Metropolia

TEKNIIKKA JA LIIKENNE

Tietotekniikka

Ohjelmistotekniikka

INSINÖÖRITYÖ

RINNAKKAISLASKENTA MPI-YMPÄRISTÖSSÄ

Työn tekijä: Juha Katajisto
Työn ohjaajat:

Miikka Mäki-Uuro
Simo Silander

Työ hyväksytty: ___. ___. 2009

Miikka Mäki-Uuro
Lehtori

Metropolia

TIIVISTELMÄ

Työn tekijä: Juha Katajisto

Työn nimi: Rinnakkaislaskenta MPI-ympäristössä

Päivämäärä: 15.11.2009 **Sivumäärä:** 62 + 4 liitettä

Koulutusohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Työn ohjaaja: Lehtori Miikka Mäki-Uuro

Työn ohjaaja: Lehtori Simo Silander

Tämän Metropolia Ammattikorkeakoululle tehdyn insinöörityön tavoitteena oli tutkia rinnakkaisia arkkitehtuureja ja viestinvälitykseen perustuvaa rinnakkaisohjelmointia, rakentaa testiympäristöksi korkean suorituskyvyn klusteri ja rinnakkaistaa alun perin sarjamuotoinen algoritmi.

Työ aloitettiin perehtymällä rinnakkaisohjelmien suorituksen mahdollistaviin laitteistoarkkitehtuureihin. Seuraavaksi syvennyttiin tarkemmin yhteen käyttöjärjestelmään (Rocks Cluster) ja rakennettiin sen avulla korkean suorituskyvyn klusteri. Klusterin tarkoituksena oli toimia rinnakkaisohjelmien testiympäristönä. Klusterin avulla voidaan rinnakkaisia ohjelmia suorittaa helposti useiden tietokoneiden avulla. Seuraavaksi perehdyttiin erityisesti viestinvälitysmallin mukaiseen rinnakkaisohjelmointiin ja MPI-standardiin. MPI-kirjastosta esitellään useimmin tarvittavat operaatiot. Lopuksi suunniteltiin ja toteutettiin sarjamuotoisesta lajittelualgoritmista neljä rinnakkaista versiota. Työssä toteutettujen algoritmien suorituskykyä vertailtiin keskenään erilaisissa tilanteissa.

Työn tuloksena saatiin toimiva korkean suorituskyvyn klusteri ja neljä toimivaa rinnakkaista versiota alun perin sarjamuotoisesta algoritmista. Tehtyjen testien perusteella huomattiin, että rinnakkaisohjelmoinnin avulla voidaan saada suuria parannuksia ohjelman suorituskykyyn, kunhan ohjelmassa esiintyvät laskennalliset ongelmat ovat luonteeltaan rinnakkaistuvia. Työn aikana huomattiin suuria eroja myös rinnakkaisten algoritmien suorituskyvyssä ja havaittiin niiden suorituskykyyn oleellisesti vaikuttavia seikkoja, kuten työmäärien jakautuminen laskentatehtävien välillä. Myös datan siirtäminen laskentatehtävien välillä on hidasta, joten on sitä parempi, mitä vähemmän sitä tarvitsee tehdä. Tähän toki vaikuttaa myös klusterin solmujen välillä käytetyn tietoliikenneverkon nopeus.

Avainsanat: MPI, Quicksort, Rinnakkaisohjelmointi

Metropolia

ABSTRACT

Name: Juha Katajisto

Title: Parallel Computing in the MPI Environment

Date: Nov. 15, 2009 **Number of pages:** 62 + 4 attachments

Department: Information Technology

Study Programme: Software Engineering

Instructor: Miikka Mäki-Uuro, Senior Lecturer

Instructor: Simo Silander, Senior Lecturer

The purpose of this Bachelor's Thesis is to examine the architectures that allow parallel computing and message passing parallel programming. In the study a high performance computing cluster was built and an originally sequential sorting algorithm was implemented in parallelized programming.

Parallel hardware architectures were explored in the study. Next a high performance computing cluster was built from eight workstations, which were linked together with Gigabit Ethernet network. The Rocks Cluster Linux distribution was chosen to be used as an operating system. The Rocks Cluster is based on the CentOS-Linux and is specially designed to make building and maintaining clusters easy. The commonly applied operations of the MPI standard as well as some more advanced ones are described. Finally, four different parallel versions of the originally sequential sorting algorithm were designed and implemented. The performance of implemented algorithms was tested and compared with each other in multiple situations.

As the result of the design and the implementation a working high performance computing cluster was built and four working parallel versions of the originally sequential algorithm were implemented. Based on the conducted tests it was found that parallelization of a program can make a significant boost to software performance assuming that the computational problems in the software are parallelizable.

The study discovered differences in the performance between the parallel versions of the algorithm. These differences resulted mainly from items being distributed unevenly between processors in some parallel versions. Transferring large amounts of data between processors can also be very expensive operation, depending on speed of the network that links the processors together.

Keywords: MPI, Quicksort, Parallel Programming

SANASTO

Algoritmi	Tarkasti määritelty äärellinen (päättyvä) vaihesarja, jota seuraamalla voidaan ratkaista tietty ongelma.
BIOS	*Basic Input-Output System*. Tietokoneohjelma, joka etsii ja lataa käyttöjärjestelmän keskusmuistiin sekä käynnistää sen tietokoneen käynnistyessä.
Daemon	Linux-käyttöjärjestelmässä taustaprosessina ajettava palvelu (prosessi).
De facto -standardi	Käytännössä syntynyt standardi. Epävirallinen standardi, joka on yleensä saavuttanut hallitsevan markkinaosuuden.
DHCP	*Dynamic Host Configuration Protocol.* Verkkoprotokolla, jonka yleisin tehtävä on jakaa IP-osoitteita lähiverkkoon kytketyille laitteille.
Ethernet	Pakettipohjainen lähiverkkoratkaisu, joka on yleisin ja ensimmäisenä laajasti hyväksytty lähiverkkotekniikka. IEEE on standardoinut Ethernet-tekniikoita 802.3-työryhmässä.
Gigabit Ethernet	IEEE 802.3ab -standardin mukainen tekniikka, jolla Ethernet-kehyksiä voidaan siirtää yhden gigabitin sekuntinopeudella.
Hyperkuutio	n-ulotteinen rakenne, jossa kukin solmu on yhteydessä n muuhun solmuun
Infiniband	Kytkinpohjainen verkkotekniikka, joka mahdollistaa tiedonsiirron jopa 96 gigabitin sekuntinopeudella.
I/O-operaatio	Operaatio, jolla siirretään tietoa tietokonelaitteiston ja sen komponenttien (esim. hiiri, näppäimistö, yms.) välillä.
IP	*Internet Protocol*. TCP/IP-mallin Internet-kerroksen protokolla, joka huolehtii IP-tietoliikennepakettien perille toimittamisesta pakettikytkentäisessä verkossa.
IP-osoite	Numeerinen osoite, jota käytetään IP-verkkoon liitettyjen laitteiden erottamiseen toisistaan.
Kääntäjä	Ohjelma, joka muuttaa ohjelmointikielellä kirjoitetun lähdekoodin konekieliseksi, suoritettavaksi versioksi.
Klusteri	Tietotekniikassa klusterilla tarkoitetaan useammasta tietokoneesta koostuvaa järjestelmää, jossa koneet on yhdistetty tietoliikenneverkolla ja käyttöjärjestelmätasolla toisiinsa.
Laskentatehtävä	Yksittäinen MPI-viestintäryhmän prosessi.
Liukuluku	Tietokoneissa käytetty esitystapa reaaliluvuille.

MMX-laajennus	Intelin vuonna 1997 Pentium-prosessoreihin kehittämä lisäkäskyjen joukko, joilla saatiin tehokkuutta lisättyä erityisesti multimedian käsittelyyn.
MPI	*Message Passing Interface*. Standardin asemassa oleva viestinvälitykseen perustuva rinnakkaisten ohjelmien kirjoittamiseen tarkoitettu ohjelmointikirjasto.
Myrinet	Lähiverkkotekniikka, joka siirtää tietoa jopa kahden gigabitin sekuntinopeudella.
Prosessi	Tietokoneen keskusyksikössä ajettava ohjelma, jolla on oma muistiavaruus.
Prosessori	Tietokoneen osa, joka suorittaa tietokoneohjelman sisältämiä konekielisiä käskyjä.
Pseudokoodi	Ohjelmointikielen tapaista koodia, jonka tarkoituksena on piilottaa eri ohjelmointikielten väliset syntaksierot ja jättää jäljelle vain algoritmin perusrakenne.
PXE	Tietokoneen verkkokäynnistyksen mahdollistava tekniikka.
Skripti	Sarja komentoja, joilla automatisoidaan toimintoja ilman ohjelmointikieltä.
Solmu	Klusteriin kuuluva yksittäinen tietokone.
Suoritin	kts. Prosessori.
SQL	*Structured Query Language*. Standardoitu kyselykieli, jolla relaatiotietokantaan voi tehdä erilaisia hakuja, muutoksia ja lisäyksiä.
Topologia	Useista komponenteista koostuvan verkon perusrakenne, eli tapa jolla verkon osat on liitetty toisiinsa.

SISÄLLYS

TIIVISTELMÄ

ABSTRACT

SANASTO

1 JOHDANTO

Rinnakkaisten ohjelmistojen tarve kasvaa jatkuvasti. Nykyään jo useissa kodeissa on tietokoneita, joissa suorittimia on kaksi, neljä tai jopa enemmän. Yksi syy rinnakkaisuuden kasvavaan suosioon on se, että tulee huomattavasti halvemmaksi valmistaa tietokone, jossa on useita samanlaisia suorittimia, kuin valmistaa yksi suoritin, jonka suorituskyky yltäisi vastaavalle tasolle. Tulevaisuudessa suorittimien määrä tietokoneissa tulee kasvamaan merkittävästi. Kalifornian yliopiston professori David Pattersonin mukaan 64-ytimisten suorittimien voidaan odottaa tulevan tuotantoon jo vuonna 2015 [Merritt].

Jotta tietokoneen useita suorittimia voitaisiin hyödyntää tehokkaasti, tarvitaan rinnakkaisia ohjelmistoja. Rinnakkaisessa ohjelmassa ainakin osa sen vaatimasta laskennasta voidaan jakaa toisistaan riippumattomiin osiin ja suorittaa samanaikaisesti useassa prosessorissa. Rinnakkaisten ohjelmistojen toteuttamiseksi on olemassa useita erilaisia ohjelmointimalleja, joista tässä työssä tutustutaan yhteen syvällisemmin.

MPI (Message Passing Interface) on niin sanottu de facto -standardi viestinvälityskirjastosta rinnakkaisille järjestelmille. Viestinvälitys on toistaiseksi lähes ainoa tapa kirjoittaa tehokasta ja skaalautuvaa koodia useammalle kuin kymmenelle prosessorille [Haataja, s. 8]. MPI-viestinvälityskirjaston mahdollisesti tärkein etu muihin viestinvälityskirjastoihin verrattuna on se, että MPI on yliopistojen, tutkimuslaitosten ja tietokonevalmistajien yhdessä sopima standardi.

Massiivista laskentaa vaativien tieteellisten ongelmien ratkaisuun käytetään usein ns. supertietokoneita. Nykyään kuitenkin klustereista on tullut niille merkittävä kilpailija. Klusterilla tarkoitetaan järjestelmää, jossa useita tietokoneita on yhdistetty verkkoyhteydellä toisiinsa, ja jotka jonkin ohjelmiston avulla yhdistetään yhdeksi suuremmaksi järjestelmäksi. Klustereiden suuri etu perinteisiin supertietokoneisiin verrattuna on se, että klusteri voidaan koota tavallisista PC-tietokoneista, jolloin sille saadaan erinomainen hinta/suorituskyky-suhde. Klustereissa käytetään usein ilmaisia Unix-pohjaisia käyttöjärjestelmiä.

Tämä insinöörityö on osa Metropolia Ammattikorkeakoulun ohjelmistotekniikan opintoja. Työssä tutustutaan rinnakkaisiin laitearkkitehtuureihin sekä rinnakkaisohjelmointiin. Testiympäristöksi rakennetaan kahdeksasta PC-tietokoneesta korkean suorituskyvyn klusteri. Rinnakkaisista ohjelmointimalleista yhtä tarkastellaan tarkemmin ja rinnakkaistetaan sen avulla sarjamuotoinen algoritmi.

2 RINNAKKAISTIETOKONEET

Rinnakkaisuutta esiintyy tietokoneissa useilla tavoilla. Rinnakkaistietokone voi koostua yhdestä tietokoneesta, jossa on useita suoritinytimiä, tai useista tietokoneista, jotka on yhdistetty tiedonsiirtoverkolla toisiinsa. Tässä luvussa käydään läpi rinnakkaisiin tietokoneisiin liittyviä perusasioita ja tutustutaan rinnakkaisiin arkkitehtuureihin.

2.1 Mitä rinnakkaisuudella tarkoitetaan

Rinnakkaisuudella tarkoitetaan sitä, että useampia asioita tehdään samalla ajanhetkellä. Tietokoneissa esiintyy sekä näennäistä että todellista rinnakkaisuutta. Näennäisellä rinnakkaisuudella tarkoitetaan yhden prosessorin järjestelmää, jossa kulloinkin suorituksessa olevaa ohjelmaa vaihdellaan, jolloin käyttäjä saa kuvan usean ohjelman suorituksesta samanaikaisesti. Näennäistä rinnakkaisuutta esiintyy tosin myös useamman prosessorin järjestelmissä, sillä lähes poikkeuksetta prosesseja on käynnissä enemmän kuin koneessa on prosessoreja. Todellisella rinnakkaisuudella taas tarkoitetaan järjestelmää, jossa on useampia prosessoreita ja eri prosessoreilla suoritetaan eri prosesseja samanaikaisesti.

Näennäisen rinnakkaisuuden hyödyt

Vaikka varsinaista laskentatehoa ei näennäisen rinnakkaisuuden keinoin saadakaan lisättyä, saadaan kuitenkin prosessoriaikaa käytettyä tehokkaammin hyödyksi. Lähes kaikille sovellusohjelmille pätee, että ne kuluttavat suurimman osan ajasta syötteen odottamiseen käyttäjältä tai jonkin I/O-operaation valmistumiseen. Varsinaista laskentaa tehdään suorittimella vain murto-osa ohjelman suoritusajasta. Näennäisen rinnakkaisuuden avulla saadaan ohjelman odotusaikana suoritukseen muita ohjelmia, jolloin prosessori saadaan hyötykäyttöön myös silloin. Järjestelmää, jossa ajetaan use-

ampaa ohjelmaa samalla suorittimella, kutsutaan moniajavaksi järjestelmäksi (vastakohta yksiajokäyttöjärjestelmälle). [Haikala, s. 30.]

Todellisen rinnakkaisuuden hyödyt

Useammalla suorittimella saadaan luonnollisesti järjestelmän laskentatehoa kasvatettua. Tällöin suoritukseen saadaan samanaikaisesti useampia ohjelmia. Laskentatehon kasvatuksen tarve onkin usein syy useamman rinnakkaisen suorittimen käyttöön. Yksittäisen ohjelman suorituskyvyn parantuminen suorittimien määrän kasvaessa riippuu kuitenkin siitä, onko rinnakkaisuus otettu ohjelmassa huomioon ja ovatko ohjelman käsittelemät laskennalliset ongelmat ylipäätään rinnakkaistettavissa.

Jokainen järjestelmään lisätty suoritin lisää mm. väylien käyttöastetta siinä määrin, että uuden suorittimen antama tehonlisäys on usein vain esimerkiksi 90 prosenttia edellisen lisätyn suorittimen antamasta lisätehosta. Tästä syystä suorittimien lukumäärä harvoin ylittää kahdeksaa, jos suorittimet jakavat saman keskusmuistin. On kuitenkin olemassa myös ns. massiivisesti rinnakkaisia järjestelmiä, joissa suorittimia voi olla jopa kymmeniä tuhansia. Näissä on kuitenkin edelliseen nähden se olennainen ero, että jokaisella suorittimella on niissä oma muistinsa. [Haikala, s. 40.]

2.2 Mihin suurta laskentatehoa tarvitaan

On olemassa jatkuva tarve suuremmalle laskentateholle, kuin mitä nykyisellä tekniikalla on mahdollista saavuttaa. Esimerkkeinä tästä voidaan mainita numeerinen simulointi ja tieteellisten ongelmien ratkaisu. Monimutkaisten ongelmien laskennallinen ratkaiseminen vaatii usein erittäin suuria määriä saman laskutehtävän toistamista suurelle datamäärälle. Nämä laskutoimitukset täytyy saada suoritetuksi kohtuullisessa ajassa. Tuotantoympäristössä monimutkaiset laskennat ja simuloinnit täytyy saada suoritetuksi usein sekunneissa tai minuuteissa (jos mahdollista). Esimerkiksi kaksi viikkoa kestävä simulaatio on aivan liian pitkä, jotta suunnittelija voisi työskennellä tehokkaasti [Wilkinson, s. 3].

Osa laskentatehtävistä on luonteeltaan sellaisia, että ne on saatava suoritetuksi tietyn aikarajan puitteissa. Esimerkiksi jos huomisen sääennusteen laskemiseen kuluu aikaa kaksi tai kolme vuorokautta, on selvää, että tulokset ovat silloin jo merkityksettömiä.

Viimeaikoina yksittäisten suorittimien suorituskyky ei ole kasvanut enää entiseen malliin, vaan suorituskykyä on parannettu lisäämällä tietokoneisiin useampia suorittimia. Tämä taas johtaa rinnakkaisohjelmoinnin kasvavaan tarpeeseen.

2.3 Rinnakkaistietokoneiden tyypit

Yleisesti käytetty, moniprosessorijärjestelmien vertailuun tarkoitettu menetelmä on ns. Flynnin taksonomia. Se jakaa moniprosessorijärjestelmät neljään luokkaan sen mukaan, miten ne käsittelevät käskyjä ja dataa.

SISD (Single Instruction, Single Data)

SISD-järjestelmiä ovat yleisesti yhden prosessorin koneet, jotka soveltavat yhtä käskysarjaa yhteen dataan kerrallaan [Quinn, s.55].

SIMD (Single Instruction, Multiple Data)

SIMD-järjestelmissä prosessorit soveltavat samaa käskysarjaa useampaan dataan samanaikaisesti. Tällaisesta järjestelmästä on hyötyä erityisesti esimerkiksi kuvankäsittelyssä, jossa kuva voidaan jakaa useiksi paloiksi ja suorittaa sama tehtävä kaikille kuvan osille samanaikaisesti. Esimerkiksi x86-arkkitehtuurin prosessoreissa on lisäkäskyjä, jotka käsittelevät useampia data-alkioita kerralla. Ensimmäinen näistä oli Intelin MMX-laajennus. [Quinn, s. 55.]

MISD (Multiple Instruction, Single Data)

MISD-järjestelmissä sovelletaan samanaikaisesti useita käskysarjoja samaan dataan. Tämäntyyppiset järjestelmät ovat hyvin harvinaisia, eikä yhtään MISD-järjestelmää ole tiettävästi ollut massatuotannossa [Viitanen]. Tätä luokitusta voidaan kuitenkin käyttää erittäin suurta luotettavuutta tarvitsevissa järjestelmissä (esimerkiksi lentokoneissa), joissa samat laskutoimitukset suoritetaan usealla suorittimella vikasietoisuuden parantamiseksi. Toinen mahdollinen käyttötarkoitus voisi olla esimerkiksi salauksen purkaminen useammalla algoritmilla samanaikaisesti. [Quinn, s. 55 - 56.]

MIMD (Multiple Instruction, Multiple Data)

MIMD on luokista yleisin. Se sisältää kaikki nykyaikaiset useampia suorittimia käsittävät järjestelmät, joissa eri suorittimet voivat suorittaa eri käskysarjoja itsenäisesti [Viitanen]. Tähän luokkaan voidaan tosin lukea myös esim. järjestelmät, joissa on erillinen liukulukusuoritin (kuten x86-perhe aina 8086:sta alkaen) [Quinn, s. 56].

2.4 Laitearkkitehtuurit

Rinnakkaistietokoneissa laitearkkitehtuurit voidaan jakaa useampaan luokkaan niiden käyttämien suoritin- ja muistiarkkitehtuurien mukaan.

SMP (Symmetric Multiprocessing)

SMP on tietokonearkkitehtuuri, jossa kaksi tai useampi samanlainen prosessori käyttää yhtä yhteistä muistia ja ovat yhteydessä siihen yhden yhteisen väylän kautta. SMP on nykyään käytetyin usean prosessorin arkkitehtuuri. Laskentatehoa voidaan tällaisessa järjestelmässä lisätä kasvattamalla prosessoreiden määrää. Koska prosessorit käyttävät samaa väylää, lisääntyy prosessoreita lisättäessä myös väyläliikenne. Tämä rajoittaakin prosessorien maksimimäärän n. 16 - 64 kappaleeseen riippuen toteutuksesta. [Stallings, s. 169.]

MPP (Massively Parallel Processing)

MPP on tietokonearkkitehtuuri, jossa kaksi tai useampi samanlainen prosessori käyttää erillisiä muisteja, mutta kuitenkin suorittaa samaa ohjelmaa rinnakkain. MPP- ja SMP-arkkitehtuurien suurin ero onkin se, että MPP-järjestelmässä jokaisella prosessorilla on oma muistinsa. Ero MPP-järjestelmän ja klusterin välillä taas on se, että MPP-järjestelmässä prosessorit ovat yhdessä koneessa, kun taas klusteri on joukko erillisiä koneita omine prosessoreineen. [Barney.]

Klusterit

Klusteri on useammasta tietokoneesta (kutsutaan usein solmuiksi, engl. node) ja niitä yhdistävästä verkosta muodostuva kokonaisuus. Klustereiden etuna muihin järjestelmiin verrattuna on muun muassa erinomainen skaalautuvuus. Klusteriin voidaan hyvin helposti liittää uusia koneita ja näin laajen-

taa järjestelmää. Klustereissa myös hinta/suorituskyky-suhde on erinomainen. Vaikka klusteri koottaisiin kalliista komponenteista, on sen hinta halvempi kuin vastaavan suorituskyvyn omaavalla supertietokoneella. Klusterit voidaan karkeasti jakaa kahteen osaan niiden käyttötarkoituksen mukaan: korkean saatavuuden (engl. High Availability, HA) ja korkean suorituskyvyn (engl. High Performance Computing, HPC) klustereihin.

Korkean saatavuuden klustereissa useampi palvelin vastaa saman palvelun saatavuudesta. Yhden hajotessa toinen ottaa tehtävät hoitaakseen. Tällaisella järjestelmällä saadaan helposti myös jaettua palvelun kuormitusta.

Korkean suorituskyvyn klustereissa solmut työskentelevät koordinoidusti yhden ongelman ratkaisemiseksi tehokkaasti. Usein yksi klusterin solmuista toimii klusterin hallintakoneena ja muut solmut vain tekevät, mitä hallintakone käskee. Näin hallintakoneella voidaan esimerkiksi raskasta laskentaa vaativa simulointi jakaa rinnakkain suoritettaviin paloihin ja lähettää muille solmuille käsky suorittaa laskenta halutulle datalle. Klusteri näyttääkin käyttäjän näkökulmasta lähinnä yhdeltä tehokkaalta tietokoneelta. Klusterin solmujen ei ole välttämätöntä olla keskenään yhtä tehokkaita, mutta työmäärien jakaminen tasaisesti tulee tällöin huomattavasti vaikeammaksi. Jokaisella klusterin solmulla on oma muistinsa, eivätkä ne näe toistensa muisteihin. Suurin ero SMP-järjestelmiin on siinä, että prosessorien määrää ei ole rajoitettu, koska prosessorit eivät käytä samaa muistia yhteisen väylän kautta. [Wilkinson, s. 26 - 28; Stallings, s. 590.]

2.5 Muistiarkkitehtuurit

Rinnakkaisjärjestelmissä on käytössä useampia erilaisia muistiarkkitehtuureja. Tässä luvussa kuvataan näitä arkkitehtuureja ja niiden välisiä eroja.

Jaettu muisti (Shared memory)

Jaetun muistin monisuoritinjärjestelmiä on olemassa hyvin monenlaisia, mutta kaikille niille on yhteistä, että jokainen suoritin omaa pääsyn kaikkeen muistiin globaalin osoiteavaruuden kautta. Suorittimet voivat siis toimia itsenäisesti, mutta ne käyttävät samoja muistiresursseja. Tämä johtaa myös siihen, että yhden suorittimen tehdessä muutoksia muistiin muutokset ovat myös muiden suorittimien nähtävissä. Yhteisen muistin järjestelmät voidaan jakaa kahteen alaluokkaan muistin hakuajan mukaan. Nämä luokat ovat

UMA (Uniform Memory Access) ja NUMA (Non-Uniform Memory Access). [Viitanen.]

UMA-luokkaan kuuluvat SMP-koneet (Symmetric Multiprocessor) ja yleisesti järjestelmät, joissa on identtisiä suorittimia. Jokaisella suorittimella on samanlainen pääsy muistiin ja sama muistin hakuaika.

NUMA-arkkitehtuuri muodostetaan yleensä yhdistämällä useita SMP-järjestelmiä. Kaikilla SMP-koneilla on pääsy toisten koneiden muisteihin. Suorittimen hakuaika omaan muistiin on nopeampaa kuin muiden suorittimien muisteihin. [Viitanen.]

Hajautettu muisti (Distributed memory)

Kuten yhteisen muistin järjestelmät myös hajautetun muistin järjestelmät voivat vaihdella suuresti, mutta kaikilla niillä on tiettyjä samoja piirteitä. Hajautetun muistin järjestelmässä jokaisella suorittimella on oma muistinsa. Suorittimet toimivat itsenäisesti eivätkä häiritse toisiaan yhteiseen muistiin kohdistuvalla liikenteellä. Suoritinten muistien yhdistäminen vaatii kommunikointiverkon (yksinkertaisimmillaan esim. Ethernet-verkon). Suorittimilla ei ole myöskään minkäänlaista yhteistä osoiteavaruutta. Kun jokin suoritin tarvitsee pääsyn toisen suorittimen muistiin, on ohjelmoijan vastuulla määrittää koska ja miten dataa siirretään. Myös prosessien synkronointi on ohjelmoijan vastuulla. [Stallings, s. 583.] Klusterit ovat tyypillisesti hajautetun muistin järjestelmiä.

Edellisten yhdistelmät

Tämän päivän suurimmat ja tehokkaimmat tietokoneet käyttävät molempia edellä mainittuja muistiarkkitehtuureja. Jaettua muistia käytetään SMP-järjestelmissä, joissa jokainen saman järjestelmän suoritin pääsee käsiksi samaan muistiavaruuteen. Hajautettua muistia käytetään SMP-järjestelmien yhdistämiseen. Koska jokainen SMP-järjestelmä näkee vain oman muistinsa, tarvitaan kommunikaatioverkko, jonka kautta dataa saadaan siirrettyä SMP-järjestelmien välillä. [Wilkinson, s. 15.]

3 KORKEAN SUORITUSKYVYN KLUSTERIT

Korkean suorituskyvyn klusterilla tarkoitetaan useista, tietoliikenneverkolla toisiinsa yhdistetyistä tietokoneista koottua järjestelmää, jota käytetään kuin yhtä erittäin suuren laskentakapasiteetin omaavaa tietokonetta. Korkean suorituskyvyn klusterin pääasiallinen käyttötarkoitus onkin yleensä raskas rinnakkaislaskenta. Tässä luvussa tutustutaan tällaisen järjestelmän asentamiseen.

3.1 Yleistä

Korkean suorituskyvyn klusterin käyttöjärjestelmäksi käy esimerkiksi Linux. Yleisimmät korkean suorituskyvyn klusterit ovat ns. Beowulf-klustereita, jotka koostuvat keskenään samanlaisista PC-koneista, käyttävät Unix-pohjaista käyttöjärjestelmää (esimerkiksi Linux tai BSD) ja siirtävät tietoa solmut toisiinsa yhdistävän Ethernet-verkon avulla. Tässä työssä perehdytään tarkemmin yhden käyttöjärjestelmän asennukseen. Tarkasteltavaksi käyttöjärjestelmäksi on valittu ilmainen Linux-jakelu Rocks Cluster, joka pohjautuu CentOS-Linux-käyttöjärjestelmään.

3.2 Rocks Cluster

Rocks Cluster on avoimeen lähdekoodiin perustuva Linux-jakelu, joka on tarkoitettu erityisesti klustereiden rakentamiseen. Käyttöjärjestelmän kehittäjien tärkein päämäärä on ollut klustereiden asentamisen ja käytön helpottaminen. Sen avulla voi helposti rakentaa laskennallisia klustereita sekä esimerkiksi ns. näyttöseiniä.

