minden csomópontból maximum nyolc él indul ki.

A levelek cellákat reprezentálnak, amelyek pontosan egy testet tartalmaznak.

A fa konstruálása után a részkocka teljes tömegét, és a tömegközéppont koordinátáját tároljuk minden csomópontban.

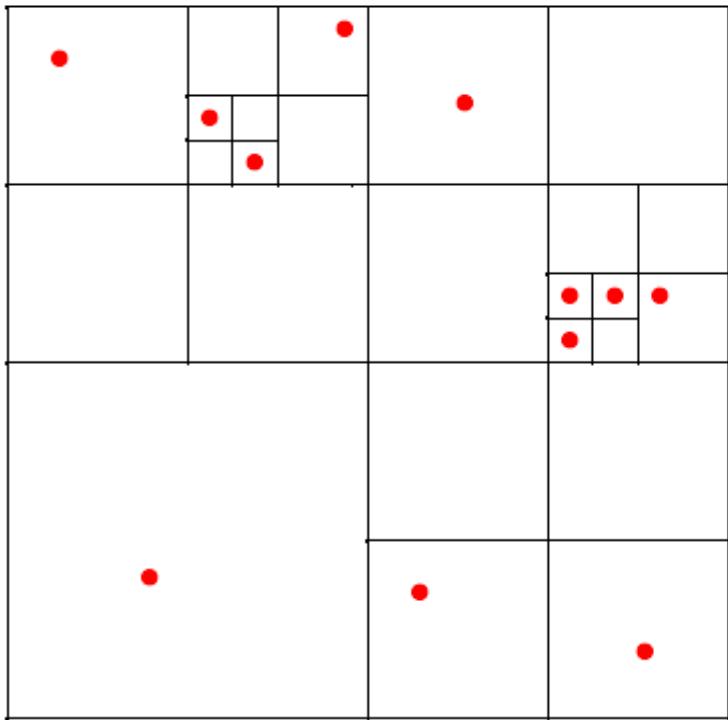
Minden testre ható erő megkapható, ha a fa gyökerétől elindulunk és eljutunk a csomópontig, ahol a klaszteres közelítés használható, azaz amikor:

$$r \geq \frac{d}{\theta}$$

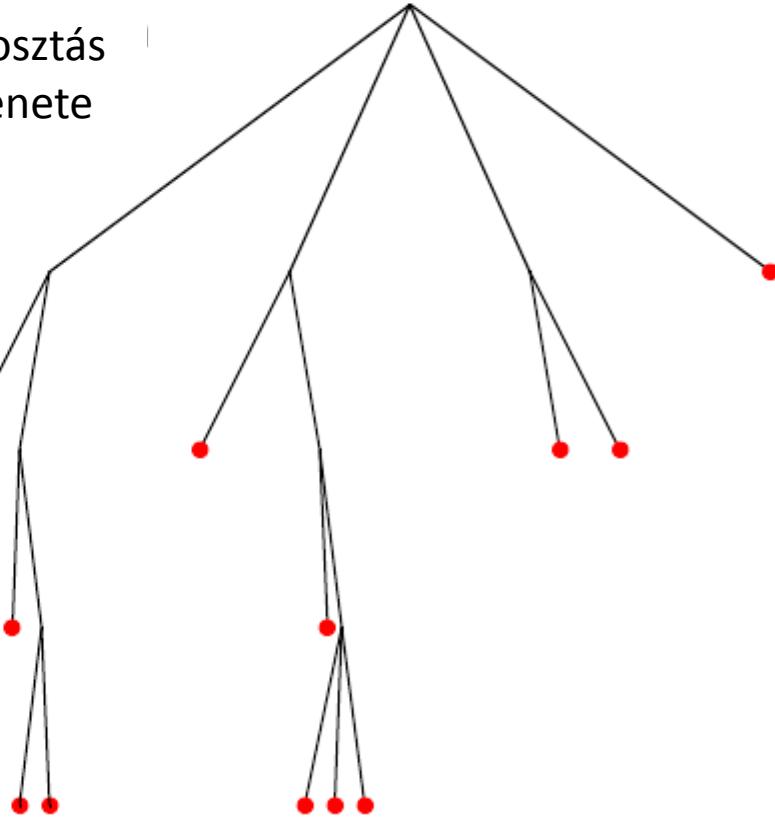
ahol θ konstans tipikusan egy, vagy annál kisebb.

