Rinnakkaisalgoritmit ja rinnakkaistietokoneet

Martti Penttonen
Tietojenkäsittelytieteen ja sovelletun matematiikan laitos
Kuopion yliopisto

18. lokakuuta 2001

1 Johdanto

Tietokoneen toimintaperiaatteita kuvattaessa käytetään usein lisänimeä *von Neumannin tietokone*. Tällöin tehdään kunniaa von Neumannin ja Turingin kaltaisille tutkijoille, jotka loivat tietokoneen ja ohjelmoinnin teoreettisen perustan ja selittivät, mitä tietokoneella on mahdollista tehdä [Davis 2000]. Von Neumannin ansioksi mainitaan erityisesti se, että tietokoneen toiminnan määräävä ohjelma on "datana" tietokoneen muistissa ja siten ohjelmaa vaihtamalla tietokone saadaan suorittamaan eri tehtävää. Von Neumann on saanut nimensä myös käsitteeseen *von Neumannin pullonkaula*. Tällä tarkoitetaan sitä, että tietokoneessa olevan tiedon käsittely tapahtuu yhdessä prosessorissa. Kun tietoa haetaan tietokoneen muistista prosessoriin käsiteltäväksi lähes joka käskyllä, tulee ainoasta prosessorista tietokoneen nopeutta rajoittava pullonkaula. Tehontarpeen alati kasvaessa, on von neumannilainen tietokonearkkitehtuuri joutunut kriittisen tarkastelun kohteeksi ja kehitys on johtanut kohti moniprosessorisia tietokoneita, rinnakkaistietokoneita.

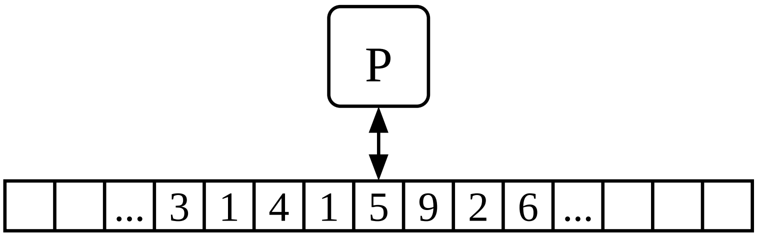
Ei ole kuitenkaan itsestään selvää, että n prosessoria suoriutuu tehtävästä n kertaa nopeammin kuin yksi prosessori. Sata metriä pitkän ojan kaivaminen on helppo jakaa sadalle työntekijälle, mutta miten jaetaan sata metriä syvän kaivon kaivaminen sadalle työntekijälle? Rinnakkaisalgoritmien löytymisen vaikeuteen voi olla monenlaisia syitä:

- Ei ole olemassakaan algoritmia, joka suoriutuu suoritettavasta tehtävästä n prosessorilla n kertaa nopeammin kuin yhdellä prosessorilla (suurella n:n arvolla).
- Ei vielä tunneta algoritmia, joka suoriutuu n prosessorilla n kertaa nopeammin.
- Vaikka tehokas algoritmi onkin olemassa, n-prosessorinen rinnakkaistietokone ei selviydy tehtävästä n kertaa nopeammin arkkitehtonisten rajoitteiden vuoksi.

Edelläkerrotusta seuraa, että peräkkäisohjelmien rinnakkaistuminen ei ole itsestäänselvyys, koska se ei ole aina edes mahdollista. Toisaalta tunnetaan myös mm. tulos [Mak 1997], jonka mukaan tietyin oletuksin ajassa $T(n)$ peräkkäistietokoneella suoritettava n-kokoinen tehtävä voidaan suorittaa lukuun $\sqrt{T(n)}\log T(n)$ verrannollisessa ajassa. Kuitenkin, vaikka rinnakkaisalgoritmi tunnettaisiinkin, tehoa ei aina saada irti koneesta, koska esim. tiedon siirtelyyn prosessorien välillä tuhlautuu liian paljon aikaa. Tämän kirjoituksen tarkoituksena on esitellä rinnakkaisalgoritmien ja rinnakkaistietokoneiden suunnitteluperiaatteita sekä hahmotella tietokonearkkitehtuureja, joilla rinnakkaisalgoritmeja voitaisiin ohjelmoijaystävällisesti ja tehokkaasti suorittaa.

2 Rinnakkaisalgoritmit

Von Neumannin arkkitehtuurin mukaista tietokonetta voidaan mallittaa kuvan 1 mukaisella kaaviolla.



Kuva 1: Peräkkäistietokoneen malli.

