RINNAKKAISOHJELMOINTI MPI:LLÄ

JUHA HAATAJA JA KAJ MUSTIKKAMÄKI
CSC – TIETEELLINEN LASKENTA OY

Tämän teoksen tekijänoikeudet kuuluvat CSC – Tieteellinen laskenta Oy:lle. Teoksen tai osia siitä voi kopioida ja tulostaa vapaasti henkilökohtaiseen käyttöön sekä Suomen yliopistojen ja korkeakoulujen kurssikäyttöön edellyttäen, että kopioon tai tulosteeseen liitetään tämä ilmoitus teoksen tekijästä ja tekijänoikeuksista. Teosta ei saa myydä tai sisällyttää osaksi muita teoksia ilman CSC:n lupaa.

© Juha Haataja, Kaj Mustikkamäki ja
CSC – Tieteellinen laskenta Oy
2001

http://www.csc.fi/oppaat/mpi/

Esipuhe

Tämä opas on tarkoitettu tiiviiksi johdatukseksi rinnakkaislaskentaan MPI-viestinvälityskirjaston avulla. Oppaan alussa kerrotaan lyhyesti rinnakkaislaskennan perusteista ja rinnakkaisohjelmoinnista yleensä. MPI:n käyttö esitellään lyhyiden esimerkkiohjelmien avulla. Erillisessä luvussa on lisäksi muutamia laajempia esimerkkiohjelmia. Oppaan lopussa on tiivistelmä MPI:stä sekä Fortran- että C-ohjelmoijille.

Teoksen alkuperäinen versio julkaistiin vuonna 1997. Juha Haataja kirjoitti pääosan oppaan tekstistä ja myös toimitti oppaan painokuntoon. Oppaassa on käytetty apuna Sami Saarisen PVM-oppaan [Saa95] esimerkkiohjelmia, joista on tehty MPI-versioita. Lisäksi Jussi Rahola antoi käyttöön MPI:llä rinnakkaistetun liittogradienttikoodin.

Oppaan syntyyn ovat suuresti vaikuttaneet Jussi Raholan ja Hannu Karttusen antamat ehdotukset ja kommentit sekä CSC:n asiakasprojekteista saadut kokemukset. Lisäksi kiitämme Sami Saarista, Juha Fagerholmia ja CSC:n järjestämien MPI-kurssien osanottajia opasta koskevasta palautteesta. Oppaan ensimmäistä versiota oikolukivat lisäksi Jussi Rahola, Juha Fagerholm, Sami Saarinen ja Yrjö Leino. Teoksen toiseen painokseen on tehty pieniä lähinnä CSC:n koneympäristöön, julkaisuihin ja yhteystietoihin liittyviä korjauksia.

Kaikki palaute on tervetullutta, jotta oppaasta tulisi mahdollisimman hyvin käytännön ohjelmointityöhön soveltuva. Kommentteja voi lähettää sähköpostitse osoitteeseen `Juha.Haataja@csc.fi`.

Espoossa 1. elokuuta 2001

Juha Haataja

Sisältö

Esipuhe **3**

1 Rinnakkaislaskennan perusteita **6**
1.1 Oppaan sisältö 6
1.2 Rinnakkaisohjelmointi ja rinnakkaiskoneet 6
1.3 Viestinvälitys ja rinnakkaislaskenta 8
1.4 CSC:n suosituksia 9
1.5 MPI:n ominaisuuksia 9
1.6 MPI-tietoutta CSC:ssä 10
1.7 Suositeltavaa kirjallisuutta 10

2 MPI:n peruskäyttö **11**
2.1 MPI:n tausta ja tavoitteet 11
2.2 Ensimmäinen esimerkki 11
2.3 Viestinnän alustaminen ja lopettaminen 16
2.4 Kahdenvälinen viestintä 17
2.5 MPI-ohjelman ajaminen 22
2.6 MPI:n valmiit tietotyypit 22
2.7 Lisätietoja 23

3 Ryhmäoperaatiot **24**
3.1 Kollektiivinen viestintä 24
3.2 Muita rutiineita 29
3.3 Laskennan tahdistus 36
3.4 Lisätietoja 36

4 MPI:n edistyneitä piirteitä **38**
4.1 MPI:n viestintämuodot 38
4.2 Välittömästi palaavat viestintärutiinit 39
4.3 Viestintäryhmät 41
4.4 Topologiat 43
4.5 Lisätietoja 46

5 Työkaluohjelmia **48**
5.1 Rinnakkaisohjelman ajanmittaus 48

5.2 Muita työkaluja 49
5.3 Lisätietoja 50

6 Esimerkkiohjelmia 51
6.1 Esimerkkien taustaa 51
6.2 Keskiarvon laskeminen 51
6.3 Kuormantasaus 56
6.4 Yhtälöryhmän ratkaisu liittogradienttimenetelmällä . . . 63
6.5 Life-peli 69
6.6 Lisätietoa 76

Liitteet 77

A Tietoa CSC:stä 78

B MPI-tietoutta Internetissä 79

C MPI Quick Reference for Fortran Programmers 80

D MPI Quick Reference for C Programmers 85

E Sanasto 89

Kirjallisuutta 92

Hakemisto 94

1 Rinnakkaislaskennan perusteita

Tässä luvussa kerrotaan MPI-oppaan sisällöstä sekä rinnakkaislaskennan peruskäsitteistä.

1.1 Oppaan sisältö

Annamme tässä oppaassa johdatuksen ohjelmointiin MPI-viestinvälityskirjastolla (Message-Passing Interface) yksinkertaisten esimerkkien avulla sekä esittelemme MPI:n edistyneitä piirteitä.

Tässä luvussa kerromme rinnakkaislaskennan malleista sekä rinnakkaiskoneista. Lisäksi esittelemme *viestinvälitystä* rinnakkaisohjelmoinnin välineenä. Luvun lopussa annamme vinkkejä lisätietojen hankkimiseen MPI:stä ja rinnakkaislaskennasta.

Oppaan luvussa 2 kerromme MPI:n peruskäytöstä. Luku on varsin itsenäinen kokonaisuus ja sitä voi pitää MPI:n perusteiden tiiviinä katsauksena.

Luvussa 3 kerromme ryhmäviestinnästä ja viestintäryhmien käytöstä. Luvussa 4 tuomme esiin muutamia MPI:n kehittyneitä piirteitä ja luvussa 5 esittelemme apuvälineitä MPI-ohjelmointiin. Esimerkiksi ajanmittaus ja viestinnän visualisointi auttavat usein nopeuttamaan koodia ja löytämään siihen mahdollisesti pujahtaneita virheitä.

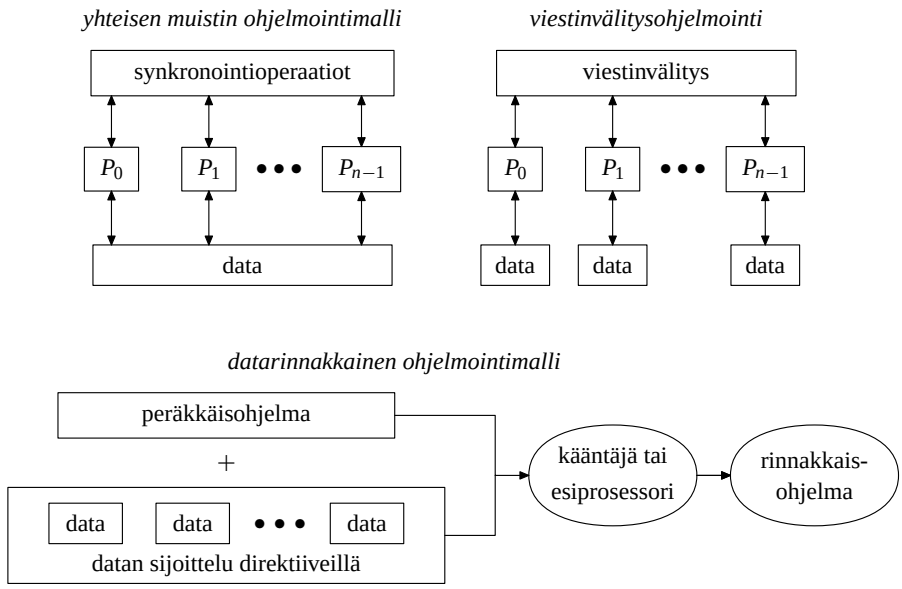
Oppaan lopussa on tietoa lisätiedon lähteistä sekä viitteitä Internetistä löytyviin MPI-dokumentteihin. Lisäksi liitteistä C ja D (sivut 80 ja 85) löytyy *MPI Quick Reference* -sivut sekä Fortran- että C-ohjelmoijille.

1.2 Rinnakkaisohjelmointi ja rinnakkaiskoneet

Käytämme tässä oppaassa seuraavia nimityksiä eräistä rinnakkaisohjelmoinnin käsitteistä:

- *keskusyksikkö* (CPU) on itsenäiseen toimintaan kykenevä prosessorista, muistista ja liitäntäväylistä koostuva laitteistokokonaisuus
- *prosessi* (process) on tietokoneen keskusyksikössä ajettava ohjelma, jolla on oma muistiavaruus
- *laskentatehtävä, tehtävä* (task) osallistuu rinnakkaislaskentaan viestimällä muiden laskentatehtävien kanssa ja suorittamalla oman osansa rinnakkaisalgoritmista
- *viestinvälitys* (message passing) mahdollistaa tiedon vaihtamisen laskentatehtävien välillä sekä laskennan eri vaiheiden synkronisoinnin.

Käytännössä kukin MPI:n laskentatehtävä on erillinen prosessi. Todellisissa rinnakkaiskoneissa pyritään laskentatehtävät lisäksi ajamaan siten, että kukin tehtävä saa käytettäväkseen keskusyksikön kaikki resurssit (muisti, levy, laskentateho).



Kuva 1.1: *Rinnakkaislaskennassa käytettyjä ohjelmointimalleja.*

Rinnakkaistietokoneen ohjelmointimallit voidaan jakaa seuraaviin kolmeen ryhmään (katso kuvaa 1.1):

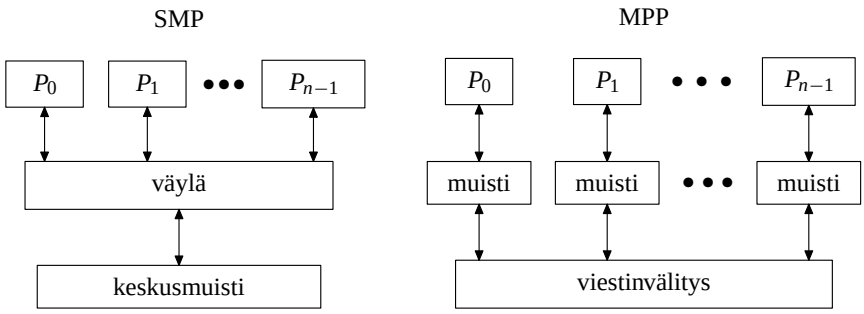
- yhteisen keskusmuistin ohjelmointimalli (shared memory, suomennettu myös ”jaettu muisti”)
- viestinvälitykseen perustuva rinnakkaisohjelmointi
- datarinnakkainen (data-parallel) ohjelmointikieli.

MPI lukeutuu viestinvälitykseen perustuviin ohjelmointimalleihin. Edellisten lisäksi voidaan ohjelmointimalliksi katsoa myös ns. *data-passing*, jossa siirretään dataa suoraan prosessorien muistien välillä.

Rinnakkaistietokoneita voi myös katsoa olevan kahta eri tyyppiä (katso kuvaa 1.2):

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

Rinnakkaiskoneen tyypin ei välttämättä tarvitse vastata käytettyä ohjelmointimallia. Esimerkiksi SMP-koneella voidaan käyttää viestinvälitysohjelmointia. Toisaalta hajautetun keskusmuistin MPP-koneilla saattaa olla mahdollista käyttää yhteisen keskusmuistin ohjelmointimallia.



Kuva 1.2: *Rinnakkaislaskennassa käytettyjä arkkitehtuureita.*

1.3 Viestinvälitys ja rinnakkaislaskenta

Viestinvälityksen avulla voidaan lähettää ja vastaanottaa dataa rinnakkaisten laskentatehtävien välillä. Viestit lähetetään yleensä tiedonsiirtoverkon tai -väylän kautta, vaikkakin yhteisen keskusmuistin koneessa voidaan käyttää lokaaleja muistiviittauksia. Ohjelmoijalle viestinvälitys näkyy käytännössä aliohjelmakutsuina, joissa on argumentteina lähetettävän ja vastaanotettavan datan sisältäviä muuttujia.

MPI:n kaltaiset viestinvälityskirjastot ovat rinnakkaisohjelmoinnin perustyökaluja, joilla voi kirjoittaa siirrettäviä ja tehokkaita ohjelmia rinnakkaistietokoneille. Vaikka viestinvälitys on melko matalan tason rinnakkaistamista, se ei ohjelmoijan kannalta eroa paljon normaalin aliohjelmakirjaston käytöstä. Lisäksi viestinvälitys on ainakin vielä lähes ainoa tapa kirjoittaa tehokasta ja skaalautuvaa rinnakkaiskoodia useammalle kuin kymmenelle prosessorille.

Viestinvälityskirjastoja tarvitaan vielä pitkän aikaa rinnakkaisten ohjelmointikielten ohella. Esimerkiksi HPF (High Performance Fortran, rinnakkainen Fortran-laajennus) ei yleensä ole tarpeeksi tehokas ohjelmointityökalu.

1.4 CSC:n suosituksia

Viestinvälityskirjastot kehittyvät niin nopeasti, että käyttäjän on usein vaikeaa valita oikeaa kirjastoa ja pysyä mukana sen kehityksessä. Kaksi tärkeintä viestinvälityskirjastoa ovat MPI (Message-Passing Interface) ja PVM (Parallel Virtual Machine).

CSC suosittelee uusille käyttäjille MPI-viestinvälityskirjaston käyttöä. Tästä huolimatta PVM-kirjasto tulee säilymään CSC:n rinnakkaistietokoneilla, ja sitä kannattaa käyttää, jos aikaisemmat ohjelmat on kirjoitettu PVM:llä tai jos ohjelmia tarvitsee ajaa muissa PVM-ympäristöissä.

MPI:n kenties tärkein etu PVM:ään nähden on se, että MPI on yliopistojen, tutkimuslaitosten ja tietokonevalmistajien yhdessä sopima standardi. Eri laitevalmistajat, esimerkkinä SGI ja IBM, ovat sitoutuneet toteuttamaan MPI-kirjaston omilla rinnakkaislaitteistoillaan. Standardoinnin takia MPI:n versiot kehittyvät paljon hallitummin kuin PVM:n versiot.

Tekniseltä toteutukseltaan MPI sisältää monia etuja PVM:ään nähden. Ohjelmakoodista tulee selkeämpää ja tiiviimpää, koska MPI:llä voi ohjelmoida korkeammalla abstraktiotasolla kuin PVM:llä. Lisäksi MPI:n *viestintäryhmät* (communicator) mahdollistavat sovelluksen oman viestinnän eristämisen mahdollisesti käytettävien aliohjelmakirjastojen viestinnästä.

PVM:n etu MPI:hin nähden on lähinnä se, että PVM toimii myös heterogeenisessa ympäristössä, jossa on monentyyppisiä koneita. Lisäksi nykyisessä MPI:n versiossa ei vielä voi dynaamisesti luoda uusia laskentatehtäviä.

1.5 MPI:n ominaisuuksia

MPI-järjestelmä sisältää suuren joukon monipuolisia rutiineita viestinvälitykseen. MPI:n vahvoja alueita ovat mm.

- monipuolinen kahdenvälinen viestintä
- paljon kollektiivisia operaatioita
- ryhmäviestintä on joustavaa ja helppoa ohjelmoida
- laskentatopologioita voi luoda tarpeen mukaan
- omia muuttujatyyppejä voi määritellä ja välittää.

MPI:n mahdollisuuksiin perehtyy parhaiten esimerkkien ja omien kokeilujen kautta. Tässä oppaassa esitettyjen rutiinien avulla voi jo toteuttaa melko haastaviakin rinnakkaistustehtäviä.

1.6 MPI-tietoutta CSC:ssä

Tämän oppaan esimerkkiohjelmien lähdekoodeja löytyy CSC:n www-palvelimesta osoitteessa

`http://www.csc.fi/oppaat/mpi/`

CSC:n julkaisemassa @CSC-lehdessä kerrotaan mm. CSC:n rinnakkaislaskentaympäristön käytöstä. Ohjelmankehitystä käsitteleviä lehdessä julkaistuja artikkeleita on koottu PDF-tiedostomuodossa osoitteeseen

`http://www.csc.fi/oppaat/ohjelmointi/`

Tietoa rinnakkaislaskennasta löytyy www-osoitteesta

`http://www.csc.fi/programming/`

Tästä www-osoitteesta löytyy PostScript-muodossa mm. MPI-standardi (231 sivua) ja MPI Quick Reference (Fortran ja C, kumpikin 2 sivua). Lisäksi tarjolla on CSC:ssä tehtyjä esimerkkejä MPI:n käytöstä.

Liitteessä B sivulla 79 on lueteltu lisää MPI-tietoutta sisältäviä www-osoitteita.

1.7 Suositeltavaa kirjallisuutta

MPI:stä on saatavilla jo monta eritasoista esitystä. Jokaisella MPI:n käyttäjällä tulisi olla ainakin standardi *MPI: A Message-Passing Interface Standard* [For95]. Myös teosta *MPI: The Complete Reference* [SOHL+96] voi käyttää käsikirjan tapaan.

Teos *Using MPI: Portable Parallel Programming with the Message-Passing Interface* [GLS94] on käyttökelpoinen johdatus MPI-ohjelmointiin. Vastaava teos on *Parallel Programming with MPI* [Pac97]. Rinnakkaislaskennan peruskäsitteitä on esitelty mm. teoksessa *Designing and Building Parallel Programs* [Fos95]. Hyvä rinnakkaislaskennan oppikirja on *Introduction to Parallel Computing — Design and Analysis of Algorithms* [KGGK94].

CSC:n asiakkaan opas [FHK00] kertoo perusasiat CSC:n asiakkaaksi tulemisesta ja CSC:n tarjoamista palveluista. CSC:n koneympäristön käyttöoppaaksi sopii teos *Metakoneen käyttöopas* [KR00b]. Rinnakkaiskoneista Cray T3E ja IBM SP on julkaistu englanninkieliset käyttöoppaat [HS01, HKR01].

CSC on myös julkaissut Fortran 90 ja Fortran 95 -kielten oppikirjan [HRR98] sekä numeeristen menetelmien oppaita [HHL+99, HKR93, Haa95].

2 MPI:n peruskäyttö

Tässä luvussa esitellään MPI:n perusteita ja annetaan yksinkertaisia käyttöesimerkkejä.

2.1 MPI:n tausta ja tavoitteet

MPI on viestinvälityskirjasto, josta on saatavilla sekä vapaasti levitettäviä että koneenvalmistajien omia tehokkaita versioita. MPI:llä kirjoitetut ohjelmat ovat siirrettävissä toiseen MPI-toteutukseen standardoinnin ansiosta. MPI on myös varsin helppokäyttöinen ja joustava rinnakkaisohjelmoinnin väline.

MPI-rutiinit on suunnitelty käytettäviksi Fortran 77 ja ANSI C -ohjelmista käsin. Mikään ei tosin estä käyttämästä Fortran 90/95 tai C++-kieliä, vaikka standardoitua tukea näiden kielten edistyksellisille piirteille (modulaarisuus, ylikuormitus jne.) ei MPI:ssä vielä olekaan.

MPI soveltuu siirrettävien sovellusohjelmistojen ja aliohjelmakirjastojen tekemiseen, joten MPI:tä voi pitää eräänlaisena ohjelmistoteollisuuden perustyövälineenä. MPI:llä voi ohjelmoida sekä hajautetun muistin rinnakkaiskoneita (esimerkkinä IBM SP ja Cray T3E) että yhteisen muistin moniprosessorikoneita (SGI, Compaq ym.).

MPI:tä nykyisellään ei toki voi pitää täydellisenä ratkaisuna rinnakkaisohjelmointiin — kyseessähän on viestinvälitykseen perustuva väline, jonka käytössä on paljon jätetty ohjelmoijan vastuulle. Mikään automaattinen rinnakkaistusväline ei MPI siis ole. Säännöllisissä tehtävissä (esimerkiksi differenssimenetelmään perustuvat koodit) voi datarinnakkaisen ohjelmointimallin käyttö olla MPI:n sijasta paras tie rinnakkaislaskentaan. Lisäksi MPI:n nykyinen versio ei ole standardoinut prosessien käsittelyä tai rinnakkaista I/O:ta.

2.2 Ensimmäinen esimerkki

Vaikka MPI on kokonaisuudessaan hyvin laaja (yli 120 rutiinia), selviää monissa rinnakkaislaskennan tehtävissä muutamalla rutiinilla — vähimmillään riittävät seuraavassa esiteltävät kuusi MPI-rutiinia.

Sama MPI-ohjelma suoritetaan kaikissa laskentatehtävissä, mutta eri tehtävät voivat laskea eri asioita esimerkiksi oman järjestysnumeronsa perusteella. Tässä luvussa käsiteltävä esimerkkiohjelma toimii seuraavasti:

1. MPI-laskennan aloittaminen
2. viestien lähetys laskentatehtävien välillä
3. MPI-laskennan lopettaminen.

Kaiken kaikkiaan esimerkkiohjelmassa käytetään kuutta MPI-rutiinia:

- MPI_INIT: rinnakkaislaskennan aloitus
- MPI_COMM_SIZE: selvitetään kuinka monta laskentatehtävää on käytettävissä
- MPI_COMM_RANK: selvitetään oma järjestysnumero
- MPI_SEND: lähetetään viesti toiselle laskentatehtävälle
- MPI_RECV: vastaanotetaan viesti toiselta laskentatehtävältä
- MPI_FINALIZE: lopetetaan rinnakkaislaskenta.

Monet rinnakkaisalgoritmit voidaan toteuttaa pelkästään näillä kuudella rutiinilla!

Listauksessa 2.1 on esitetty Fortran 90 -kielinen versio esimerkkiohjelmasta. Saman ohjelman C-kielinen versio löytyy listauksesta 2.2 sivulta 15. Fortran- ja C-kielisissä ohjelmissa on pieniä eroja MPI-rutiinien kutsusyntaksin suhteen. MPI-rutiinien kutsuja käsitellään tarkemmin kappaleessa 2.3.

2.2.1 Fortran-kielinen ohjelma

Listauksen 2.1 Fortran-koodissa otetaan heti alussa käyttöön joukko MPI:n määrittelyjä komennolla

```
INCLUDE 'mpif.h'
```

Tämän jälkeen määritellään laskentatehtävissä tarvittavat muuttujat ja vakiot. Fortranissa MPI:n tietotyyppeihin viitataan `INTEGER`-tyyppisillä muuttujilla; C-kielessä on asia hoidettu korkeamman tason olioilla (osoittimet ja tietuerakenteet).

Fortran 90:ssä voi edellä annetun `INCLUDE`-komennon sijoittaa moduuliin:

```
      MODULE mpi
        IMPLICIT NONE
! Otetaan käyttöön MPI:n määrittelyt:
        INCLUDE 'mpif.h'
      END MODULE mpi
```

Listaus 2.1: *Yksinkertainen MPI:n Fortran-kielinen käyttöesimerkki.*

```
PROGRAM esimerkki
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  INTEGER, PARAMETER :: tag = 50
  INTEGER :: id, ntasks, source_id, &
      dest_id, rc, i
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
  INTEGER, DIMENSION(2) :: msg
  CALL MPI_INIT(rc)
  IF (rc /= MPI_SUCCESS) THEN
    WRITE(*,*) 'MPI initialization failed'
    STOP
  END IF
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
  IF (id /= 0) THEN
    msg(1) = id
    msg(2) = ntasks
    dest_id = 0
    CALL MPI_SEND(msg, 2, MPI_INTEGER, dest_id, &
        tag, MPI_COMM_WORLD, rc)
  ELSE
    DO i = 1, ntasks-1
      CALL MPI_RECV(msg, 2, MPI_INTEGER, &
          MPI_ANY_SOURCE, tag, &
          MPI_COMM_WORLD, status, rc)
      source_id = status(MPI_SOURCE)
      WRITE(*,*) 'message:', msg, 'sender:', source_id
    END DO
  END IF
  CALL MPI_FINALIZE(rc)
END PROGRAM esimerkki
```

Tämän moduulin voi ottaa käyttöön lauseella `USE mpi`. Moduulin `mpi` määrittelyssä käytettiin Fortran 77:stä peräisin olevaa sarakesidonnaista lähdekoodin muotoa, jossa suoritettavat lauseet alkavat vasta 7. sarakkeesta.

Esimerkkiohjelman ensimmäinen suoritettava lause on

```
CALL MPI_INIT(rc)
```

Tällä kutsulla käynnistetään rinnakkaislaskenta. Kokonaislukumuuttujaan `rc` saadaan rutiinin `MPI_INIT` paluukoodi, jonka arvo on vakio `MPI_SUCCESS`, jos MPI:n käynnistäminen onnistuu.

Nimetty vakio `MPI_SUCCESS` on määritelty tiedostossa `mpif.h`, eikä sitä tarvitse (eikä saa!) määritellä ohjelmakoodissa. Onneksi MPI:ssä käytettyjen tunnusten uudelleen määrittely omassa ohjelmassa johtaa useimmiten käännösaikaiseen virheilmoitukseen.

Aliohjelmakutsu `MPI_COMM_SIZE` palauttaa muuttujaan `ntasks` käytettävissä olevien laskentatehtävien lukumäärän. Tehtävän järjestysnumero saadaan

selville aliohjelmakutsulla MPI_COMM_RANK. Järjestysnumero `id` on kokonaisluku välillä $0, \ldots, \texttt{ntasks} - 1$.