A fa elkészítése $O(n \log n)$ nagyságrendű időt vesz igénybe, és az erőké is, így a teljes idő $O(n \log n)$.

2 dimenziós tér rekurzív felosztása

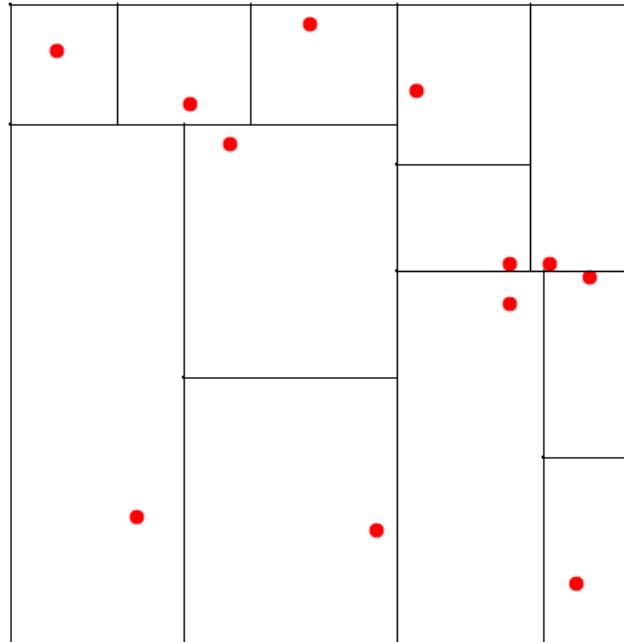


Felosztás
menete



Ortogonalis rekurzív kettéosztás

2D-s területben elsőként keressünk egy függőleges egyenest, amely úgy osztja két részre, hogy minden két részben azonos számú test essen. minden így kapott részt vízszintesen osszunk ketté egy egyenessel, hogy azonos számú testet tartalmazzanak. Amíg szükséges folytatunk a szétosztásokat.



Alacsony szintű képfeldolgozás

Nyilvánvalóan párhuzamosítható számítás.

Számos alacsony szintű képfeldolgozási művelet csak lokális adaton dolgozik, vagy esetleg egy szűk környezetből veszi inputját.

Geometriai transzformációk

- Objektum eltolása Dx értékkel x irányba és Dy értékkel y irányba:

$$x' = x + Dx$$

$$y' = y + Dy$$

ahol x és y az eredeti koordináták, x' , valamint y' az újak.

- Objektum skálázása x irányban S_x és y irányban S_y értékkel:

$$x' = x * S_x$$

$$y' = y * S_y$$

Geometriai transzformációk

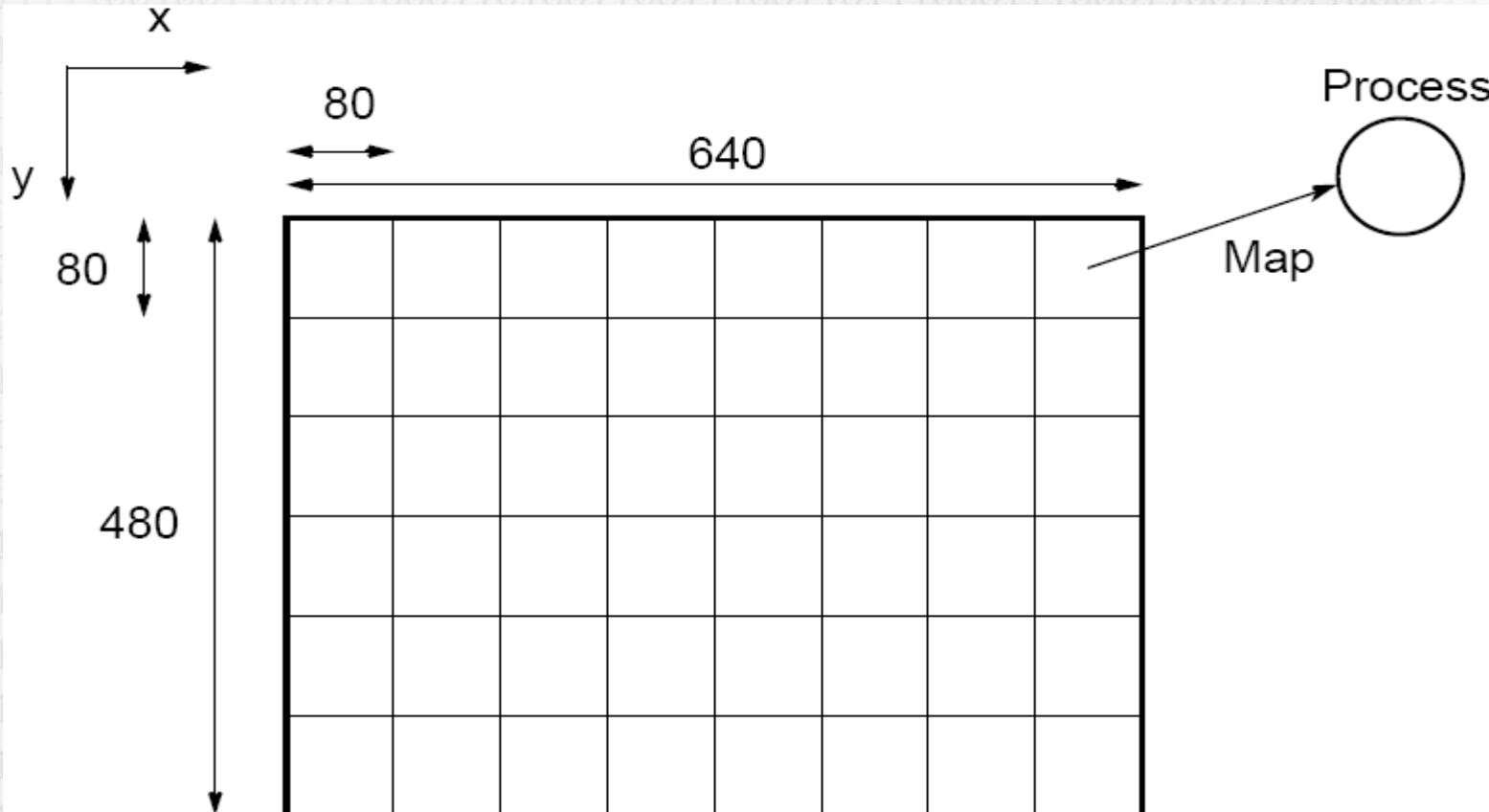
- Objektum forgatása θ szöggel a koordinátarendszer z tengelye körül:

$$x' = x * \cos \theta + y * \sin \theta$$

$$y' = -x * \sin \theta + y * \cos \theta$$

Régiókra particionálás

- minden régiót külön processzben számolhatunk



Mandelbrot halmaz

Nyilvánvalóan párhuzamosítható számítás

Mandelbrot halmaz

- A komplex sík pontjai, amelyeket iteráció során változtatunk, de egy határon belül maradnak.

$$z_{k+1} = z_k^2 + c$$

ahol z_{k+1} a $(k+1)$ -dik iterációja a $z = a + bi$ komplex számnak és c komplex szám a pont kezdeti helyzetét adja meg a komplex síkon. z kezdeti értéke nulla.

