Bölüm 9: Paralel Algoritmalar

İleri Algoritmalar

Neden paralellik? | Flynn ve PRAM modelleri | Hızlanma, Amdahl | İş ve derinlik | PRAM ile toplama | Önek toplamı | Boru hattı | Boru hatlı ağaç | Fork/Join, MapReduce

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

© Sercan Külcü. Tüm hakları saklıdır. · Sunumun etkileşimli HTML sürümü:
https://sercankulcu.github.io/files/advanced_algorithms/slides/Bolum_09_Paralel_Algoritmalar.html

GİRİŞ

Öğrenme Çıktıları

Kavramlar

Paralel algoritmanın böl–ata–birleştir yapısını, avantaj ve zorluklarını (iletişim, senkronizasyon, yük dengeleme) açıklamak.

Modeller ve ölçütler

Flynn sınıflarını ve PRAM modelini tanımak; hızlanma, verimlilik, maliyet, iş ve derinliği hesaplamak; Amdahl yasasını uygulamak.

Temel teknikler

PRAM ağaç toplamını, paralel önek toplamını, boru hattını ve boru hatlı ağacı adım adım yürütmek ve adım sayılarını bulmak.

Java'da paralellik

Fork/Join ile paralel birleştirme sıralaması, MapReduce ve veri paralelliği örneklerini okumak; yarış durumlarını fark etmek.

GİRİŞ

Birlikte Daha Hızlı

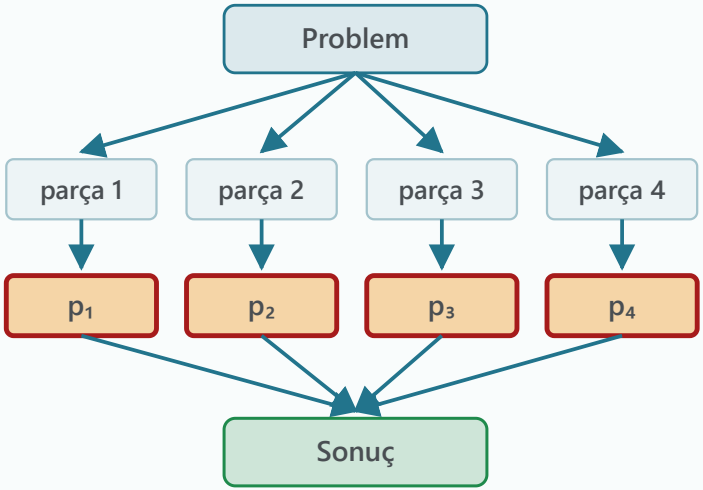
- Bazı işleri bitirmek için **çok fazla zaman** gerekir: büyük veri analizi, bilimsel simülasyon, görüntü işleme.
- **Paralel algoritmalar** (parallel algorithms) bir problemi **aynı anda birden fazla işlemci** (çekirdek) kullanarak çözer.
- Karmaşık görevler çok daha **kısa sürede** tamamlanabilir.
- Tek çekirdeğin saat hızı artık neredeyse artmıyor; performans artışı **çekirdek sayısından** geliyor.

Benzetme: 1000 sınav kâğıdını bir öğretmen 10 saatte okur. 10 öğretmen kâğıtları paylaşırsa ≈ 1 saat sürer; ama kâğıtları **dağıtmak** ve notları **toplamak** da zaman alır.

GİRİŞ

Paralel Algoritmalar

- Bir grup insanın **birlikte çalışmasına** benzer.
- Karmaşık problemi **daha küçük parçalara** ayırır.
- Parçaları aynı anda **farklı işlemcilere atayarak** işi hızlandırır.
- İşlemciler görevlerini bitirince **sonuçlar birleştirilir**.
- **Yüksek başarımlı hesaplamanın** (HPC) temelidir.
- Büyük veri analizi, bilimsel simülasyon, makine öğrenmesi gibi alanlarda kullanılır.



Dezavantajlar ve Zorluklar

Kazanımlar

+ Aynı işi daha kısa sürede bitirmek (hızlanma)
+ Tek makineye sığmayan büyük problemleri çözebilmek (ölçeklenebilirlik)
+ Donanımı (tüm çekirdekleri, GPU'yu) verimli kullanmak

Dezavantajlar

– Problemi parçalara ayırmak ve **birleştirmek** ek çaba gerektirir
– **İletişim** ve **senkronizasyon** ek yük getirir
– **Yük dengeleme**: tüm işlemcilere eşit iş düşmeli, yeterli işlem gücü olmalı
– Hata ayıklama zordur (yarış durumu, kilitlenme)
– Bazı problemler yapısı gereği **sıralıdır**

Veri bağımlılığı varsa (adım k, adım k − 1'in sonucunu bekliyorsa) o kısım paralelleştirilemez: örn. $x_k = f(x_{k-1})$ yinelemesi.

Uygulama Alanları

Matris çarpımı

Bilimsel hesaplamanın temel işlemi; sonuç matrisinin her satırı (hatta her hücresi) bağımsız hesaplanabilir.

Görüntü işleme

Kenar bulma, filtreleme, nesne tanıma: her piksel ya da blok için aynı işlem (veri paralelliği, GPU).

Hesaplamalı akışkanlar dinamiği

Uçak, gemi, araç tasarımında akış simülasyonu: alan ızgaraya bölünür, her bölge bir işlemcide.

Genetik algoritmalar

Popülasyondaki bireylerin uygunluğu birbirinden bağımsız ve paralel değerlendirilir (çizelgeleme, rotalama).

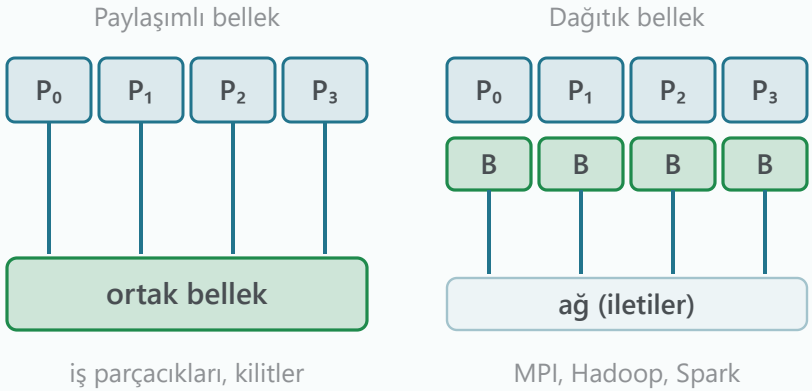
Flynn Sınıflandırması

Sınıf	Açılım	Anlamı	Örnek
SISD	Single Instruction, Single Data	tek komut akışı, tek veri	klasik tek çekirdekli işlemci
SIMD	Single Instruction, Multiple Data	aynı komut, birçok veri üzerinde aynı anda	vektör komutları (AVX), GPU
MISD	Multiple Instruction, Single Data	farklı komutlar aynı veri üzerinde	nadir; hata toleranslı sistemler
MIMD	Multiple Instruction, Multiple Data	her işlemci kendi programını kendi verisinde	çok çekirdekli CPU, kümeler

Veri paralelliği: aynı işlem verinin farklı parçalarına (SIMD tarzı).

Görev paralelliği: farklı görevler aynı anda (MIMD tarzı).

Bellek Mimarileri



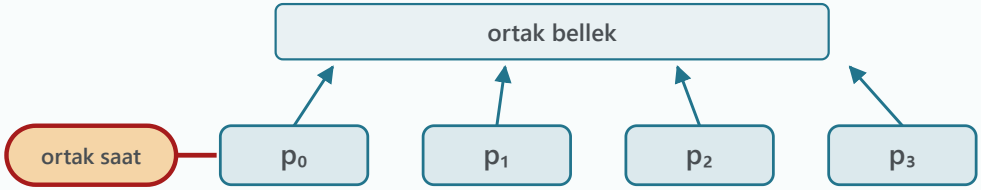
- **Paylaşımlı bellek** (shared memory): tüm işlemciler aynı adres uzayını görür. İletişim ucuzdur; eşzamanlı erişim **senkronizasyon** ister.
- Java iş parçacıkları, `ForkJoinPool`, OpenMP.

- **Dağıtık bellek** (distributed memory): her düğümün kendi belleği vardır; veri **iletiyle** taşınır.
- Ölçeklenir ama iletişim pahalıdır: MPI, Hadoop MapReduce.

PRAM Modeli

- **PRAM** (Parallel Random Access Machine): paralel algoritmaların kuramsal modeli; sıralı RAM modelinin paralel karşılığı.
- p işlemci, ortak bir bellek ve **ortak saat**: işlemciler adımları **eş zamanlı** (lockstep) atar.
- Her adımda bir işlemci: bellekten oku → yerel hesapla → belleğe yaz. Her biri O(1) sürer.
- İletişim ve senkronizasyon maliyeti **yok sayılır**; bu yüzden algoritmanın içsel paralelliğini ölçer.

Tür	Okuma	Yazma
EREW	dışlayıcı	dışlayıcı
CREW	eşzamanlı	dışlayıcı
CRCW	eşzamanlı	eşzamanlı



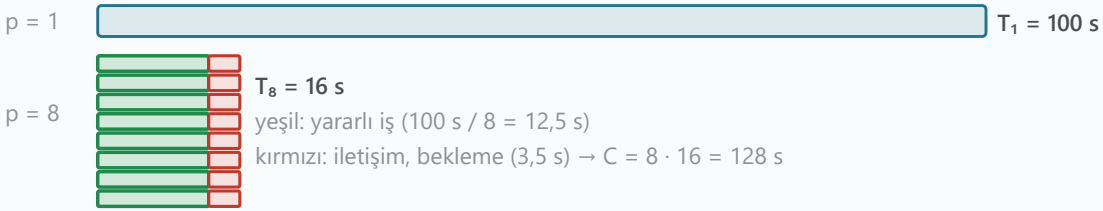
E = Exclusive (aynı hücreye bir adımda tek işlemci), C = Concurrent (birden çok işlemci). CRCW'de çakışan yazmalar bir kuralla çözülür (ortak değer, rastgele biri, öncelikli olan…).

Hızlanma, Verimlilik, Maliyet

Ölçüt	Tanım
T_1	en iyi sıralı algoritmanın süresi
T_p	p işlemcideki paralel süre
Hızlanma (speedup)	$S = T_1 / T_p$ (ideal: $S = p$)
Verimlilik (efficiency)	$E = S / p$ $(0 < E \leq 1)$
Maliyet (cost)	$C = p \cdot T_p$

Örnek: $T_1 = 100$ s, $p = 8$, $T_8 = 16$ s
$S = 100 / 16 =$ **6,25**
$E = 6{,}25 / 8 \approx$ **0,78**
$C = 8 \cdot 16 = 128$ s $(> 100$ s$)$

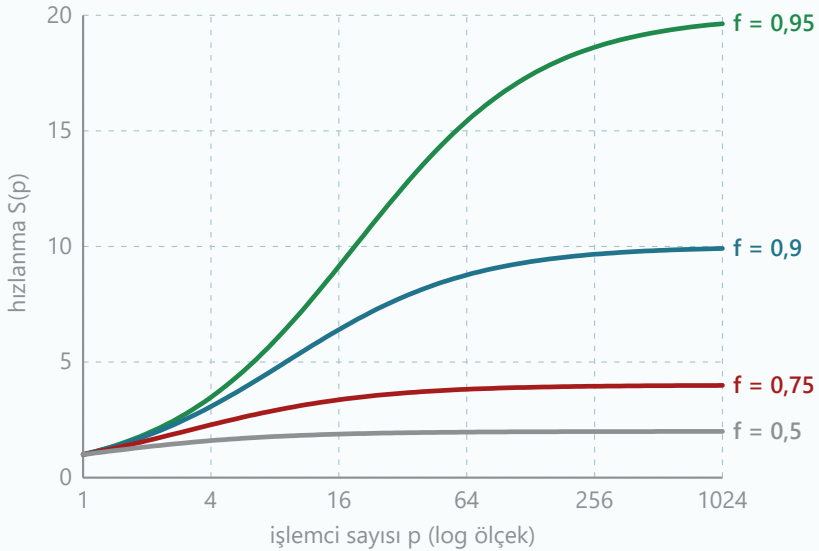
- **Maliyet-optimal**: $C = \Theta(T_1)$.
- $S < p$ olmasının nedenleri: iletişim, bekleme, yük dengesizliği, sıralı kısımlar.



Amdahl Yasası

- Programın f kesri paralelleştirilebilir, 1 − f kesri **sıralıdır**.
- S(p) = 1 / ((1 − f) + f / p)
- p → ∞ iken S → **1 / (1 − f)**: sıralı kısım hızlanmayı sınırlar.
- f = 0,9, p = 8: S = 1 / (0,1 + 0,1125) ≈ **4,71**; sınır 10.

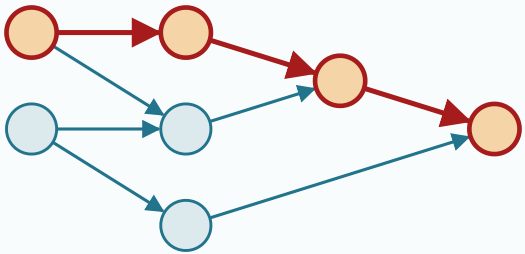
Gustafson yasası: problem boyutu p ile büyürse S = (1 − f) + f · p. f = 0,9, p = 8 → S = 7,3. Büyük problemlerde paralellik daha kârlıdır.



İş ve Derinlik

- **İş** (work) W: tüm işlemlerin toplam sayısı = T_1 (tek işlemcide süre).
- **Derinlik** (depth, span) D: en uzun bağımlılık zinciri = $T\infty$ (sınırsız işlemcide süre).
- **Paralellik** = W / D: yararlı kullanılabilecek en fazla işlemci sayısı.
- Alt sınırlar: $T_p \geq W / p$ ve $T_p \geq D$.
- **Brent teoremi**: $T_p \leq W / p + D$.
- **İş-verimli** (work-efficient): W, en iyi sıralı algoritmayla aynı mertebede.

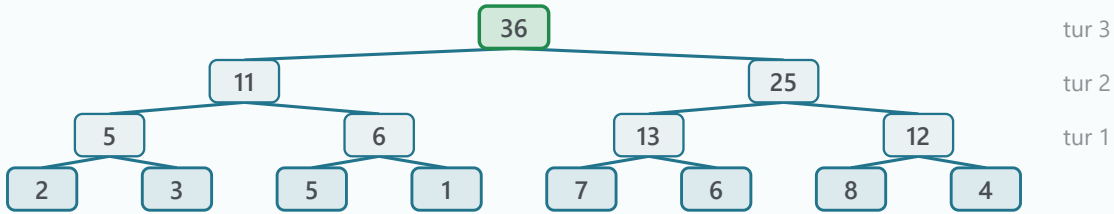
Örnek: n sayının ağaçla toplanması
W = n − 1 toplama, D = $\log_2$ n tur.
n = 1024: W = 1023, D = 10, paralellik ≈ 102.
p = 32 için Brent: $T_{32} \leq 1023/32 + 10 \approx 42$ adım.



