Marko Moilanen

Matriisikertolaskun kustannusmalli hajautetun ja jaetun muistin rinnakkaistietokoneille

Tietotekniikan
pro gradu -tutkielma
12. huhtikuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Kokkolan yliopistokeskus Chydenius

Tekijä: Marko Moilanen
Yhteystiedot: marko.j.s.moilanen@student.jyu.fi
Puhelinnumero: 040 702 8452
Ohjaaja: Risto Honkanen
Työn nimi: Matriisikertolaskun kustannusmalli hajautetun ja jaetun muistin rinnakkaistietokoneille
Title in English: Cost model for matrix multiplication in distributed and shared memory parallel computers
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 78+8
Tiivistelmä: Merkittävä rinnakkaisohjelman suorituskykyyn vaikuttava tekijä on laskennan ja viestinnän välinen tasapaino. Tutkielmassa kehitettiin rinnakkaisalgoritmi ja sitä vastaava kustannusmalli matriisikertolaskulle $C = AB$, missä A, B ja C ovat $n \times n$-matriiseja. Algoritmi ja kustannusmalli kehitettiin sekä hajautetun että jaetun muistin rinnakkaistietokoneille. Kustannusmalli esitettiin kustannusfunktiona.

Tutkielmassa sovellettiin konstruktiivista tutkimusotetta. Ensin perehdyttiin rinnakkaistietokoneisiin, rinnakkaislaskennan teoreettisiin malleihin, kahteen rinnakkaislaskennan ohjelmointiympäristöön sekä matriisikertolaskun rinnakkaistamiseen. Sen jälkeen laadittiin yksinkertaisiin oletuksiin perustuva algoritmi ja sitä vastaava kustannusfunktio sekä jaetulle että hajautetulle muistille. Algoritmit myös toteutettiin ja niitä suoritettiin CSC:n Puhti-supertietokoneessa. Suoritusaikoja mittaamalla määritettiin kustannusfunktioiden parametrit ja saatiin tietoa funktioiden tarkkuudesta.

Kumpikin kustannusmalli osoittautui tyydyttävän tarkaksi, kun $n \leq 512$. Mallien avulla voidaan karkeasti arvioida rinnakkaistamisesta saatavaa hyötyä. Tarkkaan suoritusaikojen ennustamiseen mallit eivät kuitenkaan sovi. Ne eivät myöskään huomioi nykyaikaisten rinnakkaistietokoneiden hierarkkista rakennetta. Siksi on syytä olettaa, että suurempia rinnakkaislaskentaympäristöjä varten tarvitaan yksityiskohtaisempi malli.
Avainsanat: Kustannusmalli, MPI, numeerinen lineaarialgebra, OpenMP, rinnakkaislaskenta
Abstract: For performance, it is important for a parallel program to have a balance between computation and communication. In this master's thesis, a parallel algorithm and a corresponding cost model were developed for matrix multiplication

$C = AB$, where A, B and C are $n \times n$ matrices. The algorithm and cost model were developed for both distributed and shared memory parallel computers. The cost model was represented as a cost function.

The constructive approach was applied. First, understanding was obtained about parallel computers, theoretical models of parallel computing, two parallel programming frameworks, and parallelizing matrix multiplication. After that, an algorithm and a corresponding cost function based on simple assumptions were developed for both distributed and shared memory parallel computers. Also, algorithms were implemented and executed on CSC Puhti supercomputer. By measuring the execution times, the parameters of the cost functions were determined and the information about the accuracy of the cost functions was gathered.

Both models turned out to be satisfactorily accurate, when $n \leq 512$. By using these models, it is possible to make rough estimates about what can be accomplished by parallelizing. The models are, however, not suitable for exact execution time prediction. Also, the models don't pay attention to hierarchical structure of modern parallel computers. That's why it seems likely that for larger parallel computing environments, a more detailed model is needed.
Keywords: Cost model, MPI, numerical linear algebra, OpenMP, parallel computing

Sanasto

BSP	Engl. *Bulk-Synchronous Parallel*, rinnakkaislaskennan teoreettinen malli. Yhden superaskeleen aikana kukin prosessori voi suorittaa laskentaa tai viestintää tai molempia. Superaskeleen jälkeen tapahtuu synkronointi.
De facto -standardi	Käytännössä vallitseva, tosiasiallinen standardi, useinkaan ei virallinen standardi.
Flynnin taksonomia	Tietokonearkkitehtuureita koskeva luokittelu, joka jakaa tietokoneet neljään luokkaan: SISD, SIMD, MISD ja MIMD.
LogP	Rinnakkaislaskennan kustannusmalli, joka kiinnittää erityistä huomiota prosessorien välisen viestinnän kustannuksiin. Mallin parametrit ovat L, o, g ja P, joista myös mallin nimi tulee.
MIMD	Engl. *Multiple Instruction stream, Multiple Data stream*, Flynnin taksonomiaan kuuluva tietokone, jonka jokaisella prosessorilla on oma käskyvirtansa ja datavirtansa. Tähän luokkaan kuuluvat käytännössä kaikki nykyaikaiset rinnakkaistietokoneet.
MISD	Engl. *Multiple Instruction stream, Single Data stream*, Flynnin taksonomiaan kuuluva tietokone, joka suorittaa useaa eri käskyvirtaa samanaikaisesti samalle datavirralle.
MPI	Engl. *Message-Passing Interface*, viestinvälityskirjaston määrittelevä spesifikaatio, jota käytetään rinnakkaislaskennassa.
OpenMP	Engl. *Open Multi-Processing*, jaetun muistin sovellusohjelmointirajapinta, jota käytetään rinnakkaislaskennassa.

PRAM	Engl. *Parallel Random Access Machine*, yksinkertainen rinnakkaislaskennan teoreettinen malli, jonka avulla voidaan tutkia ratkaistavan ongelman rinnakkaistuvuutta. Malli ei ota kantaa prosessorien välisen viestinnän aiheuttamiin kustannuksiin.
QRQW PRAM	Engl. *Queue-Read Queue-Write Parallel Random Access Machine*, rinnakkaislaskennan teoreettinen malli, jossa samaan muistipaikkaan kohdistuvat luku- tai kirjoituspyynnöt menevät jonoon ja yhtä pyyntöä palvellaan kerrallaan.
SIMD	Engl. *Single Instruction stream, Multiple Data stream*, Flynnin taksonomiaan kuuluva tietokone, joka suorittaa samaa käskyvirtaa samanaikaisesti usealle eri datavirralle.
SISD	Engl. *Singe Instruction stream, Single Data stream*, Flynnin taksonomiaan kuuluva tietokone, joka sisältää yhden prosessorin, jolla on yksi käskyvirta ja yksi datavirta. SISD ei ole varsinaisesti rinnakkaistietokone.

Sisältö

1 Johdanto

Peräkkäisohjelmointi on erilaisilta ohjelmoinnin peruskursseilta tuttu ja ainakin yksinkertaisia ongelmia ratkaistaessa myös helposti ymmärrettävä ohjelmointitapa. Nykyaikaisten supertietokoneiden täyden potentiaalin hyödyntäminen edellyttää kuitenkin rinnakkaisuuden käyttämistä. Rinnakkaisuus puolestaan tuo omat ongelmansa. Suorituskyvyn kannalta merkittävä tekijä on laskennan ja viestinnän välinen suhde [16]. Vaikka rinnakkaisuuden lisääminen pienentääkin laskentaan kuluvaa aikaa, se useimmiten myös lisää laskennasta vastaavien prosessoriytimien, prosessien tai säikeiden välistä viestintää. Siksi parhaan mahdollisen suorituskyvyn saavuttaminen edellyttää tasapainon löytämistä.

Yleisesti mallintamisen tarkoitus on kuvata mallinnettavan ilmiön oleelliset piirteet riittävän tarkasti ennustamista ja analyysiä varten, eikä rinnakkaislaskennan mallintaminen muodosta tästä poikkeusta [60]. Aikojen saatossa lukuisia rinnakkaislaskennan teoreettisia malleja onkin esitetty, mm. PRAM [32, 36], QRQW PRAM [38], BSP [80, 35] ja LogP [17] sekä näiden pohjalta kehitetyt muunnelmat. Teoreettisen mallin tulisi saavuttaa yksinkertaisuuden ja yksityiskohtaisuuden välinen tasapaino [17]. Mallien kehittyessä suuntauksena on ollut kuhunkin malliin kuuluvien parametrien lisääntyminen [74], mikä on johtanut tarkempiin, mutta samalla monimutkaisempiin malleihin. Kuitenkin edellä mainitut mallit ovat yleisluontoisia ja ne täytyy soveltaa aina kuhunkin tarkasteltavaan laskentatehtävään. Tilanteen mukaan osan parametreista voi mahdollisesti jättää mallia sovellettaessa pois [17, 6].

Tutkimusongelmaksi asetetaan rinnakkaisen matriisikertolaskun kustannusmallin laatiminen. Kustannusmallin täytyy ottaa huomioon sekä laskentaan että viestintään kuluvat aikakustannukset. Jotta malli ei olisi liian yleisluontoinen, ennen mallin laatimista kehitetään myös kertolaskualgoritmi, jonka kustannuksia mallin avulla pyritään arvioimaan. Malli esitetään kustannusfunktiona. Algoritmi ja kustannusfunktio laaditaan sekä hajautetun että jaetun muistin rinnakkaistietokoneille. Matriisikertolaskun osalta rajoitutaan muotoa $C = AB$ olevaan kertolaskuun, missä A, B ja C ovat neliömatriiseja.

Tutkielmassa sovelletaan konstruktiivista tutkimusotetta [49, 59]. Konstruktiivista tutkimusotetta noudattavassa tutkimuksessa tuotetaan konstruktio, joka rat-

kaisee jonkin käytännön ongelman. Muunlaisesta ongelmanratkaisusta tutkimusotteen erottaa se, että ongelma ja sen ratkaisu liitetään olemassa olevaan teoreettiseen tietoon, sekä vaatimus käytännön toimivuudesta ja uutuusarvosta. Tässä tapauksessa konstruktiona voidaan pitää kehitettäviä matriisikertolaskualgoritmeja sekä niihin liittyviä kustannusmalleja. Oleellinen osa konstruktiivista tutkimusta on perusteellinen aihepiiriin tutustuminen. Siksi ennen algoritmien ja kustannusmallien laatimista perehdytään rinnakkaistietokoneisiin, rinnakkaislaskennan teoreettisiin malleihin, kahteen yleisesti käytettyyn rinnakkaislaskennan ohjelmointiympäristöön sekä matriisikertolaskuun ja joihinkin olemassa oleviin rinnakkaisalgoritmeihin.

Algoritmit toteutetaan C-kielisinä ohjelmina ja niihin liittyviä kustannusmalleja verrataan kokeellisesti saataviin tuloksiin. Laskentaympäristönä on CSC:n Puhti-rinnakkaistietokone [27, 28]. Osoittautuu, että kumpikin kustannusmalli pätee kohtuullisen hyvin, kun $n \times n$-matriiseja kerrottaessa on $n \leq 512$. Mallien avulla voidaan arvioida karkeasti, missä kohtaa rinnakkaisuuden lisääminen ei maksa vaivaa. Sen sijaan suoritusaikojen tarkkaan ennustamiseen mallien tarkkuus ei riitä.

Luvussa 2 esitellään konstruktiivinen tutkimusote ja sen soveltaminen tässä tutkielmassa. Luvussa 3 perehdytään rinnakkaistietokoneisiin, luvussa 4 rinnakkaislaskennan teoreettisiin malleihin ja luvussa 5 tutkielmassa käytettyihin rinnakkaislaskennan ohjelmointiympäristöihin. Luvussa 6 käydään läpi matriisikertolaskua ja eräitä olemassa olevia rinnakkaisalgoritmeja. Tutkielmaa varten laaditut rinnakkaisalgoritmit ja niitä vastaavat kustannusfunktiot esitetään luvussa 7. Luvussa 8 puolestaan esitetään algoritmien toteutus ja tarkastellaan malleja kokeellisesti. Luvussa 9 esitetään johtopäätökset.

2 Konstruktiivisesta tutkimusotteesta

Kustannusmallia laadittaessa sovelletaan konstruktiivista tutkimusotetta [49, 59]. Konstruktiivinen tutkimusote esitellään lyhyesti luvussa 2.1. Luvussa 2.2 puolestaan kerrotaan, kuinka tutkimusotetta tässä tutkielmassa sovelletaan.

2.1 Tutkimusotteesta yleisesti

Kasasen ym. [49] mukaan konstruktiolla tarkoitetaan kokonaisuutta, joka tuottaa ratkaisun johonkin selkeään ongelmaan. Konstruktiota kehitettäessä tuotetaan jotain aiemmasta poikkevaa. Konstruktiivisella tutkimusotteella puolestaan tarkoitetaan tutkimusmenetelmää, jota käytetään konstruktioiden tuottamiseen. Esimerkkeinä konstruktioista kirjoittajat mainitsevat mm. matemaattiset algoritmit, keinotekoiset kielet sekä uudet lääkkeet ja hoitomuodot. He kuitenkin huomauttavat, ettei mitä tahansa ongelmanratkaisua tule pitää konstruktiivisena tutkimuksena. Ongelma ja sen ratkaisu täytyy sitoa jo olemassa olevaan teoreettiseen tietoon. Lisäksi ratkaisulta edellytetään uutuusarvoa ja käytännön toimivuutta. Konstruktiivisen tutkimusotteen keskeisiä osia on havainnollistettu kuvassa 2.1.

Kuva 2.1: Konstruktiivisen tutkimusotteen osat. Muokattu Kasasen ym. esittämästä kuvasta [49, kuva 1].

Kasasen ym. [49] mukaan konstruktiivista tutkimusotetta noudattava tutkimusprosessi voidaan jakaa kuuteen vaiheeseen, joiden järjestys toki voi vaihdella:

1. käytännön kannalta relevantin ongelman etsiminen
2. perusteellinen aihepiiriin tutustuminen
3. ratkaisun innovointi
4. ratkaisun toimivuuden demonstrointi

5. ratkaisun tuoman teoreettisen kontribuution analysointi
6. ratkaisun soveltamisalan tarkastelu.

Heidän mukaansa vaihe 3 eli innovointi on onnistuneen konstruktiivisen tutkimuksen kannalta oleellinen. Innovointi on luonteeltaan heuristista, ja ratkaisun kurinalainen teoreettinen arviointi tapahtuu tyypillisesti myöhemmin.

Lukka [59] lisää edellä esitettyyn listaan pitkän aikavälin tutkimusyhteistyön mahdollisuuksien selvittämisen. Hänen mukaansa konstruktiivisessa tutkimusotteessa tutkijan ja käytännön edustajien tulee toimia hyvin tiiviissä yhteistyössä. Tyypillisesti tutkijan pitäisi kuulua tiimiin, jossa on mukana myös yhteistyötä tekevien organisaatioiden edustajia. Jos tutkija työskentelee yksin, on hyvin todennäköistä, ettei konstruktiota koskaan toteuteta käytännössä. Lukka huomauttaa myös, että vaikka ongelman ratkaisu osoittautuisikin käytännössä toimimattomaksi, sillä voi silti olla teoreettista merkitystä.

2.2 Tutkimusotteen soveltaminen tässä tutkielmassa

Kasasen ym. [49] luettelemat tutkimusprosessin vaiheet sisältyvät tähän tutkielmaan. Matriisikertolasku on keskeinen numeerisen laskennan työkalu, ja sen kustannuksien arviointi on tärkeää, jotta laskennan rinnakkaistamisella saavutettaisiin paras mahdollinen hyöty. Aihepiiriin myös tutustutaan perusteellisesti ja olemassa olevaa teoreettista tietämystä apuna käyttäen laaditaan algoritmi sekä hajautetun että jaetun muistin rinnakkaistietokoneelle ja kumpaakin algoritmia vastaava kustannusmalli. Kustannusmallin toimivuutta tarkastellaan käytännön rinnakkaislaskentaympäristössä ja mallin toimivuutta arvioidaan tulosten pohjalta. Lisäksi pohditaan mallin teoreettista kontribuutiota.

Sen sijaan Lukan mainitseman [59] tutkimusyhteistyön liittäminen tutkielmaan ei ole aivan yhtä yksinkertaista. Lukka puhuu tutkijasta ja käytännön harjoittajista, mutta tässä tutkielmassa nämä ovat ainakin osittain sama henkilö, sillä tutkielman kirjoittaja itse myös testaa mallin toimivuutta. Lisäksi tutkimusprosessissa on keskeisesti mukana myös tutkielman ohjaaja, jonka kanssa tutkielman kirjoittaja pitää säännöllisiä palavereja. Tämän laajuista yhteistyötä on jo tutkielman työmäärän rajaamisen vuoksi pidettävä riittävänä.

3 Rinnakkaistietokoneista

Tämän luvun tarkoitus on antaa yleiskuva rinnakkaistietokoneista. Luvussa 3.1 käsitellään Flynnin taksonomiaa, joka on eräs tunnettu tapa luokitella erilaisia tietokoneita. Luvussa 3.2 käsitellään keskeisimpiä käytössä olevia rinnakkaistietokoneita ja luvussa 3.3 suorituskykyyn vaikuttavia seikkoja.

3.1 Flynnin taksonomia

Flynn esitti vuonna 1966 ilmestyneessä artikkelissaan [30] tietokonearkkitehtuureita koskevan luokittelun, jonka keskeisiä käsitteitä ovat käskyvirta (*instruction stream*) ja datavirta (*data stream*). Tietokoneet voidaan voidaan jakaa neljään luokkaan seuraavasti:

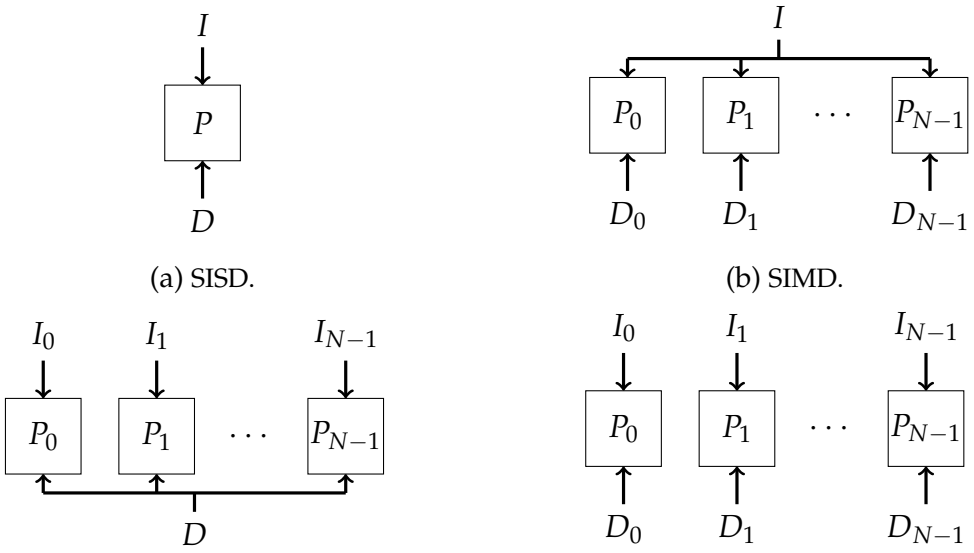
- SISD (*Single Instruction stream, Single Data stream*)
- SIMD (*Single Instuction stream, Multiple Data stream*)
- MISD (*Multiple Instruction stream, Single Data stream*)
- MIMD (*Multiple Instruction stream, Multiple Data stream*).

Luokkia havainnollistetaan kuvassa 3.1. Seuraavissa alaluvuissa kuhunkin luokkaan perehdytään tarkemmin.

3.1.1 SISD

SISD sisältää yhden prosessorin, joka käsittelee datavirtaansa suorittamalla käskyvirran käskyjä yksi kerrallaan. Tätä on havainnollistettu kuvassa 3.1a. Kyseessä ei varsinaisesti ole rinnakkaistietokone. Siitä huolimatta tällainenkin tietokone voi toteuttaa matalan tason rinnakkaisuutta. Tällaisia rinnakkaisuuden muotoja ovat käskytason rinnakkaisuus ja monisäikeistys [72, s. 27–31].

Käskytason rinnakkaisuudessa [72, s. 27–30] on kysymys siitä, että prosessorin eri osat suorittavat operaatioita samanaikaisesti. Tähän on kaksi vaihtoehtoa: liukuhihnaus ja useamman käskyn suorittaminen. Useampaa käskyä voidaan suorittaa samanaikaisesti, mikäli prosessorissa on useita funktionaalisia yksiköitä, jotka pystyvät käskyjä suorittamaan. Liukuhihnauksessa [68, s. 214–219] taas suoritettava käsky jaetaan peräkkäisiin osakäskyihin, joita prosessorin eri osat suorittavat.



(a) SISD. (b) SIMD.

(c) MISD. (d) MIMD.

Kuva 3.1: Flynnin taksonomia. *P* tarkoittaa prosessoria, *I* käskyvirtaa ja *D* datavirtaa. Muokattu Flynnin ja Ruddin esittämästä kuvasta [31, kuva 1]. Kuva (c) poikkeaa oleellisesti Flynnin ja Ruddin esityksestä.

Näin useampaa käskyä voidaan suorittaa limittäin, mikä säästää aikaa. Kuva 3.2 havainnollistaa 4 peräkkäisen 6-osaisen käskyn liukuhihnausta.

Säietason rinnakkaisuus eli monisäikeistys [72, s. 30–31] edustaa käskytason rinnakkaisuuteen verrattuna karkeampijakoista rinnakkaisuutta. Jos yhdessä säikeessä suoritettava tehtävä joutuu odottamaan esim. muistihaun valmistumista, on mahdollista antaa suoritusaikaa jollekin toiselle säikeelle. Näin odotusaika voidaan käyttää hyödyllisesti. Monisäikeistyksestä on tietenkin hyötyä vain, mikäli säikeestä toiseen siirtyminen on riittävän nopeaa.

Sykli	0	1	2	3	4	5	6	7	8
Käsky 0	0	1	2	3	4	5			
Käsky 1		0	1	2	3	4	5		
Käsky 2			0	1	2	3	4	5	
Käsky 3				0	1	2	3	4	5

Kuva 3.2: Liukuhihnaus. Muokattu Nullin ja Loburin esittämästä kuvasta [68, kuva 5.4].

3.1.2 SIMD

SIMD-tietokone suorittaa samaa käskyä usealle eri datavirralle samanaikaisesti. Tätä on havainnollistettu kuvassa 3.1b. Pacheco ja Malensek [72, s. 31–32] havainnollistavat SIMD:n toimintaa vektorien yhteenlaskulla. Oletetaan, että suoritettavana on seuraavanlainen yhteenlasku:

```
for (i = 0; i < N; i++)
    x[i] += y[i];
```

Jos prosessorien määrä on N, voi prosessori P_i ladata luvut `x[i]` ja `y[i]`, suorittaa yhteenlaskun ja tallentaa tuloksen `x[i]`:hin. Jos taas prosessoreita on vähemmän, laskenta on suoritettava riittävän monta kertaa, osalle alkioista kerrallaan. Jos jako ei mene tasan, viimeisellä kierroksella osa prosessoreista ei tee mitään.

Pacheco ja Malensek huomauttavat, että SIMD-tietokoneilla on omat rajoitteensa. Klassisen SIMD-tietokoneen prosessorit nimittäin joko suorittavat annettua käskyä tai eivät tee mitään. Lisäksi kaikki prosessorit toimivat täysin synkronisesti eivätkä ne voi varastoida käskyjä mahdollista myöhempää suoritusta varten. Siksi ehtolauseita sisältävien ohjelmien suorittaminen on SIMD-tietokoneilla tehotonta. Sen sijaan yksinkertaisten vektoreita käsittelevien silmukoiden rinnakkaistamiseen SIMD sopii hyvin.

Tärkeä esimerkki SIMD-periaatteen mukaan toimivista tietokoneista ovat vektoritietokoneet [29, s. 384–386], esim. CRAY-1 [75]. Vaikka vektoritietokoneiden varsinainen kultakausi onkin ohi [76], on nykyaikaisista vektoritietokoneista ainakin NEC:n Aurora-SX TSUBASA [51] herättänyt kiinnostusta [1]. Nykyisin merkittäviin SIMD-sovelluksiin lukeutuvat prosessorien vektorilaajennokset [8]. Myös grafiikkasuorittimet eli GPU:t noudattavat joiltain osin SIMD-mallia.

3.1.3 MISD

MISD on sikäli ongelmallinen, että siitä on tuskin ainoaakaan käytännön esimerkkiä. Flynn ja Rudd [31] esittävät MISD-tietokoneesta liukuhihnamaisen tulkinnan, jossa kunkin prosessorin (paitsi viimeisen) ulostulo ohjataan seuraavan prosessorin syötteeksi. Tämä vastaa Kungin ja Leisersonin [52] esittämää systolista matriisitietokonetta. Kuten Ngoko ja Trystram [67] toteavat, tällaisen tietokoneen luokittelusta MISD-tietokoneeksi voidaan olla eri mieltä. Onhan selvää, että esim. prosessorit P_0 ja P_1 eivät varsinaisesti käsittele samaa datavirtaa, vaan P_1 käsittelee dataa, jota P_0 on ensin käsitellyt.

Ngoko ja Trystram [67] esittävät, että MISD on laiminlyöty rinnakkaistietokoneen malli. He esittelevät MISD-mallin, jossa eri prosessorit ratkaisevat samaa ongelmaa, mutta kukin prosessori suorittaa eri algoritmia. Kun yksi prosessori löytää ratkaisun, kaikki prosessorit pysähtyvät. Ngokon ja Trystramin esittämä malli vastaa MISD:tä, sillä prosessorin suorittamaa algoritmia voidaan ajatella käskyvirtana ja ratkaistavaa ongelmaa datavirtana. Kuva 3.1c on piirretty tämän tulkinnan mukaisesti, siis Flynnin ja Ruddin esityksestä [31] poiketen.

3.1.4 MIMD

MIMD-tietokoneen prosessorit suorittavat kukin omaa käskyvirtaansa omalle datavirralleen. MIMD-tietokonetta on havainnollistettu kuvassa 3.1d. Kuten Dongarra ja van der Steen [29, s. 382] toteavat, luokka sisältää hyvin monenlaisia toisistaan poikkeavia tietokoneita. Siksi heidän mukaansa tätä luokkaa ajatellen Flynnin taksonomia onkin tietokoneiden luokittelun kannalta riittämätön.

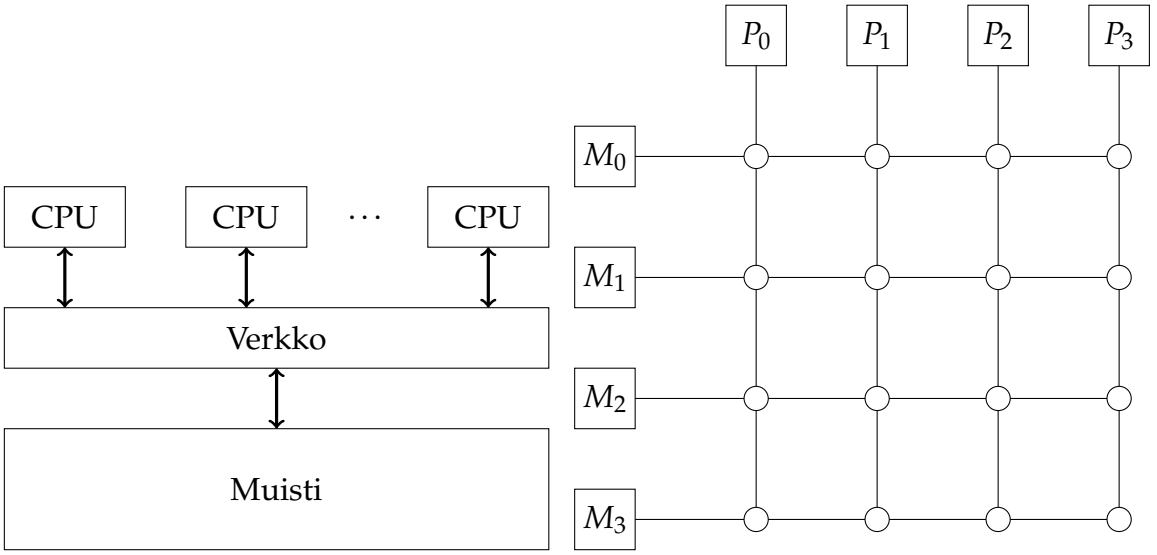
Suurin osa nykyaikaisista rinnakkaistietokoneista on MIMD-tietokoneita. Pacheco ja Malensek [72, s. 34–35] jakavat luokkaan kuuluvat tietokoneet kahteen päätyyppiin: jaetun ja hajautetun muistin järjestelmiin.