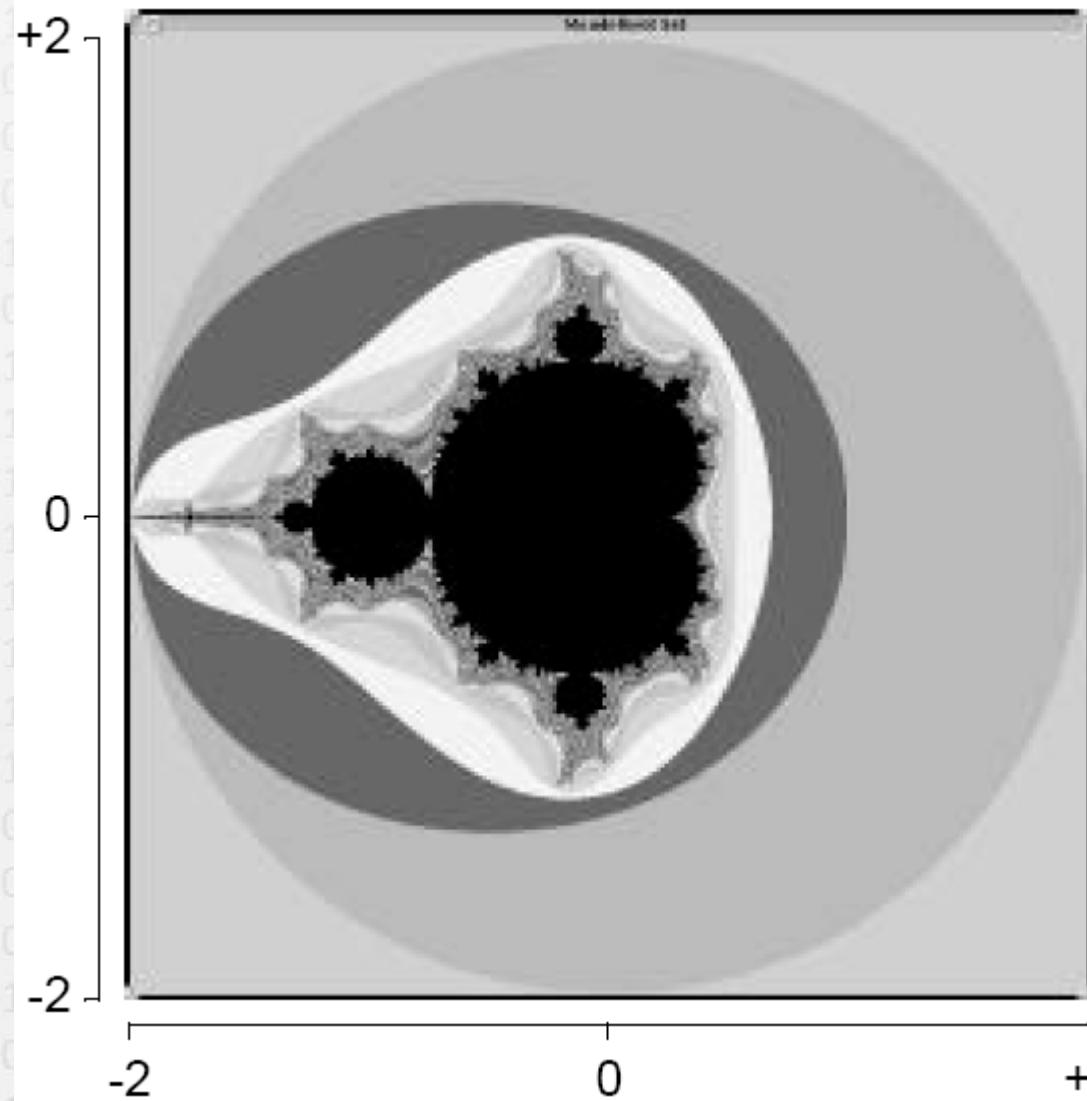
- Az iteráció addig folytatódik, amíg z nagysága kettőnél nem nagyobb, vagy az lépések száma egy határt el nem ér. A komplex z szám nagysága, mint vektorhossz:

$$z_{length} = \sqrt{a^2 + b^2}$$

Soros rutin egy pontra

```
structure complex {  
    float real;  
    float imag;  
};  
int cal_pixel(complex c)  
{  
    int count, max;  
    complex z;  
    float temp, lengthsq;  
    max = 256;  
    z.real = 0; z.imag = 0;  
    count = 0; /* number of iterations */  
    do {  
        temp = z.real * z.real - z.imag * z.imag + c.real;  
        z.imag = 2 * z.real * z.imag + c.imag;  
        z.real = temp;  
        lengthsq = z.real * z.real + z.imag * z.imag;  
        count++;  
    } while ((lengthsq < 4.0) && (count < max));  
    return count;  
}
```