Laskennan alustamisen jälkeen voidaan haarautua tekemään eri asioita eri laskentatehtävissä. Jos oma järjestysnumero ei ole nolla, lähetetään esimerkissä viesti laskentatehtävälle nolla rutiinilla MPI_SEND seuraavasti:

```
CALL MPI_SEND(msg, 2, MPI_INTEGER, dest_id, &
    tag, MPI_COMM_WORLD, rc)
```

Muuttuja `msg` sisältää kahdesta alkiosta koostuvan kokonaislukutaulukon. Argumentissa `tag` on viestin tunnistenumero, jonka ohjelmoija voi vapaasti valita.

Jos oma järjestysnumero on nolla, vastaanotetaan esimerkissä viestejä muilta laskentatehtäviltä rutiinilla MPI_RECV:

```
CALL MPI_RECV(msg, 2, MPI_INTEGER, MPI_ANY_SOURCE, tag, &
    MPI_COMM_WORLD, status, rc)
```

Viestin lähettäjän järjestysnumero talletetaan muuttujaan `source_id`. Tämän jälkeen tulostetaan vastaanotettu viesti ja lähettäjän järjestysnumero.

Kun viestit on lähetetty ja vastaanotettu, voidaan ohjelman lopussa lopettaa rinnakkaislaskenta komennolla MPI_FINALIZE.

Tätä esimerkkiä käsitellään tarkemmin tämän luvun seuraavissa kappaleissa.

2.2.2 C-kielinen ohjelma

Listauksen 2.2 esimerkkiohjelma tekee C-kielellä samat toimenpiteet kuin edellä esitelty Fortran-ohjelmakin.

MPI-ohjelmatiedoston alussa tulee olla komento `#include <mpi.h>`, jonka avulla saadaan käyttöön MPI:n määrittelemiä datatyyppejä ja vakioita.

Ohjelman alussa määritellään laskennassa tarvittavat muuttujat ja vakiot. Erona aikaisempaan Fortran-versioon on rakenteisen muuttujatyypin MPI_Status käyttö `status`-muuttujan määrittelyssä.

C-kielessä alustetaan rinnakkaislaskenta aliohjelmakutsulla MPI_Init. Rutiinin nimen alku kirjoitetaan siis isoilla kirjaimilla ja loppuosa pienillä — Fortranissahan käytetään kokonaan isoja kirjaimia. C-kielessä voidaan Fortranista poiketen välittää ohjelman komentoriviargumentit rutiinille MPI_Init. Rutiinin arvonaan palauttama paluukoodi `rc` kertoo, onnistuiko rinnakkaislaskennan käynnistäminen.

Rutiinilla MPI_Comm_size saadaan selville laskentatehtävien lukumäärä muuttujaan `ntasks` ja rutiinilla MPI_Comm_rank kunkin tehtävän järjestysnumero muuttujaan `id`.

Laskennan alustamisen jälkeen haaraudutaan `if`-lauseessa sen mukaan, onko oma järjestysnumero nolla vai ei. Mikäli järjestysnumero `id` ei ole nolla,

Listaus 2.2: *Yksinkertainen MPI:n C-kielinen käyttöesimerkki.*

```
#include <stdio.h>
#include <mpi.h>
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 initialization failed\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("message: %d %d sender: %d\n",
             msg[0], msg[1], source_id);
    }
  }
  rc = MPI_Finalize();
  exit(0);
}
```

lähetetään rutiinilla MPI_Send kahdesta kokonaisluvusta koostuva viesti `msg` laskentatehtävälle nolla. Vakiossa `tag` on viestin tunnistenumero, jonka ohjelmoija voi vapaasti valita.

Mikäli järjestysnumero on nolla, otetaan muiden laskentatehtävien lähettämiä viestejä vastaan rutiinilla MPI_Recv. Viestin lähettäjä saadaan selville `status`-muuttujan kentästä MPI_SOURCE.

Viestien lähettämisen ja vastaanottamisen jälkeen lopetetaan rinnakkaislaskenta kutsumalla rutiinia MPI_Finalize.

Tätä esimerkkiä käsitellään lisää seuraavissa kappaleissa.

Taulukko 2.1: *MPI:n aliohjelmakutsut viestinnän alustamiseen ja lopettamiseen sekä oman järjestysnumeron ja laskentatehtävien määrän selvittämiseen. Lisäksi on esitetty MPI:n aliohjelmakutsut kahdenväliseen viestintään.*

C-kieli	*Fortran*
int **MPI_Init**(int *argc, char ***argv)	**MPI_INIT**(ierror) integer ierror
int **MPI_Comm_size**(MPI_Comm comm, int *size)	**MPI_COMM_SIZE**(comm, size, ierror) integer comm, size, ierror
int **MPI_Comm_rank**(MPI_Comm comm, int *rank)	**MPI_COMM_RANK**(comm, rank, ierror) integer comm, rank, ierror
int **MPI_Finalize**(void)	**MPI_FINALIZE**(ierror) integer ierror
int **MPI_Send**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	**MPI_SEND**(buf, count, datatype, dest, tag, comm, ierror) *type* buf(*) integer count, datatype, dest, tag, comm, ierror
int **MPI_Recv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	**MPI_RECV**(buf, count, datatype, source, tag, comm, status, ierror) *type* buf(*) integer count, datatype, source, tag, comm, status(MPI_STATUS_SIZE), ierror

2.3 Viestinnän alustaminen ja lopettaminen

Käytämme tässä oppaassa pääosin Fortran 90 ja ANSI C -kielten syntaksia MPI:n havainnollistamiseen. Rutiinien kutsusyntaksi esitetään samaan tapaan kuin taulukossa 2.1, jossa on esitetty MPI-standardissa määritellyt rutiinit lihavoiduilla kirjaimilla. Liitteissä C ja D (sivut 80 ja 85) on samaa esitystapaa käyttävä kattavampi MPI:n rutiinien luettelo.

Rinnakkaislaskenta käynnistetään MPI:n rutiinilla MPI_INIT. Seuraavassa on Fortran- ja C-kieliset esimerkit käynnistyskomennosta:

```
CALL MPI_INIT(rc)                ← Fortran
rc = MPI_Init(&argc, &argv);     ← C-kieli
```

Siis Fortranissa käytetään isoja kirjaimia MPI-rutiinien nimissä; C:ssä nimet alkavat isoilla kirjaimilla ja loppuosa on pienillä kirjaimilla. C-kielessä voidaan MPI-järjestelmälle välittää ohjelman komentoriviargumentit kutsulla MPI_Init.

Fortran-rutiinien kutsusyntaksi eroaa C-kielestä siinä, että viimeisenä argu-

menttina on rutiinin paluukoodin välittävä kokonaislukumuuttuja. Tästä muuttujasta käytetään nimeä `ierror` taulukon 2.1 esitystavassa. C-kielessä rutiinit palauttavat paluukoodin arvonaan. Onnistuneen MPI-kutsun paluukoodin arvo on vakio MPI_SUCCESS.

Useimmissa MPI-rutiineissa käytetään *viestintäryhmäksi* (communicator) kutsuttua tietorakennetta. Olemassaolevaan viestintäryhmään viitataan vakiolla tai muuttujalla, joka on *kahva* (handle) MPI:n sisäiseen tietorakenteeseen. Tämän luvun esimerkeissä riittää ymmärtää, että valmiiksi määritelty viestintäryhmä MPI_COMM_WORLD sisältää kaikki rinnakkaislaskentaan osallistuvat laskentatehtävät. Taulukossa 2.1 viitataan viestintäryhmään muuttujalla `comm`.

Viestintäryhmän koon saa selville rutiinilla MPI_COMM_SIZE. Rutiinia voidaan kutsua Fortranista esimerkiksi seuraavasti:

```
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
```

Tämän kutsun tuloksena saadaan laskentatehtävien lukumäärä muuttujaan `ntasks`. C-kielessä täytyy rutiinille välittää kokonaislukumuuttujan `ntasks` osoite:

```
rc = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
```

Oma järjestysnumero `id` väliltä $0, \ldots,$ `ntasks` -1 saadaan selville Fortran- ja C-kielissä seuraavilla aliohjelmakutsuilla:

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)   ← Fortran
rc = MPI_Comm_rank(MPI_COMM_WORLD, &id);     ← C-kieli
```

Laskennan lopussa tarvitaan vielä rutiinia MPI_FINALIZE, jonka käyttö tapahtuu Fortran- ja C-kielissä seuraavasti:

```
CALL MPI_FINALIZE(rc)    ← Fortran
rc = MPI_Finalize();     ← C-kieli
```

Tämän jälkeen MPI-järjestelmää ei voi enää käynnistää uudelleen.

2.4 Kahdenvälinen viestintä

Listauksissa 2.1 ja 2.2 esitetyssä esimerkkiohjelmassa lähetetään kahdesta kokonaisluvusta koostuvia viestejä tehtävälle nolla, jos oma järjestysnumero ei ole nolla. Jos oma järjestysnumero on nolla, otetaan vastaan viestejä ja tulostetaan ne.

Taulukossa 2.1 on esitetty kahdenväliseen viestintään käytettävien rutiinien MPI_SEND ja MPI_RECV kutsusyntaksit C- ja Fortran-kielillä. Tässä teoksessa käytetään taulukossa 2.2 lueteltuja merkintöjä.

Tätä muuttujien nimeämiskäytäntöä sovelletetaan myöhemmin muiden rutiinien yhteydessä. Kannattaa huomata, että C-kielessä lähetettävä tai vastaanotettava data välitetään aina osoitinmuuttujan avulla (taulukkohan on C-kielessä oikeastaan osoitin). Fortranissa vastaava argumentti voi olla joko

Taulukko 2.2: *Teoksessa käytettyjä merkintöjä.*

Muuttuja	*Selitys*
`buf`	Lähetettävä tai vastaanotettava data
`count`	Data-alkioiden lukumäärä, viestin pituus
`datatype`	Data-alkioiden tyyppi
`dest`	Viestin vastaanottajan järjestysnumero
`source`	Viestin lähettäjän järjestysnumero
`tag`	Viestin tunnistenumero (ohjelmoijan valittavissa)
`comm`	Viestintäryhmä
`status`	Viestinvälitykseen liittyviä tietoja

skalaari- tai taulukkomuuttuja, sillä Fortran välittää aliohjelmien argumentit aina osoitteiden avulla.

Argumentiksi `datatype` kelpaavat datatyyppien nimet on esitelty kappaleessa 2.6. MPI:n käyttäjän ei tarvitse huolehtia siitä, kuinka monta tavua tietyn tyyppinen muuttuja vie, vaan ohjelmassa voi käyttää suoraan Fortranin tai C-kielen omia muuttujatyyppejä. Esimerkissämme käytetään kokonaislukujen välittämiseen datatyyppejä MPI_INTEGER (Fortran) ja MPI_INT (C-kieli).

MPI:n kahdenvälisiä viestintärutiineita voi verrata kirjeen lähettämiseen. Kirjeen sisältöä vastaa muuttuja `buf` ja kirjeen kokoa muuttuja `count`. Muuttujassa `datatype` on kirjeen tyyppi (postikortti, kirje, paketti jne.). Muuttujissa `dest` ja `source` kerrotaan vastaanottaja ja lähettäjä — voimme siis odottaa viestiä tietyltä lähettäjältä. Muuttujassa `tag` on viestin tunniste, jota voi verrata kirjeeseen leimatuksi kirjeensulkijamerkiksi. Muuttuja `comm` vastaa kirjatun kirjeen lähettämistä — vain samaan viestintäryhmään kuuluva voi ottaa viestin vastaan.

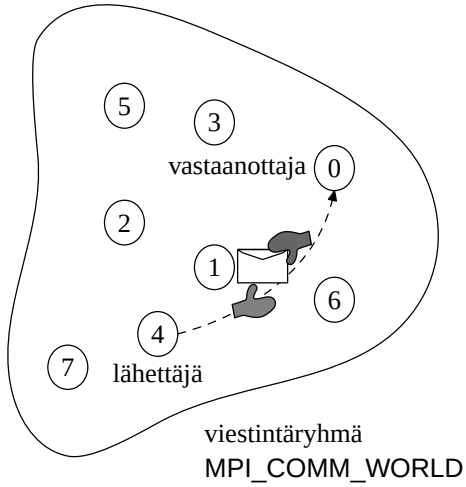
2.4.1 Viestintärutiinien ominaisuuksia

MPI:n normaalit kahdenväliset viestintärutiinit MPI_SEND ja MPI_RECV pysähtyvät odottamaan viestinnän valmistumista eli rutiinit ovat *pysähtyviä* (”blokkaavia”, blocking). Täten aliohjelmakutsusta MPI_SEND palataan kutsuvaan ohjelmaan vasta kun lähetyspuskuri voidaan käyttää uudestaan. Vastaavasti rutiinista MPI_RECV palataan vasta kun tuleva data on kopioitu vastaanottopuskuriin.

MPI:n kahdenvälinen viestintä takaa viestinvälityksen luotettavuuden, joten tarkistuksia viestien perillemenosta ei yleensä tarvitse tehdä. Lisäksi MPI-standardi takaa, että laskentatehtävän lähettämät viestit saapuvat vastaanottajalle aikajärjestyksessä. Siis jos laskentatehtävä P_1 lähettää tehtävälle P_2 ensin viestin m_1 ja tämän jälkeen toisen viestin m_2, saapuu viesti m_1 perille ensin.

Kannattaa kuitenkin huomata, että jos ensimmäinen viesti kulkee reittiä $P_1 \rightarrow P_2$ ja jälkimmäinen lähetetään ensin laskentatehtävälle P_3, joka lähettää sen edelleen P_2:lle (siis $P_1 \rightarrow P_3 \rightarrow P_2$), voi jälkimmäinen viesti olla perillä ennen ensimmäistä! MPI nimittäin takaa viestien järjestyksen säilymisen vain kahden laskentatehtävän välillä.

Kuva 2.1 havainnollistaa kahdenvälistä viestintää. MPI:ssa valmiiksi määritelty viestintäryhmä MPI_COMM_WORLD sisältää kaikki laskentatehtävät, joita on kuvassa kahdeksan kappaletta. Kuvassa tehtävä neljä lähettää viestin tehtävälle nolla.



Kuva 2.1: *Viestinvälitys kahdeksan laskentatehtävän tapauksessa.*

Viestintäryhmään viitataan C-kielessä tyyppiä MPI_Comm olevalla muuttujalla. Fortranissa viittaus hoidetaan INTEGER-tyyppisellä muuttujalla. Nämä muuttujat ovat *kahvoja* (handle), joiden avulla viitataan MPI-järjestelmän sisäisiin olioihin. Siten käytettäessä esimerkiksi valmiiksi määriteltyä viestintäryhmää MPI_COMM_WORLD viitataan itse asiassa MPI-järjestelmän sisäiseen tietorakenteeseen. Vaikka Fortranissa vakion MPI_COMM_WORLD arvo on kokonaisluku, tämä kokonaisluku on vain viittaus erääseen MPI:n sisäiseen tietorakenteeseen.

2.4.2 Kahdenvälinen viestintä Fortran-kielellä

Aliohjelmakutsu MPI_SEND lähettää viestin laskentatehtävältä toiselle. Rutiinin Fortran-kutsusyntaksi on seuraava:

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
type buf(*)
integer count, datatype, dest, tag, comm, ierror
```

Seuraavassa on esimerkki viestin lähettämisestä Fortran 90 -kielellä:

```
INTEGER, PARAMETER :: tag = 50
   ...
msg(1) = id
msg(2) = ntasks
dest_id = 0
CALL MPI_SEND(msg, 2, MPI_INTEGER, dest_id, tag, &
      MPI_COMM_WORLD, rc)
```

Rutiinin MPI_SEND ensimmäinen argumentti on esimerkissämme kahdesta alkiosta koostuva kokonaislukutaulukko `msg`. Toinen argumentti (luku 2) kertoo välitettävien alkioiden lukumäärän ja kolmas tyypin (tässä tyyppinä on MPI_INTEGER).

Neljäs argumentti `dest_id` kertoo viestin vastaanottajan järjestysnumeron ja viides argumentti `tag` viestiin liittyvän tunnistenumeron. Tämän tunnistenumeron voi ohjelmoija vapaasti valita. Tunniste pitää huolta siitä että saman tyyppiset mutta eri asioita tarkoittavat viestit eivät sekoitu keskenään.

Kuudes argumentti MPI_COMM_WORLD viittaa viestintäryhmään ja kertoo, mitkä laskentatehtävät ovat mahdollisia viestin vastaanottajia. Kaikki laskentatehtävät kuuluvat viestintäryhmään MPI_COMM_WORLD. Muuttujassa `dest_id` annettu järjestysnumero viittaa tähän viestintäryhmään. Muuttujaan `rc` saadaan rutiinin MPI_SEND paluukoodi, joka on MPI_SUCCESS, mikäli lähettäminen onnistui.

Vastaanottorutiini MPI_RECV toimii samaan tapaan kuin viestien lähetyskomento MPI_SEND. Rutiinin kutsumuoto näyttää seuraavalta:

MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
type buf(*)
integer count, datatype, source, tag, comm,
  status(MPI_STATUS_SIZE), ierror

Edellä rutiinilla MPI_SEND lähetettyjä viestejä voi vastaanottaa seuraavasti:

```
CALL MPI_RECV(msg, 2, MPI_INTEGER, MPI_ANY_SOURCE, tag, &
    MPI_COMM_WORLD, status, rc)
source_id = status(MPI_SOURCE)
```

Kolme ensimmäista argumenttia sisältävät puskurimuuttujan `msg` vastaanotettavalle datalle sekä alkioiden lukumäärän (luku 2) ja vastaanotettavan tiedon tyypin (MPI_INTEGER).

Argumentin `source` arvo MPI_ANY_SOURCE kertoo, että viestejä otetaan vastaan miltä tahansa lähettävältä laskentatehtävältä, joka kuuluu käytettyyn viestintäryhmään MPI_COMM_WORLD. Argumenttina voi tietenkin olla myös jonkin laskentatehtävän järjestysnumero, jolloin odotetaan viestiä tältä tehtävältä.

Argumentissa `tag` määrätään vastaanotettavan datan tunnistenumero. Arvona voi olla myös nimetty vakio MPI_ANY_TAG, jolloin viestin tunnisteeksi käy mikä tahansa arvo. Muuten tunnistenumeron tulee olla sama kuin siinä viestissä, jonka haluamme ottaa vastaan.

Argumentti `status` on taulukko, jonka alkioista saadaan tietoa lähetetystä viestistä. Esimerkiksi viestin lähettäjän järjestysnumero saadaan selville MPI_RECV-kutsun jälkeen taulukon `status` alkiosta MPI_SOURCE. Argumentissa `rc` saadaan puolestaan rutiinin MPI_RECV paluukoodi.

2.4.3 Kahdenvälinen viestintä C-kielellä

C-kielisen rutiinin MPI_Send kutsusyntaksi on seuraava:

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

Viestien lähettämisen voi tehdä siis esimerkiksi seuraavasti:

```
const int tag = 50;
   ...
msg[0] = id;
msg[1] = ntasks;
dest_id = 0;
rc = MPI_Send(msg, 2, MPI_INT, dest_id, tag,
              MPI_COMM_WORLD);
```

Rutiinin MPI_Send ensimmäinen argumentti sisältää lähetettävän datan. Tässä lähetetään kaksi kokonaislukua taulukossa `msg`. Jos lähetettävänä olisi skalaari, pitäisi argumentiksi antaa skalaarimuuttujan osoite.

Rutiinin MPI_Send toinen argumentti sisältää lähetettävän datan määrän (tässä kaksi kokonaislukua) ja kolmas argumentti tyypin MPI_INT. Neljännessä argumentissa `dest_id` on vastaanottajan järjestysnumero, joka on esimerkissämme nolla. Viides argumentti on tunnistenumero, joka on ohjelmoijan vapaasti valittavissa. Tunnisteen avulla voidaan samantyyppiset mutta eri asioita tarkoittavat viestit erottaa toisistaan.

Vastaanottorutiinin MPI_Recv kutsusyntaksi näyttää tältä:

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

Edellä lähetetty viesti voidaan vastaanottaa laskentatehtävässä nolla seuraavasti:

```
MPI_Status status;
int source_id;
   ...
rc = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag,
              MPI_COMM_WORLD, &status);
source_id = status.MPI_SOURCE;
```

Vakio MPI_ANY_SOURCE tarkoittaa, että viesti voidaan vastaanottaa mistä tahansa laskentatehtävästä — argumenttina voisi olla myös jonkin laskentatehtävän järjestysnumero. Vastaanottorutiinin MPI_Recv viimeisessä ar-

gumentissa `status` välitetään mm. viestin lähettäjän järjestysnumero. Viestin lähettäjä saadaan C-kielessä selville MPI_Recv-aliohjelmakutsun jälkeen `status`-muuttujan kentästä MPI_SOURCE.

2.5 MPI-ohjelman ajaminen

MPI-standardissa ei ole määritelty, miten laskentatehtävät käynnistetään rinnakkaiskoneen prosessoreilla. CSC:n Cray T3E -rinnakkaiskoneella tämä tapahtuu komennolla `mpprun`. IBM SP:ssä rinnakkaisajon voi käynnistää komennolla `poe`.

Listauksissa 2.1 ja 2.2 esitettyjen esimerkkiohjelmiemme tulostus voisi olla seuraava käytettäessä neljää laskentatehtävää:

```
message: 2 4 sender: 2
message: 1 4 sender: 1
message: 3 4 sender: 3
```

Siis tehtävä numero nolla vastaanotti viestin ensiksi tehtävältä kaksi ja viimeisenä tehtävältä yksi.

2.6 MPI:n valmiit tietotyypit

Edellisessä Fortran-esimerkissä käytettiin datan lähettämiseen ja vastaanottamiseen MPI:n tyyppiä MPI_INTEGER, joka on eräs MPI:n Fortran-versiossa määritellyistä tietotyypeistä. Muita Fortranin tietotyyppejä ovat MPI_REAL, MPI_CHARACTER, MPI_COMPLEX jne. MPI:n tietotyypit sekä Fortran- että C-kielille on lueteltu taulukossa 2.3.

Taulukko 2.3: *MPI:n tietotyyppejä.*

C-kieli	*Fortran*
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED	MPI_CHARACTER, MPI_INTEGER, MPI_REAL, MPI_COMPLEX, MPI_DOUBLE_PRECISION, MPI_DOUBLE_COMPLEX, MPI_LOGICAL, MPI_BYTE, MPI_PACKED

Siirrettävien alkioiden määrän voi antaa suoraan esimerkiksi REAL-tyyppisten alkioiden lukumääränä — MPI hoitaa datan siirron ja tarvittavat konversiot.

MPI:ssä on myös mahdollista määritellä uusia tietotyyppejä ja välittää niitä laskentatehtävien välillä. Tästä löytyy lisätietoja seuraavassa annetuista viitteistä. Vektorityypin määrittelyä ja käyttöä esitellään kappaleessa 6.5 sivulla 69.

2.7 Lisätietoja

Tässä luvussa esitellyistä MPI:n peruskäsitteistä saa lisätietoja mm. käsikirjoista *MPI: A Message-Passing Interface Standard* [For95] ja *MPI: The Complete Reference* [SOHL+96]. Teokset *Using MPI: Portable Parallel Programming with the Message-Passing Interface* [GLS94] ja *Parallel Programming with MPI* [Pac97] ovat helppolukuisia johdatuksia MPI-ohjelmointiin.

3 Ryhmäoperaatiot

Tässä luvussa esitellään MPI:n tarjoamia mahdollisuuksia kollektiiviseen eli monenväliseen viestintään. Kaikki laskentatehtävät kutsuvat MPI:n kollektiivisia rutiineja samoilla argumenteilla. Yleensä jokin laskentatehtävä on kuitenkin erikoisasemassa, esimerkiksi lähettämässä tietoa muille laskentatehtäville.

3.1 Kollektiivinen viestintä

MPI:llä voi lähettää ja vastaanottaa tietoa myös monen laskentatehtävän välillä. Seuraava esimerkki laskee laskentatehtävien järjestysnumeroiden summan ja lähettää tämän jälkeen summan arvon kaikille laskentatehtäville.

Jokainen laskentatehtävä suorittaa esimerkkiohjelmassamme aliohjelmakutsun MPI_REDUCE. Rutiinin Fortran-kutsumuoto on seuraava:

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer count, datatype, op, root, comm, ierror

Listauksessa 3.1 esitetty Fortran-koodi laskee muuttujien `data` arvojen summat laskentatehtävän nolla muuttujaan `s` seuraavasti:

```
data = id
CALL MPI_REDUCE(data, s, 1, MPI_INTEGER, &
    MPI_SUM, 0, MPI_COMM_WORLD, rc)
```

Aliohjelmakutsun ensimmäinen argumentti `data` sisältää summattavat arvot ja toinen argumentti `s` sisältää tuloksen. Kumpikin muuttuja on tyyppiä MPI_INTEGER.

Rutiinille MPI_REDUCE on tässä annettu laskentaoperaatioksi MPI_SUM. Muut mahdolliset operaatiot on lueteltu taulukossa 3.2.

Jos argumentti `data` olisi taulukko, tulisi muuttujan `s` olla saman kokoinen taulukko. Reduktio-operaatio eli tässä tapauksessa summaus tehtäisiin taulukoille alkioittain.