3.2 Tavallisesti käytettyjä rinnakkaistietokoneita

Kuten edellä on todettu, suurin osa nykyaikaisista rinnakkaistietokoneista on MIMD-koneita ja nämä taas voidaan jakaa karkeasti jaetun ja hajautetun muistin rinnakkaistietokoneisiin. Jaetun muistin tietokoneisiin perehdytään luvussa 3.2.1 ja hajautetun muistin tietokoneisiin luvussa 3.2.2. Lisäksi luvussa 3.2.3 perehdytään grafiikkasuorittimiin, jotka noudattavat joiltain osin SIMD-mallia.

3.2.1 Jaetun muistin rinnakkaistietokoneet

Jaetun muistin järjestelmä [72, s. 35–37] koostuu prosessoreista, yhteisestä muistista ja verkosta, joka yhdistää prosessorit ja muistin. Näin jokainen prosessori pääsee käsiksi mihin tahansa jaetun muistin muistisijaintiin. Tätä kautta prosessorit myös tavallisesti kommunikoivat keskenään. Kommunikointi on siksi implisiittistä. Rakennetta on havainnollistettu kuvassa 3.3a. Prosessorit ovat tavallisesti moniytimisiä.



Kuva 3.3: Kuvassa (a) jaetun muistin rinnakkaistietokoneen rakenne ja kuvassa (b) verkon ristikytkentä. Kuvassa (b) komponentit P_i ovat prosessoreita, M_i muisteja ja ympyrät kytkimiä. Muokattu Pachecon ja Malensekin esittämistä kuvista [72, kuvat 2.3 ja 2.7a].

Vaikka jaetun muistin ympäristöissä kaikki prosessorit pääsevätkin käsiksi samaan muistiin, pääsy voidaan toteuttaa eri tavoin. Pacheko ja Malensek [72, s. 36–37] luokittelevat jaetun muistin tietokoneet kahteen kategoriaan: UMA (*uniform memory access*) ja NUMA (*non-uniform memory access*). UMA-järjestelmässä verkko yhdistää kaikki prosessorit suoraan jaettuun muistiin. Muistihaun nopeus ei riipu muistisijainnista. Sen sijaan NUMA-järjestelmässä kukin prosessori on suoraan yhteydessä tiettyyn jaetun muistin lohkoon. Toisten prosessorien muistilohkoon kukin prosessori pääsee käsiksi prosessorien sisältämän erikoislaitteiston avulla. Tässä tapauksessa prosessorin omaan lohkoon kohdistuva muistihaku on nopeampi kuin jonkin toisen prosessorin lohkoon kohdistuva. NUMA:n etu on, että se skaalautuu muistin määrää kasvatettaessa paremmin kuin UMA. Toinen etu on prosessorin omaan muistilohkoon kohdistuvan muistihaun nopeus. Toisaalta UMA on ohjelmoijan kannalta helpompi, koska erilaisista hakuajoista ei tarvitse huolehtia.

Pachecon ja Malensekin [72, s. 37–38] mukaan prosessorit ja muisti voidaan yhdistää toisiinsa väylällä tai kytkinverkolla. Väylän käyttämistä puoltaa edullisuus. Varjopuolena on, että väylän johdot ovat kaikkien väylän yhdistämien laitteiden jakamia. Kun laitteiden määrä kasvaa, kasvaa myös kilpailu ja samalla suorituskyky laskee. Siksi järjestelmien kasvaessa on alettu siirtyä kytkinverkkojen käyttöön. Ku-

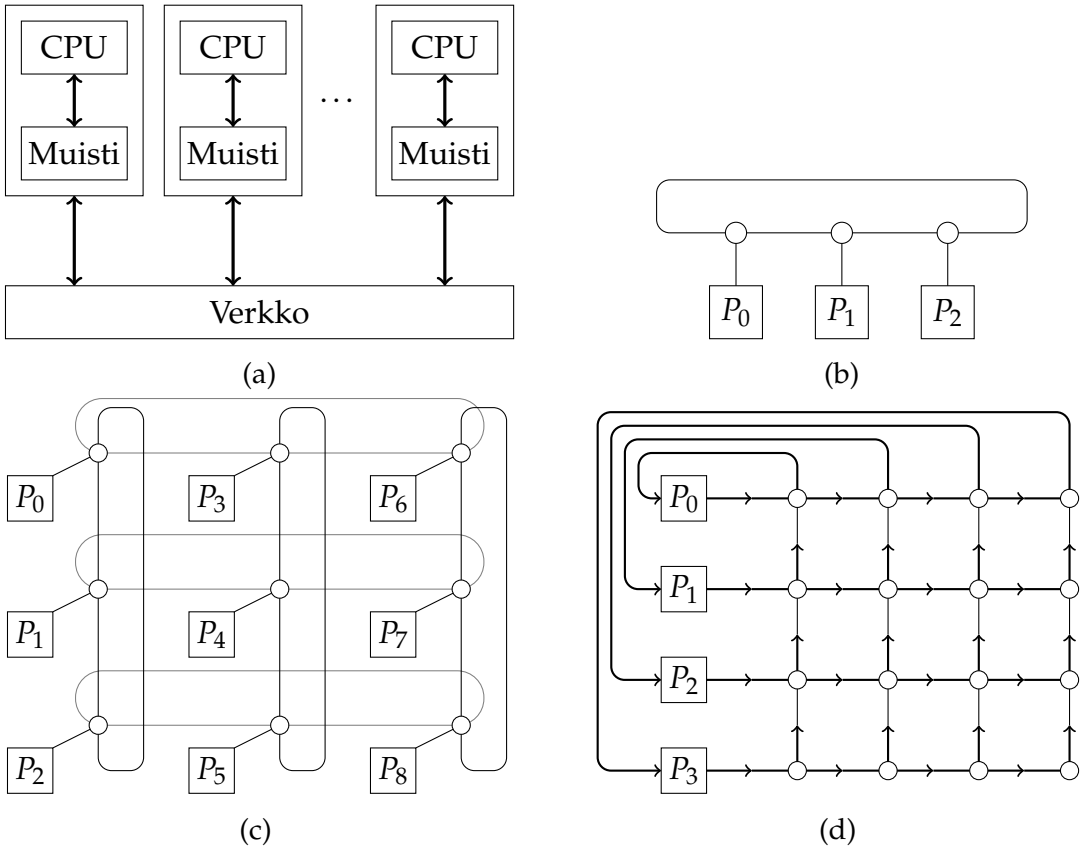
ten nimestä voi päätellä, kytkinverkot sisältävät kytkimiä, jotka kontrolloivat laitteiden välistä kommunikointia. Eräs mahdollisuus on ristikytkentä. Ristikytkentää on havainnollistettu kuvassa 3.3b. Ristikytkentä on väylää tehokkaampi, joskin kalliimpi ratkaisu.

3.2.2 Hajautetun muistin rinnakkaistietokoneet

Hajautetun muistin järjestelmät [72, s. 35, 37] koostuvat prosessoreista, joilla jokaisella on oma yksityinen muistinsa, ja prosessorit yhdistävästä verkosta. Kuva 3.4a havainnollistaa rakennetta. Prosessorien välinen kommunikointi tapahtuu eksplisiittisesti verkon kautta esim. viestinvälityksen avulla. Hajautetun muistin rinnakkaistietokoneet ovat tyypillisesti useasta kaupallisesta tietokoneesta ja kaupallisesta verkosta koostuvia klustereita. Klusterin noodit eli verkon yhdistämät tietokoneet ovat itse asiassa usein jaetun muistin laitteita.

Pachecon ja Malensekin mukaan [72, s. 38–43] hajautetun muistin verkot voidaan jakaa kahteen päätyyppiin: suoraan ja epäsuoraan. Suorassa verkossa jokainen prosessorin ja muistin muodostama kokonaisuus on yhdistetty yhteen kytkimeen ja kytkimet on yhdistetty toisiinsa. Kuvassa 3.4b on esitetty rengastopologiaa noudattava ja kuvassa 3.4c kaksiulotteinen toroidaalinen suora verkko. Suoran verkon tehoa kuvattaessa käytetään yhtenä mittana linkkien määrää. Tapana on tosin laskea vain kytkinten väliset linkit. Esim. kuvan 3.4b tapauksessa linkkejä on 3 ja kuvan 3.4c tapauksessa 18. Kuvassa 3.4d puolestaan on esimerkki epäsuorasta verkosta. Erona suoraan verkkoon on, että kytkimet eivät ole suoraan yhteydessä prosessoreihin ja linkit ovat usein yksisuuntaisia. Kullakin prosessorilla on oma ulostulo- ja sisäänmenolinkkinsä, ja kytkinten voidaan ajatella muodostavan oman verkkonsa. Kuvan 3.4d tapauksessa kyseessä on ristikytkentä (vrt. jaettu muisti, kuva 3.3b) ja kaikki prosessorit voivat kommunikoida keskenään yhtä aikaa, kunhan vain useampi prosessori ei yritä kommunikoida saman prosessorin kanssa.

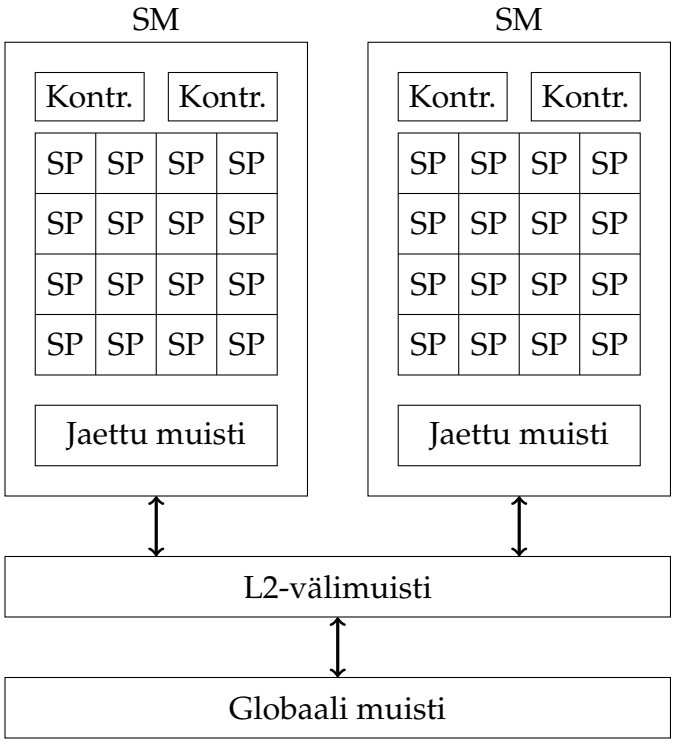
Vaikka jaetun muistin käyttäminen onkin ohjelmoijan kannalta helpompaa, Pacheco ja Malensek [72, s. 49] huomauttavat, että hajautetussa muistissa on hyvät puolensa. Laitteiston kannalta merkittävin seikka on verkon skaalautuvuus. Väylällä voi käytännössä yhdistää vain muutaman prosessorin, ja toisaalta ristikytkentä on tehokas, mutta kallis vaihtoehto. Sen sijaan hajautetun muistin verkkoratkaisut, esim. hyperkuutio ja edellä kuvattu toroidi, ovat edullisempia. Siksi hajautetun muistin järjestelmät soveltuvat paremmin paljon laskentaa vaativiin tehtäviin.



Kuva 3.4: Kuvassa (a) hajautetun muistin rinnakkaistietokoneen rakenne. Kuvissa (b) ja (c) esimerkki suorasta verkosta (rengastopologia ja toroidi) ja kuvassa (d) epäsuorasta (ristikytkentä). Kuvissa (b)–(d) ympyrät ovat kytkimiä ja kukin komponentti P_i koostuu prosessorista sekä tämän omasta muistista. Kuvat (a), (b) ja (d) on muokattu Pachecon ja Malensekin esittämistä kuvista [72, kuvat 2.4, 2.8a ja 2.14]. Kuva (c) on muokattu Pachecon ja Malensekin esittämästä kuvasta [72, kuva 2.8b], mutta sitä on tarkennettu linkkien pituuden suhteen Lain esityksen pohjalta [53, kuva 1].

3.2.3 Grafiikkasuorittimia käyttävät ympäristöt

Näytönohjaimet käyttävät grafiikkaprosessoreita eli grafiikkasuorittimia (lyh. GPU, *Graphics Processing Unit*). Pacheco ja Malensek kirjoittavat [72, s. 291], että vaikka alkuperäinen käyttötarkoitus onkin liittynyt tietokonegrafiikkaan, sittemmin grafiikkasuorittimia on alettu käyttää myös muuhun laskentaan niiden suorituskyvyn vuoksi. Usein käytetäänkin lyhennettä GPGPU (*General Purprose Computing on GPUs*). Alun perin grafiikkasuorittimia ohjelmoitaessa ei-graafisetkin algoritmit piti muotoilla graafisesti, mutta myöhemmin on kehitetty yleiskäyttöön tarkoitettuja sovellusohjelmointirajapintoja, esim. OpenCL [50] ja Nvidian CUDA [70].



Kuva 3.5: GPU:n rakenne. Muokattu Pachecon ja Malensekin esityksen pohjalta [72, kuva 6.1].

Tarkka GPU:n rakenteen kuvaus on tietenkin riippuvainen siitä, mistä järjestelmästä milloinkin on kyse. Hu ym. [46] ovat laatineet yleisen mallin, joka pyrkii kuvaamaan paitsi GPU:n rakennetta myös tehtävienhallintaa, muistinkäyttöä ynnä muita keskeisiä seikkoja. Tämän tutkielmassa kuitenkin tyydytään hyvin yleisluontoiseen kuvaukseen. Siksi seuraavassa seurataan Pachecon ja Malensekin suppeaa esitystä [72, s. 292–295], joka soveltuu ennen kaikkea Nvidian grafiikkasuorittimien kuvaamiseen. GPU koostuu useasta SM-prosessorista (*Streaming Multiprocessor*). Yhden SM:n voidaan ajatella vastaavan yhtä tai useampaa SIMD-tietokonetta. SM puolestaan sisältää useita kontrolliyksiköitä sekä SP-ytimiä (*Streaming Processor*). Yhden SM:n sisältämät SP:t jakavat saman pienen, mutta nopean muistin. Rakennetta on havainnollistettu kuvassa 3.5. Samassa tietokoneessa on myös tavanomainen prosessori eli CPU. Tällaisessa ympäristössä sitä käytetään tavallisesti muuhun kuin varsinaiseen rinnakkaislaskentaan.

Pacheco ja Malensek [72, s. 34] huomauttavat, ettei GPU ole puhtaasti SIMD-järjestelmä, sillä yksi SM pystyy suorittamaan useampaa kuin yhtä käskyvirtaa. Pikemminkin GPU sijoittuu SIMD:n ja MIMD:n väliin. Joka tapauksessa yksi SM sisäl-

tää monta SP-ydintä, mikä mahdollistaa tehokkaan SIMD-tyylisen laskennan. Kyseessä ei myöskään ole puhtaasti jaetun eikä hajautetun muistin laskentaympäristö, vaan GPU voi käyttää kumpaa tahansa.

3.3 Suorituskyvystä

Ohjelman rinnakkaistamisella pyritään kasvattamaan suorituskykyä. Esim. Crovella ym. [16] käyttävät viestinnän ja laskennan suhdetta rinnakkaisohjelman suorituskyvyn ennustamiseen. Voidaankin ajatella, että suorituskyvyn kannalta rinnakkaisohjelma koostuu laskennasta ja viestinnästä. Luvussa 3.3.1 perehdytään Amdahlin lakiin, joka antaa teoreettisen ylärajan sille, mitä laskentaa nopeuttamalla voidaan saavuttaa, ja luvussa 3.3.2 prosessorien välisen verkon ominaisuuksiin.

3.3.1 Amdahlin laki

Amhdal kritisoi vuonna 1967 ilmestyneessä artikkelissaan [7] rinnakkaisuuteen liitettyjä odotuksia. Itse artikkeli ei sisällä tarkkaan muotoiltua lakia, vaan Amdahlin esittämät havainnot on muotoiltu täsmällisesti vasta myöhemmin. Amdahlin laki voidaan esittää Gustafsonin [41] käyttämiä merkintöjä mukaillen seuraavasti. Oletetaan, että ohjelma koostuu peräkkäisestä osasta, jonka suoritusaika on t_s, ja rinnakkaistuvasta osasta, jonka suoritusaika yhdellä prosessorilla on t_p. Ohjelman suoritusaika yhdellä prosessorilla on siis $t_\mathrm{s} + t_\mathrm{p}$. Jos käytössä on N prosessoria eikä muuta kuin laskentaan kuluvaa aikaa oteta huomioon, on suoritusaika $t_\mathrm{s} + t_\mathrm{p}/N$. Näin ollen nopeutus S eli peräkkäisohjelman suoritusajan suhde vastaavan rinnakkaisohjelman suoritusaikaan on

$$S = \frac{t_\mathrm{s} + t_\mathrm{p}}{t_\mathrm{s} + t_\mathrm{p}/N}. \tag{3.1}$$

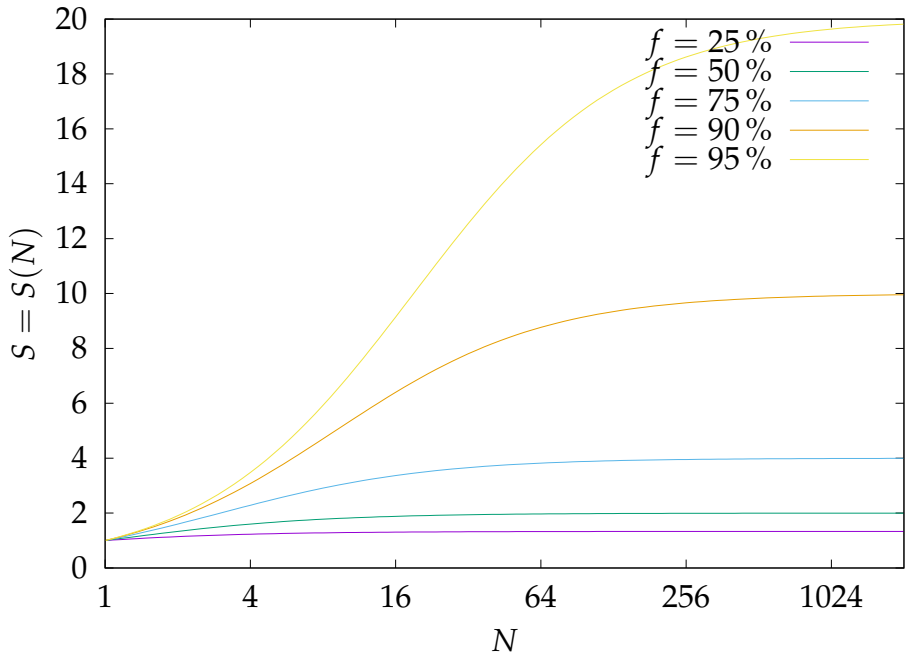
Merkitään Al-hayannin ym. [4] esitystä mukaillen rinnakkaistuvan osan osuutta kokonaisajasta f:llä, siis

$$f = \frac{t_\mathrm{p}}{t_\mathrm{s} + t_\mathrm{p}}. \tag{3.2}$$

Nyt nopeutus voidaan kirjoittaa muodossa

$$\begin{aligned} S &= \frac{t_\mathrm{s} + t_\mathrm{p}}{t_\mathrm{s} + t_\mathrm{p}/N} \\ &= \frac{N(t_\mathrm{s} + t_\mathrm{p})}{Nt_\mathrm{s} + Nt_\mathrm{p} - Nt_\mathrm{p} + t_\mathrm{p}} \\ &= \left(\frac{N(t_\mathrm{s} + t_\mathrm{p})}{N(t_\mathrm{s} + t_\mathrm{p})} - \frac{Nt_\mathrm{p}}{N(t_\mathrm{s} + t_\mathrm{p})} + \frac{t_\mathrm{p}}{N(t_\mathrm{s} + t_\mathrm{p})} \right)^{-1} \\ &= \frac{1}{1 - f + f/N}. \end{aligned} \tag{3.3}$$

Nopeutus on esitetty prosessorien määrän funktiona kuvassa 3.6. Kuten kuvasta nähdään, prosessorien määrän kasvattaminen nopeuttaa suoritusaikaa vain tiettyyn rajaan asti. Tämä johtuu siitä, että ohjelmassa on aina peräkkäinen osa, jonka suoritusaikaan prosessorien määrä ei vaikuta.



Kuva 3.6: Amdahlin lain mukainen nopeutus prosessorien määrän funktiona f:n eri arvoilla.

Gustafson [41] huomauttaa, että Amdahlin lain piilo-oletuksena on, että ongelman koko pysyy samana prosessorien määrästä riippumatta. Hänen mukaansa tämä ei pidä paikkaansa, sillä laskentaresurssien kasvaessa myös ongelman koko ja erityisesti rinnakkaistuvan osan koko kasvavat. Hän esittää jopa, että rinnakkainen

osa kasvaa lineaarisesti prosessorien määrän funktiona ja nopeutus on näin ollen

$$S = \frac{t_\mathrm{s} + t_\mathrm{p} N}{t_\mathrm{s} + t_\mathrm{p}}. \tag{3.4}$$

Schreiber [76] toteaakin, että vaikka Amdahlin laki onkin totta, se ei ole relevantti, koska ohjelman peräkkäinen osa on nykyisin varsin vähäinen kokonaisuuteen verrattuna. Kritiikistä huolimatta on syytä tutkielman kirjoittajan omana näkemyksenä todeta, että Amdahlin laista on oma hyötynsä. Jos algoritmi on laadittu ja f voidaan arvioida, yhtälö 3.3 antaa teoreettisen ylärajan sille, mitä rinnakkaisuudella voidaan saavuttaa. Lisäksi sen avulla voidaan saada käsitys siitä, missä vaiheessa prosessorien lisääminen ei enää maksa vaivaa.

Kuten A-hayanni ym. [4] toteavat, sekä Amdahl [7] että Gustafson [41] käsittelevät vain laskentaa, vaikka todelliseen rinnakkaisohjelman suorituskykyyn vaikuttavat muutkin seikat. Amdahlin laista on kuitenkin kehitetty laajennoksia, jotka pyrkivät kiinnittämään huomiota muihin suorituskykyyn vaikuttaviin tekijöihin. Esim. Li ja Malek [57] kiinnittävät huomiota kommunikaation aiheuttamiin kustannuksiin. Sun ja Chen [77] puolestaan laajentavat Amdahlin mallia vastaamaan moniytimisten prosessorien ominaisuuksia. Laajennoksiin ei tässä tutkielmassa perehdytä.

3.3.2 Kommunikaatioverkon ominaisuuksista

Pachecon ja Malensekin mukaan [72, s. 43–45] verkon suorituskyvyn kuvaamiseksi usein käytetyt suureet ovat latenssi eli viive ja kaistanleveys. Viiveeksi kutsutaan aikaa, joka kuluu siitä, kun lähettäjä aloittaa datan lähettämisen, siihen, kun vastaanottaja aloittaa samaisen datan vastaanottamisen. Kun vastaanottaja on aloittanut ensimmäisen tavun vastaanottamisen, se vastaanottaa dataa nopeudella, jota kutsutaan kaistanleveydeksi. Jos siis verkon viive on ℓ aikayksikköä ja kaistanleveys β tavua aikayksikössä, niin m tavun kokoisen viestin siirtämiseen kuluva aika on

$$T = \ell + \frac{m}{\beta}. \tag{3.5}$$

Edellä sanottu koskee paitsi prosessorien välistä myös prosessorin ja jaetun muistin välistä viestintää.

Pacheco ja Malensek huomauttavat [72, s. 45], että viiveellä ja kaistanleveydellä tarkoitetaan usein eri asioita kuin edellä on kuvattu. Viiveellä saatetaan tarkoittaa yhtälön 3.5 antamaa tiedonsiirtoon kuluvaa kokonaisaikaa. Viiveeseen saatetaan sisällyttää myös esim. viestin kokoamiseen ja purkamiseen kuluva aika. Tässä tutkiel-

massa pitäydytään kuitenkin edellä annetuissa määritelmissä, joissa viive ja kaistanleveys erotetaan toisistaan. Lisäksi viestin kokoamiseen tai purkamiseen kuluvaa aikaa kutsutaan yleiskustannukseksi eikä sitä sisällytetä viiveeseen. Näin viive kuvaa nimenomaan verkon eikä prosessorien ominaisuuksia.

Taulukko 3.1: Maailman tehokkaimmat supertietokoneet. Lyhennetty Lun ym. esittämästä taulukosta [58, taulukko 1]. Taulukko vastaa marraskuun 2021 tilannetta.

Sijoitus	Tietokoneen nimi	Valmistaja	Kaistanleveys (Gbps)	Viive (µs)
1	Fugaku	Fujitsu	180	$\leq$ 0,54
2	Summit	IBM	200	0,6
3	Sierra	IBM/Nvidia	200	0,6
4	Sunway Taihu Light	NRCPC	200	1
5	Perlmutter	HPE	400	ei tiedossa

Taulukossa 3.1 on konkreettisia esimerkkejä nykyaikaisissa rinnakkaistietokoneissa olevien verkkojen ominaisuuksista. Kuten taulukosta nähdään, yleinen kaistanleveys on 200 Gbps ja viive 1 µs tai sen alle. Taulukossa kuvataan kuitenkin vain maailman tehokkaimpia tietokoneita. Lun ym. mukaan [58] heikkotehoisemmissa rinnakkaistietokoneissa 10 Gbps on hyvin tavallinen kaistanleveys.

4 Rinnakkaislaskennan teoreettisista malleista

Mallintamisen tarkoitus on tavoittaa tutkittavan ilmiön oleelliset piirteet riittävän selvästi ja tarkasti, jotta ilmiön analysoiminen ja ennusteiden tekeminen olisi mahdollista [60]. Sama pätee rinnakkaislaskennan teoreettisiin malleihin. Laskennan toteuttaminen vaatii resursseja ja aikaa, toisin kuin mallin käyttäminen. Asian kääntöpuoli on mallin väistämätön epätarkkuus. Lisäksi malleja on hyvin paljon ja ne keskittyvät eri asioihin. Toisaalta mallien välillä on riippuvuuksia ja jotkin mallit toimivat pohjana useille toisille malleille.

Luvuissa 4.1–4.4 esitellään neljä tunnettua mallia: PRAM, QRQW PRAM, BSP ja LogP. PRAM on hyvin yksinkertainen synkroninen rinnakkaislaskennan malli, jonka avulla voi keskittyä tutkimaan ratkaistavan ongelman rinnakkaistuvuutta. Kyseessä on jaetun muistin malli. QRQW PRAM on PRAM-muunnelma, joka ratkaisee samanaikaisen kirjoittamisen ja lukemisen ongelman laittamalla samaan jaetun muistin muistisijaintiin kohdistuvat luku- ja kirjoituspyynnöt jonoon. BSP ja LogP ovat puolestaan hajautetun muistin malleja. LogP on täysin asynkroninen, BSP synkronisen ja asynkronisen mallin välimaastossa. LogP kiinnittää erityistä huomiota prosessorien välisen kommunikaation kustannuksiin. Kutakin mallia käsiteltäessä esitellään lyhyesti myös joitakin mallin laajennoksia. Konkretian saamiseksi luvussa 4.5 annetaan esimerkki siitä, kuinka yleislähetyksen suorittaminen esitellyillä malleilla tapahtuu. Luvussa 4.6 tehdään yhteenveto.

4.1 PRAM

PRAM (*Parallel Random Access Machine*) on Fortunen ja Wyllien [32] esittämä rinnakkaislaskennan malli. Alkuperäisessä Fortunen ja Wyllien mallissa (P-RAM) on äärettömän monta prosessoria, mutta muissa lähteissä mallista esitetään usein versio, jossa prosessorien määrä on äärellinen. Tässä tutkielmassa PRAM määritellään Gibbonsin [36] esitystä mukaillen.

MÄÄRITELMÄ 1. *PRAM-mallin mukainen tietokone* koostuu p prosessorista ja jaetusta muistista. Lisäksi jokaisella prosessorilla on oma paikallinen muistinsa. Prosessorit

toimivat suorittamalla synkronisia askeleita. Yhden askeleen aikana prosessori voi suorittaa jonkin seuraavista käskyistä:

- *Laskenta.* Yhden laskentaoperaation suorittaminen paikalliseen muistiin tallennetuilla tiedoilla ja tuloksen tallentaminen yhteen paikallisen muistin muistisijaintiin.
- *Lukeminen.* Tiedon lukeminen yhdestä jaetun muistin muistisijainnista yhteen paikallisen muistin muistisijaintiin.
- *Kirjoittaminen.* Tiedon kirjoittaminen yhdestä paikallisen muistin muistisijainnista yhteen jaetun muistin muistisijaintiin.

Jokaisen käskyn suorittaminen vie yhden aikayksikön.