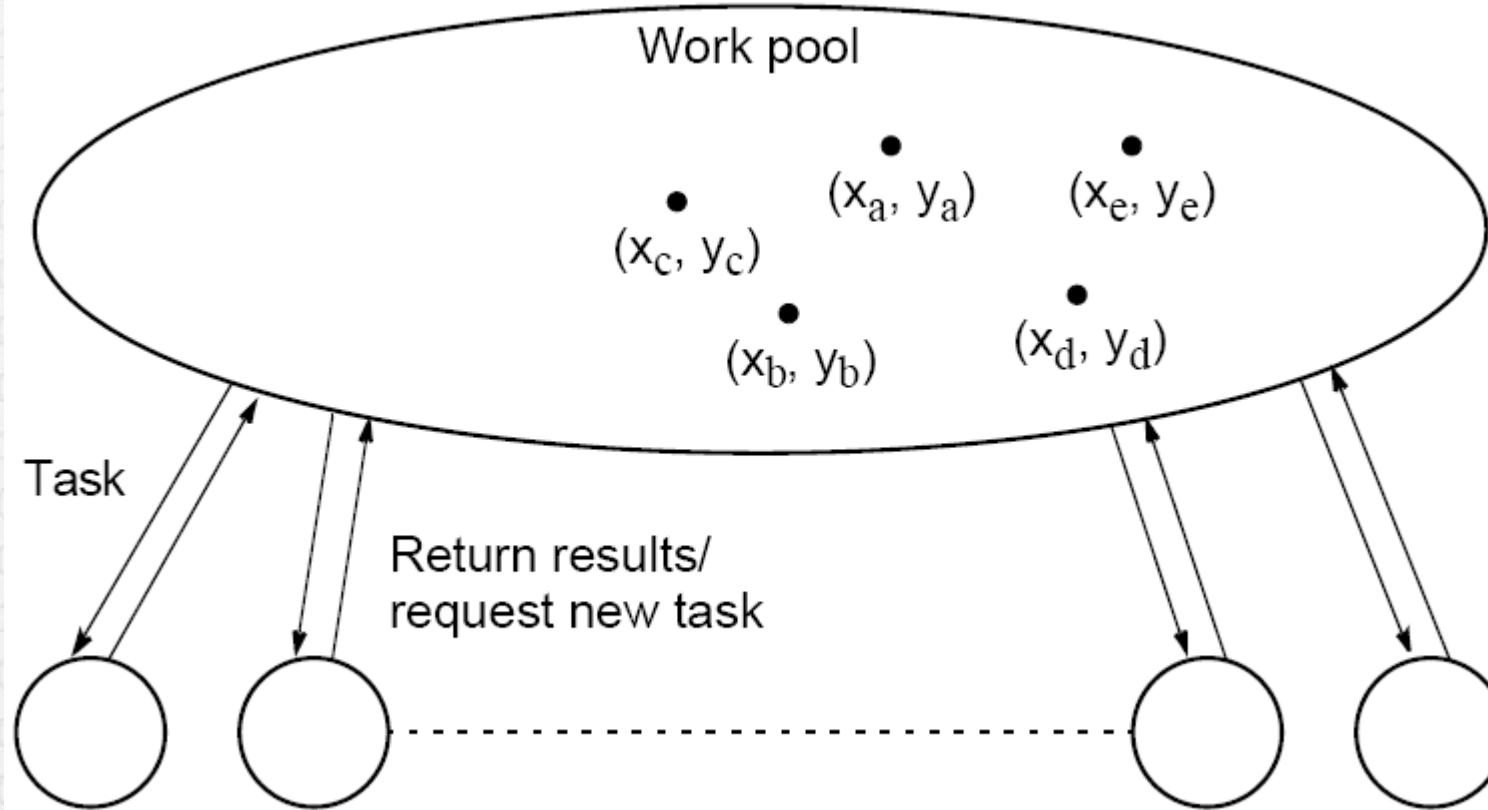
Mandelbrot halmaz



Párhuzamosítás

- **Statikus feladat kiosztás**
 - Egyszerűen osszuk a teret fix számú részre és minden részt külön processz számoljon.
 - Nem igazán hatékony, mert minden régió eltérő számú iterációt igényel.
- **Dinamikus feladat kiosztás (Work pool modell)**
 - Amikor a processz elkészül a korábbi számítással, új feladatot kap.