W = 7 işlem (düğüm), D = 4 (kırmızı yol)
paralellik W / D = 1,75

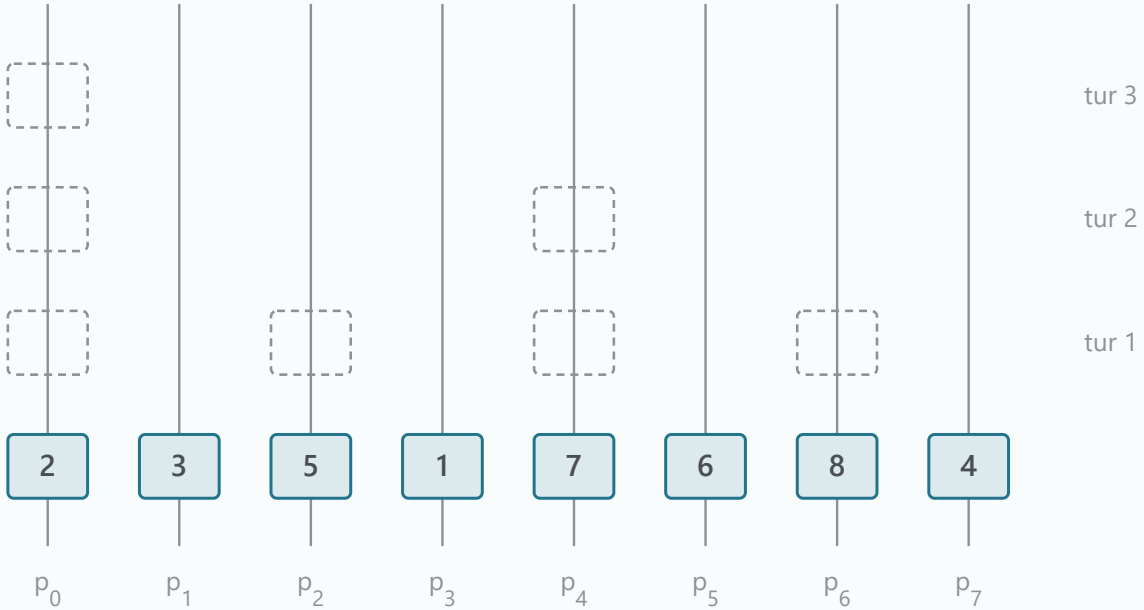
Problem: Paralel Toplama

- Girdi: n sayı; çıktı: toplamları. Sıralı algoritma n − 1 toplama yapar: **O(n)**.
- Sıralı döngüde her toplama bir öncekini bekler: zincir uzunluğu n − 1, paralellik yok.
- Toplama **birleşmeli** (associative) olduğu için parantezleri değiştirebiliriz: ((2 + 3) + (5 + 1)) + ((7 + 6) + (8 + 4)) → dengeli bir **ikili ağaç**.
- Ağacın her düzeyindeki toplamlar birbirinden bağımsızdır → aynı turda paralel yapılır.
- Aynı fikir her birleşmeli işlem için çalışır: **max, min, çarpım, AND, OR** (indirgeme, reduction).



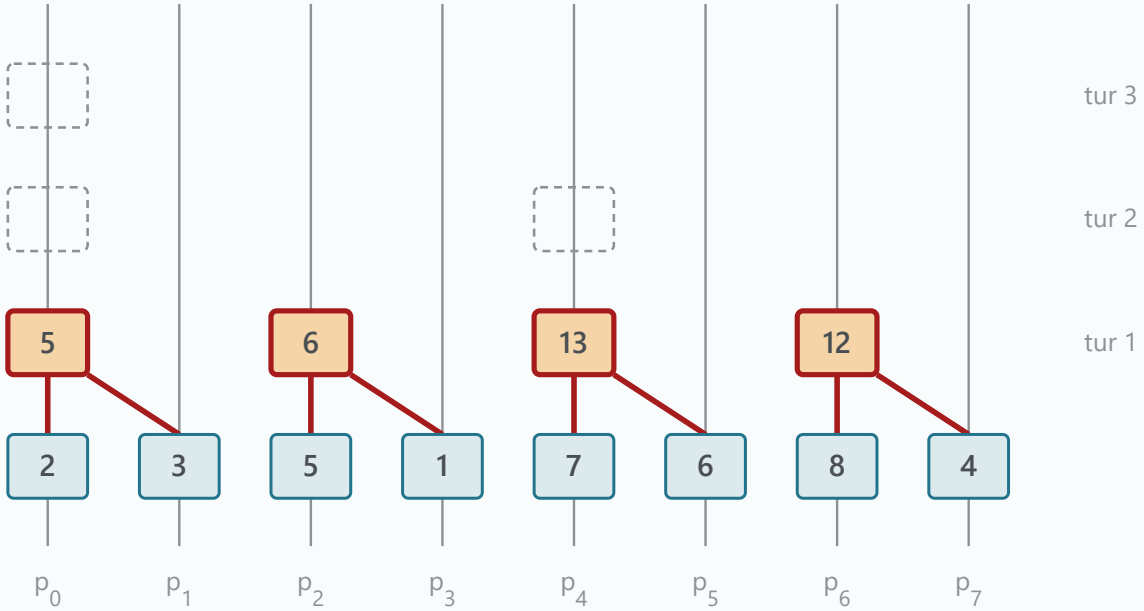
Örnek dizi: 2, 3, 5, 1, 7, 6, 8, 4 → toplam 36. 8 işlemci p_0 … p_7 birer sayı tutar.

PRAM Toplama Adım Adım



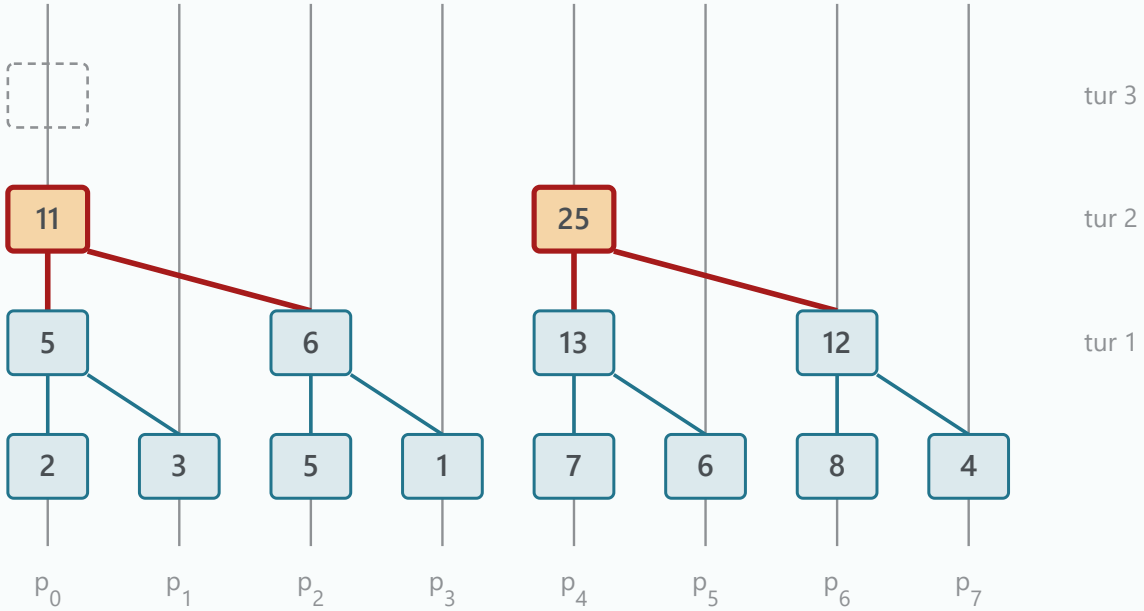
Başlangıç: 8 sayı 8 işlemciye (p_0 … p_7) birer birer dağıtılır. Kesikli kutular, sonraki turlarda yazılacak hücrelerdir.

PRAM Toplama Adım Adım



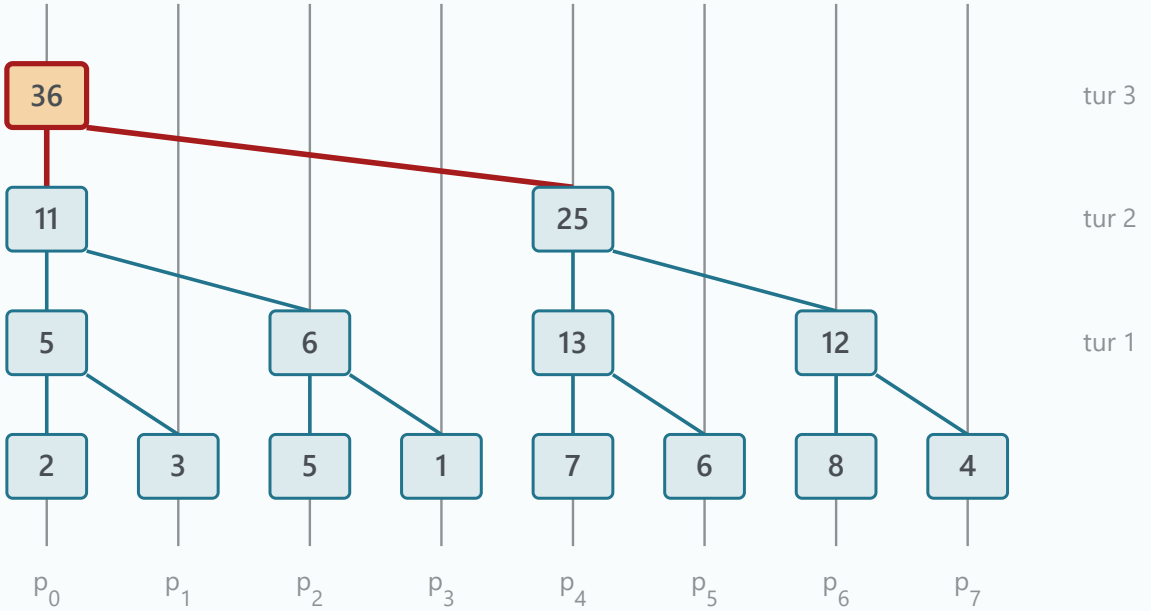
Tur 1: 4 işlemci *aynı anda* komşu çiftleri toplar: 2+3=5, 5+1=6, 7+6=13, 8+4=12.

PRAM Toplama Adım Adım



Tur 2: 2 işlemci, aradaki uzaklık 2 olan kısmi toplamları birleştirir: 5+6=11, 13+12=25.

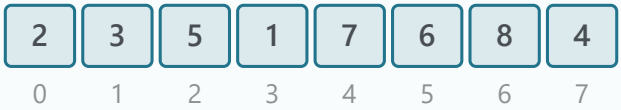
PRAM Toplama Adım Adım



Tur 3: p_0, 11+25 = **36** sonucunu bulur. 8 sayı $\log_2 8$ = **3 turda** toplandı (sıralı: 7 adım).

Ağaç Toplamı: Kodu Adım Adım İzle

```
int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
int n = a.length;
for (int adim = 1; adim < n; adim *= 2) {
  // aynı turdaki tüm i'ler paralel
  for (int i = 0; i + adim < n; i += 2 * adim)
    a[i] = a[i] + a[i + adim];
}  // tur sonu: senkronizasyon
// toplam a[0]'da
```



Dizi oluşturulur; `n = 8`. Her hücreyi bir işlemci tutar.

Adım 1 / 12

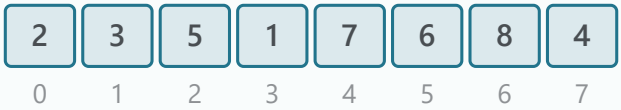
DEĞİŞKENLER

`n = 8` `adim = –` `i = –`

`a = [2, 3, 5, 1, 7, 6, 8, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
▶ for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
5   for (int i = 0; i + adim < n; i += 2 *
  adim)
6     a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



Yeni tur: `adim = 1`. Bu turda 4 toplama **paralel** yapılır.

Adım 2 / 12

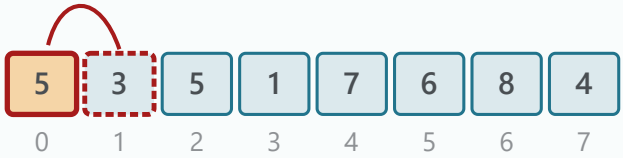
DEĞİŞKENLER

`n = 8` `adim = 1` `i = –`

`a = [2, 3, 5, 1, 7, 6, 8, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
    for (int i = 0; i + adim < n; i += 2 *
  adim)
      a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



`a[0] = a[0] + a[1]` = 2 + 3 = **5**.

Adım 3 / 12

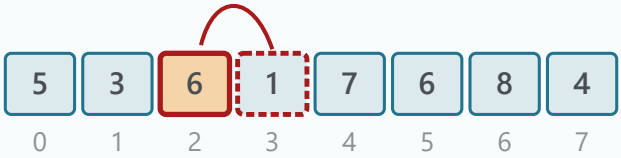
DEĞİŞKENLER

`n = 8` `adim = 1` `i = 0`

`a = [5, 3, 5, 1, 7, 6, 8, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
▶   for (int i = 0; i + adim < n; i += 2 *
    adim)
▶     a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



`a[2] = a[2] + a[3]` = 5 + 1 = **6**.

Adım 4 / 12

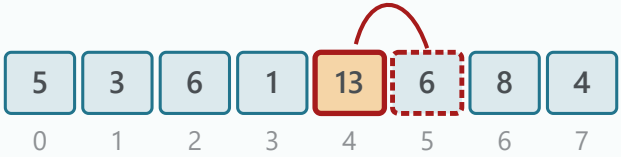
DEĞİŞKENLER

`n = 8` `adim = 1` `i = 2`

`a = [5, 3, 6, 1, 7, 6, 8, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
    for (int i = 0; i + adim < n; i += 2 *
  adim)
      a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



`a[4] = a[4] + a[5]` = 7 + 6 = **13**.

Adım 5 / 12

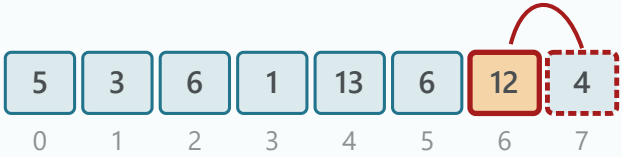
DEĞİŞKENLER

`n = 8` `adim = 1` `i = 4`

`a = [5, 3, 6, 1, 13, 6, 8, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
    for (int i = 0; i + adim < n; i += 2 *
  adim)
      a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



`a[6] = a[6] + a[7]` = 8 + 4 = **12**.

Adım 6 / 12

DEĞİŞKENLER

`n = 8` `adim = 1` `i = 6`

`a = [5, 3, 6, 1, 13, 6, 12, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
▶ for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
5   for (int i = 0; i + adim < n; i += 2 *
    adim)
6     a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```

5	3	6	1	13	6	12	4
0	1	2	3	4	5	6	7

Yeni tur: `adim = 2`. Bu turda 2 toplama **paralel** yapılır.

Adım 7 / 12

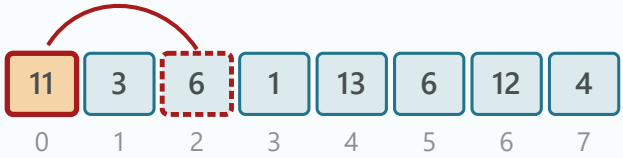
DEĞİŞKENLER

`n = 8` `adim = 2` `i = –`

`a = [5, 3, 6, 1, 13, 6, 12, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
    for (int i = 0; i + adim < n; i += 2 *
adim)
      a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



`a[0] = a[0] + a[2]` = 5 + 6 = **11**.

Adım 8 / 12

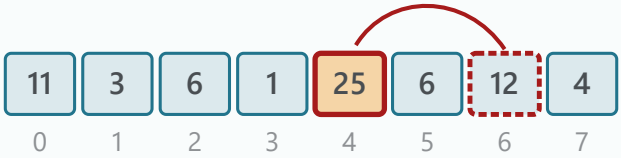
DEĞİŞKENLER

`n = 8` `adim = 2` `i = 0`

`a = [11, 3, 6, 1, 13, 6, 12, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
▶   for (int i = 0; i + adim < n; i += 2 *
    adim)
▶     a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



`a[4] = a[4] + a[6]` = 13 + 12 = **25**.

Adım 9 / 12

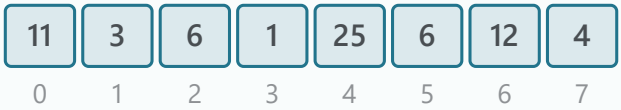
DEĞİŞKENLER

`n = 8` `adim = 2` `i = 4`

`a = [11, 3, 6, 1, 25, 6, 12, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
▶ for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
5   for (int i = 0; i + adim < n; i += 2 *
    adim)
6     a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



Yeni tur: `adim = 4`. Bu turda 1 toplama **paralel** yapılır.

