VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Bendrosios atminties lygiagretaus programavimo modelių analizė ir eksperimentinis tyrimas

Analysis and experimental evaluation of shared memory parallel programming models

Magistro baigiamasis darbas

Atliko: Aurimas Smoliakas (parašas)

Darbo vadovas: Prof. dr. Rimantas Vaicekauskas (parašas)

Recenzentas: (parašas)

Vilnius – 2020

Santrauka

Šio magistro baigiamojo darbo tikslas buvo apžvelgti egzistuojančius bendrosios atminties lygiagretaus programavimo modelius, išsiaiškinti jų savybes bei pateikti rekomendacijas, kurios esant konkrečiai problemai, leistų nustatyti, kuris išlygiagretinimo modelis yra tinkamiausias jai spręsti.

Siekiant užsibrėžto tikslo darbe buvo išanalizuoti esami bendrosios atminties išlygiagretinimo modeliai. Toliau buvo išspręsti keturi uždaviniai su kiekvienu iš išlygiagretinimo modeliu. Galiausiai eksperimentiškai įvertinti uždavinių spartinimai ir nustatyta, kaip efektyviai modeliai sprendžia uždavinius.

Atlikus darbo užduotis buvo rasta, kad skirtingose uždavinių kategorijose išlygiagretinimo modeliai pasižymi skirtingais spartinimais. Vienomis savybėmis pasižymintys uždaviniai sprendėsi efektyviau su vienu modeliu, kitomis savybėmis pasižymintys uždaviniai – su kitais modeliais. Todėl darbo apibendrinime yra rekomenduojama rinktis išlygiagretinimo modelį pagal tai, kokia problema bus sprendžiama ir kaip tikėtina atrodys sprendimo kodo struktūra.

Raktiniai žodžiai: bendroji atmintis, išlygiagretinimo modeliai, analizė, eksperimentinis tyrimas, OpenMP, Cilk Plus, Threading Building Blocks, Google Go, spartinimas.

Summary

The goal of this work was to examine existing shared memory parallel programming models, figure out their properties and provide recommendations, which would let us determinate the most fitting model for a given problem.

To achieve the intended goal, existing shared memory parallel programming models were analyzed. After that, four problems were solved with each of the parallel programming models. Finally, problems' speedups were experimentally evaluated and the efficiency of models were identified for each problem.

After completing these tasks, it was found that different parallel programming models have varying speedups for different types of problems. Problems with one type of properties could be solved more efficiently with one model, while problems with other properties could be solved more efficiently with another model. Because of that, in the work's conclusions recommendations are given to choose parallel programming model based on the type of the problem and its solution's code structure.

Keywords: sharred memory, parallel programming models, analysis, experimental evaluation, OpenMP, Cilk Plus, Threading Building Blocks, Google Go, speedup

Turinys

1. Įvadas

Techninė įranga nuolat tobulėja vienokiu ar kitokiu būdu. Ne išimtis yra ir kompiuterių procesoriai. Ilgą laiką jie greitėjo dėl didėjančio procesoriaus taktavimo dažnio. Šis dažnis nurodo, kiek operacijų per sekundę procesorius gali atlikti, todėl jis tiesiogiai įtakoja nuoseklių programų veikimo greitį. Tačiau pastaraisiais metais taktavimo dažnis beveik nebedidėja dėl tam tikrų fizinių priežasčių. Dėl to vietoje taktinio dažnio didinimo procesorių gamintojai pradėjo gaminti procesorius su keliais savarankiškai veikiančiais branduoliais ir šių branduolių kiekis didėja kas kelios procesorių kartos. Šie branduoliai leidžia procesoriui vykdyti kelias operacijas vienu metu.

Branduolių kiekis procesoriuje nuoseklių programų greičio taip tiesiogiai nebeįtakoja kaip taktavimo dažnis, todėl norint pagreitinti nuoseklią programą, ją reikia paversti į lygiagrečią, kuri per tą patį laiko tarpą gali atlikti daug daugiau skaičiavimų. Dėl tos pačios priežasties lygiagretumas yra vienas iš bazinių reikalavimų naujoms programoms rašyti.

