TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKA TEADUSKOND
Arvutiteaduse Instituut
Hajussüsteemide õppetool
Informaatika eriala

Ain Uljas

MPI2 sisend-väljund vahendid

Bakalaureusetöö, 4 AP

Juhendaja: Eero Vainikko

Autor: .... ”.......” mai 2006
Juhendaja: .... ”.......” mai 2006

Tartu 2006

Sisukord

1 Sissejuhatus....3
2 Sissejuhatus sõnumiedastusmeetodisse....4
2.1 Sõnumiedastus-programmide struktuur....4
2.2 Põhikäsud....5
2.3 Blokeeruvad sõnumiedastusoperatsioonid....5
2.3.1 Blokeeruv mitte-puhverdatud send/receive....5
2.3.2 Blokeeruv puhverdatud send/receive....7
2.4 Mitte-blokeeruvad sõnumiedastusoperatsioonid....8
3 MPI: Message Passing Interface....10
3.1 MPI teegi töö alustamine ja lõpetamine....10
3.2 Kommunikaatorid....11
3.3 Sõnumite saatmine ja vastuvõtt....11
3.3.1 Tupikute vältimine....14
3.4 Kollektiivne kommunikatsioon....15
4 Sisend/väljund....18
4.1 Mõisted....18
4.2 Lihtsamad failioperatsioonid....19
4.3 Vihjed....21
4.4 Vaated....22
4.5 Andmete ligipääs....24
4.5.1 Positsioneerimine....24
4.5.2 Sünkronisatsioon....25
4.5.3 Koordineeritus....25
4.5.4 Kokkulepped....26
4.5.5 Lõhestatud kollektiivsed rutiinid....26
4.6 Failiesitusviisid....27
4.7 Terviklikkus ja semantika....28
5 Kokkuvõte....29
6 Abstract....30
7 Kasutatud kirjandus....31

1 Sissejuhatus

Sõnumiedastus-programmeerimise paradigma on üks vanimaid ning ka levinumaid lähenemisi programmeerida paralleelselt töötavaid masinaid. Mõningast „süüd“ omab antud faktis asjaolu, et riistvarale esitatakse vaid minimaalsed nõudmised. Minu eesmärgiks on uurida, kas ja kuidas on võimalik uusimaid MPI (*Message Passing Interface*) laiendusi kasutades efektiivne töö sisend-väljundiga.

Töö esimeses osas esitatakse üldine taust sõnumiedastus-programmeerimise paradigmast. Teises osas tutvustatakse põgusalt MPI võtmevõimalusi paralleeltööks. Töö kolmandas osas vaadeldakse MPI poolt pakutud sisend-väljund mudelit ja uuritakse põhilisi funktsioone.

Üldise taustainfo saamiseks kasutati põhilise allikana *Introduction to parallel computing* [1]. Mõningaid nopped on võetud ka *Parallel programming with MPI* [2]. Sisend-väljundi osa kirjeldamiseks oli suurt abi nii standardi enda tekstist [3] kui ka selle, võib lugeda, kasutusõpetusest [4].

2 Sissejuhatus sõnumiedastusmeetodisse

Paralleelprogrammeerimise jaoks on loodud arvukalt programmeerimiskeeli ja teeke. Nad erinevad üksteisest nii selle poolest, kuidas käsitlevad programmeerija kätte antavat aadressruumi, kuidas tullakse toime paralleelsete tegevuste sünkroniseerimisega ja kui hea või halb on keele väljendusvabadus. On kaks võtmeelementi, mis iseloomustavad sõnumiedastus-programmeerimise paradigmat. Esimene neist eeldab osadeks lahutatud aadressruumi, teine aga toetab ainult ilmutatud paralleliseerimist.

Loogiline vaade sõnumiedastus-paradigmat toetavast masinast koosneb p protsessist, igal oma ainuõiguslik aadressruum. Aadressruumi tükeldatusest saab teha kaks järeldust. Esiteks, iga andmeblokk peab kuuluma ühte antud mälupartitsioonidest, seega peavad andmed olema ilmutatult ära jaotaud ja paigutatud. See teeb programmeerimise keerulisemaks, kuid julgustab kasutama rohkem lokaalset mäluruumi saavutamaks kõrgemat jõudlust, kuna protsessor saab sellele ligi kiiremini kui mittelokaalsele mäluruumile. Teine järeldus on see, et kõik omavahelised suhtlemised protsesside vahel nõuavad vastastikust valmisolekut andmevahetuseks, nii protsess, mis saadab infot kui ka protsess, mis seda vastu võtab. See nõue lisab mitmesugustel põhjustel rohkesti keerukust. Protsess, mis omab vajalikke andmeid, peab osalema suhtluses isegi siis, kui tal ei ole mingit pistmist sündmustega, mis toimuvad neid andmeid soovivas protsessis. Teatud juhtudel viib see väga veidrate programmideni. Dünaamilise ja/või halva struktuuriga suhtluse korral võib programmi kood olla väga keeruline.

Sõnumiedastus-programmeerimise paradigma nõuab, et paralleelsus on ilmutatult programmeerija poolt loodud. See tähendab, et vastava jadaprogrammi analüüs ning dekompositsioon, nii et seda saab täita paralleelselt, on puhtalt programmeerija vastutusel. Tulemuseks on see, et programmeerimine, kasutades sõnumiedastus-paradigmat, kipub olema üsna raske ning intellekti nõudev. Teisest küljest, korralikult kirjutatud sõnumiedastus-programmid võivad tihti saavutada väga kõrge kasumlikkuse ning neid saab korraga täita väga paljudel protsessidel.

2.1 Sõnumiedastus-programmide struktuur

Sõnumiedastus-programmid on tihti kirjutatud kasutades asünkroonset või lõdvalt sünkroonset lähenemist. Asünkroonse paradigma korral täidetakse kõik sünkroonsed käsud asünkroon-

selt. See teeb võimalikuks mistahes paralleelse algoritmi realiseerimise. Siiski, niisuguste programmide korrektsuse tõestamine on raskem ja nad võivad sisaldada mitte-determineeritud käitumist. Lõdvalt sünkroonsed programmid on heaks kompromissiks. Niisugustes programmides protsessid sünkroniseeruvad andmevahetuseks, kuid nende vahepeal käib töö täiesti asünkroonselt. Paljusid tuntud paralleelalgoritme saab elegantselt realiseerida kasutades selleks lõdvalt sünkroonset paradigmat.

2.2 Põhikäsud

Kuna interaktsioon saavutatakse saates ja võttes vastu sõnumeid, siis peamised operatsioonid sõnumiedastus-programmeerimise paradigmas on `send`[1] ja *`receive`*[2]. Nende lihtsaimas vormis näevad deklaratsioonid välja järgmised:

```
send(void *sendbuf, int nelems, int dest);
receive(void *recvbuf, int nelems, int source);
```

kus *sendbuf* `ja` *recvbuf* on viidad puhvritele, mille sisu tuleb saata või vastu võtta, *nelems* andmeühikute arv, mis saadetakse või vastu võetakse ning *dest* ja *source* vastavalt protsessid, millele info saadetakse või või millelt info tuleb.

Kuigi nende kahe funktsiooni semantika võib tunduda üsna triviaalne, on asjalood mõnevõrra keerulisemad, kuna poliitika nende funktsioonide implementeerimisel varieerub. Enamikel sõnumiedastus-platvormidel on andmete saatmiseks ja vastuvõtmiseks spetsiaalne riistvaraline tugi. Nad võivad toetada otsepöördusmälu (DMA-*direct memory access*) ja asünkroonset ülekannet kasutades võrguliidese riistvara. See võimaldab andmeid liigutada ilma protsessori osaluseta.