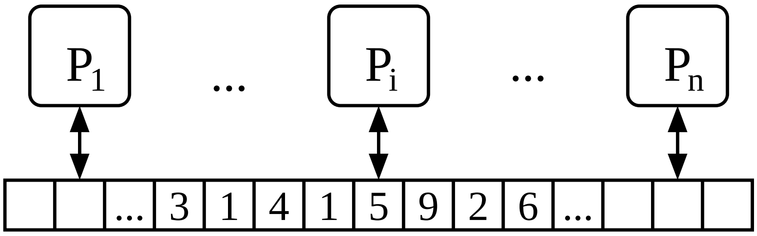
Siinä neliönmuotoinen laatikko tarkoittaa prosessoria, joka suorittaa ohjelmaa, ja pitkulainen laatikko tarkoittaa muistipaikkojen jonoa. Tässä pelkistetyssä mallissa ei tehdä eroa erityyppisten muistien välillä vaan muistiavaruuden oletetaan olevan yhtenäinen. Ohjelmaa suorittaessaan prosessori hakee muistipaikoista dataa (lukuja) viipeettä ja tallentaa käsittelyn tuloksia muistipaikkoihin. Koneen toiminnan määrää ohjelma, joka koostuu aritmeettisista operaatioista, muistioperaatioista ja haarautumisoperaatioista. Lukemisen helpottamiseksi esitämme ohjelmat rakenteisella "korkean tason" ohjelmointikielellä. Kuvan 2 ohjelma lajittelee lukujonon $L = L[1..n]$ luvut kasvavaan järjestykseen. Siinä *satunnainen(X)* tarkoittaa jonon X satunnaista alkiota ja *partitio(r,X)* jakaa jonon X kolmeen osaan, jotka sisältävät rajalukua r pienemmät, sen kanssa yhtäsuuret, ja sitä suuremmat luvut. Siten algoritmin suoritusajan suuruusluokan määrää palautuskaava $T(n) = 2T(n/2) + n$, jota toistaen nähdään, että $T(n)$ on luokkaa $n \log n$. Palautuskaavassa $2T(n/2)$ viittaa kahden puolta pienemmän lajittelutehtävän rekursiiviseen suoritusaikaan ja n partition suoritusaikaan. (Tarkka analyysi ei ole aivan näin yksinkertainen, sillä satunnaisuus ei jaa n-kokoista tehtävää kahdeksi $n/2$-kokoiseksi tehtäväksi. Kuitenkin algoritmi toimii suurella todennäköisyydellä näin nopeasti.)

Yhteistä muistia käyttävää rinnakkaistietokonetta voidaan mallittaa kuvan 3 mukaisella kaaviolla.

Olennainen ero kuvan 1 peräkkäistietokoneeseen on se, että nyt prosessoreita on useita. Oletamme, että kaikki prosessorit suorittavat samaa ohjelmaa, ei kuitenkaan joka hetki samaa kohtaa ohjelmassa. Ainoa olennainen lisäys peräkkäiskoneen käskykantaan on käsky, jolla prosessori saa tietoonsa (aivan kuin muistihaulla) oman tunnuksensa — tällainen

proc pikalajittelu(L)
if $|L| <= 1$ **then**
  return L
else
  r=satunnainen(L)
  $L_<, L_=, L_>$=partitio(r, L)
  for $i \in \{<, >\}$ **do**
    L_i=pikalajittelu(L_i)
  return $L_< * L_= * L_>$

Kuva 2: Lajittelu peräkkäisalgoritmilla.



Kuva 3: Yhteistä muistia käyttävän rinnakkaistietokoneen malli.

käsky tarvitaan prosessorien työnjaossa, jotta prosessori tunnistaisi omat työnsä. Korkean tason ohjelmointikielessä ainoa lisäys on rinnakkain suoritettava **pardo**-silmukkarakenne. Kuvan 2 ohjelma muutetaan rinnakkaisohjelmaksi 4 muuttamalla **do**-silmukan peräkkäin suoritettavat rekursiiviset kutsut **pardo**-silmukalla peräkkäin suoritettaviksi. (Alirutiini *partitio(r,X)* rinnakkaistetaan kuvan 6 ohjelmassa.)

Rinnakkaisalgoritmin **pardo**-silmukan jokainen indeksi (esimerkissämme $<$ ja $>$) aloittaa uuden rinnakkain suoritettavan prosessin. Koska satunnaisuuden ansiosta raja r jakaa jonon L todennäköisesti suunnilleen keskeltä, n alkion lajittelutehtävä tuottaa kaksi rinnakkain suoritettavaa suunnilleen $n/2$ alkion lajittelutehtävää, ne kumpikin kaksi $n/4$ alkion tehtävää jne. Noin $\log n$ vaiheen jälkeen alkuperäinen tehtävä on jakautunut (lähes) n rinnakkain suoritettavaan lajittelutehtävään, joiden suorituksen ja yhdistämisen jälkeen koko tehtäväkin on suoritettu. Koska partitio voidaan muodostaa suunnilleen ajassa $\log n$, rinnakkaisen pikalajittelun suoritusajaksi voidaan arvioida $T(n) = T(n/2) + \log n$, jota toistaen voidaan $T(n)$:n arvioida olevan suuruusluokkaa $\log^2 n$, suurella todennäköisyydellä. (Partition vaatima suoritusaika $\log n$ arvioidaan myöhemmin.)