Kita priežastis, kodėl lygiagretumas yra reikalingas, yra tai, jog pasaulis pats savaime yra labai lygiagretus. Todėl modeliuoti ar prognozuoti jo reiškinius lygiagrečiai yra daug natūraliau ir paprasčiau. Pavyzdžiui, kaip galėtume modeliuoti oro pokyčius tam tikroje teritorijoje ar planetų judėjimą nuosekliai?

Dėl lygiagrečių algoritmų efektyvumo lygiagretus programavimas, su juo susijusių mokslinių tyrimų atlikimas, karkasų, bibliotekų ar kitokios programinės įrangos kūrimas šiais laikais yra labai aktuali kompiuterių mokslo tema.

Deja, neretai lygiagretūs algoritmai būna sudėtingesni už nuoseklius, todėl atsiranda visa aibė naujų problemų, kurias tenka spręsti. Tai gali būti gijų sinchronizavimas, lenktynių būklės išvengimas ar bendros atminties dalijimasis tarp gijų. Bendrosios atminties valdymas ir naudojimas yra vienas iš sudėtingesnių lygiagretaus programavimo uždavinių. Dėl to būtent bendroji atmintis lygiagrečiame programavime ir jos modelių analizė yra šio darbo tema.

Egzistuoja daug skirtingų priemonių, kurios leidžia kurti lygiagrečius algoritmus, naudojančius bendrąją atmintį. Tai gali būti operacijų vykdymo modeliai (pvz. POSIX Threads), programavimo kalbų praplėtimai (pvz. OpenMP), programavimo kalbų bibliotekos (pvz. Intel TBB) ir t. t. Taip pat naujosios programavimo kalbos (pvz. Google Go) dažnai būna specialiai optimizuotos lygiagretiems algoritmams rašyti ir naudoja naujus lygiagretaus programavimo modelius [Sum12].

Minėtosios ir panašios priemonės įprastai būna orientuotos konkrečiam ratui uždavinių spręsti, todėl nėra absoliučiai geriausio sprendimo, kuris tiktų bet kuriam atvejui [PBF10]. Kyla problema**,** kurį lygiagretaus programavimo instrumentą ar modelį pasirinkti, kai turime tam tikro tipo uždavinį ir siekiame jį išspręsti ar optimizuoti.

Šiame magistriniame darbe minėtoji problema ir yra nagrinėjama.

Norint išspręsti išsikeltą problemą, šio darbo tikslas yra apžvelgti egzistuojančius bendrosios atminties lygiagretaus programavimo modelius, išsiaiškinti jų savybes bei pateikti rekomendacijas, kurios esant konkrečiai problemai, leistų nustatyti, kuris išlygiagretinimo modelis yra tinkamiausias jai spręsti.

Siekiant išsikelto tikslo yra sprendžiami tokie uždaviniai:

1. Detaliai išanalizuoti pasirinktus esamus bendrosios atminties išlygiagretinimo modelius. Į šį uždavinį įeina modelių struktūros ir taikymo srities nagrinėjimas, ekvivalentumas lyginant su kitais modeliais.
2. Analitiškai nustatyti tam tikrą skirtingomis savybėmis pasižyminčių uždavinių ratą, kuriam priklausantys uždaviniai gali būti išlygiagretinti vienu ar kitu modeliu.
3. Identifikuoti optimalius pasirinktų uždavinių išlygiagretinimo būdus ir eksperimentiškai įvertinti jų spartinimą. Remiantis tuo nustatyti, kurie modeliai kuriems uždaviniams geriausiai tinka.