3.2.1 Tuetut laitteistot

Prosessoriarkkitehtuureista tuettuja ovat x86 ja x86_64. Verkoksi käy normaali Ethernet-verkko, mutta myös esimerkiksi Myrinet ja Infiniband ovat tuettuja.

3.2.2 Laitteistovaatimukset

Laitteistovaatimustensa puolesta Linux-jakelu Rocks Cluster sopii käyttöjärjestelmäksi myös vanhemmille PC-koneille erittäin hyvin.

Hallintakoneen laitteistovaatimukset:

- **Kiintolevy:** 30 Gt.
- **Muisti:** 1 Gt.
- **Verkko:** kaksi fyysistä porttia.
- **BIOS-käynnistysjärjestys:** CD, Kiintolevy.

Muiden solmujen laitteistovaatimukset:

- **Kiintolevy:** 30 Gt.
- **Muisti:** 1 Gt.
- **Verkko:** yksi fyysinen portti.
- **BIOS-käynnistysjärjestys:** CD, PXE, Kiintolevy.

3.2.3 Testilaitteisto

Rinnakkaisohjelmien testiympäristönä käytettävä klusteri rakennettiin kahdeksasta HP d530 -pöytätietokoneesta, jotka yhdistettiin toisiinsa Gigabit Ethernet -verkolla (kuva 1).

Kuva 1: Testilaitteistona käytetty klusteri.

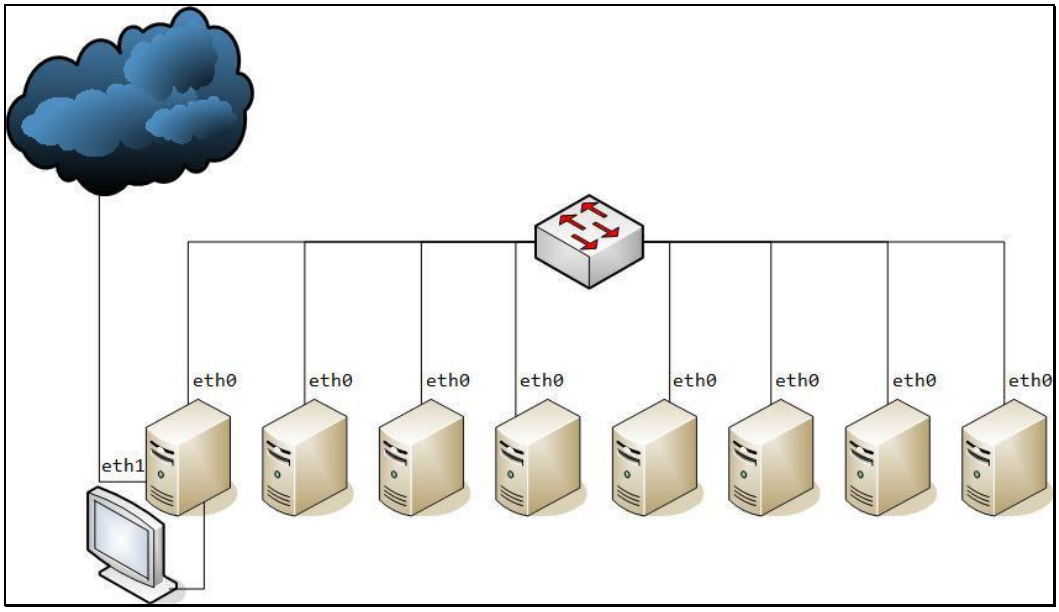
Tarkempi kuvaus laitteiston solmuista:

- **Prosessori:** Intel Pentium 4, 2666 MHz.
- **Kiintolevy:** 40 Gt.
- **Muisti:** 1 Gt DDR 333 MHz.
- **Verkko:** yksi Gigabit Ethernet -portti (Hallintakoneessa myös yksi Fast Ethernet -portti Internet-yhteyttä varten).

Kaikki klusterin koneet ovat siis kokoonpanoltaan täysin identtisiä (lukuunottamatta hallintakoneen toista verkkokorttia). Klusterin koneet on yhdistetty toisiinsa Gigabit Ethernet -yhteydellä.

3.2.4 *Fyysinen rakenne*

Rocks-klusterin voi asentaa yhtä hyvin räkkipalvelimiin kuin ihan normaaleihin pöytäkoneisiin. Koneiden täytyy kuitenkin olla yhdistetty toisiinsa kuvan 2 mukaisesti.



Kuva 2: Klusterin fyysinen rakenne.

Klusterin hallintakoneesta verkkokortti, jonka Linux määrittää eth0:ksi, kytketään samaan verkkoon muiden solmujen kanssa. Verkkokortti, jonka Linux määrittää eth1:ksi, kytketään ulkoverkkoon.

Näppäimistön, hiiren ja näytön tarvitsee vain hallintakoneeseen, ja siihenkin ne ovat välttämättömät vain asennuksen ajaksi. Periaatteessa solmuihin ei tarvita kuin virta- ja verkkokaapelit. Etäyhteys hallintakoneelle onnistuu esimerkiksi SSH-yhteydellä.

3.2.5 *Käyttöjärjestelmän asentaminen*

Rocks-käyttöjärjestelmän asennuslevyt ja yksityiskohtaiset asennusohjeet saa ladattua Internetistä, osoitteesta

http://www.rocksclusters.org/.

Rocks-käyttöjärjestelmä on jaettu useille asennuslevyille, jotta kenenkään ei tarvitse ladata ja asentaa osia, joita ei itse tarvitse. Minimivaatimuksena on kuitenkin *Base, Kernel/Boot, Web Server*, *OS 1* ja *OS 2*-levyt. Nämä kaikki sisältyvät Jumbo-DVD-levyyn, joten helpoimmalla pääsee, kun lataa sen.

Hallintakoneen asennus

Asennus käynnistetään laittamalla ensimmäinen asennuslevy koneeseen ja käynnistämällä kone sieltä. Aloitusruudun ilmestyttyä näytölle kirjoitetaan *build* ja painetaan enter.

Mikäli koneen verkkokortti ei saa DHCP-palvelimelta itselleen IP-osoitetta, kysyy asennus ensimmäiseksi käytettäviä verkkoasetuksia. Seuraavaksi valitaan paikallisesti tai verkon yli asennettavat levyt. Tämän jälkeen asennusohjelma kysyy vielä joitakin perustietoja liittyen klusteriin ja käytettäviin verkkoasetuksiin. Viimeiseksi vielä osioidaan kiintolevy halutulla tavalla ja lopulta suoritetaan varsinainen asennus. Asennusohjelma on varsin selkeä, ja jokaisella sivulla on vasemmassa reunassa lisätietoa kyseiseen sivuun liittyen.

Muiden solmujen asennus

Hallintakoneen asennuksen jälkeen voidaan asentaa klusterin muut solmut. Tämä tapahtuu kirjoittamalla hallintakoneen konsolissa komento

insert-ethers.

Listalta valitaan lisättävien koneiden tyypiksi *Compute*. Ohjelma ottaa vastaan tietokoneiden lähettämät DHCP-pyynnöt ja tallentaa tiedot niistä omaan SQL-tietokantaansa ja käynnistää asennuksen.

Asennuksen käynnistyttyä koneessa, voi sen etenemistä seurata ottamalla siihen etäyhteyden komennolla

rocks-console compute-X-Y,

jossa X ja Y ovat samat kuin insert-ethers-ohjelmassa (esim 0-0).

Testiklusterin asennus

Kahdeksan koneen testiklusterin asennuksiin meni aikaa noin tunti. Erityisen käteväksi asennuksen tekee se, että kun hallintakone on asennettu, voidaan kaikki loput solmut asentaa rinnakkain. Asennuksesta selvittiin ilman minkäänlaisia ongelmia.

4 RINNAKKAISOHJELMOINTI

Rinnakkaisohjelmointi on nopeasti yleistyvä ohjelmistokehityksen ala. Suorittimien määrän kasvaessa tietokoneissa ei sarjamuotoisilla ohjelmistoilla enää saada täyttä hyötyä koneen suoritustehosta. Rinnakkaisohjelmointiin on olemassa useita erilaisia malleja, joita käsitellään tässä luvussa.

Rinnakkaisohjelmointi tuo mukanaan myös haasteita. Rinnakkaisissa ohjelmissa esiintyvien virhetilanteiden selvittäminen on usein hyvin hankalaa, erityisesti koska ohjelmaa suoritetaan rinnakkain useammassa prosessorissa, eikä rinnakkaisten ohjelman kohtien suoritusjärjestystä voida ennalta ennustaa. Rinnakkaisuuden toteuttaminen ohjelmissa jää yleensä kokonaan ohjelmoijan vastuulle, sillä perinteisten ohjelmointikielten rakenteet eivät ota rinnakkaisuutta huomioon.

4.1 Rinnakkaisohjelmoinnin rajoitteet

Tässä luvussa esiintyvät asiat perustuvat lähteeseen: [Wilkinson, s. 6 - 12 ja 79 - 80].

Rinnakkaistuvat ja ei-rinnakkaistuvat ohjelmat

On olemassa laskennallisia ongelmia, joiden ratkaisu on rinnakkaistettavissa helposti, ja toisaalta ongelmia, joiden ratkaisua ei ole mahdollista rinnakkaistaa lainkaan. Luonnollisesti myös välimuotoja näiden edellä mainittujen välillä on olemassa.

Rinnakkaistuvat ongelmat jakautuvat selkeisiin osaongelmiin, joiden välillä ei ole paljoa laskentaa hidastavia riippuvuuksia. Rinnakkaislaskentaa voivat hidastaa useat eri tekijät, kuten laskentakäskyjen datariippuvuudet ja ohjelman rakenteesta johtuvat ongelmat. Myös suurten datamäärien siirtäminen prosessorien välillä voi muodostua suorituskyvyn pullonkaulaksi riippuen siitä, miten prosessorit ovat yhteydessä toisiinsa.

Esimerkkinä erittäin helposta rinnakkaistuvuudesta voidaan mainita kuvankäsittely, jossa esiintyy paljon sellaista laskentaa, joka voidaan suoraan jakaa täysin toisistaan riippumattomiin osiin. Laskentaan osallistuvien prosessorien ei tarvitse keskustella keskenään lainkaan, vaan ne voivat päätellä esimerkiksi omasta järjestysnumerostaan kuva-alueen, jota ne käsittelevät.

Tällaisia ongelmia kutsutaan naurettavan rinnakkaisiksi (Embarassingly Parallel).

Kaikki laskentaongelmat eivät kuitenkaan ole rinnakkaistuvia. Esimerkkinä voidaan mainita Fibonaccin lukujonon (0, 1, 1, 2, 3, 5, 8, 13, 21, …) laskeminen käyttäen kaavaa

$$F(n) = \begin{cases} 0 \text{ , kun } n = 0 \\ 1 \text{ , kun } n = 1 \\ F(n-1) + F(n-2) \text{ , kun } n > 1 \end{cases}$$

Tällaisen ongelman rinnakkainen ratkaisu on mahdotonta toteuttaa, sillä jokainen laskun osa on riippuvainen kahdesta edellisestä tuloksesta.

Rinnakkaistamisella saavutettava ohjelman nopeutuminen

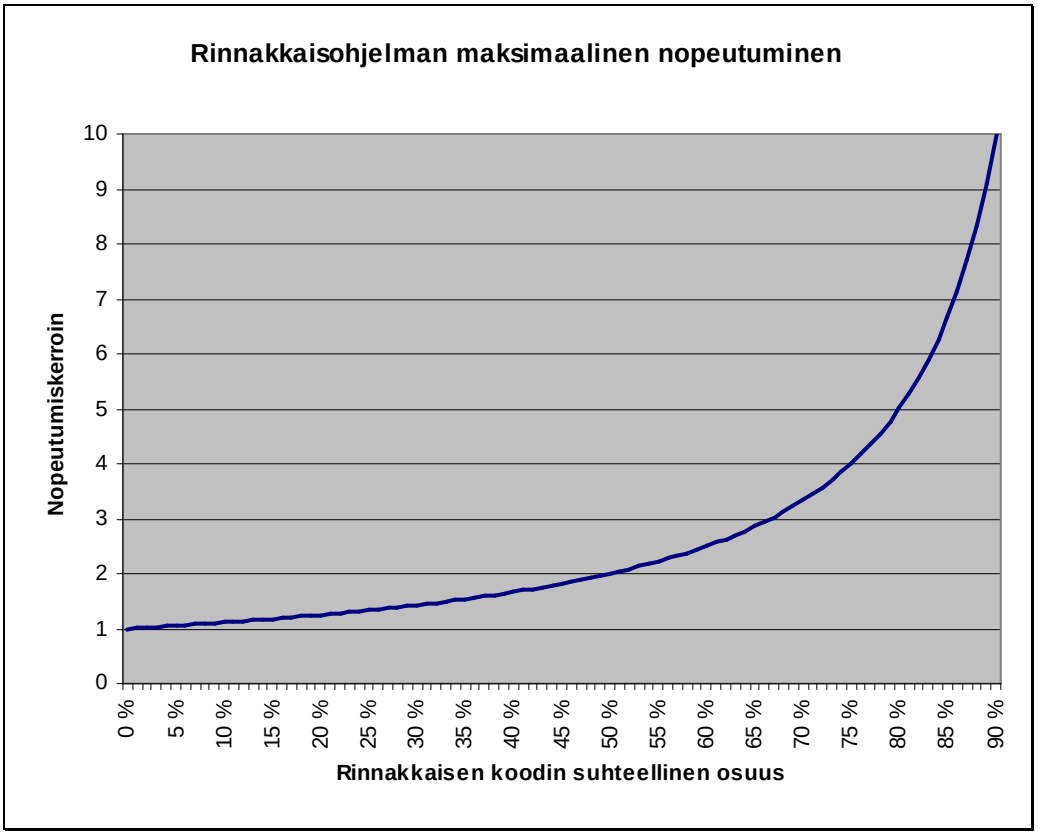
Vaikka ohjelman osien rinnakkain suorittamisella saadaan aikaan säästöjä ohjelman suoritusajassa, ohjelma ei ikinä nopeudu n-kappalleella suorittimia n-kertaiseen nopeuteen.

Amdahlin lain mukaan ohjelman nopeutumismahdollisuudet rinnakkain suoritettuna riippuvat siitä, kuinka suuri osuus koodista voidaan rinnakkaistaa.

Amdahlin laki ilman prosessoreiden määrää:

$$k = \frac{1}{1-P}$$

Yllä olevassa kaavassa k on suurin saavutettava nopeutumiskerroin ja P rinnakkain suoritettavan koodin suhteellinen osuus. Näin saatua suurinta nopeutumiskerrointa on havainnollistettu kuvassa 3. Rinnakkain suoritettavalla koodilla tarkoitetaan sellaista ohjelman osaa, joka suoritetaan rinnakkain useassa prosessorissa eri data-alkioille. Nopeutumiskertoimen ollessa k, ohjelman suoritusaika on $\frac{1}{k}$ alkuperäisestä suoritusajasta.



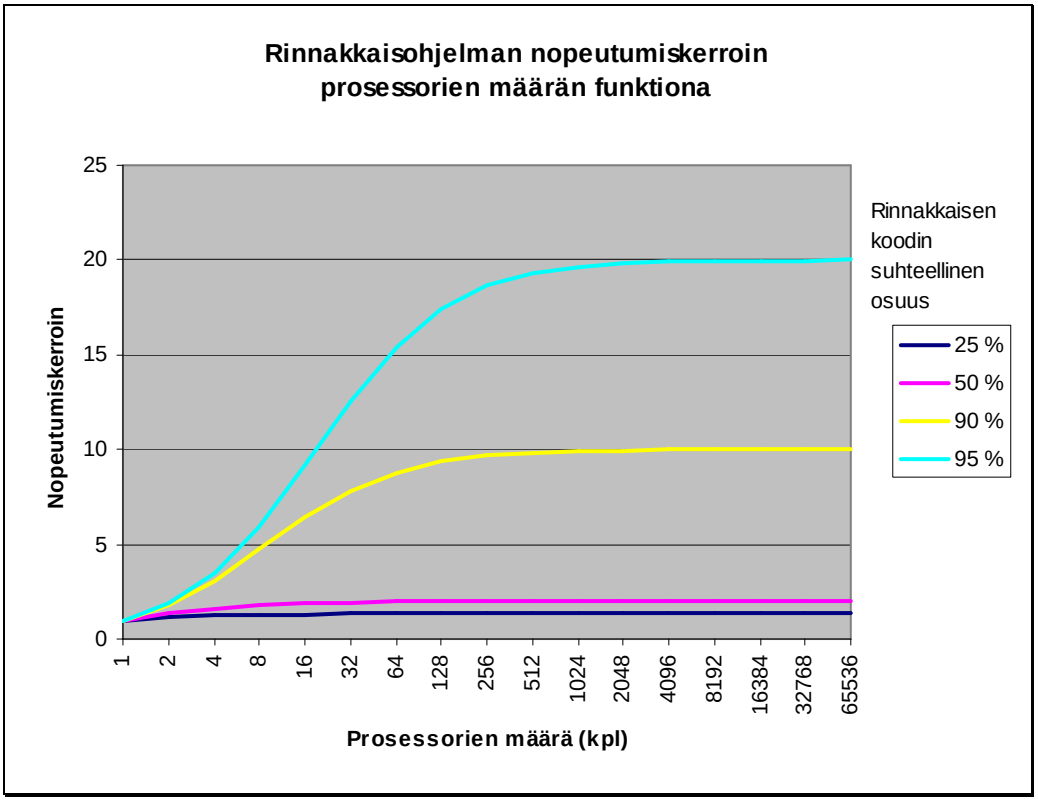
Kuva 3: Rinnakkaisohjelman maksimaalinen nopeutuminen.

Kuvasta 3 havaitaan, että suureen nopeutumiskertoimeen vaaditaan, että hyvin suuri osa koodista on rinnakkain suoritettavaa.

Kun otetaan huomioon myös käytettävien prosessorien määrä, muuttuu kaava muotoon:

$$k = \frac{1}{\frac{P}{N} + S}$$

Yllä olevassa kaavassa k on suurin saavutettava nopeutumiskerroin, N prosessorien määrä, P rinnakkaisen koodin suhteellinen osuus ja S sarjamuotoisen koodin suhteellinen osuus (1-P).



Kuva 4: Rinnakkaisohjelman nopeutumiskertoimia.

Kuvassa 4 on havainnollistettu jälkimmäisellä kaavalla saatavia nopeutumiskertoimia eri rinnakkaisen koodin osuuksilla ja prosessorien määrillä. Kuvasta havaitaan, että prosessorien määrällä on sitä pienempi merkitys, mitä pienempi osa koodista on rinnakkain suoritettavaa.

4.2 Rinnakkaisohjelmointimallit

Rinnakkaisohjelmien suunnittelun ja toteutuksen avuksi on olemassa useita erilaisia ohjelmointimalleja, joiden avulla ohjelman vaatima rinnakkaisuus saadaan toteutettua. Ohjelmointimallit eivät ole sidonnaisia alustaan, jolla ohjelmaa ajetaan. Esimerkiksi viestienvälitysmallia voidaan käyttää myös jaetun muistin järjestelmissä. [Haataja, s. 8.]

Jaetun muistin malli

Jaetun muistin mallille on ominaista, että eri prosessit voivat kirjoittaa ja lukea muistista asynkronisesti. Muistin korruptoitumisen estämiseksi on tällöin kuitenkin käytettävä esimerkiksi semaforeja, ja muistin käsittely vaatii normaaliakin huolellisempaa ohjelmointia. [Barney.]

Säiemalli

Säiemallissa ohjelma suoritetaan useana erillisenä säikeenä. Tällöin yhden säikeen jäädessä odottamaan esimerkiksi I/O-operaation valmistumista, muut säikeet voivat jatkaa toimintaansa normaalisti. Toisin sanoen ohjelman toisistaan riippumattomat osat saadaan suoritettua samanaikaisesti. Koska säikeet keskustelevat jaetun muistin avulla, täytyy ohjelmoijan huolehtia siitä, ettei useampi säie yritä samanaikaisesti käsitellä samaa muistialuetta. [Barney.]

Viestinvälitysmalli

Viestinvälitysmallissa ohjelmaa voidaan suorittaa fyysisesti samassa tietokoneessa olevissa prosessoreissa tai esimerkiksi Internetin kautta toisiinsa yhteydessä olevissa koneissa. Yhteistä kummallekin toteutukselle on, että prosessorit ovat yhteydessä toisiinsa ja jakavat yhteisen viestinvälitysketjun, jonka avulla eri prosessoreissa etenevät operaatiot voivat viestiä toistensa kanssa. Viestienvälitys on aina ennalta sovittua. Vaikka viestinvälitysmallia tyypillisesti käytetään hajautetun muistin järjestelmissä, voidaan sitä käyttää myös jaetun muistin järjestelmissä. [Barney.] Tässä työssä käsitellään MPI-viestinvälitysstandardia tarkemmin seuraavissa luvuissa.

Tiedonrinnastusmalli

Tiedonrinnastusmallia käytetään useimmiten käsiteltäessä taulukko- tai kuutiomuotoisia tietorakenteita. Jaetun muistin järjestelmässä käsiteltävä tietorakenne voidaan pitää yhtenäisenä ja määrittää kullekin prosessille tietorakenteesta tietty lohko. Hajautetun muistin järjestelmässä lohkot siirretään eri koneille prosessoitavaksi. Tällainen ohjelmointimalli sopii erityisen hyvin laskentaan, jossa eri lohkoissa tapahtuvilla muutoksilla ei ole vaikutusta toisiinsa. [Barney.]

Hybridimalli

Erilaisia yhdistelmiä edellä kuvatuista malleista sanotaan usein ns. hybridimalleiksi.

4.3 Rinnakkaisohjelmien ongelmatilanteita

Vaikka rinnakkaisuudella saadaan monesti ohjelman suoritustehokkuutta kasvatettua, tuo se mukanaan myös ongelmia, joita sarjamuotoisissa ohjelmissa ei esiinny. Näistä yleisimpiä käydään läpi tässä luvussa.

Poissulkemisongelma

Sarjamuotoista ohjelmaa suoritetaan käsky käskyltä täsmälleen siinä järjestyksessä, kuin ne ovat ohjelmakoodissa. Näin ollen ohjelmoijan on helppo varmistaa, että esimerkiksi erilaisten tietorakenteiden käsittely tapahtuu oikein eikä ongelmia tule. Rinnakkaisessa ohjelmassa yksittäisten koodirivien suoritusjärjestystä eri prosessien välillä ei voida ennustaa. Näin ollen syntyy helposti tilanteita, joissa useampi prosessi yrittää esimerkiksi muokata kaikille prosesseille yhteistä tietorakennetta samanaikaisesti, jolloin tuloksena on usein tietorakenteen korruptoituminen ja väärä tulos laskennassa. Tällaisessa tilanteessa ei ole merkitystä edes sillä, suoritetaanko ohjelma useammalla prosessorilla vai näennäisesti rinnakkaisena yhdellä prosessorilla. Tämä johtuu siitä, että keskeyttävää moniajoa käyttävässä järjestelmässä ohjelman suoritus saattaa keskeytyä milloin tahansa ja ohjaus siirtyä toiselle prosessille. Tällaista ongelmaa kutsutaan poissulkemisongelmaksi (mutual exclusion). [Stallings, s. 208 - 212.]

Poissulkemisongelma ratkaistaan etsimällä ohjelmakoodista ns. kriittiset alueet eli ohjelman osat, joissa poissulkemisongelma voi syntyä. Tällaisten ohjelman kohtien suorittaminen halutaan sallia vain yhdelle prosessille kerrallaan ja niihin pääsyä valvotaan yleensä jonkinlaisen lukitusmekanismin avulla. Itse lukitusmekanismi voidaan toteuttaa joko itse tai voidaan käyttää hyväksi valmiita lukitusmekanismeja kuten semaforeja. [Haikala, s. 91 - 92; Stallings, s. 211 ja 217.]

Synkronointiongelma

Synkronointiongelmalla tarkoitetaan prosessien suorituksen tahdistamiseen liittyvää ongelmaa esimerkiksi tilanteessa, jossa suurempi ongelma on jaettu useammaksi osaongelmaksi mutta osaongelmien ratkaisun jälkeen tarvitaan vielä yksi jakamaton vaihe tulosten yhdistämiseksi. Osaongelmien tulokset yhdistävän prosessin on odotettava, että jokainen osaongelma on saatu ratkaistua ennen kuin se voi aloittaa tulosten yhdistämisen. Synkronointiongel-

ma ratkaistaan käyttämällä erityisiä synkronointiviestejä prosessien välillä. Näin ollen prosessi voi jäädä odottamaan toisen prosessin lähettämää synkronointiviestiä, ennen kuin se jatkaa suoritustaan. [Haikala, s. 95; Stallings, s. 197.]

Lukkiutumisongelma

Lukkiutumisella (deadlock) tarkoitetaan yleensä tilannetta, jossa ohjelmaa suoritetaan useana prosessina, jotka kaikki odottavat jonkin muun samaa ohjelmaa suorittavan prosessin hallussa olevaa resurssia. Koska kaikki prosessit odottavat, ei mikään odotetuista resursseista vapaudu koskaan, ja odottelu jatkuu ikuisesti. Myös prosessien välisiä viestejä voidaan pitää resursseina. Lukkiutumiseen voi siis johtaa myös sellainen ohjelmointivirhe, jossa keskenään kommunikoivat prosessit syystä tai toisesta jäävät kehämäisesti odottamaan viestejä toisiltaan. Lukkiutumisen mahdollisuus voidaan joissain tilanteissa estää esimerkiksi asettamalla resurssin varaamiselle aikaraja, jolloin viimeistään jossain vaiheessa jokin prosessi luopuu varaamastaan resurssista. [Haikala, s. 107 - 108.]

Viestinvälitykseen perustuvassa rinnakkaisohjelmoinnissa lukkiutuminen voi tapahtua esimerkiksi tilanteessa, jossa jokaisen prosessin on sekä lähetettävä että vastaanotettava viestejä joltakin toiselta prosessilta. Tällöin lukkiutuminen voidaan välttää jakamalla prosessit kahteen osaan niin, että toinen puoli lähettää ensin ja vastaanottaa sitten, kun taas toinen puoli vastaanottaa ensin ja lähettää sitten.

Nälkiintymisongelma

Nälkiintyminen (starvation) on läheistä sukua lukkiutumiselle. Nälkiintynyt prosessi ei ole lukkiutunut, mutta sen suoritus ei silti pääse jatkumaan. Näin voi käydä esimerkiksi ohjelmassa, jossa käsitellään kaikille prosesseille yhteistä tietorakennetta lukija- ja kirjoittajaprosessien avulla. Oletetaan, että tietorakenne sallii kerralla useita lukuoperaatioita, mutta kirjoitusoperaatio voidaan suorittaa vain silloin, kun samanaikaisia lukuoperaatioita ei ole. Tällöin kirjoittajaprosessi voi nälkiintyä, mikäli lukijoita tulee jatkuvasti lisää tai jos niitä on hyvin paljon. Tässä esimerkkitapauksessa nälkiintyminen voidaan estää asettamalla tietorakenteelle lisävaatimus, että jos yksikin kirjoittaja on jonottamassa, ei uusia lukijoita päästetä lukuoperaatioon ennen kuin kirjoitusoperaatio on suoritettu. [Haikala, s. 112.]

5 MPI (MESSAGE PASSING INTERFACE)

MPI on yli 120 operaatiota käsittävä viestinvälityskirjasto. MPI määrittelee standardin näille operaatioille, mutta ei niiden toteutusta, aivan kuten ohjelmointikielet on tarkasti määritelty, mutta ei sitä, kuinka kääntäjät toteutetaan. MPI on siis toisin sanoen rajapintamäärittely. Tässä luvussa tutustutaan tarkemmin tähän viestinvälityskirjastoon.

5.1 Yleisesti

MPI on nimensä mukaisesti viestinvälityskirjasto tai pikemminkin ns. de facto -standardi sellaiselle. MPI:sta on monta ilmaista toteutusta vapaasti ladattavissa Internetistä. Eri laitevalmistajat ovat toki tehneet omille kokoonpanoilleen optimaalisia versioita, mutta niitä ei luonnollisesti ole saatavilla ilmaiseksi. Koska MPI on standardi, voidaan ohjelmia ajaa eri MPI:n toteutuksilla ilman koodimuutoksia.