Aliohjelmakutsussa annettiin `root`-argumentin arvoksi `0`. Tällöin laskentatehtävä nolla on erikoisasemassa: rutiini MPI_REDUCE on epäsymmetrinen eli tulos saadaan muuttujaan `s` vain laskentatehtävässä nolla. Muissa laskentehtävissä ei muuttuja `s` saa arvoa. Jos muuttuja `s` olisi taulukko, tarvitsisi sille varata tilaa vain laskentatehtävässä `root`. Toisaalta myös laskentatehtävän nolla `data`-muuttujan arvo lasketaan mukaan summaan.

Rutiini MPI_BCAST puolestaan lähettää `s`-muuttujan arvon laskentatehtävästä nolla muille laskentatehtäville:

```
CALL MPI_BCAST(s, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, rc)
WRITE(*,*) 'id:', id, 'sum:', s
```

Tämän rutiinin kutsumuoto näyttää seuraavalta:

MPI_BCAST(buffer, count, datatype, root, comm, ierror)
type buffer(*)
integer count, datatype, root, comm, ierror

Muuttujan `buffer` sisältämä data lähetetään argumenttia `root` vastaavasta laskentatehtävästä. Argumentissa `count` kerrotaan lähetettävien ja vastaanotettavien alkioiden lukumäärä. Siis argumentti `buffer` voi olla myös taulukko. Argumentissa `datatype` määritellään datan tyyppi, joka oli edellä MPI_INTEGER.

Seuraavassa on edellä esitetty Fortran-kielinen ohjelmakoodi C-kielellä toteutettuna (katso listausta 3.2):

```
data = id;
rc = MPI_Reduce(&data, &s, 1, MPI_INT, MPI_SUM, 0,
                MPI_COMM_WORLD);
rc = MPI_Bcast(&s, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("id: %d sum: %d\n", id, s);
```

Rutiinissa MPI_Reduce lasketaan siis laskentatehtävien paikallisissa muuttujissa `data` olevien arvojen summa laskentatehtävän nolla muuttujaan `s`. Nämä skalaariargumentit täytyy välittää C-kielessä osoitteina tyyliin `&data` ja `&s`. Rutiini MPI_Bcast lähettää muuttujan `s` arvon laskentatehtävästä nolla viestintäryhmän kaikkiin laskentatehtäviin.

Ohjelman tulostus neljällä laskentatehtävällä voisi olla seuraava:

```
data: 0 sum: 6
data: 2 sum: 6
data: 1 sum: 6
data: 3 sum: 6
```

Taulukossa 3.1 on esitetty edellä käytettyjen rutiinien kutsusyntaksi. Taulukossa 3.2 on puolestaan listattu rutiinin MPI_REDUCE käytettävissä olevat reduktio-operaatiot. Edellä käytettiin operaatiota MPI_SUM.

Rutiinien MPI_REDUCE ja MPI_BCAST toimintaa on havainnollistettu kuvissa 3.1 ja 3.2.

Listaus 3.1: *Kollektiivinen viestintä Fortran-kielellä.*

```
PROGRAM esimerkki
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  INTEGER :: ntasks, id, rc, data, s
  CALL MPI_INIT(rc)
  IF (rc /= MPI_SUCCESS) THEN
    WRITE(*,*) 'MPI initialization failed'
    STOP
  END IF
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
  data = id
  CALL MPI_REDUCE(data, s, 1, MPI_INTEGER, &
      MPI_SUM, 0, MPI_COMM_WORLD, rc)
  CALL MPI_BCAST(s, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, rc)
  WRITE(*,*) 'data:', data, 'sum:', s
  CALL MPI_FINALIZE(rc)
END PROGRAM esimerkki
```

Listaus 3.2: *Kollektiivinen viestintä C-kielellä.*

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char *argv[])
{
  int ntasks, id, rc, data, s;
  rc = MPI_Init(&argc, &argv);
  if (rc != MPI_SUCCESS) {
    printf("MPI initialization failed\n");
    exit(1);
  }
  rc = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &id);
  data = id;
  rc = MPI_Reduce(&data, &s, 1, MPI_INT, MPI_SUM, 0,
                  MPI_COMM_WORLD);
  rc = MPI_Bcast(&s, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("data: %d sum: %d\n", data, s);
  rc = MPI_Finalize();
  exit(0);
}
```

Taulukko 3.1: *Rutiinit MPI_REDUCE ja MPI_BCAST ryhmäviestintään.*

C-kieli	*Fortran*
int **MPI_Reduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)	**MPI_REDUCE**(sendbuf, recvbuf, count, datatype, op, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer count, datatype, op, root, comm, ierror
int **MPI_Bcast**(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)	**MPI_BCAST**(buffer, count, datatype, root, comm, ierror) *type* buffer(*) integer count, datatype, root, comm, ierror

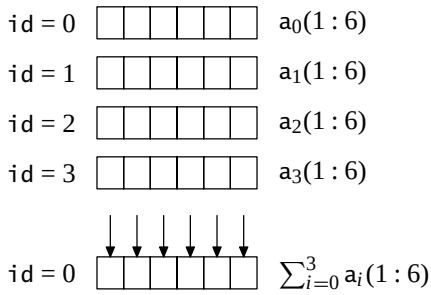
Taulukko 3.2: *MPI:n reduktio-operaatiot.*

Operaatio	*Merkitys*
MPI_MAX	maksimiarvo
MPI_MIN	minimiarvo
MPI_SUM	summa
MPI_PROD	tulo
MPI_LAND	looginen and-operaatio
MPI_BAND	biteittäinen and-operaatio
MPI_LOR	looginen or-operaatio
MPI_BOR	biteittäinen or-operaatio
MPI_LXOR	looginen xor-operaatio
MPI_BXOR	biteittäinen xor-operaatio
MPI_MAXLOC	maksimiarvo ja sijainti
MPI_MINLOC	minimiarvo ja sijainti

Kuvassa 3.1 lasketaan laskentatehtävissä $0,\ldots,3$ olevien a-taulukkojen vastinalkiot yhteen. Merkintä a_i tarkoittaa laskentatehtävän i taulukkoa a. Tätä operaatiota vastaava MPI-kutsu voisi olla C-kielellä

```
float a[6], s[6];
   ...
rc = MPI_Reduce(a, s, 6, MPI_FLOAT, MPI_SUM, 0,
                MPI_COMM_WORLD);
```

Tässä siis laskettiin alkioittain yhteen reaalilukutaulukkoja.

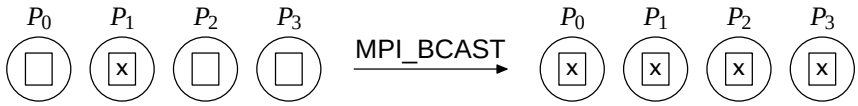


Kuva 3.1: *Rutiinin MPI_REDUCE toiminta.*

Kuvassa 3.2 lähetetään laskentatehtävästä numero yksi muuttujan x arvo muille laskentatehtäville. Vastaava MPI-kutsu voisi olla Fortran-kielellä

```
CALL MPI_BCAST(x, 1, MPI_REAL, 1, MPI_COMM_WORLD, rc)
```

Tässä muuttuja x on tyyppiä REAL oleva Fortran-kielen muuttuja.



Kuva 3.2: *Rutiinin MPI_BCAST toiminta.*

Jos summausoperaation tulos halutaan suoraan selville kaikissa laskentatehtävissä, voidaan käyttää rutiinia MPI_ALLREDUCE, jonka kutsumuoto on lähes samanlainen kun rutiinin MPI_REDUCE, vain `root`-argumentti puuttuu. Seuraavassa on esitetty rutiinin Fortran-kutsumuoto:

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm,
  ierror)
type sendbuf(*), recvbuf(*)
integer count, datatype, op, comm, ierror

Aliohjelmakutsun MPI_ALLREDUCE jälkeen ei siis tarvita erillistä operaatiota MPI_BCAST.

3.2 Muita rutiineita

3.2.1 MPI_SCATTER ja MPI_GATHER

Edellä laskettiin alkioiden summa rutiinin MPI_REDUCE avulla ja lähetettiin sama viesti viestintäryhmän muille jäsenille (MPI_BCAST). MPI:ssä voi myös lähettää eri tietoa yhtä aikaa kullekin viestintäryhmän jäsenelle tai vastaavasti vastaanottaa kaikilta viestintäryhmän jäseniltä. Taulukossa 3.3 on esitetty tähän soveltuvat rutiinit MPI_SCATTER ja MPI_GATHER.

Taulukko 3.3: *Viestien lähetys kaikille viestintäryhmän jäsenille (levitys) ja vastaanotto kaikilta viestintäryhmän jäseniltä (keräys).*

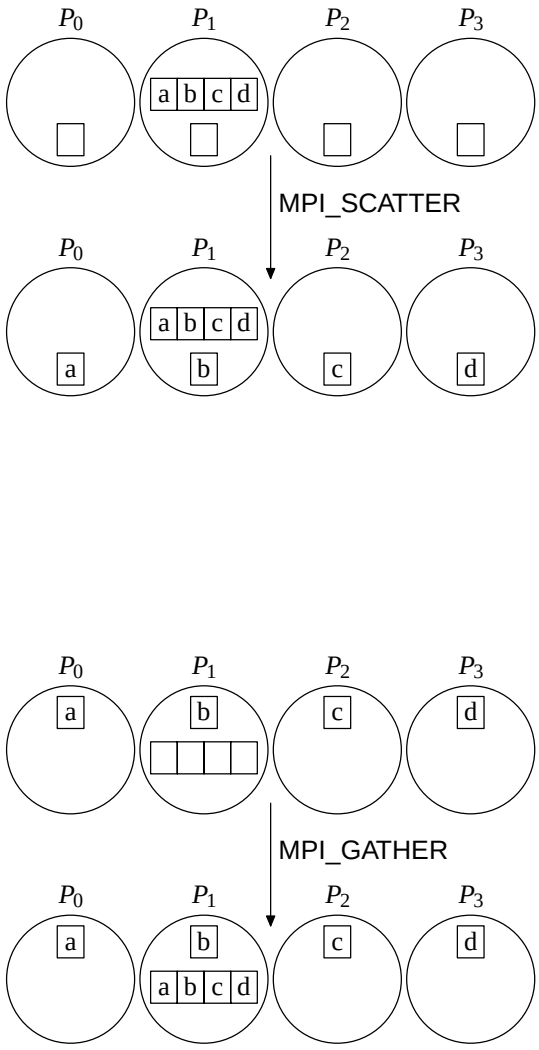
C-kieli	*Fortran*
int **MPI_Scatter**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	**MPI_SCATTER**(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror
int **MPI_Gather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	**MPI_GATHER**(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Näitä rutiineita voi käyttää esimerkiksi levittämään taulukon alkiot laskentatehtävien kesken (rutiini MPI_SCATTER) tai keräämään taulukon alkiot yhteen prosessoriin (rutiini MPI_GATHER).

Rutiinien toimintaa on havainnollistettu kuvassa 3.3 neljän laskentatehtävän tapauksessa. Rutiini MPI_SCATTER lähettää kuvan tapauksessa yhden datalkion laskentatehtävästä P_1 kullekin laskentatehtävälle, ja käänteinen rutiini MPI_GATHER vastaanottaa dataa kultakin laskentatehtävältä.

Rutiinin MPI_SCATTER argumentissa `sendbuf` on laskentatehtävästä `root` lähetettävä data. Argumentissa `sendcount` kerrotaan, kuinka monta alkiota lähetetään kullekin tehtävälle. Taulukon `sendbuf` koon tulee siis olla (vähintään) `sendcount` $\times$ p, missä p on laskentatehtävien lukumäärä. Argumenttiin `recvbuf` välitetään vastaanotettu data. Argumenttien `sendcount` ja `recvcount` tulisi yleensä olla arvoltaan samat, samoin argumenttien `sendtype` ja `recvtype`.

Argumentille `sendbuf` tarvitsee varata tilaa vain `root`-tehtävässä, mutta kaikkien laskentatehtävien tulee antaa tämä argumentti aliohjelmakutsussa. Huomaa, että rutiini MPI_SCATTER lähettää dataa myös itselleen!



Kuva 3.3: *Rutiinien MPI_SCATTER ja MPI_GATHER toiminta.*

Rutiinin MPI_GATHER toiminta on käänteinen rutiiniin MPI_SCATTER nähden. Laskentatehtäviltä kerättävä tieto on argumentissa `sendbuf`. Argumentilla `recvbuf` on merkitystä vain siinä laskentatehtävässä (`root`), joka kerää datan itselleen. Argumenteilla `sendbuf`, `sendcount`, `sendtype`, `root` ja `comm` on merkitystä kaikissa laskentatehtävissä.

Listauksissa 3.3 ja 3.4 on esimerkki MPI:n levitys- ja keräysrutiinien käytöstä (MPI_SCATTER ja MPI_GATHER). Tässä esimerkissä laskentatehtävä nolla lähettää aluksi kolme lukua kullekin laskentatehtävälle. Tämän jälkeen lasketaan sinifunktion arvot näissä pisteissä. Lopuksi kerätään funktion arvot taulukkoon `values` laskentatehtävään P_0.

C-kielisen ohjelmaversion tulostus voisi olla neljän laskentatehtävän tapauksessa seuraava:

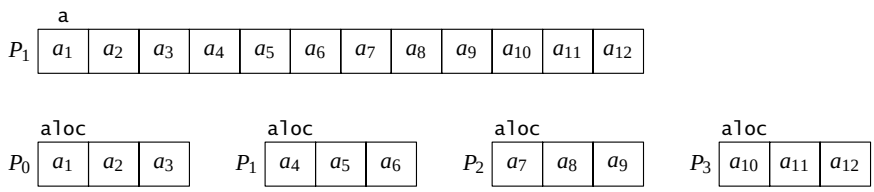
```
ntasks = 4, asize = 12
Results are:
0.000000, 0.000000
0.272727, 0.269359
0.545455, 0.518807
0.818182, 0.729904
1.090909, 0.887047
1.363636, 0.978619
1.636364, 0.997851
1.909091, 0.943322
2.181818, 0.819062
2.454545, 0.634257
2.727273, 0.402567
3.000000, 0.141120
```

Kukin laskentatehtävä siis laski sinifunktion arvon kolmelle peräkkäiselle `a`-taulukon alkiolle.

Datan lähettämistä on havainnollistettu kuvassa 3.4. Taulukon `a` arvot

$$(a_1, a_2, \ldots, a_{12})$$

lähetetään laskentatehtävästä P_1 rutiinilla MPI_SCATTER. Argumenttien `sendcount` ja `recvcount` arvo on 3. Data vastaanotetaan kaikissa laskentatehtävissä taulukkoon `aloc`. Esimerkiksi laskentatehtävä P_0 vastaanottaa alkiot (a_1, a_2, a_3).



Kuva 3.4: *Rutiinin MPI_SCATTER käyttöesimerkki.*

Listaus 3.3: *Rutiinien MPI_SCATTER ja MPI_GATHER käyttö Fortran-kielellä.*

```
PROGRAM gs

  IMPLICIT NONE

  INCLUDE 'mpif.h'

  INTEGER, PARAMETER :: local_size = 3
  INTEGER, PARAMETER :: root_id = 0
  REAL, ALLOCATABLE, DIMENSION(:) :: a, values
  REAL, DIMENSION(local_size) :: aloc, vloc
  INTEGER :: asize, ntasks, id, rc, i

  CALL MPI_INIT(rc)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)

  IF (id == root_id) THEN
     asize = ntasks * local_size
     ALLOCATE(a(asize), values(asize))
     a = (3.d0*(/ (i, i = 0, asize-1) /)) / (asize-1)
     WRITE(*,*) 'ntasks = ', ntasks, ', asize = ', asize
  END IF

  CALL MPI_SCATTER(a, local_size, MPI_REAL, &
      aloc, local_size, MPI_REAL, root_id, &
      MPI_COMM_WORLD, rc)

  vloc = SIN(aloc)

  CALL MPI_GATHER(vloc, local_size, MPI_REAL, &
      values, local_size, MPI_REAL, root_id, &
      MPI_COMM_WORLD, rc)

  IF (id == root_id) THEN
     WRITE(*,*) 'Results are:'
     DO i = 1, asize
       WRITE(*,*) a(i), values(i)
     END DO
  END IF

  CALL MPI_FINALIZE(rc)

END PROGRAM gs
```

Listaus 3.4: *Rutiinien MPI_Scatter ja MPI_Gather käyttö C-kielellä.*

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

#define local_size 3
#define root_id 0

int main(int argc, char **argv)
{
  float *a, *values;
  float aloc[local_size], vloc[local_size];
  int asize, ntasks, id, rc, i;

  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &id);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

  if (id == root_id) {
    asize = ntasks * local_size;
    a = (float *)malloc(asize*sizeof(*a));
    values = (float *)malloc(asize*sizeof(*values));
    for (i=0; i < asize; i++)
      a[i] = 3*i/((float)asize-1);
    printf("ntasks = %d, asize = %d\n", ntasks, asize);
  }

  rc = MPI_Scatter(a, local_size, MPI_FLOAT,
                   aloc, local_size, MPI_FLOAT, root_id,
                   MPI_COMM_WORLD);

  for (i=0; i < local_size; i++)
    vloc[i] = sin(aloc[i]);

  rc = MPI_Gather(vloc, local_size, MPI_FLOAT,
                  values, local_size, MPI_FLOAT, root_id,
                  MPI_COMM_WORLD);

  if (id == root_id) {
    printf("Results are:\n");
    for (i=0; i < asize; i++)
      printf("%f, %f\n", a[i], values[i]);
  }

  rc = MPI_Finalize();
}
```

3.2.2 MPI_SCATTERV ja MPI_GATHERV

MPI:ssä on edellisten rutiinien lisäksi käytettävissä ryhmäviestintärutiinit MPI_SCATTERV ja MPI_GATHERV, joilla voi lähettää ja vastaanottaa eri määrän dataa kultakin laskentatehtävältä.

Listauksessa 3.5 on esitetty MPI_SCATTERV-rutiinin käyttö. Ohjelmassa lähetetään taulukon `sendbuf` sisältö laskentatehtävästä 0. Taulukkoon `displs` sijoitetaan kullekin laskentatehtävälle lähetettävän datan etäisyys taulukon `sendbuf` alusta laskien. Etäisyydet lasketaan lähetettävinä yksiköinä eli tässä tapauksessa reaalilukuina (MPI:n tyyppi MPI_REAL). Koodissa käytetään arvoja $(0,4,8,12,\ldots)$ näinä askelpituuksina, jolloin lähetettävä taulukko jaetaan tasavälisiin osiin. Lähetettävien alkioiden lukumäärä lasketaan taulukkoon `sendcounts`. Taulukon arvoiksi asetetaan luvut $(1,2,3,4,4,\ldots)$, missä arvo 4 on maksimimäärä lähetettäviä arvoja.

Listaus 3.5: *Rutiinin MPI_SCATTERV käyttö datan lähettämiseen.*

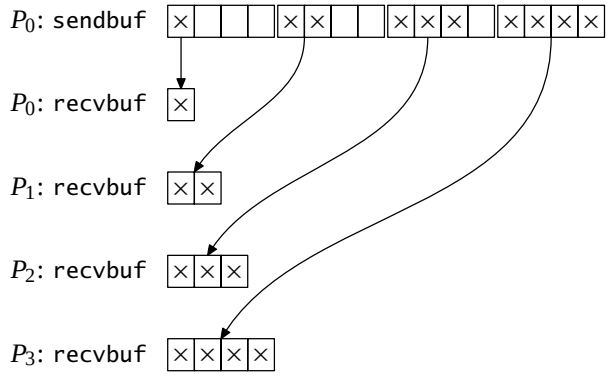
```
PROGRAM gv
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  INTEGER, PARAMETER :: n = 4, root_id = 0
  REAL, DIMENSION(:), ALLOCATABLE :: recvbuf, sendbuf
  INTEGER, DIMENSION(:), ALLOCATABLE :: &
      sendcounts, displs
  INTEGER :: ntasks, id, rc, allocstat, recvcount, i
  CALL MPI_INIT(rc)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
  IF (id == root_id) THEN
    ALLOCATE(sendbuf(n*ntasks), displs(ntasks), &
        stat=allocstat)
    IF (allocstat /= 0) STOP 'virhe muistinvarauksessa!'
    sendbuf = 10*(/ (i, i = 1, n*ntasks) /)
    displs = n*(/ (i, i = 0, ntasks-1) /)
    ALLOCATE(sendcounts(0:ntasks-1), stat=allocstat)
    IF (allocstat /= 0) STOP 'virhe muistinvarauksessa!'
    sendcounts = MIN(n, (/ (i, i = 1, ntasks) /))
  END IF
  recvcount = MIN(n, id + 1)
  ALLOCATE(recvbuf(recvcount), stat=allocstat)
  IF (allocstat /= 0) STOP 'virhe muistinvarauksessa!'
  recvbuf = 0.0
  CALL MPI_SCATTERV(sendbuf, sendcounts, displs, &
      MPI_REAL, recvbuf, recvcount, MPI_REAL, &
      root_id, MPI_COMM_WORLD, rc)
  WRITE (*,'(A,I3,",",(5F8.2))') 'id, recvbuf: ', id, recvbuf
  CALL MPI_FINALIZE(rc)
END PROGRAM gv
```

Huomaa, että taulukoille `sendbuf`, `sendcounts` ja `displs` varataan tilaa vain tehtävässä `root_id`.

Ohjelman tulostus voisi näyttää seuraavalta:

```
id, recvbuf:   1,   50.00   60.00
id, recvbuf:   2,   90.00  100.00  110.00
id, recvbuf:   0,   10.00
id, recvbuf:   3,  130.00  140.00  150.00  160.00
```

Tässä laskentatehtävien lukumäärä oli neljä.



Kuva 3.5: *Rutiinin MPI_SCATTERV toiminta. Tehtävä P_0 lähettää neljän alkion pituisista lohkoista eri määrän dataa eri laskentatehtäville.*

Viestinvälitystä on havainnollistettu kuvassa 3.5. Taulukon `displs` arvojen $(0, 4, 8, 12, \ldots)$ askelpituus on neljä, jolloin lähetettävä data haetaan neljän alkion pituisista lohkoista. Ensimmäisesta lohkosta lähetetään vain yksi alkio, toisesta kaksi jne.

Rutiinien MPI_SCATTERV ja MPI_GATHERV kutsumuoto on esitetty taulukossa 3.4.

3.2.3 MPI_ALLGATHER ja MPI_ALLREDUCE

Rutiineilla MPI_ALLGATHER ja MPI_ALLREDUCE voi kerätä tietoa yhtä aikaa moneen laskentatehtävään. Rutiinien kutsusyntaksit on esitetty taulukossa 3.5. Näiden rutiinien käytöstä löytyy esimerkkejä luvusta 6 sivulta 51 alkaen.

Näille rutiineille ei anneta juuritehtävää (`root`) kuten rutiinille MPI_REDUCE, koska kaikki laskentatehtävät ovat samanarvoisia. MPI_ALLREDUCE-aliohjelmalla voi laskea esimerkiksi summan, jonka arvo palautetaan automaattisesti kaikille laskentatehtäville. Tulos on sama kuin peräkkäin annettujen aliohjelmakutsujen MPI_REDUCE ja MPI_BCAST.

Rutiini MPI_ALLGATHER puolestaan kerää hajautetun taulukon kaikille laskentatehtäville. Tulos on sama kuin kerättäessä taulukko yhteen tehtävään rutiinilla MPI_GATHER ja lähetettäessä tulos kaikille tehtäville aliohjelmalla MPI_BCAST.

Taulukko 3.4: *Eri pituisten viestien lähetys kaikille viestintäryhmän jäsenille (levitys) ja vastaanotto kaikilta viestintäryhmän jäseniltä (keräys).*

C-kieli	*Fortran*
int **MPI_Scatterv**(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	**MPI_SCATTERV**(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcounts(*), displs(*), sendtype, recvcount, recvtype, root, comm, ierror
int **MPI_Gatherv**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)	**MPI_GATHERV**(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcounts(*), displs(*), recvtype, root, comm, ierror

3.3 Laskennan tahdistus

MPI:n ryhmäviestinnässä on käytettävissä tahdistusrutiini (synchronization) MPI_BARRIER. Laskentatehtävät siis jatkavat matkaa vasta, kun kaikki ovat suorittaneet aliohjelmakutsun. Rutiinin kutsusyntaksi on esitetty taulukossa 3.6.

Laskentatehtävien tahdistaminen voi huonontaa kuorman tasausta, sillä muut odottavat hitainta tehtävää. Onneksi rinnakkaislaskennassa selvitään usein ilman erillisiä tahdistuskutsuja.

3.4 Lisätietoja

Ryhmäviestintärutiineistä löytyy lisäesimerkkejä luvusta 6 sivulta 51 alkaen.

Tässä luvussa esitettyjen rutiinien lisäksi MPI:ssä on muitakin ryhmäviestintään soveltuvia rutiineita. Lisätietoja löytyy esimerkiksi teoksista *MPI: A Message-Passing Interface Standard* [For95] ja *MPI: The Complete Reference* [SOHL+96].

Taulukko 3.5: *Viestien keräys kaikille viestintäryhmän jäsenille sekä reduktio-operaatio yhtä aikaa kaikille laskentatehtäville.*

C-kieli	*Fortran*
int **MPI_Allgather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)	**MPI_ALLGATHER**(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror) *type* sendbuf(*), recvbuf(*) integer sendcount, sendtype, recvcount, recvtype, comm, ierror
int **MPI_Allreduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	**MPI_ALLREDUCE**(sendbuf, recvbuf, count, datatype, op, comm, ierror) *type* sendbuf(*), recvbuf(*) integer count, datatype, op, comm, ierror

Taulukko 3.6: *MPI:n tahdistusrutiini MPI_BARRIER.*

C-kieli	*Fortran*
int **MPI_Barrier**(MPI_Comm comm)	**MPI_BARRIER**(comm, ierror) integer comm, ierror

4 MPI:n edistyneitä piirteitä

Tässä luvussa esitellään MPI:n edistyneempiä piirteitä eli erilaisia viestintämuotoja sekä viestintäryhmien käyttöä. Näiden piirteiden hallinta ei ole kuitenkaan välttämätöntä MPI-ohjelmoinnissa. Esitetyistä keinoista saattaa silti olla apua joidenkin ongelmien nopeammassa ratkaisemisessa — ennen kaikkea ohjelmointiin kuluvan ajan säästämisessä.