Mallia on tarpeen täsmentää sen perusteella, kuinka se suhtautuu samanaikaiseen lukemiseen ja kirjoittamiseen [45]. Näin päädytään neljään erilaiseen PRAM-malliin: CRCW, CREW, ERCW ja EREW. Tässä kirjaimet R ja W viittaavat lukemiseen ja kirjoittamiseen (*Read* ja *Write*), C ja E puolestaan samanaikaisuuden sallimiseen tai kieltämiseen (*Concurrent* ja *Exclusive*).

Listauksessa 4.1 on pseudokoodiesimerkki kahden $n \times n$-matriisin kertolaskusta. Oletetaan, että käytössä on CRCW PRAM. Kun $i, j, k = 0, 1, \ldots n-1$, on mahdollisia kolmikon (i, j, k) arvoja yhteensä n^3, jolloin täyden rinnakkaisuuden saavuttaminen vaatii n^3 prosessoria. Sen sijaan sisimmän silmukan rungon kukin prosessori voi suorittaa vakioajassa eikä jaetun muistin lukeminen tai siihen kirjoittaminen aiheuta konflikteja. Näin ollen itse laskennan aikavaativuus on $O(1)$ ja prosessoreja tarvitaan $O(n^3)$. Jos vastaava laskutoimitus suoritettaisiin yhdellä prosessorilla, tilanne olisi päinvastainen: prosessorien määrä olisi $O(1)$ ja laskennan vaatima aika $O(n^3)$.

```
1 for i = 0 to n-1 pardo
2     for j = 0 to n-1 pardo
3         for k = 0 to n-1 pardo
4             C[i, j] += A[i, k] * B[k, j]
```

Listaus 4.1: Neliömatriisien kertolaskun suorittaminen CRCW PRAM -tietokoneella.

Esim. Culler ym. [17] kritisoivat mallia siitä, että prosessorit toimivat synkronisesti ja niiden välinen kommunikointi on ilmaista. Näitä ominaisuuksia hyödyntäen voidaan laatia hyvin tehokkaita algoritmeja, jotka kuitenkin osoittautuvat epärealistisiksi. Toisaalta Harrisin mukaan [45] PRAM on saanut suosiota juuri yksinkertai-

suutensa ja yleisyytensä vuoksi. Useita PRAM-algoritmeja on laadittu, ja katsauksessaan Harris käykin läpi simulointitekniikoita, joiden avulla PRAM-algoritmeja voidaan simuloida realistisemmilla malleilla.

Mallista on kehitetty useita muunnelmia, joskin jokainen seuraavissa alaluvuissa esitettävä malli on jo historiallisistakin syistä vähintään epäsuorasti PRAM:sta riippuvainen. Gibbons [36] kritisoi mallia siitä, että prosessorit toimivat täysin synkronisesti, ja esittää asynkronisen PRAM-mallin. Aggarwal ym. [3] puolestaan haluavat ottaa huomioon paitsi laskentaan myös kommunikointiin liittyvät kustannukset ja esittävät LPRAM-mallin (*Local-memory Parallel Random Access Machine*). LPRAM on CREW-malli, asynkroninen PRAM pikemminkin useasta mallista koostuva malliperhe.

4.2 QRQW PRAM

Kuten edellä on todettu, erilaiset PRAM-mallit voidaan jakaa karkeasti neljään luokkaan sen mukaan, kuinka ne suhtautuvat samanaikaiseen lukemiseen ja kirjoittamiseen. Gibbons ym. [38] väittävät kuitenkin, etteivät samanaikaisuuden pelkkä kieltäminen tai salliminen onnistu kuvaamaan reaalimaailman tietokoneita. He esittäväkin QRQW PRAM -mallin (*Queue-Read Queue-Write Parallel Random Access Machine*), joka sallii samanaikaisen lukemisen ja kirjoittamisen, mutta liittää niihin kustannuksen: samaan muistipaikkaan kohdistuvat luku- tai kirjoituspyynnöt menevät jonoon, ja yhtä pyyntöä palvellaan kerrallaan.

MÄÄRITELMÄ 2. *QRQW PRAM -mallin mukainen tietokone* koostuu äärellisestä määrästä (p kappaletta) prosessoreita ja jaetusta muistista. Lisäksi jokaisella prosessorilla on oma paikallinen muistinsa. Prosessorit toimivat suorittamalla synkronisia askeleita. Jokainen askel koostuu kolmesta aliaskeleesta, jotka ovat

- *lukualiaskel*: kukin prosessori P_i lukee r_i:stä jaetun muistin sijainnista
- *laskenta-aliaskel*: kukin prosessori P_i suorittaa c_i laskentaoperaatiota (ja käyttää tähän vain paikallista muistiaan)
- *kirjoitusaliaskel*: kukin prosessori P_i kirjoittaa w_i:hin jaetun muistin muistipaikkaan.

Jos useampi kuin yksi prosessori kirjoittaa askeleen aikana muistipaikkaan x, niistä jokin (sattumanvarainen) prosessori kirjoittaa muistipaikkaan arvon, joka on muistipaikassa, kun askel päättyy.

MÄÄRITELMÄ 3. Olkoon X QRQW PRAM -tietokoneen jaetun muistin muistipaikkojen joukko. Tarkastellaan yhtä tietokoneen askelta. Olkoon $r(x)$ muistipaikkaan $x \in X$ kohdistuvien lukuoperaatioiden ja $w(x)$ kirjoitusoperaatioiden määrä. Jos askeleen aikana tapahtuu vähintään yksi luku- tai kirjoitusoperaatio, askeleen *maksimikilpailu* on

$$\kappa = \max(\{ r(x) \mid x \in X \} \cup \{ w(x) \mid x \in X \}). \tag{4.1}$$

Mikäli askeleen aikana ei suoriteta yhtään luku- eikä kirjoitusoperaatiota, asetetaan $\kappa = 1$.

MÄÄRITELMÄ 4. Olkoon QRQW PRAM -tietokoneen maksimikilpailu askeleen aikana κ ja

$$m = \max_i(r_i, c_i, w_i). \tag{4.2}$$

Tällöin

- *askeleen aikakustannus* on $\max(m, \kappa)$
- QRQW PRAM -algoritmin *aika* on aikakustannusten summa
- QRQW PRAM -algoritmin *työ* on ajan ja prosessorien määrän tulo.

Määritelmä 2 saattaa herättää kysymyksen, miksi samanaikaiset kirjoitusoperaatiot sallitaan, vaikka niiden lopputulos on epävarma. Kirjoittajat toteavat, että monet satunnaisuuteen perustuvat algoritmit eivät voi täysin varmasti välttää samanaikaista kirjoittamista tai lukemista. Se on yksi syy siihen, miksi EREW on liian tiukka sääntö.

Mallin kehittäjät ovat määritelleet myös asynkronisen QRQW PRAM -mallin [37], jossa kukin prosessori etenee täysin omaan tahtiinsa eikä synkronointia ole, sekä QSM-mallin (*Queuing Shared-Memory*) [39]. Viimeksi mainittu muistuttaa QRQW PRAM -mallia, mutta ottaa huomioon myös sen, että jaetun muistin käyttö on kalliimpaa kuin paikallisen muistin käyttö. Tätä kuvataan mallissa parametrilla $g \geq 1$. Itse asiassa QRQW PRAM on QSM:n erikoistapaus, kun $g = 1$.

4.3 BSP

Valiant esittelee BSP-mallin [80] (*Bulk-Synchronous Parallel*) tarkoituksenaan tarjota rinnakkaislaskennalle ”silloittava” (Valiant: ”bridging”) malli, toisin sanoen standardi, josta kaikki voivat olla samaa mieltä. Edellä esitetyistä malleista poiketen

BSP on hajautetun muistin malli. Malli ei myöskään ole täysin synkroninen. Pikemminkin voitaisiin mallin nimeä mukaillen puhua lohkosynkronisesta mallista. Myöhemmässä artikkelissa Gerbessiotis ja Valiant [35] tarkentavat BSP:n määritelmää, ja seuraavassa otetaan nämä tarkennukset huomioon.

MÄÄRITELMÄ 5. *BSP-mallin mukainen tietokone* koostuu prosessoreista, joista kullakin on oma muistinsa, prosessorien välisestä tiedonsiirtoverkosta ja synkronointilaitteistosta. Laskennan suoritus koostuu *superaskeleista*. Yhden superaskeleen aikana kukin komponentti voi suorittaa paikallista laskentaa tai viestintää (lähettäminen ja vastaanottaminen) tai molempia. Prosessori voi käyttää vain sellaista dataa, joka sillä on jo ennen superaskeleen alkua. Tämän jälkeen tapahtuu synkronointi. Synkronointimekanismi voidaan kytkeä pois miltä tahansa prosessorien joukolta.

MÄÄRITELMÄ 6. BSP-mallin *parametrit* ovat L, g ja p. Niiden merkitys on seuraava:

- Kun aika L on kulunut, tarkistetaan, ovatko kaikki (synkronoitavat) prosessorit suorittaneet superaskeleen loppuun asti. Mikäli ovat, siirrytään seuraavaan superaskeleeseen, muussa tapauksessa varataan L aikayksikköä lisää.
- Verkko suorittaa *h-relaatioita*, joiden aikana kukin prosessori lähettää ja kullekin prosessorille lähetetään korkeintaan h viestiä. Yhden h-relaation aikakustannus on gh.
- Prosessorien määrä on p.

Yhden superaskeleen aikakustannus on siis luvun L monikerta. Valiant ja Gerbessiotis [35] antavat myös vaihtoehtoisen määritelmän, jonka mukaan superaskel voi päättyä koska tahansa ajan L jälkeen. Jos siis superaskeleen suorittaminen vaatisi enemmän kuin L aikayksikköä, ei olisi tarpeen odottaa L aikayksikköä lisää. Tässäkin tapauksessa L on yhteen superaskeleeseen kuluva vähimmäisaika. Parametrin L alarajan määrää laitteisto, sillä jo synkronoinnin toteuttaminen aiheuttaa oman kustannuksensa.

Valiant on myöhemmin esittänyt Multi-BSP-nimisen laajennoksen [81]. Mallissa on d tasoa, ja tasoa i kuvaa parametrinelikko (p_i, g_i, L_i, m_i). Tason i komponentin sisällä on p_i tason $i-1$ komponenttia. Parametri g_i on prosessorin laskentanopeuden suhde niiden sanojen määrään, jotka voidaan yhden sekunnin aikana siirtää tasojen i ja $i-1$ välillä. Superaskeleen aikana tason $i-1$ komponentit suorittavat ohjelmaa itsenäisesti, kunnes ne kohtaavat puomin. Puomisynkronointiin kuluva kustannus on L_i. Parametri m_i on i:nnen tason komponentin muistin suuruus, eikä siihen lasketa mukaan alempien tasojen muisteja.

Gerbessiotis [34] toteaa, että moniytimisten koneiden muistihierarkia täytyy huomioida, mutta pitää Multi-BSP:tä monimutkaisena ja esittää MBSP-mallin. Parametrit ovat (p, l, g, m, L, G, M). Ensimmäiset parametrit p, l ja g vastaavat alkuperäisen BSP-mallin parametreja p, L ja g. Ne kuvaavat prosessorien välistä kommunikointia. Kullakin prosessorilla on rajallinen määrä nopeaa muistia, jonka koko on M. Lisäksi on m yksikköä hitaampaa muistia, jonka I/O-operaatioihin liittyviä kustannuksia L ja G kuvaavat (vastaavasti kuin l ja g).

4.4 LogP

LogP on BSP:n tavoin hajautetun muistin malli. Se kiinnittää erityistä huomiota prossorien välisen kommunikaation kustannuksiin. Mallin parametrit käyvät ilmi mallin nimestä: L, o, g ja P.

Culler ym. [17] korostavat, että rinnakkaislaskennan mallin täytyy olla sekä tarpeeksi yksityiskohtainen että yksinkertainen: yhtäältä liian pieni määrä parametreja ei kuvaa tietokoneen toimintaa realistisesti, toisaalta liian yksityiskohtaista mallia on vaikeaa käyttää algoritmien suunnitteluun ja analysointiin. Kirjoittajien lähtökohta mallia suunniteltaessa oli BSP. He kuitenkin nostavat esille joitakin BSP:n ongelmia. Esimerkiksi superaskeleen alussa lähetettyjä viestejä voidaan käyttää vasta seuraavan superaskeleen aikana, vaikka superaskel olisikin viivettä pidempi. Lisäksi superaskeleen lopussa tapahtuva synkronointi vaatii synkronointilaitteiston. Culler ym. esittävätkin LogP-mallin, jossa edellä mainitut asiat ovat toisin. Koska prosessorit toimivat asynkronisesti, kukin prosessori voi käyttää viestiä heti, kun se saa viestin, ja synkronointi suoritetaan viestinvälityksen kautta.

MÄÄRITELMÄ 7. *LogP-mallin mukaisen tietokoneen* rakenne ja toiminta määritellään seuraavasti:

- Tietokone koostuu äärellisestä määrästä prosessoreita, joista jokaisella on oma muistinsa, ja prosessorit yhdistävästä kommunikaatioverkosta.
- Prosessorit kommunikoivat lähettämällä toisilleen viestejä. Kaikki viestit ovat lyhyitä.
- Prosessorit toimivat asynkronisesti.

MÄÄRITELMÄ 8. LogP-mallissa on seuraavat parametrit:

- L, latenssin eli viiveen yläraja.
- o, yleiskustannus, aika, joka prosessorilta kuluu yhden viestin lähettämiseen tai vastaanottamiseen. Tänä aikana prosessori ei voi tehdä mitään muuta.

- g, viestiväli, vähimmäisaika, joka kuluu kahden viestin vastaanottamisen tai lähettämisen välillä.
- P, prosessorien määrä.

L, o ja g mitataan prosessorisyklien monikertoina. Yhden prosessorin lähettämiä tai yhdelle prosessorille saapuvia viestejä voi olla yhdellä kertaa verkon kuljetettavana korkeintaan $\lceil L/g \rceil$.

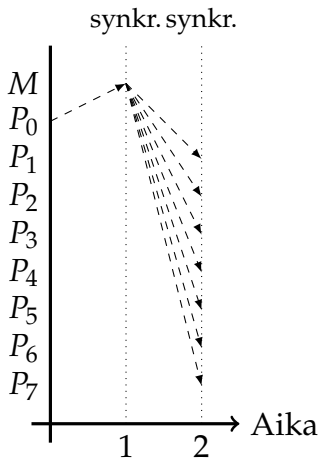
LogP-malliin on tehty useita laajennoksia. Alexandrov ym. [6] huomauttavat, että on edullisempaa koota useat lyhyet viestit yhdeksi pitkäksi viestiksi kuin lähettää useita lyhyitä viestejä, ja esittelevät LogGP-mallin. Uusi parametri G on pitkiin viesteihin liittyvä tavukohtainen väli. Pitkää viestiä lähetettäessä jokaisen lähetetyn tavun jälkeen on siis G:n mittainen väliaika.

Yan ym. [84] puolestaan huomauttavat, että suurten klustereiden tapauksessa täytyy ottaa huomioon myös hierarkia. He esittelevät LogGPH-mallin, joka laajentaa LogGP-mallia yhden parametrin verran. Uusi parametri h kertoo järjestelmän hierarkkisuusasteen. Oletetaan, että verkko koostuu noodeista, noodit prosessoreista ja prosessorit ytimistä. Tällöin $h = 3$. Kun saman prosessorin ytimet kommunikoivat keskenään, kyseessä on 1. tason kommunikointi, kun saman noodin, mutta eri prosessorin ytimet kommunikoivat, kyseessä on 2. tason kommunikointi, ja niin edelleen.

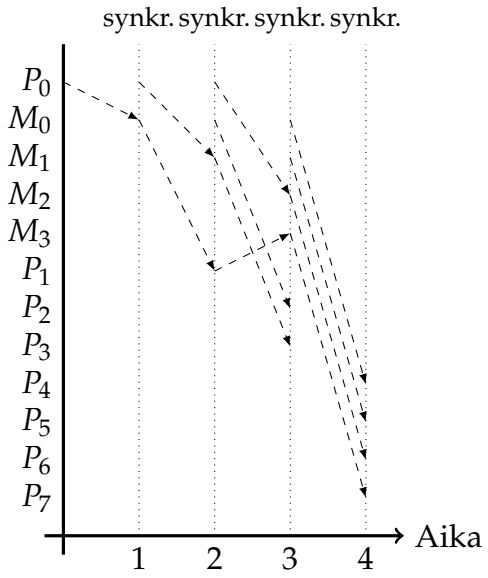
4.5 Yleislähetyksen aikakustannuksien mallintamisesta

Seuraavaksi vertaillaan, kuinka edellä esitellyt mallit suoriutuvat yleislähetyksestä. Oletetaan siis, että käytössä on prosessorit $P_0, P_1, \ldots, P_7$. Ajanhetkellä 0 prosessori P_0 alkaa lähettää dataa (lyhyen viestin) muille prossoreille. Tarkoitus on saada viesti perille mahdollisimman nopeasti. LogP:n kehittäjät Culler ym. [17] esittävät artikkelissaan algoritmin ja siihen liittyvän ajoituskaavion, muiden mallien tapauksessa algoritmi on kirjoittajan oma. Algoritmeihin liittyvät ajoituskaaviot on esitetty kuvassa 4.1.

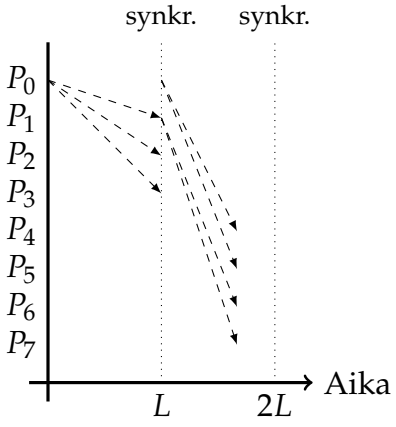
PRAM:n prosessorit kommunikoivat jaetun muistin kautta. CRCW ja CREW PRAM:n osalta algoritmi on yksinkertainen: ensimmäisen askeleen aikana P_0 kirjoittaa viestin jaettuun muistiin, ja seuraavan askeleen aikana muut prosessorit käyvät sen lukemassa. Itse asiassa aikaa kuluu 2 aikayksikköä riippumatta siitä, kuinka monelle prosessorille viesti pitää lähettää. Koska ERCW ja EREW PRAM:n prosessorit



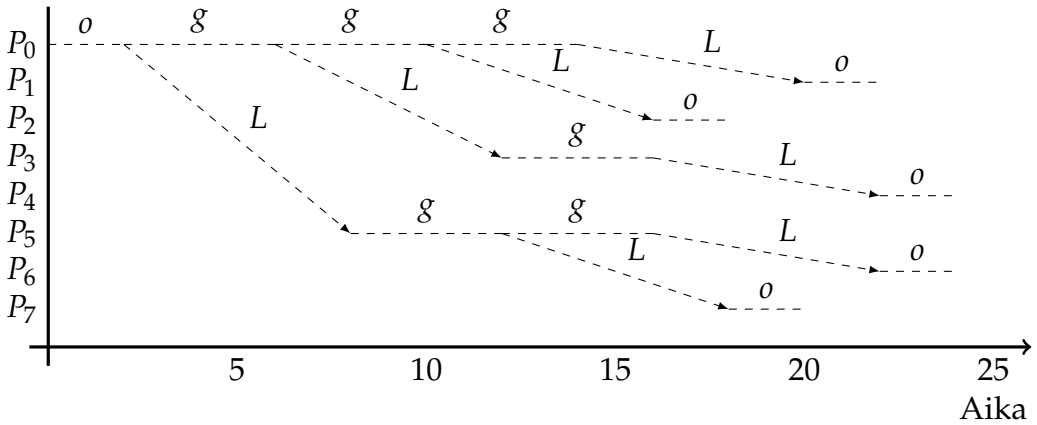
(a) CRCW ja CREW PRAM.



(b) ERCW, EREW ja QRQW PRAM.



(c) BSP.



(d) LogP.

Kuva 4.1: Yleislähetys. M ja M_i ovat jaetun muistin muistisijainteja. Kuva (d) on muokattu Cullerin ym. esittämästä kuvasta [17, kuva 3].

eivät voi lukea samaa jaetun muistin muistisijaintia samanaikaisesti, ne tarvitsevat 4 muistisijaintia ja aikaa kuluu 4 aikayksikköä. Vaikka QRQW PRAM antaa mahdollisuuden jonottaa, jonotusmahdollisuutta ei tässä ajan säästämiseksi käytetä. Siksi QRQW PRAM noudattaa tässä samaa algoritmia kuin ERCW ja EREW PRAM.

BSP:n tapauksessa viestien ajoitus riippuu parametrien arvosta. Oletetaan tässä esimerkissä, että $L = 3g$. Oletetaan myös, että kaikki prosessorit osallistuvat synkronointiin. Jotta ensimmäinen superaskel ehdittäisiin suorittaa ajassa L, voi P_0 lähettää viestin vain kolmelle prosessorille. Kun toinen superaskel alkaa, viesti on neljällä prosessorilla, joten yleislähetys saadaan suoritetuksi superaskeleen aikana. Vaihtoehtoisia strategioita on useita. Yksi mahdollisuus on esitetty kuvassa 4.1c. Tärkeintä on, ettei yksi prosessori lähetä viestiä neljälle prosessorille, koska tällöin aikaa kuluisi $4g > L$ aikayksikköä.

LogP:n osalta esimerkki noudattaa Cullerin ym. [17] esitystä. Oletetaan yksinkertaisuuden vuoksi, että viive on aina maksimaalinen L ja $g \geq o$. Prosessori P_0 aloittaa lähettämisen hetkellä 0, ja verkko saa viestin kuljetettavakseen hetkellä o. Viesti kulku kestää L aikayksikköä, ja tämän jälkeen vastaanottaja käyttää viestin vastaanottamiseen o aikayksikköä, joten vastaanottaja on vastaanottanut viestin hetkellä $L + 2o$. Sillä aikaa kun viesti on verkon kuljetettavana, P_0 voi aloittaa uuden viestin lähettämisen. Sen täytyy ensin odottaa aika g, ja viestin lähettäminen vie ajan o, mutta ajat menevät päällekkäin, ja koska $g \geq o$, toinen P_0:n lähettämä viesti on valmis verkon kuljetettavaksi hetkellä $o + g$, kolmas hetkellä $o + 2g$ ja niin edelleen. Kuvassa 4.1d on esitetty ajoituskaavio, kun $L = 6$, $o = 2$, $g = 4$ ja $P = 8$. Tässä tapauksessa yhden prosessorin lähettämiä viestejä saa olla verkon kuljetettavana kerrallaan $\lceil L/g \rceil = 2$ kappaletta. Viimeinenkin prosessori on vastaanottanut viestin hetkellä 24.

4.6 Yhteenvetoa teoreettisista malleista

Edellä esitettyjä malleja voidaan verrata ja luokitella monella tapaa. Taulukossa 4.1 malleja vertaillaan muistimallin, synkronisuuden ja parametrien suhteen. Synkronisuuden osalta kukin malli luokitellaan synkroniseksi, lohkosynkroniseksi tai asynkroniseksi. PRAM-tietokoneen prosessorit suorittavat askeleita täysin tasatahtiin, mutta BSP sallii prosessorien tehdä synkronointien välillä useita toimenpiteitä. Samaa voidaan sanoa QRQW PRAM:sta. Silti sekä BSP että QRQW PRAM edellyttävät

kaikkien prosessorien globaalia synkronointia, ja tämän suhteen LogP poikkeaa muista käsitellyistä malleista.

Taulukko 4.1: Esitettyjen mallien vertailua. BSP:n ja LogP:n parametrit L ja g eivät tarkoita samaa.

Nimi	Muistimalli	Synkronisuus	Parametrit
PRAM [32, 36]	Jaettu muisti	Synkroninen	p
QRQW PRAM [38]	Jaettu muisti	Lohkosynkroninen	p
BSP [80, 35]	Hajautettu muisti	Lohkosynkroninen	L, g, p
LogP [17]	Hajautettu muisti	Asynkroninen	L, o, g, P

Rico-Gallego ym. [74] pitävät LogP:tä tärkeänä, koska se on antanut perustan useille laajennoksille, joista luvussa 4.4 on lyhyesti esitelty LogGP ja LogGPH. Kirjoittajien mukaan mallien kehitykselle on ollut tyypillistä parametrien lisääntyminen. Tähän on vaikuttanut suurteholaskennassa käytettävien laskentaympäristöjen kehitys. Voidaan tietenkin kysyä, mikä on optimaalinen parametrien määrä, kun otetaan huomioon, että mallin täytyy saavuttaa toimiva kompromissi yhtäältä yksinkertaisuuden ja toisaalta yksityiskohtaisuuden väliltä.

5 Rinnakkaislaskennan ohjelmointiympäristöjä

Rinnakkaisohjelman toteuttaminen vaatii konkreettisen ohjelmointirajapinnan tai -ympäristön. Kuten luvussa 3 todettiin, suurin osa nykyisistä rinnakkaistietokoneista voidaan luokitella hajautetun ja jaetun muistin tietokoneisiin. Sekä hajautettua että jaettua muistia käytetään jatkossa, joten tässä luvussa tutustutaan MPI:hin, joka on hajautetun muistin osalta de facto -standardi, ja OpenMP:hen, joka puolestaan on jaetun muistin de facto -standardi [48]. MPI:hin perehdytään luvussa 5.1 ja OpenMP:hen luvussa 5.2. Luvussa 5.3 tarkastellaan MPI:n ja OpenMP:n suorituskyvyn eroa ja muita seikkoja, jotka vaikuttavat siihen, kumpaa kannattaa käyttää.

5.1 MPI

Standardin mukaan [65, s. 1] MPI (*Message-Passing Interface*) on viestinvälityskirjaston määrittelevä spesifikaatio. Klassisessa viestinvälityksessä on kysymys kahden prosessorin välisestä tiedonsiirrosta, mutta MPI laajentaa tätä klassista mallia esim. kollektiivisella viestinnällä. Spesifikaatio on määritelty C:lle ja Fortranille. Tässä tutkielmassa keskitytään kuitenkin vain C-kieleen. Seuraavissa alaluvuissa perehdytään MPI:n peruskäyttöön sekä muihin jatkon kannalta merkityksellisiin MPI:n ominaisuuksiin.

5.1.1 Peruskäyttö ja kahdenvälinen viestintä

Tarkastellaan yksinkertaista esimerkkiohjelmaa Haatajan ja Mustikkamäen esitystä [43, s. 11–17] mukaillen. Koodi on esitetty listauksessa 5.1. Esikääntäjän direktiivillä `#include<mpi.h>` saadaan käyttöön MPI:n määrittelemiä tietotyyppejä ja vakioita. Varsinainen rinnakkaislaskenta aloitetaan rivillä 11 kutsumalla funktiota `MPI_Init`. Funktio palauttaa tiedon siitä, onnistuiko rinnakkaislaskennan käynnistäminen. Rivillä 16 tallennetaan prosessien lukumäärä muuttujaan `ntasks` ja rivillä 17 kunkin prosessin järjestysnumero muuttujaan `id`. Tässä `MPI_COMM_WORLD` on valmiiksi määritelty viestintäryhmä, johon kuuluvat kaikki laskentaan osallistuvat prosessit. Riviltä 18 alkaen ohjelman suoritus haarautuu prosessin järjestysnumeron mukaan. Jos järjestysnumero ei ole 0, prosessi kokoaa viestin, joka koostuu

sen järjestysnumerosta ja prosessien määrästä, ja lähettää sen prosessille 0. Jos taas prosessin järjestysnumero on 0, prosessi ottaa vastaan jokaisen sille lähetetyn viestin, selvittää sen lähettäjän `status`-muuttujan tiedoista ja tulostaa viestin tiedot. Rivillä 34 kutsutaan funktiota `MPI_Finalize`, joka lopettaa rinnakkaislaskennan.