Toisena esimerkkinä tarkastelemme kuvan 5 ohjelmaa, joka laskee lukujonon $L = L[1..n]$ *alkusummat*, ts. tuottaa jonon, jonka i:s jäsen on summa $L[1] + L[2] + \ldots + L[i]$.

Alkusumma-algoritmin suoritusaika on **do**-silmukan suorituskertojen määrä $\log n$, sillä sisemmän **pardo**-silmukan suorittamiseen riittää vakioaika. Alkusumma-algoritmilla on rinnakkaisalgoritmiikassa paljon käyttöä. Sen avulla voidaan esimerkiksi kuvan 6 tavalla

```
proc pikalajittelu(L)
if |L| <= 1 then
    return L
else
    r=satunnainen(L)
    L<, L=, L>=partitio(r, L)
    for i ∈ {<, >} pardo
        Li=pikalajittelu(Li)
    return L< * L= * L>
```

Kuva 4: Lajittelu rinnakkaisalgoritmilla.

```
proc alkusumma(L)
for i ∈ {1 ... ⌈log(n)⌉} do
    for j ∈ {2^(i-1) + 1 ... n} pardo
        L[j] = L[j - 2^(i-1)] + L[j]
return L
```

Kuva 5: Alkusummien laskeminen.

muodostaa pikalajittelualgoritmissa tarvittava partitio.

Viimeisenä esimerkkinä tarkastelemme tehtävää, joka ensiksi tuntuu vaikeasti rinnakkaistuvalta. *Äärellinen automaatti* on matemaattinen kone, jonka avulla mallitetaan tilaa muuttavia järjestelmiä. Äärellisen automaatin määrittelee tilanmuutosfunktio, joka kertoo, miten eri syötteet muuttavat automaatin tilaa. Esimerkiksi kuvassa 7 annettu funktio määrittelee erään äärellisen automaatin. Jos automaatti on aluksi tilassa 0, se siirtyy syötteellä *abaaba* tilojen 1,2,5,2,3 kautta tilaan 5. Aina, kun luetun tekstin kolme viimeistä merkkiä ovat *aab*, automaatti on tilassa 3, ja aina, kun luetun tekstin kolme viimeistä merkkiä ovat *aba*, automaatti on tilassa 5.

Koska äärellisen automaatin toiminta on luonteeltaan peräkkäistä — uusi tila riippuu edellisistä, tuntuu aluksi vaikealta keksiä rinnakkaisohjelma, joka suorittaisi automaatin laskun. Tehtävä ratkeaa, kun yhdestä tilasta alkavan tilajonon asemesta tarkastellaan merkkijonoon liittyvää tilanmuutosfunktiota. Merkitköön $f_w(p)$ sitä tilaa q, johon syötejono w vie automaatin tilasta p lähtien. Tyhjälle merkkijonolle ϵ $f_\epsilon = Id$ eli identiteettifunktio, jolle $Id(p) = p$. Määritelmän mukaisesti taulukon 7 ylempi rivi määrittelee f_a:n ja alempi rivi määrittelee f_b:n. Soveltamalla tilafunktiota kaksi kertaa peräkkäin nähdään, että $f_a \circ f_b = f_{ab} = (4, 3, 3, 4, 4, 3)$, missä $\circ$ tarkoittaa funktioiden yhdistämistä. Merkkijono *abaaba* vie automaatin tilasta 0 tilaan 5 eli $f_{abaaba}(0) = 5$. Kuvan 8 ohjelma laskee funktion f_w. Algoritmin suoritusaika, joka on luokkaa $\log n$, johdetaan palautuskaavasta $T(n) = T(n/2) + 1$.

Esimerkkimme osoittavat, että rinnakkaisohjelmoinnin ei tarvitse olla monimutkaisempaa kuin peräkkäisohjelmoinnin. Itse asiassa se voi jopa olla helpompaakin, sillä rinnakkais-

proc partitio(r, L)
for $i \in \{1 \ldots n\}$ **pardo** $P[i] = (L[i] < r)$ % 1="tosi", 0="epätosi"
P=alkusumma(P)
$p = P[n]$ % montako pientä oli
for $i \in \{1 \ldots n\}$ **pardo** $S[i] = (L[i] > r)$
S=alkusumma(S)
$s = S[n]$ % montako suurta oli
m=n-p-s % näin monesti r
for $i \in \{1 \ldots m\}$ **pardo** $L_=[i] = r$
for $i \in \{1 \ldots n\}$ **pardo**
  if $L[i] < r$ **then** $L_<[P[i]] = L[i]$
  elsif $L[i] > r$ **then** $L_>[S[i]] = L[i]$
return $L_<, L_=, L_>$

Kuva 6: Partition muodostaminen.

f	0	1	2	3	4	5
a	1	2	2	5	5	2
b	0	4	3	0	0	4

Kuva 7: Äärellinen automaatti.

