VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
KOMPIUTERINIO MODELIAVIMO KATEDRA

Baigiamasis magistro darbas

LYGIAGRIETIEJI ALGORITMAI TIESINĖS ALGEBROS UŽDAVINIUOSE

Parallel algorithms for linear algebra problems

Atliko: II kurso, 9 grupės studentas

Andžej Šuškevič (parašas)

Darbo vadovas:

doc. Tadas Meškauskas (parašas)

Darbo recenzentas:

dr. Jolita Ignatavičiūtė (parašas)

Vilnius
2007

Turinys

Anotacija

Šiame magistro baigiamajame darbe yra nagrinėjami tiesinės algebros uždavinių sprendimai, panaudojant įvairias skaičiavimo priemones bei specializuotas bibliotekas. Pagrindinis darbo tikslas yra ištirti tokių uždavinių sprendimo būdus bei išnagrinėti jų išlygiagretinimo galimybes. Išanalizavus susijusią literatūrą, ištyrus pasirinktų tiesinių lygčių sistemų sprendimo algoritmus bei atlikus susijusius praktinius bandymus buvo pateikta detali algoritmų analizė bei jų panaudojimo rekomendacijos. Be to, panaudojant BalticGrid technologines galimybes, pavyko paskirstyti tiesinės algebros uždavinių sprendimą tarp keliasdešimties kompiuterių, tuo pačiu sumažinus bendrą užduočių skaičiavimo laiką.

Summary

The title of this work is “Parallel algorithms for linear algebra problems”. The main goal of Master thesis is to research solving possibilities of linear algebra problems, using different kind of computing machines and dedicated linear algebra libraries.

In the beginning of the work author introduces the comparison of the system for parallel computing such as symmetric multi-processing and massively parallel processing. Later in this chapter the main linear algebra problems and theirs solutions were introduced. In the beginning of the second part, different kinds of algorithms for solving linear equation systems, such as LU factorization and SVD – singular value decomposition, were researched. In the next part of this chapter author looked for possibility to make computations of linear equation system in a parallel way.

In the practical part of the work, the author developed few programs, which were used for analysis of different kind of algorithms and used BalticGrid technologies for parallel solving of linear equation systems.

In the closing part of the work author presents the main results of the work and suggests some recommendations.

Įvadas

Tikslas

Šio darbo tikslas yra ištirti tiesinės algebros uždavinių sprendimo būdus bei išnagrinėti jų išlygiagretinimo galimybes.

Uždaviniai

1. Atlikti tiesinių lygčių sprendimo algoritmų analizę, bei pateikti panaudojimo rekomendacijas.
2. Ištirti specifines bibliotekas, skirtas tiesinių algebros uždavinių sprendimui.
3. Ištirti, kaip triukšmo pridėjimas bei jo intensyvumas gali paveikti tiesinės lygčių sistemos sprendinį.
4. Parinkti tinkamą strategiją, kuri leistų išlygiagretinti didelės apimties tiesinės algebros uždavinių paketo skaičiavimus.

Rezultatai

1. Remiantis atliktais algoritmų tyrimais bei atliktais bandymais, buvo detaliai išanalizuoti atrinkti algoritmai, įvertinant jų skaičiavimo laiką, skaičiavimo paklaidas ir netikties skaičiavimą.
2. Buvo ištirtos bei panaudotos tokios tiesinės algebros bibliotekos, kaip atlas, lapack, blas, cerfacs, plapak ir scalapack bei atrinkti keli algoritmai tolimesniems tyrimams atlikti.
3. Buvo atlikti tyrimai ir bandymai, kurie leido įvertinti triukšmo įtaką galutiniam sistemos sprendiniui.
4. Buvo sukurta strategiją, kuri leido panaudojant BalticGrid skaičiavimo galimybes išlygiagretinti didelės apimties tiesinės algebros uždavinių paketo sprendimą, kas savo ruožtu, leido sumažinti bendrą užduočių skaičiavimo laiką.

Aktualumas

Laikai, kada žodis kompiuteris reiškė ne mažą dėžę, šiandien vadinama asmeniniu kompiuteriu, o didžiulį prietaisą, kuris užimdavo, didžiules patalpas, kuriose šiandien galėtų tilpti visas būrys programuotojų su savo asmeniniais kompiuteriais, tie laikai, kada tokius prietaisus sau galėjo leisti tik patys didžiausi universitetai arba didžiulės organizacijos ir kai buvo nuolat siekiama, bet kokia kaina, padidinti procesoriaus galingumą, bei pagreitinti atmintį tam, kad galima būtų pakelti bendrą

superkompiuterio galingumą, tie laikai jau seniai praėjo. Šiandien tam, kad galima būtų pasiekti didžiules skaičiavimo galimybes, yra dažniau naudojamas kitas kelias – ne vieno procesoriaus nuolatinis tobulinimas, kuris yra be galo brangus – o procesorių skaičiaus didinimas bei specializuotos programinės įrangos tobulinimas, optimizavimas ir paruošimas lygiagrečiam darbui su keliais procesoriais. Šis būdas tapo pigesne superkompiuterių alternatyva.

Viena svarbiausių rolių skaičiavimo metoduose atlieka tiesinės algebros uždaviniai, kurių pagrindą sudaro tiesinių lygčių sprendimas bei tikrinių reikšmių paieška, kurie yra praktiškai visų algoritmų sudaromoji dalis. Statiškai apie 75% superkompiuterių procesorių darbo laiko yra naudojama būtent tokiems uždaviniams spręsti. Nieko nuostabaus, kad lygiagrečiųjų algoritmų bei lygiagrečiųjų skaičiavimų taikymas tapo būtinybe, norint optimizuoti šio tipo uždavinių sprendimo paiešką. Tai patvirtina ir faktas, kad itin galingos techninės įrangos gamintojai, kartu su savo produktais teikia ir optimizuotas programų bibliotekas, dauguma iš kurių yra būtent programos, skirtos tiesinės algebros uždaviniams spręsti.

Kaip žinia, nagrinėjant tiesinių lygčių sistemų sprendimą, panaudojant įvairią skaičiavimo techniką, reikėtų nagrinėti uždavinius Ax = b tipo, kur A yra matrica, b yra duotas vektorius, o x – sprendinio vektorius. Todėl toliau šiame darbe autorius nagrinės būtent tokių uždavinių sprendimo būdus, ištirs algoritmų efektyvumą, įvertins skaičiavimo paklaidas bei atliks kitus tyrimus. Taip pat bus analizuojamas sistemos sprendinio jautrumas įnešamam triukšmui ir galiausiai, kai bus atlikta analizė ir atlikti bandymai viename kompiuteryje, autorius, užduočių išlygiagretinimui panaudos paskirstytų skaičiavimo tinklus tam, kad galima būtų sumažinti atliekamų tyrimų skaičiavimo laiką.

Pirmoje darbo dalyje, remiantis literatūra, bus išnagrinėtos lygiagrečios kompiuterių architektūros, tiesinės algebros uždavinių sprendimo algoritmų tipai bei jų realizacija, taip pat, specializuotos bibliotekos, kurių pagalba, galima atlikinėti minėtus skaičiavimus. Be to, bus išrinktas keletas įdomesnių algoritmų, kuriuos autorius po to panaudos tolimesniuose tyrimuose.

Antroje darbo dalyje autorius pateiks savo analizę aukščiau minėtoms problemoms. Pasiūlys problemų sprendimo galimybes, paruoš savo teiginių įrodymus teoriniame lygyje bei sukurs pagrindą, kuriuo remiantis po to bus kuriami praktiniai uždaviniai.

Remiantis teorinėmis žiniomis įgytomis pirmoje darbo dalyje, atlikta analize antroje darbo dalyje, praktinėje darbo dalyje bus atlikti testiniai bandymai su pasirinktais algoritmais, ištirti jų stipriosios ir silpnosios pusės bei atlikta visa aibė įvairių eksperimentų. Pirmiausia bus atrinktos kelios tiesinė algebros bibliotekos, keletas algoritmų, paruošti testiniai duomenys kurių pagrindu bus suprogramuotas testinių tiesinės algebros uždavinių paketas. Bandymus planuojama atlikti kaip su atsitiktiniu būdu sugeneruotomis matricomis, taip ir su matricomis, kurios atsiranda realiuose

taikymuose, kai diferencialiniai modeliai yra sprendžiami diskrečiųjų algoritmų pagalba. Tokių matricų taikymo sričių aibė yra labai plati, nuo cheminės inžinerijos iki branduolinės fizikos uždavinių, todėl ir matricų pasirinkimas yra gana ne mažas, kas savo ruožtu leis paįvairinti atliekamus eksperimentus.

Išlygiagretinimo darbams atlikti bus panaudotas BalticGrid servisas, kuris apjungia kelis šimtus kompiuterių ir leidžia vienu metu lygiagrečiai skaičiuoti įvairius uždavinius. Atlikus bandymus, autorius suformuluos magistrinio darbo išvadas, išdėstys tyrimų rezultatus bei pateiks pasiūlymus ir rekomendacijas.

1. Daugiaprocesorinės architektūros

Nuolat augantis poreikis naudoti kompiuterius, kaip priemonę sudėtingiems skaičiavimams atlikti, skatina kompiuterių našumo didinimą. Atliekant tokius skaičiavimus neretai vieno procesoriaus resursų nebeužtenka. Tenka ieškoti alternatyvių sistemų tam, kad galima būtų atlikti skaičiavimus greičiau arba su didesniu informacijos kiekiu ir nelaukti rezultato po keletą dienų arba mėnesių. Tokia alternatyva buvo rasta ir ja tapo daugiaprocesorinės architektūros, kurias galima suskirstyti į keturias grupes [BDZ03]:

- Vektoriniai superkompiuteriai
- Simetrinės daugiaprocesorinės sistemos su bendra atmintimi (SMP – Symmetric Multi-processing)
- Daugiaprocesorinės sistemos su paskirstyta atmintimi arba su masiniu lygiagretumu (MPP - Massively Parallel Processing)
- Klasterinės sistemos

Žemiau bus apibūdintos šių sistemų ypatumai bei išrinkta vieną iš jų būsimiems praktiniams tyrimams atlikti.

1.1. Vektoriniai kompiuteriai (superkompiuteriai naudojantys vektorinius procesorius)

Pirmas tokio tipo kompiuteris buvo Cray-1, kuris buvo pagamintas dar 1976 m. Šio superkompiuterio architektūrinis sprendimas buvo itin sėkmingas, todėl jo pagrindu buvo kuriami vis nauji modeliai. Ši kompiuterinė architektūra remiesi dviem esminiais principais:

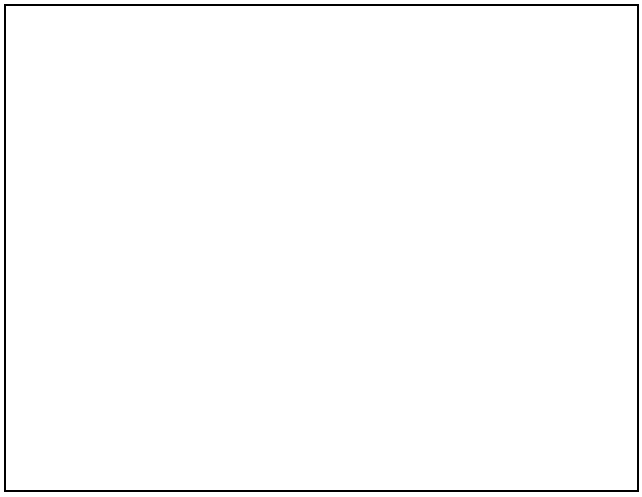
- konvejerio pagrindu apdorojamas komandų srautas;
- galimybė atlikti operacijas su vektoriais, o tai yra efektyvus darbas su dideliais duomenų masyvais.

Pagrindinis vektorinių operacijų tikslas yra išlygiagretinti ciklus, kuriems „prasukti“ sugaištama daugiausiai procesoriaus laiko. Kompiliuojant programas, kompiliatorius automatiškai pakeičia programose aprašytus ciklus, taip kad skaičiavimus galima būtų atlikti vektorinių operacijų pagalba. Verta pastebėti, kad toks paprastas programų išlygiagretinimas, t.y. galimybė naudoti esamas programavimo kalbas, o ne naudoti naują, specialiai pritaikytą vektoriniams procesoriams technologiją, buvo, be abejo, vienas iš tų faktorių, kuris nulėmė šios architektūros populiarumą.

Vektorinių kompiuterių architektūra dažniausiai yra tokio pavidalo: keli vektoriniai procesoriai (nuo 2 iki 16), kurie naudoja bendrą atmintį (SMP) sudaro sistemos mazgą. Mazgai yra sujungiami tarpusavyje komutatorių pagalba, ir tuo pačiu sudaro MPP sistemą. Tokį architektūrinį sprendimą turi tokie kompiuteriai, kaip: CRAY T90 arba NEC SX-5.

1.2.Simetrinės daugiaprocesorinės sistemos su bendra atmintimi (SMP)

Pagrindinis šios architektūros bruožas yra tas, kad visi procesoriai turi tiesioginį priėjimą prie bendros atminties. Pirmos SMP sistemos buvo sudarytos iš kelių procesorių bei bendros atminties masyvo, kuris buvo pasiekiamas per magistralę. Tačiau labai greitai buvo pastebėtas esminis šios architektūros trūkumas, padidinus procesorių skaičių, iškildavo konfliktai, kreipiantys į bendrą atmintį. Problema pavyko iš dalies išspręst, padalinus atmintį į blokus, o prisijungimą prie jų realizuoti komutatorių pagalba. Tačiau šis sprendimas dėl ekonominių sumetimų praktiškai neleidžia panaudoti daugiau negu 32 procesorius vienoje SMP sistemoje.

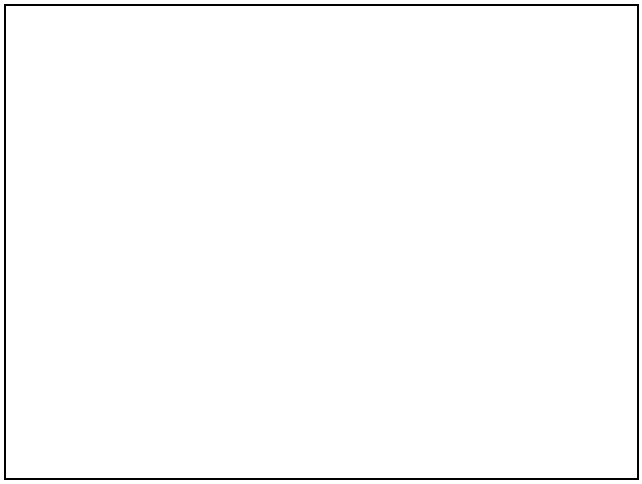


1 pav. Simetrinės daugiaprocesorinės sistemos su bendra atmintimi (SMP) architektūra

Iš kitos pusės, bendros atminties naudojimas yra ir vienas iš didžiausių šios architektūros privalumų, nes jis žymiai supaprastina programų kūrimą, o tai savo ruožtu gali būti esminiu faktoriumi renkantis ne didelę daugiaprocesorinę sistemą. Situacijoje, kai visi procesoriai, turi vienodai greitą priėjimą prie atminties, klausimas, koks procesorius atliks kokius skaičiavimus neturi didesnės įtakos, kas savo ruožtu lemia, kad visa aibė nuosekliųjų algoritmų gali būti, be didesnių problemų, pagreitintą išlygiagretinamųjų bei vektorinių transliatorių pagalba. Šie faktoriai lemia, kad SMP kompiuteriai šiandien yra gana populiarūs. Tačiau jau minėti SMP trūkumai bei tai, kad sistemos kaštai didėja neproporcingai sistemos našumui, didinant procesorių skaičių sistemoje lemia, kad dažnai ieškoma kito architektūrinio sprendimo, kuris galėtų labiau atitikti keliamus reikalavimus.

