

TARTU ÜLIKOOL
Arvutiteaduse instituut

Eero Vainikko

Fortran95 ja MPI

TARTU 2004

TARTU ÜLIKOOL
Arvutiteaduse instituut

Eero Vainikko

Fortran95 ja MPI

Tartu 2004

Selle õppevahendi koostamist toetas Eesti Infotehnoloogia Sihtasutus Tiigriülikooli projekti raames.

Toimetaja: Jaanus Pöial

© Eero Vainikko 2004

ISBN 9985-56-849-4

Tartu Ülikooli Kirjastus

www.tyk.ut.ee

Tellimus nr. 142

Sisukord

Sissejuhatus	5
I Programmeerimine keeles Fortran95	7
1 Fortrani põhjooned	9
1.1 Programmeerimiskeele FORTRAN ajalugu	11
1.2 Fortran77 puudusi	12
1.3 Mida uut on Fortran9x keeles võrreldes Fortran77-ga?	14
2 Fortran9x ja objekt-orienteeritus	17
2.1 Andmetüübid	18
2.1.1 Põhitüübid	18
2.1.2 Konstantide defineerimine; esimene näiteprogramm	19
2.1.3 Kasutaja poolt defineeritavad e. tuletatud tüübid	20
2.1.4 Abstraktsed andmetüübid ja klassid	21
2.1.5 Polümorfism OO programmeerimise kontseptsioonis	25
2.1.6 Liidesedirektiiv	31
3 Keele Fortran9x elemendid	35
3.1 Kommentaaride lisamine lähteteksti ridadele	35
3.2 Muutujate põhitüübid	35
3.3 Aritmeetilised operaatorid	36
3.4 Võrdlusoperaatorid	37
3.5 Stringitöötlus	38
3.6 Fortran95 viidad (<i>pointers</i>)	40
4 Massiivid ja operatsioonid nendega	43
4.1 Fortran90 massiivide omadused	43
4.2 Elemantaaroperatsioonid massiividega	44
4.3 Massiivide sektorid ja konstruktorid	44
4.4 Reserveeritava mälu massiivid	46
4.5 Automaatsed massiivid	47
4.6 Fiktiivsete argumentide deklareerimine	47
4.7 Standardfunktsioone massiividega	48

4.7.1	Päringufunktsioonid	48
4.7.2	Kujumuutusoperatsioonid	52
4.7.3	Vektorite ja maatriksite korrutamine	52
4.7.4	Kitsendusfunktsioonid	52
4.7.5	Asukohafunktsioonid	53
4.7.6	Massiivi muutmise funktsioonid	53
4.7.7	WHERE - direktiiv	54
4.8	Sisend-väljund	54
4.8.1	Formaadikirjeldused	55
4.8.2	Failitöötlus	56
II	MPI (<i>The Message Passing Interface</i>)	59
5	Kontseptsioon	61
6	MPI kuus põhikäsku	65
6.1	MPI konstruktor ja destruktor	66
6.2	Põhipäringud	67
6.3	Teadete saatmine ja vastuvõtmine	68
6.4	MPI-programmide kompileerimine ja käivitamine	69
7	Veel MPI käske	71
7.1	MPI käskude liigitus	71
7.2	Peamisi ühisoperatsioone	71
7.3	Näide: Iseorganiseeruv kommunikatsioonimudel	73
8	Mitteblokeeriv kommunikatsioon	77
8.1	Mitteblokeerivaid MPI käske	77
8.2	Ühesuunaline kommunikatsioon kahe protsessori vahel	78
8.3	Vastastikune kommunikatsioon ja tupikute vältimine	79
9	Näide: Hõredate maatriksite klass	83
9.1	Hõredate maatriksite kolmikformaad	83
9.2	Paralleliseerimine	89
9.3	Kaasgradientide meetodi testimine	99
	Kokkuvõte	104
	Kirjandus	105
	Indeks	106

Sissejuhatus

Käesolev õppematerjal on mõeldud abivahendiks programmeerijale, kes vajab kõrgetasemelisi keelilisi vahendeid suuremahuliste ülesannete lahendamiseks, kus üheks oluliseks näitajaks on programmi tööaeg. Materjalid on kokku pandud Tartu Ülikooli Arvutiteaduse Instituudis autori poolt 2003. a. sügissemestril loetud erikursuse “Teadusarvutused” käigus, kuid on mõeldud laiemaks kasutamiseks nii teistel õppekursustel (nagu näiteks TÜ ATI erikursus “Paralleelarvutused”) kui ka sõltumatuks kasutamiseks kõigile huvitatutele.

Me eeldame, et lugeja on tuttav programmeerimisega vähemalt ühes kõrgetasemelises keeles (nagu näiteks Java, C, C++, Pascal, MATLAB jne.) Materjali esitus baseerub suuresti näidetel ning süntaksivõrdlusel eri keelte vahel. Soovitav oleks et lugeja kasutaks käsikäes õppematerjaliga arvutit, kus on olemas Fortran90/95 kompilaator¹ ning installeeritud teatedastustek MPI².

Kõik lähtetekstidena toodud programminäited on kättesaadaval internetiaadressilt <http://www.ut.ee/~eero/F95jaMPI/Kood/>. Näiteprogrammide koostamisel olid inspiratsiooniks mitmed nii autori enda kui ka õpilaste, kolleegide kui ka kirjanduses toodud näited. Kõik teiste poolt kirjutatud näiteprogrammid on autori poolt vähemal või rohkemal määral ümber kirjutatud ning modifitseeritud ja seetõttu oleks liiga keerukas viidata iga programmi kohta eraldi algtekstide komponentallikaid. Näidete valikul sai proovitud, kus vähegi võimalik, koostada täielikult töötavad programmid, mida lugeja saaks proovida ise kompileerida, käivitada, muuta, täiustada lähtetekste, eksperimenteerida. Julgustame siin seda aktiivselt tegema, kuna parim õppimisviis käib omarada läbi isikliku kogemuse!

Õppematerjali esimene osa on pühendatud programmeerimiskeelele Fortran.

Esimeses peatükis kirjeldame programmeerimiskeele valikukriteeriume, kirjeldame Fortrani eeliseid teiste levinud keelte seas ning anname lühikese ülevaate keele ajaloost. Ka kirjeldame lühidalt, mida uut tõi Fortran90/95 võrreldes eekäijatega.

Teises peatükis alustame keele detailsemat kirjeldamist, lähtudes eesmärgist kirjutada programme objekt-orienteeritud lähenemisviisil. Räägime andmete põhitüüpidest, kasutaja poolt defineeritavatest tüüpidest, abstraktsetest tüüpidest, klasside defineerimisest ning polümorfismist keeles Fortran90/95.

Kolmas peatükk tutvustab keele elemente. Toome mõningaid võrdlevaid tabelleid eri keelte sarnaste konstruktsioonide süntaksist. Räägime muuhulgas ka sõnetööstlusest Fortranis ja viidakäsitluse eripärast.

Neljas peatükk on pühendatud Fortran90/95 massiivtööstlusele. Kirjeldame massiivi-

¹Õppematerjali kirjutamise ajal on näiteks Linux-tööjaamadele individuaalseks kasutamiseks tasuta litsentsiga saadaval Intel Fortrani kompilaator, vt. lähemalt <http://www.intel.com>.

²Vt. näiteks <http://www-unix.mcs.anl.gov/mpi/mpich/> või <http://www.lam-mpi.org/>

süntaksit, massiivobjekte ning võimalikke operatsioone nendega, mõningaid käepäraseid eeldefineeritud funktsioone massiividega. Peatüki lõpetame Fortrani sisendi- ja väljundioperatsioonide kirjeldusega.

Antud õppematerjali teine osa on pühendatud teatedastusmeetodi standardile MPI (*the Message Passing Interface*).

Viies peatükk kirjeldab lühidalt teatedastusmeetodite üldist kontseptsiooni ja metoodikat nii MPI kui ka teiste teatedastusteede puhul.

Kuuendas peatükis kirjeldame MPI kuut põhikäsku. Seitsmendas peatükis täiendame toodud käskudepagasit veel mõningate vajalike käskudega ühiskommunikatsiooni ja kahe protsessi vahelise suhtluse teostamiseks. Toome näite, kuidas kirjutada iseorganiseeruva kommunikatsiooniga programme.

Kaheksanda peatüki eesmärgiks on kirjeldada mitteblokeerivate kommunikatsioonikäskude kasutamist ning võimalusi tupikute vältimiseks paralleelprogrammides.

Viimases, üheksandas peatükis toome ühe reaalse näite paralleelprogrammeerimisest, kus demonstreerime Fortrani ja MPI kasutamist hõredate maatriksitega lineaarvõrrandite süsteemide lahendamisel kaasgradientide meetodil.

Mainime veel, et toodud materjal ei kata sugugi kogu Fortran95 ja MPI temaatikat ning seetõttu tuleks antud õppevahendisse suhtuda kui sissejuhatavasse materjali ning vajaduse korral pöörduda muu kirjanduse poole. Küll aga loodame, et käesoleva töö ilmumine eestikeelsena lihtsustab tunduvalt antud temaatika omandamist.

Autor on tänulik Merik Meristele antud õppematerjali käsikirjaga tutvumise, mitmete vigade paranduste ning kasulike soovitude eest, mis aitasid materjaliesitust täiustada.

Eero.Vainikko@ut.ee

Osa I

Programmeerimine keeles Fortran95

Peatükk 1

Fortrani põhijooned

Oletame, et meil on vaja sooritada suuremahulisi arvutusi näiteks teaduslikel eesmärkidel, modelleerimisel vms. Selleks tuleb meil leida sobiv riist- ja tarkvarast koosnev töökeskkond. Vaja on leida antud ülesande lahendamiseks sobiv riistvara piisavalt kiire protsessori-ga, piisava mäluhulgaga jne. Ilmne küsimus on aga ka: millist programmeerimiskeelt ja muud abitarvara kasutada? Keelevalikul suurte arvutusmahukate ülesannete lahendamiseks tuleks otsuse tegemisel esitada järgnevaid, omavahel paljuski seotud küsimusi:

A. Kui hästi üks või teine programmeerimiskeel saab hakkama ujukoma arvutustega?

Ujukomaoperatsioonide kiirus sõltub eelkõige ka arvuti protsessori arhitektuurist (näiteks, kui palju on konkreetset protsessoril ujukoma-registreid ja konveiereid ujukomaarvudega operatsioonide teostamiseks), kuid paraku ka keele enda omapäradest. Näiteks tihti läheb (insener)arvutustes tarvis kompleksarvulisi muutujaid. On hea kui kompleksarvu-tüüp on keeleliselt toetatud, st. et tüüp kompleksarv on keeles olemas.

Nagu me näeme alapunktis 1.1, loodigi keel Fortran algselt silmas pidades vajadust efektiivselt ning lihtsalt sooritada suurel hulgal operatsioone ujukomaarvudega. Eri-nevalt näiteks C-keelte perekonnast, (keel C loodi tegelikult eelkõige süsteemprogrammeerimiseks), on kompleksarvu-tüüp Fortanis keele elemendiks.

B. Kuidas on realiseeritud massiivioperatsioonid?

Suurte andmehulkade töötlemisel organiseerime me andmeid enamasti massiividesse. Hea on, kui massiivid on keelde "sisse ehitatud" konstruktsioonid ehk andmeobjektid. See tagab nii programmitekstide lühiduse kui ka algoritmide realisatsiooni kompaktsuse. Lisaks on sellel suur mõju kompilaatori optimeerimisvõimele.

Fortran95-s on massiivid keele üheks lahutamatuks osaks. Võib öelda et massiivid on keelde sisseehitatud andmeobjektid. Programmeerimist lihtsustab massiivisüntaks keelelisel tasemel, see võimaldab üheselt määrata ära näiteks maatriksite ja vektorite vahelisi operatsioone ning kompilaator ise valib optimaalse realisatsiooni antud olukorrast lähtudes nende realiseerimiseks. Kuigi on võimalik ka näiteks C++ keelt täiendada massiivioperatsioonidega, ei saa me öelda, et see oleks antud keele algsosa. Java-keeles on ujukomaarvude massiivid aga juba keele standardis defineeritud paraku kuiul mis teeb raskeks optimeerimise

C. Millisel tasemel on kompilaatori optimeerimisvõime?

Kuigi on olemas teatud reeglid, mida optimaalsel programmeerimisel arvestada, jääb suur osa programmikoodi optimeerimistööst kompilaatori kanda. Ilmne reegel on: mida keerulisem keel, seda raskem on kompilaatoril teha õigeid optimeerimisotsuseid. Näiteks, keeles Fortran77 kirjutatud programmi on tunduvalt lihtsam optimeerida kui keeles C++, põhjuseks Fortran77 programmide staatiline mäluhaldus.

Fortran95 täiendab Fortran77 standardit moodsate keeleliste vahenditega, arvestades seejuures, et optimeeritavus säiliks niipalju kui võimalik. Objekt-orienteeritud kontseptsioonist rakendatakse vaid teatud kitsam osa, mis ei kahjusta programmi koodi optimeeritavust arvutuskiiruse mõttes.

D. Kas ja kuidas on realiseeritud objekt-orienteeritus?

Kuigi objekt-orienteeritud keelte eeliste detailsem väljatoomine ei mahu antud kirjutise raamidesse, mainime vaid, et mida keerulisemaks muutub kirjutatav programm, seda suurem on vajadus objekt-orienteeritud lähenemise võimaluste järele. Samas jällegi, mida keerulisemad ja võimalusterohkemad on objektid vaadeldavas keeles, seda raskem on kompilaatoril tagada optimaalsust.

Keeles Fortran95 on objekt-orienteeritus realiseeritud moodulite näol. Üks moodul võib sisaldada endas ühte või tervet hulka andmestruktuure, millel on defineeritud teatud operatsioonid. Samas võib öelda, et ka massiivid on keeles justkui omaette sisseehitatud klassid koos nendel defineeritud operatsioonidega.

E. Kuivõrd kõrgetasemeline on antud programmeerimiskeel?

Selge see, et C++ on kõrgetasemelisem kui C ning annab palju uusi võimalusi. Sarnaselt on Fortran95 kõrgemasemelisem keel kui Fortran77 ning muudab märksa lihtsamaks näiteks mäluhalduse ning massiividega opereerimise.

Kuigi ka Fortran95 keeles leidub igandeid, mis võimaldavad kirjutada halbu programme, on selles keeles kirjutatud arvutusliku iseloomuga programmid märksa lihtsamad ning kergemini loetavamad kui mõnes muus programmeerimiskeeles. Seda just tänu hästiarendatud massiivisüntaksi ja -operatsioonide toele.

Fortran95 kasuks räägib muuhulgas ka võimalus teha nn. massiivide indeksikontrolli, mis muudab programmide silumise ning muidu raskestiavastatavate vigade leidmise kergemaks. Tavaliselt on vaja programmi kompileerimisel anda kompilaatorile lisaparameeter `-C` mille tulemusena kompilaator genereerib ca 10 korda aeglasema programmi, kuid käivitamisel teostatakse reaalselt massiiviindeksi piiride kontrolli (ja tihti ka näiteks mäluhalduse vigade otsingut) ning väljastatakse vastav veateade probleemide avastamisel.

Lisaks soovitaks lugeda artiklit aadressil <http://www.lahey.com/PRENTICE.HTM>, mis annab kujukaid fakte, mis demonstreerivad keeleveliku olulisust üht või teist sorti ülesande korral.

1.1 Programmeerimiskeele FORTRAN ajalugu

Programmeerimiskeele FORTRAN nimi on pärit IBM-lt, keele kompilaator *Mathematical FORmula TRANslation System* loodi 1950-ndate aastate lõpus. Antud keel oli mõeldud eelkõige matemaatiliste avaldiste ja arvutuste lihtsamaks programmeerimiseks sarnanemaks rohkem tegelikele matemaatilistele tekstidele. Eesmärgiks oli luua keel, mida oleks lihtne õppida, kuid mis oleks oma optimaalsuselt siiski suuteline võistlema assembler-keelega. Enne seda kasutati programmeerimiseks masinkeelele sarnaseid programmikoode, programmeerija pidi muuhulgas hästi tundma konkreetse arvuti arhitektuuri, registreite arvu, masinkäskusid jne. Oma eesmärgi see üks esimesi kõrgetasemekeeli saavutas, võimaldades kiiremini programmeerida vaid väikese arvutuste efektiivsuskaoga: Fortran muutus kiiresti populaarseks. Aastaks 1963 oli loodud juba 40 erinevat kompilaatorit. Sisuliselt tekkis hulganisti erinevaid keele dialekte, mis tõi kaasa vajaduse keel standardiseerida.

Fortani versioonide ajalugu illustreerib kokkuvõtvalt järgmine graaf:

Fortran66 \mapsto **Fortran77** \mapsto (**Fortran8x**) \mapsto **Fortran90** \mapsto **Fortran95** \longrightarrow (**Fortran200x**).

Kirjeldame seda arengut järgnevas veidi lähemalt.

- Aastal 1966, peale 4 aastast tööd, valmis kõige esimene programmeerimiskeele standard üldse – Fortran66. Standard sisaldas kõigi dialektide ühisosa. Seega, selleks et kirjutada programme, mis töötaksid kõigil arvutitel, oli mõttekas järgida standardit. Standard pani aluse Fortrani järgnevale tuntusele – arvutitootjad varustasid oma tooted reeglina ka Fortrani kompilaatoriga. Samas jätkus keele täiustamine, kusjuures iga tootja tegi oma laiendusi, mis väljusid standardi raamest. Programmide lähtetekstide ühelt arvutilt teisele kohandamine muutus jälle keerukaks, kuna eri tootjad kasutasid erinevaid täiendusi, et kasvavate vajadustega kaasas käia. Ühilduvuseks võeti kasutusele nn. eeltöötuskäskud (sarnast tehnikat võime tihti kohata näiteks C koodides), mis halvendas aga programmide loetavust. See kõik ning lisaks ka paljude vahendite puudumine keeles lõi vajaduse täiustada standardit.
- Aastal 1978 loodi Fortran77 standard (USAs, 1980 ISO standardina). Fortran77, sisuliselt järjekordne dialekt, on tänapäevastes normides endiselt vanamoodne ning väheste võimalustega keel. Näiteks puudub selles rekursioon, dünaamiline mäluhaldus jms., peatume nendel lähemalt järgmises osas. Tänu keele suurele populaarsusele programmeerijate seas neil aastail ning lihtsusele, mida kujutas Fortran66 keeles kirjutatud programmide tõlkimine Fortran77 keelde, leidub tohutul hulgal endiselt kasutuses olevat tarkvara Fortran77-s.
- 1980 algul algas uue standardi loomine eelkõige põhjusel, et tekkinud olid uued keeled uute võimalustega ning paljud uued rakendused kirjutati juba muudes keeltes. Teadusarvutusteks, tehnilisteks ning numbrilisteks arvutusteks on aga Fortran alati parem olnud ning vaja oli keel kaasajastada. Uut standardit nimetati luues Fortran8x-ks kuid valmides sai sellest Fortran90. See on moodne Fortran77 täiendus, mis lisab mitmeid uusi võimalusi, samas on säilinud ühilduvus f77 standardiga ja keeles on

endiselt igandeid ehk vanamoodsaid keelekonstruktsioone, mis võimaldavad “kehvasti programmeerida”. Ühilduvust oli vaja eelkõige selleks, et kergendada üleminekut uuele standardile ning et oleks võimalik kasutada tohutut hulka insenertehnilist ning teadusarvutuslikku tarkvarakogumit, mis selleks ajaks oli jõutud kirjutada.

- Järgnes Fortran95, hüpe ei ole siiski enam nii suur kui eelmise standardi loomisel. Hetkel töötatakse Fortran200x standardi kallal.

Ka keelel Fortran90/95 (ehk Fortran9x) on mitmeid täiendusi/modifikatsioone. Näiteks keel F, mille nime on inspireerinud keele C nime lühidus. Ka on keel ise Fortran90 lühendatud variant visates välja kõik igandid, mis keeles endiselt olemas selleks, et saaks kasutada Fortran77 programme. Iseenesest väga hea samm, kuid paraku tundub, et F ei ole võitnud piisavalt populaarsust. Keel F on ühilduv Fortran90/95-ga, aga mitte vastupidi. Ka on olemas F tasuta LINUXi versioon.

Teiseks tuntuimaks modifikatsiooniks on HPF – High Performance Fortran, mõeldud paralleelprogrammeerimiseks. Sisuliselt täiendab see Fortran9x keelt spetsiaalsete makrodega, mis programmitekstis esituvad eriliste kommentaaridena kompilaatorile andmete paralleelse esituse ning töötluse kohta.

1.2 Fortran77 puudusi

Loetleme siin mõningaid standardse Fortran77 puudusi. Kuna Fortran77 lähtetekste on siiski võimalik kompileerida ka Fortran90 kompilaatoriga, siis on hea neid teada.

1. Lähteteksti fikseeritud formaat (*fixed source format*)

Üks kõige ebamugavamaid omadusi Fortran77-s. Päritud Fortran66-st, st. ajast kui programmeerimise osaks oli veel lähtetekstide käsitsi spetsiaalsesse vormi kirjutamine, mis pidi lihtsustama perforaatori tööd, kes iga käsurea kohta spetsiaalse masinaga perfokaardi mulgustas. Fikseeritud nõuded aga ise on järgmised:

- (a) 5 esimest positsiooni real on reserveeritud reanumbritele, ühes reas tohib olla kuni 72 sümbolit.
- (b) 6-s positsioon on rea jätkusümboli koht. Kui käsurida või avaldis on nii pikk, et ületab 72 sümboli piiri, saab rida jätkata, pannes 6-ndale positsioonile tühikust erineva sümboli. Tavaliselt peab seal olema aga tühik.
- (c) Kommentaare saab lisada vaid eraldi reana, pannes esimeseks sümboliks **C** või **\$**. Seega, standardi järgi kommentaari rea lõppu lisada ei ole võimalik. Kommentaari saab siiski alustada suvalisest kohast real kasutades sümbolit “!”, seda lubab enamus kompilaatoreid.
- (d) Standardis kasutatakse vaid suurtähti. Tegelikult enamus kompilaatoreid lubab kasutada ka väikeseid tähti, keel on tõstetundetu, st. et näiteks kirjed “**END**”, “**End**” ja “**end**” on samatähenduslikud.

- (e) Identifikaatori maksimaalseks pikkuseks on 6 sümbolit. Tänapäevaste keeltega võrreldes on see muidugi väga ebamugav piirang, mis sunnib programmeerijat kulutama täiendavat aega sobivate lühendite väljamõtlemisele, muudab programmitextide loetavuse halvemaks ning on vigade allikaks. Enamus hetkel kasutatavaid Fortran77 kompilaatoreid on sellest nõudest loobunud hoolimata standardist.
2. **Sisseehitatud paralleelsuse puudumine.** Näiteks massiivitöötusel tuleb Fortran77 korral iga massiivi elementi eraldi käsitleda, mis ei anna kompilaatorile piisavalt vabadust, juhul, kui on tegu näiteks mitmeprotsessorilise masinaga, et seda tööd protsessorite vahel jagada.
 3. **Dünaamilise mäluhalduse puudumine.** Kogu kasutatav mälu tuleb standardi järgi ära määrata kompileerimise ajal. See on ühest küljest väga hea kompilaatorile endale – nii saab see teostada agressiivsemat optimeerimisstrateegiat. Samas on see aga ka väga piirav programmeerijale kuna tihti ei ole ette teada, kui suurt osa mälu ühel või teisel hetkel vaja läheb erinevate massiivide tarbeks. (Üsna tavaline praktika oli näiteks kirjutada omaenda **ALLOCATE**, ja **DEALLOCATE**-tüüpi käsud kus eraldati eri massiividele ühe suure algselt selleks etteantud massiivi osi vastavalt vajadusele programmi töö käigus. Vahel kutsuti aga isegi välja näiteks hoopis C-keele mäluhalduse operatsioone kui miski muu ei aidanud!) Mõnedel kompilaatoritel (näiteks SUNi f77) on aga mäluhalduse käsud standardiväliselt ka olemas.
 4. **Kehv ühilduvus eri arhitektuuridel.** Seda just tänu erinevatele täiendustele eri tootjate poolt. See tekitab järjekordselt vajaduse uue standardi järele.
 5. **Tuletatavate andmetüüpide (*ik. user-defined data types*) puudumine, rääkimata objekt-orienteerituse kontseptsiooni olemasolust.** Juhul, kui on tegemist suurema programmi või projektiga, mille kallal töötab mitu inimest või inimgruppi, siis on hästidefineeritud andmestruktuuride olemasolu ning objekt-orienteeritud lähenemisviis projekti edukuse üks eeltingimusi. Ilma kindla struktuurita programmi puhul tekib teatud piirist olukord, kus üht viga parandades tekib kümme viga juurde kuskil mujal programmmis.
 6. **Rekursiooni puudumine.** Paljud algoritmid kasutavad rekursiooni. Optimeerimine läheb küll kompilaatoril rekursiooni puhul raskemaks, kuid rekursiooni võlu algoritmides on kirjutatud programmi lihtsuses ja lühiduses.
 7. **Kõrvalefektide tekkimine ühisväljade jms. mäluoperatsioonide (nagu **COMMON**, **EQUIVALENCE**) kasutamisel.** Selliste keelekonstruktsioonide olemasolu muudab programmi muuhulgas raskesti loetavaks – ei ole võimalik nii lihtsalt aru saada, kus ja kas üks või teine teatud mäluaadressil paiknev muutuja oma väärtuse saab. See omakorda suurendab raskestiavastatavate vigade tõenäosust programmmis.

Nagu nägime, on palju põhjusi Fortrani standardi kaasajastamiseks ning see standard on uuendamisel ka praegu. Vaatleme nüüd lähemalt uuendusi, mis Fortran90/95 töid

1.3 Mida uut on Fortran9x keeles võrreldes Fortran77-ga?

1. **Kasutusel on uus, vähemate piirangutega lähteteksti formaat.** Lühidalt võiks lähteteksti formaati kirjeldada järgmiselt:
 - (a) Tegemist on nn. **vaba lähteteksti formaadiga** (*free source format*), ükski positsioon ei ole eritähenduslik. Avaldise või käsu jätkamiseks uuel real kasutatakse sümbolit “&” jätkatava rea lõpus;
 - (b) lubatud on kuni 132 sümbolit ühes reas;
 - (c) lubatud on rohkem kui üks käsk samal real, käsud tuleb eraldada sel juhul semi-kooloniga;
 - (d) on lubatud kirjutada kommentaare programmi lähtetekstiga samale reale, kommentaari algussümboliks on hüüumärk “!”;
 - (e) lubatud on nii suured- kui ka väikesed tähed (keel tõstetundetu);
 - (f) identifikaatorid võivad olla kuni 31 sümbolit pikad, lubatud kasutada allkriips-sümbolit “_” eraldajana identifikaatori siseselt – see annab võimaluse paremini ja loetavalt programmeerida.
2. **Paralleelsust saab väljendada massiivoperatsioonides ja näiteks, WHERE-konstruktsiooniga.** Kui teostatakse massiivoperatsioone kasutades massiivintatsiooni (vt. peatükk 4), saab kompilaator lisainformatsiooni massiivi elementide omavahelise sõltumatus kohta antud operatsioonis ning teab, et antud operatsioon on võimalik teostada paralleelselt, näiteks erinevatel konveieritel või protsessoritel.
3. **Dünaamilise mäluhalduse käskude olemasolu.** Lisatud on käsud **ALLOCATE** ja **DEALLOCATE** massiividele mälu eraldamiseks (vt. alapunkti 4.4). Ilma mälureserveerimise võimaluseta ei kujuta me tänapäeval ühtegi keelt ettegi.
4. **Eri arhitektuuridel ühilduvuseks on olemas konstruktsioon **KIND**.** Selliselt saab näiteks ära määrata konkreetsele muutujale määratud bittide arvu mis ei sõltu kasutatavast arhitektuurist (vt. alapunkti 2.1.1).
5. **Tuletatavad tüübid (*User-defined types*.)** Nii saab defineerida loogiliselt kokkusobivaid andmestruktuure, mis lihtsustab programmide loetavust ning programmeerimist üldse. Lähemalt teeme kasutaja poolt defineeritavatest tüüpidest e. tuletatud tüüpidest juttu alapunktis 2.1.3.
6. **Rekursioon on lubatud.** Fortran90/95 tuleb aga kompilaatorile öelda, kui tegemist on rekursiivse protseduuriga, vt. lähemalt alapunkti 2.1.5.
7. **Objektid, protseduurid, tüübid, gobaalsed ja lokaalsed definitsioonid** saab pakkida kokku **moodulitesse** (vt. Peatükki 2), s.o. objekt-orienteeritus on rakendatud keelekonstruktsiooni **module** abil

- **Massiivide kujumuutuse operatsioonid (*reshaping and retyping*)** võimaldavad vältida Fortran77 staatilist **EQUIVALENCE** käsku (vt. alapunkti 4.7.2).
- **Protseduurliideste määramise võimalus (konstruktsioon **INTERFACE**)**, mis muuhulgas aitab kompilaatoril programmi optimeerida ning täpsustada semantikat teekide puhul. Teatud juhtudel on liides isegi kohustuslik (vt. alapunkti 2.1.6).
- **Võrreldes Fortran77-ga on järgnevad juhtimiskäsud uudsed:**
 - **DO...ENDDO**-tsükkel;
 - **DO...WHILE**-tsükkel;
 - **EXIT** – tsüklist väljumine;
 - **CYCLE** – hüpe tsükli uuele ringile;
 - **SELECT CASE** konstruktsioon.
- **Saab kasutada kapseldamist (*ik. encapsulation*)**. Seesmised (ehk privaatsed) protseduurid või muutujad saab teha kättesaadavaks vaid lokaalselt antud moodulis.
- **On olemas võimalus operaatorite üledefineerimiseks**. Operaatorite üledefineerimine võimaldab lihtsustada programme keeruliste andmestruktuuride korral, muuta programmi loetavamaks ning lisada soovitud definitsioone ka moodulitele.

Märkus. Enamasti, soovist toetada täielikult ka Fortran77 standardit, leidub keeles Fortran90 mitmeid igandeid, mis on pärit juba Fortran66-st! Need on standardis varustatud märkega “*obsolescent*”. Selliste struktuuride hulka kuuluvad näiteks aritmeetiline **IF**-direktiiv, **ASSIGN**, **ASSIGN**-märgistatud **GOTO**-käsk, **FORMAT**-direktiiv, **PAUSE**-käsk, mitme **DO**-tsükli lõpetamine ühel märgendatud real. Enamikku neist Fortran95 standardis niikuinii enam ei eksisteeri. Standardis on veel ka konstruktsioone, mis on ilma “*obsolescent*” märgendita, kuid mida siiski ei soovitata kasutada. Nende hulka kuuluvad sellised konstruktsioonid nagu fikseeritud lähteteksti kuju (st. nagu Fortran77 korral); kaudselt defineeritud muutujad (**soovitav on kasutada alati **IMPLICIT NONE** käsku!**); **COMMON**-blokk; **EQUIVALENCE**-käsk (– tuleks kasutada **TRANSFER**-käsku tüübi muutusteks, võtmesõna **POINTER** muutujate aliaste defineerimiseks ning **ALLOCATABLE** atribuuti ajutise mäluruumi haldamiseks); **ENTRY**-käsk (mis lubab funktsiooni või alamprogrammi täitmist alustada mujalt kui esimeselt direktiivilt).

