TARTU ÜLIKOOL

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut
Infotehnoloogia eriala

Tõnu Jaarma

Ülevaade objektorienteeritud paralleelprogrammeerimise raamistikust Charm++

Bakalaureusetöö (6 EAP)

Juhendajad: Oleg Batrašev
prof. Eero Vainikko

Autor: …………………………………………………… "……" mai 2012

Juhendaja: …………………………………………………… "……" mai 2012

Juhendaja: …………………………………………………… "……" mai 2012

Lubada kaitsmisele
Professor: …………………………………………………… "……" mai 2012

TARTU 2012

Sisukord

Sissejuhatus 3
1 Paralleelprogrammeerimise mõte 6
1.1 Sõnumi saatmise liides 8
1.2 MPI blokeeruvad funktsioonid 9
1.2.1 Saatmise ja vastuvõtmise funktsioonid 9
1.2.2 Kollektiivside funktsioon 10
1.2.3 Andmete jagamise funktsioon 12
1.2.4 Andmete kogumise funktisoon 12
1.3 MPI mitteblokeeruvad funktsioonid 13
2 Charm++ iseärasused 15
2.1 Mõisted, mida töös kasutatakse 15
2.2 Arhitektuur 16
2.3 Charm++ ülesehitus 17
2.4 *Chare* objektid 18
2.5 *Chare* objektide kogum 19
2.6 Chare grupid 20
2.7 Sõnumid 20
2.8 *Proxy* liides 21
2.9 Sisendmeetod 22
3 Programmeerimine Charm++ raamistikus 23
3.1 Charm++ struktuur 23
3.2 Charm++ käivitamise mudel 23
3.3 Kompileerimine 24
3.4 Charm++ programmide käivitamine 25
3.5 Charm++ programmi iseärasused 25
Kokkuvõte 27
Summary 28
Kasutatud kirjandus 29
Lisad 31

Sissejuhatus

Eksisteerib palju võimsaid superarvuteid ja samas ka keerulisi algoritme, mis tõesti võimsust vajavad. Probleem seisneb selles, et riistvara ei jõua tarkvara arenguga sammu pidada ja tihti tekib vajadus suuremale jõudlusele. Lahenduseks oleks ülesannete või tööde jagamine olemasolevate ressursside vahel. Näiteks oleks väga keeruline luua protsessor, mis oleks 1000 GHz, kuid pole probleem luua tuhat 1 GHz protsessorit. Paralleelarvutuste mõte seisnebki selles, et ei peaks eksponentsiaalselt võimsamaid riistvara komponente looma, vaid kasutada efektiivselt olemasolevaid.

Käesoleva töö eesmärgiks on anda ülevaade paralleelprogrammeerimise raamistikust Charm++. Kuid enne selle raamistiku ülevaate andmist, kirjeldatakse paralleel-programmeerimisest üldisemalt. Tuuakse näide maatriksite korrutamisest jagatud protsessorite peal ja seda kahe mudeli peal: MPI ja Charm++. Charm++ on objektorienteeritud paralleel-programmeerimise raamistik, mis baseerub C++ keelel, kuid lihtsamalt võib öelda, et Charm++ on teek, mis lubab C++ objektidel suhelda omavahel efektiivsemalt. Charm++ raamistikule pani aluse L.V. Kale koos oma arendaja tiimiga paralleelprogrammeerimise laboratooriumis. Umbes sada inimest on panustanud sellesse projekti keskmiselt 15 aasta jooksul. [1]

Charm++ raamistikku kasutades on näiteks ehitatud järgmised suured rakendused:

- NAMD – *Not Another Molecular Dynamics programm*, mis on dünaamiline molekulaar simulatsiooni pakett. [2]
- *Center for Simulation of Advanced Rockets* kasutab füüsilise simulatsiooniks koodi, mis kasutab samuti Charm++ raamistikku. [3]

Charm++ programmeerimise mudel sarnaneb CORBA, Java RMI või näiteks RPC mudelile, kuid siiski on see teek mõeldud suure jõudlusega masinatele, mis on disainitud paralleelarvutuste läbiviimiseks. Charm++ kasutab „üks programm, palju andmeid“ (SPMD – *single program, multiple data*) programmeerimise mudelit, mille tegi kuulsaks MPI (*Message Passing Ingerface*). [1]

Sõnumi saatmise liides ehk MPI ilmus esimesena standardi nime all 1994 aastal ja selle üks põhilisi eesmärke on täita paralleelarvutusi jagatud ressursside vahel. MPI funktsionaalsus on äärmiselt ulatusrikas ja pakub programmeerijale:

- Sidet punktist punkti (*Point-to-point communication*)
- Kollektiivset sidet (*Collective communication*)
- Ühepoolset sidet (*One-sided communication*)
- Paralleelset sisendit ja väljundit (*Parallel I/O*)
- Dünaamilist protsessi juhtimist (*Dynamic process management*) [7]

Charm++ eelis MPI ees on see, et eksisteerib keeruline iseärasus lihtsustamaks rakendusest sõltumatut objekti ümberasustamist (*application-independent object migration*). MPI standardit kasutades on see samuti võimalik, kuid nõuab keerulist programmeerimist ja pidevat koormuse jälgimist. Charm++ parim iseärasus seisneb selles, et seda raamistikku võib võtta kui lisandit MPI standardile, sest eksisteerib AMPI (*Adaptive Message Passing Interface*), mis lubab MPI koodi jooksutada otse Charm++ raamistikku kasutades. [1]

Järgnevas töös antakse põhiline ülevaade mõlemast paralleelprogrammeerimise tehnikast. MPI standardit käsitletakse otse koodinäidetega ja seletatakse lahti iga funktsiooni tähtsus ühes tavalises paralleelselt käivitatavast programmist. Charm++ raamistikule lähenetakse rohkem teoreetilisema osaga, kus alguses pannakse rõhku struktuurile ja lõpus antakse ülevaade, kuidas programmeerida Charm++ programm.

Selles töös kirjeldatakse ka Charm++ struktuuri, mida saab kokku võtta viie sõltumatu objekti kategorisatsiooniga:

1. Järjestikused objektid (nagu ka C++)
2. *Chare* objektid – samaaegselt käivituvad objektid
3. *Chare* järjend – kogum chare objektidest ja neid objekte saab järjendis individuaalselt käivitada
4. *Chare* grupid – kogum *chare* objektidest, mis on iga protsessori grupi liige
5. Sõnumid – infovahetusega tegelevad objektid [7]

Töö koosneb kirjalikust ja praktilisest osast. Kirjalik osa on rohkem teoreetilisem ja ära jagatud kolme peatüki vahel. Kirjaliku osa eesmärgiks on anda kindel ülevaade paralleel-programmeerimisest kasutades selleks kahte erinevat lähenemist. Töö praktiline osa koosneb kahest paralleelselt käivitatavast programmist, kus esimene on kirjutatud MPI standardit kasutades ja teine Charm++ raamistikku. Lisatud on ka õpetused, kuidas endale sellised keskkonnad paigaldada, et sooritada samu paralleelarvutusi.

Esimeses peatükis antakse ülevaade paralleelprogrammeerimise kasutusest tänapäeval ja kuidas kirjutada koodi, mis arvutaks paralleelselt. Esmalt kirjeldatakse populaarsemat paralleelprogrammeerimise stiili, milleks on MPI standard ja selles stiilis koode kirjutatakse C keeles. Kui tavalist koodi käivitatakse järjest ja iga funktsioon ootab oma täitmise lõpetamist, siis mõistlikum on kirjutada mitteblokeeruvaid funktsioone paralleelprogrammeerimisel.
Näha on ka kuidas tekivad lihtsad järjekorra probleemid, ehk ei ole teada kus ja millised andmed millises järjekorras töödeldakse, kui just ei kasutata mingeid sünkroniseerimis-tehnikaid.

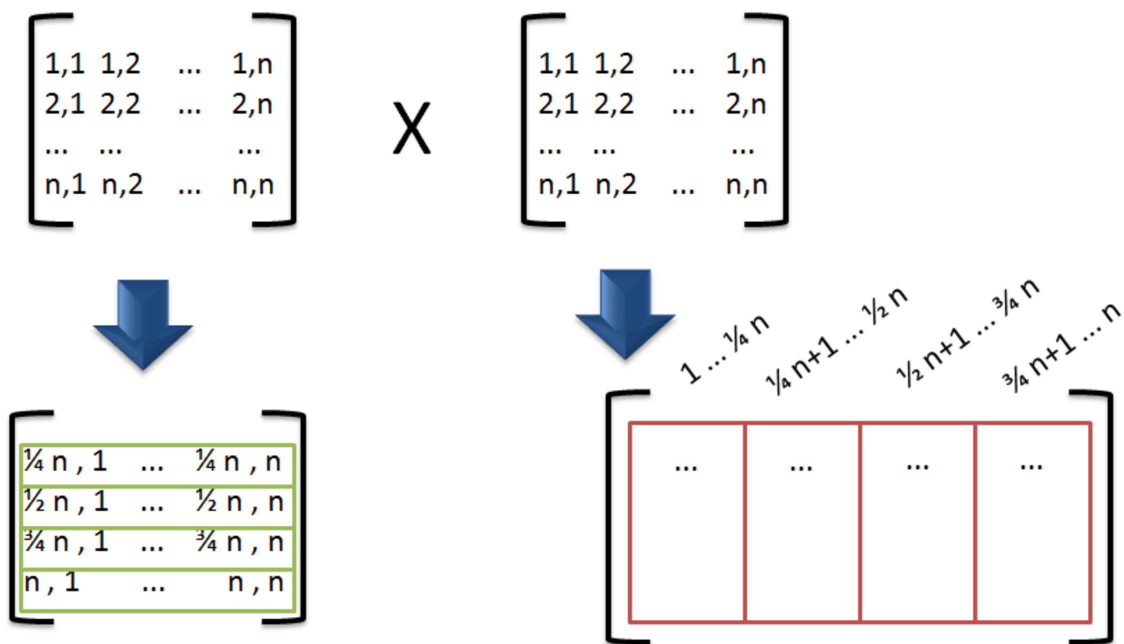
Teises peatükis kirjeldatakse selle töö põhilist osa – ülevaadet Charm++ raamistikust. Ülevaate andmiseks kirjeldatakse raamistiku ülesehitust, struktuuri ja kindlasti ka iseäraseid tunnuseid. Alustatakse väikseimatest objektidest ja sealt edasi kirjeldatakse juba nende objektide järjendit ning kuidas need objektid suhtlevad omavahel. Seal tulevad välja ka Charm++ raamistiku erinevused võrreldes MPI liidesega.