1.3.Daugiaprocesorinės sistemos su paskirstyta atmintimi (MPP)

Problemos, kurios yra būdingos SMP sistemoms, yra labai lengvai išsprendžiamos sistemose, sukurtose pagal MPP architektūra, kurios pagrindą sudaro daugiaprocesorinė sistema su paskirstyta atmintimi, kurios mazgai komunikuoja per tam tikrą sąsają.



2 pav. Daugiaprocesorinių sistemų su paskirstyta atmintimi (MPP) architektūra

Taigi kiekvieną sistemos mazgą sudaro vienas arba keli mikroprocesoriai, atskira operatyvioji atmintis, komunikavimo bei įvedimo/išvedimo įranga. Be to, kiekviename mazge, gali būti įdiegta pilna OS versija arba jos dalis, kuri atlieka branduolio funkcijas, o pilna OS versija veikia specialiai tam skirtame, valdančiame kompiuteryje. Procesoriai tokiose sistemose turi priėjimą tik prie savo lokalios atminties, o duomenims saugomiems kito mazgo atmintyje pasiekti yra naudojamas

pranešimų persiuntimo mechanizmas. Procesorius, kurio atmintyje yra saugomos reikalingos reikšmės išsiunčia juos pranešimų pagalba, o kitas procesorius, kuriam tie kintamieji yra reikalingi – juos priima. Toks sprendimas, kaip jau buvo minėta, leidžia išvengti problemos su konfliktiniais kreipiniais į atmintį, o tai savo ruožtu, suteikia galimybę didinti procesorių skaičių neribotai, tuo pačiu didinant sistemos našumą. Svarbiu sistemos privalumu yra galimybė be didesnių sunkumų keisti mazgų skaičių, t.y. surinkti sistemą, kuri turės norimą našumą.

Bet kaip žinia, vienų problemų sprendimas, sukelia naujas problemas, taigi šis sprendimas, irgi turi savo trūkumų. Pirmiausia tam, kad sistemą veiktų efektyviai, turi būti užtikrinta, kad pranešimų persiuntimo mechanizmas, nebūtų viso darbo sulėtėjimo priežastimi. Tai galėtų atsitikti dėl prastų komunikacinių galimybių tarp procesorių. Iš tikrųjų ne taip paprasta sujungti labai didelį kiekį procesorių. Pačiu paprasčiausiu būdu, būtų kiekviename mazge įrengti po N-1 komunikavimo kanalų, kur N yra mazgų skaičius, plius, kad jie dar būtų dviejų krypčiu, bet turbūt akivaizdu, kad tai neįmanoma. Būtent todėl superkompiuterių gamintojai, naudoja sudėtingas komunikacijų topologijas, kas leidžia sumažinti komunikavimo kanalų skaičių iki 4 – 6 viename mazge. Tokiu būdu mazgai, kurie nėra tiesioginiai kaimynai, persiunčia informaciją, per tarpinius mazgus. Be to, ne taip seniai pradėti naudoti itin greiti komutatoriai, kurie užtikrina bendravimą tarp visų mazgų, nedalyvaujant tarpiniams mazgams.

Techninės problemos, tai ne vienintelės problemos, kurias galima aptikti nagrinėjant MPP architektūra. Kuriant programas MPP sistemoms, būtina atsižvelgti ir į jų topologiją, ir specialiomis priemonėmis paskirstyti skaičiavimus tarp procesorių, kas lemia, kad žymiai sudėtingėja efektyvių programų kūrimas. Be to, iš karto galima pastebėti, kad jei programa yra „pririšama“ prie tam tikros architektūros topologijos, naudojamų platformų, komunikacijos priemonių, atsiranda problema, su tokių programų „pernešimų“ ant kitų platformų.

1.4.Klasterinės sistemos

Klasterinės sistemos (toliau klasteriai) tapo savotišku MPP sistemų patobulinimų. Jei MPP architektūroje mazgus sudaro užbaigta skaičiavimo sistema, tai klasteriu atveju mazgai yra atskiri kompiuteriai. Kompiuterinių tinklų nuolatinis tobulėjimas bei specialios programinės įrangos (MPI), leidžiančios persiųsti pranešimus, tinkliniais protokolais, sukūrimas leidžia sukurti daugiaprocesorinę sistemą su paskirstyta atmintimi be didesnių išlaidų ir pastangų, sujungiant pvz.: vienos įstaigos arba klasės kompiuterius.

Vienas iš didžiausių klasterių ypatumų yra galimybė sujungti į vieną sistemą įvairaus tipo kompiuterius, pradedant nuo asmeninių ir baigiant superkompiuteriais. Taip surinkta sistema, pagal našumą atitinka keliasdešimt kartu brangesnius gamyklinius superkompiuterius. Aišku ekvivalenčiomis tas sistemas pavadinti negalima, nes kaip jau buvo minėta, klasteryje labai daug priklauso nuo komunikavimo sąsajos. Jei yra naudojamas, pavyzdžiui, Fast Ethernet tinklas maksimalus duomenų kiekis, kuris gali būti perduotas tarp mazgų per 1 s yra 10 Mb, palyginimui per tą pati laiko tarpą superkompiuteris Cray T3D perduoda 480 Mb.

Dėl aukščiau aprašytų priežasčių prieš pradedami spręsti kokį nors uždavinį ir tuo labiau programuojant jį, verta panagrinėti kokią daugiaprocesorinę sistemą reikėtų pasirinkti ir ar galima bus išnaudoti visas jos skaičiavimo galimybes. ScaLAPACK paketo, kuris yra pritaikytas tiesinės algebros uždaviniams spręsti, kūrėjai pateikia tokius reikalavimus daugiaprocesorinėms sistemoms: „Duomenų mainų tarp dviejų skirtingų mazgų, kuri yra matuojama Mb/s, turi būti ne mažesnė negu 1/10 maksimalaus vieno mazgo našumo matuojamo Mflops“. Taigi, jei turime mazgus, kuriuose maksimalus našumas yra 1000 MHz, tai Fast Ethernet tinklas galės palaikyti tik 1/10 šio greičio, o tai reiškia, kad ne kiekvienas uždavinys galės būti išspręstas išnaudojant maksimalų daugiaprocesorinės sistemos našumą.

Nepaisant visų trūkumų klasterinės sistemos, kurios buvo pritaikytos lygiagrečių algoritmų realizavimui greitai tapo pigesne gamyklinių superkompiuteriu alternatyva. Lygiagretus skaičiavimas buvo pradėtas naudoti sudėtingose mokslinėse ir inžinerinėse programose.

Autorių labiausiai sudomino šį paskutinė lygiagrečių sistemų rūšis ir jis nusprendė savo tyrimams panaudoti būtent tokią architektūrą. Autoriui buvo prieinamos dvi šio tipo sistemos tai buvo Vilniaus universiteto matematikos ir informatikos fakulteto klasteris bei BalticGrid tinklas, kuris sujungė keletą klasterių į vieną didžiulį tinklą ir leido vienoje vietoje surinkti apie tūkstantį procesorių. Tolimesniuose darbo etapuose autorius numatė atlikti tyrimus, kaip su viena taip ir su kita sistema, nors daugiausiai dėmesio žadama skirti BalticGrid tinklui, todėl žemiau yra pateikta detalesnė šio tinklo informaciją.

1.5.BalticGrid

Kaip jau buvo minėta prieš tai buvusiame skyrelyje, BalticGrid yra MPP architektūros rūšis, kurią būtų galima apibrėžti, kaip servisą, leidžiantį sujungti į vieną tinklą daugybę kompiuterių, kurie „dalijasi“ procesoriais, atmintimi bei kitais resursais per internetą [Bal05]. Kadangi Gridas tai tinklas sujungiantis daugybę kompiuterių ir sudarantis didžiulį virtualų kompiuterį, kuris sugeba

paskirstyti procesų vykdymą lygiagrečiai tarp kelių mazgų, tai jis gali būti panaudotas itin sudėtingiems skaičiavimams atlikti. BalticGridas leidžia atlikti skaičiavimus su dideliais duomenų kiekiais dviem būdais:

- padalinant juos į kelis uždavinukus su mažesniais duomenų kiekiais arba
- atliekant daugiau skaičiavimo vienu metu, negu tai būtų galima padaryti ant vieno kompiuterio, modeliuojant lygiagrečius skaičiavimus tarp mazgų.

Grido architektūra dažniausiai sudaro žemiau išvardinti komponentai, kurie užtikrina pagrindines grido funkcijas:

- WMS (ang. Workload Management System) – servisas leidžiantis vartotojui persiuntinėti uždavinius į gridą bei juos valdyti.
- DMS (ang. Data Management System) – servisas, kuris rūpinasi duomenų apsikeitimu tarp grido ir vartotojo, čia įeina ir duomenų persiuntimas į gridą ir gavimas iš jo, taip duomenų saugojimas pačioje sistemoje.
- MS (ang. Monitoring System) – servisas, kuris leidžia vykdytojui tikrinti jam priklausančių uždavinių statusus bei susijusią informaciją.
- IS (ang. Information System) – teikia informaciją, apie prieinamus, šiuo momentu, resursus gride.
- AAS (ang. Authentification and Authorization System) – servisas, užtikrinantis Grid sistemos saugumą [Bal05].

Tiek trumpai, apie planuojama panaudoti sistemą, detaliau jos galimybės bus išnagrinėtos, kitose darbo skyriuose.

2. Tiesinės algebros uždaviniai

Kaip jau buvo minėta, kai kalbama apie tiesinės algebros uždavinius, dažniausiai galvojama apie tiesinių lygčių sistemų sprendimą. Maža to, kad šis uždavinių tipas, savaime yra pats svarbus mokslinėse tyrimuose, tai jis dar yra ir daugelio kitų užduočių bei algoritmų sudedamoji dalis, todėl šio tipo uždavinių svarbą yra neginčijama. Be to, tai paaiškina ir nuolatinius bei nesibaigiančius mokslinius tyrimus ir tobulinimus šioje srityje.

2.1. Tiesinių lygčių sistemų sprendimo algoritmai

Dabar yra žinoma daugybė algoritmų Ax = b uždaviniams spręsti. Yra žinoma daug įvairių tokių algoritmų klasifikavimo būdų, jie būna suskirstomi pagal darbo principą, pagal duomenis su kuriais

dirba ir t.t. Jei uždaviniams su tankiai užpildytomis matricomis galima taikyti vienus, tai išretintoms matricoms turi būti taikomi visai kitokie algoritmai, kurie leistų maksimaliai išnaudoti nulinius matricų elementus.

Pirmu algoritmu, kuris bus išnagrinėtas detaliau yra algoritmas, kuris labai gerai tinka tankioms matricoms ir tai yra Gausso eliminavimo algoritmas, kurį dar galima pavadinti matricos *LU* dekompozicija. Šio algoritmo tikslas yra padalinti pagrindinę matricą į dvi dalis trikampę apatinę *L* matricą, kurios įstrižainėje būtų visi vienetai bei trikampę viršutinę matricą *U* tokias, kad būtų teisinga sąlyga $A = LU$. Taip suskaidant matricą, galima iš esmės supaprastinti pradinį uždavinį, padalinant jį į keletą paprastesnių. Tai leidžia išvengti nekorektiškų sistemų sprendimo būdų, pavyzdžiui panaudojant Cramer‘io formules arba apskaičiuojant matricą A^{-1}, o po to sprendžiant sistemą, pagal formulę $x = A^{-1}b$.

Panaudojant LU dekompozicios algoritmą bus atlikti tokie veiksmai:

- Surastas išskaidymas A = LU;
- Išspręsta sistema Ly = b;
- Išspręsta sistema Ux = y.

Kadangi kiekvienas iš šių žingsnių yra nesunkiai užprogramuojamas, tai leidžia sukurti algoritmą, kuris yra pakankamai universalus ir leidžia atlikti skaičiavimus su kvadratinėmis matricomis, toks algoritmas efektyviausiai dirba su nelabai didelėmis matricomis iki kelių tūkstančių eilučių.

Tačiau labai dažnai realiuose taikymuose dirbama su matricomis, kurios yra išretintos, o tai reiškia, kad dauguma tos matricos elementų yra lygūs 0. Lygčių sistemos su tokiomis matricos, be abejo, gali būti skaičiuojami ir įprastų algoritmų pagalba, bet norint efektyviau spręsti tokio tipo sistemas būtina atsižvelgti ir išnaudoti tokios sistemos ypatumus. Tokių matricų naudojimas leidžia sukurti greitesnius sistemų sprendimų algoritmus, negu jų analogai, pritaikyti darbui su tankiomis matricomis. Realiuose taikymuose tokių uždavinių sprendimas atsiranda daugybėje sričių pradedant orų prognozių ir baigiant kosmetinių priemonių prasiskverbimų į oda modeliavimu. Čia tampa aišku, kodėl tokiuose modeliuose dažniausiai figūruoja išretintos matricos, taip atsitinka dėl to, kad visi sistemos nežinomieji nėra glaudžiai susiję tarpusavyje, o jų sąryšis atsiranda tik tam tikrose grupėse. Kitame skyriuje autorius detaliau išnagrinės išretintų matricų tipus bei pristatys algoritmus, kurie efektyviai dirba su tokiomis matricomis, tačiau savo tolimesniuose tyrimuose autorius naudos ir vieno ir kito tipo matricas, todėl jis privalo pasirinkti algoritmus, kurie gali dirbti, kaip su vienais taip ir su kitais duomenų tipais.

2.2. Išretintos matricos

Yra dvi pagrindinės išretintų matricų rūšys: struktūrizuotos ir nestruktūrizuotos. Struktūrizuotos matricos, tai tokios, kurių ne nulinės reikšmės sudaro, sistemingai išdėstytas struktūras. Dažniausiai tai būna ne didelis įstrižainių skaičius. Nestruktūrizuotose išretintose matricose ne nulinės reikšmės yra išmėtytos chaotiškai. Skirtumai tarp šių dviejų tipų nėra esminiai, jei kalbame apie tiesinės algebros uždavinius, kurie bus sprendžiami panaudojant nuosekliuosius algoritmus, bet situacija yra visiškai skirtinga, jei kalbame apie iteracinius metodus ir lygiagrečiųjų algoritmų panaudojimą. Šiuo atveju skaičiavimo greitis gali labai svyruoti, priklausomai nuo duomenų struktūrų ir kaip jos yra išdėstytos atmintyje.

Nagrinėjant išretintų matricų saugojimą galima pastebėti, jog tam, kad darbas su tokiomis matricomis būtų efektyvus, reikia turėti galimybė saugoti tik jos nenulines reikšmes. Tuo pačiu turi likti galimybė atlikti skaičiavimo operacijas, kaip ir su paprastomis matricomis.

Viena iš paprastesnių išretintų matricų saugojimo schemų yra taip vadinamas saugojimas koordinačių formatu. Tokia duomenų struktūra susideda iš trijų masyvų: pirmame masyve yra saugomos realios arba kompleksinės nenulinės matricos reikšmės, antrame ir trečiame masyve saugoma tos reikšmės pozicija, t.y. viename masyve saugomas eilutės, o kitame masyve yra saugomas stulpelio numeris. Bet toks metodas nėra pats geriausias. Galima pastebėti, kad jei surūšiuosime šiuos masyvus pagal eilutes arba pagal stulpelius, tai pamatysime, kad yra saugoma perteklinė informacija ir jos kiekį galima būtų sumažinti. Tam galima pasinaudoti rodyklių pagalba, kurios nurodytų, kur pirmame masyve yra naujos eilutės pradžią. Tokie saugojimo metodai, dar vadinami „Suspaustos išretintos eilutės“ arba „Suspausto išretinto stulpelio“ metodais ir yra vieni populiariausių išretintų matricų saugojimo schemų. Tokį populiarumą lėmė tai, kad su matricomis, kurios yra saugomos, panaudojant tokias schemas yra paprasčiau atlikti tipinius skaičiavimus. Iš kitos pusės koordinačių metodo panaudojimas yra labai paprastas ir lankstus, todėl jis yra dažnai naudojamas, kaip pagrindinis formatas, kuriant programinius paketus, pavyzdžiui Harwell‘o bibliotekoje.