Adım 10 / 12

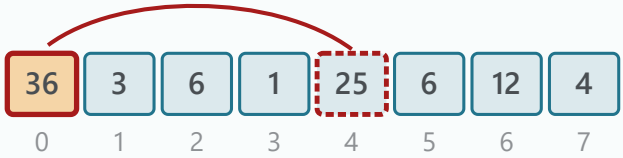
DEĞİŞKENLER

`n = 8` `adim = 4` `i = –`

`a = [11, 3, 6, 1, 25, 6, 12, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
    for (int i = 0; i + adim < n; i += 2 *
  adim)
      a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
8 // toplam a[0]'da
```



`a[0] = a[0] + a[4]` = 11 + 25 = **36**.

Adım 11 / 12

DEĞİŞKENLER

`n = 8` `adim = 4` `i = 0`

`a = [36, 3, 6, 1, 25, 6, 12, 4]`

Ağaç Toplamı: Kodu Adım Adım İzle

```
1 int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
2 int n = a.length;
3 for (int adim = 1; adim < n; adim *= 2) {
4   // aynı turdaki tüm i'ler paralel
5   for (int i = 0; i + adim < n; i += 2 *
  adim)
6     a[i] = a[i] + a[i + adim];
7 }  // tur sonu: senkronizasyon
▶ // toplam a[0]'da
```

36	3	6	1	25	6	12	4
0	1	2	3	4	5	6	7

`adim = 8` → döngü biter. Toplam `a[0] = 36`; 3 tur, 7 toplama.

Adım 12 / 12

DEĞİŞKENLER

`n = 8` `adim = 8` `i = –`

`a = [36, 3, 6, 1, 25, 6, 12, 4]`

PRAM Toplama: Analiz

Ölçüt	n işlemci (n/2 yeterli)	$p = n / \log_2 n$ işlemci
Süre T_p	$\log_2 n$	O(log n)
İş W	n − 1	n − 1
Maliyet $p \cdot T_p$	Θ(n log n)	Θ(n)
Maliyet-optimal mi?	hayır	**evet**

- Turlar ilerledikçe çalışan işlemci sayısı yarıya iner (4, 2, 1): çoğu işlemci boşta bekler, maliyet yüksektir.
- Çözüm: her işlemci önce kendi $\log_2 n$'lik bloğunu **sıralı** toplar (O(log n)), sonra $n / \log_2 n$ kısmi toplam ağaçla birleştirilir (O(log n)).
- EREW yeterlidir: bir turda her hücreyi tek işlemci okur/yazar. Her tur sonunda bir **senkronizasyon** (bariyer) gerekir.

Paralel Önek Toplamı

- **Önek toplamı** (prefix sum, scan): $y_i = x_0 + x_1 + \ldots + x_i$.
- Sıralı: $y_i = y_{i-1} + x_i$, O(n) ama tamamen zincirleme.
- **Hillis–Steele**: d = 1, 2, 4, … için her i ≥ d paralel olarak x[i] += x[i − d].
- $D = \lceil \log_2 n \rceil$ tur, W = O(n log n) → iş-verimli değil.
- **Blelloch** algoritması (yukarı/aşağı süpürme) W = O(n), D = O(log n).
- Kullanım: paralel sıralama, sıkıştırma (filter), Java'da `Arrays.parallelPrefix`.

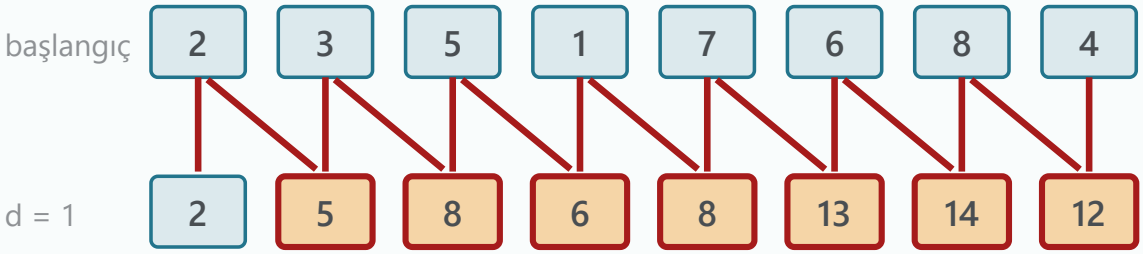
Toplamadaki ağaç yalnızca son hücreyi (toplamı) verir; önek toplamında **her** hücre kendi önekini öğrenmelidir.

Hillis–Steele Önek Toplamı Adım Adım



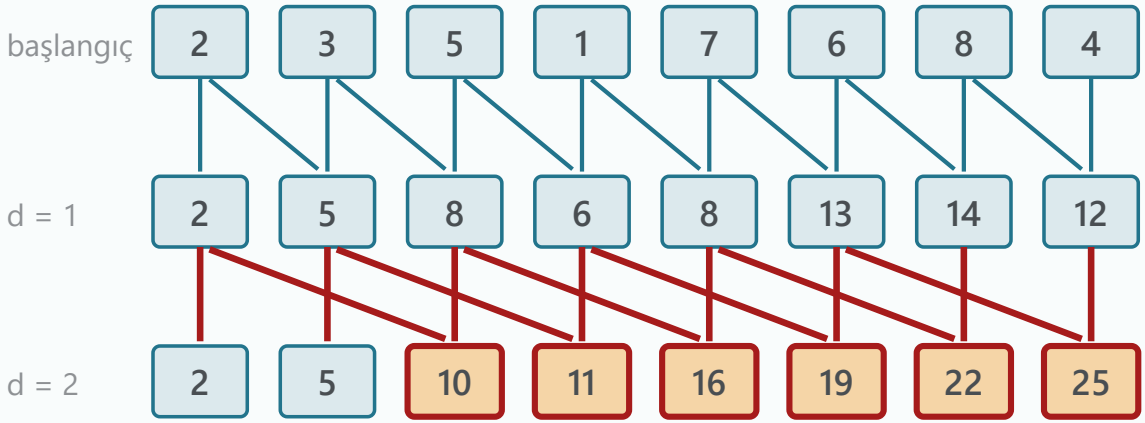
Girdi: 2 3 5 1 7 6 8 4. Amaç: her i için $x_0 + \ldots + x_i$ (kapsayıcı önek toplamı).

Hillis–Steele Önek Toplamı Adım Adım



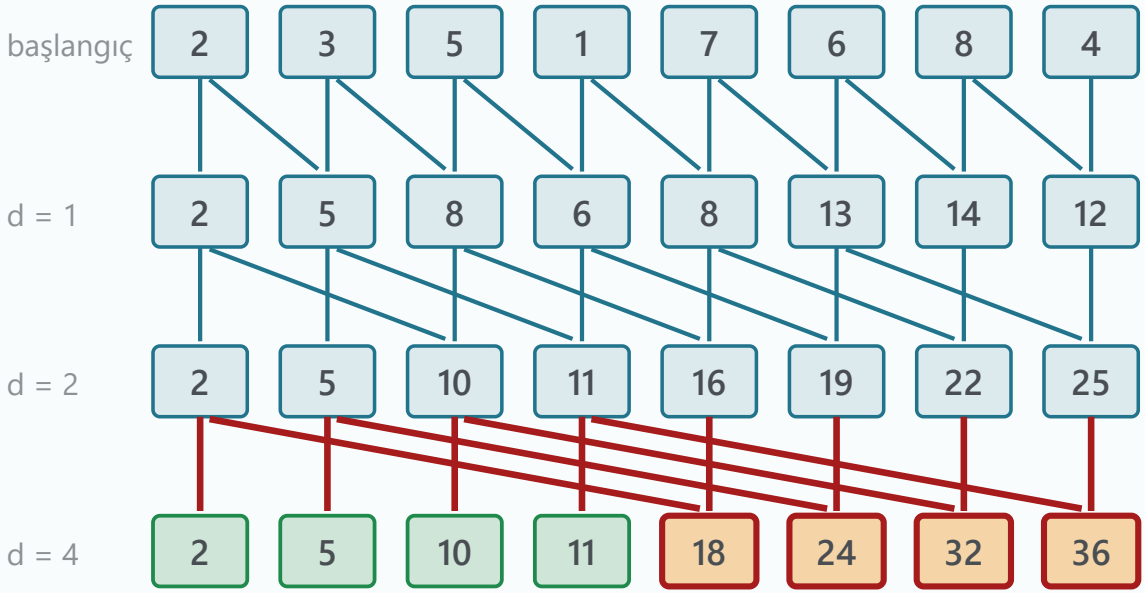
d = 1: i ≥ 1 olan her işlemci kendi değerine solundakini ekler (7 toplama, aynı anda).

Hillis–Steele Önek Toplamı Adım Adım



d = 2: i ≥ 2 için 2 konum soldakini ekler. Artık her hücre 4 öğelik pencerenin toplamıdır.

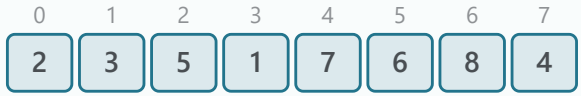
Hillis–Steele Önek Toplamı Adım Adım



d = 4: sonuç 2 5 10 11 18 24 32 36. $\log_2 8 = 3$ turda bitti; ama toplam 7+6+4 = 17 toplama yapıldı.

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
▶ // yukarı süpürme; x = 2 3 5 1 7 6 8 4
2 for (int d = 1; d < n; d *= 2)
3   for (int i = 2*d - 1; i < n; i += 2*d)
4     x[i] += x[i - d];
5 x[n - 1] = 0;
6 // aşağı süpürme (iç döngüler paralel)
7 for (int d = n / 2; d >= 1; d /= 2)
8   for (int i = 2*d - 1; i < n; i += 2*d) {
9     int t = x[i - d];
10    x[i - d] = x[i];
11    x[i] += t;
12  }
13 // x: dışlayıcı önek toplamları
```



girdi

Girdi 2 3 5 1 7 6 8 4 (n = 8). Dizi, örtük bir ikili ağaç gibi kullanılacak: önce yukarı (indirgeme), sonra aşağı.

Adım 1 / 9

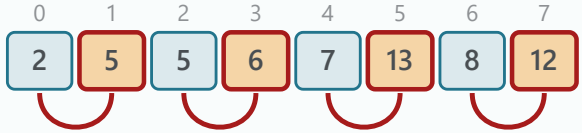
DEĞİŞKENLER

d = –

x = [2, 3, 5, 1, 7, 6, 8, 4]

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
▶  for (int d = 1; d < n; d *= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d)
▶      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: dışlayıcı önek toplamları
```



yukarı süpürme, d = 1

d = 1: x[1] += x[0], x[3] += x[2], x[5] += x[4], x[7] += x[6] (aynı turda paralel, 4 toplama).

Adım 2 / 9

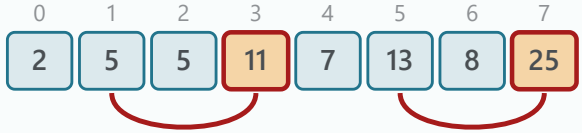
DEĞİŞKENLER

d = 1

x = [2, 5, 5, 6, 7, 13, 8, 12]

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
▶  for (int d = 1; d < n; d *= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d)
▶      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: dışlayıcı önek toplamları
```



yukarı süpürme, d = 2

d = 2: x[3] += x[1], x[7] += x[5] (aynı turda paralel, 2 toplama).

Adım 3 / 9

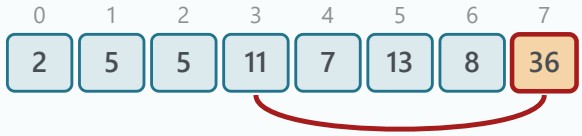
DEĞİŞKENLER

d = **2**

x = **[2, 5, 5, 11, 7, 13, 8, 25]**

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
▶  for (int d = 1; d < n; d *= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d)
▶      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: dışlayıcı önek toplamları
```



yukarı süpürme, d = 4

d = 4: x[7] += x[3] (aynı turda paralel, 1 toplama). Kök x[7] = 36: toplam.

Adım 4 / 9

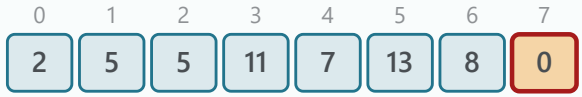
DEĞİŞKENLER

d = 4

x = [2, 5, 5, 11, 7, 13, 8, 36]

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
▶  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
13 // x: dışlayıcı önek toplamları
```



kök sıfırlandı

Toplam 36 köktedir. Kök 0 yapılır: aşağı süpürmede her düğüm "solundakilerin toplamını" taşıyacak.

Adım 5 / 9

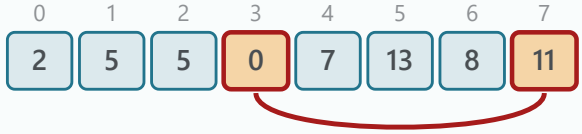
DEĞİŞKENLER

`toplam = 36`

`x = [2, 5, 5, 11, 7, 13, 8, 0]`

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
▶  for (int d = n / 2; d >= 1; d /= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d) {
▶      int t = x[i - d];
▶      x[i - d] = x[i];
▶      x[i] += t;
12   }
13 // x: dışlayıcı önek toplamları
```



aşağı süpürme, d = 4

d = 4: her (i − d, i) çiftinde sol çocuk ebeveynin değerini alır, sağ çocuk ebeveyn + eski sol değerini alır.

Adım 6 / 9

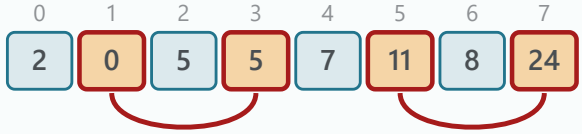
DEĞİŞKENLER

d = 4

x = [2, 5, 5, 0, 7, 13, 8, 11]

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
▶  for (int d = n / 2; d >= 1; d /= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d) {
▶      int t = x[i - d];
▶      x[i - d] = x[i];
▶      x[i] += t;
12   }
13 // x: dışlayıcı önek toplamları
```



aşağı süpürme, d = 2

d = 2: her (i − d, i) çiftinde sol çocuk ebeveynin değerini alır, sağ çocuk ebeveyn + eski sol değerini alır.

