T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Mai-Liis Karring

MPI programmide jõudlusanalüüsi meetodid ja vahendid

Magistritöö (20 AP)

Juhendaja: prof. Eero Vainikko

Autor: ..... "....." mai 2007
Juhendaja: ..... "....." mai 2007

TARTU 2007

Sisukord

Sissejuhatus **4**

1 MPI - Message Parsing Interface **5**
1.1 Teateedastus ja teateedastusliides 5
1.2 MPI programmide eripärad 7

2 Profileerimine ja jõudlusanalüüs **12**
2.1 Profileerimine 12
2.2 Lühiülevaade MPI programmide profileerijatest 16

3 Põhjalikum ülevaade mõningatest MPI profileerijatest **18**
3.1 MpiP 18
3.2 Allinea OPT 21
3.3 MPE ja Jumpshot-4 28

4 Tarkvarapakett DOUG **35**

5 Profileerimisinfo analüüs tarkvarapaketil DOUG **41**

Kokkuvõte **46**

Resümee (inglise keeles) **47**

Kasutatud kirjandus **48**

Lisad **54**

Sissejuhatus

Jõudlusanalüüs ja skaleeruvusprobleemide tuvastamine on paralleelprogrammide arendamise juures väga oluline valdkond, kuna paralleelsus kaotab oma eesmärgi, kui enamal hulgal protsessoritel paralleelselt jooksutatuna programmi töö efektiivsus hoopis langeb tõusmise asemel. Kuna paralleelprogrammide kirjutamine on üldjuhul keerukam ja aeganõudvam ülesanne, kui tavaprogrammide kirjutamine, siis ressursside paremaks ja läbimõeldud kasutamiseks on jõudlusanalüüsi läbiviimine kasulik tegu ning jõudluse tõstmine üks olulisi eesmärke. Paralleelprogrammides, mis on üles ehitatud teateedastusmudelile, võib jõudluse juures lisatakistuseks osutuda ka protesside omavaheline suhtlusalgoritm.

Käesoleva töö eesmärk on uurida, milliseid vahendeid leidub MPI programmide kommunikatsiooni- ja sünkronisatsiooniosa jõudlusanalüüsi läbiviimiseks ning tutvustada mõnda neist pikemalt. Ühtlasi on eesmärgiks selgitada välja, milliseid kitsaskohti võimaldavad MPI programmide profileerijad tuvastada ning kuidas leida skaleeruvusprobleeme tekitavaid kohti profileeritavas programmis. Lisaeesmärgiks on peale profileerimistarvaradega tutvumist, kasutada mõnda neist tarkvarapaketil DOUG [13] ning ülevaatlikult analüüsida kogutud infot.

Põhjalikumaks analüüsiks eelkõige tarkvarapaketi DOUG arendajatele aga ka kõigile teistele huvilistele on genereeritud logifailid lisana CD plaadil käesolevale tööle kaasa pandud.

Peatükk 1

MPI - Message Parsing Interface

1.1 Teateedastus ja teateedastusliides

Message passing ehk teateedastus on paralleelprogrammeerimise meetod, mille abil andmed ühe protsessori mälust kopeeritakse teise protsessori mällu [35]. Teateedastust on tarvis protsesside omavaheliseks kommunikatsiooniks ning sünkronisatsiooniks paralleelprogrammides. Teateedastuse programmeerimismudel on üks enamlevinumaid paralleelprogrammide programmeerimismudeleid tänapäeval. See koosneb paralleelselt jooksvatest töödest, mis jagatakse olemasolevate protsessorite vahel ning millel igaühel on kasutamiseks vaid talle ligipääsetavad andmed. Igal tööl on oma unikaalne nimi ning omavahel vahetatakse andmeid sõnumite saatmise teel kahe töö vahel või tööde hulgal.

Message Passing Interface (edaspidi MPI) on standard, mis kirjeldab tarkvarateeki, mille abil kirjutada teateedastusega paralleelprogramme, mis jooksevad hajuskeskkonnas [9]. Standardi loojaks on MPI Forum ning esialgne versioon MPI-1.0 lasti välja aastal 1994. Praegune kasutuselolev versioon on MPI-2.0, aastast 1998 [24]. Tänapäeval on MPI kujunenud *de facto* standardiks paralleelprogrammide kirjutamisel.

Põhjus, miks MPI laialdast kasutuspinda on leidnud, on selle porditavus. MPI programme saab käivitada igal platvormil, kuhu mõni MPI realisatsioon on paigaldatud, ilma et peaks programmi koodi muutma, seega on MPI programmid sõltumatud masina ja võrgu arhitektuurist [35].

Leidub palju erinevaid MPI realisatsioone. Enamlevinumad on vabavaralised, kuid leidub ka tasulisi. Alljärgnevalt on toodud lühiülevaade mõningatest realisatsioonidest [25].

- MPICH1 [4] - Mississippi State University Argonne National Laboratory poolt välja töötatud MPI-1.2 standardi realisatsioon, mille viimased versioonid sisaldavad ka olulisemaid osi MPI-2.0 standardist.

- MPICH2 [3] - MPICH2 on Mississippi State University Argonne National Laboratory poolt valminud MPI-2.0 standardile täielikult vastav edasiarendus realisatsioonist MPICH1.

  MPICH1 ja MPICH2 on avatud lähtekoodiga vabavara, mis on olnud aluseks paljudele teistele MPI realisatsioonidele, ka kommertsiaalsetele ning tootjapõhistele arhitektuurispetsiifilistele lahendustele.

- LAM [16] - USA Ohio osariigi poolt asutatud arvutusalaste teadusuuringute keskusest Ohio Supercomputing Centre alguse saanud vabavaraprojekt, mis hetkel on Indiana University Open Systems Laboratory hooldada. Arendustöid selles enam ei tehta. Ka LAM-MPI ei toeta täielikult MPI-2 standardit. LAM-MPI oli algusest peale kavandatud heterogeensete klastersüsteemide jaoks, mis on teinud ta populaarseks mitmeklastrilistes süsteemides. LAM-MPI sisaldab ka tuge Globus griidisüsteemile ning PBS tööjärjekorrasüsteemile. Ühtlasi kasutab LAM-MPI IMPI *Interoperable MPI* standardit, mis on ettevõtluse initsiatiivil loodud reeglistik, mis võimaldab MPI programmil kasutada üheaegselt erinevaid MPI realisatsioone, saavutamaks nõnda parema jõudluse ka heterogeensetes keskkondades.

- Open MPI [26] - OpenMPI ühendab endas mitmeid erinevaid MPI realisatsioonide projekte (kaasaarvatud LAM-MPI) ning on seadnud eesmärgiks koostada "parim olemas olev MPI teek"[26]. OpenMPI projektil on lai hulk sponsoreid ja toetajad nii eraettevõtluse, haridusasutuste kui ka teaduskeskuste näol ning seda kutsutakse ka "järgmise põlvkonna MPI realisatsiooniks".

- MPI/pro [42] - Kommertsiaallahendus Verari Systems Software Inc poolt, mis tootja sõnul ületab vabavaraliste MPI realisatsioonide jõudluse 10-20%.
- DeinoMPI [10] - MPICH2 põhjal arendatud vabavaraline MPI teek Windowsi operatsioonisüsteemi jaoks.

1.2 MPI programmide eripärad

MPI programmide loomisel on erinevalt tavaprogrammidest arendajatel vaja lahendada palju lisaülesandeid. Vaja on luua tarkvara osa, mis tegeleks programmi paralleeliseerimise juhtimisega - kommunikatsiooni ja sünkronisatsiooniga erinevate protsesside vahel, andmete partitsioneerimise ja laialijaotamisega, protsesside jagamisega protsessorite vahel ning andmestruktuuride lugemise ja kirjutamisega [33]. Suhtlusmustreid võib vaadelda üldisemalt jagatuna nelja erinevat mõõdet pidi:

- lokaalne või globaalne;
- struktureeritud või struktureerimata;
- staatiline või dünaamiline;
- sünkroonne või asünkroonne.

Kui protsess suhtleb vaid väikese hulga teiste protssessidega, oma naaberprotsessidega, siis on tegu lokaalse suhtlusega. Globaalse suhtluse korral suhtleb protsess kõigi protsessidega. Struktureeritud kommunikatsiooni korral moodustavad protsessid oma suhtluses mingi kindla struktuuri, näiteks puu või toru, struktureerimata kommunikatsiooni korral moodustatakse juhuslikke graafe. Staatilises suhtlusmudelis teateid omavahel vahetavad osapooled aja kulgedes ei muutu, dünaamilises mudelis võivad kommunikatsioonipartneri määrata näiteks käivitamisajal arvutatud andmed ning see võib pidevalt varieeruda. Sünkroonses suhtluses saatja ja vastvõtja tegevus on koordineeritud, asünkroonses suhtluses võib saaja andmeid vastu võtta ka saatja koostööta [14].

MPI suhtlusmudeli disain on keerukas ülesanne. Ian Fosteri poolt on artiklis "Designing and Building Parallel Programs"[14] pakutud välja lühike kontrollküsimustik ennetamaks skaleeruvuse ja ebaefektiivsuse probleeme juba enne planeeritud disaini realiseerimist.

1. Kas kõik programmi alamülesanded kasutavad ligikaudselt võrdsel hulgal kommunikatsiooni? Kui vaid ühel protsessil on ligipääs mingile tihedalt kasutatavale andmestruktuurile, siis tuleks kaaluda selle andmestruktuuri hajutamist või peegeldamist teistesse programmi osadesse.

2. Kas iga protsess suhtleb vaid vähese arvu naabritega? Kui igal protsessil on vaja suhelda rohkemate protsessidega, siis on soovitatav jagada globaalne kommunikatsioon lokaalseteks suhtlusgruppideks

3. Kas kommunikatsioonioperatsioonid saavad töötada paralleelselt?

4. Kas erinevate protsessidega seotud arvutusülesanded saavad töötada paralleelselt?

Nendele ja teistele skaleeruvus- ja jõudlusküsimustele aitavad olemasolevate MPI programmide korral vastuse leida ka MPI profileerijad, millest tuleb lähemalt juttu järgnevates peatükkides.

Disainides teateedastusprogramme, kasutatakse tihti **töö ja kanali** (*task-channel*) **mudelit**, kuna selle abil saab kavandada kogu paralleelprogrammmi kommunikatsioonistruktuuri. Töö ja kanali mudel paralleelselt jooksvatest töödest ja neid ühendavatest kanalitest. Iga töö jooksutab üht jadaprogrammi ning kasutab kohalikku mälu. Töö ja kanali ühenduskohtades (sissetulevates ja väljaminevates portides) on defineeritud töö liides muude töödega kanalite kaudu suhtlemiseks. Töö saab teostada 4 erinevat tüüpi operatsioone:

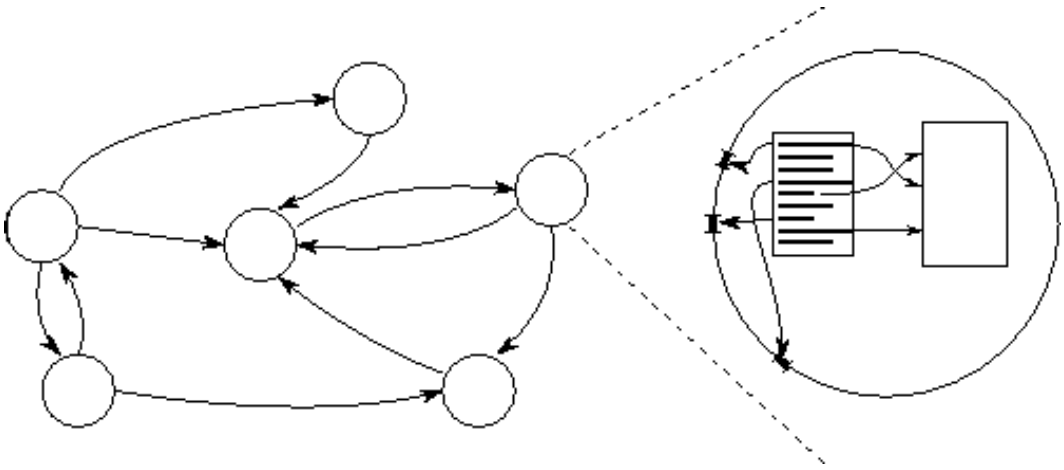
1. andmete mälust (kohalikust) lugemine ja mällu kirjutamine;

2. teadete saatmine väljaminevatest portidest ning teadete vastuvõtmine sissetulevatest portidest;

3. uute tööde loomine;

4. töö lõpetamine.

Siin tuleb tähele panna, et antud mudelis on teate saatmine asünkroonne operatsioon, st teate saatmine toimub kohe, kui vastav käsk on tulnud. Vastuvõtmine aga on sünkroonne tehe, sest käimas olev programm peatatakse ning jätkata saab vaid siis, kui oodatav teade on saabunud. Sissetulevad ja väljaminevad pordid tööde vahel on ühendatud teadetejärjekordadest koosnevatest kanalitest. Kanaleid saab paralleelprogrammi töö käigus luua ja kustudada, seega ühendus eri tööde vahel varieerub dünaamiliselt. Viis, kuidas töid erinevate protsessoritega seotakse ei muuda programmi struktuuri, üldjuhul üks või enam protsessi seotakse ühe protsessoriga. Tööde ja kanalite mudelit illustreerib joonis 1.1, kus tööd on tähistatud ringiga ja neid ühendavad kanalid nooltega [14]



Joonis 1.1: Töö ja kanali programmeerimismudel [14] (joonis 1.7)

Tööde ja kanalite mudeli eeliseks on asjaolu, et see garanteerib programmi determineerituse, st ei juhtu olukorda, kus samade sisendparameetrite korral võib programmi tulemus olla kord ühesugune, kord teistsugune (tulemus on alati ette ennustatav), sest igal kanalil on vaid üks saatja ja üks vastuvõtja. Seega, töö A saab eristada talle saabuvaid sõnumeid töödelt B ja C, kuna need saabuvad erinevaid kanaleid pidi.

Teateedastuse programmeerimismudel loob sarnaselt tööde ja kanalite mudelile mingi arvu töid (üldiselt rakendatakse teateedastusmudelid,

kus programmi käivitamisel luuakse mingi kindel arv töid, mille hulk programmi käigus ei muutu, ning mis lahendavad sama ülesannet, kuid erinevatel andmestruktuuri osadel), kuid kanaleid otseselt teateedastusmudel ei toeta. Iga tööl on oma unikaalne nimi ning selle asemel, et saata mingi teade mingisse kanalisse, saadetakse teade otse kindlale adressaadile, täpsustades saaja nime. Selles mudelis aga vaikimisi ei ole garanteeritud, et tööle A saadetud sõnumid töödelt B ja C jõuavad ka tööle A kohale samas järjekorras, kui need saadeti, mistõttu on seda mudelit realiseerivad programmid mittedetermineeritud, mis aga teeb programmi veaaltimaks ja vähem usaldatavaks. Küll aga kasutades teaateedastuseks MPI-d, on kindlustatud, et teated saabuvad samas järjekorras, kui need saadeti. MPI võimaldab info vastuvõtul täpsustada, milliselt töölt teadet oodatakse, kuid samas võimaldab ka jätta selle lahtiseks märkides saatja kohale tähise `MPI_ANY_SOURCE`. Samuti on võimalus eristada sama saatja erinevaid sõnumeid, kui määrata oodatava teate märgistus (parameeter *tag*), mille korral võetakse vastu vaid konkreetse märgistusega teateid. Samuti võimaldab MPI jätta märgistuse tähistamata, märkides parameetri *tag* kohale tähise `MPI_ANY_TAG`. Seega on võimalik kirjutada nii determineeritud kui ka mittedetermineeritud MPI programme. Esimene on neist eelistatum, sest nõnda välistatakse vead, mis võivad tuleneda sellest, et teated ei saabu oodatud järjekorras (teateid ei saadeta oodatud järjekorras).