Peatükk 2

Fortran9x ja objekt-orienteeritus

Programmeerimiskeele Fortran77 puhul on tegemist protseduurkeelega, s.t. kasutaja programm koosneb üldjuhul järjestikusest protseduuride (funktsioonide, alamprogrammide) kogumist, mis üksteist välja kutsuvad ning peale töö lõppu juhtimise oma väljakutsuja-protseduurile tagasi annavad. Andmete üleandmine ühelt protseduurilt teisele toimub kas parameetrite abil või kasutades globaalseid mäluaadresse (**COMMON**, **EQUIVALENT**-käsud). Protseduurkeelte “hädad” suurte programmipakettide kirjutamisel on üldteada (näiteks, piisavalt suure projekti korral, parandades programmi mingis osas ühe vea on kerge tekitada mitu uut juurde mingis teises kohas) ning soovitav on kasutada objekt-orienteeritud lähenemisviisi. Eelnevalt nägime, et Fortran90 standard toetab endiselt ka Fortran77 protseduur-lähenemisviisi, seega võimaldab kirjutada vanamoodsaid programme. Siin me püüame anda ülevaate, kuidas programmeerida objekt-orienteeritult Fortran9x keeles.

Kokkuvõtvalt võib öelda:

Objekt-orienteeritus (OO) on keeles Fortran90 realiseeritud moodulites; moodulid kätkevad endas klasse ning globaalseid andmestruktuure. OO paradigmast realiseerib Fortran9x vaid selle osa, mis tagab programmikoodi hea optimeeritavuse.

Loetleme siin lühidalt OO paradigma elemendid, mis on Fortran9x-s realiseeritud:

- andmetüüpide abstraktsioon – saab defineerida tuletatud tüüpe ehk kasutaja-andmetüüpe (lähemalt vt. alapunkti 2.1.3);
- andmete nähtavuspiirkondade juhtimine – **PRIVATE** ja **PUBLIC** atribuudid;
- kapseldamine – andmestruktuure ning meetodeid saab organiseerida moodulitesse ning saab kasutada eelmainitud andmete peitmise vahendeid;
- andmetüüpide ning meetodite päritavus ja laiendatavus – supertüübid, operaatorite üledefineerimine;
- taaskasutatavus – moodulid;
- polümorfism (*polymorphism*) – eri klassid ja objektid omavad sama funktsionaalsust. Saab kasutada programmikoodis, mis vajab seda funktsionaalsust sõltumata sellest, millise klassi või objektiga on tegu.

Järgnevalt käsitleme eeltoodud omaduste realiseerimise keeles Fortran9x täpsemalt

2.1 Andmetüübid

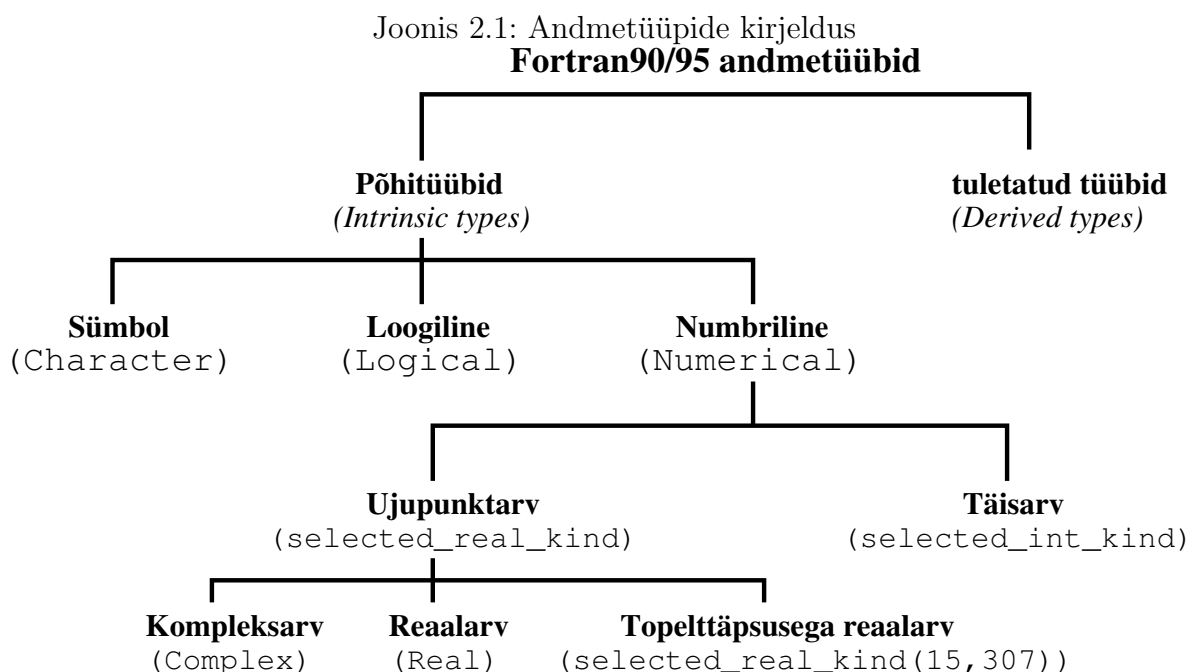
Fortran9x andmetüüpide käsitlemisel lähtume järgnevast skeemist:

Põhitüübid → tuletatud tüübid → abstraktsed tüübid → klassid

Toodud skeem illustreerib OO kontseptsioonile tuginedes klasside kujunemist lähtudes põhitüüpide organisatsioonist tuletatud tüüpidesse lisades vajalikud abstraktsioonid.

2.1.1 Põhitüübid

Fortran9x põhitüübid (*inglise k. intrinsic types*) võib jagada kolme klassi: sümboltüüp (*character*), loogiline (*logical*) ja numbrilised tüübid (vt. joonist 2.1).



Sümboltüüp on analoogiliselt teiste keeltega mõeldud positiivsete täisarvuliste väärtuste ehk ASCII-koodi elementide määranguks; loogilise tüübi puhul on võimalikud 2 väärtust: kas tõene (`.true.`) või väär (`.false.`). Numbrilised ehk arvutüübid jagunevad täisarvudeks ning ujukomaarvudeks. Arvutüübid 32-bitisel protsessoril on järgmised:

Tüüp	Bittide arv	Kümnenndkohtade arv	Piirkond
integer	16	10	-32,768 kuni 32,768
real	32	6	-10^{37} kuni 10^{37}
double precision ^{*)}	64	15	-10^{307} kuni 10^{307}
complex	2×32	2×6	2 real-tüüpi
*) F90 "igand" - vt. <code>selected_real_kind</code>			

Kuna erinevatel arvutiarkitektuuridel võivad standardse täisarvu või ujukomaarvu pikkused olla erinevad, siis ühilduvuse huvides on defineeritud käsud `selected_int_kind` ja `selected_real_kind`. Näiteks,

```

long = selected_int_kind(9)
topelt = selected_real_kind(15,307)
kvadraat = selected_real_kind(18,4932)

```

määrab täisarvutüübi **long** piirkonnaga -10^9 kuni 10^9 , ujukomaarvutüübi **topelt** 15 kümnendkohaga ja eksponendiga vahemikus ± 307 ning **kvadraat** 18 kümnendkohaga ja eksponendi piirkonnaga ± 307 . Juhul kui antud protsessor toetab toodud arvutüüpe, siis **integer(long)**, **real(topelt)** ja **real(kvadraat)** vastavad C++ tüüpidele **long int**, **double** ja **long double**. Juhul kui protsessor mõnda neist ei toeta, väljastatakse **kind**-muutuja väärtuseks negatiivne arv. Antud juhul võiks kasutada **real(topelt)** asemel ka Fortran77-st pärit **DOUBLE PRECISION** kuid seda peetakse Fortran9x igandiks.

2.1.2 Konstantide defineerimine; esimene näiteprogramm

Meie “hello-world”-programmiks on 2.1, kus on toodud näide konstantide defineerimisest ning nende organiseerimisest eraldi moodulisse. Mooduli kasutamiseks on **use**-käsk real number 17.

Lähtetekst 2.1: Matemaatiliste konstantide defineerimine

```

1  ! fail: Konstandid.f90
2  ! Moodul mis defineerib topelttäpsusega matem. konstante
3  module Konstandid ! Mooduli nimi
4  implicit none ! Identifikaatoritüübi vaikeväärtusi pole vaja
5  !integer,parameter :: dp = selected_real_kind(15,307)
6  integer,parameter :: dp = kind(1.d0) ! Alternatiivne kuju
7  real(dp),parameter :: e_Vaartus = 2.71828182845904523560287_dp
8  real(dp),parameter :: pi_Vaartus = 3.141592653589793238462643_dp
9  real(dp),parameter :: pi_Ruudus = 9.869604401089358618834491_dp
10 real(dp),parameter :: pi_Ruutjuur = 1.772453850905516027298167_dp
11 real(dp),parameter :: Ruutjuur_2st = 1.4142135623730950488_dp
12 ! kind-atribuudi alternatiivne süntaks: real(kind=dp) ...
13 real(kind=dp),parameter :: Ruutjuur_3st = 1.7320508075688772935_dp
14 end module Konstandid
15
16 program Test ! põhiprogrammi algus
17 use Konstandid ! Kasuta defineeritud konstante
18 implicit none ! Identifikaatoritüübi vaikeväärtusi pole vaja
19 real :: pi ! Lokaalse muutuja def.
20 print *, 'pi_Vaartus_on:_', pi_Vaartus ! Kuva konstant
21 pi = pi_Vaartus ; print *, 'pi=_', pi ! Esita madalama täpsusega
22 end program Test
23 ! Programmi väljund (Intel Fortran Compiler) :
24 ! pi Vaartus on: 3.14159265358979
25 ! pi = 3.141593

```

Märgime veel, et käsk **implicit none** (programmiringidel 4 ja 18) ütleb kompilaatorile, et ühelegi identifikaatorile antud blokis vaikimisi tüüpi ei määrata. Nii peab iga muutuja tüüpi olema määratud vastasel juhul genereeritakse kompileerimisel vastav viga. Juhul kui

`implicit none` ära jätta, saavad kõik muutujad, mis algavad sümboliga `i,j,k,l,m` või `n` automaatselt tüübiks `integer`, kõik ülejäänud aga tüübi `real`. Sellisel kompilaatori käitumisel ei oleks tegelikult midagi viga juhul, kui programmeerija suudab olla järjekindel, nimetades kõiki muutujaid vastavalt või defineerides tüübi vastasel juhul. Tõeline probleem võib aga tekkida juhul, kui kogemata teha mõne muutuja kirjpildis trükiviga. Tulemusena võib selline viga olla üliraskelt avastatav. Kuigi, ka siin on abi enamasti olemas: kompilaatoritel on parameeter (`-u` nii Inteli kui ka SUNi kompilaatori puhul), mis lisab `implicit none` vaikimisi justkui igale poole.

2.1.3 Kasutaja poolt defineeritavad e. tuletatud tüübid

Tuletatud tüüpide (*user-defined types*) loomisega tutvume järgneva näite varal:

Lähtetekst 2.2: Tuletatud tüüpide defineerimine

```

1  ! fail: tyybiloome.f90
2  program tyybiloome
3  implicit none
4  type keemiline_element           ! Tuletatud andmetüüp
5      character(len=2) :: symbol
6      integer          :: aatomnumber
7      real             :: aatommass
8  end type
9  type(keemiline_element) :: argoon, sysinik, neon ! elemendid
10 type(keemiline_element) :: Mendelejevi_Tabel(109) ! massiiv
11 real                      :: mass ! standardpikkusega ujupunktarv
12
13 sysinik%aatommass = 12.010 ! komponendi väärtuste
14 sysinik%aatomnumber = 6    ! omistamised
15 sysinik%symbol = "C"      !
16 argoon = keemiline_element("Ar", 18, 26.98) ! elemendi loomine
17 read *, neon              ! sisestada Ne 10 20.183
18 Mendelejevi_Tabel(5) = argoon ! elemendi lisamine massiivi
19 Mendelejevi_Tabel(17) = sysinik ! elemendi lisamine massiivi
20 Mendelejevi_Tabel(55) = neon ! elemendi lisamine massiivi
21 mass = Mendelejevi_Tabel(5)%aatommass ! komponendi väärtus
22 print *, mass              ! annab 26.98000
23 print *, neon              ! annab Ne 10 20.18300
24 print *, Mendelejevi_Tabel(17) ! annab C 6 12.01000
25 end program tyybiloome

```

Programmis 2.2 defineeritakse ridadel 4-8 tüüp `keemiline_element` ning ridadel 9-10 on näide loodud tüübi kasutamisest. Nagu näeme, kasutatakse individuaalkomponentide eraldajana märki `%` (nagu näiteks ridadel 13-15); uuele tüübimuutujale väärtustekomplekti omistamiseks võib kasutada konstruktsiooni:

```
<tüübimuutuja>=<tüüp>(<kompon.1_väärtus>,<kompon.2_väärtus>,...)
```

nagu on toodud real number 16. Märkame ka, et näiteks sisestus- ja väljastusoperatsioone võib teostada tüübimuutuja kui tervikuga (ridadel 17 ja 23).

Loomulikult võib tuletatud tüübis kasutada ka juba olemasolevaid tuletatud tüüpe.

Järgnev programm demonstreerib eeltoodud näites defineeritud tüüpi `keemiline_element` kasutamist tuletatud tüübis `ajalugu`:

Lähtetekst 2.3: Varemdefineeritud tuletatud tüüpi kasutamine tuletatud tüübis

```

1  ! fail: tyyptyybis.f90
2  program tyyptyybis
3  implicit none
4  type keemiline_element           ! Tuletatud andmetüüp
5      character(len=2) :: symbol
6      integer         :: aatomnumber
7      real            :: aatommass
8  end type
9  type ajalugu                     ! teine tüüp
10     character(len=31)           :: elemendi_nimi
11     integer                    :: avastamise_aasta
12     type(keemiline_element)    :: keemia
13 end type ajalugu
14 type(keemiline_element)        :: hapnik           ! elemendid
15 type(keemiline_element)        :: argoon, sysinik, neon ! elemendid
16 type(keemiline_element)        :: Mendelejevi_Tabel(109) ! massiiv
17 real                            :: mass ! standardpikkusega ujupunktarv
18 type (ajalugu)                  :: Joseph_Priestley ! Avastaja
19
20 sysinik%aatommass = 12.010      ! komponendi väärtuste
21 sysinik%aatomnumber = 6        ! omistamised
22 sysinik%symbol = "C"          !
23 argoon = keemiline_element("Ar", 18, 26.98) ! elemendi loomine
24 hapnik = keemiline_element("O", 76, 190.2) ! elemendi loomine
25 read *, neon                  ! sisestada Ne 10 20.183
26 Mendelejevi_Tabel( 5) = argoon ! elemendi lisamine massiivi
27 Mendelejevi_Tabel(17) = sysinik ! elemendi lisamine massiivi
28 Mendelejevi_Tabel(55) = neon   ! elemendi lisamine massiivi
29 mass = Mendelejevi_Tabel(5)%aatommass ! komponendi väärtus
30 print *, mass                  ! annab 26.98000
31 print *, neon                  ! annab Ne 10 20.18300
32 print *, Mendelejevi_Tabel(17) ! annab C 6 12.01000
33 Joseph_Priestley = ajalugu("Hapnik", 1774, hapnik) ! loomine
34 print *, Joseph_Priestley ! annab: (Intel Fortran)
35 ! Hapnik 1774 O 76 190.2000
36 end program tyyptyybis

```

2.1.4 Abstraktsed andmetüübid ja klassid

Anname siin lühikese kirjelduse abstraktsetest andmetüüpidest ning sellest, millised vahendid leiduvad Fortran9x-s nende realiseerimiseks. Võib öelda et **abstraktne andmetüüp** (*Abstract Data Type (ADT)*)

- väljendab andmetüübi põhiomadusi,
- on defineeritud programmeerimiskeelest sõltumatul kujul,
- defineeritakse eelkõige lähtudes käitumisest ning tegelik realisatsioon on teisejärguline

Toodud ADT omadused on Fortran9x keeles väljendatavad tuletatud tüüpide abil. Lisaks määrab ADT ära ka

- meetodid mis seotud antud andmetüübiga ja
- andmete ning meetodite nähtavuse.

Viimane omadus annab võimaluse peita ADT kasutaja eest ebaolulisi rakendusega seotud detaile. Me soovime et ADT kasutaja saaks ligipääsu vaid kasutajale olulistele komponentidele ning meetoditele ning ei vaeva teda üksikasjadega, kuidas miski realiseeritud on. Fortran9x annab selleks **PUBLIC** ja **PRIVATE** atribuutide lisamise võimaluse ADT eri komponentidele (nii muutujatele kui ka meetoditele).

Klass on sisuliselt vaid paar sammu edasi ADT-st. **Klass** on ADT laiendatuna kahe spetsiaalse meetodiga: **konstruktor** ja **destruktor**. Konstruktor on meetod, mis kutsutakse välja objekti loomisel – reserveeritakse mälu, algväärtustatakse muutujad. Destruktor aga vastupidi, teostab operatsioonid, mis on vajalikud objekti eksistentsi lõpetamisel: vabastab mälu jms.

Fortran95-s on automaatne mäluvabastus (Fortran90-s veel mitte). Automaatse mäluvabastuse korral tühistatakse mälueraldused automaatselt juhul, kui objekt ei ole enam aktiivne. Siiski on soovitatav organiseerida mäluvabastust ise. Programmi paremaks tööks ja paremaks optimeerimisvõimeks soovitatakse lisaks vabastada mälu vastupidises järjekorras reserveerimisele, kui muidugi võimalik. See vähendab mälu fragmenteeritust ning parandab töökiirust.

Märkus. Fortran9x võimaldab ühte moodulisse koguda rohkem kui ühe tuletatud tüüpi koos vastavate meetoditega. Lisaks saab moodulis defineerida ka globaalsed muutujad ja konstandid. Sellisena on defineeritav moodul mõnevõrra erinev klassikalise objekt-orienteeritud kontseptsiooni tavadest, võimaldades tegelikult teha rohkem kui “puhas” objekt-orienteeritud kontseptsioon lubaks.

Näide: Abstraktne andmetüüp ja klass

Järgnev näide illustreerib, kuidas defineerida abstraktseid andmetüüpe ning klasse.

Fibonacci arvudeks nimetatakse naturaalarvude jada $\{F_n\}$, kus $F_0 = F_1 = 1$ ja $n \geq 2$ korral

$$F_n = F_{n-1} + F_{n-2},$$

st. 1, 2, 3, 5, 8, 13, 21, ...

Lähtetekst 2.4: Fibonacci arvude klass

```

1 ! Fail: Fibonacci_arvud.f90
2 module klass_Fibonacci_arv
3
4 ! kõigepealt andmestruktuurid:
5 implicit none ! (Aamen kirikus.)

```

```

6  public :: Liida,Valjasta      ! meetodid kasutajale
7  type Fibonacci_arv          ! tuletatud tüüp
8      private                  ! privaatsed muutujad:
9      integer :: alumine,ylemine,piir
10 end type Fibonacci_arv
11
12 contains ! seejärel meetodid antud klassis
13
14 function uus_Fibonacci_arv(max) result(num) ! isetehtud konstruktor
15     implicit none
16     integer,optional :: max
17     type(Fibonacci_arv) :: num
18     num = Fibonacci_arv (0, 1, 0)      ! sisseehitatud konstruktor
19     if (present(max)) then ! juhul kui piir oli antud:
20         num = Fibonacci_arv (0, 1, max) ! sisseehitatud konstruktor
21     endif
22 end function uus_Fibonacci_arv
23
24 function Liida(farv) result(summa)
25     implicit none
26     type(Fibonacci_arv),intent(in) :: farv ! sisendparam.–ei muuda
27     integer :: summa
28     summa = farv%alumine + farv%ylemine ! liida komponendid
29 end function Liida
30
31 subroutine Valjasta(num)
32     implicit none
33     type (Fibonacci_arv),intent(inout) :: num ! (muudab argumenti)
34     integer :: j,summa
35     if (num%piir < 0) return ! ei ole midagi teha
36     print *, 'M_L Fibonacci(M)' ! pealkiri
37     do j = 1, num%piir ! tsüklil üle piirkonna
38         summa = Liida(num) ; print *, j, summa ! liida ja väljasta
39         num%alumine = num%ylemine ; num%ylemine = summa ! täiusta
40     end do
41 end subroutine Valjasta
42 end module klass_Fibonacci_arv
43 ! include 'Fibonacci_arv.f90' ! vajalik juhul kui moodul olnuks eraldi
44 ! failis antud nimega
45 program Fibonacci ! Põhiprogramm
46 use klass_Fibonacci_arv ! pärib muutujad ja liikmed
47 implicit none
48 integer, parameter :: lopp = 8 ! etteantud piir
49 type (Fibonacci_arv) :: num
50 num = uus_Fibonacci_arv(lopp) ! isetehtud konstruktor
51 call Valjasta (num) ! lüüa ja väljastada arvude jada
52 end program Fibonacci
53 ! käivitamine annab:
54 ! M Fibonacci(M)
55 ! 1 1
56 ! 2 2
57 ! 3 3
58 ! 4 5
59 ! 5 8
60 ! 6 13
61 ! 7 21
62 ! 8 34

```


Fortran9x keeles on tuletatud tüüpide puhul olemas nn. **sisseehitatud konstruktor**. Sisseehitatud konstruktori nimeks on tuletatud tüübi nimi; parameetritena antakse ette kõik tuletatud tüübi moodustavate muutujate soovitud väärtused. Toodud näites kasutatakse seda ridadel 18 ja 20 **kasutaja poolt defineeritava konstruktori** ehk **manuaalse konstruktori** (rida 14) loomisel. Nii on tavaks defineerida konstruktoreid mille puhul puuduvate liikmete väärtused asendatakse vaikeväärtustega.

Äsjatoodud näites kohtame ka direktiivi **intent** (ridadel 26 ja 33), mis on alati soovitatav lisada protseduuri argumentidele. See määrab ära antud funktsiooni või alamprogrammi kavatsuse antud argumendi suhtes. Võimalikud väärtused on:

- **intent(in)**, mis tähendab, et antud parameeter on vaid sisendparameeter ning selle väärtus antud blokis ei muutu. See tähendab muuhulgas, et antud protseduuris väärtuse omistamine sellele genereerib kompileerimisvea.
- **intent(out)** tähendab, et tegemist on vaid väljundparameetriga. Enne sellele väärtuse omistamist avaldistes kasutamine genereerib vea.
- **intent(inout)** – nii sise- kui ka väljundparameeter; kitsendusi ei ole.

Parameetrite liik on soovitatav ära määrata selleks, et vähendada eksimisvõimalusi programmeerimisel, kuid ka põhjusel, et nii antakse kompilaatorile optimeerimiseks vajalikku lisainformatsiooni, mis lihtsustab protsessi ja muudab tulemuse efektiivsemaks.

Toome siin ka näite **intent**-atribuutide kasutamise kohta:

Lähtetekst 2.5: Parameetrite edastamine alamprogrammidele väärtuse ja viida abil.

```

1  ! Fail: intent_tyybid.f90
2  program main
3    implicit none
4    integer :: sisestus
5    print *, "sisesta_täisarv:_ "
6    read *, sisestus; print *, "Sisestati_", sisestus
7    ! Parameetri edastus väärtuse abil:
8    call EiMuuda( (sisestus) ) ! Parameetrit mitte muuta
9    print *, "Peale_EiMuuda()_on_ta_", sisestus
10   ! Edastus viida abil:
11   call Muuda(sisestus) ! Kasuta ja muuda
12   print *, "Peale_Muuda()_on_ta_", sisestus
13 end program
14
15 subroutine Muuda(Viit)
16 ! Muuda JUHUL KUI parameeter anti ette viidana
17 implicit none
18 integer, intent(inout) :: Viit
19 Viit = 100;
20 print *, "Alamprogrammis_Muuda()_sai_ta_väärtuseks_", Viit
21 end subroutine Muuda
22 subroutine EiMuuda(Vaartus)
23 ! Mitte muuta JUHUL KUI parameeter anti ette väärtusena
24 implicit none
25 integer :: Vaartus
26 Vaartus = 100
27 print *, "Alamprogrammis_EiMuuda()_saab_ta_väärtuseks_", Vaartus
28 end subroutine EiMuuda ! Käivitamine annab:

```

```

29 ! sisesta täisarv: 82
30 ! Sisestati          82
31 ! Alamprogrammis Ei_Muuda() saab ta väärtuseks          100
32 ! Peale Ei_Muuda() on ta          82
33 ! Alamprogrammis Muuda() sai ta väärtuseks          100
34 ! Peale Muuda() on ta          100

```

2.1.5 Polümorfism OO programmeerimise kontseptsioonis

Polümorfismiks (*polymorphism*) nimetatakse eri klasside ja objektide sarnase funktsionaalsuse ühendamist. Polümorfismi abil saab programmis ühendada sarnast funktsionaalsust eri klassides ja objektides üldisesse funktsiooni või alamprogrammi nii, et programmeerimisel ei pea mõtlema, mis tüüpi objektiga on parajasti tegu.

Fortran9x lubab eri moodulitesse kuuluvatel tuletatud tüüpidel defineeritud erinevaid funktsioone ehk meetodeid ühendada ühise funktsiooni või alamprogrammi nime alla käsu `module procedure` abil liisedirektiivis. Toome siin järgneva näite:

Näide: Polümorfismi kasutamine

Lähtetekst 2.6: Geomeetrilised kujundid

```

1 ! Fail: Geomeetrilised_kujundid.f90
2 module klass_Ristkylik ! defineerime objekti esimesest klassist
3   implicit none ! ärme parem seda unusta
4   type Ristkylik
5     real :: alus, korgus
6   end type Ristkylik
7 contains ! Ristküliku pindala arvutamine
8   function ristkyliku_pindala(r) result(pindala)
9     type(Ristkylik), intent(in) :: r
10    real :: pindala
11    pindala = r%alus*r%korgus
12  end function ristkyliku_pindala
13
14  function loo_Ristkylik(kylg1, kylg2) result(nimi)
15    ! Konstruktor tüübile Ristkülik
16    real, optional, intent(in) :: kylg1, kylg2
17    type(Ristkylik) :: nimi
18    nimi = Ristkylik(1., 1.) ! Vaikimisi ühikruut
19    if (present(kylg1)) nimi = Ristkylik(kylg1, kylg1)
20    if (present(kylg2)) then
21      nimi = Ristkylik(kylg1, kylg2)
22    endif
23  end function loo_Ristkylik
24 end module klass_Ristkylik
25
26 module klass_Ring ! define the second object class
27 implicit none
28 real :: pi = 3.1415926535897931d0 ! Konstant Pi
29 type Ring
30 real :: raadius
31 end type Ring

```

```

32 contains ! Ringi pindala arvutamine
33   function ringi_pindala(c) result(pindala)
34     type (Ring), intent(in) :: c
35     real :: pindala
36     pindala = pi*c%raadius**2
37   end function ringi_pindala
38 end module klass_Ring
39
40 program geomeetrilised_kujundid ! mõlemad tüübid ühises funktsioonis
41 use klass_Ring
42 use klass_Ristkylik
43 implicit none
44 ! Ühendav interface-käsk pindala arvutamiseks mõlema tüübi korral
45 interface arvuta_pindala
46   module procedure ristkyliku_pindala, ringi_pindala
47 end interface
48 ! Deklareerime mõned geomeetrilised kujundid:
49 type (Ristkylik) :: neli_kylge, ruut, yhikruut
50 type (Ring) :: kaks_poolt ! sisemus, välimus
51 real :: pindala = 0.0 ! tulemus
52 ! Initsialiseeri ristkülik ja arvuta selle pindala
53 neli_kylge = Ristkylik(2.1,4.3) ! sisseehitatud konstruktor
54 pindala = arvuta_pindala(neli_kylge) ! üldfunktsioon
55 write(6,100) neli_kylge, pindala ! väljastatakse komponendid
56 100 format (f3.1,"_korda_",f3.1,"_ristküliku_pindala_on_",f5.2)
57 ! Initsialiseeri ring ja arvuta selle pindala
58 kaks_poolt = Ring(5.4) ! sisseehitatud konstruktor
59 pindala = arvuta_pindala(kaks_poolt) ! üldfunktsioon
60 write(6,200) kaks_poolt, pindala
61 200 format ("Ringi,_mille_raadius_on_",f3.1,"_pindala_on_",f9.5 )
62 ! Eri konstruktorite testimine:
63 neli_kylge = loo_Ristkylik(2.1,4.3) ! manuaalne konstruktor
64 pindala = arvuta_pindala(neli_kylge) ! üldfunktsioon
65 write(6,100) neli_kylge, pindala
66
67 ruut = loo_Ristkylik(2.1) ! manuaalne konstruktor 2
68 pindala = arvuta_pindala(ruut) ! üldfunktsioon
69 write(6,100) ruut, pindala
70
71 yhikruut = loo_Ristkylik() ! manuaalne konstruktor 3
72 pindala = arvuta_pindala(yhikruut) ! üldfunktsioon
73 write(6,100) yhikruut, pindala
74
75 end program geomeetrilised_kujundid ! Käivitamine annab:
76 ! 2.1 korda 4.3 ristküliku pindala on 9.03
77 ! Ringi, mille raadius on 5.4, pindala on 91.60885
78 ! 2.1 korda 4.3 ristküliku pindala on 9.03
79 ! 2.1 korda 2.1 ristküliku pindala on 4.41
80 ! 1.0 korda 1.0 ristküliku pindala on 1.00

```

Soovitav oleks uurida hoolega toodud programmi ja leida iseseisvalt vastused järgmistele küsimustele:

- Kuidas on rakendatud polümorfism? Milline osa programmist selle realiseerib (vt. ridu 45-47)?
- Kuidas kasutada sisseehitatud (*implicit*) konstruktoreid ja neid ise defineerida?

- Mida tähendab `optional` ehk suvandparameeter (vt. rida 16)?
- Pöörata muuhulgas tähelepanu `if`-direktiivi erinevatele kujudele ridadel 19 ja 20, (mõlemad on Fortran9x puhul lubatud)!

Näide: OO programmeerimine (sealhulgas `public`, `private` atribuudid ja operaatorite üledefineerimine, rekursioon)

Järgneva näite eesmärgiks on demonstreerida head programmeerimisstiili. Definieeritakse ratsionaalarvude klass, kusjuures tegelikud murdude liikmed ratsionaalarvudes on `private`-atribuudiga (vt. Programmi 2.7 ridu 6-9). Selleks, et kasutajal siiski oleks ligipääs antud andmetele, lisatakse spetsiaalsed päringufunktsioonid (read 66 ja 72). Tuleb võtta arvesse, et antud juhul väljaspool klassi enda meetodeid sisseehitatud konstruktorit ei saa kasutada ning tuleb hoolitseda ka selle eest, et klassi väljastpoolt kasutatavate meetodite hulgas leiduks vähemalt üks konstruktor vastavate õigustega (vt. ridu 101 ja 120).