Esiteltyjen rutiinien käyttöä ei käydä läpi samalla perusteellisuudella kuin edellisissä luvuissa.

4.1 MPI:n viestintämuodot

Luvussa 2 esiteltiin kahdenvälisen viestinnän peruskomennot eli aliohjelmat MPI_SEND ja MPI_RECV. Näillä komennoilla tulee yleensä hyvin toimeen, mutta joskus suorituskyvyn parantamiseksi voi käyttää hiukan edistyneempiä versioita perusrutiineista.

MPI_BSEND on *puskuroitu* (buffered) lähetysrutiini, joka vaatii käyttäjää varaamaan tarvittavan puskuritilan ennen viestin lähetystä. Puskuritila varataan komennolla MPI_BUFFER_ATTACH. Puskuritilaa ei saa vapauttaa tai käyttää uudestaan ennen kuin viesti on vastaanotettu. Puskuroitu lähetysrutiini palaa kutsuneeseen ohjelmaan heti, kun viesti on kopioitu lähetyspuskuriin, vaikkei vastaavaa vastaanottorutiinia olisikaan vielä aloitettu.

Puskuroinnin etuna on siis se, että laskentaa voidaan ruveta tekemään heti, kun viesti on kopioitu lähetyspuskuriin. Huonona puolena on se, että tämä kopiointi voi hidastaa ohjelmaa. Lisäksi lähetyspuskuri vie tilaa koneen keskusmuistista. Yleensä kannattaakin käyttää normaalia MPI_SEND-rutiinia. Puskuroinnin sijaan voi yrittää käyttää myös seuraavassa kappaleessa esiteltäviä ns. välittömästi palaavia viestintärutiineja.

MPI_SSEND on *synkronoitu* lähetysrutiini, joka palaa kutsuneeseen ohjelmaan vasta, kun vastaava vastaanottorutiini on käynnistetty. Kun aliohjelma MPI_SSEND palaa kutsuvaan ohjelmaan, saa lähetyspuskurin käyttää heti uudestaan. Synkronoitu lähetys voi olla nopeampaa kuin normaalin lähetysrutiinin MPI_SEND käyttö, mutta tämä riippuu käytetystä konearkkitehtuurista.

MPI_RSEND on *vastaanottajan valmiuteen* (ready mode) perustuva lähetysrutiini, jota saa käyttää ainoastaan, jos vastaava vastaanottorutiini on jo toiminnassa. Jos vastaanottorutiinia ei ole toiminnassa, on rutiinia MPI_RSEND käytetty väärin eikä rutiinin toimintaa ole määritelty.

Kaikkien ylläolevien rutiinien argumentit ovat täysin samat kuin rutiinilla MPI_SEND (katso taulukko 2.1 sivulla 16).

4.2 Välittömästi palaavat viestintärutiinit

Toisinaan olisi hyödyllistä tehdä laskentaa sillä aikaa, kun odotellaan viestin lähettämisen tai vastaanottamisen valmistumista. MPI sisältää mahdollisuuden myös tähän niin sanottujen *välittömästi palaavien* viestintärutiinien (non-blocking communication) avulla. Nämä rutiinit käynnistävät viestin lähetyksen tai vastaanoton, mutta palaavat heti suorittamaan ohjelmakoodin seuraavia lauseita. Viestin lähetys- tai vastaanottopuskureihin ei saa koskea, ennen kuin operaatio on valmistunut.

Normaalit viestintärutiinit (esimerkiksi MPI_SEND ja MPI_RECV) eivät palaa välittömästi, mikä tarkoittaa sitä, että rutiineista palataan ohjelmakoodiin vasta sitten, kun viesti on saatu lähetettyä tai vastaanotettua tai viestintä on ainakin käynnistynyt ja puskuritila voidaan käyttää uudestaan.

Välittömästi palaavien viestintärutiinien nimet ovat samat kun normaalien viestintärutiinien, mutta nimeen tulee lisää `I`-kirjain (immediate). Siis nimeksi tulee esimerkiksi MPI_IRSEND. Lisäksi argumenttilistaan tulee viestintäryhmän jälkeen `request`-muuttuja, jonka tyyppi on C-kielessä MPI_Request ja Fortranissa `INTEGER`. Tätä `request`-muuttujaa käytetään lähetyksen tilan testaamiseen (valmis tai kesken) rutiineilla MPI_WAIT ja MPI_TEST. Näiden rutiinien kutsusyntaksi on esitetty taulukossa 4.1.

Rutiini MPI_WAIT palaa kutsuvaan ohjelmaan vasta, kun `request`-muuttujaa vastaava viestintätapahtuma on valmis. Rutiini MPI_TEST palaa kutsuvaan

Taulukko 4.1: *Viestien valmiustilan tutkimisrutiinien käyttö C-kielestä ja Fortranista.*

C-kieli	*Fortran*
int **MPI_Wait**(MPI_Request *request, MPI_Status *status)	**MPI_WAIT**(request, status, ierror) integer request, status(MPI_STATUS_SIZE), ierror
int **MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status)	**MPI_TEST**(request, flag, status, ierror) logical flag integer request, status(MPI_STATUS_SIZE), ierror

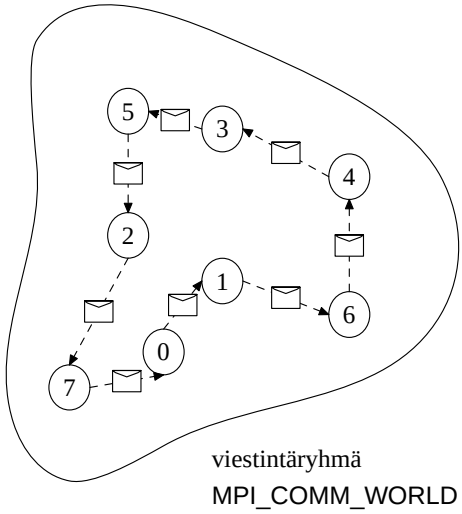
ohjelmaan välittömästi ja palauttaa arvon `flag=.true.`, jos `request`-muuttujaa vastaava tapahtuma on valmis, muuten palautettu arvo on epätosi eli `flag=.false.`.

Kaikkien ylläolevien rutiinien vastaanottorutiiniksi käy MPI_RECV eli normaali vastaanottorutiini. Viestien vastaanottamiseen on myös käytettävissä välittömästi palaava rutiini MPI_IRECV. Tämän rutiinin argumenttilistassa on myös `request`-muuttuja, jota käytetään selvittämään, onko vastaanotto jo valmistunut.

Listauksessa 4.1 on esimerkki välittömästi palaavien rutiinien käytöstä. Laskentatehtävä numero 0 lähettää tehtävälle 1 taulukon `a` sisällön aliohjelmakutsun MPI_ISEND avulla. Viestin lähtemistä tarkistetaan rutiinin MPI_TEST avulla. Laskentatehtävä 0 voi tehdä muuta laskentaa samaan aikaan kun viesti on lähetettävänä. Vastaavasti laskentatehtävä 1 vastaanottaa dataa rutiinin MPI_IRECV avulla. Viestin tuloa jäädään lopulta odottamaan aliohjelmakutsussa MPI_WAIT.

Edellä esiteltyjä lähetys- tai vastaanottorutiineita tulisi käyttää ainoastaan, jos on varma, mitä tekee. Yleensä rutiinit MPI_SEND ja MPI_RECV riittävät hyvin, sillä ne on jo optimoitu. Lisäksi niissä on varmistuksia *lukkiintumisen* (deadlock) varalta enemmän kuin muissa lähetystavoissa. Tämä varmistus on tietenkin tehty suorituskyvyn kustannuksella, joten joskus on täysin perusteltua käyttää edistyneempiä versioita kahdenvälisistä viestintärutiineista.

MPI:n hyviä puolia onkin mm. se, että normaaleilla viestinvälitysrutiineilla on melko vaikeaa saada aikaan lukkiintumista. Kuvassa 4.1 on esimerkki tilanteesta jossa tämä voisi tapahtua. Tässä kaikki laskentatehtävät lähettävät viestin eikä yksikään laskentatehtävä vastaanota, ennen kuin oma viesti on lähetetty. Käytännössä kuvassa 4.1 esitetty tilanne lukkiintuu monis-



Kuva 4.1: *Esimerkki rinnakkaisalgoritmin lukkiintumisesta.*

sa MPI-toteutuksessa vain silloin, kun käytetään synkronista lähetysrutiinia MPI_SSEND tai lähetetään hyvin suuria datamääriä ilman puskurointia.

4.3 Viestintäryhmät

Luvuissa 2 ja 3 on jo tutustuttu viestintäryhmään MPI_COMM_WORLD, joka sitoo yhteen kaikki kutsulla MPI_INIT käynnistetyt laskentatehtävät. Joskus kuitenkin tulee eteen ongelma, jossa tehtävät pitäisi loogisesti jakaa erilaisiin ryhmiin. Esimerkiksi puolet laskentatehtävistä voisivat laskea Fourier'n muunnosta ja loput voisivat tehdä data-aineiston järjestelemistä. Tällöin on loogista käyttää eri viestintäryhmää kummankin ryhmän sisäisiin viesteihin.

Uusien viestintäryhmien luominen onnistuu rutiinilla MPI_COMM_SPLIT (katso taulukkoa 4.2). Jokaisen laskentatehtävän on kutsuttava tätä kollektiivista rutiinia.

Taulukko 4.2: *Rutiinin MPI_COMM_SPLIT käyttö C-kielestä ja Fortranista.*

C-kieli	*Fortran*
int **MPI_Comm_split**(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)	**MPI_COMM_SPLIT**(comm, color, key, newcomm, ierror) integer comm, rank, key, newcomm, ierror

Rutiinin MPI_COMM_SPLIT argumentit määrittävät, miten viestintäryhmä jaetaan osiin. Argumentti `color` määrää ryhmän siten, että jokainen tehtävä, jonka MPI_COMM_SPLIT-kutsun `color`-arvo on sama, kuuluu samaan uuteen viestintäryhmään. Argumentti `key` määrää tehtävien järjestyksen uudessa viestintäryhmässä. Mitä pienempi `key`-argumentti on, sitä pienempi on järjestysnumero `rank` uudessa viestintäryhmässä. Tehtävät, joiden `key`-argumentti on sama, järjestetään samaan järjestykseen, jossa ne ovat vanhassa viestintäryhmässä.

Sekä `color`-argumentin että argumentin `key` pitää olla positiivisia. Argumentilla `color` on sallittuna arvona myös MPI_UNDEFINED, jolloin kyseinen tehtävä ei kuulu mihinkään uuteen viestintäryhmään.

Uutta viestintäryhmää voidaan käyttää kaikissa MPI-kutsuissa viestintäryhmän paikalla, ja kyseinen MPI-kutsu koskee vain tätä viestintäryhmää.

Seuraavassa esimerkkiohjelmassa käytetään viestintäryhmiä suorittamaan kaksi erilaista laskentatoimitusta yhtäaikaa. Laskutoimituksina ovat vektorin parillisten alkioiden summan laskeminen ja parittomien tulon laskeminen. Viestintäryhmä MPI_COMM_WORLD jaetaan kahteen eri aliviestintäryhmään käyttäen rutiinia MPI_Comm_split, jossa toisessa on alkuperäiset parilliset tehtävät (järjestettynä MPI_Comm_rank-arvojen mukaan) ja toisessa alkuperäiset

parittomat tehtävät. Laskentatehtävien alustusvaiheessa kaikille tehtäville lähetetään sama vektori rutiinilla MPI_Bcast.

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

#define MAX_PROCS 16

void main(int argc, char *argv[])
{
  int myid, mysubid, numprocs, color, key;
  int j;
  MPI_Status rc;
  MPI_Comm subcomm;
  MPI_Op operator;
  float a[MAX_PROCS], result_odd, result_even;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* Tehtävä 0 alustaa vektorin a satunnaisilla arvoilla. */
  if (myid == 0) {
    srand(1);
    for (j = 0; j < numprocs; j++) {
      a[j] = (float)rand()/32767.0;
      printf("%d: %f\n",j,a[j]);
    }
  }

  /* Lähetetään vektori a kaikille tehtäville. */
  MPI_Bcast(a,numprocs,MPI_FLOAT,0,MPI_COMM_WORLD);

  if (myid%2 == 0) {
    color = 1;
  } else {
    color = 2;
  }
  MPI_Comm_split(MPI_COMM_WORLD,color,myid,&subcomm);

  /* Tehtävän järjestysnumero uudessa viestintäryhmässä.*/
  MPI_Comm_rank(subcomm,&mysubid);

  /* Lasketaan toisessa uudessa viestintaryhmässä
     parillisten summa ja toisessa parittomien tulo. */

  if (myid%2 == 0) {
    operator = MPI_SUM;
  } else {
    operator = MPI_PROD;
  }
  MPI_Reduce(&a[myid],&result_odd,1,MPI_FLOAT,\
             operator,0,subcomm);
```

```
  if (myid == 1) {
    MPI_Send(&result_odd,1,MPI_FLOAT,0,99,MPI_COMM_WORLD);
  }

  if (myid == 0) {
    result_even = result_odd;
    MPI_Recv(&result_odd,1,MPI_FLOAT,1,99,\
             MPI_COMM_WORLD,&rc);
    printf("Parill. summa = %f\nParitt. tulo = %f\n",\
             result_even,result_odd);
  }

  MPI_Finalize();
}
```

Uusien viestintäryhmien sisällä katsotaan viestintäryhmän sisäinen, *uusi* järjestysnumero käyttäen jälleen rutiinia MPI_Comm_rank, kuten aivan ohjelman alussakin, viestintäryhmäargumentti on vain toinen:

```
MPI_Comm_rank(subcomm,&mysubid);
```

Uusi järjestysnumero `mysubid` pätee ainoastaan uuden viestintäryhmän sisällä. Vanhassa viestintäryhmässä järjestysnumerot pysyvät ennallaan.

Laskutoimituksen tyypin määrää järjestysnumero `myid` viestintäryhmässä MPI_COMM_WORLD. Jos järjestysnumero on parillinen, laskutoimituksena on yhteenlasku, muuten kertolasku:

```
  if (myid%2 == 0) {
    operator = MPI_SUM;
  } else {
    operator = MPI_PROD;
  }
```

Laskutoimitus suoritetaan yhdellä rutiinilla MPI_Reduce, joka operoi viestintäryhmän sisällä. Koska viestintäryhmistä toinen sisältää parilliset ja toinen parittomat tehtävät, saadaan vaaditut arvot laskettua yhdellä MPI_Reduce-kutsulla, vain operaatio on eri.

Lopussa tulokset kootaan yhdelle laskentatehtävälle, joka tulostaa vastaukset.

4.4 Topologiat

MPI:ssä voi määritellä laskentatehtäville sisäisiä topologioita, esimerkiksi kuutiohiloja. Sisäiset topologiat ovat *virtuaalisia* eli *näennäisiä* topologioita, eivätkä millään tavalla ole riippuvaisia prosessoreiden todellisesta kytkentäverkosta. Virtuaaliset topologiat ovat hyödyllisiä joissakin ongelmissa vähentäen ohjelmoijan tarvetta miettiä, mille tehtävälle viesti lähetetään ja mistä vastaanotetaan. Topologiat voivat MPI:ssä olla joko *karteesisia topologioita* tai yleisiä *graafi-topologioita*. Tässä kappaleessa käsitellään ainoastaan karteesisia topologioita.

Karteesiset topologiat (kuutiohilat) esiintyvät sen verran usein eri ongelmien

yhteydessä, että niiden käsittelyyn on MPI:ssä kehitetty omat rutiininsa. Tällöin ei tarvitse määritellä karteesisia topologioita yleisten graafien avulla ja ohjelmointivaiva vähenee hiukan. Taulukossa 4.3 on esitelty yleisimmät karteesisen topologian käsittelyyn tarvittavat rutiinit.

Rutiinilla MPI_CART_CREATE luodaan karteesisen topologian mukainen viestintäryhmä. Uusi karteesinen viestintäryhmä luodaan viestintäryhmän `oldcomm` jäsenistä. Argumentti `ndims` on ulottuvuuksien lukumäärä karteesisessa topologiassa ja taulukon `dims` alkioissa on kunkin ulottuvuuden laskentatehtävien lukumäärä. Taulukon `periods` kukin alkio kertoo, onko kyseinen ulottuvuus periodinen vai ei, ja taulukko `reorder` kertoo, saako tehtävien järjestystä muuttaa siten, että saadaan tehokkaammat viestintäpolut. Muuttujassa `newcomm` palautetaan uusi karteesinen viestintäryhmä.

Rutiinilla MPI_CARTDIM_GET voi kysyä karteesisen viestintäryhmän ulottuvuuksien lukumäärän.

Rutiinilla MPI_CART_GET kysytään tietoa karteesisesta viestintäryhmästä. Rutiinille annetaan arvot `comm` ja `maxdims`. Rutiini palauttaa halutut tiedot muissa argumenteissa. Tämä informaatio on sama kuin MPI_CART_CREATE-kutsussa.

Rutiini MPI_CART_RANK palauttaa `coords`-taulukon arvoja vastaavan tehtävän järjestysnumeron `rank`.

Rutiini MPI_CART_COORDS on MPI_CART_RANK-rutiinin käänteisrutiini ja palauttaa argumenttia `rank` vastaavat koordinaatit.

Rutiini MPI_CART_SHIFT palauttaa vastaanottajan järjestysnumeron (`dest`) ja lähettäjän järjestysnumeron (`source`), joita laskentatehtävä voi käyttää siirto-operaatiossa (shift). Tämä operaatio kohdistetaan yleensä taulukon alkioihin. Argumentti `direction` kertoo suunnan, jonne siirretään, ja `disp` kertoo, kuinka monta hyppyä tehdään. Tätä rutiinia käytetään yleensä apuna kutsuttaessa rutiinia MPI_SENDRECV.

Rutiini MPI_SENDRECV suorittaa sekä lähetyksen että vastaanottamisen yhtäaikaa. Argumentteina on samat argumentit, joita löytyy sekä MPI_SEND- että MPI_RECV-rutiinista. Rutiinia käytetään yleensä yhdessä rutiinin MPI_CART_SHIFT kanssa.

Karteesista topologiaa luotaessa täytyy antaa eri ulottuvuuksille tieto siitä, onko kyseinen ulottuvuus periodinen vai ei. Jos ulottuvuus on periodinen, silloin voidaan lähettää viesti ”oikealle” kuvan 4.2 mukaisesti ja oikeanpuoleisimmasta tehtävästä lähetetty viesti ”kiertää” vasemmalle.

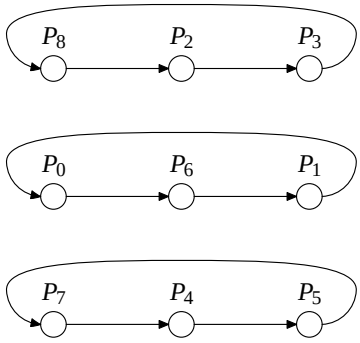
Jos ulottuvuus ei ole periodinen, lähetetty viesti kulkee kuvan 4.3 mukaisesti. Kuvaan on merkitty harmaalla reunatehtävät. Näiden arvona on valmiiksi nimetty vakio MPI_UNDEFINED, minkä esimerkiksi MPI_CART_SHIFT palauttaa reunatehtävälle. Tälläiselle haamutehtävälle voi lähettää viestin, mutta viesti häviää. Samoin haamutehtävältä voi vastaanottaa viestin, mutta viesti ei ole järjellinen.

Taulukko 4.3: *Karteesisen topologian käsittelyyn liittyviä rutiineita.*

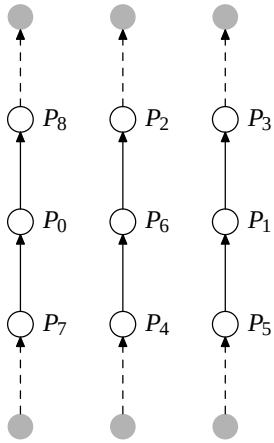
C-kieli	*Fortran*
int **MPI_Cart_create**(MPI_Comm oldcomm, int ndims,int *dims, int *periods, int reorder, MPI_Comm *newcomm)	**MPI_CART_CREATE**(oldcomm, ndims, dims, periods, reorder, newcomm, ierror) integer oldcomm, ndims, dims(ndims), periods(ndims), reorder, newcomm, ierror
int **MPI_Cartdim_get**(MPI_Comm comm, int *ndims)	**MPI_CARTDIM_GET**(comm, ndims, ierror) integer comm, ndims, ierror
int **MPI_Cart_get**(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)	**MPI_CART_GET**(comm, maxdims, dims, periods, coords, ierror) integer comm, maxdims, dims(maxdims), periods(maxdims), reorder, newcomm, ierror
int **MPI_Cart_rank**(MPI_Comm comm, int *coords, int *rank)	**MPI_CART_RANK**(comm, coords, rank, ierror) integer comm, coords(ndims), rank, ierror
int **MPI_Cart_coords**(MPI_Comm comm, int rank, int maxdims, int *coords)	**MPI_CART_COORDS**(comm, rank, maxdims, coords, ierror) integer comm, rank, maxdims, coords(ndims), ierror
int **MPI_Cart_shift**(MPI_Comm comm, int direction, int disp, int *source, int *dest)	**MPI_CART_SHIFT**(comm, direction, disp, source, dest, ierror) integer comm, direction, disp, source, dest, ierror
int **MPI_Sendrecv**(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)	**MPI_SENDRECV**(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror) *type* sendbuf(*), recvbuf(*) integer sendtype, dest, sendtag, recvcount, recvtype, source, recvtag, comm, status(MPI_STATUS_SIZE), ierror

Karteesista topologiaa luodessa voi käyttää `reorder`-argumentille arvoa tosi (C:ssä arvo $\neq 0$ ja Fortranissa `.true.`), jolloin tehtävien järjestys voi muuttua millaiseksi tahansa, kuten kuvista 4.2 ja 4.3 näkyy. Tämä tarkoittaa, että data (vektori tai taulukko) voi olla mielivaltaisessa järjestyksessä tehtävissä, jos data on jaettu ennen karteesisen topologian luomista. Data kannattaa siis jakaa vasta topologian luomisen jälkeen. Yleensä datan jakautumisen tietäminen etukäteen on hankalaa, ja koodia tehtäessä ei tulisi luottaa minkäänlaiseen järjestyksen säilymiseen virtuaalitopologioiden kanssa, vaan aina pyrkiä jakamaan data vasta topologian luomisen jälkeen.

Karteesisten topologioiden käytöstä on esimerkkinä Life-pelin toteutus MPI:llä sivulla 69.



Kuva 4.2: *Viestin kiertäminen periodisen reunaehdon tapauksessa.*



Kuva 4.3: *Viestin kulkeminen ei-periodisen reunaehdon tapauksessa.*

4.5 Lisätietoja

MPI:n edistyneistä piirteistä saa lisätietoa mm. teoksista *Parallel Programming with MPI* [Pac97], *MPI: A Message-Passing Interface Standard* [For95] ja *MPI: The Complete Reference* [SOHL+96].

Listaus 4.1: *Välittömästi palaavien rutiinien käyttö.*

```
PROGRAM nb

  IMPLICIT NONE

  INCLUDE 'mpif.h'

  INTEGER, PARAMETER :: n = 100, tag = 100
  INTEGER, DIMENSION(n) :: a
  INTEGER :: rc, id, comm_request
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
  LOGICAL :: comm_done

  CALL MPI_INIT(rc)
  IF (rc /= MPI_SUCCESS) STOP 'initialization failed!'
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)

  IF (id == 0) THEN
    CALL MPI_ISEND(a, n, MPI_INTEGER, 1, tag, &
        MPI_COMM_WORLD, comm_request, rc)
    DO
      CALL MPI_TEST(comm_request, comm_done, status, rc)
      IF (comm_done) EXIT
      ! Tehdään muuta laskentaa, ei kosketa taulukkoon a.
    END DO
  ELSE IF (id == 1) THEN
    CALL MPI_IRECV(a, n, MPI_INTEGER, 0, tag, &
        MPI_COMM_WORLD, comm_request, rc)
    ! Tehdään muuta laskentaa, ei kosketa taulukkoon a.
    CALL MPI_WAIT(comm_request, status, rc)
  END IF

  CALL MPI_FINALIZE(rc)