Adım 7 / 9

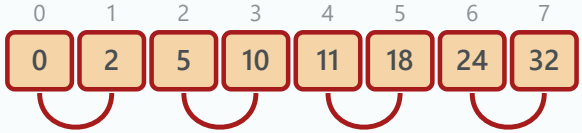
DEĞİŞKENLER

d = **2**

x = **[2, 0, 5, 5, 7, 11, 8, 24]**

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
▶  for (int d = n / 2; d >= 1; d /= 2)
▶    for (int i = 2*d - 1; i < n; i += 2*d) {
▶      int t = x[i - d];
▶      x[i - d] = x[i];
▶      x[i] += t;
12   }
13 // x: dışlayıcı önek toplamları
```



aşağı süpürme, d = 1

d = 1: son tur; 4 çift aynı anda güncellenir.

Adım 8 / 9

DEĞİŞKENLER

d = 1

x = [0, 2, 5, 10, 11, 18, 24, 32]

Blelloch Önek Toplamı: Kodu Adım Adım İzle

```
1  // yukarı süpürme; x = 2 3 5 1 7 6 8 4
2  for (int d = 1; d < n; d *= 2)
3    for (int i = 2*d - 1; i < n; i += 2*d)
4      x[i] += x[i - d];
5  x[n - 1] = 0;
6  // aşağı süpürme (iç döngüler paralel)
7  for (int d = n / 2; d >= 1; d /= 2)
8    for (int i = 2*d - 1; i < n; i += 2*d) {
9      int t = x[i - d];
10     x[i - d] = x[i];
11     x[i] += t;
12   }
▶ // x: dışlayıcı önek toplamları
```



sonuç (dışlayıcı önek)

Sonuç 0 2 5 10 11 18 24 32: dışlayıcı önek toplamı. $2 \cdot (n - 1) = 14$ işlem, iş O(n), derinlik $2 \log_2 n = 6$ tur.

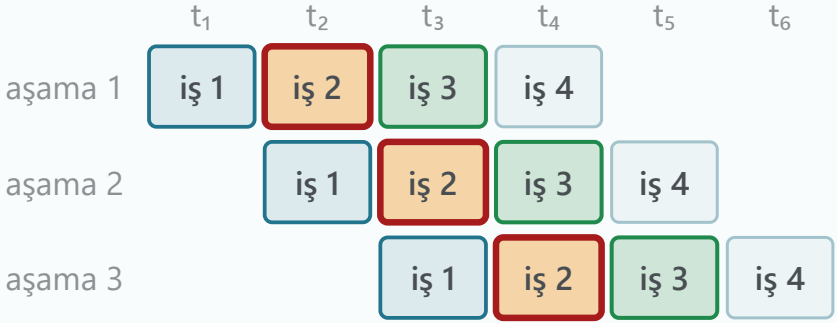
Adım 9 / 9

DEĞİŞKENLER

x = [0, 2, 5, 10, 11, 18, 24, 32]

Boru Hattı Fikri

- **Boru hattı** (pipeline): iş, ardışık **aşamalara** bölünür; her aşama bir işlemcidir.
- Fabrika montaj hattı gibi: bir iş 2. aşamaya geçerken 1. aşama bir sonraki işe başlar.
- Tek bir işin süresi kısalmaz; ama **birim zamanda biten iş** (throughput) artar.
- k aşama, m iş: sıralı k · m adım; boru hattıyla **k + m − 1** adım.
- Başta hat dolarken ve sonda boşalırken bazı aşamalar boştur.

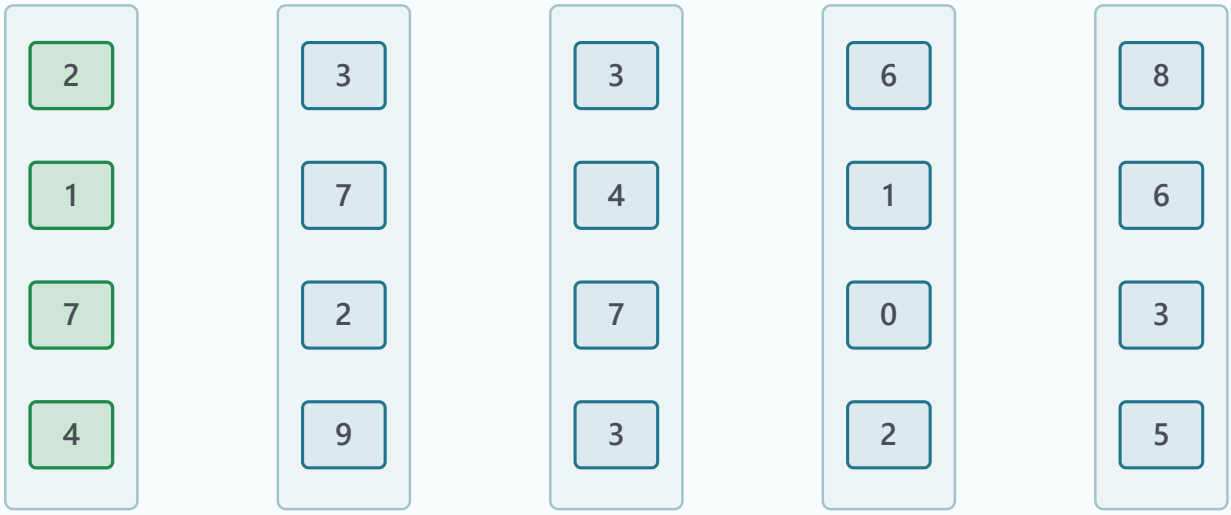


3 aşama, 4 iş: 3 · 4 = 12 yerine 3 + 4 − 1 = 6 adım.

Boru Hattı Problemi

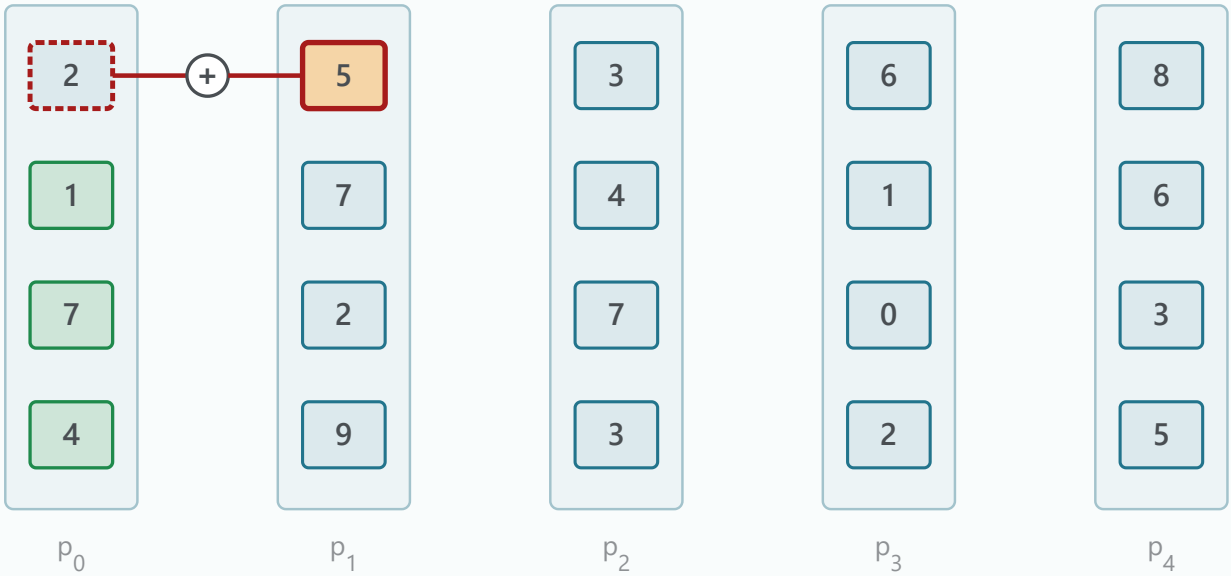
- 5 işlemci p_0 … p_4 doğrusal bir zincir oluşturur; her işlemci yalnızca **sağ komşusuna** ileti gönderebilir.
- Her p_j, 4 değerlik bir sütun tutar (satır 0 … 3).
- Amaç: her satır r için p_j'deki değer, p_0 … p_j'nin o satırdaki değerlerinin **toplamı** olsun (satır bazında önek toplamı). Böylece p_4 her satırın toplamını tutar.
- Bir adımda her işlemci **bir** değer gönderir; alıcı onu kendi değerine ekler.
- Saf yöntem: önce satır 0 zincir boyunca taşınır (4 adım), sonra satır 1… → $4 \cdot 4 =$ **16 adım**.
- Boru hattı: p_j, satır r'yi **t = r + j** anında gönderir; satırlar zincirde arka arkaya akar.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



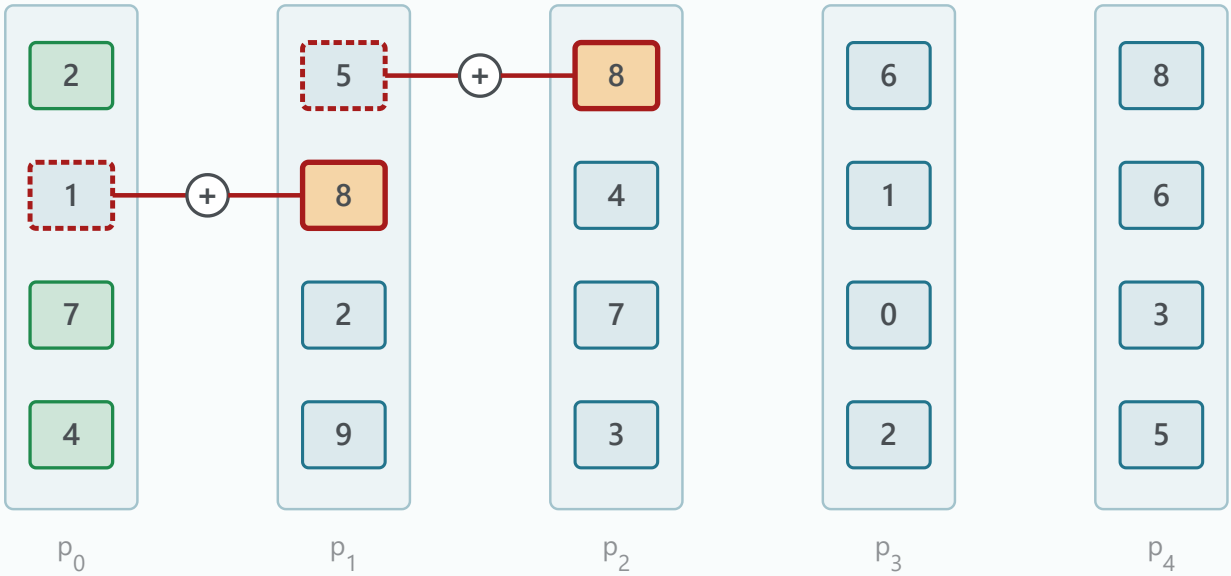
Her p_j 4 satırlık bir sütun tutar. Hedef: her satırda p_j, p_0 … p_j değerlerinin toplamını tutsun. p_0'ın değerleri zaten sonuçtur.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



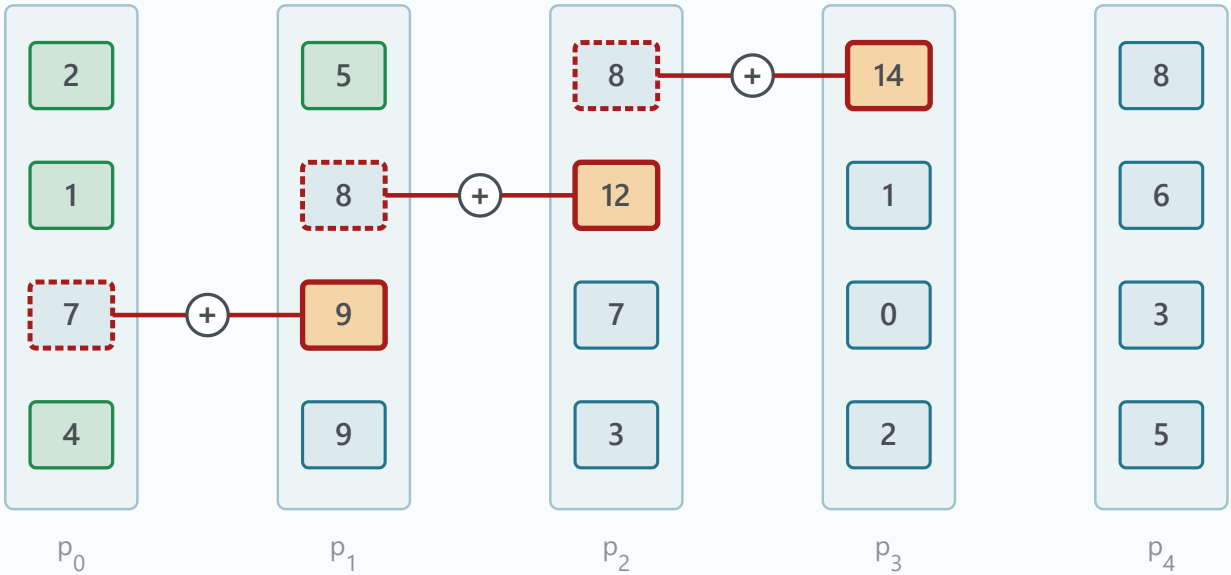
Adım 1: $p_0 \rightarrow p_1$ satır 0: 2+3=5.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



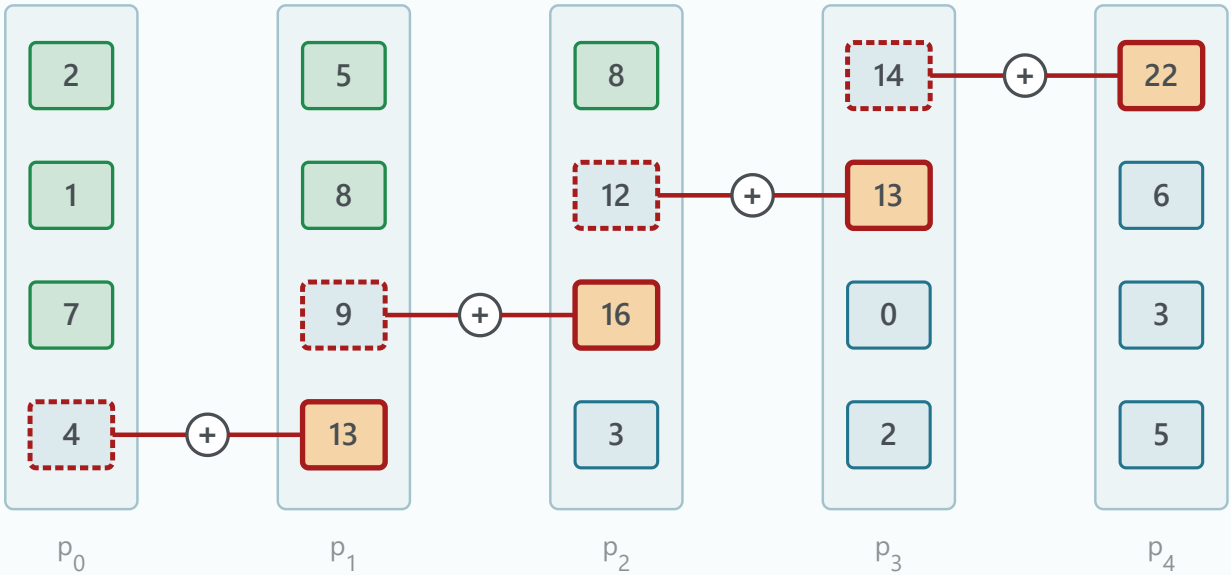
Adım 2: $p_0 \rightarrow p_1$ satır 1: 1+7=8, $p_1 \rightarrow p_2$ satır 0: 5+3=8.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



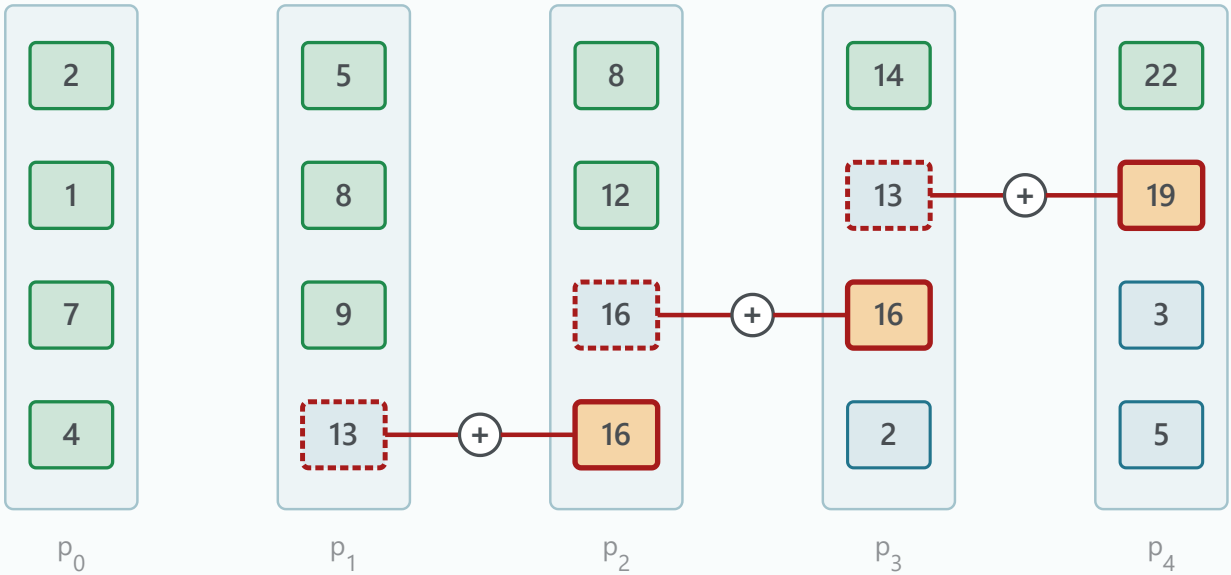
Adım 3: $p_0 \rightarrow p_1$ satır 2: 7+2=9, $p_1 \rightarrow p_2$ satır 1: 8+4=12, $p_2 \rightarrow p_3$ satır 0: 8+6=14.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



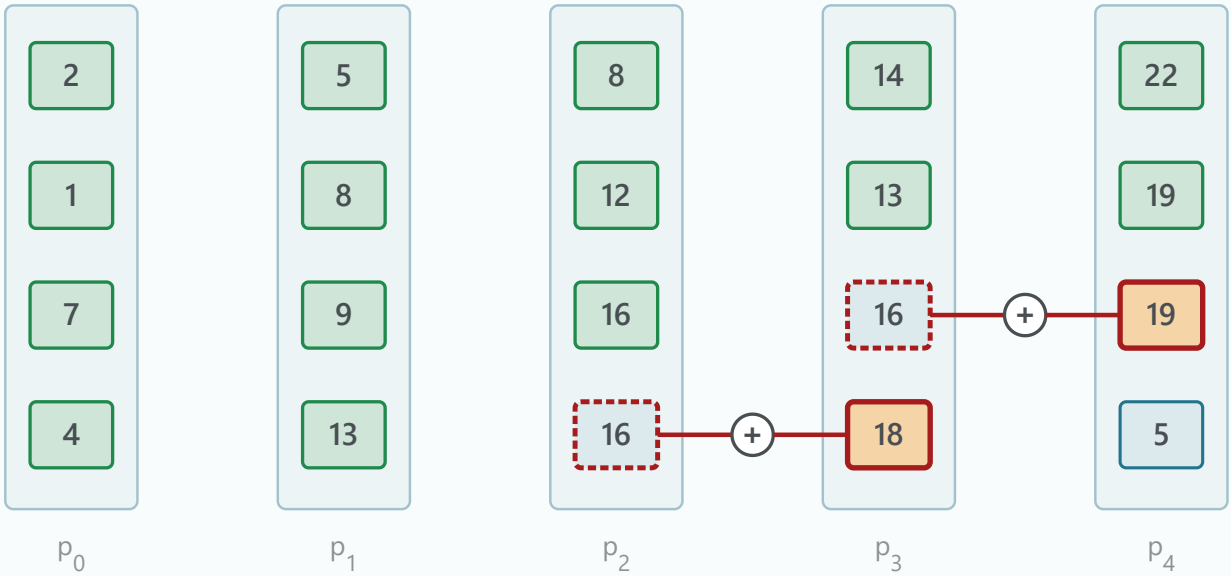
Adım 4: $p_0 \to p_1$ satır 3: 4+9=13, $p_1 \to p_2$ satır 2: 9+7=16, $p_2 \to p_3$ satır 1: 12+1=13, $p_3 \to p_4$ satır 0: 14+8=22.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



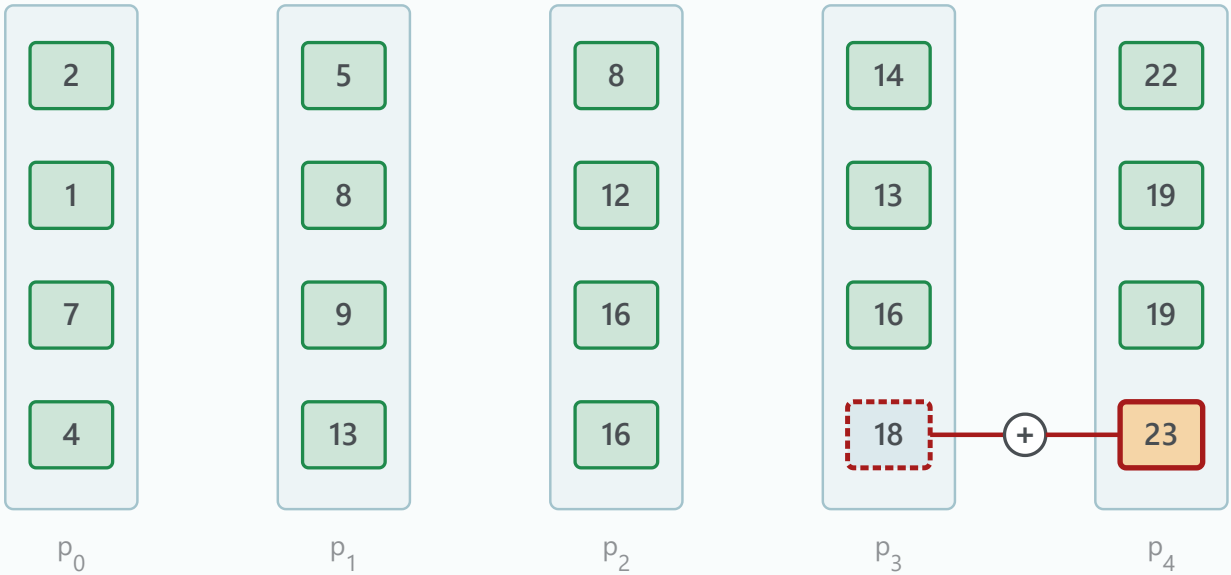
Adım 5: $p_1 \rightarrow p_2$ satır 3: 13+3=16, $p_2 \rightarrow p_3$ satır 2: 16+0=16, $p_3 \rightarrow p_4$ satır 1: 13+6=19.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



Adım 6: $p_2 \rightarrow p_3$ satır 3: 16+2=18, $p_3 \rightarrow p_4$ satır 2: 16+3=19.

Boru Hattı Adım Adım: 5 İşlemci, 4 Satır



Adım 7: $p_3 \rightarrow p_4$ satır 3: 18+5=23. Bitti: **7 adım**; p_4 satır toplamlarını (22 19 19 23) tutar.

Boru Hattı: Analiz

adım	$p_0 \to p_1$	$p_1 \to p_2$	$p_2 \to p_3$	$p_3 \to p_4$
1	satır 0	—	—	—
2	satır 1	satır 0	—	—
3	satır 2	satır 1	satır 0	—
4	satır 3	satır 2	satır 1	satır 0
5	—	satır 3	satır 2	satır 1
6	—	—	satır 3	satır 2
7	—	—	—	satır 3