ohjelmoijan ei tarvitse pohtia, tuleeko $L_<$ lajitella ennen kuin $L_>$ — rinnakkaisen ilmiön koodaaminen peräkkäiseksi voi olla joskus hankalaa. Koska ohjelmointi, ja varsinkin oikein toimivien ohjelmien tekeminen, on joka tapauksessa vaikeaa, ohjelmointikieli ja ympäristö eivät saisi tuottaa ylimääräisiä vaikeuksia ohjelmointiin. Siksi olisi toivottavaa, että peräkkäislaskennan ja rinnakkaislaskennan ero olisi vain **pardo**-silmukan lisääminen. Algoritmitutkimuksen ansiosta käytettävissämme on runsas rinnakkaisalgoritmikirjallisuus (katso esim. [Jájá 1992, Penttonen 1998]). Tutkijoiden motiivina on ollut, paitsi käytännön tarve, sen selvittäminen, kuinka paljon tietojenkäsittelytehtävissä on yleensä rinnakkaisuutta. Yleisen käsityksen mukaan kaikki "kiinnostavat" tehtävät, jotka voidaan suorittaa peräkkäiskoneella polynomiaalisessa ajassa, voidaan suorittaa rinnakkaiskoneella "polylog" ajassa, ts. ajassa joka on verrannollinen tehtävän koon logaritmin potenssiin.

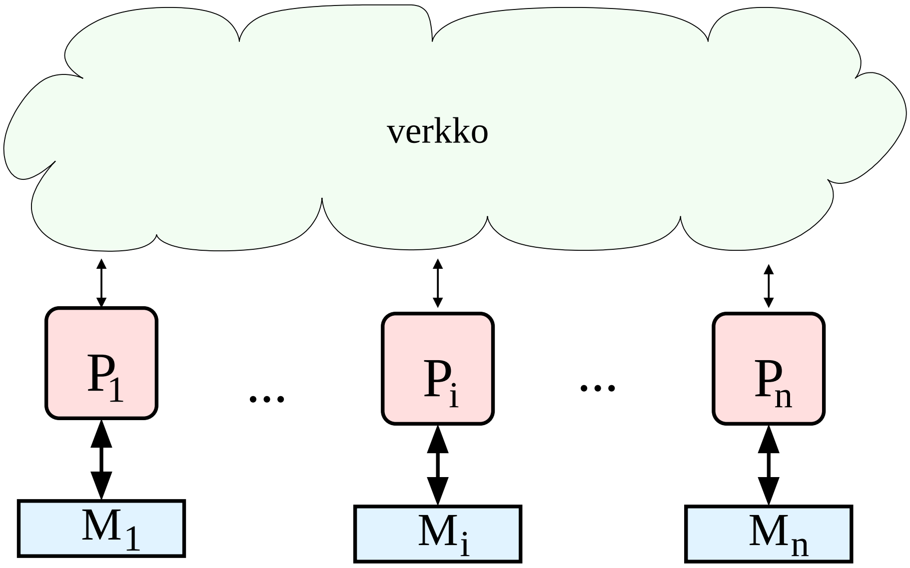
3 Hajautetun muistin malli

Yhteisen muistin konemallin vaivattomuus ohjelmoijalle perustuu juuri yhteiseen muistiin, joka on välittömästi kaikkien prosessorien käytettävissä. Pahaksi onneksi sellaisen muistin toteuttaminen on käytännössä ollut vaikeaa ja kallista. Erilaisin kytkentäratkaisuin ja nk. vektoriprosessorein toteutettuja yhteistä muistia käyttäviä supertietokoneita on ollut ja on [van der Steen et al.], mutta ne ovat kalliita ja vaikeasti laajennettavissa.

proc automaatti(f_a, f_b, w)
if $|w| = 1$ **then**
    return f_w
else
    Olkoon $w = u_1u_2$, missä $|u_1| = \lfloor |w|/2 \rfloor$ ja $|u_2| = \lceil |w|/2 \rceil$
    for $i \in \{1, 2\}$ **pardo**
        f_{u_i}=automaatti(f_a, f_b, u_i)
    return $f_{u_1} \circ f_{u_2}$