2.3 Blokeeruvad sõnumiedastusoperatsioonid

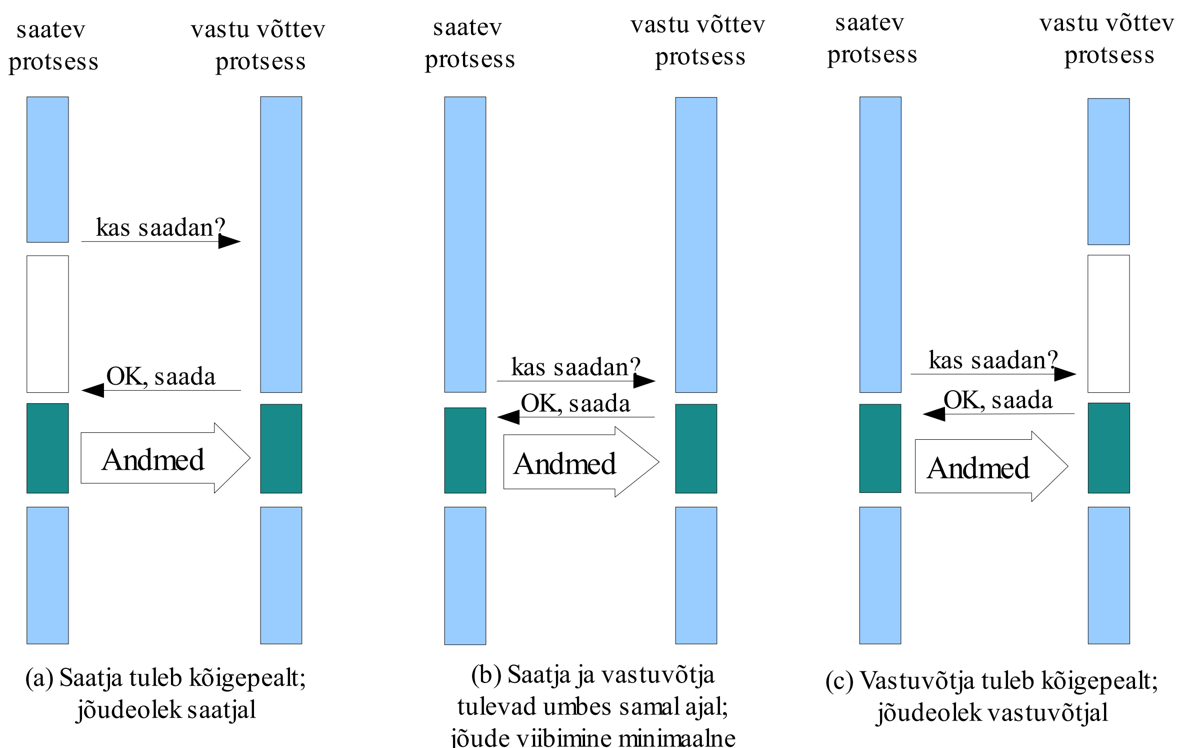
2.3.1 Blokeeruv mitte-puhverdatud `send/receive`

Esimesel juhul ei tagastu `send` funktsioon enne kui vastav `receive` käsk on sihtprotsessis välja kutsutud. Kui see juhtub, siis sõnum saadetakse ja `send` funktsioon tagastub, olles ope-

1 e. k. saada
2 e. k. võta vastu

ratsiooni edukalt lõpetanud. Tüüpiliselt on mängus ka nn käepigistus saatva ja saava protsessi vahel. Saatja protsess saadab „avalduse“ et soovib teise protsessiga suhelda. Kui teine protsess on sooviavalduse kätte saanud, ta vastab. Saatja protsess käivitab andmete ülekandmise operatsiooni. Kuna ei saatja ega vastuvõtja ei kasuta puhverdamist nimetatakse seda ka **mittepuhverdatud blokeerivaks operatsiooniks**.

Illustratsioonil 1 on näidatud kolm võimalikku juhtu, kus esimesel juhul infot tahetakse saata enne kui ollakse seda võimeline vastu võtma, saatmise ja saamise signaal postitakse üsna



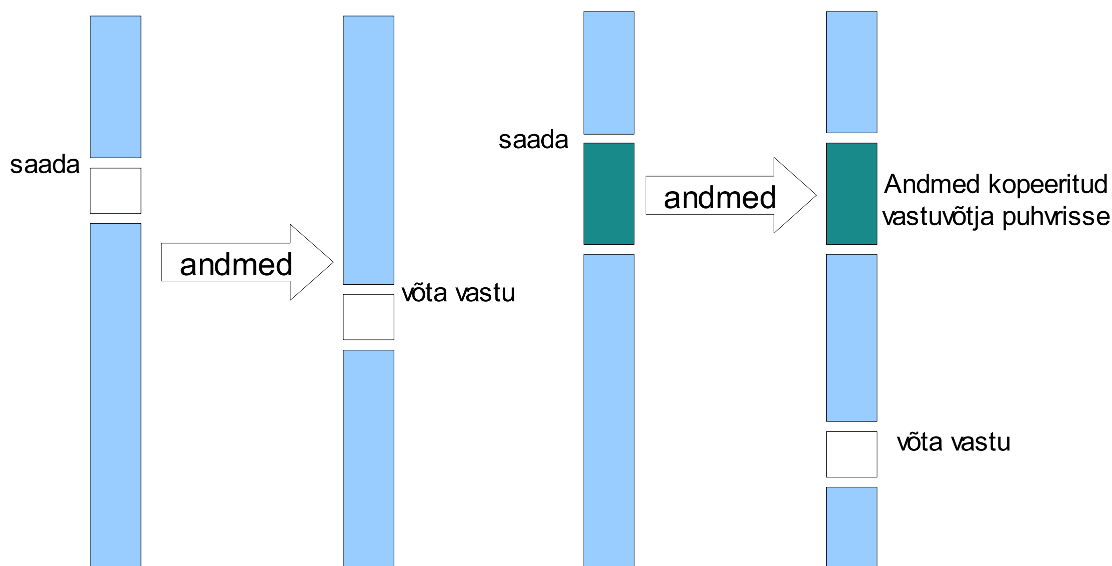
Illustratsioon 1: Käepigistus mitteblokeeruvas mittepuhverdatud saada/võta vastu operatsioonis.

üheaegselt ning kus andmeid vajatakse enne kui vastavasisuline saatmisoperatsioon on üldse alanud. Esimesel ja kolmandal juhul on tegemist emb-kumma protsessi jõudeolekus viibimisega. On selge, et blokeeruv mitte-puhverdatud protokoll on sobiv vaid siis, kui saatmist ja vastuvõtmist teostatakse enam-vähem samal ajal. Asünkroonses keskkonnas võib seda olla võimatu ennustada. Mõne protsessi jõudeolekus viibimise võimalikkus on selle meetodi suurimaks puuduseks. Protokolli kasutamise muudab veelgi keerukamaks asjaolu, et blokeeruvas kommunikatsioonis võivad tekkida kergesti tupikud. Seda võib saada lahendada operatsioonide järjekorra muutmisega, kuid mitte alati. Samuti teeb sedalaadi programmeerimine koodi raskemini jälgitavaks ja vigadele avatumaks.

2.3.2 Blokeeruv puhverdatud `send/receive`

Lihtne lahendus vältimaks jõude seismist on kasutada nii saatja kui saaja poolses otsas puhvreid. Kui saatjal on vaja mingeid andmeid edastada, siis kopeeritakse need esialgu vastavasse puhvrisse ja tagastub, kui kopeerimine on lõppenud. Saatja protsess saab nüüd jätkata enda programmi täitmist, teades, et muudatused andmetega ei muuda programmi semantikat. Tegelik kommunikatsioon saab toimuda mitmetel viisidel olenevalt allolevast riistvarast. Kui riistvara toetab asünkroonset kommunikatsiooni (sõltumatu protsessorist), siis saab võrguülekanne käivituda peale seda, kui andmed on kopeeritud puhvrisse. Paneme tähele, et vastuvõtja poolel ei saa samuti sõnumi sisu sihtkohta paigutada, kuna see läheb vastuollu programmi semantikaga. Selle asemel, andmed kopeeritakse eelnevalt puhvrisse ka siin. Kui vastuvõtja protsess avastab `receive` operatsiooni, siis kontrollitakse, kas andmed on puhvris saadaval. Kui on, kopeeritakse need sihtkohta. Olukorda iseloomustab Illustratsioon 2a.

