KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

JAVA GIJŲ PANAUDOJIMO LYGIAGREČIOMS PROGRAMOMS TYRIMAS

Rolandas Kašinskas

Java gijų panaudojimo lygiagrečioms programoms tyrimas

Magistro darbas

Darbo vadovas
doc. Romas Marcinkevičius

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Rolandas Kašinskas

Java gijų panaudojimo lygiagrečioms programoms tyrimas

Magistro darbas

Vadovas
doc. Romas Marcinkevičius
2010-05

Recenzentas
doc. dr. S.Maciulevičius
2010-05

Atliko
IFM-4/2 gr. stud.
Rolandas Kašinskas
2010-05-15

Kaunas, 2006

BAIGIAMOJO DARBO TURINYS

LENTELIŲ TURINYS

PAVEIKSLIUKŲ TURINYS

1. ĮVADAS

1.1. Dokumento paskirtis

Kam reikia lygiagretaus programavimo? Tam, kad išnaudoti geriau resursus, kad veiksmus atklikti greičiau. Pavyzdžiui, turime masyvą tūkstančio elementų ir jame yra įvairūs skaičiai. Mums reikia rasti didžiausia skaičių. Sakykime algoritmas be gijų jį visą patikrina per 2 sekundes. Dabar paimkime tą patį masyvą ir įvykdykime jį lygiagrečiame algoritme, kuris pasidalina masyvą į dvi dalis ir kiekviena gija ieško savo dalyje. Tai puse masyvo patikrintų per 1 sekundę, o kadangi dvi gijos ieško, tai jį visą patikrins maždaug per 1 sekundę, aišku jei sistema turės kelių branduolių sistemą.

Dokumentas skirtas suprasti, kaip efektyviai galima panaudoti gijas ir koks jų efektyvumas su vienu ar keliais procesoriaus branduoliais. Leis lengviau įvertinti ar verta panaudoti gijas? Ar verta įsigyti procesorių su daug branduolių. Kadangi kuriant lygiagrečias sistemas kiekvienam iškyla klausimas, kaip parašyti programą efektyviai, kad ji veiktų kiek galima greičiau ir išnaudotų resursus efektyviau. Dokumente supažindinama su esminėmis lygiagretaus programavimo problemomis. Supažindinama su daugiausiai naudojamais jų sprendimais. Apžvelgiama kas labiausiai įtakoja lygiagretųjį programavimą. Supažindinama su esamais sprendimais, nors tokių sprendimų viešam naudojimui nėra platinama. Taip pat autoriaus sukurta programa, skirta išanalizuoti gijų įtaką algoritmui. Supažindinama, kaip veikia ši programa, ką su ja galime ištirti, kuo ji mums naudinga.

Susipažinus su duotuoju dokumentu bus lengviau galima įvertinti gijų naudą kuriamom sistemoms. Bus lengviau įvertinti ar verta įsigyti naują procesorių su daugiau branduolių. Taip pat bus paprasčiau įvertinti naujas situacijas, kad būtų galima pagerinti sistemos darbą ir bus supažindinta su programa, kuri leis lengvai patikrinti kuriamos sistemos ar algoritmo efektyvumą su skirtingu skaičiumi gijų.

Santrauka

Why to use concurrent programing? Mostly for optimizing computer resources, to optimize programs, that they use the resources better and more efficient. These days most computers have processors with two and more cores and most programs don't use them efficient.

Java threads are easy to use, but they have some problems, like deadlocks, livelocks, starvation. In this document we analize what problem solution are better for the problems, locks or semaphores and when which solution to use. Another thing that is important is how many threads to use in different algorithms and what influence have they on the computer resources and run time. And of course is there a difference between threads in algoritms that make the same work and different work, what influence in runtime and resources usage has it? All the analysis is made by a specialy designed application directly for the purpose of monitoring. It can monitor the use of resources and runtime of a given application many times and give you the results. It helps to evade the problem of a single test that can be faulty.

All this analysis helps us to decide when to use more threads, when to use less. When it's time to get a multicore processor. What solutions to use. All this helps us to make better and faster decisions when we program our applications.

2. ANALITINĖ DALIS

2.1. Įvadas

Java programavimo kalba buvo viena iš pirmųjų, kuri suteikė galimybę programuotojams lengvai ir paprastai pasinaudoti lygiagretaus programavimo galimybėmis, kaip gijos, ir kurti efektyvesnes sistemas. Bet taip pat, kadangi šitą galimybę Java suteikė nuo pat pradžių, likusios tam tikros problemos. Kur slypi problemos, kokios jos yra? Gijos suteikia galimybę norimą sistemos darbo veiksmą išskaidyti į kelis mažesnius darbus. Vykdyti darbus lygiagrečiai, vykdomi keli darbai vienu metu toje pačioje programoje, bet taip pat jie naudoja tuos pačius resursus, vykdo nuskaitymą ir rašymą tuo pačiu metu. Taip atsiranda problemos, kurių nematysi programose, kurios nenaudoja gijų. Kai kurių problemų nepastebėsime sistemose su vieno branduolio procesoriumi, nes gijos nevykdomos vienu metu, o palaipsniui tai viena, tai kita. Bet dauguma naujų kompiuterių turi kelis branduolius, tai reiškia, kad gijos gali būti vykdomos lygiagrečiai. Kas atsitiks, jei turėsime programą su dviem gijomis? Pirma vykdant nuskaitys duomenis, o kita tuo metu įrašinės? Kas atsitiks, ar pirma nuskaitys senus duomenis, naujus, ar tiesiog jų kombinaciją? Tai sunku nuspėti, nes kiekvieną kart vykdant galima gauti vis skirtingą rezultatą. Bet tai tik viena problema, jų yra ir daugiau. Taip pat lygiagrečios sistemos išsiskiria pagal atminties paskirstymą, kaip sistemos su bendra ir paskirstyta atmintimi. Sistemos, naudojančios bendrą atmintį, vėlgi gali susidurti su problemomis, kaip aklavietė, badas. Programos, naudojančios paskirstytą atmintį, susiduria su problemomis, kad reikia tinkamai padalinti užduotis visoje sistemoje, reikia gauti visus rezultatus ir juos apjungti. Taip pat kuriant tokias sistemas galima susidurti ir su daugybę kitų problemų, kaip gijų prioritetai. Ar vertas juos keisti, kaip užtikrinti, kad pakeitus prioritetą nebus sudaryta badavimo problema. Kaip sistema veiks greičiau? Ar kai programos gijos atlieka tą patį darbą, ar kai skirtingą? Kas efektyviau, kas naudoja mažiau resursų? Dar iškylantis klausimas, ar kurti naują procesą ar procesui gijas. Mes turime įvertinti ir Java versiją [11]. Kaip ir minėta, daug problemų iškyla pasirenkant tinkamą metodą, būdą, sprendimą. Taigi kaip apsispręsti? Tai mes ir bandysime išsiaiškinti.

2.2. Tikslai ir uždaviniai

Šiuo metu jau dauguma turi kompiuterius su procesoriais, kurie turi kelis branduolius. Multi-procesorinės mašinos tampa standartu didėjant greičiui, nors vieno procesoriaus greitis mažėja [12]. O kas naudoja Java programavimo kalbą, net neįsivaizduoja, ar efektyviai Java išnaudoja procesorių, ar programa turi pakankamai gijų geram efektyvumui. Kaip išnaudojami branduoliai? Norint nustatyti visus šiuos dalykus, reikia išmanyti programavimą. Yra ir kiti sprendimai, bet ne Java programavimo kalbai. Biblioteka, padedanti pasinaudoti multi-branduoliniais procesoriais nežinant ir nebūnant gijų ekspertų [13]. Bet tai Intel produktas, skirtas C++ programavimo kalbai. Todėl mes vykdysime tyrimą.

AMD atliktas tyrimas apie branduolius ir programinę įranga atskleidė „Ar daugiau branduolių tikrai geriau?“[3]. Pirmi testai parodė, kad paprastos užduotys buvo atliekamos mažiau efektyviai, nei tikėtasi. Kai kurios net buvo atliekamos lėčiau. Kai kurie skubiai kaltino procesorių architektus. Bet paaiškėjo, kad problema buvo programinė įranga. Kaip ji suprogramuota, nes joje užduotys negali būtį efektyviai padalintos keliems branduoliams. Kaip išsiaiškinta, dauguma programų, kurios veikė ant vieno branduolio, ir šiomis dienomis veikia ant vieno branduolio dėl programinės sistemos struktūros.

Atliksime analizę svarbiausių ir dažniausiai iškylančių klausimų, kurie turėtų palengvinti programuotojui darbą, palengvinti jo apsisprendimą, kokį sprendimą pasirinkti, kada sistema yra efektyvesnė, su kiek gijų, su keliais branduoliais veikia greičiau. Bus taip pat pademonstruota, kaip lengvai galima nustatyti, koks gijų skaičius yra tinkamiausias.

Atliksime tyrimą, kuriame tirsime:

- Kaip kinta vykdymo laikas ir išnaudojami kompiuterio resursai priklausomai nuo (g, b) porų, kur g - gijų skaičius, b – branduolių skaičius;
- Java lygiagretaus programavimo esminės problemos (aklavietės, badas, amžini ciklai);
- Gijų pobūdžio įtaka (kai gijos atlieka tą patį, kai atlieka skirtingą darbą).

2.3. Lygiagretaus programavimo priemonių tipai

Lygiagretaus programavimo priemonėmis yra sprendžiamos įvairių tipų užduotys. Taip pat lygiagretūs uždaviniai gali būti atliekami skirtingose platformose ar įrenginiuose. Dėl šių priežasčių sukurtos programavimo priemonės skiriasi savo veikimo principais.

2.4. Atminties naudojimas

Bet kokio algoritmo vykdymui yra reikalingi duomenys. Nesvarbu, ar algoritmas yra vykdomas nuosekliai ar lygiagrečiai. Duomenys gali būti saugomi bendroje atminties vietoje, arba atskiruose atminties moduliuose, skirtinguose kompiuteriuose. Pagal tai, kokio tipo duomenų saugojimas yra naudojamas, priklauso architektūrinis sistemos sprendimas.

2.4.1. Bendra atmintis

Lygiagrečiuosiuose skaičiavimuose architektūra, naudojanti bendrą atminties vietą yra vadinama bendros atminties architektūra. Sistemos, paremtos tokio tipo architektūra, naudoja bendrą atminties vietą duomenims saugoti [5]. Kadangi skaičiavimai yra atliekami lygiagrečiai, o naudojama atmintis yra bendra, tai susiduriama su problemomis. Tokio tipo sistemose reikia užtikrinti duomenų saugumą. Vienu metu kelios skirtingos gijos gali bandyti manipuliuoti tais pačiais duomenimis, o tai gali sąlygoti klaidingus skaičiavimus ir jų rezultatus.

2.4.2. Paskirstyta atmintis

Kita pagrindinė lygiagrečiųjų skaičiavimų atminties saugojimo architektūra paremta schema, pagal kurią duomenys yra saugomi atskiruose atminties moduliuose. Tokia paskirstytos atminties architektūra yra kuriama sujungiant kiekvieną komponentą didelės spartos duomenų kanalais duomenų apsikeitimui. Dažniausiai šio tipo sprendimai yra naudojami vykdant lygiagrečius skaičiavimus paskirstytose sistemose, sujungtose į bendrą tinklą [5]. Paskirstytos atminties sistemų kūrimo metu iškyla kelios pagrindinės problemos. Pirmoji problema reikalauja tinkamai paskirstyti skaičiavimų užduotis skirtingiems procesoriams su atskiromis duomenų saugyklomis. Kita problema – surinkti apskaičiuotus rezultatus iš atskirų komponentų ir gautą informaciją sujungti į galutinį rezultatą [6]. Paskirstytos atminties algoritmai gali būti vykdomi kompiuterių paskirstytuose tinkluose – į tinklą greitais duomenų kanalais sujungtais kompiuteriais. Jei algoritmas parašytas teisingai, nesunkiai galima padidinti jo efektyvumą išplečiant tokio tipo sistemas. Projektas „Beowulf“

jungia Linux operacinės sistemos klasterius, naudojančius MPI lygiagrečiųjų skaičiavimų technologiją [4]. Ar lengva sukontroliuoti paskirstytų sistemų resursus ir apkrovimą [9]?

2.5. Operacinės sistemos lygmuo

Gijos gali veikti skirtinguose operacinės sistemos lygmenyse. Pagal tai, kokius sistemos resursus gijos gali naudoti ir kuriame operacinės sistemos lygmenyje jos veikia, jos yra skirstomos į kelias rūšis:

- procesus;
- branduolio lygmens gijas;
- vartotojo lygmens gijas;

2.5.1. Procesai

Procesas – paprasčiausias vienetas, kuris gali būti laikomas atskira programa. Jis turi atskirą atminties vietą. Procesas gali sukurti savo vaikinius procesus. Pagrindinis procesų modelio privalumas – visiškas būsenos atskyrimas. Kaip rašyta aukščiau, procesas turi savo atskirą atminties vietą, turi priėjimą prie operacinės sistemos globalių kintamųjų, atminties ir įvesties/išvesties valdymo, palaiko signalus. Tačiau lygiagretiems skaičiavimams visiškas būsenos atskyrimas nėra pliusas, o atvirkščiai – sukelia daugiau nepatogumų. Nepatogumai kyla, siekiant realizuoti bendradarbiavimą tarp procesų, nes čia nėra naudojama bendra atmintis. Procesai „bendrauti“ gali tik naudodami ganėtinai ribotus įrankius: Unix sistemų vamzdžius (angl. p*ipes*) ar lizdus (angl. s*ockets*). Dar blogiau, kai kurios operacinės sistemos turi nustatytą maksimalų galimų sukurti procesų limitą. Šios ir kitos priežastys paskatino gijų atsiradimą.

2.5.2. Branduolio lygmens gijos

Branduolio lygmens gijos yra realizuotos operacinės sistemos branduolyje. Jos gali būti pasiekiamos naudojantis tam tikromis bibliotekomis. Linux sistemoje yra naudojama POSIX standarto sąsaja – pthreads. Šio tipo gijas aptarnauja operacinės sistemos planuotojas (ang. *Scheduler*). Gijos yra aptarnaujamos sparčiau nei procesai. Gijų kūrimas, skirtingai, nei procesai, nereikalauja naujos atminties vietos išskyrimo, nes naudoja tą pačią atminties adresų sritį [9]. Dėl to jų sukūrimas užtrunka mažiau. Dauguma operacinių sistemų palaiko branduolio lygio gijas. Šio lygmens gijos dar gali būti skirstomos į du tipus:

- Branduolio gijos – veikia vartotojo lygmenyje kol vykdo vartotojo aprašytas funkcijas ar dirba su bibliotekomis. Tačiau dirbant su operacinės sistemos šaukiniais (ang.

system calls), persijungia į branduolio lygmenį.

- Tik-branduolio gijos – visą laiką dirba operacinės sistemos branduolio lygmenyje. Programuotojas dirbti su šiomis gijomis gali pasinaudodamas branduolio servisais [2].

2.5.3. Vartotojo lygmens gijos

Tai vartotojo aplinkoje veikiančios gijos, dar vadinamos „žaliosiomis gijomis“ (angl. *Green threads*). Skirtingai nei branduolio lygmens gijos, vartotojo lygmens gijos yra vykdomos vartotojo kode, dažniausiai specialios bibliotekos. Šio tipo gijos dažniausiai yra naudojamos dinaminių programavimo kalbų, tokių kaip Java ar Ruby. Java gijos yra emuliuojamos virtualioje mašinoje (JVM). Operacinės sistemos branduolys nežino apie šių gijų egzistavimą ir jų nevaldo. Atvirkščiai – branduolys mato paleistą virtualią mašiną kaip vieną giją. Tačiau iš to galime pastebėti šių gijų privalumus. Kadangi vartotojo lygmens gijos nėra susietos su branduoliu, tai jų skaičius nėra ribojamas. Galima paleisti tiek gijų, kiek leidžia sistemos resursai arba kiek priskirta konkrečiai programai. Gijų valdymas daug paprastesnis nei branduolio lygmens – jos gali būti emuliuojamos sistemose, kuriose nėra branduolio lygmens gijų palaikymo [9]. Vienas pagrindinių trūkumų – šios gijos neišnaudoja kelių procesorių ar kelių branduolių procesorių sistemų. Kitas trūkumas – nebendradarbiavimas su operacinės sistemos branduoliu. Dėl šios priežasties vartotojo lygmens gijos negali gauti atskirų signalų iš branduolio arba iškviesti sisteminių šaukinių. Taip pat, baigus vykdyti giją, branduolys negali duoti signalo pradėti vykdyti kitą vartotojo lygmens giją.

2.6. Esminės problemos

Aklavietė („deadlock“) – tai problema, kai dvi ar daugiau gijų laukia resurso, kurį turi atlaisvinti kitą gija [10]. Java virtualioji mašina neaptinka aklaviečių ir net jei galima suprogramuoti kažką, kad rastų galimas aklavietes, kol kas tokių įrankių nėra.

Kaip paprasčiausiai išspręsti aklavietės problemą? Tam reikia laikytis paprastos taisyklės, kai kažkokia gija naudoja užrakinimą („lock“) jokia kita gija tuo metu negali kviesti metodų, kuriems reikia to pačio užrakinimo taip pat [1]. Bet tai nėra labai praktiška ir patogu. Kitas būdas yra aukštesnio lygio resursų užrakinimas, kuris gali sumažinti gijų efektyvumą [1]. Užrakinus pasirinktą objektą, kuris turi daug kintamųjų, mums reikia pakeisti tam tikrus kintamuposius, bet gal pasirinkta gija keis tik vieną kintamąjį, gal kitos gijos tuo metu galėtų keisti kitus kintamuosius? Tokiu atveju kitos gijos privalės laukti, kol pirmoji įvykdys viską ir

vėl atrakins objektą. Praktiškiausias atvejis yra naudoti hierarchiją užrakinimuose. Taip užraktas apsaugo ne tik save ir savo resursus, kintamuosius, bet ir žemiau jo esančius užraktus. Dar viena išeitis yra laukimas tam tikrą laiką („timeouts“). Bet tai nėra taip paprasta, kaip skamba. Reikia paruošti sistemą, kad jei per tam tikrą laiką negavome reikiamų resursų, gija atlaisvintų savo resursus, atstatytų juos į būsenas prieš paėmimą, nurodytų kitoms gijoms, kad taip reikės vėl atlikti. Žinoma tai suteikia daugiau galimybių, bet taip pat tai yra sudėtingiau.

Amžinas užraktas („livelock“) – tai problema, kuri atsiranda dėl gijų tarpusavio veiksmų:
Amžinas užraktas, kai gijos veikia, bet darbas nėra atliekamas, pvz, jei viena gija kažką pakeitė, o antra atkeitė ir taip amžinai.

Badavimas („starvation“) – tai gijų badavimas, kai giją negauna priėjimo prie procesoriaus resursų ir ji niekad nevykdoma. Bado sprendimas vadinamas, teisingumo („fairness“), kai visos gijos gauna vienodą šansą būti įvykdytom.[10]

Java kalboje dažniausios trys situacijos, kurios gali privesti prie bado problemos:

- Gijos su aukštu prioritetu sunaudoja visą procesoriaus laiką, o gijos su žemu prioritetu negauna laiko vykdymams.

  Gijų prioritetus galima nustatyti nuo 1 iki 10. Kuo aukštesnis prioritetas, tuo daugiau vykdymo laiko gaunama. Paprastai daugumai programų rekomenduojama nekeisti prioritetų.

  Reikia nepamiršti programuojant Java, kad prioritetų tvarka buvo pakeista tarp Java 5 ir 6 versijos.

- Gijos užblokuojamos amžinai, laukdamos įėjimo į sinchronizuotą bloką, kai kitos gijos įleidžiamos, o kitos blokuojamos.