Koodi on esimerkki kahdenvälisestä viestinnästä [43, s. 17–22]. Lähettämisessä käytettävän `MPI_Send`-funktion kutsumuoto on

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Parametri `buf` kertoo lähetettävän datan, `count` alkioiden lukumäärän, `datatype` tyypin ja `dest` vastaanottajan järjestysnumeron. Parametri `tag` on vapaavalintainen tunnistenumero, jonka avulla eri asiaa tarkoittavat viestit voidaan erottaa toisistaan. Viimeinen parametri `comm` on viestintäryhmä. Esimerkissä käytetään valmista viestintäryhmää `MPI_COMM_WORLD`, eikä tässä tutkielmassa perehdytä viestintäryhmien luomiseen. Parametri on kuitenkin sikäli tärkeä, että vain samaan viestintäryhmään kuuluva prosessi voi ottaa viestin vastaan. Vastaanottavan funktion kutsumuoto on

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status status)
```

Parametri `source` kertoo lähettäjän järjestysnumeron. Esimerkissä käytetään parametrin arvona vakiota `MPI_ANY_SOURCE`, joka tarkoittaa, että viesti voidaan ottaa vastaan miltä tahansa prosessilta. Viimeisen parametrin `status` kautta välitetään mm. lähettäjän järjestysnumero.

Listauksen 5.1 ohjelmasta on syytä huomata, että prosessi 0 on erikoisasemassa. Vaikka kyseessä onkin yksi ohjelma, sen suoritus haarautuu prosessin järjestysnumeron mukaan. Tämä on Pachecon ja Malensekin mukaan [72, s. 94] useimmille MPI-ohjelmille tyypillinen ominaisuus.

5.1.2 Tulostus ja syöttö

Listauksen 5.1 koodissa vain prosessi 0 tulostaa jotakin. Kuitenkin Pachecon ja Malensekin mukaan [72, s. 105–107] käytännössä kaikki MPI-toteutukset sallivat jokaisen prosessin päästä kirjoittamaan tietovirtoihin `stdout` ja `stderr`. Tästä ei vielä seuraa, että järjestykseen olisi mahdollista vaikuttaa. Jos useampi prosessi yrittää

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    const int tag = 50;
    int id, ntasks, source_id, dest_id, rc, i;
    MPI_Status status;
    int msg[2];

    rc = MPI_Init(&argc, &argv);
    if (rc != MPI_SUCCESS) {
        printf("MPI:n alustus epäonnistui.\n");
        exit(1);
    }
    rc = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if (id != 0) {
        msg[0] = id;
        msg[1] = ntasks;
        dest_id = 0;
        rc = MPI_Send(msg, 2, MPI_INT, dest_id,
                      tag, MPI_COMM_WORLD);
    } else {
        for (i = 1; i < ntasks; i++) {
            rc = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE,
                          tag, MPI_COMM_WORLD, &status);
            source_id = status.MPI_SOURCE;
            printf("viesti: %d %d, lähettäjä: %d\n",
                   msg[0], msg[1], source_id);
        }
    }
    rc = MPI_Finalize();
    return 0;
}
```

Listaus 5.1: Yksinkertainen MPI-ohjelma. Muokattu Haatajan ja Mustikkamäen esittämästä koodista [43, listaus 2.2].

kirjoittaa ulostulovirtaan `stdout`, ei ole mahdollista ennustaa, missä järjestyksessä tämä tapahtuu.

Toisaalta, kuten Pacheco ja Malensek [72, s. 107–108] toteavat, useimmat MPI-toteutukset sallivat vain prosessin 0 päästä lukemaan tietovirtaa `stdin`. Heidän mukaansa tämä on järkevää, koska jos useampi prosessi pääsisi lukemaan syötevirtaa, ei olisi selvää, kuinka syöte jaettaisiin prosessien kesken. Siksi on käytännöllistä, että vain prosessi 0 lukee syötteen ja jakaa sen muille prosesseille tarpeen mukaan.

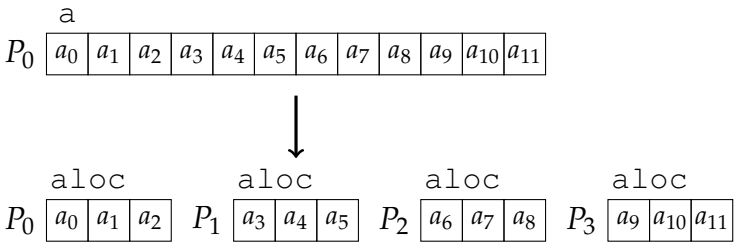
5.1.3 Kollektiivisesta viestinnästä

Kuten edellä nähtiin, kahdenvälisessä viestinnässä vastaanottaja ja lähettäjä kutsuvat eri funktioita. Kollektiivisessa viestinnässä sen sijaan kaikki prosessit kutsuvat samaa kollektiivista funktiota samoilla parametreilla [43, s. 24]. Silti kahdenvälisen viestinnän tapaan yleensä jokin prosessi on erikoisasemassa.

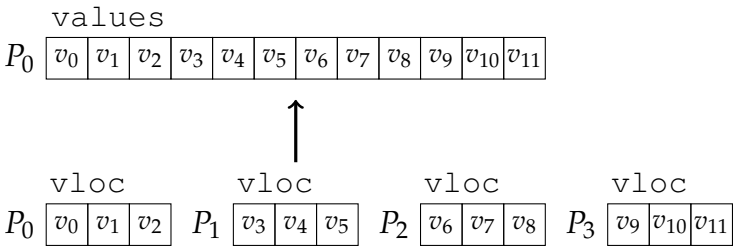
```
1  if (id == root_id) {
2      asize = ntasks * local_size;
3      a = (double *)malloc(asize*sizeof(*a));
4      values = (double *)malloc(asize*sizeof(*values));
5      for (i = 0; i < asize; i++)
6          a[i] = 3*i/((double)asize-1);
7  }
8
9  rc = MPI_Scatter(a, local_size, MPI_DOUBLE,
10                  aloc, local_size, MPI_DOUBLE,
11                  root_id, MPI_COMM_WORLD);
12 for (i = 0; i < local_size; i++)
13     vloc[i] = sin(aloc[i]);
14
15 rc = MPI_Gather(vloc, local_size, MPI_DOUBLE,
16                 values, local_size, MPI_DOUBLE,
17                 root_id, MPI_COMM_WORLD);
18
19 if (id == root_id) {
20     // Tulosten käsittely.
21 }
```

Listaus 5.2: Esimerkki funktioiden `MPI_Scatter` ja `MPI_Gather` käytöstä. Lyhennetty ja muokattu Haatajan ja Mustikkamäen esittämästä koodista [43, listaus 3.4].

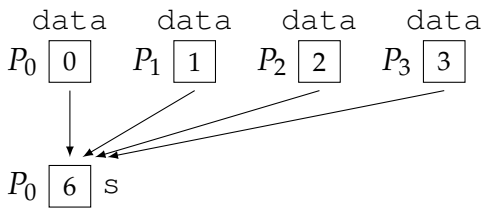
Listauksessa 5.2 on esimerkki funktioiden `MPI_Scatter` ja `MPI_Gather` käytöstä [43, s. 29–33]. Esimerkkiä on lyhennetty jättämällä pois mm. rinnakkaislaskennan aloitus ja lopetus sekä muuttujien määrittely. Riveillä 1–7 prosessi, jonka järjestysnumero on `root_id`, varaa muuttujille `a` ja `values` tilaa sekä tallentaa `a`:n muistipaikkoihin liukulukuja. Riveillä 9–11 kutsutaan funktiota `MPI_Scatter`. Funktio levittää prosessin `root_id` lähetyspuskuriin `a` tallennetut alkiot kaikille prosesseille. Kukin prosessi tallentaa vastaanottamansa alkiot vastaanottopuskuriinsa `a_local`. Prosessi 0 saa ensimmäiset `local_size` alkiota, prosessi 1 seuraavat `local_size` alkiota ja niin edelleen. Funktion toimintaa on havainnollistettu kuvassa 5.1a. Kuvan tilanteessa prosesseja on 4, paikallisen vastaanottopuskurin koko on 3 ja `root_id` on 0. Vaikka vain prosessin `root_id` täytyy varata lähetyspuskurille `a` tilaa, on kaikkien prosessien annettava tämäkin parametri funktiota `MPI_Scatter` kutsuttaessa. Lisäksi huomionarvoista on, että prosessi `root_id` lähettää dataa myös itselleen.



(a) `MPI_Scatter`.



(b) `MPI_Gather`.



(c) `MPI_Reduce`.

Kuva 5.1: Esimerkki funktioiden `MPI_Scatter`, `MPI_Gather` ja `MPI_Reduce` toiminnasta. Kuva (a) on muokattu Haatajan ja Mustikkamäen esittämästä kuvasta [43, kuva 3.4].

Riveillä 12–13 kukin prosessi käsittelee saamaansa dataa ja tallentaa laskennan tulokset lähetyspuskuriin `vloc`. Riveillä 15–17 puolestaan kutsutaan `MPI_Gather`-funktiota, joka kerää laskennan tulokset prosessille `root_id` [43, s. 31]. Tämä funktio toimii `MPI_Scatter`-funktioon nähden käänteisesti. Tässä tapauksessa kunkin prosessin lähetyspuskuriin `vlocal` tallennetut liukuluvut kerätään prosessin `root_id` vastaanottopuskuriin `values`. Funktion toimintaa on havainnollistettu kuvassa 5.1b.

Funktio `MPI_Reduce` [43, s. 24–28] suorittaa myös laskentaa. Jos esim. kullakin prosessilla on kokonaisluku tallennettuna muuttujaan `data`, saadaan lukujen summa prosessin 0 muuttujaan `s` funktiokutsulla

```
rc = MPI_Reduce(&data, &s, 1, MPI_INT, MPI_SUM, 0,
                MPI_COMM_WORDL);
```

Esimerkkiä on havainnollistettu kuvassa 5.1c, jossa muuttujaan `data` on tallennettu prosessin järjestysnumero. Funktion kolmantena parametrina annetaan yhden prosessin lähettämien alkioiden lukumäärä, joka tässä tapauksessa on 1. Reduktio-operaatio on tässä esimerkissä `MPI_SUM`, joka laskee alkioiden summan. Muita vaihtoehtoja ovat esim. `MPI_PROD`, joka laskee tulon, sekä `MPI_MAX` ja `MPI_MIN`, joilla saadaan selville maksimi ja minimi.

Kollektiiviseen viestintään kuuluu myös laskennan tahdistus eli synkronointi [43, s. 36–37]. Funktion kutsumuoto on

```
MPI_Barrier(MPI_Comm comm)
```

Prosessit jatkavat ohjelman suoritusta vasta, kun kaikki ovat kutsuneet kyseistä funktiota.

5.2 OpenMP

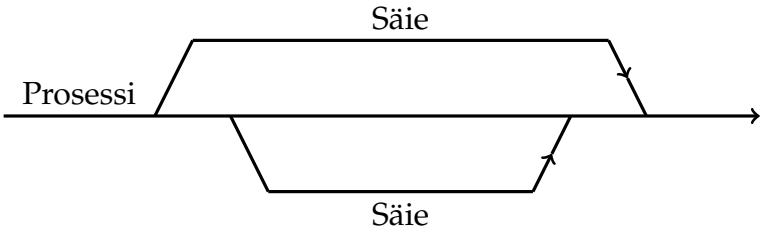
Pachecon ja Malensekin mukaan [72, s. 221–222] OpenMP (*Open Multi-Processing*) on jaetun muistin sovellusohjelmointirajapinta. Sitä suunniteltaessa on tavoitteena ollut mahdollistaa jo olemassa olevan peräkkäisen koodin rinnakkaistaminen askel askeleelta. MPI:llä tämä on käytännössä mahdotonta, mutta OpenMP:llä erityisesti for-silmukoiden rinnakkaistaminen vaatii vain vähäisiä muutoksia lähdekoodiin. Toinen merkittävä ero MPI:hin nähden on, että MPI:n käyttö vaatii tarvittavan kirjaston, OpenMP:n käyttö lisäksi ohjelmointikielen kääntäjän, joka tukee OpenMP:tä.

Seuraavissa alaluvuissa käydään läpi OpenMP:n perusominaisuuksia. Standardi [71, s. 1] on määritelty C:lle, C++:lle ja Fortranille, mutta vain C-kieleen liittyviä asioita käsitellään.

5.2.1 Perusteita

Tutustutaan yksinkertaiseen perusohjelmaan Pachecon ja Malensekin esitystä [72, s. 222–228] mukaillen. Koodi on esitetty listauksessa 5.3. Rivillä 3 otetaan käyttöön OpenMP:n tarvitsema otsikkotiedosto `omp.h`. Rivillä 10 oleva direktiivi informoi esikääntäjää siitä, että sitä seuraava koodilohko on tarkoitus rinnakkaistaa. Koodilohkossa kutsutaan funktiota `Hello()`, joka kirjoittaa ulostulovirtaan `stdout`. OpenMP:n tapauksessa rinnakkaistaminen tarkoittaa säikeiden käyttöä, ja tässä käytettävien säikeiden määräksi asetetaan muuttujaan `thread_count` tallennettu luku, joka on saatu komentoriviargumenttina. Säikeiden määrää ei ole kuitenkaan välttämätöntä antaa; jos sitä ei anneta, sen määrää ajoympäristö. Tällöin tyypillisesti säikeiden määrä on sama kuin prosessoriytimien määrä. Lisäksi ajoympäristöllä voi olla omat rajoituksensa säikeiden määrän suhteen, joten ei ole takeita, että säikeitä saadaan käyttöön juuri `thread_count` kappaletta.

Pachecon ja Malensekin mukaan [72, s. 222–228] ennen `parallel`-direktiiviä ohjelmaa suorittaa prosessi, jossa on vain yksi säie. Kun suoritus etenee `parallel`-direktiiviin, prosessi haarautuu useammaksi säikeeksi. Kun säie lopettaa ohjelman suorittamisen, se yhdistyy takaisin prosessiin, josta se aiemmin haarautui. Haarautumista ja yhdistymistä on havainnollistettu kuvassa 5.2. Säikeet jakavat suuren osan prosessin resursseista, ja tämä koskee myös tietovirtoja `stdin` ja `stdout`. Siksi jokainen säie pystyy tulostamaan. Tulostusjärjestys on kuitenkin epädeterminististinen.



Kuva 5.2: Kaksi säiettä haarautuvat prosessista ja yhdistyvät siihen takaisin. Muokattu Pachecon ja Malensekin esittämästä kuvasta [72, kuva 5.2].

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4
5  void Hello(void);
6
7  int main(int argc, char* argv[]) {
8      int thread_count = strtol(argv[1], NULL, 10);
9
10 #    pragma omp parallel num_threads(thread_count)
11     Hello();
12
13     return 0;
14 }
15
16 void Hello(void) {
17     int my_rank = omp_get_thread_num();
18     int thread_count = omp_get_num_threads();
19
20     printf("Terveiset säikeestä %d (%d)\n",
21            my_rank, thread_count);
22 }
```

Listaus 5.3: Yksinkertainen OpenMP-ohjelma. Muokattu Pachecon ja Malensekin koodin pohjalta [72, ohjelma 5.1].

5.2.2 Kilpailutilanteista

Resurssien jakaminen voi johtaa kilpailutilanteisiin. Tarkastellaan Pachecon ja Malensekin antamaa [72, s. 229–232] yksinkertaista esimerkkiä. Oletetaan, että muuttuja `global_result` on kaikkien säikeiden jakama ja sen lisäksi jokaisella säikeellä on yksityinen muuttuja `my_result`. Tarkoitus on, että jokainen säie suorittaa yhteenlaskun

```
global_result += my_result;
```

Ongelmana on kuitenkin, ettei tulos ole välttämättä oikein. Kukin säie nimittäin tallentaa laskentaa varten yhteisen muuttujan `global_result` arvon omaan rekisteriinsä. Tällöin jokin säie saattaa muuttaa yhteisen muuttujan arvoa ilman, että muut säikeet tietävät asiasta, ja muut säikeet laskevat yhteenlaskun vanhentunutta

muuttujan arvoa käyttäen. Ongelma voidaan välttää suojaamalla koodi `critical`-direktiivillä:

```
#   pragma omp critical
    global_result += my_result;
```

Nyt vain yksi säie voi kerrallaan suorittaa koodia.

Muita mahdollisuuksia ovat Pachecon ja Malensekin mukaan [72, s. 256–264] `atomic`-direktiivi ja lukot. Kaikissa menetelmissä on omat hyvät ja huonot puolensa. Lukitus on monipuolisin, mutta hitain vaihtoehto, `atomic` taas hyvin tehokas, mutta käyttöominaisuuksiltaan rajoittunut. Sen sijaan `critical` sijoittuu sekä tehokkuuden että monipuolisuuden suhteen lukituksen ja `atomic`-direktiivin väliin.

5.2.3 Silmukoiden rinnakkaistamisesta

OpenMP mahdollistaa for-silmukoiden rinnakkaistamisen [72, s. 237–244]. Tämä saadaan aikaiseksi `parallel for` -direktiivillä. Yksinkertaisimmillaan rinnakkainen silmukka on muotoa