2. Bendrosios atminties lygiagrečiųjų kompiuterių architektūra

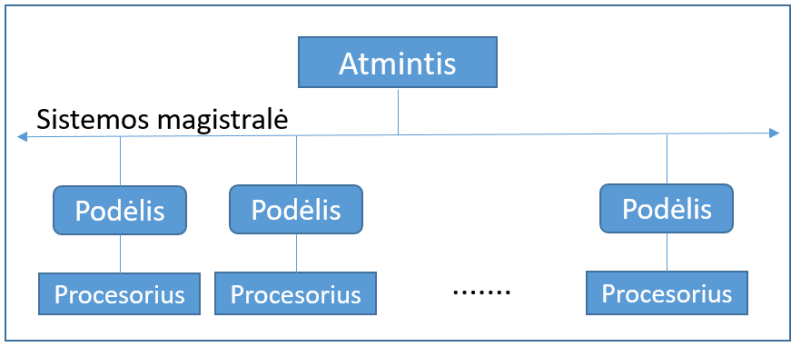
Bet kuris lygiagretus kompiuteris vienu metu įprastai naudoja daugiau nei vieną procesorių arba branduolį. Kiekvienas jų sprendžia tam tikras užduotis. Užduotims spręsti dažniausiai reikia atminties, kurioje laikoma užduoties būsena. Pagal skirtingą atminties naudojimą skiriamos dvi lygiagrečiųjų kompiuterių atminties architektūros:

1. Bendrosios atminties
2. Paskirstytos atminties

Šiame darbe nagrinėjama bendrosios atminties architektūra.

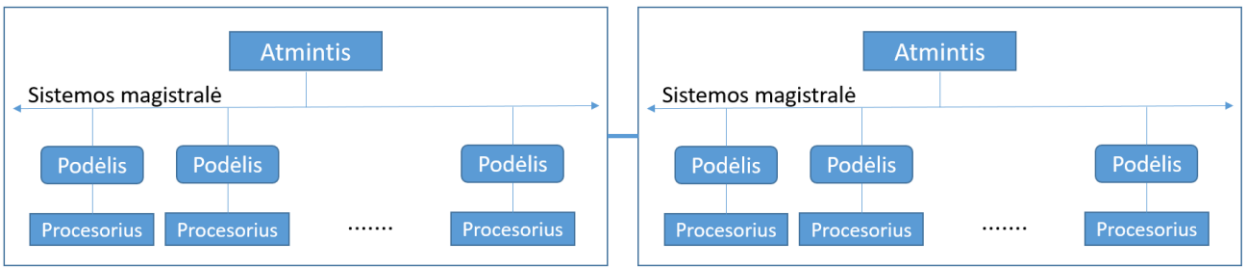
Bendrosios atminties architektūros pagrindinė savybė yra tai, jog kiekvienas kompiuterio procesorius turi prieigą prie visos kompiuterio atminties. Tai reiškia, kad keli procesoriai vienu metu dirbdami atskirai gali prieiti prie tos pačios atminties vietos ir matyti vienas kito pakeitimus atmintyje [Dun90].

Jeigu visi procesoriai atmintį pasiekia vienodu greičiu, tai toks kompiuteris turi tolygiai pasiekiamą bendrąją atmintį (angl. UMA – Uniform Memory Access) (1 pav.). Tokia atmintis dažniausiai naudojama simetriniuose multiprocesoriniuose (trump. SMP) kompiuteriuose, kurie turi kelis ar daugiau procesorių su nuosavais podėliais, tačiau dalijasi viena atmintimi, operacine sistema ir kitais komponentais.



1 pav. Tolygiai pasiekiamos atminties (UMA) kompiuterio schema

Galima sujungti kelis SMP kompiuterius fiziškai. Tada vienas SMP kompiuteris gali tiesiogiai pasiekti atmintį iš kito SMP kompiuterio. Tačiau tada viename SMP esanti atmintis iš to paties SMP procesoriaus yra pasiekiama greičiau, negu ji būtų pasiekiama iš kito SMP kompiuterio procesoriaus. Tada sakome, jog kompiuteriai yra su netolygiai pasiekiama bendrąja atmintimi (angl. NUMA – Non-Uniform Memory Access) (2 pav.).



2 pav. Netolygiai pasiekiamos atminties (UMA) kompiuterio schema

Pagrindiniai tokios architektūros privalumai yra tai, jog sistemai yra nesunku prieiti prie bet kurios atminties vietos, nes visa atmintis yra globali. Dėl tos pačios priežasties duomenų dalijimasis tarp sistemos užduočių yra labai paprastas [HJ16].