END PROGRAM nb
```

5 Työkaluohjelmia

Tässä luvussa esitellään ohjelman suorituksen seuraamiseen tarkoitettu funktio MPI_WTIME sekä eräitä työkaluohjelmia viestinvälityksen visualisointiin ja optimointiin.

5.1 Rinnakkaisohjelman ajanmittaus

MPI-ohjelman eri osien käyttämän ajan mittaaminen on usein tärkeää, sillä rinnakkaisissa ohjelmissa pyritään mahdollisimman suureen laskentatehoon ja silloin on tärkeää saada selville ohjelman pullonkaulat. Ajanmittausta tarvitaan ohjelmien ajonaikaisen ajankäytön selvittämiseen

MPI:ssä on mukana ajanmittaukseen tarkoitettu rutiini MPI_WTIME, joka palauttaa ohjelman alusta käytetyn ajan *seinäkelloaikana*. Jotta MPI_WTIME-kutsu palauttaisi järkeviä aikoja, pitää laskentatehtävää ajaa yksin prosessorissa. Muuten toiset prosessit voivat vaikuttaa ajoaikaan ja tällöin seinäkelloaika ei kuvaa laskentatehtävän käyttämää aikaa oikein.

Taulukko 5.1: *MPI:n ajanmittausrutiinien käyttö C-kielestä ja Fortranista.*

C-kieli	*Fortran*
double **MPI_Wtime**(void)	DOUBLE PRECISION **MPI_WTIME**()
double **MPI_Wtick**(void)	DOUBLE PRECISION **MPI_WTICK**()

Rutiini MPI_WTIME palauttaa ajan sekunneissa ja rutiinin MPI_WTICK palauttaa ajastimen tarkkuuden samaten sekunneissa. Näiden rutiinien kutsusyntaksi on esitetty taulukossa 5.1.

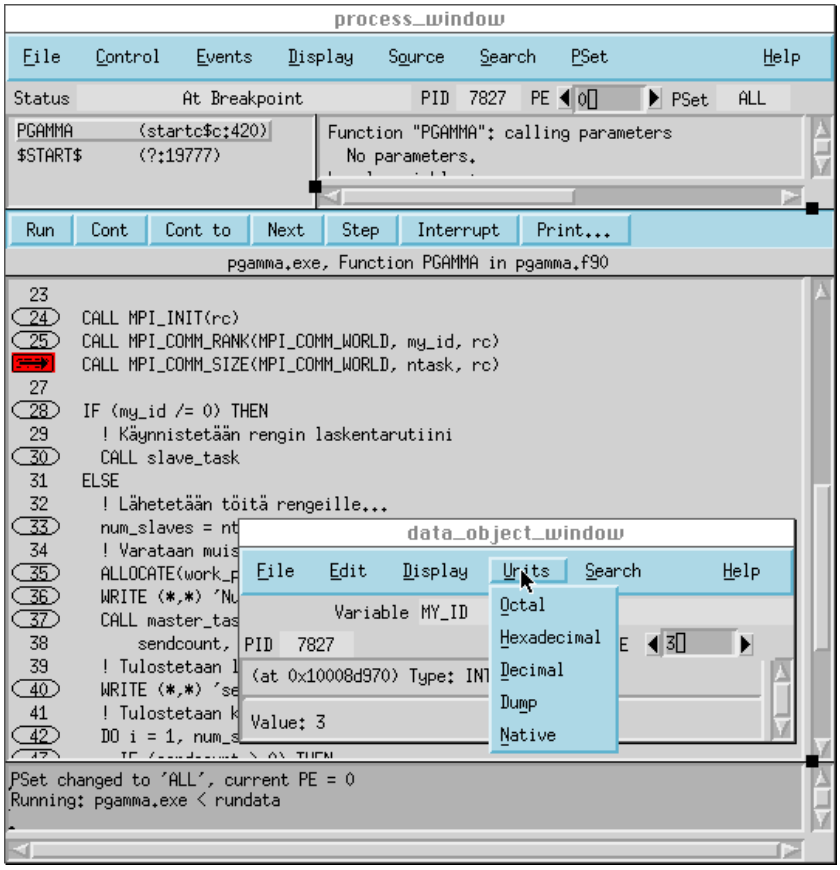
Rutiinia MPI_WTIME kannattaa käyttää siten, että rutiinia kutsutaan ennen mitattavaa ohjelmanpätkää ja otetaan aika talteen. Ohjelmanpätkän suorituksen jälkeen kutsutaan uudestaan rutiinia MPI_WTIME ja otetaan jälleen aika talteen. Näiden kahden ajanhetken erotus on kyseisen ohjelmapätkän käyttämä seinäkelloaika.

Rutiinin MPI_WTIME käytöstä löytyy esimerkki sivulta 66.

5.2 Muita työkaluja

TotalView on sekä peräkkäis- että rinnakkaisohjelmille soveltuva tutkintaohjelma. TotalView mahdollistaa ajonaikaisen ohjelmointivirheiden etsimisen MPI-ohjelmista. Ohjelmisto on saatavissa moneen eri koneympäristöön.

Kuvassa 5.1 on esitetty ohjelmiston käyttöliittymä. TotalView:n avulla voi mm. asettaa pysäytyskohtia haluttuihin laskentatehtäviin. Muuttujien arvoja voi seurata laskentatehtävittäin.



Kuva 5.1: *TotalView-ohjelmiston käyttö.*

MPI-ohjelmien viestinvälitystä on mahdollista seurata myös jälkikäteen. Esimerkiksi VAMPIR-niminen ohjelmisto mahdollista lokitiedostojen tulostamisen ohjelman ajon aikana. Ohjelman päätyttyä voi lokitiedostosta selvittää, millaisia viestejä prosessorit välittivät toisilleen kullakin ajanhetkellä. Tämä voi auttaa mm. viestinvälitykseen liittyvien pullonkaulojen poistamisessa.

5.3 Lisätietoja

Rinnakkaisohjelman vianetsintä ja optimointi voi olla hyvin aikaavievää puuhaa. Lisätietoa aiheesta löytyy mm. teoksesta *Parallel Programming with MPI* [Pac97] sekä CSC:n rinnakkaiskoneiden Cray T3E ja IBM SP käyttöoppaista [HS01, HKR01].

6 Esimerkkiohjelmia

Tässä luvussa käymme läpi muutamia laajempia esimerkkiohjelmia. Kussakin ohjelmassa esitellään eräitä tapoja toteuttaa rinnakkaisalgoritmeja MPI:llä.

6.1 Esimerkkien taustaa

Osa seuraavista esimerkeistä perustuu CSC:n PVM-oppaan [Saa95] esimerkkiohjelmiin, jotka on muunnettu MPI-ohjelmiksi. Kappaleessa 6.4 esitetty liittogradienttimenetelmän rinnakkaisversio on pääasiassa Jussi Raholan käsialaa.

Kaikissa Fortran 90/95 -esimerkkiohjelmissa käytetään seuraavaa `mpi`-moduulia:

```
        MODULE mpi
          IMPLICIT NONE
! Otetaan käyttöön MPI:n määrittelyt:
          INCLUDE 'mpif.h'
          INTEGER, SAVE ::
      &      ntask,      ! Laskentatehtävien lkm
      &      my_id,      ! Oma järjestysnumero
      &      rc          ! Paluukoodi
        END MODULE mpi
```

Tämä moduuli on useimmissa koneissa syytä määritellä käyttäen sarakesidonnaista (Fortran 77) syöttömuotoa.

Esimerkkiohjelmissa käytetään SPMD-ohjelmointimallia (Single Program, Multiple Data). Tällöin kaikki laskentatehtävät koostuvat samasta ohjelmakoodista ja tehtävän järjestysnumeron perusteella valitaan, mikä osa ohjelmakoodista suoritetaan kussakin tehtävässä.

6.2 Keskiarvon laskeminen

Tehtävänä on taulukon alkioiden keskiarvon laskeminen rinnakkaistetusti. Aluksi oletamme, että taulukon alkioiden määrä on jaollinen laskentatehtä-

vien lukumäärällä, ja tämän jälkeen esitämme yleisempään tapaukseen soveltuvan ohjelmakoodin.

6.2.1 Sama määrä laskentapisteitä

Kun kuhunkin laskentatehtävään välitetään sama määrä laskentapisteitä, tarvittava MPI-koodi on varsin yksinkertainen. Neljän laskentatehtävän tapausta on havainnollistettu kuvassa 6.1. Tässä tapauksessa taulukossa a on 12 alkiota.

a(1) a(2) a(3)	a(4) a(5) a(6)	a(7) a(8) a(9)	a(10) a(11) a(12)
P_0	P_1	P_2	P_3

Kuva 6.1: *Taulukon a alkioiden keskiarvon laskeminen.*

Seuraavassa ohjelmakoodissa generoidaan lähetettävä data laskentatehtävässä nolla taulukkoon a. Kukin laskentatehtävä ottaa vastaan 1000 alkiota, joten datan kokonaismäärä on $1000 \times p$, missä p on laskentatehtävien lukumäärä. Taulukko a alustetaan satunnaisluvuilla ennen datan lähettämistä.

```
PROGRAM average

  USE mpi  ! Tarvittavat MPI-määrittelyt

  IMPLICIT NONE

  INTEGER, PARAMETER :: &
      root_id = 0, &          ! Juurisolmun numero
      local_nr = 1000         ! Datapisteiden määrä tehtävässä
  REAL, DIMENSION(:), ALLOCATABLE :: &
      a                       ! Juurisolmun taulukko
  REAL, DIMENSION(local_nr) :: &
      aloc                    ! Paikallinen taulukko
  INTEGER :: &
      data_size, &            ! Globaalin datan määrä
      allocstat               ! Muistinvarauksen tilakoodi
  REAL :: psum, total_sum     ! Summatut arvot

  ! Alustetaan MPI-järjestelmä
  CALL MPI_INIT(rc)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntask, rc)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_id, rc)

  IF (my_id == root_id) THEN
    data_size = ntask*local_nr
    WRITE (*,*) 'Datapisteiden lkm on ', data_size
    ALLOCATE(a(data_size), STAT=allocstat)
    IF (allocstat /= 0) STOP 'Muistinvaraus epäonnistui!'
    CALL RANDOM_NUMBER(a)
  END IF

  ! Levitetään satunnaisluvut laskentatehtäville
  CALL MPI_SCATTER(a, local_nr, MPI_REAL, aloc, &
```

```
        local_nr, MPI_REAL, root_id, MPI_COMM_WORLD, rc)

    ! Lasketaan osasummat
    CALL compute(aloc, psum)

    ! Lasketaan globaali summa
    CALL MPI_REDUCE(psum, total_sum, 1, MPI_REAL, MPI_SUM, &
        root_id, MPI_COMM_WORLD, rc)

    ! Juuritehtävä tulostaa lopputuloksen
    IF (my_id == root_id) THEN
      WRITE (*,*) 'Keskiarvo on ', total_sum/data_size
    END IF

    ! Lopetetaan MPI:n käyttö
    CALL MPI_FINALIZE(rc)

  CONTAINS

    SUBROUTINE compute(array, summa)
      ! Lasketaan lukuarvojen summa
      IMPLICIT NONE
      REAL, DIMENSION(:), INTENT(IN) :: array
      REAL, INTENT(OUT) :: summa
      summa = SUM(array)
    END SUBROUTINE compute

  END PROGRAM average
```

Levitimme taulukon a alkiot laskentatehtäville käyttäen ryhmäviestintärutiinia MPI_SCATTER. Lopputuloksen laskimme rutiinin MPI_REDUCE avulla laskentatehtävään nolla.

6.2.2 Eri määrät datapisteitä

Edellisessä esimerkissä kukin laskentatehtävä vastaanottaa saman määrän taulukon a alkioita. Mikäli datan määrä ei ole jaollinen laskentatehtävien lukumäärillä, tarvitaan hiukan monimutkaisempi ohjelmakoodi.

Kukin laskentatehtävä saa oman osansa laskentapisteistä. Pisteiden määrä l_i lasketaan kaavasta

$$l_i = n/p + m, \quad i = 0, 1, \ldots, p-1,$$

missä n on datapisteiden kokonaismäärä ja p on laskentatehtävien lukumäärä. Arvo $m = 1$, jos $i < \mathrm{MOD}(n, p)$, missä i on laskentatehtävän järjestysnumero $(0, \ldots, p-1)$. Jakolasku n/p on katkaiseva kokonaislukujen jakolaskuoperaatio.

Kuvassa 6.2 on havainnollistettu datan jakamista, kun alkioiden määrä n on 11 alkiota ja laskentatehtävien lukumäärä p on neljä. Tässä tapauksessa tehtävä P_3 saa laskettavakseen kaksi alkiota ja muut saavat kolme.

a(1) a(2) a(3)	a(4) a(5) a(6)	a(7) a(8) a(9)	a(10) a(11)
P_0	P_1	P_2	P_3

Kuva 6.2: *Taulukon a alkioiden keskiarvon laskeminen, kun alkioita lähetetään eri määrä eri laskentatehtäville.*

Seuraavassa lähetetään aluksi datan kokonaismäärä `data_size` laskentatehtäville rutiinin MPI_BCAST avulla. Taulukon a alkiot levitetään laskentatehtäville käyttäen rutiinia MPI_SCATTERV, ja lopputulos lasketaan MPI_REDUCE-rutiinin avulla muuttujaan `total_sum`.

```
PROGRAM average2

  USE mpi ! Tarvittavat MPI-määrittelyt

  IMPLICIT NONE

  INTEGER, PARAMETER :: &
      root_id = 0            ! Juurisolmun numero
  REAL, DIMENSION(:), ALLOCATABLE :: &
      a, &                   ! Juurisolmun taulukko
      aloc                   ! Paikallinen taulukko
  INTEGER, DIMENSION(:), ALLOCATABLE :: &
      send_counts, &         ! Arvojen lukumäärät
      displs                 ! Arvojen alkukohdat
  INTEGER :: &
      data_size, &           ! Globaalin datan määrä
      my_num, &              ! Omien datapisteiden lkm
      allocstat, &           ! Muistinvarauksen tilakoodi
      i, &                   ! Silmukkamuuttuja
      local_nr, ival_mod     ! Apumuuttujia
  REAL :: psum, total_sum    ! Summatut arvot

  ! Alustetaan MPI-järjestelmä
  CALL MPI_INIT(rc)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntask, rc)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_id, rc)

  IF (my_id == root_id) THEN
    WRITE (*,*) 'Syötä datapisteiden lkm: '
    READ (*,*) data_size
    IF (data_size <= 0) STOP 'Datapisteiden lkm <= 0!'
  END IF
  ! Lähetä datapisteiden lkm kaikille
  CALL MPI_BCAST(data_size, 1, MPI_INTEGER, 0, &
      MPI_COMM_WORLD, rc)

  ! Lasketaan omien datapisteiden lkm
  local_nr = data_size/ntask
  ival_mod = MOD(data_size,ntask)
  IF (my_id < ival_mod) THEN
    my_num = local_nr+1
  ELSE
    my_num = local_nr
  END IF
```

```
  ALLOCATE(aloc(my_num), STAT=allocstat)
  IF (allocstat /= 0) STOP 'Muistinvaraus epäonnistui!'

  WRITE (*,*) 'Tehtävän nro ja pisteiden lkm:', my_id, my_num

  IF (my_id == root_id) THEN
    ALLOCATE(send_counts(0:ntask-1), displs(0:ntask-1), &
        STAT=allocstat)
    IF (allocstat /= 0) STOP 'Muistinvaraus epäonnistui!'
    send_counts(0:ival_mod-1) = local_nr+1
    send_counts(ival_mod:) = local_nr
    displs(0) = 0
    DO i = 0, ntask-2
      displs(i+1) = displs(i) + send_counts(i)
    END DO
    WRITE (*,*) 'Datapisteiden lkm: ', send_counts
    ALLOCATE(a(data_size), STAT=allocstat)
    IF (allocstat /= 0) STOP 'Muistinvaraus epäonnistui!'
    CALL RANDOM_NUMBER(a)
  END IF

  ! Levitetään satunnaisluvut laskentatehtäville
  CALL MPI_SCATTERV(a, send_counts, displs, MPI_REAL, &
      aloc, my_num, MPI_REAL, root_id, MPI_COMM_WORLD, rc)

  ! Lasketaan osasummat
  CALL compute(aloc, psum)

  ! Lasketaan globaali summa
  CALL MPI_REDUCE(psum, total_sum, 1, MPI_REAL, MPI_SUM, &
      root_id, MPI_COMM_WORLD, rc)

  ! Juurinoodi tulostaa lopputuloksen
  IF (my_id == root_id) THEN
    WRITE (*,*) 'Keskiarvo on ', total_sum/data_size
  END IF

  ! Lopetetaan MPI:n käyttö
  CALL MPI_FINALIZE(rc)

CONTAINS
```

Edellä esitetty aliohjelma `compute`.

```
END PROGRAM average2
```

Ohjelmakoodissa käytettiin dynaamista muistinvarausta (`ALLOCATE`) sekä paikallisille taulukoille (`aloc`) että lähetettävälle datalle (`a`). Datapisteiden lukumäärä luettiin käyttäjän antamasta syötteestä.

Seuraavassa on ohjelman käyttöesimerkki:

```
 Syötä datapisteiden lkm:
11013
 Tehtävän nro ja pisteiden lkm: 0,  1377
 Datapisteiden lkm:  5*1377,  3*1376
 Tehtävän nro ja pisteiden lkm: 4,  1377
 Tehtävän nro ja pisteiden lkm: 2,  1377
```

```
Tehtävän nro ja pisteiden lkm: 1,  1377
Tehtävän nro ja pisteiden lkm: 5,  1376
Tehtävän nro ja pisteiden lkm: 6,  1376
Tehtävän nro ja pisteiden lkm: 3,  1377
Tehtävän nro ja pisteiden lkm: 7,  1376
Keskiarvo on  0.50007123822091004
```

Tässä tapauksessa laskentatehtävät $0,\ldots,4$ saivat 1377 alkiota ja loput saivat 1376 alkiota.

6.3 Kuormantasaus

Isäntä-renki -ohjelmointimallissa (master-slave) yksi laskentatehtävistä koordinoi laskentaa ja jakaa datan muille laskentatehtäville.

Tässä esimerkissä lasketaan gammafunktion käänteisluvun $1/\Gamma$ arvoja kompleksitason eri pisteissä. Gammafunktio voidaan määritellä pisteessä $z = x + iy$ seuraavasti:

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt, \quad \mathrm{Re}(z) > 0.$$

Euler on johtanut gammafunktion käänteisarvolle laskukaavan

$$\frac{1}{\Gamma(z)} = z e^{\gamma z} \prod_{n=1}^{\infty} \left(\left(1 + \frac{z}{n}\right) e^{-\frac{z}{n}} \right).$$

Ottamalla kummaltakin puolelta luonnollinen logaritmi saadaan laskukaava summamuotoon:

$$\ln \frac{1}{\Gamma(z)} = \ln z + \gamma z + \sum_{n=1}^{\infty} \left(\ln\left(1 + \frac{z}{n}\right) - \frac{z}{n} \right). \tag{6.1}$$

Tässä vakio γ on Eulerin vakio, jonka voi laskea seuraavasta raja-arvolausekkeesta:

$$\gamma = \lim_{m \to \infty} \left(\sum_{i=1}^{\infty} \frac{1}{m} - \ln m \right) \approx 0.5772156649015329.$$

Esimerkkiohjelmassa käytetään tätä likiarvoa vakion γ arvona.

Esimerkkiohjelmassa lasketaan gammafunktion käänteisarvoja suorakulmaisessa hilassa $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. Ylä- ja alarajat eli arvot

$$\{x_{\min}, x_{\max}, y_{\min}, y_{\max}\}$$

sekä pisteiden lukumäärät n_x ja n_y annetaan syötteenä ohjelmalle. Lisäksi ohjelmalle syötetään toleranssiarvo, joka kertoo kuinka tarkkoja arvioita ohjelman on laskettava.

Funktion $1/\Gamma(z)$ arvojen laskemiseen voidaan tarvita eri määrä laskuoperaatioita eri pisteissä. Tämän vuoksi tarvitaan kuormantasausta: nopeimmin laskutoimituksista selviävät tehtävät saavat laskettavakseen lisää pisteitä, kun taas hitaimmin valmistuvat laskentatehtävät saavat vastaavasti vähemmän laskettavia pisteitä.

Ohjelmassa käytetään kahdenvälistä viestintää eli rutiineja MPI_SEND ja MPI_RECV. Kuormantasausalgoritmi voidaan toteuttaa seuraavasti käyttäen Fortran 90:n moduuleja:

```
MODULE loadbal
  IMPLICIT NONE

  INTEGER, PARAMETER :: &
      prec = SELECTED_REAL_KIND(10,100) ! Laskentatarkkuus

  INTEGER, PARAMETER, PRIVATE :: &
      buff_message_tag = 50, &
      zbuff_message_tag = 60, &
      nterms_message_tag = 70, &
      task_code_message_tag = 80

  INTEGER, PARAMETER, PRIVATE :: &
      do_task_code = 0, &
      stop_task_code = -1

  CHARACTER(LEN=*), PARAMETER, PRIVATE :: &
      format_str_0 = '(1x,i4,i9,1p,5e13.5)', &
      format_str_1 = '(1x,a4,a9,5a13)', &
      format_str_2 = '(1x,1p,e13.5,2(2g13.5,i4))', &
      format_str_3 = '(1x,a,z10,a)'
CONTAINS
```

Aliohjelmien master_task ja slave_task sekä funktion invgamma määrittelyt.

```
END MODULE loadbal
```

Aliohjelma master_task hoitaa tehtävän jakamisen rengeille:

```
SUBROUTINE master_task(num_slaves, work_piece, &
    nterms_per_cpu, sendcount, recvcount)
  USE mpi

  IMPLICIT NONE

  INTEGER, INTENT(IN) :: num_slaves
  INTEGER, DIMENSION(:), INTENT(OUT) :: work_piece
  REAL(KIND=prec), DIMENSION(:), INTENT(OUT) :: nterms_per_cpu
  INTEGER, INTENT(OUT) :: sendcount, recvcount
  REAL(KIND=prec) :: x, y, eps, xmin, xmax, &
      xdelta, ymin, ymax, ydelta
  COMPLEX(KIND=prec) :: z, res
  INTEGER :: nx, ny, i, j, k, nterms, rc, dest_id, source_id

  nterms_per_cpu(:) = 0
  work_piece(:) = 0
```

```
    WRITE (*,*) 'toleranssi, xmin, xmax, nx, ymin, ymax, ny ?'
    READ (*,*) eps, xmin, xmax, nx, ymin, ymax, ny
    WRITE (*,format_str_2) eps, xmin, xmax, nx, ymin, ymax, ny

    nx = nx - 1
    xdelta = 0
    IF (nx > 0) xdelta = (xmax-xmin)/nx
    ny = ny - 1
    ydelta = 0
    IF (ny > 0) ydelta = (ymax-ymin)/ny

    sendcount = 0
    recvcount = 0
    IF ((nx+1)*(ny+1) < num_slaves) THEN
      WRITE (*,*) 'virhe: pisteitä vähemmän kuin renkitehtäviä'
    ELSE
      WRITE (*,format_str_1) 'cpu#', 'nterms', 'x', 'y', &
          're(fun)', 'im(fun)', 'abs(fun)'
      DO j = 0, ny
        y = ymin + j*ydelta
        DO i = 0, nx
          x = xmin + i*xdelta
          IF (sendcount >= num_slaves) THEN
            ! Vastaanotetaan tuloksia rengeiltä.
            ! Rengin järjestysnro selviää muuttujasta source_id
            CALL receive_data(z, res, recvcount, &
                nterms_per_cpu, source_id)
            ! Renki nro source_id saa seuraavaksi dataa
            dest_id = source_id
          ELSE
            dest_id = sendcount + 1
          END IF
          CALL send_data(dest_id, x, y, eps, sendcount, &
              work_piece)
        END DO
      END DO
      DO k = 1, num_slaves
        ! Kerätään jäljellä olevat tulokset
        CALL receive_data(z, res, recvcount, &
            nterms_per_cpu, source_id)
      END DO
    END IF

    ! Lähetetään lopetusviestit rengeille
    DO k = 1, num_slaves
      CALL MPI_SEND(stop_task_code, 1, MPI_INTEGER, k, &
         task_code_message_tag, MPI_COMM_WORLD, rc)
    END DO
  CONTAINS
```

Aliohjelmien `send_data` ja `receive_data` määrittelyt.