Mõnikord ei oma masinad vastavat kommunikatsiooni jaoks vajalikku riistvara. Sellisel juhul võib kasutada puhverdamist ainul ühel pool. Näiteks, saatja protsess katkestab[1] vastuvõtja ning toimub ülekande operatsioon, kus andmed toimetatakse vastuvõtja protsessi puhvrisse. Kui vastuvõtja protsessi programmis jõutakse `receive` operatsioonini, siis lihtsalt kopeeri-



Illustratsioon 2: Blokeeruv puhverdatud ülekandeprotokoll: (a) kommunikatsiooni toetava riistvara olemasolul koos puhvritega saatja ja vastuvõtja otsas; (b) spetsiaalse kommunikatsiooniriistvara puudumisel katkestab saatja vastuvõtja ning paigutab andmed vastuvõtja puhvrisse.

1 i. k. interrupt

takse puhvri sisu vajalikesse mäluüksustesse. Seda illustreerib Illustratsioon 2b. Ei ole raske modelleerida protokolli, kus puhverdamine toimub ainult saatja poolel ning vastuvõtja algatab ülekande katkestades saatja.

On kerge näha, et puhverdatud protokollid leevendavad jõudeajas seismist, kuid lisavad puhvri haldamise kulusid. Üldiselt on nii, et tugevalt sünkroonne mitte-puhverdatud saatmine toimib paremini kui puhverdatud saatmine. Siiski, mitmetahulistes programmides ei ole sünkroonsus garanteeritud, mistõttu eelistatakse puhverdatud kommunikatsiooni, v.a siis, kui puhvri suurus saab takistuseks.

Kuigi puhverdamine lahendab paljud tupikud, on nende tekkimine (tekitamine) siiski võimalik. See tuleneb sellest, et `receive` käsud on alati blokeerivad käsud (et kindlustada semantika kooskõlalisust).

2.4 Mitte-blokeeruvad sõnumiedastusoperatsioonid

Blokeerivates protokollides tuli programmi semantika korrektsuse kindlustamiseks maksta andamit ootavate protsesside või puhvrihaldusele kuluva aja näol. Tihti on aga programmeerijalt võimalik nõuda semantika korrektsust ning provideerida vastutasuks kiiret `send/receive` operatsiooni, mille halduskulud[1] on minimaalsed. See klass mitte-blokeerivaid protokolle tagastub vastavasisulisest operatsioonist enne kui see semantiliselt ohutu oleks. Seega ei tohi kasutaja muuta andmeid, mis võivad parajasti osaleda andmevahetuses. Üldiselt on olemas ka `check-status` operatsioon, mis märgib seda, kas saatmises osalenud andmeid tohib juba muuta või mitte.

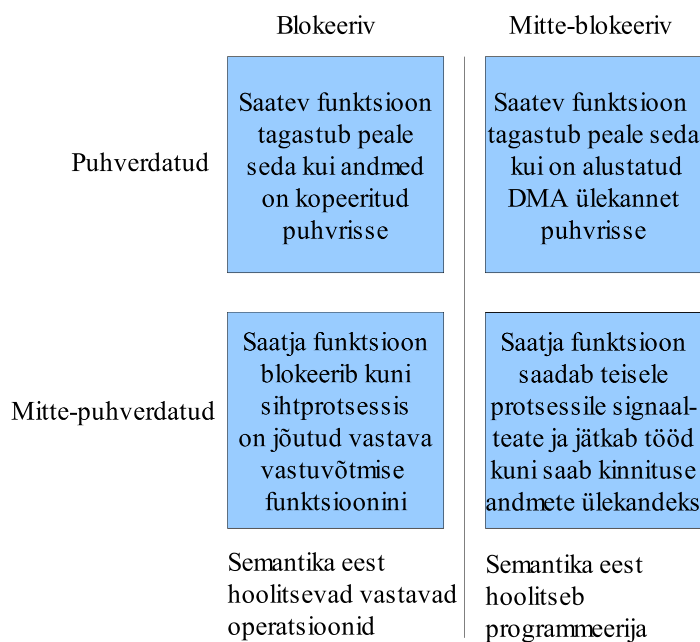
Mitteblokeerivad operatsioonid võivad ka ise olla puhverdatud või mitte-puhverdatud. Protsess, mis soovib saata andmeid, lihtsalt saadab sihtprotsessile vastavasisulise signaalteate ning tagastub; saab teisi töid edasi teha. Mõne aja pärast, kui saabub kinnitus, käivitatakse ülekandemehhanism. Kui see on lõpetatud, siis `check-status` funktsioon teatab programmeerijale, et neid andmeid on nüüd ohutu muuta.

Mitte-blokeerivate operatsioonide efektiivsust saab veelgi tõsta, kasutades spetsiaalset riistvara. Sellisel juhul on aga andmed terve `receive` operatsiooni vältel nn ohtlikus seisus.

1 i. k. overhead

Mitte-blokeerivaid operatsioone saab teha ka puhverdatult. Saatja käivitab DMA operatsiooni ja tagastub koheselt. Andmed saavad ohutuks hetkel mil DMA operatsioon on lõpetanud. Saatja poolel käivitab `receive` käsk andmete ülekande saatja puhvrist sihtkohta. Puhvrite kasutamine vähendab aega, mil andmete muutmine on lubamatu.

Message Passing Interface teek teostab nii blokeerivaid kui mitte-blokeerivaid operatsioone. Blokeerivate operatsioonidega on lihtsam ning ohutum programmeerida samas kui mitte-blokeerivad funktsioonid annavad kätte tööriistad koodi sügavaks optimeerimiseks. Viimaste kasutamisel peab olema väga ettevaatlik.



Illustratsioon 3: Mitmesuguseid andmeedastuse variante.

3 MPI: Message Passing Interface

Paljud varasemad kommertsiaalsed paralleelarvutid tuginesid sõnumiedastus-arhitektuurile, kuna see oli laias laastus odavam kui jagatud aadressruumiga arhitektuurid. Kuna sõnumiedastus on nendele arvutitele loomulik programmeerimise paradigma, siis viis see paljude erinevate teekide arendamisele. Tõtt-öelda sai sellest moderniseeritud vormiga assemblerkeel, milles iga riistvara tootja tegi oma teegi, kuid mis ei olnud kokkusobiv teiste riistvaratootjate arvutitega. Enamasti olid erinevused kõigest sünktaktilised, kuid piisavalt tihti leidus ka tõsiseid semantilisi erinevusi, mis nõudsid tõsist tööd, kui tahtsid ühele teegile kirjutatud programmi teisele portida.

Sõnumiedastusliides, MPI, loodi neid probleeme ületama. MPI defineerib standardse teegi, mida saab kasutada loomaks portatiivseid sõnumiedastusprogramme kasutades keeli C, C++, Fortran jm. MPI standard defineerib nii süntaksi kui ka tuumikhulga funktsioonide semantika. MPI lõid grupp teadlasi nii akadeemilisest kui industriaalsest ringkonnast ning on pälvinud toetuse pea kõigilt riistvaratootjatelt.

3.1 MPI teegi töö alustamine ja lõpetamine

`MPI_Init` käsk kutsutakse välja enne mistahes teist MPI rutiini. Selle eesmärgiks on initsialiseerida MPI keskkond. `MPI_Init` väljakutsumine rohkem kui korra tekitab programmis vea. `MPI_Finalize` kutsutakse välja arvutuste lõppedes. See teeb mitmesuguseid mäluvabastamisi ning lõpetab MPI keskkonna. `MPI_Init` ja `MPI_Finalize` tuleb välja kutsuda kõigil protsessidel täpselt ühe korra. Peale `MPI_Finalize` käsku ei tohi kasutada enam ühtki teist MPI teegi käsku, ka `MPI_Init` mitte. Nende funktsioonide prototüübid on järgmised:

```
int MPI_Init(int *argc, char ***argv)
void MPI::Init(int &argc, char** &argv)

int MPI_Finalize()
void MPI::Finalize()
```

Kõik meie poolt vaadeldavad funktsioonid nüüd ja edaspidi tagastavad keeles C täisarv tüüpi väärtuse, mida saab kasutada funktsiooni edu või ebaedu kontrolliks.

