Rinnakkaisuus

Tarkastelemme, miten algoritmien suoritusta voi nopeuttaa käyttämällä useaa laskentayksikköä samanaikaisesti.

- Miksi rinnakkaisuus on tärkeää?
- Millaisia nopeutuksia rinnakkaistamalla ylipäänsä voi saada aikaan?
- Mitä korkean tason työkaluja Java 8 tarjoaa rinnakkaisuuden saamiseksi käyttöön?
- Millaisia tekniikoita joissain tärkeissä erityisesti rinnakkain suoritettavaksi tarkoitetuissa algoritmeissa käytetään?

Rinnakkaisuus ei kuulu koealueeseen, mutta siitä tulee yksi ylimääräinen laskuharjoitustehtävä.

Rinnakkaisuus ja samanaikaisuus

Rinnakkaisuudella (parallelism) tarkoitetaan laskennan nopeuttamista jakamalla se useaan mahdollisimman riippumattomaan osaan ja suorittamalla kukin osa omalla laskentalaitteellaan (prosessorilla, ytimellä tms.).

Samanaikaisuudella (concurrency) tarkoitetaan yleisemmin sitä, että useita asioita tapahtuu samaan aikaan. Tämä ei välttämättä edellytä, että käytössä on usea prosessori:

- tietokanta, jolla on useita käyttäjiä
- käyttäjällä voi olla samaan aikaan auki tekstinkäsittelyohjelma ja verkkoselain, ja tausta-ajona jotain tieteellistä laskentaa.

Samanaikaisuuden hallinnassa keskeistä on yleensä varmistaa, että järjestelmän eri osien vuorovaikutukset toimivat tarkoitetulla tavalla:

- eri käyttäjien voivat yrittää lukea ja kirjoittaa samaa tietokannan tietoalkiota
- tekstinkäsittelyohjelmalla on suurempi prioriteetti kuin tausta-ajolla.

Tämän luennon tarkoituksena on antaa joitain ajatuksia siitä, miten rinnakkaisuus voi nopeuttaa laskentaa sen tyyppisissä tilanteissa, joita tällä kurssilla muutenkin on käsitelty.

- algoritmit korkean tason pseudokoodina
- esitellään vastaavia korkean abstraktiotason piirteitä Javassa
- emme tarkastele rinnakkaisuuden toteutusta (esim. prosessien välistä kommunikointia)
- emme tarkastele hajautetun laskennan erityispiirteitä.

Mistä peruskäyttäjä saa rinnakkaista laskentavoimaa?

Tyypillisessä henkilökohtaisen tietokoneen keskussuorittimessa (CPU) voi nykyään olla esim. neljä ydintä (core). Tämän esityksen kannalta jokainen ydin on itsenäinen laskentalaite.

- Yksinkertaisuuden vuoksi käytämme jatkossa termiä ”suoritin” (tai ”prosessori”) tällaisesta laskentalaitteesta, joka siis voi olla myös yksi ydin moniydinsuorittimessa.

Grafiikkasuorittimessa (GPU) on hyvin suuri määrä rinnakkaisuutta (satoja ytimiä), mutta niiden toiminnot ovat melko erikoistuneita.

- Grafiikkasuoritinta voi kuitenkin hyödyntää myös yleislaskennassa (CUDA-rajapinta, kurssi *Rinnakkaislaskenta grafiikkasuorittimilla*).

Enemmän prosessoreita voi vuokrata käyttöönsä pilvipalveluista kohtuuhintaan (esim. 32 ydintä parilla eurolla per tunti).