```
  END SUBROUTINE master_task
```

Varsinainen datan lähetys ja vastaanotto tehdään rutiineissa `send_data` ja `receive_data`:

```
SUBROUTINE send_data(dest_id, x, y, eps, sendcount, &
    work_piece)
  IMPLICIT NONE

  INTEGER, INTENT(IN) :: dest_id
  REAL(KIND=prec), INTENT(IN) :: x, y, eps
  INTEGER, INTENT(INOUT) :: sendcount
  INTEGER, DIMENSION(:), INTENT(INOUT) :: work_piece
  REAL(KIND=prec), DIMENSION(3) :: buff

  ! Kerrotaan että tulossa on laskettavaa dataa
  CALL MPI_SEND(do_task_code, 1, MPI_INTEGER, dest_id, &
     task_code_message_tag, MPI_COMM_WORLD, rc)
  ! Lähetetään dataa
  buff = (/ x, y, eps /)
  CALL MPI_SEND(buff, 3, MPI_REAL, dest_id, &
      buff_message_tag, MPI_COMM_WORLD, rc)
  work_piece(dest_id) = work_piece(dest_id) + 1
  sendcount = sendcount + 1
END SUBROUTINE send_data
```

Tässä lähetimme dataa laskentatehtävälle numero `dest_id`. Seuraavassa vastaanotetaan tuloksia rengiltä, jonka järjestysnumero on `source_id`:

```
SUBROUTINE receive_data(z, res, recvcount, &
    nterms_per_cpu, source_id)
  IMPLICIT NONE

  COMPLEX(KIND=prec), INTENT(OUT) :: z, res
  INTEGER, INTENT(INOUT) :: recvcount
  REAL(KIND=prec), DIMENSION(:), INTENT(INOUT) :: &
      nterms_per_cpu
  INTEGER, INTENT(OUT) :: source_id
  COMPLEX(KIND=prec), DIMENSION(2) :: zbuff
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
  INTEGER :: nterms

  ! Vastaanotetaan miltä tahansa rengiltä
  CALL MPI_RECV(zbuff, 2, MPI_COMPLEX, MPI_ANY_SOURCE, &
      zbuff_message_tag, MPI_COMM_WORLD, status, rc)
  z = zbuff(1); res = zbuff(2)
  CALL MPI_RECV(nterms, 1, MPI_INTEGER, MPI_ANY_SOURCE, &
      nterms_message_tag, MPI_COMM_WORLD, status, rc)
  recvcount = recvcount + 1
  ! Selvitetään rengin järjestysnro
  source_id = status(MPI_SOURCE)
  nterms_per_cpu(source_id) = &
      nterms_per_cpu(source_id) + nterms
  WRITE (*,format_str_0) source_id, nterms, z, res, ABS(res)
END SUBROUTINE receive_data
```

Renkitehtävä on isäntää yksinkertaisempi toteuttaa. Seuraava koodi hakee dataa isännältä ja lähettää vastaukset takaisin:

```
SUBROUTINE slave_task
  USE mpi
```

```
  IMPLICIT NONE

  REAL(KIND=prec) :: x, y, eps
  REAL(KIND=prec), DIMENSION(3) :: buff
  COMPLEX(KIND=prec) :: z, res
  COMPLEX(KIND=prec), DIMENSION(2) :: zbuff
  INTEGER :: master_id, task_code, nterms, rc
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

  master_id = 0
  DO
    ! Tarkistetaan tulevan viestin tyyppi
    CALL MPI_RECV(task_code, 1, MPI_INTEGER, master_id, &
        task_code_message_tag, MPI_COMM_WORLD, status, rc)
    ! Tarkistetaan onko tulossa dataa
    IF (task_code == stop_task_code) EXIT
    ! Vastaanotetaan data
    CALL MPI_RECV(buff, 3, MPI_DOUBLE_PRECISION, master_id, &
        buff_message_tag, MPI_COMM_WORLD, status, rc)
    x = buff(1); y = buff(2); eps = buff(3)
    z = CMPLX(x,y)
    ! Lasketaan funktion arvo
    res = invgamma(z,eps,nterms)
    ! Palautetaan tulokset
    zbuff = (/ z, res /)
    CALL MPI_SEND(zbuff, 2, MPI_COMPLEX, master_id, &
        zbuff_message_tag, MPI_COMM_WORLD, rc)
    CALL MPI_SEND(nterms, 1, MPI_INTEGER, master_id, &
        nterms_message_tag, MPI_COMM_WORLD, rc)
  END DO
END SUBROUTINE slave_task
```

Varsinainen työ tehdään `DO`-silmukassa, jonka toistaminen lopetetaan, kun isäntä lähettää lopetuskoodin. Gammafunktion käänteisarvot lasketaan funktiokutsulla `invgamma`. Funktion `invgamma` voi määritellä esimerkiksi seuraavalla tavalla:

```
FUNCTION invgamma(zinput,eps,nterms)
  IMPLICIT NONE

  COMPLEX(KIND=prec), INTENT(IN) :: zinput
  REAL(KIND=prec), INTENT(IN) :: eps
  INTEGER, INTENT(OUT) :: nterms
  COMPLEX(KIND=prec) :: invgamma
  INTEGER, PARAMETER :: ninf = 1000000000
  REAL(KIND=prec), PARAMETER :: &
      g = 0.5772156649015329_prec, &
      smalleps = 100*EPSILON(g)
  COMPLEX(KIND=prec) :: zz, zn, fun, term
  REAL(KIND=prec) :: magn, oldmagn
  INTEGER :: n, iz

  zz = zinput
  iz = zz
  IF (zz == iz) zz = zz + smalleps
```

```
    term = LOG(zz) + g*zz
    fun = EXP(term)
    magn = ABS(fun)
    loop: DO n = 1, ninf
      oldmagn = magn
      zn = zz/n
      term = term + LOG(1 + zn) - zn
      fun = EXP(term)
      magn = ABS(fun)
      IF (ABS(magn - oldmagn) <= eps) THEN
        nterms = n
        invgamma = fun
        EXIT loop
      END IF
    END DO loop
  END FUNCTION invgamma
```

Funktio perustuu yhtälössä (6.1) esitettyyn laskukaavaan. Tämä ei ole optimaalinen tapa laskea gammafunktion käänteisarvo!

Pääohjelma voisi näyttää seuraavalta:

```
  PROGRAM pgamma
    USE mpi, rc => ierror
    ! Otetaan käyttöön laskentarutiinit
    USE loadbal, ONLY : prec, master_task, slave_task

    IMPLICIT NONE

    ! Tulostusformaatti
    CHARACTER(LEN=*), PARAMETER :: &
        format_str = ‘‘(1x,’cpu#’,i3,’: nterms=’,f10.0, &
        & ’ work pieces=’,i6,’. average=’,f10.2)’’

    INTEGER, DIMENSION(:), ALLOCATABLE :: &
        work_piece   ! Taulukko tulosten talletusta varten
    REAL(KIND=prec), DIMENSION(:), ALLOCATABLE :: &
        nterms_per_cpu ! Kunkin rengin laskemien tulosten lkm
    INTEGER :: &
        num_slaves, &! Renkien lkm
        sendcount, & ! Viestien lkm
        recvcount, & ! Vastausten lkm
        i            ! Silmukkamuuttuja

    CALL MPI_INIT(rc)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_id, rc)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntask, rc)

    IF (my_id /= 0) THEN
      ! Käynnistetään rengin laskentarutiini
      CALL slave_task
    ELSE
      ! Lähetetään töitä rengeille...
      num_slaves = ntask - 1
      ! Varataan muistia kirjanpidolle
      ALLOCATE(work_piece(num_slaves), nterms_per_cpu(num_slaves))
```

```
      WRITE (*,*) 'Renkien lukumäärä', num_slaves
      CALL master_task(num_slaves, work_piece, nterms_per_cpu, &
          sendcount, recvcount)
      ! Tulostetaan lähetettyjen viestin lukumäärät
      WRITE (*,*) 'Lähetetyt ja vastaanotetut viestit:', &
          sendcount, recvcount
      ! Tulostetaan kunkin rengin tekemän työn määrä
      DO i = 1, num_slaves
        IF (sendcount > 0) THEN
          WRITE (*,format_str) i, nterms_per_cpu(i), &
              work_piece(i), nterms_per_cpu(i)/work_piece(i)
        END IF
      END DO
    END IF

    CALL MPI_FINALIZE(rc)
  END PROGRAM pgamma
```

Ohjelmassa käynnistetään rengit aliohjelmakutsulla `slave_task`. Isäntätehtävässä varataan puolestaan tilaa kirjanpitoa varten, jolloin ohjelman lopussa voidaan tulostaa yhteenveto lähetetyistä ja vastaanotetuista viesteistä.

Ohjelman tulostus näyttää esimerkiksi seuraavalta:

```
 Renkien lukumäärä 7
 toleranssi, xmin, xmax, nx, ymin, ymax, ny ?
1e-4 -1 1 10 -1 1 10
   1.00000E-04  -1.0000       1.0000      10  -1.0000       1.0000      10
 cpu#   nterms            x            y       re(fun)      im(fun)     abs(fun)
    1       27 -1.00000E+00 -1.00000E+00 -1.34798E+00  2.35125E+00  2.71025E+00
    2       71 -7.77778E-01 -1.00000E+00 -2.09270E+00  1.17634E+00  2.40066E+00
    3       88 -5.55556E-01 -1.00000E+00 -2.18171E+00 -5.73822E-02  2.18246E+00
   ...
    7       81  5.55556E-01  1.00000E+00  1.22718E+00  1.47424E+00  1.91816E+00
    1       61  7.77778E-01  1.00000E+00  1.62128E+00  1.04961E+00  1.93139E+00
    2       23  1.00000E+00  1.00000E+00  1.80629E+00  6.46247E-01  1.91842E+00
 Lähetetyt ja vastaanotetut viestit: 2*100
 cpu#  1: nterms=      626. work pieces=    15. average=     41.73
 cpu#  2: nterms=      604. work pieces=    15. average=     40.27
 cpu#  3: nterms=      586. work pieces=    14. average=     41.86
 cpu#  4: nterms=      633. work pieces=    14. average=     45.21
 cpu#  5: nterms=      598. work pieces=    14. average=     42.71
 cpu#  6: nterms=      514. work pieces=    14. average=     36.71
 cpu#  7: nterms=      587. work pieces=    14. average=     41.93
```

Tässä annettiin tarkkuusvaatimukseksi arvo 10^{-4}. Gammafunktion käänteisarvoja laskettiin alueella $[-1,1] \times [-1,1]$, ja laskentahilan koko oli 10×10 pistettä.

Tätä ohjelmaa voisi tehostaa huomattavasti mm. vähentämällä lähetettävien viestien lukumäärää. Kerralla voisi esimerkiksi lähettää laskettavaksi isomman joukon pisteitä, jolloin viestinnän määrä putoaisi vastaavasti. Toisaalta voi käydä niin, että jokin laskentatehtävä saa laskettavakseen joukon työläimpiä arvoja. Tällöin ohjelman kuormantasaus huononee.

6.4 Yhtälöryhmän ratkaisu liittogradienttimenetelmällä

Tehtävänä on ratkaista lineaarinen yhtälöryhmä

$$A\mathbf{x} = \mathbf{b}, \tag{6.2}$$

kun matriisi A on tiheä (vähän nollasta poikkeavia alkioita). Yhtälöryhmä (6.2) voidaan ratkaista iteratiivisesti käyttämällä vaikkapa liittogradienttimenetelmää (conjugate gradient method).

Seuraavassa on kuvaus yksinkertaisesta liittogradienttimenetelmästä:

$$\begin{aligned}
&\mathbf{r}^0 = \mathbf{b} - A\mathbf{x}^0 \\
&\mathbf{p}^0 = \mathbf{r}^0 \\
&k = 0 \\
&\textbf{DO} \\
&\qquad \begin{aligned}
\alpha_k &= \frac{\langle \mathbf{r}^k, \mathbf{r}^k \rangle}{\langle \mathbf{p}^k, A\mathbf{p}^k \rangle} \\
\mathbf{x}^{k+1} &= \mathbf{x}^k + \alpha_k \mathbf{p}^k \\
\mathbf{r}^{k+1} &= \mathbf{r}^k - \alpha_k A\mathbf{p}^k \\
\beta_k &= \frac{\langle \mathbf{r}^{k+1}, \mathbf{r}^{k+1} \rangle}{\langle \mathbf{r}^k, \mathbf{r}^k \rangle} \\
\mathbf{p}^{k+1} &= \mathbf{r}^{k+1} + \beta_k \mathbf{p}^k \\
k &= k+1
\end{aligned} \\
&\textbf{UNTIL } \|\mathbf{r}^k\| < \epsilon
\end{aligned}$$

Algoritmin lopetusehdossa vaaditaan, että *residuaalin* $\mathbf{r}^k = \mathbf{b} - A\mathbf{x}^k$ normi on tarpeeksi pieni.

Esimerkkikoodiemme laskentatarkkuus määritellään seuraavasti moduulissa `cg_precision`:

```
MODULE cg_precision
  ! Määritellään liukulukujen tarkkuus
  IMPLICIT NONE
  INTEGER, PARAMETER :: &
      prec = SELECTED_REAL_KIND(12,50)
END MODULE cg_precision
```

Tätä moduulia käytetään puolestaan muissa moduuleissa. Varsinainen liittogradienttialgoritmi on toteutettu moduulissa `cgalg`:

```
MODULE cgalg

  USE mpi
  USE cg_precision, ONLY : prec

  IMPLICIT NONE

CONTAINS
```

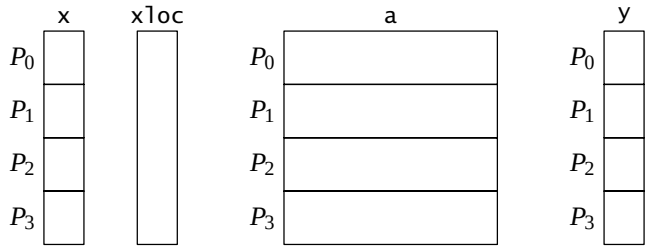
```
SUBROUTINE cg(n, nstart, nend, b, x, itmax, eps, matr_mul)
  ! Aliohjelma ratkaisee lineaarisen yhtälöryhmän Ax=b
  ! liittogradienttialgoritmilla (CG).
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: &
      n, &                ! Yhtälöryhmän dimensio
      nstart, nend        ! Oman vektorin alku ja loppu
  REAL(KIND=prec), DIMENSION(nstart:nend), &
      INTENT(IN) :: b     ! Oikean puolen vektori
  REAL(KIND=prec), DIMENSION(nstart:nend), &
      INTENT(INOUT) :: x  ! Alkuarvaus ja ratkaisu
  INTEGER, INTENT(IN) :: itmax  ! Iteraatioiden lukumäärä
  REAL(KIND=prec), INTENT(IN) :: eps  ! Suppenemiskriteeri
  INTERFACE
    SUBROUTINE matr_mul(n, nstart, nend, x, y)
      ! Aliohjelma laskee matriisi-vektoritulon y=A*x
      USE cg_precision, ONLY : prec
      INTEGER, INTENT(IN) :: n, nstart, nend
      REAL(KIND=prec), DIMENSION(nstart:nend), &
          INTENT(IN) :: x
      REAL(KIND=prec), DIMENSION(nstart:nend), &
          INTENT(OUT) :: y
    END SUBROUTINE matr_mul
  END INTERFACE
  REAL(KIND=prec) :: &
      alpha, beta, & ! Askelpituudet
      rnorm, &       ! Residuaalin normin neliö
      rnnorm, &      ! Uuden residuaalin normin neliö
      rnorm0, &      ! Alkuresiduaalin normin neliö
      temp, &        ! Tilapäismuuttuja
      pap            ! Arvo <p, A*p>
  REAL(KIND=prec), DIMENSION(nstart:nend) :: &
      p, &           ! Etsintäsuunta
      ap, &          ! A*p
      res            ! Residuaali
  INTEGER :: k       ! Iteraatiokierros
  INTEGER :: nloc    ! Paikallisen vektorin pituus

  ! Laske alkuresiduaali b-Ax
  nloc = nend - nstart + 1
  CALL matr_mul(n, nstart, nend, x, res)
  res = b - res
  p = res ! Alkuresiduaali etsintäsuunnaksi
  k = 1
  temp = DOT_PRODUCT(res,res)
  CALL MPI_ALLREDUCE(temp, rnorm, 1, MPI_REAL, MPI_SUM, &
      MPI_COMM_WORLD, rc)
  rnorm0 = rnorm ! Alkuresiduaalin normin neliö
  DO WHILE(k < itmax .AND. SQRT(rnorm)/SQRT(rnorm0) > eps)
    CALL matr_mul(n, nstart, nend, p, ap)
    temp = DOT_PRODUCT(p,ap)
    CALL MPI_ALLREDUCE(temp, pap, 1, MPI_REAL, MPI_SUM, &
        MPI_COMM_WORLD, rc)
    alpha = rnorm / pap
    x = x + alpha * p      ! Päivitä iteraattia
    res = res - alpha * ap ! Päivitä residuaalia
```

```
        temp = DOT_PRODUCT(res,res)
        CALL MPI_ALLREDUCE(temp, rnnorm, 1, MPI_REAL, MPI_SUM, &
            MPI_COMM_WORLD, rc)
        beta = rnnorm / rnorm
        p = res + beta * p     ! Päivitä etsintäsuuntaa
        rnorm = rnnorm
        IF (my_id == 0) WRITE (*,*) k, SQRT(rnorm)
        k = k + 1
      END DO
      IF (my_id == 0) WRITE (*,'(/,"CG suppeni ",&
          &i3," iteraatiossa",/)') k - 1
    END SUBROUTINE cg

  END MODULE cgalg
```

Liittogradienttialgoritmissa tarvitaan matriisi-vektoritulon laskeva aliohjelma. Moduulissa `cgalg` määriteltyyn aliohjelmaan `cg` välitetään tämä aliohjelma nimellä `matr_mult`.



Kuva 6.3: *Matriisin jakaminen laskentatehtäville riveittäin.*

Kuvassa 6.3 on esitetty eräs tapa toteuttaa rinnakkaistettu matriisi-vektoritulo. Tässä versiossa kukin laskentatehtävä saa ”omakseen” joukon matriisin A päällekkäisiä rivejä. Seuraavassa moduulissa on toteutettu matriisitulon tällä tavalla laskeva rutiini `matrix_mult`. Moduulissa `matrix` on lisäksi aliohjelma `matrix_init` matriisin alustamiseen rinnakkaistetusti.

```
  MODULE matrix

    USE mpi
    USE cg_precision, ONLY : prec

    IMPLICIT NONE

    PUBLIC :: matrix_init, matrix_mult
    PRIVATE

    REAL(KIND=prec), DIMENSION(:,:), &
        ALLOCATABLE, SAVE :: coeffmat

  CONTAINS

    SUBROUTINE matrix_init(n, nstart, nend)
      ! Aliohjelma alustaa kerroinmatriisin
      INTEGER, INTENT(IN) :: n, nstart, nend
```

```
    INTEGER :: allocstat, i, j

    ALLOCATE(coeffmat(nstart:nend, n), STAT=allocstat)
    IF (allocstat /= 0) STOP 'muistinvaraus epäonnistui!'

    DO j = 1, n
       DO i = nstart, nend
          coeffmat(i,j) = 1/(1.0_prec+ABS(i-j))
       END DO
    END DO
  END SUBROUTINE matrix_init

  SUBROUTINE matrix_mult(n, nstart, nend, x, y)
    ! Lasketaan matriisi-vektoritulo y=A*x
    INTEGER, INTENT(IN) :: n, nstart, nend
    REAL(KIND=prec), DIMENSION(nstart:nend), INTENT(IN) :: x
    REAL(KIND=prec), DIMENSION(nstart:nend), INTENT(OUT) :: y
    REAL(KIND=prec), DIMENSION(n) :: xloc
    INTEGER :: vlen

    vlen = nend - nstart + 1

    CALL MPI_ALLGATHER(x, vlen, MPI_REAL, xloc, vlen, &
        MPI_REAL, MPI_COMM_WORLD, rc)
    y = MATMUL(coeffmat,xloc)
  END SUBROUTINE matrix_mult

END MODULE matrix
```

Moduulien `cgalg` ja `matrix` avulla voimmekin nyt toteuttaa rinnakkaistetun liittogradienttimenetelmän. Tarvittava pääohjelma on esitetty seuraavassa:

```
PROGRAM iterate
  USE mpi
  USE cg_precision, ONLY : prec
  USE cgalg, ONLY : cg
  USE matrix, ONLY : matrix_init, matrix_mult

  IMPLICIT NONE

  REAL(KIND=prec), DIMENSION(:), ALLOCATABLE :: rhs, x
  INTEGER :: n, vlen, nstart, nend, allocstat
  REAL(KIND=prec) :: t0, t1, toler = 1e-5

  CALL MPI_INIT(rc)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntask, rc)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_id, rc)

  IF (my_id == 0) THEN
    WRITE (*,*) 'Anna matriisin koko n'
    READ (*,*) n
  END IF

  t0 = MPI_WTIME()

  CALL MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, rc)
```

```
  IF (MODULO(n, ntask) /= 0) THEN
    IF (my_id == 0) WRITE (*,*) 'Virhe: n ei ole jaollinen &
        &prosessorien lukumäärällä'
    CALL MPI_FINALIZE(rc)
    STOP
  END IF
  vlen = n/ntask
  nstart = 1 + my_id*vlen
  nend = (my_id + 1)*vlen
  ALLOCATE (rhs(nstart:nend), x(nstart:nend), STAT=allocstat)
  IF (allocstat /= 0) STOP 'muistinvaraus epäonnistui!'
  rhs = 1; x = 0

  CALL matrix_init(n, nstart, nend)
  CALL cg(n, nstart, nend, rhs, x, 100, toler, matrix_mult)

  t1 = MPI_WTIME()
  IF (my_id == 0) WRITE (*,*) 'Suoritusaika: ', t1 - t0

  CALL MPI_FINALIZE(rc)
END PROGRAM iterate
```

Seuraavassa on ohjelman ajoesimerkki kahdeksalla Cray T3E -rinnakkaiskoneen prosessorilla:

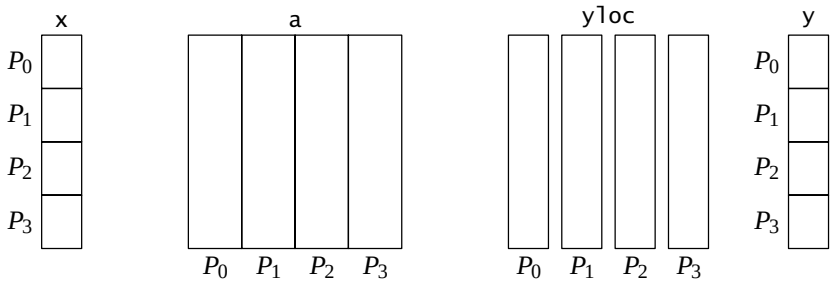
```
 Anna matriisin koko n
4000
 1,  3.5441064507725124
 2,  0.8786666230856649
 3,  0.39260895383962724
 4,  0.22371697755487119
 5,  0.14036037346798103
 6,  9.1621175058730922E-2
 7,  6.07628842315540083E-2
 8,  4.05105344626063152E-2
 9,  2.70204583072113612E-2
 10,  1.79917624832130855E-2
 11,  1.19483478294735296E-2
 12,  7.91115076460293352E-3
 13,  5.22186939459730153E-3
 14,  3.43615763756852056E-3
 15,  2.2542773613242334E-3
 16,  1.47456003705305895E-3
 17,  9.61775018421270585E-4
 18,  6.25568087407620595E-4

CG suppeni  18 iteraatiossa

 Suoritusaika:  1.8440357148647308
```

Tässä matriisin A koko oli 4000×4000. Ratkaisuun tarvittiin 18 liittogradientti-iteraatiota. Ratkaisuun kului seinäkelloaikaa 1.8 sekuntia.

Seuraavassa on esitetty matriisin alustusrutiini sekä matriisi-vektoritulon toteutus siinä tapauksessa, että matriisi halutaan jakaa prosessoreille sarakkeit-



Kuva 6.4: *Matriisin jakaminen laskentatehtäville sarakkeittain.*

tain (katso kuvaa 6.4):

```
SUBROUTINE matrix_init(n, nstart, nend)
  ! Aliohjelma alustaa kerroinmatriisin
  INTEGER, INTENT(IN) :: n, nstart, nend
  INTEGER :: allocstat, i, j

  ALLOCATE(coeffmat(n, nstart:nend), STAT=allocstat)
  IF (allocstat /= 0) STOP 'muistinvaraus epäonnistui!'

  DO j = nstart, nend
    DO i = 1, n
      coeffmat(i,j) = 1/(1.0_prec+ABS(i-j))
    END DO
  END DO
END SUBROUTINE matrix_init

SUBROUTINE matrix_mult(n, nstart, nend, x, y)
  ! Lasketaan matriisi-vektoritulo y=A*x
  INTEGER, INTENT(IN) :: n, nstart, nend
  REAL(KIND=prec), DIMENSION(nstart:nend), INTENT(IN) :: x
  REAL(KIND=prec), DIMENSION(nstart:nend), INTENT(OUT) :: y
  REAL(KIND=prec), DIMENSION(n) :: yloc
  INTEGER, DIMENSION(ntask) :: recvlen

  recvlen = nend - nstart + 1
  yloc = MATMUL(coeffmat, x)
  CALL MPI_REDUCE_SCATTER(yloc, y, recvlen, MPI_REAL, &
      MPI_SUM, MPI_COMM_WORLD, rc)
END SUBROUTINE matrix_mult
```

Rutiini MPI_REDUCE_SCATTER laskee vektorien `yloc` alkioittaisen summan sekä jakaa tulosvektorin laskentatehtäville. Tulos on sama kuin peräkkäin suoritetuilla aliohjelmakutsuilla MPI_REDUCE ja MPI_SCATTER.

Edellä on laskettu matriisi-vektoritulo $A\mathbf{x}$ käyttäen Fortran 90:n standardifunktiota `MATMUL`. Monissa laitteistoissa on kuitenkin käytettävissä lineaarialgebraan BLAS-rutiinit (Basic Linear Algebra Subprograms), jolloin matriisitulon voi toteuttaa tehokkaasti BLAS-rutiinilla `sgemv`. Esimerkiksi sivulla 65 esitettyyn aliohjelmaan `matrix_mult` tarvitaan vain seuraavat muutokset:

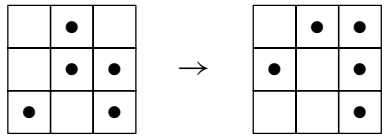
```
EXTERNAL sgemv ! BLAS-rutiini matriisi-vektoritulolle

vlen = nend - nstart + 1
CALL MPI_ALLGATHER(x, vlen, MPI_REAL, xloc, vlen, &
     MPI_REAL, MPI_COMM_WORLD, rc)
CALL sgemv('n', vlen, n , 1.0_prec, coeffmat, vlen, &
    xloc, 1, 0.0_prec, y, 1)
```

Aliohjelmakutsun `sgemv` jälkeen vektorissa `y` on matriisi-vektoritulon arvo. Samoin voi pistetulon laskevan Fortran 90/95:n funktion `DOT_PRODUCT` korvata BLAS-rutiinilla `sdot`.

6.5 Life-peli

Life-peli on kaksiulotteinen soluautomaatti. Koska systeemin vuorovaikutukset ovat paikallisia (naapurisolut vaihtavat tietoa keskenään), rinnakkaistaminen onnistuu kohtuullisen tehokkaasti.



Kuva 6.5: *Life-pelin tilanne kahdella peräkkäisellä ajanhetkellä.*

Life-pelin ideaa on havainnollistettu kuvassa 6.5. Solu voi olla kahdessa tilassa, joita kuvataan usein sanoilla ”elossa” tai ”kuollut”. Kunkin solun naapuristossa on kahdeksan solua, ja solun tila seuraavalla ajanhetkellä saadaan seuraavan säännön perusteella:

- jos naapureista on kolme elossa, naapuriston keskellä syntyy uusi solu
- jos naapureista on kaksi tai kolme elossa, vanha solu säilyy hengissä
- muussa tapauksessa elossa oleva solu kuolee joko ”yksinäisyyteen” tai ”liikakansoitukseen”.

Seuraavassa ohjelmakoodissa käytetään pelilautana kaksiulotteista kokonaislukutaulukkoa, jonka arvot voivat olla joko nollia tai ykkösiä. Nämä arvot vastaavat solun tilaa: $0 =$ kuollut ja $1 =$ elossa.

Tässä ohjelmassa käytetään karteesisen topologian luomiseen aliohjelmia `MPI_CART_CREATE` ja `MPI_CART_SHIFT`. Uuden tietotyypin luomiseen käytetään rutiineja `MPI_TYPE_VECTOR` ja `MPI_TYPE_COMMIT`. Laskettavan datan määrä välitetään rutiinien `MPI_BCAST` ja `MPI_SCATTER` avulla. Vierekkäisten laskentatehtävien väliseen viestinvälitykseen käytetään rutiinia `MPI_SENDRECV`. Laskennan lopettaminen hoidetaan rutiineilla `MPI_TYPE_FREE`, `MPI_COMM_FREE` ja `MPI_FINALIZE`.