Lähtetekst 2.7: Ratsionaalarvude klass

```

1  ! Fail: klass_Ratsionaalarv.f90
2  module klass_Ratsionaalarv
3      implicit none ! Õnneks ei läinud meelest lisada...:-)
4      ! kõik public-attribuudiga välja arvatud järgmised protseduurid:
5      private :: syt, taanda
6      type Ratsionaalarv
7          private ! privaatsed komponendid lugeja ja nimetaja
8          integer :: lugeja, nimetaja
9      end type Ratsionaalarv
10     ! operaatorite üledefineerimine interface-käsuga:
11     interface assignment (=)
12         module procedure omista_taisarv
13     end interface
14     interface operator (+)
15         module procedure liida_Ratsionaalarv
16     end interface
17     interface operator (*)
18         module procedure korruta_Ratsionaalarv
19     end interface
20     interface operator (==)
21         module procedure vordlus
22     end interface
23 contains                                     ! funktsioonid mida vaja aritmeetikaks
24
25     function liida_Ratsionaalarv(a,b) result(c) ! op. + üledefineerimine
26         type(Ratsionaalarv), intent(in) :: a,b
27         type(Ratsionaalarv) :: c
28         c%lugeja = a%lugeja * b%nimetaja + a%nimetaja * b%lugeja
29         c%nimetaja = a%nimetaja * b%nimetaja
30         call taanda(c)
31     end function liida_Ratsionaalarv
32
33     function konverteeri(nimi) result(value) ! ratsionaalarv reaalarvuks
34         type(Ratsionaalarv), intent(in) :: nimi
35         real :: value ! kümnenndmurru kuju
36         value = float(nimi%lugeja) / nimi%nimetaja

```

```

37  end function konverteeri
38
39  function kopeeri_Ratsionaalarv (nimi) result (uus)
40      type(Ratsionaalarv), intent(in) :: nimi
41      type(Ratsionaalarv)           :: uus
42      uus%lugeja = nimi%lugeja
43      uus%nimetaja = nimi%nimetaja
44  end function kopeeri_Ratsionaalarv
45
46  subroutine kustuta_Ratsionaalarv(nimi)
47      ! hõivatud ressursside vabastamine, siin lihtsalt nullimine
48      type(Ratsionaalarv), intent(inout) :: nimi
49      nimi = Ratsionaalarv(0,1)
50  end subroutine kustuta_Ratsionaalarv
51
52  subroutine omista_taisarv(uus,I) ! op. "=" üledef. täisarvu puhul
53      type(Ratsionaalarv), intent(out) :: uus ! operaatori vasak pool
54      integer, intent(in)             :: I    ! ja parem pool
55      uus%lugeja = I ; uus%nimetaja = 1
56  end subroutine omista_taisarv
57
58  recursive function syt(j,k) result(s) ! Suurim ühistegur
59      integer, intent(in) :: j, k ! lugeja, nimetaja
60      integer             :: s
61      if ( k == 0 ) then ; s = j
62      else ; s = syt(k,modulo(j,k)) ! rekursiivne käsk
63      endif
64  end function syt
65
66  function anna_Nimetaja(nimi) result(n) ! päringufunktsioon
67      type(Ratsionaalarv), intent(in) :: nimi
68      integer                         :: n ! nimetaja
69      n = nimi%nimetaja
70  end function anna_Nimetaja
71
72  function anna_Lugeja(nimi) result(n) ! päringufunktsioon
73      type(Ratsionaalarv), intent(in) :: nimi
74      integer                         :: n ! lugeja
75      n = nimi%lugeja
76  end function anna_Lugeja
77
78  subroutine poora(nimi) ! ratsionaalarvu pöördväärtus
79      type(Ratsionaalarv), intent(inout) :: nimi
80      integer                             :: temp
81      temp = nimi%lugeja
82      nimi%lugeja = nimi%nimetaja
83      nimi%nimetaja = temp
84  end subroutine poora
85
86  function vordlus(a_sisend, b_sisend) result(t_f) ! Võrdlus ==
87      type(Ratsionaalarv), intent(in) :: a_sisend, b_sisend ! vasak == parem
88      type(Ratsionaalarv)           :: a, b ! koopiad taandamiseks
89      logical                       :: t_f ! TRUE või FALSE
90      a = kopeeri_Ratsionaalarv(a_sisend)
91      b = kopeeri_Ratsionaalarv(b_sisend)
92      call taanda(a) ; call taanda(b) ! taanda väikseimale kujule
93      t_f = (a%lugeja==b%lugeja).and.(a%nimetaja==b%nimetaja)
94  end function vordlus
95

```

```

96  subroutine valjasta(nimi)                                ! murru väljastamiseks
97      type(Ratsionaalarv), intent(in) :: nimi
98      print *, nimi%lugeja, "/", nimi%nimetaja
99  end subroutine valjasta
100
101  function loo_Ratsionaalarv(lug, nim) result(nimi)
102  ! ratsionaalarvtüübi suvandkonstruktor
103      integer, optional, intent(in) :: lug, nim
104      type(Ratsionaalarv)           :: nimi
105      nimi = Ratsionaalarv(0,1) ! vaikeväärtused
106      if (present(lug)) nimi%lugeja = lug
107      if (present(nim)) nimi%nimetaja = nim
108      if (nimi%nimetaja == 0) nimi%nimetaja = 1
109      call taanda(nimi) ! lihtsusta
110  end function loo_Ratsionaalarv
111
112  function korruta_Ratsionaalarv(a,b) result(c) ! Op. "*" üledefin.
113      type(Ratsionaalarv), intent(in) :: a, b
114      type(Ratsionaalarv)           :: c
115      c%lugeja = a%lugeja * b%lugeja
116      c%nimetaja = a%nimetaja * b%nimetaja
117      call taanda(c)
118  end function korruta_Ratsionaalarv
119
120  function Ratsionaalarv_(lug, nim) result(nimi)
121  ! Public Konstruktor ratsionaalarvtüübile
122      integer, optional, intent(in) :: lug, nim
123      type(Ratsionaalarv)           :: nimi
124      if (nim==0) then ; nimi=Ratsionaalarv(lug,1)
125      else ; nimi = Ratsionaalarv(lug, nim)
126      end if
127  end function Ratsionaalarv_
128
129  subroutine taanda(nimi) ! lihtsaima ratsionalarvkujule leidmine
130      type(Ratsionaalarv), intent(inout) :: nimi
131      integer                             :: g ! suurim ühistegur
132      g = syt(nimi%lugeja, nimi%nimetaja)
133      nimi%lugeja = nimi%lugeja/g
134      nimi%nimetaja = nimi%nimetaja/g
135  end subroutine taanda
136  end module klass_Ratsionaalarv

```

Toodud näites tuleks tähelepanu pöörata eelkõige järgnevale:

- Kuidas toimub operaatorite üledefineerimine? (Vaata ridu 11-13, kus toimub omistamise üledefineerimine ning ridu 14-22, kus näitena on defineeritud üle mõned aritmeetilised operaatorid.)
- Rekursiooni puhul tuleb kompilaatorile öelda eraldi, et tegu on rekursiivse protseduuriga (vt. rida 58).
- Funktsioonides on soovitatav kasutada `result`-atribuuti, nii nagu toodud näites igal pool ka on tehtud. Märkime siin siiski, et näiteks ridade 66-70 asemel võiks kasutada ka kuju


```

66 integer function anna_Nimetaja(nimi)
67   anna_Nimetaja = nimi%nimetaja
68 end function anna_Nimetaja

```

Siiski on `result`-atribuut alati kohustuslik rekursiivsete funktsioonide korral (read 58-64).

Järgneb näide klass_Ratsionaalarv kasutamise kohta:

Lähtetekst 2.8: Ratsionaalarvude klassi kasutamine (põhiprogramm)

```

1  ! Fail: Ratsionaalarvu_test.f90
2  include 'klass_Ratsionaalarv.f90'
3  program main
4  use klass_Ratsionaalarv
5  implicit none
6  type(Ratsionaalarv) :: x, y, z
7
8  ! — Saab kasutada vaid juhul kui Ratsionaalarv ei ole private: —
9  ! x = Rational(23,7) ! sissehitatud konstr. kui public-komponendid
10 ! —
11 x = Ratsionaalarv_(23,7) ! public-atribuudiga konstruktor
12 write (*, '( "Ratsionaalarv_x_x==_" )', advance='no'); call valjasta(x)
13 write (*, '( "Kümnendkujul_x_x==_", g9.4) ') konverteeri(x)
14 call poora(x)
15 write (*, '( "pööratuna_1/x_x==" )', advance='no'); call valjasta(x)
16 x = loo_Ratsionaalarv() ! manuaalne konstruktor
17 write (*, '( "tegime_nullilise_x_x==" )', advance='no')
18 call valjasta(x)
19 y = 11 ! ratsionaalarv = taisarv üledefineeritud
20 write (*, '( "taisarv_y_y==_" )', advance='no'); call valjasta(y)
21 z = loo_Ratsionaalarv(23,7) ! manuaalne konstruktor
22 write (*, '( "tegime_ratsionaalarvu_z_z==" )', advance='no')
23 call valjasta(z)
24 ! Päringufunktsioonide testid:
25 write (*, '( "muru_z_peal_on_" , g4.0) ') anna_Lugeja(z)
26 write (*, '( "muru_z_all_on_" , g4.0) ') anna_Nimetaja(z)
27 ! Mitmesuguste funktsioonide teste:
28 write (*, '( "Teeme_ratsionaalarvu_x_x=20/192_" )', advance='no')
29 x = loo_Ratsionaalarv(20,192) ! manuaalne konstruktor
30 write (*, '( "lihtsusatult_x_x==" )', advance='no'); call valjasta(x)
31 write (*, '( "x_kopeerimine_y-sse_annab_" )', advance='no')
32 y = kopeeri_Ratsionaalarv(x)
33 write (*, '( "peale_kopeerimist_y_y==" )', advance='no'); call valjasta(y)
34 ! Üledefineeritud operaatorite testimine:
35 write (*, '( "z*_x_x_annab_" )', advance='no'); call valjasta(z*x)
36 write (*, '( "z+_x_x_annab_" )', advance='no'); call valjasta(z+x)
37 y = z ! Üledefineeritud omistamine
38 write (*, '( "y_y=z_z_annab_y_y_väärtuseks_" )', advance='no')
39 call valjasta(y)
40 write (*, '( "loogiline_y_y==_x_x_annab_" )', advance='no'); print *, y==x
41 write (*, '( "loogiline_y_y==_z_z_annab_" )', advance='no'); print *, y==z
42 ! Destruktor:
43 call kustuta_Ratsionaalarv(y) ! tegelikult vaid nulli siin...
44 write (*, '( "y kustutamine annab y = " )', advance='no')

```

```

45  call valjasta(y)
46  end program main ! Programm väljastab (SUN Fortran):
47  ! Ratsionaalarv x = 23 / 7
48  ! Kümnendkujul x = 3.286
49  ! pööratuna 1/x = 7 / 23
50  ! tegime nullilise x = 0 / 1
51  ! taisarv y = 11 / 1
52  ! tegime ratsionaalarvu z = 23 / 7
53  ! murru z peal on 23
54  ! murru z all on 7
55  ! Teeme ratsionaalarvu x = 20/192, lihtsusatult x = 5 / 48
56  ! x kopeerimine y-sse annab peale kopeerimist y = 5 / 48
57  ! z * x annab 115 / 336
58  ! z + x annab 1139 / 336
59  ! y = z annab y väärtuseks 23 / 7
60  ! loogiline y == x annab F
61  ! loogiline y == z annab T
62  ! y kustutamine annab y = 0 / 1

```

Nagu juba eelnevalt mainisime, ei saa (erinevalt lähteteksti 2.7 ridadega 49,105,124,125), programmis `Ratsionaalarvu_test.f90` (lähteteksti 2.8 ridadel 11,16,21,29) kasutada sisesehitatud konstruktorit. Põhjuseks on see, et tüüpi `Ratsionaalarv` liikmed on `private`-atribuudiga ning kättesadavad vaid mooduli enda seest, `use`-käsk `real 4` selleks veel õigust ei anna.

Antud näites on destruktor (lähteteksti 2.7 ridadel 46-50) vaid illustratiivne, kuna mälueraldust `Ratsionaalarv`-tüübis endas ei teostata. Vastasel korral oleks seal vaja kutsuda välja vastavad `deallocate`-käsud.

2.1.6 Liidesedirektiiv

Eelnevates näidetes esines juba `interface`-käsu erinevaid vorme (lähtetekst 2.6 read 45-47, lähtetekst 2.7 read 11-22), mis on seotud polümorfismi ja operaatorite üledefineerimisega. `Interface`-direktiivi üks põhilisi kasutuseesmärke tuleneb vajadusest anda kompilaatorile lisainformatsiooni välise funktsiooni või alamprogrammi parameetrite kohta. Millistel juhtudel on vaja kasutada `interface`-käsku?

Lihtne vastus oleks: siis kui kompilaator kurdab selle puudumise üle. Loetleme siin veel mõningaid juhtusid, kus `interface`-käsk peab kindlasti olema toodud:

- andes edasi alamprogrammi massiivi, millel on teada vaid järk (näiteks `A(:, :)`, `B(:)`);
- kutsudes välja funktsioon mis tagastab:
 - teadmata suurusega massiivi;
 - sõne mille pikkus ei ole ette teada (vt. näiteks programmi 3.4 read 5-14) või
 - viida;
- andes alamprogrammile parameetrina ette funktsiooni nime, mida välja kutsuda töö käigus. Selline praktika on üsna levinud Fortran77 puhul. Kuigi antud konstruktsiooni

väga ei julgustata Fortran9x puhul, toome siin näite, kuidas seda saab teha – kasutada tuleb `interface`-konstruktsiooni:

Lähtetekst 2.9: Arvu π arvutamine kasutades Simpsoni valemit (põhiprogramm)

```

1  ! Fail: Pi/leiapi.f90
2  ! Programm mis arvutab Pi nii täpselt kui võimalik kasutamata
3  !   mingit eelteadmist pi kohta integreerides  $f(x) = \sqrt{1-x**2}$ 
4  !   lõigul [0 ,1/2]
5  ! Informatsiooniks Pi väärtus 25 tüvekohaga:
6  !   real (kind=rk) :: PI25DT
7  !   parameter (PI25DT = 3.141592653589793238462643_rk)
8  program leiapi
9  use RealKind ! uju.arv-tüübi (rk) äramääramine moodulis RealKind
10 implicit none
11 real(kind=rk) :: alpha, beta
12 real(kind=rk) :: integraal, diff, diff_eelmine, theta, c
13 real(kind=rk) :: pi, pi_eelmine, err, err_eelmine
14 real(kind=rk) :: csimpson ! anname ette funktsiooni tüübi
15 integer :: i, n
16 interface
17   function func_pi(x) result(f)
18     use RealKind
19     real(kind=rk) :: f
20     real(kind=rk), intent(in) :: x
21   end function func_pi
22 end interface
23 alpha = 0.0_rk
24 beta = 0.5_rk
25 ! Lähendame integraali kasutades komposiit-Simpsoni valemit võttes
26 !   n = 2**i, kus i = 1, ... ,30
27 integraal = csimpson(func_pi, alpha, beta, 2)
28 pi = 12.0_rk * (integraal - sqrt(3.0_rk)/8.0_rk)
29 diff_eelmine = pi
30 pi_eelmine = pi
31 print*, 'i_n_PI_Väärtuse_muutumine'
32 print*, 1, 2, pi
33 do i=2,30
34   n = 2**i
35   integraal = csimpson(func_pi, alpha, beta, n)
36   pi = 12.0_rk*(integraal-sqrt(3.0_rk)/8.0_rk)
37   ! Leia eelneva ja praeguse lähendi erinevus,
38   !   kui see enam ei vähene, lõpeta
39   diff = abs(pi-pi_eelmine)
40   if (diff_eelmine/diff < 1.0_rk) exit
41   print*, i, n, pi, diff
42   diff_eelmine = diff
43   pi_eelmine = pi
44 end do
45 end program leiapi
46 ! Programmi väljund (SUN Fortran):
47 ! i n PI Väärtuse_muutumine
48 ! 1 2 3.1415454321631157
49 ! 2 4 3.141589571116926 4.413895381016886E-5
50 ! 3 8 3.1415924586206944 2.887503768533861E-6
51 ! 4 16 3.141592641366758 1.8274606361501355E-7
52 ! 5 32 3.1415926528252624 1.1458504367567457E-8
53 ! 6 64 3.141592653512002 7.16739556591318E-10

```

```

54 ! 7 128 3.1415926535868043 4.480238402493342E-11
55 ! 8 256 3.1415926535896083 2.8039792709932953E-12
56 ! 9 512 3.141592653589785 1.7674750552032492E-13
57 ! 10 1024 3.1415926535897935 8.43769498715119E-15

```

Märgime siin, et programmi 2.9 real 14 toodud väljakutsutava funktsiooni tüübimäärang peab olema antud, kui me tahame antud funktsiooni välja kutsuda (ridadel 27,35), vastasel korral annab kompilaator vea. Samas, ilma `interface`-blokit (read 16-22) antud programmi vigadeta kompileerida ei õnnestu. See on defineeritud ka integraali arvutavas funktsioonis, mis realselt antud funktsiooni kasutab:

Lähtetekst 2.10: Simpsoni valemi rakendamine integraali leidmiseks

```

1 ! Fail:Pi/csimpson.f90 — Lihtne programm mis lähendab f.-i func
2 !   integraali lõigul [alpha,beta] kasutades Simpsoni valemit
3 function csimpson(func, alpha, beta, n) result(integraal)
4   use RealKind
5   implicit none
6   real(kind=rk)          :: integraal
7   real(kind=rk), intent(in) :: alpha, beta
8   integer, intent(in) :: n
9   ! Interface-blokk mis tagab et argumendid oleksid õiged
10  interface
11    function func(x) result(f)
12      use RealKind
13      real(kind=rk) :: f
14      real(kind=rk), intent(in) :: x
15    end function func
16  end interface
17  real(kind=rk) :: f_vasak, f_kesk, f_parem, h
18  integer :: i
19  h = (beta-alpha)/dble(n) ! dble() - topelttäpsusega uju.arvuks
20  f_vasak = func(alpha)
21  integraal = 0.0_rk
22  do i=1,n
23    f_kesk = func(alpha+(i-0.5_rk)*h)
24    f_parem = func(alpha+i*h)
25    integraal = integraal+(h/6.0_rk) &
26              * (f_vasak+4.0_rk*f_kesk+f_parem)
27    f_vasak = f_parem
28  enddo
29 end function csimpson

```

Järgnevas on realiseeritud funktsiooni $\sqrt{1-x^2}$ arvutamine:

Lähtetekst 2.11: Funktsiooni rakendus π arvutamiseks.

```

1 ! Fail: Pi/func_pi.f90
2 function func_pi(x) result(f)
3   use RealKind
4   implicit none
5   real(kind=rk) :: f
6   real(kind=rk), intent(in) :: x
7   if (abs(x) <= 1.0_rk) then
8     f = sqrt(1.0_rk-x**2)

```

```

9   else
10  print *, 'Viga: ', x, ' ei ole func_pi(x) määramispiirkonnas.'
11  stop
12  end if
13 end function func_pi

```

Ujukomaarvu tüüpi määranguks on siin defineeritud moodul:

Lähtetekst 2.12: Moodul topelttäpsuse määranguks

```

1  ! Fail: Pi/RealKind.f90 — Moodul topelttäpsuse määratlemiseks
2  Module RealKind
3  implicit none
4  !integer, parameter :: rk = selected_real_kind(6,37) ! real
5  integer, parameter :: rk = selected_real_kind(15,307) ! double
6  !integer, parameter :: rk = selected_real_kind(18,4932) ! 4-kordne
7  end module RealKind

```

Märgime, et toodud programminäites topelttäpsuse asendamiseks `real`-tüüpi ujuko-
maarvudega piisab vaid muudatusest failis 2.12. Väljakommenteeritud 4-kordne täpsusaste
(rida 6) on paraku küll vähestel kompilaatoritel defineeritud (näiteks CRAY arhitektuuril).

Peatükk 3

Keele Fortran9x elemendid

Järgnevas toome mõningaid võrdlevaid tabeleid erinevate keelte süntaksi kohta. Toodud valik ei pretendeeri täielikkusele, vaid on pigem illustratiivse iseloomuga, lihtsustamaks Fortran9x keelekonstruktsioonide mõistmist eelneva kogemuse baasil mõnest teisest keelest.

3.1 Kommentaaride lisamine lähteteksti ridadele

Kommentaari lisamiseks Fortran9x programmiteksti suvalisele reale piisab sümbolist `!`. Kommentaar kestab kuni reavahetuseni. See on üsna sarnane näiteks MATLABi puhul, kus kommentaari alustussümboliks on `%` või C, C++ ja Java oma `//`-kommentaaridega. Järgnevas tabelis on toodud kokkuvõtte eri keelte kommentaaride kujust:

Keel	Süntaks	Asukoht lähtekoodis
MATLAB	<code>% kommentaar (rea lõpuni)</code>	suvaline
C, C++, Java	<code>// kommentaar (rea lõpuni)</code>	suvaline
F90	<code>! kommentaar (rea lõpuni)</code>	suvaline
F77	<code>* kommentaar</code>	1. veerg

Märgime, et C, C++ ja Java `/*...*/`-kommentaarile sarnast konstruktsiooni, kus kommentaar võib jätkuda ka peale reavahetust, Fortran9x ei oma. Seega, juhul kui on näiteks vaja kommenteerida välja pikem programmilõik lähtetekstis, lisatakse kogu bloki iga rea ette sümbol `!`.

3.2 Muutujate põhitüübid

Järgnevas tabelis on toodud eri keelte põhiliste andmetüüpide võrdlus:

Tüüp	C++	F90	F77
byte	char	character ::	character
integer	int	integer ::	integer
ujukomaarv	float	real ::	real
topelttäpsusega ujup.	double	real*8 ::	double precision
kompleksarv		complex ::	complex
loogiline tüüp	bool	logical ::	logical
osuti tüüp	*	pointer ::	
struktuur	struct	type ::	

Märgime, et Fortrani keeles on muutujatel vaikimisi tüüp ette antud muutuja esitähed järgi. Antud käitumist on võimalik muuta `implicit`-käsu abil. Vaikeväärtused on näiteks defineeritavad järgnevatel käskudega:

```
IMPLICIT INTEGER (I-N) ! F77 ja F90 vaikeväärtus
IMPLICIT REAL (A-H,O-Z) ! F77 ja F90 vaikeväärtus
```

Käsuga `implicit none` saab aga automaatselt tüübimäärangu üldse välja lülitada. `Implicit`-käsk tuleb kirjutada vahetult peale `use`-käskusid enne muutujate deklaratsioone.

3.3 Aritmeetilised operaatorid

Aritmeetilised operaatorid on enamasti üsna standardsed eri keeltes. MATLABis on lisaks olemas elemendiviisilised aritmeetilised operatsioonid (sel juhul kirjutatakse operaatori ette “.”, et eristada näiteks maatriksite korrutamist elementviisilisest korrutamisest). C++ keelele omaseid “++”-tüüpi operaatoreid Fortranis ei leidu ning tuleks kasutada kuju `muutuja = muutuja+1`. Ka on Fortranis täisarvulisel jagamisel jäägi leidmiseks sisseehitatud funktsioon `modulo(a,b)`, kus `b` on `a` jagaja.

Kirjeldus	MATLAB	C++	Fortran
liitmine	+	+	+
lahutamine	-	-	-
korrutamine	* ja .*	*	*
jagamine	/ ja ./	/	/
eksponentsiaal	^ ja .^	pow(x,y)	**
jääk		%	
inkrement		++	
dekrement		--	

3.4 Võrdlusoperaatorid

Aritmeetiliste ja loogiliste operaatorite puhul võib öelda, et Fortran9x on pooled C++ keele operaatorite kujud omaks võtnud, samas kui loogilised operaatorid käivad endist viisi veel Fortran77 stiilis. (Samas, näiteks, ka `.eq.` `==` asemel on lubatud.)

Kirjeldus	MATLAB	C++	F90	F77
võrdne	<code>==</code>	<code>==</code>	<code>==</code>	<code>.EQ.</code>
mittevõrdne	<code>~=</code>	<code>!=</code>	<code>/=</code>	<code>.NE.</code>
väiksem kui	<code><</code>	<code><</code>	<code><</code>	<code>.LT.</code>
väiksem või võrdne	<code><=</code>	<code><=</code>	<code><=</code>	<code>.LE.</code>
suurem kui	<code>></code>	<code>></code>	<code>></code>	<code>.GT.</code>
suurem või võrdne	<code>>=</code>	<code>>=</code>	<code>>=</code>	<code>.GE.</code>
loogiline eitus	<code>~</code>	<code>!</code>	<code>.NOT.</code>	<code>.NOT.</code>
loogiline JA	<code>&</code>	<code>&&</code>	<code>.AND.</code>	<code>.AND.</code>
loogiline inklusiivne VÕI	<code>!</code>	<code> </code>	<code>.OR.</code>	<code>.OR.</code>
loogiline eksklusiivne VÕI	<code>xor</code>		<code>.XOR.</code>	<code>.XOR.</code>
loogiline ekvivalents	<code>==</code>	<code>==</code>	<code>.EQV.</code>	<code>.EQV.</code>
loogiline mitteekvivalents	<code>~=</code>	<code>!=</code>	<code>.NEQV.</code>	<code>.NEQV.</code>

Näide. `Select case` süntaks

Järgnevas programmis on toodud näide, kuidas kasutatakse muutuja väärtusel põhinevat hargnemist:

Lähtetekst 3.1: Näide valikuoperaatori kasutamisest

```

1 program main
2 ! Näide valikuoperaatori select case süntaksi kasutamisest:
3 implicit none
4 integer :: taisarv ! kasutaja poolt sisestatav
5 print *, 'Sisesta_kas_0_või_1:'
6 read *, taisarv
7 select case(taisarv)
8   case (0)
9     print *, 'Valisid_0'
10  case (1)
11    print *, 'Valisid_1'
12  case default
13    print *, 'Vale_valik:_', taisarv
14 end select
15 end program main

```

3.5 Stringitöötlus

Stringi- ehk sõnetöötlusega tutvume järgneva kolme lihtsa näite varal.

Lähtetekst 3.2: Kahe stringi võrdlus, ühendamine ning pikkuse leidmine

```

1  ! Fail: stringide_kasutamine.f90
2  program stringide_kasutamine
3      ! Kahe stringi võrdlus, ühendamine ning pikkuse leidmine
4      implicit none
5      character(len=10) :: string1, string2
6      character(len=20) :: string3
7      integer           :: pikkus
8      print *, 'Sisesta esimene string (max 10 tähemärki): '
9      read  '(a)', string1    ! etteantud formaadiga lugemine
10     print *, 'Sisesta teine string (max 10 tähemärki): '
11     read  '(a)', string2    ! etteantud formaadiga lugemine
12     if (string1==string2) then ! (võrdlus)
13         print *, "On samad."
14     else
15         print *, "On erinevad."
16     end if
17     string3 = trim(string1)//trim(string2) ! Ühendamine
18     print *, 'Ühendatud string on: ', string3
19     pikkus = len_trim(string3)
20     print *, 'Ühendatud stringi pikkus on: ', pikkus
21 end program stringide_kasutamine

```

Toodud programmis on `trim()` Fortran9x põhifunktsioon, mis väljastab stringi, milles on etteantud stringist lõputühikud tagant ära jäetud. Sarnaselt, `len_trim()` väljastab vastava stringi pikkuse.

Järgnevas näites demonstreeritakse, kuidas on võimalik teisendada stringi täisarvuks:

Lähtetekst 3.3: Stringi konverteerimine täisarvuks

```

1  ! Fail: numbriline_string.f90
2  program numbriline_string
3      ! Numbrilise stringi konverteerimine täisarvuks
4      implicit none
5      character(len=5) :: vanusestring
6      integer         :: vanus
7      print *, "Sisesta oma vanus:_"
8      read *, vanusestring
9      ! konverteerime kasutades isesfaili lugemist:
10     read (vanusestring, fmt='(i5)') vanus ! konverteeri täisarvuks
11     print *, "Su täisarvuline vanus on _____", vanus
12     print '(("_Su binaarne vanus on _____",_b8)', vanus
13     print '(("_Su vanus kuueteistkümnendsüsteemis on_",_z8)', vanus
14     print '(("_Su vanus kaheksandsüsteemis _____",_o8)', vanus
15 end program numbriline_string

```

Lõpetuseks veel näide, kuidas saab muuta stringide sisu kas väikesteks tähtedeks või suurteks:

Lähtetekst 3.4: Tõste muutmine stringides

```

1  ! Fail: tosteteisendus.f90
2  program tosteteisendus
3  implicit none
4  ! Deklareerime funktsioonide interfeisid:
5  interface ! vaikesteks and suurteks
6      function vaikesteks(string) result(uus_string)
7          character(len=*), intent(in) :: string ! teadmata pikkus
8          character(len=len(string)) :: uus_string ! sama pikkusega
9      end function vaikesteks
10     function suurteks(string) result(uus_string)
11         character(len=*), intent(in) :: string ! teadmata pikkus
12         character(len=len(string)) :: uus_string ! sama pikkus
13     end function suurteks
14 end interface
15 ! Testime tõsteteisendusi
16 character(len=34) :: test='Ahvidele_võimalikult_palju_BANAANE'
17 print *, "Algselt_string:_", test
18 test = vaikesteks(test)
19 print *, "Peale_funktsiooni_vaikesteks:_", test
20 test = suurteks(test)
21 print *, "Peale_funktsiooni_suurteks:_", test
22 end program tosteteisendus
23
24 function vaikesteks(string) result(uus_string)
25 implicit none
26 character(len = *), intent(in) :: string ! teadmata pikkus
27 character(len = len(string)) :: uus_string ! sama pikkusega
28 character(len=30), parameter :: &
29     SUURED = 'ABCDEFGHJKLMNOPQRSTUVWXYZ', &
30     vaiksed = 'abcdefghijklmnopqrstuvwöäöüxyz'
31 integer :: k ! tsükliloendur
32 integer :: pos ! positsioon alfabeedis
33 uus_string = string ! kopeeri kogu string
34 do k = 1, len(string) ! tähtede muutmine
35     pos = index(SUURED, string(k:k)) ! kas täht SUURED hulgas?
36     if (pos/=0) uus_string(k:k) = vaiksed(pos:pos) ! muuda
37 end do
38 end function vaikesteks
39
40 function suurteks(string) result(uus_string)
41 implicit none
42 character(len=*), intent(in) :: string ! teadmata pikkus
43 character(len=len(string)) :: uus_string ! sama pikkusega
44 character(len=30), parameter :: &
45     SUURED = 'ABCDEFGHJKLMNOPQRSTUVWXYZ', &
46     vaiksed = 'abcdefghijklmnopqrstuvwöäöüxyz'
47 integer :: k ! tsükliloendur
48 integer :: pos ! positsioon alfabeedis
49 uus_string = string ! kopeeri kogu string
50 do k = 1, len(string) ! tähtede muutmine
51     pos = index(vaiksed, string(k:k)) ! kas täht vaiksed hulgas?
52     if (pos/=0) uus_string(k:k) = SUURED(pos:pos) ! muuda
53 end do

```

```

54 end function suurteks
55 ! Programm annab käivitamisel:
56 ! Algselt string: Ahvidele võimalikult palju BANAANE
57 ! Peale funktsiooni vaikesteks: ahvidele võimalikult palju banaane
58 ! Peale funktsiooni suurteks: AHVIDELE VÕIMALIKULT PALJU BANAANE

```

Toodud näiteks kasutatakse sisefunktsiooni `index` (rida 51), mis väljastab alamstringi (teine parameeter) positsiooni etteantud stringis (esimene parameeter) või nulli, kui sellist alamstringi ei leidu.