  Java neturi jokios griežtos eilės, kaip patekti į sinchronizavimo bloką ir tai gali iškelti prieš tai minėtą problemą.

- Gijos laukia objekte („wait()“) ir laukia, kol bus sužadintos, nes kitos gijos sužadinamos visada pirmiau.

  Funkcija „notify()“ nesuteikia jokios garantijos, kuri gija bus sužadinta, tai gali būti bet kuri gija, kuri iškvietė metodą „wait()“. Taip gali iškilti problema, jei gija niekad nebus sužadinta.

Kaip įgyvendinti teisingumo sprendimą? Java neleidžia šimtu procentų įgyvendinti šito sprendimo, bet mes vis tiek galime padidinti teisingumą tarp gijų. Pirmiausia vietoj sinchronizacijos blokų vertėtų naudoti užrakinimą („lock“). Antra, reikia laukimą padaryti taip pat teisingesnį įvedant savo sukurtą eilę, į kurią būtų įrašomos visos laukiančios gijos ir vis sužadinama pirma eilėje. Taip išvengsime galimų problemų. Tai žinoma gali atsiliepti greitaveikai, bet tai priklausys nuo to, kaip mes įgyvendinsime užrakinimo ir laukimo funkcijas reikiamuose kodo vietose.

2.7. Priemonės

2.7.1. Java

JAVA lygiagretumą palaiko gijų pagalba. JVM gali vienu metu vykdyti daug gijų. Sukurti giją JAVA kalboje galima vienu arba dviem metodais:

- Naudojant java.lang.Thread klasę ir perdengiant run() metodą.
- Sukuriant naują paleidžiamą interfeisą.

Galima paleisti daug gijų, kurios aptarnauja vieną metodą, tam reikia, kad gijos būtų sinchronizuotos. Gijų komunikavimui dėl užraktintos sekcijos naudojamas semaforas. Vienas iš semaforų variantų yra „mutex“. Viena gija, vienu metu, gali turėti objektą. Kiekviena kita gija, bandanti prieiti prie objekto, bus blokuojama ir turės laukti. Jei laukia kelios gijos, tik viena gija prieis prie objekto.

Taip pat svarbus dalykas yra wait/notify komandos, tai palaukti ir pranešti komandos. Wait, priverčia laukti savo eilės, kol kita gija įvykdys kažkokią tai užduotį, kurios laukia gija su wait. Tada paleidžiamas notify, kad pranešti tai gijai.

2.7.2. Java CSP

Pagrindiniai tipai CSP bibliotekoje yra procesai ir įvairios komunikacijos formos tarp jų. Viskas CSP bibliotekoje yra procesas, net tinklas iš subprocesų. Bet tarp šių procesų nėra tiesioginio komunikavimo, visas komunikavimas vyksta tik per CSP sinchronizavimo objektus, tokius kaip kanalai ar įvykių barjerai, prie kurių prieina procesai [7].

Naudojant CSP biblioteką, priešingai nei Java standartinę biblioteką, nekyla tokių problemų, kaip Deadlocks ar Livelocks. Nes Java, naudodama gijas, kviečia kažkokį tai metodą, tada kada reikalinga, bet esmė tokia, kad visos gijos gali kviesti atitinkamą metodą vienu metu, taip sudarydamos tokias problemas, o procesai kiekvienas turi savo metodus.

Komunikacija per kanalus ir procesų tinklus

Paprasčiausias būdas komunikacijai yra įrašant ir nuskaitant duomenis iš kanalo. Galimi keli tipai kanalų, kaip:

- vienas-vienam
- vienas-betkuriam
- betkuris-vienam
- betkuris-betkuriam.

Procesai CSP gali būti susieti viens su kitu, arba vienas su daug procesų taip sukurdami tinklą, kuris vėlgi gali būti interpretuojamas kaip procesas ir jungiamas toliau.

JCSP Procesai

JCSP kalboje procesas yra klasė, kurioje įtraukta CSProcess sąsaja.

CSProcess sąsaja

```
package jcsp.lang;
public interface CSProcess
{
   public void run();
}
```

Taigi Java CSP biblioteka turi tikrai didelį privalumą prieš Java standartinę biblioteką. Nes išvengiama daug problemų. Viskas yra atskirta.

2.8. Galimi sprendimai

2.8.1. Sprendimai Lietuvoje

Kaip ir likusiame pasaulyje, taip ir Lietuvoje, nepavyko rasti panašią problemą nagrinėjančios programinės įrangos. Tačiau tai anaiptol nereiškia, jog problema nėra aktuali. Didžiuosiuose Lietuvos universitetuose yra dėstomi moduliai, supažindinantys su lygiagrečiuoju programavimu. Kiekvienais metais studentai yra supažindinami su lygiagretaus programavimo priemonėmis. Iš to galima spręsti, jog problema yra tikrai aktuali Lietuvoje ir ją reikia spręsti.

2.8.2. Sprendimai pasaulyje

Lygiagretusis programavimas įgauna pagreitį vis plintant kompiuteriams su kelių branduolių procesoriais. Todėl auga poreikis programinės įrangos, skirtos kurti ir testuoti lygiagretaus programavimo algoritmus. Tačiau tokio tipo programinės įrangos kūrimas yra sudėtingas, tad ir pasiūla nėra didelė. Nepavyko rasti analogišką problemą sprendžiančios programinės įrangos. Tai visiškai nereiškia, jog tokios nėra. Iš to galima spręsti, kad lygiagrečiųjų priemonių palyginimo įranga dar tik kūrimo stadijoje ir nėra plačiai taikoma.

Intel korporacija turi įrankius, skirtus tirti lygiagrečias programas, jų veikimą su atitinkamais skaičiais branduolių, bet sistemos nėra viešai prieinamos ir yra naudojamos tik pačioje įmonėje.

2.8.3. Vartotojo problemos

Kai kurios technologijos, kurios galės būti tiriamos, naudojant programinę įrangą, skirtą lygiagrečių algoritmų analizei, yra skirtos skirtingoms operacinėms sistemoms. Dažnai, konkrečioje operacinėje sistemoje veikiančios lygiagretaus programavimo bibliotekos turi atitikmenis kitose operacinėse sistemose, tačiau vistiek tai yra skirtingos bibliotekos, nors ir panašiai (ar taip pat) realizuotos. Tai gali sukelti skaičiavimo ir algoritmų vykdymo netikslumų arba išvis neleisti vykdyti palyginimo.

Lygiagretieji algoritmai, ypač sudėtingesni, ne visada yra vykdomi analogiškai. Tai nutinka dėl įvairių priežasčių: kompiuterio apkrovimo, atsitiktinių faktorių. Ne visada vienodai vienos gijos pabaigia skaičiavimus anksčiau už kitas. Dėl šių priežasčių kelis kartus vykdomi analogiškų skaičiavimų matavimai gali pateikti skirtingus rezultatus. Rekomenduojama matavimus vykdyti kelis kartus.

Programa negali užtikrinti, kad analizuojamas algoritmas yra parašytas teisingai. Ji negali nuspręsti, ar algoritmas užstrigo kritinėje sekcijoje, ar tiesiog yra lėtai vykdomas. Vartotojas pats privalo nuspręsti, kada skaičiavimai yra vykdomi pernelyg ilgai. Tokiu atveju gali nutraukti vykdymą. Taip pat, pridėdamas naujus algoritmus, programuotojas pats atsako už jų korektiškumą.

2.8.4. Esminis sprendimo aspektas

Didžiausias lygiagretaus programavimo iššūkis yra, kaip suskaldyti užduotį, kad veiktų lygiagrečiai [8]. Taigi reikia visas užduotis paruošti taip, kad jos lengvai būtų padalinamos tarp procesoriaus branduolių. Kad gerai veiktų, su atitinkamais skaičiais branduolių ir gijų. Kaip tai pasiekti? Kaip visa tai išbandyti? Dažniausia tokiais atvejais

padeda praktika, jei esate daug dirbę su gijomis, jūs nujausite, kaip reikia skaidyti užduotis, kaip jas paruošti. O Jūs tik pradedate, ar susidūrėte su tuo pirmą kartą, kaip išbandyti, kas gerai veikia, kas ne? Kaip nustatyti, koks vienos ar kitos problemos sprendimo būdas yra geriausias? Pirmiausia reikia paruošti atitinkamą algoritmą, sprendžiantį problemą. Tada reikia paruošti pasirinktą algoritmą su visais įmanomais sprendimo būdais ir juos testuoti. Kad testavimas būtų pakankamai tikslus, reikia testavimą atlikti ne vieną kartą, kad išvengti galimų netikslumų, kurie gali kilti dėl kompiuterio procesoriaus apkrovimo, sistemos gaunamo procesoriaus laiko.

Išanalizuoti skirtingus problemų sprendimus ir užduočių padalinimus lengva pasitelkiant papildomą programinę įrangą, kuri įvykdys norimą algoritmą keletą kartų, bus galima vykdyti visus skirtingus problemų sprendimus ir vėliau juos palyginti, tiek grafiškai, tiek paprastų lentelių pagalba. Taigi, paruošus skirtingus dalinimo ir problemų sprendimus, mes juos galėsime vis vykdyti, patobulint ir vėl vykdyti, taip ieškant geriausia sprendimo. Programinė įranga padės mum visa tai ištestuoti su pasirinktu skaičium gijų ir branduolių.

2.9. Tyrimo tikslas

Tyrimas suteikia galimybę palyginti įvairių sprendimų efektyvumą, jo naudingumą panaudojant sistemose. O tai leis programuotojams lengviau pasirinkti tinkamą sprendimą ir taip pagerinti sistemų efektyvumą. Taip pat leis įvertinti, kiek verta investuoti į techninę įrangą, kaip multi branduolinis procesorius, operatyvioji atmintis ir pan. Tyrimas bus skirtas stebėti kaip kinta vykdymo laikas, kompiuterio resursų išnaudojimas. Taip pat kaip įtakoja įvairūs problemų sprendimai tuos pačius laiko ir resursų kriterijus.

Bus analizuojamas tokiais aspektais:

- Kaip kinta programos vykdymo laikas ir išnaudojami kompiuterio resursai priklausomai nuo (g, b) porų, kur g - gijų skaičius, b – branduolių skaičius.
- Java lygiagretaus programavimo esminių problemų (aklavietės, badas, amžini ciklai) efektyviausi sprendimo būdai. Palengvinti, kokį problemos sprendimą naudoti.
- Gijų pobūdžio įtaka (kai gijos atlieka tą patį, kai atlieka skirtingą darbą), kaip tai įtakoja vykdymo laiką ir reikalingus kompiuterio resursus.
- Gijų prioritetai – ar turi įtakos sistemos vykdymui, vykdymo laikui ir resursams, jei nesukelia bado problemos.

1 lentelė. Tyrimo aspektai

Tyrimo tikslas, problema	**Įvertinamas vykdymo laikas**	**Įvertinama kompiuterio resursų išnaudojimas**	**Įvertinamas problemos sprendimas**
Gijų ir branduolių skaičiaus įtaka	+	+	
Gijų problemos (aklavietės, badas, amžini ciklai)	+	+	+
Gijų pobūdžio įtaka	+	+	

3. PROJEKTINĖ DALIS

3.1. Projekto kūrimo pagrindas (pagrindimas)

Išlygiagretinimo metodų tyrimo sistemos vystymas yra sudėtingas ir daug laiko reikalaujantis procesas. Kuriant lygiagrečias sistemas reikia nustatyti, kuri technologija labiau tinka tam tikriems algoritmams, kuri kalba yra efektyvesnė. Kuriamas įrankis yra taikomas būtent šiai problemai išspręsti. Sistemos dėka galima pasirinkti, kiek kartų vykdyti konkretų algoritmą, algoritmo vykdymo technologiją ir taip gauti duomenis apie tos kalbos efektyvumą konkretiems skaičiavimams:

- Sugaištas vykdymo laikas;
- Vykdymų skaičius;
- Atminties sunaudojimas;
- Procesoriaus aprovimas;

3.2. Sistemos tikslai (paskirtis)

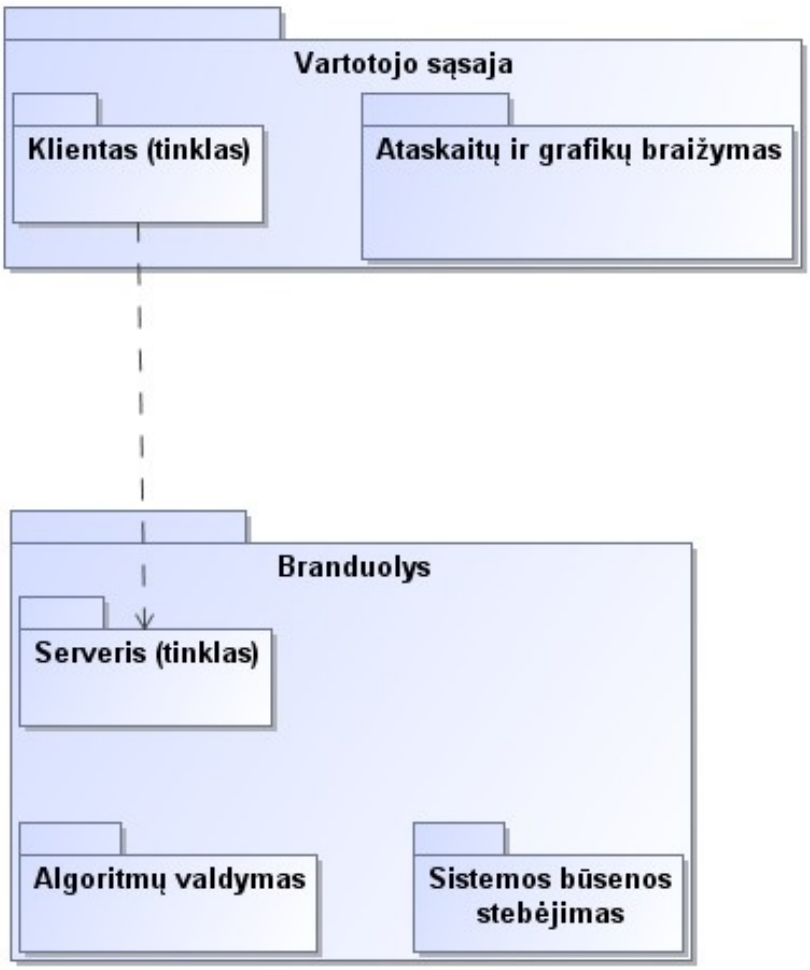
Projekto tikslas – sukurti pagalbinę priemonę, lygiagrečiųjų skaičiavimų technologijoms ir algoritmams palyginti, bei analizuoti jų veiklą. Toks įrankis leis greitai ir paprastai nustatyti tinkamą lygiagrečiųjų skaičiavimų technologiją konkrečiai problemai spręsti. Jis leis greitai ir efektyviai išbandyti savo algoritmą, jį paleidus kelis kartus su skirtinais parametrais. Atsižvelgiant į rezultatus bus galima pagerinti kuriamų lygiagrečių sistemų efektyvumą. Programinės įrangos galimybės:

- Lygiagrečiųjų skaičiavimų algoritmų vykdymas nustatant parametrus;
- Algoritmų parametrų (naudojamos atminties ir procesoriaus) matavimas ir fiksavimas;
- Išmatuotų parametrų pateikimas vartotojui;
- Naujų algoritmų įtraukimas;

3.3. Sistemos sudėtis

Sistema susideda iš dviejų pagrindinių dalių: serverio ir kliento, arba kitaip – vartotojo sąsajos ir branduolio. Abi sistemos dalys bendrauja tinklu, todėl abi turi įdiegtus tinklo modulius: „Klientas“ ir „Serveris“. Grafinė vartotojo sąsaja dar turi ataskaitų generavimo ir grafikų braižymo modulį, skirtą vartotojui pateikti analizės rezultatus suprantamesniu formatu. Sistemos branduolys, be tinklo apdorojimo modulio, turi dar du paketus: algoritmo valdymą – skirtą darbui su algoritmais (paleidimas, parametrų perdavimas, sąrašo skaitymas),

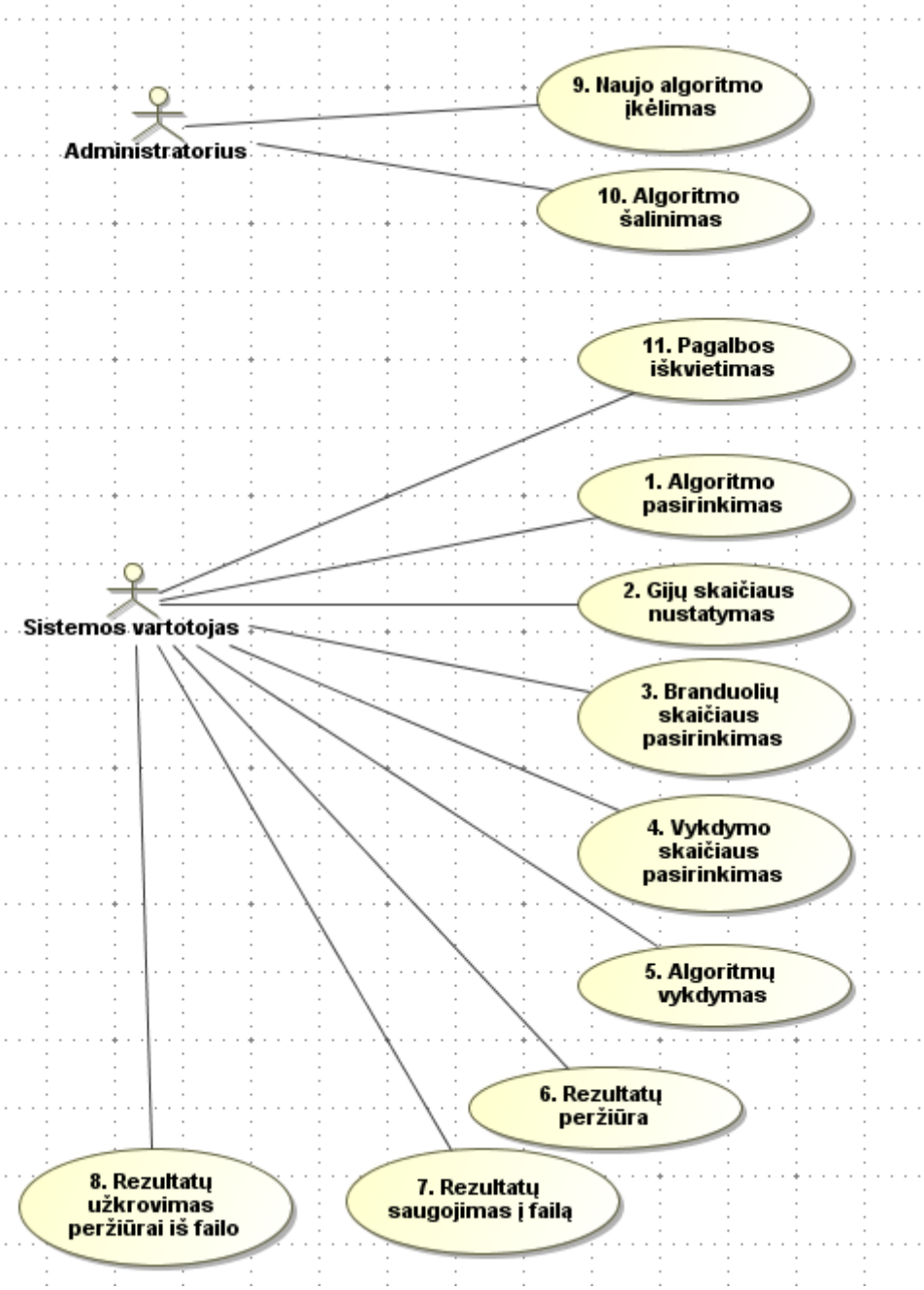
bei sistemos būsenos stebėjimą – skirtą gauti informaciją apie konkrečių procesų (vykdomų algoritmų) sunaudojamus operacinės sistemos resursus.