MPI-standardia ei ole virallisesti hyväksytty minkään standardointiorganisaation toimesta, vaan siitä on tullut ns. de facto -standardi, eli se on käytännön kautta saavuttanut merkittävän markkinaosuuden. MPI-standardista on olemassa useita versioita, mutta jokainen niistä on alaspäin yhteensopiva. Siis jos ohjelma on kirjoitettu noudattaen MPI 1 -standardia, se voidaan suorittaa myös MPI 2 -toteutuksella.

5.2 Syitä MPI:n käyttöön

Viestienvälitys on toistaiseksi lähes ainoa tapa kirjoittaa tehokasta ja skaalautuvaa koodia useammalle kuin kymmenelle prosessorille [Haataja, s. 8]. MPI-viestinvälityskirjaston mahdollisesti tärkein etu muihin viestinvälityskirjastoihin verrattuna on se, että MPI on yliopistojen, tutkimuslaitosten ja tietokonevalmistajien yhdessä sopima standardi. MPI:n hyviin puoliin lukeutuu myös suhteellisen korkealla abstraktiotasolla tapahtuva prosessorien välisen viestinnän ohjelmointi, jolloin ohjelmakoodista tulee tiiviimpää ja selkeämpää. Ohjelmointia selkeyttää myös se, että MPI:n viestintäryhmät mahdollistavat prosessorien välisen viestinnän erottamisen mahdollisesta muusta ohjelmassa tapahtuvasta viestinnästä. MPI-kirjastoa myös päivitetään jatkuvasti. Uusin versio standardista on 2.2, joka julkaistiin 4. syyskuuta 2009.

5.3 MPI-ohjelmien kääntäminen

MPI-ohjelmat on helpointa kääntää niin sanottujen kääreskriptien avulla. Kääreskripti on pieni ohjelma, joka liittää mukaan kaikki MPI:n tarvitsemat kirjastot ja valitsimet ja kutsuu sopivaa kääntäjää niillä. Kääreskriptiä kutsutaan aivan kuten vastaavaa kääntäjää muutenkin. Yleisimmin käytetyt kääreskriptit on esitetty taulukossa 1

Taulukko 1: Kääntäjäohjelmien kääreskriptit.

Kieli	Kääntäjä	Kääreskripti
C	gcc	mpicc
C++	g++	mpiCC
Fortran 77	f77	mpif77
Fortran 90	f90	mpif90

Esimerkiksi C-kielistä MPI-ohjelmaa *esimerkki.c* käännettäessä olisi kääreskriptin kutsu seuraavanlainen:

```
mpicc esimerkki.c -o esimerkki.
```

MPI-ohjelmat on kuitenkin mahdollista kääntää myös käyttämättä edellämainittuja kääreskriptejä, kunhan osaa käyttää oikeita valitsimia haluamansa kääntäjän kanssa.

5.4 MPI-ohjelmien suorittaminen

MPI-ohjelmat suoritetaan erillisen käynnistysohjelman avulla. Itse käynnnistysohjelmaa ei ole määritelty vanhemmissa MPI-standardeissa lainkaan, joten se vaihtelee toteutuksittain. Useissa ilmaisissa MPI-toteutuksissa käynnistysohjelma on nimeltään mpirun. MPI-2-standardi suosittelee (mutta ei edellytä) standardin toteuttajia tarjoamaan mpiexec-käynnistysohjelman. [Gropp, s. 28.] Unix-pohjaisissa käyttöjärjestelmissä käytettävä käynnistysohjelma parametrikuvauksineen selviää usein komennolla *man mpi*. Mpirun-ohjelmalle voidaan antaa erilaisia komentoriviparametreja, joista tärkeimpiä ovat:

- -machinefile <machine-file name>
- -np <np>.

Machinefile-parametrilla voidaan määrittää tiedosto, joka sisältää niiden solmujen nimet tai IP-osoitteet, joissa ohjelmaa halutaan ajaa. Np-

parametrilla voidaan määrittää, kuinka monta prosessia käynnistetään. Jos np-parametrin arvo on suurempi kuin machinefile-parametrilla määritetyn tiedoston sisältämien solmujen määrä, käynnistetään osaan solmuista useampia prosesseja.

MPI-ohjelmien suorittamiseksi Linux-järjestelmässä on oltava käynnissä ns. MPI-daemon (mpd), eli taustaprosessina toimiva MPI-palvelu. MPI-daemonin tehtävänä on mm. ohjelmaa käynnistettäessä huolehtia ohjelman käynnistämisestä muissa suoritukseen osallistuvissa prosessoreissa.

6 MPI-OHJELMOINNIN PERUSTEET

Tässä luvussa esitettävät asiat perustuvat MPI-1.3-standardiin [MPI-1.3].

Tässä luvussa käsitellään MPI-kirjaston tärkeimpiä funktioita C- ja C++ -kielissä. C-kielisten MPI-funktioiden nimet alkavat aina sanalla MPI, jonka jälkeen tulee alaviiva ja varsinaisen funktion nimi isolla alkukirjaimella ja loppuosa pienillä kirjaimilla.

Suoritettaessa ohjelmia MPI-ympäristössä laskentaan osallistuu yleensä useita prosesseja. Koska yhdelläkin prosessorilla voidaan suorittaa useampaa MPI-prosessia samanaikaisesti, kutsutaan näitä prosesseja tässä työssä laskentatehtäviksi.

6.1 MPI-ympäristön alustus ja lopetus

Jotta muita MPI-funktioita voidaan käyttää, täytyy ensimmäiseksi alustaa viestintäympäristö. Alla on esitetty viestinnän alustusfunktion prototyyppi

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

Kutsun parametreina voidaan siis välittää MPI:lle osoittimet ohjelman komentoriviargumentteihin. C-kielessä MPI-funktiot palauttavat kokonaislukutyyppisen paluukoodin. Onnistuneen paluukoodin arvo on vakio MPI_SUCCESS.

Viestintäympäristön alustamisen jälkeen on yleensä syytä selvittää, kuinka monta laskentatehtävää työhön osallistuu. Tämä tapahtuu kutsumalla funktiota, jonka prototyyppi on

int MPI_Comm_Size(MPI_Comm comm, int *size).

Ensimmäisenä parametrina annetaan ohjelman alussa yleensä vakio MPI_COMM_WORLD, johon kaikki laskentatehtävät alustuksen yhteydessä sijoitetaan. Parametri size on kokonaislukuosoitin, johon laskentatehtävien määrä tallennetaan.

Viestintäympäristön alustamisen yhteydessä jokaiselle laskentatehtävälle annetaan myös järjestysnumero (väliltä 0…p - 1, laskentatehtävien määrän ollessa p). Järjestysnumeron selvittäminen on oleellista ohjelmoinnin kannalta. Yleinen organisointi on, että yksi laskentatehtävä suorittaa eri asioita kuin toiset. Tämä järjestysnumero voidaan selvittää kutsumalla funktiota, jonka prototyyppi on

int MPI_Comm_rank(MPI_Comm comm, int *rank).

Ensimmäisenä parametrina annetaan viestintäryhmä, johon laskentatehtävä kuuluu ja toisena kokonaislukuosoitin, johon laskentatehtävän järjestysnumero kyseisessä viestintäryhmässä tallennetaan.

MPI-viestintäympäristön käyttö lopetetaan kutsumalla siihen tarkoitettua funktiota. Alla on esitetty viestinnän lopetusfunktion prototyyppi

int MPI_Finalize(void).

Luonnollisesti viestintäympäristön lopetusfunktion kutsumisen jälkeen ei enää saa kutsua mitään MPI-standardissa määriteltyä funktiota.

6.2 Viestintäryhmät

MPI-ympäristössä kaikki laskentaan osallistuvat laskentatehtävät liitetään osaksi viestintäryhmää. Viestintäryhmän ideana on, että kaikki samaan viestintäryhmään kuuluvat laskentatehtävät voivat lähettää ja vastaanottaa viestejä keskenään. MPI-ympäristön alustuksen, eli MPI_Init -kutsun yhteydessä kaikki laskentatehtävät liitetään viestintäryhmään MPI_COMM_WORLD. Joitakin ongelmia ratkaistaessa on kuitenkin loogista jakaa laskentatehtävät pienempiin viestintäryhmiin, jolloin samaan ryhmään kuuluvat laskentatehtävät voivat keskittyä saman osaongelman ratkaisuun. Näin ollen eri viestintäryhmissä voidaan hyödyntää esimerkiksi kollektiivisia viestintäoperaatioita häiritsemättä muissa viestintäryhmissä olevia laskentatehtäviä.

Viestintäryhmä voidaan jakaa pienempiin osiin kutsumalla kaikissa laskentatehtävissä funktiota:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm *newcomm).
```

Parametreilla määritetään, miten viestintäryhmä jaetaan osiin. Ensimmäinen MPI_Comm-tyyppinen parametri määrittää pilkottavan ryhmän. Parametri color määrää uuden ryhmän siten, että jokainen kutsuja, jonka color-parametrin arvo on sama, kuuluu samaan uuteen viestintäryhmään. Parametri key määrää kutsujien järjestyksen uudessa ryhmässä: mitä pienempi key-parametrin arvo on, sitä pienempi on kutsujan järjestysnumero uudessa viestintäryhmässä. Mikäli useammalla kutsujalla on sama key-parametrin arvo, järjestetään ne uudessa ryhmässä samaan järjestykseen kuin missä ne olivat aiemmin. Parametrien color ja key on oltava positiivisia. Parametrilla color on myös sallittu arvo MPI_UNDEFINED, jolloin kutsuja ei kuulu mihinkään uuteen viestintäryhmään. Viimeinen parametri määrittää nimen uudelle viestintäryhmälle.

6.3 MPI:n valmiit tietotyypit

MPI-kirjasto on suunniteltu toimivaksi heterogeenisessa ympäristössä, eli periaatteessa laskentaan osallistuvissa solmuissa voi olla käytössä eri arkkitehtuureja. Tästä syystä MPI:n viestintäoperaatioissa täytyy ilmaista siirrettävien alkioiden tyyppi MPI-kirjastossa määriteltyjen vakioiden (taulukko 2) avulla.

Taulukko 2: MPI-kirjaston määrittelemät tietotyypit ja niiden vastineet C- ja C++ -kielissä.

MPI:n tietotyyppi	C-kielen tietotyyppi	C++ -kielen tietotyyppi
MPI_CHAR	char	char
MPI_WCHAR	wchar_t	wchar_t
MPI_SHORT	signed short	signed short
MPI_INT	signed int	signed int
MPI_LONG	signed long	signed long
MPI_SIGNED_CHAR	signed char	signed char
MPI_UNSIGNED_CHAR	unsigned char	unsigned char
MPI_UNSIGNED_SHORT	unsigned short	unsigned short
MPI_UNSIGNED	unsigned int	unsigned int
MPI_UNSIGNED_LONG	unsigned long	unsigned long int
MPI_FLOAT	float	float
MPI_DOUBLE	double	double
MPI_LONG_DOUBLE	long double	long double
MPI_BOOL		bool
MPI_COMPLEX		Complex<float>
MPI_DOUBLE_COMPLEX		Complex<double>
MPI_LONG_DOUBLE_COMPLEX		Complex<long double>
MPI_BYTE		
MPI_PACKED		

Kuten taulukosta 2 nähdään, MPI-kirjastossa määritellyt tietotyypit vastaavat vastaavasti nimettyjä C- ja C++ -kielen tietotyyppejä. Taulukossa 2 on myös kaksi vain MPI-kirjastossa määriteltyä tietotyyppiä: MPI_BYTE ja MPI_PACKED. Tietotyyppi MPI_BYTE viittaa tavuun, joka on esitetty kahdeksan bitin avulla. MPI-ympäristössä on mahdollista myös määritellä uusia tietotyyppejä ja siirtää niitä laskentatehtävien välillä käyttäen tietotyyppiä MPI_PACKED. Dataa lähetettäessä tai vastaanotettaessa MPI hoitaa tiedon konvertoinnin (esitystapamuunnokset) automaattisesti.

6.4 Kahden prosessin välinen viestintä

MPI-kirjaston määrittelemät tärkeimmät kahden prosessin väliset viestintärutiinit on listattu taulukossa 3.

Taulukko 3: MPI:n tarjoamia viestintärutiineja kahdenväliseen viestintään.

Viestintärutiinin prototyyppi	**Parametrit**
int **MPI_Send**(void *sendbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm)	sendbuf = osoitin lähetettävään dataan count = lähetettävien alkioiden lkm datatype = alkioiden tyyppi dest = vastaanottajan järjestysnumero tag = viestin tunnistenumero (vapaasti valittavissa) comm = viestintäryhmä, johon viesti lähetetään
int **MPI_Recv**(void *recvbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	recvbuf = osoitin vastaanottopuskuriin count = vastaanotettavien alkioiden lkm datatype = alkioiden tyyppi dest = lähettäjän järjestysnumero tag = viestin tunnistenumero (vapaasti valittavissa) comm = viestintäryhmä, josta vastaanotetaan
int **MPI_Get_count**(MPI_Status *status, MPI_Datatype datatype, int *count)	status = osoitin status-olioon datatype = alkioiden tyyppi count = osoitin muuttujaan, johon lkm tallennetaan
int **MPI_Probe**(int source, int tag, MPI_Comm comm, MPI_Status *status)	source = lähettäjän järjestysnumero tag = viestin tunnistenumero (vapaasti valittavissa) comm = viestintäryhmä, josta vastaanotetaan status = osoitin status-olioon

Dataa voidaan lähettää laskentatehtävältä toiselle operaatiolla MPI_Send(). Jokaista MPI_Send()-operaation kutsua vastaan täytyy jollakin laskentatehtävällä olla vastaava MPI_Recv()-operaation kutsu.

Operaatiolla MPI_Probe() voidaan odottaa tietyn viestin saapumista. Kun viesti on saapunut, voidaan operaatiolla MPI_Get_count() selvittää, kuinka monta alkiota viestin mukana tuli. Näistä operaatioista on hyötyä etenkin vastaanotettaessa vaihtelevan mittaisia taulukoita. Hyvin yksinkertainen esimerkki tästä on esitetty lähdekoodissa 1.

```
1   #include "mpi.h"
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <time.h>
5
6   int main(int argc, char *argv[]){
7     int nTasks, myID, tag=0;
8     int *myData;
9     MPI_Status status;
10    MPI_Init(&argc, &argv);
11    MPI_Comm_size(MPI_COMM_WORLD, &nTasks);
12    MPI_Comm_rank(MPI_COMM_WORLD, &myID);
13    if(myID == 0){
14      int dest_ID, sendCount, i;
15      srand(time(NULL));
16      for(dest_ID = 1; dest_ID < nTasks; dest_ID++){
17        sendCount = (rand() % 10) + 1;
18        myData = (int*)malloc(sendCount*sizeof(int));
19        for(i=0; i<sendCount; i++)
20          myData[i] = rand() % 10;
21        printf("myID: 0, I will send %d numbers to %d\n", sendCount, dest_ID);
22        MPI_Send(myData, sendCount, MPI_INT, dest_ID, tag, MPI_COMM_WORLD);
```