3.2 Kommunikaatorid

Võtmeelement läbi MPI on „kommunikatsiooni domeen“. Kommunikatsiooni domeen on hulk protsesse, millel on lubatud üksteisega suhelda. Info kommunikatsiooni domeenist on salvestatud muutujasse tüüpi `MPI_Comm` (või `MPI::Comm` tüüpi objekti), mida kutsutakse kommunikaatoriks. Kommunikaatorid on argumendiks paljudele MPI rutiinidele ja nad üheselt identifitseerivad protsessid, mis osalevad ülekande operatsioonil. Iga protsess võib kuuluda rohkemasse kui ühte domeeni. Eristatakse sise- ja väliskommunikaatoreid[1]. Interkommunikaatoreid kasutatakse infovahetuseks erinevatesse kommunikaatoritesse kuuluvate protsesside, intrakommunikaatoreid samadesse kommunikaatoritesse kuuluvate protsesside vahel. Antud töös vaatleme ainult sisekommunikaatoritega seonduvat.

Üldiselt, kõik protsessid võivad tahta suhelda kõigi teistega. Selleks on MPI'l vaikimisi defineeritud kommunikaator `MPI_COMM_WORLD`, mis sisaldab kõiki protsesse, mis selles paralleelprogrammis parajasti jooksevad. Siiski, suure tõenäosusega soovime omavahel suhtlema panna vaid teatud gruppi protsesse. Kasutades erinevat kommunikaatorit iga grupi jaoks, saame kindlustada selle, et ükski sõnum ei hakka segama teisi sõnumeid teistes gruppides. Juhul kui programm on väga suur, on meil sellest tükeldatud kujul ka tunduvalt parem ülevaade.

Funktsioone `MPI_Comm_size` ja `MPI_Comm_rank` kasutame selleks, et teada saada vastavalt antud kommunikaatoris olevate protsesside arvu (suurus) ja iseenda kui protsessi identifitseerimisnumbrit (järku) mingis kommunikaatoris. Järk on täisarv, mis võib varieeruda nullist kuni kommunikaatori suurus miinus üks. Neid saab välja kutsuda nii:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int Comm::Get_size() const

int MPI_Comm_rank(MPI_Comm comm, int *rank)
int Comm::Get_rank() const
```

3.3 Sõnumite saatmine ja vastuvõtt

Põhilised funktsioonid sõnumite saatmiseks ja vastuvõtmiseks on `MPI_Send` ja `MPI_Receive`. Parameetritena antakse kaasa viit puhvrile, mille sisu saadetakse või kuhu see

1 i. k. intra-/inter-communicator

vastuvõtmisel kirjutatakse, andmetüüp, mis tüüpi andmeid edastatakse/vastu võetakse (vt Tabel 1) ja mitu ühikut, siht/lähteprotsessi identifikaator, kommunikatsioonidomeen ja lipik (int tüüpi täisarv) tegevuse täpsemaks identifitseerimiseks protsessisiseselt. Andmetüüp MPI_BYTE vastab ühele baidile ning MPI_PACKED vastab andmete kogumile, mis on saadud mitte-järjestikulise info pakkimisel. Kõik vastavad parameetrid peavad saatja/vastuvõtja funktsioonis omavahel klappima. Lähteprotsessi ja lipiku märkimiseks saab kasutada ka metaparameetreid: MPI_ANY_SOURCE sobib mistahes saatjaga ning MPI_ANY_TAG mistahes lipiku numbriga.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
void Comm::Send(const void* buf, int count, const Datatype&
datatype, int dest, int tag) const
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
void Comm::Recv(void* buf, int count, const Datatype&
datatype, int source, int tag, Status& status) const
```

Vastuvõtja funktsioonis on veel parameeter MPI_Status[1], eeldefineeritud andmestruktuur, mis sisaldab infot saatja protsessinumbri, lipiku ja veakoodi kohta. See on juhuks, kui kasutatakse metaparameetreid, kuid saatja protsessi identifikaatorit või tagi oleks saajal siiski vaja teada.

Vastuvõtja poolel võib sõnumi pikkuseks olla märgitud ka suurem arv, kui reaalselt saatja sõnumi suurus. See lubab vastuvõtja protsessil mitte teada sõnumi täpset suurust. Kui vastuvõetav sõnum on suurem kui selleks reserveeritud puhvri pikkus, tekib ületäitumise viga ning funktsioon tagastab veakoodi MPI_ERR_TRUNCATE. MPI_Status parameeter annab infot ka sõnumi täpse suuruse kohta. See ei ole kättesaadav otse status muutujast, vaid läbi funktsiooni MPI_Get_count.

1 e. k. staatus, olek

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

Tabel 1: MPI, C ja C++ andmetüüpide vastavus

`MPI_Recv` tagastub alles siis, kui soovitud sõnum on saabunud ja vastavasse protsessi puhvrisse kopeeritud. See tähendab, `MPI_Recv` on blokeeriv funktsioon. MPI lubab aga kahte erinevat `MPI_Send` realiseeringut. Esimesel juhul tagastub see siis, kui vastav `MPI_Recv` on välja kutsutud ja sõnum saadetud. Teisel juhul kopeerib ta saadetava sõnumi süsteemsesse puhvrisse ja tagastub, ilma et ootaks teiselt protsessilt mingitki vastust. Mõlemal juhul saab saatja protsessi algset sõnumiga puhvrit kasutada juba millekski muuks. MPI programmid peavad olema võimelised töötama korrektselt sõltumata sellest, kuidas `MPI_Send` on realiseeritud. Niisuguseid programme kutsutakse ohutuks.

3.3.1 Tupikute vältimine

Sõnumite saatmise ja vastuvõtmise funktsioonide semantika seab mõningaid piiranguid, kuidas me neid kasutada saame. Vaatleme järgmist näidet, kus protsess *i* saadab sõnumi protsessile *i*+1 (protsesside arvu jäägiklassiringis) ja saab sõnumi protsessilt *i*-1 (protsesside arvu jäägiklassiringis).

```
1  int a[10], b[10], npes, myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5  MPI_Comm_size(MPI_COMM_WORLD, &npes);
6  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
   MPI_COMM_WORLD);
7  MPI_Recv(b, 10, MPI_INT,(myrank-1+npes)%npes, 1,
   MPI_COMM_WORLD);
8  ...
```

Kui `MPI_Send` on realiseeritud kasutades puhverdamist, töötab programm korrektselt. Kui aga `MPI_Send` blokeerib kuni teine protsess sõnumi vastu võtab, viimne kui üks protsess hangub, kuna oodatakse, millal naaberprotsess kutsub välja MPI_Recv operatsiooni. Tupik tekib ka juhul kui meil on vaid kaks protsessi. Seega ei ole antud näide ohutu. Seda viga annab aga parandada:

```
1  int a10, b10, npes, myrank;
2  MPI_Status status;
3  ...
4  MPI_Comm_size(MPI_COMM_WORLD, &npes);
```

```
5  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

6  if (myrank%2==1) {

7    MPI_Send(a, 10, MPI_INT,(myrank+1)%npes, 1, MPI_COMM_WORLD);

8    MPI_Recv(b, 10, MPI_INT,(myrank-1+npes)%npes, 1,
   MPI_COMM_WORLD);

9  } else {

10   MPI_Recv(b, 10, MPI_INT,(myrank-1+npes)%npes, 1,
   MPI_COMM_WORLD);

11   MPI_Send(a, 10, MPI_INT,(myrank+1)%npes, 1, MPI_COMM_WORLD);

12 }

13 ...
```

3.4 Kollektiivne kommunikatsioon

MPI pakub suure hulga funktsioone teostamaks tavapäraseid ühis-kommunikatsiooni operatsioone. Kõik kollektiivsed funktsioonid, mida MPI pakub, võtavad argumendina kommunikaatori, mis defineerib grupi nende protsessidega, mis osalevad operatsioonis. Kõik sellesse gruppi kuuluvad protsessid peavad antud kollektiivse rutiini välja kutsuma. Mõningates kollektiivsetes funktsioonides saadab või võtab andmeid vastu üksik protsess. Siis on ka allikas või sihtprotsess funktsiooni üheks argumendiks; kõikides protsessides selles grupis peab olema märgitud sama protsessi identifikaator. Järgmisena esitan komplekti kõige olulisematest üldistest kollektiivsetest funktsioonidest.

```
int MPI_Barrier(MPI_Comm comm)
void Intracomm::Barrier() const
```

Funktsioon tagastub ainult siis, kui kõik protsessid grupis on selle funktsiooni välja kutsunud. Kasutatakse töö käigu sünkroniseerimiseks protsesside vahel. [1]

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
```

```
int source, MPI_Comm comm)
void Intracomm::Bcast(void* buf, int count, const Datatype&
datatype, int source) const
```

Üks protsess `source` saadab samad andmed `buf` igale protsessile kommunikaatoris. Saadud andmed igas protsessis salvestatakse muutujasse `buf`. Saadetud/vastuvõetud andmete hulk (`count`) peavad olema kõigis protsessides samad. See käib loomulikult ka andmetüübi kohta.