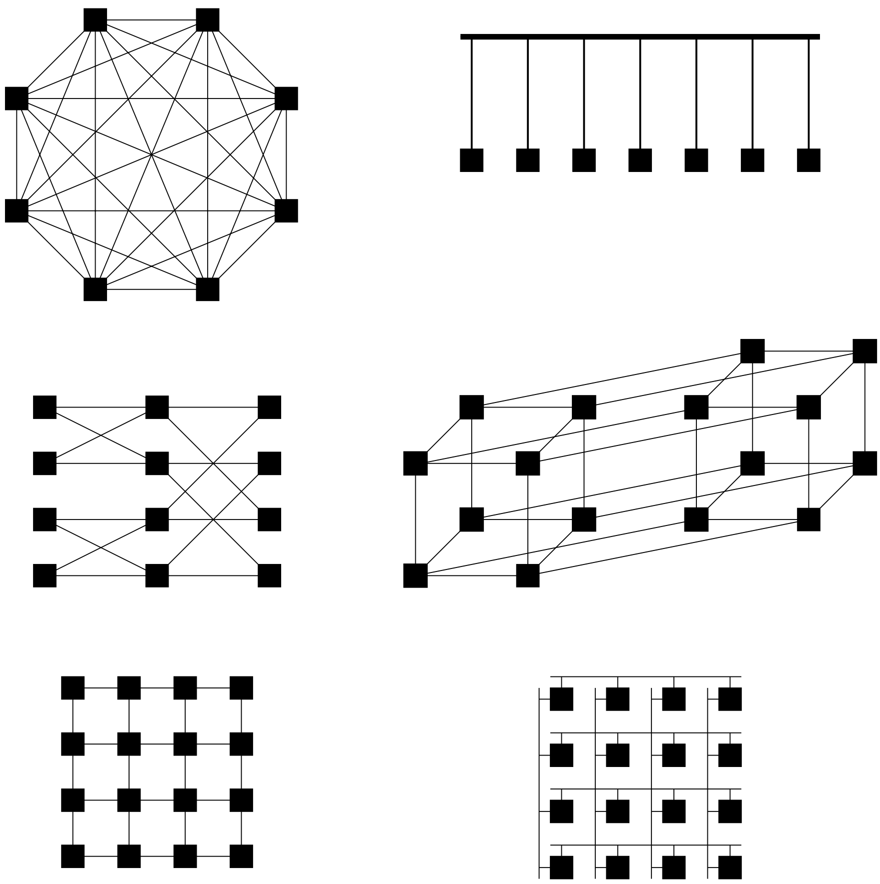
Kuva 8: Äärellinen automaatti rinnakkaisohjelmana.

Markkinakehityksen johdosta tietokonevalmistajien mielenkiinto on viime aikoina kohdistunut peräkkäistietokoneiden ja erityisesti henkilökohtaisten työasemien kehittämiseen. Työasemista saakin laskentatehoa halvimmalla. Tarvittaessa paljon laskentatehoa tulee houkuttelevaksi kuvan 9 kaltainen laskentamalli, jossa joukko erillisiä tietokoneita yhdistetään rinnakkaistietokoneeksi tietoliikenneverkon avulla.



Kuva 9: Hajautetun muistin malli.

Koska prosessorien välisen kommunikoinnin tulee olla nopeampaa kuin Internetissä suoritettavien peräkkäissovellusten tiedonsiirto, hajautetun muistin rinnakkaiskoneen tai työasemaklusterin verkko perustuu yleensä johonkin rakenteeltaan säännölliseen "tiheään" kytkentään. Kuvassa 10 on esitetty kuusi tapaa kytkeä prosessorit toisiinsa. Täydellisessä verkossa, hyperkuutiossa ja perhosessa etäisyydet ovat lyhyitä, mutta ristikkorakenne mahtuu paremmin kolmiuloitteiseen maailmaan. Erilaisia verkkoratkaisuja on esitelty mm. lähteissä [Almasi et al. 1994, van der Steen et al.]. Beowulf [Beowulf] ja Grid [Grid] tuovat hajautetun muistin rinnakkaislaskennan "jokamiehen" ulottuville. Tietokoneteollisuus on myös alkanut kehittää standardeja, jotka helpottaisivat ja tehostaisivat prosessorien välistä kommunikointia [InfiniBand].



Kuva 10: Verkkotopologioita: Täydellinen verkko, väylä, perhonen, hyperkuutio, ristikko ja väyläristikko.

Jos tällaisella n-prosessorisella *hajautetun muistin koneella* suoritetaan n toisistaan riippumatonta tehtävää, koneesta saadaan luonnollisesti n-kertainen teho verrattuna yhteen koneeseen. Mutta rinnakkaiskonetta tarvitaan yhden tehtävän nopeaan suorittamiseen. Tällöin tulee ratkaistavaksi:

1. Miten laskenta levitetään prosessoreille niin, että kuormitus on tasainen?
2. Miten data levitetään muistiin niin, että se on tehokkaasti käytettävissä?
3. Miten ratkaistaan tiedonsiirto niin, että aikaa ei kulu turhaan odottamiseen?

4. Miten hajautetun muistin konetta ohjelmoidaan?

Tavoitetilanne olisi se, että ohjelmoijan ei tarvitsisi tietää mitään prosessorien ja muistin määrästä ja sijainnista ja kuitenkin kone suorittaisi tehokkaasti (yhteenlaskettua prosessoritehoa vastaavasti) hänen ohjelmaansa.