Kolmandas peatükis muutub teoreetiline osa natukene praktilisemaks. Kirjeldatakse kuidas Charm++ programm käivitatakse ja kuidas kulgeb selle käivitusprotsess. Lihtsate näidete abil demonstreeritakse, kuidas programmeerija ei pea teadma tervet käitussüsteemi, vaid piisab teadmisest, kuidas luua objekte, mida saab ümber migreerida.

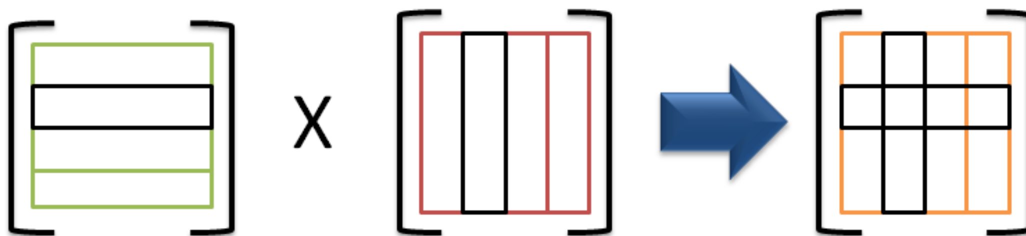
Lisana on välja toodud kaks programmi: C keeles kirjutatud maatriksite korrutamine (MPI liidese demonstratsioon) ja C++ keeles kirjutatud maatriksite korrutamine (Charm++ raamistiku demonstratsioon). Mõlemad programmid kasutavad sama algoritmi paralleelselt arvutamiseks ja mõlemad on kirjutatud mitteblokeeruvaid funktsioone kasutades. Samuti on lisana esitatud õpetused, kuidas paigaldada vastav keskkond ja neid programme paralleelselt käivitada.

1 Paralleelprogrammeerimise mõte

Programme saab paralleelseks teha mitmel erineval viisil ja neist kaks kõige kasutavamat on andmete paralleelsus ja ülesannete paralleelsus. Andmete paralleelsus on kergemini saavutatav ja seega ka enim kasutatav. [6] Ka lisades olev maatriksite korrutamise näide käib pigem andmete paralleelselt töötlemise alla. Selle asemel, et tervet maatriksit mingile protsessorile saata, saadetakse vaid seda, mida on vaja teada, et sooritada mingi konkreetne operatsioon. Maatrikstite korrutamises jagatakse maatriksid ridadeks ja veergudeks, sest üks koht tulevases maatriksis leitakse vaid sellele vastava esimese maatriksi rea ja teise maatriksi veeru korrutiste summana (Joonis 1). Kõige mõistlikum oleks näidata, kuidas töötab lisades olev Charm++ programm, sest see maatriksite korrutamine on lahendatud andmete paralleelselt töötlemise tulemusel (Joonis 2). Tegelikult programmis ei arvutata konkreetselt ridade ja veergudega, vaid teine maatriks transponeeritakse, et saadud transponeeritud maatriksis saaks arvutusi sooritada ridadega. Tulemuseks oleks kahe maatriksi ridade korrutiste summa, mis on kergemine programmeeritav ja samuti ka kergemini loetav (lõpptulemust see ei muuda, endiselt on tegu kahe algse maatriksi korrutisega).



Iga protsessor saab ühe rea ja veeru ning arvutab tulemus maatriksis vastava koha:



Joonis 1. Maatriksite paralleelselt korrutamise mõte.

1. Protsessor

Read ja veerud saadetakse laiali

1. Protsessor 2. Protsessor 3. Protsessor 4. Protsessor

Iga protsessor arvutab oma rea ja veeruga.

Tulemuseks on korrutatud maatriksi diagonaal.

Rida jäetakse alles ja saadetakse naabritele oma veerg.

Pärast 1. tsüklit Pärast 2. tsüklit Pärast 3. tsüklit

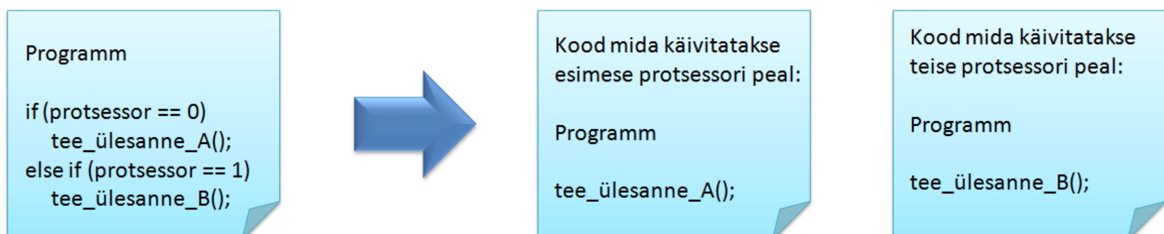
Iga protsessor sooritab arvutused oma rea ja saadud veeruga

Tulemused kogutakse kokku.

Tulemuseks on täidetud maatriks.

Joonis 2. Maatriksite korrutamine paralleelselt.

Teine erinev meetod paralleelsuse saavutamiseks on ülesannete sooritamine paralleelselt. Kus siis näiteks protsessor A teeb ainult teatud tüüpi ülesandeid ja protsessor B teist tüüpi ülesandeid (Joonis 3). [5]



Joonis 3. Ülesannete paralleelsus.

1.1 Sõnumi saatmise liides

Üks populaarsemaid paralleelarvutamiseks kasutatavaid liideseid on sõnumi saatmise liides ehk MPI (*Message Passing Interface*). MPI ilmus esimesena standardi nime all 1994. aastal ja selle liidese üks põhilisi eesmärke oli täita paralleelarvutusi, kasutades hajusat mälu ja tänu sellele sai see liides kõige populaarsemaks paralleelarvutuste sooritamisel.

Sõnumiedastus mudel on paralleelprogrammeerimise mudel, kus protsessid saavad andmeid jagada ainult sõnumeid edastades. Kui üks protsess tahab andmeid edastada teisele, siis esimene protsess algatab saatmise funktsiooni, edastab sõnumit ja samal ajal kui sõnumit edastatakse blokeerub protsess seniks kuni sõnum on täies mahus edastatud. Teine protsess peab algatama sõnumi saamise funktsiooni ja ootama seni, kuni sõnum on endale täies ulatuses ära kopeeritud.

Sellisel sundühenduste loomisel on mitmeid eeliseid paralleelprogrammeerimisel, näiteks sõnumeid saab üle väga suure ala edastada. MPI programm saab töötada arvutitel sõltumata nende asukohtadest. Programmide silumine ja vigade otsimine on samuti mugav, sest kaob põhjus muretsemiseks, et üks protsess kirjutaks üle mingi võõra mälu aadressi. Eranditest antakse ülevaade peatükis 1.3, kus räägitakse mitteblokeeruvatest funktsioonidest. [7]

MPI liidesel on mõningad klassikalised mõisted, mis rõhuvad selgele paralleelsele programmi disainile. Esmalt on vaja aru saada edasiandja või kommunikaatori mõistest (*comminicator*). Kommunikaator määrab ära grupi protsesse, millel on omadus suhelda omavahel. Selle grupi igale liikmele omistatakse unikaalne järk (*rank*) ja omavahel suheldakse oma unikaalset järku kasutades.

Kommunikatsiooni vundament on üles ehitatud lihtsatele saatmise ja vastuvõtmise toimingutele. Protsess võib saata sõnumi teisele protsessile andes oma järgu ja unikaalse sildi (*tag*), et sõnum oleks identifitseeritav. Vastuvõtja saab siis tagasi saata kinnituse, et sõnum on kätte saadud, kasutades seda sama unikaalset silti ja pärast seda täita tööjuhiseid. Just selliseid kommunikatsioone, kus on üks saatja ja üks vastuvõtja, nimetatakse sidet punktist punkti (*point-to-point communication*). Leidub ka teisi variante, kus näiteks üks protsess tahab suhelda kõigi ülejäänutega. Sellisel juhul oleks koormav kirjutada koodi, mis tegeleks kõikide saatjatega ja vastuvõtjatega. See isegi ei kasutaks võrku optimaalselt ära. MPI saab hakkama igasuguste variantidega sellist tüüpi kollektiivse side loomisega (*collective communication*), mis hõlmaks kõiki protsesse (nendest pikemalt järgnevates funktsioonide kirjeldustes). [7]

1.2 MPI blokeeruvad funktsioonid

Saatmise ja vastuvõtmise funktsioonid on kaks fundamentaalset kontseptsiooni MPI liideses. Peaaegu kõiki MPI funktsioone saab rakendada baas saatmise ja vastuvõtmise kutseid kasutades. Eranditeks on ainult ülesanded, mis on oma mahult nii suured, et andmeid ei saa hoida ühes masinas (lahenduseks on mitteblokeeruvad sõnumite edastused, kuid sellest pikemalt peatükis 1.2.5). Lihtsamalt öeldes on tegu saatmisfunktsiooniga, mis ei oota et kogu sõnum oleks ära saadetud, vaid täidab juba ülesandeid.