```
int MPI_Reduce(void *operand, void *result, int count,
MPI_Datatype datatype, MPI_Op op, int root, MP_Comm comm)
```

```
void Intracomm::Reduce(const void* operand, void* result, int
count, const Datatype& datatype, const Op& op, int root)
const
```

`MPI_Reduce` kombineerib operandid viidatud muutujaga `operand` kasutades operatsiooni `op` ja salvestab tulemuse protsessi `root` muutujasse `result`. Nii operand kui ka result viitavad mäluaadressile, mis sisaldab count järjestikust andmeelementi, mille tüüp on datatype. Operaatoriks on üks MPI poolt eeldefineeritud funktsioonidest (miinimum, maksimum, summa, korrutis, loogikatehted jt) või mõni enda poolt loodud. Kõik argumendid peale operandi(de) ja tulemuse aadressi peavad olema samad kõigis protsessides. Eksisteerib ka funktsioon `Allreduce`, mille ainukeseks vaheks on see, et tulemus salvestatakse kõikides protsessides

```
int MPI_Gather( void* send_data, int send_count, MPI_Datatype
send_type, void* recv_data, int recv_count, MPI_Datatype
recv_type, int root, MPI_Comm comm)
```

```
void Intracomm::Gather(const void* sendbuf, int sendcount,
const Datatype& sendtype, void* recvbuf, int recvcount, const
Datatype& recvtype, int root) const
```

`MPI_Gather` korjab iga protsessi `send_data`ga viidatud andmed ja salvestab `root` protsessi `recv_data`'ga viidatud mälupessa. Tulemuses järjest andmed protsessilt 0, 1, 2,.... Parameetrid `recv_count` ja `recv_type` on tavaliselt vastavalt samad mis `send_count` ja `send_type`. Nad märgivad elementide arvu, mis saadi igalt protsessilt eraldi. `Recv`

parameetrid on olulised ainult `root` protsessi jaoks.

```
int MPI_Scatter(void* send_data, int send_count, MPI_Datatype
send_type, void* recv_data, int recv_count, MPI_Datatype
recv_type, int root, MPI_Comm comm)
```

```
void Intracomm::Scatter(const void* sendbuf, int sendcount,
const Datatype& sendtype, void* recvbuf, int recvcount, const
Datatype& recvtype, int root) const
```

`MPI_Scatter` jagab muutujaga `send_data` viidatud andmed protsessil `root` p (– protsesside arv) segmendiks, igaüks neist koosneb `send_count` elemendist, mille tüüp on `send_type`. Esimene segment saadetakse protsessile 0, teine protsessile 1, jne. `Send` parameetrid on olulised ainult protsessis identifikaatoriga `root`. [2]

4 Sisend/väljund

Korralikku optimeeritust saab rakendada ainult siis, kui paralleelne S/V pakub kõrgtasemel liidest andmete partitsioneerimiseks ja globaalsete andmestruktuuride ülekannet protsesside mälu ja failide vahel. Lisaefektiivsus saavutatakse, kui süsteemil on toetus asünkroonsele S/V-le ning kui omatakse kontrolli andmete füüsilise paiknemise üle andmekandjal (kettal).

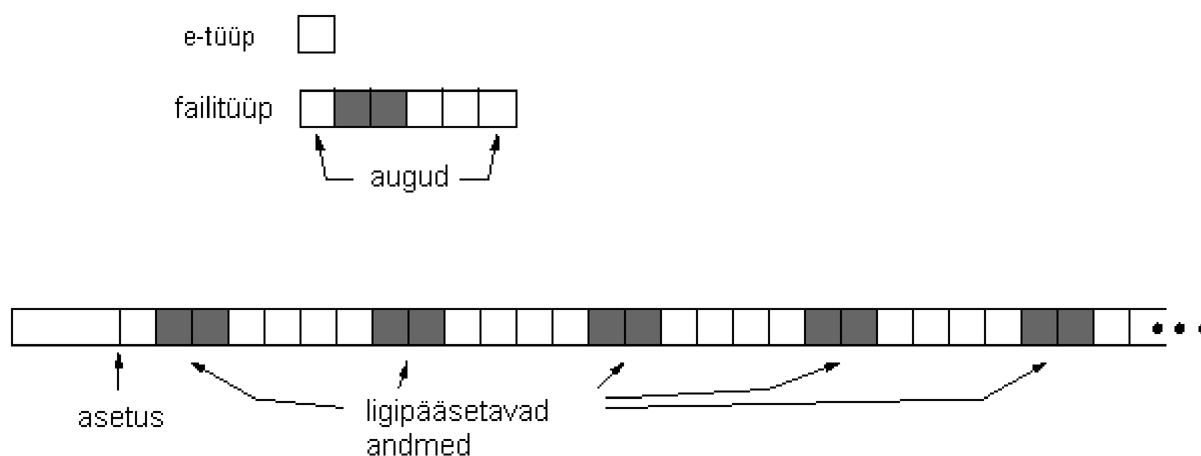
Selle asemel et defineerida S/V ligipääsu moodused erinevatele faili juurdepääsumustritele, valiti veidi teistsugune lähenemine, kus andmete partitsioneerimine väljendatakse tuletatud andmetüüpide abil. Võrreldes lõpliku hulga eeldefineeritud mustritega annab niisugune lähenemine juurde paindlikkust ja väljendusrikkust.

4.1 Mõisted

fail (*file*)MPI fail on kollektsioon andmetükke. MPI toetab suva- või järjestikpöördust mistahes failide tervikhulgale. Fail avatakse kollektiivselt grupi protsesside poolt. Kõik kollektiivsed väljakutsed failile on kollektiivsed grupile.

asetus (*displacement*) Faili asetus on absoluutne baidi positsioon faili alguse suhtes. Asetus defineerib koha, kust algab **vaade**.

e-tüüp (*etype*) Elementaartüüp on ühik andmete lugemisel ja positsioneerimisel. Selleks võib olla mistahes eeldefineeritud MPI andmetüüp. Tuletatud andmetüüpe saab teha, kasutades MPI andmetüübi loomise konstruktoreid. Sõltuvalt kontekstist võib sõna e-tüüp kirjeldada kolme elementaarandmetüübi aspekti: mingi kindel MPI tüüp, andmeüksus seda tüüpi või



Illustratsioon 4: E-tüübid ja failitüübid

selle tüübi ulatus.

failitüüp (*filetype*) on alus faili jagamisel protsesside vahel ja defineerib ligipääsumalli failile. Failitüüp on kas üksik e-tüüp või tuletatud MPI andmetüüp, mis on konstrueeritud mitmest sama e-tüübi instantsist.

vaade (*view*) Vaade defineerib mingist failist praegusel hetkel nähtava ja kättesaadava andmehulga. Tegemist on järjestatud e-tüüpide hulgaga. Igal protsessil on failile oma vaade, mis on defineeritud kolme parameetriga: asetus, e-tüüp ja failitüüp. Failitüübi mustrit korratakse alustades asetusest; see defineeribki vaate. Vaateid saab muuta ka programmi töö ajal. Vaikimise vaade on lineaarne baidijada (asetus on null, e-tüüp ja failitüüp võrdne MPI_BYTE-ga).

Nihe (*offset*) Nihe on positsioon vaate suhtes väljendatuna arvuna e-tüüpides. Augud failitüübis jäetakse selle arvutamisel vahele. Nihe 0 on esimene nähtav e-tüüp vaates (jättes vahele asetuse ja augud alguses).

faili suurus ja faili lõpp MPI faili suurust mõõdetakse baitides faili algusest. Vastloodud fail omab suurust 0 baiti. Mistahes vaate jaoks faili lõpp on nihe esimesest loetavast e-tüübist viimase baidini failis.

failiviit (*file* pointer)Failiviit on mõeldav nihe, mida kasutab MPI. *Individuaalsed failiviidad* on lokaalsed iga protsessi jaoks, mis faili avas. *Jagatud failiviit* on jagatud grupi protsesside poolt, mis faili avasid.

failipide (*file handle*)Failipide on nn läbipaistmatu objekt, mille loob MPI_FILE_OPEN ja vabastab MPI_FILE_CLOSE. Kõik operatsioonid avatud failil kulgevad läbi failipideme.

4.2 Lihtsamad failioperatsioonid

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode,
MPI_Info info, MPI_File *fh)
static MPI::File MPI::File::Open(const MPI::Intracomm& comm,
const char* filename, int amode, const MPI::Info& info)
```