MPI programmide kirjutamise juures tuleb arvestada ka tupikseisude (*deadlock*) tekkevõimalusega. Tupikseis on olukord, millal erinevad MPI käsud tekitavad omavahel sellise sõltuvuse, mida ei ole võimalik täita ning programm jääb lõpmatuseni ootama [12]. Näiteks `MPI_Send` käsk on blokeeriv käsk, mis ei saa enne tööd jätkata, kui vastav teade on lõplikult teele saadetud, ent kui saadetav teade on pikem, kui süsteemne teateedastuspuhver võimaldab mahutada, tuleb teade saata mitmes jaos ehk siis tuleb oodata kõigepealt, millal teine töö on andmed vastu võtnud ning süsteemne teateedastuspuhver taas vaba. Kui aga vastuvõtja ise aga samal ajal ei oota teate vastuvõtmist, vaid üritab samuti saata süsteemsest puhvrist pikemat teadet esimesele tööle, siis tekibki hetk, mil kumbi töö ei saa jätkuda, sest vastaspool ei võta teadet vastu ning teadet ei saa vastu võtta, sest eelmine käsk `MPI_Send` pole lõpule viidud [39]. Ühtlasi tekib tupik ka siis, kui mõlemad tööd kutsuvad välja blokeeriva MPI_Receive käsu, jäädes üksteiselt teadet

ootama, kuid kumbki ei ole eelnevalt üksteisele teadet saatnud, mida teine pool võiks vastu võtta. Kuna teadete vastuvõtmisel võib kasutada ka metamärke (`MPI_ANY_TAG` ja `MPI_ANY_SOURCE`), siis võib tekkida olukord, kus tupikud programmi käivitamisel tekivad vaid aeg-ajalt, sõltuvalt asjaolust, millal mingi protsess missuguse teate saadab. Selliseid olukordi nimetatakse ajastusest sõltuvateks tupikuteks (*timing-dependent deadlock*) [12].

Peatükk 2

Profileerimine ja jõudlusanalüüs

2.1 Profileerimine

Paralleelprogrammid oma ülesehituselt on keerukamad kui tavaprogrammid ning nende puhul on jõudlus üheks väga oluliseks aspektiks. Kui programm ei skaleeru ning üha suuremal arvul protsessoritel joostes kulub programmi tööks üha enam aega, siis kaotab paralleelsus oma eesmärgi ning tavaprogramm võiks sama töö ära teha kiiremini. Lisaks on paralleelprogrammidest ka raskem leida vigade põhjustajaid ning jõudluse kitsaskohti. Vigade leidmiseks on programmeerijatele abiks loodud mitmeid silumistööriistu ning jõudlusprobleemide lahendamiseks profileerijaid. Käesolevas töös vaadeldakse MPI programmide suhtlusosa profileerijaid ning keskendutakse protsesside omavahelise kommunikatsiooni jälgimist võimaldavatele vahenditele.

Profileerimine (*profiling*) ehk profiili koostamine programmide puhul on jõudlusandmete ja ressursikasutusinfo kogumine tervikuna kogu programmi või programmiosa kohta ning salvestatavate sündmuste ajaline järjestus ei selles ei kajastu. Programmide jälgimine (*tracing*) on kogutava info ajatempliga märgistamine ning kronoloogilise järjestuse loomine ning võimaldab kasutajal analüüsida andmeid toimunu taustal. Programmide profiilide koostajaid ehk profileerimisprogramme nimetatakse profileerijateks. On olemas erinevate ressursside profileerimiseks loodud profileerijaid ja jälgijaid, näiteks mälu või protsessori toimingute uurimiseks programmi töö käigus. Käesolev töö nagu eelnevalt mainitud keskendub teateedastusliidese profileerijatele.

Profileerimisprogrammid on aja arenedes muutunud üha mitmekesisemaks ja rohkemaid võimalusi pakkuvateks ning valdav enamus profileerijaid sisaldavad endas ka jälgimisfunktsioone ning pakuvad kasutajatele infot sündmuste ajalise järjestuse kohta, seega termin "profileerimine" on tihti kasutuses ka laiemas mõistes hõlmates ka jälgimist. Selleski töös ei ole jälgimist rangelt profileerimisest eristatud vaid jälgimisinfot on tõlgendatud profileerimisandmete osana.

MPI programmide profileerijad koosnevad üldiselt kahest osast - programmi tööd jälgivast ja infot koguvast teegist ning kasutajaliidesest tulemuste kuvamiseks. Paljud profileerijad suudavad kogutud info inimesele ülevaate andmiseks graafiliselt visualiseerida. Põhiline info, mida kogutakse, on ajakulu. Mõõdetakse, kui palju aega kulus protsessidel erinevate MPI käskude täitmiseks, kui palju aega kulus kommunikatsioonile ja sünkronisatsioonile. Sõltuvalt profileerijast võidakse pakkuda ka muid lisavõimalusi - näiteks edastatavate sõnumite suuruse ülestähendamist. Võidakse kasutada ka lisaliideseid riistvaraloendurite andmete jälgmiseks. Leidub ka jõudlusanalüüsi tööriistapakette, mis koosnevad erinevatest andmekogumisteekidest, visualiseerijatest ja logifailide konverteerijatest erinevatesse formaatidesse ning pakuvad kasutajale laiemat valikut kui ainult suhtluse osa profileerimist.

Enamasti ei pea profileermiseks programmi koodi uuendama ega sinna vastavaid käske sisse lisama või MPI funktsioonide väljakutseid asendama profileeritavate MPI funktsioonide väljakutsetega. Sõltuvalt profileerijast on võimalused erinevad. Osa profileerijaid võimaldavad kasutajal koodis kasutada profileerimise alustamist ja lõpetamist tähistavaid käske. See on hea puhkudel, millal kasutaja ei soovigi kogu programmi analüüsida lasta, vaid ainult ettemääratud lõike sellest. Samas võidakse lasta ka kasutajal defineerida ise profileeritavaid koodiosasid, sest vaikimisi MPI profileerijad töötlevad iga logitavat MPI käsku eraldi osana, kuid oma lõigu defineerimisel on hiljem logifaili analüüsil võimalik saada koondinfot just selle piirkonna kohta.

Enamlevinult seotakse programm peale kompileerimist profileerimisteegi külge ning selle tulemusel saadakse profileerimisinfot genereeriv programm. Siiski, see ei ole ainuke võimalus. On ka profileerijaid, mille puhul ei ole vaja uuritavat koodi eelnevalt profileerimisteegiga siduda, vaid kasutades spetsiaalseid programmeerimisliideseid (näiteks DPCL[29] või DynInst[37]) suu-

davad nad profileerimiseks vajaliku koodi sisestada dünaamiliselt programmi mälu aadressiruumi selle jooksmise ajal või vahetult enne jookmise alustamist. Kolmandaks võimaluseks on näiteks programmi lähtekoodi automaatne ümberkodeerimine vahetamaks MPI funktsioonide väljakutsed profileerija poolt pakutavate jõudlusinfo kogumisega MPI funktsioonide väljakutsete vastu. Kui programm on ühel või teisel moel lülitatud profileerimise režiimi, siis tegeleb programmeerimisteek määratud info kogumisega ning logifailidesse kirjutamisega. Töö lõppedes kasutatakse sobivat vaatlusprogrammi logifailis oleva info analüüsimiseks. Siingi on olemas alternatiiv - osa profileerijatest analüüsib infot dünaamiliselt ehk siis programmi töötamise ajal.

Profileerijate eesmärk on leida programmi kitsaskohti, kuhu kulub palju aega ning mille taha protsessid ootama võivad jääda. Paralleelselt jooksvate protsesside arvu kasvades võib kitsaskohti lisanduda, seega tihti ei paista need vähese arvu protsesside korral välja. Ühtlasi mõõdavad nad paralleeliseerimise üldkulusid (kas ikka tõesti tagab paralleelsus optimaalsema ressursijaotuse või tekitab hoopis lisaprobleeme ja takistusi). Profileerijad aitavad ka hinnata tulu erinevate koodiosade optimeerimisest. Näiteks mõni koht, kus oleks võimalik algoritmi muutmise korral vähendada ajakulu, ei pruugigi tekitada erilisi probleeme ka siis, kui paralleelselt jooksvate protsesside arv kasvab, kuid mõni teine samalaadne probleem seevastu ainult võimendub protsesside lisandudes.

Profileerimisel võib esineda kõrvalnähtusid - mida suuremat hulka andmeid koguda, seda kauem selleks aega läheb ning seda enam saavad mõjutatud programmi erinevate etappide ajakulu mõõtmistulemused.

Erinevad profileerijad kasutavad erinevaid profileerimistaktikaid. Üheks valikuks on näiteks lamedate (*flat profiles*) või täisprofiilide (*full profiles*) koostamine. Lamedad profiilid sisaldavad infot ainult mingil ajahetkel süsteemi funktsioonide väljakutsemagasinis (*call stack*) kõige pealmisel tasemel oleva funktsiooni kohta, arvestamata funktsioonide omavahelisi suhteid. Täisprofiilide korral salvestatakse info vastavusega väljakutsemagasinis olevatele andmetele. Väljakutseteekonna profiilid (*callpath profiles*) mõõdavad ajakulu iga väljakutsemagasini täissisule (funktsioonijärjekorrale) vastavalt, mitte ainult pealmise taseme funktsiooni arvestades. Väljakutseasukoha profiliid (*callsite profiles*) on veelgi täpsemad, eristades kohti programmikoodis, kust

vastav funktsioon välja kutsuti [38].

Eelneva selgitamiseks näite varal on aluseks võetud alljärgnev programmikood [38].

```
void B() {
   sleep(1);
}
void A() {
   sleep(1);
   B();
}
int main() {
   A();
   sleep(1);
   B();
   B();
   return 0;
}
```

Kui seda programmi profileerida lamedalt siis oleks tulemuseks tulemuseks kolm summaarset ajakulu: funktsioonile A, funktsioonile B ja funktsioonile *main*. Iga kord kui mingit funktsiooni välja kutsutakse, lisatakse selle ajakulu sama funktsiooni juba olemasolevale ajakuluarvestile. Kui selle programmi kohta koostada väljakutseteekonna profiil, oleks tulemuseks neli erinevat summaarset ajakulu: *main*, *main-A*, *main-A-B*, *main-B* ehk siis vastavalt kõikidele võimalikele väljakutsemagasini olukordadele selle programmi kasutamisel. Väljakutseasukoha profiili korral lisanduks eelnevale veel üks *main-B*, kusjuures need kaks erineksid väljakutseasukoha poolest, seega tulemus oleks: *main* ilma väljakutseasukohata, *main-A* väljakutseasukohaga *main*, *main-A-B* väljakutseasukohaga *A*, *main-B* väljakutseasukohaga *main-esimene* ja *main-B* väljakutseasukohaga *main-teine*. [38].

Teiseks võimaluseks on valida kaasa arvava (*inclusive*) või välja arvava (*exclusive*) ajamõõtmise vahel. Esimese korral loetakse funktsiooni A tööaja hulka kogu funktsiooni täitmisele kuluv aeg, kaasa arvatud sealt seest

väljakutsutavate funktsioonide täitmiseks kulunud aeg, teise korral jäetakse muude A seest väljakutsutavate funktsioonide ajakulu A tööaja sisse arvestamata [38].

2.2 Lühiülevaade MPI programmide profileerijatest

Nagu eelnevalt kirjeldatud on olemas erinevaid profileerimistehnikaid ning loodud on mitmeid profileerimistarkvarasid. Kõige lihtsam ning vähem funktsionaalsust omav MPI profileerija on MpiP [43], millest räägib pikemalt peatükk 3.1. MpiP on väljakutseasukoha-põhine profileerija, mis ei vaja programmi taaskompileerimist, vaid ainult vastava teegi külge sidumist. MpiP näol on tegu lihtsa vabavaralise avatud lähtekoodiga lahendusena teateedastusliidese töö kohta info kogumiseks ning ka logifailid, kuhu tulemus kirjutatakse, ei vaja eraldi tõlgendusprogrammi, vaid on tavalisel teksti kujul. MpiP ei sisalda jälgimisvahendeid.

TAU *Tuning and Analysis Utilities*) [30] seevastu on üks mahukamaid saadaval olevaid vabavaralisi profileerimislahendusi, mille jaoks teateedastusliidese profileerimine on vaid üks alamosadest. TAU toetab PAPI (*Performance Application Programming Interface*) teeki ning suudab seetõttu genereerida profiile ka põhinedes PAPI poolt pakutavatele riistvaraloenduritele. TAU sisaldab ka mälu profileerimise vahendeid. TAU kasutamiseks on mitmeid võimalusi. Üheks neist on programmi lähtekoodi modifitseerimine, lisades sinna profileerimist võimaldavad käsud. Selle automaatseks sooritamiseks sisaldab TAU profileerimispakett spetsiaalseid tööriistu, kuid ainult C++ keele jaoks [23]. Teiste programmide puhul võib kasutada TAU dünaamilisi profileerimivõtteid, mis põhinevad DynInst [37] programmeerimisliidesel, mille abil suudab TAU profileerimiseks vajaliku koodi sisestada programmi mälu aadressiruumi selle tööajal. TAU logifailide analüüsiks on paketis vaatlusprogramm Racy, kuid sellega on võimalik analüüsida ka MpiP poolt koostatavaid faile.

Paradyn [28] on mõeldud jõudluse mõõtmiseks paralleel- ja hajusprogrammidel. Paradyn töötab dünaamilise koodisisestuse põhimõttel, seega ka-

sutajad ei pea oma programmi uuesti kompileerima ega vastava teegi külge siduma. Ka kogutud infot analüüsib Paradyn programmi töö ajal, mis tähendab tulemuste kuvamist reaalajas. Otseselt ei ole Paradyn mõeldud teateedastusliidese profileerimiseks, kuid võimaldab seda siiski vähesel määral teha, piirdudes MPI realisatsioonidest MPICH1 ning AIX operatsioonisüsteemi POE MPI toetamisega [23]. Paradyn on vabavara.

Kommertsiaalsetest profileerijatest/jälgijatest rääkides on Vampir Trace Analyzer üks tuntumaid. Nüüdseks on Vampir läbinud nimemuutuse ning sellest on saanud Intel®Trace Analyzer and Collector [17]. Oma rohkete andmete visuaalsete kuvamisviiside rohkuse tõttu võimaldab Intel®Trace Analyzer and Collector analüüsida andmeid väga mitmekesiselt ja tulemulikult. Intel®Trace Analyzer and Collector hõlmab endas ka MPI-2 standardis kirjeldatud sisend/väljund operatsioonide logimist ja teeki MPI korrektsuse kontrolliks. [17].