1 pav. Komponentų diagrama

3.4. Panaudojimo atvejų diagrama

Programinės įrangos panaudojimo atvejų diagrama pateikta paveikslėlyje (2 pav.).



2 pav. Sistemos panaudojimo atvejai

9 ir 10 panaudos atvejai nėra realizuojami programiškai. Diagramoje jie pateikiami tik siekiant aiškiau atvaizduoti sistemos funkcionalumą. Algoritmai įtraukiami į sistemą įkeliant juos į tam skirtus katalogus. Programa pati atpažįsta naujus algoritmus.

3.4.1. Reikalavimai sistemos išvaizdai

Vartotojo sąsaja turi būti:

- Lengvai skaitoma;
- Paprasta naudoti;
- Intuityvi, paprasta rasti norimas funkcijas;
- Spalvos turi būti patrauklios, „nerėžti“ akies;
- Neįkyri, kad nereiktų vis pakartotinai patvirtinti;

3.4.2. Reikalavimai panaudojamumui

Sistemos panaudojamumas:

- Neturi būti reikalingi kursai, kad išmokti naudotis sistema;
- Klaida sistemoje neturi sukelti rimtų pasekmių.

3.4.3. Reikalavimai vykdymo charakteristikoms

Sistemos charakteristikos:

- Sistema turi pateikti tiksliai vykdyme metu naudotų resursų ataskaitą;
- Rezultatų pateikimas turi būti atliktas per kuo trumpesnį laiko tarpą;
- Sistema turi būti patikima, kad jos rezultatais būtų galima pasikliauti;
- Sistema turi būti praplėčiama.

3.4.4. Reikalavimai veikimo sąlygoms

Sistemos turi:

- Veikti turimuose kompiuteriuose;
- Yra pateikti testo rezultatai su skirtingomis programavimo kalbomis realizuotų algoritmų;
- Netrikdyti kitų kompiuteryje esančių programų darbo;
- Būti greit paleidžiama bet kokiomis sąlygomis.

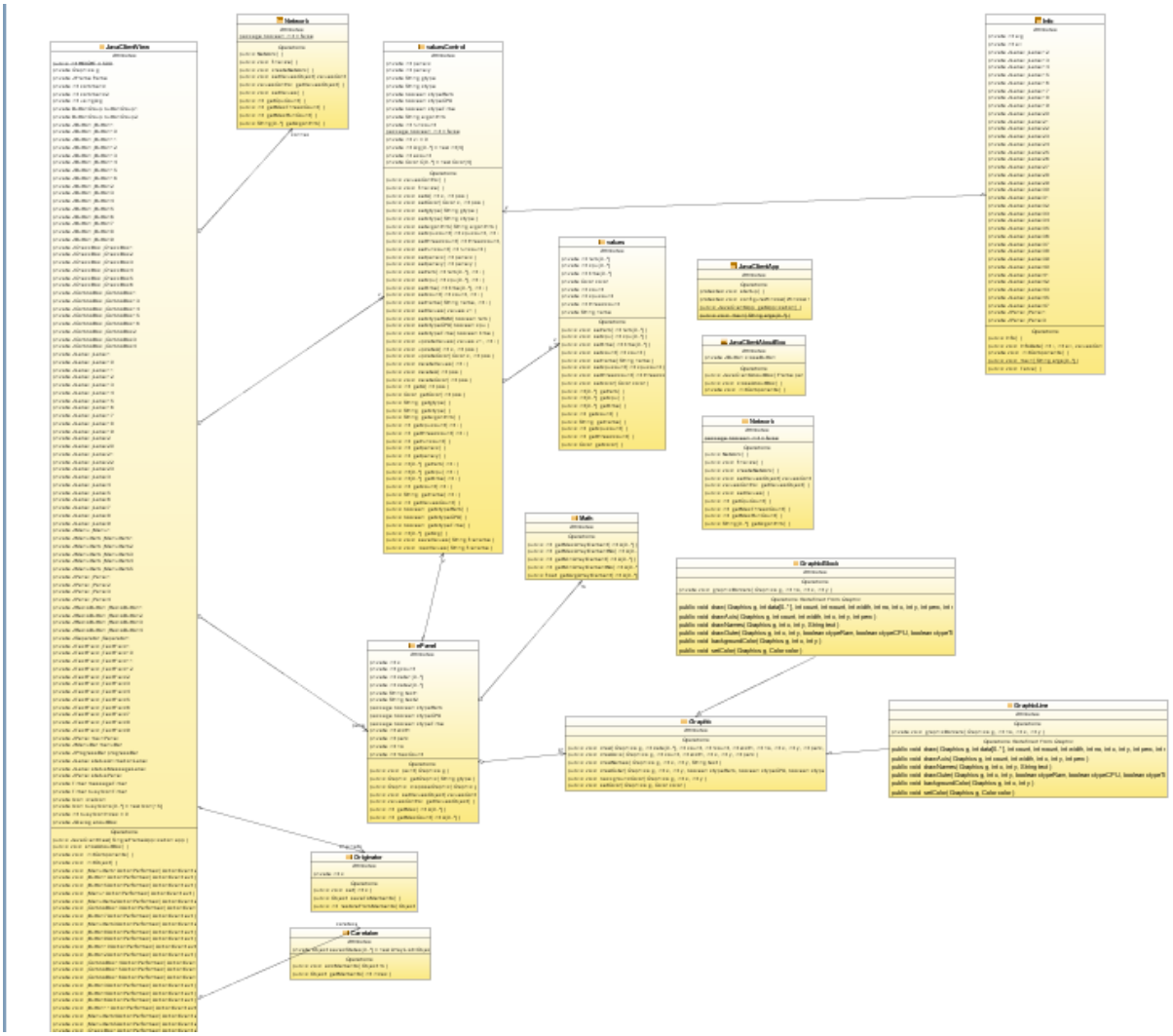
3.4.5. Reikalavimai sistemos priežiūrai

Sistema turi:

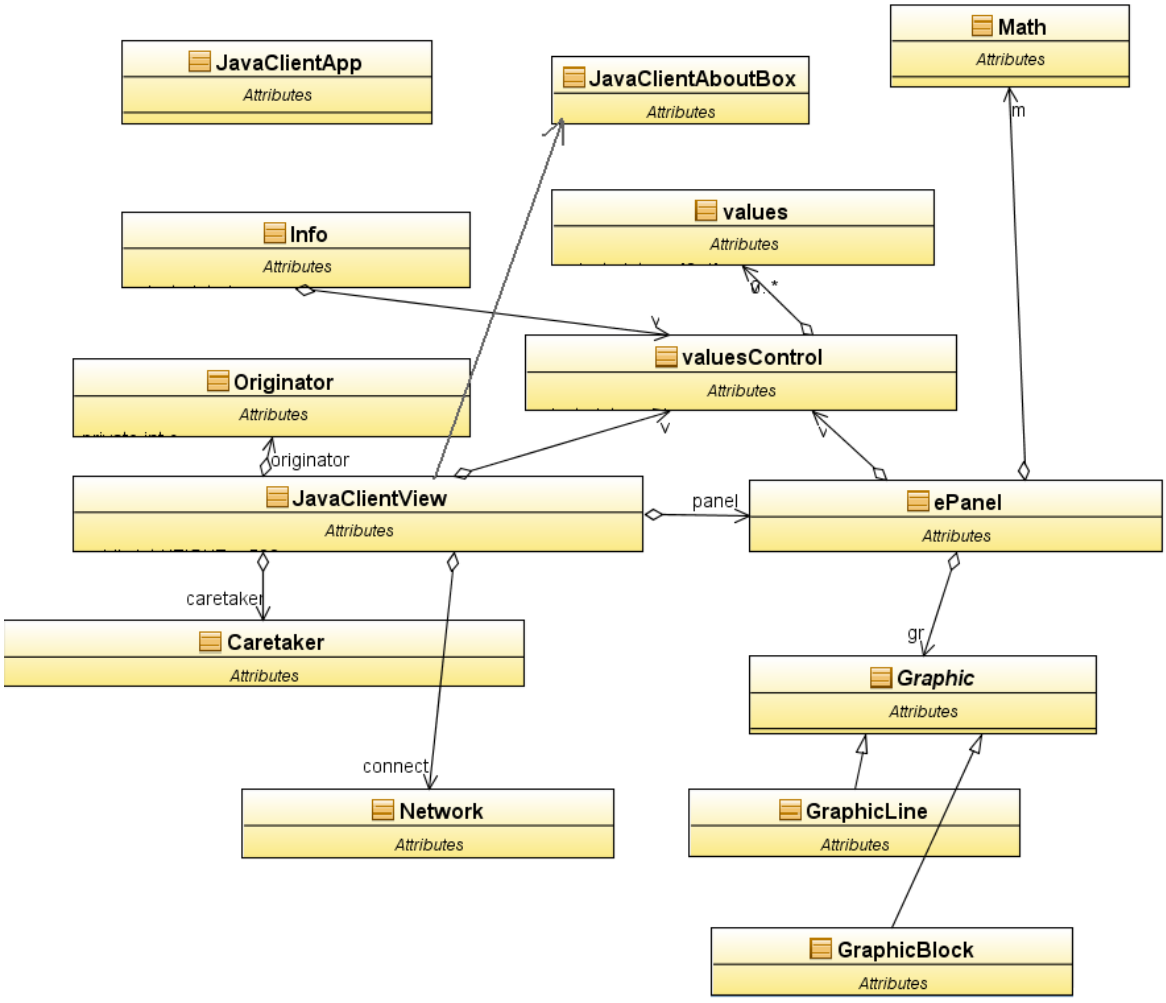
- Veikti įvairiose operacinėse sistemose, kaip Windows, Linux. (Vartotojo sąsaja)
- Būti lengvai papildoma naujais testavimo algoritmais.

3.5. Klasių diagrama

3.5.1. Klientinė dalis (vartotojo sąsaja)

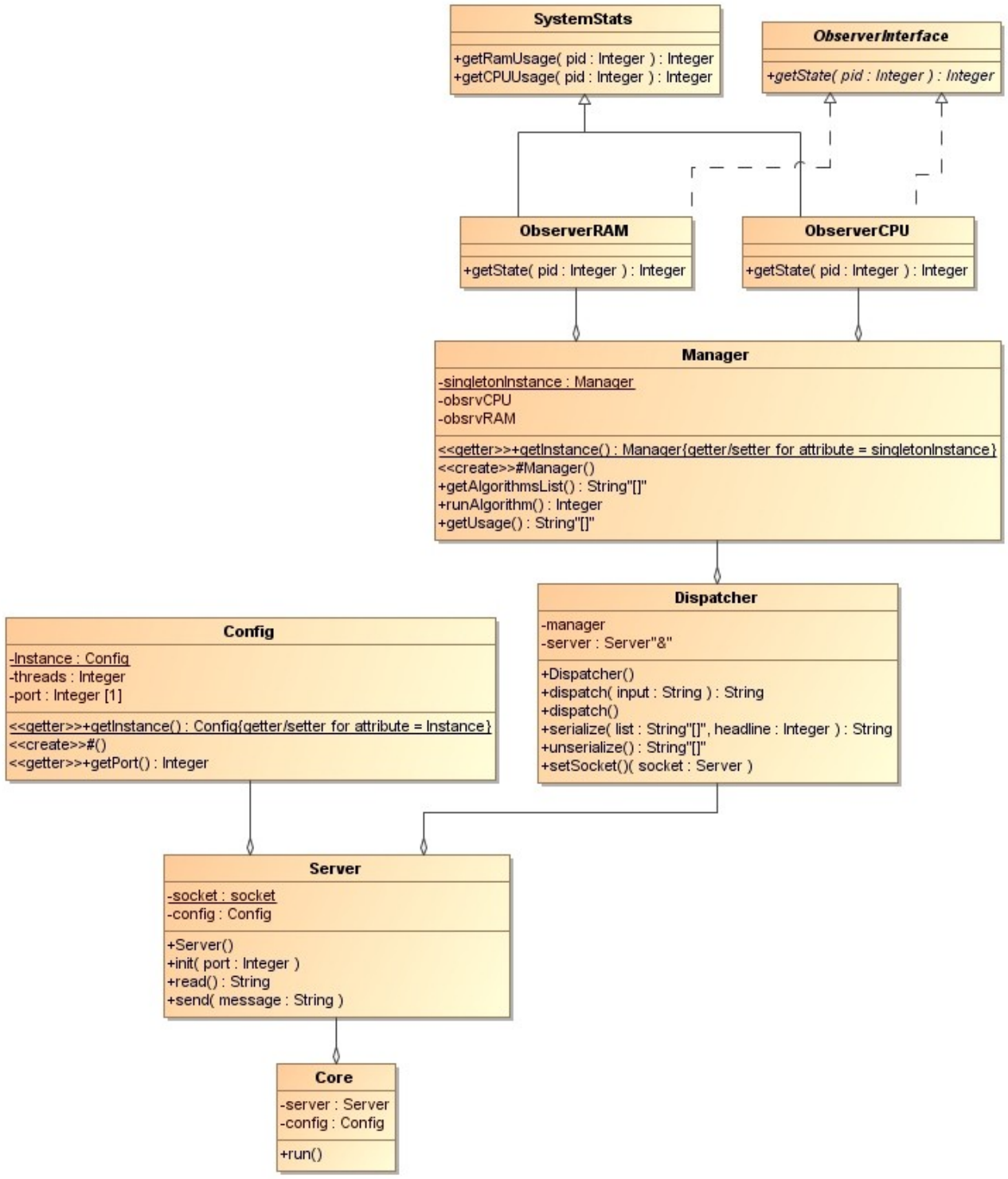


3 pav. Klientinės dalies klasių diagrama



4 pav. Klientinės dalies klasių diagrama sutraukta

3.5.2. Serverinė dalis

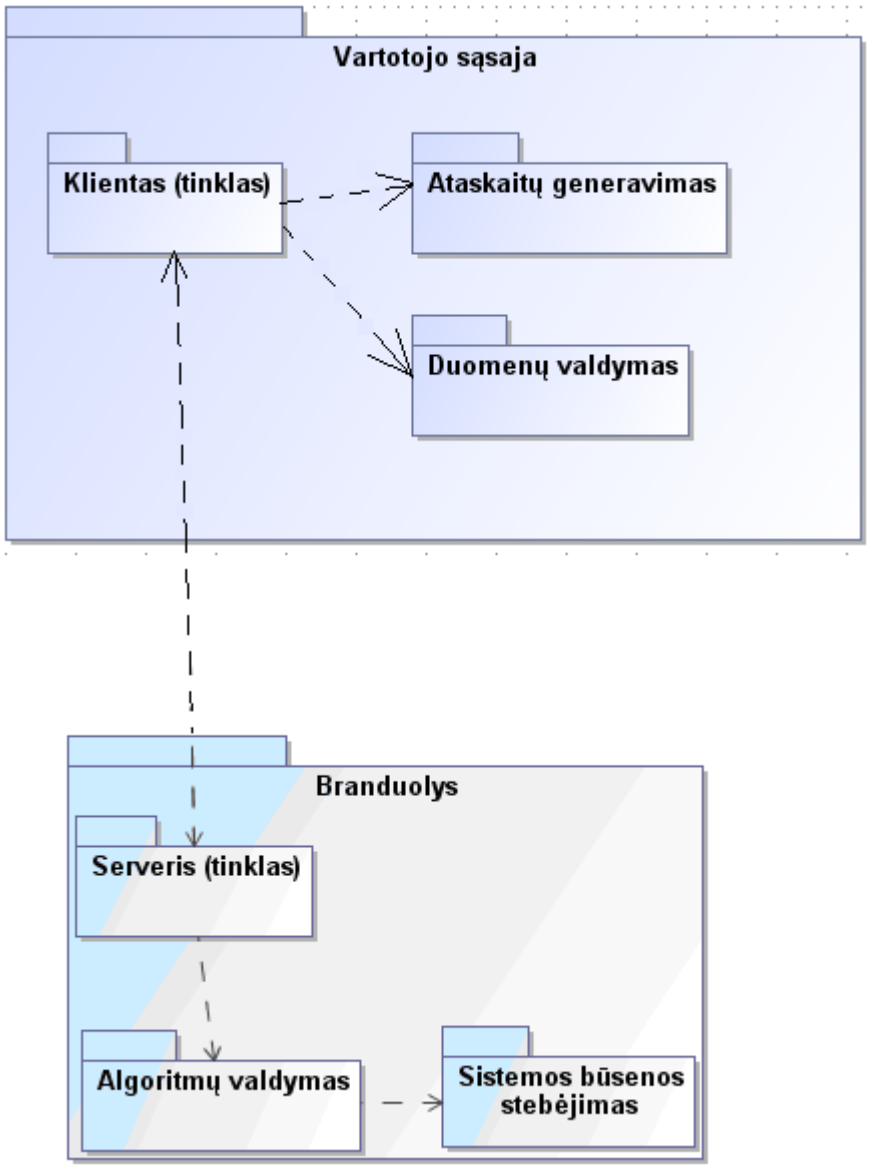


5 pav. Serverio klasių diagrama

3.5.3. Klasių paketai

Sistema susideda iš dviejų pagrindinių dalių: serverio ir kliento, arba kitaip – vartotojo sąsajos ir branduolio. Abi sistemos dalys bendrauja tinklu, todėl abi turi įdiegtus tinklo modulius: „Klientas“ ir „Serveris“. Grafinė vartotojo sąsaja dar turi ataskaitų generavimo ir

grafikų braižymo modulį, skirtą vartotojui pateikti analizės rezultatus suprantamesniu formatu. Sistemos branduolys, be tinklo apdorojimo modulio, turi dar du paketus: „Algoritmo valdymas“ – skirtą darbui su algoritmais (paleidimas, parametrų perdavimas, sąrašo skaitymas), bei „Sistemos būsenos stebėjimas“ – skirtą gauti informaciją apie konkrečių procesų (vykdomų algoritmų) sunaudojamus operacinės sistemos resursus.



6 pav. Klasių paketai

4. TYRIMO DALIS

Programinė įranga yra skirta lygiagrečių Java algoritmų įvairių problemų sprendimų efektyvumui nustatyti. Taip pat išanalizuoti, kaip kinta sistemos greitaveika, kai sistema veikia su daugiau gijų ir kaip veikia, kai sistemai suteikiama daugiau procesoriaus branduolių vykdyti algoritmus. Visa tai turi palengvinti vartotojui rasti geriausią variantą iš jam priimtinų. Bet, kad nustatyti, kaip veikia sistema ir ar ji tikrai tinka vartotojui naudotis, reikia ją įvertinti tam tikrais kriterijais, tokiais kaip stabilumas, greitaveika ir pan.

4.1. Vertinimo kriterijai ir rezultatai

Programinės įrangos vertinimo kriterijai ir aprašymas pateikiami žemiau esančioje lentelėje. Kadangi nėra panašių sistemų, tai vertiname tik pačią sistemą.