```
#   pragma omp parallel for num_threads(thread_count)
    for (i = 0; i < n; i++)
        // Silmukan runko.
```

Kun ohjelman suoritus saapuu `parallel for` -direktiiviin, prosessista haarautuu `thread_count` -1 uutta säiettä ja silmukan kierrokset jaetaan säikeiden kesken. Rinnakkaistamisella on rajoitteensa. OpenMP voi rinnakkaistaa vain for-silmukoita, ja for-silmukan kierrosluvun täytyy olla tiedossa jo ennen silmukan suorittamista. Luonnollisesti myös silmukan kierrosten täytyy olla toisistaan riippumattomia, jotta rinnakkaistaminen olisi mielekästä.

Oma kysymyksensä on, kuinka sisäkkäisten silmukoiden kanssa tulee menetellä. Zheng ym. [85] tarkastelevat asiaa sekä analyyttisesti että kokeellisesti ja päätyvät siihen lopputulokseen, että vain uloin silmukka kannattaa rinnakkaistaa. Aldinucci ym. [5] suosittelevat samaa strategiaa yleissäännöksi, mutta antavat ymmärtää, ettei sääntö ole ehdoton. Jos silmukan kierrosluku on pieni, yhden tai useamman sisemmän silmukan rinnakkaistaminen saattaa johtaa parempaan suorituskykyyn. Handhika ym. [44] puolestaan esittävät kahdelle sisäkkäiselle silmukalle esikäsittelyalgoritmin, jonka avulla vähän kokemusta omaavan ohjelmoijan pitäisi pystyä päättelemään, kannattaako ulompi vai sisempi silmukka rinnakkaistaa.

5.2.4 Säieturvallisuudesta

Koodi on Pachecon ja Malensekin mukaan [72, s. 274–277] säieturvallinen, jos useampi kuin yksi säie voi suorittaa sen samanaikaisesti ilman ongelmia. Heidän mukaansa säieturvallisuuden puuttuminen C:n funktioista ei ole tavatonta. Joissain tapauksissa tällaisesta funktiosta on kuitenkin olemassa myös vaihtoehtoinen, säieturvallinen versio. Yksi esimerkki ei-säieturvallisesta funktiosta on otsikkotiedostossa `string.h` esitelty `strtok`, jota käytetään merkkijonon paloitteluun. Funktio tallentaa osoittimen paloiteltavaan merkkijonoon. Osoitin on kuitenkin kaikille säikeille yhteinen, mikä johtaa kilpailutilanteisiin. Tästä funktiosta on tosin olemassa myös säieturvallinen versio nimeltä `strtok_r`.

Säieturvallisuuden puuttuminen tulee esille myös Barreton ja Bauerin tutkimuksessa [10]. Kirjoittajat rinnakkaistivat lineaarisessa sekalukuoptimoinnissa käytetyn *branch and bound* -algoritmin sekä MPI:llä että OpenMP:llä. Vaikka OpenMP vaikuttikin lupaavalta osaongelmien ratkaisemisessa, kokonaisuutena se suoriutui silti MPI:tä heikommin. Käytetty LP-ratkaisija ei nimittäin ollut säieturvallinen, mikä monimutkaisti algoritmin toteuttamista ja heikensi OpenMP:n suorituskykyä.

5.3 Suorituskyvystä ja muista valintaan vaikuttavista seikoista

Kang ym. [48] vertailivat OpenMP:n, MPI:n ja MapReducen suorituskykyä. MapReduce [18] on suurten datamäärien käsittelyyn soveltuva ohjelmointimalli. Toinen Kangin ym. käyttämistä testiongelmista oli hyvin dataintensiivinen, ja siitä MapReduce suoriutuikin MPI:tä ja OpenMP:tä paremmin. Toinen ongelma puolestaan oli verkkoteorian alaan kuuluva kaikkien lyhinten polkujen ongelma. Siinä MapReduce pärjäsi MPI:hin ja OpenMP:hen verrattuna huomattavasti huonommin. Tämän tutkielman kannalta mielenkiintoisin on kuitenkin MPI:n ja OpenMP:n keskinäinen marssijärjestys. MPI:tä käytettiin sekä viiden tietokoneen klusterissa että yksittäisessä tietokoneessa, OpenMP:tä pelkästään yhdessä tietokoneessa. Huomattavaa on, että ongelman koosta riippumatta OpenMP suoriutui MPI:tä paremmin. Lisäksi ongelman koon kasvaessa MPI:tä käyttänyt viiden tietokoneen klusteri oli hitaampi kuin MPI:tä käyttänyt yksittäinen tietokone. Kirjoittajien mukaan klusterin tietokoneiden välinen kommunikointi osoittautui pullonkaulaksi, vaikka koneet olikin yhdistetty 1 Gbps:n kytkimellä. Tutkimuksen perusteella kirjoittajat suosittelevat OpenMP:tä, jos ongelma on riittävän pieni ja yhden tietokoneen resurssit riit-

tävät. Jos taas ongelma on suurempi ja vaatii paljon laskentaa, MPI on hyvä vaihtoehto. Jälkimmäinen johtopäätös antaa ymmärtää, että jos ongelmaa olisi kasvatettu tarpeeksi paljon, MPI olisi jossain vaiheessa osoittautunut OpenMP:tä tehokkaammaksi. Tätä ei kuitenkaan tutkimuksessa kokeellisesti osoitettu.

Barreton ja Bauerin tutkimus [10] on mainittu jo edellä. Sekalukuoptimoinnissa käytettävä *branch and bound* -algoritmi rinnakkaistettiin sekä MPI:llä että OpenMP:llä. Koska algoritmissa käytetty LP-ratkaisija ei ollut säieturvallinen, ongelma jouduttiin kiertämään ratkaisuilla, jotka hidastivat OpenMP:tä. MPI ei tästä ongelmasta kärsinyt, ja se olikin lopulta nopeampi kuin OpenMP. Huonomman suorituskyvyn lisäksi on otettava huomioon inhimilliset seikat. Tässä tapauksessahan algoritmin toteuttaminen OpenMP:llä vaati ylimääräistä vaivannäköä. Barreto ja Bauer kirjoittavatkin saaneensa myönteisen kuvan MPI:n toimivuudesta suhteessa OpenMP:hen.

Mallón ym. [61] vertailivat MPI:n, OpenMP:n ja UPC:n suorituskykyä useilla erilaisilla testiongelmilla. UPC (*Unified Parallel C*) [79] on C-kielen laajennos, joka toimii hajautetun muistin ympäristöissä, vaikka ohjelmoija näkeekin yhden globaalin osoiteavaruuden. MPI:tä ja UPC:tä käytettiin sekä hajautetun että jaetun, OpenMP:tä vain jaetun muistin ympäristössä. Kummassakin tapauksessa MPI osoittautui yleisesti ottaen tehokkaimmaksi, vaikka tästä säännöstä joitain poikkeuksia olikin. Muutenkin kirjoittajat pitävät MPI:tä parhaana vaihtoehtona. Heidän mukaansa jaettuun muistiin rajoittuminen heikentää OpenMP:n skaalautuvuutta. Lisäksi OpenMP ei yleensä tue datan paikallisuuden huomiointia.

Myös MPI:n ja OpenMP:n yhdistäminen on mahdollista [19]. Toteutustapoja on useita, mutta karkeasti kuvattuna tällainen ohjelma on MPI-ohjelma, jossa kunkin prosessin suorittamaan peräkkäiskoodiin on lisätty OpenMP:n direktiivejä. Cabral ym. [11] toteuttivat kolme erilaista osittaisdifferentiaaliyhtälön ratkaisumenetelmää. Vertailussa olivat mukana tavallinen OpenMP, OpenMP:n parannettu versio OMP-EWS (*OpenMP-Explicit Work-Sharing*), MPI ja MPI/OMP-EWS-hybridi. Sekä tavallinen OpenMP että OMP-EWS kärsivät synkronointiin liittyvistä kustannuksista ja suoriutuivat heikosti. Kun algoritmi sisälsi usean synkronointipisteen, MPI/OMP-EWS-hybridi osoittautui tehokkaimmaksi vaihtoehdoksi, MPI taas ohitti muut, kun käytössä oli yksi synkronointipiste.

Kysymykseen MPI:n ja OpenMP:n keskinäisestä paremmuudesta ei edellä referoitujen tutkimusten perusteella voida antaa yksinkertaista vastausta. Algoritmit, laitteistot ja muut suorituskykyyn vaikuttavat tekijät ovat erilaisia. Käytännössä suorituskyvyn lisäksi ohjelmointiympäristön valintaan vaikuttavat myös inhimil-

liset syyt. Cabral ym. [11] toteavat, että OpenMP on suosittu juuri helppokäyttöisyytensä vuoksi, mutta huomauttavat, ettei tehokkaan OpenMP-ohjelman tekeminen ole yksinkertaista. Erityisesti Kang ym. [48] korostavat OpenMP:n helppoutta. Heidän mukaansa peräkkäiskoodin rinnakkaistaminen vaatii vain tarvittavien direktiivien lisäämistä eikä ohjelmoijan tarvitse ymmärtää monisäikeistystä erityisen syvällisesti. Näkemys ei ole täysin perusteeton, mutta sitä voidaan kyllä kritisoida liiasta optimismista tai ainakin yksipuolisuudesta. Gonçalves ym. [40] huomauttavat, että täysin triviaaleja ohjelmia lukuun ottamatta oikein toimivan ja suorituskykyisen OpenMP-koodin kirjoittaminen edellyttää rinnakkaislaskennan peruskäsitteiden hallintaa.

6 Matriisikertolaskun rinnakkaistamisesta

Koska tarkasteltavaksi laskentatehtäväksi on valittu matriisikertolasku, seuraavaksi perehdytään matriiseihin ja matriisikertolaskun rinnakkaistamiseen. Luvussa 6.1 käsitellään matriiseja ja matriisilaskentaa. Luku on suppea, sillä vain jatkon kannalta merkitykselliset asiat käsitellään. Luvussa 6.2 puolestaan tutustutaan joihinkin tunnettuihin rinnakkaisalgoritmeihin.

6.1 Matriiseista

Matriisi [54, s. 23–27] on kaksiulotteinen reaaliluvuista koostuva taulukko. Matriisia, jossa on m riviä ja n saraketta, sanotaan $m \times n$-matriisiksi. Esim. 2×3-matriisi A voidaan kirjoittaa alkioittain muodossa

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}, \tag{6.1}$$

lyhyemmin $A = (a_{ij})$, missä i on rivin ja j sarakkeen indeksi. Huomautettakoon, että useista matemaattisista esityksistä poiketen tässä tutkielmassa indeksointi aloitetaan luvusta 0. Tarvittaessa rivin ja sarakkeen indeksit erotetaan toisistaan pilkulla, esim. $a_{i,j+1}$. Jos $m = n$, kyseessä on neliömatriisi. Tällöin alkioita, joiden rivi- ja sarakeindeksit ovat samoja, kutsutaan diagonaalialkioiksi.

Tyypillisiin matriisien laskutoimituksiin kuuluvat vakiolla kertominen sekä matriisien yhteen- ja vähennyslasku [54, s. 25]. Vakiolla kertominen ja yhteenlasku määritellään alkioittain ja vähennyslasku näiden avulla. Tämän tutkielman kannalta tärkeä on myös matriisikertolasku [54, s. 31–32].

MÄÄRITELMÄ 9. Olkoot $A = (a_{ij})$ ja $B = (b_{ij})$ $m \times n$-matriiseja sekä $\lambda \in \mathbb{R}$. *Vakion λ ja matriisin A tulo* on

$$\lambda \cdot A = \lambda A = (\lambda a_{ij}). \tag{6.2}$$

Lisäksi *matriisien A* ja *B summa* on

$$A + B = (a_{ij} + b_{ij}) \tag{6.3}$$

ja *erotus*

$$A - B = A + (-1) \cdot B. \tag{6.4}$$

Matriisit λA, $A + B$ ja $A - B$ ovat $m \times n$-matriiseja.

MÄÄRITELMÄ 10. Olkoon $A = (a_{ij})$ $m \times n$-matriisi ja $B = (b_{ij})$ $n \times s$-matriisi. *Matriisien A ja B tulo* on matriisi $A \cdot B = AB = (c_{ij})$. Tulomatriisin alkiot ovat

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}. \tag{6.5}$$

Tulomatriisi on $m \times s$-matriisi.

Matriisien laskutoimituksissa dimensioilla on merkitystä. Kuten määritelmästä 9 huomataan, vain samankokoisia matriiseja voidaan laskea yhteen ja vähentää toisistaan. Määritelmän 10 mukaan matriisien A ja B tulo on määritelty vain, mikäli A:ssa on täsmälleen yhtä monta saraketta kuin B:ssä on rivejä. Tästä seuraa myös, ettei BA ole välttämättä määritelty, vaikka AB olisikin määritelty.

6.2 Olemassa olevia rinnakkaisalgoritmeja

Matriisikertolaskun rinnakkaistamiseen on useita tunnettuja vaihtoehtoja. Li ym. [56] toteavat, ettei mikään algoritmi ole joka tilanteessa suorituskykyisin vaihtoehto. Siksi he esittävätkin polyalgoritmista lähestymistapaa, jossa päätöksentekomekanismi valitsee tilanteeseen parhaiten sopivan algoritmin. Li ym. jakavat kertolaskualgoritmit Cannonin algoritmiin, Foxin algoritmiin ja vain yleislähetystä käyttäviin algoritmeihin. Seuraavissa alaluvuissa noudatetaan samaa jaottelua. Kutakin algoritmia käsiteltäessä tuodaan esille myös Lin ym. esittämiä näkökohtia, mutta varsinaisiin polyalgoritmeihin ei perehdytä.

Huomautettakoon vielä, että tarkastelun kohteena ovat nimenomaan tiheiden matriisien kertolaskualgoritmit. Tiheiden matriisien vastakohta ovat harvat matriisit, joiden alkioista vain pieni osa poikkeaa nollasta [42, s. 47–48]. Harvojen matriisien käsittely on oma aihekokonaisuutensa, sillä tavanomaisten menetelmien käyttäminen harvoille matriiseille olisi laskentaresurssien tuhlausta. Tulevissa alaluvuissa siis A ja B ovat tiheitä $n \times n$-matriiseja ja $C = AB$.

6.2.1 Cannon

Cannon on esittänyt algoritmin vuonna 1969 ilmestyneessä väitöskirjassaan [12]. Ideana on, että A, B ja C jaetaan lohkoihin ja yhden matriisin lohkojen määrä on yhtä suuri kuin käytettävien prosessoreiden määrä. Prosessorit siis muodostavat kaksiulotteisen taulukon eli matriisin, jolloin prosessori P_{ij} vastaa lohkojen A_{ij}, B_{ij} ja C_{ij} käsittelemisestä.

$$A(0) = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \quad B(0) = \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix} \quad C(0) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$A(1) = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{11} & a_{12} & a_{10} \\ a_{22} & a_{20} & a_{21} \end{pmatrix} \quad B(1) = \begin{pmatrix} b_{00} & b_{11} & b_{22} \\ b_{10} & b_{21} & b_{02} \\ b_{20} & b_{01} & b_{12} \end{pmatrix} \quad C(1) = \begin{pmatrix} a_{00}b_{00} & a_{01}b_{11} & a_{02}b_{22} \\ a_{11}b_{10} & a_{12}b_{21} & a_{10}b_{02} \\ a_{22}b_{20} & a_{20}b_{01} & a_{21}b_{12} \end{pmatrix}$$

$$A(2) = \begin{pmatrix} a_{02} & a_{00} & a_{01} \\ a_{10} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{20} \end{pmatrix} \quad B(2) = \begin{pmatrix} b_{20} & b_{01} & b_{12} \\ b_{00} & b_{11} & b_{22} \\ b_{10} & b_{21} & b_{02} \end{pmatrix}$$

$$C(2) = \begin{pmatrix} a_{00}b_{00} + a_{02}b_{20} & a_{01}b_{11} + a_{00}b_{01} & a_{02}b_{22} + a_{01}b_{12} \\ a_{11}b_{10} + a_{10}b_{00} & a_{12}b_{21} + a_{11}b_{11} & a_{10}b_{02} + a_{12}b_{22} \\ a_{22}b_{20} + a_{21}b_{10} & a_{20}b_{01} + a_{22}b_{21} & a_{21}b_{12} + a_{20}b_{02} \end{pmatrix}$$

$$A(3) = \begin{pmatrix} a_{01} & a_{02} & a_{00} \\ a_{12} & a_{10} & a_{11} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \quad B(3) = \begin{pmatrix} b_{10} & b_{21} & b_{02} \\ b_{20} & b_{01} & b_{12} \\ b_{00} & b_{11} & b_{22} \end{pmatrix}$$

$$C(3) = \begin{pmatrix} a_{00}b_{00} + a_{02}b_{20} + a_{01}b_{10} & a_{01}b_{11} + a_{00}b_{01} + a_{02}b_{21} & a_{02}b_{22} + a_{01}b_{12} + a_{00}b_{02} \\ a_{11}b_{10} + a_{10}b_{00} + a_{12}b_{20} & a_{12}b_{21} + a_{11}b_{11} + a_{10}b_{01} & a_{10}b_{02} + a_{12}b_{22} + a_{11}b_{12} \\ a_{22}b_{20} + a_{21}b_{10} + a_{20}b_{00} & a_{20}b_{01} + a_{22}b_{21} + a_{21}b_{11} & a_{21}b_{12} + a_{20}b_{02} + a_{22}b_{22} \end{pmatrix}$$

Kuva 6.1: Esimerkki Cannonin algoritmista. Muokattu Cannonin esityksen pohjalta [12, s. 23–27].

Algoritmin kulku voidaan kuvata Cannonin esitystä [12, s. 22–27] mukaillen seuraavasti. Olkoot aluksi $A(0) = A$, $B(0) = B$ ja $C(0)$ nollamatriisi. Yksinkertaisuuden vuoksi oletetaan Cannonin esimerkin tapaan myös, että kukin lohko sisältää vain yhden alkion. Siirretään jokaista matriisin $A(0)$ riviä i $(i = 0, 1, \ldots, n-1)$ i sarakkeen verran vasemmalle ja jokaista matriisin $B(0)$ saraketta j $(j = 0, 1, \ldots n-1)$ j rivin verran ylös. Näin on muodostettu $A(1)$ ja $B(1)$, ja matriisi $C(1)$ saadaan nyt kertomalla matriisien $A(1)$ ja $B(1)$ vastinalkiot keskenään. Sen jälkeen $A(k)$ muodostetaan siirtämällä jokaista matriisin $A(k-1)$ alkiota yhden sarakkeen verran oikealle ja $B(k)$ siirtämällä jokaista matriisin $B(k-1)$ lohkoa yhden rivin verran alas. $C(k)$

muodostetaan kertomalla matriisien $A(k)$ ja $B(k)$ vastinalkiot keskenään ja lisäämällä näin saatu matriisi matriisiin $C(k-1)$. Algoritmin suoritus päättyy, kun $C(n)$ on laskettu, sillä $C(n) = AB$. Erityistapausta $n = 3$ on havainnollistettu kuvassa 6.1. Kuten kuvasta käy ilmi, esim. matriisin oikeanpuoleisimman sarakkeen alkioita oikealle siirrettäessä alkiot siirtyvät vasemmanpuoleisimpaan sarakkeeseen.

Lee ym. [55] pitävät Cannonin algoritmin etuna sitä, että kaikki prosessorit työskentelevät jokaisen iteraation aikana, ja sitä, että prosessorit viestivät keskenään vain lohkoja siirrettäessä. He yleistävät algoritmin muille kuin neliömatriiseille. Yleistetty algoritmi ottaa huomioon myös sen, että fyysisten prosessoreiden määrä ei välttämättä vastaa lohkojen määrää. Bae ym. [9] puolestaan esittävät alkuperäistä Cannonin algoritmia nopeamman algoritmin, joka tosin vaatii kultakin prosessorilta myös useampaa rekisteriä. Li ym. [56] kutsuvat tavanomaista Cannonin algoritmia C-stationaariseksi, koska laskennan aikana matriisien A ja B alkiot liikkuvat, mutta C:n alkiot eivät liiku. He esittävät algoritmista lisäksi A-stationaarisen ja B-stationaarisen version. Heidän mukaansa sopivan version valinnalla voidaan pienentää viestinnän määrää. Jos esim. matriisi A on huomattavasti suurempi kuin matriisit B ja C, on hyödyllistä käyttää A-stationaarista algoritmia, koska tällöin A:n alkioita ei tarvitse siirtää.

6.2.2 Fox

Fox ym. [33] ovat esittäneet algoritmin, jota on myöhemmin alettu kutsua Foxin algoritmiksi. Tässäkin ideana on, että matriisit jaetaan lohkoihin ja kukin prosessori P_{ij} vastaa sille kuuluvista lohkoista A_{ij}, B_{ij} ja C_{ij}.

Algoritmi voidaan kuvata Foxin ym. esitystä [33] mukaillen seuraavasti. Oletetaan taaskin yksinkertaisuuden vuoksi, että jokainen lohko sisältää vain yhden alkion. Olkoot $A(0) = A$, $B(0) = B$ ja $C(0)$ nollamatriisi. Kun $k > 0$, matriisi $A(k)$ muodostetaan siten, että kaikki rivin i alkiot $(i = 0, 1, \ldots, n-1)$ ovat $a_{i,i+k}$, siis diagonaali- tai sivudiagonaalialkioita. Nyt matriisi $C(k)$ saadaan kertomalla matriisien $A(k)$ ja $B(k-1)$ vastinalkiot keskenään ja lisäämällä näin saatu matriisi matriisiin $C(k-1)$. Sen jälkeen muodostetaan matriisi $B(k)$ siirtämällä jokaista $B(k-1)$:n alkiota yhden rivin verran ylös. Lopulta $C(n) = AB$, ja tässä vaiheessa myös $B(n) = B$. Kuvassa 6.2 on havainnollistettu erityistapausta $n = 3$. Kuvasta käy ilmi myös, että matriisien oikeanpuoleisin ja vasemmanpuoleisin sarake sekä ylin ja alin rivi ajatellaan toisiinsa yhdistetyiksi. Tällöin esim. matriisin ylimmällä rivillä oleva alkio siirtyy ylöspäin siirrettäessä alimmalle riville.

$$A(0) = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \quad B(0) = \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix} \quad C(0) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$A(1) = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{11} & a_{12} & a_{10} \\ a_{22} & a_{20} & a_{21} \end{pmatrix} \quad B(1) = \begin{pmatrix} b_{00} & b_{11} & b_{22} \\ b_{10} & b_{21} & b_{02} \\ b_{20} & b_{01} & b_{12} \end{pmatrix} \quad C(1) = \begin{pmatrix} a_{00}b_{00} & a_{01}b_{11} & a_{02}b_{22} \\ a_{11}b_{10} & a_{12}b_{21} & a_{10}b_{02} \\ a_{22}b_{20} & a_{20}b_{01} & a_{21}b_{12} \end{pmatrix}$$

$$A(2) = \begin{pmatrix} a_{02} & a_{00} & a_{01} \\ a_{10} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{20} \end{pmatrix} \quad B(2) = \begin{pmatrix} b_{20} & b_{01} & b_{12} \\ b_{00} & b_{11} & b_{22} \\ b_{10} & b_{21} & b_{02} \end{pmatrix}$$

$$C(2) = \begin{pmatrix} a_{00}b_{00} + a_{02}b_{20} & a_{01}b_{11} + a_{00}b_{01} & a_{02}b_{22} + a_{01}b_{12} \\ a_{11}b_{10} + a_{10}b_{00} & a_{12}b_{21} + a_{11}b_{11} & a_{10}b_{02} + a_{12}b_{22} \\ a_{22}b_{20} + a_{21}b_{10} & a_{20}b_{01} + a_{22}b_{21} & a_{21}b_{12} + a_{20}b_{02} \end{pmatrix}$$

$$A(3) = \begin{pmatrix} a_{01} & a_{02} & a_{00} \\ a_{12} & a_{10} & a_{11} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \quad B(3) = \begin{pmatrix} b_{10} & b_{21} & b_{02} \\ b_{20} & b_{01} & b_{12} \\ b_{00} & b_{11} & b_{22} \end{pmatrix}$$

$$C(3) = \begin{pmatrix} a_{00}b_{00} + a_{02}b_{20} + a_{01}b_{10} & a_{01}b_{11} + a_{00}b_{01} + a_{02}b_{21} & a_{02}b_{22} + a_{01}b_{12} + a_{00}b_{02} \\ a_{11}b_{10} + a_{10}b_{00} + a_{12}b_{20} & a_{12}b_{21} + a_{11}b_{11} + a_{10}b_{01} & a_{10}b_{02} + a_{12}b_{22} + a_{11}b_{12} \\ a_{22}b_{20} + a_{21}b_{10} + a_{20}b_{00} & a_{20}b_{01} + a_{22}b_{21} + a_{21}b_{11} & a_{21}b_{12} + a_{20}b_{02} + a_{22}b_{22} \end{pmatrix}$$

Kuva 6.2: Esimerkki Foxin algoritmista. Muokattu Foxin ym. esimerkistä [33, kuva 2].

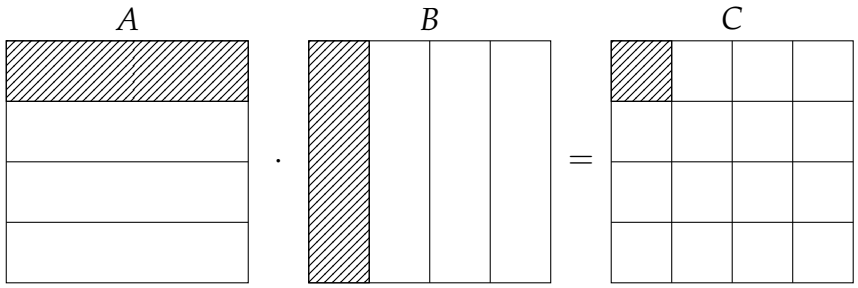
Algoritmi muistuttaa Cannonin algoritmia [12, s. 22–27], mutta eräs oleellinen ero on yleislähetyksen käyttö. Matriisin A diagonaali- tai sivudiagonaalialkiothan yleislähetetään kaikille samaa riviä käsitteleville prosessoreille. Esim. Lee ym. [55] luokittelevat matriisikertolaskun rinnakkaisalgoritmit yleislähetystä käyttäviin ja niihin, jotka eivät käytä yleislähetystä. Cannonin algoritmi sijoittuu viimeksi mainittuun luokkaan, sillä siinä prosessorit kommunikoivat vain vierekkäisten prosessorien kanssa.

Choin ym. kehittämä [14] PUMMA (*Parallel Universal Matrix Multiplication Algorithms*) sisältää muiden algoritmien lisäksi myös Foxin algoritmin yleistyksen. Myös Li ym. [56] ovat kehittäneet Foxin algoritmista useita muunnelmia. Heidän mukaansa Foxin algoritmia kannattaa käyttää, jos prosessorimatriisin muoto on äärimmäinen, esim. rivejä on paljon enemmän kuin sarakkeita. Tässä tapauksessa tietenkin täytyy valita algoritmin sopiva muunnelma.

6.2.3 Vain yleislähetystä käyttäviä algoritmeja

Vaikka Foxin algoritmissa [33] käytetäänkin yleislähetystä, siinä on mukana myös vierekkäisten lohkojen välistä viestintää, nimittäin lohkoja ylöspäin siirrettäessä. Sen sijaan tähän luokkaan kuuluvat algoritmit käyttävät viestinnässään vain yleislähetystä. Tunnettu esimerkki on van de Geijnin ja Wattsin kehittämä [82] SUMMA (*Scalable Universal Matrix Multiplication Algorithm*). Van de Geijnistä ja Wattsista riippumatta Agarwal ym. [2] ovat kehittäneet hyvin samanlaisen algoritmin.

Algoritmin idea voidaan esittää seuraavasti. Matriisit A, B ja C jaetaan lohkoihin, ja prosessori P_{ij} vastaa lohkoista A_{ij}, B_{ij} ja C_{ij}. Oletuksena on myös, että alussa prosessorilla on muistissa kyseiset A:n ja B:n lohkot. Jotta prosessori pystyisi laskemaan lohkon C_{ij} arvot, se tarvitsee A:sta yhden lohkorivin ja B:stä lohkosarakkeen, kuten nähdään kuvasta 6.3. Siksi jokainen prosessori P_{ij} yleislähettää lohkon A_{ij} samaa riviä käsitteleville prosessoreille ja lohkon B_{ij} samaa saraketta käsitteleville prosessoreille.



Kuva 6.3: Tulomatriisin lohkon laskentaan tarvitaan yksi lohkorivi ja lohkosarake. Yksinkertaistaen muokattu Josén ym. esittämästä kuvasta [47, kuva 1].

Myös José ym. [47] esittävät hyvin samantapaisen algoritmin, joskin heidän esittämässään algoritmissa matriisit luetaan ulkoisesta muistista. José ym. analysoivat algoritmia arkkitehtuurin näkökulmasta ja esittävät tähän perustuvan rinnakkaistietokoneen arkkitehtuuria koskevan suunnitelman.

Lin ym. mukaan [56] vain yleislähetystä käyttävien algoritmien etuna on helppo toteutettavuus. Toisaalta yleislähetyksen aiheuttamien kustannusten vuoksi ne ovat tehokkaita vain silloin, kun prosessoreita on vähän.

7 Kertolaskualgoritmit ja niiden kustannusfunktiot

Tässä luvussa esitetään matriisikertolaskualgoritmit ja niitä vastaavat kustannusfunktiot, joita luvussa 8 tarkastellaan kokeellisesti. Tarkastelu aloitetaan perehtymällä peräkkäisalgoritmiin ja sen kustannuksiin luvussa 7.1. Rinnakkaisalgoritmit noudattavat pitkälti samaa ideaa kuin aiemmin esitellyt vain yleislähetystä käyttävät algoritmit [82, 2, 47]. Prosessorien väliselle viestinnälle puolestaan esitetään LogGP-mallin [6] inspiroima kustannusmalli. Rinnakkaisalgoritmit ja niiden kustannusfunktiot esitetään luvussa 7.2.

7.1 Peräkkäisalgoritmi

Olkoot A, B ja C $n \times n$-matriiseja ja $C = AB$. Tietokonetoteutusta ajatellen oletetaan myös matriisien koostuvan liukuluvuista. Kun A ja B esitetään tietokoneessa kaksiulotteisten taulukoiden avulla, on niiden kertolaskun toteuttaminen suoraviivaista. C-kielinen peräkkäiskoodi on esitetty listauksessa 7.1