Hajautetun muistin rinnakkaistietokoneita ohjelmoidaan yleisesti nk. *viestinvälitysmallin* mukaisesti. Tässä mallissa ohjelmaan kirjoitetaan käskyjä, jotka lähettävät viestejä prosessorien kesken. Viesteissä siirretään dataa tai alitehtäviä toisessa prosessorissa suoritettavaksi. Ylläolevan luettelon tehtävistä 1, 2 ja 4 jäävät ohjelmoijan vastuulle, 3 riippuu koneesta. Yleisimmin käytetty viestinvälitysmallin toteutus on MPI-ohjelmakirjasto (Message Passing Interface), jota voidaan käyttää monien ohjelmointikielten kanssa rinnakkaislaskentaan. MPI:n laaja käyttö todistaa, että se on käyttökelpoinen ratkaisu moneen laskentatehoa vaativaan tehtävään. Se ei kuitenkaan sovellu hienorakeisen rinnakkaisuuden toteutukseen, jossa prosessorit hyvin usein (lähes joka käskyllä) tarvitsevat yhteistä tietoa. MPI-ohjelmointi myös vaatii ohjelmoijalta peräkkäisohjelmointiin verrattuna ylimääräistä päänvaivaa ja resurssien hallinta voi käydä ylivoimaiseksi, ellei tehtävä ole riittävän säännöllinen rakenteeltaan. Siksi MPI tai viestinvälitykseen perustuva ohjelmointi ei saisi olla lopullinen ratkaisu korkean tason ohjelmointimalliksi. Kuvassa 11 on yksinkertainen esimerkki MPI-kirjaston käytöstä. Ohjelmassa esiintyy neljä MPI:n kuudesta yleisimmästä käskystä, mutta siitä puuttuvat MPI_Send ja MPI_Receive -käskyt.

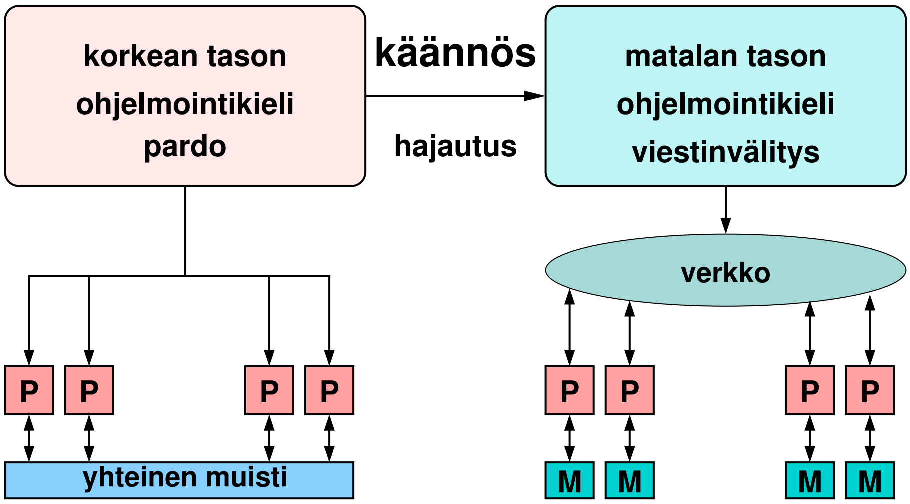
```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char **argv;
{
int rank,size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
printf("Terve! Olen prosessori %d / %d \n",rank,size);
MPI_Finalize();
return 0;
}
```

Kuva 11: “Terve!” rinnakkaisesti MPI:llä.

4 Yhteisen muistin malli hajautetun muistin koneessa

Yhteisen muistin malli on epäilemättä ohjelmoijalle helpompi, koska se vapauttaa ohjelmoijan tietokoneen resurssien hallinnasta. Kuitenkin suuren, nopean, suurelle prosessorimäärälle yhteisen ja halvan yhteisen muistin suora rakentaminen saattaa olla mahdotonta. Hajautetun muistin koneista sen sijaan saa tehoa halvalla. Kun myös tietoliikennetekniikassa kehitys on huimaa, on järkevää kysyä, voitaisiinko hajautetun muistin koneita käyttää yhteisen muistin mallin mukaiseen laskentaan.

Tässä luvussa tutkitaan sitä, millä edellytyksillä yhteisen muistin mallin mukainen rinnakkaisohjelmointi voitaisiin tehokkaasti toteuttaa hajautetun muistin koneessa, vrt kuva 12.



Kuva 12: Yhteisen muistin ohjelman suorittaminen hajautetulla muistilla.

Jos haluamme yleiskäyttöistä rinnakkaislaskentaa, meidän on hyväksyttävä seuraavat vaatimukset, jotka saattavat olla vaikeita toteuttaa:

1. Kaikki muistipaikat ovat kaikkien prosessorien saatavilla.

2. Mikä tahansa konekäsky voi sisältää viittauksen muistiin.

Kohdasta 1 seuraa, että yhteinen muisti on hajautettava prosessori/muisti -moduuleille jonkin säännön mukaisesti. Jos tämä hajautus tehdään huonosti, jotkin moduulit voivat kuormittua pahoin ja kone jumiutuu. Hajautus voidaan tehdä hyvin käyttäen *satunnaistettua hajautusta*. Sen sijaan mikään deterministinen hajautusfunktio ei toimi hyvin, sillä viipeen alaraja on vähintään luokkaa $\log^2 p/\log\log p$ vakioasteisessa verkossa, jossa on p prosessoria [Karlin et al. 1988]. Hajautuksen käyttö voi tuntua hullulta, koska olisi luonnollista sijoittaa "yhteenkuuluva" data lähekkäin. Muistin hallinnoinnin siirtäminen ohjelmoijan vastuulle on kuitenkin kohtuuttoman vaikeaa ja monimutkaisessa laskennassa johtaa suurella todennäköisyydellä ruuhkautumiseen. Satunnaistuksen etu on siinä, että se takaa muistin saatavuuden ruuhkitta suurella todennäköisyydellä.