2 lentelė. Programinės įrangos vertinimo kriterijai

Parametras	Aprašymas
Saugumas	Nėra. Programoje nėra realizuota jokios vartotojų autorizavimo ar kitos priemonės.
Išplečiamumas	Yra. Sistema suprojektuota taip, kad į ją lengva būtų įtraukti naujus algoritmus. Sistemos pakeitimus taip pat galima atlikti, nes pateikiamas sistemos kodas ir dokumentacija.
Pernešamumas	Yra tik klientinės dalies. Klientinė dalis parašyta JAVA programavimo kalba ir gali veikti tiek Windows, tiek Linux aplinkoje, serverinės dalies pernešti negalima, nes pritaikyta veikti tik Linux aplinkoje.
Sąsajos galimybės	Nėra. Sistema naudoja integruotus algoritmus, kurie aprašyti sistemoje, nėra galimybės nurodyti kitų algoritmų kaip papildomą šaltinį.
Panaudojamumas	Yra. Pateikiama vartotojo dokumentacija.
Patvarumas	Nėra. Sistema netikrina, ar įtrauktas algoritmas veikia gerai, ar neįvyksta kažkokių tai aklaviečių ir pan.
Išbaigtumas	Nėra. Sistema nebuvo pilnai naudojama normaliomis sąlygomis, buvo tik testuojama. Todėl vartotojui gali prireikti naujų funkcijų ar pakeisti esamas.
Efektyvumas	Yra. Sistemos pagrindinį skaičiuojamąjį darbą atlieka serverio dalis, kuri yra serveryje, o klientas tik atvaizduoja duomenis, todėl

	duomenų atvaizdavimas nereikalauja daug resursų, priešingai nei serveris.
Ištestavimo lygis	Yra. Testavimas vyko su įvairiais algoritmais. Sistema nėra išbandyta normaliomis darbo sąlygomis, kaip minėta „Išbaigtumas“ punkte.
Lankstumas	Nėra. Sistema negali būti modifikuojama paprastai, be programuotojo įsikišimo, negali būti keičiami skaičiavimo parametrai.

4.2. Tobulino galimybės

Sistemos tobulinimo galimybės nustatomos naudojantis sistema. Nors ir buvo įgyvendinti visi pradiniai reikalavimai, sistema nėra išbaigta. Kodėl taip sakoma, kad nėra išbaigta? Žinoma todėl, kad naudojantis sistema pastebimi jos trūkumai, trūkstamos funkcijos, trūkstamos galimybės, ar tai tam tikri vartotojo sąsajos nepatogumai.

Kol kas pasinaudojus sistema iškilo keletas sistemos patobulinimų pasiūlymų.

- Formulės rankinis įvedimas, kad paskaičiuoti atitinkamus koeficientus priklausomai nuo formulės. Kad įvertinti procesoriaus apkrovimo kriterijų lyginant algoritmus.
- Kelių algoritmų detalios informacijos palyginimas tarpusavyje viename lange.
- Parametrų nustatymo galimybė, kad ištestuotu algoritmą su visomis galimomis branduolių skaičiaus kombinacijomis.

5. EKSPERIMENTINĖ DALIS

Šioje dalyje pateikiama Java gijų veikimo įvairiose programose analizė, analizavimo rezultatai, rezultatų įvertinimas. Eksperimentas atliktas su tam tikslui paruošta programine įranga, kuri lengvai leidžia peržiūrėti svarbiausius rezultatų duomenis tokius, kaip laikas, procesoriaus išnaudojimas ir operatyviosios atminties išnaudojimas.

5.1. Programinė įranga ir naudojimas

Pagrindinė programinės įrangos funkcija – suteikti galimybę vartotojui palyginti skirtingas lygiagrečiųjų skaičiavimų technologijas panašiomis sąlygomis – vykdant analogiškus skaičiavimus atliekančius algoritmus.

5.1.1. Programinės įrangos dalys

- Serverinė (centrinė) sistemos dalis – centrinė kuriamos sistemos dalis, diegiama serveryje arba kitame tinklu pasiekiamame kompiuteryje. Šiame kompiuteryje taip pat turi būti įdiegta ir paruošta naudojimui visa programinė įranga ir technologijos, reikalingos konkrečių algoritmų paleidimui: JavaCSP, Java ir k.t.
- Klientinė sistemos dalis (klientas)– šioje sistemoje tai vartotojo sąsają turinti grafinė aplinka, leidžianti per nuotolį prisijungti prie anksčiau minėtos serverinės sistemos dalies. Prisijungus, kliento dėka, galima valdyti serverinę pusę – gauti galimų vykdyti algoritmų informaciją, juos išsirinkti ir paleisti, nurodyti parametrus algoritmų vykdymui ir atvaizduoti vykdymo metu ar po jo gautus rezultatus ir informaciją.

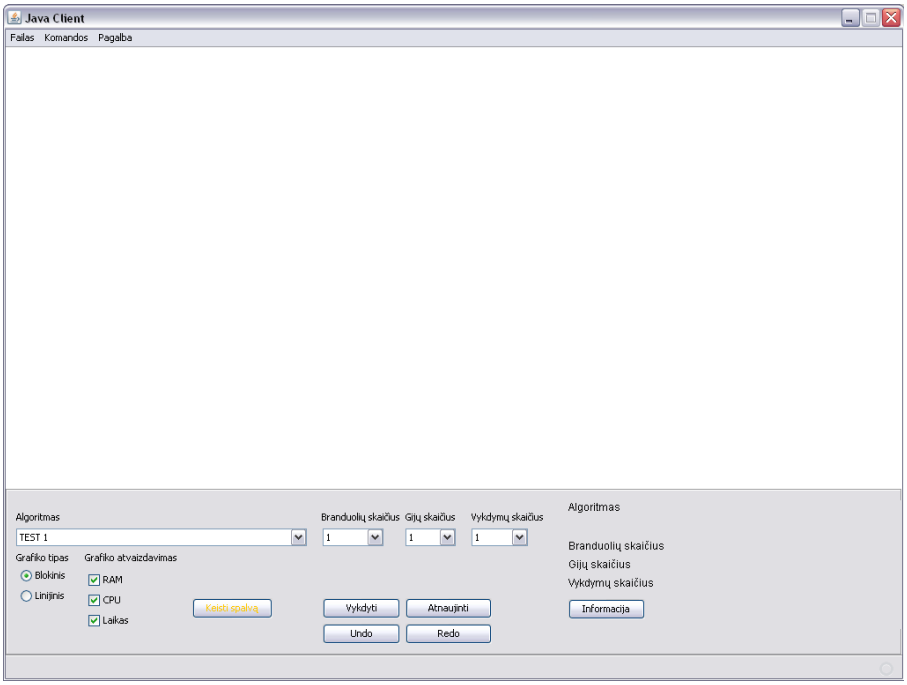
5.1.2. Algoritmo vykdymas

Vykdant skirtingus algoritmus, galima pasirinkti daug nustatymo galimybių, kaip:

- Algoritmas – pasirenkamas atitinkamas algoritmas, kurį ruošiamės vykdyti.
- Branduolių skaičius – pasirenkama branduolių skaičių, tai nusako, su kiek branduolių serverinė dalis vykdys algoritmą.
- Vykdymų skaičius – nusakome, kelis kartus bus kartojamas algoritmo vykdymas.

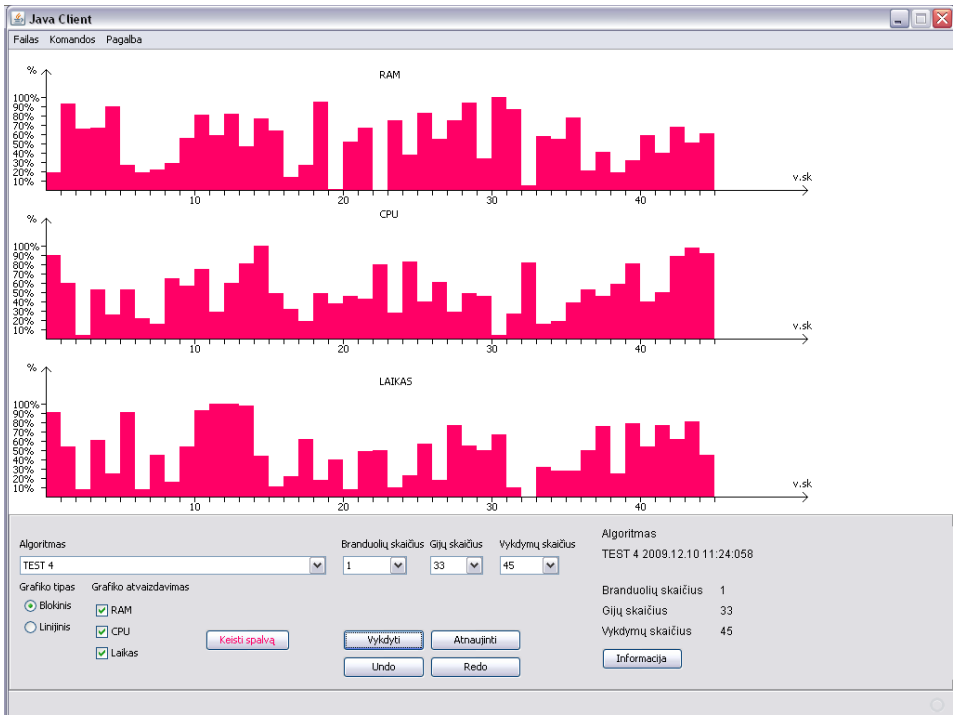
- Grafiko tipas – nustatome, kokį grafiko tipą norime matyti.
- Grafiko atvaizdavimas – nustatome, kuriuos parametrus norime matyti, RAM – operatyvioji atmintis, CPU – procesorius ar Laikas.
- Keisti spalvą – nustatome grafiko spalvą.
- Vykdyti – pradedame algoritmo vykdymą.
- Atnaujinti – Atnaujiname duomenis apie algoritmus iš serverinės dalies.
- Undo – sugrįžtame į prieš tai vykdytą algoritmą.
- Redo – Priešingas veiksmas Undo.

Pagrindinis algoritmų vykdymo langas leidžia nustatyti prieš tai minėtus parametrus ir pasirinkti norimą algoritmą (7 pav.)



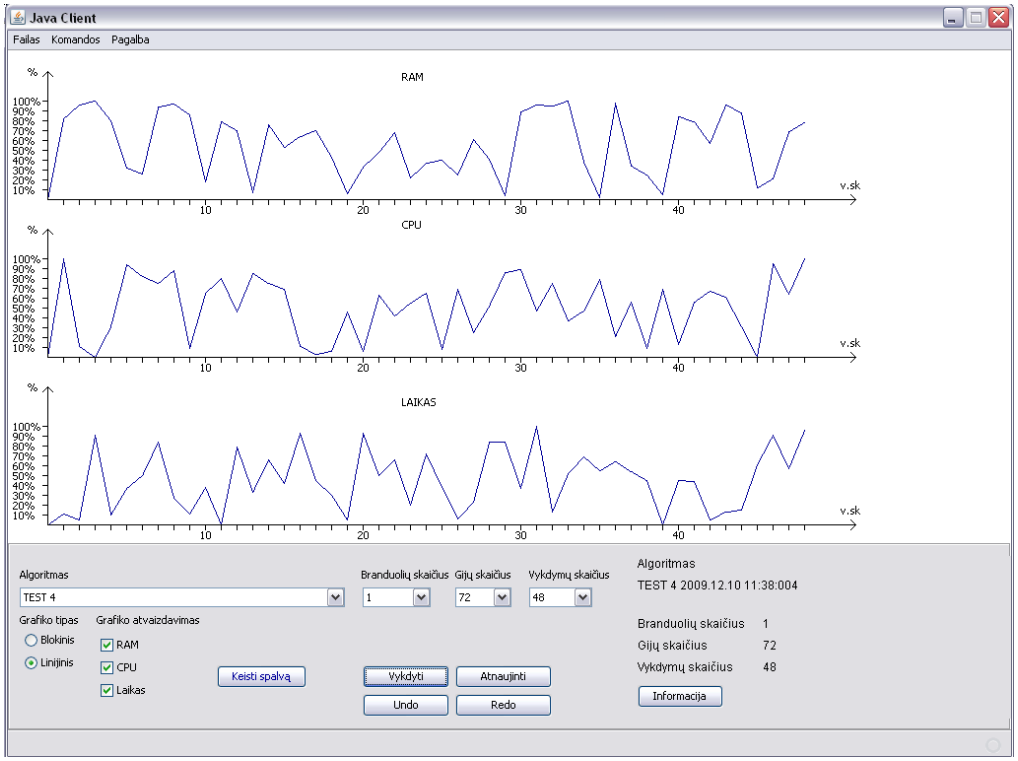
7 pav. Vykdymo langas

Įvykdžius algoritmą su pasirinktai parametrai gauname atitinkamus rezultatus, kurie atvaizduojami grafiku (8 pav.).



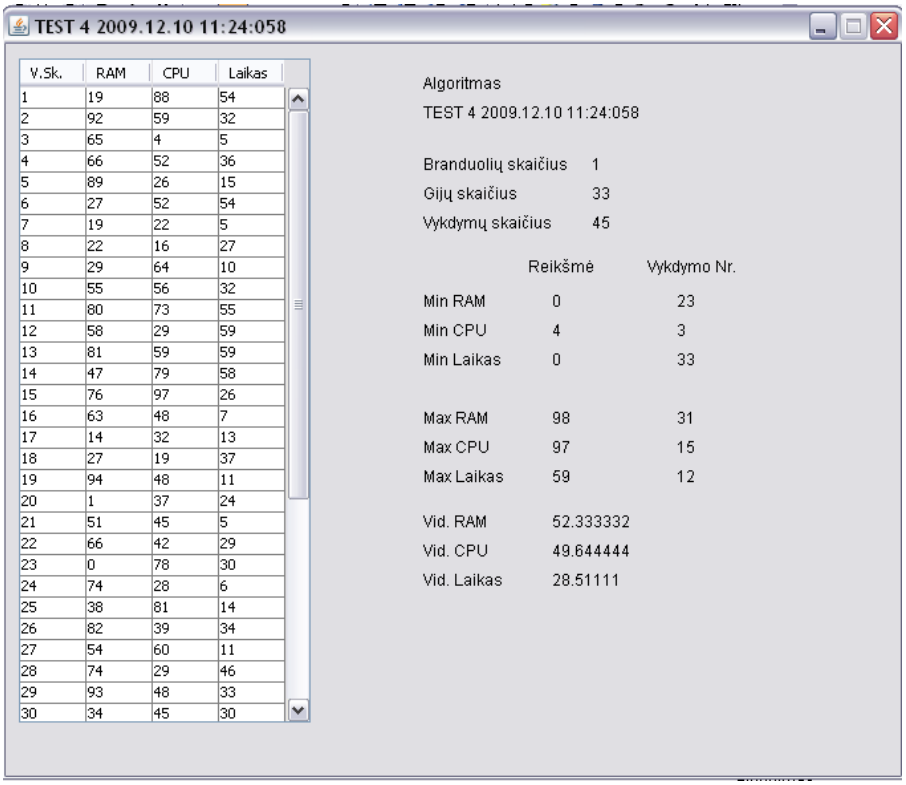
8 pav. Vykdymo langas su algoritmo rezultatais

Taip pat galima pasirinkti skirtingus rezultatų atvaizdavimo būdus ir skirtingas atvaizdavimo duomenų kombinacijas (9 pav.).



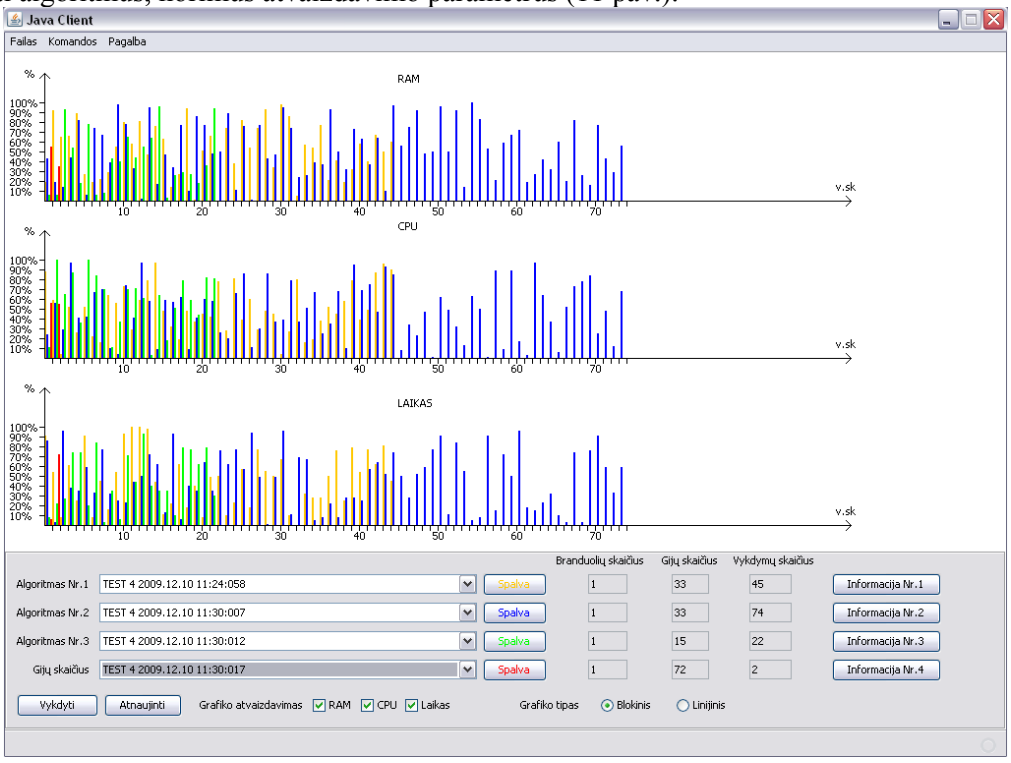
9 pav. Vykdymo langas su algoritmo rezultatais, linijinis atvaizdavimas

Detalius įvykdyto algoritmo duomenis galima pasirinkus atitinkamą meniu punktą. Detalūs duomenys bus atvaizduojami lentelės forma (10 pav.).



10 pav. Detali rezultatų peržiūra

Visus įvykdytų algoritmų duomenis galima palyginti tarpusavyje grafiškai. Pasirintkti norimus palyginti algoritmus, norimus atvaizdavimo parametrus (11 pav.).



11 pav. Algoritmų grafinis palyginimas

5.2. Algoritmų parinkimas eksperimentui

Atliekant eksperimentą neįmanoma patikrinti visų galimų algoritmų, kuriuose naudojamos Java gijos, todėl pasirinkti keli algoritmai, kurie bus naudojami eksperimente.

Šie algoritmai pasirinkti, nes rikiavimo algoritmia dažnai naudojami. Taip pat juos dažniausiai naudoja nemodifikavus. Taip pat dažnai atliekami skaičiavimai. Pasirinkti labai specifinį algoritmą nėra tikslo, nes jis mažai naudojamas ir svarbesni dažnai naudojamų algoritmų rezultatai. Tiek gijų, branduolių skaičiaus įtakai, tiek aklavietės, bado problemos sprendimams analizuoti bus naudojami tie patys algoritmai, tik su skirtingais problemų sprendimo būdais.

Gijų, branduolių skaičiaus įtakai nustatyti bus naudojami trys algoritmai:

- MergeSort – rikiavimo algoritmas
- QuickSort – rikiavimo algoritmas
- Calculation – skaičiavimo algoritmas

Abu rikiavimo algoritmai rikiuoja 10 mln ilgio atsitiktinių skaičių masyvą, kad užtikrinti didesnį įvairiškumą, o atsitiktiniai skaičiai neturės didelės įtakos, nes bus vertinamas veikimo vidurkis. Calculation algoritmas atlieka operacijas taip pat 10mln ilgio masyve.

Algoritmai MergeSort ir QuickSort yra suprogramuoti taip, kad būtų galimybė jiems perduoti parametrą, kuris keičia gijų skaičių programoje. Ir tai vienintelis parametras, perduodamas algoritmams. Rikiuojamo masyvo ilgis, atsitiktinių skaičių reikšmės yra nustatytos algoritmo kode ir nekeičiamos iš vykdomosios programos. Taip buvo pasirinkta, kad sukurtos programos kode būtų kuo mažiau įvairių sąlygų, kuo mažiau pašalinių veiksmų, kuriuos reikia įvykdyti, norint vykdyti programą. Taip bandoma užtikrinti vienodesnį vykdymų rezultatą ir labiau atspindintį gijų įtaką algoritmui, o ne papildomų parametrų įtaką.