Miksi rinnakkaisuuden merkitys kasvaa

- Mooren lain (1965) mukaan transistorien määrä mikropiirillä kaksinkertaistuu 2 vuodessa.
- Tämä kehitys jatkuu edelleen.
- Viime vuosiin asti transistorimäärän kasvu on heijastunut suoraan suorittimien kellotaajuuksin ja siten laskennan nopeuteen.
- Tämä kehitys on nyt oleellisesti pysähtynyt.
- Jatkossa transistorien lisääntyminen nopeuttaa laskentaa vain epäsuorasti:
  - moniydinsuorittimet
  - enemmän nopeaa muistia.
- **Johtopäätös:** jos halutaan jatkossa yhä nopeampia ohjelmia, niin kuin tähänkin asti, on pakko ruveta hyödyntämään rinnakkaisuutta.

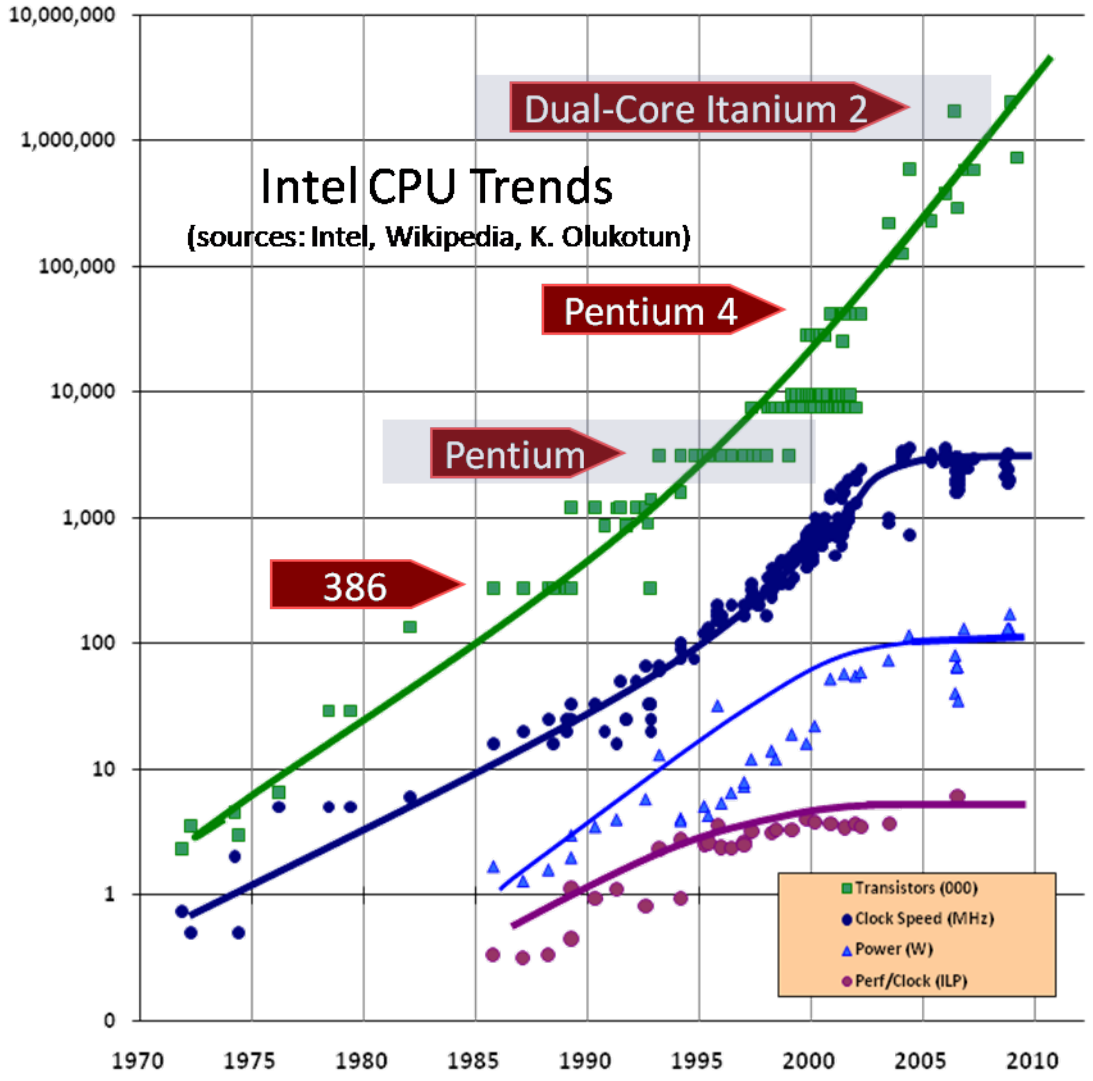
Seuraavan sivun kuvan kaksi ylintä käyrää havainnollistavat tilannetta Intelin suorittimien osalta:

- **vihreä**/neliöt: transistorien lukumäärä
- **sininen**/pallot: kellotaajuus.

Pystyakselin asteikko on logaritminen, joten nouseva suora esittää eksponentiaalista kasvua.

Kuvassa on myös

- **vaaleansininen**/kolmiot: suorittimen teho
- **sinipunainen**: suorituskyky kellosykliä kohti (ILP, Instruction Level Parallelism)



Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal* 30(3), March 2005 (kuva päivitetty 8/2009)

Paljonko rinnakkaisuudesta voi hyötyä

Tarkastellaan esimerkkinä tehtävää valmistaa kattilallinen vihanneskeittoa seuraavalla hieman yksinkertaistetulla algoritmilla:

1. Pilko vihannekset.
2. Laita ainekset kattilaan ja anna kiehua, kunnes keitto on kypsä.

Oletetaan, että yhdeltä kokilta vaihe 1 vie A minuuttia ja vaihe 2 B minuuttia.

Jos kokkeja on useita, vaihe 1 voidaan rinnakkaistaa, vaihetta 2 ei voi. Saman keiton valmistamiseen menisi K kokin joukkueelta parhaimmillaan $A/K + B$ minuuttia. Siis nopeutus voi olla korkeintaan

$$S = \frac{A + B}{A/K + B}.$$

Käytännössä tähän ei yleensä päästä, sillä etenkin suurilla K aiheutuu lisäkuormaa kommunikoinnista ym.

Olkoon nyt yleisemmin $T(p)$ laskenta-aika, kun jokin ongelma ratkaistaan käyttäen p suoritinta.

Erityisesti $T(1)$ on sarjalliseen ratkaisemisee kuluva aika. Tarkastelemme nopeutusta

$$S(p) = \frac{T(1)}{T(p)}.$$

Oletetaan, että laskennasta osuus r voidaan rinnakkaistaa.

Edellisen sivun tilanteessa siis $T(1) = A + B$ ja $r = A/(A + B)$.

Kun käytetään p prosessoria, laskenta-aika on ainakin

$$T(p) \geq (1 - r)T(1) + rT(1)/p$$

ja nopeutus korkeintaan

$$S(p) \leq \frac{T(1)}{(1 - r)T(1) + rT(1)/p} = \frac{p}{(1 - r)p + r}.$$

Tämä tunnetaan nimellä Amdahlin laki (1967).

Kun $p \to \infty$, yläraja lähestyy arvoa $1/(1 - r)$.

Millaiset algoritmit rinnakkaistuvat

Jotta jotain laskentaa voisi merkittävästi nopeuttaa rinnakkaisuudella, sen pitää olla jaettavissa osaongelmiin, jotka ovat ratkaistavissa

- toisistaan riippumatta
- missä tahansa järjestyksessä.

Tarkastelemme esimerkkinä kolmea Java 8:n metodia, joiden toteutus hyödyntää rinnakkaisuutta:

- Arrays.parallelSet: taulukon alustaminen
- Arrays.parallelSort: taulukon järjestäminen
- Arrays.parallelPrefix: ns. alkuosasummien laskeminen.

Ohjelmoijan näkökulmasta

- näitä metodeja voi käyttää tuntematta rinnakkaisuuden teknistä toteutusta
- näitä metodeja luultavasti kannattaa käyttää, jos tietää, että laitteisto tarjoaa paljon rinnakkaista laskentakapasiteettia.

