S-114.240 Rinnakkaislaskenta laskennallisessa tieteessä: Matriisilaskenta

Mika Prunnila
S4
44608T

Sisältö

1. Johdanto **2**

2. Rinnakkais ohjelmointi ja rinnakkaiskoneet **2**

3. Matriisien ja vektoreiden lohkominen rinnakkaislaskennassa **3**

4. Valikoituja rinnakkaistettuja matriisi operaatiota **4**

4.1 Hajautetun muistin järjestelmä.. 4

4.1.1 Matriisi-vektori tulo.. 5

4.1.2 Matriisi tulo.. 6

4.2 Yhteisen keskusmuistin järjestelmä.. 8

4.2.1 Matriisi-vektori tulo.. 8

4.2.2 Matriisi tulo.. 9

5. Matriisi kirjastorutiineista **10**

5.2 ScaLAPACK..10

5.3 PBLAS testiajoja..10

6. Yhteenveto **12**

Viitteet **13**

1. Johdanto

Moni tieteellinen ongelma on joko suoranaisesti matriisi ongelma tai se voidaan palauttaa tällaiseksi esimerkiksi diskretoinnilla. Tästä johtuen matriisilasku algoritmit ovatkin selvästi erityisasemassa tietotekniikassa ja ne ovat yhä jatkuvan kehittämisen kohteena.

Matriisi probleemoissa alkioiden määrä nousee helposti yli miljoonien, ja jotta laskenta voitaisiin suorittaa siedettävässä ajassa saatikka tutkia ratkaisujen erilaisia parametri riippuvuuksia, on järkevää turvautua rinnakkaislaskentaan. Tämä tuo huomattavia lisähaasteita itse mallintajalle: on huomioitava esim. latenssit sekä kuorman tasaus, jolloin rinnakkaistamisesta saadaan suurin mahdollinen hyöty.

Tässä esitelmässä käydään läpi muutamia rinnkkaistetun matriisi laskun perusasioita. Teksti on pääosin referoitu Gene H. Golubin ja Chrles F. Van Loanin kirjan Matrix Computations luvusta 6. Myös muita lähteitä on käytetty, mutta näihin ei itse tekstissä muutamaa poikkeusta lukuun ottamatta viitata. Kappale kohtaiset viitteet on kuitenkin annettu tekstin lopussa.

(Golub ja Loan käyttävät kirjassaan hyväksi *matlab*in notaatioita vektoreiden ja matriisien määrittelyissä sekä algoritmeissa, ja näitä notaatioita hyödynnetään myös tässä esityksessä.Tekstin seuraaminen edellyttää siis perus *matlab* osaamista)

2. Rinnakkais ohjelmointi ja rinnakkaiskoneet

Rinnakkaislaskennassa ohjelmointimallit voidaan jakaa kolmeen ryhmään :

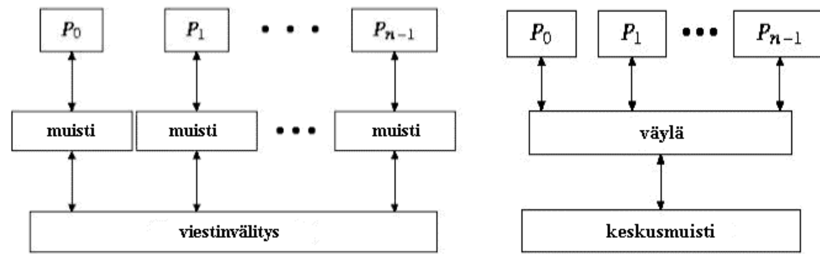
- jaetun keskusmuistin ohjelmointimalli
- viestin välitykseen perustuva ohjelmointi
- datarinnakkainen ohjelmointi.

Tässä työssä käsitellään lähinnä kahta ensiksi mainittua mallia.

Ohjelmointi mallien lisäksi myös rinnakkaistietokoneet voidaan jakaa eri tyyppeihin (kts kuva 2.1):

- SMP-*tietokone* (symmetric multiprocessor): prosessoreilla yhteinen keskus muisti
- MPP-*tietokone* (massively parallel processor): prosessoreilla on omat hajautetut keskusmuistit

Rinnakkaiskoneen tyypin ei kuitenkaan välttämättä tarvitse vastata käytettyä ohjelmointimallia.



Kuva 2.1 . Rinnakkaislaskennan arkkitehtuureja

Jaetun muistin järjestelmissä prosessorit kommunikoivat lukemalla ja kirjoittamalla globaaleja muuttujia, jotka sijaitsevat yhteisessä (globaalissa) keskusmuistissa. Jokainen prosessori kuitenkin suorittaa omaa lokaalia ohjelmaansa, joka puolestaan sijaitsee kyseisen prosessorin lokaalissa muistissa.

Hajautetun muistin järjestelmissä prosessorien välinen kommunikointi on järjestetty prosessorien välisellä viestinnällä.

3. Matriisien ja vektoreiden lohkominen rinnakkaislaskennassa

Rinnakkaisalgoritmien kannalta on oleellista miten data jaetaan prosessorieiden lokaalien muistien kesken. Datan sijoittelu on erityisen tärkeää kuorman tasauksen kannalta.