Suhteliselt uus kuid perspektiivikas tulija turul on Alline Software poolt loodud Allinea OPT (*Optimizing and Profiling Tool)*. OPT pakub kommertsiaaltarkvarale omast mugavust logifailide kuvamisel pakkudes laialdasi võimalusi andmete visuaalseks analüüsiks, osutades problemaatilistele kohtadele. Allinea OPT programmile on pühendatud. Allinea OPT tarkvara tutvustatakse pikemalt peatükis 3.2.

MPE ehk *MPI Parallel Environment* [7] ja sellega kaasnev logifailide visualiseerija Jumpshot-4 [6] on vast ehk kõige mugavamini paigaldatav ja kasutatav profileerimisvahend. Vaikimisi tuleb see kaasa MPICH1 ja MPICH2 MPI realisatsioonidega, kuid toetab ka laialdasel hulgal teisi MPI realisatsioone. Jumpshot-4 omadustest saab lugeda peatükist 3.3 ning sellega on viidud läbi ka 5. peatükis kirjeldatud analüüs tarkvarapaketil DOUG.

Peatükk 3

Põhjalikum ülevaade mõningatest MPI profileerijatest

3.1 MpiP

MpiP on lihtne vabavaraline ja avatud lähtekoodiga profileerimisteek MPI programmide jaoks. Erinevalt paljudest teistest profileerijatest kogub MpiP statistilisi andmeid ainult MPI funktsioonide kohta, mistõttu on kogutud andmehulk ning lisakulu resurssidele MpiP rakendamise korral väiksemad [43]. MpiP on loodud California ülikoolis Lawrence Livermore riiklikus laboratooriumis. MpiP kasutamiseks tuleb programm linkida MpiP profileerimisteegi *libmpip* külge. Seejärel tuleb programm käivitada nagu tavaliselt. Programmi töö lõppedes kuvatakse standardväljundisse andmed *.mpiP faili kohta, kuhu kogutud info salvestati. Kui programm on kompileeritud koos info genereerimisega siluri jaoks (tavaliselt kompilaatori võti *-g*), siis MpiP suudab automaatselt kuvada ka koodifaili nime ja reanumbri, kust vastav väljakutse MPI funktsioonile tuli.

MpiP eeliseks on asjaolu, et tekitatavad logifailid on tavalised tekstifailid ega vaja mingit spetsiifilist programmi nende vaatamiseks. Statistiliste analüüside tegemiseks on nõnda sealt hõlbus andmeid kätte saada. Teisest küljest ei paku teksti kujul olevad jõudlusandmed kõige paremat ülevaadet ning inimesele analüüsimiseks on mugavam kasutada neid profileerijaid, mille salvestatud infot on mõni visualiseerija suuteline kuvama (MpiP logifailide

vaatamiseks on võimalik kasutada TAU [30] visualiseerijat).

Profileerija salvestab ajakulu kasutatud MPI käskudele, teateedastusega seotud käskude kohta salvestatakse ka sõnumi suurus baitides ning faili lugemise ja kirjutamisega seotud MPI käskude korral samuti andmete suurus baitides. MpiP ei kogu informatsiooni kohalike MPI käskude kohta (näiteks `MPI_Comm_Size`, mis tagastab protsesside arvu kogu süsteemis).

```
---------------------------------------------------------------------------
@--- MPI Time (seconds) ------------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0         53    3.5e-05     0.00
   *         53    3.5e-05     0.00
---------------------------------------------------------------------------
@--- Callsites: 5 ------------------------------------------------------------
---------------------------------------------------------------------------
 ID Lev File/Address        Line Parent_Funct                              MPI_Call
  1   0 Mesh.F90            1453 __mesh_class__mesh_paramsmpibcast         Bcast
  2   0 DOUG_utils.F90      1407 __doug_utils__sharedctrldata_mpitypecreate Type_commit
  3   0 main.F90             232 MAIN__                                    Reduce
  4   0 DOUG_utils.F90       437 __doug_utils__util_finalizempi            Barrier
  5   0 DOUG_utils.F90      1422 __doug_utils__sharedctrldata_bcast        Bcast
```

Joonis 3.1: MpiP poolt kogutav üldinfo

Joonis 3.1 demonstreerib profileerija poolt kogutavaid üldiseid andmeid. *Task* on töö unikaalne nimi programmis (antud näites on ainult üks töö), tärniga tähistakse kogu programmi (kõiki töid summaarselt). *AppTime* lahtrisse on märgitud aeg (sekundites), mis kulus protsessil oma töö tegemiseks käsu `MPI_Init` lõpetamisest kuni käsu `MPI_Finalize` alguseni, *MPITime* on ajakulu selles vahemikus kasutatavatele MPI funktsioonidele. *MPI%* on *MPITime* ja *AppTime* suhe.

Callsite lõik ehk funktsiooni väljakutseasukoha lõik seab igale kasutatavale MPi funktsioonile vastavusse ID-numbri. Kui siluri info on saadaval (programm on kompileeritud koos suvandiga *-g*), siis on näha ka funktsiooni nimi, kust vastav MPI käsk välja kutsuti ning ka vastava faili nimi ning reanumber.

Joonis 3.2 illustreerib *Mpip* poolt kuvatavaid koondandmeid ajakulu ja sõnumi suuruse poolest 20 juhtiva MPI käsu väljakutse kohta. *Time* tähistab summarset ajakulu (millisekundites) vastava MPI käsu (*Call* täitmisele kogu programmi töö jooksul, mis on välja kutsutud kohast, mille ID-number on märgitud tulbas *Site*. *App%* ja *MPI%* tähistavad vastava tehte summaarse ajakulu suhet kogu programmi ajakulusse ja kogu programmi MPI käskude

```
---------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------
---------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Reduce                  3      0.018    0.00   51.43    0.00
Barrier                 4      0.006    0.00   17.14    0.00
Bcast                   5      0.005    0.00   14.29    0.00
Type_commit             2      0.003    0.00    8.57    0.00
Bcast                   1      0.003    0.00    8.57    0.00
---------------------------------------------------------------------
@--- Aggregate Sent Message Size (top twenty, descending, bytes) ----
---------------------------------------------------------------------
Call                 Site      Count      Total       Avrg  Sent%
Bcast                   5          1        116        116  80.56
Bcast                   1          1         20         20  13.89
Reduce                  3          1          8          8   5.56
```

Joonis 3.2: MpiP poolt kogutavad koondandmed

täitmise ajakulusse. Viimane näitaja, *COV* on antud tehte ajakulu variatsioonikordaja, mis on arvutatud töös osalenud individuaalsete protsesside vastava tehte ajakulu põhjal. Mida suurem on variatsioonikordaja (tunnuse standardhälbe ja keskväärtuse suhe), seda rohkem erinevad protsesside ajakulud sellele tehtele. Joonisel 3.2 antud näites on kasutusel vaid üks protsess, mistõttu variatsioonikordaja on 0.

Lisaks koondandmetele on välja toodud ka iga protsessi iga tehte (funktsiooni väljakutse koos konkreetse väljakutseasukohaga) ajakulu ning saadetud sõnumite hulga ja suuruse kohta käiv statistika (joonis 3.3). Kui programm kasutab ka MPI failitöötlemise käske, siis lisandub sellele veel failist loetava / faili kirjutatava andmehulga statistika, mis kasutab samu näitajaid, kui on kuvatud saadetud sõnumite kohta käivas tabelis.

MpiP failide lugemise hõlbustamiseks on ka olemas spetsiaalne programm nimega ToolGear Viewer [20], mis lisaks teistelegi formaatidele toetab *.mpiP failiformaati. ToolGearViewer ei visualiseeri *.mpiP failide andmeid, vaid võimaldab selekteerida kuvatavat infot ning kuvada sama akna alumises osas vastavat osa programmi koodis (kui koodi asukoht on määratud).

```
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 5 ----------------------
---------------------------------------------------------------------------
Name                 Site Rank  Count      Max     Mean      Min   App%   MPI%
Barrier                 4    0      1    0.006    0.006    0.006   0.00  17.14
Barrier                 4    *      1    0.006    0.006    0.006   0.00  17.14

Bcast                   1    0      1    0.003    0.003    0.003   0.00   8.57
Bcast                   1    *      1    0.003    0.003    0.003   0.00   8.57

Bcast                   5    0      1    0.005    0.005    0.005   0.00  14.29
Bcast                   5    *      1    0.005    0.005    0.005   0.00  14.29

Reduce                  3    0      1    0.018    0.018    0.018   0.00  51.43
Reduce                  3    *      1    0.018    0.018    0.018   0.00  51.43

Type_commit             2    0      1    0.003    0.003    0.003   0.00   8.57
Type_commit             2    *      1    0.003    0.003    0.003   0.00   8.57
---------------------------------------------------------------------------
@--- Callsite Message Sent statistics (all, sent bytes) -------------------
---------------------------------------------------------------------------
Name                 Site Rank   Count       Max      Mean       Min       Sum
Bcast                   1    0       1        20        20        20        20
Bcast                   1    *       1        20        20        20        20

Bcast                   5    0       1       116       116       116       116
Bcast                   5    *       1       116       116       116       116

Reduce                  3    0       1         8         8         8         8
Reduce                  3    *       1         8         8         8         8
```

Joonis 3.3: MpiP poolt kogutavate protsesside ajakulu tehte kohta

3.2 Allinea OPT

Allinea OPT (Optimization and Profiling Tool) on Suurbritannia ettevõtte Allinea Software poolt loodud kommertsiaalne MPI programmmide profileerija. Allinea OPT on turul suhteliselt uus produkt, ametlikult välja lastud aastal 2006, mistõttu ei ole see pikemat kajastust leidnud varasemates paralleelprogrammide profileerimise ja optimeerimise vahendeid kirjeldavates või võrdlevates uurimistöödes ja ettekannetes [31], [8], [22], [23], mis on ka põhjuseks, miks käesoleva töö autor on selle valinud üheks tutvustavaks näiteks MPI programmide profileerijate hulgast.

Lisaks MPI kommunikatsiooni ajakulu ning sõnumi suuruste jälgmisele võimaldab OPT ka protsessori ja PAPI (*Performance Application Programming Interface*) [27] programmeerimisteegi abil ligipääsetavate riistvaraloendurite kasutamist, seda viimast küll vaid juhul kui Linux operatsioonisüsteemis, kus OPT parasjagu jookseb, on tuumas laetud moodul *perfctr*. Tänu

riistvaraloenduritele on lisaks kommunikatsioonile võimalik jälgida ka potsessorite tegevusi või tegevusetust. Programmi optiseerimise aspektist on see väga kasulik info.

Allinea OPT programmi eeliseks on laiad valikuvõimalused informatsiooni visuaalseks kuvamiseks. Lisaks enamlevinud visualiseeritud aegjoonele (*timeline*) on võimalik kasutada ka histogrammi, joon-, tulp- ja sektordiagrammi, väljakutsete graafi ning teateedastustabelit. Allinea OPT ei kuva kasutajale informatsiooni reaalajas ning jõudlusanalüüs saab toimuda vaid peale programmi töö lõpetamist.

OPT koosneb kolmest komponendist, millest igaüks võib joosta erineval riistvaral [1]:

1. profileerimisteek, mille külge tuleb siduda uuritav rakendus ning mis tegeleb jõudlusinfo kogumisega;

2. andmebaas/server, kuhu kogutud info salvestatakse;

3. kasutajaliides, mis kuvab kogutud andmed kasutajale analüüsimiseks.

Jõudlusanalüüsi sooritamiseks tuleb kasutajaliideses (*optgui*) luua uus töö. Töö loomisel tuleb määrata töö nimi, käivitatav fail (analüüsitav programm, mis on juba seotud profileerimisteegi külge), MPI protsesside arv ning vajadusel ka käsurea parameetrid. Lisavõimalustena saab valida, milliste MPI funktsioonide kohta infot koguda ning millised protsessori ressursikasutuse loendureid kasutada. MPI funktsioonide valiku korral pakutakse kasutajale nimekirja kõigist olemasolevatest MPI käskudest (va staatiliselt seotud teeke kasutavate programmide korral, kui ette on antud nimekiri ainult programmis kasutusel olevatest MPI käskudest). Protsessori loenduritest saab valida sobivaid alljärgnevast hulgast.

- Mälu leheküljetõrked (*page faults*) - virtuaalmälu katkestuste arv, mis viitavad saalimisvajadusele kõvakettalt, kuna järgneva käsku jaoks vaja minev andmekogum ei asu mälus.

- Mälu ümberjaotamise järjekorras viibivate lehekülgede poole pöördumiste arv (*page reclaims*).

- Kohustuslikud kontekstivahetused (*involuntary context switches*) - kordade arv, millal protsess on olnud sunnitud loovutama protsessori kasutusjärje järgmisele protsessile. Tavaliselt juhtub see, kui protsessile eraldatud protsessoriaeg saab läbi või tööjärjekorda on saabunud mõni kõrgema prioriteediga protsess. Protsess jääb ootele, millal taaskord protsessori kasutamise kord temani jõuab.

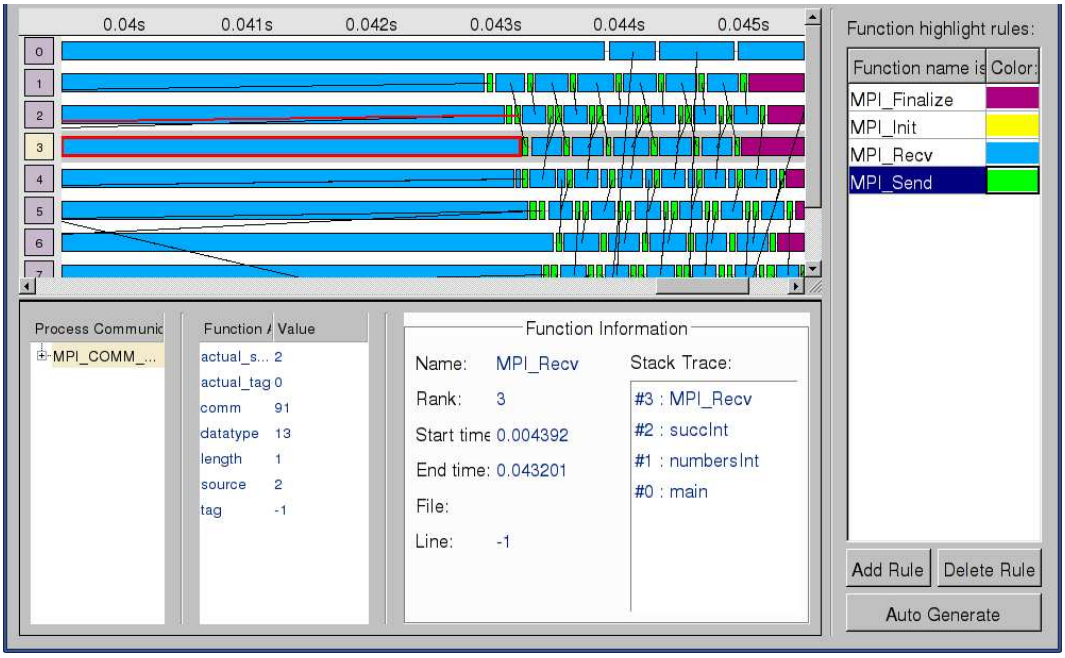
- Vabatahtlikud kontekstivahetused (*voluntary context switches*) - kordade arv, millal protsess on vabatahtlikult talle eraldatud protsessori kasutusaja jooksul järjekorra käest loovutanud järgmise protsessi kasuks. Selline olukord tekib näiteks juhul, kui protsessil on tarvis kasutada sisend- või väljundoperatsioone.