Didžiausias šios architektūros trūkumas yra plėtimasis tarp procesoriaus ir atminties. Naujų procesorių pridėjimas padidina magistralės užimtumą tarp procesoriaus ir atminties. Jeigu sistema naudoja podėlius, padidėja ir magistralės užimtumas tarp podėlių ir atminties.

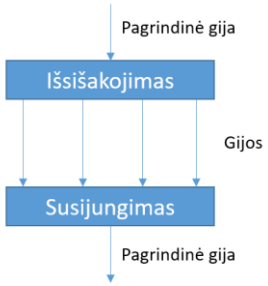
3. Bendrosios atminties lygiagretaus programavimo modeliai

Šiame skyriuje bus nagrinėjami bendrosios atminties lygiagretaus programavimo modeliai.

3.1. OpenMP

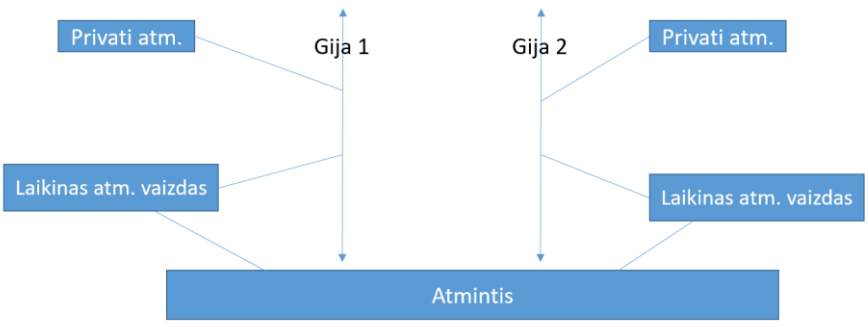
OpenMP yra programavimo standartas, skirtas lygiagretiems skaičiavimams naudojant bendrosios atminties kompiuterius. Jis yra skirtas C, C++ ir Fortran programavimo kalboms ir didžioji dalis kompiliatorių palaiko šį standartą [OAB18]. Standartas yra kuriamas konsorciumo, vadinamo "OpenMP Architecture Review Board". Jį sudaro nariai iš stambiausių techninės ir programinės įrangos gamintojų, tokių kaip IBM, Intel, Nvidia, Red Hat, Oracle ir kitų.

OpenMP remiasi šakok-sujunk (fork-join) lygiagretaus vykdymo modeliu. Modelis veikia taip, jog programa yra pradedama vykdyti nuosekliai pagrindinėje gijoje. Priėjusi tam tikrą programos dalį, vadinamą lygiagrečiu regionu, ši gija "išsišakoja" į kelias gijas, kurios kiekviena vykdo sekančią programos kodo dalį lygiagrečiai. Pasibaigus lygiagrečiam regionui, naujosios gijos yra sunaikinamos ir iš jų lieka viena, originalioji pagrindinė gija – gijos "susijungia". Kodo dalis tarp išsišakojimo ir susijungimo yra vadinama lygiagrečiuoju regionu.



3 pav. Šakok-sujunk modelis

Išsišakojimo metu sukurtos gijos yra pagrindinės gijos kopijos – jos turi tokią pačią prieigą prie to, kas buvo apskaičiuota prieš išsišakojimą. Tačiau kiekviena gija gali turėti ir privačius duomenis bei gija turi priėjimą prie informacijos apie save ir kitas gijas. Supaprastinta programos atminties struktūra pavaizduota 4 paveiksle.



4 pav. OpenMP atminties modelio struktūra

Kaip matoma iš paveikslėlio, OpenMP programose visos gijos gali skaityti ir rašyti kintamuosius į tam tikrą atminties vietą, kuri standarte tiesiog vadinama *atmintimi*. Papildomai

kiekviena gija gali turėti savo laikiną atminties vaizdą (angl. temporary view of the memory). Tai gali būti mašinos registrai, podėlis (angl. cache) ar kitokia lokali duomenų talpykla, esanti tarp gijos ir atminties. Laikinas atminties vaizdas leidžia podėliuoti kintamuosius ir išvengti kreipimosi į atmintį dėl kiekvieno kintamojo. SMP kompiuteriuose tai galėtų būti problema – kai vienas procesorius pakeičia skaičiavimų reikšmes ir jas patalpina į podėlį, naujas reikšmes mato tik tas kodas, kuris yra vykdomas ant to procesoriaus. Kiti SMP kompiuterio procesoriai matytų tik senas reikšmes. Šis reiškinys vadinamas atminties vientisumo problema (angl. memory consistency problem). Kai kurie kompiuteriai turi įmontuotus sprendimus šiai problemai. Jie tada vadinami nuoseklaus podėlio (angl. cache coherent) kompiuteriais.