Arrays.parallelSet(⟨***taulukko***⟩ , ⟨***funktio***⟩)

Jos halutaan esim. alustaa taulukon $neliot$ alkioksi $0, 1, 4, 9, 16, \ldots$, tämä tapahtuu kutsulla

```
Arrays.parallelSet(neliot, i -> i*i);
```

Alustusarvot määräävän funktion esitys i -> i*i on lambdalauseke. Meidän kannaltamme se on yksinkertainen tapa ilmaista yksinkertaisia funktioita. Emme tässä mene syvemmälle Javan lambdalausekkeiden filosofiaan.

Oletetaan, että alustettavan taulukon koko on n ja yhden alkion alustaminen sujuu vakioajassa.

Sarjallisesti aikavaativuus siis on $O(n)$.

Jos käytettävissä on p prosessoria, kullekin niistä voi antaa vastuulle noin n/p alkion mittaisen pätkän taulukosta. Kukin prosessori alustaa oman pätkänsä sarjallisesti ajassa $O(n/p)$.

Siis laskenta on kokonaisuudessaan rinnakkaistuva, ja saavutamme nopeutuksen p.

Taulukon alustaminen on esimerkki ns. nolottavan rinnakkaisesta (embarassingly parallel) ongelmasta. Ongelma rinnakkaistuu maksimaalisen tehokkaasti ilman, että asiaa tarvitsee juuri edes miettiä.

Muita tämän tyyppisiä ongelmia ovat esim. monet

- raakaan voimaan perustuvat etsinnät
- optimointimenetelmät ja simulaatiot, joissa tehdään useita toistoja satunnaisilla alkuarvoilla.

Arrays.parallelSort(⟨*taulukko*⟩)

Lähtökohtana Javan rinnakkaisjärjestämisessä on lomitusjärjestäminen.
Pseudokoodi sivulta 104:

```
merge-sort(A,vasen,oikea)
1   if vasen < oikea
2       keski = ⌊(vasen + oikea)/2⌋
3       merge-sort(A,vasen,keski)
4       merge-sort(A,keski+1,oikea)
5       merge(A,vasen,keski,oikea)
```

Selvästi rivien 3 ja 4 rekursiiviset kutsut eivät mitenkään vaikuta toisiinsa ja voidaan suorittaa rinnakkain.

Rivin 5 lomituksen rinnakkaistaminen ei ole yhtä helppoa. Normaali lomitusalgoritmi tarkastelee alkioita tiukasti peräkkäin. Jotain voidaan kuitenkin tehdä. Esitämme seuraavassa perusidean siitä, miten lomitus rinnakkaistetaan Javan parallelSortissa.

Tarkastellaan lomituksen perustilannetta: Taulukot $A[1 \ldots n]$ ja $B[1 \ldots n]$ ovat järjestyksessä. Haluamme lomittaa ne taulukkoon $C[1 \ldots 2n]$. Siis taulukkoon C pitäisi tulla taulukoiden A ja B alkiot niin, että se on kokonaan järjestyksessä. Merkitsemme tätä operaatiota $merge(A, B, C)$.

Tunnetusti tämän voitaisiin tehdä sarjallisesti ajassa $O(n)$ (sivu 108).

Rinnakkaisuuden mahdollistamiseksi sovellamme hajoita ja hallitse -periaatetta ja jaamme taulukon A puoliksi. Siis

$$merge(A, B, C)$$

palautetaan siihen, että suoritetaan

$$merge(A[1 \ldots n/2], B[1 \ldots x], C[1 \ldots n/2 + x]) \quad \text{ja}$$
$$merge(A[n/2 + 1 \ldots n], B[x + 1 \ldots n], C[n/2 + x + 1 \ldots 2n]).$$

Taulukon B jakokohta x valitaan siten, että $B[x] < A[n/2] \leq B[x + 1]$. Tämä löytyy binäärihaulla. (Lisäksi pitää ottaa huomioon erikoistapaukset, joissa B:n alkiot ovat kaikki suurempia kuin $A[n/2]$ tai kaikki pienempia kuin $A[n/2]$.)