- Kulunud süsteemiaeg (*system time used*) - aeg, mis kulus protsessoril operatsioonisüsteemi käskude täitmiseks, mida oli tarvis jooksutamaks kasutaja programmi.

- Kulunud kasutajaaeg (*user time used*) - aeg, mis kulus protsessoril kasutaja poolt kirjutatud (käivitatud programmi) koodi täitmiseks. Tasub tähele panna, et süsteemiaeg ja kasutajaaeg on üksteist välistavad - st sama tegevuse ajakulu ei saa minna korraga kirja nii süsteemiaja kui ka kasutajaaja alla.

- Saalimiste arv (*swaps*) - näitab mitu korda programmi tööajal protsess mälust välja saaliti.

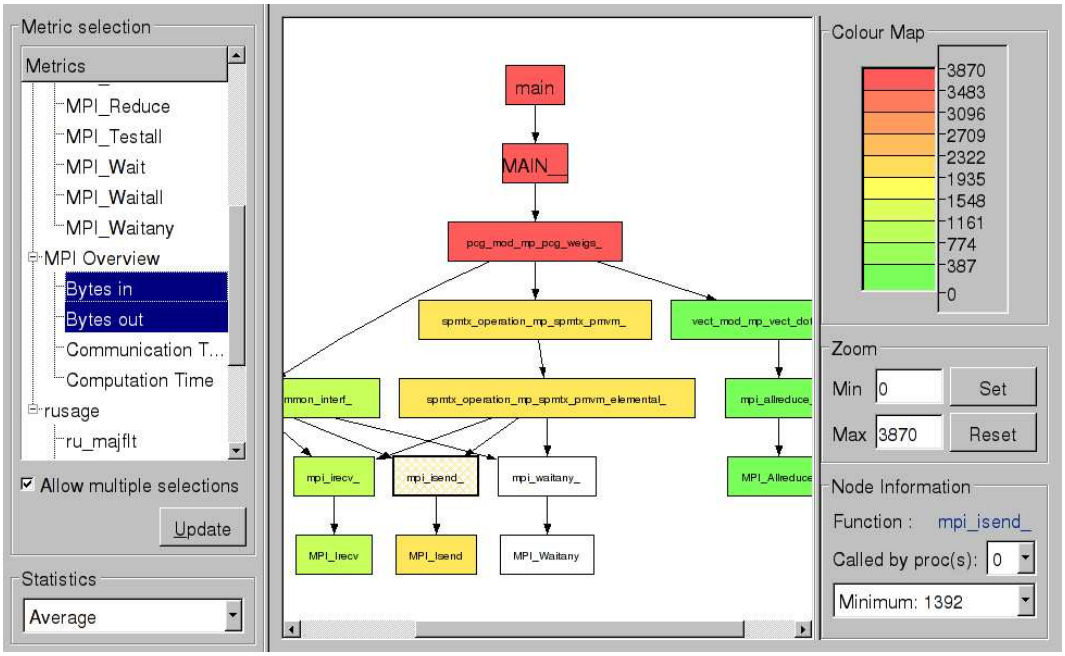
Kui programm on oma töö lõpetanud, on kasutajal võimalik vaadata ja analüüsida kogutud jõudlusandmeid. Esimene avanev vaade on aegjoon (joonis 3.4). Aegjoonelt on hästi näha protsesside omavaheline töökoormuse jaotus. Värvilised kastid tähistavad erinevaid MPI funktsioone. Eri protsesside MPI funktsioone ühendavad jooned tähistavad saadetavaid sõnumeid. Aegjoon võimaldab hõlpsalt tuvastada sõnumi saatmisega hiljaksjäämise probleemi. Joonisel 3.4 on näide sõnumi hilinemise probleemist - `MPI_Recv` kastid on ajajoonel suhteliselt pikad ning täiendavad jooned, mis tähistavad saadetavaid sõnumeid erinevate protsesside vahel näitavad, et sõnumeid hakatakse ootama juba tunduvalt varem, kui neid teele saadetakse. Lisainfona



Joonis 3.4: OPT aegjoon

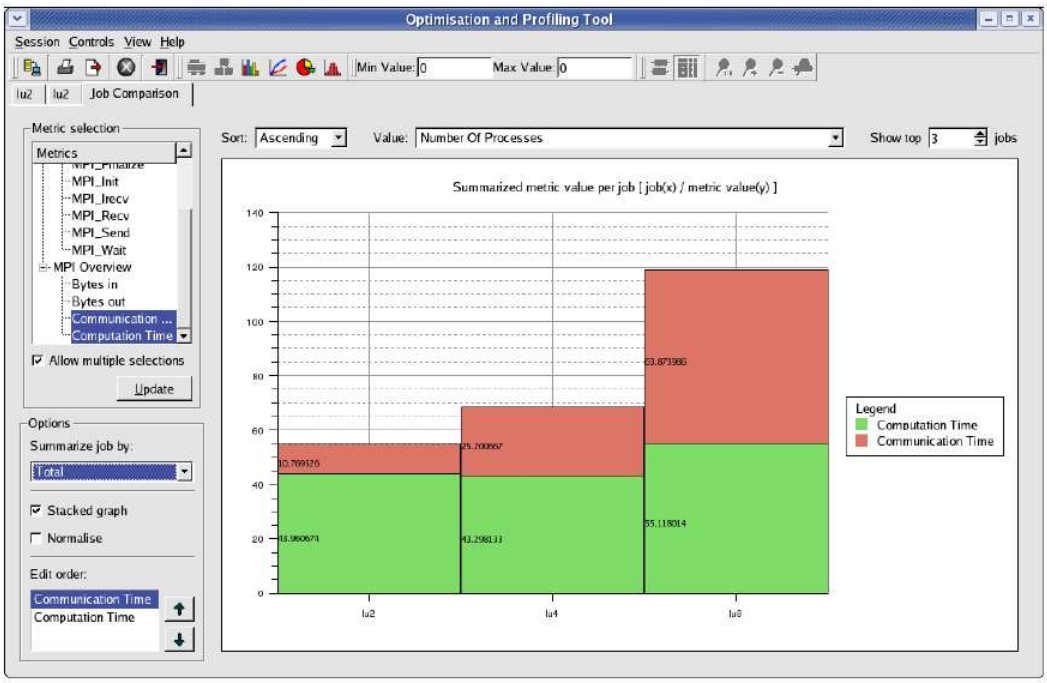
kuvatakse akna alumises osas informatsiooni vastava ajajoonelt valitud MPI funktsiooni kohta. Kasutaja saab ise valida värve erinevate MPI funktsioonide tähistamiseks, mis on väga hea, sest nõnda on võimalik jätta enamus MPI käske värvituks ning värvida vaid otsitavad, saavutades parema ülevaatlikuse. Ajajoonelt saab veel ka tuvastada üleliiase suhtluse probleemi. Kui on märgata, et joonisel on palju väikeseid üksteisele vastavaid saatmisi ja vastuvõtmisi sümboliseerivaid kaste, mis tähistavad paljude väikesete järjestike sõnumite edastamist, siis võiks kaaluda mitme lühikese sõnumi kombineerimist üheks pikemaks ja saata korraga üks mahukam teade mitme järjestikuse väikese teate asemel [1].

Väljakutsete graaf (*call graph*) on otseselt seotud aegjoonega, kajastades protsessorite magasinis olevaid funktsioone hetkel, mil asuti täitma mõnda jälgitavat MPI käsku aegjoonel nähtaval olevast ajavahemikust. Ekraani vasakust servast saab valida sobivaid mõõdikuid ja soovitavaid statistilisi andmeid (miinimum, maksimum, keskmine, summa, dispersioon), vastavalt mille väärtustele toimub funktsioonide automaatne värvimine (joonis 3.5). Valitud



Joonis 3.5: OPT väljakutsete graaf

mõõdikud mõõdavad tulemusi ainult MPI funktsioonidel, muude funktsioonide puhul sumeeritakse kõikide protsesside nende MPI funktsioonide tulemused, mis on välja kutsutud selle funktsiooni kaudu. Väljakutsete graafi saab hästi kasutada programmi kitsaskohtade leidmiseks, näiteks otsides erinevate näitude kohal jooniselt kõige punasemaid piirkondi. Üheks strateegiaks on võrrelda protsesside keskmist suhtlusaega (*communication time*) ehk siis aega, mis kulub MPI käskude täitmiseks, protsesside ja sissetulevate ning väljasaadetavate keskmise baitide hulgaga. Näiteks, kui jooniselt on näha, et `MPI_Recv` käsk kasutab palju suhtlusaega, ent läbiva baitide hulga järgi väljakutsete graafi vaadates ei olegi see funktsioon kõige "kuumem piirkond", siis võib taaskord tegu olla sõnumite hiljaks jäämise probleemiga - kulutatakse palju aega ootamisele. Samuti tasub punaste piirkondade korral vaadata ekraani paremas all nurgas pakutavat detailsemat infot - võrrelda erinevate protsesside samale funktsioonile sama mõõdiku alusel kogutud statistikat. Näiteks suhtlusaja võrdlemisel võiks vaadata, kas samale funktsioonile on erinevad protsessid kulutanud minimaalselt, maksimaalselt, keskmiselt ja sum-



Joonis 3.6: OPT tulpdiagramm [2]

maarselt enam-vähem sama palju aega. Kui erinevused erinevate protsesside vahel on suured, siis võib olla probleeme protsessidevahelises tööjaotuses [1].

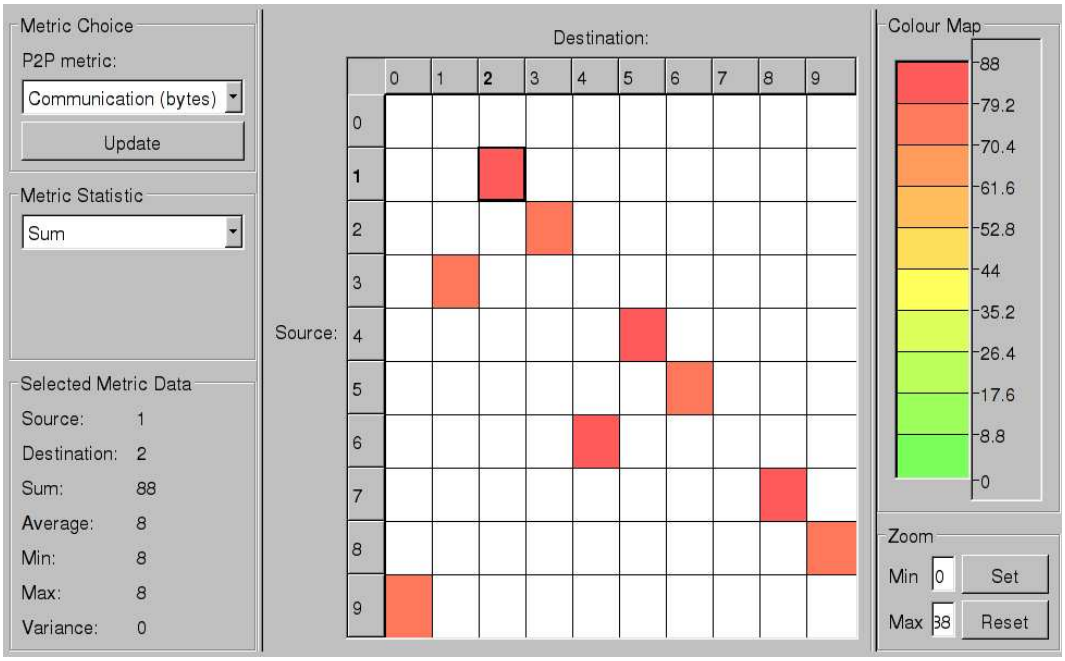
Tulp-, joon- ja sektordiagramm on samuti seotud aegjoonega ning kajastavad tulemusi vaid aegjoonel nähtava perioodi kohta va juhul, kui tegu on erinevate tööde võrdlemisega. Nimelt on Allinea OPTi üheks suureks eeliseks kahe või enama töö omavahelise võrdlemise võimalus. Näiteks võib sama programmi käivitada erineval arvul protsessoritel ning seejärel tulemusi omavahel võrrelda, märkamaks skaleeruvusprobleeme. Tööde võrdlemisel ei ole võimalik kasutada aegjoont, väljakutsete graafi ning teateedastustabelit.

Tulpdiagramm võimaldab vaadata statistikat protsessipõhiselt ja võrrelda seda teiste protsessidega. Erinevate tööde võrdlemise korral on tulpdiagramm eriti hea skaleeruvusprobleemide tuvastamiseks. Joonisel 3.6 võrreldakse programmi suhtlus- ja arvutusaega (*computation time*) 2, 4 ja 8 protsessi korral (arvutusaeg on aeg, mis kulus programmi täitmisele MPI käskude vahel). Jooniselt on näha, et tegu on halvasti skaleeruva programmiga, kuna

protsesside arvu kasvamisel neljalt kaheksale on suhtlemisele kuluv aeg kasvanud rohkem kui kaks korda.

Sagedustabel näitab protsesside lõikes jagunemist valitud mõõdiku erinevate väärtuste vahel. Sagedustabelist paistavad hästi silma üksikud protsessid, kes teistest mingite näitude poolest erinevad.

Teateedastustabel (*message profile*) annab kasutajale ülevaate protsesside omavahelisest üks-ühele suhtlusest nii ajakulus kui ka baitides. Ka teateedastustabel kajastab olukorda vaid aegjoonelt valitud perioodil. Teateedastustabelit illustreerib joonis 3.7.



Joonis 3.7: OPT teateedastustabel

Kuigi Allinea OPT on tasuline programm ning sellele pakutakse aktiivset tuge, on selle installeerimis- ja kasutusjuhendid käesoleva töö autori arvates kohati veidi puudulikud ning võivad kasutajatele, kellel pole laiemaid süsteemiadministreerimisalaseid teadmisi osutuda ebapiisavateks. Samas jõudlusanalüüsi meetodi ja vahendid, mida OPT pakub, on võrreldes teiste MPI profileerijatega ühed kasutajasõbralikumatest ja mugavamatest.