Tarkastellaan ensin vektoreita. Oletetaan, että $x \in \Re^n$ ja että prosessorien määrälle p pätee $n = pr$. x voidaan jakaa prosessoreille riveittäin tai sarakkeittain:

- *Sarake* jaossa x ajatellaan $r \times p$ matriisina,

$$x_{r\times p} = [x(1:r) \quad x(r+1:2r) \quad \cdots \quad x(1+(p-1)r:n)],$$

ja jokainen sarake annetaan eri prosessorille.

- *Rivi* jaossa x ajatellaan puolestaan $p \times r$ matriisina,

$$x_{p\times r} = [x(1:p) \quad x(p+1:2p) \quad \cdots \quad x((r-1)p+1:n)].$$

Nyt x jaetaan riveittäin prosessoreille.

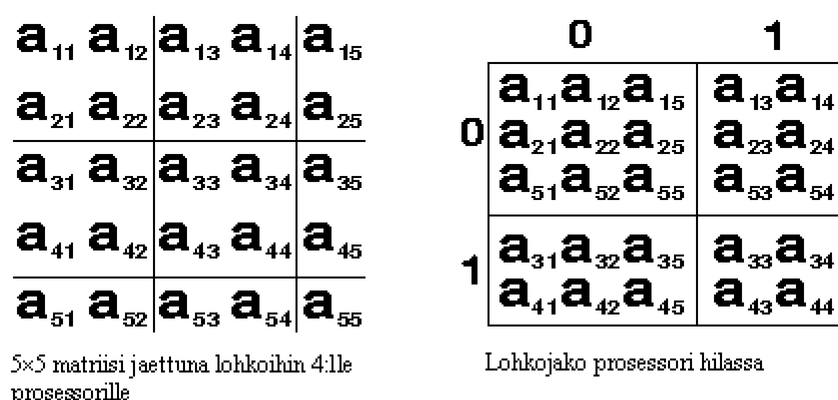
Vastaavasti nämä jaetaan tylillisesti jatkuvaan ja jaettuun (wrap).

Jos n ei ole p:n moninkerta eli, $n = pr + q \ (0 \le q < p)$,niin prosessoreista $1 \ldots q$ tallettaa jokainen $r+1$ alkiota ja prosessorit $q+1 \ldots p$ tallettavat r alkiota.

Matriisit voidaan tallettaa vastaavalla tavalla kuin vektorit. Taulukossa 1. löytyy neljä tapaa tallettaa neliö matriisi $A \in \Re^{n\times n}$, ja näille jokaiselle löytyy analoginen lohko tallennus. Matriiseille löytyy myös ns. 2D syklinen lohko tallennus (kuva 3.1), jota käyttää esim. ScaLAPACK kirjasto. Tämä jako tapa takaa erittäin hyvän skaalautuvuuden ja kuormantasauksen.

Suunta	Tyyli	Prosessorin μ data
Sarake	Jatkuva	$A(:, 1+(\mu-1)r:\mu r)$
Sarake	Jaettu	$A(:,\mu:p:n)$
Rivi	Jatkuva	$A(1+(\mu-1)r:\mu r, :)$
Rivi	Jaettu	$A(\mu:p:n,:)$

Taulukko 1. Erilaisia neliö matriisin $A \in \Re^{n \times n}$ jako tapoja.



Kuva 3.1 . Syklinen 2D lohkojako 5×5 matriisille 2×2 prosessori hilassa

4.Valikoituja matriisi operaatioita rinnakkaislaskennassa

4.1 Hajautetun muistin järjestelmä

Hajautetun muistin järjestelmissä voi olla jopa useita tuhansia prosessoreita, jolloin prosessoreiden välisestä kommunikaatiosta aiheutuvat viiveet ovat laskenta tehoa rajoittava tekiä. Tämän takia algoritmit pyritäänkin toteuttamaan siten, että vain lähi naapurit kommunikoivat keskenään.

Viestien välitykseen kuluvaa aikaa τ voidaan kuvata yksinkertaisella mallilla, jossa viestit lähetetään **send**({matriisi}, {vastaanottan prosessorin numero}) funktiolla ja vastaanotetaan **recv**({matriisi}, {lähettävän prosessorin numero}) funktiolla. Jos edellä mainittujen funktioden alustus vie ajan t_s ja yhden liukuluvun siirto ajan t_d , kuluu viestiin joka sisältää m liukulukua aika

$$\tau(m) = t_s + t_d m. \qquad (4.1)$$

Seuraavaksi esitellään viestinvälitykseen perustuvat matriisi ja matriisi-vektori tulo. Viesteihin kuluvaa aikaa arvioidaan käyttämällä kaavaa (4.1).