3.6 Fortran95 viidad (*pointers*)

Andmed paiknevad arvuti mälus mingil kindlal mäluaadressil. Etteantud andmete mäluaadressi nimetatakse antud andmete **viidaks**, muutujat, mis võib antud aadressi hoida – **viitmuutujaks**. Fortran9x viidad sarnanevad pigem C++ viitadele, olles muutuja või massiivi osa aliaseks, samas võimaldamata näiteks mäluaadresside peenaritmeetikat nagu keeles C. Samas, erinevalt C++ keelest, on säilitatud viitade puhul hea optimeerimisvõime kompilaatorile.

Viida määranguks tuleb tüübidefinitsoonis lisada atribuut `pointer`. Viidale sihi (*target*) omistamise operaatoriks on: `=>`. Muutuja, millele tahame viidata, peab omama atribuuti `target`.

Üks Fortran9x sisefunktsioone on:

```
associated (viida_nimi , sihi_nimi)
```

mis väljastab `.true.` juhul kui antud viit on seotud valitud sihiga, vastasel juhul `.false.`

Viida sihi tühistamine (ehk nullimine) toimub eri keeltes järgmiselt:

C, C++	<code>pointer_name = NULL</code>
Fortran90	<code>nullify(viitade_nimede_loetelu)</code>
Fortran95	<code>pointer_name = NULL()</code>

Tähtis reegel: kus iganes ka viit programmis ei esine, seostatakse seda alati vastava sihiga e. viidatava konstruktsiooniga. Antud omaduse esitab programminäide:

Lähtetekst 3.5: Avaldised viitadega

```

1 ! Fail: avaldised_viitadega.f90
2 program avaldised_viitadega
3 ! Näide viitade kasutamise kohta avaldistes
4 implicit none
5 integer, pointer :: p, q, r      ! viidad
6 integer, target  :: i=1, j=2, k=3 ! sihid
7 q => j           ! q viitab täisarvule j
8 p => i           ! p viitab täisarvule i

```

```

9  ! Avaldis mis näeb küll välja justkui viidaaritmeetika asendab
10 ! viidad automaatselt sihiga:
11 q = p+2      ! tähendab: j = i + 2 = 1 + 2 = 3
12 print *,i,j,k ! väljasta sihtide väärtused: 1, 3, 3
13 p => k      ! p viitab nüüd k-le
14 print *,(q-p) ! tähendab väljastada: j - k = 3 - 3 = 0
15 ! kontrollime viidete seotust sihtidega:
16 print *,associated(r) ! false
17 r => k      ! r viitab nüüd k-le
18 print *,associated(p,i) ! false
19 print *,associated(p,k) ! true
20 print *,associated(r,k) ! true
21 end program avaldised_viitadega
22 ! Väljund (Intel Fortran):
23 !           1           3           3
24 !           0
25 ! F
26 ! F
27 ! T
28 ! T

```

Järgnevas toome näite kuidas saab kasutada viiteid massiivi eri osadele osutamiseks:

Lähtetekst 3.6: Viitade kasutamine massiivi eri osadele osutamiseks

```

1  ! Fail: massiiviviidad.f90
2  program massiiviviidad
3  ! Näiteid viitadest massiivi erinevatele osadele
4  implicit none
5  integer,parameter :: max = 5
6  integer           :: kesk, tsykliloendaja
7  integer,target   :: massiiv(max)= (/10,20,30,40,50 /)
8  integer,pointer  :: esiosa(:), tagaosa(:), otsad(:)
9  kesk = max/2 ! jagame viideteks esiosa ja tagaosa
10 esiosa => massiiv(::kesk)
11 tagaosa => massiiv(kesk+1:)
12 otsad => massiiv(1:max:(max-1))
13 do tsykliloendaja = 1,max ! tsükkel üle massiivi pikkuse
14   print *, 'massiiv( ', tsykliloendaja, &
15   ')=', massiiv( tsykliloendaja )
16 end do
17 print *, 'Massiivi_esiosa_on:', esiosa
18 print *, 'Massiivi_tagaosa_on:', tagaosa
19 print *, 'Massiivi_otsad_on:', otsad
20 end program massiiviviidad
21 ! Käivitamine annab (SUN):
22 ! massiiv( 1 )= 10
23 ! massiiv( 2 )= 20
24 ! massiiv( 3 )= 30
25 ! massiiv( 4 )= 40
26 ! massiiv( 5 )= 50
27 ! Massiivi esiosa on: 10 20
28 ! Massiivi tagaosa on: 30 40 50
29 ! Massiivi otsad on: 10 50

```

Viimases näites kasutasime juba mitmeid konstruktsioone, mis on seotud Fortran9x massiividega. Vaatlemegi neid lähemalt järgnevas peatükis.

Peatükk 4

Massiivid ja operatsioonid nendega

Massiivid ja nende käsitus keeles Fortran9x teeb antud keele tõeliselt atraktiivseks programmeerijale, kel on vaja töödelda suurel hulgal andmekogumeid. Massiive võib vaadelda kui keelde sisseehitatud objekte, on olemas nii konstruktori-laadsed käsud kui ka destruktori tüüpi operatsioonid, lisaks suurel hulgal vajalikke meetodeid massiividega. Massiivitöötlemise notatsioon on sarnane sellele, mis on kasutusel näiteks MATLAB-is. Kes on MATLAB-is (või sellele sarnanevas, tasuta litsentsiga interpretaatorikeeles SciLab) maatriksitega töötanud, oskab hinnata selle lihtsust ning väljenduslikkust.

4.1 Fortran90 massiivide omadused

Loetleme järgnevas põhilisi omadusi Fortran9x massiivide kohta:

- On olemas spetsiaalne massiivi-notatsioon, mis võimaldab lihtsasti töötada massiivi osadega ehk alammassiividega. Antud süntaks laieneb tegelikult kogu keelele. Näiteks on defineeritud kõik aritmeetilised operatsioonid ka massiivide puhul.
- Massiivid võivad olla nii funktsioonide või alamprogrammide parameetriteks kui ka funktsioonide poolt tagastatavateks väärtusteks.
- Massiivide tüübid seotuna mälueraldusviisiga võib jagada järgmisse kolme liiki:
 - etteantud kujuga (*assumed-shape*) massiivid,
 - reserveeritava mäluuga (*allocatable*) massiivid ja
 - automaatsed (*automatic*) massiivid.
- Leiduvad spetsiaalsed operaatorid tööks massiividega, nagu näiteks [WHERE](#)-direktiiv.

Et järgnev oleks lihtsamini arusaadav, siis lepime kõigepealt kokku järgnevate terminite suhtes:

Massiivi järk (*rank*) – antud massiivi dimensioonide arv;

massiivi ulatus (*extent*) – massiivi alumise raja ja ülemise raja vahe igas dimensioonis;

massiivi kuju (*shape*) – massiivi järk koos massiivi ulatusega;

massiivi suurus (*size*) – antud massiivi kõigi elementide arv.

4.2 Elementaaroperatsioonid massiividega

Avaldistes võib kasutada massiivi indekseerimata identifikaatorit. Sel juhul tehakse operatsioon kõigi massiivi elementidega. (Peab ainult hoolitsema selle eest, et kõigi avaldistes olevate massiivide kujud oleksid sarnased.) Toome siin lihtsa võrdleva näite Fortran77 ja Fortran90 massiivioperatsioonide kohta:

Fortran77	Fortran90
REAL a(100), b(100)	<code>real, dimension(100) :: a, b</code>
DO 1 i = 1,100	
a(i) = 2.0	<code>a = 2.0</code>
1 b(i) = b(i)*a(i)	<code>b = b * a</code>

Märgime ühtlasi, et enamus standardfunktsioone (nagu näiteks `SIN`, `ABS`, `DBLE` jne.) töötavad ka massiivide korral:

```
real, dimension(2,3) :: h, f
real :: g=.5
f = 10.0
h = sin(f)
g = sin(f(2,1))
```

Mõnedele standardfunktsioonidele saab lisada parameetri `dim`, mis näitab ära, millise dimensiooniga on tegu. Toome näiteks järgneva programmilõigu:

```
integer :: i, j
real, dimension(2,3) :: g
i = size(g) ! annab 6
i = size(g,dim=1) ! annab esimese dimensiooni pikkuse, 2
j = size(g,dim=2) ! annab teise dimensiooni pikkuse, 3
```

Funktsioon `shape` annab matriksi kuju, nagu on toodud järgnevas näitelõigus:

```
integer, dimension(2,3) :: s
s = shape(g) ! annab mõlemad dimensioonid, (/2,3/)
```

4.3 Massiivide sektorid ja konstruktorid

Juhul, kui on vaja teha mingit toimingut või operatsiooni teatud alammassiiviga etteantud massiivist, siis võib kasutada massiivide alamindeksite määratlemise kolmikut järgneval kujul:

```
algindeks : lõppindeks : samm
```

Toome järgnevas lihtsa võrdleva näite;

Fortran77:	Fortran9x:
<pre>REAL A(10,10),B(10,10) DO 1 J=1,8 DO 2 I=1,8 A(I,J) = B(I+1,J+1) 2 CONTINUE 1 CONTINUE</pre>	<pre>REAL, DIMENSION(10,10) :: A,B A(1:8,1:8) = B(2:9,2:9)</pre>

Juhul, kui kasutatavas kolmikus `samm` ära jätta, siis loetakse selle vaikeväärtuseks 1. Samas, juhul kui `algindeks` ja/või `lõppindeks` ära jätta, loetakse see vaikimisi väikseimaks või vastavalt suurimaks indeksiks antud massiivi dimensioonis. Seega näiteks

```
integer, dimension(2,3) :: g
integer, dimension(2,3,3) :: f
g = f(:, :, 2) ! tähendab sama mis g(1:2,1:3)=f(1:2,1:3,2)
g = f(:, 1, 1:3) ! tähendab sama mis: g(1:2,1:3)=f(1:2,1,1:3)
```

Massiivkonstante (st. massiive, mille kõigil või teatud osal elementidel on üks konstantne väärtus), saab defineerida näiteks nii:

```
real, dimension(1024) :: a
a(1:1024) = 1.0 ! sama mis: a = 1.0
a(::1024:2) = 2.0 ! a(1)=a(3)=...= 2.0
```

Teiseks mugavaks võimaluseks on kasutada järgmist tüüpi **massiivkonstruktorit**:

```
a = (/ (i, i=1,1024) /) ! a(1)=1, a(2)=2, a(3)=3, ...
a = (/ (i, i=1,4096,4) /) ! a(1)=1, a(5)=5, ...
```

Massiivkonstruktor töötab vaid 1-dimensionaalse massiivi korral, kuid kui on vaja mitmemõõtmelisi massiive initsialiseerida, võib kasutada sisseehitatud funktsioone `reshape` ja `shape` nagu toodud järgnevas näites:

Lähtetekst 4.1: Reshape ja shape kasutamine massiivi kujumuutuseks

```
integer, dimension(2,3) :: g
g = reshape( (/ 1., 2., 3., 4., 5., 6. /), shape(g))
```

Funktsioon `reshape` ütleb antud juhul, et lineaarset massiivi tuleb tegelikult interpreteerida 2-mõõtmelise massiivi `g` kujul.

Fortran interpreteerib mitmedimensionaalseid massiive kui ühedimensionaalseid, kusjuures esimene indeks varieerub kõige kiiremini. Öeldakse, et Fortranis salvestatakse matrisid veergude kaupa, erinevalt näiteks C keelest, kus salvestatakse ridade kaupa. Seega, massiivi `g` elementi indeksitega `(i,j)` interpreteeritakse tegelikkuses kui:

```
size(g, dim=1)*(j - 1) + i
```

Näiteks, massiiv:

```
g(1,1) = 1.; g(2,1) = 2.
```

```
g(1,2) = 3.; g(2,2) = 4.
g(1,3) = 5.; g(2,3) = 6.
```

on tegelikult salvestatud kujul (/ 1.,2.,3.,4.,5.,6. /).

Kordame veelkord, et Fortran9x massiiviavaldistes peavad massiivide kujud olema sarnased, st. dimensioonide arv ja ulatus igas dimensioonis peavad olema võrdsed. Kusjuures skalaare võib kasutada avaldistes koos mistahes kujul massiividega. Õnneks suudab enamuse sellistest eksimustest avastada kompilaator lähteteksti töötamise käigus.

4.4 Reserveeritava mäluga massiivid

Fortran9x-s saab dünaamiliselt reserveeritava mäluga massiividele mälu eraldada käsuga **ALLOCATE**. Massiivikirjelduses peab aga esinema **ALLOCATABLE** atribuut, mis deklareerib, et tegu on reserveeritava mäluga massiiviga:

```
1 program simulate
2   implicit none
3   integer :: n
4   integer, dimension (:, :) , allocatable :: a
```

See defineerib massiivi nime ja järgu (e. dimensioonide arvu) kuid jätab massiivi ulatuse määratlemata kuni näiteks massiivi suurus selgub peale sisselugemist:

```
5   print *, 'Sisesta n: '
6   read *, n
7   if (.NOT. allocated(a)) allocate(a(n, 2*n))
```

Kui massiivi **a** enam vaja ei ole, tuleb anda käsk:

```
108 deallocate(a)
109 end program simulate
```

NB! Eksisteerib oht mälulekkeks programmis juhul, kui uuesti reserveerida mälu samale massiivile ilma, et eelnevalt oleks kutsutud välja vastav **DEALLOCATE** käsk!

Märkus. Praktikaks kasutatakse atribuudi **allocatable** asemel tihti hoopis **pointer**-atribuuti. Põhjuseks on asjaolu, et **pointer**-atribuudi puhul **allocate-deallocate** käsud töötavad samuti nagu **allocatable**-atribuudi puhul, kuid mälueralduse või -vabastuse võib sooritada ka näiteks alamprogrammis. Ka ei ole **allocatable**-tüüpi massiivid nähtavad mälu reserveerinud alamprogrammiblokist väljaspool! Muuhulgas, ka kompilaatorid võivad anda sellist laadi vigade korral üsna ebaadekvaatset informatsiooni, kui üldse.

4.5 Automaatsed massiivid

Automaatseid massiive kasutatakse situatsioonides, kus alamprogramm või funktsioon vajab lokaalset massiivi, mille suurus sõltub sisendparameetritest. Seda saaks realiseerida näiteks nii:

```
1 subroutine vanaviisi(a,n,m,lokaalne_massiiv)
2   implicit none
3   integer :: n,m
4   real :: a(n,m),lokaalne_massiiv(n,m)
```

eeldades, et `lokaalne_massiiv` on saanud mälueralduse juba enne alamprogrammi `vanaviisi()` väljakutset. Juhul, kui `lokaalne_massiiv` on vajalik vaid lokaalselt alamprogrammi sees, võib kasutada automaatse massiivi konstruktsiooni. Sel juhul näeb see välja järgnevalt:

```
1 subroutine automaatne(a,n,m)
2   implicit none
3   integer :: n,m
4   real :: a(n,m),lokaalne_massiiv(n,m)
```

Mälueraldus massiivile `lokaalne_massiiv` toimub automaatselt alamprogrammi `automaatne()` sisenemisel ning mälu vabastatakse samuti automaatselt alamprogrammist väljumisel.

4.6 Fiktiivsete argumentide deklareerimine

Alamprogrammi parameetrites olevate massiivide korral on kasutajal teatud vabadus, milles suhtes alamprogrammi või funktsiooni tegelikud parameetriteks olevad massiivid on fiktiivsete parameetritega (st. vastavad parameetrid, mis on defineeritud alamprogrammi enda seemiseks kasutamiseks). On kolm erinevat võimalust: **etteantud kujuga**, **eeldatud suurusega** ning **eeldatud kujuga** fiktiivsed massiivparameetrid.

A. Etteantud kujuga (*explicit-shape*) fiktiivsed massiivargumendid

Ka Fortran77-s võib alamprogrammile edasiantav massiiv olla ettemääratud kujuga (järgu ja ulatusega), kuid see etteantud kuju ei pea tingimata kokku langema massiiviga, mis on alamprogrammi tegelikuks parameetriks. Alamprogrammi selline deklaratsioon määrab vaid ära lokaalselt, kuidas antud massiivi interpreteeritakse. Massiivide ulatused võivad olla kas staatilised või häälestatavad (*adjustable*) alamprogrammi argumentidega. Näiteks:

```
1 subroutine s1(a,b,c,k,m,n)
2   real :: a(100,100) ! on staatiline massiiv
3   real :: b(m,n) ! häälestatud
4   real :: c(-10:20,k:n) ! häälestatud
```

B. Eeldatava suurusega (*assumed-size*) fiktiivsed massiivargumendid (vananenud, mittesoovitav konstruktsioon)

FORTRAN 77 lubab ka nn. eeldatava suurusega massiive, kuid ainult viimases indeksis. Selleks asendatakse vastav raja sümboliga “*”. (Põhineb sellel, et massiivide tegelik salvestus mälus on lineaarne. Kuid selline mälumudel ei ole tegelikult hea näiteks hajussüsteemide korral ning on mittesoovitav kuju.) Toome siin näite:

```

1 subroutine s2(a,b,c,k,m,n)
2   real :: a(100,100) ! staatiline
3   real :: b(m,*) ! eeldatava suurusega
4   real :: c(-10:20,k:*) ! eeldatava suurusega

```

C. Eeldatava kujuga (*assumed-shape*) fiktiivsed massiivargumentid

Sellisel juhul tuleb ette anda vaid massiivi järk. Ulatuset ette ei määrata. Iga dimensiooni kohta on vaja anda kirjeldus kujul: [algindeks]:[lõppindeks]. Näiteks,

```

1 subroutine s3(a,b,c,k)
2   real :: a(100,100) ! staatiline
3   real :: b(:, :) ! eeldatava kujuga
4   real :: c(-10:20,k:) ! häälestatud

```

NB! Antud juhul peab seda alamprogrammi väljakutsuvas blokis olema defineeritud `interface`-liides.

Kokkuvõtvalt toome erinevad fiktiivsete massiivargumentide võimalused ära järgnevas näites alamprogrammi päisest:

```

subroutine massiivtybid(f,g,nx,ny,nz)
  implicit none
  integer :: nx, ny, nz
  real, dimension(2,3,4) :: ff ! määratud kuju
  real, dimension(nx,ny,ny) :: f ! eeldatav suurus
  real, dimension(:, :) :: g ! eeldatav vorm
  real, dimension(size(g,1), size(g,2)) :: s ! automaatne
  real, dimension(:, :), allocatable :: t ! allokeeritav massiiv
  !.....
end subroutine

```

4.7 Standardfunktsioone massiividega

Vaatleme mõningaid Fortran90 sisseehitatud (ehk standard-)funktsioone massiivide kohta päringute tegemiseks. Need võimaldavad kirjutada üldisemaid protseduure, mis ei sõltu massiivi tüübist tegelikes argumentides:

4.7.1 Päringufunktsioonid

Loetleme järgnevalt mõningaid päringufunktsioone massiivide kohta:

`size(A[,dim])`: Väljastab massiivi `A` suuruse, või juhul kui antud `dim`, siis antud mõõtme ulatus.

`shape(B)`: Väljastab 1-mõõtmelise täisarvulise massiivi, mille elementideks on `B` iga mõõtme ulatus.

`lbound(C[,dim])`: Annab massiivi `C` iga mõõtme kohta alumise indeksi, või siis vastava arvu dimensiooni `dim` olemasolu korral.

`ubound(D[,dim])`: Sarnane funktsiooniga `lbound`, kuid väljastatakse ülemised indeksid. Illustreerime toodud funktsioone lihtsa näitega:

Lähtetekst 4.2: Lihtne näide päringufunktsioonide kohta

```

1  ! Fail: paring.f90
2  program paring
3      integer , dimension(-30:-25,12:18,0:4,5) :: a
4      print *, 'size(a): ', size(a)
5      print *, 'size(a,1): ', size(a,1)
6      print *, 'lbound(a): ', lbound(a)
7      print *, 'lbound(a,2): ', lbound(a,2)
8      print *, 'ubound(a): ', ubound(a)
9      print *, 'ubound(a,3): ', ubound(a,3)
10     print *, 'shape(a): ', shape(a)
11 end program paring
12 ! Programmi väljund (Intel Fortrani kompil.)
13 !   size(a):           1050
14 !   size(a,1):         6
15 !   lbound(a):         -30           12           0           1
16 ! lbound(a,2):         12
17 !   ubound(a):         -25           18           4           5
18 ! ubound(a,3):         4
19 !   shape(a):          6           7           5           5

```

Fortran9x massiive võib sisuliselt lugeda teatud spetsiaalseteks **objektideks** millel on olemas kindlaksmääratud omadused ja meetodid päringute sooritamiseks nende omaduste kohta. Järgnev näide demonstreerib muuhulgas, kuidas saab alamprogrammile anda ette parameetreid (rida 49) olles kindel, et tegelik parameeter vastab õigele fiktiivsele parameetrile:

Lähtetekst 4.3: Päringufunktsioonide kasutamine alamprogrammides

```

1  ! Fail: testarrfun.f90
2  module vahetus
3  contains
4      subroutine vahetaja(a1,b1,a2,b2)
5          implicit none
6          integer , optional , dimension(:) :: a1,b1
7          integer , optional , dimension(:,:) :: a2,b2
8          integer , dimension(size(a1)) :: work1
9          !integer , dimension(size(a2,1),size(a2,2)) :: work2 ! ifc annab
10                                     !runtime-vea:
11          !Allocate error 494: Allocation of Array with extent of
12          !           -459085316 failed
13          integer , dimension(:,:), allocatable :: work2
14          if (present(a1)) then
15              work1=a1
16              a1=b1
17              b1=work1
18          endif
19          if (present(a2)) then
20              allocate(work2(size(a2,1),size(a2,2)))

```

```

21     work2=a2
22     a2=b2
23     b2=work2
24     deallocate(work2)
25     endif
26 end subroutine vahetaja
27 end module vahetus
28 program testarrfun
29 use vahetus
30 implicit none
31 integer , dimension(4)    :: a,b
32 integer , dimension(3,3) :: aa,bb
33 integer , dimension(2) , parameter :: d=shape(bb)
34 integer :: i
35 a=(/(i , i=1,size(a))/)
36 b=(/(i , i=size(a),1,-1)/)
37 print *, 'a_□= ', a
38 print *, 'b_□= ', b
39 aa=reshape((/(i , i=1,size(aa))/) , (/3,3/))
40 !bb=reshape((/(i , i=size(bb),1,-1)/) , (/3,3/))
41 bb=reshape((/(i , i=size(bb),1,-1)/) , d)           ! (sama tulemus:)
42 print *, 'aa_□= ', aa
43 print *, 'bb_□= ', bb
44 call vahetaja(a,b)
45 print *, '_____□Peale□vahetusi:□_____'
46 print *, 'a_□= ', a
47 print *, 'b_□= ', b
48 !call vahetaja(a,b,aa,bb) ! aga a ja b juba vahetatud! :
49 call vahetaja(a2=aa,b2=bb)
50 print *, 'aa_□= ', aa
51 print *, 'bb_□= ', bb
52 end program testarrfun
53 ! Programmi väljund (Sun Fortran komp.) :
54 ! a  = 1 2 3 4
55 ! b  = 4 3 2 1
56 ! aa = 1 2 3 4 5 6 7 8 9
57 ! bb = 9 8 7 6 5 4 3 2 1
58 ! _____ Peale vahetusi: _____
59 ! a  = 4 3 2 1
60 ! b  = 1 2 3 4
61 ! aa = 9 8 7 6 5 4 3 2 1
62 ! bb = 1 2 3 4 5 6 7 8 9

```

Toome siin veel sellesama ülesande lahenduse, kuid kirjutatuna veidi erinevalt, arvestades polümorfismi võimalusi Fortran9x keeles::

Lähtetekst 4.4: Eelnev programm kirjutatuna arvestades polümorfismi võimalusi Fortran9x-s

```

1 ! Fail: testarrfun2.f90
2 module vahetus2
3   interface vahetaja
4     module procedure vahetaja1 , vahetaja2
5   end interface
6 contains
7   subroutine vahetaja1(a1,b1)
8     implicit none
9     integer , dimension(:) :: a1,b1

```



```

10     integer , dimension( size( a1 ) ) :: work1
11     work1=a1
12     a1=b1
13     b1=work1
14 end subroutine vahetaja1
15 subroutine vahetaja2( a2 , b2 )
16     implicit none
17     integer , dimension( : , : ) :: a2 , b2
18     integer , dimension( : , : ) , allocatable :: work2
19     allocate( work2( size( a2 , 1 ) , size( a2 , 2 ) ) )
20     work2=a2
21     a2=b2
22     b2=work2
23     deallocate( work2 )
24 end subroutine vahetaja2
25 ! Et täiesti universaalset protseduuri saada , tuleks jätkata :
26 ! subroutine vahetaja3
27 ! . . .
28 ! jne. kuni vahetaja7
29 end module vahetus2
30
31 program testarrfun2
32 use vahetus2
33 implicit none
34 integer , dimension( 4 ) :: a , b
35 integer , dimension( 3 , 3 ) :: aa , bb
36 integer , dimension( 2 ) , parameter :: d=shape( bb )
37 integer :: i
38 a=( / ( i , i=1 , size( a ) ) / )
39 b=( / ( i , i=size( a ) , 1 , -1 ) / )
40 print * , '_____Massiivid_____algul_____ ,
41 print * , 'a_____ , a
42 print * , 'b_____ , b
43 aa=reshape( ( / ( i , i=1 , size( aa ) ) / ) , d )
44 bb=reshape( ( / ( i , i=size( bb ) , 1 , -1 ) / ) , d )
45 print * , 'aa_____ , aa
46 print * , 'bb_____ , bb
47 call vahetaja( a , b )
48 print * , '_____Peale_____vahetusi : _____ ,
49 print * , 'a_____ , a
50 print * , 'b_____ , b
51 call vahetaja( aa , bb )
52 print * , 'aa_____ , aa
53 print * , 'bb_____ , bb
54 end program testarrfun2
55 ! Programm väljastab ( SUN ) :
56 ! _____ Massiivid algul : _____
57 ! a = 1 2 3 4
58 ! b = 4 3 2 1
59 ! aa = 1 2 3 4 5 6 7 8 9
60 ! bb = 9 8 7 6 5 4 3 2 1
61 ! _____ Peale vahetusi : _____
62 ! a = 4 3 2 1
63 ! b = 1 2 3 4
64 ! aa = 9 8 7 6 5 4 3 2 1
65 ! bb = 1 2 3 4 5 6 7 8 9

```

4.7.2 Kujumuutusoperatsioonid

Eepooltõudud näites 4.1 kasutasime juba sisefunktsiooni `reshape`, mis transformeerib massiivi ühest kujust teise. Sellised funktsioonid on veel:

Funktsioon `spread(A,dim,ntk)` tekitab massiivist `A` `ntk` koopiat suunas `dim`. Näiteks, juhul kui `A=(/7,4/)`, siis

$$\text{spread}(A,2,3) = \begin{pmatrix} 7 & 7 & 7 \\ 4 & 4 & 4 \end{pmatrix}; \quad \text{spread}(A,1,3) = \begin{pmatrix} 7 & 4 \\ 7 & 4 \\ 7 & 4 \end{pmatrix}.$$

Funktsioon `pack(B,mask[,vektor])` pakib suvalise kujuga massiivi ühedimensionaalsesse massiivi kasutades loogilist massiivi `mask`. Juhul kui vektor on antud, siis tulemusvektori pikkuseks on `vektor`-i pikkus, vastasel juhul on pikkuseks väärtuste `.true.` arv massiivis `mask`. Näiteks, juhul kui

$$\text{mask} = \begin{pmatrix} T & F & T \\ F & T & F \end{pmatrix}$$

ja

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix},$$

siis `pack(A,mask)` on `(/1,3,5/)`; `pack(A,mask,(/7,8,9,10/))` on `(/1,3,5,10/)`. Viimasel juhul võetakse tulemusvektoriks `vektor`, milles on esimesed kolm komponenti asendatud.

4.7.3 Vektorite ja maatriksite korrutamine

Juhul kui meil on antud kaks sarnase kujuga massiivi `A` ja `B`, siis `A*B` tähendab nende elementviisilist korrutamist. Kui me tahame aga leida `A` ja `B` korrutist maatriksite korrutamise mõttes, siis on selleks olemas sisefunktsioon `matmul`. Näiteks,

```
integer :: n,k,m
real    :: A(n,k),B(k,m),C(n,m)
C=matmul(A,B) ! Realiseerib maatriksite korrutamise
```

Tuleb vaid hoolt kanda, et tegurite kujud ühilduksid maatriksite korrutamiseks sobivalt.

Sisefunktsioon `dot_product(A,B)` realiseerib skalaarkorrutise, ehk `matmul` erijuhu juhul kui `A` oleks reavektor (`A(1,n)`) ning `B` veervektor (`B(n,1)`). `Dot_product(A,B)` puhul peavad aga `A` ja `B` olema sama kujuga massiivid (näiteks `A(n),B(n)`).

4.7.4 Kitsendusfunktsioonid

Kitsendusfunktsioonid massiividega on näiteks `sum`, `product`, `count`, `maxval`, `any`,

`all`. Toome siin vaid paar illustreerivat näidet:

```

real :: a(1024), b(4,1024), skaalar
skaalar = sum(a) ! kõigi elementide summa
a = product(b,DIM=1) ! esimeses dimensioonis olevate elementide
! korrutis
skaalar = count(a==0) ! annab nulliliste elementide arvu
skaalar = maxval(a,mask=a<0) ! nullilähedaseim negatiivne element

```

On olemas loogilised kitsendusfunktsioonid nagu näiteks **all** ja **any**:

```

logical :: a(n)
real, dimension(n) :: b, c
if (all(a)) ... ! globaalne .AND.
if (all(b==c)) ... ! tõene juhul kui võrdus kehtib igal pool
if (any(a)) ... ! globaalne .OR.
if (any(b<0.0)) ... ! tõene juhul kui suvaline element < 0.0

```

4.7.5 Asukohafunktsioonid

Asukohafunktsioonide abil saab leida massiivi teatud elemendi positsiooni.

Sisefunktsioonid **maxloc** ja **minloc** väljastavad vastavalt massiivi minimaalse või maksimaalse elemendi positsiooni.

Funktsioon **maxloc(A[,mask])** väljastab ühedimensionaalse massiivi, mille elementide arvuks on massiivi **A** dimensioonide arv. Selle väärtuseks on indeksite loetelu, mis määrab ära massiivi **A** maksimaalse elemendi asukoha (nende elementide hulgast kus **mask** väärtus on **.true.** juhul kui **mask** antud on). Funktsioon **minloc** leiab analoogselt minimaalse väärtuse.

4.7.6 Massiivi muutmise funktsioonid

Sellesse kategooriasse kuuluvad sisefunktsioonid **cshift**, **eoshift** ja **transpose**.

Funktsioon **transpose(maatriks)** tagastab 2-dimensionaalse massiivi **maatriks** transponeeritud, st. tulemuses on element **(i,j)** asendatud elemendiga **(j,i)**.

Funktsioon **cshift(A,shift,dim)** tagastab sama tüüpi ja kujuga massiivi nagu **A**, kuid nihutab tsükliliselt maatriksit **shift** sammu suunas **dim**.

Funktsioon **eoshift(A,shift,dim[,boundary])** töötab sarnaselt funktsiooniga **cshift**, kuid tsirkulatsiooni ei toimu. Tühjad kohad asendatakse nullilise väärtusega või parameetri **boundary** väärtusega selle olemasolu korral.