```
23        free(myData);
24       }
25    }
26    else{
27      int source_ID=0, recvCount;
28      MPI_Probe(source_ID, tag, MPI_COMM_WORLD, &status);
29      MPI_Get_count(&status, MPI_INT, &recvCount);
30      myData = (int*)malloc(recvCount*sizeof(int));
31      MPI_Recv(myData, recvCount, MPI_INT, source_ID, tag, MPI_COMM_WORLD, &status);
32      printf("myID: %d, I got %d numbers from 0\n", myID, recvCount);
33      free(myData);
34    }
35  }
```

Lähdekoodi 1: MPI_Probe() ja MPI_Get_count() -operaatioiden käyttö.

Lähdekoodissa 1 esitetyssä ohjelmassa laskentatehtävä, jonka järjestysnumero on nolla, lähettää arpomansa määrän kokonaislukuja kaikille muille laskentatehtäville (rivit 14 - 24). Muut laskentatehtävät odottavat ensin viestin saapumista (rivi 28), selvittävät sitten, montako alkiota niille lähetettiin (rivi 29), ja ottavat sitten alkiot vastaan (rivi 31).

6.5 Kollektiivinen viestintä

Kollektiivisella viestinnällä tarkoitetaan viestintää, johon osallistuu tietty laskentatehtävien joukko. Tällaisissa viestintäfunktioissa yleensä yksi lähettää kaikille tai kaikki lähettävät yhdelle. Osassa MPI-toteutuksista kollektiiviset viestintärutiinit on toteutettu käyttäen alemman kerroksen protokollien broadcast- ja multicast-toiminnallisuutta. Tällöin monen laskentatehtävän välinen viestintä on tehokkainta ja nopeinta toteuttaa käyttäen kollektiivisia viestintäoperaatioita. MPI-kirjaston määrittelemät tärkeimmät kollektiiviset viestintärutiinit on listattu taulukossa 4.

Taulukko 4: MPI:n tarjoamia kollektiivisia viestintärutiineja.

Viestintärutiinin prototyyppi	Parametrit
int **MPI_Barrier**(MPI_Comm comm)	comm = synkronoitava viestintäryhmä
int **MPI_Bcast**(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)	buffer = osoitin datapuskuriin count = lähetettävien alkioiden lkm datatype = lähetettävien alkioiden tyyppi int root = lähettäjän järjestysnumero comm = viestintäryhmä, jolle lähetetään
int **MPI_Reduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) int **MPI_Allreduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)	sendbuf = osoitin lähetettävään dataan recvbuf = osoitin vastaanottopuskuriin count = laskettavien alkioiden lukumäärä datatype = laskettavien alkioiden tyyppi op = alkioille suoritettava laskutoimitus (kts. teksti) root = laskentatehtävä, jolle vastaus annetaan comm = viestintäryhmä, jossa laskenta tehdään
int **MPI_Scatter**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recv-count, MPI_Datatype recvtype, int root, MPI_Comm comm) int **MPI_Gather**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recv-count, MPI_Datatype recvtype, int root, MPI_Comm comm)	sendbuf = osoitin lähetettävään dataan sendcount = lähetettävien alkioiden lukumäärä sendtype = lähetettävien alkioiden tyyppi recvbuf = osoitin vastaanottopuskuriin recvcount = vastaanotettavien alkioiden lukumäärä recvtype = vastaanotettavien alkioiden tyyppi root = laskentatehtävä, jolle vastaus a comm = viestintäryhmä
int **MPI_Scatterv**(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendty-pe, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) int **MPI_Gatherv**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)	sendbuf = osoitin lähetettävään dataan sendcount(s) = lähetettävien alkioiden lukumää-rä(t) displs = lähetettävien alkioiden etäisyydet sendtype = lähetettävien alkioiden tyyppi recvbuf = osoitin vastaanottopuskuriin recvcount(s) = vast.otettavien alkioiden lukumää-rä(t) recvtype = vastaanotettavien alkioiden tyyppi root = lähettävä / vastaanottava laskentatehtävä comm = viestintäryhmä

Kollektiivisia viestintäoperaatioita käytetään niin, että jokainen laskentatehtävä kutsuu samaa operaatiota. Lähes kaikissa taulukossa 4 esitellyistä viestintärutiineista yksi laskentatehtävistä on aina erikoisasemassa.

Laskentatehtävien synkronointi on tehty erittäin helpoksi. Riittää, että jokainen laskentatehtävä kutsuu MPI_Barrier()-operaatiota. Yksikään laskentatehtävä ei jatka suoritustaan ennen kuin kaikki ovat kutsuneet operaatiota.

MPI-Bcast()-operaatiolla voidaan samat data-alkiot lähettää useille laskentatehtäville samanaikaisesti. Parametri buffer on lähettävällä laskentatehtävällä osoitin lähetettävään dataan ja muilla vastaanottopuskuriin.

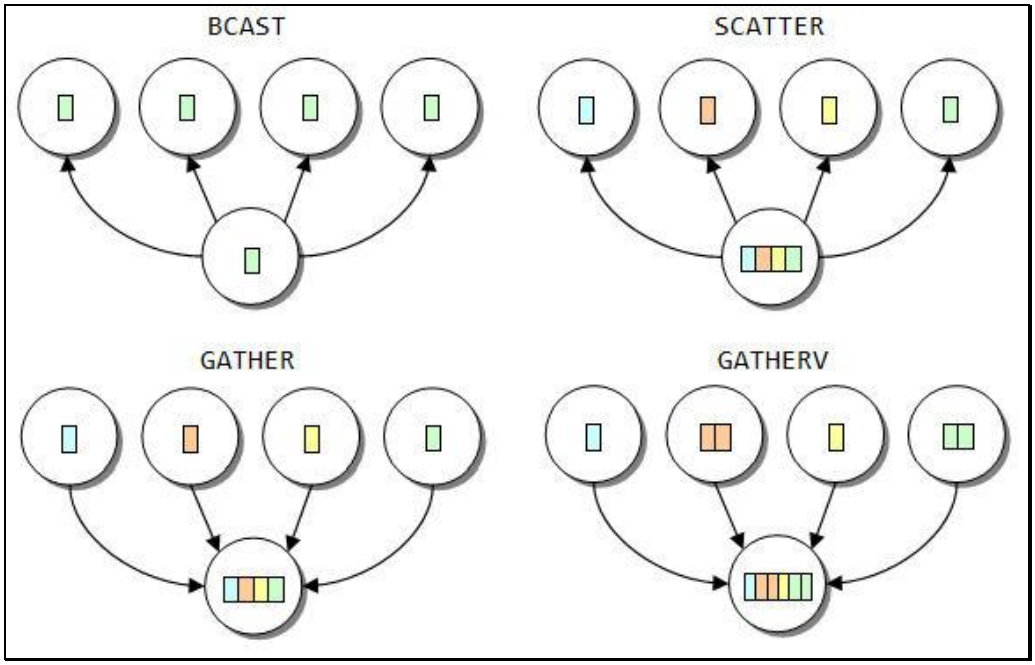
MPI-kirjasto tarjoaa myös mahdollisuuden lähettää samanaikaisesti eri tietoa eri laskentatehtäville. Tähän tarkoitukseen on kaksi operaatiota:

MPI_Scatter() ja MPI_Scatterv(). Erona näillä kahdella on, että ensimmäisellä lähetetään sama määrä alkioita jokaiselle laskentatehtävälle. Jälkimmäinen taas antaa mahdollisuuden lähettää eri määriä dataa eri laskentatehtäville. Vain lähettäjän tarvitsee varata tilaa sendbuf-taulukolle (tästä huolimatta se on jokaisen kutsun parametrina). On tärkeää huomata, että nämä operaatiot lähettävät alkioita myös itselle (lähettäjälle). Näin ollen sendbuf-muistialueen tuleekin olla pituudeltaan vähintään n * p, jossa n on lähetettävien alkioiden lukumäärä ja p laskentatehtävien lukumäärä.

Operaatiota MPI_Scatterv() käytettäessä tulee taulukossa sendcounts olla omissa sarakkeissaan lähetettävien alkioiden määrä kullekin laskentatehtävälle. Taulukossa displs kerrotaan vastaavasti lähetettävän datan etäisyys muistialueen sendbuf alusta.

Tietoa voidaan myös vastaanottaa samanaikaisesti useammalta laskentatehtävältä. Tähän tarkoitukseen on kaksi operaatiota: MPI_Gather() ja MPI_Gatherv(). Ne toimivat aivan kuten edellä mainitut MPI_Scatter() ja MPI_Scatterv(), mutta tiedon kulkusuunta on käänteinen. Esimerkiksi taulukolle recvbuf tarvitsee varata tilaa vain vastaanottavassa laskentatehtävässä.

Kollektiivisten viestintäoperaatioiden toimintaa on havainnollistettu kuvassa 5. Siinä nähdään, kuinka eri kollektiiviset viestintäoperaatiot siirtävät data-alkioita laskentatehtävien välillä (ympyrät kuvaavat laskentatehtäviä ja suorakulmiot data-alkioita).



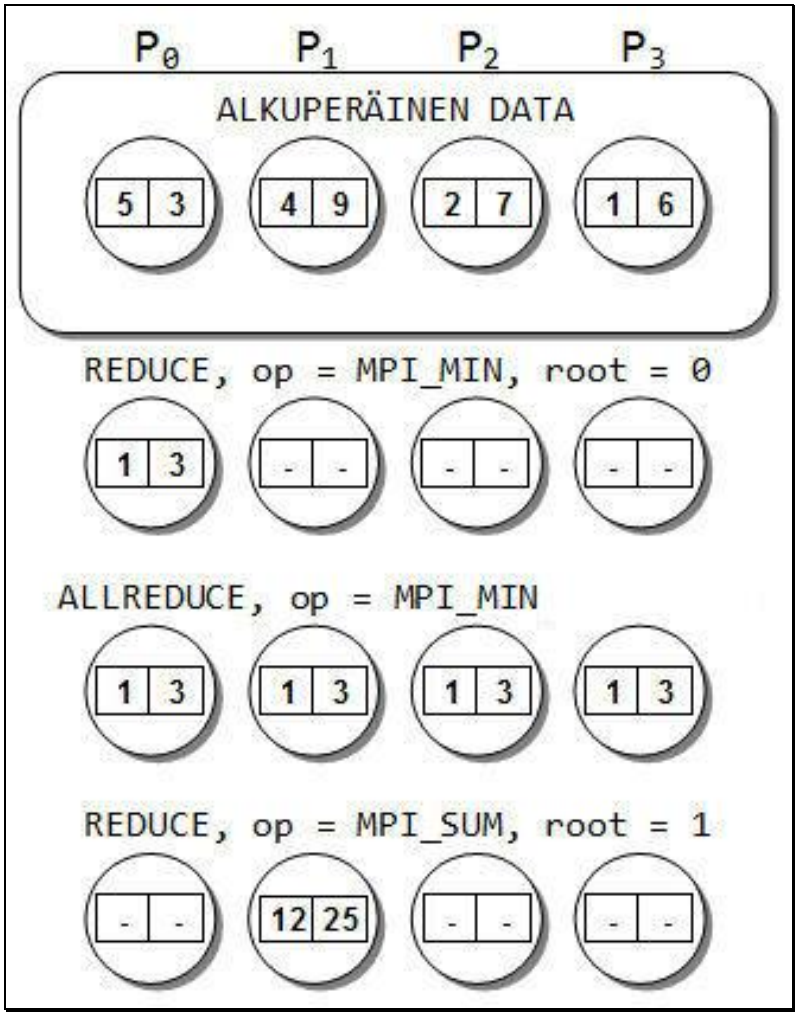
Kuva 5: Kollektiivisten viestintäoperaatioiden toiminta.

MPI-kirjasto määrittelee myös useita ns. reduktio-operaatioita, joilla voidaan suorittaa erilaisia matemaattisia operaatioita kaikkien laskentatehtävien alkioille. Reduktio-operaatioita on kaksi: MPI_Reduce(), jossa vain yksi laskentatehtävistä saa tietää lopputuloksen ja MPI_Allreduce(), jossa lopputulos kerrotaan jokaiselle laskentatehtävälle. Kutsuissa parametrin op arvoksi annetaan jokin MPI-kirjaston määrittelemistä operaatiovakioista. MPI:n tarjoamia reduktio-operaatioita on listattu taulukossa 5.

Taulukko 5: MPI:n määrittelemiä reduktio-operaatioita.

Reduktio-operaatiot	**Kuvaus**
MPI_SUM	Summa
MPI_PROD	Tulo
MPI_MAX	Maksimiarvo
MPI_MIN	Minimiarvo
MPI_MAXLOC	Maksimiarvo ja sijainti
MPI_MINLOC	Minimiarvo ja sijainti
MPI_BAND	Biteittäinen AND-operaatio
MPI_BOR	Biteittäinen OR-operaatio
MPI_BXOR	Biteittäinen XOR-operaatio
MPI_LAND	Looginen AND-operaatio
MPI_LOR	Looginen OR-operaatio
MPI_LXOR	Looginen XOR-operaatio

Osaa MPI:n reduktio-operaatioista on havainnollistettu kuvassa 6.



Kuva 6: MPI:n reduktio-operaatioiden toiminta.

Kuten kuvasta 6 havaitaan, MPI:n reduktio-operaatiot tehdään laskentatehtävien välillä sarakkeittain. Esimerkiksi reduktio-operaatio MPI_MIN etsii parametrina määritellystä muistialueesta sarakkeittain pienimmän laskentatehtävillä esiintyvän luvun. Ensimmäisen sarakkeen pienin arvo on yksi, koska laskentatehtävillä on ensimmäisessä sarakkeessa luvut 5, 4, 2 ja 1. Yhteenvetona MPI-kirjastoa käyttävistä ohjelmista voidaan sanoa, että ne suoritetaan laskentatehtävissä, jotka kommunikoivat keskenään send-receive-pareilla tai niiden varianteilla.

6.6 Uudistuksia MPI-2-standardissa

Tässä luvussa esitettävät asiat perustuvat MPI-2.0-standardiin [MPI-2.0].

MPI-2-standardi toi mukanaan joitakin merkittäviä uudistuksia edellisiin versioihin verrattuna. Tässä luvussa on käyty lyhyesti läpi joitakin näistä uudistuksista.

Dynaaminen prosessien hallinta

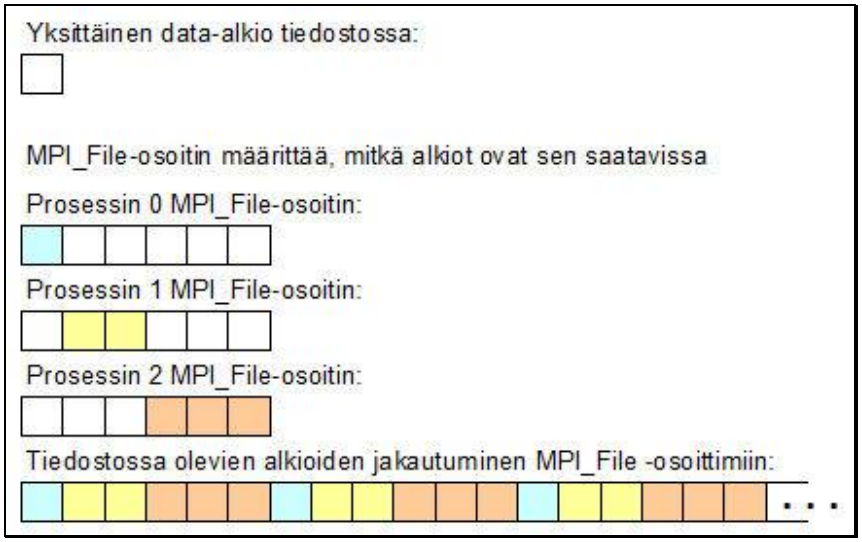
Aiemmissa MPI-standardeissa laskentatehtäviä on ollut sama, ohjelman käynnistämisen yhteydessä määritelty, määrä koko laskennan ajan. MPI-2-standardissa on määritelty myös dynaaminen prosessienhallinta, jolla laskentatehtäviä voidaan lisätä ja poistaa suorituksen aikana. Uusi standardi määrittelee jopa mekanismeja, joilla kaksi erillistä MPI-ohjelmaa voivat viestiä keskenään, vaikka kumpikaan ohjelmista ei olisi käynnistänyt toista ohjelmaa. Dynaamisesti luodut laskentatehtävät lisätään omaan viestintäryhmäänsä, jolloin ne on mahdollista liittää myös osaksi jotain toista viestintäryhmää.

Yksipuolinen viestintä

Eräs MPI-2-standardin uusista ominaisuuksista on mahdollisuus yksipuoliseen viestintään. Aiemmin jokaista MPI_Send()-operaation kutsua kohti on jollain laskentatehtävällä täytynyt olla vastaava MPI_Recv()-operaation kutsu. Yksipuolisessa viestinnässä viestinnän aloittaja määrittää kaikki lähettämiseen ja vastaanottoon liittyvät parametrit. Tästä on hyötyä erityisesti tilanteissa, joissa esimerkiksi aloittava laskentatehtävä A tarvitsee tiettyjä dataalkioita laskentatehtävältä B. Ilman vastaanottamiaan parametreja lähettävä laskentatehtävä B ei kuitenkaan tietäisi mitä sen pitäisi lähettää ja kenelle. Aiemmin tällaisiin tilanteisiin on tarvittu lisää viestejä, joissa osapuolet ovat voineet sopia, mitä lähetetään ja kenelle. MPI-2-standardin myötä vastaanottavan laskentatehtävän A on mahdollista määrittää kaikki viestintään tarvittavat parametrit molempien laskentatehtävien osalta.

Rinnakkainen I/O

Rinnakkainen I/O tuli mukaan MPI-standardiin versiossa 2.0. MPI-standardissa tämä rinnakkainen tiedostojenkäsittely on toteutettu ns. johdettujen tietotyyppien (derived datatypes) avulla. Rinnakkaisen tiedostonkäsittelyn aloittamiseksi jokaisen käsittelyyn osallistuvan laskentatehtävän on kutsuttava samaa kollektiivista operaatiota, jossa määritetään muun muassa avattava tiedosto, jne. Paluuarvona tämä operaatio palauttaa MPI_File-tyyppisen osoittimen tiedostoon. Rinnakkaista tiedostonkäsittelyä on selvitetty kuvassa 7.



Kuva 7: Tiedoston rinnakkainen käsittely MPI_File-osoittimien avulla.

MPI osaa käsitellä tiedostoja, joissa on mielivaltainen määrä samaa tietotyyppiä edustavia alkioita. MPI_File-tyyppinen osoitin sisältää tietyn määrän alkuperäisiä alkioita, mutta osa niistä näkyy ”aukkoina”.

7 RINNAKKAISTAMISESIMERKKI: QUICKSORT-ALGORITMI

Tässä luvussa esitettävät teoria-asiat perustuvat seuraaviin lähteisiin: [Wilkinson, s. 111 - 113, 313 - 314 ja 323 - 326; Cormen, s. 145 - 154; Quinn, s. 33 - 35 ja 338 - 349].

Tässä luvussa havainnollistetaan MPI-viestinvälityskirjaston käyttöä sarjamuotoisen lajittelualgoritmin rinnakkaistamiseen liittyvien ongelmien ratkaisussa. Esitetyissä algoritmeissa esiintyvät MPI-kirjaston operaatiot kuuluvat MPI-1.3-standardiin.

Koejärjestely

Testiympäristönä työssä toteutettaville rinnakkaisille algoritmeille käytetään luvussa 3.2 esitettyä korkean suorituskyvyn klusteria. Klusterin rakentaminen ja asentaminen oli myös osa tätä työtä.

Työssä toteutettavia rinnakkaisia algoritmeja testataan mittaamalla niiden suoritusaikoja sekä työmäärien jakautumista laskentatehtävien välillä. Mittauksia tehdään erilaisilla prosessorien ja lajiteltavien alkioiden määrillä. Suoritusaika mitataan algoritmeihin sisällytettävällä ajanmittauksella. Työmäärien jakautumista mitataan laskentatehtävillä olevien alkioiden määrien suhteelli-

sena erona ideaalitilanteeseen (alkiot jakautuneet tasan kaikille laskentatehtäville). Näin ollen ihannetulos jälkimmäisessä mittauksessa on 0 %.

Quicksort

Quicksort (Hoare, 1962) on yksi suosituimmista ja tehokkaimmista sarjamuotoisista lajittelualgoritmeista. Quicksort-algoritmin toiminta (kuva 8) perustuu niin sanottuihin sarana-alkioihin. Sarana-alkio on alkio, jonka mukaan lista voidaan jakaa kahteen osaan: sarana-alkiota pienempiin tai yhtäsuuriin ja sarana-alkiota suurempiin. Syntyneille osalistoille voidaan taas suorittaa sama operaatio uudelleen niin kauan, että lajiteltavana on ainoastaan yhden alkion mittaisia osalistoja. Quicksort-algoritmin aikavaatimus on yleensä $O(n*\log(n))$, mutta pahimmassa tapauksessa $O(n^2)$. Pahin mahdollinen aineisto Quicksort-algoritmille on sellainen, joka on jo valmiiksi järjestyksessä. Pahimman tilanteen välttämiseksi voidaan algoritmin alkuun kuitenkin lisätä jokin toinen algoritmi, joka sekoittaa lajiteltavat alkiot.

Kuva 8: Quicksort-algoritmin eteneminen.

Rinnakkaisia lajittelutilanteita on pääasiassa kahdenlaisia: lajiteltavat alkiot voivat olla aluksi yhden prosessorin hallussa tai ne voivat olla hajallaan useammilla prosessoreilla. Koska hajautetun muistin järjestelmissä ei voida osoittaa muiden prosessorien muistia, tarvitaan näihin tilanteisiin sopiva algoritmi. Tässä luvussa on käyty läpi kumpaankin tilanteeseen sopivat versiot

rinnakkaisesta Quicksort-algoritmista. Esiteltävissä esimerkeissä lajiteltavina alkioina on käytetty lukuja.

Lajiteltu rinnakkainen aineisto

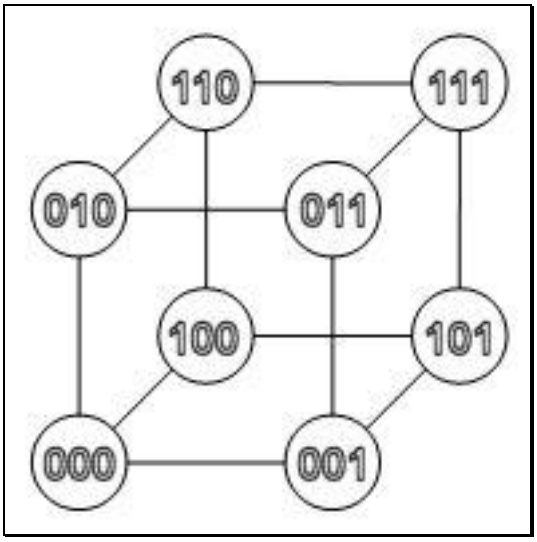
Rinnakkaisessa järjestelmässä aineiston voidaan todeta olevan lajiteltu, kun seuraavat kaksi ehtoa ovat voimassa:

1. Jokaisen prosessorin hallussa oleva osa aineistosta on lajiteltu (jokaiselle alkiolle x pätee: $x_i > x_{(i-1)}$, kun $i > 0$).

2. Jokaisen prosessorin p hallussa olevan aineiston ensimmäinen alkio x_0 on suurempi kuin prosessorin p - 1 hallussa olevan aineiston viimeinen alkio $x_{(n-1)}$, kun $p > 0$.

7.1 Hyperkuutio

Geometriassa hyperkuutio on n-ulotteinen rakenne, jossa on 2^n kulmaa ja jokainen kulma on yhteydessä n kappaleeseen muita kulmia. Näin ollen voidaan sanoa pisteen olevan 0-ulotteinen, viivan 1-ulotteinen, neliön 2-ulotteinen, kuution 3-ulotteinen jne. Tästä johtuen voidaan todeta myös esimerkiksi 3-ulotteisen hyperkuution koostuvan kahdesta 2-ulotteisesta alikuutiosta. Rinnakkaislaskennassa käytettynä jokaista hyperkuution kulmaa edustaa laskentatehtävä. Kuvassa 9 on esitetty kolmiulotteinen hyperkuutio, jossa kulmia edustavat laskentatehtävät (laskentatehtävien numerot on esitetty binäärimuodossa).



Kuva 9: 3-ulotteisen hyperkuution rakenne.

Hyperkuutiossa jokainen laskentatehtävä saa keskustelukumppaninsa järjestysnumeron vaihtamalla yhden bitin arvoa omassa järjestysnumerossaan. Vaihdettava bitti riippuu kyseisellä hetkellä käsiteltävästä ulottuvuudesta. Usein onkin niin, että hyperkuutio käydään läpi suurimmasta ulottuvuudesta pienimpiin. Tällöin bitti, jota vaihdetaan, vaihtuu joka kierroksella.

Hyperkuutio-topologiassa on hyviä rakenteellisia ominaisuuksia, joita voidaan hyödyntää esimerkiksi hajota ja hallitse -tyyppisissä algoritmeissa. Osa seuraavaksi esitettävistä rinnakkaisista algoritmeista muodostaa laskentatehtävistä hyperkuutioita, joissa varsinainen algoritmi suoritetaan.

7.2 Sarjamuotoinen Quicksort

Sarjamuotoinen Quicksort-algoritmi on melko yksinkertainen. Lähdekoodissa 2 on kuvattu algoritmin toiminta pseudokoodina.

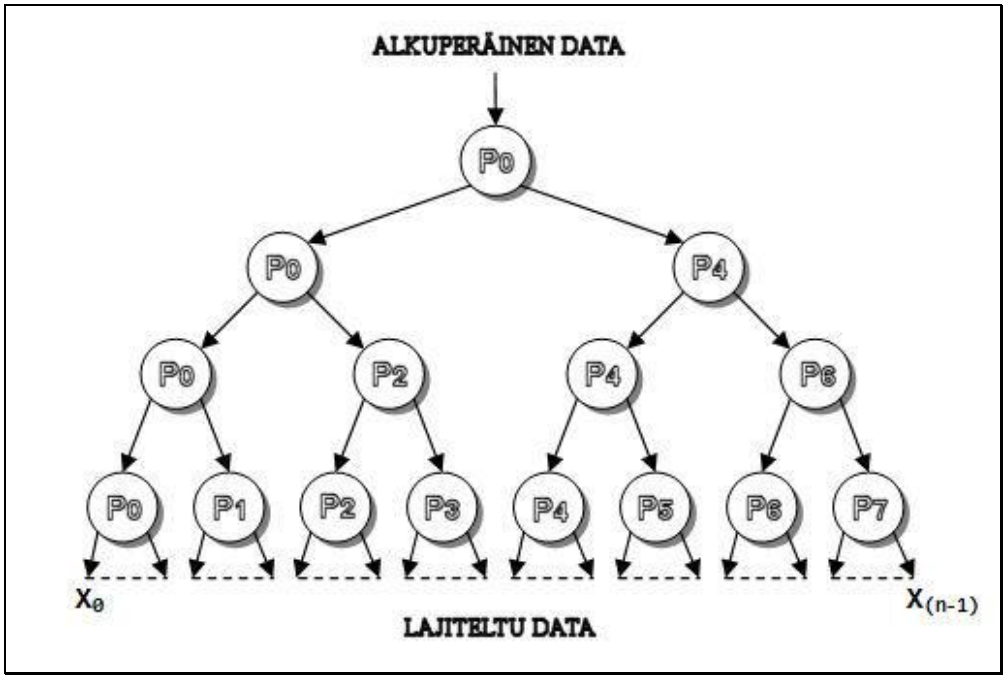
```
Funktio quicksort(lista, alku, loppu){
    Jos alku < loppu{
        osita(lista, alku, loppu, sarana)
        quicksort(lista, alku, sarana-1)
        quicksort(lista, sarana+1, loppu)
    }
}
```

Lähdekoodi 2: Quicksort-funktio pseudokoodina.

Algoritmi saa siis parametrina listan, jossa lajiteltavat alkiot ovat, sekä lajiteltavan alueen alku- ja loppukohdan. Osita-funktio siirtää kaikki alkiot, jotka ovat pienempiä tai yhtäsuuria kuin sarana-alkio, listan alkuun. Vastaavasti sarana-alkiota suuremmat alkiot siirretään listan loppuun. Näin saadut osalistat taas lajitellaan kutsumalla rekursiivisesti quicksort-funktiota. Algoritmin suoritus loppuu, kun rekursiivisten kutsujen alku-parametri ei ole enää pienempi kuin loppu-parametri, eli on päästy yhden alkion mittaisiin osalistoihin. Tässä vaiheessa alkuperäinen lista on järjestyksessä.

7.3 Rinnakkainen Quicksort

Eräs helposti mieleen tuleva tapa rinnakkaistaa Quicksort-algoritmi on aloittaa yhdellä prosessorilla ja välittää aina toinen osalista jollekin toiselle prosessorille. Näin tehtäessä lajitteluprosessista syntyy puurakenne. Kuvassa 10 on esitetty algoritmin eteneminen, kun käytössä on 8 laskentatehtävää (P_0 - P_7). Oletuksena kuvassa on sarana-alkion valinta niin, että se jakaa listan aina kahteen yhtäsuureen osaan (listan mediaani on tällainen luku) ja että kaikki alkiot ovat erisuuruisia.



Kuva 10: Algoritmin suorituksesta syntyvä puurakenne.

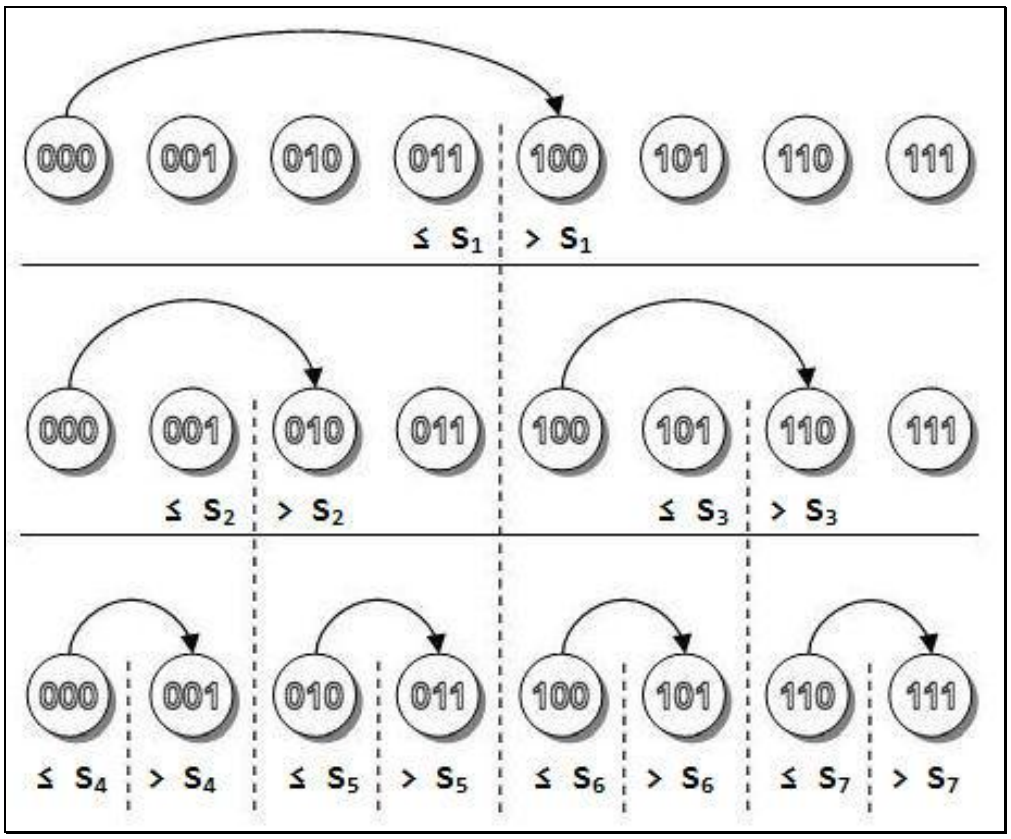
Kuten kuvasta 10 huomataan, laskentatehtävä P_0 esiintyy muista poiketen jokaisella puun tasolla ja tekee näin ollen myös suurimman määrän työtä. Puurakenteen huonona puolena on, että ensimmäisen kahtiajaon suorittaa yksi prosessori, joka rajoittaa tällaisten algoritmien suorituskykyä.

Rinnakkaisohjelmissa suorituskyvyn kannalta parhaassa tilanteessa kaikilla laskentaan osallistuvilla prosessoreilla on yhtä paljon laskettavaa. Quicksort-algoritmin tapauksessa tämä tarkoittaa sitä, että sarana-alkioksi saataisiin aina listan (kaikkien alkioiden) mediaani. Mediaanin valitsemista varten listan pitäisi olla kuitenkin jo valmiiksi järjestyksessä, joten käytännössä tämä on mahdotonta. Monissa Quicksort-algoritmin toteutuksissa sarana-alkiona käytetään listan ensimmäistä tai viimeistä alkiota [Quinn, s. 339; Cormen, s. 146]. Tämä ei kuitenkaan ole välttämättä järkevää, sillä jos näin saatu sarana-alkio on aina osalistan suurin tai pienin, on päädytty suorituskyvyn kannalta pahimpaan mahdolliseen tilanteeseen. Tämän vuoksi onkin järkevämpää valita sarana-alkioksi esimerkiksi listan ensimmäisen ja viimeisen alkion keskiarvo. Näin ei ainakaan yhtä todennäköisesti päädytä tilanteeseen, jossa sarana-alkio olisi listan suurin tai pienin.

7.3.1 Lajiteltava data yhdessä solmussa

Ensimmäisenä rinnakkaisena versiona Quicksort-algoritmista toteutetaan tässä työssä ohjelma, jossa kaikki lajiteltava data on aluksi yhden laskenta-

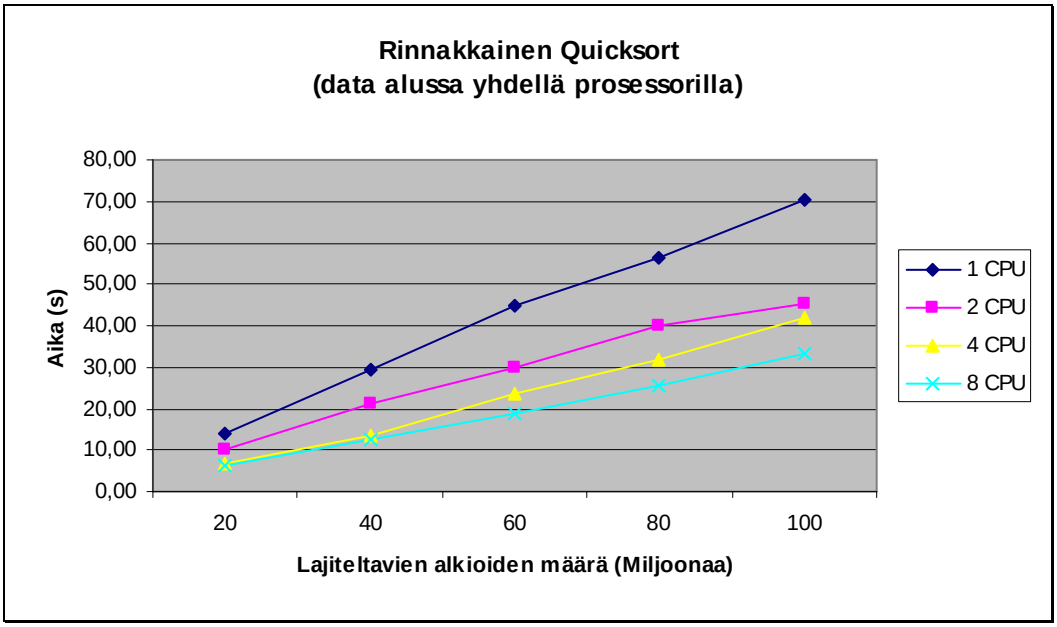
tehtävän hallussa. Lajittelun vaiheet 3-ulotteisessa hyperkuutiossa on esitetty kuvassa 11.



Kuva 11: Rinnakkaisen Quicksort-algoritmin toiminta.

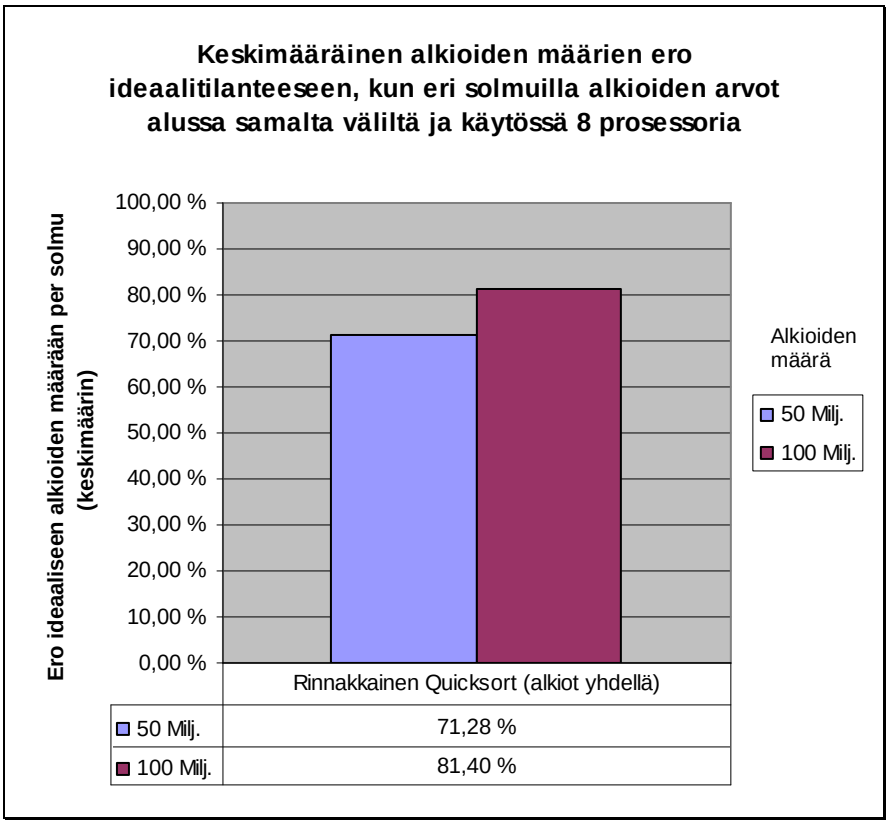
Kuvassa 11 kaikki lajiteltava data on aluksi laskentatehtävässä 0 (000). Laskentatehtävien numerot on kuvassa esitetty selkeyden vuoksi binäärimuodossa. Laskentatehtävä 0 valitsee ensimmäisen sarana-alkion (S_1) ja jakaa datan sen mukaan kahteen osaan. Koska tässä vaiheessa ollaan hyperkuution suurimmassa ulottuvuudessa, sarana-alkiota suuremmat alkiot lähetetään laskentatehtävälle, jonka järjestysnumeron vasemmanpuoleisin bitti on eri kuin lähettäjällä.

Sama operaatio toistuu ulottuvuudesta toiseen sillä erolla, että lähettäjiä ja vastaanottajia on jokaisessa alikuutiossa kaksinkertainen määrä. Kun viimein tullaan 0-ulotteisiin hyperkuutioihin, data on valmiina lajiteltavaksi rinnakkain kaikissa laskentatehtävissä. Tämän jälkeen molemmat järjestetyn aineiston ehdot ovat voimassa. Jos sarana-alkiot valittiin hyvin, on jokaisella laskentatehtävällä suunnilleen saman verran laskettavaa.



Kuva 12: Algoritmin suoritusajat eri alkioiden ja prosessorien määrillä.

Kuvassa 12 nähdään algoritmin suoritusajat eri alkioiden ja prosessorien määrillä. Suoritusajat pienenevät useampaa prosessoria käytettäessä merkittävästi, etenkin suurilla alkioiden määrillä.



Kuva 13: Solmujen hallussa olevien alkioiden määrän ero ideaalitilanteeseen keskimäärin.

Kuvasta 13 huomataan, että alkiot jakautuvat solmuihin erittäin epätasaisesti. Tämä johtuu erityisesti siitä, että sarana-alkio muodostetaan yhden laskentatehtävän epäjärjestyksessä olevista alkioista.

Lähdekoodin läpikäynti

Tässä osiossa käydään läpi algoritmin lähdekoodia. Lähdekoodi löytyy kokonaisuudessaan liitteestä 1. Lähdekoodin alussa esitellään tarvittavat muuttujat (lähdekoodi 3). Muuttujat on tässä yhteydessä esitelty seuraavaksi esitettävien lähdekoodien tulkitsemisen helpottamiseksi.

```
32 MPI_Status status;     // MPI-funktioiden tarvitsema status-muuttuja
33    double elapsedTime; // Ajanmittaukseen käytettävä muuttuja
34       int i,           // Apumuuttuja silmukoihin
35           j,           // Apumuuttuja silmukoihin
36           tag=0,       // MPI-viestintäfunktioissa käytettävä tunniste
37           p,           // Laskentatehtävien lukumäärä
38           myID,        // Oma järjestysnumero
39           partnerID,   // Keskustelukumppanin järjestysnumero
40           cubeSize,    // Hyperkuution koko
41           dim,         // Suurimman hyperkuution ulottuvuuksien lkm
42           curDim,      // Nykyisen hyperkuution ulottuvuuksien lkm
43           n,           // Data-alkioiden kokonaismäärä
44           nPerProc=0,  // Yhden laskentatehtävän hallussa olevien
                             alkioiden lkm
45           nSmaller,    // Sarana-alkiota pienempien alkioiden lkm
46           nLarger;     // Sarana-alkiota suurempien alkioiden lkm
47     float pivot,       // Sarana-alkio
48           *myData,     // Laskentatehtävän hallussa oleva data
49           *smallerThanPivot, // Sarana-alkiota pienemmät alkiot
50           *largerThanPivot;  // Sarana-alkiota suuremmat alkiot
```

Lähdekoodi 3: Muuttujien esittelyt.

Muuttujien esittelyn yhteydessä on selvitetty myös hieman jokaisen muuttujan käyttötarkoitusta.