```
1 for (int i = 0; i < n; i++)
2     for (int j = 0; j < n; j++)
3         C[i][j] = 0;
4         for (int k = 0; k < n; k++)
5             C[i][j] += A[i][k]*B[k][j];
```

Listaus 7.1: Matriisikertolasku C-peräkkäiskoodina.

Algoritmissa on kolme sisäkkäistä silmukkaa. Keskimmäinen silmukka sisältää yhden sijoitusoperaation rivillä 3 ja sisimmän silmukan riveillä 4 ja 5. Sisimmän silmukan runko sisältää yhden lauseen, joka voidaan laskea kahdeksi lauseeksi (yhteenlasku ja kertolasku). Kun nyt uloimman silmukan runko suoritetaan n kertaa, operaatioita suoritetaan yhteensä

$$n(n(1+2n)) = 2n^3 + n^2. \tag{7.1}$$

Aikavaativuus on näin ollen $O(n^3)$ ja peräkkäiskoodin kustannusfunktio

$$T_{\text{sr}}(n) = (2n^3 + n^2)\tau_{\text{sr}}, \tag{7.2}$$

missä τ_{sr} on yhteen liukulukuoperaatioon kuluva aika.

7.2 Rinnakkaisalgoritmit

Käytetty rinnakkaisalgoritmi perustuu sekä hajautetun että jaetun muistin tapauksessa oikeastaan usean peräkkäisalgoritmin rinnakkaiseen suorittamiseen. Tämä edellyttää matriisien lohkojakoa. Rinnakkaisuus tuo mukaan myös prosessorien välisen viestinnän, ja sitä myötä kustannusparametrien määrä lisääntyy. Luvussa 7.2.1 esitetään, kuinka matriisit jaetaan lohkoihin ja miten prosessorien välinen viestintä tapahtuu. Luvuissa 7.2.2 ja 7.2.3 esitetään itse algoritmit ja niiden kustannusfunktiot.

7.2.1 Lohkojako ja viestintä

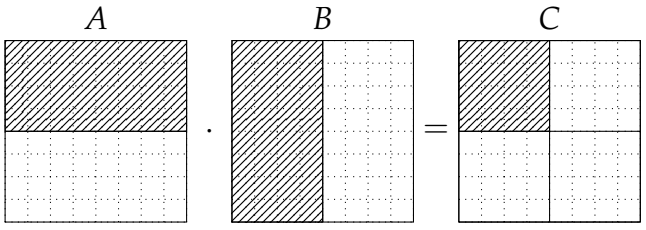
Olkoot edelleen A, B ja C liukuluvuista koostuvia $n \times n$-matriiseja sekä $C = AB$. Yksinkertaisuuden vuoksi oletetaan myös, että $n = 2^d$ $(d = 1, 2, \ldots)$. Olkoon prosessorien määrä $N = 4^1, 4^2, \ldots, 4^d$. Nyt matriisi C jaetaan N:ään neliömatriisin muotoiseen lohkoon,

$$C = \begin{pmatrix} C_0 & C_1 & \cdots & C_{\sqrt{N}-1} \\ \vdots & \vdots & & \vdots \\ C_{(\sqrt{N}-1)\sqrt{N}} & C_{(\sqrt{N}-1)\sqrt{N}+1} & \cdots & C_{N-1} \end{pmatrix}, \tag{7.3}$$

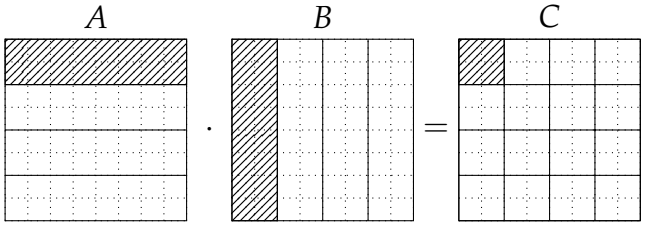
ja sovitaan, että prosessori P_i vastaa lohkon C_i alkioiden laskemisesta. Matriisit A ja B jaetaan $\sqrt{N}$ lohkoon, A vaakasuuntaisesti ja B pystysuuntaisesti. Tapausta $n = 8$ vastaavia lohkojakoja on havainnollistettu kuvassa 7.1. Kuvasta nähdään, että kun N:ää kasvatetaan, lopulta $N = n^2$ ja C:n lohkot ovat yhden alkion kokoisia.

Kuvasta 7.1 saadaan myös käsitys siitä, millaisia A:n ja B:n lohkoja kukin prosessori tarvitsee. Yhden matriisin C lohkon laskemiseksi tarvitaan A:sta $n/\sqrt{N} \times n$-lohko ja B:stä $n \times n/\sqrt{N}$-lohko, siis yhteensä $2n^2/\sqrt{N}$ alkiota. Tällöin yksittäinen prosessori suorittaa $(2n^3 + n^2)/N$ liukulukuoperaatiota.

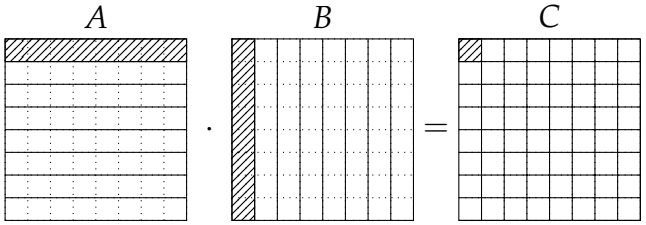
Hajautetun muistin tapauksessa kaikki prosessorien välinen viestintä toteutetaan kahdenvälisen viestinvälityksen avulla. Viestinnän toteutus ja sen kustannukset riippuvat käytetystä topologiasta. Sekä Pješivac-Grbović ym. [73] että Nuriyev ym. [69] luettelevat viisi erilaista topologiaa: lineaarinen eli litteä puu, ketjupuu, binääripuu, K-ketjupuu ja binomipuu. Näitä on havainnollistettu kuvassa 7.2. Kuten Pješivac-Grbović ym. [73] toteavat, topologioiden paremmuus riippuu tilanteesta.



(a) $N = 4$.



(b) $N = 16$.

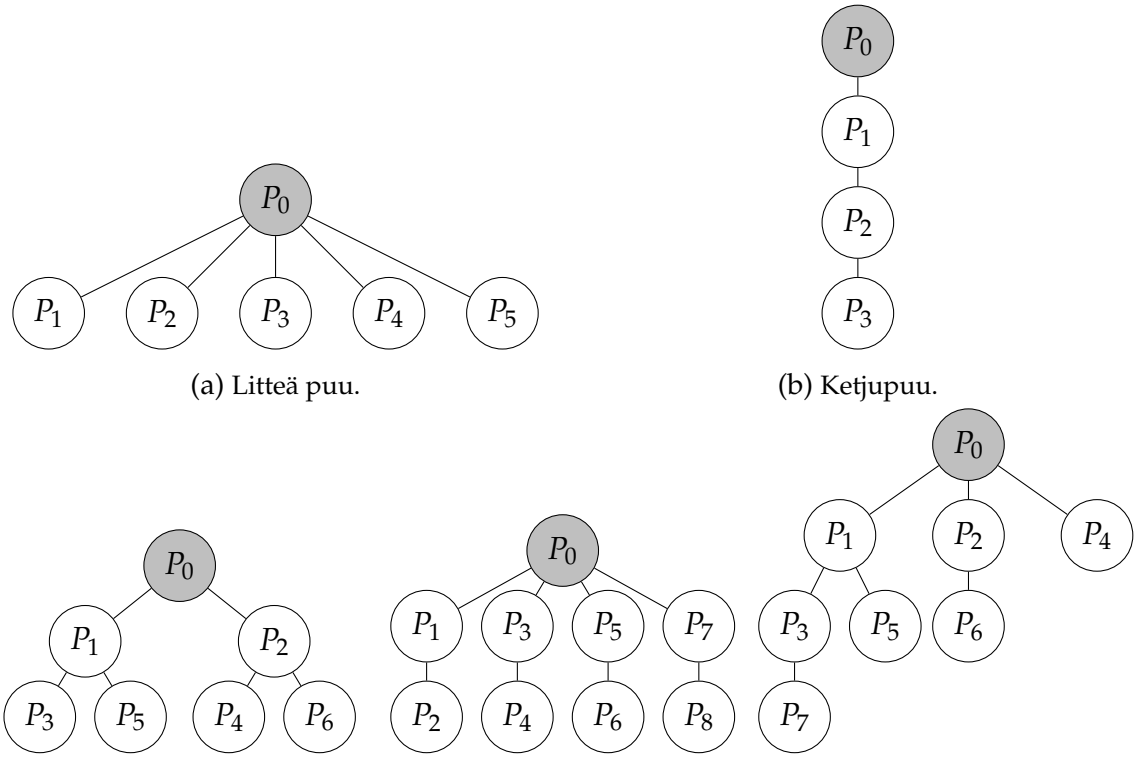


(c) $N = 64$.

Kuva 7.1: Matriisien jako lohkoihin, kun $n = 8$. Prosessorin P_0 laskema matriisin C lohko ja sen laskentaa varten tarvitsemat matriisien A ja B lohkot on väritetty.

Kustannusfunktioita laadittaessa pyrittiin käyttämään mahdollisimman yksinkertaista mallia. Siksi mainituista tutkimuksista [73, 69] poiketen viestin segmentointia ei oteta huomioon, vaan oletetaan, että prosessori lähettää viestin toiselle prosessorille yhdellä kertaa. Näin vastaanottaja ei voi lähettää mitään saamansa viestin osaa eteenpäin, ennen kuin se on vastaanottanut koko viestin. Toisaalta lineaarisen puun ja ketjupuun oleellinen ero on siinä, että ensiksi mainittu ei käytä segmentointia ja viimeksi mainittu käyttää, joten ketjupuulla ei tässä mallissa ole merkitystä. Alustavissa mittauksissa osoittautui, että hajautetun muistin tapauksessa binääripuutopologiaan perustuva kustannusfunktio onnistui mallintamaan kustannuksia tyydyttävästi. Siksi hajautetun muistin kollektiivinen viestintä esitetään binääripuun avulla.

Viestinvälitysmallia hyödynnetään myös jaetun muistin tapauksessa. Erona edellä sanottuun on kuitenkin se, että nyt keskenään viestivät prosessori ja jaettu muisti. Toisin sanoen jaettua muistia voidaan ajatella ylimääräisenä prosessorina, joka



(a) Litteä puu. (b) Ketjupuu.

(c) Binääripuu. (d) K-ketjupuu ($K = 4$). (e) Binomipuu.

Kuva 7.2: Rinnakkaistietokoneiden topologioita. Muokattu Nuriyevin ym. esittämästä kuvasta [69, kuva 3].

ei varsinaisesti laske mitään. Alustavissa mittauksissa huomattiin, että lineaariseen topologiaan perustuva kustannusfunktio tuottaa tyydyttävän tuloksen, joten topologia oletetaan lineaariseksi.

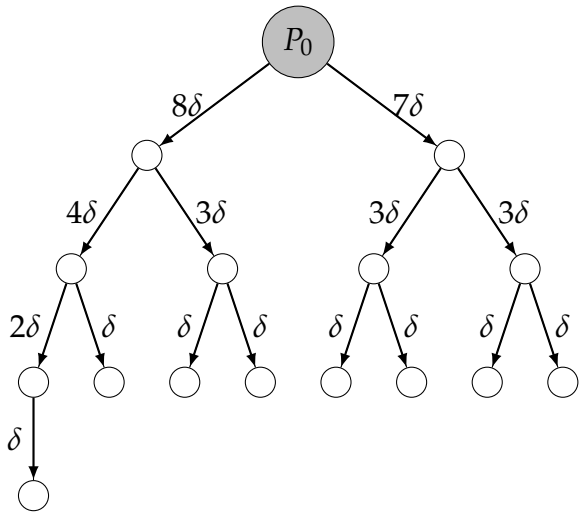
Kun kahden prosessorin tai prosessorin ja muistin välillä siirtyy k liukulukua, tähän kuluva aika on $\alpha + \gamma \cdot (k - 1) \approx \alpha + \gamma \cdot k$. Tässä α muodostuu lähettäjän ja vastaanottajan yleiskustannuksesta ja verkon viiveestä. Toinen parametri γ on puolestaan yhtä liukulukua vastaava viestiväli. Ideana on, että yhden liukuluvun lähettämisen jälkeen lähettäjän on odotettava γ:n verran. Toisin sanoen $1/\gamma$ on yhtä liukulukua vastaava kaistanleveys. Hajautetun ja jaetun muistin parametrit erotetaan toisistaan alaindeksien avulla. Niinpä hajautetun muistin kustannusparametrit ovat α_{dm} ja γ_{dm} ja jaetun muistin puolestaan α_{sm} ja γ_{sm}.

7.2.2 Hajautettu muisti

Alussa matriisit A ja B ovat prosessorin P_0 hallussa, ja laskennan päätyttyä myös kaikki C:n alkiot lähetetään P_0:lle. Algoritmissa on näin ollen kolme vaihetta:

1. Prosessori P_0 lähettää jokaiselle prosessorille niiden tarvitsemat A:n ja B:n lohkot.
2. Kukin prosessori P_i laskee sille osoitetun lohkon C_i alkiot.
3. Kukin prosessori P_i lähettää laskemansa lohkon C_i alkiot prosessorille P_0.

Tarkastellaan seuraavaksi kuhunkin vaiheeseen liittyviä aikakustannuksia.



Kuva 7.3: Datan levitys prosessorilta P_0 prosessoreille $P_1, P_2, \ldots, P_{N-1}$, kun $N = 16$. Prosessoreista vain P_0 on nimetty. Se, missä järjestyksessä muut prosessorit nimetään, ei vaikuta aikakustannuksiin.

Vaiheessa 1 prosessori P_0 siis lähettää prosessoreille $P_1, P_2, \ldots, P_{N-1}$ liukulukuja. Olkoon yhden prosessorin tarvitsemien liukulukujen määrä δ. Lähetettävänä on näin ollen yhteensä $(N-1)\delta$ liukulukua. Lähettäminen suoritetaan peräkkäisinä kahdenvälisinä viestinvälitysoperaatioina siten, että koko viestin saatuaan prosessori lähettää dataa aina kahdelle seuraavalle prosessorille, kunnes jokainen prosessori on saanut δ liukulukua. Erityistapausta $N = 16$ on havainnollistettu kuvassa 7.3. Aluksi P_0 lähettää toiselle prosessorille 8δ ja toiselle 7δ lukua. Vastaanottajat tallentavat saamastaan viestistä δ lukua ja lähettävät lopun datan taas eteenpäin.

Tarkastellaan sitten kaikista pisintä P_0:sta lehtisolmuun kulkevaa polkua, koska se määrää suoritusajan. Koska pisimmän polun pituus on $\log_2(N)$ ja kulkevan datan määrä tällä polulla $N\delta/2 + N\delta/4 + \cdots + N\delta/N$, saadaan polun aikakustannukseksi

$$\begin{aligned}
&\left(\alpha_{\text{dm}} + \gamma_{\text{dm}} \cdot \frac{N\delta}{2}\right) + \left(\alpha_{\text{dm}} + \gamma_{\text{dm}} \cdot \frac{N\delta}{4}\right) + \cdots + \left(\alpha_{\text{dm}} + \gamma_{\text{dm}} \cdot \frac{N\delta}{N}\right) \\
=&\alpha_{\text{dm}} \log_2(N) + \gamma_{\text{dm}} N\delta \left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{N}\right) \\
=&\alpha_{\text{dm}} \log_2(N) + \gamma_{\text{dm}}(N-1)\delta.
\end{aligned} \tag{7.4}$$

Kukin prosessori tarvitsee A:sta yhden $n/\sqrt{N} \times n$-lohkon ja B:stä yhden $n \times n/\sqrt{N}$-lohkon, joten $\delta = 2n^2/\sqrt{N}$. Huomautettakoon, että tähän määrään sisältyy paljon toistoa. Esim. kaikki matriisin C samaa riviä laskevat prosessorit tarvitsevat A:sta saman lohkon. Algoritmia toteutettaessa kuitenkin osoittautuu, että on yksinkertaisinta käsitellä jokaista lohkoa erillisenä. Siksi lähetettävän datan määrä on näin suuri. Sijoittamalla ja käyttämällä approksimaatiota $N - 1 \approx N$ saadaan

$$\begin{aligned}
\alpha_{\text{dm}} \log_2(N) + \gamma_{\text{dm}}(N-1)\delta &= \alpha_{\text{dm}} \log_2(N) + \gamma_{\text{dm}} \cdot \frac{2n^2(N-1)}{\sqrt{N}} \\
&\approx \alpha_{\text{dm}} \log_2(N) + 2\gamma_{\text{dm}} n^2 \sqrt{N}.
\end{aligned} \tag{7.5}$$

Vaiheessa 2 kukin prosessori laskee oman $n/\sqrt{N} \times n/\sqrt{N}$-lohkonsa alkiot. Algoritmi vastaa aiemmin kuvattua peräkkäisalgoritmia. Kun yhteen liukulukuoperaatioon kuluvaa aikaa merkitään τ_{dm}:llä, aikakustannus on $(2n^3 + n^2)\tau_{\text{dm}}/N$.

Vaiheessa 3 prosessorit $P_1, P_2, \ldots, P_{N-1}$ lähettävät laskemansa $n/\sqrt{N} \times n/\sqrt{N}$-lohkon alkiot prosessorille P_0. Algoritmi on vaiheeseen 1 nähden käänteinen, mutta aikakustannusten kannalta ei ole väliä, kumpaan suuntaan data liikkuu. Kun tarkastellaan pisintä polkua lehtisolmusta prosessoriin P_0 ja huomioidaan, että $\delta = n^2/N$, saadaan aikakustannukseksi

$$\begin{aligned}
\alpha_{\text{dm}} \log_2(N) + \gamma_{\text{dm}}(N-1)\delta &= \alpha_{\text{dm}} \log_2(N) + \gamma_{\text{dm}} \cdot \frac{n^2(N-1)}{N} \\
&\approx \alpha_{\text{dm}} \log_2(N) + \gamma_{\text{dm}} n^2.
\end{aligned} \tag{7.6}$$

Algoritmin kokonaiskustannus saadaan laskemalla vaiheiden 1, 2 ja 3 kustannukset yhteen. Hajautetun muistin algoritmin kustannusfunktio on näin ollen

$$T_{\text{dm},n}(N) = 2\alpha_{\text{dm}} \log_2(N) + \frac{2n^3 + n^2}{N}\tau_{\text{dm}} + 2\gamma_{\text{dm}} n^2 \sqrt{N} + \gamma_{\text{dm}} n^2. \tag{7.7}$$

Funktio $N \mapsto 1/N$ on aidosti vähenevä, funktiot $N \mapsto \log_2(N)$ ja $N \mapsto \sqrt{N}$ puolestaan aidosti kasvavia. Näistä ensiksi mainittu vähenee nopeasti ja viimeksi mainitut

kasvavat hitaasti. Tämä antaa aiheen olettaa, että kustannusfunktio vähenee ensin voimakkaasti ja alkaa sitten hiljalleen kasvaa. Tällöin funktiolla olisi yksi globaali minimi. On kuitenkin huomattava, että myös kertoimet vaikuttavat funktion käyttäytymiseen. Joka tapauksessa funktion derivaatta on

$$T'_{\mathrm{dm},n}(N) = \frac{2\alpha_{\mathrm{dm}}}{N\ln(2)} - \frac{2n^3+n^2}{N^2}\tau_{\mathrm{dm}} + \frac{\gamma_{\mathrm{dm}}n^2}{\sqrt{N}}. \tag{7.8}$$

Derivaatan nollakohdan ratkaiseminen johtaa neljännen asteen ja sieventämisen jälkeen kolmannen asteen yhtälöön, kun muuttujaksi valitaan $\sqrt{N}$. Siksi nollakohtaa ei tässä tutkielmassa ratkaista analyyttisesti.

7.2.3 Jaettu muisti

Alkutilanteessa matriisit A ja B on tallennettu jaettuun muistiin, ja lopussa myös matriisi C tallennetaan sinne. Algoritmissa on kolme vaihetta:

1. Kukin prosessori lukee vuorollaan tarvitsemansa matriisien A ja B alkiot jaetusta muistista.
2. Kukin prosessori laskee sille osoitetut C:n alkiot.
3. Kukin prosessori tallentaa laskemansa C:n alkiot jaettuun muistiin.

Yksinkertaisuuden vuoksi oletetaan, että jokaisella prosessorilla on riittävä tallennuskapasiteetti matriisien A, B ja C lohkojen väliaikaista tallentamista varten.

Vaiheessa 1 kukin prosessori siis lukee tarvitsemansa matriisien A ja B alkiot jaetusta muistista. Jaetun muistin lukeminen ja siihen kirjoittaminen mallinnetaan kahdenvälisenä viestintänä. Luettavia alkioita on yhtä prosessoria kohti $2n^2/\sqrt{N}$. Koska vain yksi prosessori kerrallaan voi tässä mallissa lukea jaettua muistia, saadaan kokonaisajaksi

$$N\left(\alpha_{\mathrm{sm}} + \gamma_{\mathrm{sm}}\frac{2n^2}{\sqrt{N}}\right) = \alpha_{\mathrm{sm}}N + 2\gamma_{\mathrm{sm}}n^2\sqrt{N}. \tag{7.9}$$

Vaiheessa 2 kukin prosessori laskee sille osoitetut matriisin C alkiot. Kun yhteen liukulukuoperaatioon kuluvaa aikaa merkitään kirjaimella τ_{sm}, on laskentaan kuluva aika $(2n^3+n^2)\tau_{\mathrm{sm}}/N$. Vaiheessa 3 kukin prosessori tallentaa vuorollaan laskemansa C:n lohkon jaettuun muistiin. Koska lohkon alkioiden määrä on n^2/N, saadaan aikakustannukseksi

$$N\left(\alpha_{\mathrm{sm}} + \gamma_{\mathrm{sm}}\frac{n^2}{N}\right) = \alpha_{\mathrm{sm}}N + \gamma n^2. \tag{7.10}$$

Jaetun muistin kustannusfunktio on siis

$$T_{\mathrm{sm},n}(N) = 2\alpha_{\mathrm{sm}}N + \frac{2n^3 + n^2}{N}\tau_{\mathrm{sm}} + 2\gamma_{\mathrm{sm}}n^2\sqrt{N} + \gamma_{\mathrm{sm}}n^2 \tag{7.11}$$

ja funktion derivaatta

$$T'_{\mathrm{sm},n}(N) = 2\alpha_{\mathrm{sm}} - \frac{2n^3 + n^2}{N^2}\tau_{\mathrm{sm}} + \frac{\gamma_{\mathrm{sm}}n^2}{\sqrt{N}}. \tag{7.12}$$

Kuten hajautetun muistin tapauksessa, tämänkin kustannusfunktion lauseke antaa olettaa, että funktiolla on globaali minimi. Kuitenkin lopullinen totuus selviää vasta sitten, kun kustannusparametrit on määritetty. Derivaatan nollakohdan ratkaiseminen johtaa neljännen asteen yhtälöön, kun muuttujaksi valitaan $\sqrt{N}$. Tämänkään derivaatan nollakohtaa ei tässä tutkielmassa ratkaista analyyttisesti.

8 Tulokset

Luvuissa 6 ja 7 matriisikertolaskua ja sen rinnakkaistamista on tarkasteltu teoreettisesti. Nyt luvussa 7 esitettyjä algoritmeja ja niihin liittyviä kustannusfunktioita aletaan tarkastella kokeellisesti. Apuna käytetään luvussa 5 esiteltyjä ohjelmointiympäristöjä. Luvussa 8.1 esitellään käytetty laitteisto ja luvussa 8.2 algoritmien toteutus. Luvussa 8.3 kuvataan mittaukset ja mittauksissa saadut tulokset. Tuloksien pohjalta määritetään kustannusfunktioiden parametrit luvussa 8.4. Kustannusmallia arvioidaan luvussa 8.5.

8.1 Laitteistosta ja sen käyttämisestä

Ohjelmia ajettaessa käytettiin CSC:n Puhti-supertietokonetta. Puhti [27, 28] on Atos BullSequana X400 -klusteri, joka on otettu käyttöön syyskuussa 2019. Puhdissa on 682 CPU-noodia, ja näiden teoreettinen huippusuorituskyky on 1,8 petaflops. Kussakin noodissa on kaksi Intel Xeon Cascade Lake -prosessoria ja kussakin prosessorissa 20 ydintä, joiden kellotaajuus on 2,1 GHz. Noodit on yhdistetty toisiinsa 100 Gbps:n linkillä. Lisäksi Puhdissa on 80 GPU-noodia, joiden teoreettinen huippusuorituskyky on 2,7 petaflops. Tähän tutkielmaan liittyvissä mittauksissa käytettiin kuitenkin vain CPU-noodeja.

Puhtiin saadaan yhteys komentoriviltä komentoa

```
ssh tunnus@puhti.csc.fi
```

käyttäen [24]. Kun ohjelmakoodi on kirjoitettu ja käännetty, sitä ei ajeta suoraan. Sen sijaan käytetään suoritusskriptiä [22], joka ohjelman ajamisen lisäksi varaa tarvittavat resurssit. Komennolla

```
sbatch skriptitiedoston_nimi
```

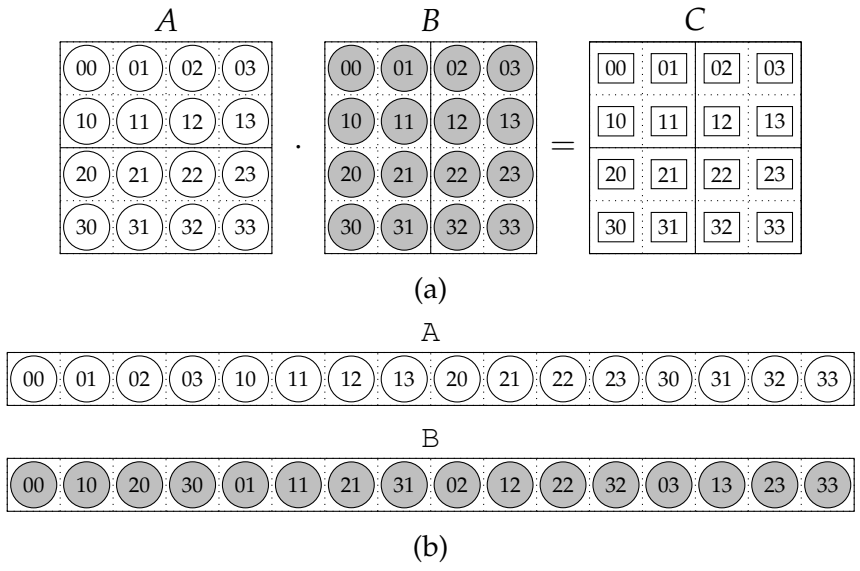
saadaan työ suoritusjonoon [26]. Työn aloittamisen ajankohtaa ei ole mahdollista ennustaa [23]. Siihen vaikuttavat resurssien saatavuus ja työn prioriteetti.

Suoritusskripti sisältää muiden tietojen lisäksi tiedon käytettävästä partitiosta [20], joka määrää, kuinka montaa prosessoriydintä on mahdollista käyttää. Mittauksissa käytettiin partitioita `small`, jossa voi käyttää vain yhtä noodia, ja `large`, jo-

ka sallii enintään 26 noodin käyttämisen. Näin ollen käytettävissä oli enimmillään $26 \cdot 40 = 1\,040$ prosessoriydintä.

8.2 Algoritmien toteutus

Matriisikertolasku $C = AB$ (matriisit $n \times n$-matriiseja) toteutettiin peräkkäiskoodina sekä MPI- ja OpenMP-koodina. Jokaisessa toteutuksessa käytettiin C-kieltä. Yhteistä kaikille toteutuksille on matriisien täyttäminen (pseudo-)satunnaisilla liukuluvuilla ja niiden esittäminen yksiulotteisina taulukoina. Matriisi A esitettiin riveittäin ja matriisi B sarakkeittain. Kertolaskua suoritettaessa nimittäin matriisin A rivit kerrotaan alkioittain matriisin B sarakkeiden kanssa. Näin ollen B:n alkioiden tallentaminen sarakkeittain mahdollistaa peräkkäisten alkioiden käyttämisen. Esitystapaa on havainnollistettu kuvassa 8.1. Matriisi C esitettiin riveittäin peräkkäis- ja OpenMP-algoritmissa. MPI:n tapauksessa esitystapa on monimutkaisempi.



Kuva 8.1: Kuvassa (a) on esitetty matriisit ja kuvassa (b) niiden tallentaminen yksiulotteisina taulukoina, kun $n = 4$. Matriisi A tallennetaan riveittäin ja B sarakkeittain. Matriisin C tallentaminen riippuu algoritmista. Kuvaan (a) on merkitty myös tapausta $N = 4$ vastaava lohkojako.

8.2.1 Peräkkäisalgoritmi

Peräkkäisalgoritmin toteutus on varsin suoraviivainen. Koodi on listattu liitteessä A. Lyhennettynä koodi on esitetty listauksessa 8.1. Riveillä 1–8 muuttujiin `A` ja `B` tallennetaan satunnaisia liukulukuja. Varsinaisen laskenta tapahtuu riveillä 11–18. Ajanmittaus aloitetaan rivillä 10 ja lopetetaan rivillä 19. Lopuksi tulostetaan laskentaan kulunut aika millisekunteina.

```
1  // A tallennetaan riveittäin.
2  for (i = 0; i < n*n; i++) {
3      A[i] = drand48();
4  }
5  // B tallennetaan sarakkeittain.
6  for (i = 0; i < n*n; i++) {
7      B[i] = drand48();
8  }
9
10 gettimeofday(&start, NULL);
11 for (i = 0; i < n; i++) {
12     for (j = 0; j < n; j++) {
13         C[i*n+j] = 0;
14         for (k = 0; k < n; k++) {
15             C[i*n+j] += A[i*n+k] * B[j*n+k];
16         }
17     }
18 }
19 gettimeofday(&end, NULL);
20
21 // Aika millisekunteina.
22 time = (end.tv_sec - start.tv_sec) * 1000
23     + (double) (end.tv_usec - start.tv_usec) / 1000;
24 printf("%f\n", time);
```

Listaus 8.1: Peräkkäisalgoritmi lyhennettynä.

Huomautettakoon, ettei koko ohjelman suoritusaikaa mitata. Algoritmeja koskevat kustannusfunktiot nimittäin mallintavat vain itse kertolaskun laskemiseen ja viestintään kuluvaa aikaa eivätkä ota huomioon mitään muuta. Peräkkäisohjelman tapauksessa kiinnostuksen kohteena on tietenkin pelkkä laskenta.

8.2.2 MPI

MPI-koodi on listattu liitteessä B. Teoreettisesta tarkastelusta poiketen nyt N on prosessien, ei prosessoreitten lukumäärä. Algoritmissa on kolme vaihetta:

1. Matriisien A ja B lohkot jaetaan prosessilta P_0 prosesseille $P_0, P_1, \ldots, P_{N-1}$.
2. Kukin prosessi laskee oman lohkonsa matriisista C.
3. Matriisin C lohkot eli laskennan tulokset kerätään prosesseilta $P_0, P_1, \ldots, P_{N-1}$ prosessille P_0.

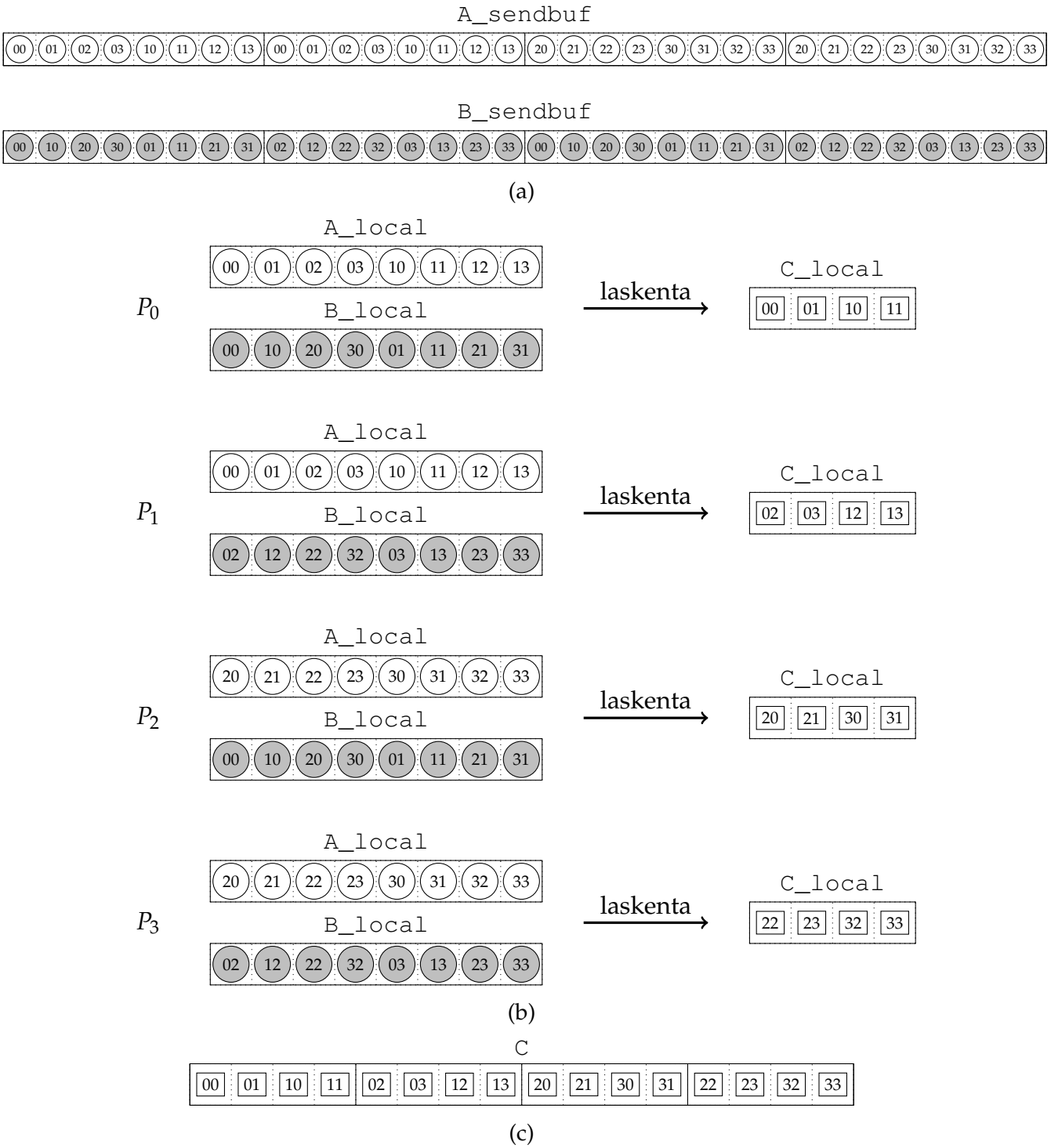
Joitakin tarvittavia tietorakenteita on havainnollistettu kuvassa 8.2. Vaiheessa 1 käytetään `MPI_Scatter`-funktiota ja vaiheessa 3 `MPI_Gather`-funktiota.

Vaiheen 1 kannalta on huomattavaa, ettei `MPI_Scatter`-funktio voi jakaa samoja lähetyspuskurin alkioita eri prosesseille. Vaikka esim. kaikki matriisin C samaa riviä laskevat prosessit tarvitsevat saman matriisin A lohkon, on kaikki nämä lohkot tallennettava lähetyspuskuriin erikseen. Kuva 8.2a havainnollistaa asiaa. Kuten kuvasta 8.2b nähdään, kukin prosessi vastaanottaa datan vastaanottopuskureihin `A_local` ja `B_local`. Laskenta tapahtuu peräkkäisesti, ja laskennan tulokset tallennetaan muuttujaan `C_local`. Vaiheessa 3 `MPI_Gather`-funktio kerää laskennan tulokset kunkin prosessin muuttujasta `C_local` prosessin P_0 vastaanottopuskuriin `C`.

Kuvasta 8.2c nähdään muuttujan `C` alkioiden järjestys, joka ei vastaa matriisin C alkioiden järjestystä. Alkiot on toki mahdollista järjestää uudestaan, mutta koska `C` sisältää tarvittavan informaation ja koska tarkoitus on mitata vain viestintään ja laskentaan kuluvaa aikaa, ajanmittaus alkaa juuri ennen `MPI_Scatter`-funktion kutsumista ja päättyy heti `MPI_Gather`-funktion suorittamisen jälkeen.

```
1  MPI_Barrier(MPI_COMM_WORLD);
2  local_start = MPI_Wtime();
3
4  // Tietojen levitys prosesseille, laskenta ja tulosten keruu.
5  ...
6
7  local_finish = MPI_Wtime();
8  local_elapsed = (local_finish - local_start) * 1000;
9  MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
10             MPI_MAX, ROOT, MPI_COMM_WORLD);
```

Listaus 8.2: MPI-ohjelman ajanmittaus.



Kuva 8.2: MPI-algoritmin datan levitys, laskenta ja tulosten keruu, kun $n = 4$ ja $N = 4$. Kuvassa (a) on esitetty `MPI_Scatter`-funktion käyttämät lähetyspuskurit ja kuvassa (b) kunkin prosessin paikalliset vastaanottopuskurit sekä laskennan tuloksena syntyvien alkioiden tallentaminen. Kuvassa (c) `MPI_Gather`-funktion keräämät tulokset on tallennettu prosessin P_0 muuttujaan `C`.

Ajanmittauksessa sovelletaan Pachecon ja Malensekin ohjetta [72, s. 129–130]. Toteutus on esitetty listauksessa 8.2. Ensinnäkin rivillä 1 prosessit synkronoidaan. Riveillä 2 ja 7 kutsutaan `MPI_Wtime`-funktiota. Laskemalla funktion palauttamien arvojen `local_start` ja `local_finish` erotus saadaan kulunut seinäkelloaika eli `local_elapsed`. Tämä on kuitenkin koodia suorittavan prosessin mittaama aika, joten eri aikoja saadaan N kappaletta. Sen vuoksi riveillä 9 ja 10 kutsutaankin `MPI_Reduce`-funktiota, jolla saadaan suurin mitattu aika.

8.2.3 OpenMP

OpenMP:n tapauksessa N on säikeiden lukumäärä. Toteutus poikkeaa varsin vähän peräkkäisalgoritmin toteutuksesta. Koko koodi on listattu liitteessä C ja koodin keskeisin osa listauksessa 8.3. Ajanmittaus aloitetaan listauksen 8.3 rivillä 1 ja lopetetaan rivillä 20. Rivillä 2 koodin suoritus haarautuu N säikeeseen, ja säikeet yhdistyvät rivillä 19. Riveillä 11–18 kukin säie laskee oman lohkonsa matriisista C.