Funktsioonide **cshift** ja **eoshift** kasutamise kohta toome järgneva näite:

Lähtetekst 4.5: Sisefunktsioonide **cshift** ja **eoshift** kasutamine.

```

1 program cshift_test
2   interface
3     subroutine valjasta_maatriks(a)
4       integer :: A(:,:)
5     end subroutine valjasta_maatriks
6   end interface
7   integer, dimension(4,5) :: G,F
8   integer :: i,j
9   G = reshape( (/ (i,i=1,size(G) /) , shape(G) )
10  print *, 'maatriks G on:'

```

```

11  call valjasta_maatriks(G)
12  F = cshift(G, shift=-1, dim=1)
13  print *, 'maatriks_cshift(G)_on: '
14  call valjasta_maatriks(F)
15  F = eoshift(G, shift=-2, dim=2, boundary=88)
16  print *, 'maatriks_eoshift(G)_on: '
17  call valjasta_maatriks(F)
18  end program cshift_test
19
20  subroutine valjasta_maatriks(A)
21  integer :: A(:, :), i, j
22  do i=1, size(A, dim=1)
23  print *, (A(i, j), j=1, size(A, dim=2))
24  enddo
25  end subroutine valjasta_maatriks
26  ! Programm väljastab (Intel Fortran kompilaator):
27  ! maatriks G on:
28  !      1      5      9      13      17
29  !      2      6     10     14     18
30  !      3      7     11     15     19
31  !      4      8     12     16     20
32  ! maatriks cshift(G) on:
33  !      4      8     12     16     20
34  !      1      5      9     13     17
35  !      2      6     10     14     18
36  !      3      7     11     15     19
37  ! maatriks eoshift(G) on:
38  !     88     88      1      5      9
39  !     88     88      2      6     10
40  !     88     88      3      7     11
41  !     88     88      4      8     12

```

4.7.7 WHERE - direktiiv

Operatsioone saab kitsendada vaid teatud massiivi elementidele **where**-direktiiviga. See on ühetasemeline käsk, (uus **where** ei tohi välimises **where**-blokis esineda). Sisuliselt nagu **if**-käsk kus argumendiks on loogiline massiiv. Olemas on ka **else where** klausel:

```

real, dimension(7,7) :: a, b
!...
where (a/=0) b=1/a
!...
where (a>2.0)
  b = a*9.0
else where
  b = a/3.0
end where

```

4.8 Sisend-väljund

Anname siin vaid mõningaid kasulikke näpunäiteid Fortran9x sisendi- ja väljundioperatsioonide kohta. Detailsemaks tutvuseks soovitame lugeda Fortran9x kirjandust.

4.8.1 Formaadikirjeldused

Sisestamiseks on käsk `read`, väljastamiseks käsud `print` ja `write`. Nagu eelpooltoodud näidetes näha võis, on kõige lihtsamaks väljastamismooduseks käsk `print *`, sisestamiseks aga käsk `read`, näiteks:

```
read *,muutuja
print *, 'muutuja=',muutuja [, 'muutuja2=' ] [ , muutuja2 ] , ...
```

Seejuures tähistab `'*'` standardväljundit või -sisendit, mis üldjuhul on vastavalt arvuti-ekraan ja klaviatuur. Käsk `print *` on tegelikult pärit Fortran77-st ning on ekvivalentne Fortran9x `write(*,*)`-käsuga (kirjutada standardväljundisse standardses formaadis iga trükitava välja kohta). Käsk `read *` on samas ekvivalentne `read(*,*)`-käsuga. Märgime siin, et see “standardne formaat” on paraku kompilaatoriti erinev. Sellepärast, saavutamaks kompilaatorist sõltumatut väljundikuju, tuleks kasutada formaadikirjeldust `format`.

`Format`-kirjeldusega saab ette anda `read(*,*)` ja `write(*,*)` käsule väljastatava või sisestatava väärtuse kuju asendades teise `'*'` vastava kirjega. `Format`-kirje võib olla defineeritud neljal erineval viisil:

1. Parameetrina otse käsus endas, formaadikirje peab olema ümbritsetud sulgudega. Näiteks:

```
write (*, '(I5 ,F10.2) ') ... muutujad ...
```

2. `Format`-kirje võib olla defineeritud eraldi parameetrina:

```
character (len=*) , parameter :: FMT1 = "(I5 ,F10.2) "
write (*,FMT1) ... muutujad ...
```

3. `Format`-kirje saab defineerida ka string-muutujas:

```
character (len=20) :: FMT1 = "(I5 ,F10.2) "
write (*,FMT1) ... muutujad ...
```

4. Võimalik on ka eraldiseisev `format`-direktiiv:

```
write (*,10) ... muutujad ...
10 format (I5 ,F10.2)
```

Viimast kuju kasutatakse juhul, kui mitmes eri kohas on vaja samas formaadis midagi väljastada ning üldiselt keerukamatel juhtudel.

Näide:

```

! Variant 1:
character(len=*),parameter :: fmt='(2i3 ,i9 ,A11) '
write (*,fmt) 1,2,3, '_on_numbrid '
! Variant 2:
write (*,'(2i3 ,i9 ,A11) ') 1,2,3, '_on_numbrid '
! Variant 3:
write (*,'(2i3 ,i9 ,A) ') 1,2,3, '_on_numbrid '
! Variant 4:
write (*,10) 1,2,3
10 format(2i3 ,i9 , '_on_numbrid ')
! väljastavad kõik sama tulemuse:
! 1 2      3 on numbrid

```

Loetleme siin mõningate formaadisümbolite tähendusi:

Formaadisümbol **i** tähistab täisarvu. Sellele peab järgnema täisarvu kohtade arv, näiteks **i4** tähendab, et tuleb trükkida täisarv nelja sümboli pikkusele väljale. Kui tegelik väljastatava arvu kohtade arv on väiksem kui 4, siis lisatakse ette tühikud.

Igale kirjeldusele võib ette kirjutada korduste arvu. Näiteks, **3i5** tähendab, et trükitakse 3 viiekohalist täisarvu.

Sümbol **f** on ujukomaarvu tähis. Kirje **f10.2** tähendab, et trükitava arvu kogupikkus on maksimaalselt 10 ja 2 kohta on peale koma. NB! kui arv on negatiivne, siis ka miinusmärk võtab ühe koha (nii nagu ka “.”).

Näiteks:

```
write (*,'(f10.2) ') -1234567.89
```

annab väljundis vea, selle asemel tuleks kirjutada:

```
write (*,'(f11.2) ') -1234567.89
```

Sümboliga **e** tähistatakse ujukomaarvu eksponentsiaalsel kujul, näiteks:

```
write (*,'(e10.2) ') -1234567.89 ! annab: -0.12E+07
```

Formaadisümbol **a** ütleb, et tegemist on kirjega.

Juhul kui on vaja midagi lugeda või kirjutada ilma reavahetusega, võib lisada atribuudi **advance='no'**. Näiteks:

```

do i=1,5
  write (*,'(A,I2) ',ADVANCE='NO') '_i_=',i
enddo
write (*,'(A) ') '_rea_lõpp.'
! väljastab:
! i = 1 i = 2 i = 3 i = 4 i = 5 rea lõpp.

```

4.8.2 Failitöötlus

Seni oli toodud näidetes **write** käskudes esimeseks parameetriks (ehk välisseadme tunnuseks) olnud ***** mis on vaikimisi number **6** ehk käsuaken. Selleks parameetriks saab võtta aga ka näiteks stringimuutuja. Näiteks:

```
character(len=80) :: kirje
write (kirje, '(e15.9)') -1234567.89
write (*,*) 'kirje on: ', kirje
```

annab tulemuseks:

```
kirje on: -.123456787E+07
```

Välisseadmeks võib aga defineerida hoopis faili, avades selle eelnevalt `open`-käsuga, nagu näiteks:

```
open(2, file='failinimi.txt')
write (2,*) 'Faili kirjutamine'
close(2)
```

mis loob faili nimega `failinimi.txt` mille sisuks saab:

```
Faili kirjutamine
```

(NB! Juhul kui selline fail juba eksisteerib, kirjutatakse see üle (nii et parem on mitte võtta failinimeks `minu_dissertatsioon.doc`, kui selline juhtub eksisteerima ja on ainus koopia).)

Failist lugemine toimub sarnaselt:

```
open(3, FILE='failinimi.txt')
read (3, '(80A)') kirje
close(3)
```

Käsul `open` on palju atribuute, millest vaid mõnda olulisemat siin lähemalt käsitleme (mõnda ülejäänut vaid mainime ning vajaduse korral tuleks manuaalidest ise täpsemalt nende kohta lugeda):

- Atribuut `FORM`. Väärtusteks kas `FORMATTED` (vaikeväärtus) või `UNFORMATTED`. `FORMATTED` tähendab, et faili puhul on tegemist tekstifailiga, `UNFORMATTED` puhul tehakse operatsioone binaarsel kujul, mis on üldjuhul palju kiirem. (Kui tegemist on binaarsete failidega, siis peab arvestama muuhulgas, et eri arvutiarhitektuurid salvestavad 4- või 8-baidilisi sõnu erinevalt. Enimlevinud on nn. *little endian* (näiteks Inteli protsessorid) ja *big endian* (näiteks SUN) formaadid. Seega, juhul kui fail on salvestatud binaarselt üht tüüpi arvutil, ei pruugi seda saada niisama lihtsalt sisse lugeda teist tüüpi arvutiarhitektuuril.)

Näide käsu `open` atribuudi `FORM` kasutamisest:

```
open(3, FILE='failinimi.dat', FORM='UNFORMATTED')
```

- Atribuut `ACTION`. Väärtusteks kas `READWRITE` (vaikimisi), `READ`, või `WRITE`. Väljendab eesmärki, mis antud failiga on plaanis peale avamist. Näiteks juhul, kui `ACTION='READ'` kuid proovida faili kirjutada, välistatakse viga

- Atribuut **STATUS**. Väärtusteks kas **OLD** (fail peab avades juba eksisteerima), **NEW** (faili ei tohi avades eksisteerida), **REPLACE** (juhul kui avatav fail juba olemas, siis see asendatakse uuega), **SCRATCH** (fail säilitatakse vaid kuni vastava **close** käsuni või programmi töö lõppemiseni) või **UNKNOWN** (vaikeväärtus, tegelik käitumine sõltub arvutisüsteemist).
- Atribuut **ACCESS** määrab faili võimalike operatsioonide tüübid; väärtusteks kas **SEQUENTIAL** (vaikimisi) või **DIRECT**. (**SEQUENTIAL** puhul lubatud käsud **BACKSPACE** (tagasiliikumine), **REWIND** (algusse liikumine) ja **ENDFILE** (võimalus ise panna faililõpusümbol).)
- Atribuut **POSITION** määrab positsiooni failitöötuse alghetkel. Väärtusteks **ASIS** (vaikeväärtus, juhul kui fail juba eksisteerib ja on avatud, siis positsiooni ei muudeta), **REWIND** (minnakse faili algusse) või **APPEND** (kui fail eksisteerib, siis asutakse selle lõppu).

Üks vajalikke atribuute, mis on lubatud kõigi, **read**, **write**, **open**, **close** korral on **IOSTAT**. Selle abil on võimalik veasituatsioone “kinni püüda”. Demonstreerime seda järgneva programmilõigu abil:

```
character (len=80)  :: kirje , kirje2
integer  :: ios
open(4, FILE='failinimi.txt', ACTION='READ', IOSTAT=ios)
if (ios/=0) then
  write (*,*) 'faili_avamine_lugemiseks_ebannestus'
else
  read (4, '(80a)') kirje
  write (*,*) 'kirje_on:', kirje
  read (4, '(80a)', IOSTAT=ios) kirje2
  if (ios/=0) then ! juhul kui failis pole rohkem ridu:
    print *, 'faili_lpp!'
  else
    write (*,*) '2_kirje_on:', kirje2
  endif
close(4)
endif
```

Osa II

MPI (*The Message Passing Interface*)

Peatükk 5

Kontseptsioon

Paralleelprogrammeerimises on kasutusel kaks põhilist mudelit: **jagatud mäluga multiprotsessori mudel** ning **teatedastusmudel**. Antud õppematerjalis tutvustame lähemalt vaid **teatedastusmudelit**. **Jagatava mälu** (*shared memory*) mudeli puhul on eelduseks ühise mäluruumi olemasolu paralleelselt töötavatel protsessoritel; informatsiooni vahetamine eri protsessidel toimub selles hästiorganiseeritud mälupöördumiste kaudu. Enimkasutatavaks standardiks sedalaadi programmeerimisel on OpenMP standard, kuid rohkem me sellel siin ei peatu. **Teatedastusmudeli** eeliseks tuleb pidada võimalust realiseerida seda ka jagatava mäluga arhitektuuride korral, mis on seega universaalsem. Kirjutades paralleelprogrammi lähtekoodi teatedastusmeetodil saab seda edukalt kasutada lisaks hajussüsteemidele (*distributed systems*) ka jagatud mäluga multiprotsessor-arvutil.

Teatedastusmeetodil on mitmeid realisatsioone. Kaks tuntumat nendest on PVM (*Parallel Virtual Machine*) ja MPI (Message Passing Interface). Esimene neist, PVM, sai tuntuks veidi varem tänu tasuta kättesaadava PVM-teegi olemasolule. PVM on üks konkreetne projekt, samas üsna edukas, nii et paljud kasutavad seda endiselt. MPI on aga vastukaaluks kujunenud tegelikult standardiks (MPI, nn MPI-1 standard aastast 1992, MPI-2 standard aastast 1998), millel leidub mitmeid realisatsioone. Populaarsemad realisatsioonidest on MPICH (*MPI Chameleon Implementation*) ja LAM-MPI, mis on mõlemad tasuta kopeeritava lähtekoodiga. Lisaks leidub mitmeid arhitektuuri-põhiseid kommertsiaalseid realisatsioone (SUN-MPI, SGI-MPI, Scali-MPI jne.) Märgime, et tegelikult saab MPI programme käivitada ka näiteks üheprotsessorilisel arvutil – käivitatakse mitu UNIXi protsessi samal masinal, mis omavahel suhtlevad MPI käskude abil sarnaselt mitneprotsessorilise juhuga. Nii saab programme kirjutada ja siluda suvalisel arvutil, millel on MPI installeeritud.

MPI-s kasutatavat teatedastusmudelit võib lühidalt iseloomustada järgnevate omadustega:

- Kõik protsessid on omavahel sõltumatud üksteisest sõltumatu mäluga. Enamus MPI programme kasutab järgnevat struktuuri (kuigi ka mõned teised kujud, nagu näiteks klient-serveri mudel, on võimalikud):
 - Kõigi protsesside juhtimiseks kirjutatakse üks ja sama programm;
 - eri protsessidel on erinevad andmete alamhulgad.

- Kommunikatsioon ehk protsessidevaheline andmetevahetus toimub teadete saatmise ja vastuvõtmise teel.
- Teade sisaldab edastatavaid andmeid ning lisaks teatud informatsiooni:

```

dest - vastuvõtva protsessi ID
srce - saatja ID
tag - teate pealkiri
len - teate pikkus
comm - kommunikaator
type - teatetüüp
buffer - tegelikud andmed mida antud teatega edastatakse

```

Enne, kui saab ja on vajadust üldse mingeid teateid saata, tuleb moodustada paralleelsed andmestruktuurid. Toome näiteks programmilõigu kahe vektori (massiivi) summa arvutamiseks:

```

do i=1,n
  z(i)=x(i)+y(i)
end do

```

Paralleliseerides selle kahel protsessoril, peame eelkõige otsustama, kuidas me soovime protsessorite vahel jagada ära andmed (antud juhul massiivi elemendid)? Oletame, et jagame massiivid pooleks, siis andmete paiknemine on järgmine:

<pre> 1 ! Protsess 0 2 real, dimension(1:n/2) :: x,y,z </pre>	<pre> 1 ! Protsess 1 2 real, dimension(n/2+1:n) :: x,y,z </pre>
---	---

Paralleliseerides näeb eeltoodud programmilõigu tsükkel välja selline:

<pre> 3 ! Protsess 0 4 do i=1,n/2 5 z(i)=x(i)+y(i) 6 end do </pre>	<pre> 1 ! Protsess 1 2 do i=n/2+1,n 3 z(i)=x(i)+y(i) 4 end do </pre>
--	--

Näeme et antud juhul kommunikatsiooni vaja ei ole. Kui aga soovime paralleliseerida järgnevat programmilõiku:

```

1 real(kind=rk) :: a,x(n),y(n)
2 a=0.0_rk
3 do i=1,n
4   a=a+x(i)*y(i)
5 end do

```

kahel protsessoril, saame analoogselt:

<pre> 1 <i>! Protsess 0</i> 2 real(rk) :: a0, x(n/2), y(n/2) 3 a0=0.0_rk 4 do i=1,n/2 5 a0=a0+x(i)*y(i) 6 end do 7 a = a0+a1 </pre>	<pre> <i>! Protsess 1</i> real(rk) :: a1, x(n/2+1:n), y(n/2+1:n) a1=0.0_rk do i=n/2+1,n a1=a1+x(i)*y(i) end do a = a0+a1 </pre>
--	--

Näeme, et real number 7 olevate omistamisoperatsioonide teostamiseks on tarvis kommunikatsiooni. Antud näidetes oli paralleelsete andmestruktuuride loomine lihtne. Praktikas ettetulevates ülesannetes moodustab paralleelsete andmestruktuuride ülesehitus tihti mahukaima osa. Me veendume selles ka ise käesoleva õppematerjali viimases peatükis esitatava veidi suurema, reaalsest elust toodud näite varal.

Järgnevalt vaatleme, kuidas protsessidevahelist kommunikatsiooni teostada kasutades MPI võimalusi.

Peatükk 6

MPI kuus põhikäsku

MPI koosneb spetsiaalsetest andmestruktuuridest (nagu näiteks tüübidefinitsioonid, kommunikaatorid (mis kujutavad endast teatud protsesside alamhulka), eeldefineeritud konstantidest jms.) ning MPI käskudest. Kui lugeda programmiteksti, mis kasutab MPI-d siis eristuvad MPI-ga seotud direktiivid tekstis eesliidese `MPI_` kasutamisega, mis tähendab, et tegu on MPI käsu või andmestruktuuriga.

Võib tunda huvi, kui suur on MPI standard? Esmapilgul tundubki ta üsna mahukas sisaldades päris suure arvu käskusid. Samas võib öelda, et MPI on lihtne ja väike, kuna piisab vaid mõne loetud põhikäsu teadmisesest.

Selleks, et kirjutada valmis paralleelprogramm, piisab põhimõtteliselt vaid kuuest MPI käsust. Loetleme siin need käsud, (käskudest täpsemalt veidi hiljem):

1. `MPI_Init` – MPI andmestruktuuride initsialiseerimiskäsk. Alati esimene MPI-käsk suvalises MPI programmis.
2. `MPI_Comm_Size` – Selle käsu abil saab teha päringu teiste protsesside arvu kohta antud paralleelprogrammis või protsesside alamhulgas..
3. `MPI_Comm_Rank` – Päring, mille abil saab protsess teada oma järgu ehk ID antud grupis.
4. `MPI_Send` – Teate saatmine.
5. `MPI_Receive` – Teate vastuvõtmine.
6. `MPI_Finalize` – MPI andmestruktuuride destruktor, töö lõpetamine. Viimane MPI-käsk MPI-programmis.

Suur osa paralleelprogrammide funktsionaalsusest on toodud kuues käsus juba olemas. Paljud teised käsud baseeruvad nendel põhikäskudel või on nende modifikatsioonid.

Järgnevalt meie “hello world” paralleelprogramm:

Lähtetekst 6.1: Lihtne MPI teatesaatmise -vastuvõtu programm

```
1 ! Fail: mpi_tervitus.f90
2 program mpi_tervitus
3   use mpi ! MPI-andmestruktuuride konstantide interfeiside moodul
```



```

4  integer :: minuid, proarv ! MinuID ja PROtsessideARV
5  integer :: pikkus ! teate pikkus
6  parameter (pikkus=MPLMAX_PROCESSOR_NAME+1) ! MPI eeldef.-d konst.
7  character*(pikkus) :: nimi, charpuhver
8  integer :: stat(MPLSTATUS_SIZE), ierr, i, dest, tag, tegpikk
9  ! ----- Ettevalmistav osa -----
10 call MPI_Init(ierr) ! Initsialiseerimine. Vea korral ierr 0-st erinev
11 ! MPICOMM_WORLD: Eeldefineeritud kommunikaator kuhu kuuluvad kõik
12 ! protsessid mis antud programmi käivitasid
13 call MPI_Comm_size(MPICOMM_WORLD, proarv, ierr) ! Protsesside arv?
14 call MPI_Comm_rank(MPICOMM_WORLD, minuid, ierr) ! MinuID päring
15 ! (0 <= minuid < proarv)
16 call MPI_Get_processor_name(nimi, tegpikk, ierr) ! Protsessori
17 ! nimepäring (protsessori või tööjaama süsteemne nimi, kus
18 ! antud protsess jookseb (ebaolulisi MPI abikäske) )
19 ! ----- Ettevalmistava osa lõpp -----
20 if (minuid/=0) then
21   dest = 0 ! sihtprotsessi järk
22   tag = minuid ! Teate
23   ! Saata teade pealkirjaga "tag" protsessile järguga "dest"
24   ! kommunikaatoris "MPICOMM_WORLD", mille sisuks on andmed
25   ! tüübist "MPL_CHARACTER" algusaadressil "nimi" pikkusega "pikkus"
26   call MPL_Send(nimi, pikkus, MPL_CHARACTER, dest, tag, &
27     MPICOMM_WORLD, ierr)
28 else
29   print *, 'Kokku on ', proarv, ' paralleelset protsessi'
30   print *, 'Põhiprotsess ', minuid, ' jookseb protsessoril ', nimi
31   do i=1, proarv-1
32     ! Võtta vastu teade mis saadetud teele käsuga MPL_Send. Esimesed
33     ! kolm argumenti määravad ära teate aadressi, pikkuse ja tüübi
34     ! teate sisu salvestamiseks. MPLANY_SOURCE ja MPLANY_TAG on
35     ! jokkerväärtused saatja-protsessi järgu ja teate TAGi jaoks.
36     ! (võiks asendada mõlemad arvuga i antud juhul)
37     ! Seitsmes argument "stat" on massiiv pikkusega MPLSTATUS_SIZE
38     ! mille abil saab teha päringuid saabunud teate kohta.
39     call MPL_Recv(charpuhver, pikkus, MPL_CHARACTER, MPLANY_SOURCE, &
40       MPLANY_TAG, MPICOMM_WORLD, stat, ierr)
41     print *, 'Protsess ', stat(MPL_SOURCE), ' &
42       ' jookseb protsessoril ', charpuhver
43   end do
44 endif
45 ! ----- Teadete lõpp -----
46 call MPI_Finalize(ierr) ! (Lõpetab MPI töö, vabastab mälu, tühistab
47 ! saabumata teadete järjekorra jms.)
48 stop
49 end program mpi_tervitus

```

6.1 MPI konstruktor ja destruktor

Selleks, et programmis saaks kasutada MPI teegi käskude, tuleb kompilaatorile kuidagi teatavaks teha, kust neid leida. Keele C korral on selleks käsk

```
#include "mpi.h"
```

Fortran77 korral saab MPI-teadlikkuse programmile lisada käsuga:

```
include 'mpif.h'
```

Keele Fortran9x jaoks on aga üldjuhul kõik MPI definitsioonid, konstandid, andmestruktuurid ja interfeisid antud moodulis nimega `mpi` mille kaasamiseks tuleb lisada rida:

```
use mpi
```

Kõige esimene MPI käsk, mis programmis on `MPI_Init`, on defineeritud kujul:

```
subroutine MPI_Init(ierr)
integer, intent(out) :: ierr
```

`MPI_Init` tuleb alati välja kutsuda enne kõiki teisi MPI käske, juhul kui MPI initialiseerimine õnnestus, siis täisarvuline väljundparameeter `ierr` saab väärtuseks konstandi `MPI_SUCCESS`, vastasel korral saab `ierr` väärtuseks veakoodi, mis on sõltuv realisatsioonist. (Praktiliselt kõikidel MPI käskudel on Fortrani puhul viimaseks parameetriks `ierr`. C-keele korral toimub aga MPI-käskude poole pöördumine sarnaselt käsuga: `ierr=MPI_Init()`.)

Vastupidiselt `MPI_Init` käsule, peab käsk `MPI_Finalize` defineerituna kujul

```
subroutine MPI_Finalize(ierr)
integer, intent(out) :: ierr
```

ilmuma peale kõiki teisi MPI käske; `MPI_Finalize` lõpetab kõik pooleliolevad kommunikatsioonid, vabastab mälu jne. Seega võib seda nimetada MPI destruktoriks.

6.2 Põhipäringud

Käsk `MPI_Comm_Rank` kujul

```
subroutine MPICommRank(komm, jark, ierr)
integer, intent(in) :: komm !näiteks MPICOMM_WORLD
integer, intent(out) :: jark, ierr
```

tagastab muutujas `rank` väljakutsuva protsessi ID täisarvuna vahemikus `0` kuni `proarv-1`, kus `proarv` on protsesside arv antud kommunikaatoris. MPI kommunikaator on viide paralleelprogrammis osalevate protsesside teatud (alam)hulgale. Fortrani puhul on selleks viiteks teatud täisarvuline väärtus. Kommunikaator, millele viitab konstant `MPI_COMM_WORLD` ühendab endas kõiki protsesse, mis algselt käivitati ühtse MPI programmina.

Käsu `MPI_Comm_Size` abil saab teada protsesside koguarvu toodud kommunikaatoris; alamprogramm on kujul:

```
subroutine MPICommSize(komm, proarv, ierr)
integer, intent(in) :: komm !näiteks MPICOMM_WORLD
integer, intent(out) :: proarv, ierr
```

6.3 Teadete saatmine ja vastuvõtmine

Teate saatmiseks on käsk `MPI_Send`:

```
subroutine MPLSend( buffer , len , <MPItype> , dest , tag , comm , ierr )
integer , intent ( in ) :: len , <MPItype>
<type> , intent ( in ) :: buffer ( len )
integer , intent ( in ) :: dest , tag , comm
integer , intent ( out ) :: ierr
```

Käsuga saadetakse teade protsessile järguga `dest`, mille sisuks on massiiv mälu algusaadressilt `buffer` andmetüübist `<MPI_type>` elementide arvuga `len`. Juhul kui `len=1`, siis võib `buffer` olla ka lihtmuutuja tüübist `<type>`. Teate identifitseerimiseks lisatakse sellele `tag` – positiivne täisarv.

Me kirjutame `<MPI_type>` nurksulgudes `<...>` seetõttu, et see tuleb tegelikult asendada ühe täisarvulise MPI poolt eeldefineeritud tüübikonstandiga vastavalt tegelikule muutujatüübile või eritähendust omavaga MPI tüübikonstandiga. `<MPI_type>` võib näiteks olla üks eeldefineeritud tüüpidest:

<code><MPI_type></code>	Andmetüüp
<code>MPI_CHARACTER</code>	sümboltüüp
<code>MPI_INTEGER</code>	täisarvutüüp
<code>MPI_LOGICAL</code>	Fortrani <code>LOGICAL</code> -tüüpi
<code>MPI_REAL</code>	ujukomaarvutüüp
<code>MPI_DOUBLE_PRECISION</code> , ka <code>MPI_REAL8</code>	topelttäpsusega ujukomaarvutüüp
<code>MPI_COMPLEX</code>	kompleksarvu tüüp
<code>MPI_DOUBLE_COMPLEX</code>	topelttäpsusega kompleksarvutüüp

Lisaks on olemas ka mehhanism isedefineeritud tüüptide loomiseks, mis võimaldab luua struktuurseid andmekooslusi või kasutada tüüpi `MPI_PACKED`, mille korral saab spetsiaalsete käskude abil “pakkida” ka erinevat tüüpi muutujate või massiivide väärtusi kokku ühte teatesse.

Teate vastuvõtmiseks on käsk `MPI_Recv`:

```
call MPLRecv( buffer , len , <MPItype> , srce , tag , comm , status , ierr )
integer , intent ( in ) :: len , <MPItype>
integer , intent ( in ) :: srce , tag , comm
<type> , intent ( out ) :: buffer ( len )
integer , intent ( out ) :: status ( MPLSTATUS_SIZE )
integer , intent ( out ) :: ierr
```

Siin on parameetrite `buffer`, `len` ja `<MPI_type>` tähendus sama, mis käsu `MPI_Send` korral eelpool. Sealjuures, teate saatja järguks on `srce`. Selleks võib kasutada ka nn. jokerit `MPI_ANY_SOURCE`; sarnaselt võib `tag` olla `MPI_ANY_TAG`. Parameetri `status(MPI_STATUS_SIZE)` abil saab teha päringu saamaks teada saabunud teate tegeliku saatja, `tag` väärtuse, teate pikkuse (vt. programminäidet 6.1, rida 41).

6.4 MPI-programmide kompileerimine ja käivitamine

Tavaliselt on **MPI-programmide kompileerimiseks** olemas vastav skript või programm, mis automaatselt annab põhikompilaatorile ette vajalikud teed päisefailidele ning teekide leidmiseks. Näiteks `f95` programmide kompileerimiseks on olemas käsk `mpif95` (`mpf95` SUN arhitektuuril), C-keeles kirjutatud MPI programmide jaoks on vastavalt käsk `mpicc` (ja `mpcc` SUN arvutitel).

Programmi käivitamiseks näiteks 4 protsessoril tuleks anda käsk:

Linux: `mpirun -np 4 a.out`

või: `mpirun -machinefile masinad -np 4 a.out` (MPICH korral)

kus failis `masinad` on loetletud arvutite IP-aadressid millel tahetakse programmi käivitada.

SUN: `mprun -np 4 a.out .` (Kui on soov kasutada kõiki tööjaamu antud klastril, siis `mprun -np 0 a.out.`)

Peatükk 7

Veel MPI käske

Toodud kuue MPI käsu abil saab põhimõtteliselt paralleelprogrammi kirjutatud, kuid enamus MPI programme kasutavad siiski mõned korrad rohkemat arvu erinevaid MPI käske. Paljud neist muudavad protsessidevahelise kommunikatsiooni lihtsamaks ja mugavamaks lähtuvalt konkreetsest situatsioonist ning eesmärkidest. Alljärgnevas esituses püüame lähendada järgnevast MPI käskude liigitusest.

7.1 MPI käskude liigitus

MPI käsud võib jagada eri rühmadesse vastavalt sellele, kas need on **kahe protsessi vahelised** (*point2point*) käsud, **ühiskäsud** ehk **kollektiivkäsud**; **blokeerivad** või **mitteblokeerivad**. Kui käsu väljakutse piirkond on terve kommunikaator, näiteks `MPI_COMM_WORLD` (vt. alapunkti 6.2), siis on tegemist ühiskommunikatsiooni (*collective communication*) käsuga. Sellisel juhul peavad kõik protsessid antud kommunikaatoris selle käsu välja kutsuma. On olemas mehhanismid alamhulkade loomiseks etteantud kommunikaatoril, st., alamkommunikaatorite tekitamiseks, millel siis on võimalik ühiskäske täitvate protsesside hulka vajaduse järgi kujundada. Kõik ühiskommunikatsiooni käsud on **blokeerivad**, st., et ühegi teise protsessi töö ei saa jätkuda enne, kui kõik kommunikaatori protsessid on vastava käsu jõudnud. Ka kahe protsessi vaheline kommunikatsioon võib olla blokeeriv, aga ka **mitteblokeeriv**. Näites 6.1 toodud teate saatmise ja vastuvõtu käsud olid blokeerivad. Efektivsema paralleelprogrammi saavutamiseks on aga tihti käepärasem kasutada mitteblokeerivat teate saatmist ja vastuvõtmist millel peatume lähemalt peatükis 8.