Algoritmas Calculation skirtas išanalizuoti gijų pobūdžio įtaką. Pirmasis Calc1 algoritmas skaičiuoja skaičių sumą ir ieško minimalios reikšmės 10mln ilgio masyve. Paiešką ir skaičiavimus atlikdamas tose pačiose gijose. Antrasis Calc2 algoritmas skaičiavimą atlieka vienose gijose, minimalios reikšmės paiešką - kitose.

Aklavietės, bado, amžinų ciklų problemų sprendimams buvo pasirinkti užraktai (lock) ir semaforo (semaphore), nes šitie sprendimai yra patys universaliausi paprastiems

uždaviniams ir dauguma šaltinių siūlo naudoti būtent šituos sprendimus. Sprendimams įvertinti bus naudojami tie patys algoritmai, tik su skirtingais užraktais. Kadangi vykdant lygiagrečius algoritmus reikalingi užraktai, tai naudosime abejus sprendimus prieš tai minėtiems kriterijams nustatyti.

5.3. Eksperimento vykdymas ir rezultatai

5.3.1. Sutrumpinimai ir paaiškinimai

Serverio parametrai:

- Procesorius – Intel Core 2 CPU 1.8Ghz
- Operatyvioji atmintis (RAM) – 1024 MB
- Operacinė sistema – Ubuntu 9.10

Sutrumpinimai:

- Gijų skaičius – G.sk
- Operatyvioji atmintis – RAM
- Procesorius - CPU

Operatyvioji atmintis ir procesoriaus apkrovimas matuojamas procentais.

Palyginimo grafikas pateikiamas keturių skirtingų vykdymų, su 1,2,4,10 gijų. Grafike vykdymo laikas atvaizduojamas procentine reikšme ir grafiko viršuje pateikiamas maksimalus vieno vykdymo laikas milisekundėmis. 100% atitinka maksimal 100% atitinka maksimalę vykdymo vertę, o kitos vertės atitinka maksimalios vertės atitinkamą procentinę dalį. Laikas matuojamas milisekundėmis.

5.3.2. QuickSort algoritmas

Vykdomas QuickSort rikiavimo algoritmas su dviem skirtingais užrakto tipais. Vykdymas bus atliekamas 30 kartų, su 1, 2, 4, 10 gijų su vienu branduoliu ir dviem.

Rezultatai su vienu branduoliu ir vykdoma trisdešimt kartų.

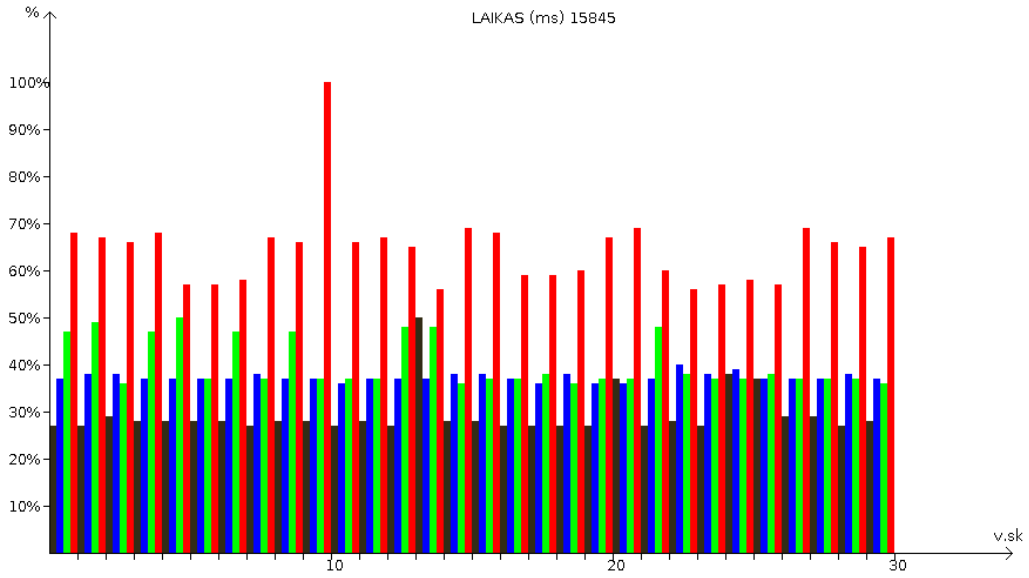
3 lentelė.QuickSort algoritmo vieno branduolio rezultatai

Algoritmas	G.sk	Min RAM	Min CPU	Min Laikas	Max RAM	Max CPU	Max Laikas	Vid Ram	Vid CPU	Vid Laikas
QuickSort1 lock	1	5	44	**4325**	8	71	**8035**	5	63	**4734**
QuickSort1 lock	2	7	51	**5768**	8	74	**6480**	7	58	**5979**
QuickSort1 lock	4	8	52	**5831**	11	71	**7977**	9	65	**6448**
QuickSort1 lock	10	10	46	**8937**	14	65	**15845**	11	58	**10296**
QuickSort1 semaphore	1	5	42	**4411**	8	64	**8135**	6	54	**5640**
QuickSort1 semaphore	2	7	44	**5867**	8	66	**8002**	7	54	**6273**
QuickSort1 semaphore	4	8	51	**7390**	11	66	**9543**	8	56	**7704**
QuickSort1 semaphore	10	10	48	**9083**	13	73	**12931**	10	56	**11225**

Rezultatai su dviem branduoliais ir vykdoma trisdešimt kartų.

4 lentelė.QuickSort algoritmo dviejų branduolių rezultatai

Algoritmas	G.sk	Min RAM	Min CPU	Min Laikas	Max RAM	Max CPU	Max Laikas	Vid Ram	Vid CPU	Vid Laikas
QuickSort1 lock	1	5	52	**4273**	5	74	**4915**	5	58	**4516**
QuickSort1 lock	2	6	54	**4297**	10	77	**6338**	6	71	**4722**
QuickSort1 lock	4	8	49	**5698**	11	79	**10696**	8	66	**6365**
QuickSort1 lock	10	10	49	**7192**	13	75	**15217**	11	65	**9324**
QuickSort1 semaphore	1	5	51	**4373**	5	72	**5214**	5	56	**4650**
QuickSort1 semaphore	2	6	51	**4301**	8	78	**6255**	6	71	**4666**
QuickSort1 semaphore	4	8	55	**5786**	9	81	**7980**	8	67	**6098**
QuickSort1 semaphore	10	10	59	**7166**	14	77	**9282**	10	71	**7733**



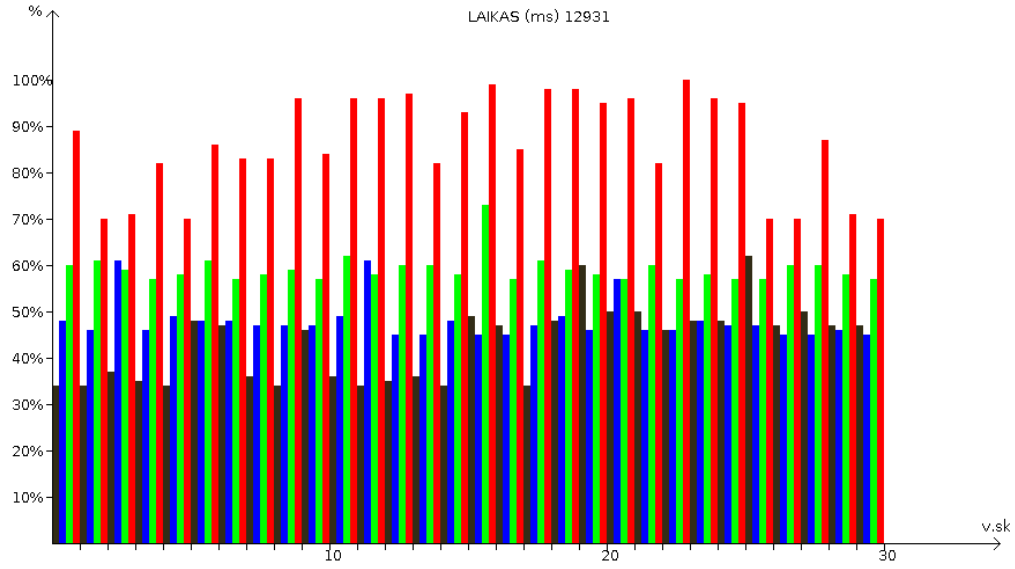
12 pav. QuickSort algoritmo vykdymo palyginimas

5 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
QuickSort1 lock	1	1	30	
QuickSort1 lock	1	2	30	
QuickSort1 lock	1	4	30	
QuickSort1 lock	1	10	30	

QuickSort algoritmo vykdymo parametrai su vienu branduoliu ir „lock“ tipo užraktu.

Grafike vykdymo laikas atvaizduojamas procentine reikšme ir grafiko viršuje pateikiamas maksimalus vieno vykdymo laikas milisekundėmis. 100% atitinka maksimalę vykdymo vertę, o kitos vertės atitinka maksimalios vertės atitinkamą procentinę dalį.

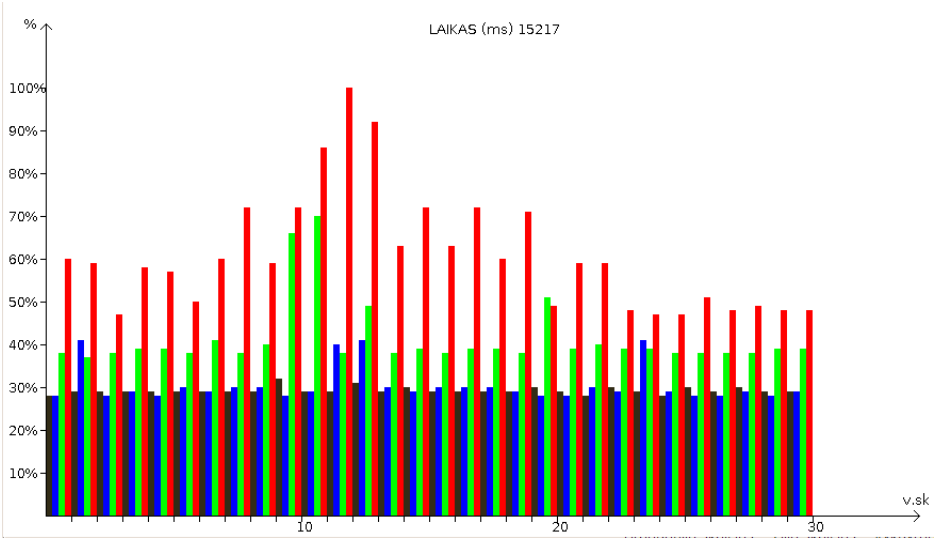


13 pav. QuickSort algoritmo vykdymo palyginimas

6 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
QuickSort1 semaphore	1	1	30	
QuickSort1 semaphore	1	2	30	
QuickSort1 semaphore	1	4	30	
QuickSort1 semaphore	1	10	30	

QuickSort algoritmo vykdymo parametrai su vienu branduoliu ir „semaphore“ tipo užraktu



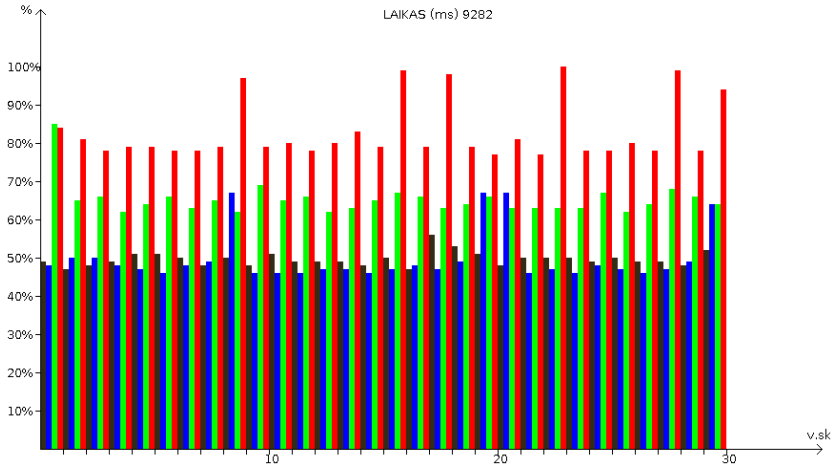
14 pav. QuickSort algoritmo vykdymo palyginimas

7 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
QuickSort1 lock	2	1	30	
QuickSort1 lock	2	2	30	
QuickSort1 lock	2	4	30	
QuickSort1 lock	2	10	30	

QuickSort algoritmo vykdymo parametrai su dviem branduoliais ir „lock“ tipo užraktu.

Grafike vykdymo laikas atvaizduojamas procentine reikšme ir grafiko viršuje pateikiamas maksimalus vieno vykdymo laikas milisekundėmis. 100% atitinka maksimalę vykdymo vertę, o kitos vertės atitinka maksimalios vertės atitinkamą procentinę dalį.



15 pav. QuickSort algoritmo vykdymo palyginimas

8 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
QuickSort1 semaphore	2	1	30	
QuickSort1 semaphore	2	2	30	
QuickSort1 semaphore	2	4	30	
QuickSort1 semaphore	2	10	30	

QuickSort algoritmo vykdymo parametrai su dviem branduoliais ir „semaphore“ tipo užraktu.

5.3.3. MergeSort algoritmas

Vykdomas MergeSort rikiavimo algoritmas su dviem skirtingais užrakto tipais. Vykdymas bus atliekamas 30 kartų, su 1, 2, 4, 10 gijų. Taip pat su vienu branduoliu ir dviem.

Rezultatai su vienu branduoliu ir vykdoma trisdešimt kartų.

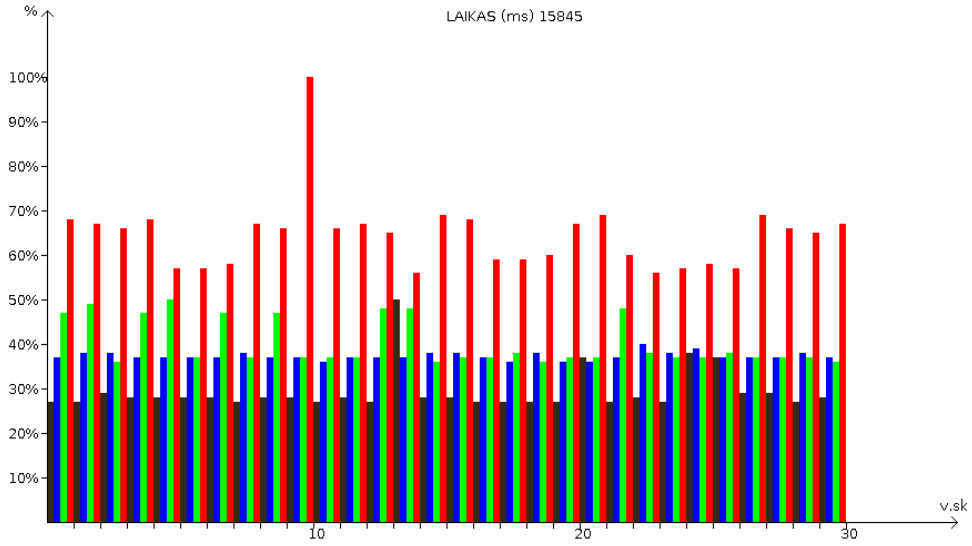
9 lentelė.MergeSort algoritmo vieno branduolio rezultatai

Algoritmas	G.sk	Min RAM	Min CPU	Min Laikas	Max RAM	Max CPU	Max Laikas	Vid Ram	Vid CPU	Vid Laikas
MergeSort1 lock	1	9	45	**10475**	11	60	**18103**	10	51	**14351**
MergeSort1 lock	2	8	49	**10287**	10	71	**14838**	8	59	**11297**
MergeSort1 lock	4	7	43	**10597**	10	67	**19677**	8	56	**13535**
MergeSort1 lock	10	8	45	**13448**	11	66	**19434**	9	56	**15142**
MergeSort1 semaphore	1	8	44	**9041**	11	63	**18583**	9	53	**13703**
MergeSort1 semaphore	2	8	54	**10256**	10	70	**12633**	8	58	**10782**
MergeSort1 semaphore	4	7	54	**10201**	10	68	**12899**	8	61	**11142**
MergeSort1 semaphore	10	8	50	**11893**	11	70	**17528**	9	60	**13572**

Rezultatai su dviem branduoliais ir vykdoma trisdešimt kartų.

10 lentelė.MergeSort algoritmo dviejų branduolių rezultatai

Algoritmas	G.sk	Min RAM	Min CPU	Min Laikas	Max RAM	Max CPU	Max Laikas	Vid Ram	Vid CPU	Vid Laikas
MergeSort1 lock	1	9	50	**8715**	11	68	**13028**	10	62	**9739**
MergeSort1 lock	2	8	53	**9169**	10	71	**12587**	8	59	**10777**
MergeSort1 lock	4	8	54	**8709**	10	71	**12584**	8	62	**10701**
MergeSort1 lock	10	8	53	**10261**	11	71	**14262**	9	62	**12146**
MergeSort1 semaphore	1	9	55	**8788**	11	74	**11459**	10	63	**9541**
MergeSort1 semaphore	2	8	53	**8724**	10	69	**11900**	8	58	**10505**
MergeSort1 semaphore	4	8	54	**10204**	10	70	**12453**	8	61	**10763**
MergeSort1 semaphore	10	9	51	**10664**	12	70	**14088**	9	61	**12480**



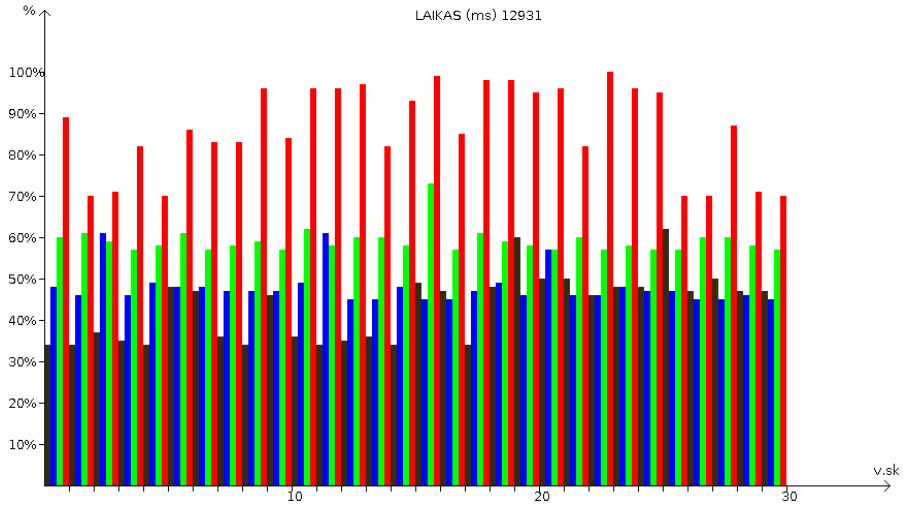
16 pav. MergeSort algoritmo vykdymo palyginimas

11 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
MergeSort1 lock	1	1	30	
MergeSort1 lock	1	2	30	
MergeSort1 lock	1	4	30	
MergeSort1 lock	1	10	30	

MergeSort algoritmo vykdymo parametrai su vienu branduoliu ir „lock“ tipo užraktu.

Grafike vykdymo laikas atvaizduojamas procentine reikšme ir grafiko viršuje pateikiamas maksimalus vieno vykdymo laikas milisekundėmis. 100% atitinka maksimalę vykdymo vertę, o kitos vertės atitinka maksimalios vertės atitinkamą procentinę dalį.