```
69  // --- LUKUJEN LATAUS ---
70  if(myID == 0){
71    nPerProc = n;
72    if( (myData = (float*)malloc(n*sizeof(float))) == 0)
73      printf("Not enough memory for myData[%d]\n",n);
74    srand48(time(NULL));
75    for(i=0; i<n; i++){
76      myData[i] = drand48();
77    }
78  }
79  // ----------------------
80
81  // Odotetaan että kaikki laskentatehtävät pääsevät tähän asti ja
82  // aloitetaan ajan mittaaminen
83  MPI_Barrier(MPI_COMM_WORLD);
84  elapsedTime = MPI_Wtime();
```

Lähdekoodi 4: Alkioiden lataus ja laskentatehtävien synkronointi.

Ohjelman alussa on silmukka (rivit 75 - 77), jossa laskentatehtävä 0 arpoo lajiteltavat alkiot. Jotta ajanmittauksesta saadaan mahdollisimman luotettava, on se aloitettu vasta laskentatehtävien synkronoinnin jälkeen (rivit 83 ja 84).

```
86  // Jokainen laskentatehtävä kuuluu ensimmäiseen hyperkuutioon
87  cubeSize = p;
88
89  // Lasketaan käytettävissä olevien ulottuvuuksien lukumäärä
90  dim = ceil(log((double)p) / (log(2.0)));
91
92  // Käydään hyperkuution ulottuvuudet läpi suurimmasta pienimpiin
93  for(curDim=dim; curDim>0; curDim--){
94    if(myID % (cubeSize/2) == 0){ // Olen lähettäjä tai vastaanottaja
95      // Lasketaan keskustelukumppanin järjestysnumero
96      partnerID = myID ^ (1 << (curDim-1));
97      if(myID < partnerID){ // Olen lähettäjä
              .
              .
              .
122     }
123     else{ // Olen vastaanottaja
              .
              .
              .
129     }
130   }
131   // Puolitetaan kuutiot
132   cubeSize /= 2;
133 }
```

Lähdekoodi 5: Hyperkuution läpikäyntisilmukka.

Lähdekoodissa 5 on pelkistetty hyperkuutioiden läpikäyntisilmukka. Ensimmäisellä kierroksella kaikki laskentatehtävät kuuluvat samaan hyperkuutioon, joten sen koko on laskentatehtävien lukumäärä (rivi 87). Laskentatehtävien lukumäärä on aina kahden potenssi, joten hyperkuution suurin ulottuvuus saadaan selville ratkaisemalla eksponenttiyhtälö $2^x = p$, jossa p on laskentatehtävien lukumäärä (rivi 90).

Tässä algoritmissa lähettäjiä ovat hyperkuution alempien puolien ensimmäiset laskentatehtävät ja vastaanottajia ylempien puolien ensimmäiset laskentatehtävät. Kommunikoivat laskentatehtävät tunnistavat itsensä ehtolauseessa rivillä 94. Hyperkuutioissa keskustelukumppanien järjestysnumerot eroavat toisistaan vain yhdellä bitillä. Näin ollen voidaan kumppanin järjestysnumero selvittää tekemällä biteittäinen XOR-operaatio: a XOR $2^{(d-1)}$, jossa a on oma järjestysnumero ja d on nykyinen ulottuvuus (rivi 96).

```
97  if(myID < partnerID){ // Olen lähettäjä
98    pivot = (myData[0]+myData[nPerProc-1]) / 2;
99    // Muodostetaan osalistat sarana-alkiota pienemmille ja yhtäsuurille
100   // sekä suuremmille
101   if( (smallerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0)
102     printf("Not enough memory for smallerThanPivot[%d]\n",nPerProc);
103   if( (largerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0)
104     printf("Not enough memory for largerThanPivot[%d]\n",nPerProc);
105   nSmaller = 0;
106   nLarger = 0;
107   for(j = 0; j < nPerProc; j++){
108     if (myData[j] <= pivot) {
109       smallerThanPivot[nSmaller] = myData[j];
110       nSmaller++;
111     }
112     else {
113       largerThanPivot[nLarger] = myData[j];
114       nLarger++;
115     }
116   }
117   MPI_Send(largerThanPivot, nLarger, MPI_FLOAT, partnerID, tag,
              MPI_COMM_WORLD);
118   free(myData);
119   myData = smallerThanPivot;
120   nPerProc = nSmaller;
121   free(largerThanPivot);
122 }
```

Lähdekoodi 6: Lähettäjän koodi.

Lähdekoodissa 6 on lähettävän laskentatehtävän läpikäymä koodi. Ensin valitaan käytettävä sarana-alkio (rivi 98). Valinnan jälkeen alkioista täytyy erotella omiin osalistoihinsa sarana-alkiota pienemmät ja yhtäsuuret sekä suuremmat (rivit 99 - 116). Koska tässä algoritmissa lähettäjät kuuluvat aina kuution alempaan puoleen, lähetetään keskustelukumppanille sarana-alkiota suurempien osalista (rivi 117). Lopuksi vielä vaihdetaan omiksi alkioiksi pelkkä pienempien alkioiden osalista (rivit 118 - 121).

```
123 else{ // Olen vastaanottaja
124   MPI_Probe(partnerID, tag, MPI_COMM_WORLD, &status);
125   MPI_Get_count(&status, MPI_INT, &nPerProc);
126   if((myData = (float *) malloc(nPerProc*sizeof(float))) == 0)
127     printf("Not Enough Memory for myData[%d]\n",nPerProc);
128   MPI_Recv(myData, nPerProc, MPI_FLOAT, partnerID, tag, MPI_COMM_WORLD,
              &status);
129 }
```

Lähdekoodi 7: Vastaanottajan koodi.

Vastaanottajan koodi (Lähdekoodi 7) on tässä algoritmissa hyvin yksinkertainen, sillä vastaanottavilla laskentatehtävillä ei ennen vastaanottoa ole lainkaan alkioita. Alkioiden vastaanotto tapahtuu niin, että ensin odotetaan viestin saapumista (rivi 124). Viestin saavuttua selvitetään kuinka monta alkiota se sisältää (rivi 125), varataan alkioille muistia (rivi 126) ja vastaanotetaan saapuneet alkiot (rivi 128).

```
135 // Lajitellaan alkiot jos niitä on enemmän kuin yksi
136 if(nPerProc > 1)
137   qsort(myData, nPerProc, sizeof(float), compare);
138
139 // Lopetetaan ajan mittaaminen
140 elapsedTime = MPI_Wtime() - elapsedTime;
```

Lähdekoodi 8: Alkioiden lajittelu ja ajan mittaamisen lopetus.

Lähdekoodissa 8 nähdään lajittelualgoritmin loppu. Algoritmin lopussa jokainen laskentatehtävä lajittelee omat alkionsa, jos niitä on enemän kuin yksi (rivit 136 ja 137). Lajittelun jälkeen alkiot ovat järjestyksessä, joten ajan mittaaminen voidaan lopettaa (rivi 140).

```
142 //-------VIRHEIDEN TARKISTUS------
143 int errors = 0;
144 for(i=0; i<nPerProc-1; i++){
145   if(myData[i] > myData[i+1])
146     errors++;
147 }
148 if(errors > 0) //tieto virheistä vain jos niitä on
149   printf("myID: %d, number of errors: %d\n", myID, errors);
150
151 if(myID==0){
152   errors = 0;
153   float *tests;
154   if( (tests = (float*)malloc((p*2)*sizeof(float))) == 0)
155     printf("Not enough memory for tests[%d]\n",(p*2));
156   tests[0] = myData[0];
157   tests[1] = myData[nPerProc-1];
158   for(i=1; i<p; i++){
159     MPI_Recv(tests+(i*2), 2, MPI_FLOAT, i, tag, MPI_COMM_WORLD,
                &status);
160   }
161   for(i=1; i<(p*2); i++){
162     if(tests[i-1] > tests[i])
163       errors++;
164   }
165   if(errors > 0) //tieto virheistä vain jos niitä on
166     printf("Number of errors between processors: %d\n",errors);
167 }
168 else{
169   float tests[2];
170   tests[0] = myData[0];
171   tests[1] = myData[nPerProc-1];
172   MPI_Send(tests, 2, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
173 }
174 //--------------------------------
```

Lähdekoodi 9: Virheiden tarkistus.

Lähdekoodissa 9 on virheiden tarkistukseen liittyvä osuus koodista. Virheillä tarkoitetaan tässä yhteydessä lajitteluvirheitä. Virheiden määrän ollessa nolla molemmat järjestetyn rinnakkaisen aineiston ehdot ovat voimassa. Samaa virheentarkistusmenetelmää on käytetty jokaisen neljän algoritmin koodissa. Aluksi tarkistetaan, ovatko laskentatehtävän hallussa olevat data-alkiot järjestyksessä. Tämän jälkeen jokainen laskentatehtävä lähettää laskentatehtävälle 0 ensimmäisen ja viimeisen alkionsa (rivit 169 - 172). Näiden alkioi-

den avulla saadaan varmistettua, että myös toinen järjestetyn rinnakkaisen aineiston ehdoista on voimassa.

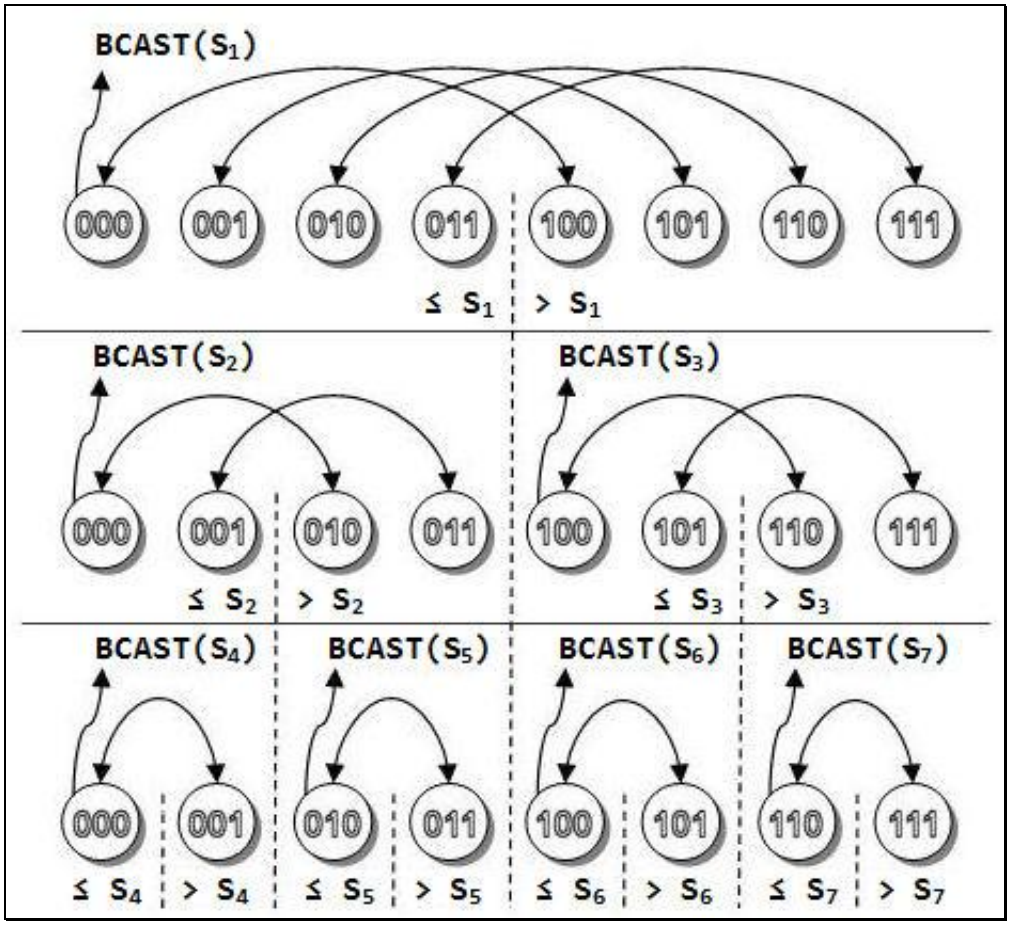
```
176 // Tulostetaan näytölle oma järjestysnumero, käytetty prosessoriaika ja
177 // hallussa olevien alkioiden lukumäärä
178 fprintf(stdout,"Processor Id: %d;  The CPU Time: %10.6f;  SubArray
                    Length: %d.\n", myID, elapsedTime, nPerProc);
179 fflush(stdout);
```

Lähdekoodi 10: Laskentatehtävään liittyvien suoritustietojen tulostus.

Lähdekoodissa 10 on viimeiseksi tehtävä laskentatehtävään liittyvien suoritustietojen tulostus (rivit 178 ja 179). Tulostettaviin tietoihin kuuluvat laskentatehtävän järjestysnumero, käytetty prosessoriaika sekä laskentatehtävän hallussa olevien alkioiden lukumäärä.

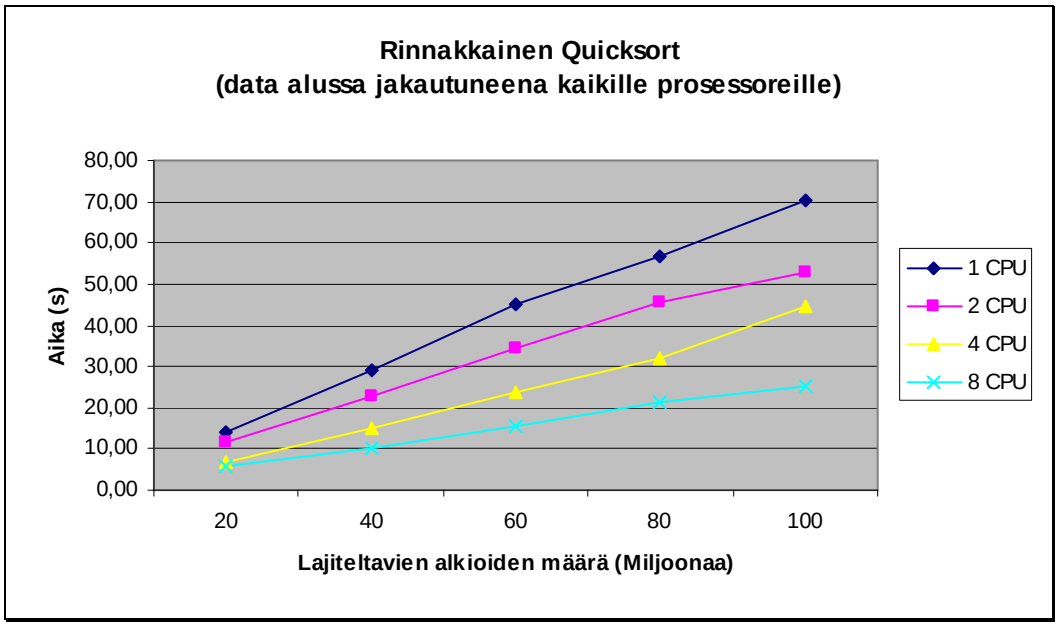
7.3.2 Lajiteltava data jakautuneena useampaan solmuun

Toisena rinnakkaisena versiona Quicksort-algoritmista toteutettiin ohjelma, jossa lajiteltava data on jo valmiiksi jakautunut kaikille laskentatehtäville. Lajittelun vaiheet 3-ulotteisessa hyperkuutiossa on esitetty kuvassa 14.



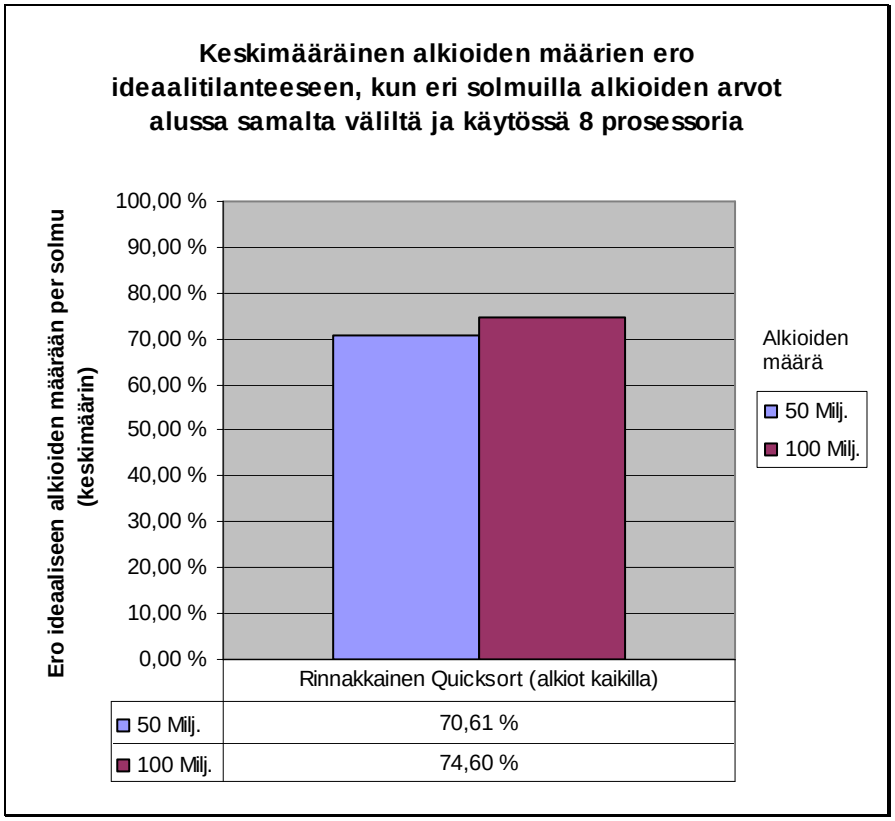
Kuva 14: Rinnakkaisen Quicksort-algoritmin toiminta.

Tässä algoritmissa laskentatehtävä 0 valitsee sarana-alkion (S_n) ja lähettää sen kaikille laskentatehtäville. Tämän jälkeen jokainen laskentatehtävä jakaa hallussaan olevan aineiston kahteen osaan: sarana-alkiota pienempiin ja yhtäsuuriin sekä sarana-alkiota suurempiin. Oltaessa hyperkuution ulottuvuudessa n, laskentatehtävät, joiden järjestysnumeron oikealta vasemmalle laskettuna n:s bitti on yksi, lähettävät sarana-alkiota pienemmät ja yhtäsuuret alkiot laskentatehtävälle, jonka n:s bitti on nolla. Vastaavasti laskentatehtävät, joiden n:s bitti on nolla, lähettävät sarana-alkiota suuremmat alkiot laskentatehtävälle, jonka n:s bitti on yksi. Tämän jälkeen hyperkuutiot pilkotaan kahdeksi osakuutioksi ja samat vaiheet toistetaan niissä. Tätä toistetaan niin pitkään, että tullaan 0-ulotteisiin hyperkuutioihin. Silloin jokainen laskentatehtävä lajittelee oman osansa aineistosta, jonka jälkeen molemmat järjestetyn aineiston ehdot ovat voimassa.



Kuva 15: Algoritmin suoritusajat eri alkioiden ja prosessorien määrillä.

Kuvassa 15 nähdään algoritmin suoritusajat eri alkioiden ja prosessorien määrillä. Suoritusajat pienenevät useampaa prosessoria käytettäessä merkittävästi etenkin suurilla alkioiden määrillä.



Kuva 16: Solmujen hallussa olevien alkioiden määrän ero ideaalitilanteeseen keskimäärin.

Kuvasta 16 huomataan, että myös tässä algoritmissa alkiot jakautuvat hyvin epätasaisesti laskentatehtäville. Suurin syy tähän on sama kuin algoritmin ensimmäisessä versiossa: sarana-alkion muodostamiseen käytetään yhden laskentatehtävän epäjärjestyksessä olevia alkioita.

Lähdekoodin läpikäynti

Tässä osiossa käydään läpi algoritmin lähdekoodia siltä osin, kuin se eroaa edellä esitetyistä. Lähdekoodi löytyy kokonaisuudessaan liitteestä 2. Lähdekoodin alussa esitellään tarvittavat muuttujat (lähdekoodi 11).

```
33  MPI_Status status;    // MPI-funktioiden tarvitsema status-muuttuja
34    MPI_Comm MPI_COMM_CUBE; // Viestintäryhmä alikuutioille
35      double elapsedTime; // Ajanmittaukseen käytettävä muuttuja
36         int i,          // Apumuuttuja silmukoihin
37             j,          // Apumuuttuja silmukoihin
38             tag=0,      // MPI-viestintäfunktioissa käytettävä tunniste
39             p,          // Laskentatehtävien lukumäärä
40             myID,       // Oma järjestysnumero
41             partnerID,  // Keskustelukumppanin järjestysnumero
42             numCubes,   // Hyperkuutioiden lukumäärä
43             sizeCubes,  // Hyperkuutioiden koko
44             myCubeNum,  // Oman hyperkuution numero
45             myCubeID,   // Järjestysnumero nykyisessä hyperkuutiossa
46             dim,        // Suurimman hyperkuution ulottuvuuksien lkm
```

```
47              curDim,     // Nykyisen hyperkuution ulottuvuuksien lkm
48              n,          // Data-alkioiden kokonaismäärä
49              nPerProc=0, // Yhden laskentatehtävän hallussa olevien
                               alkioiden lkm
50              nRecv,      // Vastaanotettavien / -otettujen alkioiden lkm
51              nSmaller,   // Sarana-alkiota pienempien alkioiden lkm
52              nLarger;    // Sarana-alkiota suurempien alkioiden lkm
53        float pivot,      // Sarana-alkio
54              *myData,    // Laskentatehtävän hallussa oleva data
55              *smallerThanPivot, // Sarana-alkiota pienemmät alkiot
56              *largerThanPivot, // Sarana-alkiota suuremmat alkiot
57              *buf;       // Tilapäinen puskuri alkioille
```

Lähdekoodi 11: Muuttujien esittelyt.

Lähdekoodissa 11 on muuttujien esittelyn yhteydessä selvitetty myös niiden käyttötarkoitus.

```
90  // Jaetaan luvut tasan kaikille laskentatehtäville
91  if( (myData = (float*)malloc(nPerProc*sizeof(float))) == 0 )
92      printf("Not enough memory for myData[%d]\n",nPerProc);
93  MPI_Scatter(buf, nPerProc, MPI_FLOAT, myData, nPerProc, MPI_FLOAT, 0,
                MPI_COMM_WORLD);
94
95  // Odotetaan että kaikki laskentatehtävät pääsevät tähän asti ja
96  // aloitetaan ajan mittaaminen
97  MPI_Barrier(MPI_COMM_WORLD);
98  elapsedTime = MPI_Wtime();
```

Lähdekoodi 12: Lajiteltavien alkioiden jako laskentatehtäville ja ajan mittaamisen aloitus.

Lähdekoodissa 12 jaetaan lajiteltavat alkiot kaikille laskentatehtäville. Koska jokainen laskentatehtävä saa aluksi saman määrän alkioita, voidaan alkioiden jakamiseen käyttää operaatiota MPI_Scatter() (rivi 93). Tämän jälkeen synkronoidaan laskentatehtävät (rivi 97) ja aloitetaan ajan mittaaminen (rivi 98).

```
103 j=0;
104 for(curDim=dim; curDim>0; curDim--){
105   // Lasketaan hyperkuutioiden lukumäärä
106   numCubes = 1 << j;
107   // Lasketaan hyperkuutioiden koko
108   sizeCubes = p / numCubes;
109   // Lasketaan oman hyperkuution numero
110   myCubeNum = myID >> curDim;
111   // Jaetaan tarvittaessa nykyinen kuutio alikuutioiksi
112   MPI_Comm_split(MPI_COMM_WORLD, myCubeNum, myID, &MPI_COMM_CUBE);
113   // Selvitetään oma järjestysnumero uudessa hyperkuutiossa
114   MPI_Comm_rank(MPI_COMM_CUBE, &myCubeID);
115
116   if(myCubeID == 0){
117     // Muodostetaan sarana-alkio
118     pivot = (myData[0]+myData[nPerProc-1]) / 2;
119   }
120   // Lähetetään sarana-alkio kaikille
121   MPI_Bcast(&pivot, 1, MPI_INT, 0, MPI_COMM_CUBE);
122   // Lasketaan keskustelukumppanin järjestysnumero
123   partnerID = myCubeID ^ (1 << (curDim-1));
124
125   if(myCubeID < (sizeCubes / 2)){ //1. Lähetä, 2. Vastaanota
```

```
        .
        .
        .
164   }
165   else{// 1. Vastaanota, 2. Lähetä
        .
        .
        .
197   }
198   j++;
199 }
```

Lähdekoodi 13: Hyperkuution läpikäyntisilmukka.

Lähdekoodissa 13 on pelkistetty hyperkuution läpikäyntisilmukka. Jokaisen kierroksen (ulottuvuuden) alussa lasketaan käytettävissä olevien hyperkuutioiden määrä ja koko (rivit 106 ja 108). Tämän jälkeen selvitetään oman hyperkuution numero (rivi 110). Seuraavaksi jaetaan hyperkuutio alikuutioiksi (rivi 112) ja selvitetään oma järjestysnumero uudessa kuutiossa (rivi 114). Alikuutioiksi jakoa ei suoriteta ensimmäisellä kierroksella ollenkaan, vaan silloin jokaisen laskentatehtävän käyttämä MPI_Comm_split()-kutsun color-arvo (myCubeNum) on sama.

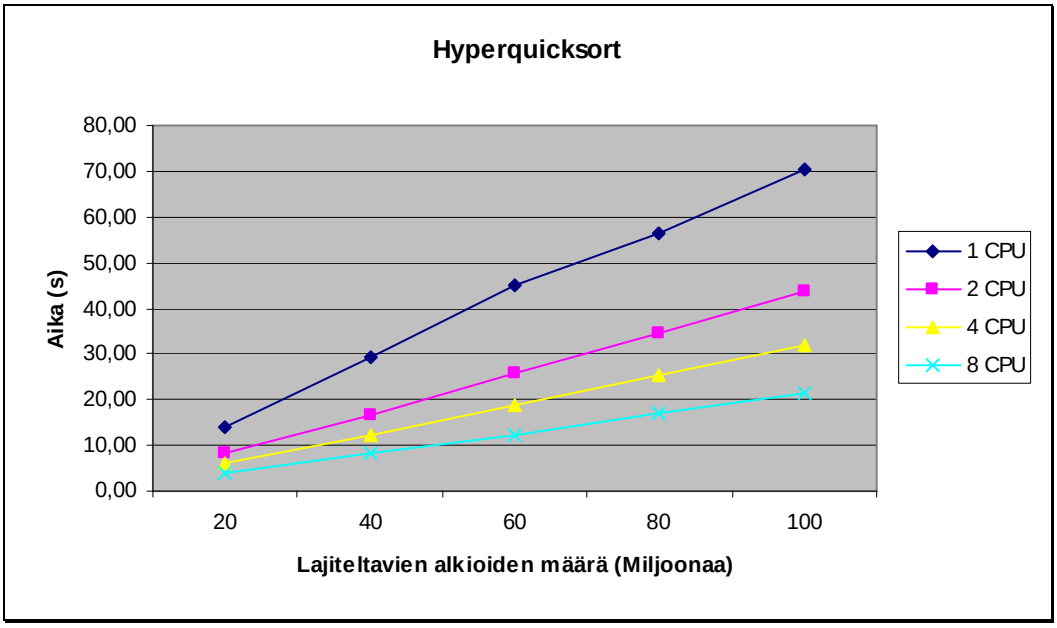
Tässä algoritmissa jokaisessa kuutiossa pienimmän järjestysnumeron omaava laskentatehtävä muodostaa käytettävän sarana-alkion (rivit 116 - 119). Tämän jälkeen käytettävä sarana-alkio lähetetään kaikille samaan kuutioon kuuluville laskentatehtäville (rivi 121). Tässä algoritmissa jokainen laskentatehtävä lähettää ja vastaanottaa, joten laskentatehtävät pitää jakaa kahteen osaan niin, että samaan aikaan toinen puoli lähettää ensin ja vastaanottaa sen jälkeen (rivit 125 - 164) ja toinen puoli vastaanottaa ensin ja lähettää sen jälkeen (rivit 165 - 197).

7.4 Hyperquicksort

Hyperquicksort-algoritmi (Wagar, 1987) alkaa siitä, mihin kaksi edellistä loppuivat. Ensimmäiseksi siis jokainen laskentatehtävä lajittelee oman osansa aineistosta. Tässä vaiheessa on jo ensimmäinen lajitellun aineiston ehdoista voimassa. Koska alkiot ovat nyt järjestyksessä, on mediaanin valitseminen sarana-alkioksi erittäin helppoa (se on listan keskimmäinen alkio tai kahden keskimmäisen keskiarvo). Tämä lajitteluvaihe tehdään hyperkuution kaikissa nollaa suuremmissa ulottuvuuksissa. Edellisissä algoritmeissa ollut viimeinen lajitteluvaihe siis jää pois. Sarana-alkion mukaan muodostetuista osalistoista toinen lähetetään hyperkuution toisessa puoliskossa sijaitsevalle las-

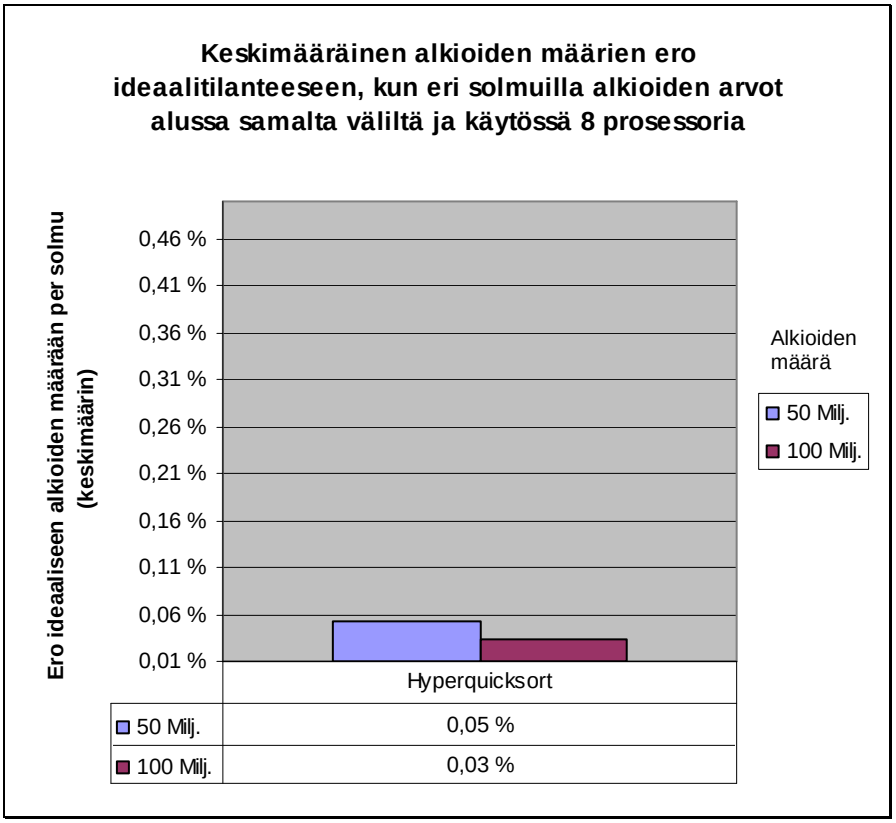
kentatehtävälle, jonka järjestysnumeron n:s bitti (ulottuvuuden ollessa n) on eri kuin lähettäjällä.

Tämän algoritmin etuna on, että mikäli alkiot ovat jakautuneet (arvojen suhteen) suunnilleen tasaisesti eri laskentatehtäville, niin laskentatehtävän 0 valitsema sarana-alkio on hyvin lähellä koko aineiston todellista mediaania.



Kuva 17: Algoritmin suoritusajat eri alkioiden ja prosessorien määrillä.

Kuvassa 17 nähdään algoritmin suoritusajat eri alkioiden ja prosessorien määrillä. Yhden prosessorin suoritusaikoihin verrattuna erot ovat huomattavia.



Kuva 18: Solmujen hallussa olevien alkioiden määrän ero ideaalitilanteeseen keskimäärin.

Kuvasta 18 (huomaa vaihtunut asteikko) huomataan, että nyt kun saranaalkion muodostamiseen käytetään lajiteltua aineistoa, saadaan työmäärät jaettua erittäin tasaisesti eri laskentatehtäville.

Lähdekoodin läpikäynti

Tässä osiossa käydään läpi algoritmin lähdekoodia siltä osin, kuin se eroaa edellä esitetyistä. Lähdekoodi löytyy kokonaisuudessaan liitteestä 3. Lähdekoodin alussa esitellään tarvittavat muuttujat (lähdekoodi 14).