Neretai yra naudojami ir tam tikri šių schemų patobulinimai, kai kurie iš jų remiasi tuo, kad įstrižainėse visos esančios reikšmės yra nenulinės arba/ir jos yra naudojamos dažniau negu visi kiti matricos elementai. Būtent todėl, tokias reikšmes apsimoka saugoti atskirai. Tokia saugojimo schema yra dar vadinama „Modifikuota išretinta eilutė“. Šiuo atveju yra naudojami tik du masyvai, pirmo masyvo pradžioje yra saugomos reikšmės, kurios yra išdėstytos matricos įstrižainėje, po jų eina visi kiti elementai, t.y. tie elementai, kurie buvo išdėstyti žemiau arba aukščiau įstrižainės.

Antrame masyve yra saugomos rodyklės į pirmą masyvą, kurios parodo, nuo kurio elemento prasideda nauja eilutė.

Kalbant apie įstrižai struktūrizuotas matricas, verta paminėti, kad reikšmes išdėstytas įstrižainėse galima vaizduoti ir stačiakampio masyvo pagalba. Šiam metodui realizuoti yra naudojami du masyvai: pirmame yra saugomos įstrižainių reikšmės, trumpesnėms įstrižainiams vietoj trūkstamų reikšmių pridedame kokį nors simbolį, antrame masyve yra saugomas postūmis, kuris priklausomai nuo elemento numerio rodys, kurioje įstrižainėje yra talpinamos išsaugotos reikšmės.

2.3.Lygiagretieji algoritmai skirti darbui su išretintomis matricomis

Taip pat analizuojant išretintas matricas verta paminėti algoritmus, skirtus atlikti skaičiavimus su jais, todėl žemiau bus pristatyti algoritmai, kurie tinka darbui su tokiomis matricomis ir sugeba išnaudoti matricų nulinius elementus tam, kad galima būtų sumažinti skaičiavimų laiką.

Išretintų matricų daugyba iš vektoriaus yra labai pigi (sąnaudos yra proporcingos nenulinių elementų matricoje skaičiui) todėl, kada mus labiausiai domina skaičiavimo sąnaudų sumažinimas verta paminėti iteracinius algoritmus. Paprasčiausi iteraciniai metodai, jei kalbame apie analizę ir įgyvendinimą, panašiai kaip ir Gausso eliminavimo algoritmas, yra pagrįsti matricos suskaidymu, į dvi dalis. Skirtumas yra tame, kad matrica suskaidoma į dalį M – dalis kuriai lengvai galima surasti atvirkštinę matricą, bei likusią dalį Z. Taigi mums gerai žinoma lygtį Ax = b galima užrašyti taip:

$Mx = Zx + b$, kur $Z = M - A$ arba

$x = M^{-1}(Zx + b)$, šitai formulei dabar galima taikyti paprastą Banach'o iteraciją:

$x_{k+1} = M^{-1}(Zx_k + b)$.

Tokie metodai yra vadinami stacionariais iteraciniais metodais. Tam, kad galima būtų ištirti tokio metodo konvergavimą, reikėtų išanalizuoti bendresnį atvejį. Tarkime turime matricą B ir vektorių c (Stacionariam iteraciniam metodui: $B = M^{-1}Z$ bei $c = M^{-1}b$), tokius kad:

$x_k = Bx_{k-1} + c$.

Tada $x_k - x^* = B^k(x_0 - x^*)$, iš čia ir nelygybės $||B^k|| \leq ||B||^k$, turime

$||x_k - x^*|| \leq ||B||^k \, ||x_0 - x^*||$.

Iš čia turime, kad pakankama sąlyga iteracijų konvergavimui yra $||B|| < 1$, o pakankama ir būtina sąlyga, bet kokiam pradiniam vektoriui x0 yra:

$\rho = \max\{|\lambda| : \lambda$ yra matricos B tikrinė reikšmė$\} < 1$.

Šios iteracinių metodų grupės labiausiai žinomais ir plačiai naudojamais yra šie du metodai: Jacobi ir Gausso-Seidelio metodai.

2.3.1. Jacobi metodas

Paimkime matricą M tokią, kad M = diag(A), kur diag(A) yra diagonali matrica susidedanti iš elementų esančių matricos A pagrindinėje įstrižainėje. Daroma prielaida, kad visi elementai esantys šioje įstrižainėje yra nenuliniai.

Tada Ax = b galima užrašyti taip:

Mx = Zx + b, iš šios sąlygos sudarome iteracinį metodą, vadinama Jacobi metodu:

$x_k = Bx_{k-1} + c$, kur B = $-M^{-1}Z$ ir c = $M^{-1}b$.

Šį metodą aprašydami matricos koordinačių pagalba, gauname tokią lygtį:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)} \right),$$

Iteracijos numeris yra žymimas kaip viršutinis indeksas.

2.3.2. Gausso – Seidelo metodas

Šio metodo esminis tikslas yra patobulinti Jacobi metodą taip, kad kiekvienoje iteracijoje naudoti tikslesnes xj(k) reikšmes, kai j < i, tai yra patobulindami formulę

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)} \right),$$

gauname formulę:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)} \right).$$

Tai yra patobuliname uždavinį Ax = b, taip kad gauname:

A = M – Z ir $x_{k+1} = M^{-1}(Zx_k + b)$, kur M – apatinis matricos A trikampis.

Verta pastebėti, kad jei pradinė matrica A yra įstrižai dominuojanti matrica, šis metodas leidžia išspręsti sistemą esant bet kokiam startiniam x0. Be to, daugumoje atveju, šis metodas yra greičiau konverguojantis negu Jacobi metodas.

3. Programinė įranga skirta tiesinės algebros uždavinių sprendimui

Šioje dalyje autorius trumpai pristatys bibliotekas, kurias teko analizuoti ir išbandyti. Kai kurios iš jų bus panaudotos praktinėje dalyje reikalingiems tyrimams atlikti. Taigi autoriui teko susipažinti su tokiomis bibliotekomis [Net05]:

- BLAS (ang. Basic Linear Algebra Subprograms) – šioje bibliotekoje yra sukauptos paprogramės, kurios yra skirtos pagrindinėms operacijoms su matricomis be vektoriais atlikti. Jos struktūra yra tokia: 1 lygio BLAS algoritmai atlieka skaičiaus ir vektoriaus bei vektoriaus ir vektoriaus operacijas, 2 lygio BLAS algoritmai atlieka matricos ir vektoriaus operacijas ir paskutinio 3 lygio BLAS algoritmai atlieka standartines operacijas su matricomis. Kadangi ši biblioteka yra labai efektyvi ir paplitusi yra sudaro, kai kurių kitų bibliotekų pagrindus, taip pat ir žemiau aprašomos lapack bibliotekos.
- Lapack (ang. Linear Algebra PACKage) – viena populiariausių pasaulyje tiesinė algebros bibliotekų, parašyta Fortan77 kalba. Skirta įvairaus tipo uždavinių sprendimui, pradedant nuo tiesinių lygčių sistemos sprendimų, panaudojant įvairias faktorizacijas (pvz.: LU, Cholesky, SVD ir t.t.) ir užbaigiant tikrinių reikšmių paieška.
- ATLAS (ang. Automatically Tuned Linear Algebra Software) – paketas, kuris iš tikrųjų yra dviejų aukščiau paminėtų bibliotekų optimizuota versiją. Diegiant šį paketą sistemoje yra atliekama visą aibė testų, kurie leidžia maksimaliai optimizuoti ir pritaikyti algoritmus darbui konkrečioje sistemoje.
- ScaLapack – Atsiradus daugiaprocesorinėms sistemoms su paskirstyta atmintimi prireikė bibliotekos, kuri galėtų panaudoti visus LAPACK bibliotekos teikiamus privalumus ir pritaikyt jas darbui su naujomis platformomis. Tokia biblioteka buvo sukurta ir tai yra ScaLAPACK paketas, kuris tapo programinės įrangos, pritaikytos darbui ant daugiaprocesorinių architektūrų, standartu. Tokį didelį projekto populiarumą lėmė tai, kad šios bibliotekos pagrindu tapo labai populiarios LAPACK bei BLAS bibliotekos.
- Cerfacs – mažiau žinoma biblioteka, naudojanti lygiagrečius algoritmus CG ir GMRES, kurie leidžia spręsti tiesinių lygčių sistemas iteraciniais metodais. Šiuos algoritmus geriausia naudoti su išretintomis matricomis.

Savo tyrimuose autorius nori atkreipti dėmesį į nuosekliuosius skaičiavimo algoritmus tam, kad tolimesnėse tyrimuose pasinaudoti BalticGrido teikiamais privalumais ir jo, o ne MPI sąsajos pagalba, išlygiagretinti sukonstruotų uždavinių skaičiavimą. Tyrimuose bus panaudotos lapack,

optimizuota jos versija atlas bei blas bibliotekos. Kitos minėtos bibliotekos buvo panaudotos susipažinimui su egzistuojančiomis sprendimo alternatyvomis ir gauti rezultatai nebus įtraukti į šį darbą. Bibliotekos, kurios buvo panaudotos praktiniuose bandymuose, bus išanalizuotos dar papildomai kitose darbo dalyse.

4. Analitinė dalis

Šioje darbo dalyje autorius pristatys esmines problemas, kurios turi būti išspręstos šio darbo kontekste. Toliau bus pateikta šių problemų galimų sprendinių analizė. Labiausiai tinkamas sprendimas bus papildomai tiriamas praktinėje darbo dalyje.

Esminė šio darbo problema yra tiesinės algebros uždavinių sprendimų pagreitinimas, toliau sekantys argumentai patvirtins šį teiginį. Norėdami sumodeliuoti tam tikrą realią situaciją, dažniausiai reikia atlikti labai didelį kiekį įvairių skaičiavimų su skirtingais duomenimis. Kuo sudėtingesnis yra modelis tuo daugiau yra duomenų ir tuo pačiu būtina atlikti daugiau skaičiavimų. Suradus sistemos sprendinį, neužteks vien tik apskaičiuotos reikšmės, papildomai reikės mažiausiai paklaidų bei netikties skaičiavimų. Modelio tyrimai būtų dar labiau priartinti prie realios situacijos, jei pavyktų įvertinti sprendinio stabilumą, t.y. įvertinti, kaip stipriai gali būti paveiktas sistemos sprendinys, jei į pradinius duomenis bus įneštas tam tikras triukšmas. Įvertinus kiek daug skaičiavimų reikės atlikti nesunku pastebėti, kad reikia priemonių, kurios leistų pagreitinti uždavinių sprendimą.

4.1. Sprendžiamų uždavinių pagreitinimas

Galima išskirti kelis būdus, kurių pagalba galima pagreitinti uždavinio sprendinių paiešką:

- Uždavinio sprendimui panaudoti galingesnį kompiuterį, turintį galingesnį procesorių, daugiau operatyvios atminties arba greitesnę bendravimo magistralę;
- Perrašyti programas taip, kad jos efektyviau išnaudotų turimus kompiuterio resursus;
- Išlygiagretinti turimas programas, t.y. paskirti skaičiavimo darbus tarp kelių procesorių, panaudojant komunikavimo sąsajas, pavyzdžiui MPI;
- Panaudojant specializuotas priemones, paskirstyti skaičiavimus tarp kelių kompiuterių, neperrašant nuosekliųjų programos versijų.

Atsižvelgiant į darbo tematiką autorių labiausiai sudomino trečias bei ketvirtas punktai, kurie ir bus išanalizuoti detaliau, po ko vienas iš jų bus parinktas ir išbandytas praktinėje darbo dalyje.

4.2. Individualios programos išlygiagretinimas

4.2.1. Programų išlygiagretinimas pritaikant jas darbui su MPP sistemomis

Kalbant apie lygiagrečiųjų programų kūrimą, būtinai reikia atsižvelgti ne tik į tai, kad parašyti efektyviai veikiančią programą yra labai sunku, bet ir į tai kokio rezultato tikimasi pasiekti, atlikus šį darbą. Juk labai dažnai nesudėtingos, blogai išlygiagretintos arba netinkamai paleistos (pavyzdžiui parinkus neoptimalų procesorių skaičių) programos, veiks lėčiau negu jų tiesinė realizaciją (žr. įrodymą praktinėje darbo dalyje). Tai patvirtiną ir bandymai atliekami su superkompiuteriais. Atliekant tiesinės algebros uždavinių sprendimą, pastebimas skirtumas, tarp tiesinės ir lygiagrečios programos realizacijos našumo, atsiranda, kai nežinomųjų skaičius viršija šimtą nežinomųjų. Tuo tarpu maksimaliai efektyviai išnaudoti superkompiuterio skaičiavimo galimybes kartais galima, jei uždavinyje yra tūkstantis arba net daugiau nežinomųjų. Situacija, kai skaičiavimams atlikti yra naudojamas klasteris, dėl prasto komunikavimo tarp mazgų, gali dar pablogėti [BDZ03].

ScaLAPACK paketo kūrėjai, pateikia formulę, kurios pagalba galima išskaičiuoti rekomenduojama procesorių skaičių, kuris gali būti panaudotas tiesinės algebros uždaviniams spręsti, maksimaliai išnaudojant daugiagprocesorinės sistemos našumą [Sca97]:

$$P = (M \times N)\, 10^6, \text{ kur } M \times N - \text{matricos dydis}$$

Remiantis formulę gauname, kad kiekvienam mazgui reikėtų paskirti apdoroti matricos bloką, kurio dydis būtų apie 1000 x 1000 dydžio. Nepaisant to, kad ši formulė yra tik rekomendacinio pobūdžio, ji gerai iliustruoja, kokio sudėtingumo uždaviniams spręsti buvo kuriamas ir optimizuojamas ScaLAPACK paketas. Išlygiagretinimo efektyvumo augimą, didinant sprendžiamų uždavinių sudėtingumą, lemia tai, kad padidinus tiesinės algebros uždavinių sudėtingumą, skaičiavimų kiekis išauga proporcingai n^3, o apsikeitimo duomenimis tarp mazgų padidėjimas yra proporcingas n^2, kas be abejo įtakoja silpniausios MPP sistemos – komunikavimo sąsajos, mažesnį apkrovimą.

Galima išskirti du lygiagretaus programavimo modelius, skirtus MPP sistemoms (pagal Flynn‘o klasifikacija [Fly72]):

- Viena programa ir daug duomenų (SPMD – Single Program Multiple Data) – visose procesoriuose vykdomos vienos programos kopijos su skirtingais duomenų blokais;
- Daug programų ir daug duomenų (MPMD - Multiple Program Multiple Date) – procesoriuose vykdomos skirtingos programos, dirbančios su skirtingais duomenimis.

MPMD modelis yra dar kitaip vadinamas funkcinių išlygiagretinimų ir yra naudojamas dažniausiai vaizdo informacijos apdorojimui. Jam yra būdingas konvejeriaus sudarymas, kur kiekvienas sekantis apdorojimo etapas yra vykdomas atskirame procesoriuje. Tačiau šis modelis nėra paplitęs tiesinės algebros skaičiavimo uždaviniuose, kadangi yra sunku sudaryti pakankamai ilgą konvejerių ir maksimaliai efektyviai apkrauti kiekvieną iš mazgų.

Tiesinės algebros uždaviniams daugiaprocesorinėse sistemose su paskirstyta atmintimi spręsti, dažniausiai yra naudojamas SPMD programavimo modelis. Šio modelio pagrindinis principas yra – keliuose procesoriuose užkrauti tos pačios programos kopija, kuri dirbs su tik jai skirtais duomenimis, kitais žodžiais tariant, lygiagreti programa suskaido pagrindinį uždavinį į N mažesnių uždavinių – použdavinių, iš kurių kiekvienas turi būti vykdomas atskirame mazge. Schematiškai tokį modelį galima būtų pavaizduoti taip:

```
IF (proc_ident = 0) {
    RUN uzdavinys1;
}
IF (proc_ident = 1) {
    RUN uzdavinys2;
}
...
IF (proc_ident = N) {
    RUN uzdavinysN;
}
rezultatas = apjungti(rezultatas1, rezultatas2, ..., rezultatasN)
```

Proc_ident žymi procesoriaus identifikatorių, o funkcija apjungti – suformuoja rezultatą, remiantis atskirų mazgų lokaliais rezultatais. Šiuo atveju tos pačios programos versija bus vykdoma kiekviename iš N mazgų, bet atliks tik jam skirtą použdavinį. Jeigu uždavinio struktūra nereikalauja didelio duomenų apsikeitimo tarp mazgų, o procesorių apkrovimas daugmaž vienodas, tai galima tikėtis, kad uždavinio sprendimas padidės N kartų. Tokį rezultatą pasiekti yra pratiškai neįmanoma.