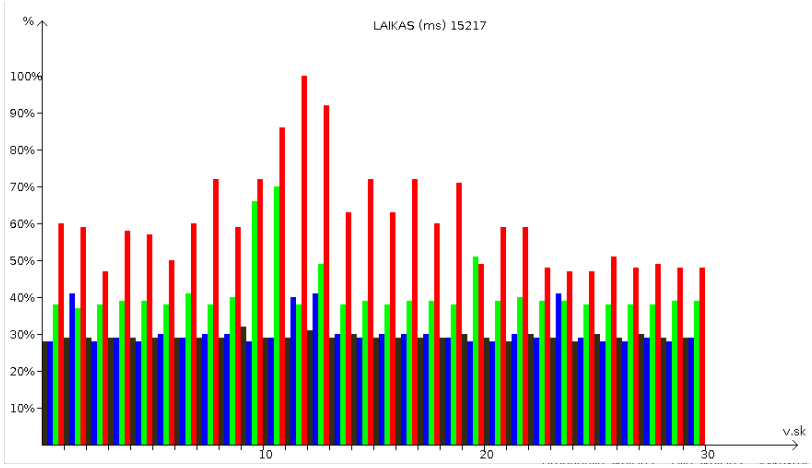


17 pav. MergeSort algoritmo vykdymo palyginimas

12 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
MergeSort1 semaphore	1	1	30	
MergeSort1 semaphore	1	2	30	
MergeSort1 semaphore	1	4	30	
MergeSort1 semaphore	1	10	30	

MergeSort algoritmo vykdymo parametrai su vienu branduoliu ir „semaphore“ tipo užraktu.

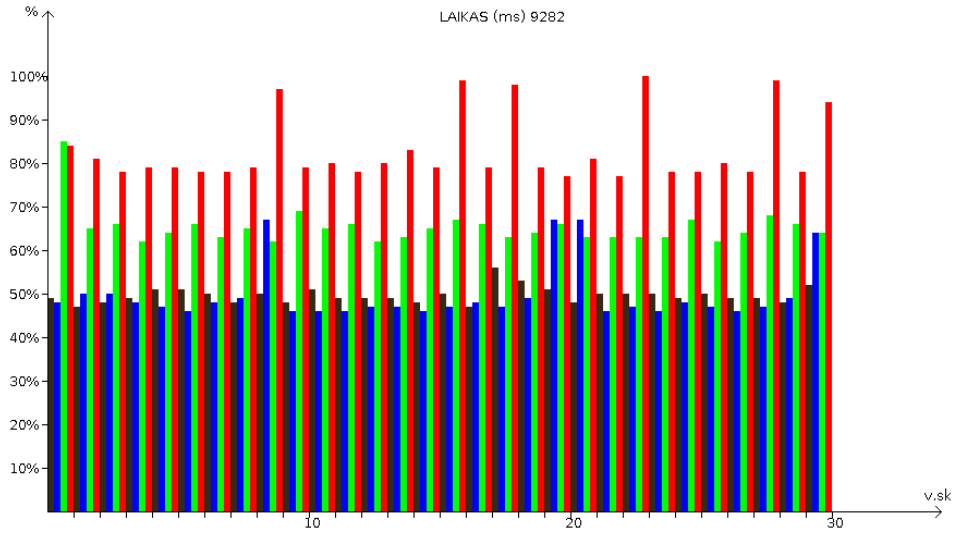


18 pav. MergeSort algoritmo vykdymo palyginimas

13 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
MergeSort1 lock	2	1	30	
MergeSort1 lock	2	2	30	
MergeSort1 lock	2	4	30	
MergeSort1 lock	2	10	30	

MergeSort algoritmo vykdymo parametrai su dviem branduoliais ir „lock“ tipo užraktu.



19 pav. MergeSort algoritmo vykdymo palyginimas

14 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
MergeSort1 semaphore	2	1	30	
MergeSort1 semaphore	2	2	30	
MergeSort1 semaphore	2	4	30	
MergeSort1 semaphore	2	10	30	

MergeSort algoritmo vykdymo parametrai su dviem branduoliais ir „semaphore“ tipo užraktu.

5.3.1. Calculation algoritmas

Vykdomi Calc1 ir Calc2 skaičiavimo algoritmai. Vykdymas bus atliekamas 30 kartų, su 1, 2, 4, 10 gijų. Taip pat su vienu branduoliu ir dviem.

Rezultatai su vienu branduoliu ir vykdoma trisdešimt kartų.

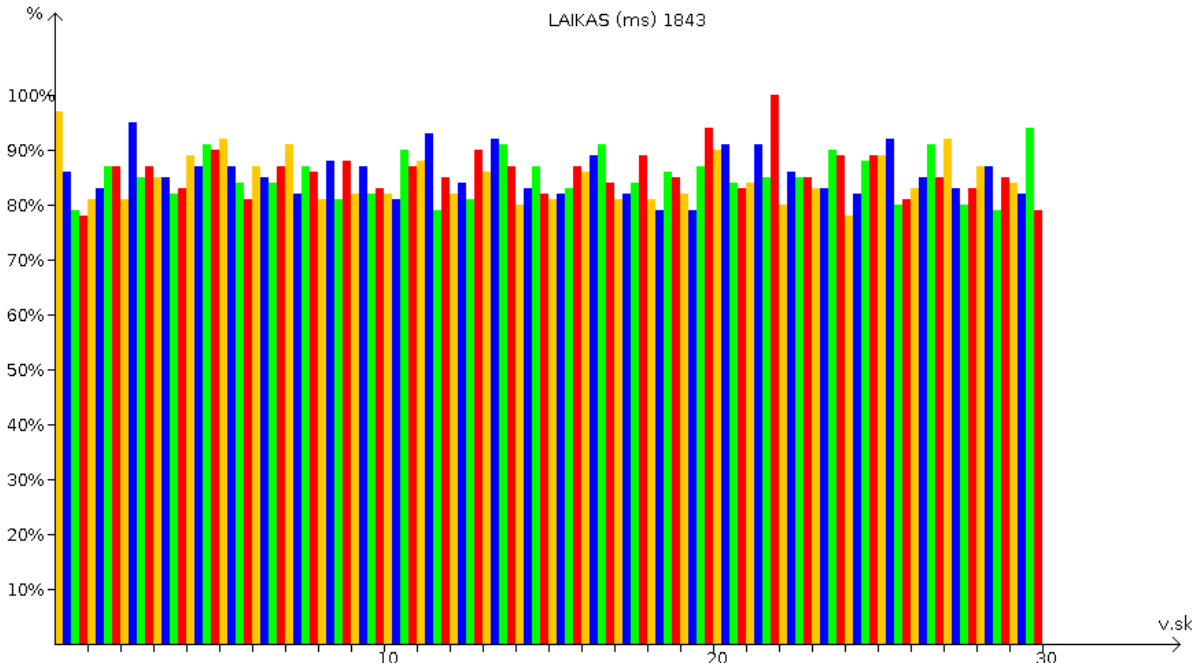
15 lentelė. Calc1 ir Calc2 algoritmo vieno branduolio rezultatai

Algoritmas	G.sk	Min RAM	Min CPU	Min Laikas	Max RAM	Max CPU	Max Laikas	Vid Ram	Vid CPU	Vid Laikas
Calc1	1	1	43	**1446**	4	61	**1800**	2	52	**1573**
Calc1	2	2	21	**1460**	2	58	**1762**	2	36	**1587**
Calc1	4	2	16	**1464**	2	57	**1750**	2	28	**1579**
Calc1	10	2	19	**1449**	2	55	**1843**	2	29	**1594**
Calc2	1	1	17	**1449**	2	54	**1809**	1	25	**1571**
Calc2	2	1	17	**1447**	2	57	**1745**	1	27	**1575**
Calc2	4	2	12	**1441**	2	47	**1725**	2	24	**1571**
Calc2	10	2	20	**1473**	2	51	**1690**	2	27	**1568**

Rezultatai su dviem branduoliais ir vykdoma trisdešimt kartų.

16 lentelė. Calc1 ir Calc2 algoritmo dviejų branduolių rezultatai

Algoritmas	G.sk	Min RAM	Min CPU	Min Laikas	Max RAM	Max CPU	Max Laikas	Vid Ram	Vid CPU	Vid Laikas
Calc1	1	2	20	**1411**	2	56	**1662**	2	30	**1553**
Calc1	2	2	21	**1427**	2	53	**1729**	2	28	**1579**
Calc1	4	2	20	**1429**	2	60	**1681**	2	29	**1561**
Calc1	10	2	16	**1488**	2	60	**1734**	2	28	**1593**
Calc2	1	2	22	**1428**	2	56	**1837**	2	25	**1557**
Calc2	2	2	18	**1423**	2	45	**1629**	2	26	**1551**
Calc2	4	2	18	**1422**	2	52	**1730**	2	25	**1554**
Calc2	10	2	15	**1443**	2	49	**1765**	2	27	**1565**



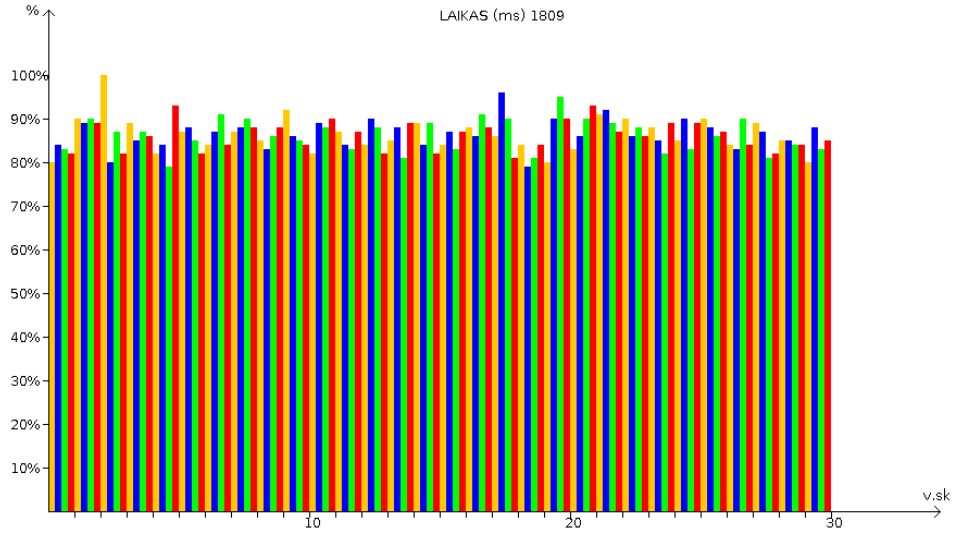
20 pav. Calc1 algoritmo vykdymo palyginimas su vienu branduoliu

17 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
Calc1	1	1	30	
Calc1	1	2	30	
Calc1	1	4	30	
Calc1	1	10	30	

Calc1 algoritmo vykdymo parametrai su vienu branduoliu.

Grafike vykdymo laikas atvaizduojamas procentine reikšme ir grafiko viršuje pateikiamas maksimalus vieno vykdymo laikas milisekundėmis. 100% atitinka maksimalę vykdymo vertę, o kitos vertės atitinka maksimalios vertės atitinkamą procentinę dalį.

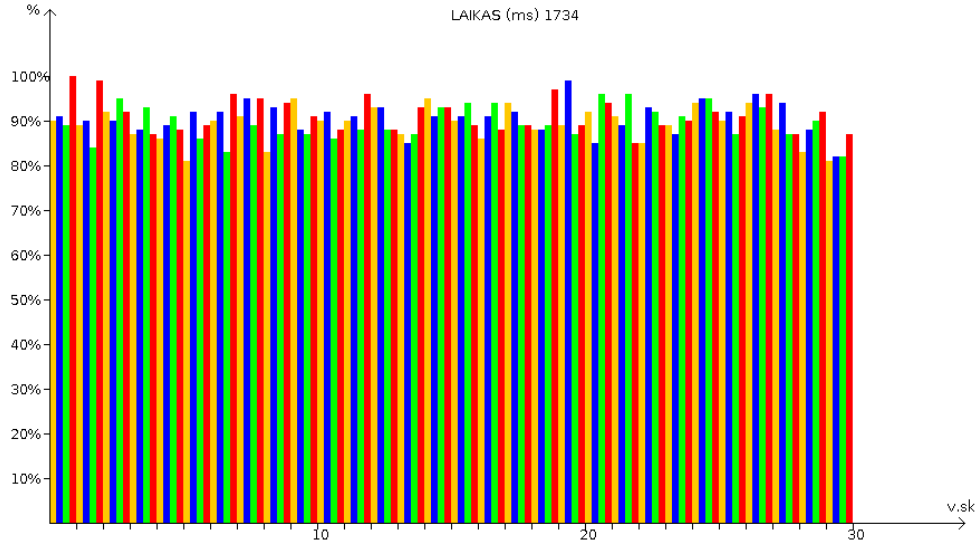


21 pav. Calc2 algoritmo vykdymo palyginimas su vienu branduoliu

18 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
Calc2	1	1	30	
Calc2	1	2	30	
Calc2	1	4	30	
Calc2	1	10	30	

Calc2 algoritmo vykdymo parametrai su vienu branduoliu.

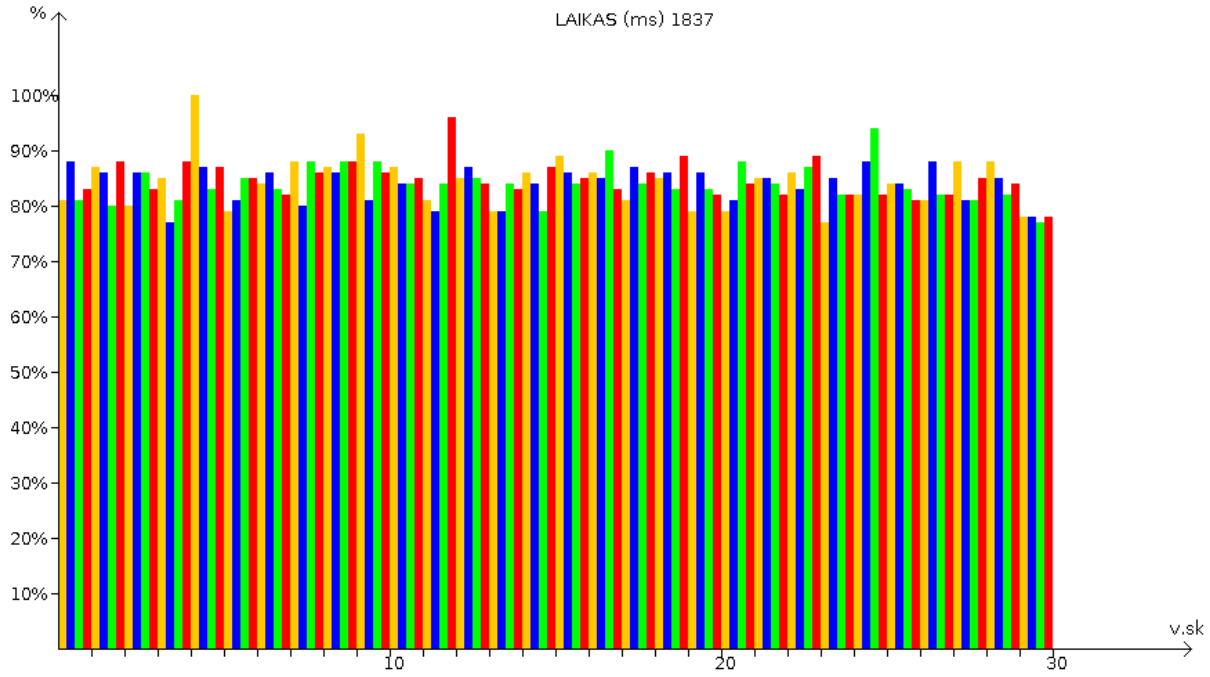


22 pav. Calc1 algoritmo vykdymo palyginimas su dviem branduoliais

19 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
Calc1	2	1	30	
Calc1	2	2	30	
Calc1	2	4	30	
Calc1	2	10	30	

Calc1 algoritmo vykdymo parametrai su dviem branduoliais.



23 pav. Calc2 algoritmo vykdymo palyginimas su dviem branduoliais

20 lentelė. Algoritmo parametrai

Algoritmas	B.sk	G.sk	V.sk	Spalva
Calc2	2	1	30	
Calc2	2	2	30	
Calc2	2	4	30	
Calc2	2	10	30	

Calc2 algoritmo vykdymo parametrai su dviem branduoliais.

5.3.2. Eksperimento analizė

Gauti duomenys pateikiami lentelėse. Pateikiama palyginamoji veikimo sparta tarp 1 branduolio procesoriaus ir 2 branduolių, veikiant tam pačiam algoritmui, tomis pačiomis sąlygomis, su viena, dviem, keturiom, dešimt gijų su skirtingais užrakto tipais.

Nebuvo testuojama su daugiau branduolių procesoriais, nes nebuvo galimybės prieiti prie kompiuterių su daugiau branduolių.

„RAM“ ir „Laikas“ stulpelyje esantys skaičiai atspindi, kiek yra suvartojama mažiau resursų ir laiko. Kuo skaičius didesnis, tuo daugiau laimima. Stulpelyje „CPU“ esantys skaičiai atspindi, kiek daugiau buvo apkrautas procesorius. Procesorius buvo apkrautas daugiau, jei skaičius neigiamas.

Skaičiai palyginimui gauti pasitelkus formulę:

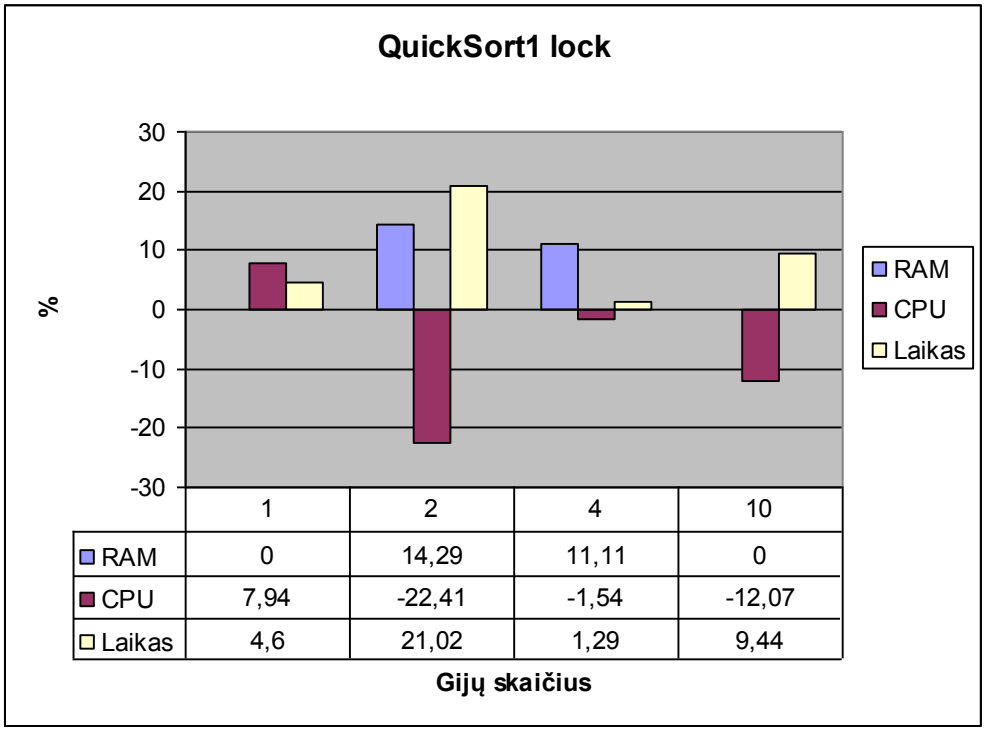
$$x = 100 - \frac{n_1}{n_2} \times 100$$

n1 – vertė gauta atliekant skaičiavimus su vienu branduoliu

n2 – vertė gauta atliekant skaičiavimus su dviem branduoliais

Gauti rezultatai atitinka procentines reikšmes.