```
33   MPI_Status status;    // MPI-funktioiden tarvitsema status-muuttuja
34     MPI_Comm MPI_COMM_CUBE;// Viestintäryhmä alikuutioille
35       double elapsedTime;// Ajanmittaukseen käytettävä muuttuja
36          int i,         // Apumuuttuja silmukoihin
37              j,         // Apumuuttuja silmukoihin
38              tag=0,     // MPI-viestintäfunktioissa käytettävä tunniste
39              p,         // Laskentatehtävien lukumäärä
40              myID,      // Oma järjestysnumero
41              partnerID, // Keskustelukumppanin järjestysnumero
42              numCubes,  // Hyperkuutioiden lukumäärä
43              sizeCubes, // Hyperkuutioiden koko
44              myCubeNum, // Oman hyperkuution numero
45              myCubeID,  // Järjestysnumero nykyisessä hyperkuutiossa
46              dim,       // Suurimman hyperkuution ulottuvuuksien lkm
47              curDim,    // Nykyisen hyperkuution ulottuvuuksien lkm
48              n,         // Data-alkioiden kokonaismäärä
```

```
49              nPerProc=0,// Yhden laskentatehtävän hallussa olevien
                           alkioiden lkm
50              nRecv,     // Vastaanotettavien / -otettujen alkioiden lkm
51              ownCount,  // Apumuuttuja listojen yhdistämiseen
52              recvCount, // Apumuuttuja listojen yhdistämiseen
53              nSmaller,  // Sarana-alkiota pienempien alkioiden lkm
54              nLarger;   // Sarana-alkiota suurempien alkioiden lkm
55         float pivot,     // Sarana-alkio
56              *myData,   // Laskentatehtävän hallussa oleva data
57      *smallerThanPivot,  // Sarana-alkiota pienemmät alkiot
58        *largerThanPivot, // Sarana-alkiota suuremmat alkiot
59              *recvBuf,  // Vastaanottopuskuri
60              *buf;      // Tilapäinen puskuri alkioille
```

Lähdekoodi 14: Muuttujien esittelyt.

Lähdekoodissa 14 on muuttujien esittelyn yhteydessä myös selvitetty niiden käyttötarkoitus.

```
106 j=0;
107 for(curDim=dim; curDim>0; curDim--){
108   // Lajitellaan omat alkiot
109   qsort(myData, nPerProc, sizeof(float), compare);
110
111   // Lasketaan hyperkuutioiden lukumäärä
112   numCubes = 1 << j;
113   // Lasketaan hyperkuutioiden koko
114   sizeCubes = p / numCubes;
115   // Lasketaan oman hyperkuution numero
116   myCubeNum = myID >> curDim;
117   //Jaetaan tarvittaessa nykyinen kuutio alikuutioiksi
118   MPI_Comm_split(MPI_COMM_WORLD, myCubeNum, myID, &MPI_COMM_CUBE);
119   // Selvitetään oma järjestysnumero uudessa hyperkuutiossa
120   MPI_Comm_rank(MPI_COMM_CUBE, &myCubeID);
121
122   if(myCubeID == 0){
123     // Muodostetaan sarana-alkio
124     if(nPerProc % 2 == 0)
125       pivot = (myData[nPerProc / 2 - 1]+myData[nPerProc / 2]) / 2;
126     else
127       pivot = myData[nPerProc / 2];
128   }
129   // Lähetetään sarana-alkio kaikille
130   MPI_Bcast(&pivot, 1, MPI_INT, 0, MPI_COMM_CUBE);
131   // Lasketaan keskustelukumppanin järjestysnumero
132   partnerID = myCubeID ^ (1 << (curDim-1));
133
134   if(myCubeID < (sizeCubes / 2)){ // 1. Lähetä, 2. Vastaanota
        .
        .
        .
195   }
196   else{// 1. Vastaanota, 2. Lähetä
        .
        .
        .
260   }
261   j++;
262 }
```

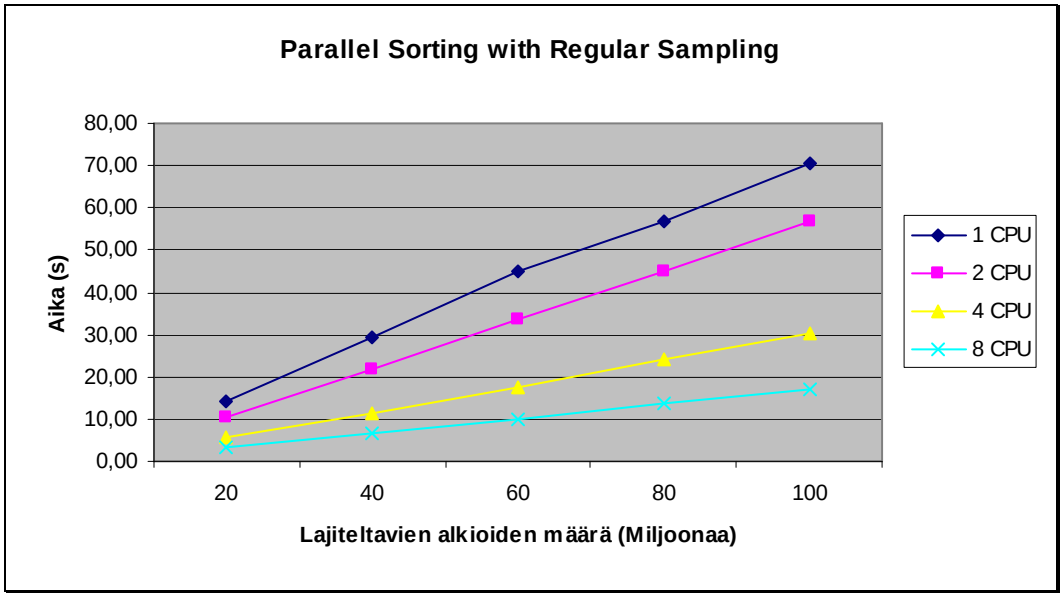
Lähdekoodi 15: Hyperkuution läpikäyntisilmukka.

Lähdekoodissa 15 oleva hyperkuution läpikäyntisilmukka on sama kuin edellisessä algoritmissa sillä poikkeuksella, että nyt alkiot lajitellaan jokaisen kierroksen alussa (rivi 109) ja sarana-alkioksi valitaan alkioiden todellinen mediaani (rivit 122 - 128).

7.5 PSRS (Parallel Sorting with Regular Sampling)

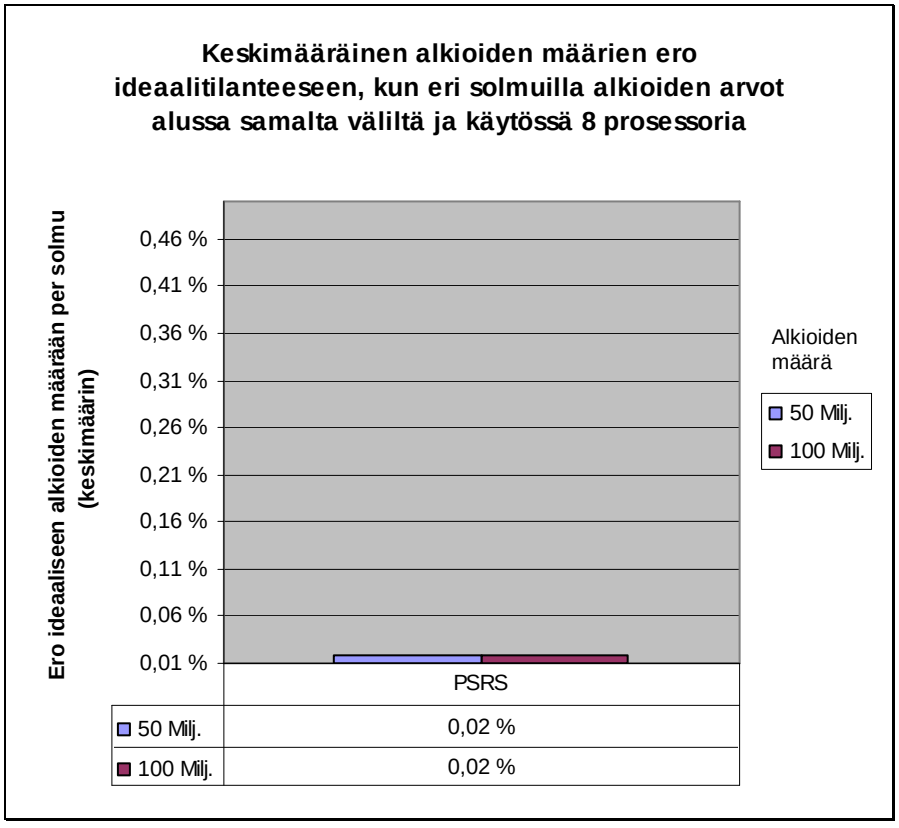
PSRS-algoritmilla on kolme etua verrattuna edellä esitettyihin algoritmeihin. Se pitää eri laskentatehtävillä olevien listojen koot vielä tarkemmin samoina, samoja alkioita ei lähetetä montaa kertaa laskentatehtävältä toiselle, eikä laskentatehtävien määrän tarvitse olla kahden potenssi.

Hyperquicksort-algoritmin tavoin myös PSRS-algoritmissa jokainen laskentatehtävä lajittelee ensin omat alkionsa järjestykseen. Tämän jälkeen jokainen laskentatehtävä ottaa säännöllisin välein (näytealkiot yhtä kaukana toisistaan) omista alkioistaan p kappaletta näytteitä, kun p on laskentatehtävien lukumäärä. Nämä näytteet lähetetään laskentatehtävälle 0, joka lajittelee kaikki näytteet järjestykseen. Tämän jälkeen laskentatehtävä 0 valitsee näytteistä p - 1 kappaletta käytettäviä sarana-alkioita ja lähettää ne kaikille laskentatehtäville. Seuraavaksi jokainen laskentatehtävä käy omat alkionsa läpi ja lähettää sarana-alkiota s_n pienemmät alkiot laskentatehtävälle p_n ja suuremmat alkiot laskentatehtävälle $p_{(n+1)}$, niin että n saa kaikki arvot välillä $0 \leq n \leq p - 1$. Kun alkiot on vastaanotettu kaikilta muilta laskentatehtäviltä, lajitellaan ne vielä kerran. Tässä vaiheessa molemmat järjestetyn aineiston ehdot ovat voimassa.



Kuva 19: Algoritmin suoritusajat eri alkioiden ja prosessorien määrillä.

Kuvassa 19 on havainnollistettu PSRS-algoritmin suoritusaikoja eri prosessorien ja lajiteltavien alkioiden määrillä. Kuvasta huomataan, että erot yhden prosessorin suoritusaikoihin ovat huomattavia, etenkin kun käytössä on 4 tai 8 prosessoria.



Kuva 20: Solmujen hallussa olevien alkioiden määrän ero ideaalitilanteeseen keskimäärin.

Kuvasta 20 nähdään, että työmäärät jakautuvat erittäin tasaisesti laskentatehtäville. Vielä Hyperquicksort-algoritmiakin tasaisemmat työmäärät johtuvat siitä, että lajittelussa huomioidaan jokaisen laskentatehtävän hallussa oleva aineisto.

Lähdekoodin läpikäynti

Tässä osiossa käydään läpi algoritmin lähdekoodia siltä osin, kuin se eroaa edellä esitetyistä. Lähdekoodi löytyy kokonaisuudessaan liitteestä 4. Lähdekoodin alussa esitellään tarvittavat muuttujat (lähdekoodi 16).

```
42  MPI_Status status;     // MPI-funktioiden tarvitsema status-muuttuja
43      double elapsedTime;// Ajanmittaukseen käytettävä muuttuja
44         int i,          // Apumuuttuja silmukoihin
45             j,          // Apumuuttuja silmukoihin
46             tag=0,      // MPI-viestintäfunktioissa käytettävä tunniste
47             p,          // Laskentatehtävien lukumäärä
48             myID,       // Oma järjestysnumero
49             partnerID,  // Keskustelukumppanin järjestysnumero
50             n,          // Data-alkioiden kokonaismäärä
51             nPerProc=0, // Yhden laskentatehtävän hallussa olevien
                              alkioiden lkm
52             check = 0,  // Apumuuttuja
53             sampleSize, // Näytteessä olevien alkioiden lkm
54             nRecvSum,   // Apumuuttuja vastaanotettujen alkioiden
                              laskemiseen
55             *nRecv,     // Taulukko vastaanotettujen alkioiden määristä
56             *sendCounts,// Lähetettävien alkioiden lukumäärät
57             *dataStart; // Indeksit, joista lähetettävät alkiot alkavat
58       float *myData,    // Laskentatehtävän hallussa oleva data
59             *buf,       // Tilapäinen puskuri alkioille
60             **recvData, // 2-ulotteinen taulukko vastaanottopuskureille
61             *pivots,    // Taulukko sarana-alkioille
62             *samples,   // Omat näytealkiot
63             *samplesFromAll;// Kerätyt näytealkiot
```

Lähdekoodi 16: Muuttujien esittelyt.

Lähdekoodissa 16 on muuttujien esittelyn yhteydessä myös selvitetty niiden käyttötarkoitus.

```
69  // Asetetaan näytealkioiden määrä per laskentatehtävä
70  sampleSize = p;
```

Lähdekoodi 17: Näytealkioiden määrän valitseminen.

Lähdekoodissa 17 on valittu käytettäväksi näytealkioiden määräksi laskentatehtävien lukumäärä. Näin toteutettuna otettavien näytteiden määrää on helppo muuttaa.

```
110 // Lajitellaan alkiot jos niitä on enemmän kuin yksi
111 if(nPerProc > 1)
112   qsort(myData, nPerProc, sizeof(float), compare);
113
114 // --- OTETAAN NÄYTTEET ---
115 if( (samples = (float*)malloc(sampleSize*sizeof(float))) == 0)
116   printf("Not enough memory for myData[%d]\n",n);
117 for(i=0; i<sampleSize; i++){
118   samples[i] = myData[((i+1)*(nPerProc/sampleSize)) - 1 -
                          ((nPerProc/sampleSize)/2)];
119 }
120 // ------------------------
121 // --- KERÄTÄÄN NÄYTTEET KAIKILTA ---
122 if(myID==0){
123   if( (samplesFromAll = (float*)malloc((p*sampleSize)*sizeof(float)))
== 0 )
124     printf("Not enough memory for myData[%d]\n",n);
125  }
126  MPI_Gather(samples, sampleSize, MPI_FLOAT, samplesFromAll, sampleSize,
MPI_FLOAT, 0, MPI_COMM_WORLD);
127  free(samples);
128  // --------------------------------
```

Lähdekoodi 18: Näytteiden ottaminen ja kerääminen.

Algoritmin alussa jokainen laskentatehtävä ottaa omista lajitelluista alkioistaan asetetun määrän näytteitä. Näytteet otetaan algoritmin nimen mukaisesti säännöllisin välein (rivit 117 - 119). Näytealkioiden indeksien (paikka taulukossa) laskemisessa on huomioitu alkioiden ja otettavien näytteiden lukumäärät. Näytteiden ottamisen jälkeen laskentatehtävä 0 kerää kaikki otetut näytteet itselleen (rivi 126).

```
130 // --- MUODOSTETAAN SARANA-ALKIOT ---
131 if( (pivots = (float*)malloc((p-1)*sizeof(float))) == 0 )
132   printf("Not enough memory for pivots[%d]\n",(p-1));
133 if(myID == 0){
134   // Lajitellaan näytteet
135   qsort(samplesFromAll, (p*sampleSize), sizeof(float), compare);
136   // Valitaan p-1 käytettävää sarana-alkiota
137   for(i=0; i<(p-1); i++){
138     pivots[i] = keskiarvo(samplesFromAll+(i*sampleSize), sampleSize*2);
139   }
140   free(samplesFromAll);
141 }
142 // --------------------------------
143
144 // Lähetetään käytettävät sarana-alkiot kaikille
145 MPI_Bcast(pivots, (p-1), MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Lähdekoodi 19: Lajittelussa käytettävien sarana-alkioiden muodostaminen.

Lähdekoodissa 19 nähdään, kuinka seuraavaksi muodostetaan käytettävät sarana-alkiot. Ensimmäiseksi kerätyt näytteet lajitellaan järjestykseen (rivi 135). Tämän jälkeen näytealkioista muodostetaan käytettävät sarana-alkiot liukuvan keskiarvon avulla (rivi 138). Lopuksi muodostetut sarana-alkiot lähetetään kaikille laskentatehtäville (rivi 145).

```
147  // --- ETSITÄÄN OMISTA ALKIOISTA LÄHETETTÄVÄT ALKIOT ---
148  if((sendCounts = (int*)calloc(p,sizeof(int)))==0)
149    printf("Not enough memory for sendCounts[%d]\n",p);
150  if((dataStart = (int*)calloc(p,sizeof(int)))==0)
151    printf("Not enough memory for dataStart[%d]\n",p);
152
153  j=1;
154  for(i=0; i<nPerProc; i++){
155    if(myData[i] < pivots[j-1]){
156      dataStart[j] = i+1;
157    }
158    else{
159      j++;
160      if(j==p)
161    break;
162      i--;
163    }
164  }
165  for(i=j; i<p; i++){
166    dataStart[i] = nPerProc;
167  }
168  for(i=1; i<p; i++){
169    if(dataStart[i] == 0){
170      dataStart[i] = dataStart[i-1];
171    }
172  }
173  sendCounts[0] = dataStart[1];
174  check = 0;
175  for(i=1; i<(p-1); i++){
176    if(dataStart[i] == nPerProc -1){
177      sendCounts[i] = 1;
178      check = 1;
179      break;
180    }
181    sendCounts[i] = dataStart[i+1] - dataStart[i];
182  }
183  if(check == 0)
184    sendCounts[p-1] = nPerProc - dataStart[p-1];
185
186  for(i=0; i<p; i++){
187    if(dataStart[i] > (nPerProc-1))
188      dataStart[i] = nPerProc-1;
189  }
190  // ------------------------------------------------
```

Lähdekoodi 20: Lähetettävien alkioiden etsintä.

Koska kaikki käytettävät sarana-alkiot muodostettiin kerralla ja ilmoitettiin kaikille, jokainen laskentatehtävä voi nyt jakaa alkionsa yhtä moneen osaan, kuin on laskentatehtäviä. Lähdekoodissa 20 nähdään, kuinka alkioista etsitään eri laskentatehtäville lähetettävien alkioiden alkupositio ja määrä (rivit 148 - 189).

```
194 // --- SIIRRETÄÄN ALKIOT OIKEILLE LASKENTATEHTÄVILLE ---
195 if( (recvData = (float**)malloc(p*sizeof(float*))) == 0 )
196   printf("Not enough memory for **recvData[%d]\n",p);
197 for(i=0; i<p; i++){
198   if((recvData[i] = (float*)malloc(nPerProc*sizeof(float))) == 0)
199     printf("Not enough memory for recvData[%d]\n",i);
200 }
201 if( (nRecv = (int*)malloc(p*sizeof(int))) == 0 )
202   printf("Not enough memory for nRecv[%d]\n",p);
```

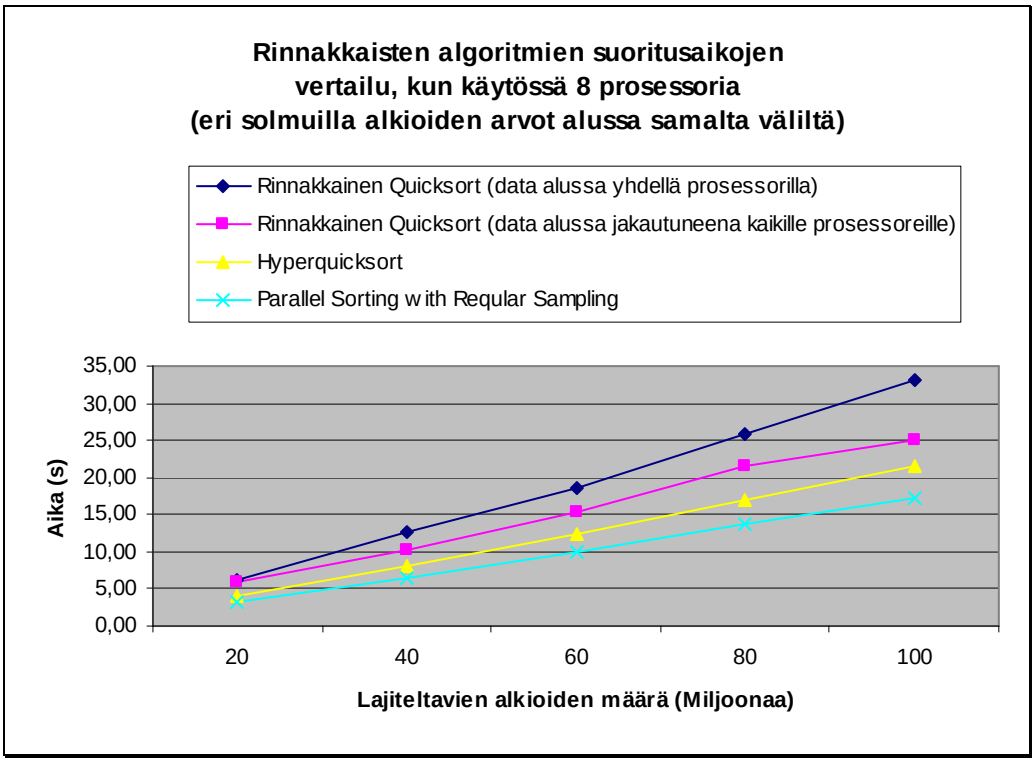
```
203
204 for(i=0; i<p; i++){
205   MPI_Scatter(sendCounts, 1, MPI_INT, &nRecv[i], 1, MPI_INT, i,
                  MPI_COMM_WORLD);
206   MPI_Scatterv(myData, sendCounts, dataStart, MPI_FLOAT, recvData[i],
                   nRecv[i], MPI_FLOAT, i, MPI_COMM_WORLD);
207 }
208 // ------------------------------------------------
```

Lähdekoodi 21: Alkioiden siirto laskentatehtävien välillä.

Varsinainen alkioiden siirto laskentatehtävien välillä tapahtuu lähdekoodissa 21. Aluksi jokainen varaa muistia jokaiselta laskentatehtävältä vastaanotettavia alkioita varten (rivit 195 - 202). Tämän jälkeen jokainen laskentatehtävä lähettää ensin kaikille lähetettävien alkioiden lukumäärän (laskentatehtäväkohtaisesti, rivi 205) ja tämän jälkeen tapahtuu itse alkioiden siirto (rivi 206).

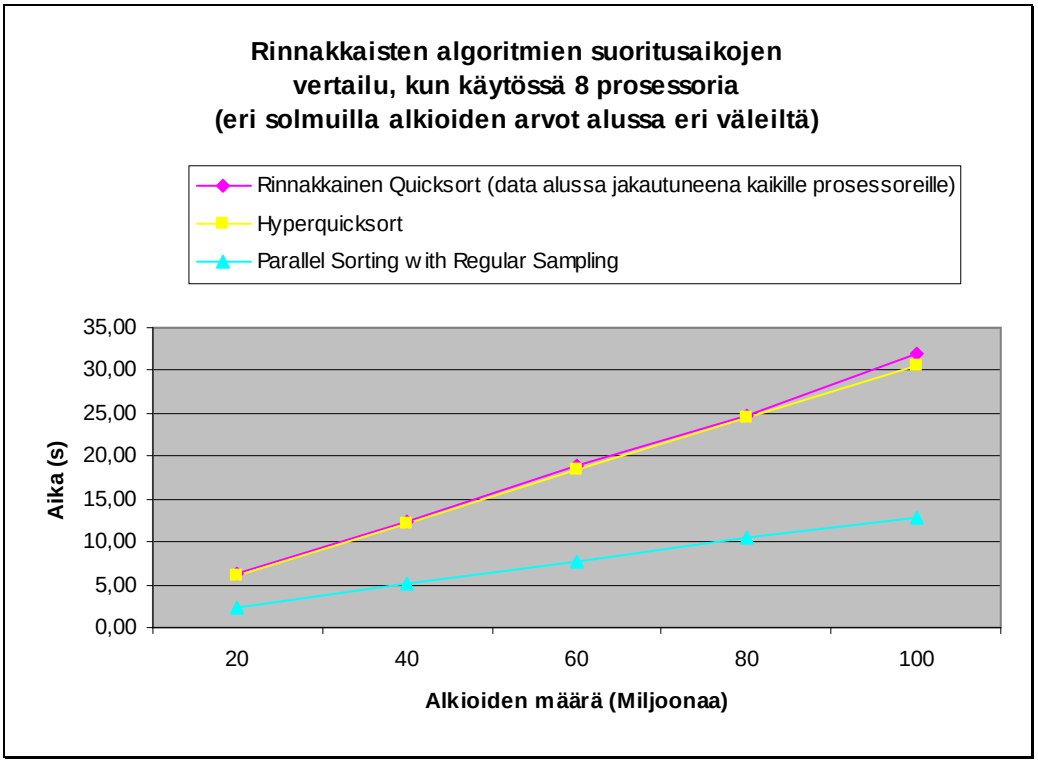
7.6 Mittaustulokset

Tässä luvussa on esitetty eri algoritmeilla saatuja mittaustuloksia ja vertailtu niitä keskenään. Aidot rinnakkaiset lajittelutilanteet voidaan jakaa karkeasti kahteen ryhmään sen mukaan, ovatko eri solmujen hallussa olevien alkioiden arvot samalta väliltä vai onko niillä keskenään erisuuruista dataa. Tästä syystä myös mittaustulokset on esitetty molemmista tilanteista.