Kiekviename modelyje použdavinys gali būti suprantamas kitaip, kai MPMD modelyje použdavinys yra funkciškai atskirta programos dalis, tai SPMD modelyje použdavinys – programos darbas su atskiru duomenų bloku. Dažniausiai išlygiagretinimas yra taikomas ciklams, tokiu atveju použdaviniai yra to ciklo veiksmai atliekami su tam tikromis ciklo reikšmėmis. Išnagrinėkime paprasčiausią pavyzdį, kuriame yra išskaičiuojami vektoriaus B elementai:

```
FOR (i = 0; i < 2000; i++) {
    B(i) = B(i) + A(i);
}
```

Iš šio ciklo galima sudaryti 2000 atskirų použdavinių iš kurių kiekvienas praktiškai galėtų būti vykdomas atskirame procesoriuje. Tarkime turime daugiaprocesorinę sistemą, kurį susideda iš 20 procesorių, tada iš šio ciklo galime sudaryti 20 použdavinių, kur kiekviename iš jų bus apskaičiuojami 100 vektoriaus B elementų. Be to, reikia apsispręst, kur bus saugomi šie masyvai procesoriaus atmintyje, yra du galimi variantai:

- Visi masyvai yra saugomi kiekviename procesoriuje, taigi ciklo išlygiagretinimui dirbti tik su elementų intervalu, kurį turės suskaičiuoti kiekvienas iš procesorių. Taigi kiekvienas procesorius turi viso masyvo kopija, bet skaičiuoja ir keičia tik jam priskirtą intervalą elementų. Programos pabaigoje reikia turėti mechanizmą, kuris surinks, visus modifikuotus elementus ir sujungs juos į vieną masyvą.
- Kiekviename iš procesorių yra saugomas na visas masyvas, o jo dalis, tiksliau 1/N pagrindinio masyvo dalis. Šiam metodui reikia turėti mechanizmą, kuris leistų lokalųjį masyvą, indeksų pagalba, patalpinti į bendrą masyvą. Jei kuriam nors procesoriui, prireiks elemento, kuris yra kito procesoriaus atmintyje, reikės pasinaudoti pranešimų persiuntimo mechanizmu.

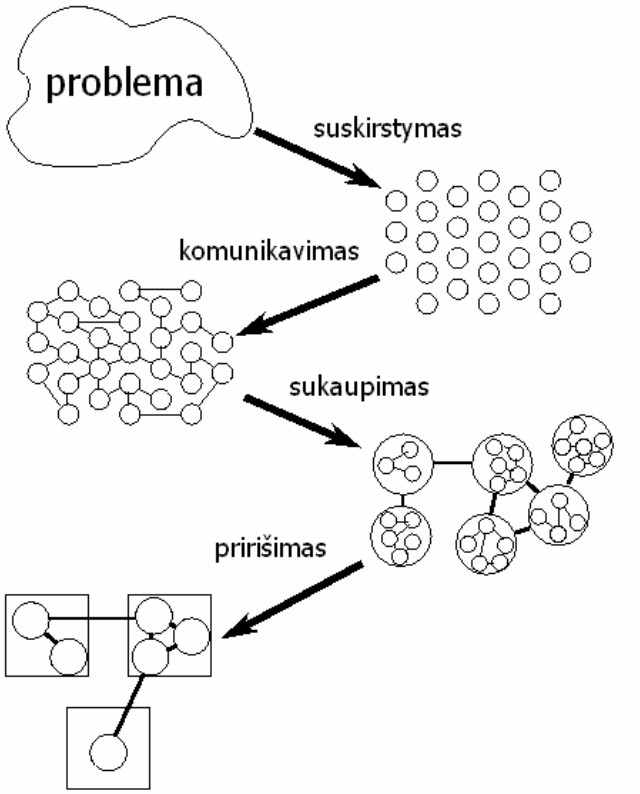
Nuo varianto, kuris bus pasirinktas labai stipriai priklauso busimos lygiagrečios programos efektyvumas. Pirmu atveju, galima žymiai sumažinti komunikavimo sąsajos panaudojimą, bet užtat šis variantas neleidžia spręsti didžiulių ir sudėtingų uždavinių, be to gali iškilti problemos sinchronizuojant kiekvieno procesoriaus turimą masyvo kopija. Antras variantas leidžia spręsti žymiai sudėtingesnius uždavinius, bet tada labai opi tampa duomenų persiuntimo tarp mazgų problemą.

Aukščiau aprašytas pavyzdys, gerai iliustruoja lygiagrečių algoritmų kūrimo metodologinę schemą, sukurtą Ian‘o Fosterio. Autorius išskaido algoritmų kūrimo etapą į keturias stadijas [Fos95]:

1. Užduoties suskirstymas arba suskaidymas į minimalias nepriklausomas užduotis. Praktiniai dalykai, tokie kaip procesorių skaičius sistemoje, šioje stadijoje mūsų nedomina.
2. Užtikrinamas komunikavimas tarp suskaidytų užduočių, t.y. parenkamos komunikavimo struktūros bei algoritmai.

3. Mažesnių užduočių sujungimas į didesnes tam, kad galima būtų padidinti efektyvumą ir sumažinti komunikavimą tarp mazgų.
4. Sugrupuotų užduočių paskirstymas tarp procesorių, maksimaliai apkraunant kiekvieną iš jų atskirai ir tuo pačiu maksimaliai išnaudojant sistemos našumą. Tai gali būti padaryta statiškai arba panaudojant balansavimo algoritmus.

Visi šie žingsniai yra pavaizduoti žemiau pateiktoje schemoje:



3 pav. Lygiagrečių algoritmų kūrimo metodologija

Taipogi verta būtų paminėti ir apie minimalius reikalavimus, kurie yra keliami daugiaprocesorinėms sistemos su paskirstyta atmintimi tam, kad jas būtų galima naudoti lygiagrečių algoritmų vykdymui. Minimalių reikalavimų rinkinys yra:

1. Procesoriai sistemoje privalo turėti unikalius identifikatorius.
2. Turi būti galimybė procesoriui identifikuoti save patį.
3. Turi būti realizuotas bendravimas tarp mazgų, t.y. vienas procesorius turi išsiųsti pranešimą, o kitas procesorius turi jį gauti.

Pranešimų perdavimą inicijuoja procesorius, kuriam reikia perduoti pranešimą, procesorius – adresatas gali tik priimti jam siųstus pranešimus.

4.2.2. Lygiagrečių programų efektyvumo tyrimas

Idealiu atveju išlygiagretinta programa, vykdoma panaudojant N procesorių, turėtų pradėti veikti N kartų greičiau negu šios programos tiesinė realizacija arba per tą patį laiko intervalą apdoroti N kartų daugiau duomenų. Deja, bet tokio efektyvumo praktiškai neįmanoma pasiekti, tai patvirtina ir Amdahlio dėsnis [Amd67]:

$$S \le \frac{1}{f + \frac{(1-f)}{N}}$$

Čia *S* pagreitinimas, kurį galima pasiekti po programos išlygiagretinimo, *N* – procesorių skaičius, *f* – tiesinio kodo dalis programoje. Šis dėsnis galioja sistemoms ir su bendra atmintimi ir su paskirstyta atmintimi.

Remiantis šia formule, nesunku pastebėti, kad tam, kad programos darbas pagreitėtų N kartų, reikšmė *f* turi būti lygi 0, o tai reikštų pilną tiesiškai vykdomo kodo pašalinimą iš programos, ką praktiškai pasiekti yra neįmanoma.

4.3.Bendro uždavinių paketo išlygiagretinimas

Autorių labiausiai sudomino tokio tipo uždavinių sprendinio paieškos pagreitinimas, ne tik parenkant algoritmą, kuris tam tikro dydžio arba tankumo matricai sugeba apskaičiuoti atsakymo vektorių greičiau bei tiksliau nei kiti, bet šiek tiek platesne prasme. Turima omeny, kad šiandien problemos dažniausiai kyla ne sprendžiant kažkokį vieną, net ir labai sudėtingą tiesinių lygčių sistemą (šiandien prieinamos skaičiavimo technikos pagalba, tai galima padaryti pakankamai greitai), bet dėl visai kitos priežasties. Kaip jau buvo minėta aukščiau, vykdydami tam tikrą projektą arba spręsdami tam tikros srities modeliavimo uždavinį, gali dažnai kyla poreikis atlikti visą aibę panašių skaičiavimų, turint daugybę skirtingų duomenų. Natūralu, kad atsiranda poreikis atlikti tai patogiai bei panaudojant sakykim turimas nuoseklias programas, neperrašant jų visų ir nepritaikant pvz. MPI sąsajos. Būtent todėl, autoriaus nuomone, specialių priemonių, skirtų būtent panašių problemų sprendimui, poreikis nuolat auga. Tai patvirtina ir skaičiavimo tinklo BalticGrid atsiradimas, kuris yra nuolat plečiamas bei tobulinamas. Autoriaus nuomone, šį priemonė yra pakankamai naują ir ji buvo žymiai mažiau tiriama, negu individualios programos išlygiagretinimas. Todėl autorius nusprendė praktinėje savo darbo dalyje išbandyti BalticGrid skaičiavimo tinklą, kaip priemonę viso skaičiavimo paketo išlygiagretinimui.

5. Praktinė dalis

5.1. Skaičiavimo priemonės

Šią tiriamojo darbo dalį autorius nusprendė pradėti nuo keleto paprastų eksperimentų atlikimo, kurie turėtų paremti daromą prielaidą, kad lygiagrečiai atliekami skaičiavimai žymiai pagreitina užduočių vykdymą, bet tik tuo atveju, jei buvo atliktas efektyvus programos išlygiagretimas arba buvo atliekamas paleidimas tinkamai parinktoje sistemoje, t.y. tinkamai parinkus resursus programos vykdymui. Taigi, kaip jau buvo minėta ankstesniuose darbo dalyse, nagrinėjimams autorius pasirinko daugiaprocesorinę sistemą su paskirstyta atmintimi, o tiksliau sakant klasterinę sistemą BalticGrid. Šią sistemą sudaro klasteris esantis VU MIF fakultete bei keletas kitų klasterių iš Lietuvos bei kitų šalių, kurie buvo apjungti į vieną tinklą. Tokia architektūra buvo pasirinkta dėl kelių priežasčių. Visų pirma atlikus daugiaprocesorinių sistemų analizę, autorių labiausiai sudomino būtent ši sistema dėl savo tobulinimo bei konfigūravimo galimybių. Be to, tokį „superkompiuterį“ pigu ir nesudėtinga įrengti iš jau turimų resursų, kas lemia itin dideli jos populiarumą, kuris nuolat auga.

Darbo pradžioje buvo nuspręsta bandymus atlikti su keletu nesudėtingų programų, skirtų atlikti įvairias standartines operacijas su matricomis, pvz.: sudėtį, daugybą ir pan. Autorius parašė keletą programų C kalba, kurios ir leido atlikti keletą pradinių bandymų. Kaip pavyzdys prieduose yra pateiktos programos, kurios realizuoja dviejų matricų daugybą. Pirma programa (priedas Nr. 1) sudaugina dvi matricas panaudojant vieną procesorių (tiesinė realizacija), kita programa (priedas Nr. 2) yra to paties skaičiavimo algoritmo lygiagreti realizacija. Programa buvo išlygiagretinta remiantis dekompozicijos principu, t.y. pasinaudojant tuo, kad dauginant matricas, kiekviena matricos A eilutė gali būti sudauginta su matrica B atskirai. Galutinė matrica C buvo gaunama sujungiant gautus iš skirtingų procesorių duomenis į vieną matrica. Iš techniškos pusės išlygiagretinta programos versija veikia taip: turime vieną pagrindinį kompiuterį, kuris padalina matricą A į kelias dalis (su tam tikru eilučių skaičiumi) ir persiunčia jas, keliems kompiuteriams, iš kurių kiekvienas skaičiuoja tik savo užduoties dalį. Apskaičiavus savo eilutes, kompiuteriai persiųsdavo informaciją atgal pagrindiniam (master) kompiuteriui, kuris sujungdavo visas apskaičiuotas eilutes į vieną bendrą matricą.

Kadangi buvo panaudota MPP architektūra, o tiksliau sakant BalticGrid skaičiavimų tinklas, kuris apjungia kelis šimtus kompiuterių, tai kiekvienas iš jų dirbo tik su savo lokalia atmintimi ir tam, kad galima būtų užtikrinti lygiagretų matricos dalių suskaičiavimą bei galutinių rezultatų surinkimą į vieną matricą, buvo naudojama pranešimų persiuntimo mechanizmas – MPI sąsaja.

Atlikus eilę bandymų, autorius apskaičiavo vidutinius programų vykdymo laikus. Gauti rezultatai yra pateikti žemiau esančioje lentelėje nr 1.:

Matricų dydžiai A ir B	Tiesinė realizacija	Lygiagreti realizacija			
		2 PC	4 PC	6PC	8PC
100x100 ir 100x100	0,02	0,03	0,06	0,06	0,06
200x250 ir 250x200	0,24	0,42	0,35	0,19	0,19
300x300 ir 300x300	0,81	1,18	0,53	0,41	0,32
500x300 ir 300x500	1,51	1,99	0,95	0,64	0,60
600x600 ir 600x600	3,25	3,33	1,67	1,07	0,93
700x700 ir 700x700	5,71	5,90	2,73	1,91	1,21

Lent 1. Vidutinis programos vykdymo laikas (s), panaudotų matricų dydžiai, naudojamų procesorių skaičius.

Didelės mokslinės vertės atlikti bandymai, be abejo, neturėjo, bet jie leido susipažinti su MPI sąsajos programavimo bei paskirstytų skaičiavimų tinklo panaudojimo ypatumais. Be to, autorius užsibrėžė sau kitą tikslą – atlikti tiesinės algebros uždavinių sprendimo paketų detalią analizę ir patikrinti kiek kartų optimizuoti be specialiai, konkrečiai užduočiai pritaikyti algoritmai, pagreitina nagrinėjamus darbe skaičiavimus.

Šiandien prieinamų bibliotekų skaičius, skirtas aukščiau aprašytiems tikslams yra iš tikrųjų įspūdingas. Vien tik Tennessee Universiteto www.netlib.org svetainėje yra jų pateikta apie šimtą, todėl pasirinkimas buvo iš tikrųjų platus. Kadangi autorių domino konkrečių uždavinių sprendimas buvo išrinkti keli kriterijai pagal kuriuos ir buvo parinktos kelios įdomesnės bibliotekos. Pasirenkant paketus labiausiai dėmesys buvo atkreiptas į galimybė atlikinėti veiksmus su matricomis bei vektoriais ir aišku esminis dominantis autorių faktorius buvo tiesinių lygčių sistemų sprendimo algoritmai. Iš kitos pusės reikėjo atkreipti dėmesį ir į algoritmų įvairovę, vartotojų atsiliepimus, bibliotekų palaikymą, populiarumą bei tobulinimą. Pagal visus minėtus kriterijus iš karto išsiskyrė kelios bibliotekos: lapack, blas, atlas, cerfacs, plapack, kurias autorius ir nusprendė panaudoti savo tyrimuose. Kai kurios iš šių bibliotekų, pavyzdžiui lapack yra turbūt viena dažniausiai naudojamų bibliotekų tiesinės algebros uždavinių sprendimui (pagal netlib svetainės pateikta populiarumo statistiką [Bre07]). Tarp pasirinktų bibliotekų yra kaip bibliotekos skirtos nuosekliajam skaičiavimui, tarp jų jau minėta lapack ir atlas bibliotekos, taip ir bibliotekos skirtos lygiagretiesiems skaičiavimams cerfacs ir plapack.