```
for (int t = 0; t < m + p - 2; t++)
  // tüm j'ler aynı anda
  for (int j = 0; j < p - 1; j++) {
    int r = t - j;
    if (r >= 0 && r < m)
      x[j + 1][r] += x[j][r];
  }
```

- Adım sayısı: **m + p − 2** = 4 + 5 − 2 = 7 (saf: m(p − 1) = 16).
- Verimlilik: m(p − 1) toplama / ((p − 1)(m + p − 2)) = m / (m + p − 2) = 4/7 ≈ %57.
- m ≫ p iken verimlilik → 1.

Boru Hattı: Kodu Adım Adım İzle

```
▶ for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
3   for (int j = 0; j < p - 1; j++) {
4     int r = t - j;
5     if (r >= 0 && r < m)
6       x[j + 1][r] += x[j][r];
7   }
```

	p_0	p_1	p_2	p_3	p_4
0	2	3	3	6	8
1	1	7	4	1	6
2	7	2	7	0	3
3	4	9	3	2	5

p = 5 işlemci, m = 4 satır. Her sütun bir işlemcinin değerleri; satır r, p_j'ye t = r + j anında ulaşır.

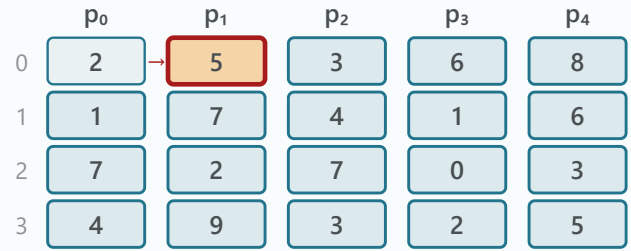
Adım 1 / 9

DEĞİŞKENLER

t = –

Boru Hattı: Kodu Adım Adım İzle

```
1 for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 0: $p_0 \rightarrow p_1$ satır 0: 2 + 3 = 5.

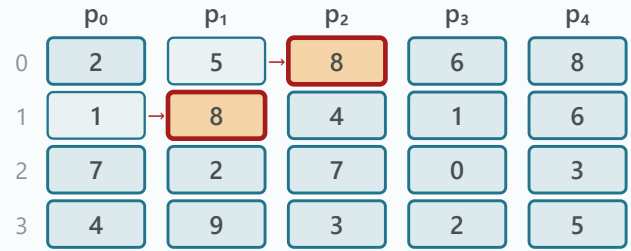
Adım 2 / 9

DEĞİŞKENLER

`t = 0` `p₄ = [8, 6, 3, 5]`

Boru Hattı: Kodu Adım Adım İzle

```
1 for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 1: $p_0 \rightarrow p_1$ satır 1: 1 + 7 = 8; $p_1 \rightarrow p_2$ satır 0: 5 + 3 = 8.

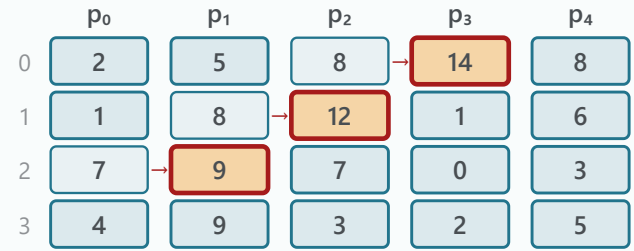
Adım 3 / 9

DEĞİŞKENLER

`t = 1` `p4 = [8, 6, 3, 5]`

Boru Hattı: Kodu Adım Adım İzle

```
1 for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 2: $p_0 \rightarrow p_1$ satır 2: 7 + 2 = 9; $p_1 \rightarrow p_2$ satır 1: 8 + 4 = 12; $p_2 \rightarrow p_3$ satır 0: 8 + 6 = 14.

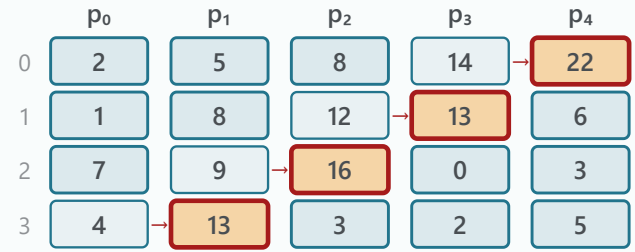
Adım 4 / 9

DEĞİŞKENLER

`t = 2` `p₄ = [8, 6, 3, 5]`

Boru Hattı: Kodu Adım Adım İzle

```
1 for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 3: $p_0 \rightarrow p_1$ satır 3: 4 + 9 = 13; $p_1 \rightarrow p_2$ satır 2: 9 + 7 = 16; $p_2 \rightarrow p_3$ satır 1: 12 + 1 = 13; $p_3 \rightarrow p_4$ satır 0: 14 + 8 = 22. Hat dolu: 4 bağlantı birden çalışıyor.

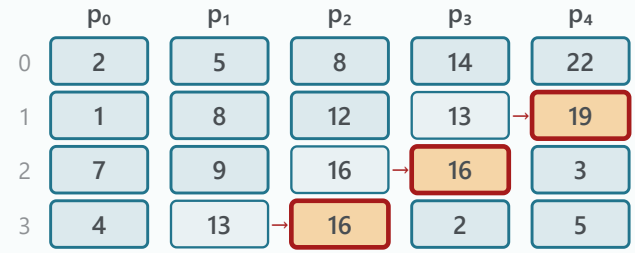
Adım 5 / 9

DEĞİŞKENLER

`t = 3` `p₄ = [22, 6, 3, 5]`

Boru Hattı: Kodu Adım Adım İzle

```
1 for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 4: $p_1 \rightarrow p_2$ satır 3: 13 + 3 = 16; $p_2 \rightarrow p_3$ satır 2: 16 + 0 = 16; $p_3 \rightarrow p_4$ satır 1: 13 + 6 = 19.

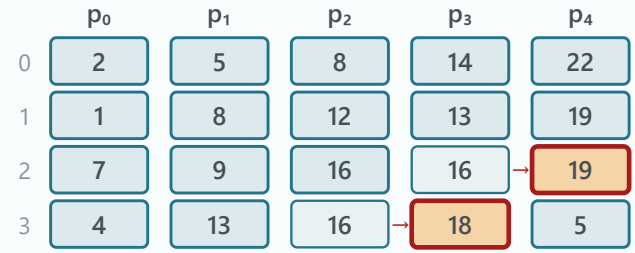
Adım 6 / 9

DEĞİŞKENLER

`t = 4` `p₄ = [22, 19, 3, 5]`

Boru Hattı: Kodu Adım Adım İzle

```
1 for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```



t = 5: $p_2 \rightarrow p_3$ satır 3: 16 + 2 = 18; $p_3 \rightarrow p_4$ satır 2: 16 + 3 = 19.

Adım 7 / 9

DEĞİŞKENLER

`t = 5` `p₄ = [22, 19, 19, 5]`

Boru Hattı: Kodu Adım Adım İzle

```
1 for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
    for (int j = 0; j < p - 1; j++) {
      int r = t - j;
      if (r >= 0 && r < m)
        x[j + 1][r] += x[j][r];
7   }
```

	p_0	p_1	p_2	p_3	p_4
0	2	5	8	14	22
1	1	8	12	13	19
2	7	9	16	16	19
3	4	13	16	18	→ 23

t = 6: $p_3 \rightarrow p_4$ satır 3: 18 + 5 = 23.

Adım 8 / 9

DEĞİŞKENLER

t = 6 p_4 = [22, 19, 19, 23]

Boru Hattı: Kodu Adım Adım İzle