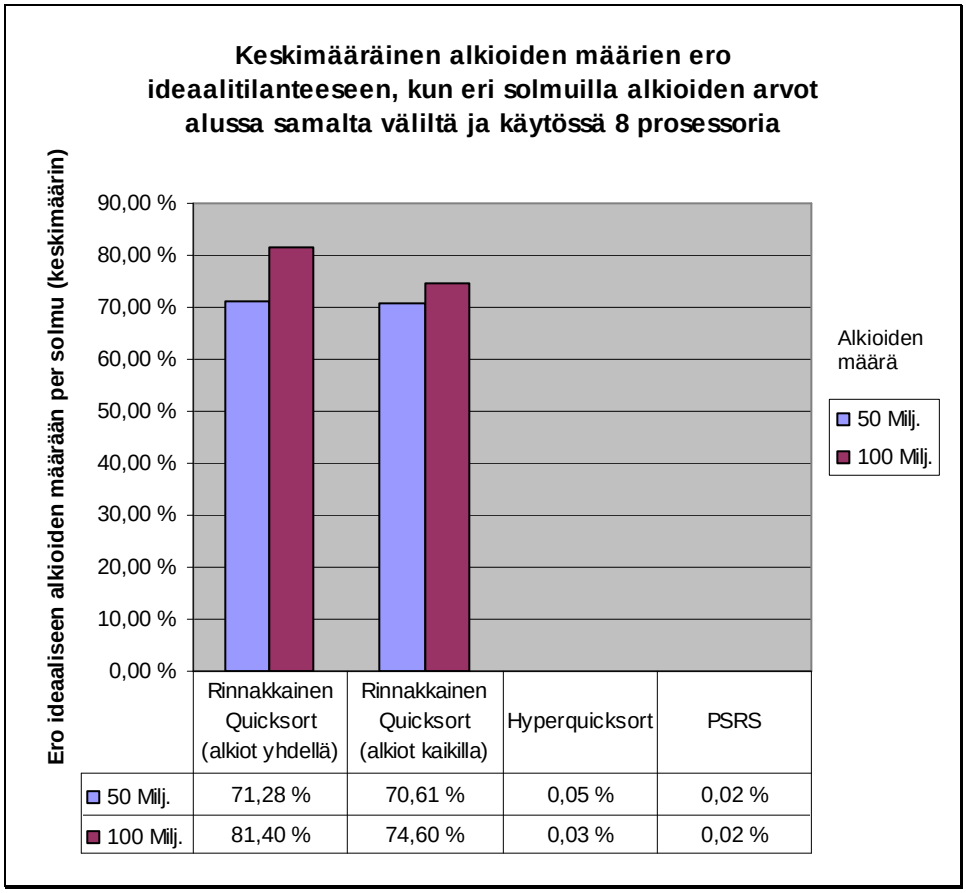
Kuva 21: Algoritmien suoritusajat, kun eri solmuilla alkioiden arvot alussa samalta väliltä.

Kuvassa 21 on vertailtu työssä esitettyjen rinnakkaisten algoritmien suoritusaikoja eri lajiteltavien alkioiden määrillä. Kuvasta huomataan, että PSRS-algoritmi on kahdeksaa prosessoria käytettäessä kaikkia muita nopeampi.



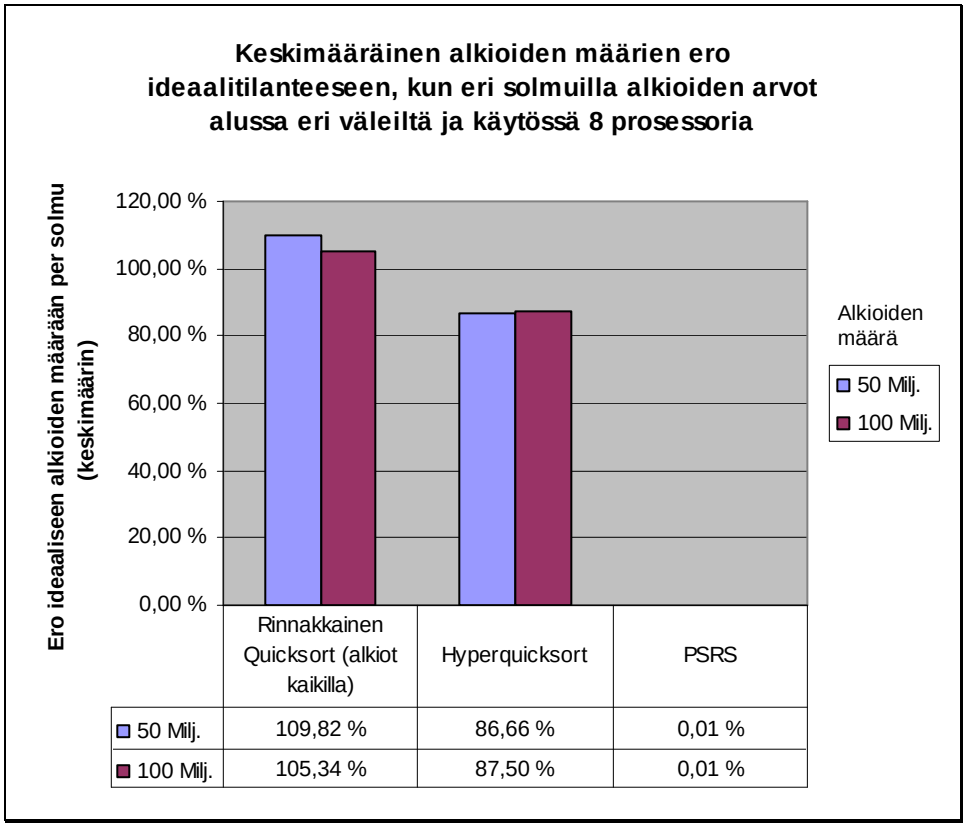
Kuva 22: Algoritmien suoritusajat, kun eri solmuilla alkioiden arvot alussa eri väleiltä.

Kuvasta 22 huomataan, että eri solmuilla olevien alkioiden arvojen ollessa eri väleiltä, ero PSRS-algoritmin ja muiden välillä kasvaa huomattavasti. Ensimmäiseksi esitetty rinnakkainen algoritmi on tästä kuvasta jätetty pois, sillä siinä kaikki alkiot ovat alussa yhdellä solmulla. Kuvaan 21 verrattuna on mielenkiintoista, että tällaisessa tilanteessa muiden algoritmien suorituskyky muuttuu huonommaksi ja PSRS-algoritmin paremmaksi.



Kuva 23: Solmujen hallussa olevien alkioiden määrän ero ideaalitilanteeseen keskimäärin, kun eri solmuilla alkioiden arvot alussa samalta väliltä.

Kuvassa 23 on esitetty työmäärien jakautumisessa esiintyneitä eroja. Algoritmeissa, jotka valitsevat sarana-alkion lajitellusta aineistosta (Hyperquicksort ja PSRS), työmäärät jakautuvat huomattavasti tasaisemmin.



Kuva 24: Solmujen hallussa olevien alkioiden määrän ero ideaalitilanteeseen keskimäärin, kun eri solmuilla alkioiden arvot alussa eri väleiltä.

PSRS-algoritmin suurin etu verrattuna muihin esitettyihin rinnakkaisiin algoritmeihin on se, että sarana-alkioita valittaessa otetaan huomioon kaikilla laskentatehtävillä oleva data. Erityisesti jos jokaisella laskentatehtävällä on keskenään erisuuruista dataa, esim. laskentatehtävällä 0 numeroalkioita väliltä 0 - 100, laskentatehtävällä 1 väliltä 101 - 200, jne, on PSRS-algoritmi ylivertaisesti tehokkain esitetyistä algoritmeista. Tällaista tilannetta on havainnollistettu kuvassa 24. Näissä tilanteissa työmäärien tasainen jakautuminen epäonnistuu pahasti myös Hyperquicksort-algoritmissa.

8 YHTEENVETO

Tässä insinöörityössä tutkittiin aluksi rinnakkaisohjelmoinnin mahdollistavia arkkitehtuureja. Seuraavaksi rakennettiin kahdeksasta tietokoneesta korkean suorituskyvyn klusteri yhdistämällä tietokoneet verkkoyhteydellä toisiinsa ja asentamalla niihin klusteroinnin mahdollistava käyttöjärjestelmä. Tämän jälkeen käytiin läpi rinnakkaisohjelmoinnin perusteita ja tutkittiin erityisesti viestinvälitysmallin mukaista rinnakkaista ohjelmointia. Lopuksi suunniteltiin

ja toteutettiin alun perin sarjamuotoisesta algoritmista neljä erilaista rinnakkaista versiota.

Suoritetuissa testeissä vertailtiin koodattuja rinnakkaisia algoritmeja mittaamalla niiden suoritusaikoja sekä työmäärien jakautumista eri prosessorien ja lajiteltavien alkioiden määrillä. Algoritmien suunnittelu, koodaus ja tehdyt mittaukset onnistuivat erittäin hyvin ja niiden perusteella havaittiin rinnakkaisten ohjelmien suoritusaikojen pienenevän huomattavasti jo kahta prosessoria käytettäessä. Algoritmeja kehitettäessä ja testattaessa huomattiin eräitä suorituskykyyn merkittävästi vaikuttavia asioita, kuten sarana-alkion muodostaminen ja sen perusteella työmäärien jakautuminen laskentatehtäville.

Rinnakkaisohjelmoinnilla voidaan saada ohjelmiin merkittävästi lisää nopeutta. Rinnakkaisten ohjelmien kehitys on kuitenkin hankalampaa kuin sarjamuotoisten ohjelmien ja vaatii siksi huolellista perehtymistä rinnakkaislaskennan mahdollistavaan ympäristöön ja sen tuomiin haasteisiin.

VIITELUETTELO

[Barney] Barney Blaise. Introduction to Parallel Computing. Lawrence Livermore National Laboratory. [Verkkodokumentti, viitattu 25.9.2009]. Saatavissa: https://computing.llnl.gov/tutorials/parallel_comp/.

[Cormen] Cormen T, Leiserson C, Rivest R, Stein C. Introduction To Algorithms, Second Edition. 2003, The MIT Press. ISBN 0-07-013151-1

[Gropp] Gropp William, Lusk Ewing, Skjellum Anthony. Using MPI - Portable Parallel Programming with the Message Passing Interface, Second Edition. 1999, The MIT Press. ISBN 0-262-57132-3.

[Haataja] Haataja J, Mustikkamäki K. Rinnakkaisohjelmointi MPI:llä. 1997, CSC - Tieteellinen laskenta Oy. ISBN 952-9821-41-7.

[Haikala] Haikala Ilkka, Järvinen Hannu-Matti. Käyttöjärjestelmät, toinen painos. 2004, Talentum Media Oy. ISBN 952-14-0851-0.

[Merritt] Merritt Rick. Researchers report progress on parallel path. EE Times, 24.8.2009. [Verkkodokumentti, viitattu 25.9.2009]. Saatavissa: http://www.eetimes.com/showArticle.jhtml?articleID=219401095&pgno=1.

[MPI-1.3] MPI-1.3 Standard. [Verkkodokumentti, viitattu 25.9.2009]. Saatavissa: http://www.mpi-forum.org/docs/mpi-report-1.3-2008-05-30.pdf.

[MPI-2.0] MPI-2.0 Standard. [Verkkodokumentti, viitattu 25.9.2009]. Saatavissa: http://www.mpi-forum.org/docs/mpi2-report.pdf.

[Quinn] Quinn Michael. Parallel Programming in C with MPI and OpenMP. 2004, McGraw Hill. ISBN 0-07-282256-2.

[Stallings] Stallings William. Operating Systems. 2001, Prentice-Hall Inc. ISBN 0-13-032986-6.

[Viitanen] Viitanen Arto. Rinnakkaislaskenta. Tampereen yliopisto 2004. [Verkkodokumentti, viitattu 25.9.2009]. Saatavissa: http://www.cs.uta.fi/tarkki/suoritus/luennot/parallel.html.