Pirma patirtis su minėtomis bibliotekomis įrodė, kad nevertėtų „išradinėti dviračio“, t.y. pačiam programuoti visų reikalingų tyrimams algoritmų bei operacijų atliekamų su matricomis ir vektoriais. Jau pirmi bandymai parodė, kad algoritmai pateikti bibliotekose puikiai atlieka visą juodą darbą ir

atlieka jį žymiai greičiau negu neoptimizuotos programos. Buvo nuspręsta pačių algoritmų neprogramuoti, tuo labiau, kad tai ir nebuvo pagrindinis darbo tikslas. Tai leido autoriui visą dėmesį sutelkti į tyrimo objektą, greitesnių algoritmų bei įdomesnių išlygiagretinimo sprendimų paiešką.

Kaip pavyzdį pateikiu autoriaus rašytų programų ir panaudotos BLAS bibliotekos procedūros vykdymo laikų palyginimą. Matomas neginčijamas bibliotekinių procedūrų privalumas, ne tik jei yra panaudojama tiesinė programų realizacija, bet ir išlygiagretinta jos versija.

Matricų dydžiai A ir B	**Blas procedūros xGEMM vykdymo laikas**	**Tiesinė realizacija**	**Lygiagreti realizacija**			
			2 PC	**4 PC**	**6PC**	**8PC**
100x100 ir 100x100	0,001	0,02	0,03	0,06	0,06	0,06
200x250 ir 250x200	0,02	0,24	0,42	0,35	0,19	0,19
300x300 ir 300x300	0,04	0,81	1,18	0,53	0,41	0,32
500x300 ir 300x500	0,095	1,51	1,99	0,95	0,64	0,60
600x600 ir 600x600	0,32	3,25	3,33	1,67	1,07	0,93
700x700 ir 700x700	0,49	5,71	5,90	2,73	1,91	1,21

Lent 2. Vidutinis programos vykdymo laikas (s), panaudotų matricų dydžiai, naudojamų procesorių skaičius. Parodomas GEMM procedūros pranašumas prieš autoriaus rašytas programas.

5.2.Tyrimuose panaudoti duomenis

Kita dilema autoriui iškilo po tai, kai jis pradėjo ieškoti duomenų savo bandymams atlikti. Kadangi daugumą algoritmų dirba su duomenų masyvais, paprasčiausias būdas buvo sugeneruoti atsitiktinių skaičių sekas ir bandyti spręsti, taip sugeneruotas sistemas. Toks būdas ir buvo panaudotas pirmuose eksperimentuose, norint nustatyti vieno algoritmo pranašumą prieš kitą. Tačiau šis būdas autoriui pasirodė ne itin įdomus ir šiek tiek per paprastas, taip buvo pradėta įdomesnių duomenų paieška. Autoriui pavyko surasti matricų talpykla „Matrix Market“ [Mat00], kurioje yra patalpintos matricos, kurios yra naudojamos realiuose taikymuose, pradedant cheminių reakcijų ir užbaigiant oro prognozių modeliavimais. Tokių duomenų panaudojimas autoriui pasirodė labiau tikslingas ir jo nuomone, tai leido įnešti į atliekamus bandymus daugiau praktinės naudos. Atsitiktinių būdų buvo atrinktos keliasdešimt matricų įvairių dydžių ir įvairaus tankumo su double tikslumo realiais elementais. Šių duomenų panaudojimas leido paįvairinti atliekamus tyrimus ir padaryti juos labiau objektyvesniais.

5.3.Tyrimo metodika

Norėdami sumodeliuoti tam tikrą realią situaciją, būtina atlikti visą eilę skaičiavimų. Iš šiame darbe nagrinėjamos pozicijos, tai būtų visos aibės tiesinių lygčių Ax = b sprendimas. Kai domina tiesiog sprendinių vektorius x, sistema turi būti sprendžiama nepriklausomai nuo kitų, kitais atvejais,

po vienos sistemos išsprendimo, mums reikia ką tik surastą vektorių x, panaudoti kitoje sistemoje, kaip vektorių b ir taip sudaryti skaičiavimo grandinę. Kaip reikėtų žiūrėti į tokią problemą ir tokio uždavinio optimizavimą bei pagreitinimą? Darbo autoriaus nuomone, darbas turi būti suskaidytas į kelis etapus:

1. Pirmiausia visą dėmesį reikėtų skirti individualios tiesinių lygčių sistemos sprendimui. Tinkamų algoritmų paieška, algoritmų greičių bei tikslumų nustatymai ir palyginimai.
2. Antras etapas yra uždavinio sprendinio stabilumo tyrimas, kuriame reikėtų ištirti, kaip įnešamas triukšmas gali paveikti galutinį sistemos sprendinį. Toks tyrimas padeda nustatyti, individualios matricos priklausomybę nuo įnešamo triukšmo ir labiau priartinti atliekamus skaičiavimus prie realios situacijos.
3. Tinkamos strategijos, kuri leistų aibę atskirų uždavinių sujungti į vieną projektą ir kuo patogiau jį valdyti, sukūrimas ir pritaikymas. Strategijų palyginimas ir efektyvumo tyrimas.

5.3.1. Tiesinių lygčių sistemų sprendimas

Pirmiausia buvo lyginami įvairūs tiesinių lygčių sistemų sprendimo algoritmai iš įvairių bibliotekų tam, kad galima būtų atrinkti keletą greičiausių bei skaičiuojančių su mažiausia paklaida ir panaudoti juos tolimesniuose tyrimuose. Šiam tikslui pasiekti buvo sukurtas šablonas pagal kurį buvo galima spręsti paduodamas lygčių sistemas bei atlikinėti įvairius tyrimus, tokius kaip algoritmo paklaidų vertinimą arba triukšmo įnešimo analizę.

5.3.2. Uždavinio sukūrimas

Kaip jau buvo minėta, tyrimuose šalia atsitiktinai sugeneruotų matricų buvo paimtos matricos, kurios atsiranda realiuose taikymuose. Čia iškilo dar viena problema: dažniausiai matricų kolekcijoje buvo pateikiama tik matrica A, t.y. iš uždaviniui Ax = b reikalingų duomenų, buvo duota tik matrica A. Autoriui kilo klausimas iš kur paimti trūkstamus duomenis, t.y. vektorių b. Atsitiktinai generuoti vektorių b ir po to spręsti sistemą būtų ne labai korektiška. Toks būdas, be abejo, leistų išmatuoti skirtingų algoritmų sprendinio paieškos greitį, bet kaip išmatuoti algoritmų tikslumą? Todėl buvo panaudotas kitas būdas, kuris nors ir reikalauja daugiau skaičiavimo resursų (su matricomis ir vektoriais atliekama daugiau veiksmų), tačiau leidžia išmatuoti algoritmų daromą paklaidą bei netiktį. Generuojamas buvo ne vektorius b, o vektorius x, kurį galima pavadinti x_tikslus. Toliau šis vektorius buvo naudojamas vektoriaus b paieškai, t.y. buvo atliekama matricos

A ir vektoriaus x_tikslus sandauga: b = A * x_tikslus ir taip buvo gaunamas vektorius b, kurį jau galima buvo naudoti lygčių sistemoje.

Tokiu būdu, buvo sukuriamas uždavinys Ax = b, kurio tikslus atsakymas - vektorius x yra lygus mūsų generuotam vektoriui x_tikslus. Kitas žingsnis buvo, šios sistemos sprendimas, tarsi vektorius x būtų nežinomas. Taip įvairių algoritmų pagalba buvo apskaičiuojamas sistemos sprendinys, kurį galima vadinti x_apytikslis. Tokiu būdu buvo gaunami du vektoriai: x_tikslus (generuotas) ir x_apytikslis (apskaičiuotas). Šie atlikti žingsniai leido ne tik nustatyti skirtingų algoritmų skaičiavimo greitį, bet ir nustatyti jų skaičiavimo paklaidas, kurias autorius nustatė pasinaudojant vektoriaus Euklidinės normos formulę:

$$|\mathbf{x}|_2 = |\mathbf{x}| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}\,.$$

kur || x || = || x_tikslus - x_apytikslis ||. Tai padėjo vertinti algoritmus ne tik pagal skaičiavimo laiką bet ir pridėti dar vieną algoritmo vertinimo kriterijų – skaičiavimo paklaidas.

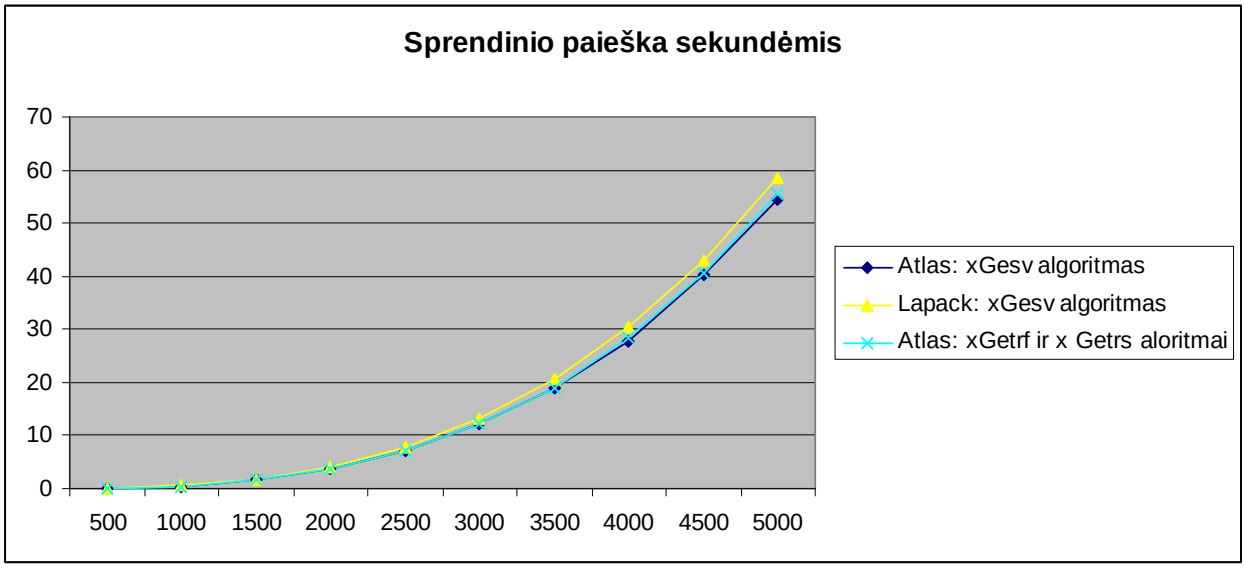
Prie aukščiau aprašytų kriterijų buvo nuspręsta įvesti dar vieną – netikties skaičiavimą (ang. Residual error arba backward error). Tam vėlgi reikėjo panaudoti Euklidinės normos formulę, tik kad šį kartą pirmiausia reikia apskaičiuoti vektorių b, kurį sąlyginai galima pavadinti b_apytikslis: b_apytikslis = A * x_apytikslis. Panaudojant šią apskaičiuotą reikšmę bei anksčiau prieš tai apskaičiuotą vektorių b, galima apskaičiuoti netiktį:

|| b || = || b – b_apytikslis || arba || A x_apytikslis – b ||

Pagal tokius žingsnius buvo atlikta visa eilė bandymų. Žemiau pateiktose diagramose autorius norėjo pavaizduoti dalį iš gautų rezultatų. Pirmoje diagramoje yra pavaizduotas sprendinių paieškos laikas sekundėmis, panaudojant skirtingus algoritmus iš skirtingų bibliotekų.

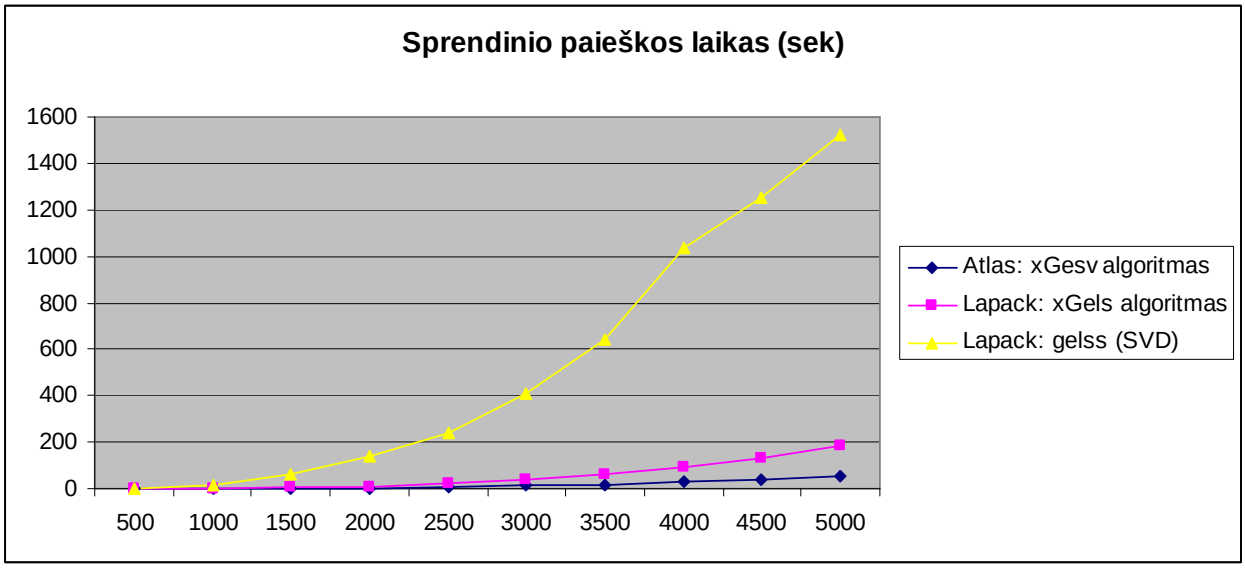
5.3.3. Algoritmų darbo laiko vertinimas

Rezultatai specialiai parodomi skirtinguose diagramose, kadangi atskirų algoritmų skaičiavimo laikai labai stipriai skiriasi.Iš diagramų matome, kad algoritmai, kurie yra paremti matricos LU dekompozicija tarp jų yra xGesv ir xGetrs algoritmai dirba keliolika kartų greičiau negu algoritmai, kurie yra paremti SVD (ang. Singular Value Decomposition) principu, pvz.: xGelss algoritmas.