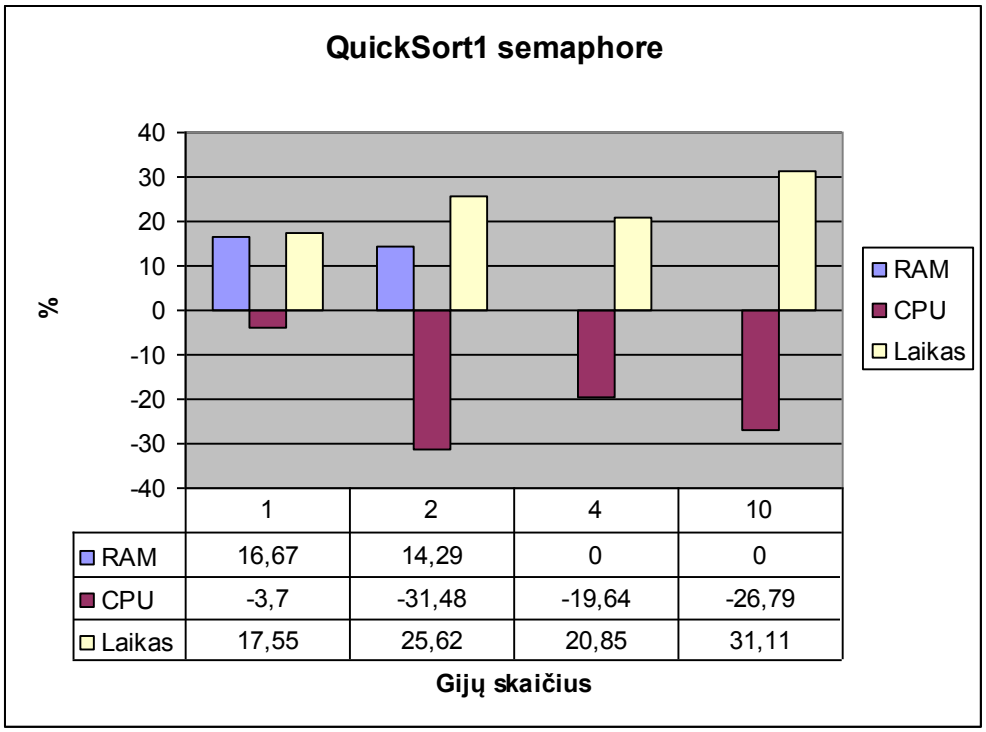
QuickSort1 algoritmas su užraktu (lock):



	1	2	4	10
RAM	0	14,29	11,11	0
CPU	7,94	-22,41	-1,54	-12,07
Laikas	4,6	21,02	1,29	9,44

24 pav. QuickSort algoritmo „lock“ užrakto grafikas

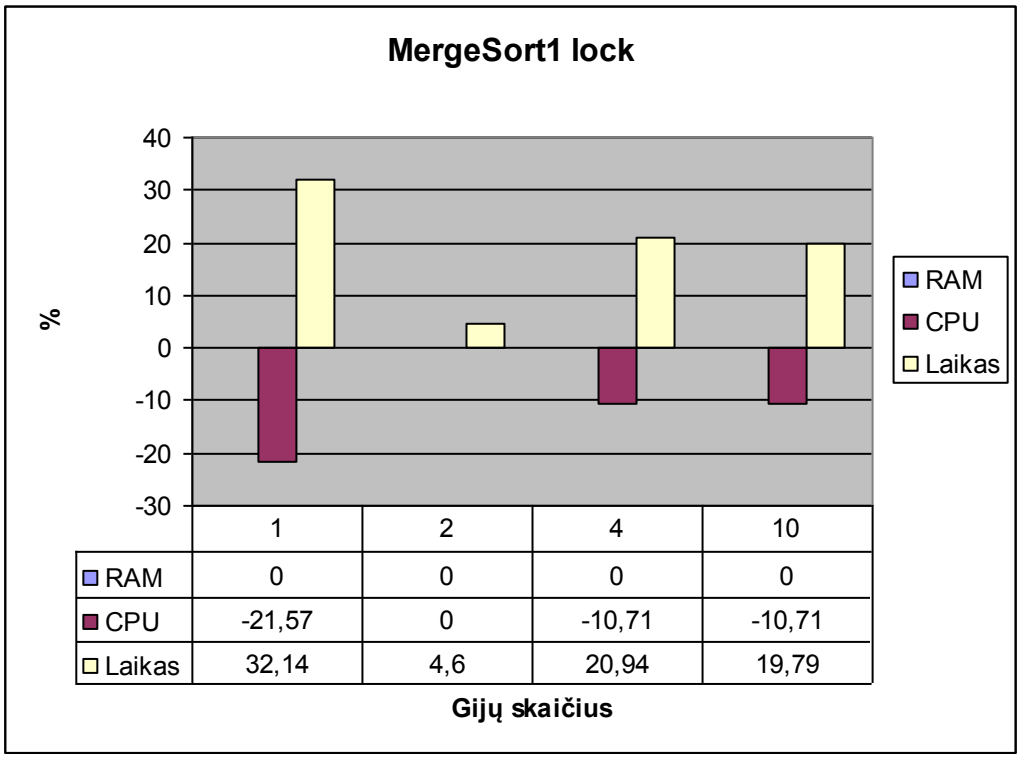
QuickSort1 algoritmas su užraktu (semaphore):



	1	2	4	10
RAM	16,67	14,29	0	0
CPU	-3,7	-31,48	-19,64	-26,79
Laikas	17,55	25,62	20,85	31,11

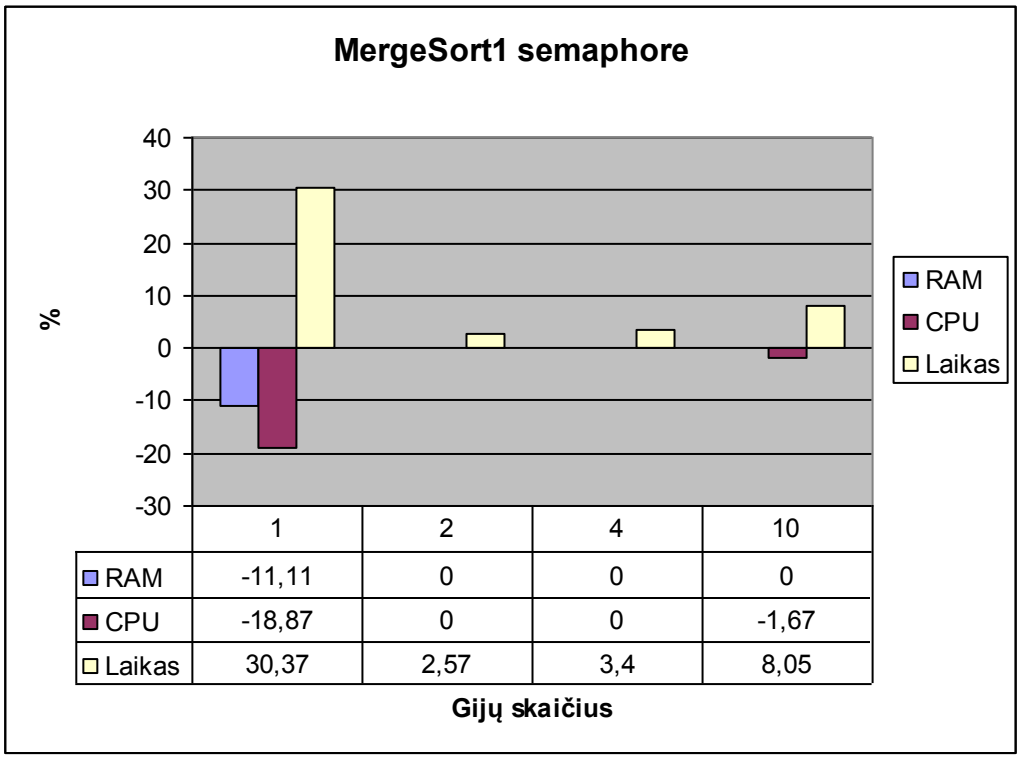
25 pav. QuickSort algoritmo „semaphore“ užrakto grafikas

MergeSort1 algoritmas su užraktu (lock):



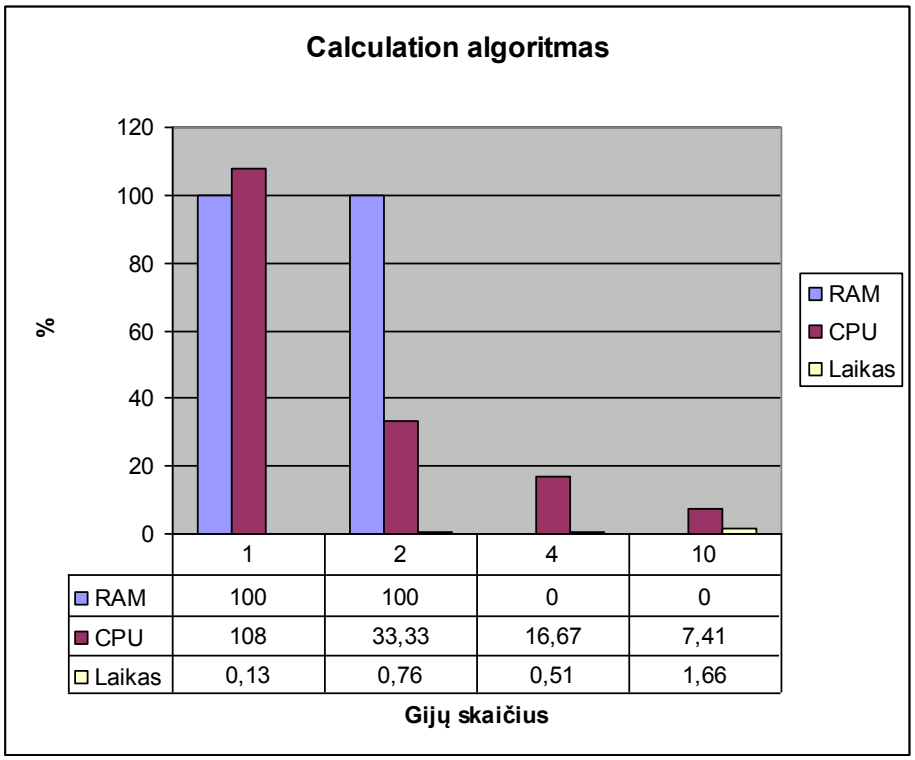
26 pav. MergeSort algoritmo „lock“ užrakto grafikas

MergeSort1 algoritmas su užraktu (semaphore):



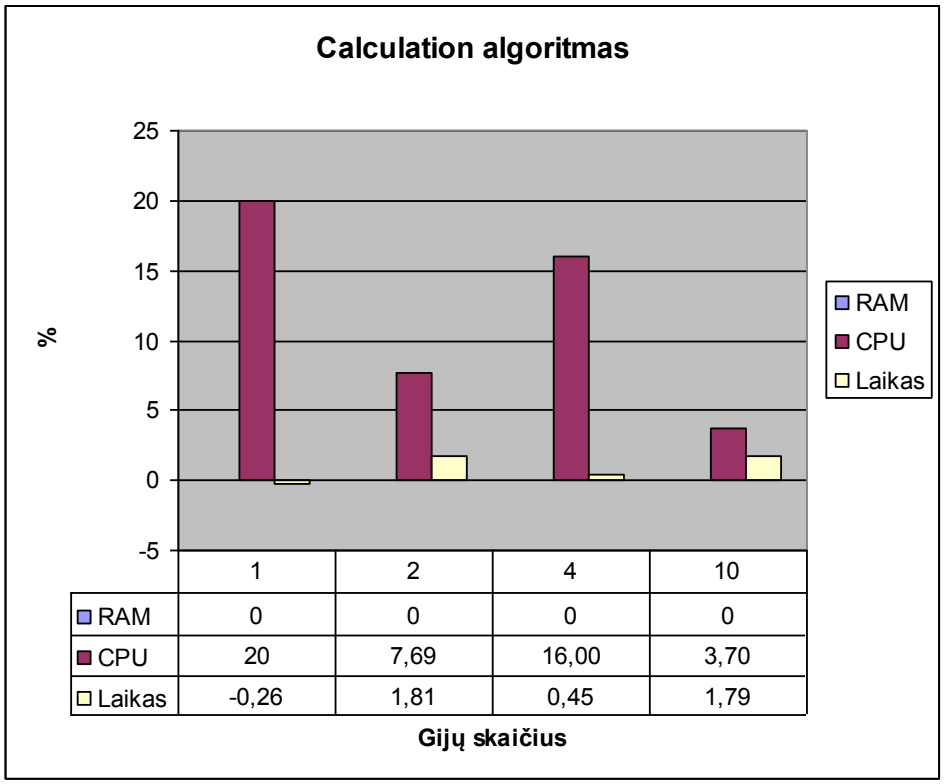
27 pav. MergeSort algoritmo „semaphore“ užrakto grafikas

Calculation algoritmas kai gijos atlieką tą patį veiksmą:



28 pav. pav. Calculation algoritmas kai gijos atlieką tą patį veiksmą

Calculation algoritmas kai gijos atlieką skirtingus veiksmus:



	1	2	4	10
RAM	0	0	0	0
CPU	20	7,69	16,00	3,70
Laikas	-0,26	1,81	0,45	1,79

29 pav. Calculation algoritmas kai gijos atlieka skirtingus veiksmus

Įvykdžius algoritmus su skirtingais parametrais, gavome rezultatus, kurie panašūs į tikėtosiuos. Bet, žinoma, atsirado ir tam tikrų keistų rezultatų, bet juos galima paaiškinti taip pat.

Pastebėti faktai vykdant algoritmus:

- Didinant gijų skaičių, skaičiavimo trukmė didėjo.
- QuickSort algoritmas didinat gijų skaičių, padidino ir operatyviosios atminties sunaudojimą.
- MergeSort algoritmo operatyviosios atminties sunaudojimas didinant gijų skaičių išliko beveik nepakitęs.
- QuickSort ir MergeSort algoritmų procesoriaus resursų išnaudojimas nepadidėjo didinant gijų skaičių, o padidėjo tik panaudojus du branduolius.
- Akivaizdus laiko sumažėjimas buvo pastebėtas panaudojant du branduolius.
- Vykdant algoritmus buvo pastebėti laiko šuoliai, kurie atsirasdavo tik pavieniais atvejais, labiausia tikėtina, kad sistema neskirdavo procesoriaus laiko algoritmams arba vykdydavo kažkokį tai operacinei sistemai reikalingą veiklą.

Šiek tiek smulkiau aptarsime rezultatus, kodėl kai kur laikai didėjo, kai kur trumpėjo. Kokią įtaka turi užrakto tipas ir pats algoritmas.

Pirmiausia buvo pastebėta, kad ilgėja vykdymo laikas, didinant gijų skaičių. Nors buvo manoma, kad tai turėtų sutrumpinti sistemos veikimo laiką, taip neįvyko. Kodėl? Pirmiausia buvo pasirinkti paprasti ir nesudėtingi algoritmai, kad mažiau būtų nenumatytų veiksmų juose, lengviau būtų galima suprasti, kas vyksta jų viduje. Pasirinkti rikiavimo algoritmai ilgiau dirba su daugiau gijų, nes kiekvienai gijai paskyrus darbą ir jai atlikus jį, rezultatus reikia sujungti. Kuo daugiau dalių reikia sujungti, tuo ilgiau viskas užtrunka. Be to, vykdant algoritmą su daug gijų ant vieno branduolio, gijos gauna procesoriaus laiką viena po kitos, o ne kartu. Tai taip pat prisideda prie lėtesnio veikimo. Jei viena gija gavo laiko, kita gali negauti ir, kol negaus, sujungimas galutinių duomenų turi laukti. Jei vykdoma su viena gija, gavus laiką, įvykdomas visas algoritmas iškarto.

Kitas pastebėjimas susijęs su QuickSort algoritmu. Padidinus gijų skaičių, išaugo ir sunaudojamos operatyviosios atminties kiekis. Kaip tai paaiškinti? Algoritmas realizuotas taip, kad jame įgyvendinta rekursija ir kviečiama vėl tas pats algoritmas mažesnei daliai duomenų, tai padalinus duomenis gijoms, daugiau gijų, daugiau kartų gaunama rekursija, daugiau sunaudojama atminties. O kadangi MergeSort algoritmas įgyvendintas neišskiriant papildomos atminties ir nėra rekursijų, tai jisai naudoja panašų kiekį atminties. Šiek tiek daugiau sunaudojama, nes sukuriama daugiau gijų, bet tai neženklus padidėjimas.

Procesoriaus resursų išnaudojimui neturėjo įtakos nei gijos, nei skirtingi užrakto tipai. Kadangi pirmi algoritmų eksperimentui atlikti su vienu procesoriaus branduoliu, tai buvo neženklus svyravimas ir didesnio procesoriaus resursų išnaudojimo nebuvo pastebėta. Priešingai nei panaudojus du procesoriaus branduolius, tokiu atveju buvo pastebėtas procesoriaus didesnis išnaudojimas nuo 1,5 iki 32 procentų. Bet jei atkreipsime dėmesį i visus analizavimus su dviem branduoliais gijos neturėjo įtakos procesoriaus resursų išnaudojimui.

Algoritmų vykdymo laikas sumažėjo panaudojus du procesoriaus branduolius nuo 1,5 iki 32 procentų. Taip pat buvo pastebėti laiko šuoliai, kurie kartais pasirodydavo tam tikrais algoritmo vykdymo momentais. Ką vadiname laiko šuoliu? Tai situacija, kai algoritmas vykdomas ženkliai ilgiau, nei vidutinis vykdymo laikas. Tokie šuoliai gali atsirasti, kai operacinė sistema fone vykdo kažkokius tai veiksmus, kai sistemai reikia kompiuterio resursų, kad palaikyti sistemos darbą, taip atsitikus kažkokia tai gija nėra vykdoma, kol neatsilaisvina procesoriaus resursai ir taip užlaikomas algoritmo įvykdymas. Kad išvengti

tokių atsitiktinių atvejų, kiekvienas algoritmas buvo vykdomas trisdešimt kartų ir imama vidutinė vykdymo vertė.

Bandant nustatyti koks užrakto tipas geresnis ir kurį geriau taikyti tokioms problemoms, kaip aklavietės, badas, amžini ciklai pastebėta, kad į šį klausimą sunku atsakyti. QuickSort algoritmas veikė greičiau su „semaphore“ tipo užraktu, o MergeSort - su „lock“ tipo užraktu. Taigi vienareikšmiškai sunku pasakyti, kas geriau. Todėl, norint naudoti kažkurį tai sprendimą, reiktų pirma paruošti testinį algoritmą, kuris bus panašus į vėliau naudotiną arba paimti būtent tą algoritmą, kurį vėliau naudosime ir išbandyti su abiem atvejais pasitelkus tam skirtą programinę įrangą ir taip nustatyti, su kuria priemone algoritmas veiks greičiau.

Bandant nustatyti, ar gijos pobūdis turi įtakos, gavome labai neaiškius rezultatus. Kas geriau? Ar kai gijos vykdo tą patį? Ar kai vykdo skirtingus veiksmus? Sunku atsakyti. Tai labai gali priklausyti nuo to, ką reikia įvykdyti. Nes pasirinktam algoritme, nors ir gavome, kad vykdant veiksmus atskirose gijose, gaunam našesnę programą, bet laiko sutaupymas buvo labai neženklus, nuo 0,13 iki 1,81 procento. Tai nėra didelis padidėjimas. Taigi paprastiems algoritmams tai didelės įtakos neturi.