4.1.1 Matriisi-vektori tulo

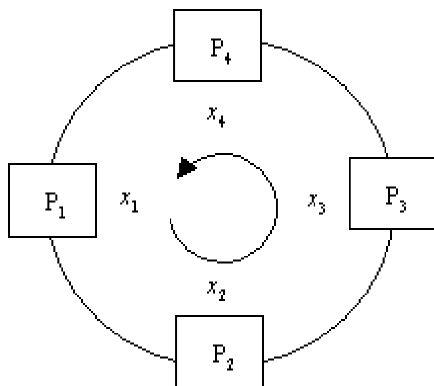
Tarkastellaan matriisioperaatiota $z = y + Ax$ ($A \in \Re^{n\times n}$, $x,y,z \in \Re^{n}$) rengasmaisessa prosessori geometriassa. Oletetaan yksinkertaisuuden vuoksi, että $n = rp$, ja jaetaan ongelma lohko muotoon

$$\begin{bmatrix} z_1 \\ \vdots \\ z_p \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_p \end{bmatrix} + \begin{bmatrix} A_{11} & \cdots & A_{1p} \\ \vdots & \ddots & \vdots \\ A_{p1} & \ldots & A_{pp} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix}, \tag{4.2}$$

missä $A_{ij} \in \Re^{r\times r}$ ja x_i, y_i, $z_i \in \Re^{r}$. Oletetaan vielä, että x_μ, y_μ, z_μ ja A:n μ:s lohko rivi sijaitsevat μ:nnen prosessorin lokaalissa muistissa. Nyt μ:nnen prosessorin tulee tehdä laskutoimitus

$$z_\mu = y_\mu + \sum_{\tau=1}^{p} A_{\mu\tau} x_\tau ,$$

joka sisältää muutakin kuin lokaalia dataa. Jotta ei-lokaali data saadaan kunkin prosessorin ulottuville, kierrätetään lohko vektoreita x_i renkaan ympäri (kuva 4.1).



Kuva 4.1 . Lohko vektoreiden x_i syklinen kierto rengas geometriassa.

Oletetaan nyt, että prosessorissa μ on olemassa lokaalit alustukset:
p, μ, *left* ja *right (lähinaapurit), n,* $row=1 + (\mu-1)r:\mu r$, $A_{loc} = A(row,:)$,
$x_{loc} = x(row)$, $y_{loc} = y(row)$, jolloin algoritmia voidaan kuvata pseudo koodilla:

```
for t = 1:p
        send(x_loc, right)
        recv(x_loc , left)
        τ = μ - t
        if τ ≤ 0
                τ = τ + p
        end
        y_loc = y_loc + A_loc(:, 1 + (τ - 1)r: τr) x_loc
end
```

Jos laskenta etenee nopeudella R liukulukuoperaatiota/s (flops/s), saadaan k:n prosessorin vaatimalle laskenta-ajalle $T(k)$ arvio

$$T(k) = \frac{2n^2}{Rk} + 2t_s k + 2t_d n\,, \quad (k>1).$$

Jos $k=1$, on kommunikaatio tarpeetonta ja $T(k) = 2n^2/R$.

Matriisin A ollessa esimerkiksi alakolmio matriisi tulee edellä kuvattuun algoritmiin paljon nollalla kertomisia ja summaamisia. Nämä turhat laskutoimitukset poistuvat, jos y_{loc} päivitetään seuraavasti,

if $\tau \le \mu$

$y_{loc} = y_{loc} + A_{loc}(:, 1 + (\tau - 1)r: \tau r)\, x_{loc}$

end.

Nyt laskutoimitusten määrä puolittuu, mutta kuorma jakautuu epätasaisesti prosessorien kesken: prosessori μ joutuu tekemään noin $\mu r^2/2$ laskutoimitusta.

Kuormantasaus ongelma voidaan ratkaista jakamalla matriisi riveittäin prosessoreille, eli käyttämällä taulukon 1. tapaa 4. (lohko muodossa) datan talletukseen. Vieläkin laskutoimitusten määrä kasvaa μ:n kasvaessa, mutta alkuperäiseen tilanteeseen verrattuna kuorma jakautuu huomattavasti tasaisemmin.Yleinen algoritmi voidaan toteuttaa indeksi manipulaatiolla, mutta edellä kuvattu algoritmi säilyy hyvin saman kaltaisena.