[Wilkinson] Wilkinson B, Allen M. Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition. 2005, Pearson Prentice Hall. ISBN 0-13-191865-6.

```
1 /* Rinnakkainen versio Quicksort-algoritmista
2  * ========================================
3  *
4  * Lajiteltavat alkiot aluksi laskentatehtävässä 0
5  * ja lopuksi jaettuna kaikille laskentatehtäville niin,
6  * että molemmat järjestetyn rinnakkaisen aineiston
7  * ehdot ovat voimassa.
8  *
9  * Copyright (c) Juha Katajisto
10 */
11
12 #include <stdlib.h>
13 #include <stdio.h>
14 #include <time.h>
15 #include <math.h>
16 #include "mpi.h"
17
18 // qsort-funktion tarvitsema alkioiden vertailufunktio
19 int compare (const void * a, const void * b)
20 {
21     if(*(float*)a != *(float*)b){
22         if(*(float*)a < *(float*)b)
23             return -1;
24         else
25             return 1;
26     }
27     else
28         return 0;
29 }
30
31 main(int argc, char **argv ){
32   MPI_Status status;     // MPI-funktioiden tarvitsema status-muuttuja
33       double elapsedTime;// Ajanmittaukseen käytettävä muuttuja
34          int i,          // Apumuuttuja silmukoihin
35              j,          // Apumuuttuja silmukoihin
36              tag=0,      // MPI-viestintäfunktioissa käytettävä tunniste
37              p,          // Laskentatehtävien lukumäärä
38              myID,       // Oma järjestysnumero
39              partnerID,  // Keskustelukumppanin järjestysnumero
40              cubeSize,   // Hyperkuution koko
41              dim,        // Suurimman hyperkuution ulottuvuuksien lkm
42              curDim,     // Nykyisen hyperkuution ulottuvuuksien lkm
43              n,          // Data-alkioiden kokonaismäärä
44              nPerProc=0, // Yhden laskentatehtävän hallussa olevien
                               alkioiden lkm
45              nSmaller,   // Sarana-alkiota pienempien alkioiden lkm
46              nLarger;    // Sarana-alkiota suurempien alkioiden lkm
47        float pivot,      // Sarana-alkio
48              *myData,    // Laskentatehtävän hallussa oleva data
49       *smallerThanPivot,// Sarana-alkiota pienemmät alkiot
50        *largerThanPivot;// Sarana-alkiota suuremmat alkiot
51
52   MPI_Init(&argc, &argv);
53   MPI_Comm_size(MPI_COMM_WORLD,&p);
54   MPI_Comm_rank(MPI_COMM_WORLD, &myID);
55
56   n=0;
57   if(argc != 2){
58     if(myID == 0)
```

```
59        printf("Usage: parallel-1 n, where n is number of items to be sorted\n"
);
60      exit(1);
61    }
62    sscanf(argv[1],"%d",&n);
63    if(n<1){
64      if(myID == 0)
65        printf("Usage: parallel-1 n, where n is number of items to be sorted\n"
);
66      exit(1);
67      }
68
69    // --- LUKUJEN LATAUS ---
70    if(myID == 0){
71      nPerProc = n;
72      if( (myData = (float*)malloc(n*sizeof(float))) == 0)
73        printf("Not enough memory for myData[%d]\n" ,n);
74      srand48(time(NULL));
75      for(i=0; i<n; i++){
76        myData[i] = drand48();
77      }
78    }
79    // ----------------------
80
81    // Odotetaan että kaikki laskentatehtävät pääsevät tähän asti ja
82    // aloitetaan ajan mittaaminen
83    MPI_Barrier(MPI_COMM_WORLD);
84    elapsedTime = MPI_Wtime();
85
86    // Jokainen laskentatehtävä kuuluu ensimmäiseen hyperkuutioon
87    cubeSize = p;
88
89    // Lasketaan käytettävissä olevien ulottuvuuksien lukumäärä
90    dim = ceil(log((double)p) / (log(2.0)));
91
92    // Käydään hyperkuution ulottuvuudet läpi suurimmasta pienimpiin
93    for(curDim=dim; curDim>0; curDim--){
94      if(myID % (cubeSize / 2) == 0){ // Olen lähettäjä tai vastaanottaja
95        // Lasketaan keskustelukumppanin järjestysnumero
96        partnerID = myID ^ (1 << (curDim-1));
97        if(myID < partnerID){ // Olen lähettäjä
98          pivot = (myData[0] + myData[nPerProc - 1]) / 2;
99          // Muodostetaan osalistat sarana-alkiota pienemmille ja yhtäsuurille
100         // sekä suuremmille
101         if( (smallerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0)
102           printf("Not enough memory for smallerThanPivot[%d]\n" ,nPerProc);
103         if( (largerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0)
104           printf("Not enough memory for largerThanPivot[%d]\n" ,nPerProc);
105         nSmaller = 0;
106         nLarger = 0;
107         for(j = 0; j < nPerProc; j++){
108           if(myData[j] <= pivot) {
109             smallerThanPivot[nSmaller] = myData[j];
110             nSmaller++;
111           }
112           else {
113             largerThanPivot[nLarger] = myData[j];
114             nLarger++;
115           }
```

```
116         }
117         MPI_Send(largerThanPivot, nLarger, MPI_FLOAT, partnerID, tag,
                    MPI_COMM_WORLD);
118         free(myData);
119         myData = smallerThanPivot;
120         nPerProc = nSmaller;
121         free(largerThanPivot);
122       }
123       else{ // Olen vastaanottaja
124         MPI_Probe(partnerID, tag, MPI_COMM_WORLD, &status);
125         MPI_Get_count(&status, MPI_INT, &nPerProc);
126         if((myData = (float *) malloc(nPerProc*sizeof(float))) == 0)
127           printf("Not Enough Memory for myData[%d]\n" ,nPerProc);
128         MPI_Recv(myData, nPerProc, MPI_FLOAT, partnerID, tag, MPI_COMM_WORLD,
                    &status);
129       }
130     }
131     // Puolitetaan kuutiot
132     cubeSize /= 2;
133   }
134
135   // Lajitellaan alkiot jos niitä on enemmän kuin yksi
136   if(nPerProc > 1)
137     qsort(myData, nPerProc, sizeof(float), compare);
138
139   // Lopetetaan ajan mittaaminen
140   elapsedTime = MPI_Wtime() - elapsedTime;
141
142   //-------VIRHEIDEN TARKISTUS------
143   int errors = 0;
144   for(i=0; i<nPerProc-1; i++){
145     if(myData[i] > myData[i+1])
146       errors++;
147   }
148   if(errors > 0) //tieto virheistä vain jos niitä on
149     printf("myID: %d, number of errors: %d\n" , myID, errors);
150
151   if(myID==0){
152     errors = 0;
153     float *tests;
154     if( (tests = (float*)malloc((p*2)*sizeof(float))) == 0)
155       printf("Not enough memory for tests[%d]\n" ,(p*2));
156     tests[0] = myData[0];
157     tests[1] = myData[nPerProc-1];
158     for(i=1; i<p; i++){
159       MPI_Recv(tests+(i*2), 2, MPI_FLOAT, i, tag, MPI_COMM_WORLD, &status);
160     }
161     for(i=1; i<(p*2); i++){
162       if(tests[i-1] > tests[i])
163     errors++;
164     }
165     if(errors > 0) //tieto virheistä vain jos niitä on
166       printf("Number of errors between processors: %d\n" ,errors);
167   }
168   else{
169     float tests[2];
170     tests[0] = myData[0];
171     tests[1] = myData[nPerProc-1];
172     MPI_Send(tests, 2, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
```

```
173   }
174   //--------------------------------
175
176   // Tulostetaan näytölle oma järjestysnumero, käytetty prosessoriaika ja
177   // hallussa olevien alkioiden lukumäärä
178   fprintf(stdout,"Processor Id: %d;  The CPU Time: %10.6f;  SubArray Length:
                  %d.\n", myID, elapsedTime, nPerProc);
179   fflush(stdout);
180
181   free(myData);
182   MPI_Finalize();
183 }
```

```
1 /* Rinnakkainen versio Quicksort-algoritmista
2  * ========================================
3  *
4  * Lajiteltavat alkiot jakautuneet aluksi kaikille
5  * laskentatehtäville (ei kuitenkaan missään erityisessä
6  * järjestyksessä). Lopuksi alkiot jaettuna kaikille
7  * laskentatehtäville niin, että molemmat järjestetyn
8  * rinnakkaisen aineiston ehdot ovat voimassa.
9  *
10 * Copyright (c) Juha Katajisto
11 */
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 #include <time.h>
16 #include <math.h>
17 #include "mpi.h"
18
19 // qsort-funktion tarvitsema alkioiden vertailufunktio
20 int compare (const void * a, const void * b)
21 {
22   if(*(float*)a != *(float*)b){
23     if(*(float*)a < *(float*)b)
24       return -1;
25     else
26       return 1;
27   }
28   else
29     return 0;
30 }
31
32 main(int argc, char **argv ){
33   MPI_Status status;       // MPI-funktioiden tarvitsema status-muuttuja
34     MPI_Comm MPI_COMM_CUBE;// Viestintäryhmä alikuutioille
35       double elapsedTime;  // Ajanmittaukseen käytettävä muuttuja
36          int i,            // Apumuuttuja silmukoihin
37              j,            // Apumuuttuja silmukoihin
38              tag=0,        // MPI-viestintäfunktioissa käytettävä tunniste
39              p,            // Laskentatehtävien lukumäärä
40              myID,         // Oma järjestysnumero
41              partnerID,    // Keskustelukumppanin järjestysnumero
42              numCubes,     // Hyperkuutioiden lukumäärä
43              sizeCubes,    // Hyperkuutioiden koko
44              myCubeNum,    // Oman hyperkuution numero
45              myCubeID,     // Järjestysnumero nykyisessä hyperkuutiossa
46              dim,          // Suurimman hyperkuution ulottuvuuksien lkm
47              curDim,       // Nykyisen hyperkuution ulottuvuuksien lkm
48              n,            // Data-alkioiden kokonaismäärä
49              nPerProc=0,   // Yhden laskentatehtävän hallussa olevien
                                 alkioiden lkm
50              nRecv,        // Vastaanotettavien / -otettujen alkioiden lkm
51              nSmaller,     // Sarana-alkiota pienempien alkioiden lkm
52              nLarger;      // Sarana-alkiota suurempien alkioiden lkm
53        float pivot,        // Sarana-alkio
54              *myData,      // Laskentatehtävän hallussa oleva data
55          *smallerThanPivot,// Sarana-alkiota pienemmät alkiot
56           *largerThanPivot,// Sarana-alkiota suuremmat alkiot
57              *buf;         // Tilapäinen puskuri alkioille
58
```

```
59   MPI_Init(&argc, &argv);
60   MPI_Comm_size(MPI_COMM_WORLD,&p);
61   MPI_Comm_rank(MPI_COMM_WORLD, &myID);
62
63   n=0;
64   if(argc != 2){
65     if(myID == 0)
66       printf("Usage: parallel-n x, where x is number of items to be sorted\n"
);
67     exit(1);
68   }
69   sscanf(argv[1],"%d",&n);
70   if(n<1){
71     if(myID == 0)
72       printf("Usage: parallel-n x, where x is number of items to be sorted\n"
);
73     exit(1);
74   }
75
76   // Oletus: n on jaollinen p:llä
77   nPerProc = n/p;
78
79   // --- LUKUJEN LATAUS ---
80   if(myID == 0){
81     if( (buf = (float*)malloc(n*sizeof(float))) == 0 )
82       printf("Not enough memory for buf[%d]\n" ,n);
83     srand48(time(NULL));
84     for(i=0; i<n; i++){
85       buf[i] = drand48();
86     }
87   }
88   // ----------------------
89
90   // Jaetaan luvut tasan kaikille laskentatehtäville
91   if( (myData = (float*)malloc(nPerProc*sizeof(float))) == 0 )
92       printf("Not enough memory for myData[%d]\n" ,nPerProc);
93   MPI_Scatter(buf, nPerProc, MPI_FLOAT, myData, nPerProc, MPI_FLOAT, 0,
               MPI_COMM_WORLD);
94
95   // Odotetaan että kaikki laskentatehtävät pääsevät tähän asti ja
96   // aloitetaan ajan mittaaminen
97   MPI_Barrier(MPI_COMM_WORLD);
98   elapsedTime = MPI_Wtime();
99
100   // Lasketaan käytettävissä olevien ulottuvuuksien lukumäärä
101   dim = ceil(log((double)p) / (log(2.0)));
102
103   j=0;
104   for(curDim=dim; curDim>0; curDim--){
105     // Lasketaan hyperkuutioiden lukumäärä
106     numCubes = 1 << j;
107     // Lasketaan hyperkuutioiden koko
108     sizeCubes = p / numCubes;
109     // Lasketaan oman hyperkuution numero
110     myCubeNum = myID >> curDim;
111     // Jaetaan tarvittaessa nykyinen kuutio alikuutioiksi
112     MPI_Comm_split(MPI_COMM_WORLD, myCubeNum, myID, &MPI_COMM_CUBE);
113     // Selvitetään oma järjestysnumero uudessa hyperkuutiossa
114     MPI_Comm_rank(MPI_COMM_CUBE, &myCubeID);
```

```
115
116     if(myCubeID == 0){
117       // Muodostetaan sarana-alkio
118       pivot = (myData[0] + myData[nPerProc - 1]) / 2;
119     }
120     // Lähetetään sarana-alkio kaikille
121     MPI_Bcast(&pivot, 1, MPI_INT, 0, MPI_COMM_CUBE);
122     // Lasketaan keskustelukumppanin järjestysnumero
123     partnerID = myCubeID ^ (1 << (curDim - 1));
124
125     if(myCubeID < (sizeCubes / 2)){ //1. Lähetä, 2. Vastaanota
126       // --- LÄHETETÄÄN ALKIOT, JOTKA SUUREMPIA KUIN SARANA-ALKIO ---
127       if( (smallerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
128         printf("Not enough memory for smallerThanPivot[%d]\n" ,nPerProc);
129       if( (largerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
130         printf("Not enough memory for largerThanPivot[%d]\n" ,nPerProc);
131       nSmaller = 0;
132       nLarger = 0;
133       for(i=0; i<nPerProc; i++){
134         if (myData[i] <= pivot) {
135           smallerThanPivot[nSmaller] = myData[i];
136           nSmaller++;
137         }
138         else {
139           largerThanPivot[nLarger] = myData[i];
140           nLarger++;
141         }
142       }
143       MPI_Send(largerThanPivot, nLarger, MPI_FLOAT, partnerID, tag,
                  MPI_COMM_CUBE);
144       free(myData);
145       myData = smallerThanPivot;
146       nPerProc = nSmaller;
147       free(largerThanPivot);
148       // ------------------------------------------------------------
149       // --- VASTAANOTETAAN ALKIOT, JOTKA PIENEMPIÄ TAI YHTÄSUURIA KUIN S-
             ALKIO ---
150       MPI_Probe(partnerID, tag, MPI_COMM_CUBE, &status);
151       MPI_Get_count(&status, MPI_INT, &nRecv);
152       if(nRecv > 0){
153         nPerProc += nRecv;
154         if((buf = (float *) malloc(nPerProc*sizeof(float))) == 0 )
155           printf("Not Enough Memory for buf[%d]\n" ,nPerProc);
156         MPI_Recv(buf, nRecv, MPI_FLOAT, partnerID, tag, MPI_COMM_CUBE,
                    &status);
157         for(i=nRecv; i<nPerProc; i++){
158           buf[i] = myData[i-nRecv];
159         }
160         free(myData);
161         myData = buf;
162       }
163       // ---------------------------------------------------------------
164     }
165     else{// 1. Vastaanota, 2. Lähetä
166       // --- VASTAANOTETAAN ALKIOT, JOTKA SUUREMPIA KUIN SARANA-ALKIO ---
167       MPI_Probe(partnerID, tag, MPI_COMM_CUBE, &status);
168       MPI_Get_count(&status, MPI_INT, &nRecv);
169       nPerProc += nRecv;
170       if((myData = (float *) realloc(myData,nPerProc*sizeof(float))) == 0)
```

```
171         printf("Not Enough Memory for myData[%d]\n" ,nPerProc);
172       MPI_Recv(myData+(nPerProc-nRecv), nRecv, MPI_FLOAT, partnerID, tag,
                  MPI_COMM_CUBE, &status);
173       // ------------------------------------------------------------
174       // --- LÄHETETÄÄN ALKIOT, JOTKA PIENEMPIÄ TAI YHTÄSUURIA KUIN S-ALKIO
175       if( (smallerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
176         printf("Not enough memory for myData[%d]\n" ,nPerProc);
177       if( (largerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
178         printf("Not enough memory for myData[%d]\n" ,nPerProc);
179       nSmaller = 0;
180       nLarger = 0;
181       for(i=0; i<nPerProc; i++){
182         if (myData[i] <= pivot) {
183             smallerThanPivot[nSmaller] = myData[i];
184           nSmaller++;
185         }
186         else {
187           largerThanPivot[nLarger] = myData[i];
188           nLarger++;
189         }
190       }
191       MPI_Send(smallerThanPivot, nSmaller, MPI_FLOAT, partnerID, tag,
                  MPI_COMM_CUBE);
192       free(myData);
193       myData = largerThanPivot;
194       nPerProc = nLarger;
195       free(smallerThanPivot);
196       // ------------------------------------------------------------------
197     }
198     j++;
199   }
200
201   // Lajitellaan alkiot jos niitä on enemmän kuin yksi
202   if(nPerProc > 1)
203     qsort(myData, nPerProc, sizeof(float), compare);
204
205   // Lopetetaan ajan mittaaminen
206   elapsedTime = MPI_Wtime() - elapsedTime;
207
208   //-------VIRHEIDEN TARKISTUS------
209   int errors = 0;
210   for(i=0; i<nPerProc-1; i++){
211     if(myData[i] > myData[i+1])
212       errors++;
213   }
214   if(errors > 0) // Tieto virheistä vain jos niitä on
215     printf("myID: %d, number of errors: %d\n" , myID, errors);
216
217   if(myID==0){
218     errors = 0;
219     float *tests;
220     if( (tests = (float*)malloc((p*2)*sizeof(float))) == 0)
221       printf("Not enough memory for tests[%d]\n" ,(p*2));
222     tests[0] = myData[0];
223     tests[1] = myData[nPerProc-1];
224     for(i=1; i<p; i++){
225       MPI_Recv(tests+(i*2), 2, MPI_FLOAT, i, tag, MPI_COMM_WORLD, &status);
226     }
227     for(i=1; i<(p*2); i++){
```

```
228       if(tests[i-1] > tests[i])
229     errors++;
230     }
231     if(errors > 0) // Tieto virheistä vain jos niitä on
232       printf("Number of errors between processors: %d\n" ,errors);
233   }
234   else{
235     float tests[2];
236     tests[0] = myData[0];
237     tests[1] = myData[nPerProc-1];
238     MPI_Send(tests, 2, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
239   }
240   //---------------------------------
241
242   // Tulostetaan näytölle oma järjestysnumero, käytetty prosessoriaika ja
243   // hallussa olevien alkioiden lukumäärä
244   fprintf(stdout,"Processor Id: %d;  The CPU Time: %10.6f;  SubArray Length:
                  %d.\n", myID, elapsedTime, nPerProc);
245   fflush(stdout);
246
247   free(myData);
248   MPI_Finalize();
249 }
```

```
1 /* Hyperquicksort-algoritmi
2  * ========================
3  *
4  * Lajiteltavat alkiot jakautuneet aluksi kaikille
5  * laskentatehtäville (ei kuitenkaan missään erityisessä
6  * järjestyksessä). Lopuksi alkiot jaettuna kaikille
7  * laskentatehtäville niin, että molemmat järjestetyn
8  * rinnakkaisen aineiston ehdot ovat voimassa.
9  *
10 * Copyright (c) Juha Katajisto
11 */
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 #include <time.h>
16 #include <math.h>
17 #include "mpi.h"
18
19 // qsort-funktion tarvitsema alkioiden vertailufunktio
20 int compare (const void * a, const void * b)
21 {
22   if(*(float*)a != *(float*)b){
23     if(*(float*)a < *(float*)b)
24       return -1;
25     else
26       return 1;
27   }
28   else
29     return 0;
30 }
31
32 main(int argc, char **argv ){
33   MPI_Status status;     // MPI-funktioiden tarvitsema status-muuttuja
34   MPI_Comm MPI_COMM_CUBE;// Viestintäryhmä alikuutioille
35       double elapsedTime;// Ajanmittaukseen käytettävä muuttuja
36          int i,          // Apumuuttuja silmukoihin
37              j,          // Apumuuttuja silmukoihin
38              tag=0,      // MPI-viestintäfunktioissa käytettävä tunniste
39              p,          // Laskentatehtävien lukumäärä
40              myID,       // Oma järjestysnumero
41              partnerID,  // Keskustelukumppanin järjestysnumero
42              numCubes,   // Hyperkuutioiden lukumäärä
43              sizeCubes,  // Hyperkuutioiden koko
44              myCubeNum,  // Oman hyperkuution numero
45              myCubeID,   // Järjestysnumero nykyisessä hyperkuutiossa
46              dim,        // Suurimman hyperkuution ulottuvuuksien lkm
47              curDim,     // Nykyisen hyperkuution ulottuvuuksien lkm
48              n,          // Data-alkioiden kokonaismäärä
49              nPerProc=0, // Yhden laskentatehtävän hallussa olevien
                               alkioiden lkm
50              nRecv,      // Vastaanotettavien / -otettujen alkioiden lkm
51              ownCount,   // Apumuuttuja listojen yhdistämiseen
52              recvCount,  // Apumuuttuja listojen yhdistämiseen
53              nSmaller,   // Sarana-alkiota pienempien alkioiden lkm
54              nLarger;    // Sarana-alkiota suurempien alkioiden lkm
55        float pivot,      // Sarana-alkio
56              *myData,    // Laskentatehtävän hallussa oleva data
57        *smallerThanPivot,// Sarana-alkiota pienemmät alkiot
58         *largerThanPivot,// Sarana-alkiota suuremmat alkiot
```

```
59              *recvBuf,   // Vastaanottopuskuri
60              *buf;       // Tilapäinen puskuri alkioille
61
62    MPI_Init(&argc, &argv);
63    MPI_Comm_size(MPI_COMM_WORLD,&p);
64    MPI_Comm_rank(MPI_COMM_WORLD, &myID);
65
66    n=0;
67    if(argc != 2){
68      if(myID == 0)
69        printf("Usage: hyperqsort n, where n is number of items to be sorted\n"
);
70      exit(1);
71    }
72    sscanf(argv[1],"%d",&n);
73    if(n<1){
74      if(myID == 0)
75        printf("Usage: hyperqsort n, where n is number of items to be sorted\n"
);
76      exit(1);
77    }
78
79    // Oletus: n on jaollinen p:llä
80    nPerProc = n/p;
81
82    // --- LUKUJEN LATAUS ---
83    if(myID == 0){
84      if( (buf = (float*)malloc(n*sizeof(float))) == 0)
85        printf("Not enough memory for buf[%d]\n" ,n);
86      srand48(time(NULL));
87      for(i=0; i<n; i++){
88        buf[i] = drand48();
89      }
90    }
91    // ----------------------
92
93    // Jaetaan alkiot tasan kaikille laskentatehtäville
94    if( (myData = (float*)malloc(nPerProc*sizeof(float))) == 0 )
95      printf("Not enough memory for myData[%d]\n" ,nPerProc);
96    MPI_Scatter(buf, nPerProc, MPI_FLOAT, myData, nPerProc, MPI_FLOAT, 0,
MPI_COMM_WORLD);
97
98    // Odotetaan että kaikki laskentatehtävät pääsevät tähän asti ja
99    // aloitetaan ajan mittaaminen
100    MPI_Barrier(MPI_COMM_WORLD);
101    elapsedTime = MPI_Wtime();
102
103    // Lasketaan käytettävissä olevien ulottuvuuksien lukumäärä
104    dim = ceil(log((double)p) / (log(2.0)));
105
106    j=0;
107    for(curDim=dim; curDim>0; curDim--){
108      // Lajitellaan omat alkiot
109      qsort(myData, nPerProc, sizeof(float), compare);
110
111      // Lasketaan hyperkuutioiden lukumäärä
112      numCubes = 1 << j;
113      // Lasketaan hyperkuutioiden koko
114      sizeCubes = p / numCubes;
```

```
115     // Lasketaan oman hyperkuution numero
116     myCubeNum = myID >> curDim;
117     //Jaetaan tarvittaessa nykyinen kuutio alikuutioiksi
118     MPI_Comm_split(MPI_COMM_WORLD, myCubeNum, myID, &MPI_COMM_CUBE);
119     // Selvitetään oma järjestysnumero uudessa hyperkuutiossa
120     MPI_Comm_rank(MPI_COMM_CUBE, &myCubeID);
121
122     if(myCubeID == 0){
123       // Muodostetaan sarana-alkio
124       if(nPerProc % 2 == 0)
125         pivot = (myData[nPerProc / 2 - 1]+myData[nPerProc / 2]) / 2;
126       else
127         pivot = myData[nPerProc / 2];
128     }
129     // Lähetetään sarana-alkio kaikille
130     MPI_Bcast(&pivot, 1, MPI_INT, 0, MPI_COMM_CUBE);
131     // Lasketaan keskustelukumppanin järjestysnumero
132     partnerID = myCubeID ^ (1 << (curDim-1));
133
134     if(myCubeID < (sizeCubes / 2)){ // 1. Lähetä, 2. Vastaanota
135       // --- LÄHETETÄÄN ALKIOT, JOTKA SUUREMPIA KUIN SARANA-ALKIO ---
136       if( (smallerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
137         printf("Not enough memory for smallerThanPivot[%d]\n" ,nPerProc);
138       if( (largerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
139         printf("Not enough memory for largerThanPivot[%d]\n" ,nPerProc);
140       nSmaller = 0;
141       nLarger = 0;
142       for(i = 0; i < nPerProc; i++){
143         if (myData[i] <= pivot) {
144           smallerThanPivot[nSmaller] = myData[i];
145           nSmaller++;
146         }
147         else {
148           largerThanPivot[nLarger] = myData[i];
149           nLarger++;
150         }
151       }
152       MPI_Send(largerThanPivot, nLarger, MPI_FLOAT, partnerID, tag,
                  MPI_COMM_CUBE);
153       free(myData);
154       myData = smallerThanPivot;
155       nPerProc = nSmaller;
156       free(largerThanPivot);
157       // ------------------------------------------------------------
158       // --- VASTAANOTETAAN ALKIOT, JOTKA PIENEMPIÄ TAI YHTÄSUURIA KUIN S-
ALKIO ---
159       MPI_Probe(partnerID, tag, MPI_COMM_CUBE, &status);
160       MPI_Get_count(&status, MPI_INT, &nRecv);
161       if(nRecv > 0){
162         nPerProc += nRecv;
163         if((recvBuf = (float *) malloc(nRecv*sizeof(float))) == 0)
164           printf("Not Enough Memory for recvBuf[%d]\n" ,nRecv);
165         MPI_Recv(recvBuf, nRecv, MPI_FLOAT, partnerID, tag, MPI_COMM_CUBE,
                    &status);
166         // --- YHDISTETÄÄN LISTAT ---
167         if( (buf=(float*)malloc(nPerProc*sizeof(float))) == 0)
168           printf("Not enough memory for myData[%d]\n" ,nPerProc);
169         ownCount=0;
170         recvCount=0;
```

```
171         for(i=0; i<nPerProc; i++){
172           if(ownCount == nPerProc-nRecv){
173             buf[i] = recvBuf[recvCount];
174             recvCount++;
175           }
176           else if(recvCount == nRecv){
177             buf[i] = myData[ownCount];
178             ownCount++;
179           }
180           else if(myData[ownCount] > recvBuf[recvCount]){
181             buf[i] = recvBuf[recvCount];
182             recvCount++;
183           }
184           else{
185             buf[i] = myData[ownCount];
186             ownCount++;
187           }
188         }
189         free(recvBuf);
190         free(myData);
191         myData = buf;
192         // ---------------------------
193       }
194       // ------------------------------------------------------------------
195     }
196     else{// 1. Vastaanota, 2. Lähetä
197       // --- VASTAANOTETAAN ALKIOT, JOTKA SUUREMPIA KUIN SARANA-ALKIO ---
198       MPI_Probe(partnerID, tag, MPI_COMM_CUBE, &status);
199       MPI_Get_count(&status, MPI_INT, &nRecv);
200       if(nRecv>0){
201         nPerProc += nRecv;
202 
203         if((recvBuf = (float *) malloc(nRecv*sizeof(float))) == 0)
204           printf("Not Enough Memory for recvBuf[%d]\n" ,nRecv);
205         MPI_Recv(recvBuf, nRecv, MPI_FLOAT, partnerID, tag, MPI_COMM_CUBE,
                  &status);
206 
207         // --- YHDISTETÄÄN LISTAT ---
208         if( (buf=(float*)malloc(nPerProc*sizeof(float))) == 0 )
209           printf("Not enough memory for buf[%d]\n" ,nPerProc);
210         ownCount=0;
211         recvCount=0;
212         for(i=0; i<nPerProc; i++){
213           if(ownCount == nPerProc-nRecv){
214             buf[i] = recvBuf[recvCount];
215             recvCount++;
216           }
217           else if(recvCount == nRecv){
218             buf[i] = myData[ownCount];
219             ownCount++;
220           }
221           else if(myData[ownCount] > recvBuf[recvCount]){
222             buf[i] = recvBuf[recvCount];
223             recvCount++;
224           }
225           else{
226             buf[i] = myData[ownCount];
227             ownCount++;
```

```
228           }
229         }
230         free(recvBuf);
231         free(myData);
232         myData = buf;
233         // -------------------------
234       }
235       // ----------------------------------------------------------------
236       // --- LÄHETETÄÄN ALKIOT, JOTKA PIENEMPIÄ TAI YHTÄSUURIA KUIN S-ALKIO
237
238       if( (smallerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
239         printf("Not enough memory for smallerThanPivot[%d]\n" ,nPerProc);
240       if( (largerThanPivot = (float*)malloc(nPerProc*sizeof(float))) == 0 )
241         printf("Not enough memory for largerThanPivot[%d]\n" ,nPerProc);
242       nSmaller = 0;
243       nLarger = 0;
244       for(i = 0; i < nPerProc; i++){
245         if (myData[i] <= pivot) {
246           smallerThanPivot[nSmaller] = myData[i];
247           nSmaller++;
248           }
249         else {
250           largerThanPivot[nLarger] = myData[i];
251           nLarger++;
252         }
253       }
254       MPI_Send(smallerThanPivot, nSmaller, MPI_FLOAT, partnerID, tag,
                  MPI_COMM_CUBE);
255       free(myData);
256       myData = largerThanPivot;
257       nPerProc = nLarger;
258       free(smallerThanPivot);
259       // ----------------------------------------------------------------
260     }
261     j++;
262   }
263
264   // Lopetetaan ajan mittaaminen
265   elapsedTime = MPI_Wtime() - elapsedTime;
266
267   //-------VIRHEIDEN TARKISTUS------
268   int errors = 0;
269   for(i=0; i<nPerProc-1; i++){
270     if(myData[i] > myData[i+1])
271       errors++;
272   }
273   if(errors > 0) // Tieto virheistä vain jos niitä on
274     printf("myID: %d, number of errors: %d\n" , myID, errors);
275
276
277   if(myID==0){
278     errors = 0;
279     float *tests;
280     if( (tests = (float*)malloc((p*2)*sizeof(float))) == 0)
281       printf("Not enough memory for tests[%d]\n" ,(p*2));
282     tests[0] = myData[0];
283     tests[1] = myData[nPerProc-1];
284     for(i=1; i<p; i++){
285       MPI_Recv(tests+(i*2), 2, MPI_FLOAT, i, tag, MPI_COMM_WORLD, &status);
```

```
286     }
287     for(i=1; i<(p*2); i++){
288       if(tests[i-1] > tests[i])
289     errors++;
290     }
291     if(errors > 0) // Tieto virheistä vain jos niitä on
292       printf("Number of errors between processors: %d\n" ,errors);
293   }
294   else{
295     float tests[2];
296     tests[0] = myData[0];
297     tests[1] = myData[nPerProc-1];
298     MPI_Send(tests, 2, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
299   }
300   //--------------------------------
301
302   // Tulostetaan näytölle oma järjestysnumero, käytetty prosessoriaika ja
303   // hallussa olevien alkioiden lukumäärä
304   fprintf(stdout,"Processor Id: %d;  The CPU Time: %10.6f;  SubArray Length:
                    %d.\n",myID, elapsedTime, nPerProc);
305   fflush(stdout);
306
307   free(myData);
308   MPI_Finalize();
309 }
```

```
1 /* Parallel Sorting with Regular Sampling -algoritmi
2  * ================================================
3  *
4  * Lajiteltavat alkiot jakautuneet aluksi kaikille
5  * laskentatehtäville (ei kuitenkaan missään erityisessä
6  * järjestyksessä). Lopuksi alkiot jaettuna kaikille
7  * laskentatehtäville niin, että molemmat järjestetyn
8  * rinnakkaisen aineiston ehdot ovat voimassa.
9  *
10 * Copyright (c) Juha Katajisto
11 */
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 #include <time.h>
16 #include <math.h>
17 #include "mpi.h"
18
19 // qsort-funktion tarvitsema alkioiden vertailufunktio
20 int compare (const void * a, const void * b)
21 {
22   if(*(float*)a != *(float*)b){
23     if(*(float*)a < *(float*)b)
24       return -1;
25     else
26       return 1;
27   }
28   else
29     return 0;
30 }
31
32 // funktio keskiarvon laskemiseen
33 float keskiarvo(float *data, int n){
34   float sum=0;
35   int i;
36   for(i=0; i<n; i++)
37     sum += data[i];
38   return (sum / n);
39 }
40
41 int main(int argc, char *argv[]){
42   MPI_Status status;    // MPI-funktioiden tarvitsema status-muuttuja
43       double elapsedTime;// Ajanmittaukseen käytettävä muuttuja
44          int i,         // Apumuuttuja silmukoihin
45              j,         // Apumuuttuja silmukoihin
46              tag=0,     // MPI-viestintäfunktioissa käytettävä tunniste
47              p,         // Laskentatehtävien lukumäärä
48              myID,      // Oma järjestysnumero
49              partnerID, // Keskustelukumppanin järjestysnumero
50              n,         // Data-alkioiden kokonaismäärä
51              nPerProc=0,// Yhden laskentatehtävän hallussa olevien
                           alkioiden lkm
52              check = 0, // Apumuuttuja
53              sampleSize,// Näytteessä olevien alkioiden lkm
54              nRecvSum,  // Apumuuttuja vastaanotettujen alkioiden laskemiseen
55              *nRecv,    // Taulukko vastaanotettujen alkioiden määristä
56             *sendCounts,// Lähetettävien alkioiden lukumäärät
57              *dataStart;// Indeksit, joista lähetettävät alkiot alkavat
```

```
58        float *myData,   // Laskentatehtävän hallussa oleva data
59              *buf,      // Tilapäinen puskuri alkioille
60              **recvData,// 2-ulotteinen taulukko vastaanottopuskureille
61              *pivots,   // Taulukko sarana-alkioille
62              *samples,  // Omat näytealkiot
63         *samplesFromAll;// Kerätyt näytealkiot
64
65   MPI_Init(&argc, &argv);
66   MPI_Comm_size(MPI_COMM_WORLD,&p);
67   MPI_Comm_rank(MPI_COMM_WORLD, &myID);
68
69   // Asetetaan näytealkioiden määrä per laskentatehtävä
70   sampleSize = p;
71
72   n=0;
73   if(argc != 2){
74     if(myID == 0)
75       printf("Usage: psrs n, where n is number of items to be sorted\n" );
76     exit(1);
77   }
78   sscanf(argv[1],"%d",&n);
79   if(n<1){
80     if(myID == 0)
81       printf("Usage: psrs n, where n is number of items to be sorted\n" );
82     exit(1);
83   }
84
85   // Oletus: n on jaollinen p:llä
86   nPerProc = n/p;
87
88   // --- LUKUJEN LATAUS ---
89   if(myID == 0){
90     if( (buf = (float*)malloc(n*sizeof(float))) == 0)
91       printf("Not enough memory for myData[%d]\n" ,n);
92     srand48(time(NULL));
93     for(i=0; i<n; i++){
94       buf[i] = drand48();
95     }
96   }
97   // ----------------------
98
99   // Jaetaan luvut tasan kaikille laskentatehtäville
100   if( (myData = (float*)malloc(nPerProc*sizeof(float))) == 0 )
101     printf("Not enough memory for myData[%d]\n" ,nPerProc);
102   MPI_Scatter(buf, nPerProc, MPI_FLOAT, myData, nPerProc, MPI_FLOAT, 0,
                 MPI_COMM_WORLD);
103
104   // Odotetaan että kaikki laskentatehtävät pääsevät tähän asti ja
105   // aloitetaan ajan mittaaminen
106   MPI_Barrier(MPI_COMM_WORLD);
107   elapsedTime = MPI_Wtime();
108   //----------------
109
110   // Lajitellaan alkiot jos niitä on enemmän kuin yksi
111   if(nPerProc > 1)
112     qsort(myData, nPerProc, sizeof(float), compare);
113
114   // --- OTETAAN NÄYTTEET ---
115   if( (samples = (float*)malloc(sampleSize*sizeof(float))) == 0)
```

```
116     printf("Not enough memory for myData[%d]\n" ,n);
117   for(i=0; i<sampleSize; i++){
118     samples[i] = myData[((i+1)*(nPerProc/sampleSize))-1-((nPerProc/sampleSize)/2)];
119   }
120   // ------------------------
121   // --- KERÄTÄÄN NÄYTTEET KAIKILTA ---
122   if(myID==0){
123     if( (samplesFromAll = (float*)malloc((p*sampleSize)*sizeof(float))) == 0 )
124       printf("Not enough memory for myData[%d]\n" ,n);
125   }
126   MPI_Gather(samples, sampleSize, MPI_FLOAT, samplesFromAll, sampleSize,
                MPI_FLOAT, 0, MPI_COMM_WORLD);
127   free(samples);
128   // --------------------------------
129
130   // --- MUODOSTETAAN SARANA-ALKIOT ---
131   if( (pivots = (float*)malloc((p-1)*sizeof(float))) == 0 )
132     printf("Not enough memory for pivots[%d]\n" ,(p-1));
133   if(myID == 0){
134     // Lajitellaan näytteet
135     qsort(samplesFromAll, (p*sampleSize), sizeof(float), compare);
136     // Valitaan p-1 käytettävää sarana-alkiota
137     for(i=0; i<(p-1); i++){
138       pivots[i] = keskiarvo(samplesFromAll+(i*sampleSize), sampleSize*2);
139     }
140     free(samplesFromAll);
141   }
142   // --------------------------------
143
144   // Lähetetään käytettävät sarana-alkiot kaikille
145   MPI_Bcast(pivots, (p-1), MPI_FLOAT, 0, MPI_COMM_WORLD);
146
147   // --- ETSITÄÄN OMISTA ALKIOISTA LÄHETETTÄVÄT ALKIOT ---
148   if((sendCounts = (int*)calloc(p,sizeof(int)))==0)
149     printf("Not enough memory for sendCounts[%d]\n" ,p);
150   if((dataStart = (int*)calloc(p,sizeof(int)))==0)
151     printf("Not enough memory for dataStart[%d]\n" ,p);
152
153   j=1;
154   for(i=0; i<nPerProc; i++){
155     if(myData[i] < pivots[j-1]){
156       dataStart[j] = i+1;
157     }
158     else{
159       j++;
160       if(j==p)
161     break;
162       i--;
163     }
164   }
165   for(i=j; i<p; i++){
166     dataStart[i] = nPerProc;
167   }
168   for(i=1; i<p; i++){
169     if(dataStart[i] == 0){
170       dataStart[i] = dataStart[i-1];
171     }
172   }
173   sendCounts[0] = dataStart[1];
```

```
174   check = 0;
175   for(i=1; i<(p-1); i++){
176     if(dataStart[i] == nPerProc -1){
177       sendCounts[i] = 1;
178       check = 1;
179       break;
180     }
181     sendCounts[i] = dataStart[i+1] - dataStart[i];
182   }
183   if(check == 0)
184     sendCounts[p-1] = nPerProc - dataStart[p-1];
185
186   for(i=0; i<p; i++){
187     if(dataStart[i] > (nPerProc-1))
188       dataStart[i] = nPerProc-1;
189   }
190   // ------------------------------------------------------
191
192   free(pivots);
193
194   // --- SIIRRETÄÄN ALKIOT OIKEILLE LASKENTATEHTÄVILLE ---
195   if( (recvData = (float**)malloc(p*sizeof(float*))) == 0 )
196     printf("Not enough memory for **recvData[%d]\n" ,p);
197   for(i=0; i<p; i++){
198     if((recvData[i] = (float*)malloc(nPerProc*sizeof(float))) == 0)
199       printf("Not enough memory for recvData[%d]\n" ,i);
200   }
201   if( (nRecv = (int*)malloc(p*sizeof(int))) == 0 )
202     printf("Not enough memory for nRecv[%d]\n" ,p);
203
204   for(i=0; i<p; i++){
205     MPI_Scatter(sendCounts, 1, MPI_INT, &nRecv[i], 1, MPI_INT, i,
                  MPI_COMM_WORLD);
206     MPI_Scatterv(myData, sendCounts, dataStart, MPI_FLOAT, recvData[i],
                    nRecv[i], MPI_FLOAT, i, MPI_COMM_WORLD);
207   }
208   // ------------------------------------------------------
209
210   // --- KÄSITELLÄÄN VASTAANOTETTU DATA ---
211   nRecvSum = 0;
212
213   for(i = dataStart[myID]; i < (dataStart[myID] + sendCounts[myID]); i++){
214     recvData[myID][i - dataStart[myID]] = myData[i];
215   }
216   nRecv[myID] = sendCounts[myID];
217   free(myData);
218   for(i=0; i<p; i++){
219     nRecvSum += nRecv[i];
220   }
221   if( (myData = (float*)malloc(nRecvSum*sizeof(float))) == 0 )
222     printf("Not enough memory for myData[%d]\n" ,n);
223
224   nPerProc=0;
225
226   for(i=0; i<p; i++){
227     for(j=0; j<nRecv[i]; j++){
228       myData[nPerProc+j] = recvData[i][j];
229     }
230     nPerProc += nRecv[i];
```

```
231   }
232   // ------------------------------------
233
234   // Lajitellaan alkiot jos niitä on enemmän kuin yksi
235   if(nPerProc > 1)
236     qsort(myData, nPerProc, sizeof(float), compare);
237
238   // Lopetetaan ajan mittaaminen
239   elapsedTime = MPI_Wtime() - elapsedTime;
240
241   //-------VIRHEIDEN TARKISTUS------
242   int errors = 0;
243   for(i=0; i<nPerProc-1; i++){
244     if(myData[i] > myData[i+1])
245       errors++;
246   }
247   if(errors > 0) // Tieto virheistä vain jos niitä on
248     printf("myID: %d, number of errors: %d\n" , myID, errors);
249
250
251   if(myID==0){
252     errors = 0;
253     float *tests;
254     if( (tests = (float*)malloc((p*2)*sizeof(float))) == 0)
255       printf("Not enough memory for tests[%d]\n" ,(p*2));
256     tests[0] = myData[0];
257     tests[1] = myData[nPerProc-1];
258     for(i=1; i<p; i++){
259       MPI_Recv(tests+(i*2), 2, MPI_FLOAT, i, tag, MPI_COMM_WORLD, &status);
260     }
261     for(i=1; i<(p*2); i++){
262       if(tests[i-1] > tests[i])
263     errors++;
264     }
265     if(errors > 0) // Tieto virheistä vain jos niitä on
266       printf("Number of errors between processors: %d\n" ,errors);
267   }
268   else{
269     float tests[2];
270     tests[0] = myData[0];
271     tests[1] = myData[nPerProc-1];
272     MPI_Send(tests, 2, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
273   }
274   //---------------------------------
275
276   // Tulostetaan näytölle oma järjestysnumero, käytetty prosessoriaika ja
277   // hallussa olevien alkioiden lukumäärä
278   fprintf(stdout,"Processor Id: %d;  The CPU Time: %10.6f;  SubArray Length:
                  %d.\n", myID, elapsedTime, nPerProc);
279
280   for(i=0; i<p; i++){
281     free(recvData[i]);
282   }
283   free(recvData);
284   free(myData);
285
286   MPI_Finalize();
287 }
```