5.3.3. Eksperimento rekomendacijos

Atlikus algoritmų analizę, gavome rezultatus, kuriuos galime panaudoti kuriant lygiagrečias sistemas. Žinoma, rezultatai nenurodo konkrečiam algoritmui patarimų, ką reikia patobulinti, ką reikia pakeisti, kiek gijų ar kiek branduolių reikia optimaliam darbui. Bet rezultatai rodo, kad didinant gijų skaičių, nekeičiant branduolių skaičiaus, mažėja sistemos greitaveika. Taip pat tendencija yra, kad algoritmas veikia greičiau su daugiau branduolių. Visus šiuos rezultatus galima panaudoti, kad sudaryti bendrines rekomendacijas kuriant sistemas su Java gijomis. Rekomendacijos galėtų būti:

- Daugiau gijų geriau, jei sistema veikia kompiuteryje su procesorium su keliais branduoliais.
- Nenaudoti gijų, jei kompiuteris turi procesorių su vienu branduoliu. Tai prailgina programos veikimo laiką. Visi atlikti bandymai pagrindžia šitą teiginį.
- Lygiagretaus programavimo problemų sprendimams naudoti užraktus „lock“ arba „semaphore“. Tyrimo rezultatai rodo, kad „semaphore“ užraktas yra efektyvesnis ir sutrumpiną programos veikimo laiką daugiau nei „lock“.

- Geriausi laikai pasiekiami, kai gijų skaičius atitinka procesoriaus branduolių skaičių.
- Kad įvertinti naujai sukurto algoritmo efektyvumą, greitaveiką su skirtingais parametrai patartina naudoti tam skirtą programinę įrangą, kuri buvo naudojama šio eksperimento metu.
- Esant galimybei atlikti tyrimus su daugiau branduolių, kaip 4-8 branduoliais. Kadangi vis plinta procesoriai su daugiau branduolių, reikalingi ir rezultatai, kad galima būtų daryti tinkamus sprendimus.

6. IŠVADOS

- Išsiaiškintos Java gijų panaudojimo svarbiausios problemos, sudaryti galimi sprendimo ir analizės būdai.
- Sukurta programinė įranga, kuri buvo panaudota algoritmų analizei atlikti. Programa padėjo atlikti tyrimus daug papraščiau ir sutaupant laiko. Taip pat leido išvengti netikslumų resursų matavimuose.
- Pastebėtos sukurtos sistemos tobulinimo galimybės, kurios programą praplėstų papildomais funkcionalumais ir ateityje, dar palengvintų naujų tyrimų atlikimą. Kaip papildomo parametrų nustatymas, sistemos apkrovimo koeficiento įvedimas.
- Sukurtoji sistema labai palengvina įvairių programų pagrindinių parametrų analizę. Galimybės leidžia analizuoti ne tik Java programas, bet ir programas kitomis kalbomis.
- Atliktas Java gijų tyrimas, kuris leido geriau supratsi gijų įtaką lygiagrečiom sistemoms.
- Sudaryti ir išanalizuoti algoritmai su skirtingais svarbiausių problemų sprendimais. Suprasta skirtingų sprendimų įtaka greitaveikai ir resursams.
- Išanalizavus visus eksperimento rezultatus, buvo pastebėta ir užfiksuota aiški gijų panaudojimo įtaka resursams. Tokiems, kaip, procesoriaus apkrovimas, operatyviosios atminties išnaudojimas, vykdymo laikas.
- Buvo išanalizuota tik maža dalis algoritmų ir jų variacijų. Likusios spendimų variacijos paskatina toliau vystyti šiuos tyrimus.
- Taip pat pastebėtas būtinumas vystyti tyrimus su didesniu branduolių skaičiumi, nes jau galima įsigyti tiek šešių, tiek aštuonių branduolių procesorius.
- Sudarytos bendrinės rekomendacijos, kurios turėtų pagreitinti apsisprendimą naudoti ar nenaudoti Java gijas ir kiek branduolių procesorių geriausia naudoti.

7. LITERATŪRA

[1] Straipsnis „Advanced Synchronization in Java Threads“ [Žiūrėta 2010-03-03], prieiga internete: http://onjava.com

[2] AIX Documentation. Kernel Extensions and Device Support Programming Concepts [Žiūrėta 2008 11 09], prieiga internete <http://rzdocs.uni-hohenheim.de/aix_4.33/ext_doc/nav/Books/nav_1.htm>

[3] Scott M.Fulton, AMD: Will more CPU cores always mean better performance? 2007 [Žiūrėta 2010 01 25], prieiga internete <http://www.betanews.com/article/AMD-Will-More-CPU-Cores-Always-Mean-Better-Performance/1187125077>

[4] Beowulf Project, The [Žiūrėta 2008 11 09], prieiga internete <http://www.beowulf.org>

[5] Hennessy, J. and Patterson, D. Computer Organization & Design. Morgan Kaufmann Publishers. San Francisco, 1998.

[6] Cory Quammen Introduction to Programming Shared-Memory and Distributed-Memory Parallel Computers, 2001.

[7] CSP for Java programmers, Part 2 Concurrent programming with JCSP [Žiūrėta 2008 10 20], prieiga internete <http://www.ibm.com/developerworks/java/library/j-csp2/>

[8] Eric Bruno [Žiūrėta 2010-05-03], prieiga internete <http://java.dzone.com/news/building-multi-core-ready-java>

[9] Gengbin Zheng, Orion Sky Lawlor, Laxmikant V. Kalé Multiple Flows of Control in Migratable Parallel Programs. University of Illinois at Urbana-Champaignm. Department of Computer Science, 2006.

[10] Straipsnis „Introduction to Java Concurrency / Multithreading“ [Žiūrėta 2010-03-05], prieiga internete: http://tutorials.jenkov.com/java-concurrency/index.html

[11] Scott Oaks, Henry Wong, JAVA threads 3rd Edition. O‘Reilly Media, Inc, 2004 [Žiūrėta 2010-02-03],

[12] Optimize Managed Code For Multi-Core Machines [Žiūrėta 2008 11 01], prieiga internete <http://msdn.microsoft.com/en-us/magazine/cc163340.aspx>

[13] Threading Building Blocks [Žiūrėta 2008 11 13], prieiga internete <http://www.threadingbuildingblocks.org/>.

8. TERMINŲ IR SANTRUMPŲ ŽODYNAS

POSIX - *Portable Operating System Interface*, operacinės sistemos sąsajos (*interface*) standartas.

JVM – Java virtualumo variklis (Java virtual machine).

Deadlock – situacija kai viena ar daugiau gijų laukia pabaigti darbą, bet nesulaukia.

Livelock – Kaip ir Deadlock, tik kad gijos viena kitai perduoda pirmenybę arba informaciją, bet niekad nebaigia darbo.

CSP – kalba aprašanti šablonus arba komunikavimą tarp procesų (Communicating Sequential Processes).

Kritinė sekcija – Kodo vieta, kurioje naudojami bendri resursai (duomenys), pasiekiami skirtingų gijų.

RAM – Laikinoji kompiuterio atmintis;

CPU – Centrinis kompiuterio procesorius;

Centrinė programa / sistemos branduolys – Programa, vykdanti algoritmų skaičiavimus serveryje;

Klientinė programa – programa skirta per nuotolį valdyti centrinę sistemos programą;

9. PRIEDAI

5 skyriuje naudotų algoritmų išeities tekstai:

MergeSort algoritmas su užraktu „lock“ ir „semaphore“. Abiejų sprendimų struktūra identiška, skiriasi, tik naudojamas užraktas, todėl pateiksime vieną algoritmo variantą su intarpais, kur naudojamas kitas sprendimo būdas. Pateikimas bus išeities tekstuose, komentaruose.

MAIN klasė:

```
package MergeSort1;
import java.util.Random;
public class Main {
 public static void main(String[] args) throws InterruptedException {
      String tc = args[0]; //Thread count
      int count = Integer.parseInt(tc);
      int count2 = 10000000;
      Random gen = new Random();
      int[] a = new int[count2];
      for( int i = 0; i < count2; i++)
        a[i] = gen.nextInt(1000000);
     int lenght = a.length;
     MergeSorter m = new MergeSorter();
      int lo;
      int hi;
      int step;
      int div;
      step = lenght/count;
      if (step < 0) step = 0;
      div = lenght - count*step;
      Threads[] t = new Threads[count+1];
      Lock[] lock =  new Lock[count+1];
 //Semaphore[] semaphore = new Semaphore[count+1];
      lo = 0;
      hi = 0;
      for (int i = 0; i < count+1; i++)
      {
       t[i] = new Threads();
       lock[i] = new Lock();
//semaphore[i] = new Semaphore(1);
      }
      for (int i = 0; i < count-1; i++)
```

```
        {
          hi = lo + step-1;
          t[i].run(lock[i], a, lo, hi, i+1);
//t[i].run(semaphore[i], a, lo, hi, i+1);
          lo = lo + step;
        }
        hi = lo + step-1;
        t[count].run(lock[count], a, lo, (hi+div), count);
//t[count].run(semaphore[count], a, lo, (hi+div), count);
        for (int i = 0; i < count; i++)
         lock[i].lock();
// semaphore[i].take();
        lo = 0;
        hi = 0;
        int hi2 = lo + step-1;
        for (int i = 0; i < count-1; i++)
        {
          hi = hi2;
          hi2 = hi + 1 + step-1;
          m.mergeArray(a, lo, hi, hi2);
        }
        for (int i = 0; i < count; i++)
         lock[i].unlock();
//semaphore[i].releas
    }
}
```

LOCK klasė:

```
package MergeSort1;
public class Lock{
  private boolean isLocked = false;
  public synchronized void lock()
  throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }

  public synchronized void unlock(){
    isLocked = false;
    notify();
  }
}
```

SEMAPHORE klasė:

```
package MergeSort1;
public class Semaphore {
  private int signals = 0;
  private int bound   = 0;
  public Semaphore(int upperBound){
    this.bound = upperBound;
  }
  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }
  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

MERGESORTER klasė:

```
package MergeSort1;
public class MergeSorter
{
   private static int[] a, b, c;    // auxiliary array b
   public static void sort(int[] a0)
   {
      a=a0;
      int n=a.length;
      b=new int[n];
      mergesort(0, n-1);
   }
   private static void mergesort(int lo, int hi)
   {
      if (lo<hi)
      {
         int m=(lo+hi)/2;
         mergesort(lo, m);
         mergesort(m+1, hi);
         merge(lo, m, hi);
      }
   }
   private static void merge(int lo, int m, int hi)
   {
```

```
        int i, j, k;
    for (i=lo; i<=hi; i++)
        b[i]=a[i];
    i=lo; j=m+1; k=lo;
    while (i<=m && j<=hi)
        if (b[i]<=b[j])
            a[k++]=b[i++];
        else
            a[k++]=b[j++];
    while (i<=m)
        a[k++]=b[i++];
    }
    //---------------------------------
    public static void mergeArray(int[] a, int lo, int m, int hi)
    {
        int i, j, k;
        int n=a.length;
        c=new int[n];
    for (i=lo; i<=hi; i++)
        c[i]=a[i];
    i=lo; j=m+1; k=lo;
    while (i<=m && j<=hi)
        if (c[i]<=c[j])
            a[k++]=c[i++];
        else
            a[k++]=c[j++];
    while (i<=m)
        a[k++]=c[i++];
    }
}   // end class MergeSorter
```

THREADS klasė:

```
package MergeSort1;
public class Threads extends Thread {
public void run(Lock lock, int[] a, int lo, int hi, int no) throws InterruptedException {
// public void run(Semaphore lock, int[] a, int lo, int hi, int no) throws InterruptedException {
        lock.lock();
// lock.take();
        int[] b = new int[hi-lo+1];
        for (int i=0; i <= hi-lo; i++)
            b[i] = a[lo+i];
        MergeSorter.sort(b);
        for (int i=0; i <= hi-lo; i++)
            a[lo+i] = b[i];
```

```
        lock.unlock();
// lock.release(
    }
    public static void main(String args[]) {
        (new Threads()).start();
    }
}
```

QuickSort algoritmas su užraktu „lock“ ir „semaphore“. Abiejų sprendimų struktūra identiška, skiriasi, tik naudojamas užraktas, todėl pateiksime vieną algoritmo variantą su intarpais, kur naudojamas kitas sprendimo būdas. Pateikimas bus išeities tekstuose, komentaruose. Lock ir Semaphore klasės nepateiksime, nes jos tokios pat, kaip ir MergeSort algoritme.

MAIN klasė:

```
package QuickSort;
import java.util.Arrays;
import java.util.Random;
public class Main {
  public static void main(String[] args) throws InterruptedException {
      String tc = args[0]; //Thread count
      int count = Integer.parseInt(tc);
      int count2 = 10000000;
      Random gen = new Random();
      int[] a = new int[count2];
      for( int i = 0; i < count2; i++)
        a[i] = gen.nextInt(1000000);
      int lenght = a.length;
      MergeSorter m = new MergeSorter();
      QuickSort q = new QuickSort();
      int lo;
      int hi;
      int step;
      int div;
      step = lenght/count;
      if (step < 0) step = 0;
      div = lenght - count*step;
      Threads[] t = new Threads[count+1];
      Lock[] lock =  new Lock[count+1];
// Semaphore[] semaphore = new Semaphore[count+1];
      lo = 0;
      hi = 0;
      for (int i = 0; i < count+1; i++)
      {
```

```
        t[i] = new Threads();
        lock[i] = new Lock();
//semaphore[i] = new Semaphore(1);
      }
      for (int i = 0; i < count-1; i++)
      {
        hi = lo + step-1;
        t[i].run(lock[i], a, lo, hi, i+1);
// t[i].run(semaphore[i], a, lo, hi, i+1
        lo = lo + step;
      }
      hi = lo + step-1;
      t[count].run(lock[count], a, lo, (hi+div), count);
// t[count].run(semaphore[count], a, lo, (hi+div), cou
      for (int i = 0; i < count; i++)
       lock[i].lock();
// semaphore[i].take();
      lo = 0;
      hi = 0;
      int hi2 = lo + step-1;
      for (int i = 0; i < count-1; i++)
      {
        hi = hi2;
        hi2 = hi + 1 + step-1;
        m.mergeArray(a, lo, hi, hi2);
      }
    for (int i = 0; i < count; i++)
      lock[i].unlock();
// semaphore[i].release();
   }
}
```

THREADS klasė:

```
package QuickSort;
public class Threads extends Thread {

   public void run(Lock lock, int[] a, int lo, int hi, int no) throws InterruptedException {
//public void run(Semaphore semaphore, int[] a, int lo, int hi, int no) throws InterruptedException {
      lock.lock();
// semaphore.take();
      System.out.println("Thread" + no);
      int[] b = new int[a.length];
      for (int i=0; i <= hi-lo; i++)
         b[i] = a[lo+i];
```

```
        QuickSort.quicksort(a, lo, hi);
        lock.unlock();
//semaphore.release();
   }
   public static void main(String args[]) {
      (new Threads()).start();
   }
}
```

Calc algoritmas su užraktu „semaphore“. Bus pateikti išeities tekstai tiek vienodam gijų darbui, tiek skirtingam. Semaphore klasė nebus pateikiama, nes ji tokia pat, kaip ir MergeSort arba QuickSort algoritmuose.

Gijos atlieką skirtingą darbą:

MAIN klasė:

```
package calc1_lock;
import java.util.Random;
public class Main {
 public static void main(String[] args) throws InterruptedException {
      String tc = args[0]; //Thread count
      int count = Integer.parseInt(tc);
      int count2 = 10000000;
      count = count/2;
      Random gen = new Random();
      int[] a = new int[count2];
      int[] b = new int[count];
      int[] c = new int[count];
      for( int i = 0; i < count2; i++)
        a[i] = gen.nextInt(10000);
     int lenght = a.length;
      int lo;
      int hi;
      int step;
      int div;
      step = lenght/count;
      if (step < 0) step = 0;
      div = lenght - count*step;
      Threads[] t = new Threads[count+1];
      Semaphore[] semaphore = new Semaphore[count+count+1];
      lo = 0;
      hi = 0;
      for (int i = 0; i < count+count+1; i++)
      {
       t[i] = new Threads();
```

```
            semaphore[i] = new Semaphore(1);
           }
           for (int i = 0; i < count-1; i++)
           {
            hi = lo + step-1;
            t[i].run(semaphore[i], a, b, lo, hi, i+1);
            lo = lo + step;
           }
           hi = lo + step-1;
           t[count].run(semaphore[count], a, b,lo, (hi+div), count);
           int j = count-1;
           //-----------------------------------
           for (int i = 0; i < count-1; i++)
           {
            hi = lo + step-1;
            t[i].run2(semaphore[j], a, c, lo, hi, i+1);
            lo = lo + step;
            j++;
           }
           hi = lo + step-1;
           t[count+count].run2(semaphore[count+count], a, c, lo, (hi+div), count);
           //---------------------------
           for (int i = 0; i < count+count; i++)
            semaphore[i].take();
           int sum = 0;
           int min = c[0];
           for (int i = 0; i < count; i++)
           {
            sum += b[i];
            if ( min < c[i] ) min = c[i];
           }
           for (int i = 0; i < count; i++)
             semaphore[i].release();
         }
       }
```

THREADS klasė:

```
       package calc1_lock;
       public class Threads extends Thread {
        public void run(Semaphore semaphore, int[] a, int[] b, int lo, int hi, int no) throws
InterruptedException {

           semaphore.take();
            System.out.println("Thread" + no);
```

```
            b[no-1] = 0;
            for (int i=lo; i <= hi; i++)
            {
               b[no-1] += a[i];
            }
             semaphore.release();
          }
        public void run2(Semaphore semaphore, int[] a, int[] c, int lo, int hi, int no) throws InterruptedException {
           semaphore.take();
            System.out.println("Thread" + no);
            c[no-1] = a[lo];
            for (int i=lo; i <= hi; i++)
            {
               if (c[no-1] < a[i] ) c[no-1] = a[i];
            }
             semaphore.release();
          }
          public static void main(String args[]) {
             (new Threads()).start();
          }
        }
```

Gijos atlieką tą patį darbą:

MAIN klasė:

```
package calc1_lock;
import java.util.Random;
public class Main {
 public static void main(String[] args) throws InterruptedException {
      String tc = args[0]; //Thread count
      int count = Integer.parseInt(tc);
      int count2 = 10000000;
      Random gen = new Random();
      int[] a = new int[count2];
      int[] b = new int[count];
      int[] c = new int[count];
      for( int i = 0; i < count2; i++)
        a[i] = gen.nextInt(10000);
      int lenght = a.length;
      int lo;
      int hi;
      int step;
      int div;
```

```
            step = lenght/count;
            if (step < 0) step = 0;
            div = lenght - count*step;
            Threads[] t = new Threads[count+1];
            Semaphore[] semaphore = new Semaphore[count+count+1];
            lo = 0;
            hi = 0;
            for (int i = 0; i < count+count+1; i++)
            {
             t[i] = new Threads();
             semaphore[i] = new Semaphore(1);
            }
            for (int i = 0; i < count-1; i++)
            {
              hi = lo + step-1;
             t[i].run(semaphore[i], a, b, c, lo, hi, i+1);
              lo = lo + step;
            }
            hi = lo + step-1;
            t[count].run(semaphore[count], a, b, c, lo, (hi+div), count);
            int j = count-1;
            //---------------------------
            for (int i = 0; i < count+count; i++)
              semaphore[i].take();
            int sum = 0;
            int min = c[0];
            for (int i = 0; i < count; i++)
            {
              sum += b[i];
              if ( min < c[i] ) min = c[i];
            }
            for (int i = 0; i < count; i++)
               semaphore[i].release();
          }
        }
```

THREADS klasė:

```
        package calc1_lock;
        public class Threads extends Thread {
         public void run(Semaphore semaphore, int[] a, int[] b, int[] c, int lo, int hi, int no) throws
InterruptedException {
            semaphore.take();
             System.out.println("Thread" + no);
             b[no-1] = 0;
```

```
            c[no-1] = a[lo];
            for (int i=lo; i <= hi; i++)
            {
               b[no-1] += a[i];
               if (c[no-1] < a[i] ) c[no-1] = a[i];
            }
             semaphore.release();
         }
         public static void main(String args[]) {
            (new Threads()).start();
         }
      }
```