MPI saatmine ja vastuvõtmine käib järgnevalt:

1. Esmalt protsessis A määratakse andmed, mida tahetakse saata protsessile B.
2. Protsess A seab kõik vajaminevad andmed oma puhvrisse, et saata B protsessile (neid puhvreid nimetatakse ka ümbrikuteks, sest kõik andmed pannakse ühte faili enne saatmist).
3. Järgneb paki kohaletoimetamine, tavaliselt vastutab selle eest mingi sidevahend (tihti on selleks võrk) ja kohad kuhu saadetakse on järgu (*rank*) defineeritud.
4. Nüüd kui sõnum on jõudnud protsess B kätte, peab B ikkagi kinnitama, et ta on nõus neid andmeid vastu võtma.
5. Kui protsess B nõustub, siis loetakse andmed edastatuks ja protsess B saadab A protsessile tagasi kinnituse.
6. Kui kinnitus on vastu võetud, siis saab protsess A oma blokeeruvast positsioonist välja minna ja jätkata tööga.[8]

1.2.1 Saatmise ja vastuvõtmise funktsioonid

MPI liideses peab kirjeldama saatmise ja vastuvõtmise funktsioone *MPI_Send* ja *MPI_Recv* funktsioonidena:

- MPI_Send(void *buffer, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator) [9]
- MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator, MPI_Status* status) [10]

Peaaegu iga MPI funktsioon on sarnase ülesehitusega ja kasutavad samu argumente:

- Void *buffer – andmete puhver
- int count – määrab ära kui palju andmeid saadetakse
- MPI_Datatype datatype – määrab ära, milliseid andmeid saadetakse
- int destination – määrab ära, millisele protsessile andmed saadetakse
- int tag – identifitseerimiseks, kui ühele protsessile saabub mitu teadet, siis protsess on võimeline nende vahel vahet tegema

- MPI_Comm communicator – määratakse ära kõik võimalikud protsesside ruum, kuhu on võimalik andmeid saata ja vastu võtta
- MPI_Status* status – pakub informatsiooni saadud sõnumist. Neid on põhiliselt nelja tüüpi:
    - MPI_STATUS_IGNORE – mis ei tagasta informatsiooni
    - MPI_SOURCE – tagastab protsessi, kust see sõnum tuli
    - MPI_TAG – tagastab identifikaatori, mis määrati sellele sõnumile
    - int* count – tagastab lihtsalt kogu andmete mahu arvu

Staatuse kohta informatsiooni saamine on oluline sellepärast, et *MPI_Recv* funktsioon ehk saaja, võib võtta ükskõik milliselt protsessilt sõnumit (*MPI_ANY_SOURCE*) ja ükskõik millist identifikaatoriga sõnumit (*MPI_ANY_TAG*). Selline uue töö kohene vastuvõtmine, kui vana töö on sooritatud, tõstab kõvasti efektiivsust. Olgu siia vahele veel öeldud, et *MPI_Send* funktsioon saadab täpselt nii palju elemente, kui ette määratud ja *MPI_Recv* funktsioon võtab vastu maksimaalselt nii palju elemente, kui palju ette antud.

1.2.2 Kollektiivside funktsioon

Peale saatmise ja vastuvõtmise funktsioonide, pakub MPI ka andmete saatmist korraga kõigile protsessidele. Seda kutsutakse laiali saatmiseks (*Broadcasting*) ja on standardne kollektiivse side tehnika. Kui üks protsess kutsub kollektiivset side, siis ta saadab täpselt samad andmed kõigile teistele protsessidele. Selles mõttes on tegu dünaamilise saatmisega, sest saadetakse kõigile ülejäänutele automaatselt, ilma lisaprogrammeerimiseta. [9]
Kollektiivse side põhiliseks eesmärgiks võib pidada näiteks kasutaja sisendi saatmist paralleelsele programmile või lihtsalt mingi konfiguratsiooni parameetri saatmist kõigile protsessidele. Funktsioon näeb välja järgmine: [8]

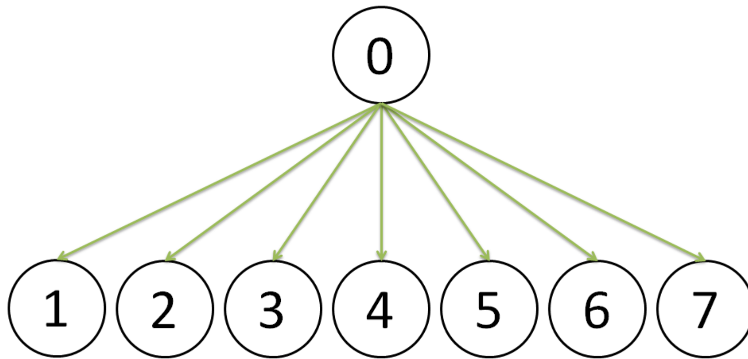
- MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm communicator) [11]

Nagu näha siis tõesti MPI funktsioonid sarnanevad saatmis ja vastuvõtmis baas-funktsioonidele. Ainus erinevus *MPI_Bcast* funktsioonil on see, et ei määrata ära, millistele protsessidele see sõnum läheb, sest niikuinii kõik protsessid, kes kutsuvad seda sama funktsiooni, saavad määratud andmed. Näiteks:

```
Muutuja = 5;
MPI_Bcast(Muutuja, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

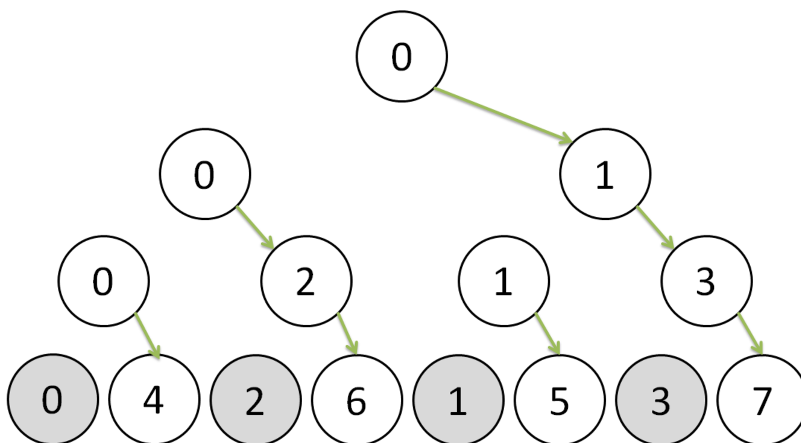
Siin näites saadetakse „Muutuja“ (mille väärtus on 5) kõigile, kes kutsuvad *MPI_Bcast* funktsiooni. Teine argument on antud juhul 1, sest on ainult üks number mida saadetakse. Null seal neljandal argumendi kohal määrab ära, mis protsess on vastutaja ehk põhiline protsess (*Master* või *Root*). *MPI_COMM_WORLD* on vaikimisi defineeritud protsesside ruumistik. See tähendab seda, et kasutatakse kõiki protsesse, mida kasutusele määratakse.

Kui nüüd vaadata, milliseid eeliseid see *MPI_Bcast* funktsioon pakub peale selle, et seda on kompaktsem kirjutada. Kujutame ette, et meil on üks sisse- ja väljaminev võrguühendus ning iga ühendus võtaks aega täpselt 1 sekund, siis *MPI_Send* ja *MPI_Recv* funktsioone kasutades läheks meil täpselt 7 sekundit. Protsess 0 saadab esimesele, siis teisele, siis kolmandale jne. kuni seitsmendale (Joonis 4. *MPI_Send* ja *MPI_Recv*).



Joonis 4. *MPI_Send* ja *MPI_Recv* funktsioon.

MPI_Bcast funktsioon saaks selle ülesandega hakkama kõigest kolme sekundiga. Kui protsess null kutsub välja *MPI_Bcast* funktsiooni, siis see saadetakse esmalt esimesele protsessile (esimene sekund), siis protsess saadab edasi teisele protsessile, kuid samaaegselt on äratatud ka protsess üks ja ta saab samaaegselt saata edasi kolmandale protsessile (teine sekund). Sama moodi jätkates saadab null protsess neljandale (sest eelnevatele on juba saadetud), protsess üks saadaks viiendale ja samaaegselt on teine ja kolmas protsess ka valmis saatma ja nad saadavad viimasele kahele (Joonis 5. *MPI_Bcast* funktsioon).



Joonis 5. *MPI_Bcast* funktsioon.

Seega juba teoorias saame järeldada kiiruste vahe, kui näiteks ühenduse loomine ja andmete saatmine kestab üks sekund: *MPI_Bcast* funktsiooni kasutades saadetakse info 2^n protsessile n sekundiga ning *MPI_Send* ja *MPI_Recv* funktsioone kasutades saadetakse info 2^n protsessile 2^n - 1 sekundiga.

1.2.3 Andmete jagamise funktsioon

Järgmisena tuleb käsitlusele MPI funktsioon, millele leidub rohkem kasutust kui *MPI_Bcast* funktsioonil. Selleks on *MPI_Scatter* funktsioon, mille põhiline eesmärk on jagada andmekogum teatud osadeks ja seda laiali saata teistele protsessidele.

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

Kus siis argumentide tähendus sarnaneb *MPI_Send* ja *MPI_Recieve* funktsioonidele:

- Sendbuf – saadetava puhvri aadress
- Sendcount – elementide arv, kui palju saadetakse igale protsessile
- Sendtype – millist tüüpi elemente saadetakse
- Recvcount – elementide arv vastuvõetavas puhvris
- Recvtype – millist tüüpi elemente võetakse vastu
- Root – milline protsess saadab andmekogust moodustatud juppe teistele protsessidele
- Comm – kommunikaator

Olgu juurde veel mainitud, et *sendbuf* argumente ignoreerivad kõik protsessid peale juht-protsessi. Lisaks veel, et signatuur, mis on seotud „*sendcount*“ ja „*sendtype*“ argumentidega, peab olema sama signatuuriga, mis on seotud „*recvcount*“ ja „*recvtype*“ argumentidega. Seda sellepärast, et tagada sama arv andmete saatmist ja vastuvõtmist, et midagi ei tuleks üle ega jääks puudu. [12]

Oletame, et meil on 4 protsessi ja defineeritud 8 elemendiline jada numbritest:

```
Muutujad = [1,2,3,4,5,6,7,8]
MPI_Scatter(Muutujad, 2, MPI_INT,  vastuvõetav_muutuja, 2, MPI_INT, 0,    MPI_COMM_WORLD);
```

Nüüd on igale protsessile defineeritud uus muutuja: „vastuvõetav_muutuja“. Sellel muutujal on vastavad väärtused: esimesel protsessil on 1 ja 2, teisel 3 ja 4 jne. Igale protsessile saadetakse 2-st elemendist koosnev jada ja iga protsess võtab vastu kahest elemendist koosneva jada – 8 elementi jaotakse kõigi 4 protsessi peal võrdselt.

1.2.4 Andmete kogumise funktsioon

MPI_Scatter funktsioonist ei ole kasu, kui pole defineeritud ka *MPI_Gather* funktsioon, mille mõte on andmeid kinni püüda protsessidelt. Iga protsess, millele on *MPI_Scatter* andmeid saatnud peab kutsuma välja ka *MPI_Gather* funktsiooni, et saata mingid tulemused peaprotsessile tagasi.

- MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) [13]

Kui nüüd võtame eelmise näite ja oletame, et andmed on protsessidele laiali saadetud ja nüüd oleks vaja uued muutjad kokku võtta, siis võime tuua illustreeriva näite:

```
uued_numbrid = vastuvõetav_muutuja + 1;
MPI_Gather(uued_numbrid,2,MPI_INT,kogutud_tulemused,2,MPI_INT,0,MPI_COMM_WORLD);
```

Siis saaksime:

```
kogutud_tulemused = [2,3,4,5,6,7,8,9]
```

1.3 MPI mitteblokeeruvad funktsioonid

Nende eelnevate funktsioonidega oleks võimalik peaaegu kõik paralleelsust nõudvad ülesanded ära lahendada, kuid nüüd tulevad mängu jõudlusprobleemid. Demonstreerimaks esmaseid jõudluse ja mälu probleeme, on näiteks käsitletud maatriksite korrutamist. Kui kasutada ainult *MPI_Bcast*, *MPI_Scatter* ja *MPI_Gather* funktsioone, siis on täiesti võimalik ära lahendada maatriksite korrutamine paralleelselt. Kus siis ühte maatriksit saadetakse kõigile protsessidele ja teist maatriksit jagatakse täpselt nii mitmeks osaks, kui mitmel protsessil nad jooksma peaksid. Tõesti toimub maatriksite korrutamine kiiremini kui ühe protsessi peal, kuid see ei oleks efektiivne.

Me saaks efektiivsemaks siis, kui saadetakse igale protsessile vaid seda, mida neil parasjagu vaja läheb. Näiteks *MPI_Scatter* funktsiooniga mõlemad maatriksid laiali saata. Nii saaks iga protsess parajasti vaid ühe veeru ja ühe rea, et need korrutiste summana tagastada. Edasi tuleks igal protsessil oma veerg edasi naabrile saata, et kõik veerud saaks kõigi ridadega korrutatud. Tõesti see süsteem toimib blokeeruvaid MPI funktsioone kasutades, kuid kahjuks vaid kuni teatud piirini. Blokeerumine toimub sellepärast, et mõni protsess on saatmisega hakkama saanud ja jääb ootama naabrilt uut veergu ja blokeerub seniks kuni saab uue veeru. Probleem seisneb selles, et kood ei lähegi sealt edasi, et vastavat veergu saata, vaid on ootamise taga kinni.

Lahenduseks on mitteblokeeruvad MPI funktsioonid, millest üks populaarsemaid on *MPI_Isend*:

- int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request) [14]

MPI_Isend töötab nii, et protsess hakkab määratud aadressil olevat andmestikku saatma ja samas programm läheb edasi koodi käivitamisega. *MPI_Isend* funktsiooniga tekib esmapilgul väga palju probleeme, millest esimesena hakkab silma just see, et andmeid saadetakse mingilt mäluaadressilt, kuid võib juhtuda, et sinna samale mäluaadressile kirjutatakse samaaegselt andmeid – tekib mälu ülekirjutamine ja vastus pole kunagi korrektne. Selliseid olukordi saab vältida rakendades *MPI_Wait* funktsiooni:

- MPI_Wait(MPI_Request *request, MPI_Status *status) [15]

See tähendaks seda, et protsess ootab kuni *MPI_Recv* on saanud täiel määral oma andmed kätte (selleks ajaks on programm blokeerunud). Edasi järgneb programmi blokeerumisest vabastamine ja koodi täitmist jätkatakse.

Kõige mõistlikum oleks nüüd lisada veel üks muutuja – ajutine puhver. Et ei tekiks kunagi sellist võimaluski, et kellegi andmeid võidakse üle kirjutada. *MPI_Isend* saadab alati ühelt mäluaadressilt, *MPI_Recv* kirjutab samaaegselt teisele mäluaadressile ja kui on vajalikud muudatused tehtud, siis vabastatakse teine ajutine mäluaadress ja kirjutakse see esimesele.

2 Charm++ iseärasused

2.1 Mõisted, mida töös kasutatakse

Chare – Charm++ klassis defineeritud objekt, mis sisaldab:

1. mingeid andmeid
2. saadetavaid ja saadavaid sõnumeid
3. sisendmeetodi käivitamist (täidetakse mingeid ülesandeid vastavalt saadud sõnumile)

Main chare – rakenduse käivitamine algab sellest spetsiaalses *chare* objektis.

Chare arrays – see on põhimõtteliselt järjend *chare* objektidest. Sõnumi saatmine käib ühele järjendi objektile viidates ja sihtpunkti määrates (objekt[mis_kohal].kuhu_saata()).

Proxy – on lokaalne C++ klass, mis esindab eemalt käivitavat C++ klassi.

Entry method – Charm++ sisendmeetod on meetod, mis käitub nagu sõnumite vastuvõtupunkt (kirjutatakse nagu tavalist meetodit C++ keeles).

Charmc – on tööriist mille läbi toimub Charm++ koodi kompileerimine.

Charmrun – programm millega käivitatakse Charm++ koodi.

Message driven – ülesandeid täidetakse sõnumi saabumise järjekorras.

Charm++ Runtime System (teisiti ka *Charm Kernel*) – on Charm++ käitussüsteem. mis tegeleb arhitektuuri võimalikult efektiivse ärakasutamisega, samas ka toetab kirjutatud ja antud platvormil töötavat tarkvara.

Dynamic load balancing framework või siis *LB Framework* – on raamistik, mille ainus eesmärk on jagada erinevaid töid erinevate protsessorite vahel, et tasakaalustada koormust.

Marshalled parameter – enne, kui mingi meetodi parameetreid hakatakse saatma, kodeeritakse need ümber sõnumiteks (andmevahetus toimub ainult sõnumite saatmisega) ja neid ümberkodeeritud parameetreid nimetaksegi *marshalled parameters*).

Mainmodule – põhimoodul, mis on kättesaadav kõikidele ülejäänud moodulitele.

entity – süsteemi või valdkonna konkreetne või abstraktne komponent.

2.2 Arhitektuur

Charm++ on objektorienteeritud paralleelprogrammeerimise raamistik. Charm++ erineb teistest traditsioonilistest sõnumiedastuse ja jagatud muutujatega programmeerimise mudelitest selle poolest, et Charm++ on sõnum orienteeritud[1]. See tähendab, et mingi töö Charm++ raamistikus käivitatakse alles siis, kui vastav sõnum seda ütleb. Kasulik on see sellepärast, et iga ülesandeid täitev protsessor saab saata veel omakorda sõnumeid teistele protsessoritele (kasvõi iseendale) saavutades sellega rohkem paralleelsust. Selle mõttekäik sarnaneb *MPI_Send* ja *MPI_Bcast* kiiruste analüüsile (Joonis 4. ja Joonis 5.). Sellele, et mudel on sõnumorienteeritud, tuleb kasuks üks olulisemaid arhitektuurilisi iseärasusi Charm++ raamistikus, milleks on asünkroonne sõnumi edastus. Asünkroonne tähendab seda, et objektid (esialgu lihtsalt objektid, hiljem Charm++ raamistikule omased objektid – *chare* objektid) suhtlevad omavahel saates sõnumeid mitteblokeeruvalt. Mitteblokeerumise all mõeldakse siin seda, et kui kood on täitmisega jõudnud objekti saatmiseni, siis saadetakse vastav objekt sõnumiga ja tagastust ei oodata – kood jätkab täitmist. Funktsioon on oma struktuurilt sarnane *MPI_Isend* funktsioonile.

Järgmiseks heaks omaduseks on see, et rakendused ei sisalda töötlevate elementide kirjeldust, näiteks mitu *chare* objekti ühele töötlevale elemendile määratakse. Lisaks ei ole vaja määrata, kuidas andmeid omavahel vahetatakse. Kõik need detailsed probleemid lahendab Charm++ käitussüsteem[2] (*Charm++ Runtime System*). Süsteem hoolitseb selle eest, et kõik *chare* objektid saaks kõikide füüsiliste töötlevate elementide vahel ära jaotatud ja lisaks marsruutida sõnumeid üle ühenduste.

Charm++ pakub dünaamilist koormuse tasakaalustamist ja sellepärast ei pea ülesande sooritamiseks määrama protsessorit, kui soovitakse luua eemalt käivitavat *chare* objekti. Käitussüsteem valib ise *chare* objekte ja käivitab neid kõige vähem koormatud protsessoritel. Käitusmootor tuvastab iga *chare* objekti identifikaatori *ChareID* järgi ja kasutaja ei pea ise teadma, kus parasjagu mingi *chare* objekt on.

Charm++ pakub veel mõningaid mehhanisme pakkumaks programmeerijale lihtsamat programmeerimist, et kirjutatud programm kohaneks paremini dünaamilises käitusmootori[3] keskkonnas. Teisisõnu, et oleks lihtsam programmeerijal kirjutada programmi erinevatele platvormidele ja samas ka erinevatele süsteemiarhitektuuridega masinatele.

---

[1] *Message driven* – arvutusi täidetakse sõnumi saabumise järjekorras.
[2] *Charm++ Runtime System* (teisiti ka *Charm Kernel*) on Charm++ käitussüsteem, mis tegeleb arhitektuuri võimalikult efektiivse kasutamisega, samas ka toetab kirjutatud ja antud platvormil töötavat tarkvara.