Kohdasta 2 seuraa, että muistiviittauksia on paljon ja että laskun eteneminen riippuu datan saannista. Muistiviittausten määrä on kohtalokas, ellei verkko "vedä" tarpeeksi hyvin. Muistioperaation hitaus haittaa laskennan edistymistä, ellei prosessorilla ole muuta tekemistä. Kumpikin ongelma on ratkaistavissa.

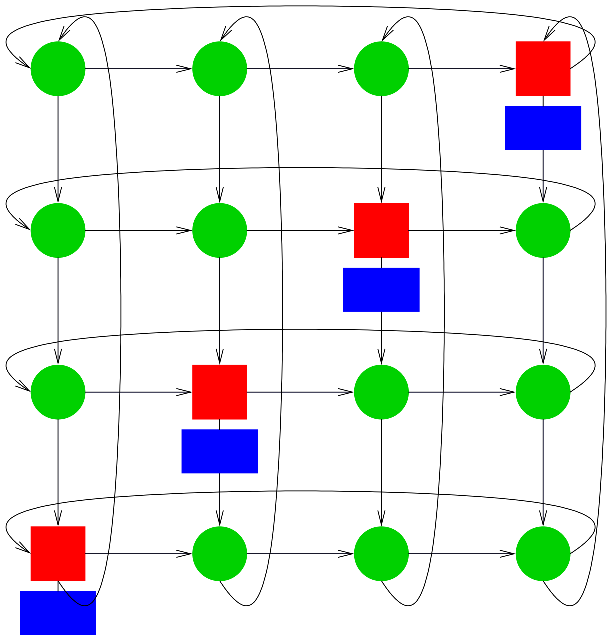
Viime vuosikymmeninä niin prosessorien, muistien kuin tietoliikenteenkin nopeus on kasvanut huimasti, ei kuitenkaan samalla tavalla. Tietoliikenteen nopeutuminen on ollut kaikkein huiminta — rinnakkaislaskennan kannalta tosin kytkentänopeuksissa on toivomisen varaa. Muistien nopeutuminen on ollut paljon hitaampaa kuin prosessorien. Kun keskusmuistin saantiajat ovat monikymmenkertaisia käskyn suoritukseen kuluvaan aikaan verrattuna, muistin hitautta on korvattu käyttämällä nopeita rekistereitä ja *välimuisteja*. Välimuistien käyttöön liittyy kuitenkin se ongelma, että tieto on päivitettävä varsinaiseen muistiin ennen kuin toinen muistiviittaus hakee sieltä vanhentunutta tietoa. Rinnakkaislaskennan tapauksessa välimuistien käyttö olisi vielä ongelmallisempaa, koska samaan muistiin viittaavia prosessoreja on paljon. Onneksi välimuisti voidaan korvata rinnakkaisuudella. Jos rinnakkaisalgoritmi tuottaa enemmän rinnakkain suoritettavia prosesseja kuin tietokoneessa on prosessoreja, yhdelle prosessorille tulee samanaikaisesti suoritettavaksi useita prosesseja, sanokaamme s kpl. Näitä ei voida tietenkään suorittaa samaan aikaan vaan vuorotellen ajan s kuluessa. Näin ollen muistioperaation suorittamiseen on käytettävissä s yksikköä aikaa, eikä välimuistia tarvita. Kutsumme tätä *ylimääräisen rinnakkaisuuden* periaatteeksi (engl. *slackness*) [Valiant 1990].

Oletetaan, että muistimoduulit ovat verkossa, jonka *halkaisija* on ϕ, ts. muistisaannin aikana prosessori ehtii suorittamaan ϕ käskyä. Jotta muistisaannin *viipeen* aikana prosessorin aika ei kuluisi hukkaan, sillä on oltava $s \geq 2\phi$ toisistaan riippumatonta käskyä suoritettavanaan. Ajattelemme, että hakupyyntö on viesti, joka välisolmulta toiselle edetessään käyttää ajan ϕ ja paluumatkallaan data mukanaan toisen kerran ajan ϕ. Kun kaikki s käskyä on suoritettu, ensimmäinen niistä on jo saanut pyytämänsä tiedon muistista ja suoritus voi jatkua. Jotta tämä toimisi sujuvasti, täytyy jokaiselta n prosessorilta 2ϕ viestiä mahtua verkkoon samanaikaisesti. Toisin sanoen verkon *kaistanleveyden* on oltava vähintään luokkaa $2\phi n$. Tämä toteutuu esim. (2- tai 3-ulotteisessa) ristikkoverkossa, jossa prosessorit ovat lävistäjällä ja muut solmut ovat reitittimiä, jotka vain välittävät dataa, kts. kuvaa 13.