```
▶ for (int t = 0; t < m + p - 2; t++)
2   // tüm j'ler aynı anda
3   for (int j = 0; j < p - 1; j++) {
4     int r = t - j;
5     if (r >= 0 && r < m)
6       x[j + 1][r] += x[j][r];
7   }
```

	p_0	p_1	p_2	p_3	p_4
0	2	5	8	14	22
1	1	8	12	13	19
2	7	9	16	16	19
3	4	13	16	18	23

m + p − 2 = 7 adımda bitti; p_4 satır toplamlarını tutar: 22 19 19 23. Saf yöntem 16 adım sürerdi.

Adım 9 / 9

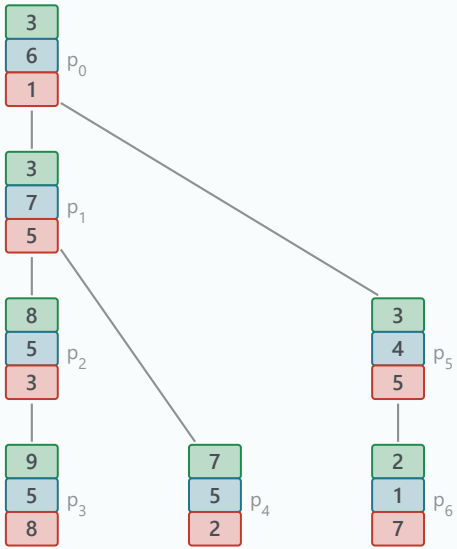
DEĞİŞKENLER

`t = 7 (bitti)` `p₄ = [22, 19, 19, 23]`

Ağaçta İndirgeme

- İşlemciler bir **ağaç** biçiminde bağlıdır; her düğüm değerini **ebeveynine** gönderir, ebeveyn kendi değerine ekler. Sonuç kökte toplanır.
- Ağacın derinliği h ise tek bir değerin indirgenmesi h tur sürer (örneğimizde h = 3).
- Şimdi her işlemcide tek sayı değil, **k bileşenli bir vektör** var (örnekte k = 3: yeşil, mavi, kırmızı). Her bileşen ayrı ayrı toplanacak.
- Saf yöntem: bileşenler sırayla indirgenir → $k \cdot h = 9$ tur.
- **Boru hatlı ağaç** (pipelined tree): bir bileşen yukarı çıkarken bir sonraki bileşen yapraklardan yola çıkar.
- Kural: bir düğüm bir turda **en fazla bir ileti alabilir** (aynı turda bir ileti de gönderebilir).

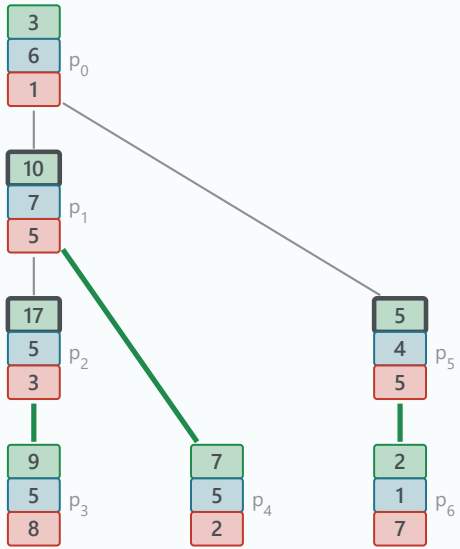
Boru Hatlı Ağaç Adım Adım



7 işlemcinin her biri 3 değerlik bir vektör tutar (yeşil, mavi, kırmızı). Hedef: üç bileşenin toplamı kökte (p_0) toplansın. Değerler alttan köke doğru akar.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

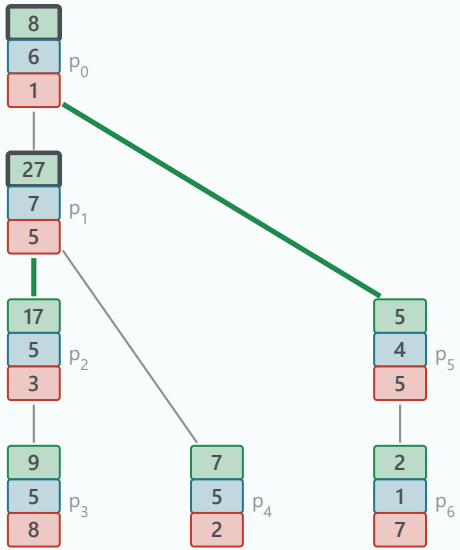
Boru Hatlı Ağaç Adım Adım



Tur 1: yeşil $p_3 \to p_2$: 9+8=17, $p_4 \to p_1$: 7+3=10, $p_6 \to p_5$: 2+3=5.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

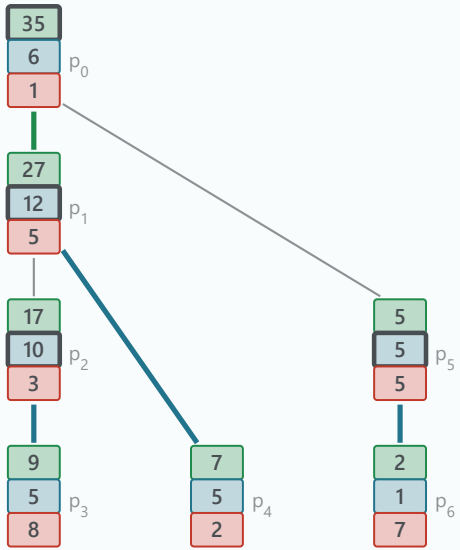
Boru Hatlı Ağaç Adım Adım



Tur 2: yeşil $p_2 \to p_1$: 17+10=27, $p_5 \to p_0$: 5+3=8.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

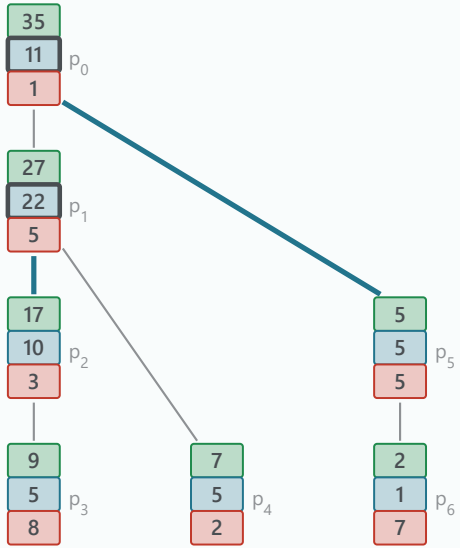
Boru Hatlı Ağaç Adım Adım



Tur 3: yeşil $p_1 \to p_0$: 27+8=35; mavi $p_3 \to p_2$: 5+5=10, $p_4 \to p_1$: 5+7=12, $p_6 \to p_5$: 1+4=5.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

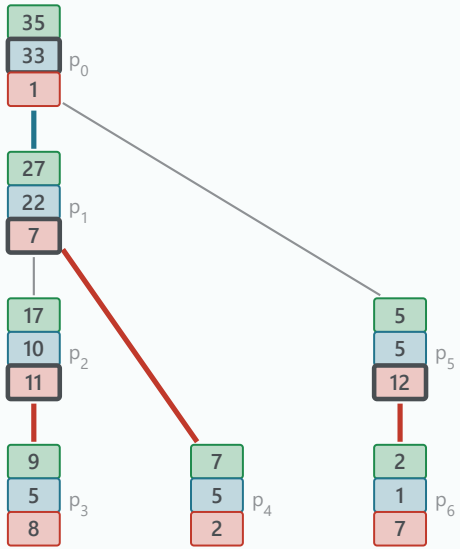
Boru Hatlı Ağaç Adım Adım



Tur 4: mavi $p_2 \to p_1$: 10+12=22, $p_5 \to p_0$: 5+6=11.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

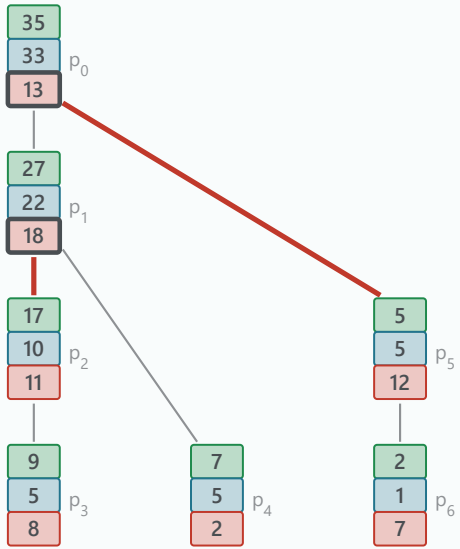
Boru Hatlı Ağaç Adım Adım



Tur 5: mavi $p_1 \to p_0$: 22+11=33; kırmızı $p_3 \to p_2$: 8+3=11, $p_4 \to p_1$: 2+5=7, $p_6 \to p_5$: 7+5=12.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

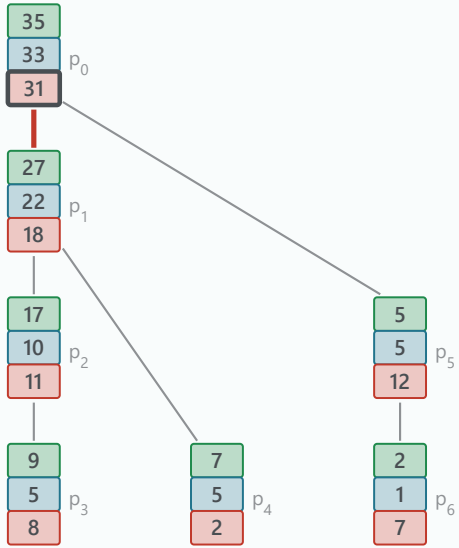
Boru Hatlı Ağaç Adım Adım



Tur 6: kırmızı $p_2 \to p_1$: 11+7=18, $p_5 \to p_0$: 12+1=13.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

Boru Hatlı Ağaç Adım Adım



Tur 7: kırmızı $p_1 \to p_0$: 18+13=31. Kökte (35, 33, 31): 7 turda bitti.

- Kenarın rengi, o turda taşınan bileşeni gösterir; ok ebeveyne doğrudur.
- Kalın çerçeveli kutu, bu turda güncellenen değerdir.
- Her bileşen 3 aşamada köke çıkar:
  (1) $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$
  (2) $p_2 \to p_1$, $p_5 \to p_0$
  (3) $p_1 \to p_0$.
- Yeni bileşen **2 turda bir** başlar.

Boru Hatlı Ağaç: Analiz

tur	aşama 1 $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$	aşama 2 $p_2 \to p_1$, $p_5 \to p_0$	aşama 3 $p_1 \to p_0$
1	yeşil	—	—
2	—	yeşil	—
3	mavi	—	yeşil
4	—	mavi	—
5	kırmızı	—	mavi
6	—	kırmızı	—
7	—	—	kırmızı

- Neden 2 turda bir? p_1 iki çocuğundan (p_4 ve p_2) ileti alır; bir turda tek ileti alabildiği için bir bileşen p_1'i **2 tur** meşgul eder. p_0 için de aynısı.
- Toplam tur: $h + (k - 1) \cdot 2 = 3 + 2 \cdot 2 =$ **7** (saf: $k \cdot h = 9$).
- k büyüdükçe: $2k + 1 \approx 2k$ tur; saf yöntem $3k$. Genel olarak tur $\approx h + (k - 1) \cdot$ (en fazla çocuk sayısı).
- Sonuç kökte: yeşil 35, mavi 33, kırmızı 31.

Paralel Algoritma Türleri

Böl ve fethet

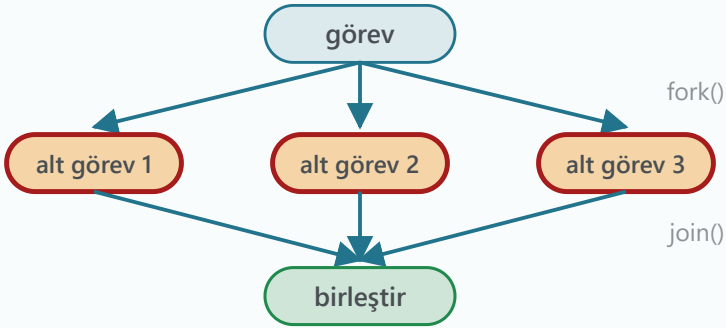
Problem bağımsız alt problemlere bölünür; alt problemler farklı çekirdeklerde aynı anda çözülür, sonuçlar birleştirilir. Paralel birleştirme sıralaması, hızlı sıralama, matris çarpımı.

MapReduce

Büyük veri dağıtık düğümlerde işlenir: **map** adımı ara anahtar–değer çiftleri üretir, **reduce** adımı bunları birleştirir. Hadoop, Spark; sayma, dizinleme, günlük analizi.

Veri paralel

Aynı hesaplama verinin farklı parçalarına aynı anda uygulanır (SIMD komutları, GPU). Bilimsel simülasyon, görüntü işleme, sayısal hesaplama.



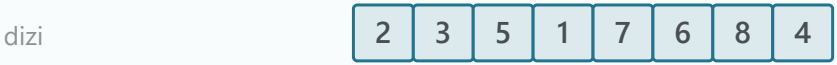
Paralel Birleştirme Sıralaması

```
class SiralamaGorevi extends RecursiveTask<int[]> {
  private final int[] dizi;
  SiralamaGorevi(int[] dizi) { this.dizi = dizi; }