[3] Käitusmootor kui vahetarkvara, mis on vajalik rakendusprogrammi käitamiseks antud tüüpi protsessoriga arvutil. Käitusmootor tõlgib rakendusprogrammi käsud protsessorile mõistetavateks masinakäskudeks.

Olulisemad mehhanismid on näiteks:

- Prioriteetide määramine – erinevatel meetoditel erinevad prioriteedid.
- Sõnumite kokku- ja lahtipakkimist – vähendamaks sõnumite suurusi.
- Puhkeoleku tuvastus – tuvastamaks programmis mõningate faaside valmisolekut.
- Dünaamiline koormuse tasakaalustamine – määratakse eemalt juhitavate objektide loomise ajal selleks, et protsessorite vahel koormust vähendada. [4]

2.3 Charm++ ülesehitus

Objektipõhine programmeerimine on ülesehitatud andmete kapseldamise ideele. Andmete kapseldamine saavutatakse, kui võetakse objektisiseselt kokku andmed ja meetodid (või siis ka funktsioonid, alamfunktsioonid ja protseduurid), kus tüüpiliselt vastutab konstruktori meetod kapseldatud andmete ja objektide käivitamise eest. Iga meetod (kaasaarvatud konstruktor), võib valikuliselt saada andmeid parameetritena või argumentidena. Iga kapseldatud andmete kogum moodustab kokku objekti ja Charm++ programm koosneb teatud arv Charm++ objektidest, mis on jagatud kõigi saadavalolevate protsessorite vahel. Seega baaselement, mis sooritab paralleelseid arvutusi, nimetatakse siin töös *chare* objektiks. *Chare* objekti saab luua igale vabale protsessorile ja lisaks saab sellele ligi eemalt juhitavast protsessorist. *Chare* objekti tähendus on sarnane *actor*[4] ja ADA ülesandega[5]. Ideeliselt võib ette kujutada, et süsteemil on kogumik töödest (*work-pool*), mis sisaldab sõnumeid olemasolevatele *chare* objektidele. Käitussüsteem valib mittedeterministlikult kogumikust mõne sõnumi ja viib selle läbi (käivitab).

Chare objektidel olevad meetodeid, mida saab käivitada eemalt, kutsutakse sisendmeetoditeks (*entry method*). Sisendmeetodid võivad võtta ümbertõlgitud parameetreid (*marshalled parameters*) või viidet sõnumi objektile. Kuna *chare* objekte on võimalik luua eemalt juhitavatele protsessoritele, seega mõningad (vähemalt üks) *chare* objektide konstruktorid peavad olema sisendmeetoditeks. Charm++ ei sega kunagi vahele käivitatud meetodile, et saaks ise alustada mingi teise tööga, eranditeks on lõimitud meetodid (*threaded*) ja sünkroonmeetodid (*synchronous*).

Charm++ on ehitatud C++ keele peale ja sarnaselt C++ keelele kasutab ka Charm++ privaatseid andmeid ja avalikke meetodeid. Erinevus seisneb selles, et neid meetodeid saab kutsuda eemalt juhitavatest protsessoritelt asünkroonselt. Asünkroonselt tähendab seda, et programm ei oota meetodi käivitamist ega lõpetamist, vaid jätkab koodi täitmist. Sellest järeldades puudub Charm++ meetoditel tagastatav väärtus. Kuna Charm++ objektid, millel meetodid käivitatakse, võivad olla teistel protsessoritel, siis C++ keeles objektile viitamine, kasutades selleks viita[6] ei kehti Charm++ raamistikus. Selle asemel on kasutusel viit läbi *proxy*. [4]

---

[4] Baseerudes *Actor* mudelile saab iga *Actor* saata, luua ja muuta käitumist vastavalt sõnumitele [19]
[5] ADA programmeerimise keeles üks ülesanne on kui objekt [20]
[6] Viit (pointer) on muutuja, mille väärtus on mäluaadress

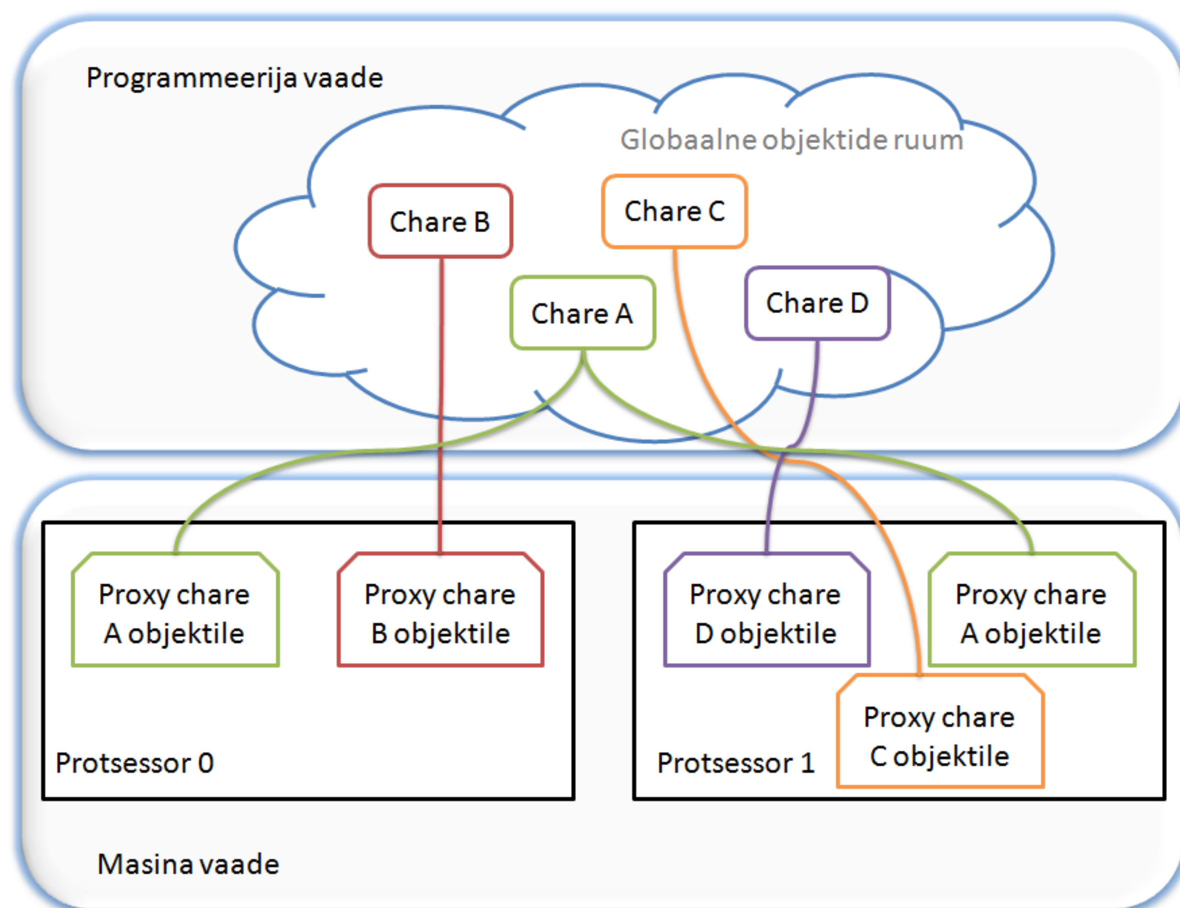
Järgnevas neljas peatükis (peatükid 2.4 kuni 2.7) kirjeldatakse detailsemalt Charm++ raamistiku fundamentaalsemaid objekte. Charm++ raamistikku saab kategoriseerida viieks ühisosata objektiks:

- järjestikused objektid nagu ka C++ keeles
- *chare* objektid
- *chare* järjendid
- *chare* grupid
- sõnumid

2.4 *Chare* objektid

Chare objektid on kõige tähtsamad elemendid Charm++ programmis. Süntaktiliselt on need samaaegselt käivitatavad objektid C++ klasside objektid, mis on tuletatud süsteemi poolt loodud klassist nimega *chare*. Võrreldes tavaliste C++ keele privaatsete ja avalike andmetega ning meetoditega, sisaldavad *chare* objektid avalikke sisendmeetodeid, mis ei tagasta midagi ja võtavad maksimaalselt ühe argumendi, milleks on sõnumi viit. Neid objekte luuakse dünaamiliselt ja rohkem kui üks *chare* objekt võib olla aktiivne samaaegselt. Seega põhiline paralleelsus saavutataksegi *chare* objektide jagamisel erinevate protsessorite peal (siin käivitub ka käitussüsteemi eriline oskus tasakaalustada erinevate protsessorite koormust jagades efektiivselt neid *chare* objekte). *Chare* objektide põhine eesmärk on omavahel suhelda, saates sõnumeid ja käivitada meetodeid. Sõnumite saatmine näeb välja samasugune nagu meetodite kutsumine C++ keeles, sest saatja objekt lihtsalt kutsub saaja objekti meetodi (peatükis 2.9 seletatakse pikemalt, miks saaja objekti meetod on sisendmeetodiks).

Nagu eelnevalt mainitud koosneb Charm++ programm teatud arv Charm++ objektidest, mis on jagatud kõigi saadavalolevate protsessorite vahel. Need kõik *chare* objektid moodustavad kokku globaalse objektide ruumi (sarnaneb MPI kommunikaatorile). Kuna neid objekte võib olla väga palju ja samuti võivad nad olla erinevates masinates, peab leiduma mingi viis, kuidas need objektid omavahel suhelda saavad. Lahenduseks on Charm++ liidese translaatori poolt loodud *proxy* liidesed ja lisaks nendele saab kasutada ka sündmusetöötlejat (*handler*), milleks võib näiteks olla *CkChareID* struktuur. Demonstreerimaks eelnevat kirjeldust, on järgmisel lehel kokkuvõttev joonis (Joonis 6). Nagu näha jooniselt, siis programmeerija tõesti ei pea teadma, kus individuaalne *chare* objekt on, vaid käitussüsteem valib ise mingid objektid ja rakendab koorma tasakaalustamist (järgmises peatükis pikemalt). Jooniselt tuleb ka välja, et sama objekti võib esindada ka mitu *proxy* liidest (peatükis 2.8 sellest pikemalt). [4]



Joonis 6. *Charm* objektide tasakaalustamine protsessorite peal.

2.5 *Chare* objektide kogum

Chare objektide kogum (*chare array*) on kogum *chare* objektidest või teisisõnu järjend *chare* objektidest. See järjend saab sisaldada väga palju elemente ja need elemendid on üksikud *chare* objektid, mida saab ühekaupa käivitada järjest või suvalises soovitud järjekorras. Igal elemendil on globaalne unikaalne identifikaator ja sellele samale identifikaatorile adresseeritakse sõnumeid.
Täpselt neid objekte käideldakse eraldi käitussüsteemis, kus dünaamiline koorma tasakaalustusraamistik[7] käsitleb neid elemente järjendis nagu objekte, mida saab ümber asustada (*migrate*) protsessorite vahel. Seega käitusmootor jälgib arvutuste koormust üle süsteemi ja võtab aega, kui palju iga järjendis oleva elemendi meetodi käivitamine aega võttis. Nendest järeldustest sõltuvalt jaotatakse ülejäänud elemendid saadaolevate protsessorite vahel. [4]   Nagu ka joonisel näha (Joonis 6), ei pea iga protsessor täitma täpselt sama palju ülesandeid.

[7] *Dynamic load balancing framework* või siis *LB Framework* on raamistik, mille ainus eesmärk on jagada erinevaid töid erinevate protsessorite vahel, et tasakaalustada koormust

Esialgu kõik *chare array* elemendid jagatakse laiali „*round-robin*“ algoritmi järgi. Kuna kõik *chare array* elemendid on iseseisvad *chare* objektid, saab neid individuaalselt liigutada tööde vahel. Charm++ käitussüsteem jagab *chare* järjendi elemendid protsesside vahel nii, et kõigile protsessidele tuleks võrdselt koormust. Programmeerija ei pea teadma, kus iga individuaalne *chare* järjendi element on. Isegi siis kui elemente saadetakse mitmeid kordi protsesside vahel, programmeerija saab ikkagi viidata elemendile samamoodi:

```
mingi_järjendi_objekt[mis_kohal]
```

„mis_kohal“ element on programmeerija jaoks alati sama. Charm++ käitusmootor hoolitseb selle eest, et sõnum saadetakse õigele protsessorile. [16]

2.6 Chare grupid

Igale protsessorile jagatakse teatud koormus *chare* objekte ja seega ühele protsessorile satub teatud kogus neid objekte. Siit tekibki veel üks tüüp objekte – *Chare* grupid (*chare group*). Iga *chare* grupp on kogum *chare* objektidest, mis on iga protsessori grupi liige. Kõik liikmed *chare* grupist omavad globaalselt unikaalset nime ja *CkGroupID* tüüpi sündmusetöötlejat. Tervet *chare* gruppi saab adresseerida, kasutades seda globaalset sündmusetöötlejat. [1]

2.7 Sõnumid

Eelnevalt oli teemaks tähtsamatest objektidest Charm++ raamistikus – *chare* objektid. Lisaks oli mainitud, et nende põhiliseks ülesandeks on suhelda omavahel saates sõnumeid ja neid sõnumeid saab saata sõltumatult. Seega sõnumite töötlemine on teine grupp paralleelselt käivitavatest objektidest. Kogu suhtlus käib sõnumite kaudu ja isegi parameetreid tõlgitakse ümber sõnumiteks. Seega sõnumeid käsitletakse teisiti ülejäänud objektidest Charm++ käitussüsteemis. Sellepärast peabki neid eelnevalt defineerima liidese failis (.ci laiendiga, sellest pikemalt peatükis 3.3). Tänu sellele liidese failile, teab käitussüsteem kõiki sõnumeid, mida luuakse terve programmi käigus, seega sõnumite vahel valimisega tegelebki käitussüsteem ise[8]. Põhiliselt jaotatakse sõnumeid kahte tüüpi:

- objektid C++ klassidest[9], mis on alamklassid spetsiaalsetest Charm++ liidese translaatori poolt genereeritud klassidest
- sõnumiteks kapseldatud parameetrid[10], kui soovitakse sõnumites edastada ka parameetreid ja ei kasutata parameetrite ümberkodeerimist

---

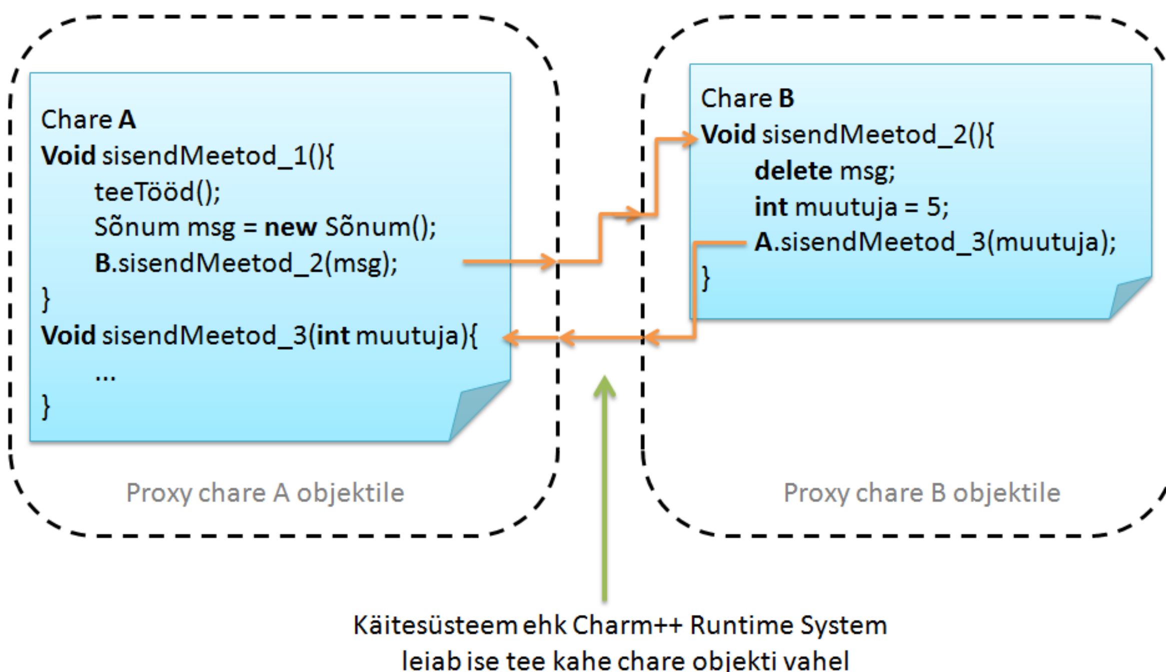
[8] Igas Charm++ programmi sisemuses on planeerija, mis valib kõigist olemasolevatest sõnumitest mingeid sõnumeid mida käivitada.

[9] Isendid C++ klassidest

Siit edasi minnes eksisteerib kaks võimalust parameetrite edastamiseks. Nendeks on parameetrite kapseldamine (enamasti kasutatakse seda, sest sõnumite saatmine on kiirem kui kodeeritud parameetrite saatmine) ja parameetrite ümberkodeerimine (*marshalled parameters*). Suurim erinevus kodeerimise ja kapseldamise vahel on see, et sisendmeetodid jätavad alles kapseldatud sõnumeid, mida neile saadetakse (erinevalt kodeeritud parameetritest). Seega peab sisendmeetod alati kustutama sõnumi (et vältida mälu lekkimist), sest seda ei tehta vaikimisi automaatselt. Kui need eelnevad olid mittepakitud sõnumid, siis eksisteerib veel ka pakitud sõnumid. Pakitud sõnumeid tuleks kasutada siis, kui soovitakse saata sõnumeid, mis sisaldavad viiteid andmetele, mitte andmeid ise. [4]

2.8 *Proxy* liides

Hajussüsteemides kirjeldatakse, et *proxy* imiteerib iseseisvat üksust (*entity*). Kuid meie jaoks on *proxy* kui lokaalne C++ klass, mis esindab eemalt käivitavat C++ klassi. Alati kui mõni *chare* objekt käivitab teise *chare* objekti sisendmeetodi, siis saatjal peab olema alati viide saajale – see viide ongi *proxy*. Charm++ käitussüsteem määrab ära kõigile *chare* objektidele kohad globaalses objektide ruumis ja *proxy* väärtusi ei pea programmeerija samuti määrama. *Proxy* klassi meetod vastab eemaltjuhitava klassi meetodile ja käitub nagu saatja. Kui kutsutakse meetod *proxy* peale, siis *proxy* saadab selle sama kutse üle võrgu päris objektile, mida lokaalne *proxy* esindab ja seega Charm++ raamistikus kogu suhtlus käib läbi *proxy* liidese. Eelneva kirjelduse demonstreerimiseks on järgnev joonis (Joonis 7). [4]



Joonis 7. Suhtlemine kahe *chare* objekti vahel toimub läbi *proxy* liidese.

---

[10]Kapseldatud parameetrid (*Marshalled parameter*) – enne kui mingi meetodi parameetreid hakatakse saatma, tehakse need (kodeeritakse) ümber sõnumiteks, sest andmevahetus toimub ainult sõnumite näol, ja neid ümbertõlgituid (kodeeritud) parameetreid nimetataksegi „marshalled parameters".

Proxy klass genereeritakse liidese translaatori poolt, baseerudes sisendmeetodite kirjeldusele. Igale *chare* tüübile eksisteerib *proxy* klass ja kõigil *proxy* liidestel on samad omadused nagu sündmusetöötlejatel mis hoolitsevad ülesannete täitmise eest. Töötleja (näiteks *CkChareID*, *CkArrayID*) ja *proxy* (näiteks *Cproxy_meetod*) koosnevad mõlemad baitidest ja neid saab saata lihtsalt sõnumites, lahti- ja kokkupakitud sõnumites ning ümberkodeeritud parameetritega sõnumites. Kokkuvõtvalt võib öelda, et kõikidel üksustel, mis on eemalt juhitavad, sisaldavad *proxy* klasse. On võimalik, et paljud erinevad *proxy* klassis olevad meetodid vastavad samale objektile ja samas on võimalik, et paljud erinevad *proxy* liidesed vastavad samadele objektidele. [4]

2.9 Sisendmeetod

Sisendmeetod on *chare* klassi funktsioonide liige. Erinevus C++ klasside funktsioonide liikmest on see, et Charm++ sisendmeetod käitub nagu sõnumite vastuvõtupunkt. Kui üks *chare* objekt kutsub teise *chare* objekti, siis saadetav andmekogu pakitakse ühte sõnumisse ja saadetakse (läbi *proxy*) otse teisele *chare* objektile. Kui saaja on sõnumi saanud, siis ta käivitab sõnumis määratud sisendmeetodi ja edastab sinna sõnumis olevad andmed. Põhiliselt on kaks väga suur erinevust võrreldes C++ meetoditega:

- Esiteks sisenevad meetodid ei saa sisaldada tagastusväärtus (funktsioon ei saa tagastada midagi – *return void*).
- Teiseks on mitteblokeeruv saatmise arhitektuur. Kui kood jõuab käivitamisega saatmiseni, siis käivitatakse sisendmeetod ja minnakse käivitusega kohe edasi, see tähendab seda, et sisendmeetod on täitmisel (kontroll puudub kas ja kuna täidetakse).

Eelnevalt oli kirjeldatud, et kogu suhtlus toimub sõnumite vahetamisega, kuid nüüd võib natukene detailsemalt öelda: Chamr++ raamistikus toimub kogu suhtlus sõnumite abil eemaltjuhitavate meetodite kutsumistega (Joonis 7). Need eemaltjuhitavad sisendmeetodid võivad võtta sõnumitesse pakitud parameetreid või sõnumi objekte. Sõnumid on madalama tasemega ja palju paindlikumad kui sõnumiteks kodeeritud parameetrid. Sisendmeetod on alati üks osa *chare* objektist, ehk siis ei eksisteeri globaalset sisendmeetodit Charm++ raamistikus. Sisendmeetodit, mis on sama nimega nagu enda klass, kutsutakse konstruktoriks. [4]

3 Programmeerimine Charm++ raamistikus

3.1 Charm++ struktuur

Charm++ struktuurilt sarnaneb C++ programmeerimiskeelele, sest enamus Charm++ programm koosnebki C++ koodist. Kasutaja lisab vaid standardsele C++ koodile (muidugi on ka Charm++ spetsiifilist koodi) spetsiaalseid Charm++ teeke, mis sisaldavadki kogu Charm++ raamistiku baas klasse, funktsioone jne. Kusjuures põhiline Charm++ programmi süntaks on klassi definitsioonid. Edasise töö teeb translaator, mis genereerib lisa koodi, et käidelda Charm++ konstruktoreid. Kõik liidesed Charm++ objektidele (sõnumid, *chare* objektid, ainult lugemiseks väärtused jne.) peavad olema deklareeritud Charm++ liidese failis. Tüüpiliselt grupeeritakse neid mooduliteks ja Charm++ programm võib koosneda mitmest moodulist. Üks moodulitest peab alati olema põhimoodul (*mainmodule*) ja see peab olema kättesaadav kõikidele ülejäänud moodulitele. [4]

3.2 Charm++ käivitamise mudel

Charm Kernel (käitussüsteem) alustab käivitamist, luues põhilise eksemplari 0 protsessorile. Peale seda loob põhikonstruktor *chare* grupi ja salvestab enda töötleja ning väljub selle koodi täitmisest. Kohe peale seda luuakse igale protsessorile grupp ja kutsutakse iga protsessori konstruktorid. Edasised funktsioonid algavad spetsiaalses *chare* objektis nimega *main chare* (umbes sama on ka C++ programmeerimiskeelele omane – *main* meetod ehk põhimeetod), mis hakkab kõiki teisi meetodeid või funktsioone välja kutsuma. Näiteks *mainchare* nimega *Mingi_nimi* käivitamine algab konstruktoris `Mingi_nimi()` või `Mingi_nimi(CkArgMsg *)`, mis on omavahel ekvivalentsed. Nüüd toimub *chare* objektide omavaheline suhtlus läbi *proxy*, kus siis üritatakse käivitada teiste objektide sisendmeetodeid läbi *proxy* liidese ja kogu suhtlus käib ikkagi sõnumitega. Kui konstruktorid lõpetavad oma töö, siis lõpus kutsutakse `CkExit()` meetod (nagu C++ keeles sama ka charm++ raamistikus, et süsteemi lõpetamise kutsega midagi ei tagastata). *Charm Kernel* hoolitseb selle eest, et pärast selle meetodi kutsumist ei töödelda ühtegi sõnumit ja samuti ühtegi sisendmeetodit ei kutsuta. Seda meetodit ei pea kutsuma kõikidel protsessoritel, vaid kui ühel protsessoril see välja kutsutakse, siis lõpetakse töö kõigil protsessoritel. Siis kui enam ei ole ühtegi sõnumit saata, toimub üleliigsete andmete kustutamine ja seejärel Charm++ programm lõpetab oma töö. [4]

3.3 Kompileerimine

Charm++ baseerub C++ keelel, seega kogu kompileerimise protsess on sama vaid väikeste erinevustega.

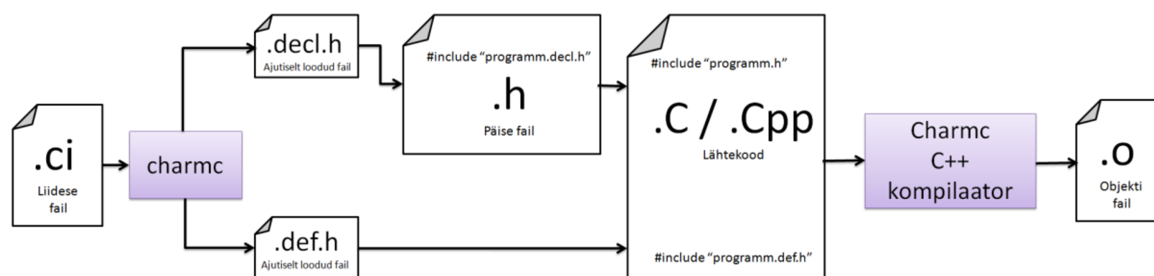
C++ keeles:
Iga klass vajab kahte faili ja nendeks on klassi definitsioon (.h failid) ja funktsiooni keha (.c või .cpp failid). Klassi definitsioon kirjeldab, millised funktsioone ja muutujaid keha sisaldab ja lähtekoodi failis on kood, mida tõlgitakse masinkoodi.

Charm++ raamistikus:
Iga klass vajab samuti kahte faili, kuid lisaks veel ka liidese kirjeldust. Liidese kirjelduses kirjeldatakse, milliseid funktsiooni liikmeid saadakse välja kutsuda teiste *chare* objektide poolt globaalses objektide ruumis.

Seega, kui kogu protsess on peaaegu sama ja tuleb vaid arvestada ühte liidese faili, on Charm++ programmi kompileerimiseks välja töödeldud lihtne tööriist, milleks on *Charmc*. *Charmc* on tööriist, mis ümbritseb põhilist kompileerijat. Charm++ koodi kompileerimine käib läbi Charmc programmi, mille käivitamisel võetakse .ci fail ja genereeritakse kaks faili (.decl.h ja .def.h), mis sisaldavad automaatselt genereeritud koodi. Need failid abistavad C++ keeles kirjutatud Charm++ programmi koodi ühildada käitussüsteemiga:

- decl.h fail pakub deklaratsioone paralleelselt käivitavate objektide *proxy* klassidele
- def.h failis on registreeritud funktsioonid, mis kutsuvad kõiki funktsioone vastavalt ainult lugemiseks väärtustele ja sisendmeetoditele [18]

Peale selle järgneb sama süsteem nagu C++ koodi kompileerimiselgi (Joonis 8).



Joonis 8. Charm++ programmi kompileerimine.

3.4 Charm++ programmide käivitamine

Kui *charmc* teeb viimase käivitatava programmi, siis luuakse veel üks programm samasse kausta, kuhu kõik ülejäänudki failid loodi – *charmrun*. Tekib kaks võimalust charm++ programmide käivitamiseks:

Esimene on lihtsalt programmi käivitamine ühe protsessori peal käsuga:

```
./programm
```

Teine on läbi *charmrun* programmi, kui tahetakse käivitada programmi mitme protsessori peal. Et määrata ära, mitmel protsessoril peaks programm jooksma, saab anda *charmrun* programmile *+p* argumendi. Näiteks käivitada programm neljal protsessoril:

`charmrun +p4 ./programm` [17]

3.5 Charm++ programmi iseärasused

Kuna lisades on välja toodud Charm++ programmi koodid, siis eelnevalt oleks kasulik Charm++ raamistiku põhilised funktsioonid selgeks teha.

`Main(CkArgMsg* msg)` - kui rakendus käivitatakse, siis kutsutakse kõige esimesena just see konstruktor Charm++ käitussüsteemi poolt.

`Main(CkMigrateMessage* msg)` - konstruktor, mida kasutatakse *chare* objektide siirdamiseks.

`CkPrintf("Mingi tekst")` - põhimõtteliselt sama funktsioon ekraanile kuvamiseks nagu `printf()` funktsioon C/C++ keeles, kuid see funktsioon on disainitud kuvama teksti, mida peaksid protsessorid tagastama ja sõltumata kus nad parajasti viibivad.

`CkExit()` - seda funktsiooni peab alati kutsuma kui protsessorid oma töö lõpetavad, sest see funktsioon lõpetab programmi. Kui kutsuda välja lihtsalt `Exit()` funktsioon ühele protsessorile, siis see protsessor tõesti lõpetab oma töö, kuid kõik ülejäänud protsessorid jäävad ootama järgmisi käske.

`#include "main.decl.h"` – lähtekoodi alguses vajalik, et saavutada ühilduvus Charm++ käitussüsteemiga.

`mainmodule main` – deklareerib liidese mooduli, ehk teeb kaks faili: main.decl.h ja main.def.h.

mainchare Main – deklareerib põhilise *chare* objekti rakendusele. Rakenduse käivitamine algabki just sellest põhilisest objekti konstruktorist.

entry Main(CkArgMsg* msg) - deklareerib põhilise *chare* klassi sisendmeetodiks, see tähendab, et ülejäänud *chare* objektid saavad sellesse objekti pöörduda nagu põhilisse objekti.

ckNew() - loob järjendi *chare* objektidest, et neid sõltumata käivitada järjendis vastavalt soovile.

CProxy_chareObjektiTüüp – loob etteantud tüüpi uue chare objekti. [18]

Kokkuvõte

Tehnika areneb pidevalt, kuid siiski ei suuda see sammu pidada tarkvara kiiruste nõuetele. Vähemalt eksisteerib üks viis, kuidas väga keerulisi algoritme kiiremaks teha. Selleks on paralleelselt töötavad arvutused. Programmi paralleelsuse saavutamiseks leidub palju keeli ja raamistikke, kuid kõigi nende mõte jääb samaks – kas paralleelsus saavutatakse andmete või ülesannete jagamisega. MPI standard on endiselt populaarseim liides paralleel-programmide kirjutamiseks, sest igasugune paralleelsus kiirendab arvutusi. Tuleviku poole liikudes märgatakse, et tegelikult saavutatakse veel kiirem tulemus, kui programm suudab keskkonda, kus see programm töötab, efektiivsemalt ära kasutada. Üks selliseid paralleel-programmeerimise mudeleid on Charm++. Charm++ käitusmootor tegeleb koormuste jagamisega erinevate protsessorite peal ja samas pakub andmete sõltumatut migreerumist üle suure ala. Charm++ programm koosneb objektidest ja need suhtlevad omavahel, saates sõnumeid, millega kutsutakse saava objekti sisendmeetod. Kogu vahepealset protsessi ei pea programmeerija isegi teadma. Piisab vaid teadmisest, kuidas luua uus objektide järjend ja nad mingisse gruppi määrata. Paralleelsus saavutataksegi siis, kui need objektid teevad oma tööd samaaegselt erinevatel protsessoritel.

Charm++ sisaldab palju erilisi iseärasusi ja alles pärast nende teadmist avastatakse, et tegelikult on selles raamistikus mugavam programmeerida kui mingis muus sõnumiedastus mudelis. Üks kindel eelis seisneb ka selles, et programmeerija näeb, kus asub koodijupp, mida täidetakse paralleelselt, sest see on defineeritud teises klassis. MPI puhul pidi hoolikalt jälgima, mis andmed kellel parasjagu on, sest koodi täitmine käib järjest ja igat funktsiooni täitsid ettenähtud protsessorid.

Kokkuvõtteks võib öelda, et mõistlik on käia tarkvara arenguga kaasas ja pidevalt kasutada uuemat tarkvara arendamiseks. Kogu aeg tehakse juurde riistvara ja seega eksisteerib rohkesti erinevaid arhitektuure. Mõistlikum on kasutada arendamisvahendeid, mis suudaksid palju tööd programmeerija eest ära teha ja Charm++ pakubki mõningaid keerulisi iseärasusi, millest programmeerija ei pea isegi teadma.

Summary

Review of object oriented parallel programming framework Charm++

Bachelor Thesis

Tõnu Jaarma

The aim of this bachelor thesis is to give and overview of parallel programming and what does running programs in parallel mean. Today most of the parallel programs are written in MPI, but that does not mean that there does not exist another parallel programming models. This work gives short overview of another great parallel programming framework which is Charm++. Charm++ is an object oriented parallel programming framework written in C++. Basically this document is divided into theoretical and practical part. In theoretical part there are described common parallel programming problems and solutions in both models – MPI and Charm++. There is also practical part of this document and it consists of programs written in C (for MPI) and in C++ (for Charm++). Also included the tutorials how to set up Charm++ and MPI environments on Linux (tested with Ubuntu).

In the first chapter there is described that parallel programming is divided into data parallelism and task parallelism. All the giving examples and programs use mainly data parallelism. As mentioned in first paragraph the best way to solve matrix multiplication is to divide the first matrix to set of rows and the second matrix to columns. Now every processor will have only one row and column to multiply and in the end all the calculated results are collected together.

The second chapter is about Charm++ and its structure. It is important to know all the concepts before programming and this bachelor thesis describes all of those main categories of objects. Charm++ consists of five major disjoint categories of objects:

1. Sequential objects
2. Chares
3. Chare groups
4. Chare arrays
5. messages

Third chapter describes how to write Charm++ code and how to compile it that the Charm++ Runtime System does most of the job. In this chapter there is also mentioned what are the differences between C++ code and Charm++ code. All the needed functions are described to program a matrix multiplication. Also those programs are in the end of this work.

Kasutatud kirjandus

[1] Charm++ Frequently Asked Questions [Online][11]

http://charm.cs.illinois.edu/manuals/pdf/faq.pdf

[2] NAMD research [Online]

http://www.ks.uiuc.edu/Research/namd/

[3] Center for simulation of advanced rockets [Online]

http://www.csar.illinois.edu/rocstar/index.html

[4] The Charm++ Programming Language Manual [Online]

http://charm.cs.illinois.edu/manuals/pdf/charm++.pdf

[5] Task parallelism [Online]

http://en.wikipedia.org/wiki/Task_parallelism

[6] Data parallelism [Online]

http://en.wikipedia.org/wiki/Data_parallelism

[7] MPI tutorial [Online]

http://www.mpitutorial.com/mpi-introduction/

[8] MPI tutorial [Online]

http://www.mpitutorial.com/mpi-send-and-receive/

[9] MPI documentation for MPI_Send [Online]

http://www.open-mpi.org/doc/v1.5/man3/MPI_Send.3.php

[10] MPI documentation for MPI_Recv [Online]

http://www.open-mpi.org/doc/v1.5/man3/MPI_Recv.3.php

[11] MPI documentation for MPI_Bcast [Online]

http://www.open-mpi.org/doc/v1.5/man3/MPI_Bcast.3.php

[12] MPI documentation for MPI_Scatter [Online]

http://www.open-mpi.org/doc/v1.5/man3/MPI_Scatter.3.php

[13] MPI documentation for MPI_Gather [Online]

http://www.open-mpi.org/doc/v1.5/man3/MPI_Gather.3.php

[14] MPI documentation for MPI_Isend [Online]

http://www.open-mpi.org/doc/v1.5/man3/MPI_Isend.3.php

[15] MPI documentation for MPI_Wait [Online]

[11] Kõiki neid lehti kontrolliti viimati 11.05.2012.

http://www.open-mpi.org/doc/v1.5/man3/MPI_Wait.3.php

[16] Introduction to Charm++ concepts [Online]

http://charm.cs.illinois.edu/tutorial/CharmConcepts.htm

[17] Introduction to the Charm++ Runtime System [Online]

http://charm.cs.illinois.edu/tutorial/CharmRuntimeSystem.htm

[18] Introduction for programming in Charm++ [Online]

http://charm.cs.illinois.edu/tutorial/BasicHelloWorld.htm

[19] Actor model [Online]

http://en.wikipedia.org/wiki/Actor_model

[20] ADA programming language [Online]

http://en.wikipedia.org/wiki/Ada_%28programming_language%29

Lisad

Lisadeks on pandud eraldi CD plaadile MPI maatriksi programm ja Charm++ maatriksi programm. Mõlemad programmid arvutavad maatriksite korrutamist ja nende lähtekoodid paiknevad kahes kaustas järgnevalt:

- Charm++ maatriksite korrutamine
  - main.C – selles failis on kirjutatud kood, mida peab vaid põhiprotsess tegema, samuti selles failis alustatakse programmi käivitamist.
  - main.ci – sisaldab sisendmeetodite ja ainult lugemiseks väärtuste definitsioone.
  - main.h – päise fail, kus on defineeritud avalikud ja privaatsed muutujad ning meetodid.
  - Makefile – sisaldab kõike vajalikke koode, et kasutajal oleks mugavam koodi kompileerida. Olgu juurde veel öeldud, et esimene rida tuleb välja vahetada vastavalt oma arvutis installeeritud Charm++ asukohale.
  - multiply.C – selles failis olevad meetodid käivitatakse paralleelselt.
  - multiply.ci – sisaldab samuti sisendmeetodeid ja ainult lugemiseks väärtuseid.
  - multiply.h – päise fail, kus on defineeritud avalikud muutujad ja meetodid.

- MPI maatriksite korrutamine
  - matrix_multiplication.c – kogu maatriksite korrutamine on paigaldatud sellesse faili.

Lisaks on kaasa ka pandud õpetused kuidas endale need keskkonnad peale panna .pdf faili kujul:

- Charm++ ja MPI keskkondade ülesseadmise õpetus

Õpetus on kirjutatud linux keskkonnale ja testitud Ubuntu peal (täpsemalt „ubuntu-10.04.2-desktop-i386“).