7.2 Peamisi ühisoperatsioone

Tihti esineb paralleelprogrammeerimisel situatsioone, kus on vaja kõigi kommunikaatori protsesside sünkronisatsiooni. Vajalik on see siis, kui peab olema kindel, et kõik protsessid on oma töö käigus jõudnud teatud kindla verstapostini. Selliseks käsuks on

```
MPI_Barrier(comm, ierr)
```

Käsku `MPI_Barrier` kasutatakse kõigi kommunikaatori `comm` protsesside sünkroniseerimi-

seks. Tegelikult on nii, et mida vähem sünkronisatsioonipunkte paralleelprogrammis leidub, seda efektiivsemaid programme saab kirjutada. Kuid alati ilma läbi ei saa, eriti programmide silumisel. Käsk `MPI_Barrier` on aga kasulik vaid seal, kus mingit andmetevahetust vaja ei lähe ja tähtis on vaid sünkronisatsioon.

Käsuga

```
MPI_Bcast( buffer , len , <MPItype> , root , comm , ierr )
```

saadab protsess järguga `root` kõigile protsessidele kommunikaatoris `comm` andmed `<MPI_type>`-tüüpi massiivist pikkusega `len` aadressilt `buffer`. Tegemist on blokeeriva käsuga!

Käsk

```
MPIReduce( sndbuf , rcvbuf , len , <MPItype> , <MPIop> , root , comm , ierr )
```

kogub kõigi protsesside `sndbuf`-väärtused kokku protsessile `root` sooritades ühtlasi operatsiooni `<op>`. Sooritatavaks operatsiooniks `<MPI_op>` võib olla:

MPIop	Operatsioon (op)
<code>MPI_MAX</code>	Maksimum
<code>MPI_MIN</code>	Miimum
<code>MPI_PROD</code>	Korrutis
<code>MPI_SUM</code>	Summa
<code>MPI_LAND</code>	Loogiline <code>.AND.</code>
<code>MPI_LOR</code>	Loogiline <code>.OR.</code>
<code>MPI_LXOR</code>	Loogiline <code>.XOR.</code>
<code>MPI_BAND</code>	Bitiviisiline <code>.AND.</code>
<code>MPI_BOR</code>	Bitiviisiline <code>.OR.</code>
<code>MPI_BXOR</code>	Bitiviisiline <code>.XOR.</code>

MPI standardis leidub mehhanism ka programmeerija enda poolt defineritavate operatsioonide lisamiseks. Siin sellel lähemalt ei peatu.

Erinevalt järgnevast käsust muutub käsu `MPI_Reduce` korral `rcvbuf` väärtus vaid protsessil järguga `root`.

Käsk

```
MPI_Allreduce( sndbuf , rcvbuf , len , <MPItype> , <MPIop> , comm , ierr )
```

on täiesti sarnane eelmise käsuga selle vahega, et kõik protsessorid saavad tulemuse muutujasse või massiivi `rcvbuf`.

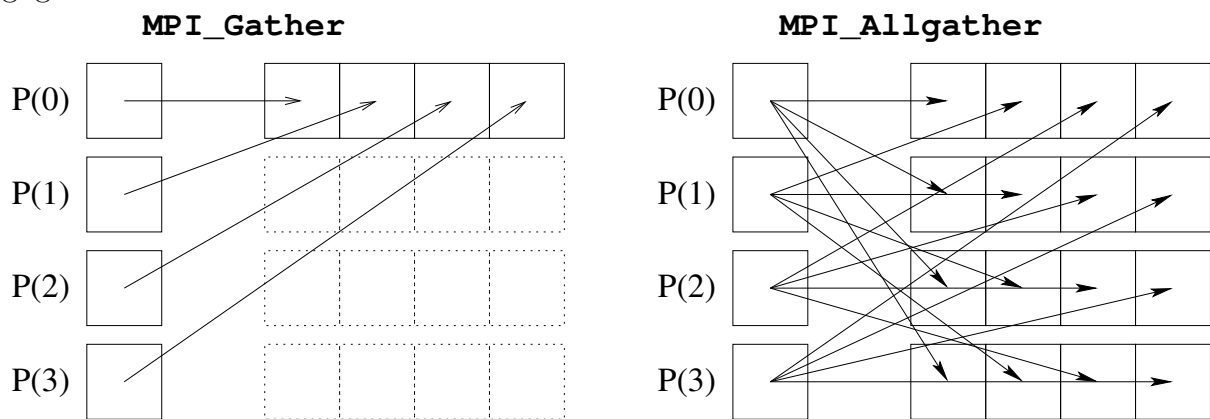
Ka läheb vahel vaja käske `MPI_Gather` ja `MPI_Allgather`, mis on mõeldud väärtuste kokkukorjamiseks eri protsessidelt ühte massiivi. Käsud on järgneva süntaksiga:

```
MPI_Gather( sndbuf , sndlen , <MPIsend_type> , &  
            rcvbuf , rcvlen , <MPIrecv_type> , root , comm , ierr )
```

ja

```
MPI_Allgather(sndbuf, sndlen, <MPI_send_type>, &  
               rcvbuf, recvlen, <MPI_recv_type>, comm, ierr )
```

Sarnaselt `MPI_(All)reduce` käskudega, saab käsu `MPI_Allgather` abil iga kommunikaatori liige omale tulemuse erinevalt käsust `MPI_Gather`, kus tulemuse saab vaid protsessor järguga `root=0`:



Käsuga `MPI_Gather` vastupidiseks operatsiooniks on `MPI_Scatter`:

```
MPI_Scatter(sndbuf, sndlen, <MPI_send_type>, &  
            rcvbuf, recvlen, <MPI_recv_type>, root, comm, ierr )
```

Selle abil saab igale protsessile saata erinevaid andmeid protsessilt järguga `root`.

Lõpetuseks siin veel ühest käsust, mis ei ole ei blokeeriv ei ka kommunikatsiooni teostav, kuid on üsna käepärane mõõtmaks programmi tööaega:

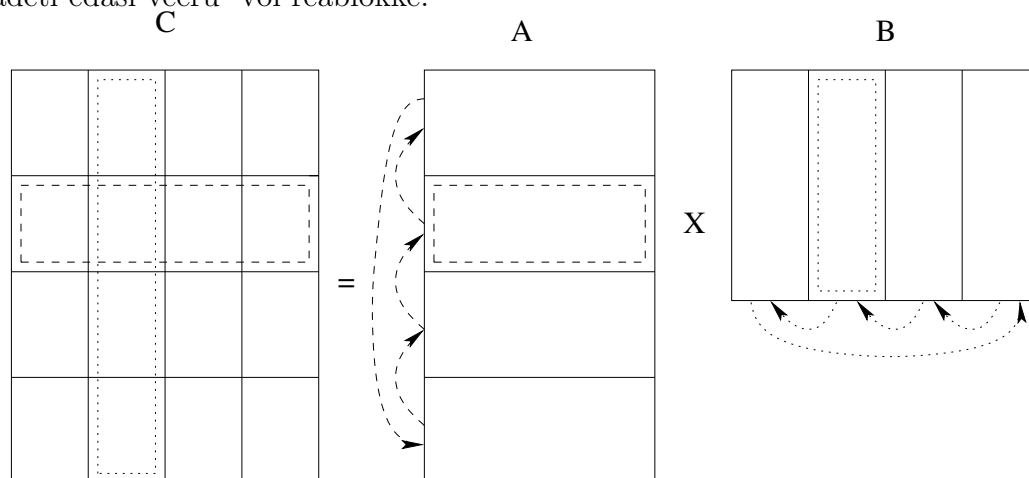
```
real*8  :: t  
t = MPLWtime( )
```

on funktsioon mis annab süsteemse kella väärtuse topelttäpsusega ujukomaarvuna. Tihti annab see täpsema tulemuse kui funktsioon `cpu_time`.

7.3 Näide: Iseorganiseeruv kommunikatsioonimudel

Olgu meil vaja arvutada maatrikskorrutis $C = A \times B$, kus C on $n \times m$ -maatriks, A on $n \times k$ -maatriks ning B on $k \times m$ -maatriks. Selleks leidub mitmeid rakendusvõimalusi. Näiteks võib jagada maatriksi A ridadekaupa P blokiks (n/P rida blokis) ning B jagada veergudekaupa P blokiks (m/P veergu blokis), kus P on protsesside arv. Algselt saab iga protsess järguga i kummastki maatriksist i -nda bloki, arvutab alamblokkide korrutise ning kasutades ringkommunikatsiooni, saadab oma rea- või veerubloki naabrile $i-1$ (kusjuures 0-järku protsess saadab oma bloki protsessile järguga $P-1$). Tulemuseks saame nii maatriksi

C hajutatud kujul vastavalt kas reablokkidena või veerublokkidena sõltuvalt sellest, kas saadeti edasi veeru- või reablokke.



Selline lähenemine on põhimõtteliselt igati mõistlik, eriti juhul, kui kasutada mitteblokeerivat kommunikatsiooni nagu näeme järgnevas peatükis. Ainus probleem võib tekkida olukorras, kus protsessorite jõudlused ei ole mingil põhjusel võrdsed. Sellisel juhul võib kasutada hoopis **iseorganiseeruvat kommunikatsioonimudelit**, kus iga nn. tööprotsess (*slave*) küsib juhtprotsessilt (*master*) uut tööd niipea, kui ta on oma eelmise ülesandega valmis saanud.

Järgnevas näites saadetakse kõigile protsessidele algselt laiali terve maatriks B ning iga tööprotsess saab algselt ühe maatriksi A rea. Selline lähenemisviis erineb eeltoodud skeemist, kuid tagab protsessorite ühtlase tööjaotuse hoolimate igaihe jõudlusest konkreetsel ajahetkel.

Lähtetekst 7.1: Iseorganiseeruva kommunikatsiooniga maatriksite korrutamine

```

1 ! Fail: mpi_mat_korda_mat.f90
2 ! Iseorganiseeruva kommunikatsioonimudeliga näide kahe maatriksi
3 ! maatrikskorrutise leidmiseks.
4 program mpi_mat_korda_mat
5   use mpi
6   use RealKind ! kind-parametri rk definitsioon
7   implicit none
8   integer, parameter :: MAXARIDU=100, MAXAVEERGE=10000, MAXBVEERGE=100
9   real(kind=rk) :: A(MAXARIDU,MAXAVEERGE), B(MAXAVEERGE,MAXBVEERGE)
10  real(kind=rk) :: C(MAXARIDU,MAXBVEERGE)
11  real(kind=rk) :: buffer(MAXAVEERGE), vastus(MAXAVEERGE)
12  real(kind=rk) :: t1, t2
13  integer :: minuid, master, parv, ierr, status(MPLSTATUS_SIZE)
14  integer :: i, j, saadet, saatja, k
15  integer :: vastuse_koht, rida, aridu, aveerge
16  integer :: bridu, bveerge, cridu, cveerge
17  integer :: MPLrk !Ei pruugi ette teada, kas MPLREAL või MPLREALS
18  call MPLINT(ierr) ! MPI saagu
19  call MPLCOMMRANK(MPLCOMM_WORLD, minuid, ierr)
20  call MPLCOMMSIZE(MPLCOMM_WORLD, parv, ierr)
21  print *, "Protsess_", minuid, "_koguarvust_", parv, "_alustab"
22  master = 0
23  aridu = 100
24  aveerge = 10000
25  bridu = 10000

```

```

26  bveerge = 100
27  cridu   = aridu
28  cveerge = bveerge
29  if ( minuid .eq. master ) then
30      print *, 'tüvekohtade_arv_(precision):', precision(1.0_rk)
31      print *, 'eksponendi_ulatus_(range):', range(1.0_rk)
32      do i=1,aveerge ! Lihtsalt mingid väärtused testimiseks...
33          do j=1,aridu
34              A(j,i)=i
35          enddo
36      enddo
37      do i=1,bveerge
38          do j=1,bridu
39              B(j,i)=i
40          enddo
41      enddo
42  endif
43  if ( precision(1.0_rk)>=15) then
44      MPLrk=MPLDOUBLEPRECISION
45  else
46      MPLrk=MPLREAL
47  endif
48  call MPLBARRIER(MPLCOMM_WORLD, ierr)
49  t1=MPLWTime(ierr)
50  if (parv==1) then
51      C=matmul(A,B)
52  else
53      if (minuid.eq.master) then
54          saadet=0
55          ! saadame B kõigile ülejäänud protsessidele:
56          do i = 1, bveerge
57              call MPLBCAST(B(1,i),bridu,MPLrk, master, &
58                          MPLCOMM_WORLD, ierr)
59          enddo
60          ! igale protsessile üks A rida; tag-iks reanumber:
61          do i = 1, parv-1
62              do j = 1, aveerge
63                  buffer(j) = A(i,j)
64              enddo
65              call MPLSEND(buffer, aveerge, MPLrk, i, &
66                          i, MPLCOMM_WORLD, ierr)
67              saadet=saadet+1
68          enddo
69          do i=1,cridu
70              call MPLRECV(vastus, cveerge, MPLrk, MPLANY_SOURCE, &
71                          MPLANY_TAG, MPLCOMM_WORLD, status, ierr)
72              saatja      = status(MPLSOURCE)
73              vastuse_koht = status(MPLTAG)
74              do j=1,cveerge
75                  C(vastuse_koht, j)=vastus(j)
76              enddo
77              if (saadet<aridu) then
78                  do j=1,aveerge
79                      buffer(j)=A(saadet+1,j)
80                  enddo
81                  call MPLSEND(buffer, aveerge, MPLrk, &
82                              saatja, saadet+1, MPLCOMM_WORLD, ierr)
83                  saadet=saadet+1
84              else ! teade töö lõpetamiseks:

```

```

85         call MPLSEND(1.0,1,MPLrk,saatja, &
86                     0,MPICOMM_WORLD,ierr)
87     endif
88 enddo
89     !Vastuse väljatrükk:
90     do i = 1,cridu;do j = 1,cveerge
91         print *, "C(", i, ", ", j, ") = ", C(i,j)
92     enddo;enddo
93 else
94     ! võtta vastu B, seejärel arvutada C ridu kuni lõputeade
95     do i=1,bveerge
96         call MPLBCAST(B(1,i),bridu,MPLrk, &
97                     master,MPICOMM_WORLD, ierr)
98     enddo
99     call MPLRECV(buffer,aveerge,MPLrk,master, &
100                MPLANY_TAG,MPICOMM_WORLD,status,ierr)
101     do while (status(MPLTAG).ne.0)
102         rida=status(MPLTAG)
103         do i=1,bveerge
104             vastus(i)=0.0
105             do j=1,aveerge
106                 vastus(i)=vastus(i)+buffer(j)*B(j,i)
107             enddo
108         enddo
109         call MPLSEND(vastus,bveerge,MPLrk,master, &
110                    rida,MPICOMM_WORLD,ierr)
111         call MPLRECV(buffer,aveerge,MPLrk,master, &
112                    MPLANY_TAG,MPICOMM_WORLD,status,ierr)
113     enddo
114     endif
115 endif
116 call MPLBARRIER(MPICOMM_WORLD,ierr) ! veendumaks et kõik siin
117 t2=MPLWTime(ierr)
118 if (minuid.eq.0) then
119     write(*,*) 'Ajakulu:',t2-t1
120 endif
121 call MPLBARRIER(MPICOMM_WORLD,ierr)
122 call MPLFINALIZE(ierr) ! MPI lõpetamine
123 stop
124 end program mpi_mat_korda_mat
125 ! Kompileerimiseks SUNil:
126 ! mpf95 -dalign -c RealKind.f90;mpf95 -dalign mpi_mat_korda_mat.f90 -
    lmpi

```

Peatükk 8

Mitteblokeeriv kommunikatsioon

Senivaadeldud MPI käsud olid blokeerivad, st. programm jääb käsku täites ootele kuni vastav MPI käsk on lõpetanud. Kuna kommunikatsiooniooperatsioonid on üldjuhul palju aeganõudvamad kui näiteks mäloperatsioonid, siis tähendab see tihti tühja ootamist mõne teate saabumise või ärasaatmise taga selle asemel, et näiteks sooritada mõnda vajalikku arvutust. Sellepärast on olemas mitteblokeerivad MPI käsud.

8.1 Mitteblokeerivaid MPI kärke

Tihti on paremaks paralleelsuseks vaja kommunikatsioonist tulenevaid ooteaegu täita kasulike arvutustega. Selline strateegia võimaldab kirjutada paremini skaleeruvaid paralleelprogramme. MPI standardis on olemas mitteblokeerivad kommunikatsioonikäsud nagu näiteks mitteblokeeriv teate saatmisfunktsioon `MPI_Isend`:

```
subroutine MPI_Isend( buffer , len , <MPItype> , dest , tag , comm , request , ierr )
integer , intent ( in ) :: len , <MPItype>
<type> , intent ( in ) :: buffer ( len )
integer , intent ( in ) :: dest , tag , comm
integer , intent ( out ) :: request , ierr
```

ja mitteblokeeriv teate vastuvõtmisfunktsioon `MPI_Irecv`:

```
call MPI_Irecv( buffer , len , <MPItype> , srce , tag , comm , request , ierr )
integer , intent ( in ) :: len , <MPItype>
integer , intent ( in ) :: srce , tag , comm
<type> , intent ( out ) :: buffer ( len )
integer , intent ( out ) :: status ( MPLSTATUS_SIZE )
integer , intent ( out ) :: request , ierr
```

Toodud käsud alustavad kommunikatsiooniooperatsiooniga “tagaplaanil” andes juhtimise kohe üle vahetult järgnevale programmiblokile. Selleks, et nii oleks võimalik korrektselt programmeerida, peab leiduma mehhanism, kuidas kindlaks teha, kas üks või teine mitteblokeeriv kommunikatsiooniooperatsioon on juba lõpetanud või mitte. Kõikidel mitteblokeerivatel käskudel on üks lisaparameeter `request`, mis on sisuliselt viit tagaplaanil

toimuva operatsiooniobjektile. Selle abil saab kontrollida teate staatust. Blokeerivalt saab sellist päringut teha käsuga `MPI_Wait`:

```
call MPIWait(request, status, ierr)
integer, intent(in) :: request
integer, intent(out) :: status(MPLSTATUS_SIZE)
integer, intent(out) :: ierr
```

Programmi töö ei jätku enne, kui vastav kommunikatsioonioperatsioon on lõppenud. Mitteblokeeriva päringu `MPI_Test` korral aga kommunikatsioonioperatsiooni lõppemist ootama ei jääda:

```
call MPITest(request, flag, status, ierr)
integer, intent(in) :: request
logical, intent(out) :: flag
integer, intent(out) :: status(MPLSTATUS_SIZE)
integer, intent(out) :: ierr
```

Juhul, kui kommunikatsioonioperatsioon viidaga `request` on lõppenud, saab loogiline muutuja `flag` väärtuseks `.true.`, vastasel korral on väärtus `.false.`

Mitteblokeerivad käsud on tihti kokkuvõttes kiiremad.

8.2 Ühesuunaline kommunikatsioon kahe protsessori vahel

Oletame, et meil on kaks protsessi järguga `minuid=0` ja `minuid=1` ning meil on vaja saata protsessilt järguga `0` mingi teade protsessile järguga `1`. Seega on meil tegemist ühesuunalise kommunikatsiooniga nende protsesside vahel ning meil on põhimõtteliselt neli erinevat võimalust:

1. Blokeeriv `send` ja blokeeriv `receive`:

```
if (minuid==0) then
  call MPISend(sendbuf, len, <MPLtype>, 1, tag, comm, ierr)
elseif (minuid==1) then
  call MPLRecv(recvbuf, len, <MPLtype>, 0, tag, comm, status, ierr)
endif
```

2. Mitteblokeeriv `send` ja blokeeriv `receive`:

```
if (minuid==0) then
  call MPIIsend(sendbuf, len, <MPLtype>, 1, tag, comm, request, ierr)
  . . .
  call MPIWait(request, status, ierr)
elseif (minuid==1) then
  call MPLRecv(recvbuf, len, <MPLtype>, 0, tag, comm, status, ierr)
endif
```

3. Blokeeriv `send` ja mitteblokeeriv `receive`:

```

if ( minuid==0) then
  call MPLSend(sendbuf , len , <MPLtype> , 1 , tag , comm , ierr )
elseif ( minuid==1) then
  call MPLIrecv(recvbuf , len , <MPLtype> , 0 , tag , comm , request , ierr )
  . . .
  call MPLWait(request , status , ierr )
endif

```

4. Mittelekeeriv `send` ja mittelekeeriv `receive`:

```

if ( minuid==0) then
  call MPLIsend(sendbuf , len , <MPLtype> , 1 , tag , comm , request , ierr )
  . . .
  call MPLWait(request , status , ierr )
elseif ( minuid==1) then
  call MPLIrecv(recvbuf , len , <MPLtype> , 0 , tag , comm , request , ierr )
  . . .
  call MPLWait(request , status , ierr )
endif

```

Käsu `MPI_Wait` võib panna programmis suvalisse kohta peale vastavat mittelekeerivat käsku, kuid enne `sendbuf` (või vastavalt `recvbuf`) järgmist kasutamist. Ühesuunaline kommunikatsioon on lihtne kuid siiski ka juba veaaldis – peab alati hoolt kandma, et igal `receive`-käsul oleks vastav `send`-käsk olemas ja vastupidi. Vastasel korral jääb üks protsessidest “lõpmatuseni” teadet ootama, juhul kui tegemist oli blokeeriva `receive`-käsu; juhul, kui aga kasutada mittelekeerivat `receive`-käsku, ei saa `recvbuf` oodatud väärtust iial kätte.

8.3 Vastastikune kommunikatsioon ja tupikute vältimine

Juhul, kui meil on tegu kahe protsessiga, mis mõlemad tahavad midagi üksteisele saata ja üksteiselt ka teadet samaaegselt vastu saada, on asi juba keerulisem. Järgnevalt demonstreerime erinevaid variante **tupikute** tekkimise võimaluse seisukohalt.

Tupik (*deadlock*) ehk nn. “surnud seis” on olukord, kus protsessorid jäävad üksteise järel ootama ilma, et ükski midagi kasulikku suudaks teha.

Tupikud võivad tekkida:

- `send`- ja `receive`-käsu valest järjekorrast tingituna
- süsteemse teatedastuspuhvri ületäitumisest.

Kahesuunalise kommunikatsiooni puhul on kolm erinevat võimalust:

1. Mõlemad protsessorid alustavad `send`-käsu ja seejärel tuleb `receive`-käsk.
2. Mõlemad protsessorid alustavad `receive`-käsu ja seejärel `send`-käsk.

- Üks protsess alustab `send`-käsuga ja seejärel tuleb `receive`-käsk, teine protsess sooritab need käsud aga vastupidises järjekorras

Tupikute vältimiseks võib kasutada mitteblokeerivaid käske. Sõltuvalt blokeeringu olemasolust tuleneb siit eri võimalusi.

1. `Send`-käsk kõigepealt, seejärel `receive`-käsk:

Vaatleme järgnevat programmilõiku, kus kasutame blokeerivaid kommunikatsioonioperatsioone:

```
if (minuid==0) then
  call MPLSend(sendbuf, len, <MPLtype>, 1, tag, comm, ierr)
  call MPLRecv(recvbuf, len, <MPLtype>, 1, tag, comm, status, ierr)
elseif (minuid==1) then
  call MPLSend(sendbuf, len, <MPLtype>, 0, tag, comm, ierr)
  call MPLRecv(recvbuf, len, <MPLtype>, 0, tag, comm, status, ierr)
endif
```

See töötab probleemideta seni, kuni `sendbuf` pikkus `len` on väiksem kui süsteemi teatedastuspuhver. Vastasel juhul tekib tupik. **Miks?** Probleem on nimelt selles, et `MPI_Send`-käsu korral kantakse `sendbuf` sisu esmalt eraldiseisvasse mälu regiooni, teatedastuspuhvrisse, kust siis seejärel tegelik andmete ülekanne toimub. Juhul, kui `len` on pikem kui see puhver, siis kantakse `sendbuf` sisu puhvrisse osade kaupa – peale seda, kui esimene osa puhvrist on teisele protsessile juba üle kantud, saab alles hakata `sendbuf` teise osa sisu puhvrisse kandma. Seega tekib mõlemal protsessil juba enne `MPI_Send`-käsu lõppu vajadus, et teine protsess alustaks oma `MPI_Recv`-käsuga, see pole aga võimalik, kuna `MPI_Send` ei ole ka seal veel lõpetanud.

Antud olukorras saab toodud probleemi vältida kirjutades `send`-käsu mitteblokeerivana:

```
if (minuid==0) then
  call MPIIsend(sendbuf, len, <MPLtype>, 1, tag, comm, request, ierr)
  call MPLRecv(recvbuf, len, <MPLtype>, 1, tag, comm, status, ierr)
  call MPLWait(request, status, ierr)
elseif (minuid==1) then
  call MPIIsend(sendbuf, len, <MPLtype>, 0, tag, comm, request, ierr)
  call MPLRecv(recvbuf, len, <MPLtype>, 0, tag, comm, status, ierr)
  call MPLWait(request, status, ierr)
endif
```

Küsimus: Miks ei tohi kirjutada käsku `MPI_Wait` kohe peale `MPI_Isend`-käsku?

2. `Receive` kõigepealt, seejärel `send`.

Järgnev programmilõik, mis kasutab blokeerivaid käske, põhjustab tupiku sõltumatult süsteemse teatedastuspuhvri suurusel:

```
if (minuid==0) then
  call MPLRecv(recvbuf, len, <MPLtype>, 1, tag, comm, status, ierr)
  call MPLSend(sendbuf, len, <MPLtype>, 1, tag, comm, ierr)
elseif (minuid==1) then
  call MPLRecv(recvbuf, len, <MPLtype>, 0, tag, comm, status, ierr)
  call MPLSend(sendbuf, len, <MPLtype>, 0, tag, comm, ierr)
endif
```

Kasutades aga mitteblokeerivat `MPI_Irecv`-käsku saame tupikuvaba lahenduse:

```

if ( minuid==0) then
  call MPIIrecv( recvbuf , len , <MPLtype> , 1 , tag , comm , request , ierr )
  call MPLSend( sendbuf , len , <MPLtype> , 1 , tag , comm , ierr )
  call MPLWait( request , status , ierr )
elseif ( minuid==1) then
  call MPIIrecv( recvbuf , len , <MPLtype> , 0 , tag , comm , request , ierr )
  call MPLSend( sendbuf , len , <MPLtype> , 0 , tag , comm , ierr )
endif

```

Küsimus: Kas `MPI_Isend` kasutamisest oluks kasu?

3. Üks protsessidest alustab käsuga `send` teine aga käsuga `receive`.

Sõltumata sellest, kas kasutada `MPI_(I)Send` või `MPI_(I)Recv`, on järgnev lahendus tupikuvaba:

```

if ( minuid==0) then
  call MPLSend( sendbuf , len , <MPLtype> , 1 , tag , comm , ierr )
  call MPLRecv( recvbuf , len , <MPLtype> , 1 , tag , comm , status , ierr )
elseif ( minuid==1) then
  call MPLRecv( recvbuf , len , <MPLtype> , 0 , tag , comm , status , ierr )
  call MPLSend( sendbuf , len , <MPLtype> , 0 , tag , comm , ierr )
endif

```

Üldjuhul soovitatakse kasutada järgnevat mudelit:

```

if ( minuid==0) then
  call MPLIsend( sendbuf , len , <MPLtype> , 1 , tag , comm , request1 , ierr )
  call MPIIrecv( recvbuf , len , <MPLtype> , 1 , tag , comm , request2 , ierr )
elseif ( minuid==1) then
  call MPLIsend( sendbuf , len , <MPLtype> , 0 , tag , comm , request1 , ierr )
  call MPIIrecv( recvbuf , len , <MPLtype> , 0 , tag , comm , request2 , ierr )
endif
call MPLWait( request1 , status1 , ierr )
call MPLWait( request2 , status2 , ierr )

```


Peatükk 9

Näide: Hõredate maatriksite klass

Antud õppematerjali lõpetuseks toome ühe veidi suuremahulisema näite Fortran95 ja MPI kasutamisest reaalse ülesande lahendamiseks. Püüame siin demonstreerida käsitletud programmeerimisvõtteid ja -vahendeid ning näeme, kuidas lihtsate vahenditega võib jõuda reaalsete tulemusteni.

Näide: Kaasgradientide meetod hõredate maatriksitega süsteemide lahendamiseks.

Andmestruktuurid hõredate maatriksite korral

Tihti on teadusarvutustes ja üldse matemaatilisel modelleerimisel kasutuses lõplike diferentside meetod (*FDM, finite differences method*), lõplike elementide meetod (*FEM, finite element method*) või lõplike mahtude meetod (*FVM, finite volume method*). Kõigi nende meetodite tulemuseks on hõredate maatriksitega lineaarvõrrandite süsteemid. Maatriksit loetakse hõredaks juhul, kui enamus maatriksi elementidest on nullid. Tavalise, kahemõõtmelise massiivina selliste maatriksite kujutamine arvutis oleks ebaefektiivne. Järgnevalt vaatleme üht lihtsat formaati hõredate maatriksite arvutis kujutamiseks.

9.1 Hõredate maatriksite kolmikformaad

Kolmikformaadis (*triple storage format*) antakse maatriksi A iga nullist erinev element a_{ij} kolme arvu abil: täisarvulised indeksid i ja j ning (enamuses rakendustes) reaalarvuline maatriksi elemendi väärtus a_{ij} . Seega, saame maatriksi A kujutamiseks kolm massiivi:

```
integer, dimension(1:nz) :: indi, indj  
real(kind=rk), dimension(1:nz) :: vals
```

Üks enamlevinumaid praktilisi ülesandeid on võrrandisüsteemi

$$Ax = b$$

(9.1)

lahendamine. Hõredate maatriksite korral kasutatakse tihti iteratiivseid meetodeid. Juhul, kui $N \times N$ maatriks A on lisaks veel ka sümmeetriline, siis üheks levinumaks lahendusalgoritmiks on **kaasgradientide meetod**. Meie eesmärk siin on luua üldine hõredate maatriksite klass, täiustada seda paralleeltöötluks vajaminevate mehhanismidega ning rakendada seda ühe konkreetse ülesande – Poissoni ülesande lahendamiseks kasutades paralleliseeritud kaasgradientide meetodit.

Selleks, et oleks lihtne valida ujukomaarvude täpsusastet kogu programmis tervikuna vaid ühest kohast, defineerime sobiva mooduli

Lähtetekst 9.1: Moodul ujukomaarvude täpsuse etteandmiseks MPI programmidele.

```

1 ! Fail: mpi_CG/RealKind.f90
2 ! Moodul topelttäpsuse määratlemiseks
3 module RealKind
4   use mpi
5   implicit none
6   ! Tavaline ujukoma täpsus: (kommenteerida välja üks kahest)
7   ! integer, parameter :: rk = selected_real_kind(6,37)
8   ! integer, parameter :: MPLRK = MPLREAL
9   ! Topelttäpsus:
10  integer, parameter :: rk = selected_real_kind(15,307)
11  integer, parameter :: MPLRK = MPLDOUBLEPRECISION
12 end module RealKind

```

Toodud moodulis on rida 11 vajalik MPI-käskudele õige täpsusastme määramiseks (lisaks sisetfunktsiooni `selected_real_kind` kasutamisele `kind`-parameetri `rk` määramisele).