3.3 MPE ja Jumpshot-4

MPE ehk *MPI Parallel Environment* on tarkvarapakett, mis pakub mitmeid kasulikke tööriistu MPI programmide kirjutajatele. MPE on originaalselt kirjutatud MPICH jaoks ning see vaikimisi lisatud ka MPICH ja MPICH2 MPI realisatsioonidesse, kuid MPE toetab ametlikult ka teisi standardseid MPI realisatsioone ning seda on võimalik süsteemi paigaldada ka eraldiseisvalt. MPE on samuti nagu MPICH ja MPICH2 välja arendatud Mississippi State University Argonne National Laboratory poolt. Käesolev peatükk keskendub MPE profileerimisteegi poolt genereeritava logifaili analüüsija Jumpshot-4 tutvustamisele. MPE paketis on läbi aegade kasutatud erinevaid logifailide formaate. Sündmuspõhine logifailiformaat (*event-based logging format*) kirjeldab kõike programmi käivitamise ajal toimuvat sündmuste ahelana, kus iga tegevus on eraldi sündmus ning igal sündmusel on üks ajatempel. Algupärane sündmustepõhine logifaili formaat on *ALOG*, millele vaatamisprogrammideks on Upshot ja Nupshot. *BLOG* oli selle edasiarendus ning *CLOG* on *ALOG* ja *BLOG* järeltulija ning selle kuvamiseks on programm Jumpshot-2. Olekupõhises logiformaadis (*state-based logging format*) koosneb olek kahest ajatemplist - alguse ja lõpuaja tähisest. Olek on kirjeldatud alati kahe sündmuse poolt: algussündmuse ja lõpusündmuse poolt. Oleku sees võib toimuda ka muid vahepealseid sündmusi. SLOG on olekupõhine logiformaat, mis tähistab skaleeruvat logifaili formaati. *SLOG* formaat loodi adresseerimaks suurte logifailide andmete skaleeruvuse probleemi visualiseerimisel. *SLOG-1* failiformaadi visualiseerija on Jumpshot-3. *SLOG-2* on omakorda edasiarendus *SLOG-1* formaadist ning see on juba joonistatavate elementide põhine failiformaat (*drawable-based logging format*), mis kirjeldab graafilisi objekte. *SLOG-2* failide kuvamiseks on loodud Jumpshot-4. MPE sisaldab ka logifailide konverteerijaid, seega on võimalik tulemusi ühest formaadist teise konverteerida [7].

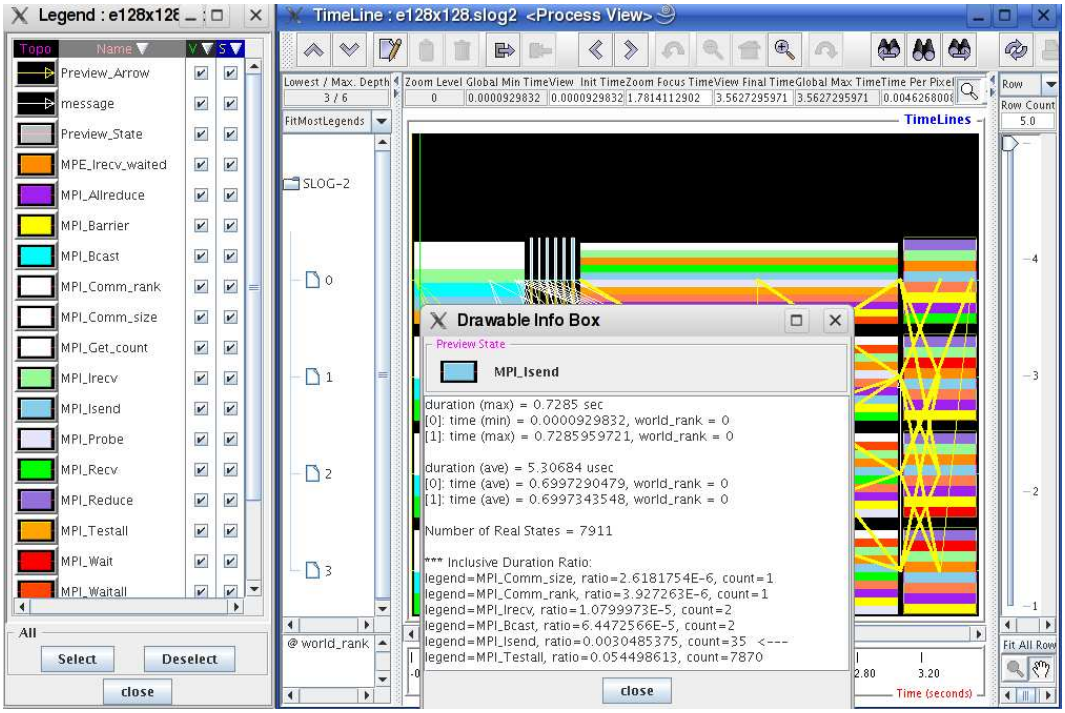
Profileeritav programm seotakse nagu eelnevaltki tutvustatud programmide Allinea OPT ning MpiP puhul kompileerimisjärgselt profileerimisteegi külge ning saadakse seejärel profileerimisandmeid genereeriv versioon programmist. Peale viimase jooksutamist kirjutatakse tulemused vaikimisi *CLOG* logifaili. Selleks, et tulemusi Jumpshot-4 programmiga analüüsida, tuleb *CLOG*

fail kõigepealt konverteerida *SLOG-2* formaati. Selleks võib kasutada eraldi MPE käsku *clog2TOslog2*, aga seda võib ka teha Jumpshot-4 programmi vahendusel konverteerimisaknas.

Jumpshot-4 on *SLOG-2* failiformaadi visualiseerija. Põhivaateks on protsesside aegjooned, mille alamosade kohta on võimalik kuvada MPI funktsioonide sagedustabelit. Tänu erinevate täpsusastmetele, mis on saavutatud eelvaate-jooniselementide (*preview drawables*) kasutusele võtu abil, võimaldab Jumpshot-4 kõrgetasemelist andmete abstraktsiooni ilma suure hulga detailandmete graafilisse kasutajaliidesesse lugemiseta [6].

Jumpshot-4 aegjoonte vaade on suuredamist ja kerimist võimaldav vaade, kus teljestiku mõõtmeteks on protsessi ID-number ja aeg. Ajajoontel kajastatakse *SLOG-2* failist loetavaid jooniselemente (*drawables*). Jooniselemente on kahesuguseid: algelemendid (*primitive drawables*) ja liitelemendid (*composite drawables*). Liitelemendid koosnevad algelementidest ning neid kasutatakse eelvaate-jooniselementidena juhtudel, mil aegjoonte vaadet ei ole piisava tasemeni suurendatud ning kõik algelemendid ei mahu ekraanile ära - siis kuvatakse liitelemente, millel hiirega klõpsates on võimalik saada detailsemat infot selles sisalduvate algelementide kohta. Joonistatavaid algelemente on omakorda kolme laadi: olek, nool ja sündmus. Eelvaate-jooniselemente on samuti kolme sorti, sest liitelement saab koosneda vaid ühte tüüpi algelementidest: eelvaate-olek (*preview-state*), eelvaate-nool (*preview arrow*) ja eelvaate-sündmus (*preview event*). Olek ja nool on joonisel määratud kahe erineva punktiga (protsessi ID-numbri ja ajatempliga), kusjuures noole puhul on algus- ja lõpp-punkti protsessi ID erinev (nooled sümboliseerivad saadetud sõnumeid), olekul aga sama. Sündmus on määratud aegjoonte vaates vaid ühe punktiga. Legendiakna legenditabelis esindatud kõik eksisteerivad olekud, nooled ja sündmused ning see on kui *SLOG-2* failiformaadi joonistatavate elementide visuaalne esitus. Legendiaken on illustreeritud joonisel 3.8. Kasutaja saab valida, kas üldse ja millise värviga joonisel erinevaid jooniselemente kujutada. Iga logitav MPI funktsioon on eraldi olek. Lisaks võimaldab MPE ka kasutajal endal programmikoodis defineerida uusi olekuid ja sündmusi [6].

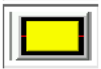
Eelvaate-olekute kuvamiseks pakub Jumpshot-4 mitmeid erinevaid võimalusi. Joonisel 3.8 on näha *FitMostLegends* esitus, mis tähendab, et eelvaate-



Joonis 3.8: Jumpshot-4 legendiaken koos aegjoonte vaatega *FitMostLegends* esitusena ning eelvaate-oleku infoaken

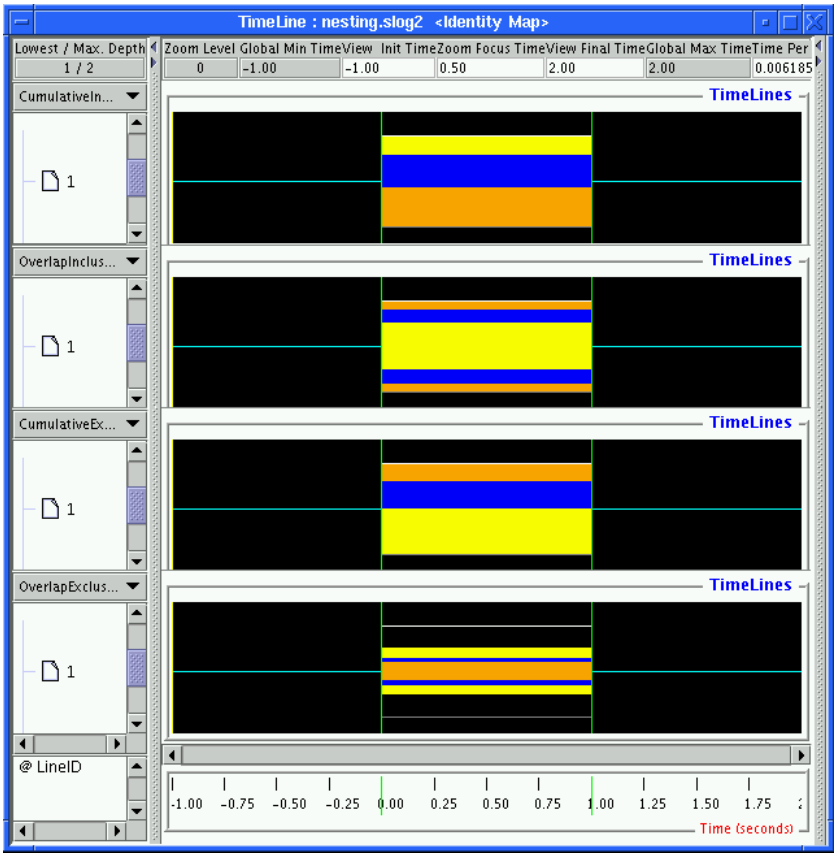
olekutel näidatakse võimalikult palju seal sisalduvaid erinevaid olekuid, seega olekute omavahelised proportsioonid eelvaate-olekus ei mängi selle esituse puhul rolli. Joonisel 3.8 on näha ka eelvaate-oleku infoaken, kus on ära toodud eelvaate-jooniselemendina kujutatud programmilõigu kestvus ajas, selle lõigu kõige esimese oleku algusaeg (minimaalne algusaeg) ning kõige viimase oleku lõpuaeg (maksimaalne lõpuaeg) ja keskmine oleku kestvusaeg. Lisaks on märgitud sisalduvate algolekute arv, algolekute kategooriad ning iga kategooria summarse kestvuse suhe kogu programmilõigu ajakulusse. *SLOG2* logifaili on salvestatud iga oleku kohta eelvaate-olekus nii kaasa arvav suhe (*inclusion ratio*) kui ka välja arvav suhe (*exclusion ratio*) eelvaate-olekusse. Kui profileerija mõõdab olekute ajakulusid, siis on seda võimalik teha kaasa arvavalt, mis tähendab, et kõikide selle oleku sees välja kutsutud teiste olekute ajakulu läheb mõõdetava oleku ajakulu hulka kirja, või välja arvavalt, mis tähendab,

et oleku siseselt teiste väljakutsutavate olekute tööaega ei arvestata [38]. Kui kasutaja on programmi koodi sisestanud mitmeid oma defineeritud olekuid, siis profileerimisel on ta tõenäoliselt huvitatud kogu oleku tööks kuluvast ajast (kaasa arvatud seal sisalduvad muude funktsioonide ja MPI käskude väljakutsed), samas olles huvitatud vaid kommunikatsiooni osa käitumisest ning madalatasemelisest protsesside omavahelisest suhtlusest, võidakse soovida näha vaid välja arvavat ajakulu [6].

Ikoon	Kirjeldus	Kestvus	*Inclusion ratio*	*Exclusion ratio*
	Kõige sisemine olek	0.5 sec	50%	50%
	Vahepealne seesmine olek	0.8 sec	80%	30%
	Kõige välimine olek	1.0 sec	100%	20%

Joonis 3.9: Olekutetabel [6], tabel 2.2

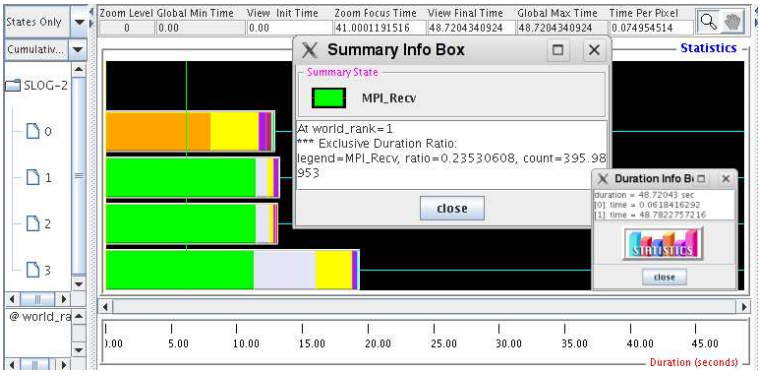
Aegjoonte vaates on võimalik kujutada eelvaate-olekuid nii kumulatiivselt (*cumulative*) kui ka kattuvalt (*overlap*) ja seda saab teha kas välja arvava või kaasa arvava ajakulu suhte kaudu. Kõik olekud eelvaate-olekus võtavad võrreldes aegjoone kõrgusega ruumi proportsionaalselt vastavalt oma summaarse ajakulu suhtele programmilõigu tööaega, kus aegjoone kõrgus sümboliseerib 100% eelvaate-oleku ajakulu ehk kogu vastavale programmilõigule kulunud aega. Näiteks, kui mingi olekute kategooria tegevusaeg moodustas eelvaate-oleku tööajast 80%, siis on vastav olek kuvatud aegjoonel eelvaate-olekus ka 80% ulatuses selle kõrgusest. Erandiks on kumulatiivse kaasa arvava ajakulu suhe (*CumulativeInclusionRatio*), sest kui iga olek arvestab oma tööaja sisse ka kõikide sisemiste olekute tööaja, siis olekute tööaegade summa võib olla suurem kui 1. Näiteks joonisel 3.9 toodud tabelis on *inclusion ratio* summa üle kõigi olekute 230%, siis eelvaate oleku kõrgus sümboliseerib 230% mitte 100% programmilõigu ajakulust. Seetõttu ei saa ka *CumulativeInclusionRatio* esituses omavahel võrrelda erinevaid eelvaate-olekuid komponentide osakaalu suhtes. Joonis 3.10 illustreerib nii kumulatiivseid kui ka kattuvaid eelvaate-olekute esitusi, kus olekute kaasa ja välja arvavad ajakulud programmilõigul on ära toodud joonisel 3.9.