```
PROGRAM Life

  USE mpi

  IMPLICIT NONE

  ! Halutaanko tulostusta?
  LOGICAL, PARAMETER :: do_output = .TRUE.
  INTEGER, PARAMETER :: &
      max_iter = 1000, &  ! Iteraatioiden maksimimäärä
      max_width = 1000, & ! Laudan maksimileveys
      max_height = 100, & ! Laudan maksimikorkeus
      print_id = 0, &     ! Tulostusnoodin järjestysnro
      board_tag_1 = 50, & ! Tunniste sarakedatalle
      board_tag_2 = 51, & ! Tunniste sarakedatalle
      print_unit = 101    ! Tulostuskanava
  CHARACTER(LEN=*), PARAMETER :: &
      filename = 'life.gen' ! Tulostustiedosto
  INTEGER, DIMENSION(:,:), POINTER :: &
      board_1, board_2    ! Life-pelin laudat
  INTEGER :: &
      comm, &             ! Karteesisen hilan viestintäryhmä
      right, left, &      ! Naapurien tunnisteet
      col_type, &         ! Saraketyyppi
      height, width, &    ! Laudan koko
      iter, &             ! Iteraatioiden lkm
      i                   ! Silmukkamuuttuja

  ! Alustetaan viestintä
  CALL init_comm(comm, ntask, my_id, right, left)
  ! Luetaan alkukonfiguraatio
  CALL read_data(iter, height, width, comm, ntask, my_id)
  ! Alustetaan Life-pelin laudat
  CALL init_board(board_1, board_2, height, width, &
      my_id, col_type)
  ! Lasketaan haluttu määrä iteraatioita
  DO i = 1, iter
    CALL iterate(board_1, board_2, comm, my_id, &
        right, left, col_type, i)
  END DO
  ! Lopetetaan laskenta
  CALL end_run(board_1, board_2, comm, my_id, col_type)

CONTAINS
```

Tähän tulevat aliohjelmien `init_comm`, `read_data`, `init_board`, `iterate` ja `end_run` määrittelyt.

```
END PROGRAM Life
```

Aliohjelma `init_comm` voidaan toteuttaa seuraavasti:

```
SUBROUTINE init_comm(comm, ntask, id, right, left)
  IMPLICIT NONE

  INTEGER,  INTENT(OUT) :: comm, ntask, id, right, left

  ! Käytetään 1-ulotteista topologiaa
```

```
  INTEGER, PARAMETER :: DIM = 1

  INTEGER, DIMENSION(mpi_status_size) :: status
  INTEGER, DIMENSION(0:DIM-1) :: sizes
  LOGICAL, DIMENSION(0:DIM-1) :: periods
  LOGICAL :: reorder
  INTEGER :: rc

  CALL MPI_INIT(rc)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntask, rc)

  ! Luodaan periodinen 1D-topologia
  sizes(DIM-1) = ntask
  periods(DIM-1) = .TRUE.
  reorder = .TRUE.
  CALL MPI_CART_CREATE(MPI_COMM_WORLD, 1, sizes, &
      periods, reorder, comm, rc)
  CALL MPI_CART_SHIFT(comm, 0, 1, right, left, rc)

  ! Selvitetään oma järjestysnro
  CALL MPI_COMM_RANK(comm, id, rc)
END SUBROUTINE init_comm
```

Tässä luotiin viestintäryhmään `comm` yksiulotteinen karteesinen topologia rutiinilla `MPI_CART_CREATE`. Topologiasta tehtiin periodinen ja MPI-järjestelmälle annettiin lupa järjestää laskentatehtävät uudelleen (`reorder`) topologian sisällä. Rutiinilla `MPI_CART_SHIFT` saatiin selville kunkin laskentatehtävän vasen ja oikea naapuritehtävä viestintäryhmässä `comm`.

Aliohjelmassa `read_data` luetaan sisään syöttötiedot:

```
SUBROUTINE read_data(iter, height, width, comm, ntask, id)
  IMPLICIT NONE

  INTEGER, INTENT(OUT) :: iter, height, width
  INTEGER, INTENT(IN) :: comm, id, ntask

  INTEGER, DIMENSION(0:ntask-1) :: num
  INTEGER :: task_cols, col_mod, i, rc

  IF (id == print_id) THEN
    WRITE (*,*) 'Anna iteraatioiden lukumäärä: '
    READ(*,*) iter
    IF (iter < 1 .OR. iter > max_iter) iter = max_iter
    WRITE (*,*) iter
    WRITE (*,*) 'Anna pelilaudan koko (korkeus, leveys):'
    READ(*,*) height, width
    IF (height < 1 .OR. height > max_height) &
        height = max_height
    IF (width < 1 .OR. width > max_width) &
        width = max_width
    WRITE (*,*) height, width
    ! Lasketaan sarakkeiden lukumäärä kussakin tehtävässä
    task_cols = width/ntask
    col_mod = MOD(width,ntask)
    DO i = 0, ntask-1
```

```
        IF (i < col_mod) THEN
          num(i) = task_cols + 1
        ELSE
          num(i) = task_cols
        END IF
      END DO
    END IF

    ! Lähetetään tiedot muille laskentatehtäville
    CALL MPI_BCAST(iter, 1, MPI_INTEGER, print_id, comm, rc)
    CALL MPI_BCAST(height, 1, MPI_INTEGER, print_id, comm, rc)
    CALL MPI_SCATTER(num, 1, MPI_INTEGER, width, 1, &
        MPI_INTEGER, print_id, comm, rc)
  END SUBROUTINE read_data
```

Laskennassa tarvittavat taulukot alustetaan aliohjelmassa `init_board`:

```
  SUBROUTINE init_board(board_1, board_2, height, width, &
      id, col_type)
    IMPLICIT NONE

    INTEGER, DIMENSION(:,:), POINTER :: board_1, board_2
    INTEGER, INTENT(IN) :: height, width, id
    INTEGER,  INTENT(OUT) :: col_type

    INTEGER :: i, stride, rc, ios

    ALLOCATE(board_1(1:height,0:width+1), &
        board_2(1:height,0:width+1))
    board_1(:,:) = 0
    board_2(:,:) = 0
    board_1(height/(id+2),:) = 1
    board_1(:,width/(id+2)) = 1
    ! Luodaan datatyyppi pelilaudan saraketta varten
    stride = 1
    CALL MPI_TYPE_VECTOR(height, 1, stride, &
        MPI_INTEGER, col_type, rc)
    CALL MPI_TYPE_COMMIT(col_type, rc)

    IF (do_output .AND. id == print_id) THEN
      OPEN(print_unit, file=filename, status='replace', &
          action='write', position='rewind', iostat=ios)
      WRITE(print_unit,'(A)') CHAR(27)//'[2J'  ! Pyyhi ruutu
      CALL draw(board_1, board_2, 0)
      ! Piirrä reunat Life-laudalle
      DO i = 1, height
        WRITE(print_unit,'(A,$)') CHAR(27)//'['
        IF (i < 10) THEN
          WRITE(print_unit,'(I1,A,$)') i, ';'
        ELSE
          WRITE(print_unit,'(I2,A,$)') i, ';'
        END IF
        IF (width < 10) THEN
          WRITE(print_unit,'(I1,A)') width, 'H|'
        ELSE
          WRITE(print_unit,'(I2,A)') width, 'H|'
```

```
        END IF
      END DO
      IF (height < 9) THEN
        WRITE(print_unit,'(A,I1,A,$)') &
            CHAR(27)//'[', height+1, ';0H'
      ELSE
        WRITE(print_unit,'(A,I2,A,$)') &
            CHAR(27)//'[', height+1, ';0H'
      END IF
      WRITE(print_unit,'(A,$)') ('-', i = 1, width-1)
      WRITE(print_unit,'(A)') '+'
    END IF
  END SUBROUTINE init_board
```

Tässä aliohjelmassa luotiin uusi MPI:n tietotyyppi pelilaudan sarakkeiden lähettämiseen ja vastaanottamiseen:

```
    stride = 1
    CALL MPI_TYPE_VECTOR(height, 1, stride, &
        MPI_INTEGER, col_type, rc)
    CALL MPI_TYPE_COMMIT(col_type, rc)
```

Tämän jälkeen tyyppiä `col_type` voi käyttää MPI:n sisäänrakennettujen tyyppien (esim. `MPI_REAL`) paikalla. Aliohjelma `init_board` piirtää haluttaessa Life-pelin alkutilanteen laskentatehtävässä `print_id`. Piirtämiseen käytetty aliohjelma `draw` määritellään myöhemmin.

Varsinainen laskentatyö tehdään aliohjelmassa `iterate`:

```
  SUBROUTINE iterate(board_1, board_2, comm, id, &
      right, left, col_type, iter)
    IMPLICIT NONE

    INTEGER, DIMENSION(:,:), POINTER :: board_1, board_2
    INTEGER, INTENT(IN) :: comm, id, right, left, col_type, iter

    INTEGER, DIMENSION(:,:), POINTER :: temp_board
    INTEGER, DIMENSION(SIZE(board_1,1),0:SIZE(board_1,2)-1) :: &
        sumval
    INTEGER, DIMENSION(mpi_status_size) :: status
    INTEGER :: wl, wu, rc

    wl = LBOUND(board_1,2)
    wu = UBOUND(board_1,2)

    CALL MPI_SENDRECV(board_1(:,wu-1), 1, col_type, right, &
        board_tag_1, board_1(:,wl), 1, col_type, left, &
        board_tag_1, comm, status, rc)
    CALL MPI_SENDRECV(board_1(:,wl+1), 1, col_type, left, &
        board_tag_2, board_1(:,wu), 1, col_type, right, &
        board_tag_2, comm, status, rc)

    sumval = CSHIFT(board_1,shift=1,DIM=1) + &
        CSHIFT(board_1,shift=-1,DIM=1) + &
        EOSHIFT(board_1,shift=1,DIM=2) + &
        EOSHIFT(board_1,shift=-1,DIM=2) + &
```

```
        CSHIFT(EOSHIFT(board_1,shift=1,DIM=2),shift=1,DIM=1) + &
        CSHIFT(EOSHIFT(board_1,shift=1,DIM=2),shift=-1,DIM=1) + &
        CSHIFT(EOSHIFT(board_1,shift=-1,DIM=2),shift=1,DIM=1) + &
        CSHIFT(EOSHIFT(board_1,shift=-1,DIM=2),shift=-1,DIM=1)

    WHERE (sumval(:,wl+1:wu-1) < 2 .OR. sumval(:,wl+1:wu-1) > 3)
      board_2(:,wl+1:wu-1) = 0
    ELSEWHERE
      board_2(:,wl+1:wu-1) = board_1(:,wl+1:wu-1)
    END WHERE
    WHERE (sumval(:,wl+1:wu-1) == 3)
      board_2(:,wl+1:wu-1) = 1
    END WHERE

    temp_board => board_1
    board_1 => board_2
    board_2 => temp_board

    IF (do_output .and. id == print_id) THEN
      CALL draw(board_1, board_2, iter)
    END IF

  END SUBROUTINE iterate
```

Aliohjelman `iterate` alussa välitetään taulukon reuna-alkioiden arvot rutiinin `MPI_SENDRECV` avulla sekä vasemmalle että oikealle. Tämän rutiinin avulla vältetään myös lukkiintumisen mahdollisuus.

Seuraavan sukupolven tilanne lasketaan taulukkoon `board_2` käyttämällä Fortran 90/95:n taulukkosyntaksia ja `WHERE`-rakennetta. Funktiot `CSHIFT` ja `EOSHIFT` siirtävät laskentahilaa haluttuun suuntaan. Funktio `CSHIFT` tekee siirtämisen periodisesti. Aliohjelman lopussa vaihdetaan taulukkojen osoitinmuuttujat keskenään, jolloin uudesta taulukosta `board_2` tulee seuraavalle kierrokselle vanha taulukko `board_1`.

Tulostaminen tehdään aliohjelmalla `draw` vain yhdessä laskentatehtävässä, jonka järjestysnumero on määrätty vakiossa `print_id`. Tämä laskentatehtävä tulostaa vain oman ”viipaleensa” laskenta-alueesta. Tässäkin tapauksessa tulostaminen on ohjelman pullonkaula, sillä näyttöpääte ei yleensä kykene esittämään laudalla tapahtuvia muutoksia riittävän nopeasti.

Aliohjelma `draw` hoitaa tulostuksen päätteelle seuraavasti käyttäen ns. VT100-päätteen tekstigrafiikkaa:

```
  SUBROUTINE draw(board_1,board_2, iter)
    IMPLICIT NONE

    INTEGER, DIMENSION(:,:), POINTER :: board_1, board_2
    INTEGER, INTENT(IN) :: iter

    INTEGER :: i, j, h

    DO i = LBOUND(board_1,1), UBOUND(board_1,1)
      DO j = LBOUND(board_1,2)+1, UBOUND(board_1,2)-1
        IF (board_1(i,j) /= board_2(i,j)) THEN
```

```
          WRITE(print_unit,'(A,$)') CHAR(27) // '['
          IF (i < 10) THEN
            WRITE(print_unit,'(I1,A,$)') i, ';'
          ELSE
            WRITE(print_unit,'(I2,A,$)') i, ';'
          END IF
          IF (j-1 < 10) THEN
            WRITE(print_unit,'(I1,A,$)') j-1, 'H'
          ELSE
            WRITE(print_unit,'(I2,A,$)') j-1, 'H'
          END IF
          IF (board_1(i,j) == 1) THEN
            WRITE(print_unit,'(A)') '*'
          ELSE
            WRITE(print_unit,'(A)') ' '
          END IF
        END IF
      END DO
    END DO
    h = SIZE(board_1,1) + 2
    IF (h < 10) THEN
      WRITE(print_unit,'(A,I1,A,I4)') &
          CHAR(27) // '[', h, ';0HIter:', iter
    ELSE
      WRITE(print_unit,'(A,I2,A,I4)') &
          CHAR(27) // '[', h, ';0HIter:', iter
    END IF
  END SUBROUTINE draw
```

Merkki `CHAR(27)` on Esc-merkki ja merkkiyhdistelmä Esc `[` on VT100-ohjauskoodien aloitustunnus.

Laskenta lopetetaan seuraavasti:

```
  SUBROUTINE end_run(board_1, board_2, comm, id, col_type)

    ! Lopetetaan MPI-viestinvälitys

    IMPLICIT NONE

    INTEGER, DIMENSION(:,:), POINTER :: board_1, board_2
    INTEGER, INTENT(IN) :: comm, id, col_type

    INTEGER :: ios, h, rc

    IF (do_output .AND. id == print_id) THEN
      h = SIZE(board_1,1)+3
      IF (h < 10) THEN
        WRITE(print_unit,'(A,I1,A,I6)') &
            CHAR(27) // '[', h, ';0HEnd of life'
      ELSE
        WRITE(print_unit,'(A,I2,A,I6)') &
            CHAR(27) // '[', h, ';0HEnd of life'
      END IF
      CLOSE(print_unit, status='keep', iostat=ios)
    END IF
```

```
    ! Vapautetaan MPI:n oliot
    CALL MPI_TYPE_FREE(col_type, rc)
    CALL MPI_COMM_FREE(comm, rc)
    ! Lopetetaan viestinvälitys
    CALL MPI_FINALIZE(rc)

  END SUBROUTINE end_run
```

Tässä esitetyn Life-ohjelman peräkkäisversio löytyy CSC:n Fortran 90/95 -oppikirjasta [HRR98]. Kyseisessä teoksessa on myös esimerkki ohjelman tulostuksesta. Rinnakkaisversiossa ei tulosteta kuitenkaan koko laskentahilan tilannetta vaan pelkästään yhden laskentatehtävän tilanne. Tällöin näyttöpäätteelle tulostuu vain osa soluautomaatin kehityksestä — viipaleen reunoilla vaihdetaan tietoa vasemman- ja oikeanpuoleisten laskentatehtävien kanssa.

6.6 Lisätietoa

Tässä luvussa käytettiin Fortran 90/95 -kieltä esimerkkiohjelmien esittämiseen. Fortran 90/95 -kieleen voi tutustua CSC:n julkaiseman oppikirjan avulla [HRR98].

MPI:n käyttöesimerkkejä löytyy lisäksi mm. käsikirjoista *MPI: A Message-Passing Interface Standard* [For95] ja *MPI: The Complete Reference* [SOHL+96]. Teokset *Using MPI: Portable Parallel Programming with the Message-Passing Interface* [GLS94] ja *Parallel Programming with MPI* [Pac97] ovat helppolukuisia johdatuksia MPI-ohjelmointiin.

Liitteet

A Tietoa CSC:stä

CSC – Tieteellinen laskenta Oy (CSC) on opetusministeriön omistama laskennallisen tieteen keskus. CSC:ssä on töissä noin 90 eri tieteenalojen ja tietotekniikan asiantuntijaa.

CSC tarjoaa laajoja palvelukokonaisuuksia tieteellistekniseen laskentaan. CSC:n asiantuntijat neuvovat esimerkiksi matemaattisten mallien ja numeeristen menetelmien käyttöön liittyvissä asioissa. CSC myös järjestää eri sovellusalojen koulutusta, esimerkiksi mallintamisesta, numeerisista menetelmistä, visualisoinnista ja ohjelmankehityksestä.

CSC on julkaissut tämän oppaan lisäksi mm. teokset

- *Rinnakkaislaskennan perusteet PVM-ympäristössä* [Saa95]
- *Fortran 90/95* [HRR98]
- *Numeeriset menetelmät* [HKR93]
- *Numeeriset menetelmät käytännössä* [HHL+99]
- *Matemaattiset ohjelmistot* [Haa98]
- *Optimointitehtävien ratkaiseminen* [Haa95]
- *Datan käsittely* [Kar01]
- *Elementtimenetelmä virtauslaskennassa* [HJ94]
- *Metakoneen käyttöopas* [KR00b]
- *CSC User's Guide* [KR00a]
- *Cray T3E User's Guide* [HS01]
- *IBM SP User's Guide* [HKR01].

Lisätietoja CSC:stä saa www-osoitteesta `http://www.csc.fi/`.

B MPI-tietoutta Internetissä

Seuraavassa on kerrottu muutamista hyödyllisistä MPI-viestinvälitykseen liittyvistä tiedonlähteistä Internetissä.

Tämän oppaan esimerkkiohjelmien lähdekoodeja löytyy www-osoitteesta

`http://www.csc.fi/oppaat/mpi/`

Myös mahdollisten painovirheiden korjaukset löytyvät tästä osoitteesta.

MPI:n käyttöesimerkkejä löytyy www-osoitteesta

`http://www.csc.fi/programming/examples/mpi/`

Käyttöesimerkit ovat osaksi samoja kuin CSC:n PVM-oppaassa [Saa95]. Nämä esimerkit löytyvät puolestaan osoitteesta

`http://www.csc.fi/programming/examples/pvm/`

Rinnakkaiskoneiden Cray T3E ja IBM SP englanninkieliset käyttöoppaat löytyvät www-osoitteista