  protected int[] compute() {
    if (dizi.length <= 1) return dizi;
    int orta = dizi.length / 2;
    var sol = new SiralamaGorevi(
        Arrays.copyOfRange(dizi, 0, orta));
    var sag = new SiralamaGorevi(
        Arrays.copyOfRange(dizi, orta, dizi.length));
    sol.fork();               // paralel başlat
    sag.fork();
    return birlestir(sol.join(), sag.join());
  }
}
// kullanım:
int[] s = new ForkJoinPool().invoke(
    new SiralamaGorevi(dizi));
```

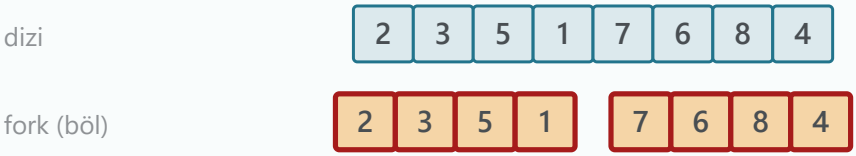
- `RecursiveTask` : sonuç döndüren Fork/Join görevi.
- `fork()` : alt görevi havuza verir (asenkron).
- `join()` : alt görevin bitmesini bekler, sonucunu alır.
- `birlestir` : bilinen sıralı birleştirme, O(n).
- `ForkJoinPool` iş çalma (work stealing) ile boşta kalan çekirdeğe iş verir.

Paralel Birleştirme Sıralaması: 2 3 5 1 7 6 8 4



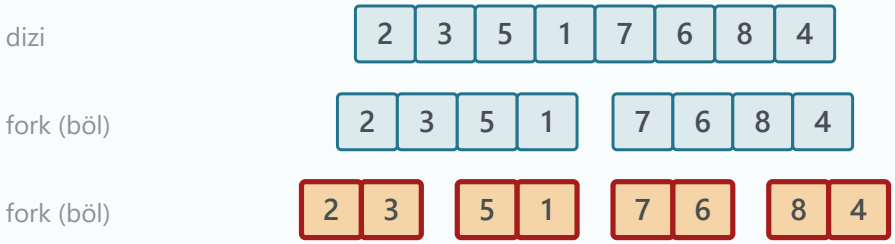
Görev: 8 öğeli diziyi sırala. `SiralamaGorevi` kök görevdir.

Paralel Birleştirme Sıralaması: 2 3 5 1 7 6 8 4



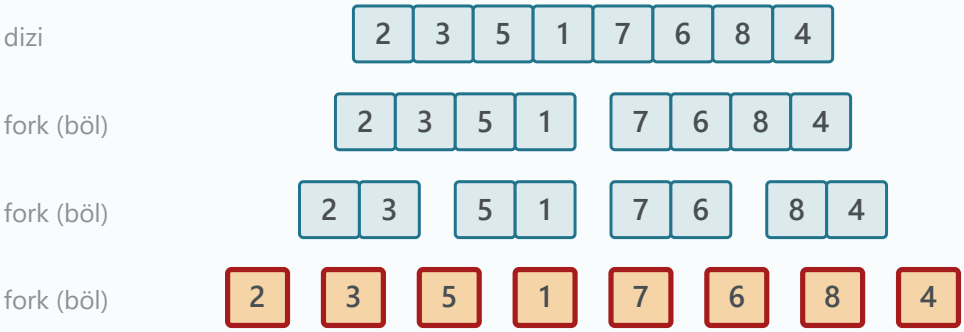
Görev diziyi ikiye böler ve iki alt görevi `fork()` ile havuza verir; iki yarı **aynı anda** işlenebilir.

Paralel Birleştirme Sıralaması: 2 3 5 1 7 6 8 4



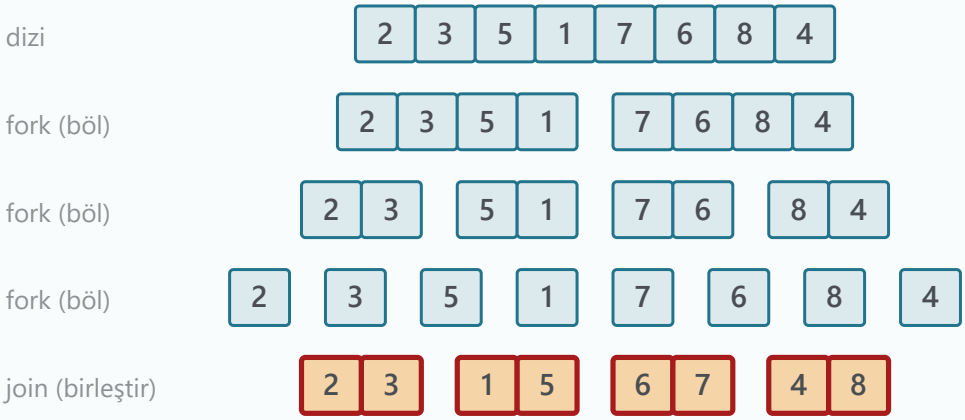
Her alt görev de böler: artık 4 görev paralel çalışabilir.

Paralel Birleştirme Sıralaması: 2 3 5 1 7 6 8 4



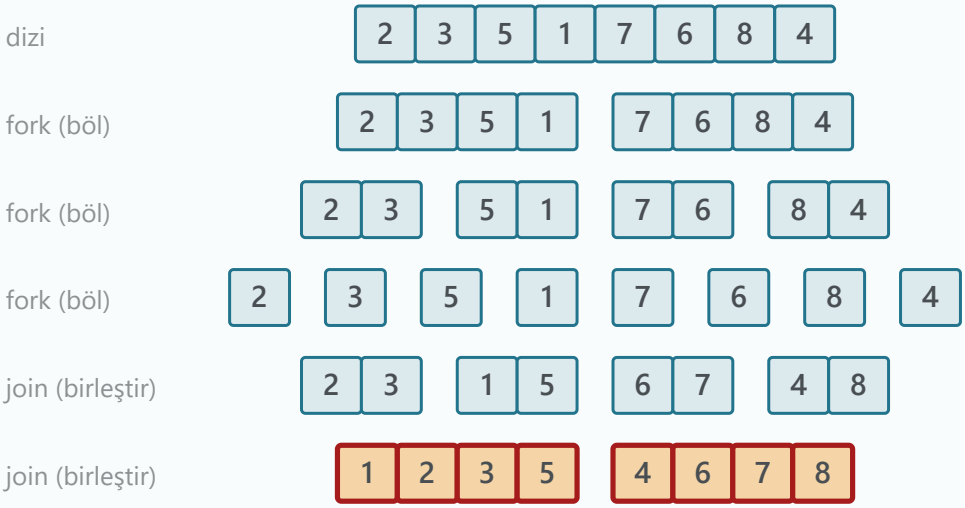
Taban durumu: 1 öğeli diziler zaten sıralıdır (8 yaprak görev).

Paralel Birleştirme Sıralaması: 2 3 5 1 7 6 8 4



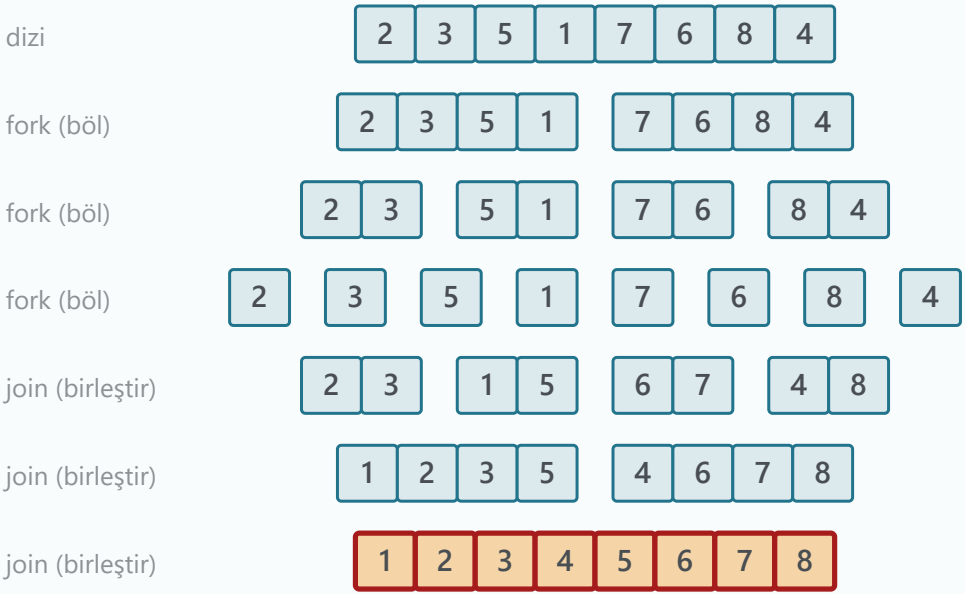
`join()` : her ebeveyn iki çocuğunun sonucunu bekler ve birleştirir: [2 3] [1 5] [6 7] [4 8].

Paralel Birleştirme Sıralaması: 2 3 5 1 7 6 8 4



Bir üst düzeyde iki birleştirme paralel yapılır: [1 2 3 5] ve [4 6 7 8].

Paralel Birleştirme Sıralaması: 2 3 5 1 7 6 8 4



Kök görev son birleştirmeyi **tek başına** yapar: [1 2 3 4 5 6 7 8]. Bu sıralı birleştirme, derinliği (span) Θ(n) yapar.

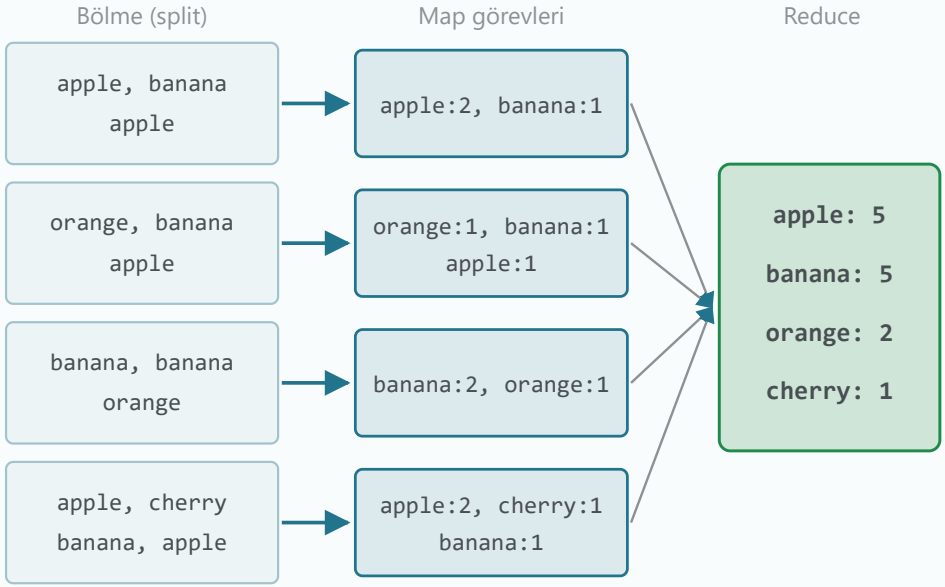
Paralel Sıralama: Analiz

Sürüm	İş W	Derinlik D	Paralellik W / D
Sıralı birleştirme sıralaması	$\Theta(n \log n)$	$\Theta(n \log n)$	1
Paralel özyineleme + sıralı birleştirme	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(\log n)$
Paralel özyineleme + paralel birleştirme (CLRS)	$\Theta(n \log n)$	$\Theta(\log^3 n)$	$\Theta(n / \log^2 n)$

- Derinlik yinelemesi: $D(n) = D(n/2) + \Theta(n) = \Theta(n)$, çünkü iki yarı paralel ama birleştirme sıralıdır.
- Paralellik yalnızca $\Theta(\log n)$: $n = 10^6$ için ≈ 20 çekirdekten fazlası pek fayda etmez.
- Uygulamada: küçük parçalarda (örn. < 1000 öğe) **sıralı** sıralamaya geçilir; aksi hâlde görev oluşturma yükü kazancı yer.
- Hazır çözüm: `Arrays.parallelSort(dizi)`.

MapReduce

- **Map**: her parça bağımsız işlenir ve ara **anahtar–değer** çiftleri üretir (kelime → adet).
- **Karıştır** (shuffle): aynı anahtarlar aynı reducer'a toplanır.
- **Reduce**: her anahtarın değerleri birleştirilir (toplanır).
- Örnekte: 13 kelime, 4 iş parçacığı → parça boyu ⌊13/4⌋ = 3; son parça 4 kelime.



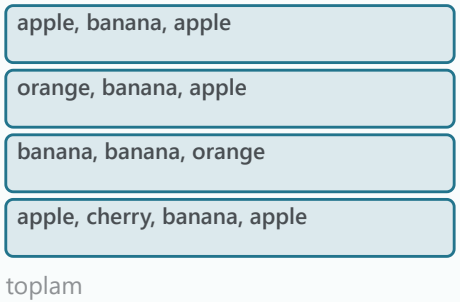
MapReduce: Kod

```
// Map: bir parçadaki kelimeleri say
protected Map<String, Integer> compute() {
  Map<String, Integer> sonuc = new HashMap<>();
  for (String kelime : parca)
    sonuc.merge(kelime, 1, Integer::sum);
  return sonuc;
}
// Ana akış
for (MapGorevi g : mapGorevleri) g.fork();
List<Map<String, Integer>> ara = new ArrayList<>();
for (MapGorevi g : mapGorevleri) ara.add(g.join());
// Reduce: ara sonuçları birleştir
Map<String, Integer> toplam = new HashMap<>();
for (Map<String, Integer> m : ara)
  m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```

- Her map görevinin **kendi** HashMap'i vardır: paylaşılan durum yok, kilit gerekmez.
- Sonuç: apple 5, banana 5, orange 2, cherry 1.
- Bu örnekte reduce tek görevdir; gerçek MapReduce'ta anahtarlar birçok reducer'a **bölüştürülür** (hash(k) mod r).

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
▶  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
11 for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
14 for (Map<String, Integer> m : ara)
15   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



13 kelime 4 parçaya bölündü; her parça için bir `MapGorevi` `fork()` ile havuza verilir ve görevler paralel çalışır.

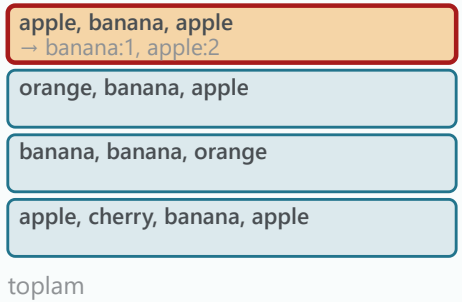
Adım 1 / 10

DEĞİŞKENLER

`görev = 4`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
▶ for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
14 for (Map<String, Integer> m : ara)
15   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



`join()` : 0. görevin kendi HashMap'i hazır → banana 1, apple 2. Paylaşılan durum yok, kilit gerekmez.

Adım 2 / 10

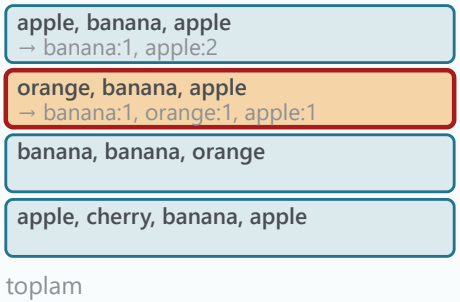
DEĞİŞKENLER

`g = 0`

`ara[0] = {banana:1, apple:2}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
▶ for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
14 for (Map<String, Integer> m : ara)
15   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



`join()` : 1. görevin kendi HashMap'i hazır → banana 1, orange 1, apple 1. Paylaşılan durum yok, kilit gerekmez.

Adım 3 / 10

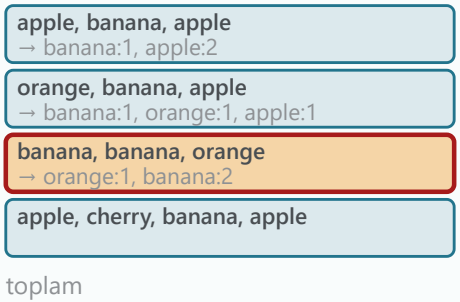
DEĞİŞKENLER

`g = 1`

`ara[1] = {banana:1, orange:1, apple:1}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
▶ for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
14 for (Map<String, Integer> m : ara)
15   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



`join()` : 2. görevin kendi HashMap'i hazır → orange 1, banana 2. Paylaşılan durum yok, kilit gerekmez.

Adım 4 / 10

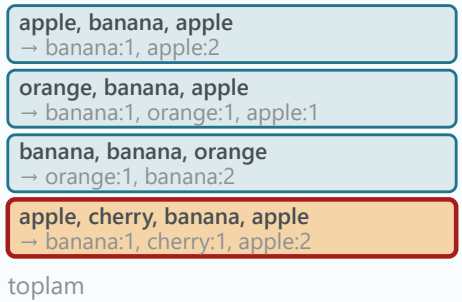
DEĞİŞKENLER

`g = 2`

`ara[2] = {orange:1, banana:2}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
▶ for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
14 for (Map<String, Integer> m : ara)
15   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



`join()` : 3. görevin kendi HashMap'i hazır → banana 1, cherry 1, apple 2. Paylaşılan durum yok, kilit gerekmez.

Adım 5 / 10

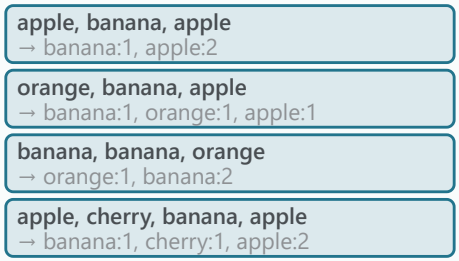
DEĞİŞKENLER

`g = 3`

`ara[3] = {banana:1, cherry:1, apple:2}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
11 for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
▶ Map<String, Integer> toplam = new HashMap<>();
14 for (Map<String, Integer> m : ara)
15   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



Reduce başlar: boş `toplam` haritası.

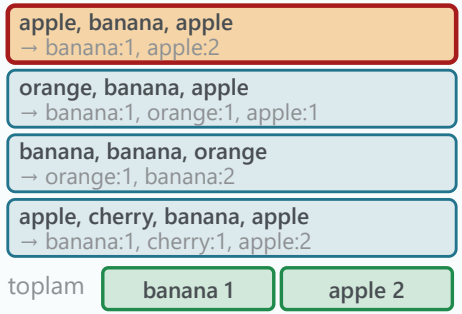
Adım 6 / 10

DEĞİŞKENLER

`toplam = {}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
11 for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
▶ for (Map<String, Integer> m : ara)
▶   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



ara[0] birleştirildi: aynı anahtarlar `Integer::sum` ile toplanır.

Adım 7 / 10

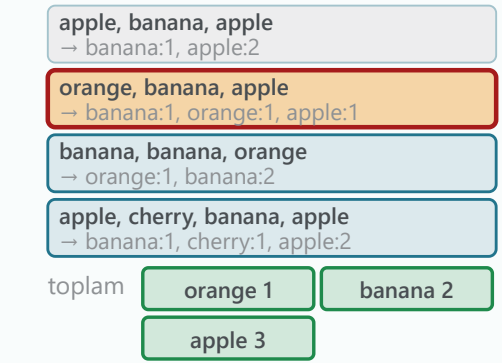
DEĞİŞKENLER

`m = ara[0]`

`toplam = {banana:1, apple:2}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
11 for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
▶ for (Map<String, Integer> m : ara)
▶   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



ara[1] birleştirildi: aynı anahtarlar `Integer::sum` ile toplanır.

Adım 8 / 10

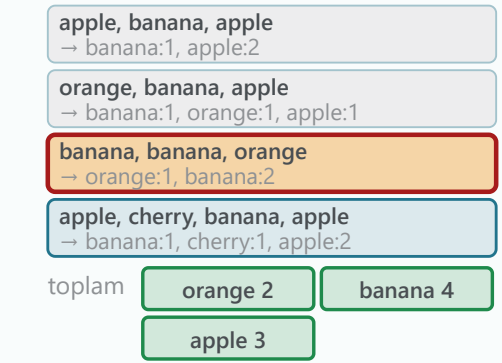
DEĞİŞKENLER

`m = ara[1]`

`toplam = {orange:1, banana:2, apple:3}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
11 for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
▶ for (Map<String, Integer> m : ara)
▶   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



ara[2] birleştirildi: aynı anahtarlar `Integer::sum` ile toplanır.

Adım 9 / 10

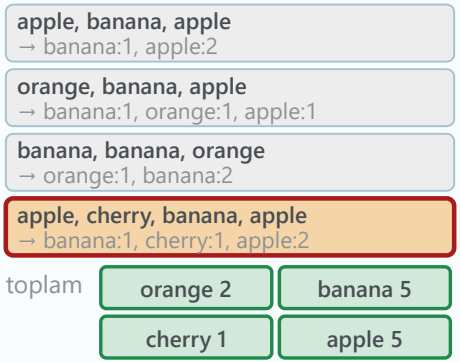
DEĞİŞKENLER

`m = ara[2]`

`toplam = {orange:2, banana:4, apple:3}`

MapReduce: Kodu Adım Adım İzle

```
1  // Map: bir parçadaki kelimeleri say
2  protected Map<String, Integer> compute() {
3    Map<String, Integer> sonuc = new HashMap<>();
4    for (String kelime : parca)
5      sonuc.merge(kelime, 1, Integer::sum);
6    return sonuc;
7  }
8  // Ana akış
9  for (MapGorevi g : mapGorevleri) g.fork();
10 List<Map<String, Integer>> ara = new ArrayList<>();
11 for (MapGorevi g : mapGorevleri) ara.add(g.join());
12 // Reduce: ara sonuçları birleştir
13 Map<String, Integer> toplam = new HashMap<>();
▶ for (Map<String, Integer> m : ara)
▶   m.forEach((k, v) -> toplam.merge(k, v, Integer::sum));
```



ara[3] birleştirildi: aynı anahtarlar `Integer::sum` ile toplanır. Sonuç: orange 2, banana 5, cherry 1, apple 5.