Muistimoduulien välisen tietoliikenteen toteuttamiseen tarvitaan vielä *reititysalgoritmi*, joka ohjaa datapaketit kohteisiinsa ruuhkitta. Reitityksessä on erityisesti varottava *lukkiumatilanteita*, joissa paketit estävät toistensa etenemisen. Yleispätevä keino lukkiuman murtamiseen on lisätä paketin lähettämiseen sen verran satunnaisuutta, että lähetystilanne ei toistu askeleesta seuraavaan samanlaisena.

Yhteinen muisti voidaan siis toteuttaa hajautetuilla muistimoduuleilla seuraavien periaatteiden mukaisesti:

- Yhteinen muisti hajautetaan muistimoduuleille satunnaistetulla hajautuksella.
- Muistisaannin viive korvataan käyttämällä hyväksi algoritmin tarjoamaa ylimääräistä rinnakkaisuutta.



Kuva 13: Harva torus, prosessorit lävistäjällä.

- Tietoliikenneverkon kaistanleveyden tulee olla prosessorien lukumäärän ja verkon halkaisijan tulon suuruusluokkaa.
- Reititysalgoritmin tulee välittää paketit maaleihinsa ajassa, joka on verkon lävistäjän suuruusluokkaa.

Näillä periaatteilla yhteisen muistimallin ohjelmia voidaan suorittaa hajautetun muistin koneessa. Suurimpana pullonkaulana tällä hetkellä on nopeiden kytkinten puuttuminen, sillä ilman niitä suuria tiedonsiirtonopeuksia ei pystytä käyttämään hyväksi prosessorin kellotaajuudella tehtävissä muistioperaatioissa. Koska kytkintenkin kehitys on nopeaa, on toiveita, että yleiskäyttöinen, halpa rinnakkaislaskenta tulisi mahdolliseksi muutaman vuoden sisällä. Onkin mielenkiintoista havaita, että supertietokoneiden valmistaja Cray myy perinteisen yhteisen muistin vektoriprosessorin Cray SV2 ja hajautetun muistin koneen Cray T3E lisäksi myös Cray MTA-2:ta (Multi Threaded Architecture), joka soveltaa ylimääräistä rinnakkaisuutta viipeen peittämiseen [Cray].

5 Yhteenveto

Uskomattoman nopeasta ja pitkään jatkuneesta kehityksestä huolimatta peräkkäislaskennan tehon kasvulla on rajansa. Vaikka prosessorien nopeudet ovat kasvaneet, muistien nopeutuminen on ollut hitaampaa. Myös valon nopeus asettaa rajoituksia: 1 GHz kellosyklin aikana valo etenee tyhjiössä vain 30 cm — nopeiden tietokoneiden pitäisi olla pieniä!

Rinnakkaislaskenta tarjoaa taloudellisen tavan hankkia lisätehoa laskentaan. Tällä hetkellä kehitys kulkee hajautettujen tietokoneklusterien suuntaan, jossa laskennan hajautus prosessoreille jää ohjelmoijan vastuulle ja maksimiteho koneesta saadaan vain, jos rinnakkaisuus on "suurirakeista". On kuitenkin lupa toivoa, että ennen pitkää yhteisen muistin mallille suunniteltujen algoritmien tehokas suorittaminen hajautetun muistin koneissa tulee mahdolliseksi. Tämä voi merkitä rinnakkaislaskennan nopeaa läpimurtoa. Vaikka näin ei kävisikään, rinnakkaislaskenta tulee joka tapauksessa laajenemaan, joskin hitaammin ja huomaamattomammin, sillä lisää laskentatehoa saadaan halvimmalla rinnakkaistamalla.

Viitteet

[Almasi et al. 1994] G. S. Almasi, A. Gottlieb: *Highly Parallel Computing* The Benjamin/Cummings Publishing Company, 1994.

[Beowulf] http://www.beowulf.org

[Cray] http://www.cray.com

[Davis 2000] M. Davis. *The Universal Computer.* Norton & Company, 2000.

[Grid] http://www.gridcomputing.org

[InfiniBand] http://www.infinibandta.com

[Jájá 1992] J. Jájá: *An Introduction to Parallel Algorithms.* Addison Wesley 1992.

[Karlin et al. 1988] A. Karlin, E. Upfal. Palallel hashing — an efficient implementation of shared memory. *Journal of the ACM* **35**:876–892 (1988).

[Mak 1997] L. Mak. Parallelism always helps. *SIAM J. Comput.* **26:** 153-172 (1997)

[Penttonen 1998] M. Penttonen. *Johdatus algoritmien suunnitteluun ja analysointiin.* Otatieto 1998.

[Valiant 1990] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, 943-971. Elsevier, 1990.

[van der Steen et al.] A. J. van der Steen, J. J. Dongarra. Overview of recent supercomputers. http://www.top500.org/ORSC/