Joonis 3.10: Jumpshot-4 aegjoonte vaated *CumulativeInclusionRatio*, *OverlapInclusionRation*, *CumulativeExclusionRatio* ja *OverlapExclusionRation* esitustena [6], joonis 2.7

Lisaks aegjoonte vaatele pakub Jumpshot-4 võimalust kuvada informatsiooni ka sagedustabelina - kas siis kogu programmi või vaid valitud lõikude ulatuses, kus samuti saab hiireklõpsu abil joonisel lisainfo akna avada. Joonisel 3.11 on illustreeritud ainult olekute sagedustabel, võimalus on lisada ka nooled. Kahjuks on sagedustabeli esitusel Jumpshot-4 visualiseerijal mõningaid puudusi - näiteks soovib kasutaja saada infot sagedustabeli kujul programmi mingi konkreetse etapi kohta, kuid programmi üldvaates (suurendustaseme 0 juures) ei ole eelvaate-olekute tõttu võimalik määrata täpset kohta aegjoonel, kus vastav etapp algab/lõpeb, seega ta kasutab suurendus-

funktsiooni. Suurendades aegjoonte vaadet sobiva tasemeni, kuni on näha algolekud, fikseerib kasutaja sobiva ajahetke aegjoonte vaatel, tehes seda nii etapi alg- kui ka lõpuaja kohta. Nüüd soovib kasutaja märgitud ajavahemiku kohta saada sagedustabelit, kuid selle jaoks on tarvis hiirega märkida ära vastav ala, kuid kuna kasutaja kasutas suurendustasemeid, siis etapi alg- ja lõpphetk ei pruugi mahtuda ära ühele ekraanile. Aegjoonte vaadet kerida samaaegselt hiirega ala märkides ei ole võimalik. Seega ei jää kasutajal muud üle kui vähendada suurendustaset kuni ekraanil on paista nii etapi algust kui ka lõppu tähistavad rohelised vertikaaljooned, mille vahelise piirkonna kohta siis hiirega märgituna saab lasta koostada sagedustabelit. Kui aga alg- või lõpphetke tähistav marker antud suurendustaseme juures mõne eelvaate-oleku keskele, siis kaasatakse sagedustabelisse kõik eelvaate-olekus esinevad algolekud ehk siis ning seetõttu võib tulemuses kajastuda ka neid olekuid, mida tegelikult kasutaja poolt märgitud ajavahemikus ei esine. Kahjuks ei korrigeeri sellisel juhul Jumpshot-4 kasutaja poolt valitud ajahetke algus- ja lõppaega infoaknas *Duration Info Box* ega ka sagedustabeli ajateljel ning seetõttu võib eksitavalt kuvada määratud ajavahemikku seal tegelikult mitteesinevaid olekuid.



Joonis 3.11: Jumpshot-4 histogramm

Üldjoontes on MPE ning selle logifailide visualiseerija Jumpshot-4 hõlpsalt kasutatavad ning väga hea dokumentatsiooni ja kasutajajuhenditega. Täiendav ajakulu MPE rakendamisel programmile on umbes 5%, kuid üha rohkem logitavaid olekuid ja sündmusi defineerides see tõuseb. Erandiks on

`MPI_Barrier` käsu (kasutatakse kõigi protsesside sünkroniseerimiseks käsu logimine, mis on teistest veidi kulukam. Enamustest programmides siiski väga palju `MPI_Barrier` funktsiooni ei kasutata. Miinuseks on puuduv ühendus programmikoodi ja logifaili vahel. Seda probleemi leevendab mõnevõrra võimalus kasutajal endal defineerida olekuid ja sündmusi otse programmikoodis.

Peatükk 4

Tarkvarapakett DOUG

Do*main Decomposition on* ***U****nstructured* ***G****rids* ehk DOUG on kiire paralleelne iteratiivne lineaarvõrrandite lahendaja. Tegu on avatud lähtekoodiga parelleeltarkvara projektiga, mis on arendamisel Tartu Ülikoolis prof. Eero Vainikko juhendamisel koostöös Bathi Ülikooli [36] (Suurbritannia) partneritega (prof. Alastair Spence, prof. Ivan G. Graham, dr. Robert Scheihl, dr. Jan Van lent). DOUG on musta kasti (*black box*) meetodil töötav automaatselt paralleeliseeruv programmiteek, mis kasutab alampiirkondadeks jagamise meetodit suurte lineaarvõrrandisüsteemide lahendamiseks. Järgneva teksti ülesehituse aluseks on võetud [40].

DOUG baseerub järgnevatel tarkvaraliidestel ja -pakettidel:

- **MPI** - DOUG on MPI programm ning paralleelsed tööd kasutavad omavaheliseks kommunikatsiooniks ja sünkronisatsiooniks teateedastusliidest.
- **UMFPACK** - mittesümmeetriliste hõredate lineaarsüsteemide $Ax = b$ otsene multifrontaalne lahendaja [11]. DOUG kasutab UMFPACK lahendajat alampiirkondade alamülesannete lahendamiseks.
- **METIS** - programmipakett, mis sisaldab meetodeid graafide ja lõplike elementide võrkude alampiirkondadeks jagamiseks [18]. METIS-paketti kasutatakse DOUG töös alampiirkondade genereerimiseks etteantud graafi tükeldamise teel vastavalt ette antud protsesside arvule.

- **BLAS** - *Basic Linear Algebra Subroutines*, lineaaralgebra põhitehete (maatriksite ja vektorite vaheliste tehete) sooritamiseks loodud funktsioonide kogum, mis on optimeeritud ette antud arvuti arhitektuuri jaoks. BLAS kasutamine kiirendab tunduvalt arvutusmahukate programmide tööd. BLAS standardile on loodud mitmeid erinevaid realisatsioone, näiteks GotoBLAS [34] või ATLAS [5].

Kus võimalik, kasutab DOUG mitteblokeerivat kommunikatsiooni. Ülesandeid lahendadakse iteratiivselt, baseerudes Krõlovi meetoditele [44]. Eelkonditsioneerijaks [44] on alampiirkondadeks jagamise meetodil baseeruv paralleelne 2-tasemeline lahendaja. Eelkonditsioneerijaga iteratiivsete meetodite puhul on vajalikud allloetletud põhioperatsioonid.

- Maatriksi ja vektori korrutis: $y := Ax$. Maatriksi ja vektori korrutamistehte juures kasutab DOUG protsesside omavaheliseks kommunikatsiooniks mitteblokeerivaid käske `MPI_Isend` ja `MPI_Irecv`. Selle tehte korral toimub suhtlus vaid nende protsesside vahel, kes tegutsevad elemendigraafi alampiirkondadel, mis on omavahel naabrid (alampiirkonnad on genereerinud METIS).

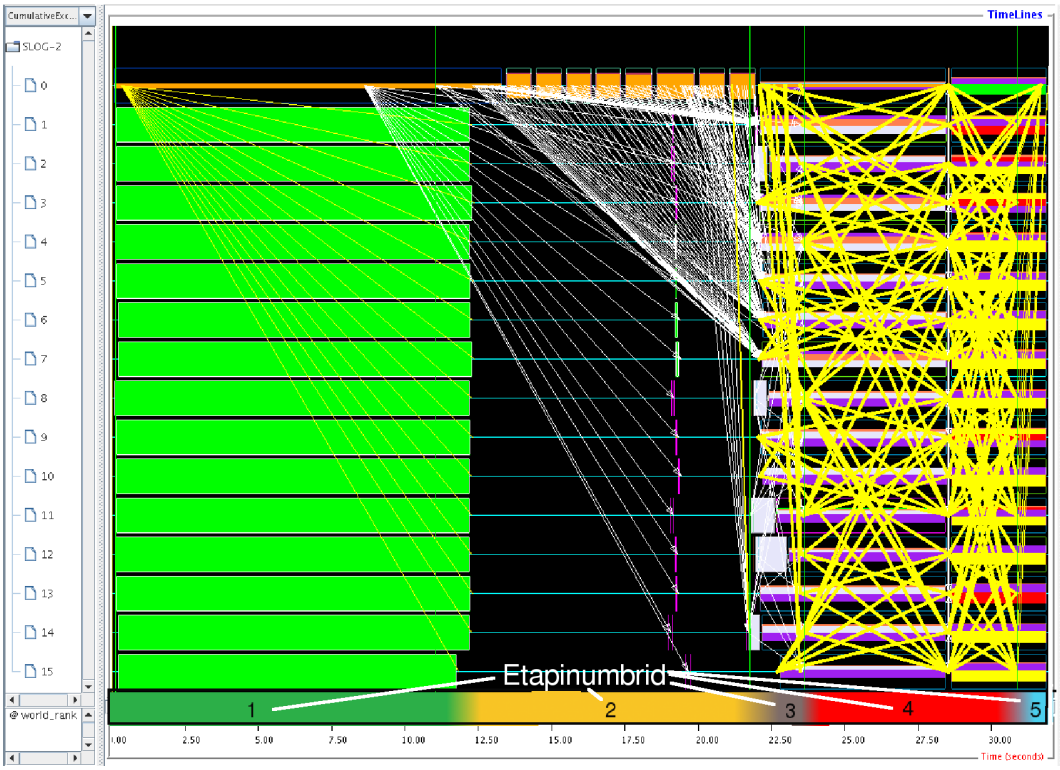
- Skalaarkorrutis: (x, y). Skalaarkorrutise kasutab DOUG blokeerivat kollektiivset teateedastust (MPI käsku `MPI_Allreduce`), kus leitakse kõikide protsesside pealt skalaarkorrutiste summa.

- Eelkonditsioneerija P rakendamine $Pz = r$.

DOUG lahendajale saab andmeid ette anda kahel viisil: elementide kujul (*elemental form*), milleks on failid, mis on genereeritud mõne lõplike elementide meetodi (*finite element method* [45]) paketi poolt, mis kirjeldavad elementide võrku (*element mesh*) ja elementide jäikusmaatrikseid (*element stifness matrices*), või siis assembleeritud kujul (*assembled form*), kus sisendfailid kirjeldavad juba kokku pandud süsteemimaatriksit [13]. DOUG paketis on kaks käivitatavat faili ülesannete lahendamiseks: *doug_main* ja *doug_aggr*, viimane kasutab jämeda võrgu eelkonditsioneerija loomiseks isevoolu strateegiat, mille idee pärineb algselt meetodist AAMM *(Aggregation-based Algebraic Multigrid Method)* [32] ning kasutab kombineeritult alampiirkondadeks

jagamise meetodite vahendeid. *Doug_ aggr* on esialgu veel arendusjärgus ja seda käesolevas töös vaatluse alla ei võeta.

Doug_ main käivitamisel loeb *master*-protsess sisendfailidest mällu elementide võrgu, elemendi jäikusmaatriksid ning elemendi parempoolsed vektorid (*element right hand side vectors*) ning genereerib neist elemendigraafi (*dual graph*). See info jagatakse *master*-protsessi poolt teistele alamprotsessidele. METIS abil jagatakse elemendigraaf alampiirkondadeks vastavalt loodud protesside arvule ning ülesanne lahendatakse iteratiivselt teostades eelkonditsioneerimise operatsioone alamosadel eraldi, kuni saavutatakse piisavat järku koonduvus. Programmi lõpufaasis kogub *master*-protsess tulemused kokku ja kirjutab/kuvab lahendi.



Joonis 4.1: *Doug_ main* programmi kommunikatsiooni üldpilt 16 protsessi puhul

Doug_ main tööd kujutab joonis 4.1, kus on Jumpshot-4 abil visualiseeri-

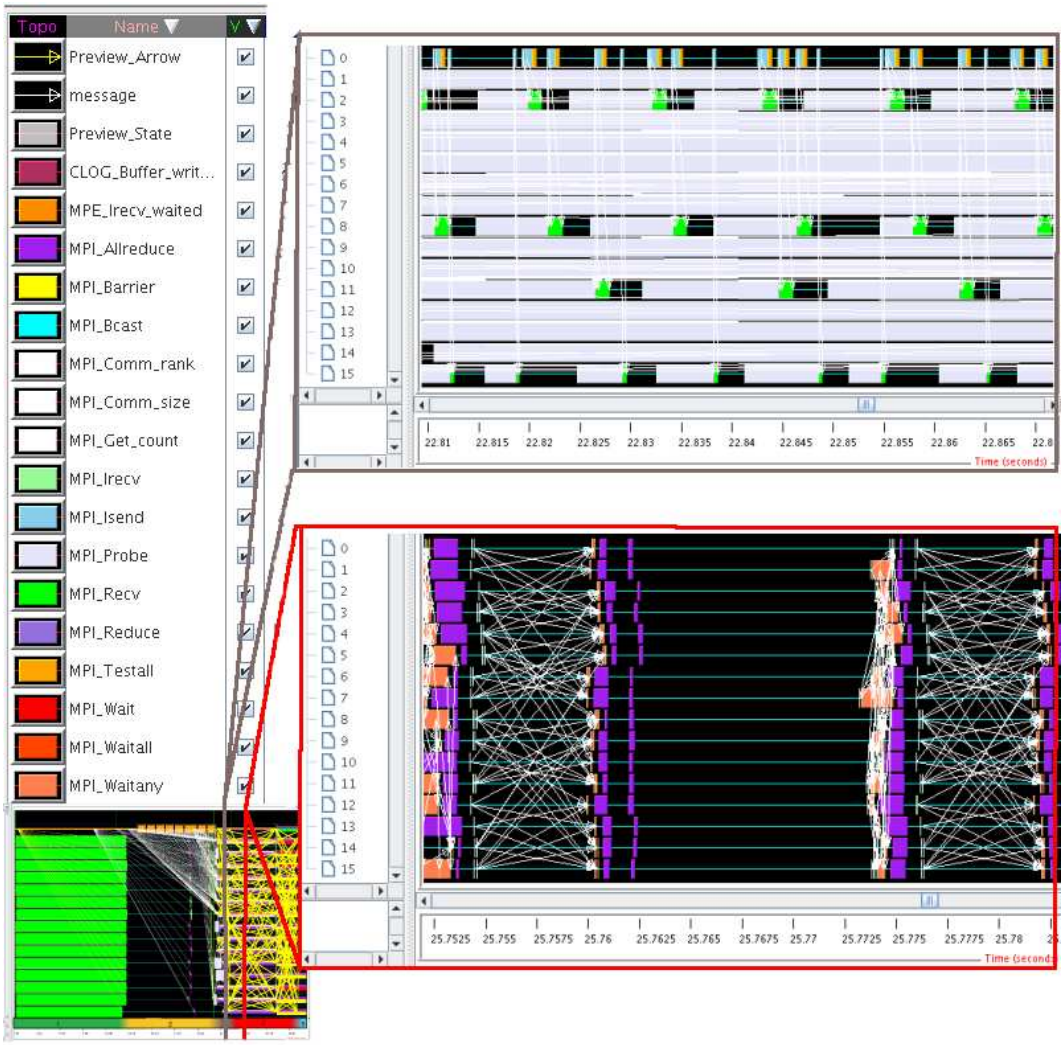
tud ülesande lahendamine paralleelselt 16. protsessoril. Värvide tähendused on välja toodud joonisel 4.2. Jumpshot-4 visualiseeritud kommunikatsioonilogile on lisatud alla aegjoonele ka programmi etappe illustreeriv värviriba.