`MPI_File_open` avab etteantud kommunikaatori (`comm`), failinime (`*filename`), viisi (`amode`) ja vihjete (`info`) abil antud faili kõigis kommunikaatori protsessides. Failipide

tagastatakse viidas (fh). Tegemist on kollektiivse funktsiooniga, mis tähendab, et kõik etteantud parameetrid (va vihje) peavad olema samad. Protsess saab faili avada ka sõltumatult teistest protsessides, kasutades kommunikaatorina MPI_COMM_SELF muutujat. Enne MPI_Finalize kutsumist tuleb sulgeda kõik avatud failid. Kommunikaator on täielikult sõltumatu S/V käitumisest.

Faili avamine toetab järgmisi avamisviise

MPI_MODE_RDONLY --- ainult lugemine,

MPI_MODE_RDWR --- lugemine ja kirjutamine,

MPI_MODE_WRONLY --- ainult kirjutamine,

MPI_MODE_CREATE --- tekita fail, kui seda pole,

MPI_MODE_EXCL --- viga, kui tegemas faili, mis juba olemas,

MPI_MODE_DELETE_ON_CLOSE --- sulgemisel kustutatakse fail,

MPI_MODE_UNIQUE_OPEN --- faili ei saa paralleelselt avada ka mujal,

MPI_MODE_SEQUENTIAL --- failipöördus toimub ainult jadamisi,

MPI_MODE_APPEND --- seadista kõik failiviidad viitama faili lõppu.

Neid muutujaid on võimalik bitthaaval VÕI tehtega (|) omavahel kombineerida (mõistlikkuse piirides, näiteks ei saa kasutada korraga MPI_MODE_SEQUENTIAL ja MPI_MODE_RDWR).

MPI_MODE_UNIQUE_OPEN lubab rakendada optimeeritud ligipääsumeetodeid, elimineerides vajaduse fail lukustada. Kasutaja vastutab selle eest, et väliseid pöördusi failile ei toimuks. MPI_Info muutujat kasutatakse selleks, et anda süsteemile vihjeid juurdepääsumustrite ja mõnede teiste eriärasuste kohta failiga ümberkäimisel.

```
int MPI_File_close(MPI_File *fh)
void MPI::File::Close()
```

Faili sulgemisel teostatakse kõigepealt sünkroniseerimine ja alles seejärel faili sulgemine.

`MPI_File_close` on kollektiivne funktsioon. Kasutaja vastutab, et kõikvõimalikud mitteblokeeruvad ja lõhestatud funktsioonid sellel failil on eelnevalt oma töö lõpetanud. Failipideme väärtuseks seatakse MPI_FILE_NULL.

```
int MPI_File_delete(char *filename, MPI_Info info)
static void MPI::File::Delete(const char* filename, const
MPI::Info& info)
```

`MPI_File_delete` kustutab antud failipidemega viidatud faili. Info argumenti saab kasutada andmaks infot failisüsteemi iseärasustest. Kui faili ei ole olemas, tekib viga MPI_ERR_NO_SUCH_FILE.

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
void MPI::File::Set_size(MPI::Offset size)
```

`MPI_File_Set_size` seab failipidemega viidatud faili suuruse. Suurust mõõdetakse baitides faili algusest. Kui `size` on väiksem kui faili endine suurus, siis seda ületav osa lõigatakse lihtsalt maha. Kui aga suurem, saab faili suuruseks `size`. Olemasolevad andmed jäävad puutumata ning juurdetuleva osa sisu ei ole defineeritud. Kui faili avamine toimus viisil `MPI_MODE_SEQUENTIAL`, siis on selle funktsiooni väljakutsumine vigane. Ka siin on oluline, et mitteblokeeruvad ning lõhestatud funktsioonid oleksid oma töö lõpetanud. Tegemist on kollektiivse protseduuriga.

Leiduvad ka funktsioonid avamismeetodi ja faili suuruse küsimiseks.

4.3 Vihjed

Info on läbipaistmatu[1] objekt, mis koosneb (võti, väärtus) paaridest (nii võti kui väärtus on sõned). Ühel võtmel võib olla ainult üks väärtus. MPI-l on mitmed reserveeritud võtmed ning nõuab, et kui MPI implementatsioon kasutab reserveeritud võtit, peab ta ka pakkuma sellega seotud funktsionaalsust. Samas, ei ole nõutud MPI reserveeritud võtme(te) toetamist. Saab ka defineerida oma võtmeid. Võtmetele on määratud realisatsioonist sõltuv maksimaalne pikkus MPI_MAX_INFO_KEY, mille väärtus peab olema vahemikus 32 kuni 255. Sarnaselt on väärtustel olemas MPI_MAX_INFO_VAL. Nii võti kui väärtus on tõstutundetud.

1 *i k opaque*

Leiduvad funktsioonid Info objekti loomiseks, võtme/väärtuse seadmiseks/muutmiseks/pärimiseks/kustutamiseks. On võimalik küsida ka võtmete arvu, i-ndat võtit, kopeerida ja kustutada info.

Vihjed, mida saab anda info objekti kaudu, lubavad kasutajal anda süsteemile ette faili ligipääsu mustreid ja muud süsteemispetsiifilist infot. See võimaldab süsteemi optimeerida nii et jõudlus ja/või ressursikasutus on nö paremad. Siiski, vihjed ei muuda MPI toimimise semantikat, ehk teisisõnu, vihjeid võidakse ignoreerida. Vihjeid on võimalik anda faili avamisel, kustutamisel, vaate seadmisel ning muidugimõista faili vihjete seadmisel. Info seadmine on kollektiivne rutiin, mis vastavalt võtmele võib protsessiti seada ka erinevaid vihjeid.

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
MPI::Info MPI::File::Get_info() const
```

Funktsioon tagastab uue info objekti, mis sisaldab vihjeid antud faili kohta. Hetkel reaalselt kehtivad parameetrid sellel failiga sisalduvad muutujas `info_used`.

MPI standardis on kirjeldatud 15 „potentsiaalselt kasulikku“ võtit seoses failidega. Need hõlmavad peamiselt juurdepääsumustreid ja andmete füüsilist paigutust andmekandjal.