Adım 10 / 10

DEĞİŞKENLER

`m = ara[3]`

`toplam = {orange:2, banana:5, cherry:1, apple:5}`

Veri Paralelliği: Matris Çarpımı

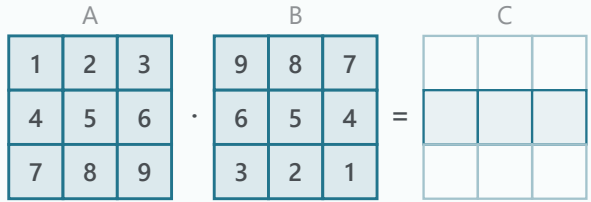
```
// her satır ayrı bir görev
class SatirGorevi extends RecursiveAction {
  protected void compute() {
    for (int j = 0; j < n; j++) {
      C[satir][j] = 0;
      for (int k = 0; k < n; k++)
        C[satir][j] += A[satir][k] * B[k][j];
    }
  }
}
try (var havuz = new ForkJoinPool()) {
  for (int i = 0; i < n; i++)
    havuz.submit(new SatirGorevi(i));
}   // close(): tüm görevleri bekler
```

C = A · B		
30	24	18
84	69	54
138	114	90

- A = [[1,2,3],[4,5,6],[7,8,9]], B = [[9,8,7],[6,5,4],[3,2,1]].
- Satırlar bağımsız: n görev, her biri $\Theta(n^2)$ iş. $W = \Theta(n^3)$, $D = \Theta(n^2)$ (satır başına).
- Her görev yalnızca **kendi satırına** yazar → yarış yok.

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
▶ protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[satir][j] = 0;
5     for (int k = 0; k < n; k++)
6       C[satir][j] += A[satir][k] * B[k][j];
7   }
8 }
```



Bu görev yalnızca C'nin 1. satırını hesaplar: A'nın 1. satırı (4, 5, 6) ile B'nin her sütunu. Öteki satırlar başka görevlerde.

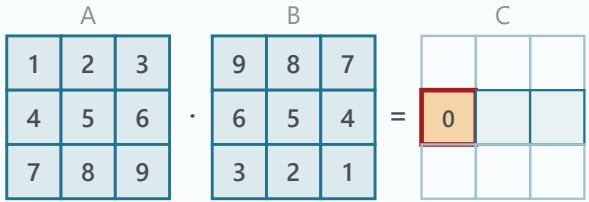
Adım 1 / 8

DEĞİŞKENLER

`satir = 1` `C[1] = [–, –, –]`

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
2 protected void compute() {
▶   for (int j = 0; j < n; j++) {
▶     C[satir][j] = 0;
5     for (int k = 0; k < n; k++)
6       C[satir][j] += A[satir][k] * B[k][j];
7   }
8 }
```



j = 0: C[1][0] = 0 ile başlar.

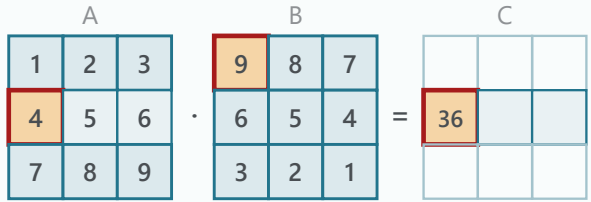
Adım 2 / 8

DEĞİŞKENLER

`satir = 1` `j = 0` `C[1][0] = 0`

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[satir][j] = 0;
      for (int k = 0; k < n; k++)
        C[satir][j] += A[satir][k] * B[k][j];
7   }
8 }
```



k = 0: C[1][0] += A[1][0] · B[0][0] = 4 · 9 → **36**.

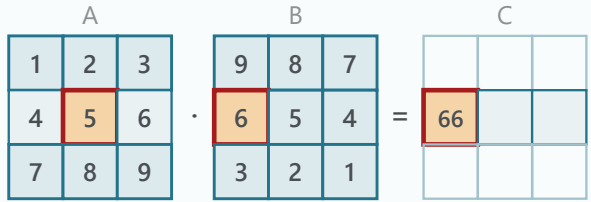
Adım 3 / 8

DEĞİŞKENLER

`satir = 1` `j = 0` `k = 0` `C[1][0] = 36`

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[satir][j] = 0;
▶     for (int k = 0; k < n; k++)
▶       C[satir][j] += A[satir][k] * B[k][j];
7   }
8 }
```



k = 1: C[1][0] += A[1][1] · B[1][0] = 5 · 6 → **66**.

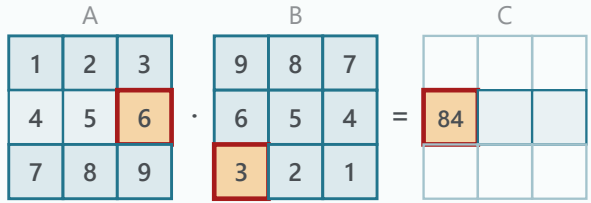
Adım 4 / 8

DEĞİŞKENLER

`satir = 1` `j = 0` `k = 1` `C[1][0] = 66`

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[satir][j] = 0;
▶     for (int k = 0; k < n; k++)
▶       C[satir][j] += A[satir][k] * B[k][j];
7   }
8 }
```



k = 2: C[1][0] += A[1][2] · B[2][0] = 6 · 3 → **84**.

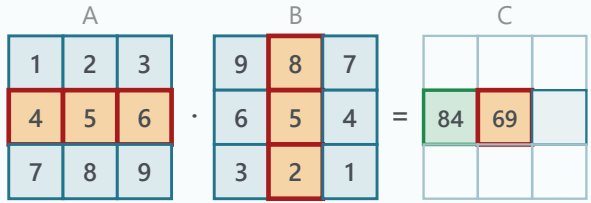
Adım 5 / 8

DEĞİŞKENLER

`satir = 1` `j = 0` `k = 2` `C[1][0] = 84`

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
2 protected void compute() {
    for (int j = 0; j < n; j++) {
      C[satir][j] = 0;
      for (int k = 0; k < n; k++)
        C[satir][j] += A[satir][k] * B[k][j];
7   }
8 }
```



j = 1: 4 · 8 + 5 · 5 + 6 · 2 = **69**.

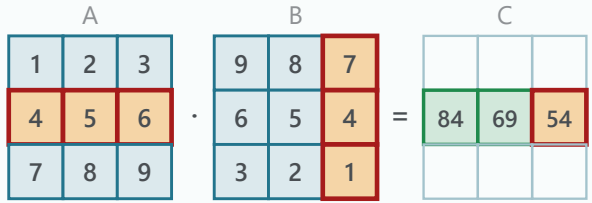
Adım 6 / 8

DEĞİŞKENLER

`satir = 1` `j = 1` `C[1][1] = 69`

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
2 protected void compute() {
    for (int j = 0; j < n; j++) {
      C[satir][j] = 0;
      for (int k = 0; k < n; k++)
        C[satir][j] += A[satir][k] * B[k][j];
7   }
8 }
```



j = 2: 4 · 7 + 5 · 4 + 6 · 1 = **54**.

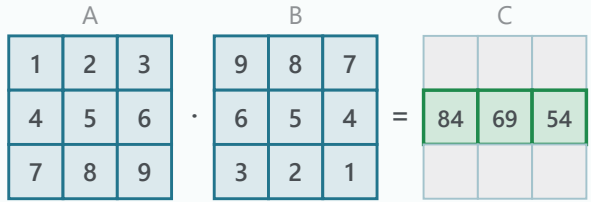
Adım 7 / 8

DEĞİŞKENLER

`satir = 1` `j = 2` `C[1][2] = 54`

Satır Görevi: Kodu Adım Adım İzle

```
1 // SatirGorevi, satir = 1
2 protected void compute() {
3   for (int j = 0; j < n; j++) {
4     C[satir][j] = 0;
5     for (int k = 0; k < n; k++)
6       C[satir][j] += A[satir][k] * B[k][j];
7   }
▶ }
```



Satır 0 ve 2 başka görevlerde, aynı anda hesaplanır.

Görev bitti: C[1] = (84, 69, 54). Her görev yalnızca kendi satırına yazdığı için yarış durumu yok.

Adım 8 / 8

DEĞİŞKENLER

`satir = 1` `C[1] = [84, 69, 54]`

PARALEL ALGORİTMA TÜRLERİ

Yarış Durumu

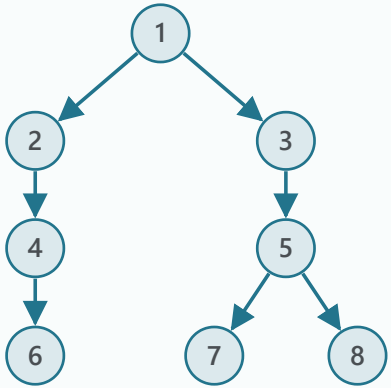
İki iş parçacığı paylaşılan `sayac`'ı aynı anda artırır. `sayac++` aslında üç adımdır: **oku → artır → yaz**.

zaman	İş parçacığı 1	İş parçacığı 2	sayac
1	oku (0)		0
2		oku (0)	0
3	yaz 0 + 1		1
4		yaz 0 + 1	**1**

Beklenen 2, sonuç 1: **kayıp güncelleme** (lost update).

- **Yarış durumu** (race condition): sonuç, iş parçacıklarının zamanlamasına bağlıdır; hata her çalıştırmada ortaya çıkmayabilir.
- Çözümler: `synchronized` blok / kilit, `AtomicInteger.incrementAndGet()`, eşzamanlı koleksiyonlar (`ConcurrentHashMap`).
- En iyisi paylaşılan durumdan kaçınmaktır: her görev kendi kısmi sonucunu üretir, sonuçlar sonra birleştirilir (indirgeme, MapReduce).
- Kilitler de bedelsiz değildir: sıralı bölge oluşturur (Amdahl) ve **kilitlenme** (deadlock) riski getirir.

Eşzamanlı Çizge Gezinme



- Fikir: her ziyaret edilmemiş komşu için yeni bir gezinme görevi `fork()` edilir; alt ağaçlar paralel gezilir.
- Görevler `ziyaret` kümesini **paylaşır**. Bu küme sıradan bir `HashSet` olursa eşzamanlı erişimde güvenli değildir.
- Ayrıca `contains` + `add` iki ayrı adımdır: iki görev aynı düğümü birlikte ziyaret edebilir.

```
Set<Dugum> ziyaret = ConcurrentHashMap.newKeySet();
if (!ziyaret.add(komsu)) continue; // atomik kontrol+ekle
```

Düzey eşzamanlı BFS: her düzeyin tüm düğümleri paralel genişletilir, düzey sonunda bariyer; D = O(çap).

PARALEL ALGORİTMA TÜRLERİ

Java'da Paralellik Araçları

Araç	Ne zaman?
`Thread`, `Runnable`	en alt düzey; elle yönetim, nadiren doğrudan kullanılır
`ExecutorService`	bağımsız görevleri sabit bir iş parçacığı havuzunda çalıştırmak
`ForkJoinPool`, `RecursiveTask`	böl ve fethet; iş çalma ile özyinelemeli görevler
`list.parallelStream()`	veri paralel map / filter / reduce (ortak ForkJoinPool kullanır)
`Arrays.parallelSort`	büyük dizileri paralel sıralamak
`Arrays.parallelPrefix`	paralel önek toplamı (scan)
`AtomicInteger`, `ConcurrentHashMap`	paylaşılan sayaç/küme gerektiğinde güvenli erişim

```
Map<String, Long> adet = kelimeler.parallelStream()
    .collect(Collectors.groupingByConcurrent(k -> k, Collectors.counting()));
```

Özet

Konu	Hatırlanacaklar
Fikir	böl → işlemcilere ata → aynı anda çöz → birleştir; bedeli iletişim, senkronizasyon, yük dengeleme
Modeller	Flynn: SISD/SIMD/MISD/MIMD; PRAM: EREW, CREW, CRCW; paylaşımlı / dağıtık bellek
Ölçütler	$S = T_1/T_p$, $E = S/p$, $C = p \cdot T_p$; Amdahl $S \le 1/(1 - f)$; Brent $T_p \le W/p + D$
PRAM toplama	$\log_2 n$ tur, $W = n - 1$; $p = n/\log n$ ile maliyet-optimal
Önek toplamı	Hillis–Steele: $\log_2 n$ tur, $O(n \log n)$ iş; Blelloch: $O(n)$ iş
Boru hattı	p işlemci, m satır: $m + p - 2$ adım (saf $m(p - 1)$)
Boru hatlı ağaç	$h + (k - 1) \cdot 2$ tur (örnek 7; saf $k \cdot h = 9$)
Java	Fork/Join (fork, join), MapReduce, satır başına görev; paylaşılan durumda yarış → atomik/eşzamanlı yapılar

ÖZET

Sık Yapılan Hatalar

p işlemci = p kat hız sanmak

Sıralı kısım (Amdahl), iletişim ve bekleme yüzünden S < p'dir. %10 sıralı kod hızlanmayı 10 ile sınırlar.

Paylaşılan HashSet / sayaç

Eşzamanlı yazma yarış durumuna yol açar. `ConcurrentHashMap`, `AtomicInteger` ya da yerel sonuç + birleştirme kullanılır.

Çok ince taneli görevler

Her öğe için `fork()` yapmak, görev oluşturma yükünü artırır. Eşiğin altındaki parçalar sıralı işlenir.

Turlar arası senkronizasyonu unutmak

PRAM toplamada tur t + 1, tur t'nin **tüm** yazmaları bitmeden başlarsa eski değerler okunur.

Yalnızca derinliğe bakmak

Hillis–Steele O(log n) turda biter ama O(n log n) iş yapar; az işlemcide sıralı taramadan yavaş olabilir.

Boru hattı adımını yanlış saymak

Hattın dolma süresi unutulur: m satır, p işlemci için m adım değil, **m + p − 2** adım.

ÖZET

Örnek Problemler

1. n = 16 sayı PRAM ağacıyla toplanıyor. Kaç tur sürer, ilk turda kaç işlemci çalışır, toplam kaç toplama yapılır?
2. Boru hattında p = 5 işlemci ve m = 100 satır varsa kaç adım gerekir? Saf yöntemle karşılaştırın.
3. Bölümdeki boru hatlı ağaç örneğinde (7 işlemci, h = 3) her işlemci k = 10 bileşenli vektör tutsaydı kaç tur gerekirdi?
4. Programın %80'i paraleldir. p = 4 için Amdahl hızlanması nedir, üst sınır kaçtır?
5. T_1 = 120 s, T_6 = 30 s. Hızlanma, verimlilik ve maliyeti bulun.
6. [1, 2, 3, 4] dizisinin Hillis–Steele önek toplamını turlar hâlinde yazın; kaç toplama yapılır?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. PRAM ağaç toplamını **en büyük öğe** bulacak şekilde değiştirin. Süre ve iş nedir?
2. n = 8 için Hillis–Steele önek toplamı kaç toplama yapar? Sıralı algoritmayla karşılaştırın.
3. Paralel gezinmedeki `ziyaret` kümesini iş parçacığı güvenli yapın.
4. Paralel birleştirme sıralamasına eşik ekleyin; neden iki alt görevden birini `fork()` yerine doğrudan `compute()` etmek daha iyidir?
5. p = 32 işlemcide S = 16 hızlanma için programın en az ne kadarı paralel olmalıdır (Amdahl)?
6. m satır, p işlemcili boru hattında adım sayısını ve verimliliği genel olarak ifade edin.

▶ **Cevaplar / ipuçları**

Kaynaklar

- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 26: Parallel Algorithms).
- J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
- A. Grama, A. Gupta, G. Karypis ve V. Kumar, *Introduction to Parallel Computing*, 2. baskı, Addison-Wesley, 2003.
- G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", AFIPS, 1967.
- J. Dean ve S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI, 2004.
- D. Lea, "A Java Fork/Join Framework", ACM Java Grande Conference, 2000.

Son

Bölüm 9: Paralel Algoritmalar

Dersin sonu · Terim sözlüğü