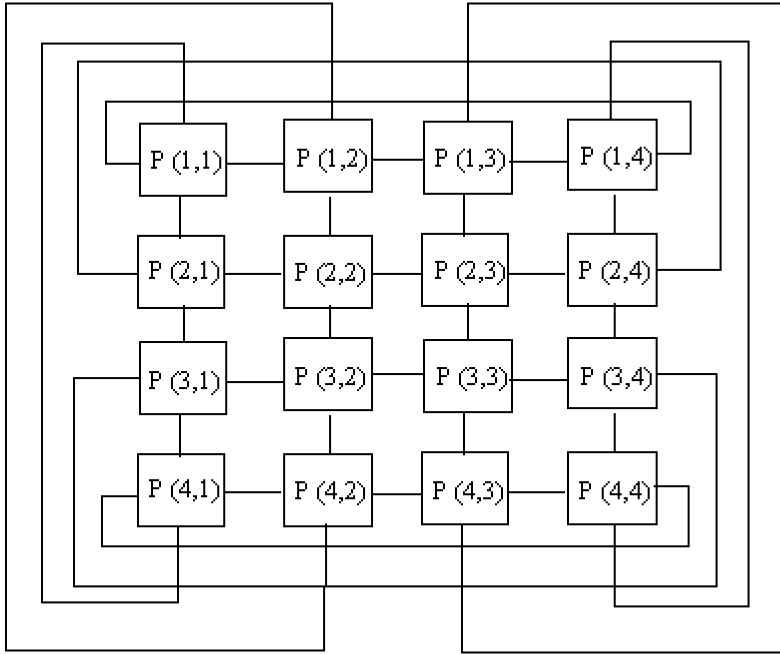
4.1.2 Matriisi tulo

Valitaan prosessori geometriaksi 2D torus (kts. kuva 4.2), ja tarkastellaan matriisi operaatiota $D = C + AB$, missä $A, B, C \in \Re^{n \times n}$. Oletetaan, että toruksessa on $p_1 \times p_1$ prosessoria ja että $n = r\, p_1$. Tällöin matriisit voidaan jakaa $r \times r$ blokkeihin, eli toruksen prosesori (i,j) tallettaa lokaaliin muistiinsa lohko matriisit A_{ij}, B_{ij}, C_{ij} ja D_{ij} ($A_{ij}, C_{ij}, D_{ij} \in \Re^{r \times r}$). Nyt prosessorin (i,j) tehtävänä on kirjoittaa yli C_{ij} laskutoimituksella

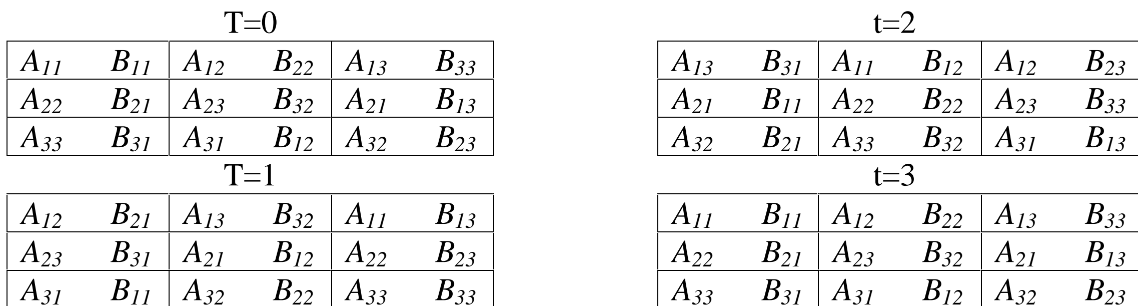
$$D_{ij} = C_{ij} + \sum_{k=1}^{p_1} A_{ik} B_{kj}\,. \qquad (4.3)$$

Kuten matriisi vektori tulon tapauksessa tässäkin operaatiossa tarvitaan paljon ei-lokaalia dataa, ja prosessoreiden kommunikaatiosta on jälleen huolehdittva. Lohko matriisien siirrot voidaan tehdä liu-uttamalla matriiseja A ja B toruksessa syklisesti siten, että A:n alkiot liikkuvat länteen ja B:n pohjoiseen. Tällöin ideana on, että jokainen prosessori saa laskettua yhtälössä (4.3) esiintyvän summan niin, että sen muistissa on aina vain kumuloituva tulos D_{ij} sekä kaksi lohkomatriisia A_{ik} *ja* B_{kj} . Kun tulo A_{ik} B_{kj} on laskettu ja summattu lähetetään matriisit eteenpäin (A_{ik} länteen, B_{kj} etelään) ja vastaan otetaan uudet lohko matriisit (A_{im} idästä , B_{mj} etelästä). Tätä kutsutaan Cannonin algoritmiksi.

Algoritmin toteutus vaati kuitenkin lohko matriisien syklistä uudelleen sijoittelua toruksessa: A:n n:ttä riviä on siirrettävä n-1 prosessoria itään ja B:n n:ttä saraketta siirretään n-1 prosessoria etelään. Kuvassa 4.3 on esitetty algoritmin toiminta tapauksessa $p_1 = 3$. Kuva sarjasta voidaan todeta, että prosessorin (i,j) kohtaavat vain lohkot A_{ik} ja B_{kj}, $k = 1,2,3$.



Kuva 4.2 . *4×4* 2D torus.



T=0		
A_{11} B_{11}	A_{12} B_{22}	A_{13} B_{33}
A_{22} B_{21}	A_{23} B_{32}	A_{21} B_{13}
A_{33} B_{31}	A_{31} B_{12}	A_{32} B_{23}

t=2		
A_{13} B_{31}	A_{11} B_{12}	A_{12} B_{23}
A_{21} B_{11}	A_{22} B_{22}	A_{23} B_{33}
A_{32} B_{21}	A_{33} B_{32}	A_{31} B_{13}

T=1		
A_{12} B_{21}	A_{13} B_{32}	A_{11} B_{13}
A_{23} B_{31}	A_{21} B_{12}	A_{22} B_{23}
A_{31} B_{11}	A_{32} B_{22}	A_{33} B_{33}

t=3		
A_{11} B_{11}	A_{12} B_{22}	A_{13} B_{33}
A_{22} B_{21}	A_{23} B_{32}	A_{21} B_{13}
A_{33} B_{31}	A_{31} B_{12}	A_{32} B_{23}

Kuva 4.3 . Lohkojen kulku 3×3 toruksessa.

Jos oletetetaan, että laskennan alkaessa prosessori *(i,j)*:llä on muistissaan kerrottavien matriisien lohkot *(i,j)* saadaan laskentaan kuluvaksi kokonaisajaksi *k*:lla prosessorilla (laskenta nopeus *R* flops/s)

$$T(k) = 2(k-1)(t_s + (n^2/k)t_d) + 2n^3/(Rk).$$

Laskennan jälkeen lohko matriisit eivät ole oikeilla prosessoreilla toruksessa, vaan on tehtävä alustuksena tehdyt sykliset siirrot takaperoisessa järjestyksessä. Tämä vie ajan $(k-1)(t_s + (n^2/k)t_d)$, joka on lisättävä aikaan $T(k)$.