Tässä ratkaisussa kokoa $2n$ oleva ongelma jaetaan osiin, joiden koot ovat $n/2 + x$ ja $3n/2 - x$. Aikavaativuuden kannalta olisi hyvä, jos jako olisi mahdollisimman tasainen eli $x \approx n/2$.

Tästä ei kuitenkaan ole mitään takeita. Pahimmassa tapauksessa x on taulukon B jommassa kummassa päässä ja osaongelmien koot ovat $3n/n$ ja $n/2$.

Tämän epätasapainoisempia jakoja ei kuitenkaan pääse syntymään, koska kumpaankin osaongelmaan tulee ainakin $n/2$ taulukon A alkiota.

Algoritmia voi soveltaa, vaikka taulukot eivät olisi saman kokoisia. Tällöin kannattaa valita suurempi taulukko A:n roolin, jolloin taas pienempäänkin osaongelmaan tulee aina vähintään 1/4 kaikista alkioista.

Arrays.parallelPrefix(⟨*taulukko*⟩, ⟨*binäärioperaattori*⟩)

Perusesimerkki tämän metodin käytöstä on alkuosasummien laskeminen. Jos alkutilanteessa taulu on `int`-taulukko ja alkion taulu[i] arvo on a_i, niin kutsun

```
Arrays.parallelPrefix(taulu, (int a, int b) -> a+b))
```

sijoittaa kaikilla i alkion taulu[i] arvoksi $\sum_{j=0}^{i} a_j$. Tässä esitämme toisena argumenttina olevan binäärioperaattorin käyttämällä taas lambdalauseketta.

Voimme antaa argumentiksi minkä tahansa binäärioperaattorin $\oplus$, joka on liitännäinen eli toteuttaa ehdon $a \oplus (b \oplus c) = (a \oplus b) \oplus c$. Esim. maksimi $a \oplus b = \max(a, b)$ kelpaa. Suorittamalla

```
Arrays.parallelPrefix(taulu, (int a, int b) -> Math.max(a,b))
```

alkion taulu[i] arvoksi tulee $\max\{a_0, \ldots, a_i\}$.

Monimutkaisempia binäärioperaattoreita voidaan määritellä rajapinnan BinaryOperator avulla. Emme kuitenkaan käsittele tässä asiaa enempää.

Alkusumman laskemista voidaan käyttää osana hyvin monenlaisia rinnakkaisia algoritmeja.

Esimerkki Reikäkorttijärjestämisessä (radix sort) eräs vaihe on järjestellä taulukon luvut siten, että parilliset luvut tulevat ennen parittomia. Parillisten lukujen keskinäinen järjestys säilyy, samoin parittomien. Esim. taulukosta

$$[27, 8, 5, 12, 4, 17, 13, 16]$$

tulee

$$[8, 12, 4, 16, 27, 5, 17, 13].$$

Alkion $A[i]$ uusi sijainti voidaan laskea alkuosasummaa soveltamalla.

Tehdään tämä järjestämisaskel taulukolle taulu[0. . . n-1].

1. Merkitään aputaulukkoon paikka ykkönen parittomien alkioiden kohdalle:

```
Arrays.parallelSet(parittomia, i -> taulu[i]%2);
```

2. Suoritetaan

```
Arrays.parallelPrefix(parittomia, (int a, int b) -> a+b);
```

minkä jälkeen parittomia[i] kertoo parittomien lukujen määrän osataulukossa taulu[0. . . i].

Erityisesti koko taulukon parillisten lukujen määrä on n-parittomia[n-1].

3. Jos taulu[i] on parillinen, sen uusi paikka on i-parittomia[i].

Jos taulu[i] on pariton, sen uusi paikka on (n-parittomia[n-1])+parittomia[i]-1.