```
1  time_start = omp_get_wtime();
2  #pragma omp parallel num_threads(N)
3  {
4      int id, row, col, row_start, col_start, row_max, col_max, kk;
5      id = omp_get_thread_num();
6
7      row_start = (id / BLOCKS_PER_LEN) * RC_PER_BLOCK;
8      row_max = row_start + RC_PER_BLOCK;
9      col_start = (id % BLOCKS_PER_LEN) * RC_PER_BLOCK;
10     col_max = col_start + RC_PER_BLOCK;
11     for (row = row_start; row < row_max; row++) {
12         for (col = col_start; col < col_max; col++) {
13             C[row*n+col] = 0;
14             for (kk = 0; kk < n; kk++) {
15                 C[row*n+col] += A[row*n+kk]*B[col*n+kk];
16             }
17         }
18     }
19 }
20 time_finish = omp_get_wtime();
21 time_elapsed = (time_finish - time_start) * 1000;
```

Listaus 8.3: Osa OpenMP-ohjelmasta.

MPI:n tapauksessa prosessit ensin synkronoidaan, tämän jälkeen kukin prosessi mittaa aikaa ja lopulta mitatuista ajoista valitaan suurin. OpenMP-ohjelmassa sen sijaan ajanotto aloitetaan ja lopetetaan varsinaisen rinnakkaiskoodin ulkopuolella. Erillistä synkronointia ei tarvita, sillä säikeiden yhdistyessä tapahtuu implisiittinen synkronointi. Tämä tosin vaikeuttaa MPI:n ja OpenMP:n suoritusaikojen vertailua, sillä OpenMP:n tapauksessahan osa mitatusta ajasta kuluu myös rinnakkaislaskennan aloittamiseen ja lopettamiseen. On kuitenkin ymmärrettävä, että kyseessä on kaksi erilaista ohjelmointiympäristöä. OpenMP-ohjelma voi sisältää useita rinnakkaisia lohkoja, MPI-ohjelmassa taas rinnakkaislaskenta aloitetaan ja lopetetaan yhden kerran.

8.3 Mittaukset

Käytetty MPI-toteutus oli Open MPI, jolla CSC myös suosittelee aloittamaan [21]. Peräkkäisohjelma käännettiin komennolla

```
gcc ohjelman_nimi
```

MPI-ohjelma komennolla

```
mpicc -lm ohjelman_nimi
```

ja OpenMP-ohjelma komennolla

```
gcc -fopenmp -lm ohjelman_nimi
```

MPI- ja OpenMP-ohjelmien kääntämisessä käytetty valitsin `-lm` linkittää mukaan tarvittavan matematiikkakirjaston. Valitsin `-fopenmp` tarvitaan OpenMP-ohjelmaa käännettäessä, ja `mpicc` on MPI-ohjelman kääntämiseksi tarvittava kääreskripti [21].

Ohjelmia ajettaessa rinnakkaisohjelmien suoritusajoissa havaittiin suurta heilahtelua, vaikka n:n ja N:n arvoja ei muutettu. Kun esim. $n = 16$ ja $N = 4$, pienin mitattu MPI-ohjelman suoritusaika oli 1,00 ms ja suurin 64,01 ms. OpenMP:n osalta vastaavat luvut olivat 0,08 ms ja 46,18 ms. Sen sijaan peräkkäisohjelman pienin mitattu aika n:n arvolla 16 oli 0,02 ms ja suurin 0,03 ms.

Pachecon ja Malensekin mukaan [72, s. 96, 130] suoritusaikojen vaihtelu johtuu ohjelman vuorovaikutuksesta järjestelmän muiden osien kanssa. Puhdin tapauksessa suoritusaikaan vaikuttavat epäilemättä myös muiden käyttäjien työt. Pacheco ja Malensek neuvovat valitsemaan mitatuista suoritusajoista pienimmän; heidän mukaansa on hyvin epätodennäköistä, että vuorovaikutus muun järjestelmän kanssa

Taulukko 8.1: Peräkkäisalgoritmin suoritusajat.

n	Aika (ms)	n	Aika (ms)	n	Aika (ms)
16	0,02	128	10,15	1 024	5 073,58
32	0,16	256	79,48	2 048	40 378,74
64	1,26	512	603,36	4 096	310 332,24

Taulukko 8.2: MPI-algoritmin suoritusajat.

n	N	Aika (ms)	n	N	Aika (ms)	n	N	Aika (ms)
16	4	0,24	128	4	2,96	1 024	4	992,64
	16	0,30		16	1,53		16	293,36
	64	2,55		64	14,40		64	95,14
	256	8,16		256	20,21		256	74,23
32	4	0,23	256	4	19,58	2 048	4	9 255,96
	16	0,31		16	6,09		16	2 330,78
	64	5,87		64	14,26		64	631,23
	256	13,46		256	20,70		256	270,66
64	4	0,52	512	4	146,99	4 096	4	75 369,14
	16	0,45		16	39,26		16	18 844,18
	64	5,98		64	24,79		64	4 934,32
	256	18,34		256	37,81		256	1 582,31

Taulukko 8.3: OpenMP-algoritmin suoritusajat.

n	N	Aika (ms)	n	N	Aika (ms)	n	N	Aika (ms)
16	4	2,25	128	4	2,64	1 024	4	1 208,70
	16	9,69		16	5,71		16	317,03
	64	5,92		64	3,00		64	218,22
	256	9,29		256	8,83		256	191,00
				1 024	29,59		1 024	202,52
32	4	1,48	256	4	20,30	2 048	4	9 620,51
	16	2,33		16	18,91		16	2 522,31
	64	2,20		64	9,53		64	1 383,20
	256	7,85		256	12,78		256	1 273,84
	1 024	29,51		1 024	32,64		1 024	1 255,85
64	4	0,46	512	4	159,14	4 096	4	80 688,15
	16	2,07		16	48,14		16	20 371,35
	64	2,94		64	41,56		64	10 778,49
	256	7,34		256	45,35		256	10 079,54
	1 024	28,12		1 024	57,70		1 024	10 033,42

nopeuttaisi ohjelman suoritusta. Käytäntö vaikuttaa kuitenkin kirjavalta. Kattavan vertailun tekeminen ei kuulu tähän tutkielmaan, mutta eri tutkimuksissa on suoritusaikoja mitattaessa käytetty mm. kolmen mittauksen pienintä arvoa [15], kymmenen mittauksen keskiarvoa [13, 66] ja tuhannen mittauksen keskiarvoa [64]. Kaikissa artikkeleissa ei ole edes mainintaa siitä, miten suoritusaika on mitattu.

Mazouz ym. [63] huomasivat, että peräkkäisohjelman tapauksessa suoritusaikojen heilahtelu ei ole merkittävää, OpenMP-ohjelman tapauksessa sen sijaan on. He kuitenkin pitävät pienimmän ajan mittaamista huonona käytäntönä mm. siksi, ettei pienintä suoritusaikaa välttämättä saavuteta kovin usein, ja suosittelevat mediaanin käyttämistä. Touati ym. [78] pitävät mediaania parempana kuin keskiarvoa, koska se ei ole yhtä herkkä poikkeaville havainnoille. Poikkeavien havaintojen vuoksi tässäkin tutkielmassa päätettiin käyttää mediaania. Peräkkäisohjelman tapauksessa keskiarvo olisi luultavasti riittänyt, mutta varmuuden saamiseksi ja vertailukelpoisuuden parantamiseksi sekä peräkkäis- että rinnakkaisohjelman suoritusajat mitattiin samaan tapaan. Peräkkäisohjelman tapauksessa kullakin luvun n ja rinnakkaisohjelmien tapauksessa kullakin lukuparin (n, N) arvolla mitattiin sata mittausaikaa. Mittausajoista lasketut mediaaniajat on esitetty kahden desimaalin tarkkuuteen pyöristettyinä taulukoissa 8.1, 8.2 ja 8.3.

Kuten taulukosta 8.2 nähdään, käytettyjen MPI-prosessien enimmäismäärä oli 256. Myös 1 024 prosessin käyttäminen olisi ollut mahdollista, mutta sata tällaista työtä olisi luultavasti vaatinut käytännössä liian pitkän jonotusajan. CSC:n ohjeissa [25] huomautetaan, että Puhdin noodien välinen viestintä on kalliimpaa kuin noodin sisäinen viestintä. Siksi työn jakamista moneen noodiin on syytä välttää. Tämän vuoksi yhdessä noodissa suoritettiin N prosessia, kun $N \leq 16$, ja suuremmilla N:n arvoilla 32 prosessia; myös 40 prosessin suorittaminen olisi ollut mahdollista, mutta 32 valittiin symmetrian vuoksi. OpenMP-työt pitää puolestaan suorittaa yhdessä noodissa, mutta käytettyjen prosessoriydinten määrään on mahdollista vaikuttaa [22]. Vastaavaa logiikkaa noudattaen yhtä OpenMP-työtä kohti varattiin 4, 16 tai 32 prosessoriydintä.

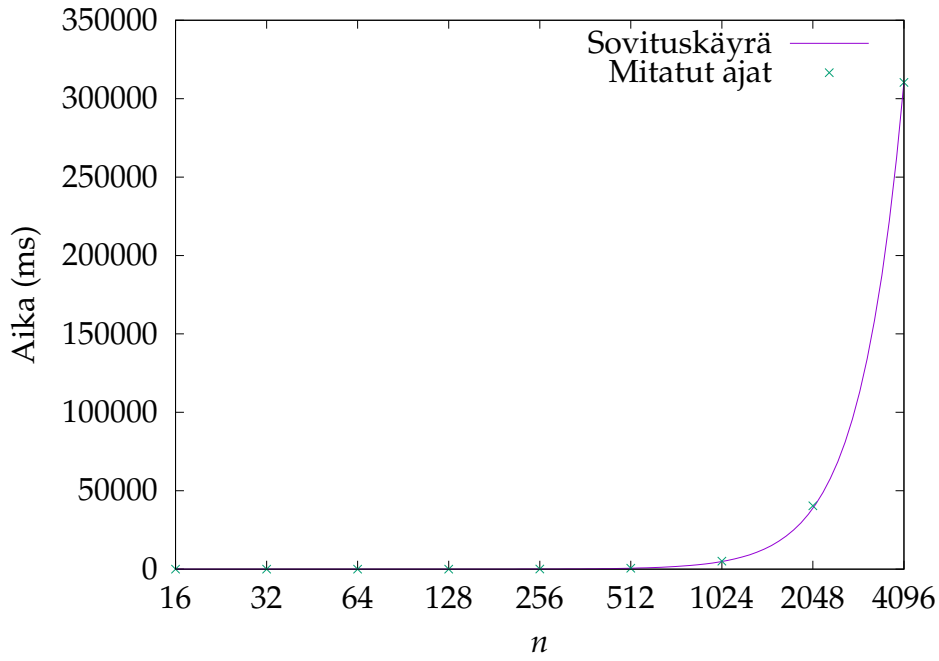
8.4 Kustannusparametrien määrittäminen

Peräkkäisohjelma käyttäytyi odotusten mukaisesti. Kustannusfunktio sovitettiin mitattuihin aikoihin Gnuplot-ohjelmaa [83, s. 74–81] käyttäen. Sovitus on on esitetty

kuvassa 8.3. Parametrin arvoksi saatiin

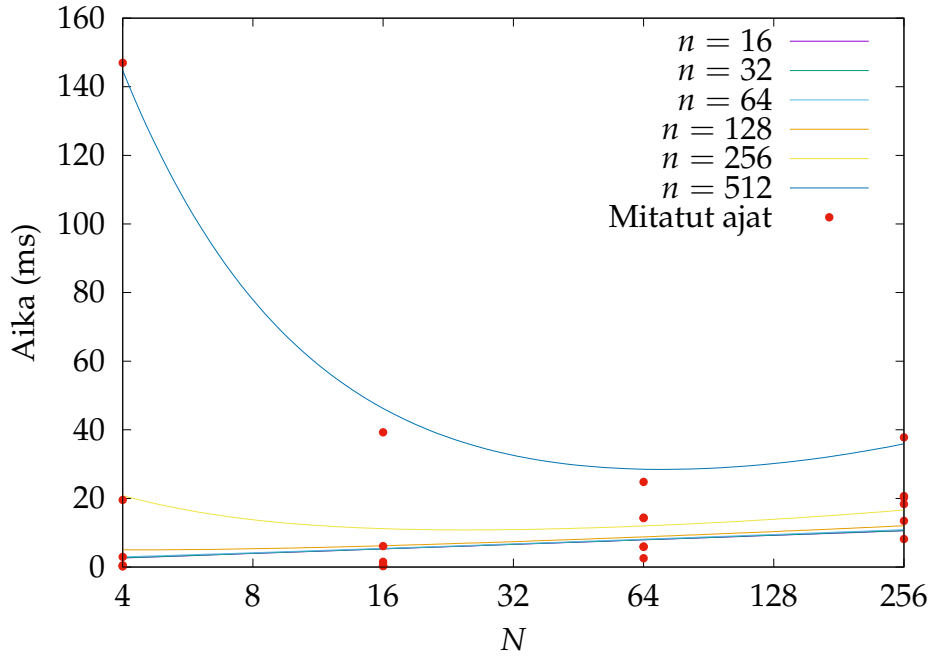
$$\tau_{\text{sr}} = (2{,}256 \pm 0{,}004)\,\text{ns}. \tag{8.1}$$

Arvo vastaa siis yhteen liukulukuoperaatioon kuluvaa aikaa. Koska prosessorin kellotaajuus on 2,1 GHz [28], yhteen liukulukuoperaatioon kuluu laskennallisesti $2{,}256 \cdot 10^{-9} \cdot 2{,}1 \cdot 10^{9} \approx 4{,}7$ prosessorisykliä.

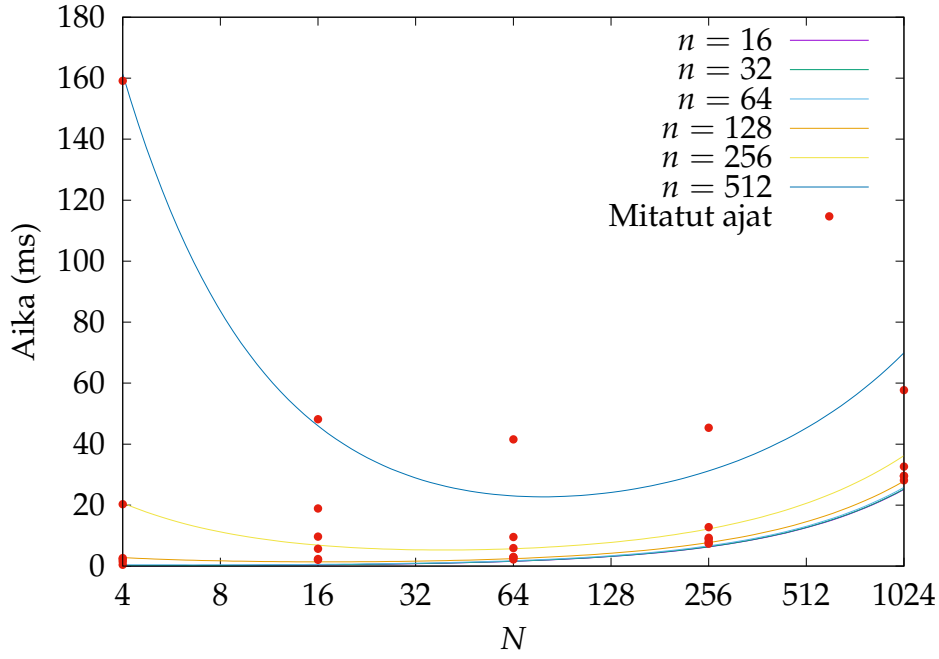


Kuva 8.3: Peräkkäisohjelman tulosten käyränsovitus.

Myös rinnakkaisohjelmien tulokset vaikuttavat pääsääntöisesti odotusten mukaisilta, kuten taulukoista 8.2 ja 8.3 nähdään. Kun $n = 16, 32$ tai 64, prosessien tai säikeiden lisääminen hyödyttää vähän tai ei ollenkaan, kun taas $n = 128, 256$ tai 512, se hyödyttää aluksi, mutta hyvin pian suoritusajat alkavat taas kasvaa. Sen sijaan isoimmilla matriiseilla suoritusajan kasvua ei useimmiten nähdä. On kuitenkin syytä olettaa, että samanlainen ilmiö olisi nähty myös suurimmilla matriiseilla, jos prosessorien tai säikeiden määrää olisi tarpeeksi kasvatettu. OpenMP:n tapauksessa arvoja $(n, N) = (16, 16), (32, 16)$ ja $(128, 16)$ vastaavat ajat tosin vaikuttavat poikkeuksellisilta jo pelkästään niitä edeltäviin ja seuraaviin aikoihin verrattuna. Yksi mahdollinen selitys on jo aiemmin mainittu mittausaikojen heilahtelu. On mahdollista myös, että kyseessä on ilmiö, jota ei ole osattu mallintaa. Havainnot eivät kuitenkaan vaikuta kokonaiskuvaan tai johtopäätöksiin.



(a) MPI.



(b) OpenMP.

Kuva 8.4: Rinnakkaisohjelmien käyränsovitus.

Vaikka rinnakkaisohjelmien kustannusfunktioissa muuttujana onkin N, tuloksia sovitettaessa otettiin samaan sovitukseen mukaan useampi n:n arvo eli useampi kustannusfunktio. Muuten sovituksessa käytettävien havaintopisteiden määrä olisi

jäänyt hyvin vähäiseksi, koska eri N:n arvoja on vähän. Sovituksen tarkkuutta arvioitiin Gnuplotin antamia kustannusparametrien virherajoja [83, s. 78–79] ja varsinkin suhteellista virhettä tarkastelemalla. Peräkkäisohjelmaa vastaavaan tarkkuuteen ei päästy, ja yhden tai useamman parametrin suhteellinen virhe kasvoi varsinkin matriisien koon kasvaessa. Tyydyttävä tulos saatiin sekä MPI:lle että OpenMP:lle, kun $n \in \{ 16, 32, 64, 128, 256, 512 \}$. Tätä vastaavat sovitukset on esitetty kuvassa 8.4. MPI-ohjelman kustannusparametreiksi saatiin

$$\begin{cases} \alpha_{\mathrm{dm}} = (700 \pm 100)\,\mu\mathrm{s} \\ \gamma_{\mathrm{dm}} = (2{,}7 \pm 0{,}5)\,\mathrm{ns} \\ \tau_{\mathrm{dm}} = (2{,}06 \pm 0{,}07)\,\mathrm{ns} \end{cases} \tag{8.2}$$

ja OpenMP-ohjelman

$$\begin{cases} \alpha_{\mathrm{sm}} = (12 \pm 2)\,\mu\mathrm{s} \\ \gamma_{\mathrm{sm}} = (2{,}6 \pm 0{,}4)\,\mathrm{ns} \\ \tau_{\mathrm{sm}} = (2{,}35 \pm 0{,}09)\,\mathrm{ns} \end{cases} . \tag{8.3}$$

Parametrit τ_{sr}, τ_{dm} ja τ_{sm} poikkeavat kaikki toisistaan, mutta ovat kuitenkin samaa suuruusluokkaa. Sen sijaan yleiskustannuksia ja viivettä kuvaavat parametrit α_{dm} ja α_{sm} poikkeavat toisistaan huomattavasti. Tämä johtuu ilmeisesti MPI:n ja OpenMP:n eroista. MPI-ohjelmassa kyse on prosessien välisestä kommunikoinnista, OpenMP-ohjelmassa taas jaetun muistin lukemisesta ja siihen kirjoittamisesta. Sitä paitsi suurimmilla N:n arvoilla yksi prosessoriydin suoritti useampaa OpenMP-säiettä, jolloin suurin osa säikeiden välisestä kommunikaatiosta tapahtui saman prosessoriytimen sisällä. Erilaisuudesta huolimatta liukulukukohtaista kaistanleveyden käänteislukua kuvaavat parametrit γ_{dm} ja γ_{sm} ovat kuitenkin hyvin lähellä toisiaan. Kustannusfunktioita laadittaessa oletettiin, että käytettävissä on aina N erillistä prosessoria. MPI:n tapauksessa tämä pitikin paikkansa, koska N oli sama kuin käytettyjen prosessoriydinten määrä. OpenMP:n tapauksessa voidaan suurimmilla N:n arvoilla puhua vain N virtuaalisesta prosessorista. Siihen nähden myös OpenMP-ohjelma noudatti kustannusmallia kohtuullisen hyvin.

Koska algoritmeissa käytettiin 64 bitin kaksoistarkkoja liukulukuja, saadaan esim. γ_{dm}:n arvoa käyttämällä kaistanleveydeksi $64\,\mathrm{b}/(2{,}7\,\mathrm{ns}) \approx 24\,\mathrm{Gbps}$. Tämä herättää kysymyksiä, koska Puhdin noodit on yhdistetty toisiinsa 100 Gbps:n linkillä [28] ja noodien sisäisen liikenteen pitäisi olla noodien välistä liikennettä hitaampaa [25]. Yksi mahdollinen selitys näin pienelle havaitulle kaistanleveydelle on yksinkertaisesti se, että Puhdissa oli laskentatehtäviä suoritettaessa muitakin käyttäjiä.

On mahdollista myös, että malli on epätarkka ja todellisuudessa γ_{dm} sisältää muutakin kuin kaistanleveydestä johtuvan viestivälin.

Taulukko 8.4: Kustannusfunktioiden derivaattojen nollakohdat. Muuttujana on N.

	Derivaatan nollakohta	
n	Hajautettu muisti	Jaettu muisti
16	< 1	< 1
32	< 1	2,4
64	< 1	6,6
128	4,4	16,8
256	24,9	39,1
512	70,1	79,0

Vaikka prosessien tai säikeiden määrä onkin luvun 4 potenssi, voidaan jokaista kustannusfunktiota silti käsitellä kuten mitä tahansa jatkuvaa ja derivoituvaa funktiota. Jos kustannusfunktion antama ennuste pitää paikkansa, derivaatan nollakohta vastaa sitä N:n arvoa, jolla saavutetaan pienin suoritusaika. Siksi kullekin kustannusfunktiolle laskettiin derivaatan nollakohta Maximalla [62] Newtonin menetelmää [42, s. 193–195] käyttäen. Ykköstä pienempiä nollakohtia ei kuitenkaan laskettu. Nollakohdat on lueteltu taulukossa 8.4. Taulukoita 8.2, 8.3 ja 8.4 vertaamalla nähdään, että nollakohtien arvot ovat pääsääntöisesti sopusoinnussa mitattujen tulosten kanssa, vaikka poikkeuksiakin on. Esim. MPI-ohjelman tapauksessa $n = 64$ voisi mitatuista arvoista päätellä, että nollakohta on jossain N:n arvojen 4 ja 64 välillä, mutta toisaalta kustannusfunktion nollakohta on ykköstä pienempi.

8.5 Kustannusmallin arviointia

Kustannusmalli on sidoksissa käytettyyn kertolaskualgoritmiin, ja tässä tapauksessa algoritmi kuuluu Lin ym. luokittelussa [56] vain yleislähetystä käyttäviin algoritmeihin. Samaan algoritmiluokkaan kuuluvat van de Geijnin ja Wattsin SUMMA [82], Agarwalin ym. algoritmi [2] sekä Josén ym. [47] algoritmi. Van de Geijn ja Watts [82] johtavat algoritmille kustannusmallin olettamalla, että kahden prosessorin välisen viestinvälityksen kustannus on tämän tutkielman merkintöjä käyttäen muotoa $\alpha + \gamma \cdot k$, missä k on lähetettävien liukulukujen määrä. Samaa oletusta päädyttiin

käyttämään myös tässä tutkielmassa. Kirjoittajat esittävät algoritmille myös MPI-koodin, mutta eivät testaa mallin toimivuutta. Agarwal ym. [2] puolestaan testaavat algoritmin suorituskykyä, mutta eivät esitä kustannusmallia. Sekä van de Geijnin ja Wattsin [82] SUMMA että Agarwalin ym. [2] algoritmi on pelkästään hajautetun muistin algoritmi. Tässä tutkielmassa samaa ideaa sovelletaan sekä hajautetulle että jaetulle muistille.

Hajautetun muistin algoritmi eroaa van de Geijnin ja Wattsin [82] sekä Agarwalin ym. [2] esittämistä algoritmeista ainakin kahdessa suhteessa. Ensinnäkin edellä mainituissa algoritmeissa alkutilanne on, että matriisilohkojen A_i ja B_i alkiot on tallennettu prosessorin P_i muistiin. Toiseksi kertolaskun suorittaminen etenee askeleina, jotka koostuvat prosessorien välisestä viestinnästä ja laskennasta. Sen sijaan tässä tutkielmassa esitetty hajautetun muistin algoritmi lähtee tilanteesta, jossa kaikki matriisien A ja B alkiot ovat prosessorin P_0 hallussa. Sen jälkeen data levitetään prosessoreille, kukin prosessori suorittaa laskennan ja lopuksi tulokset kerätään prosessorille P_0. Tietenkin sovelluksia ajatellen riippuu tilanteesta, voidaanko olettaa, että alkutilanteessa matriisit A ja B on jaettu lohkoittain kaikille prosessoreille. Siksi kumpaakaan oletusta ei voi sinänsä pitää toista parempana. Kuitenkin useille MPI-ohjelmille on tyypillistä, että yksi prosessi on erityisasemassa, eikä tutkielman kirjoittajalla ole tiedossa yhtään tutkimusta, jossa matriisikertolaskun algoritmi ja sitä vastaava kustannusmalli laadittaisiin tästä oletuksesta lähtien. Lisäksi tässä esitetyn algoritmin rakenne on varsin yksinkertainen, koska data levitetään ja kerätään vain kerran. Yksinkertaisuuden varjopuolena on `MPI_Scatter`-funktion käyttämisestä johtuva tehottomuus. Näin ollen tutkielman uutuusarvona edellä viitattuihin tutkimuksiin [82, 2] nähden voidaan pitää ensinnäkin erilaista algoritmia ja toiseksi sitä, että algoritmille laaditaan kustannusmalli, jota myös testataan.

José ym. [47] esittävät algoritmin, jossa kaikki tiedot luetaan ulkoisesta muistista ja tulokset tallennetaan ulkoiseen muistiin, kun laskenta on suoritettu. He esittävät algoritmille myös kustannusmallin ja mallin pohjalta rinnakkaistietokoneen arkkitehtuuria koskevan suunnitelman. Rinnakkaistietokoneesta rakennetaan prototyyppi, jossa mallin toimivuutta testataan. Tässä tutkielmassa esitetty jaetun muistin algoritmi on sen sijaan kehitetty yleisesti jo olemassa olevia rinnakkaistietokoneita varten. Josén ym. [47] esittämä kustannusmalli on myös huomattavasti yksityiskohtaisempi, sillä se ottaa huomioon myös jaetun muistin suorituskykyyn liittyvän muistihierarkian. Tämän tutkielman mallissa sen sijaan niin hajautetun kuin jaetunkin muistin kustannukset mallinnetaan samoista oletuksista lähtien eikä muistihie-

rarkiaan kuuluvia erilaisia hakuaikoja oteta huomioon. Siksi tässä esitettyä mallia voi yksinkertaisuutensa vuoksi pitää yleisempänä, mutta samalla tietysti myös epätarkempana.

Kustannusparametreja määritettäessä oletettiin, että parametrit ovat riippumattomia matriisien dimensiosta n. Tämä oli käyränsovitusten tekemiseksi välttämätöntä, koska prosessien tai säikeiden määrä N kasvoi hyvin nopeasti ja yhtä n:n arvoa vastaavia mittauspisteitä saatiin hyvin vähän. Tyydyttäviä tuloksia saatiinkin, kun $n \in \{ 16, 32, 64, 128, 256, 512 \}$. Suuremmilla matriiseilla sovituksen tarkkuus heikkeni huomattavasti. On tietenkin mahdollista, että tulokset olisivat olleet tarkempia, mikäli olisi ollut mahdollista määrittää kutakin lukua n vastaavat kustannusparametrit erikseen. Tämä olisi edellyttänyt suurempia rinnakkaislaskentaresursseja. Toisaalta suurempien rinnakkaislaskentaresurssien käyttö olisi luultavasti vaatinut myös mallin tarkentamista. Esitetty malli ei huomioi nykyaikaisten rinnakkaistietokoneiden hierarkiaa, jossa noodit koostuvat prosessoreista ja prosessorit ytimistä ja jossa noodien välinen tietoliikenne on kalliimpaa kuin noodien sisäinen. Sen sijaan mallin mukainen rinnakkaistietokone koostuu yksinkertaisesti prosessoreista, jotka ovat yhteydessä toisiinsa, tai prosessoreista, jotka ovat yhteydessä jaettuun muistiin.

Kaiken kaikkiaan esitettyjen kustannusmallien avulla voidaan karkeasti arvioida matriisikertolaskun kustannuksia Puhdin kaltaisessa rinnakkaistietokoneessa ainakin silloin, kun $n \leq 512$. Kummankaan mallin tarkkuus ei kuitenkaan riitä tarkan suoritusajan ennustamiseen.

9 Yhteenveto ja johtopäätökset

Jotta rinnakkaisohjelma olisi suorituskykyinen, on laskennan ja viestinnän oltava tasapainossa, sillä rinnakkaisuuden lisääntyessä myös viestinnän määrä tavallisesti lisääntyy. Rinnakkaislaskennan mallintamista varten onkin vuosien varrella kehitetty lukuisia teoreettisia malleja, joskaan kaikki mallit eivät keskity kuvaamaan viestintään liittyviä kustannuksia.

Tutkimusongelmaksi asetettiin rinnakkaisen matriisikertolaskun kustannusmallin laatiminen. Jotta malli ei jäisi liian yleisluontoiseksi, ennen mallin laatimista laadittiin myös vastaava algoritmi. Algoritmi ja malli laadittiin sekä hajautetun että jaetun muistin rinnakkaistietokoneille. Kustannusmalli esitettiin kustannusfunktiona. Tutkimusmenetelmänä oli konstruktiivinen tutkimusote [49, 59]. Aihepiiriin tutustuttiin perehtymällä rinnakkaistietokoneisiin, rinnakkaislaskennan teoreettisiin malleihin, kahteen yleisesti käytettyyn rinnakkaislaskennan ohjelmointiympäristöön sekä eräisiin jo olemassa oleviin matriisikertolaskun rinnakkaisalgoritmeihin.

Hajautetun muistin algoritmi toteutettiin C-kielisenä MPI-koodina ja jaetun muistin algoritmi C-kielisenä OpenMP-koodina. Ohjelmat suoritettiin CSC:n Puhti-supertietokoneessa [27, 28]. Mittaamalla relevanttien koodilohkojen suorittamiseen kuluvaa aikaa voitiin määrittää kustannusfunktioihin sisältyvät kustannusparametrit. Näin pystyttiin myös arvioimaan mallin luotettavuutta. Odottamattomana seikkana mittauksissa havaittiin yllättävän voimakasta suoritusaikojen heilahtelua, vaikkei ohjelmaa tai ohjelman syötettä suorituskertojen välissä muutettukaan. Tästä johtuvaa epätarkkuutta pyrittiin eliminoimaan tekemällä yhtä suoritusaikaa kohti sata mittausta, joiden tuloksista laskettiin mediaani. Heilahtelu selittynee ainakin osittain sillä, että Puhdissa oli mittauksia suoritettaessa muitakin käyttäjiä.

Mittausten perusteella $n \times n$-matriiseja Puhdin kaltaisessa tietokoneessa kerrottaessa kustannusmallin tarkkuus on tyydyttävä, kun $n \leq 512$. Mallin avulla saadaan karkea käsitys siitä, milloin rinnakkaisuuden lisääminen ei maksa vaivaa. Sen sijaan tarkkaan suoritusaikojen ennustamiseen mallin tarkkuus ei riitä. Mallia kehitettäessä on tehty yksinkertaistavia oletuksia, jotka eivät huomioi nykyisten rinnakkaistietokoneiden hierarkiaa. Siksi on syytä olettaa, että Puhtia merkittävästi suurempien rinnakkaistietokoneiden tapauksessa mallin tarkkuus heikkenee entisestään.

Mahdollinen jatkotutkimusaihe on samankaltaisen kustannusmallin laatiminen Puhtia suuremmille rinnakkaistietokoneille. Tällöin joudutaan luultavasti käyttämään nyt esitettyä monimutkaisempaa mallia, joka huomioi paremmin nykyaikaisten rinnakkaistietokoneiden hierarkian. Toinen mahdollinen jatkotutkimusaihe on algoritmin ja kustannusmallin kehittäminen grafiikkasuorittimia käyttäville rinnakkaistietokoneille.

Lähteet

[1] AFANASYEV, I. V., VOEVODIN, V. V., VOEVODIN, V. V., KOMATSU, K., JA KOBAYASHI, H. Analysis of relationship between SIMD-processing features used in NVIDIA GPUs and NEC SX-Aurora TSUBASA vector processors. Julkaisusarjassa *International Conference on Parallel Computing Technologies* (Almaty, Kazakhstan, elokuu 2019), Springer, 125–139.

[2] AGARWAL, R. C., GUSTAVSON, F. G., JA ZUBAIR, M. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal of Research and Development 38*, 6 (1994), 673–681.

[3] AGGARWAL, A., CHANDRA, A. K., JA SNIR, M. Communication complexity of PRAMs. *Theoretical Computer Science 71*, 1 (1990), 3–28.

[4] AL-HAYANNI, M. A. N., XIA, F., RAFIEV, A., ROMANOVSKY, A., SHAFIK, R., JA YAKOVLEV, A. Amdahl's law in the context of heterogeneous many-core systems — a survey. *IET Computers & Digital Techniques 14*, 4 (2020), 133–148.

[5] ALDINUCCI, M., CESARE, V., COLONNELLI, I., MARTINELLI, A. R., MITTONE, G., CANTALUPO, B., CAVAZZONI, C., JA DROCCO, M. Practical parallelization of scientific applications with OpenMP, OpenACC and MPI. *Journal of Parallel and Distributed Computing 157* (2021), 13–29.

[6] ALEXANDROV, A., IONESCU, M. F., SCHAUSER, K. E., JA SCHEIMAN, C. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. Julkaisusarjassa *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures* (Santa Barbara, CA, USA, kesäkuu 1995), Association for Computing Machinery, 95–105.

[7] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. Julkaisusarjassa *Proceedings of the April 18–20, 1967, spring joint computer conference* (Atlantic City, NJ, USA, huhtikuu 1967), Association for Computing Machinery, 483–485.

[8] AMIRI, H., JA SHAHBAHRAMI, A. SIMD programming using Intel vector extensions. *Journal of Parallel and Distributed Computing 135* (2020), 83–100.

[9] BAE, S. E., SHINN, T.-W., JA TAKAOKA, T. A faster parallel algorithm for matrix multiplication on a mesh array. *Procedia Computer Science 29* (2014), 2230–2240.

[10] BARRETO, L., JA BAUER, M. Parallel branch and bound algorithm — a comparison between serial, OpenMP and MPI implementations. Julkaisusarjassa *Journal of Physics: Conference Series* (Toronto, Canada, kesäkuu 2010), vol. 256, IOP Publishing, 012018.

[11] CABRAL, F. L., GONZAGA DE OLIVEIRA, S. L., OSTHOFF, C., COSTA, G. P., BRANDÃO, D. N., JA KISCHINHEVSKY, M. An evaluation of MPI and OpenMP paradigms in finite-difference explicit methods for PDEs on shared-memory multi-and manycore systems. *Concurrency and Computation: Practice and Experience 32*, 20 (2020), e5642.

[12] CANNON, L. E. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, elokuu 1969.

[13] CHANG, C.-H., LU, C.-W., YANG, C.-T., JA CHANG, T.-C. An approach of performance comparisons with OpenMP and CUDA parallel programming on multicore systems. *Concurrency and Computation: Practice and Experience 28*, 16 (2016), 4230–4245.

[14] CHOI, J., WALKER, D. W., JA DONGARRA, J. J. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience 6*, 7 (1994), 543–570.

[15] CHORLEY, M. J., JA WALKER, D. W. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science 1*, 3 (2010), 168–174.

[16] CROVELLA, M., BIANCHINI, R., LEBLANC, T., MARKATOS, E., JA WISNIEWSKI, R. Using communication-to-computation ratio in parallel program design and performance prediction. Julkaisusarjassa *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing* (Arlington, TX, USA, joulukuu 1992), IEEE, 238–245.

[17] CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., JA VON EICKEN, T. LogP: Towards a realistic model of parallel computation. Julkaisusarjassa *PPOPP '93: 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, USA, toukokuu 1993), Association for Computing Machinery, 1–12.

[18] DEAN, J., JA GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[19] DIAZ, J., MUÑOZ CARO, C., JA NIÑO, A. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems 23*, 8 (2012), 1369–1386.

[20] DOCS CSC. Available batch job partitions. URL `https://docs.csc.fi/computing/running/batch-job-partitions/`, viitattu 14.2.2023.

[21] DOCS CSC. Compiling applications in Puhti. URL `https://docs.csc.fi/computing/compiling-puhti/`, viitattu 14.2.2023.

[22] DOCS CSC. Creating a batch job script for Puhti. URL `https://docs.csc.fi/computing/running/creating-job-scripts-puhti/`, viitattu 14.2.2023.

[23] DOCS CSC. Getting started. URL `https://docs.csc.fi/computing/running/getting-started/`, viitattu 14.2.2023.

[24] DOCS CSC. Overview. URL `https://docs.csc.fi/computing/overview/`, viitattu 14.2.2023.

[25] DOCS CSC. Performance checklist. URL `https://docs.csc.fi/computing/running/performance-checklist/`, viitattu 14.2.2023.

[26] DOCS CSC. Submitting a batch job. URL `https://docs.csc.fi/computing/running/submitting-jobs/`, viitattu 14.2.2023.

[27] DOCS CSC. Systems. URL `https://docs.csc.fi/computing/available-systems/`, viitattu 14.2.2023.

[28] DOCS CSC. Technical details about Puhti. URL `https://docs.csc.fi/computing/systems-puhti/`, viitattu 14.2.2023.

[29] DONGARRA, J. J., JA VAN DER STEEN, A. J. High-performance computing systems: Status and outlook. *Acta Numerica 21* (2012), 379–474.

[30] FLYNN, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54*, 12 (1966), 1901–1909.

[31] FLYNN, M. J., JA RUDD, K. W. Parallel architectures. *ACM computing surveys (CSUR) 28*, 1 (1996), 67–70.

[32] FORTUNE, S., JA WYLLIE, J. Parallelism in random access machines. Julkaisusarjassa *Proceedings of the tenth annual ACM symposium on Theory of computing* (San Diego, CA, USA, toukokuu 1978), Association for Computing Machinery, 114–118.

[33] FOX, G. C., OTTO, S. W., JA HEY, A. J. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel computing 4*, 1 (1987), 17–31.

[34] GERBESSIOTIS, A. V. Extending the BSP model for multi-core and out-of-core computing: MBSP. *Parallel Computing 41* (2015), 90–102.

[35] GERBESSIOTIS, A. V., JA VALIANT, L. G. Direct bulk-synchronous parallel algorithms. Julkaisusarjassa *SWAT '92: Proceedings of the Third Scandinavian Workshop on Algorithm Theory 1992* (Helsinki, Finland, heinäkuu 1992), Springer, 1–18.

[36] GIBBONS, P. B. A more practical PRAM model. Julkaisusarjassa *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures* (Santa Fe, NM, USA, kesäkuu 1989), Association for Computing Machinery, 158–168.

[37] GIBBONS, P. B., MATIAS, Y., JA RAMACHANDRAN, V. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science 196*, 1–2 (1998), 3–29.

[38] GIBBONS, P. B., MATIAS, Y., JA RAMACHANDRAN, V. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing 28*, 2 (1998), 733–769.

[39] GIBBONS, P. B., MATIAS, Y., JA RAMACHANDRAN, V. Can a shared-memory model serve as a bridging model for parallel computation? *Theory of Computing Systems 32*, 3 (1999), 327–359.

[40] GONÇALVES, R., AMARIS, M., OKADA, T., BRUEL, P., JA GOLDMAN, A. OpenMP is not as easy as it appears. Julkaisusarjassa *HICSS '16: Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS)* (Koloa, HI, USA, tammikuu 2016), IEEE, 5742–5751.

[41] GUSTAFSON, J. L. Reevaluating Amdahl's law. *Communications of the ACM 31*, 5 (1988), 532–533.

[42] HAATAJA, J., HEIKONEN, J., LEINO, Y., RAHOLA, J., RUOKOLAINEN, J., JA SAVOLAINEN, V. *Numeeriset menetelmät käytännössä*. CSC — Tieteellinen laskenta Oy, Espoo, 2002.

[43] HAATAJA, J., JA MUSTIKKAMÄKI, K. *Rinnakkaisohjelmointi MPI:llä*. CSC — Tieteellinen laskenta Oy, Espoo, 2001.

[44] HANDHIKA, T., BUSTAMAM, A., ERNASTUTI, JA KERAMI, D. Data preprocessing for determining outer/inner parallelization in the nested loop problem using OpenMP. Julkaisusarjassa *AIP Conference Proceedings* (Depok, Jawa Barat, Indonesia, marraskuu 2016), AIP Publishing LLC, 030138.

[45] HARRIS, T. J. A survey of PRAM simulation techniques. *ACM Computing Surveys (CSUR) 26*, 2 (1994), 187–206.

[46] HU, L., CHE, X., JA ZHENG, S.-Q. A closer look at GPGPU. *ACM Computing Surveys (CSUR) 48*, 4 (2016), 1–20.

[47] JOSÉ, W. M., SILVA, A. R., VÉSTIAS, M. P., JA NETO, H. C. Algorithm-oriented design of efficient many-core architectures applied to dense matrix multiplication. *Analog Integrated Circuits and Signal Processing 82*, 1 (2015), 147–158.

[48] KANG, S. J., LEE, S. Y., JA LEE, K. M. Performance comparison of OpenMP, MPI, and MapReduce in practical problems. *Advances in Multimedia 2015* (2015).

[49] KASANEN, E., LUKKA, K., JA SIITONEN, A. The constructive approach in management accounting research. *Journal of management accounting research 5*, 1 (1993), 243–264.

[50] KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification, Version v3.0.12*, 2022.

[51] KOMATSU, K., MOMOSE, S., ISOBE, Y., WATANABE, O., MUSA, A., YOKOKAWA, M., AOYAMA, T., SATO, M., JA KOBAYASHI, H. Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. Julkaisusarjassa *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA, marraskuu 2018), IEEE, 685–696.

[52] KUNG, H., JA LEISERSON, C. E. Systolic arrays (for VLSI). Julkaisusarjassa *Sparse Matrix Proceedings 1978* (Knoxville, TN, USA, marraskuu 1979), Society for industrial and applied mathematics, 256–282.

[53] LAI, C.-N. Constructing all shortest node-disjoint paths in torus networks. *Journal of Parallel and Distributed Computing 75* (2015), 123–132.

[54] LANG, S. *Linear algebra*, 3rd ed. Springer-Verlag, New York, NY, USA, 1987.

[55] LEE, H.-J., ROBERTSON, J. P., JA FORTES, J. A. Generalized Cannon's algorithm for parallel matrix multiplication. Julkaisusarjassa *ICS '97: Proceedings of the 11th international conference on Supercomputing* (Vienna Austria, heinäkuu 1997), Association for Computing Machinery, 44–51.

[56] LI, J., SKJELLUM, A., JA FALGOUT, R. D. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience 9*, 5 (1997), 345–389.

[57] LI, X., JA MALEK, M. Analysis of speedup and communication/computation ratio in multiprocessor systems. Julkaisusarjassa *Proceedings. Real-Time Systems Symposium* (Huntsville, AL, USA, joulukuu 1988), IEEE, 282–288.

[58] LU, P.-J., LAI, M.-C., JA CHANG, J.-S. A survey of high-performance interconnection networks in high-performance computer systems. *Electronics 11*, 9 (2022), 1369.

[59] LUKKA, K. The key issues of applying the constructive approach to field research. Kirjassa *Management expertise for the new millennium: In commemoration of the 50th anniversary of the Turku School of Economics and Business Administration*, T. Reponen, Ed. Turku School of Economics and Business Administration, 2000, ss. 113–128.

[60] MAGGS, B. M., MATHESON, L. R., JA TARJAN, R. E. Models of parallel computation: A survey and synthesis. Julkaisusarjassa *Proceedings of the Twenty-Eighth*

Annual Hawaii International Conference on System Sciences (Maui, HI, USA, tammikuu 1995), vol. 2, IEEE, 61–70.

[61] MALLÓN, D. A., TABOADA, G. L., TEIJEIRO, C., TOURINO, J., FRAGUELA, B. B., GÓMEZ, A., DOALLO, R., JA MOURINO, J. C. Performance evaluation of MPI, UPC and OpenMP on multicore architectures. Julkaisusarjassa *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting* (Espoo, Finland, syyskuu 2009), Springer, 174–184.

[62] MAXIMA. Maxima 5.46.0 manual. URL `https://maxima.sourceforge.io/docs/manual/maxima_singlepage.html`, viitattu 11.3.2023.

[63] MAZOUZ, A., TOUATI, S., JA BARTHOU, D. Study of variations of native program execution times on multi-core architectures. Julkaisusarjassa *2010 International Conference on Complex, Intelligent and Software Intensive Systems* (Krakow, Poland, helmikuu 2010), IEEE, 919–924.

[64] MENTONE, A., DI LUCCIO, D., LANDOLFI, L., KOSTA, S., JA MONTELLA, R. CUDA virtualization and remoting for GPGPU based acceleration offloading at the edge. Julkaisusarjassa *Internet and Distributed Computing Systems: 12th International Conference, IDCS 2019* (Naples, Italy, lokakuu 2019), Springer, 414–423.

[65] MESSAGE PASSING INTERFACE FORUM. *MPI: A Message-Passing Interface Standard Version 4.0*, 2021.

[66] MICHAILIDIS, P. D., JA MARGARITIS, K. G. Scientific computations on multicore systems using different programming frameworks. *Applied Numerical Mathematics 104* (2016), 62–80.

[67] NGOKO, Y., JA TRYSTRAM, D. Revisiting Flynn's classification: The portfolio approach. Julkaisusarjassa *Euro-Par 2017: Parallel Processing Workshops* (Santiago de Compostela, Spain, elokuu 2017), Springer, 227–239.

[68] NULL, L., JA LOBUR, J. *Essentials of Computer Organization and Architecture*. Jones and Bartlett Publishers, Sudbury, MA, USA, 2003.

[69] NURIYEV, E., RICO-GALLEGO, J.-A., JA LASTOVETSKY, A. Model-based selection of optimal MPI broadcast algorithms for multi-core clusters. *Journal of Parallel and Distributed Computing 165* (2022), 1–16.

[70] NVIDIA. *CUDA C++ Programming Guide*, 2022.

[71] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP application programming interface Version 4.5*, 2021.

[72] PACHECO, P., JA MALENSEK, M. *An introduction to parallel programming*, 2nd ed. Elsevier, Cambridge, MA, USA, 2022.

[73] PJEŠIVAC-GRBOVIĆ, J., ANGSKUN, T., BOSILCA, G., FAGG, G. E., GABRIEL, E., JA DONGARRA, J. J. Performance analysis of MPI collective operations. *Cluster Computing 10*, 2 (2007), 127–143.

[74] RICO-GALLEGO, J. A., DÍAZ-MARTÍN, J. C., MANUMACHU, R. R., JA LASTOVETSKY, A. L. A survey of communication performance models for high-performance computing. *ACM Computing Surveys (CSUR) 51*, 6 (2019), 1–36.

[75] RUSSELL, R. M. The CRAY-1 computer system. *Communications of the ACM 21*, 1 (1978), 63–72.

[76] SCHREIBER, R. A few bad ideas on the way to the triumph of parallel computing. *Journal of Parallel and Distributed Computing 74*, 7 (2014), 2544–2547.

[77] SUN, X.-H., JA CHEN, Y. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and distributed Computing 70*, 2 (2010), 183–188.

[78] TOUATI, S.-A.-A., WORMS, J., JA BRIAIS, S. The speedup-test: A statistical methodology for programme speedup analysis and computation. *Concurrency and computation: practice and experience 25*, 10 (2013), 1410–1426.

[79] UPC CONSORTIUM. *UPC language and library specifications, Version 1.3*, 2013.

[80] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[81] VALIANT, L. G. A bridging model for multi-core computing. *Journal of Computer and System Sciences 77*, 1 (2011), 154–166.

[82] VAN DE GEIJN, R. A., JA WATTS, J. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience 9*, 4 (1997), 255–274.

[83] WILLIAMS, T., JA KELLEY, C. *Gnuplot 5.2*, 2019.

[84] YUAN, L., ZHANG, Y., TANG, Y., RAO, L., JA SUN, X. LogGPH: A parallel computational model with hierarchical communication awareness. Julkaisusarjassa *2010 13th IEEE International Conference on Computational Science and Engineering* (Hong Kong, China, joulukuu 2010), IEEE, 268–274.

[85] ZHENG, Z., CHEN, X., WANG, Z., SHEN, L., JA LI, J. Performance model for OpenMP parallelized loops. Julkaisusarjassa *Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE)* (Changchun, China, joulukuu 2011), IEEE, 383–387.

A Peräkkäiskoodi

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4
5  int main(int argc, char **argv) {
6      double *A, *B, *C;
7      int i, j, k;
8      const int n = strtol(argv[1], NULL, 10);
9      struct timeval start, end;
10     double time;
11
12     A = (double *)malloc(n*n*sizeof(double));
13     B = (double *)malloc(n*n*sizeof(double));
14     C = (double *)malloc(n*n*sizeof(double));
15
16
17     /************************/
18     /* Matriisien kokoaminen */
19     /************************/
20     // A tallennetaan riveittäin.
21     for (i = 0; i < n*n; i++) {
22         A[i] = drand48();
23     }
24     // B tallennetaan sarakkeittain.
25     for (i = 0; i < n*n; i++) {
26         B[i] = drand48();
27     }
28
29
30     /***********/
31     /* Laskenta */
32     /***********/
33     gettimeofday(&start, NULL);
34     for (i = 0; i < n; i++) {
35         for (j = 0; j < n; j++) {
36             C[i*n+j] = 0;
37             for (k = 0; k < n; k++) {
```

```
38                  C[i*n+j] += A[i*n+k] * B[j*n+k];
39              }
40          }
41      }
42      gettimeofday(&end, NULL);
43
44      // Aika millisekunteina.
45      time = (end.tv_sec - start.tv_sec) * 1000
46          + (double) (end.tv_usec - start.tv_usec) / 1000;
47      printf("%f\n", time);
48
49
50      free(A); free(B); free(C);
51      return 0;
52  }
```