Algoritmia voidaan parantaa huomattavasti käyttämällä putkitettua broadcast viestintää. Näin on tehty esimerkiksi SUMMMA (Scalable Universal Matrix Multiplication Algorithm) algoritmissa, jota on käytetty esim. joissain matriisi aliohjelma kirjastoissa.

4.2 Yhteisen keskusmuistin järjestelmä

Yhteisen keskusmuistin rinnakkaisarkkitehtuureissa väylän kapasiteetti ja prosessorien lokaali muisti yhdessä rajoittavat laskennan tehokkuutta, ja väylän nopeus asettaa ylärajan prosessorien määrälle. Jos ohjelmaa ajetaan MPP-tietokoneessa, rajoitukset ovat luonnollisesti erilaisia.

Datan siirto lokaalien ja globaalin muistin välillä sekä globaalien muuttujien päivitykset täytyy tehdä huolellisesti. Nämä toimenpiteet täytyy ajastaa dynaamisesti, sillä muuten saattaa käydä niin, että esim. jotkut globaalien muuttujien päitykset pyyhkiytyvät yli.

Tutkitaan jälleen perus matriisioperaatioita, ja havainnollistetaan globaalien muutujien päivitystä sekä kuormantasausta.

4.2.2 Matriisi vektori tulo

Oletetaan edelleen, että $n = rp$, ja muutetaan operaatio $z = y + Ax$ lohko muotoon:

$$\begin{bmatrix} z_1 \\ \vdots \\ z_p \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_p \end{bmatrix} + \begin{bmatrix} A_1 \\ \vdots \\ A_p \end{bmatrix} x, \qquad (4.4)$$

missä $A_\mu \in \Re^{r\times n}$ ja y_μ, $z_\mu \in \Re^r$ sekä $x \in \Re^n$. Nyt A, x ja z sijaitsevat globaalissa muistissa, jota kaikki prosessorit voivat lukea ja kirjoittaa.

Nyt laskenta voidaan toteuttaa yksinkertaisesti siten, että prosessori μ ensin lukee y_μ:n , A_μ:n ja x:n lokaaliin muistiinsa ja tämän jälkeen laskee kaavan (4.4) rivin μ ja päivittää tuloksen yhteiseen muistiin. Algoritmin nopeuteen vaikuttaa oleellisesti se miten A_μ luetaan yhteisestä keskusmuistista.

Ongelman voi jakaa lohkoihin myös toisella tavalla

$$z = y + \begin{bmatrix} A_1 & \cdots & A_p \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix}, \qquad (4.5)$$

missä $A_\mu \in \Re^{n\times r}$, $x_\mu \in \Re^r$ ja $y,z \in \Re^n$. Laskenta voidaan toteuttaa esimerkiksi käyttämällä globaalia taulukkoa $W \in \Re^{n\times p}$, jonka pysty vektoreihin talletetaan lohko tulot $A_\mu x_\mu$. Kun kaikki lohko tulot on laskettu, voidaan laittaa esim. yksi prosessori laskemaan W:n sarakkeet yhteen. Toinen vaihtoehto on antaa kunkin prosessorin huolehtia vektori summan päivityksestä. Jos nyt algoritmi kirjoitetaan niin, että z kirjoitetaan y:n päälle, voi joidenkin prosessorien kontribuutio vektori summaan kirjoittua yli. Tästä ongelmasta päästään, jos prosessori μ suorittaa laskennan esimerkiksi seuraavasti:

$r = n/p; col = 1 + (\mu\text{-}1)r: \mu r; A_{loc} = A(:, col); x_{loc} = x(col);$
$w_{loc} = A_{loc}\, x_{loc}$

$y_{loc} = y$
$y_{loc} = y_{loc} + w_{loc}$
$y = y_{loc}$
----

missä katkoviivoin rajoitetun osan voi suorittaa vain yksi prosessori kerrallaan. Jos näin ei menetellä, voi ohjelmassa tapahtua esimerkiksi suoritus

Prosessori 1 lukee y:n
Prosessori 2 lukee y:n
Prosessori 1 päivittää y:n
Prosessori 2 päivittää y:n,

jossa prosessori *1*:n kontribuutio vektorisummaan häviää.

4.2.2 Matriisi tulo

Olkoon laskettavana $D = C + AB$ kohdan 4.1.2 tapaan sillä poikkeuksella, että B on yläkolmio matriisi. Jaetaan matriisit jälleen lohko muotoon p:n prosessorin kesken:

$$[D_1, \cdots, D_{k\cdot p}] = [C_1, \cdots, C_{k\cdot p}] + [A_1, \cdots, A_{k\cdot p}] \cdot [B_1, \cdots, B_{k\cdot p}] \tag{4.6}$$

missä jokaisen lohkon koko on $r = n/(kp)$. Merkitään

$$B_j = \begin{bmatrix} B_{1j} \\ \vdots \\ B_{jj} \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \qquad B_{ij} \in \Re^{r\times r},$$

jolloin D_j voidaan laskea seuraavasti:

$$D_j = C_j + \sum_{\tau=1}^{j} A_{ik} B_{\tau j} \,. \tag{4.7}$$

D_j:n laskemiseen tarvittavien liukulukuoperaatioiden määrä on

$$f_j = 2nr^j = \left(\frac{2n^3}{k^2 p^2} \right) j \,,$$

joka kasvaa j:n funktiona. Kuorma siis jakautuu epätasaisesti. Kohdassa 4.2.1 saatiin kuormantasaus aikaan käyttämällä jaettua datan sijoittelua prosessoreille, ja tässä

voidaan menetellä vastaavalla tavalla. Asetetaan prosessori μ siis laskemaan D_j j:n arvoilla $j = \mu{:}p{:}kp$, jolloin sen tarvitsee tehdä

$$F(\mu) = \sum_{i=1}^{k} f_{\mu+(i-1)p} \approx \left(k\mu + \frac{k^2 p}{2} \right) \frac{2n^3}{k^2 p^2}$$

liukuluoperaatiota. Kuorman tasauksen mittana voidaan käyttää suhdetta $F(p)/F(1)$. Koska

$$\frac{F(\mu)}{F(1)} = 1 + \frac{2(p-1)}{2+kp},$$

laskenta työ jakaantuu sitä tasaisemmin mitä suurempi k on. On kuitenkin huomioitava , että globaalin muistin ja prosessorien välinen kommunikaatio kasvaa k:n kasvaessa. Eli latenssien ollessa suuria suuren k:n valinta saattaa lopulta hidastaa laskentaa.

5. Matriisi kirjastorutiineista

5.1 ScaLAPACK

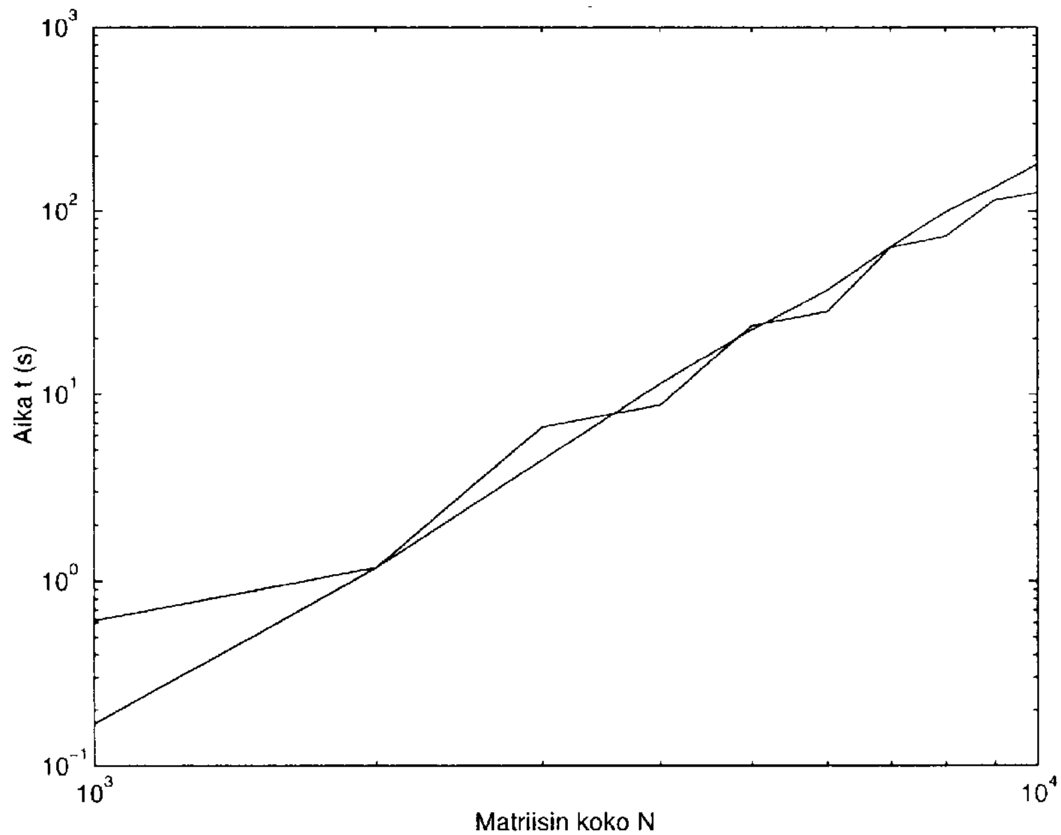
LAPACK (Linear Algebra PACKage) kirjastolle löytyy rinnakkaistettu versio ScaLAPACK (Scalable LAPACK), joka on suunniteltu hajautetun muistin MIMD tietokoneille. Se on kirjoitettu SPMD-ohjelmointityylillä (Single Program, Multiple Data), ja kuten jo aiemmin mainittiin ScaLAPACK:issa oletetaan, että matriisit ovat jaettu sykliseen 2-dimensionaaliseen lohko muotoon (kts kuva 3.1).

ScaLAPACKin rakennuspalikoita ovat BLAS kirjaston (Basic Linear Algebra Subprograms) rinnakkaistetun version PBLAS (Parallel BLAS) funktiot. Prosessorien välisestä viestien välityksestä huolehditaan BLACS funktioiden (Basic Linear Algebra Communication Subprograms) avulla. ScaLAPACK on pyritty rakentamaan siten, että se muistuttaa käyttäjän kannalta mahdollisimman paljon LAPACK:ia.