1 diagrama. Sprendinių vektoriaus x paieškos laikas (sek.) pirmai grupei algoritmų

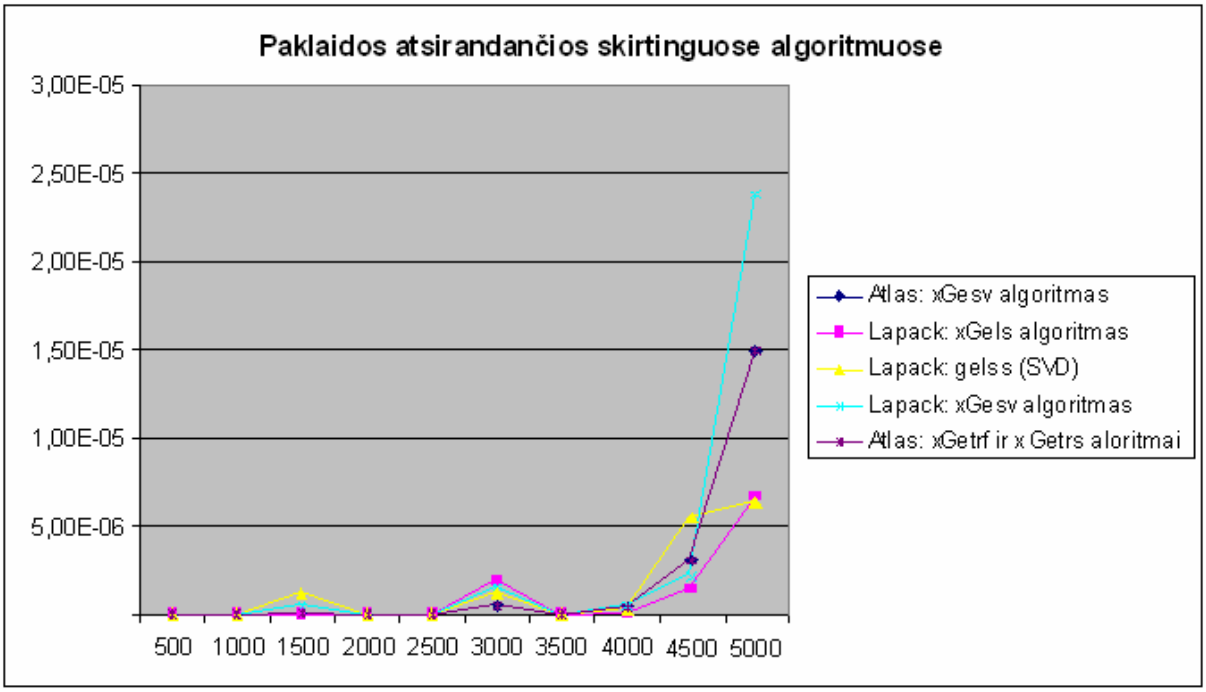
Žemiau pateiktoje diagramoje itin stipriai išsiskiria xGelss algoritmas, kurio vykdymo laikas itin stipriai išaugą, kai pradedame skaičiuoti dideles matricas, šito reiškinio priežastį bei tikslą dėl kurio galima naudoti tokį lėtą algoritmą pamatysime iš kitų diagramų.



2 diagrama. Sprendinių vektoriaus x paieškos laikas (sek.) antrai grupei algoritmų

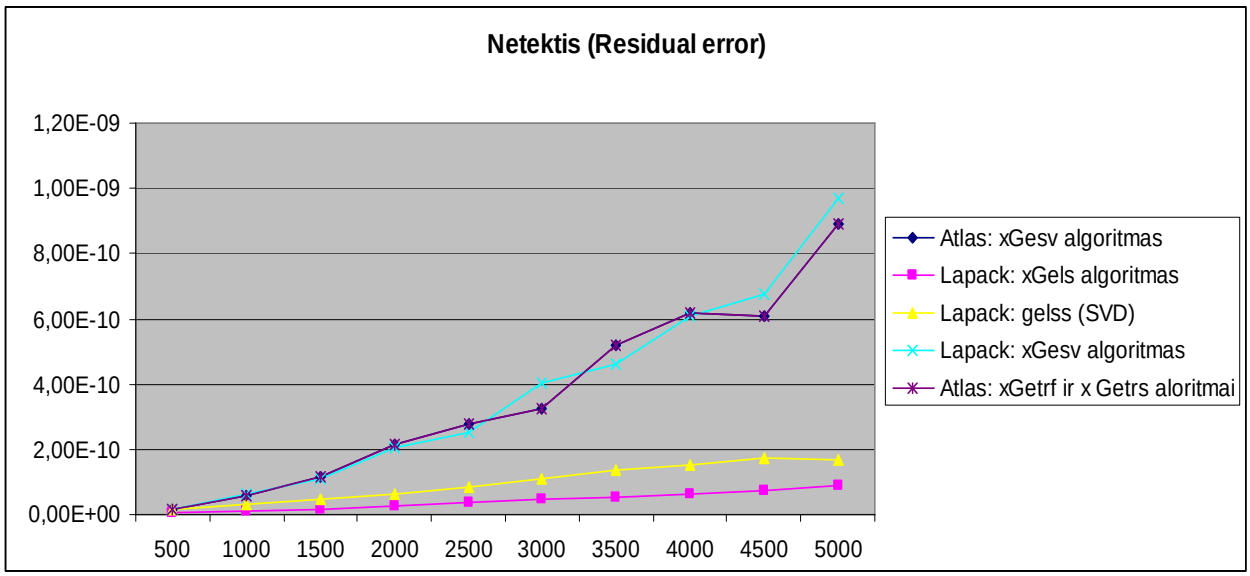
5.3.4. Algoritmų skaičiavimo paklaidos bei apskaičiuota netiktis

Žemiau pateiktose diagramose yra pavaizduotas kitas gautas rezultatas, o tiksliau sakant paklaidos atsirandančios skirtinguose algoritmuose. Nesunku pastebėti, kad algoritmai, kurie dirba greičiausiai, tuo pačiu skaičiuoja ir su didesnėmis paklaidomis negu kiti. Itin gerai, tai galima pastebėti tiriant xGesv algoritmo skaičiavimus, jo paklaidos yra didesnės negu kitų algoritmų. Tuo pačiu verta paminėti, kad atlas bibliotekos algoritmai, kurie iš esmės yra lapack bibliotekos optimizuoti algoritmai, parodo geresnį darbą negu paprastos jų versijos, kaip paklaidų taip ir laiko atžvilgiu.



3 diagrama. Skaičiavimo paklaidos atsirandančios skirtinguose algoritmuose

Panašią situaciją matome ir su netiktimi, lapack algoritmų xGels ir xGelss algoritmai skaičiuoja su mažesne netiktimi negu greiti dekompozicijos algoritmai.



4 diagrama. Skirtingų algoritmų apskaičiuota netektis

Trumpai apibendrinant šį skyrių, autorius nori pabrėžti, kad priklausomai nuo to ar mums reikia greito ar tikslesnio atsakymo reikia ieškoti skirtingų algoritmų. Dažnai norima rezultatą gauti kuo greičiau, tokiu atveju reikia paieškoti algoritmų iš atlas bibliotekos, priešingu atveju, jei svarbiausias yra tikslumas, reikia panagrinėti lapack bibliotekos teikiamus algoritmus.

5.3.5. Antras etapas, uždavinio sprendinio stabilumo tyrimas

Antrame etape buvo atlikta sprendinio paieškos stabilumo analizė, kuri leido dar labiau priartinti atliekamus bandymus prie bandymų paimtų iš realių taikymų. Kadangi modeliuojant tam tikras situacijas visada verta atkreipti dėmesį, kaip modeliuojamą objektą gali paveikti pradinių duomenų pakeitimas arba kitaip sakant triukšmo įnešimas. Šio darbo kontekste, objektas, kuris gali būti paveiktas yra sprendinių vektorius, o duomenų pakeitimas, tai į pradinę matricą įvedamas triukšmas.

Analizuodami tą patį uždavinį Ax = b, pradžioje buvo atlikti tie patys veiksmai, kaip ir prieš tai buvusiame etape, t.y. buvo sugeneruotas vektorius x_tikslus kuris buvo padaugintas iš turimos pradinės matricos A (duota matrica, be triukšmo), taip buvo apskaičiuotas vektorius b bei vektorius x_apytikslis, kuris buvo gautas išsprendus tiesinių lygčių sistemą su vektoriumi b. Šie duomenys buvo panaudoti kitame tyrimo etape.

Kitas žingsnis buvo triukšmo matricos sukūrimas. Šiam tikslui buvo konstruojama nauja matrica U, kuri privalėjo būti to pačio dydžio, kaip ir duota pradinė matrica A. Naujos matricos U elementai buvo generuojami atsitiktinai iš intervalo -½ ir ½. Turėdami atsitiktinio triukšmo matricą, galima

atlikti sprendinio stabilumo analizę ir ištirti, kaip stipriai bus paveiktas sprendinys priklausomai nuo to, kokio intensyvumo bus įnešamas triukšmas.

Kitas žingsnis buvo matricos su triukšmu arba perturbuotos matricos A‘ apskaičiavimas. Šiam tikslui pasiekti autorius panaudojo formulę A‘ = α * U + A, kur koeficientas α apibrėžia triukšmo intensyvumą, t.y. kuo didesnis yra šis koeficientas, tuo didesni bus elementai esantis matricoje U ir tuo labiau matricą A‘ skirsis nuo pradinės, duotos matricos A ir tuo labiau bus paveiktas sistemos sprendinys. Kitame žingsnyje autorius spręsdamas sistemą:

A` * x_su_triuksmu = b, suranda sprendinių vektorių x_su_triuksmu. Dabar pasinaudojant ta pačia Euklidinės normos formule || x_apytikslis – x_su_triuksmu ||, autorius vertina, kokio jautrumo yra sistemos sprendinys, t.y. kaip stipriai jis reaguoja į triukšmo didinimą – koeficientą α pakeitimą.

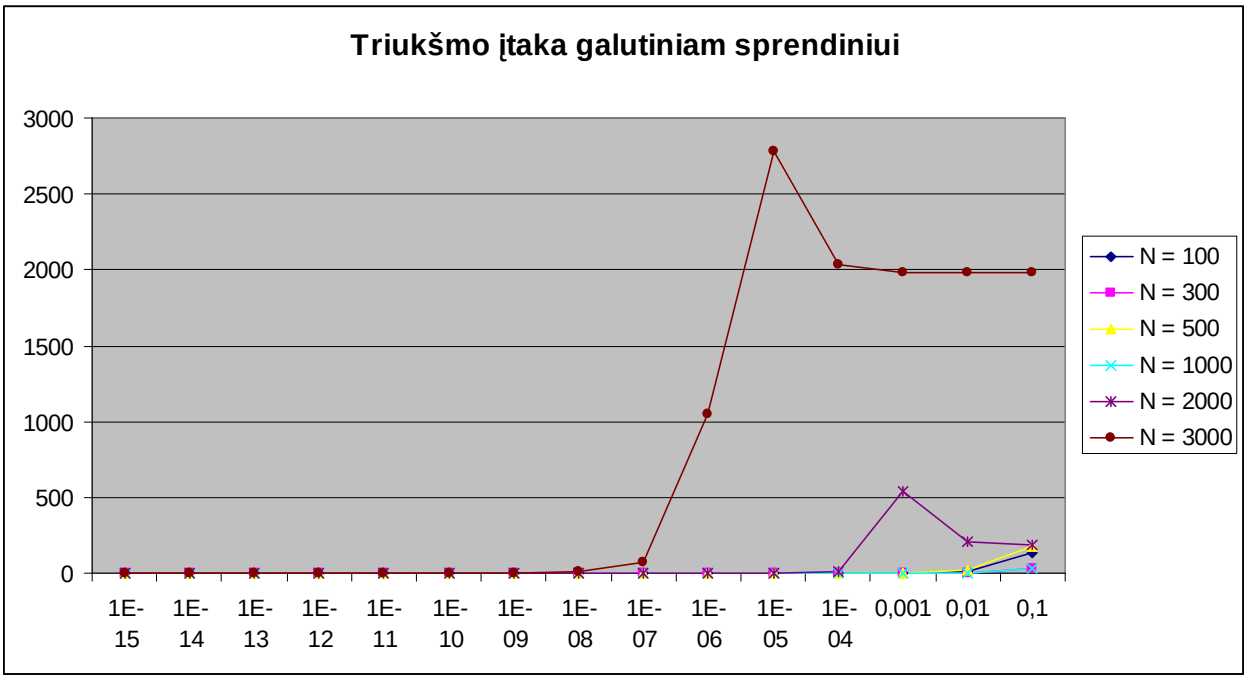
Tam, kad galima būtų gauti pilną vaizdą būtina kelis kartus skaičiuoti uždavinį, kiekvieną kartą didinant koeficientą α. Autorius pasirinko dvi šio koeficiento didinimo strategijas, kaip viena taip ir kita prasideda nuo labai mažo α, kurio eilė yra 10^{-15}, ir užsibaigia ties 10^{-1} arba 0,1, kai triukšmas tampa jau toks didelis, kad gali užgožti pradinę matricą. Du būdai skiriasi žingsniais tarp dviejų gretimų α reikšmių. Pirmu atveju**,** buvo parinktas didinimas po eilę, t.y. su kiekvienu žingsniu α buvo 10 kartų didesnis negu prieš tai buvusi reikšmė, tai buvo generuojama α seka:

10^{-15},10^{-14},10^{-13}, ..., 10^{-1}.

Antras vertinimo būdas reikalavo dar daugiau skaičiavimų, kadangi žingsnis buvo parinktas dar mažesnis ir kiekvienas iš prieš tai buvusių intervalų buvo padalintas dar į 9 dalis

10^{-15} * 9, 10^{-15},* 8, ... ir t.t:

Dėl daug kartų kartojamų sistemos sprendimų (keičiamas triukšmo intensyvumas), šiems bandymams atlikti reikėjo labai greito algoritmo, būtent todėl ir buvo parinktas ATLAS bibliotekos dgesv algoritmas, kuris prieš tai atliktuose eksperimentuose parodė geriausią laiką.

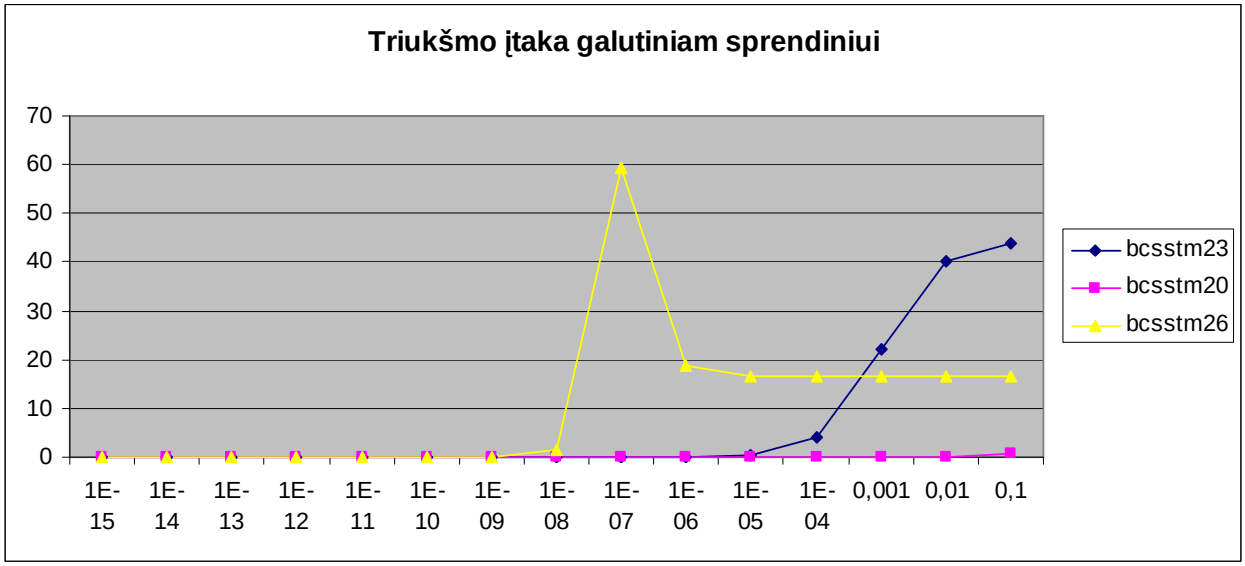


4 diagrama. Triukšmo įtaka galutiniam sprendiniui, kai matrica A buvo generuojama

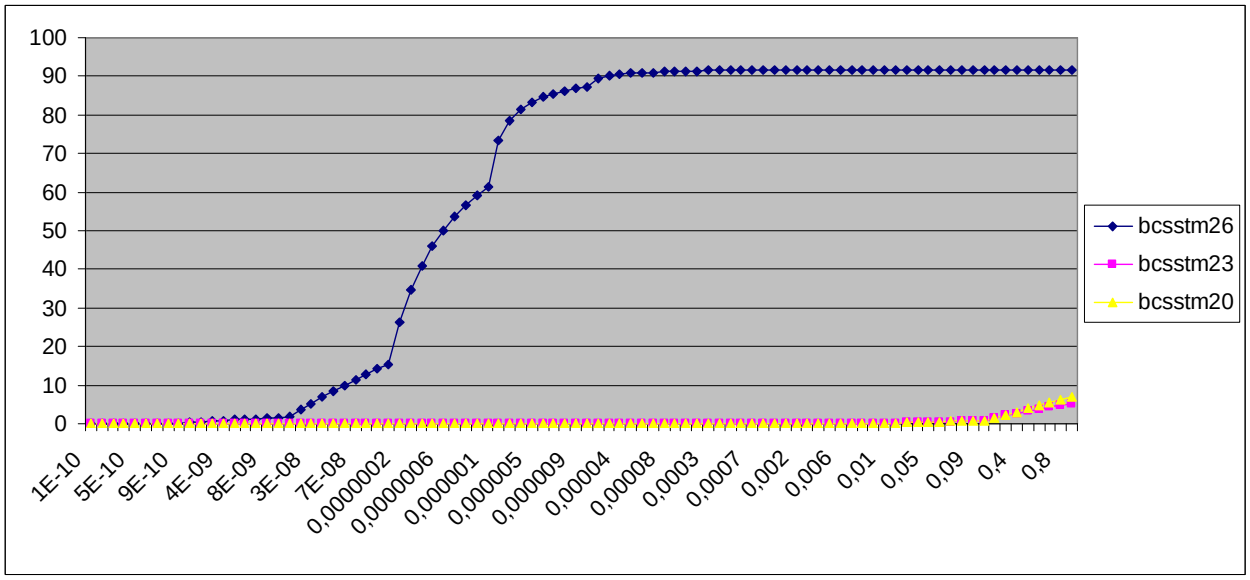
Iš pateiktos diagramos matome, kad sistemos sprendinys bendru atveju yra labiausiai paveikiamas, kai matrica yra didelė. Taip pat būtina pastebėti, kad pridedamo triukšmo intensyvumas turi lemiama poveikį galutiniam sprendiniui. Kuo didesnė yra matrica tuo reikia pakankamai mažo triukšmo, kad jis paveiktų galutinį sprendinį ir atvirkščiai kuo mažesnė matrica tuo didesnio reikia triukšmo.