B MPI-koodi

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <mpi.h>
5
6  #define ROOT 0
7  #define n 8
8
9  int main(int argc, char **argv) {
10     double *A, *B, *C, *A_sendbuf, *B_sendbuf;
11     double local_start, local_finish, local_elapsed, elapsed;
12     int N, id, i, j, k;
13
14     MPI_Init(&argc, &argv);
15     MPI_Comm_rank(MPI_COMM_WORLD, &id);
16     MPI_Comm_size(MPI_COMM_WORLD, &N);
17
18     // Rivien tai sarakkeiden määrä yhtä tulomatriisin lohkoa kohti.
19     const int RC_PER_BLOCK = (int) (n / sqrt((double) N));
20     // Lohkojen määrä matriisin sivua kohti.
21     const int BLOCKS_PER_LEN = n / RC_PER_BLOCK;
22
23     double A_local[RC_PER_BLOCK*n], B_local[RC_PER_BLOCK*n],
24         C_local[RC_PER_BLOCK*RC_PER_BLOCK];
25
26     if (id == ROOT) {
27         A = (double *)malloc(n*n*sizeof(double));
28         B = (double *)malloc(n*n*sizeof(double));
29         C = (double *)malloc(n*n*sizeof(double));
30
31
32         /************************/
33         /* Matriisien kokoaminen */
34         /************************/
35         // A tallennetaan riveittäin.
36         for (i = 0; i < n*n; i++) {
37             A[i] = drand48();
```

```
38          }
39          // B tallennetaan sarakkeittain.
40          for (i = 0; i < n*n; i++) {
41              B[i] = drand48();
42          }
43
44
45          /*****************/
46          /* Lähetyspuskurit */
47          /*****************/
48          // Koska Scatter-rutiini ei salli päällekkäisen datan
49          // lähettämistä eri prosesseille, eri prosessien tarvitsema
50          // data laitetaan lähetyspuskuriin yksinkertaisesti peräkkäin.
51          A_sendbuf = (double *)malloc(N
52                                       *RC_PER_BLOCK*n*sizeof(double));
53          B_sendbuf = (double *)malloc(N
54                                       *RC_PER_BLOCK*n*sizeof(double));
55          // Tallennetaan A lähetyspuskuriin.
56          for (i = 0; i < N; i++) {
57              // Yhdessä lohkossa on RC_PER_BLOCK*n alkiota.
58              for (j = 0; j < RC_PER_BLOCK*n; j++) {
59                  A_sendbuf[i*RC_PER_BLOCK*n+j] =
60                      A[(i/BLOCKS_PER_LEN)*RC_PER_BLOCK*n+j];
61              }
62          }
63          // Tallennetaan B lähetyspuskuriin.
64          for (i = 0; i < N; i++) {
65              // Yhdessä lohkossa on RC_PER_BLOCK*n alkiota.
66              for (j = 0; j < RC_PER_BLOCK*n; j++) {
67                  B_sendbuf[i*RC_PER_BLOCK*n+j] =
68                      B[(i%BLOCKS_PER_LEN)*RC_PER_BLOCK*n+j];
69              }
70          }
71      }
72
73      /****************/
74      /* Ajanotto alkaa */
75      /****************/
76      MPI_Barrier(MPI_COMM_WORLD);
77      local_start = MPI_Wtime();
78
79
```

```
80      /**********************/
81      /* Jaetaan prosesseille */
82      /**********************/
83      MPI_Scatter(A_sendbuf, RC_PER_BLOCK*n, MPI_DOUBLE,
84                  A_local, RC_PER_BLOCK*n, MPI_DOUBLE,
85                  ROOT, MPI_COMM_WORLD);
86      MPI_Scatter(B_sendbuf, RC_PER_BLOCK*n, MPI_DOUBLE,
87                  B_local, RC_PER_BLOCK*n, MPI_DOUBLE,
88                  ROOT, MPI_COMM_WORLD);
89
90
91      /***********/
92      /* Laskenta */
93      /***********/
94      for (i = 0; i < RC_PER_BLOCK; i++) {
95          for (j = 0; j < RC_PER_BLOCK; j++) {
96              C_local[i*RC_PER_BLOCK+j] = 0;
97              for (k = 0; k < n; k++) {
98                  C_local[i*RC_PER_BLOCK+j] +=
99                      A_local[i*n+k] * B_local[j*n+k];
100             }
101         }
102     }
103
104
105     /*****************/
106     /* Tulosten keruu */
107     /*****************/
108     MPI_Gather(C_local, RC_PER_BLOCK*RC_PER_BLOCK, MPI_DOUBLE,
109                C, RC_PER_BLOCK*RC_PER_BLOCK, MPI_DOUBLE,
110                ROOT, MPI_COMM_WORLD);
111
112
113     /******************/
114     /* Ajanotto päättyy */
115     /******************/
116     local_finish = MPI_Wtime();
117     local_elapsed = (local_finish - local_start) * 1000;
118     MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
119                MPI_MAX, ROOT, MPI_COMM_WORLD);
120     // Aika millisekunteina.
121     if (id == ROOT)
```

```
122             printf("%f\n", elapsed);
123
124         if (id == ROOT) {
125             free(A); free(B); free(C); free(A_sendbuf); free(B_sendbuf);
126         }
127         MPI_Finalize();
128         return(0);
129     }
```

C OpenMP-koodi

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <omp.h>
5
6  int main(int argc, char **argv) {
7      double *A, *B, *C;
8      double time_start, time_finish, time_elapsed;
9      int i, j;
10
11     // Matriisin dimensio.
12     const int n = strtol(argv[1], NULL, 10);
13     // Säikeiden määrä, sama kuin lohkojen määrä.
14     const int N = strtol(argv[2], NULL, 10);
15
16     // Rivien tai sarakkeiden määrä yhtä tulomatriisin lohkoa kohti.
17     const int RC_PER_BLOCK = (int) (n / sqrt((double) N));
18
19     // Lohkojen määrä matriisin sivua kohti.
20     const int BLOCKS_PER_LEN = n / RC_PER_BLOCK;
21
22     A = (double *)malloc(n*n*sizeof(double));
23     B = (double *)malloc(n*n*sizeof(double));
24     C = (double *)malloc(n*n*sizeof(double));
25
26
27     /************************/
28     /* Matriisien kokoaminen */
29     /************************/
30     // A tallennetaan riveittäin.
31     for (i = 0; i < n*n; i++) {
32         A[i] = drand48();
33     }
34     // B tallennetaan sarakkeittain.
35     for (i = 0; i < n*n; i++) {
36         B[i] = drand48();
37     }
```

```
38
39
40      /*****************/
41      /* Ajanotto alkaa */
42      /*****************/
43      time_start = omp_get_wtime();
44  #   pragma omp parallel num_threads(N)
45      {
46          int id, row, col, row_start, col_start, row_max, col_max, kk;
47          id = omp_get_thread_num();
48
49
50          /**********/
51          /* Laskenta */
52          /**********/
53          row_start = (id / BLOCKS_PER_LEN) * RC_PER_BLOCK;
54          row_max = row_start + RC_PER_BLOCK;
55          col_start = (id % BLOCKS_PER_LEN) * RC_PER_BLOCK;
56          col_max = col_start + RC_PER_BLOCK;
57          for (row = row_start; row < row_max; row++) {
58              for (col = col_start; col < col_max; col++) {
59                  C[row*n+col] = 0;
60                  for (kk = 0; kk < n; kk++) {
61                      C[row*n+col] += A[row*n+kk]*B[col*n+kk];
62                  }
63              }
64          }
65      }
66
67
68      /******************/
69      /* Ajanotto päättyy */
70      /******************/
71      time_finish = omp_get_wtime();
72      time_elapsed = (time_finish - time_start) * 1000;
73      // Aika millisekunteina.
74      printf("%f\n", time_elapsed);
75
76      free(A); free(B); free(C);
77      return(0);
78  }
```