5.2 PBLAS testiajoja

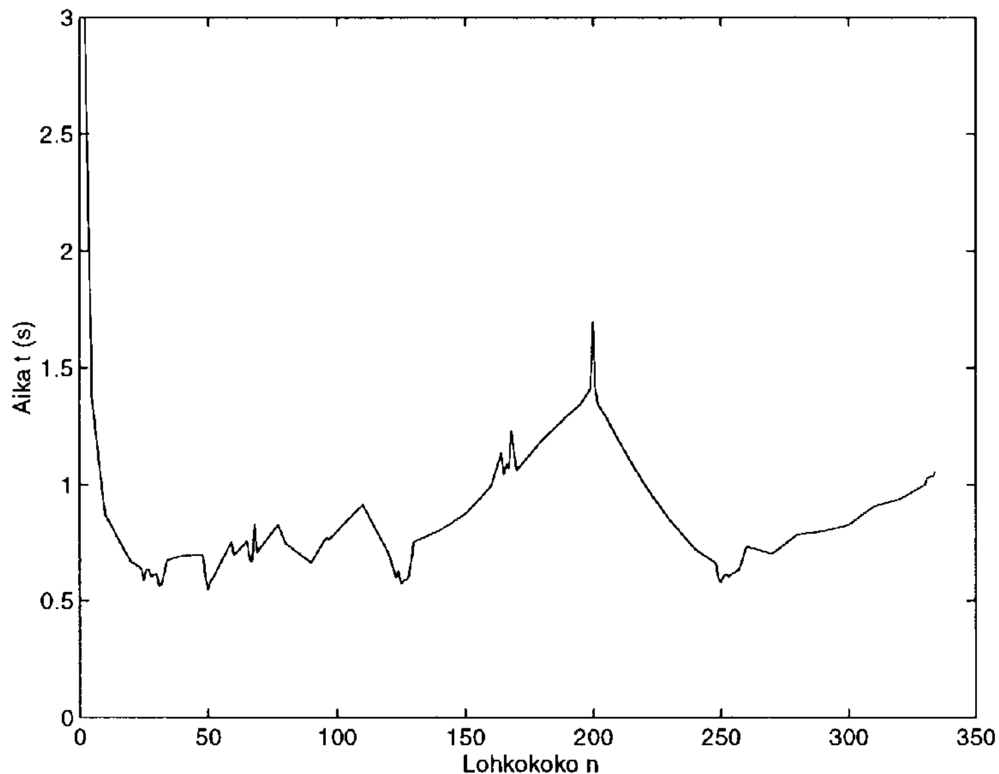
PGEMM on PBLAS-rutiini, joka suorittaa matriisi kertolaskun käyttämällä SUMMAa vastaavaa algoritmia. Seuraavaksi tarkastellaan CSC:n 3TE:llä tälle tehtyjen testiajojen tuloksia (Yrjö Leino, @CSC 3/97).

Ensimmäisessä testisarjassa tutkitaan matriisin koon vaikutusta laskenta-aikaan, ja prosessorien lukumäärä ($8{\times}8$ hila eli 64 prosessoria) pidetään vakiona. Matriisilohkolle on käytetty kahta arvoa: $n=25$ ja $n=250$. Kuvassa 5.1 on esitetty ajon tulokset. Isolla lohkokoolla laskenta-aika oskilloi hieman, kun taas pienellä lohkolla kasvu on lineaarista. Oskillaatio johtuu yksinkertaisesti siitä, että lohkot eivät jakaannu tasaisesti hilaan, jos matriisin koko ei ole jaollinen luvulla 2000. Pienimmällä matriisin koolla $N = 1000$ vain $4{\times}4$ hila prosessoreista on työllistettynä $n=250$ jaolla, mistä aiheutuu kuvaajassa nähtävä melko suuri ero $n=25$ jakoon verrattuna.



Kuva 5.1 . Laskenta-aika matriisin koon funktiona

Toisessa testisarjassa matriisin koko (*N=1000*) ja prosessorimäärä (*4×4*) pidetään vakiona, jolloin päästään tutkimaan kuorman tasauksen vaikutusta laskenta-aikaan piirtämällä laskenta-aika lohkokoon funktiona. Testin tulos on esitetty kuvassa 5.2. Kuvaajan minimit löytyvät kohdista, joissa kuorma jakaantuu mahdollisimman tasaisesti hilaan. Näin käy, jos *1000/4n* on jokin kokonaisluku, eli jos *n = 250, 125, 50, 25, 10, 5, 2, 1*. Näistä neljä ensimäistä voidaan paikantaa kuvasta 5.2, mutta kun *n< 25* alkaa laskenta-aikaa hallitsemaan prosessorien välinen viestintä. Voimakas piikki arvolla *n = 200* aiheutuu kuorman epätasauksesta: *N/n = 1000/200 = 5* ja hilassa on *4×4* prosessoria, eli hilan ensimmäiselle prosessorille tulee 4 lohkoa, ensimmäisen rivin ja sarakkeen muille prosessoreille 2 lohkoa ja 9 prosessorille vain 1 lohko. Kun *n* lähestyy arvoa 333 tekee työn oleellisesti vain 9 prosessoria, mistä aiheutuu loiva maksimi käyrän lopussa.