Žemiau yra parodyta keleto matricos iš Matrix Market svetainės analizė. Kaip pavyzdys buvo paimtos kelios matricos iš Harwell-Boing kolekcijos:

BCSSTM20	Struktūrinė inžinierinės matricos. Kabančio tilto korpusas; N = 485
BCSSTM23	Struktūrinė inžinierinės matricos. Part of a 3D globally triang. Building; N = 485
BCSSTM24	Struktūrinė inžinierinės matricos. Calgary olimpinis stadionas; N = 3562



5 diagrama. Triukšmo įtaka galutiniam sprendiniui, kai žingsnis yra didelis



6 diagrama. Triukšmo įtaka galutiniam sprendiniui, kai žingsnis yra mažas

Pateiktos diagramos patvirtina teiginį, kad kuo matrica yra didesnė ir tankesnė tuo reikia, mažesnio triukšmo intensyvumo, kad jis galėtų paveikti galutinį sprendinį.

5.3.6. Užduoties išlygiagretinimas – BalticGrid

Jau atliekant aukščiau aprašytus bandymus, buvo susidurta su laiko problema t.y. atliekami bandymai buvo atliekami vienas po kito ant vieno kompiuterio (Intel M procesorius 1,7Ghz taktinis dažnis, 512 Mb operatyvios atminties, 330 Mb laisvos atminties). Tokio darbo trūkumai paaiškėjo iš

karto, kadangi vienu metu gali būti sprendžiamas tik vienas uždavinys, tai susidarydavo eilė uždavinių, kurie buvo sprendžiami griežtai vienas po kito. Jei programos testavimui užtekdavo ją praleisti su trimis – penkiomis matricomis, tai norint gauti realius rezultatus buvo atliekami bandymai su keturiasdešimt matricų, kur kiekviena matrica buvo sprendžiama keliais algoritmais, be to dar buvo tiriamas sprendinio stabilumas, o tai reiškia, kad ta pati sistemą buvo sprendžiama po keliasdešimt kartų. Panaudojant paprastą aritmetiką galima pabandyti apytiksliai apskaičiuoti, kiek reikėtų laiko visam tyrimui atlikti. Turėdami 5 skirtingus algoritmus, apie 40 įvairių dydžių matricų, kurių vidutinis dydis yra virš dviejų tūkstančių eilučių jau reikėtų išspręsti apie 200 sistemų. Be to, reikia nepamiršti apie triukšmo pridėjimą ir analizę, o tai prideda dar mažiausiai 15 kartų kiekvienos sistemos sprendimų, o tai reiškia dar 600 sistemų sprendimų, taigi net jei bus panaudotas pats greičiausias algoritmas, tokie skaičiavimai tikrai užtruks. Norint sumažinti rankinį darbą, be abejo buvo paruošta aplinka, kurį leido visus tyrimus atlikti vienas po kito ir saugoti duomenis patogiu formatu. Bet, tai neišsprendė pagrindinės problemos, visos užduotys vis tiek buvo skaičiuojamos nuosekliai, tik užbaigus vieną užduotį, buvo pradedama skaičiuoti kita.

Reikėjo sprendimo, kuris leistų išlygiagretinti tyrimo procesą. Dažnai, kaip išlygiagretinimas suprantamas pačios programos arba atskirų jos dalių perrašymas, t.y. kodo išlygiagretinimas, pavyzdžiui, tam kad galima būtų panaudoti MPI sąsają. Tačiau, kaip jau buvo minėta, tai nebuvo šio darbo tikslas. Autorius sukūręs programų paketą, kuris leido spręsti ir vertinti individualių uždavinių sprendinio paieškos laiką bei tikslumą bei parinkęs visą aibę duomenų, tikėjosi surasti kitokį sprendinį, negu kiekvieno algoritmo išlygiagretinimas, kuris galėtų sumažinti individualios užduoties skaičiavimo laiką. Autorius pradėjo ieškoti priemonės, kurį leistų panaudoti visas sukurtas nuoseklias programas, bibliotekas bei paruoštus testavimo duomenis bet tuo pačiu visus skaičiavimus atliktų ne nuosekliai, bet lygiagrečiai ant daugelio kompiuterių. Toks sprendinys buvo rastas ir tai buvo BalticGrid servisas.

5.3.7. BalticGrid, kaip priemonė, leidžianti atlikti lygiagrečius skaičiavimus

Išanalizavęs BalticGrido galimybes, autorius iškėlė sau tikslą sukurti strategiją, kurį leistų išlygiagretinti viso projekto skaičiavimus, tuo pačiu sumažinant bendrą skaičiavimo laiką. Tai reiškia, kad autorius užsibrėžė tikslą, sukurti mechanizmą, kuris leistų panaudoti jau parašytas programas bei paruoštus duomenis ir paskirstyti darbą tarp keliasdešimties kompiuterių esančių BalticGrido tinkle, taip sumažinant bendrą uždavinio skaičiavimo laiką. Kaip papildomas

reikalavimas, mechanizmas turėjo būti patogiai konfigūruojamas, paprastai paleidžiamas bei procesas turėjo būti patogiai stebimas.

Visų pirma reikėjo pasirūpinti tuo, kad paruoštos programos pasileistų BalticGrid tinkle. Deja, tai pavyko ne iš karto. Paprastos programos nenaudojančios jokių bibliotekų, pasileisdavo BalticGride ir sėkmingai užbaigdavo savo darbą, bet kadangi autoriaus atliekamiems bandymams rašytos programos naudodavo išorines bibliotekas, atsirado problemos su programų paleidimu gride. Teko kiekvieną kartą siunčiant uždavinį į gridą siųsti kartu ir visas bibliotekas, tai sudarė daug sunkumų. Tuo labiau, kad kai kurios bibliotekos yra gana didelės net po keliasdešimt megabaitų, todėl uždavinių pateikimas skaičiavimams tapo gana sudėtingas. Tik suinteresuotų asmenų dėka BalticGride atsirado algebros programinei paketai, kurie padėjo išspręsti šią problemą ir atsisakyti bibliotekų persiuntimo kiekvieną kartą siunčiant uždavinį į klasterį. Verta paminėti, kad kol bibliotekos nebuvo įdiegtos klasteryje buvo rastas laikinas sprendimas. Bibliotekoms saugoti buvo naudojamas grido failo saugyklą. Pasinaudojus šia teikiama galimybe pavyko sumažinti duomenų kiekį persiunčiamų į gridą ir gaunamų iš grido.

Sutvarkius šį darbų etapą ir užtikrinus sėkmingą programų kompiliavimą bei paleidimą gride, autorius perėjo prie uždavinių sprendimo schemos kūrimo. Jei pavieniams uždaviniams spręsti užteko paprasto jdl (ang. Job Description Language) failo su tipu JobType = Normal, tai visos aibės uždavinių sprendimui reikėjo, sudėtingesnio mechanizmo. Todėl buvo pradėti gilesnį grido galimybių tyrimai ir buvo rastas keletas tinkamų sprendimų. Visų pirma, tai parametriniai jdl failai, kurie leidžia nustatyti visą aibę parametrų, o taip pat ir paduodamų duomenų būdus bei kiekį. Be to, jie leidžia sukurti pseudo ciklą panašų i for ciklą, kuriame galima nustatyti ciklo pradžią, pabaigą bei žingsnį. Pavyko parašyti tokį jdl falą, kuris kiekviename žingsnyje generavo naujus, paduodamus į programą, parametrus. Autorius, panaudojo tai, kaip triukšmo intensyvumo generatorių. Programa paduodamus parametrus naudojo, kaip triukšmo matricos koeficientą α. Esminis dalykas čia buvo tai, kad WMS (ang. Workload Management System) paskirstydavo kiekvieną iš šių užduočių atskiram kompiuteriui, todėl, pavyzdžiui, uždavinys su sprendinio stabilumo analize buvo sprendžiamas lygiagrečiai, panaudojant 15 kompiuterių. Žemiau yra pateikta ištrauka iš paleidimo failo:

```
[
    JobType = "Parametric";
    Executable = "runAtlas.sh";
    StdInput = "m_PARAM_.mtx";
    StdOutput = "m_PARAM_.out";
    StdError = "m_PARAM_.err";
    Parameters = 40;
```

```
    ParameterStart = 1;
    ParameterStep = 1;

    RetryCount = 1;
    Requirements = other.GlueCEInfoTotalCPUs > 2;
    Rank = other.GlueCEStateFreeCPUs;
    InputSandbox  = {"atlas","runAtlas.sh","m_PARAM_.mtx"};
    InputSandboxBaseURI = "gsiftp://pupa.elen.ktu.lt/storage/balticgrid/ansu1485/";
    OutputSandbox = {"m_PARAM_.out","m_PARAM_.err"};
    OutputSandboxBaseURI = "gsiftp://pupa.elen.ktu.lt/storage/balticgrid/ansu1485/";
...
];
```

Kitas būdas, kuris padėjo organizuoti BalticGrido kompiuterių darbą yra collection tipo jdl failas, kuris leidžia sujungti kelis uždavinius į vieną ir valdyti jį pasinaudojant tik vienu id. Paruošęs matricų sąrašą jos buvo paduodamos programoms ir kiekviena iš jų buvo sprendžiama atskirame kompiuteryje.

```
[
   Type = "collection";
   VirtualOrganisation = "balticgrid";
   StdError = "main.err";
   InputSandbox = {"runAtlas.sh","m236x236.mtx","m900x900.mtx","atlas"};
   InputSandboxBaseURI = "gsiftp://pupa.elen.ktu.lt/storage/balticgrid/ansu1485/";
   OutputSandbox = {"main.err"};
   OutputSandboxBaseURI = "gsiftp://pupa.elen.ktu.lt/storage/balticgrid/ansu1485/";
   Requirements=Member("VO-balticgrid-ktu/APPS/MATH/ATLAS_LAPACK/3.6.0",
    other.GlueHostApplicationSoftwareRunTimeEnvironment) &&
        Member("VO-balticgrid-ktu",other.GlueHostApplicationSoftwareRunTimeEnvironment) &&
    !(other.GlueCEUniqueID == "grid5.mif.vu.lt:2119/blah-pbs-sdj" ||
    other.GlueCEUniqueID=="grid2.mif.vu.lt:2119/jobmanager-lcgpbs-sdj")&&
    other.GlueCEInfoTotalCPUs > 20;
   RetryCount = 0;
   max_nodes_running = 50;

   nodes = {
      [
       NodeName = "job1";
            Executable = "runAtlas.sh";
            Arguments = "m236x236.mtx";
            RetryCount = 1;
            StdOutput = "job1.out";
            StdError = "job1.err";
            OutputSandbox = {"job1.out","job1.err"};
      ],
      [
            NodeName = "job2";
            Executable = "runPaklaida.sh";
```

```
            Arguments = "m700x700.mtx 1 ";
            RetryCount = 2;
            StdOutput = "job2.out";
            StdError = "job2.err";
            OutputSandbox = {"job2.out","job2.err"};
        ]
....

    };
]
```

Paskutinis naudojamas jdl failo tipas yra dag (directed acyclic graph) failas, kuris leido sujungti keletą uždavinių taip, kad vieno įvedimas, išvedimas arba vykdymas būtų priklausomas nuo kito, tai leido sukurti hierarchinę darbų vykdymo struktūrą. Žemiau pateiktame pavyzdyje yra parodytas sąryšis tarp mazgų nodeComp ir nodeA bei nodeA mazgo sąryšis su mazgu nodeB. Toks sąryšis apibrėžia tokią vykdymų seką: užduotis A galės būti įvykdytą tik tada, kai pasibaigs mazgo nodeComp darbas, o nodeB darbas galės būti pradėtas tik tada, kai pasibaigs skaičiavimai mazge nodeA. Tokia logika buvo panaudota, triukšmų matricų sudarymui ir perdavimui kitiems mazgams tolimesniam apdorojimui.

Dag.jdl

```
[
    Type = "dag";
    VirtualOrganisation = "balticgrid";
    StdError = "main.err";
    InputSandbox = {"runAtlas.sh","m236x236.mtx"};
    OutputSandbox = {"main.err"};
    RetryCount = 0;
    max_nodes_running = 20;

    nodes = [
        nodeComp = [
            description = [
                JobType = "Normal";
                Executable = "comp.sh";
                InputSandbox = {"comp.sh","Makefile","atlas2.c","interface.h","mmio.c","mmio.h"};
                OutputSandbox = {"atlas"};
                Requirements=
            Member("VO-balticgrid-ktu/APPS/MATH/ATLAS_LAPACK/3.6.0",
            other.GlueHostApplicationSoftwareRunTimeEnvironment) &&
Member("VO-balticgrid-ktu",other.GlueHostApplicationSoftwareRunTimeEnvironment) &&
!(other.GlueCEUniqueID == "grid5.mif.vu.lt:2119/blah-pbs-sdj" ||
  other.GlueCEUniqueID == "grid2.mif.vu.lt:2119/jobmanager-lcgpbs-sdj");
            ];
        ];
```

```
    nodeA = [
      description = [
        JobType = "Normal";
        Executable = "runAtlas.sh";
        Arguments = "m3000x3000.mtx";
        RetryCount = 1;
        StdOutput = "nodeA.out";
        StdError = "nodeA.err";
        InputSandbox  = {root.InputSandbox,root.nodes.nodeComp.OutputSandbox[0]};
        OutputSandbox = {"nodeA.out","nodeA.err"};
      ];
    ];

     nodeB = [
      description = [
        JobType = "Normal";
        Executable = "triuksmas.sh";
        Arguments = "m2000x2000.mtx";
        RetryCount = 1;
        Requirements = other.GlueCEInfoTotalCPUs > 2;
        Rank = other.GlueCEStateFreeCPUs;
        StdOutput = "nodeB.out";
        StdError = "nodeB.err";
        InputSandbox  = {root.InputSandbox,root.nodes.nodeA.OutputSandbox[0]};
        OutputSandbox = {"nodeB.out","nodeB.err"};
      ];
    ];

....

  ];

  dependencies = {
    { nodeComp, nodeA },
    { nodeA, nodeB },
   ...
  };
];
```

Taigi pasinaudojant aukščiau aprašytomis priemonėmis buvo sukurta tokia paleidimo schema, kuri yra nesunkiai konfigūruojama, papildoma, pakartotinai perleidžiama įvykus klaidai ir patogiai stebima ir aišku esminis jos privalumas yra tai, kad panaudojus BalticGridą užduočių paskirstymui buvo sumažintas bendras užduočių skaičiavimo laikas. Jei viename kompiuteryje atlikti visus skaičiavimus, autoriui užtruko apie 26 val., tai BalticGrido pagalba, tie patys rezultatai buvo gauti jau po ~14 val, kai skaičiavimams buvo panaudota 20 procesorių. Taigi bendrą tyrimų laiką, pavyko

sumažinti beveik dvigubai. Rezultatas pakankamai geras, tik iškilo klausimas, kodėl taip stipriai padidinus kompiuterių skaičių, nepavyko dar labiau sumažinti skaičiavimų laiką. Autoriaus nuomone, tai atsitiko dėl dviejų priežasčių, pirmiausia tai, įvyko todėl, kad kai kurios užduotis vis tik turėjo laukti savo vykdymo pradžios be to užtruko duomenų persiuntimas tarp mazgų bei failų saugyklos. Bet tai buvo tik vienas minusas, dar labiau koja pakišo jau minėtas WMS, kuris paskirsto darbus tarp kompiuterių. Autorius pastebėjo, kad labai dažnai, nors kompiuteriai jau galėtų skaičiuoti kitą uždavinį, jų statusas vis dar buvo „laukiantis“ ir nauja užduotis jiems iš karto nebuvo paskiriama, taip buvo prarandamas brangus laikas. Tačiau, kadangi BalticGrid projektas yra nuolat tobulinamas, tai manau ateityje ši problema bus išspręsta ir tai padės efektyviau panaudoti laisvus resursus, o tai, savo ruožtu, reikš dar greitesnį uždavinių skaičiavimą.