Esimesel etapil loeb *master-protsess* sisendfailidest sisse lähteülesande andmed ning edastab info elementide võrgu kohta teistele protsessidele. Alamprotsessid ootavad info saabumist, sest vastuvõtt on blokeeriv, kuna enne saabuvat infot ei ole võimalik arvutustöödega alustada. Teine etapp algab *masteri* jaoks kohe, kui ta esimese etapi info teele on saatnud ning teisel etapil tegeleb *master* elemendigraafi konstrueerimisega ja selle METIS abil alampiirkondadeks jagamisega. Peale seda saadab ta teistele protsessidele (mitteblokeeruvalt) info elementide jagunemise kohta alampiirkondade vahel elemendigraafis. Kohal, kus joonisel 4.1 on risti üle aegjoonte esimene roheline vertikaaljoon (tähistamaks ajahetke programmi käimise 11. sekundi lähistel) arvutab *master* välja, kes on tema naabrid, lähtudes alampiirkonnast, mis sai talle määratud elemendigraafi partitsioneerimise käigus ning alustab kolmandat etappi - lokaalse assembleeritud süsteemimaatriksi koostamist, mille käigus *master* tükkhaaval jagab infot teistele protsessidele.

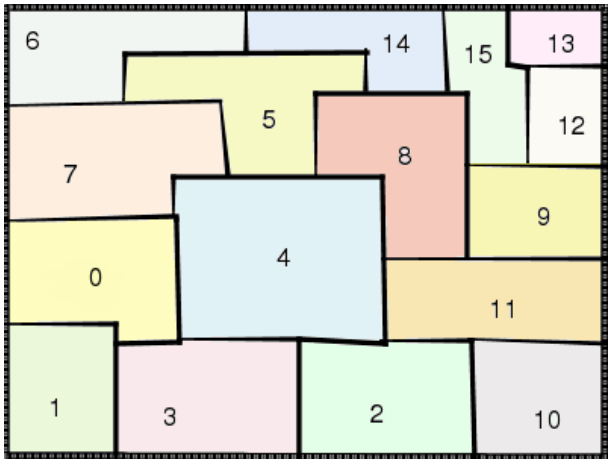
Alamprotsesside jaoks koosneb teine etapp lokaalsel tasemel elemendigraafi alampiirkondadeks jagamisega ning lõpeb info saamisega *masterilt* elementide jagunemise kohta elemendigraafi alampiirkondade vahel. Kolmandal etapil alustavad ka alamprotsessid lokaalse assembleeritud süsteemimaatriksi koostamist, saades vajaliku info *master-protsessilt*. Kolmanda etapi suhtluse illustratsioon on suurendatult toodud välja ka joonise 4.2 ülaosas. *Master* edastab infot mitteblokeerivalt, kuid alamprotsessid peavad selle vastu võtmiseks kasutama blokeerivat MPI käsku, kuna arvutustega (assembleeritud süsteemimaatriksi koostamisega) ei saa enne info saabumist jätkata. Enne teadete vastuvõttu kontrollitakse blokeeruva `MPI_Probe` käsu abil, kas vastav teade on juba saabunud.

Vahetult enne kolmanda etapi algust tuvastavad oma naabrid ka teised alamprotsessid, seda ajahetke sümboliseerib joonisel 4.1 teine roheline aega tähistav vertikaaljoon ajatelge mööda liikudes. Esimest kuni kolmandat etappi kõikidel protsessidel võib käsitleda ka ühe ühtse ülesseadmise etapina, sest ülesande lahendamine toimub alles neljandal etapil.

Neljandal etapil toimuvad iteratsioonid, mis jäävad viimase kahe rohelise



Joonis 4.2: *Doug_ main* programmi üksikud kommunikatsiooniosad suurendatult



Joonis 4.3: Elemendigraafi alampiirkonnad/naaberpiirkonnad käesolevas näites

vertikaaljoone poolt märgitud ajavahemikku joonisel 4.1. Iteratsioonide ajal toimuv on suurendatult välja toodud ka joonise 4.2 alaosas. Nagu eelnevalt mainitud, siis on ka siit jooniselt näha, et iteratsioonide puhul toimub kahte sorti suhtlust - naabritevahelist mitteblokeerivat ning kollektiivset ja blokeeruvit. Blokeerivaks käsuks on siinkohal `MPI_Allreduce`, mis kogub kokku kõikide protsesside saatepuhvri info, sooritab nendel määratud tehte (antud juhul summeerimise) ning kirjutab tulemuse kõikide protsesside vastuvõtupuhvrisse. Antud näiteülesande ning 16 protsessi korral näeksid alampiirkonnad välja nagu toodud joonisel 4.3. Joonis on toodud vaid naabrussuhteid illustreerival eesmärgil ning ei kajasta elemendigraafi elementide jagunemist partitsioonide vahel ega ka alampiirkondade vastavaid suurusi. Kuna protsessi 1 naabriteks on vaid protsessid 0 ja 3, siis saadab protsess 1 teateid vaid protsessidele 0 ja 3 iteratsiooni vastaval sammul nagu näha jooniselt 4.2.

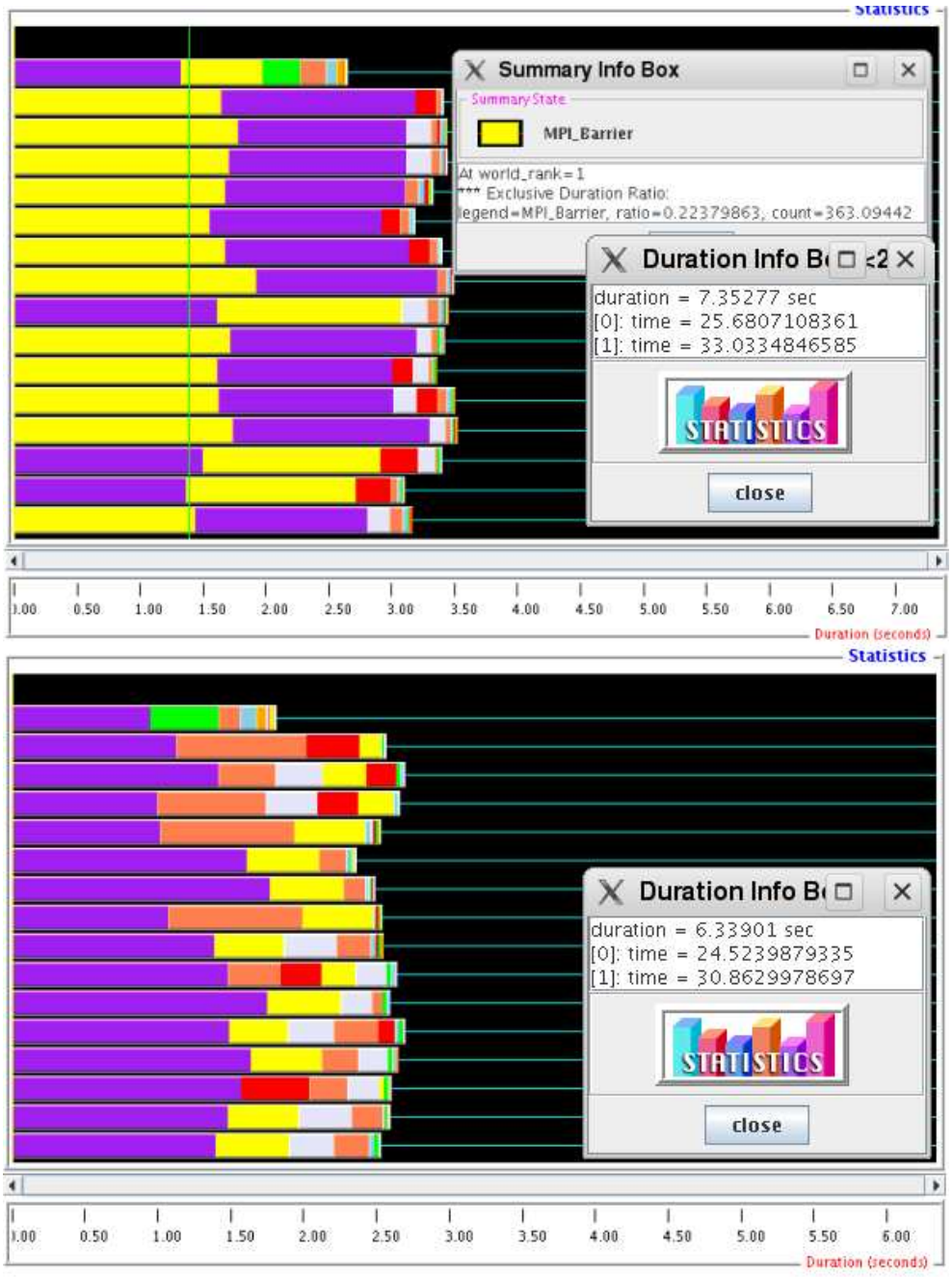
Viimasel viiendal etapil summeerib *master* tulemused kokku (taaskord blokeeriv kommunikatsioon kasutades `MPI_Reduce` käsku, mis erineb eelpool mainitud `MPI_Allreduce` käsust vaid selle võrra, et tulemus kirjutatakse vaid juurprotsessi vastuvõtupuhvrisse), lisaks saadavad kõik alamprotsessid osa tulemusandmeid *masterile* veel ka mitteblokeerivalt. Kui *master* on tulemused kätte saanud, siis sellega lõppeb ka programmi kommunikatsiooniosa.

Peatükk 5

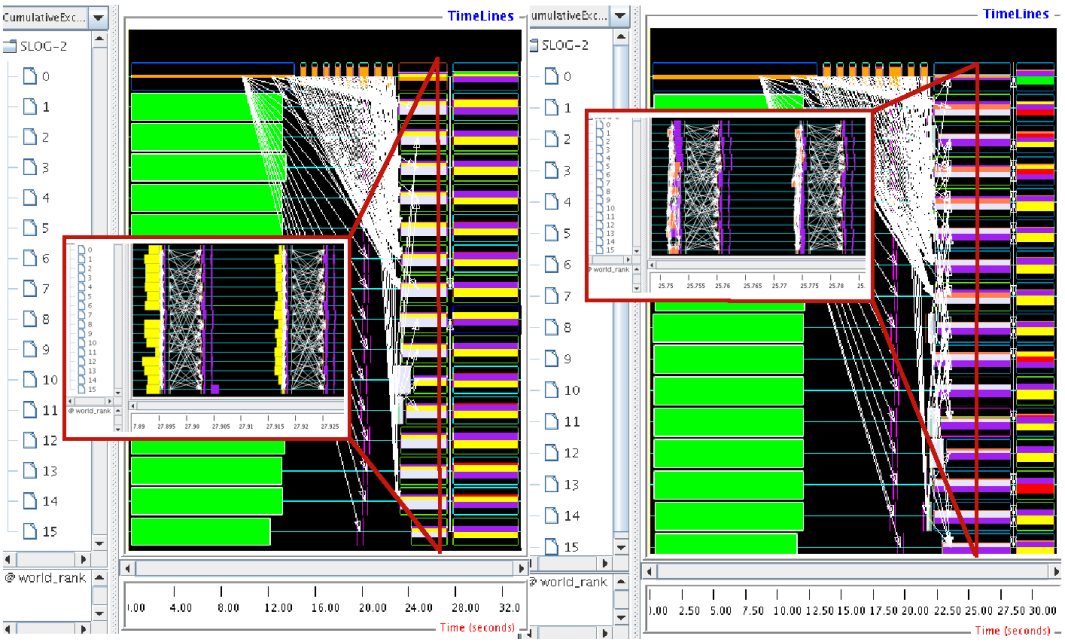
Profileerimisinfo analüüs tarkvarapaketil DOUG

Käesoleva töö üheks eesmärgiks on ka koguda profileerimisinfot tarkvarapaketi DOUG kohta. Lähtudes eelnevalt tutvustatud vahenditest on käesoleva töö autor valinud kasutatavaks profileerijaks MPE ja tulemuste analüüsiks Jumpshot-4. Kõige rohkem võimalusi pakuks kindlasti Allinea OPT ning oleks autori esimene eelistus, kuid kuna autoril puudub selle kasutamislitsents ning proovilitsentsi võib kasutada vaid programmiga tutvumise eesmärgil, siis pole Allinea OPT kasutamine antud hetkel võimalik. MpiP profileerijaga tekkisid aga DOUG paketiga kasutamisel segmentatsioonivead, kui seda jooksutada rohkem kui ühel protsessoril ning programm katkestast tulemusteta oma töö. Käesoleva töö kirjutamise käigus tuli ka MpiP profileerijast välja uus versioon, kuid sellegi puhul on probleem sama. Segmentatsioonivead DOUG profileerimisel MpiP profileerijaga erinevatel arvutiarhitektuuridel ja operatsioonisüsteemidel. Põhjus on senini teadmata. Selle probleemi silumine ei kuulu kahjuks töö autori pädevusvaldkonda ning MpiP kasutamisest antud eesmärgi saavutamiseks tuli loobuda.

MPE abil tehtud profileerimiskatsed on sooritatud *kuu*-arvutiklastril, mis Tartu Ülikooli Arvutiteaduse Instituudi Hajussüsteemide õppetooli tarbeks loodud kaheksast *Dual Core AMD Opteron(tm)* 2,2GHz protsessoriga 64-bitise aadressiedastusega arvutist koosnev klaster, seega maksimaalselt sai kasutada paralleelselt 16 protsessorit. Genereeritud logifailid on käesolevale



Joonis 5.1: Iteratsioonietapi sagedustabel. Ülemine kajastab tulemusi enne *MPI_ Barrier* käsu eemaldamist iteratsioonidest ja alumine pärast



Joonis 5.2: *Doug_ main* üldpilt enne (vasakul) ja pärast sünkronisatsiooni eemaldamist

tööle lisana kaasa pandud CD plaadil.

Logifailides olevat infot analüüsides esimese iseloomuliku tunnusena jäi silma `MPI_Barrier` käsu rohkus iteratsioonide etapis. Koodi lähemalt uurides selgus, et sinnagi oli kommentaarideks kirjutatud, et `MPI_Barrier` käsud tuleks sealt eemaldada. Tõenäoliselt on tegu silumise käigus lisatud sünkronisatsiooniga, mis on sinna ka ununenud. Iteratsioonietapi sagedustabelit uurides (joonise 5.1 ülemine osa) on näha, et kokku saaks antud näites hoida umbes 2 sekundit, kui `MPI_Barrier` funktsioon eemaldada. Peatükis 3.3 kirjeldatud Jumpshot-4 programmi iseärasuse tõttu on käesolevasse iteratsioonietappi sagedustabelil sisse arvestatud ka `MPI_Barrier` väljakutsed iteratsioonietapi ümbruses, seega ei anna joonis 5.1 väga korrektset ülevaadet - nagu näha sama joonise alumiselt osalt, siis pärast iteratsioonietapist eemaldamist on `MPI_Barrier` siiski kajastatud endiselt sagedustabelis, kuigi väiksemal määral. Antud juhul on siiski võimalik neid kahte histogrammi omavahel võrrelda, sest need on genereeritud samalt suurendustasemelt

ning samal suurendustasemel on nende kahe logifaili eelvaate-olekud enamvähem sarnased, sest tegu on sama lähteülesande ja sama protsesside arvuga, seega mõlemasse sagedustabelisse on kaasatud iteratsioonietapi lähiümbruse `MPI_Barrier` olekud.