Miksi alkuosasummien laskeminen sitten on rinnakkaistuvaa?

Tarkastellaan esimerkkinä 8-alkioista taulukkoa $A[1 \ldots 8]$.

Taulukon koko summan $A[1] + \ldots + A[8]$ laskeminen voidaan rinnakkaistaa seuraavasti:

vaiheessa 1 lasketaan rinnakkain
$A[2] = A[1] + A[2]$, $A[4] = A[3] + A[4]$, $A[6] = A[5] + A[6]$ ja $A[8] = A[8] + A[7]$.

vaiheessa 2 lasketaan rinnakkain
$A[4] = A[4] + A[2]$ ja $A[8] = A[8] + A[6]$.

vaiheessa 3 lasketaan $A[8] = A[8] + A[4]$.

Yleisemmin n alkion summan laskemisessa vaiheita tulee $\log n$.

Kukin vaihe menee vakioajassa, jos käytössä on ainakin n prosessoria.

Jos prosessorien lukumäärä on $p < n$, voidaan jakaa taulukko ensin p osataulukkoon, joiden koko on n/p.

(Oletetaan yksinkertaisuuden vuoksi, että n ja p ovat kakkosen potensseja.)

Ensin kukin prosessori summaa oman osataulukkonsa sen kokoon verrannollisessa ajassa $O(n/p)$.

Sen jälkeen saadut p osasummaa voidaan laskea yhteen edellisen sivun menettelyllä ajassa $O(\log p)$.

Aikavaativuus koko taulukon summan laskemiselle on siis $O(n/p + \log p)$. Jos $n \gg p \log p$, niin termi $O(n/p)$ dominoi, joten p prosessoria antaa melkein nopeutuksen p.

Samalla taulukkoon tulee muita osasummia, joita sopivasti yhdistelemällä saadaan lasketuksi muutkin alkuosasumman alkiot.

Kaikkien osasummien laskeminen taulukossa $A[1 \ldots n]$ on helpointa esittää rekursiivisesti. (Oletetaan taas, että n on kakkosen potenssi.)

1. Jos $n = 1$, palaa aliohjelmasta tekemättä mitään.
2. Muodosta aputaulukko $B[1 \ldots n/2]$.
3. Laske rinnakkain $B[i] = A[2i] + A[2i-1]$ kun $i = 1, \ldots, n/2$.
4. Laske rekursiivisesti taulukon B alkusummat.
5. Laske rinnakkain $A[2i] = B[i]$ ja $A[2i-1] = B[i-1] + A[2i-1]$ kun $i = 1, \ldots, n/2$.

Aikavaativuudeksi nähdään $O(n/p + \log p)$ samaan tapaan kuin edellä.

Lisää alkuosasumman laskemisesta ja soveltamisesta löytyy esim. Guy E. Blellochin artikkelista `https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf`.

Esimerkki: Primin algoritmi

Esimerkkinä rinnakkaisuuden soveltamisessa osana monimutkaisempaa algoritmia tarkastelemme Primin algoritmia pienimmälle virittävälle puulle tiheässä verkossa.

Lähdetää liikkeelle sarjallista algoritmista (s. 568):

```
Prim-Dense(G,w,r)
1   S = {r}
2   distance[r] = 0
3   for kaikille solmuille v ∈ V \ {r}
4       distance[v] = w(r,v) // ääretön, jos kaari puuttuu
5       parent[v] = r
6   while S ≠ V
7       valitse u ∈ V \ S, jolla distance[u] on pienin
8       T = T ∪ {(parent[u],u)}
9       for jokaiselle solmulle v ∈ vierus[u]
10          if v ∉ S and w(u,v) < distance[v]
11              parent[v] = u
12              distance[v] = w(u,v)
13  return T
```

Kuten muistetaan, sarjallisen algoritmin aikavaativuus on $O(n^2)$, missä $n = |V|$ on solmujen lukumäärä:

- rivit 1–2 menevät vakioajassa
- rivin 3 for-silmukka suoritetaan n kertaa, ja silmukan runko riveillä 4–5 vie vakioajan
- rivin 6 while-silmukka suoritetaan $n - 1$ kertaa
  - rivillä 7 pienin alkio etsitään ajassa n
  - rivin 9 silmukka suoritetaan n kertaa, ja runko vie vakioajan.