Järgnevalt defineerime mooduli hõredate maatriksite kujutamiseks kolmikformaadis koos konstruktori (rida 16), destruktori (rida 32) ja meie jaoks tähtsaima funktsiooni $y = Ax$ operatsiooniga (rida 43):

Lähtetekst 9.2: Hõredate maatriksite klass.

```

1 ! Fail: mpi_CG/sparse_mat.f90
2 module sparse_mat
3   ! _____
4   ! Hõredate maatriksite klass
5   ! _____
6   use RealKind
7   implicit none
8   type sparse_m
9     integer :: nearv ! nullist erin. el. : arv
10    integer, dimension (:), pointer :: indj, indj ! mat(i:rida, j:veerg)
11    real(kind=rk), dimension (:), pointer :: val ! mat(i, j) väärtus
12 end type sparse_m
13
14 contains
15
16 function Loo_sparse_m(nearv, Aindi, Aindj, Aval) result(mat)
17   ! _____
18   ! Konstruktor
19   ! _____
20   implicit none

```

```

21  type(sparse_m) :: mat
22  integer, intent(in) :: nearv
23  integer, intent(in), dimension(nearv), optional :: Aindi, Aindj
24  real(kind=rk), intent(in), dimension(nearv), optional :: Aval
25  allocate(mat%indi(nearv), mat%indj(nearv), mat%val(nearv))
26  mat%nearv=nearv
27  if (present(Aindi)) mat%indi=Aindi
28  if (present(Aindj)) mat%indj=Aindj
29  if (present(Aval)) mat%val=Aval
30  end function Loo_sparse_m
31
32  subroutine Kustuta_sparse_m(mat)
33  !-----
34  ! Destruktor
35  !-----
36  implicit none
37  type(sparse_m), intent(in out) :: mat
38  mat%nearv=0
39  deallocate(mat%indi, mat%indj)
40  deallocate(mat%val)
41  end subroutine Kustuta_sparse_m
42
43  function sparse_Ax(A,x) result(y)
44  !-----
45  ! funktsioon y=Ax hõredate maatriksite korral
46  !-----
47  implicit none
48  type(sparse_m), intent(in) :: A
49  real(kind=rk), intent(in), dimension(:) :: x
50  real(kind=rk), dimension(size(x)) :: y ! sama pikk kui x !
51  integer :: i, j
52
53  y=0.0_rk ! nullime, kuna y automaatne massiiv
54  do j=1,A%nearv
55     i=A%indi(j)
56     y(i)=y(i)+A%val(j)*x(A%indj(j))
57  end do
58  end function sparse_Ax
59
60  subroutine tryki_sparse_m(A)
61  !-----
62  ! Väljatrükk (kasulik debugimisel;)
63  !-----
64  implicit none
65  type(sparse_m), intent(in) :: A
66  integer :: j
67  do j=1,A%nearv
68     print *, 'sparse_m( ', j, ' ): ', A%indi(j), A%indj(j), A%val(j)
69  end do
70  end subroutine tryki_sparse_m
71
72  function Laplace_M(N) result(mat)
73  !-----
74  ! Klassi sparse_m testimiseks: Funktsioon mis loob Laplace'i
75  !   operaatori diskretisatsioonimaatriksi ühtlasel ruudukujulisel
76  !   2D-võrgul Dirichlet nulliliste rajatingimuste korral
77  !-----
78  implicit none
79  type(sparse_m) :: mat

```

```

80  integer , intent ( in ) :: N ! Laplace 'i matriksi mõõtmed:
81                                     !   (N^2 x N^2)
82  integer :: nearv , i , k
83  integer :: el !
84  nearv=N*(N+2*(N-1))+2*(N-1)*N ! Nullist erin. elem. arv
85  mat=Loo_sparse_m(nearv) ! Hõreda matriksi konstruktor
86  el=0
87  ! Põhidiagonaalblokk
88  !   elemendid põhidiagonaalil
89  do i=1,N**2
90      el=el+1
91      mat%indi( el)=i
92      mat%indj( el)=i
93      mat%val( el)=4.0_rk
94  end do
95  !   elemendid kõrvaldiagonaalidel
96  do k=1,N ! tsükkel üle diagonaalblokkide
97      el=el+1 !   iga diagonaalbloki teine nullist erin.el.:
98      mat%indi( el)=(k-1)*N+1
99      mat%indj( el)=(k-1)*N+2
100     mat%val( el)=-1.0_rk
101     el=el+1 !   iga diagonaalbloki eelviimane nullist erin.el.:
102     mat%indi( el)=k*N
103     mat%indj( el)=k*N-1
104     mat%val( el)=-1.0_rk
105     do i=(k-1)*N+2,k*N-1
106         el=el+1 ! alamdiagonaal:
107         mat%indi( el)=i
108         mat%indj( el)=i-1
109         mat%val( el)=-1.0_rk
110         el=el+1 ! ülemdiagonaal:
111         mat%indi( el)=i
112         mat%indj( el)=i+1
113         mat%val( el)=-1.0_rk
114     end do
115 end do
116 ! Ülejäänud Laplace 'i matriksi blokid:
117 do i=N+1,N**2-N ! teisest kuni eelviimase blokireani:
118     el=el+1 ! All-diagonaalblokid:
119     mat%indi( el)=i
120     mat%indj( el)=i-N
121     mat%val( el)=-1.0_rk
122     el=el+1 ! Peal-diagonaalblokid:
123     mat%indi( el)=i
124     mat%indj( el)=i+N
125     mat%val( el)=-1.0_rk
126 end do
127 ! Esimene peal-diagonaalblokk:
128 do i=1, N
129     el=el+1
130     mat%indi( el)=i
131     mat%indj( el)=i+N
132     mat%val( el)=-1.0_rk
133 end do
134 ! Viimane all-diagonaalblokk:
135 do i=N**2-N+1,N**2
136     el=el+1
137     mat%indi( el)=i

```

```

138     mat%indj(e1)=i-N
139     mat%val(e1)=-1.0_rk
140 end do
141 ! Kontroll:
142 if (e1/=nearv) print *, "VIGA: _vale_nearv!"
143 end function Laplace_M
144 end module sparse_mat

```

Klassi `sparse_mat` testimiseks on toodud alamprogramm `Laplace_M` (Lähteteksti 9.2 rida) Laplace'i maatriksite genereerimiseks. Laplace'i maatriks tekib näiteks Poissoni võrrandi

$$u_{xx} + u_{yy} = f, \quad \Omega = \{(x, y) : 0 \leq x \leq 1, 0 \leq y \leq 1\}, \quad (9.2)$$

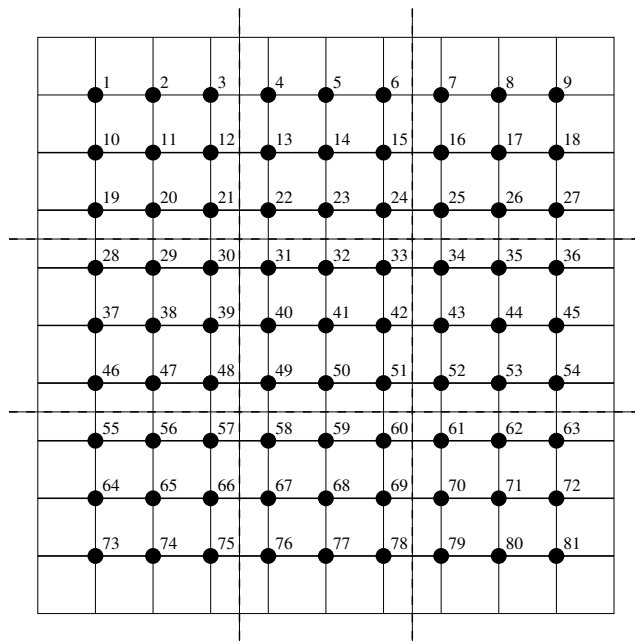
diskretiseerimisel nulliliste Dirichlet' rajatingimuste korral. Kui diskretiseerimisvõrk on ühtlane ning koosneb $n \times n$ sõlmest, siis näeb $n^2 \times n^2$ Laplace'i maatriks välja järgmine:

$$A = \begin{bmatrix} B & -I & 0 & \cdots & 0 \\ -I & B & -I & \ddots & \vdots \\ 0 & -I & B & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -I \\ 0 & \cdots & 0 & -I & B \end{bmatrix}, \quad (9.3)$$

kus $n \times n$ maatriks B on kujul:

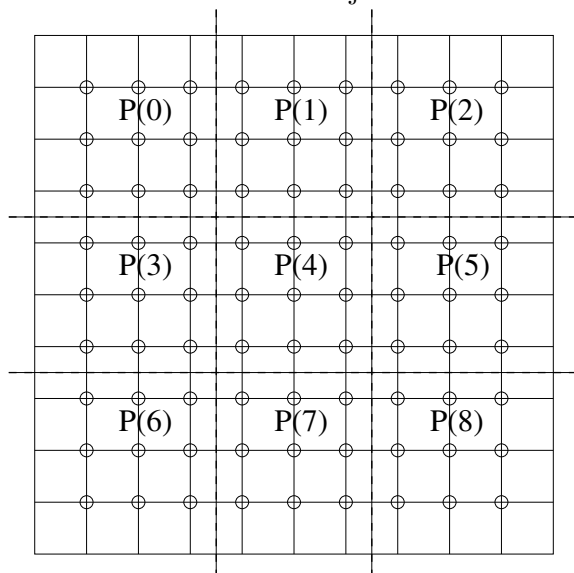
$$B = \begin{bmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & \ddots & \vdots \\ 0 & -1 & 4 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 4 \end{bmatrix}$$

ja I on ühikmaatriks. Funktsioon `Laplace_M` saab ette vaid ühe parameetri – n – sõlmede arvu nii x kui ka y -telje suunas. Järgneval joonisel on toodud eeldatav (üks võimalikest) võrgusõlmede numeratsioon `Laplace_M` juhul kui $n = 9$.

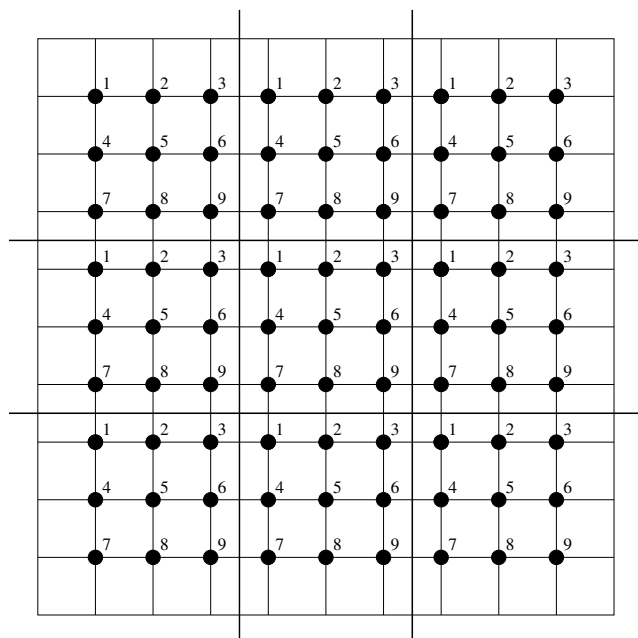


9.2 Paralleliseerimine

Paralleliseerimiseks jagame ülesande lahenduspiirkonna Ω alampiirkondadeks. Lihtsuse mõttes teeme tükelduse p võrdseks osaks nii x kui ka y -suunas, saades $P = p^2$ mittekattuvat alampiirkonda Ω_k , $k = 0, \dots, P - 1$, kus P on etteantud protseside arv. Järgneval joonisel on toodud võimalik jaotus $P = 9$ alampiirkonnaks $p = 3$ korral:



Jaotuse tulemusena saab iga protsess endale teatud arvu sõlmi, mis saavad igal protsessil lokaalse numeratsiooni. Üks võimalik lokaalne numeratsioon on toodud järgneval joonisel:

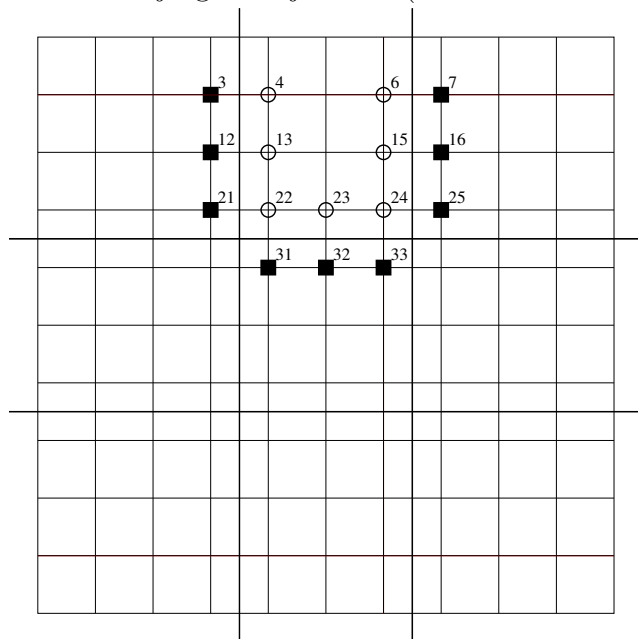


(Märgime, et tegelik lokaalne numeratsioon võib vabalt olla erinev joonisel toodust.)

Me eeldame, et protsess järguga 0 ($P(0)$) genereerib Laplace'i maatriksi A , jagab selle vastavalt toodud skeemile alammatriksiteks $A^{(k)}$, $k = 0, \dots, P - 1$ selliselt, et suvaline maatriksi A element $a_{ij} \in A^{(k)}$ parajasti siis, kui mõlemad indeksid $i, j \in \Omega_k$. Seejärel jaotab protsess $P(0)$ alammatriksids laiali (jättes ka iseendale ühe alammatriksidi).

Saanud kätte oma alammatriksidi, nummerdab iga protsess oma sõlmed lokaalselt ringi ning samas genereerib massiivi `global2local`, kus `global2local(k)` annab k -nda globaalse sõlme numbrit lokaalses järjestuses.

Selleks, et kogu maatriks A saaks kaetud, tuleb lisada ka sellised maatriksi A elemendid a_{ij} , mille korral $i \in \Omega_k$ ja $j \in \Omega_l$, kus $k \neq l$ (st. ühendused joonisel, mis on alampiirkondadevahelise joonega läbi lõigatud.) Selleks saadab protsess $P(0)$ igale ülejäänule ka vastava täiendmaatriksi. Kõik protsessid lisavad vastavalt saadud täiendmaatriksile oma lokaalsete sõlmede massiivi lõppu niinimetatud vari-sõlmed. Varisõlmede asukoht protsessi $P(1)$ korral on toodud järgneval joonisel (tähistatud ruudukestega):



(Märgime, et joonisel on sõlmede numbrid toodud globaalses numeratsioonis, tegelikult saab ka iga varisõlm lokaalse järjestuse, mis kajastub massiivis `global2local`.)

Etteantud protsessil on oluline teada, millisele naaberprotsessile tema suvaline varisõlm tegelikult vastab. Selleks saadab protsess $P(0)$ laiali ka massiivi `so_reg(1:n2)`, (“sõlmeomechanike register”) mis annab iga sõlme kohta alampiirkonna numbril, kuhu antud sõlm kuulub. Selle abil leitakse iga varisõlmele vastava naaberpiirkonna tegeliku sõlme lokaalne järk kasutades massiivi `global2local` naaberprotsessil.

Kirjeldatud andmestruktuurid luuakse järgnevas programmilõigus, mis kirjeldab ära klassi `par_sparse_mat` konstruktori, destruktori ning kõik ülejäänud vajaminevad meetodid, ka need mis on loodud andmete hajutamiseks protsessorite vahel ning tulemuse kokkukogumiseks protsessile $P(0)$. Märgime, et toodud `par_sparse_mat` meetodid ja andmestruktuurid on realiseeritud nii, et need töötavad suvalise `so_reg(1:n2)` massiivi korral.

Lähtetekst 9.3: Hajusate hõredate maatriksite klass

```

1 ! File: mpi_CG/par_sparse_mat.f90
2 module par_sparse_mat
3   use sparse_mat
4   use RealKind
5   use mpi
6   type par_sparse_m
7     integer :: siseyhendusi ! lokaalse maatriksi nullist erin. el.
8                       ! arv ilma yhendusteta varisõlmedesse
9     integer :: variyhendusi ! lok. maatriksi yhend. arv varisõlmedesse
10    type(sparse_m) :: sise_mat ! yhendustega vaid sisesõlmedel
11    type(sparse_m) :: vari_mat ! yhendustega sise- ja varisõlmede vahel
12 end type par_sparse_m

```

```

13  type alamhulk ! kasutame alamhulkade defineerimiseks sõlmede hulgal
14  integer :: solmi
15  integer , dimension (:) , pointer :: solmenumbrid
16  end type alamhulk
17  type(alamhulk) , dimension (:) , allocatable :: varisolmed_naabrilt , &
18  minusolmed_naabrile , partitsioon
19  integer , dimension (:) , pointer :: varisolmede_algus_naabrile
20  real(kind=rk) , dimension (:) , pointer :: puhver , masteri_puhver
21  integer :: omasolmi , varisolmi , myid , numprocs , master
22  integer :: glob_solmi ! sõlmede koguarv
23
24  contains
25
26  function Loo_par_sparse_mat(comm,so_reg ,A) result(par_mat)
27  !-----
28  ! Konstruktor
29  ! Saab sisendiks matriksi A mis on defineeritud vaid
30  ! kommunikaatori comm 0-ndal protsessil.
31  ! so_reg — (sõlmeomanike register) antud algselt vaid
32  ! 0-ndal protsessil , väljundiks ülejäänutel
33  ! Väljundiks on par_mat objekt tüübist par_sparse_m ,
34  ! (defineeritud varisolmede
35  !-----
36  implicit none
37  integer , intent(in) :: comm ! kommunikaator
38  type(sparse_m) , intent(in) :: A ! suur matriks 0ndal prots.
39  integer , dimension (:) , pointer :: so_reg ! iga sõlme omaniku järk
40  type(par_sparse_m) :: par_mat ! konstrueeritav hajusobjekt
41  integer , dimension (:) , pointer :: global2local
42  integer :: stat(MPLSTATUS_SIZE) , ierr , sisepiir
43  call MPLCOMM_RANK(comm,myid,ierr) ! protsessi järk
44  call MPLCOMMSIZE(comm,numprocs,ierr) ! protsesside arv
45
46  master=0 ! 0 on peamine
47  if (myid==master) then
48  glob_solmi=size(so_reg)
49  endif
50  ! Saadame laiali sõlmede arvu ja kuulumise:
51  call MPLBCAST(glob_solmi , 1 , MPLINTEGER, master , comm, ierr)
52  if (myid/=master) then
53  allocate(so_reg(glob_solmi))
54  endif
55  call MPLBCAST(so_reg , glob_solmi , MPLINTEGER, master , comm, ierr)
56  par_mat = hajuta_mat(A,so_reg)
57  call suhtluse_ettevalmistus(so_reg , par_mat)
58  if (myid==master) then
59  call tee_partitsioonid(comm,so_reg)
60  endif
61
62  contains ! abiprotseduurid:
63
64  function hajuta_mat(A,so_reg) result(par_A)
65  !-----
66  ! hajuta_mat tulemusel hajutatakse masteri A protsesside vahel
67  ! hajusobj.-ks par_mat lähtudes sõlmeomanike registrist so_reg
68  ! par_A objektis on aga matriksites indeksid veel globaalsed
69  !-----
70  implicit none
71  type(sparse_m) :: A ! defineeritud vaid protsessil 0

```

```

72  type(par_sparse_m) :: par_A ! tulemuseks maatriks A hajusana
73  integer :: so_reg(:) ! seda teavad kõik
74  integer :: i,j,id,esiloendur,tagaloendur,mitu
75  integer,dimension(:),allocatable :: Aelarv ! A elementide arv,
76                                     ! mis antud protsessile kuulub
77  integer,dimension(:),allocatable :: Aindi,Aindj
78  real(kind=rk),dimension(:),allocatable :: Aval
79  if (myid.eq.master) then
80     allocate(Aelarv(0:numprocs-1))
81     Aelarv=0
82     do i=1,A%nearv
83        j=so_reg(A%indi(i))
84        Aelarv(j)=Aelarv(j)+1
85     end do
86     ! vaatame maatriksi läbi iga protsessi seisukohalt:
87     do id=1,numprocs-1
88        mitu=Aelarv(id)
89        call MPLSEND(mitu,1,MPLINTEGER,id,master,comm,ierr)
90        allocate(Aindi(mitu),Aindj(mitu),Aval(mitu))
91        esiloendur=0
92        tagaloendur=mitu
93        ! kõigepealt pannakse massiividesse Aindi, Aindj ja Aval
94        ! need maatriksi A elemendid, mille mõlemad indeksid
95        ! (i ja j) kuuluvad samale protsessile
96        do i=1,A%nearv
97           if(so_reg(A%indi(i))==id) then ! juhul kui id sõlm
98              if (so_reg(A%indj(i))==id) then ! sisõlm-sisesõlm
99                 esiloendur=esiloendur+1
100                Aindi(esiloendur)=A%indi(i)
101                Aindj(esiloendur)=A%indj(i)
102                Aval(esiloendur)=A%val(i)
103             else ! sisesõlm-varisõlm: ühendused lisame tagaossa:
104                Aindi(tagaloendur)=A%indi(i)
105                Aindj(tagaloendur)=A%indj(i)
106                Aval(tagaloendur)=A%val(i)
107                tagaloendur=tagaloendur-1
108             end if
109           endif
110        end do
111        if (esiloendur/=tagaloendur) then ! otsad koos?
112           print *, 'VIGA: _loenduriviga.'
113        endif
114        call MPLSEND(esiloendur,1,MPLINTEGER,id,master,comm,ierr)
115        ! saadame A tüki ära:
116        call MPLsend(Aindi,mitu,MPLINTEGER,id,master,comm,ierr)
117        call MPLsend(Aindj,mitu,MPLINTEGER,id,master,comm,ierr)
118        call MPLsend(Aval,mitu,MPLRK,id,master,comm,ierr)
119        deallocate(Aindi,Aindj,Aval)
120     end do
121  else ! slave võtab pakutu vastu:
122     call MPLRECV(mitu,1,MPLINTEGER,master,master,comm,stat,ierr)
123     call MPLRECV(sisepiir,1,MPLINTEGER,master,master,&
124                comm,stat,ierr)
125     allocate(Aindi(mitu),Aindj(mitu),Aval(mitu))
126     call MPLRECV(Aindi,mitu,MPLINTEGER,master,master,&
127                comm,stat,ierr)
128     call MPLRECV(Aindj,mitu,MPLINTEGER,master,master,&
129                comm,stat,ierr)
130     call MPLRECV(Aval,mitu,MPLRK,master,master,comm,stat,ierr)

```

```

131     par_A%siseyhendusi=sisepiir
132     par_A%variychendusi=mitu-sisepiir
133     ! Loodud maatriksites on indeksid globaalses järjestuses...
134     par_A%sise_mat=Loo_sparse_m(sisepiir, Aindi, Aindj, Aval)
135     par_A%vari_mat=Loo_sparse_m(par_A%variychendusi, &
136         Aindi(sisepiir+1), Aindj(sisepiir+1), Aval(sisepiir+1))
137     deallocate(Aindi, Aindj, Aval)
138 endif
139 ! Masteri enda alammaatriks ka:
140 if (myid.eq.master) then
141     mitu=Aelrv(master)
142     allocate(Aindi(mitu), Aindj(mitu), Aval(mitu))
143     esiloendur=0
144     tagaloendur=mitu
145     ! kõigepealt pannakse massiividesse Aindi, Aindj ja Aval kirja
146     ! need maatriksi A elemendid, mille mõlemad indeksid (i ja j)
147     ! kuuluvad samale protsessile
148     do i=1,A%nearv
149         if (so_reg(A%indi(i))==master) then ! juhul kui masteri sõlm
150             if (so_reg(A%indj(i))==master) then ! sisõlm-sisesõlm
151                 esiloendur=esiloendur+1
152                 Aindi(esiloendur)=A%indi(i)
153                 Aindj(esiloendur)=A%indj(i)
154                 Aval(esiloendur)=A%val(i)
155             else ! sisesõlm-varisõlm: ühendused lisame tagaossa:
156                 Aindi(tagaloendur)=A%indi(i)
157                 Aindj(tagaloendur)=A%indj(i)
158                 Aval(tagaloendur)=A%val(i)
159                 tagaloendur=tagaloendur-1
160             end if
161         endif
162     end do
163     if (esiloendur/=tagaloendur) then ! otsad koos?
164         print *, 'VIGA: _loenduriviga.'
165     endif
166     sisepiir=esiloendur
167     par_A%siseyhendusi=esiloendur
168     par_A%variychendusi=mitu-esiloendur
169     ! Loodud maatriksites indeksid veel globaalses järjestuses...
170     par_A%sise_mat = &
171         Loo_sparse_m(par_A%siseyhendusi, Aindi, Aindj, Aval)
172     par_A%vari_mat=Loo_sparse_m(par_A%variychendusi, &
173         Aindi(sisepiir+1), Aindj(sisepiir+1), Aval(sisepiir+1))
174     deallocate(Aindi, Aindj, Aval)
175     deallocate(Aelrv)
176 endif
177 end function hajuta_mat
178
179 subroutine suhtluse_ettevalmistus(so_reg, par_A)
180     integer, intent(in) :: so_reg(:) ! seda teavad kõik
181     type(par_sparse_m) :: par_A ! maatriks hajusobjektina
182     ! solmenumbrite teisenduseks:
183     integer, dimension(:), pointer :: global2local
184     integer, dimension(:), allocatable :: ootus, ireqs, ireqr
185     integer :: i, j, k, loendur, naaber, id, maxsaata
186
187     allocate(varisolmed_naabrilt(0:numprocs-1))
188     allocate(minusolmed_naabrile(0:numprocs-1))
189     allocate(ireqs(0:numprocs-1))

```

```

190 allocate(ireqr(0:numprocs-1))
191 allocate(global2local(glob_solmi))
192 global2local=0
193 loendur=1
194 do i=1,glob_solmi
195     if(so_reg(i)==myid) then
196         global2local(i)=loendur
197         loendur=loendur+1
198     end if
199 end do
200 omasolmi=loendur-1 ! antud protsessi oma sõlmede arv
201 allocate(ootus(0:numprocs-1)) ! sõlmede arv protsessilt
202 ootus=0
203 ! Märgistame global2local kohad mis globaalses numeratsioonis
204 ! saavad olema meie varisõlmede kohad esialgu arvuga -1,
205 ! samas loendame ootuse igalt naabrilt:
206 do i=1,par_A%variuhendus
207     if(global2local(par_A%vari_mat%indj(i))==0) then
208         global2local(par_A%vari_mat%indj(i))=-1
209         naaber=so_reg(par_A%vari_mat%indj(i))
210         ootus(naaber)=ootus(naaber)+1
211     end if
212 end do
213 ! Varisõlmed iga naabri jaoks asuvad eraldi:
214 allocate(varisolmede_algus_naabrile(0:numprocs-1))
215 varisolmede_algus_naabrile=0
216 do id=0,numprocs-1
217     if(id/=myid) then
218         varisolmede_algus_naabrile(id)=loendur
219         loendur=loendur+ootus(id)
220     end if
221 end do
222 varisolmi=loendur-omasolmi-1 ! varisolmede arv antud protsessil
223 ! INFOVAHETUS. Kõigepealt võtame mälu:
224 do id=0,numprocs-1
225     varisolmed_naabrilt(id)%solmi=0
226     if(ootus(id)/=0) then
227         allocate(varisolmed_naabrilt(id)%solmenumbrid(ootus(id)))
228     end if
229 end do
230 do i=1,glob_solmi
231     if(global2local(i)==-1) then !Vaatama uuesti varislm. asukohti
232         j=so_reg(i) ! omanik on j
233         k=varisolmed_naabrilt(j)%solmi+1
234         varisolmed_naabrilt(j)%solmi=k ! suurendame ühevõrra
235         varisolmed_naabrilt(j)%solmenumbrid(k)=i ! salvest.glob.num.
236         global2local(i)=varisolmede_algus_naabrile(j)+k-1!uus väärt.
237     end if
238 end do
239 ! Nüüd võime lok. maatriksite indeksid teisendada lokaalseks:
240 do i=1,par_A%siseyhendusi
241     par_A%sisemat%indi(i)=global2local(par_A%sisemat%indi(i))
242     par_A%sisemat%indj(i)=global2local(par_A%sisemat%indj(i))
243 end do
244 do i=1,par_A%variuhendus
245     par_A%vari_mat%indi(i)=global2local(par_A%vari_mat%indi(i))
246     par_A%vari_mat%indj(i)=global2local(par_A%vari_mat%indj(i))
247 end do
248 ! Teavitame protsesse mitu sõlme neilt (nemad salvestavad selle

```

```

249      ! muutujas minusolmed_naabrile (:)%solmi)
250      do id=0,numprocs-1
251          if(id/=myid) then
252              call MPLSend(varisolmed_naabrilt(id)%solmi,1, &
253                          MPLINTEGER,id,myid,comm,ierr)
254          end if
255      end do
256      minusolmed_naabrile(myid)%solmi=0
257      maxsaata=0
258      do id=0,numprocs-1
259          if(id/=myid) then
260              call MPLRecv(k,1,MPLINTEGER,MPLANY_SOURCE, &
261                          MPLANY_TAG,comm,stat,ierr)
262              minusolmed_naabrile(stat(MPLTAG))%solmi=k
263              if(k>maxsaata) maxsaata=k
264          end if
265      end do
266      allocate(puhver(maxsaata)) ! kasut. hiljem tegel. väärt. saadm.
267      call MPLBARRIER(comm,ierr)
268
269      ! saadame teistele varisolmed_naabrilt(id)%solmenumbrid
270      ! tulemus salvest. kui
271      ! minusolmed_naabrile(protsessi_nr)%solmenumbrid
272      do id=0,numprocs-1
273          if(ootus(id)/=0) then
274              call MPLIsend(varisolmed_naabrilt(id)%solmenumbrid, &
275                          ootus(id),MPLINTEGER,id,myid,comm,ireqs(id),ierr)
276          end if
277      end do
278      do id=0,numprocs-1
279          k=minusolmed_naabrile(id)%solmi
280          if(k/=0) then
281              allocate(minusolmed_naabrile(id)%solmenumbrid(k))
282              call MPLIrecv(minusolmed_naabrile(id)%solmenumbrid,k, &
283                          MPLINTEGER,id,id,comm,ireqr(id),ierr)
284          end if
285      end do
286      do id=0,numprocs-1 ! ootame, kuni kõik saadetud
287          if(ootus(id)/=0) then
288              call MPLWait(ireqs(id),stat,ierr)
289          end if
290      end do
291      do id=0,numprocs-1 ! ootame, kuni kõik saabunud
292          if(minusolmed_naabrile(id)%solmi/=0) then
293              call MPLWait(ireqr(id),stat,ierr)
294          end if
295      end do
296      ! lõpuks teisendame indeksid lokaalseteks ja OK:
297      do id=0,numprocs-1
298          do i=1,minusolmed_naabrile(id)%solmi
299              minusolmed_naabrile(id)%solmenumbrid(i) = &
300                  global2local(minusolmed_naabrile(id)%solmenumbrid(i))
301          end do
302      end do
303      deallocate(ootus)
304      deallocate(global2local)
305      deallocate(ireqr)
306      deallocate(ireqs)
307      end subroutine subtluse_ettevalmistus