```
http://www.csc.fi/oppaat/t3e/
http://www.csc.fi/oppaat/ibmsp/
```

MPI-standardista löytyy tietoa osoitteesta

`http://www.mcs.anl.gov/mpi/`

Teoksen *Designing and Building Parallel Programs* [Fos95] www-versio löytyy osoitteesta

`http://www.mcs.anl.gov/dbpp`

C MPI Quick Reference for Fortran Programmers

Numbers in brackets (e.g. [3.2.1]) refer to the MPI standard, see URL `http://www.csc.fi/programming/web/`.

MPI header file to include

INCLUDE 'mpif.h'

Initialization and shutdown

Initialize MPI [7.5]

MPI_INIT(ierror)
integer ierror

Cleanup MPI [7.5]

MPI_FINALIZE(ierror)
integer ierror

Basic communicators

Count group members in a communicator [5.4.1]

MPI_COMM_SIZE(comm, size, ierror)
integer comm, size, ierror

Determine rank of self in the group [5.4.1]

MPI_COMM_RANK(comm, rank, ierror)
integer comm, rank, ierror

Additional routines

MPI_COMM_CREATE, MPI_COMM_FREE, MPI_COMM_DUP, MPI_COMM_SPLIT, MPI_COMM_COMPARE, MPI_COMM_TEST_INTER, MPI_COMM_REMOTE_SIZE, MPI_COMM_REMOTE_GROUP, MPI_INTERCOMM_CREATE, MPI_INTERCOMM_MERGE

Blocking point-to-point communication

Send a message to one process [3.2.1]

MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
type buf(*)
integer count, datatype, dest, tag, comm, ierror

Receive a message from one process [3.2.4]

MPI_RECV(buf, count, datatype, source, tag, comm, status, ierror)
type buf(*)

integer count, datatype, source, tag, comm, status(MPI_STATUS_SIZE), ierror

Send and receive data [3.10]

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror)
type sendbuf(*), recvbuf(*)
integer sendcount, sendtype, dest, sendtag, recvcount, recvtype, source, recv tag, comm, status(MPI_STATUS_SIZE), ierror

Count received data elements [3.2.5]

MPI_GET_COUNT(status, datatype, count, ierror)
integer status(MPI_STATUS_SIZE), datatype, count, ierror

Wait for message arrival [3.8]

MPI_PROBE(source, tag, comm, status, ierror)
integer source, tag, comm, status(MPI_STATUS_SIZE), ierror

Additional routines

MPI_BSEND, MPI_SSEND, MPI_RSEND, MPI_BUFFER_ATTACH, MPI_BUFFER_DETACH, MPI_SENDRECV_REPLACE, MPI_GET_ELEMENTS

Non-blocking point-to-point communication

Begin to send a message [3.7.2]

MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierror)
type buf(*)
integer count, datatype, dest, tag, comm, request, ierror

Begin to receive a message [3.7.2]

MPI_IRECV(buf, count, datatype, source, tag, comm, request, ierror)
type buf(*)
integer count, datatype, source, tag, comm, request, ierror

Complete a non-blocking operation [3.7.3]

MPI_WAIT(request, status, ierror)
integer request, status(MPI_STATUS_SIZE), ierror

Check or complete a non-blocking operation [3.7.3]

MPI_TEST(request, flag, status, ierror)
logical flag
integer request, status(MPI_STATUS_SIZE), ierror

Check message arrival [3.8]

MPI_IPROBE(source, tag, comm, flag, status, ierror)
logical flag
integer source, tag, comm, status(MPI_STATUS_SIZE), ierror

Additional routines

MPI_IBSEND, MPI_ISSEND, MPI_IRSEND, MPI_WAITANY, MPI_TESTANY, MPI_WAITALL, MPI_TESTALL, MPI_WAITSOME, MPI_TESTSOME, MPI_CANCEL, MPI_TEST_CANCELLED, MPI_REQUEST_FREE

Collective communication

Send a message to all group members [4.4]

MPI_BCAST(buffer, count, datatype, root, comm, ierror)
type buffer(*)

integer count, datatype, root, comm, ierror

Combine messages from all group members [4.9.1]

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer count, datatype, op, root, comm, ierror

Result is returned to all processes in a group [4.9.5]

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, ierror)
type sendbuf(*), recvbuf(*)
integer count, datatype, op, comm, ierror

Reduction operators [4.9.2]

MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR

Send different messages to all group members [4.6]

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Send differently sized messages to all group members [4.6]

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcounts(*), displs(*), sendtype, recvcount, recvtype, root, comm, ierror

Receive from all group members [4.5]

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcount, sendtype, recvcount, recvtype, root, comm, ierror

Receive different amount of data from group members [4.5]

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcount, sendtype, recvcounts(*), displs(*), recvtype, root, comm, ierror

All processes receive the gather result [4.7]

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierror)
type sendbuf(*), recvbuf(*)
integer sendcount, sendtype, recvcount, recvtype, comm, ierror

Reduction and scatter operation [4.10]

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror)
type sendbuf(*), recvbuf(*)
integer recvcounts(*), datatype, op, comm, ierror

Additional routines

MPI_BARRIER, MPI_ALLGATHERV, MPI_ALLTOALL, MPI_ALLTOALLV, MPI_SCAN, MPI_OP_CREATE, MPI_OP_FREE

Group management routines

MPI_GROUP_SIZE, MPI_GROUP_RANK, MPI_GROUP_TRANSLATE_RANKS, MPI_GROUP_COMPARE, MPI_COMM_GROUP, MPI_GROUP_UNION, MPI_GROUP_INTERSECTION, MPI_GROUP_DIFFERENCE, MPI_GROUP_INCL, MPI_GROUP_EXCL, MPI_GROUP_RANGE_INCL, MPI_GROUP_RANGE_EXCL, MPI_GROUP_FREE

Routines for persistent communication

MPI_SEND_INIT, MPI_BSEND_INIT, MPI_SSEND_INIT, MPI_RSEND_INIT, MPI_RECV_INIT, MPI_START, MPI_STARTALL

Derived MPI data types

Create a homogeneous vector [3.12.1]

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierror)
integer count, oldtype, newtype, ierror

Create a strided homogeneous vector [3.12.1]

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype, ierror)
integer count, blocklength, stride, oldtype, newtype, ierror

Save a derived type [3.12.4]

MPI_TYPE_COMMIT(datatype, ierror)
integer datatype, ierror

Free a derived type [3.12.4]

MPI_TYPE_FREE(datatype, ierror)
integer datatype, ierror

Additional routines

MPI_PACK, MPI_UNPACK, MPI_PACK_SIZE, MPI_TYPE_HVECTOR, MPI_TYPE_INDEXED, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_SIZE, MPI_TYPE_LB, MPI_TYPE_UB

Communicators with topology

Create a cartesian topology [6.5.1]

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
comm_cart, ierror)
integer comm_old, ndims, dims(*), comm_cart, ierror
logical periods(*), reorder

Determine ranks for cartesian shift [6.5.5]

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest, ierror)
integer comm, direction, disp, rank_source, rank_dest, ierror

Determine rank from cartesian coordinates [6.5.4]

MPI_CART_RANK(comm, coords, rank, ierror)
integer comm, coords(*), rank, ierror

Determine cartesian coordinates from rank [6.5.4]

MPI_CART_COORDS(comm, rank, maxdims, coords, ierror)
integer comm, rank, maxdims, coords(*), ierror

Additional routines

MPI_DIMS_CREATE, MPI_CART_SUB, MPI_CARTDIM_GET, MPI_CART_GET, MPI_CART_NEIGHBORS_COUNT, MPI_CART_MAP, MPI_GRAPH_CREATE, MPI_GRAPHDIMS_GET, MPI_GRAPH_GET, MPI_GRAPH_NEIGHBORS, MPI_GRAPH_MAP, MPI_TOPO_TEST

Routines for communicator caches

MPI_KEYVAL_CREATE, MPI_KEYVAL_FREE, MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE

Error handling routines

MPI_ERRHANDLER_CREATE, MPI_ERRHANDLER_SET, MPI_ERRHANDLER_GET, MPI_ERRHANDLER_FREE, MPI_ERROR_STRING, MPI_ERROR_CLASS, MPI_ABORT

Environment inquiry

Determine wall clock time [7.4]

double precision **MPI_WTIME**()

Additional routines

MPI_GET_PROCESSOR_NAME, MPI_WTICK, MPI_INITIALIZED, MPI_PCONTROL

MPI constants

Wildcard constants [3.2.4]

MPI_ANY_TAG, MPI_ANY_SOURCE

Elementary MPI data types [3.2.2]

MPI_CHARACTER, MPI_INTEGER, MPI_REAL, MPI_COMPLEX, MPI_DOUBLE_PRECISION, MPI_LOGICAL, MPI_BYTE, MPI_PACKED

Reserved communicators [5.2.4]

MPI_COMM_WORLD, MPI_COMM_SELF

MPI null objects

MPI_PROC_NULL, MPI_COMM_NULL, MPI_GROUP_NULL, MPI_TYPE_NULL

Additional constants

MPI_SUCCESS, MPI_SOURCE, MPI_TAG, MPI_UNDEFINED

D MPI Quick Reference for C Programmers

Numbers in brackets (e.g. [3.2.1]) refer to the MPI standard, see URL `http://www.csc.fi/programming/web/`.

MPI header file to include

#include <mpi.h>

Initialization and shutdown

Initialize MPI [7.5]

int **MPI_Init**(int *argc, char ***argv)

Cleanup MPI [7.5]

int **MPI_Finalize**(void)

Basic communicators

Count group members in a communicator [5.4.1]

int **MPI_Comm_size**(MPI_Comm comm, int *size)

Determine rank of self in the group [5.4.1]

int **MPI_Comm_rank**(MPI_Comm comm, int *rank)

Additional routines

MPI_Comm_create, MPI_Comm_free, MPI_Comm_dup, MPI_Comm_split, MPI_Comm_compare, MPI_Comm_test_inter, MPI_Comm_remote_size, MPI_Comm_remote_group, MPI_Intercomm_create, MPI_Intercomm_merge

Blocking point-to-point communication

Send a message to one process [3.2.1]

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

Receive a message from one process [3.2.4]

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

Send and receive data [3.10]

int **MPI_Sendrecv**(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)

Count received data elements [3.2.5]

int **MPI_Get_count**(MPI_Status *status, MPI_Datatype datatype, int *count)

Wait for message arrival [3.8]

int **MPI_Probe**(int source, int tag, MPI_Comm comm, MPI_Status *status)

Additional routines

MPI_Bsend, MPI_Ssend, MPI_Rsend, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Sendrecv_replace, MPI_Get_elements

Non-blocking point-to-point communication

Begin to send a message [3.7.2]

int **MPI_Isend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

Begin to receive a message [3.7.2]

int **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

Complete a non-blocking operation [3.7.3]

int **MPI_Wait**(MPI_Request *request, MPI_Status *status)

Check or complete a non-blocking operation [3.7.3]

int **MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status)

Check message arrival [3.8]

int **MPI_Iprobe**(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)

Additional routines

MPI_Ibsend, MPI_Issend, MPI_Irsend, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome, MPI_Cancel, MPI_Test_cancelled, MPI_Request_free

Collective communication

Send a message to all group members [4.4]

int **MPI_Bcast**(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Combine messages from all group members [4.9.1]

int **MPI_Reduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Result is returned to all processes in a group [4.9.5]

int **MPI_Allreduce**(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Reduction operators [4.9.2]

MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR

Send different messages to all group members [4.6]

int **MPI_Scatter**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Send differently sized messages to all group members [4.6]

int **MPI_Scatterv**(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Receive from all group members [4.5]

int **MPI_Gather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Receive different amount of data from group members [4.5]

int **MPI_Gatherv**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)

All processes receive the gather result [4.7]

int **MPI_Allgather**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Reduction and scatter operation [4.10]

int **MPI_Reduce_scatter**(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Additional routines

MPI_Barrier, MPI_Allgatherv, MPI_Alltoall, MPI_Alltoallv, MPI_Scan, MPI_Op_create, MPI_Op_free

Group management routines

MPI_Group_size, MPI_Group_rank, MPI_Group_translate_ranks, MPI_Group_compare, MPI_Comm_group, MPI_Group_union, MPI_Group_intersection, MPI_Group_difference, MPI_Group_incl, MPI_Group_excl, MPI_Group_range_incl, MPI_Group_range_excl, MPI_Group_free

Routines for persistent communication

MPI_Send_init, MPI_Bsend_init, MPI_Ssend_init, MPI_Rsend_init, MPI_Recv_init, MPI_Start, MPI_Startall

Derived MPI data types

Create a homogeneous vector [3.12.1]

int **MPI_Type_contiguous**(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

Create a strided homogeneous vector [3.12.1]

int **MPI_Type_vector**(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Save a derived type [3.12.4]

int **MPI_Type_commit**(MPI_Datatype *datatype)

Free a derived type [3.12.4]

int **MPI_Type_free**(MPI_Datatype *datatype)

Additional routines

MPI_Pack, MPI_Unpack, MPI_Pack_size, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct, MPI_Address, MPI_Type_extent, MPI_Type_size, MPI_Type_lb, MPI_Type_ub

Communicators with topology

Create a cartesian topology [6.5.1]

int **MPI_Cart_create**(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

Determine ranks for cartesian shift [6.5.5]

int **MPI_Cart_shift**(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

Determine rank from cartesian coordinates [6.5.4]

int **MPI_Cart_rank**(MPI_Comm comm, int *coords, int *rank)

Determine cartesian coordinates from rank [6.5.4]

int **MPI_Cart_coords**(MPI_Comm comm, int rank, int maxdims, int *coords)

Additional routines

MPI_Dims_create, MPI_Cart_sub, MPI_Cartdim_get, MPI_Cart_get, MPI_Cart_neighbors_count, MPI_Cart_map, MPI_Graph_create, MPI_Graphdims_get, MPI_Graph_get, MPI_Graph_neighbors, MPI_Graph_map, MPI_Topo_test

Routines for communicator caches

MPI_Keyval_create, MPI_Keyval_free, MPI_Attr_put, MPI_Attr_get, MPI_Attr_delete

Error handling routines

MPI_Errhandler_create, MPI_Errhandler_set, MPI_Errhandler_get, MPI_Errhandler_free, MPI_Error_string, MPI_Error_class, MPI_Abort

Environment inquiry

Determine wall clock time [7.4]

double **MPI_Wtime**(void)

Additional routines

MPI_Get_processor_name, MPI_Wtick, MPI_Initialized, MPI_Pcontrol

MPI constants

Wildcard constants [3.2.4]

MPI_ANY_TAG, MPI_ANY_SOURCE

Elementary MPI data types [3.2.2]

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED

Reserved communicators [5.2.4]

MPI_COMM_WORLD, MPI_COMM_SELF

MPI null objects

MPI_PROC_NULL, MPI_COMM_NULL, MPI_GROUP_NULL, MPI_TYPE_NULL

Additional constants

MPI_SUCCESS, MPI_SOURCE, MPI_TAG, MPI_UNDEFINED

E Sanasto

Seuraavassa on selitetty eräitä keskeisiä rinnakkaisohjelmoinnin ja MPI:n termejä.

all-to-all: täydellinen viestien vaihto
allgather: yleiskeräys
asynchronous: tahdistamaton, asynkroninen
bandwidth: kaistanleveys, tiedonsiirtonopeus
barrier: tahdistus(operaatio)
blocking: odottamaan pysähtyvä, blokkaava
broadcast: yleislähetys
buffer: puskuri, viestipuskuri
buffered: puskuroitu
cache memory: välimuisti
cartesian topology: karteesinen topologia
collective communication: ryhmäviestintä, kollektiivinen viestintä
collective operations: ryhmäoperaatiot
communication domain: viestintäalue
communicator cache: viestintäryhmän välivarasto
communicator: viestintäryhmä
conjugate gradient method: liittogradienttimenetelmä
daemon: demoni, taustapalvelinohjelma
datatype: tiedon tyyppi
data distribution: datan sijoittelu (laskentatehtäville)
debugging: vianetsintä
deadlock: lukkiutuminen
derived data types: käyttäjän määrittelemät datatyypit
destination: viestin kohde, viestin kohdeprosessi
displacement: peräkkäisten alkioiden välimatka
distributed memory: hajautettu keskusmuisti
elementary data types: perusdatatyypit
finalize: lopeta (esim. viestintä)
gather: keräys
global operation: globaali operaatio
group communication: ryhmäviestintä
handle: kahva
homogeneous vector: samanlaisista alkioista koostuva vektori

initialize: alusta (esim. viestintä)
initialization: alustaminen, käynnistäminen
latency: lähetysviive, saantiviive, latenssiaika
load balancing: kuormantasaus
malleable: mukautuva, vaihtelevalla prosessorimäärällä toimiva
message: viesti
message buffer: viestipuskuri
message passing: viestinvälitys
MPP (massively parallel processor): massiivisesti rinnakkainen tietokone (prosessoreilla on omat hajautetut keskusmuistit)
neighbor: naapuri
non-blocking: välittömästi palaava, ei-blokkaava
non-buffered: puskuroimaton
null object: tyhjä olio
pack: pakata (dataa lähettämistä varten)
parallel: rinnakkainen
parallelism: rinnakkaisuus
parallelize: rinnakkaistaa
parallel computing: rinnakkaislaskenta
persistent communication: pysyvä viestintä
periodic: jaksollinen, periodinen
point-to-point communication: kahdenvälinen viestintä
process: prosessi
protocol: protokolla, tietoliikenneyhteyskäytäntö
rank: järjestysnumero
ready mode send: vastaanottajan valmiuteen perustuva lähetys
receive: vastaanottaa; viestin vastaanottokutsu
receiver: vastaanottaja
reduce: reduktio(-operaatio)
reduction: reduktio
reordering: uudelleenjärjestely (esim. uudessa topologiassa)
request: (palvelu)pyyntö
reserved communicator: varattu viestintäryhmä
scan: kertymä, selaus
scatter: levitys
send: lähettää; viestin lähetyskutsu
sender: lähettäjä
sequential: peräkkäinen, peräkkäis-
shared memory: yhteinen keskusmuisti, jaettu keskusmuisti
shift: siirto
shutdown: pysäytys
SMP (symmetric multiprocessor): symmetrinen moniprosessori (prosessoreilla on yhteinen keskusmuisti)
source: viestin lähettäjä, viestin lähettäjäprosessori
stride: askelpituus
synchronize: tahdistaa
synchronous: tahdistettu, synkronoitu

tag: tunniste
task: tehtävä, laskentatehtävä
thread: säie
topology: topologia
unpack: purkaa (dataa vastaanottamisen jälkeen)
virtual topology: näennäinen topologia, virtuaalinen topologia
wall clock time: seinäkelloaika
wildcard: jokerimerkki

Kirjallisuutta

[FHK00] Juha Fagerholm, Juha Haataja ja Sirpa Kotila, toim. *CSC:n asiakkaan opas*. CSC, 2000. `http://www.csc.fi/oppaat/asopas/`.

[For95] Message-Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, 1995.

[Fos95] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. Internet-osoite `http://www.mcs.anl.gov/dbpp`.

[GLS94] William Gropp, Ewing Lusk ja Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[Haa95] Juha Haataja. *Optimointitehtävien ratkaiseminen*. CSC, 1995. 2. painos.

[Haa98] Juha Haataja, toim. *Matemaattiset ohjelmistot*. CSC, 1998. `http://www.csc.fi/oppaat/mat.ohj/`.

[HHL+99] Juha Haataja, Jussi Heikonen, Yrjö Leino, Jussi Rahola, Juha Ruokolainen ja Ville Savolainen. *Numeeriset menetelmät käytännössä*. CSC, 1999.

[HJ94] Jari Hämäläinen ja Jari Järvinen. *Elementtimenetelmä virtauslaskennassa*. CSC, 1994.

[HKR93] Juha Haataja, Juhani Käpyaho ja Jussi Rahola. *Numeeriset menetelmät*. CSC, 1993.

[HKR01] Juha Haataja ja Tiina Kupila-Rantala, toim. *IBM SP User's Guide*. CSC, 2001. `http://www.csc.fi/oppaat/ibmsp/`.

[HRR98] Juha Haataja, Jussi Rahola ja Juha Ruokolainen. *Fortran 90/95*. CSC, 1998.

[HS01] Juha Haataja ja Ville Savolainen, toim. *Cray T3E User's Guide*. CSC, 2001. `http://www.csc.fi/oppaat/t3e/`.

[Kar01] Hannu Karttunen. *Datan käsittely*. CSC, 2001.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta ja George Karypis. *Introduction to Parallel Computing — Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[KR00a] Tiina Kupila-Rantala, toim. *CSC User's Guide*. CSC, 2000. `http://www.csc.fi/oppaat/cscuser/`.

[KR00b] Tiina Kupila-Rantala, toim. *Metakoneen käyttöopas*. CSC, 2000. `http://www.csc.fi/oppaat/metakone/`.

[Pac97] Peter S. Pachero. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[Saa95] Sami Saarinen. *Rinnakkaislaskennan perusteet PVM-ympäristössä*. CSC, 1995.

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker ja Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

Hakemisto

Symbolit
`#include`, 14

A
aikajärjestys, 18
ajaminen, 22
ajanmittaus, 48
alkioiden lukumäärä, 18
alkioiden tyyppi, 18
`ALLOCATE`, 34, 55
aloitus, 12
ANSI C -kieli, 11, 16
arkkitehtuurit, 8
askelpituus, 34

B
Basic Linear Algebra Subprograms, 68
BLAS-rutiinit, 68
blocking, 18
blokkaavat rutiinit, 18
`buf`, 18
buffered, 38

C
C++-kieli, 11
C-kielinen ohjelma, 14
`comm`, 18
communicator, 9, 17
Compaq, 11
conjugate gradient method, 63
`count`, 18
CPU, 7
Cray T3E, 11
Cray TotalView, 49
CSC – Tieteellinen laskenta Oy, 78

D
data-parallel, 7
data-passing, 8
datarinnakkainen ohjelmointi, 7, 8
`datatype`, 18
deadlock, 40
`dest`, 18
`DOT_PRODUCT`, 65, 69

E
esiprosessori
    rinnakkaistava, 7

F
Fortran 77 -kieli, 11
Fortran 90 -kieli, 10, 11, 16
Fortran 95 -kieli, 10, 11
Fortran-kielinen ohjelma, 12

G
gammafunktio, 56

H
hajautetun muistin rinnakkaiskone, 11
handle, 17, 19
High Performance Fortran, 9
HPF, 9

I
IBM SP, 11
`INCLUDE`, 12
isäntä-renki -ohjelmointimalli, 56

J
järjestysnumero, 12–14, 17, 18, 20

K
kahdenvälinen viestintä, 17, 19
    C-kieli, 21
    Fortran-kieli, 19
kahva, 17, 19
karteesinen topologia, 43, 45, 69, 71
keräys, 29
keskiarvon laskeminen, 51
keskusyksikkö, 7
kollektiivinen viestintä, 24
komentoriviargumentit, 14, 16
kuormantasaus, 56, 57, 62
kuutiohila, 43
käynnistys, 16

L

laskentatehtävä, 7
    lukumäärä, 13, 14
levitys, 29
Life-peli, 69
liittogradienttimenetelmä, 63
lineaarinen yhtälöryhmä, 63
lopettaminen, 14
lukkiintuminen, 40, 74
lähettäjä, 18

M

massively parallel processor, 8
master-slave, 56
`MATMUL`, 66, 68
matriisi-vektoritulo, 68
message passing, 7
Message-Passing Interface, 6, 9
`MODULE`, 12, 51
monenvälinen viestintä, 24
MPI, 9
MPI-ohjelman ajaminen, 22
MPI-viestinvälityskirjasto, 6
`mpi.h`, 14
MPI:n avainsanat
    MPI_ALLGATHER, 35, 37, 65, 82
    MPI_ALLREDUCE, 28, 35, 37, 65, 82
    MPI_ANY_SOURCE, 20, 21
    MPI_ANY_TAG, 20
    MPI_Allgather, 37, 87
    MPI_Allreduce, 37, 86
    MPI_BAND, 27
    MPI_BARRIER, 36, 37
    MPI_BCAST, 25, 27–29, 35, 54, 66, 69, 81
    MPI_BOR, 27
    MPI_BSEND, 38
    MPI_BUFFER_ATTACH, 38
    MPI_BXOR, 27
    MPI_BYTE, 22
    MPI_Barrier, 37
    MPI_Bcast, 25, 27, 42, 86
    MPI_CARTDIM_GET, 44, 45
    MPI_CART_COORDS, 44, 45, 83
    MPI_CART_CREATE, 44, 45, 69, 71, 83
    MPI_CART_GET, 44, 45
    MPI_CART_RANK, 44, 45, 83
    MPI_CART_SHIFT, 44, 45, 69, 71, 83
    MPI_CHARACTER, 22
    MPI_CHAR, 22
    MPI_COMM_FREE, 69, 76
    MPI_COMM_RANK, 12, 14, 16, 80
    MPI_COMM_SIZE, 12, 13, 16, 17, 80
    MPI_COMM_SPLIT, 41
    MPI_COMM_WORLD, 17, 19, 20, 41, 43
    MPI_COMPLEX, 22
    MPI_Cart_coords, 45, 88
    MPI_Cart_create, 45, 88
    MPI_Cart_get, 45
    MPI_Cart_rank, 45, 88
    MPI_Cart_shift, 45, 88
    MPI_Cartdim_get, 45
    MPI_Comm_rank, 14, 16, 41, 43, 85
    MPI_Comm_size, 14, 16, 85
    MPI_Comm_split, 41
    MPI_Comm, 19
    MPI_DOUBLE_COMPLEX, 22
    MPI_DOUBLE_PRECISION, 22
    MPI_DOUBLE, 22
    MPI_FINALIZE, 12, 14, 16, 17, 69, 80
    MPI_FLOAT, 22
    MPI_Finalize, 15, 16, 85
    MPI_GATHERV, 34–36, 82
    MPI_GATHER, 29, 31, 35, 82
    MPI_GET_COUNT, 81
    MPI_Gatherv, 36, 87
    MPI_Gather, 29, 87
    MPI_Get_count, 86
    MPI_INIT, 12, 13, 16, 41, 80
    MPI_INTEGER, 18, 20, 22, 24, 25
    MPI_INT, 18, 21, 22
    MPI_IPROBE, 81
    MPI_IRECV, 40, 47, 81
    MPI_IRSEND, 39
    MPI_ISEND, 40, 47, 81
    MPI_Init, 14, 16, 85
    MPI_Iprobe, 86
    MPI_Irecv, 86
    MPI_Isend, 86
    MPI_LAND, 27
    MPI_LOGICAL, 22
    MPI_LONG_DOUBLE, 22
    MPI_LONG, 22
    MPI_LOR, 27
    MPI_LXOR, 27
    MPI_MAXLOC, 27
    MPI_MAX, 27
    MPI_MINLOC, 27
    MPI_MIN, 27
    MPI_PACKED, 22
    MPI_PROBE, 81
    MPI_PROD, 27, 43
    MPI_Probe, 86

MPI_REAL, 22, 34, 73
MPI_RECV, 12, 14, 16–18, 20, 21, 38–40, 44, 57, 80
MPI_REDUCE_SCATTER, 68, 82
MPI_REDUCE, 24, 25, 27–29, 35, 53, 54, 68, 82
MPI_RSEND, 39
MPI_Recv, 15, 16, 21, 22, 85
MPI_Reduce_scatter, 87
MPI_Reduce, 25, 27, 43, 86
MPI_Request, 39
MPI_SCATTERV, 34–36, 54, 82
MPI_SCATTER, 29, 31, 53, 68, 69, 72, 82
MPI_SENDRECV, 44, 45, 69, 73, 74, 81
MPI_SEND, 12, 14, 16–20, 38–40, 44, 57, 80
MPI_SHORT, 22
MPI_SOURCE, 15, 21, 22
MPI_SSEND, 38, 41
MPI_SUCCESS, 13, 17, 20, 47
MPI_SUM, 24, 25, 27, 43
MPI_Scatterv, 36, 87
MPI_Scatter, 29, 86
MPI_Sendrecv, 45, 85
MPI_Send, 15, 16, 21, 85
MPI_Status, 14
MPI_TEST, 39, 40, 47, 81
MPI_TYPE_COMMIT, 69, 72, 73, 83
MPI_TYPE_CONTIGUOUS, 83
MPI_TYPE_FREE, 69, 76, 83
MPI_TYPE_VECTOR, 69, 72, 73, 83
MPI_Test, 39, 86
MPI_Type_commit, 87
MPI_Type_contiguous, 87
MPI_Type_free, 87
MPI_Type_vector, 87
MPI_UNDEFINED, 41, 44
MPI_UNSIGNED_CHAR, 22
MPI_UNSIGNED_LONG, 22
MPI_UNSIGNED_SHORT, 22
MPI_UNSIGNED, 22
MPI_WAIT, 39, 40, 47, 81
MPI_WTICK, 48
MPI_WTIME, 48, 66, 84
MPI_Wait, 39, 86
MPI_Wtick, 48
MPI_Wtime, 48, 88
`mpif.h`, 12, 51
MPP-tietokone, 8
`mpprun`, 22

N

naapuritehtävä, 71
non-blocking communication, 39

O

ohjelman ajaminen, 22
ohjelmointimallit, 7

P

paluukoodi, 13, 14, 17
Parallel Virtual Machine, 9
`poe`, 22
process, 7
prosessi, 7
pullonkaula, 48
puskuritila, 38
puskuroitu, 38
PVM, 9
pysähtyvät rutiinit, 18

R

ready mode, 39
reduktio-operaatiot, 25, 27
rinnakkaiskoneet, 6
rinnakkaisohjelmointi, 6
ryhmäoperaatiot, 24

S

`sdot`, 69
seinäkelloaika, 48
`sgemv`, 68
SGI, 11
shared memory, 7
shift, 44
Single Program, Multiple Data, 51
SMP-tietokone, 8
soluautomaatti, 69
`source`, 18
SPMD-ohjelmointimalli, 51
`status`, 18
symmetric multiprocessor, 8
synchronization, 36
synkronoitu, 38

T

`tag`, 18
tahdistus, 36
task, 7
tehtävä, 7
tiedonsiirtoväylä, 8
tiedonsiirtoverkko, 8
tietotyypit, 22
topologia, 43, 45
karteesinen, 69, 71
TotalView, 49
tunnistenumero, 14, 15, 21

työkaluohjelmat, 48

V
VAMPIR, 49
vastaanottaja, 18, 20, 21
vastaanottajan valmius, 39
viesti
    lähettäjä, 18
    pituus, 18
    tunnistenumero, 14
    tyyppi, 18
    vastaanottaja, 18
viestintä
    alustaminen, 16
    kahdenvälinen, 17
    kollektiivinen, 24
    lopettaminen, 16
viestintämuodot, 38
viestintäryhmä, 9, 17, 18, 20, 41
    luominen, 41
viestinvälitys, 6–8
viestinvälitysohjelmointi, 7, 8
välittömästi palaavat rutiinit, 39

Y
yhteisen muistin moniprosessorikone, 11
yhteisen muistin ohjelmointi, 7, 8
yhtälöryhmä, 63