Aikavaativuutta siis dominoivat

- pienimmän alkion etsiminen rivillä 7 ja
- sisempi silmukka rivillä 9.

Oletetaan nyt, että käytettävissä on p suoritinta, missä $p \leq n$. Oletetaan yksinkertaisuuden vuoksi, että n/p on kokonaisluku.

Pienimmän alkion etsiminen taulukosta distance voidaan rinnakkaistaa seuraavasti:

1. Jaetaan taulukko distance p osataulukkoon, joiden koko on n/p.

   Varataan aputaulukko apu$[1 \ldots p]$.

   Suoritin numero i saa vastuulleen osataulukon numero i, eli distance$[(i-1)n/p + 1 \ldots in/p]$, sekä alkion apu$[i]$.

2. Kukin suoritin i toisistaan riippumatta etsii oma osataulukkonsa pienimmän alkion ja sijoittaa sen paikkaan apu$[i]$. Tämä vie ajan $O(n/p)$.

3. Lasketaan arvoksi apu$[p]$ koko taulukon apu$[1 \ldots p]$ pienin arvo. Tämä tapahtuu ajassa $O(\log p)$ samaan tapaan kuin alkuosasummien laskemisessa. Pidetään samalla kirjaa, millä indeksillä u minimi löytyi.

4. Välitetään globaali minimi apu$[p]$ ja indeksi u kaikille suorittimille. Tämä tapahtuu ajassa $O(\log p)$:
   - alussa vain suoritin numero p tietää arvot
   - laskenta-askelessa k tieto on jo levinnyt 2^k suorittimelle, ja kukin niistä kertoo sen yhdelle uudelle suorittimelle.

Rivien 9–12 silmukka voidaan samoin suorittaa ajassa $O(n/p)$ yksinkertaisesti antamalla kunkin suorittimen vastuulle n/p solmua.

Kun muistetaan, että edellä analysoidut osat suoritetaan $n-1$ kertaa, saadaan kokonaisaikavaativuudeksi $O(n^2/p + n \log p)$.

Jos $p \log p \ll n$, termi n^2/p dominoi, ja saavutamme jälleen lähes optimaalisen nopeutuksen p.

Huom. Tällaisiin O-arvioihin on syytä suhtautua varauksella:

- Käytännössä p on usein melko pieni. Tällöin O-merkinnän piilottamat vakiokertoimet tulevat merkitseviksi. Yleensä rinnakkaisen ratkaisun vakiokertoimet ovat suuremmat.
- Suurilla p suorittimien välinen kommunikointi voi muodostua pullonkaulaksi. Tämä pätee etenkin, jos laskenta hajautetaan usealle eri tietokoneelle. Suoritusaika saattaa kääntyä jopa nousuun, jos suorittimia lisätään liikaa.

Yhteenveto

- Tehokkaassa ohjelmoinnissa pitää algoritmien teoreettisen aikavaativuuden lisäksi ymmärtää, millaisessa ympäristössä ohjelma suoritetaan.
- Nykyaikaisissa laskentaympäristöissä rinnakkaisuus on erittäin tärkeä tapa lisätä laskentatehoa, ja sen merkitys ei ainakaan vähene lähitulevaisuudessa.
- Jos tietää perusoperaatioita, jotka rinnakkaistuvat hyvin, niitä voi käyttää osana ohjelmia murehtimatta rinnakkaisuuden toteutuksesta.
- Rinnakkaisuutta ja samanaikaisuutta käsitellään enemmän muilla kursseilla (esim. *Tietokoneen toiminta* ja *Käyttöjärjestelmät*).