Joonis 5.2 illustreerib *doug_ main* programmi üldpilti ja iteratsioonietapi detaile enne ja pärast `MPI_Barrier` eemaldamist. Parema vaadeldavuse huvides on eelvaate-noolte kuvamine välja lülitatud.

Infot analüüsides ei olnud märgata sõnumite saatmisega hiljaksjäämise probleemi. See on hea tulemus. Kuid teisest küljest on märgata, et kulub väga palju aega kuni *masteri* poolt saadetud lähteülesande info alamprotsessidele kohale jõuab, kuna see on niivõrd mahukas. Selle koha peal oleks ruumi optimeerimiseks. Info sisselugemiseks võiks kasutada MPI-2 standardis [24] kirjeldatud paralleelseid sisend- ja väljundvõimalusi, et lugeda elementide võrk sisse paralleelselt ning koostada elemendigraaf hajusalt. Kui METIS asemel võtta elemendigraafi partitsioneerimiseks kasutusele parMETIS [19], mis on paketi METIS paralleelversioon, siis oleks võimalik hajusana asuv elemendigraaf ka partitsioneerida hajusalt, hetkel on vaja, et elemendigraafi info kajastuks tervikuna ühe protsessi andmetes. Antud näidisülesande puhul kestab partitsioneerimine *master*-protsessil alla sekundi, seega partitsioneerimise paralleeliseerimine ei ole kõige olulisem, kuigi muutub olulisemaks lähteülesannete keerukuse tõusul. Andmete sisselugemise ning elemendigraafi koostamise paralleeliseerimiseks on rohkem põhjust, esimest eelkõige põhjusel, et sisendinfo saatmine teistele protsessidele on ajakulukas.

MPE ei mõõda kogu programmi tööaega vaid ajakulu teateedastusliidese initsialiseerimise hetkest kuni selle lõpetamiseni, seega terve programmi ajakulu mõiste tähendab antud juhul ainult programmi kommunikatsiooniosa alustamise ja lõpetamise vahelist ajakulu. Käivitades programmi erinevatel hetkedel sama lähteülesandega ning sama paralleelsete protsesside arvuga

Tabel 5.1: DOUG kommunikatsiooniosa minimaalne ajakulu sekundites vastavalt protsesside arvule

4	8	12	16
43,89	35,47	32,63	31,96

on tulemus varieeruv ning sõltuv arvutiklastri protsessorite hetkekoormusest. Tabelis 5.1 on toodud minimaalne ajakulu 4, 8, 12 ja 16 protsessi korral vähemalt kolmest eri aegadel sooritatud katsetest samal lähteülesandel (programmiversioonis, kus üleliigne sünkronisatsioon on juba eemaldatud).

Keskmise ajakulu arvestamiseks rohkete katsete korral eri aegadel oleks kõige mugavam kasutada MpiP profileerijat, sest selle rakendamise lisakulud on võrreldes teiste profileerijatega väiksemad ning tulemusena saadavaid tekstifaile oleks hõlbus automaatselt töödelda, sealt vajalikud andmed kokku koguda. MPE korral on esiteks logifailid juba väga mahukad (antud näite korral enamasti vahemikus 50-70 MB) ning teiseks toimub nende analüüs manuaalselt Jumpshot-4 visualiseerija abil, seega antud juhul on piirdutud 3-6 katsega iga kombinatsiooni kohta. Kõige suurem kasu tundub olevalt neljalt protsessorilt üleminekul kaheksale, kuid sealt edasi aja kokkuhoid aeglustub, kuna nagu eelnevalt mainitud, siis suur hulk ajast kulub programmi töö käigus elementide võrgu info saatmisele ja elemendigraafi koostamisele, mis ei ole hetkel paralleeliseeritud, seega protsesside lisandudes saab ainult lüheneda iteratsioonide (neljas) etapp ning teisest küljest iga protsessi lisandudes lisandub ka rohkem kommunikatsiooni.

Kokkuvõte

Käesolevas töös tutvustati teateedastusliidest, selle erinevaid realisatsioone ning MPI programmide eripärasid. Selgitati, mis on programmide profileerimine ja jälgimine, ning mis kasu sellest saada on võimalik. Tutvuti erinevate jõudlusanalüüsi läbiviimise vahendite ja meetoditega ning peatuti mõne profileerija omaduste ja võimaluste tutvustamisel pikemalt. Seejärel valiti sobilik vahend ja profileeriti selle abil eelnevalt kirjeldatud suurte lineaarsüsteemide lahendamiseks mõeldud tarkvarapaketti DOUG. Teostati tulemuste esialgne analüüs ning suuri ning leiti optimiseerimist võimaldavaid kohti. Käesolevale tööle on lisatud valik logifaile DOUG-paketi arendajatele soovi korral sügavamaks analüüsiks. Edasiarendusena tuleks samu katseid korrata kordi mahukamate lähteülesannetega tunduvalt suuremal arvutiklastril saamaks paremat ülevaadet programmi skaleeruvusest. Ühtlasi oli käesolevas töös välja jäetud DOUG-paketi arendusjärgus olev osa.

Performance Analysis Tools and Methods for MPI programs

Master thesis

Mai-Liis Karring

Abstract

The main purpose of this Master thesis was investigate the performance analysis tools and methods for *Message Passing Interface* programs for the communication and synchronization parts. Performance issues are very important in context of parallel programs due to the extra effort and resources they require during the development phase compared to serial programs. There is a danger that in case of bad design the parallel program will work less efficiently than a serial program which solves the same issue. Additional purpose was to use some of the introduced tool and gather profiling data for DOUG software, which is a *black box* solver for large sparse linear systems being developed at Tartu University in collaboration with University of Bath (United Kingdom). As a result it was found that some optimising could be done at the setup phase. Further tests should be carried out in the future on larger computer clusters and with more complex tasks.

Kirjandus

[1] Allinea Software. *OPT User Guide.*
`Version 1.2`.

[2] Allinea Software. *UsingăOPT A White Paper.*
`http://www.allinea.com/downloads/OPTWhite.pdf` - viimati väisatud 26.04.2007.

[3] Argonne National Laboratory, Mississippi State University. *MPICH2.*
`http://www-unix.mcs.anl.gov/mpi/mpich` - viimati väisatud 2.03.2007.

[4] Argonne National Laboratory, Mississippi State University. *MPICH - A Portable Implementation of MPI.*
`http://www-unix.mcs.anl.gov/mpi/mpich1/` - viimati väisatud 21.03.2007.

[5] *Automatically Tuned Linear Algebra Software (ATLAS).*
`http://math-atlas.sourceforge.net/` - viimati väisatud 15.05.2007.

[6] Anthony Chan, David Ashton, Rusty Lusk, William Gropp. *Jumpshot-4 Users Guide.*
`Mathematics and Computer Science Division, Argonne National Laboratory`.

[7] Anthony Chan, Bill Gropp, Rusty Lusk. *Performance Visualization for Parallel Programs.*
`http://www-unix.mcs.anl.gov/perfvis/software/log_format/index.htm` - viimati väisatud 3.05.2007.

[8] Michael Collette, Bob Corey, John Johnson. *High Performance Tools & Technologies.*
`http://www.llnl.gov/computing/tutorials/performance_tools/HighPerformanceToolsTechnologiesLC.pdf` - viimati väisatud 30.04.2007.

[9] Critical Software SA. *What is MPI.*
`http://www.criticalsoftware.com/hpc/what_is_mpi.php#5` - viimati väisatud 12.03.2007.

[10] *DeinoMPI.*
`http://mpi.deino.net/index.htm` - viimati väisatud 04.04.2007.

[11] Department of Computer and Information Science and Engineering, University of Florida. *UMFPACK.*
`http://www.cise.ufl.edu/research/sparse/umfpack/` - viimati väisatud 15.05.2007.

[12] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski. *Automated, scalable debugging of MPI programs with Intel®Message Checker.*
`http://csdl.ics.hawaii.edu/se-hpcs/papers/11.pdf` - viimati väisatud 4.04.2007.

[13] *DOUG Development Site.*
`http://www.dougdevel.org/` - viimati väisatud 21.05.2007.

[14] Ian Foster. *Designing and Building Parallel Programs,* peatükid 1.3, 2.3 ja 8.
`http://www-unix.mcs.anl.gov/dbpp/` - viimati väisatud 4.04.2007.

[15] William Gropp. *Tutorial on MPI: The Message-Passing Interface.*
`http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/node11.html#Node11` - viimati väisatud 02.04.2007.

[16] Indiana University LAM Team. *LAM-MPI.*
`http://www.lam-mpi.org/` - viimati väisatud 21.03.2007.

[17] ©Intel®Corporation. *Intel Trace Analyzer and Collector* .
http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/306321.htm - viimati väisatud 21.05.2007.

[18] George Karypis. *METIS*.
http://glaros.dtc.umn.edu/gkhome/metis/metis/overview - viimati väisatud 15.05.2007.

[19] George Karypis. *ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering*.
http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview - viimati väisatud 22.05.2007.

[20] Lawrence Livermore National Laboratory. *ToolGear Viewer*.
http://www.llnl.gov/CASC/tool_gear/ - viimati väisatud 20.05.2007.

[21] Adam Leko, Hans Sherburne, UPC Group HCS Research Laboratory University of Florida. *MPE/Jumpshot Evaluation Report*.
http://www.hcs.ufl.edu/upc/archive/toolevals/Presentations/MPEJumpshotEvaluation.ppt - viimati väisatud 8.05.2007.

[22] Bernd Mohr. *The Future of Performance Optimization Tools*.
http://www.fz-juelich.de/zam/datapool/KojakTalks/isc2005.pdf - viimati väisatud 25.04.2007.

[23] Shirley Moore, David Cronk, Kevin London, Jack Dongarra. *Review of Performance Analysis Tools for MPI Parallel Programs,* Computer Science Department, University of Tennessee Knoxville.
http://www.cs.utk.edu/ cronk/papers/epvm01.ps - viimati väisatud 27.04.2007.

[24] MPI Forum. *MPI Documents*.
http://www.mpi-forum.org/docs/ - viimati väisatud 2.05.2007.

[25] NSF Engineering Research Center, Mississippi State University. *MPI Implementations*.
http://hpcl.cse.msstate.edu/projects/mpi/implementations.html - viimati väisatud 21.03.2007.

[26] Open MPI Development Team. *Open-MPI.*
`http://www.open-mpi.org/` - viimati väisatud 21.03.2007.

[27] PAPI Group. *Performance Application Programming Interface.*
`http://icl.cs.utk.edu/papi/` -viimati väisatud 21.04.2007.

[28] Paradyn Project, Computer Sciences Department, University of Wisconsin. *Paradyn.*
`http://www.paradyn.org/` -viimati väisatud 21.05.2007.

[29] Dr. Douglas M. Pase. *Dynamic Probe Class Library (DPCL).*
`http://www.cs.wisc.edu/ paradyn/DPCL/` - viimati väisatud 19.05.2007.

[30] Performance Research Lab, University of Oregon. *Tuning and Analysis Utilities.*
`http://www.cs.uoregon.edu/research/tau/` - viimati väisatud 21.05.2007.

[31] Rolf Rabenseifner. *Parallel Performance Analysis and Profiling.*
`http://www.hlrs.de/organization/par/par_prog_ws/2006F/16_performance_analysis_mpi.pdf` - viimati väisatud 29.04.2007.

[32] R. Scheichl, E. Vainikko. *Robust Aggregation-Based Coarsening for Additive Schwarz in the Case of Highly Variable Coefficients*, Proceddings of the European Conference on Computational Fluid Dynamics, ECCOMAS CFD 2006 (P. Wesseling, E. ONate, J. Periaux, Eds.), TU Delft, 2006.

[33] Luis Moura E Silvay, Rajkumar Buyyaz. *Parallel Programming Models and Paradigms.*
`http://www.gridbus.org/ raj/cluster/v2chap1.pdf` - viimati väisatud 21.03.2007.

[34] Texas Advanced Computing Center, The University of Texas at Austin. *GotoBLAS.*
`http://www.tacc.utexas.edu/resources/software/software.php` - viimati väisatud 15.05.2007.

[35] UCLA Academic Technology Services. *An Introduction to MPI – the Message-Passing Interface.* `http://www.ats.ucla.edu/rct/hpc/parallel_computing/mpi-intro.htm` - viimati väisatud 10.03.2007.

[36] *University of Bath.* `http://www.bath.ac.uk/` - viimati väisatud 15.05.2007.

[37] University of Maryland. *An Application Program Interface (API) for Runtime Code Generation.* `http://www.dyninst.org/` - viimati väisatud 22.05.2007.

[38] UPC Group, High-performance Computing Lab, University of Florida. *Parallel Performance Wizard v1.0 User Manual*, peatükk 1.2 "Profile Terminology". `http://ppw.hcs.ufl.edu/docs/htmlsplit/manual/Profile-Terminology.html` - viimati väisatud 8.05.2007.

[39] Eero Vainikko. *Fortran95 ja MPI.* `Tartu Ülikooli kirjastus, 2004`.

[40] Eero Vainikko. *Suuremahulised teadusarvutused, paralleelsus ja GRID.* `http://www.ut.ee/ eero/talks/VeniaLegendi.pdf` - viimati väisatud 15.05.2007.

[41] Heikki Vallaste. *E-Teatmik.* `http://vallaste.ee/` - viimati väisatud 21.05.2007.

[42] Verari Systems Software. *MPI/pro.* `http://www.mpi-softtech.com/mpi_pro.php` - viimati väisatud 21.03.2007.

[43] Jeffrey Vetter, Chris Chambreau. *MpiP: Lightweight, Scalable MPI Profiling.* `http://mpip.sourceforge.net/` - viimati väisatud 19.05.2007.

[44] Henk A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems.* Cambridge University Press, 2003.

[45] Roger Young, Ian MacPhedran. *Internet Finite Element Resources.* `http://homepage.usask.ca/~ijm451/finite/fe_resources/fe_resources.html` - viimati väisatud 20.05.2007.

Lisad

Lisa 1: CD-ROM tarkvarapaketi DOUG logifailide valikuga visualiseerija Jumpshot-4 jaoks.

Lisa 1: CD-ROM tarkvarapaketi DOUG logifailide valikuga visualiseerija Jumpshot-4 jaoks

CD-plaadil asetseb valik *MPI Parallel Environment 2* (MPE2) teegi abil kogutud profileerimis- ja jälgmimislogisid tarkvarapaketile DOUG. Logifailid on *SLOG-2* failiformaadis ja sobivad vaatamiseks MPE2 visualiseerijaga Jumpshot-4. MPE2 versioon 1.0.4 on lähtekoodina samuti lisatud.

Logifaili nime algus viitab lähteülesandele. Faili nime teine osa viitab paralleelsete protsesside arvule. Märgistus "with_barrier" failinimes tähistab neid logifaile, kust üleliigsed väljakutsed funktsioonile `MPI_Barrier` on eemaldatud, nagu on pikemalt kirjeldatud peatükis 5.