Work pool modell



Celluláris automata

Számítások randevú típusú szinkronizálással

Celluláris automata

- A feladatteret cellákra osztjuk.
- minden cella véges állapotok közül pontosan egyet vesz fel.
- A cellákra a szomszédjai valamelyen hatással vannak bizonyos szabály szerint, és minden cella egy „generációs” szabállyal is rendelkezik, amely szimultán fut.
- A szabályok újra és újra alkalmazásra kerülnek minden generációs lépésben, így a cellák fejlődnek, vagy megváltoztatják állapotukat generációról generációra.

A legismertebb celluláris automata John Horton Conway Cambridge-i matematikus „Élet játéka” (“Game of Life”).

Az élet játéka

- **Táblás játék – elméletben végtelen méretű kétdimenziós cellatömbbel. minden cellában egy organizmus lehet, nyolc szomszédja van. Kezdetben néhány cella foglalt.**
- **A következő szabályokat alkalmazzuk:**
 1. **Minden organizmus, amelynek két, vagy három szomszédja van, életben marad a következő generációra.**
 2. **Minden organizmus, amelynek négy, vagy több szomszédja van, meghal a túlnépesedés miatt.**
 3. **Minden organizmus, amelynek maximum egy szomszédja van, elpusztul az izoláltság miatt.**
 4. **Minden üres cellában, amelynek pontosan három nem üres szomszédja van egy új organizmus születik.**

E szabályokat Conway határozta meg hosszas kísérletek után.

Más egyszerű példa* – halak és cápák

- Az óceán három dimenziós tömbbel modellezhető
- minden cella tartalmazhat vagy halat, vagy cápát
- A halak és a cápák szabályokat követnek

*: Más példák is lehetnek: nyulak – rókák

Halak

- A következő szabályok szerint élnek:
 1. Ha valamely szomszédos cella üres, akkor odaúszik.
 2. Ha több szomszédos cella üres, akkor véletlenszerűen választ egyet.
 3. Ha nincs üres szomszédos cella, akkor helyben marad.
 4. Ha a hal mozog és eléri nemzési idejét (paraméter), akkor egy bábihalnak életet ad, amely a megüresedő cellába kerül.
 5. A hal x generáció után meghal.

Cápák

- A következő szabályok vonatkoznak rájuk:
 1. Ha valamely szomszédos cellában hal van, akkor a cápa abba a cellába megy és megeszi a halat.
 2. Ha több szomszédos cellában van hal, akkor a cápa véletlenszerűen választ egyet, abba a cellába megy és megeszi a halat.
 3. Ha nincs hal a szomszédos cellában, akkor a cápa egy nem foglalt szomszédos cellába mozog olyan szabályok szerint, mint a hal.
 4. Ha a cápa mozog és eléri nemzési idejét, akkor egy békicápának életet ad, amely a megüresedő cellába kerül.
 5. Ha a cápa y generación keresztül nem eszik, akkor meghal.

Valós CA példák

- **Folyadék/gáz dinamika**
- **Folyadékok és gázok áramlása testek körül**
- **Gázok diffúziója**
- **Biológiai fejlődés**
- **Légáramlás repülőgép légcsavarjánál**
- **Homok eróziója/mozgása folyóparton**

Részben szinkron számítás

- Nem minden iterációnál végezzük el a szükséges szinkronizációt
- Ez jelentősen csökkennetheti az átfutási időt, mert a randevú típusú szinkronizálás általában nagyon lelassítja a számításokat
- A globális szinkronizációt randevú rutinnal végezzük bizonyos időnként

Felhasznált és javasolt irodalom

- [2] B. Wilkinson, M. Allen

Parallel Programming

Prentice Hall, ISBN 978-0131405639, 2004, 2nd ed., angol, 496 o.