```

```

308 end function Loo_par_sparse_mat
309
310 function par_sparse_Ax(comm,A,par_x0 , nulli) result(par_y)
311     ! _____
312     ! funktsioon y=Ax hõredate hajusmaatriksite korral
313     ! _____
314     implicit none
315     integer ,intent(in) :: comm ! kommunikaator
316     type(par_sparse_m) ,intent(in) :: A
317     real(kind=rk) ,intent(in) ,dimension(:) :: par_x0
318     logical ,optional ,intent(in) :: nulli ! .true., => par_y=0 algul
319     real(kind=rk) ,dimension(size(par_x0)) :: par_y ! sama pikk kui x !
320     ! vajame massiivi, kus ruumi ka varisolmedele:
321     real(kind=rk) :: par_x(omasolmi+varisolmi) ! automaatne massiiv
322     integer :: i,j,k,id
323     integer ,dimension(:) ,allocatable :: ireq
324     integer :: stat(MPLSTATUS_SIZE) ,ierr
325
326     par_x(1:size(par_x0)) = par_x0
327     allocate(ireq(0:numprocs-1))
328     do id=0,numprocs-1 ! varisolmede väärtuste vastuvõtu alustus
329         k=varisolmed_naabrilt(id)%solmi
330         if (k/=0) then
331             call MPI_Irecv(par_x(varisolmede_algus_naabrile(id)),k, &
332                 MPLRK,id ,id ,comm,ireq(id) ,ierr)
333         end if
334     end do
335     do id=0,numprocs-1 ! teistele protsessidele nende varisolmed
336         k=minisolmed_naabrile(id)%solmi
337         if (k/=0) then
338             do i=1,k ! kogume puhvrise
339                 puhver(i)=par_x(minisolmed_naabrile(id)%solmenumbrid(i))
340             enddo ! ... ja saadame teele, TAG=myid
341             call MPLSend(puhver,k,MPLRK,id ,myid, &
342                 comm,ierr)
343         endif
344     enddo
345     if (present(nulli)) then ! algväärtustamine kui vaja
346         if (nulli) par_y=0.0_rk
347     end if
348     ! Korrutame siseõlmede osa:
349     par_y = sparse_Ax(A%sise_mat ,par_x)
350     ! Veendume, et varisolmede väärtused kohal:
351     do id=0,numprocs-1
352         if (varisolmed_naabrilt(id)%solmi/=0) then
353             call MPLWAIT(ireq(id) ,stat ,ierr)
354         endif
355     enddo
356     ! lisame varisolmede osa korrutise:
357     par_y = par_y + sparse_Ax(A%vari_mat ,par_x)
358     deallocate(ireq)
359 end function par_sparse_Ax
360
361 ! _____
362 ! Alamprogramm, mis vajalikud tööks vektoritega:
363 ! _____
364 subroutine tee_partitsioonid(comm,so_reg)
365     ! _____
366     ! masteri ettenalmistus nektorite haitutuseks ja kokkukoostamiseks

```



```

367  ! (alternatiiv oleks koguda kõigi global2local>0 asukohad)
368  !-----
369  implicit none
370  integer , intent ( in ) :: comm ! kommunikaator
371  integer , intent ( in ) :: so_reg ( : )
372  integer , dimension ( : ) , allocatable :: solmedearv
373  integer :: i , k , id , omanik , maxsaata
374  integer :: numprocs , ierr
375  call MPLCOMM_SIZE ( comm , numprocs , ierr ) ! protsesside arv
376
377  allocate ( solmedearv ( 0 : numprocs - 1 ) ) ! loendab solmi igal protsessil
378  solmedearv = 0
379  do i = 1 , glob_solmi
380     omanik = so_reg ( i )
381     solmedearv ( omanik ) = solmedearv ( omanik ) + 1
382  end do
383  allocate ( partitsioon ( 0 : numprocs - 1 ) ) ! tüübist alamhulk
384  ! leiame partitsioonide suurused:
385  do id = 0 , numprocs - 1
386     partitsioon ( id ) % solmi = 0
387     allocate ( partitsioon ( id ) % solmenumbrid ( solmedearv ( id ) ) )
388  end do
389  do i = 1 , glob_solmi ! partitsioonidesse jagamine
390     omanik = so_reg ( i )
391     k = partitsioon ( omanik ) % solmi + 1
392     partitsioon ( omanik ) % solmi = k
393     partitsioon ( omanik ) % solmenumbrid ( k ) = i
394  end do
395  maxsaata = MAXVAL ( solmedearv )
396  deallocate ( solmedearv )
397  allocate ( masteri_puhver ( maxsaata ) )
398 end subroutine tee_partitsioonid
399
400 subroutine hajuta_vektor ( comm , vec , minu_osavec )
401  !-----
402  ! master hajutab vec laiali osavektoriteks
403  !-----
404  implicit none
405  integer :: comm ! kommunikaator
406  real ( kind = rk ) , dimension ( : ) :: vec ! antud vaid masteril
407  real ( kind = rk ) , dimension ( : ) :: minu_osavec ! tulemusena kõigil
408  integer :: id , stat ( MPLSTATUS_SIZE ) , ierr
409  integer :: i , k
410  if ( myid == master ) then
411     do id = 1 , numprocs - 1
412        k = partitsioon ( id ) % solmi
413        do i = 1 , k
414           masteri_puhver ( i ) = vec ( partitsioon ( id ) % solmenumbrid ( i ) )
415        end do
416        call MPLSend ( masteri_puhver , k , MPLRK , id , myid , comm , ierr )
417     end do
418     do i = 1 , partitsioon ( master ) % solmi ! masteri enda osa
419        minu_osavec ( i ) = vec ( partitsioon ( master ) % solmenumbrid ( i ) )
420     end do
421  else ! slave 'id:
422     call MPLRecv ( minu_osavec , omasolmi , MPLRK , &
423                  master , master , comm , stat , ierr )
424  endif
425 end subroutine hajuta_vektor

```



```

426
427 subroutine kogu_vektor_kokku(comm, minu_osavec, vec)
428     !-----
429     ! master kogub osavektorid kokku suureks vektoriks vec
430     !-----
431     implicit none
432     integer :: comm ! kommunikaator
433     real(kind=rk), dimension(:) :: vec ! defineeritud vaid masteril
434     real(kind=rk), dimension(:) :: minu_osavec ! defineeritud kõigil
435     integer :: stat(MPLSTATUS_SIZE), ierr, saatja
436     integer :: id, i
437     if (myid/=master) then ! minu vektori osa masterile, TAG=myid
438         call MPLSend(minu_osavec, omasolmi, MPLRK, &
439             master, myid, comm, ierr)
440     else
441         do i=1, partitsioon(master)%solmi ! master alustab enda osaga
442             vec(partitsioon(master)%solmenumbrid(i))=minu_osavec(i)
443         end do
444         do id=1, numprocs-1 ! (saabuva teate pikkus pole tegelikult õige)
445             call MPLRecv(masteri_puhver, size(masteri_puhver), MPLRK, &
446                 MPLANY_SOURCE, MPLANY_TAG, MPICOMMWORLD, stat, ierr)
447             saatja=stat(MPLTAG)
448             do i=1, partitsioon(saatja)%solmi
449                 vec(partitsioon(saatja)%solmenumbrid(i))=masteri_puhver(i)
450             end do
451         end do
452     endif
453 end subroutine kogu_vektor_kokku
454
455 subroutine Kustuta_par_sparse_mat(A) ! destruktor TODO !
456     !-----
457     ! Destruktor
458     !-----
459     implicit none
460     type(par_sparse_m), optional :: A
461     integer :: id, k
462     if (present(A)) then ! Kustutame maatriksi
463         call Kustuta_sparse_m(A%sisem_mat)
464         call Kustuta_sparse_m(A%vari_mat)
465     else ! vabastame kõik loodud mälustruktuurid (soovitatakse
466         ! teha tagurpidises järjekorras allokeerimisega)
467         ! mälueraldused alamprogrammist tee_partitsioonid:
468         if (myid==master) then
469             deallocate(masteri_puhver)
470             do id=numprocs-1, 0, -1
471                 deallocate(partitsioon(id)%solmenumbrid)
472             end do
473             deallocate(partitsioon)
474         endif
475         ! alamprogramm suhtluse_ettevalmistus(so_reg, par_A):
476         do id=numprocs-1, 0, -1
477             if (minusolmed_naabrile(id)%solmi/=0) then
478                 deallocate(minusolmed_naabrile(id)%solmenumbrid)
479             end if
480         end do
481         deallocate(puhver)
482         do id=numprocs-1, 0, -1
483             if (varisolmed_naabrilt(id)%solmi/=0) then
484                 deallocate(varisolmed_naabrilt(id)%solmenumbrid)

```

```

485         end if
486     end do
487     deallocate ( varisolmede_algus_naabrile )
488     deallocate ( minusolmed_naabrile )
489     deallocate ( varisolmed_naabrilt )
490 endif
491 end subroutine Kustuta_par_sparse_mat
492 end module par_sparse_mat

```

Sellega ongi loodud kõik antud näites vajalikud andmestruktuurid paralleelse kaasgradientide meetodi realiseerimiseks.

9.3 Kaasgradientide meetodi testimine

Lisaks hajutatud maatriksile A jaotatakse protsessi $P(0)$ poolt laiali vastavalt ka vektor \mathbf{b} , see on realiseeritud alamprogrammis `hajuta_vektor` (vt. lähteteksti 9.3 rida 400). Kõik kaasgradientide algoritmis vajaminevad abivektorid on vaid hajutatud kujul. Üksnes vastusvektor \mathbf{x} kogutakse peale soovitud täpsuse saavutamist kokku protsessile $P(0)$ alamprogrammi `kogu_vektor_kokku` abil (lähteteksti 9.3 rida 427).

Paralleelse kaasgradientide meetodi tööks on vaja sooritada paralleelselt kahte põhioperatsiooni:

1. **Skalaarkorrutised**, mis on ühel protsessil kõige parem realiseerida Fortran95 sisefunktsiooni `dot_product` abil (järgnevas lähtekoodis 9.4 ridadel 123 ja 129). Selleks, et igalt protsessilt saadud tulemused kokku liita nii, et kõik protsessid saaksid tulemusena teada summa, on mugavaim ja efektiivseim võimalus kasutada käsku `MPI_Allreduce` nagu on toodud lähtekoodi 9.4 ridadel 124 ja 130.
2. **Maatriksi korrutamine vektoriga**, mis on realiseeritud `par_sparse_mat` klassifunktsioonina `par_sparse_Ax` (vt. lähteteksti 9.3 rida 310). Antud operatsiooni paralleelne realisatsioon on toodud programmi efektiivsuse võtmeküsimusi – kasutatakse mitteblokeerivat kommunikatsiooni, täites teadete vahetamisele kuluva aja kasulike arvutustega. Operatsioon teostatakse viies etapis:
 - A. Alustatakse varisõlmede mitteblokeerivaid vastuvõtuoperatsioone käskudega `MPI_Irecv` kõigilt naaberprotsessidelt, kellega antud protsessi seob mõni variimaatriksi ühendus (vt. lähteteksti 9.3 ridu 328-334).
 - B. Alustatakse mitteblokeerivaid saatmisoperatsioone `MPI_Isend`-käskudega, mille sisuks on antud protsessilt naabritele vajaminevad väärtused (naaberprotsessid kaasajastavad oma vastavate varisõlmede väärtused peale saabunud etapi A teadete vastuvõtmist).
 - C. Sooritatakse Ax -operatsioon sisemaatriksiga (lähteteksti 9.3 rida 349) kutsudes välja klassi `sparse_mat` funktsiooni `sparse_Ax` vt. (lähteteksti 9.2 rida 43).
 - D. Veendutakse, et kõik varisõlmede väärtused on naabritelt saabunud (lähteteksti 9.3 read 351-355).
 - E. Sooritatakse Ax -operatsioon sisemaatriksiga (lähteteksti 9.3 rida 357)

Järgnevalt toome kaasgradientide meetodi paralleelse realisatsiooni ja põhiprogrammi:

Lähtetekst 9.4: Suurte hõredate maatriksitega süsteemide paralleelne lahendamine kaasgradientide meetodiga

```

1  ! Fail: mpi_CG/par_sparse_lahenda.f90
2  !
3  ! Programm, mis lahendab hõreda maatriksiga sümmeetrilisi
4  ! lineaarvõrrandite süsteeme kasutades kaasgradientide
5  ! meetodit. Prooviks lahendatakse Poissoni ülesanne
6  ! Laplace'i maatriksiga, mis tekib Dirichlet' nulliliste
7  ! rajaväärtuste korral ühikruudus ühtlase võrgu puhul.
8  !
9  program par_sparse_lahenda
10 use par_sparse_mat
11 implicit none
12 type(sparse_m) :: A
13 type(par_sparse_m) :: par_A
14 real(rk), dimension(:), allocatable :: par_x,x0,y,par_y,vastus
15 integer,dimension(:),pointer :: so_reg ! sõlmeomanike register
16 real(rk) :: t1,t2,t3,t4
17 real(rk) :: eps=1.0E-10_rk
18 integer :: ierr,p,n,solmi,i,it
19
20 call MPLINT(ierr) ! initsialiseerime MPI
21 call MPLCOMMRANK(MPICO MMWORLD,myid,ierr) ! järk myid
22 call MPLCOMMSIZE(MPICO MMWORLD,numprocs,ierr) ! prots.arv
23 master=0
24 if (myid==master) then
25   p=INT(SQRT(DBLE(numprocs))) ! p - ruutjuur protsesside arvust
26   ! mitu alampiirkonda kummaski suunas
27   if (p**2/=numprocs) then
28     print *, 'Prots. arv peab olema arvu ruut! (1,4,9,16,25,...)'
29     stop
30   endif
31   print *, 'Sisestage n (peab jaguma', p, '-ga):'
32   read *, n ! n - sõlmede arv nii x kui ka y suunas
33   print *, 'n=', n
34   solmi = n*n ! ruudukujuline piirkond
35   A=Laplace_M(n) ! A - hõre, (n*n)X(n*n) Laplace'i maatriks
36   print *, 'Otsitavaid:', solmi
37   print *, 'Maatriksis nullist erinevaid elemente:', A%nearv
38   allocate(so_reg(solmi)) ! so_reg - sõlmeomanike register, massiiv
39   ! mis annab iga sõlme protsessi numbri
40   do i=1,solmi
41     so_reg(i)=(((i-1)/n)/(n/p))*p+MOD(i-1,n)/(n/p)
42   end do
43 endif
44
45 ! Käivitame stopperi:
46 call MPI_Barrier(MPICO MMWORLD,ierr)
47 if (myid==master) then
48   t1 = MPLWtime()
49 endif
50 par_A = Loo_par_sparse_mat(MPICO MMWORLD,so_reg,A)
51 ! Paneme stopperi seisma:
52 call MPI_Barrier(MPICO MMWORLD,ierr)
53 if (myid==master) then

```

```

54     t2 = MPLWtime()
55 endif
56
57 ! Teeme testimiseks valmis juhusliku parema poole vektori
58 if (myid==master) then
59     allocate(x0(solmi),y(solmi))
60     call random_seed()
61     call random_number(x0)
62     ! x0=1.0_rk ! testimiseks
63     y=sparse_Ax(A,x0) ! süsteemi Ax=y lahendiks seega x0 !
64 endif
65 ! Käivitame stopperi:
66 call MPI_Barrier(MPI_COMM_WORLD,ierr)
67 if (myid==master) then
68     t3 = MPLWtime()
69 endif
70 allocate(par_y(omasolmi),par_x(omasolmi))
71 call hajuta_vektor(MPI_COMM_WORLD,y,par_y)
72 par_x = Kaasgradientide_meetod(par_A,par_y,eps,it)
73 if (myid.eq.master) then
74     allocate(vastus(solmi))
75 endif
76 call kogu_vektor_kokku(MPI_COMM_WORLD,par_x,vastus)
77 ! Paneme stopperi seisma:
78 call MPI_Barrier(MPI_COMM_WORLD,ierr)
79 if (myid==master) then
80     t4 = MPLWtime()
81 endif
82
83 if (myid.eq.master) then
84     call Kustuta_sparse_m(A)           ! sparse_m destruktor
85 endif
86 call Kustuta_par_sparse_mat(par_A) ! par_sparse_m destruktor
87 call Kustuta_par_sparse_mat()      ! par_sparse_mat destruktor
88
89 if (myid.eq.master) then
90     print *, 'Lahendi_maksimaalne_viga:', MAXVAL(ABS(vastus-x0))
91     print *, 'Initsialiseerimisaeg:', t2-t1
92     print *, 'Lahendamisaeg:', t4-t3, ',_kokku_',it, ',_iteratsioonid_'
93     deallocate(vastus)
94 endif
95 deallocate(par_x,par_y)
96 if (myid==master) then
97     deallocate(y,x0)
98     deallocate(so_reg) ! teistel protsessidel allokeeriti mujal...
99 endif
100 call MPI_Finalize(ierr)
101
102 contains
103     function Kaasgradientide_meetod(par_A,par_y,eps,it) result(par_x)
104         ! _____
105         ! Lahendab hajusa süsteemi Ax=y täpsusega eps
106         ! _____
107         implicit none
108         type(par_sparse_m) :: par_A
109         real(rk) :: par_y(:),eps
110         integer,intent(out) :: it ! iteratsioonide arv
111         real(rk) :: par_x(omasolmi)
112         real(rk) :: vanarho rho alpha beta tmp

```

```

113      ! automaatsed abimassiivid:
114      real(rk) :: p(omasolmi), q(omasolmi), r(omasolmi)
115      integer :: ierr
116
117      par_x=0.0_rk
118      vanarho=1.0_rk
119      r=par_y-par_sparse_Ax(MPILCOMMWORLD, par_A, par_x, nulli=.true.)
120      p=0.0_rk
121      it=0
122      do while (vanarho>eps**2.AND.it <3000)
123          tmp=dot_product(r, r) ! sisefunktsioon
124          call MPI_Allreduce(tmp, rho, 1, MPLRK, MPLSUM, &
125                          MPILCOMMWORLD, ierr)
126
127          beta=rho/vanarho
128          p=r+beta*p
129          q=par_sparse_Ax(MPILCOMMWORLD, par_A, p, nulli=.true.)
130          tmp=dot_product(p, q)
131          call MPI_Allreduce(tmp, alpha, 1, MPLRK, MPLSUM, &
132                          MPILCOMMWORLD, ierr)
133
134          alpha=rho/alpha
135          par_x=par_x+alpha*p
136          r=r-alpha*q
137          vanarho=rho
138          it = it+1
139          if (myid==0) then
140              print *, it, dsqrt(rho)
141          endif
142      end do
143  end function Kaasgradientide_meetod
144 end program par_sparse_lahenda

```

Märgime, et antud kujul tuleks toodud näites protsessorite arvuks valida kas 1, 4, 9, 16, jne (st. mingi arvu ruut). Lisame täielikkuse huvides ka toodud programmi kompilleerimiseks vajamineva [Makefile](#)-i:

Lähtetekst 9.5: Makefile toodud näiteprogrammi kompilleerimiseks.

```

# Fail: mpi_CG/Makefile

# SUN arhitektuur:
F90 = mpf95 # SUN
# FLAGS = -u -dalign -g -C # debugimiseks
FLAGS = -u -fast -dalign -xarch=v8plusa
LIBS = -lmpi -lmopt -xlic_lib=sunperf -lnsl -ls3l -lmvec

#Linux:
# F90 = ifc # Intel Fortran Compiler Linux korral
# #FLAGS = -u -g -C # debugimiseks
# FLAGS = -u -O

MOD0 = RealKind.o sparse_mat.o par_sparse_mat.o
TESTO = par_sparse_lahenda.o

PRG = par_sparse_lahenda

all: $(PRG)

$(PRG): $(MOD0) $(TESTO)

```

```
$(F90) $(FLAGS) -o $(PRG) $(MODO) $(TESTO) $(LIBS)

.SUFFIXES:      .o .f90
.f90.o:
    $(F90) $(FLAGS) -c $*.f90

clean:
    rm -f core* *.o *.mod $(PRG)
```

Kokkuvõte

Antud õppematerjal oli sissejuhatuseks paralleelprogrammeerimisele kasutades Fortran90/95 keelt ja MPI teatedastusstandardit. Märgime, et mõlemad standardid, nii Fortrani kui ka MPI standard, on tegelikult pidevas edasiarenduses. Seega on kirjeldatud materjal ise pidevas muutuses ning täiustamises. (Selle õppematerjali kirjutamise ajal on valmimas Fortran200x standard; siinkirjeldatud MPI käsud katavad vaid osa MPI-1 standardist, MPI-2 standard on valmis ning lisab mitmeid uusi kontseptsioone nagu näiteks dünaamiline protsesside tekitamine jms. Ettevalmistamisel on juba ka MPI-3 standard.) Küll aga on mõlema standardi areng pidevalt ees tegelikest, realselt olemasolevatest realisatsioonidest, st. Fortrani kompilaatoritest ning MPI teekidest. Võime öelda, et siintoodud materjalid on kindlaks baasiks ka tulevastele arengutele, moodustades mõlema käsitletud tarkvarasüsteemi põhituuma. Edasiseks materjali omandamiseks soovitame konsulteerida aga juba kirjandust, millest siin anname väikese ülevaate.

Fortran90/95 kohta võib täpsemalt lugeda raamatust [1], mis annab põhjaliku ülevaate kogu keele kohta. Raamatus [2] käsitletakse objekt-orienteeritud kontseptsiooni programmeerimisel keeles Fortran90/95. Iseseisvaks õppimiseks sobivad ka ülevaated [3,4], mis on saadaval Internetist. Teine neist ([4]) kirjeldab küll Fortran90 edasiarendust paralleelarvutitele, HPF-i (*High Performance Fortran*), kuid annab ka Fortran90 kohta üsna põhjaliku ülevaate. Konkreetsete tehniliste detailide kohta on kõige hõlpsam konsulteerida (SUNi ja Inteli) kompilaatorite manuaale [5,6,7], mis on kättesaadavad Internetist.

MPI standardist (täpsemalt MPI-1) võib lugeda raamatust [8]. Selles põhjalikus MPI käsitluses leidub muuhulgas soovitusi ning tehnikat paralleelprogrammide silumiseks. Raamatus [9] antakse sissejuhatus paralleelprogrammeerimisse kasutades MPI-d, kusjuures kirjeldatakse üsna põhjalikult ka MPI standardit ennast kasutaja seisukohalt. Ühe või teise MPI käsu kuju ja argumentide kohta saab aga teha järelepärimisi kasutades tavalist UNIXi [man](#) käsku, eeldades et MPI teegi dokumentatsioon on tööjaamale paigaldatud; vastasel korral on sama informatsioon kättesaadav ka Internetist [10].

Kirjandus

1. M Metcalf and J Reid, Fortran 90/95 Explained; ISBN: 0198505582, Oxford University Press, 1999.
2. Ed Akin, Object-Oriented Programming via Fortran90/95. Cambridge University Press, 2003.
3. [Dr. C.-K. Shene Fortran 90 Tutorial](http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html) (<http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html>).
4. [Dr. A C Marshall. HPF Programming Course Notes](http://hkusuc.hku.hk/cc/sp2/ftp/hpf/5days/courseno.ps) (<http://hkusuc.hku.hk/cc/sp2/ftp/hpf/5days/courseno.ps>).
5. [SUNi kompilaatorite dokumentatsioon](http://developers.sun.com/prodtech/cc/reference/docs/index.html) (<http://developers.sun.com/prodtech/cc/reference/docs/index.html>).
6. [Intel Fortran Programmers' Reference](http://www.intel.com/software/products/compilers/techttopics/for_prg.htm) (http://www.intel.com/software/products/compilers/techttopics/for_prg.htm).
7. [Intel Fortran Libraries Reference](http://www.intel.com/software/products/compilers/techttopics/for_lib.htm) (http://www.intel.com/software/products/compilers/techttopics/for_lib.htm).
8. W Gropp. Using MPI : portable parallel programming with the message-passing interface; ISBN: 0262571048, The MIT Press, 1994.
9. P S Pacheco. Parallel programming with MPI; ISBN 1558603395, Morgan Kaufmann Publishers, 1997.
10. [MPI käskude manuaal](http://www-unix.mcs.anl.gov/mpi/www/index.html) (<http://www-unix.mcs.anl.gov/mpi/www/index.html>).

Indeks

- abs, 44
- abstraktne andmetüüp, 18, 21, 22
- ADT, 21
- ADVANCE, 56
- all, 52, 53
- allocatable, 46
- allocate, 13, 14, **46**, 51, 85, 91–97, 100, 101
- .AND., 37
- any, 52, 53
- associated, 40
- automaatne
 - massiiv, 47

- BACKSPACE, 58

- character, 36
- close, 57
- complex, 36
- count, 52, 53
- cshift, 53, 54

- db1e, 44
- deallocate, 13, 14, **46**, 51, 85, 92, 93, 95–99, 101
- destruktor, 22
- dimension, 44
- dot_product, 52
- double precision, 18, 19

- else where, 54
- ENDFILE, 58
- eoshift, 53, 54
- .EQV., 37

- F - keel, 12
- failitöötus, 56
- .false., 18, 40
- fikseeritud lähteteksti formaat, 12
- fiktiivsed massiivargumendid
 - eeldatava
 - kujuga, 48
 - suurusega, 47
 - etteantud kujuga, 47
- formaadisümbol
 - a, 56
 - e, 56
 - f, 56
 - i, 56
- format, 55, 56
- Fortran77, 14
- Fortran9x, 12

- hõredate maatriksite kolmikformaad, 83
- hajussüsteemid (*distributed systems*), 61
- HPF - High Performance Fortran, 104
- HPF - High Performance Fortran, 12

- implicit integer, 36
- implicit none, 19, 20, **36**
- implicit real, 36
- include, 23, 30
- index, 40
- integer, 36
- intent, 24
- interface, 15, 32, 33, 48
- IOSTAT, 58

- jagatava mälu (*shared memory*) mudel, 61

- kaasgradientide meetod, 84
- kind*, 14
- klass, 22
- klassid, 18, **22**
- kommentaaride lisamine, 35
- konstruktor, 22
 - manuaalne, 24
 - sisseehitatud, 24

- LAM-MPI, 61
- Laplace'i matriks, 87
- lbound, 49
- len_trim, 38
- liidesedirektiiv, 31
- logical, 36
- loogiline tüüp, 18
- matriksite korrutamine, 52
- massiivi
 - alamindeksid, 44
 - asukohafunktsioonid, 53
 - järk, 43, 46
 - kitsendusfunktsioonid, 52
 - konstruktor, 45
 - kuju, 44, 46
 - kujumuutusoperatsioonid, 52
 - muutmise funktsioonid, 53
 - päringufunktsioonid, 48
 - suurus, 44
 - ulatus, 43, 46
- matmul, 52
- maxloc, 53
- maxval, 52, 53
- minloc, 53
- module, 14
- module procedure, 25, 26
- moodul, 17
- MPI
 - blokeerivad käsud, 71
 - destruktor, 67
 - kommunikaator, 67
 - mitteblokeerivad käsud, 77
 - realisatsioonid, 61
 - ühisoperatsioonid, 71
- MPI_Allgather, 72, 73
- MPI_Allreduce, 72, 99
- MPI_ANY_SOURCE, 68
- MPI_ANY_TAG, 68
- MPI_BAND, 72
- MPI_Barrier, 71, 72
- MPI_Bcast, 72
- MPI_BOR, 72
- MPI_BXOR, 72
- MPI_CHARACTER, 68
- MPI_Comm_Rank, 65, 67
- MPI_Comm_Size, 65, 67
- MPI_COMM_WORLD, 67, 71
- MPI_COMPLEX, 68
- MPI_DOUBLE_COMPLEX, 68
- MPI_DOUBLE_PRECISION, 68
- MPI_Finalize, 65, 67, 67
- MPI_Gather, 72
- MPI_Init, 65, 67
- MPI_INTEGER, 68
- MPI_Irecv, 77, 99
- MPI_Isend, 77, 99
- MPI_LAND, 72
- MPI_LOGICAL, 68
- MPI_LOR, 72
- MPI_LXOR, 72
- MPI_MAX, 72
- MPI_MIN, 72
- MPI_PACKED, 68
- MPI_PROD, 72
- MPI_REAL, 68
- MPI_REAL8, 68
- MPI_Receive, 65
- MPI_Recv, 68
- MPI_Reduce, 72
- MPI_Scatter, 73
- MPI_Send, 65, 68
- MPI_STATUS_SIZE, 68
- MPI_SUCCESS, 67
- MPI_SUM, 72
- MPI_Test, 78
- MPI_type, 68
- MPI_Wait, 78, 79
- MPI_Wtime(), 73
- mpicc, 69
- MPICH, 61
- mpif95, 69
- mpirun, 69
- mprun, 69
- .NEQV., 37

- .NOT.*, 37
- NULL()*, 40
- nullify*, 40
- open*, 57
- open atribuut*
 - ACCESS*
 - DIRECT*, 58
 - SEQUENTIAL*, 58
 - ACTION*
 - READWRITE*, 57
 - FORM*, 57
 - FORMATTED*, 57
 - UNFORMATTED*, 57
 - IOSTAT*, 58
 - POSITION*
 - APPEND*, 58
 - ASIS*, 58
 - REWIND*, 58
 - STATUS*
 - OLD*, 58
 - REPLACE*, 58
 - SCRATCH*, 58
 - UNKNOWN*, 58
- OpenMP*, 61
- optional*, 27
- .OR.*, 37
- pack*, 52
- parameter*, 19
- pointer*, 36, **40**, 41, 46, 84, 91, 93, 100
- polümorfism*, 17, **25**
- present*, 49
- print*, **55**
- private*, 17, 22, 27
- product*, 52, 53
- public*, 17, 22, 27
- PVM*, 61
- read*, **55**, 57
- real*, 36
- real*8*, 36
- recursive*, 28
- rekursioon*, 14, 27
- request*, 77
- reserveeritav massiiv*, 46
- reshape*, 45, 50, 52
- result*, 29, 30
- REWIND*, 58
- select case*, 37
- selected_int_kind*, 18
- selected_real_kind*, 84
- selected_real_kind*, 19, 34
- shape*, 44, 45, **48**
- sin*, 44
- sisend-väljund*, 54
- size*, 44, 45, **48**, 49, 50
- spread*, 52
- standardsisend*, 55
- standardväljund*, 55
- sum*, 52, 53
- SUN-MPI*, 61
- tag*, 62
- target*, 40, **40**, 41
- teatedastusmudel*, 61
- transpose*, 53
- trim*, 38
- .true.*, 18, 40, 52
- tuletatud andmetüübid*, 13, 14, 18
- tupik*, 79, 80
- type*, 20, 21, 23, 25, 26, **36**, 84, 85, 90, 91, 93, 96, 98, 100, 101
- ubound*, 49
- ujukomaoperatsioonide kiirus*, 9
- use*, 19
- use mpi*, 67
- vaba lähteteksti formaat*, 14
- viit*, 40
- where*, 14, 43, **54**
- write*, 30, **55**, 56
- .XOR.*, 37