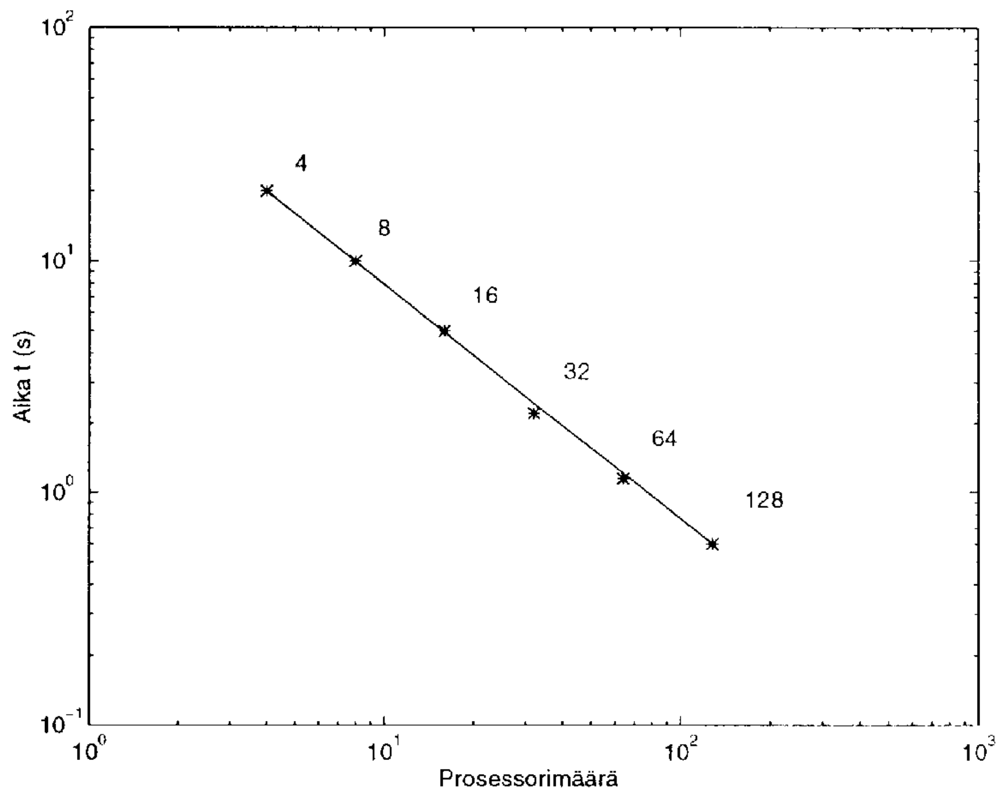
Kuva 5.2 . Laskenta-aika lohkokoon funktiona.

Viimeisessä testisarjassa tutkitaan skaalautuvuutta. Nyt matriisin koko pidetään vakiona, ja prosessori määrää kasvatetaan. Kuvassa 5.3 olevan sovitetun suoran kulma kerroin on hyvin lähellä arvoa –1. PGEMM skaalautuu siis lähes ideaalisesti.

6. Yhteenveto

Matriisi laskut ovat peräkkäisen ohjelmoinnin ”oppikirja tapauksia” ja sama tuntuu pätevän myös rinnakkaisohjelmoinnissa. Suurten ongelmien tehokas ratkaisu edellyttää kuitenkin monien asioiden, kuten viestinvälityksen ja kuormantasauksen, huolellista tarkastelua. Etenkin tiheiden matriisien tapauksessa tarvitaan paljon viestintää, ja laskenta saattaa muuttua tehottomaksi ellei viestintä kustannuksia optimoida.

Koska on jo olemassa tehokkaita matriisi aliohjelmakirjastoja, on kyseenalaista kannattaako itse painiskellä näiden ongelmien kanssa, etenkään jos matriisi laskut ovat pelkkä mallinnuksen työkalu. On kuitenkin syytä olla perillä perusperiaatteista ja rajoittavista tekijöistä, jotta pystyisi mahdollisimman hyvin hyödyntämään näitä kirjastoja.



Kuva 5.3 . Laskenta-aika prosessorimäärän funktiona

Viitteet

Luku 2. Juha Haataja ja Kaj Mustikkamäki, *Rinnakkaisohjelmointi MPI:llä (Yliopistopaino, Helsinki, 1997).* (luku 1)

Luvut 3-4. Gene H. Golubin ja Chrles F. Van Loanin, *Matrix Computations (The John Hopkins University Press ,London, 1996).* (luku 6)

Barry Wilkinson, Michael Allen, *Parallel Programming : Techniques and Applications Using Networked Workstations and Parallel Computers, Prentice Hall, 1998.* (luku 10)

Robert van de Geijn and Jerrell Watts, ‘‘SUMMA: Scalable Universal Matrix Multiplication Algorithm,” Department of Computer Sciences, The Unversity of Texas, TR-95-13, April 1995. Also: LAPACK Working Note #96 , May 1995.

Luku 5. ScaLAPACK Home Page: http://www.netlib.org/scalapack/scalapack_home.html

Yrjö Leino. ”Huimia matriisikertolaskuja T3E:llä,” @CSC 3/97, s. 31