5.4.Tyrimo rezultatai

Apibendrinant praktinėje darbo dalyje atliktus tyrimus, autorius pateikia žemiau surašytus pagrindinius gautus rezultatus. Visų pirma, remiantis atliktais algoritmų tyrimais bei atliktais bandymais, buvo detaliai išanalizuoti atrinkti algoritmai, įvertinant jų skaičiavimo laiką, paklaidas bei netiktį. Skaičiavimo laiko atžvilgiu, geriausiai pasirodė LU dekompozicijos algoritmai GESV ir GETRS, kaip iš ATLAS taip ir iš LAPACK bibliotekų, šiek tiek blogesnį laiką parodė QR faktorizacija paremtas LAPACK bibliotekos algoritmas GELS. Panaudojant SVD algoritmą GELSS tiesinės lygčių sistemos sprendinys buvo ieškomas ilgiausiai, užtat jis ir GELS algoritmai parodė geriausią rezultatą, jei kalbama apie atsakymo tikslumą, kadangi šių dviejų algoritmų skaičiavimo klaidos buvo mažiausios. Panaši situacija buvo ir su netikties skaičiavimu. Be to, atlikti pirmoje praktinio darbo dalyje tyrimai parodė optimizuotos bibliotekos ATLAS pranašumą prieš kitas bibliotekas, tokias kaip LAPACK ir BLAS, ir įrodė, kad kartais gerai optimizuota nuosekli programa dirba dar greičiau negu išlygiagretinta ir paleista ant didesnio skaičiaus procesorių programos versija.

Antroje tyrimų dalyje buvo atlikti bandymai, kurie leido įvertinti triukšmo įtaką galutiniam tiesinių lygčių sistemos sprendiniui. Remiantis gautais rezultatais, galima daryti išvadą, kad sprendinių vektorius yra labiau paveikiamas triukšmo didelėse matricose ir kuo matrica yra didesnė tuo reikia mažesnio triukšmo, kad sistemos sprendinys pasikeistų ir atvirkščiai – kuo matrica yra mažesnė tuo didesnio reikia triukšmo, kad tai paveiktų galutinį sprendinį.

Panaudojant BalticGrid skaičiavimo galimybes pavyko, neperrašant nuosekliųjų programų versijų, išlygiagretinti didelės apimties tiesinės algebros uždavinių paketo sprendimą, kas savo ruožtu leido sumažinti beveik dvigubai bendrą užduočių skaičiavimo laiką.

Išvados ir rekomendacijos

Iš atliktų tyrimų bei atliktos analizės autorius priėjo kelių išvadų, kurios bus išdėstytos žemiau. Panaudojant specializuotas tiesinės algebros paketus, galima pasiekti įspūdingų užduočių skaičiavimo rezultatų. Visų pirma, konkrečiai kiekvienam uždaviniui pritaikytas bei maksimaliai optimizuotas algoritmas leidžia kelis kartus greičiau atlikti dominančius skaičiavimus, negu parašytos ir neoptimizuotos programos. Esminis yra ne tik bibliotekos bei atskiro algoritmo parinkimas, kadangi, priklausomai nuo algoritmo gali kardinaliai skirtis uždavinio sprendimo efektyvumas. Jei pirmoje vietoje yra sprendinio radimo greitis reikia ieškoti tarp greitesnių algoritmų, pavyzdžiui, paremtus LU dekompozicija, jei domina kuo mažesnė skaičiavimų paklaida reikėtų atkreipti dėmesį į SVD algoritmus.

Tačiau, net panaudojant itin greitus algoritmus, kartais itin dideliuose projektuose to neužtenka. Reikia ieškoti galimybės paskirstyti skaičiavimus tarp kelių kompiuterių, kadangi tik tai gali padėti sumažinti bendrą užduočių skaičiavimo laiką. Lygiagrečių skaičiavimų ir daugiaprocesorinės sistemos, tokios kaip BalticGrid tinklas, panaudojimas leidžia žymiai sumažinti bendro uždavinių paketo skaičiavimo laiką arba per tą patį laiko tarpą apdoroti didesnį kiekį duomenų. Tai įmanoma pasiekti tik tuo atveju, jei yra apdorojami iš tikrųjų dideli duomenų kiekiai, kitaip laikas, kuris bus išnaudojamas komunikacijai tarp mazgų užtikrinti bus palyginamai didelis, o tai savo ruožtu neleis pagreitinti uždavinių skaičiavimo. Panaši situacija yra gaunama, kai yra netinkamai parinktas procesorių skaičius arba ne maksimaliai apkrauti visi mazgai. Tik atsižvelgus į šias pastabas galima pasiekti maksimalų sistemos našumo išnaudojimą.

Be to, verta paminėti, kad BalticGrid yra patogus, nesunkiai konfigūruojamas servisas, leidžiantis atlikinėti lygiagrečius skaičiavimus net ir su nuosekliomis programos versijomis bei nebūtinai panaudojant MPI sąsają. Jei dar ateityje pavyktų patobulinti Grido darbų paskirstymo mechanizmą, kuris greičiau apkrautų laisvus mazgus, tai jo populiarumas nuo to tik išaugtų.

Literatūros sąrašas

[Saa00a] Yousef Saad. Iterative Methods for Sparse Linear Systems.

[Fly72] Flynn, M., Some Computer Organizations and Their Effectiveness, 1972

[Fos95] Ian Foster, Designing and Building Parallel Programs, 1995

[Amd67] G. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities.AFIPS Press. – 1967

[BDZ03] А. А. Букатов, В. Н. Дацюк, А. И. Жегуло, Программирование многопроцессорных вычислительных систем, Ростов-на-Дону 2003

[Mpi95] MPI: The Complete Reference
http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

[Sca97] ScaLAPACK Users' Guide
http://www.netlib.org/scalapack/scalapack_home.html

[Mat04] Matrix Market – matricų saugykla.
http://gams.nist.gov/MatrixMarket/

[Bal05] BalticGrid project.
http://www.balticgrid.org/

[Lit07] Lietuvos akademinių institucijų lygiagrečiųjų ir paskirstytų skaičiavimų tinklas
http://www.litgrid.lt

[Net05] Netlib – collection of mathematical software, papers, and databases.
http://www.netlib.org

[Bre07] Breakdown of requests to each Netlib library
http://www.netlib.org/master_counts2.html

[Jdl06] JDL Attributes Specification (submission via WMS WMProxy)
https://edms.cern.ch/file/590869/1/EGEE-JRA1-TEC-590869-JDL-Attributes-v0-8.pdf

[Gli07] gLite 3.0 User Guide
https://edms.cern.ch/file/722398//gLite-3-UserGuide.html

[Atl00] Automatically Tuned Linear Algebra Software (ATLAS)
http://math-atlas.sourceforge.net/

[Lap07] LAPACK -- Linear Algebra PACKage
http://www.netlib.org/lapack/index.html

[Pet06] Portable, Extensible Toolkit for Scientific Computation
http://www-unix.mcs.anl.gov/petsc/petsc-as/

[FG03] V. Fraysse, L. Giraud , 2003 A Set of Conjugate Gradient Routines for Real and Complex Arithmetics
http://www.cerfacs.fr/algor/reports/2000/TR_PA_00_47.pdf

[FGG+03] V. Fraysse, L. Giraud, S. Gratton, J. Langou 2003, A Set of GMRES Routines for Real and Complex Arithmetics.
http://www.cerfacs.fr/algor/reports/2003/TR_PA_03_03.pdf

[ZX06] Selective Data Distortion via Structural Partition and
SSVD for Privacy Preservation, Wang Weijun Zhong, Shuting Xu, Jun Zhang, Southeast University, Nanjing, 210096, P.R. China, 2006

[ZW07] Matrix Decomposition-Based Data Distortion Techniques for Privacy Preservation in Data Mining, Jun Zhang, Jie Wang; Technical Report TR 472-07, Department of Computer Science, University of Kentucky, Lexington, KY, 2007

Priedas Nr. 1

Tiesinė matricų daugybos realizacija

```
#include <stdio.h>
#include <time.h>
#define AEilut 350      /* eiluciu skaicius matricoje A */
#define AStulp 300      /* stulpeliu skaicius matricoje A */
#define BStulp 500      /* stulpeliu skaicius matricoje B */

main(int argc, char **argv) {
   int intsize, dbsize, i, j, k;

   int    a[AEilut][AStulp],   /* matrica A */
          b[AStulp][BStulp],   /* matrica B */
          c[AEilut][BStulp],   /* matrica C */
          temp;

   intsize = sizeof(int);
   dbsize  = sizeof(double);

   clock_t start = clock();
   // uzpildom matricas duomenimis
   for (i=0; i<AEilut; i++) {
      for (j=0; j<AStulp; j++) {
         a[i][j]= (i + j) / 100;
      }
   }

   for (i=0; i<AStulp; i++) {
      for (j=0; j<BStulp; j++) {
         b[i][j]= (i * j) / 100;
      }
   }

   for (k=0; k<BStulp; k++) {
      for (i=0; i<AEilut; i++) {
         c[i][k] = 0.0;
         for (j=0; j<AStulp; j++) {
            c[i][k] = c[i][k] + a[i][j] * b[j][k];
         }
      }
   }

   printf("Uztruko laiko: %f\n", ((double)clock() - start) / CLOCKS_PER_SEC);

   for (j=0; j<BStulp; j++) {
      temp = temp + c[AEilut-1][j];
   }

   printf("%d   ", temp );
   printf ("\n");
}
```

Priedas Nr. 2

Lygiagreti matricų daugybos realizacija

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
#define AEilut       350   /* eiluciu skaicius matricoje A */
#define AStulp_BEilut 300   /* stulpeliu skaicius matricoje A */
#define BStulp       500   /* stulpeliu skaicius matricoje B */

#define MASTER 0          /* taskid of first task */
#define FROM_MASTER 1         /* setting a message type */
#define FROM_WORKER 2         /* setting a message type */

MPI_Status status;

main(int argc, char **argv) {
   int numtasks,      /* number of tasks in partition */
      taskid,         /* a task identifier */
      numworkers,          /* number of worker tasks */
      source,              /* task id of message source */
      proc,           /* task id of message destination */
      nbytes,              /* number of bytes in message */
      mtype,          /* message type */
      intsize,        /* size of an integer in bytes */
      dbsize,         /* size of a double float in bytes */

      eilutes,       /* Matricos A eilutes siunciamos kiekvienam procesoriui */
      kiekeil, liko, postumis,
      i, j, k, count
      ;

   int a[AEilut][AStulp_BEilut],   /* matrica A */
        b[AStulp_BEilut][BStulp],   /* matrica B */
        c[AEilut][BStulp],
        temp;   /* matrica C */

        double mytime;



   intsize = sizeof(int);
   dbsize = sizeof(double);

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   numworkers = numtasks - 1;
```

```
/**************************** MASTER ***********************************/
if (taskid == MASTER) {

   printf("Number of worker tasks = %d\n",numworkers);
   mytime = MPI_Wtime();

   // uzpildom matricas duomenimis
   for (i=0; i<AEilut; i++) {
      for (j=0; j<AStulp_BEilut; j++) {
         a[i][j]= (i + j) / 100;
      }
   }

   for (i=0; i<AStulp_BEilut; i++) {
      for (j=0; j<BStulp; j++) {
         b[i][j]= (i * j) / 100;
      }
   }

   /* siunciam matricos duomenis skirtingiems procesioriams */
   kiekeil     = AEilut/numworkers;
   liko        = AEilut%numworkers;
   postumis    = 0;
   mtype       = FROM_MASTER;

   for (proc=1; proc<=numworkers; proc++) {
      eilutes = (proc <= liko) ? kiekeil+1 : kiekeil;

      MPI_Send(&postumis, 1,                    MPI_INT, proc, mtype, MPI_COMM_WORLD);
      MPI_Send(&eilutes,  1,                    MPI_INT, proc, mtype, MPI_COMM_WORLD);
      MPI_Send(&a[postumis][0], eilutes*AStulp_BEilut,MPI_INT, proc, mtype, MPI_COMM_WORLD);
      MPI_Send(&b, AStulp_BEilut*BStulp,        MPI_INT, proc, mtype, MPI_COMM_WORLD);

      postumis = postumis + eilutes;
   }

   /* surenka visus duomenis i viena matrica C */
   mtype = FROM_WORKER;
   for (i=1; i<=numworkers; i++) {
      source = i;
      MPI_Recv(&postumis,      1,          MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&eilutes,       1,          MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&c[postumis][0], eilutes*BStulp, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
   }


   mytime = MPI_Wtime() - mytime;
   printf("Programos vykdymo laikas kompiuteryje %d yra %lf sek..\n",taskid, mytime);

   for (j=0; j<BStulp; j++) {
      temp = temp + c[AEilut-1][j];
   }

   printf("%d   ", temp );
   printf ("\n");
  }
```

```
  /**************************** SLAVE ***********************************/
  if (taskid > MASTER) {
     mtype   = FROM_MASTER;
     source  = MASTER;
     //printf ("Master =%d, mtype=%d\n", source, mtype);
     MPI_Recv(&postumis, 1,             MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&eilutes,  1,             MPI_INT, source, mtype, MPI_COMM_WORLD, &status);

     MPI_Recv(&a, eilutes*AStulp_BEilut, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&b, AStulp_BEilut*BStulp,  MPI_INT, source, mtype, MPI_COMM_WORLD, &status);

     //printf ("postumis =%d\n", postumis);
     //printf ("eilutes =%d\n", eilutes);
     //printf ("a[0][0] =%e\n", a[0][0]);

     for (k=0; k<BStulp; k++) {
        for (i=0; i<eilutes; i++) {
           c[i][k] = 0.0;
           for (j=0; j<AStulp_BEilut; j++) {
              c[i][k] = c[i][k] + a[i][j] * b[j][k];
           }
        }
     }

     mtype = FROM_WORKER;
     MPI_Send(&postumis, 1,      MPI_INT,    MASTER, mtype, MPI_COMM_WORLD);
     MPI_Send(&eilutes,  1,      MPI_INT,    MASTER, mtype, MPI_COMM_WORLD);
     MPI_Send(&c, eilutes*BStulp,MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  }

  MPI_Finalize();
}
```