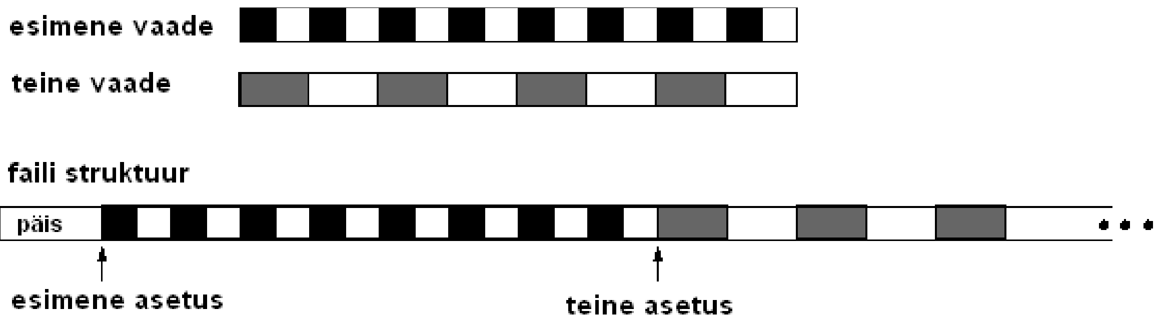
4.4 Vaated

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
MPI_Datatype etype, MPI_Datatype filetype, char *datarep,
MPI_Info info)
void MPI::File::Set_view(MPI::Offset disp, const
MPI::Datatype& etype, const MPI::Datatype& filetype, const
char* datarep, const MPI::Info& info)
```

MPI_File_set_view rutiin muudab protsessi vaadet andmetele antud failis. Ette antakse fail, asetus (`offset`), e-tüüp, failitüüp, andmete esitlusviis (*„native“/“internal“/“external32“)* ja vihjed. Funktsioon seab individuaalsed ja jagatud failiviidad nulliks. Ta on kollektiivne funktsioon, esitlusviis ja e-tüüp peavad olema kõikidel protsessidel samad, teised võivad erineda.

Kui e-tüüp on porditav andmetüüp, siis arvutatakse e-tüübi ulatus[1] muutes[2] asetusi nii, et need klapivad andmete esitlusviisiga. Kui e-tüüp ei ole porditav, siis skaleerimist ulatuse arvutamisel ei tehta ning kasutaja peab ise hoolitsema andmete ühildumise eest.

Tuletame meelde, et ligipääs andmetele toimub e-tüüpide kaupa, lugedes või kirjutades andmeid, mille tüübiks on e-tüüp. Nihe väljendatakse e-tüüpides, failiviidad viitavad e-tüüpide algusele. Kui fail avatakse kirjutamiseks, siis ei e-tüüp ega failitüüp ei tohi sisaldada kattuvaid



Illustratsioon 5: Asetused

regioone. See reegel ei kehti failitüüpidele erinevatest protsessidest. Kui failitüüp sisaldab endas auke, siis on andmed nendes sellele protsessile kättesaamatud. Siiski, e-tüüpi, asetust ja failitüüpi saab ligipääsu huvides muuta, kutsudes välja `MPI_Set_file_view`. E-tüübi ja failitüübi loomisel on viga kasutada absoluutaadresse. Argument `datarep` on sõne, mis väljendab andmeesitlusviisi failis. Kasutaja vastutab selle eest, et kõik mitteblokeeruvad ning lõhestatud operatsioonid sellel failil on oma töö lõpetanud, enne kui kutsutakse `MPI_Set_file_view`. Vastasel juhul tekib viga.

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp,
     MPI_Datatype *etype, MPI_Datatype *filetype, char
*datarep)
void MPI::File::Get_view(MPI::Offset& disp, MPI::Datatype&
     etype, MPI::Datatype& filetype, char* datarep) const
```

Ette antakse failipide, kõik teised väärtused on need, mida teada tahetakse saada.

1 *i k extent*
2 i k *to scale*

Positsioneeri-mine	***Sünkronisat-sioon***	***Koordineeritus***	
		Mittekollektiivne	*Kollektiivne*
Ilmutatud nihe	*Blokeeriv*	MPI_FILE_READ_AT MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	Mitteblokeeriv & lõhestatud kollektiivne	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
Individuaalne failiviit	*Blokeeriv*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	Mitteblokeeriv & lõhestatud kollektiivne	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
Jagatud failiviit	*Blokeeriv*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	Mitteblokeeriv & lõhestatud kollektiivne	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Tabel 2: Mitmesugused juurdepääsufunktsioonid

4.5 Andmete ligipääs

Andmeid liigutatakse failide ja protsesside vahel, kutsudes välja lugemis- ja kirjutamisoperatsioone. Seejuures leidub kolm sõltumatut aspekti:

- positsioneerimine (ilmutatud vs failiviit)
- sünkronisatsioon (blokeeriv vs mitteblokeeriv, lõhestatud kollektiivne)
- koordineeritus (kollektiivne vs mittekollektiivne)

Järgmisena toome ära kombinatsioonid nendest funktsioonidest, kusjuures eristame ka individuaalseid ja jagatud failiviitu.

4.5.1 Positsioneerimine

MPI pakub kolme erinevat andmepositsioneerimist: ilmutatud nihe, individuaalne failiviit ja jagatud failiviit. Erinevad meetodeid võib samas programmis omavahel segada, nad ei mõjuta

üksteist.

Juurdepääsufunktsioonid, mis aktsepteerivad ilmutatud nihet, sisaldavad oma nimes täheühendit _AT (näit MPIFILE_WRITE_AT). Andmete lugemine/kirjutamine toimub faili antud positsioonis, failiviita ei kasutata ega uuendata. Individuaalse failiviidaga funktsioonid ei sisalda oma nimes mingit identifikaatorit, küll aga saab nii eristada jagatud failiviidaga funktsioone, millede nimes on _SHARED või _ORDERED.

Peamine semantiline küsimus MPI failiviitadega on see, kuidas ja millal neid S/V operatsioonide poolt muudetakse. Üldiselt jätab operatsioon failiviida viitama järgmisele andmeühikule peale seda, mida ta on juba kasutanud.

4.5.2 Sünkronisatsioon

MPI toetab blokeerivaid ja mitteblokeerivaid S/V rutiine. Blokeeriv S/V kutse ei tagastu kuni kutsung on lõpetanud. Mitteblokeeriv käivitab S/V operatsiooni, kuid ei jää ootama selle lõpetamist. Kindlustamaks, et töö on lõpetatud ning et puhvreid tohib taaskasutada, on eraldi vaja kutsuda teatud funktsioone (MPI_TEST, MPI_WAIT). Mitteblokeeriva funktsiooni nimi sisaldab sõne „_I“. On viga kasutada lokaalset puhvrit andmete allika või sihtkohana teistes funktsioonides või seda lugedes/kirjutades, kui mitteblokeeriv kutsung, mis seda puhvrit kasutab, ei ole oma tööd veel lõpetanud.

4.5.3 Koordineeritus

Igale mittekollektiivsele ligipääsufunktsioonile vastab oma kollektiivne funktsioon. Enamike rutiinide jaoks on selleks MPI_FILE_{READ|WRITE|...}_ALL või paar MPI_FILE_ {READ_ALL|WRITE_ALL|...}_{BEGIN|END}.

Kui mittekollektiivsete funktsioonide puhul oleneb kutsungi lõpetamine vaid väljakutsuva protsessi tegevustest, siis kollektiivse protsessi puhul võib funktsiooni edukus sõltuda teistest protsessidest. Kollektiivsed rutiinid võivad olla tunduvalt kiiremad kui nende mittekollektiivsed vasted, kuna globaalsel andmejuurdepääsul on suur potentsiaal automaatse optimiseerimise rakendamiseks.

4.5.4 Kokkulepped

Andmeid mälus piiritleb kolmik `buf`, `count` ja `datatype`. Operatsiooni lõppemisel sisaldub muutujas `status` loetud/kirjutatud andmete hulk. Keeles C++ pole staatus-argument kohustuslik. Juurdepääsufunktsioonid püüavad andmeid (mille tüübiks on `datatype`) liigutada kasutaja puhvri `buf` ja faili vahel. Andmetele saab ligi vastavalt seatud vaatele.

Mitteblokeerivad funktsioonid osutavad seda, et MPI võib alustada andmete lugemist/kirjutamist ning seovad päringu pideme, `request`, S/V operatsiooniga. Mitteblokeerivad operatsioonid täidetakse funktsioonide MPI_TEST, MPI_WAIT või mõne nende variatsiooni kaudu. Tüüpiline S/V funktsioon näeb välja umbes nii:

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void
*buf, int count, MPI_Datatype datatype, MPI_Status *status)
void MPI::File::Read_at(MPI::Offset offset, void* buf, int
     count, const MPI::Datatype& datatype, MPI::Status&
status)
void MPI::File::Read_at(MPI::Offset offset, void* buf, int
     count, const MPI::Datatype& datatype)
```

4.5.5 Lõhestatud kollektiivsed rutiinid

MPI pakub piiratud kujul kollektiivseid mitteblokeeruvaid funktsioone kasutades lõhestatud[1] kollektiivseid rutiine. Viitan nendele funktsioonidele kui „lõhestatud“, sest üks kollektiivne operatsioon on jagatud kaheks: algusfunktsioon ja lõpufunktsioon. Algusfunktsioon, nagu nimigi ütleb, alustab operatsiooni ja lõpufunktsioon lõpetab selle. Nende vahepeal on keelatud kasutada puhvrit, mis kasutaja algusfunktsioonile parameetriks andis. Lõhestatud kollektiivsetele funktsioonidele failipidemega `fh` kehtivad allpoolsed reeglid.

- Mistahes MPI protsessil ei tohi ühel failipidemel mis tahes hetkel korraga käia ühtki teist (lõhestatud või regulaarset) kollektiivset operatsiooni, kui lõhestatud kollektiivne rutiin käib.

1 i k *split*

- Algus- ja lõpufunktsioonid on kollektiivsed selle protsesside grupi jaoks, kes osalesid faili kollektiivsel avamisel. Igale lõpufunktsioonile peab vastama täpselt üks eelnevalt väljakutsutud paariliseta algusfunktsioon.
- Lõhestatud kollektiivsed funktsioonid ei paaru regulaarsete kollektiivsete operatsioonidega.
- Algus- ja lõpufunktsioonid peavad argumendiks andma sama puhvri.
- Mitmelõimelises programmis peavad algus/lõpufunktsiooni välja kutsuma sama lõim.

Nende funktsioonide argumentidel on täpselt sama tähendus, mis nendega ekvivalentsetel kollektiivsetel versioonidel. Lõhestatud kollektiivne algusfunktsioon ning lõpufunktsioon viivad sama tulemuseni, mis nendele vastatav kollektiivne funktsioon. [3]

4.6 Failiesitusviisid

Kuna erinevatel masinatel on erinevad esitusviisid binaarsete andmete jaoks—baidijärjestus, andmetüüpide suurused, *etc*—siis ei pruugi ühe masina peal loodud failid olla niisama lihtsalt arusaadavad ka teistes masinates: tuleb sellega arvestada. MPI pakub kasutajatele võimalust luua portatiivseid faile, mida on võimalik lugeda ka teistes masinates. Asi toimib läbi funktsiooni `MPI_File_set_view` argumendi `datarep`.

Parameeter `datarep` märgib, kuidas tuleb mitmeid andmetüüpe (täisarvud, ujukomaarvud) püsisalvestusseadmele paigutada. MPI toetab mitmeid esitusi. Kolm neist on eeldefineeritud MPIs endas, nimeliselt `native`, `internal`, ja `external32`. MPI standardi realiseerija võib pakkuda veel omapoolseid esitusviise. MPI lubab ka kasutajail endil defineerida uusi viise andmete säilitamiseks ja siis käitusaegselt neid töösse lisada (seda läbi konverteerimisfunktsioonide).

Esitusviisi `native` puhul salvestatakse andmed faili täpselt nii, nagu nad paikevad mälus, andmete konverteerimist ei tehta. See on ühtlasi vaikeväärtus. Kuna puudub konverteerimine, siis puudub ka jõudluskadu S/V operatsioonides või andmete täpsuses. Seda meetodit ei saa kasutada masinas, kus on mälus kasutusel teistsugune andmete esitusviis. Teiste sõnadega, `native` ei ole portatiivne.

Esitusviis `internal` on realiseerija poolt defineeritud esitusviis, mis võib pakkuda (olenevalt realiseerijast) mõningat portatiivsust. Näiteks, realiseerija võib defineerida `internal` esitusviisi, mis on portatiivne kõikides selle tootja selle MPI realisatsiooniga masinates. Teised tootjad võivad, kuid ei pea olema võimelised sellest aru saama. Võimalik on ka see variant, et realiseerija valib sisemiseks esitlusviisiks `external32`-e.

Kõige keerulisem, kuid samas kasutajale kõige suuremat funktsionaalsust pakkuv meetod on `external32`. Faili, mis on kirjutatud `external32` esitlusviisist lähtudes, on võimeline lugema mistahes realisatsioon mistahes masinas. Kuna see meetod võib nõuda realiseerijalt andmete teatud konverteerimist, siis võib juhtuda, et S/V jõudlus on väiksem ning et esineb teatavat täpsuse kadu ujukomaarvude esitamisel. Ehk siis: seda esitlusviisi tuleks kasutada vaid siis, kui portatiivsus on elutähtis.[4]

4.7 Terviklikkus ja semantika

Terviklikkuse semantika defineerib tulemuse, kui failile toimub pöördus mitmest kohast korraga. Kõik failipöördused MPIs on seotud teatud tüüpi failipidemega, mis luuakse, kui fail kollektiivselt avatakse. MPI pakub kolme terviklikkuse taset: järjestikuline terviklikkus üle kõigi pöörduste läbi üheainsa failipideme, järjestikuline terviklikkus üle kõigi pöörduste kasutades failipidemeid, mis tekitatud ühest kollektiivsest avamisest, atomaarne laad sisse lülitatud, ning kasutaja enda poolt järgitav terviklikkusmudel, mis ei ole mõni eelpoolmainitud. Järgnevalt vaatleme paari olulist abivahendit-funktsiooni.

```
int MPI_Sile_set_atomicity(MPI_File fh, int flag);
void MPI::File::Set_atomicity(bool flag)
```

Funktsioon lülitab antud failipidemel sisse/välja atomaarse laadi. Laad võimaldab S/V operatsioone täita kui hetkega rakendatud tegevusi. Funktsioon rakendub ainult uutele pöördustele.

```
int MPI_file_sync(MPI_file fh)
void MPI::File::Sync()
```

Funktsiooni kutsumine põhjustab kõikide eelnevate kirjutamisoperatsioonide kandmise püsisalvestusseadmele.[3]

5 Kokkuvõte

Töös tutvustati üldisi sõnumiedastusprintsiipe ja tehnikaid. Seejärel vaadeldi mõningaid olulisemaid MPI funktsioone, mille najal kogu standard üles on ehitatud. See oli vajalik, et anda taustateadmised ja valimisolek uurida, mida uut pakub MPI2 sisend-väljundi osas. Nägime, et MPI2'e poolt pakutav mudel on vägagi võimalusterohke ning paindlik. Oli olemas suur hulk sisseehitatud võimalusi, kuid ettenägelikult oli jäetud ruumi ka kasutajapoolsete laienduste jaoks.

6 Abstract

Numerous programming languages and libraries have been developed for explicit parallel programming. The message-passing programming paradigm is on of the oldest and most widely used approaches for programming parallel computers. Message-passing programs are often written using the asynchrounous or loosely synchronous paradigms.

Since interactions are accomplished by sending and receiving messsages, the basic operations in the messag-passing programming paradigm are send and receive. We could implement interactions between sender and receiver as buffered blocking, non-buffered blocking, buffered non-blocking or non-blocking non-buffered, each one having its own benefits and drawbacks.

MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using C, C++ , Fortran, etc. The MPI standard defines both the syntax as well as the semantics of a core set of library routines. The very first ones a programmer must now are for intitializing and terminating the MPI environment: Init and Finalize. A key concept used throughout MPI is that of the communication domain. Information about communication domains is stored in a variable called communicator. There are functions used to determine the number of processes in a communicator and the label of the calling process. All of the collective communication functions provided by MPI take as an argument a communicator.

The significant optimizations required for efficiency (e.g. grouping, collective buffering, and disc-directed I/O) can only be implemented if the parallel I/O system provides a high-level interface supporting complete transfers of global data structures between process memories and files.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), MPI chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

7 Kasutatud kirjandus

1. Grama, A., Gupta, A., Karypis, G., Kumar, V (2003) Introduction to parallel computing. Addison-Wesley, 635 lk.

2. Pacheco, P. S. (1997) Parallel programming with MPI. Morgan Kaufman Publishers Inc, 419 lk.

3. Gropp, W., Huss-Lederman S., Lumsdaine A., jt (1998) MPI—the complete reference, Volume 2, The MPI Extensions. The MIT Press, 344 lk; http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

4. Gropp, W., Lusk, E., Thakur, R. (1999) Using MPI2, Advanced Features of the Message-Passing Interface. The MIT Press, 382 lk.