Laimei, programos kūrėjui nėra svarbu, kaip podėlio nuoseklumas veikia kompiuteryje, kuriame paleista OpenMP programa. Tame kompiuteryje gali net nebūti nuoseklaus podėlio, nes OpenMP specifikacija apibrėžia savas taisykles, kaip reikia sinchronizuoti bendrus duomenis tarp skirtinguose procesoriuose veikiančių gijų. Šiose taisyklėse teigiama, kad bendri duomenys turi būti matomi visoms gijoms sinchronizacijos taškuose. Tarp sinchronizacijos taškų gijos gali laikyti jų pakeistus duomenis lokaliuose podėliuose. Dėl to įmanoma situacija, kuomet gijos laikinai turi skirtingas bendrų duomenų reikšmes. Jeigu vienai gijai reikia reikšmės iš kitos gijos, tuomet toje kodo vietoje turi būti pridėtas sinchronizacijos taškas.

Pagal nutylėjimą OpenMP užtikrina, kad gija turi laukti, kol visos gijos lygiagrečiame regione baigs darbą. Baigus darbą bendra atmintis yra susinchronizuojama. Taip pat galima įdėti barjerą į kodą, ties kuriuo reikėtų susinchronizuoti duomenis.

Kitas būdas susinchronizuoti duomenis yra užtikrinti, jog tik viena gija vykdys kodo bloką vienu metu. Tokie kodo blokai vadinami kritinėmis sekcijomis. Jų pradžioje ir pabaigoje bendra atmintis yra susinchronizuojama. Panašus būdas yra naudoti užraktus (angl. locks), kuriais irgi galima užtikrinti, kad tik viena gija vykdo kodo dalį ir sinchronizuoti duomenis.

OpenMP taip pat turi atminties sinchronizavimo funkciją *flush*. Ši funkcija užtikrina, kad ją kviečianti gija turi tokias pačias bendrų duomenų reikšmes, kaip ir bendroji atmintis. Tai pasiekiama įrašius gijoje esančias reikšmes į bendrąją atmintį ir gijai gavus visas naujas bendrų duomenų reikšmes iš kitų gijų pakeitimų.

Kiekviena gija taip pat turi priėjimą prie dar vienos atminties rūšies – privačios gijos atminties (angl. threadprivate memory). Privačioje gijos atmintyje yra laikomi tik tai gijai matomos kintamųjų reikšmės. Priklausomai nuo situacijos, privačius kintamuosius gali sukurti programa (pvz. For cikle kiekviena gija turi savo privačią iteracijos kintamojo reikšmę) arba programos kūrėjas turi apibrėžti, kurie kintamieji lygiagrečiame regione bus privatūs, o kurie – vieši. [CJP08] Privati atmintis pravarti tuo, jog ji sumažina poreikį sinchronizuotis tarp gijų ir atnaujinimų bendrojoje atmintyje, taip minimizuojant vietas, į kurias gali kreiptis daug gijų.

Vienas iš trūkumų, jog privatiems kintamiesiems reikia daugiau atminties, negu tokiems pat viešiems.

Vykdymo metu kiekvienos gijos privačiai atminčiai yra išskiriami atminties blokai, vadinami gijos steku (angl. thread stack). Steko dydis įprastai būna kompiliatoriaus parenkamas pagal nutylėjimą, bet programos kūrėjas gali jį padidinti, jeigu gijai reikia saugoti daug duomenų.

OpenMP standartas suteikia įrankius, kurie yra skirstomi į tris grupes:

- kompiliatoriaus direktyvos (angl. compiler directives)
- bibliotekos funkcijos (angl. runtime functions/routines)
- aplinkos kintamieji (angl. environment variables)

3.1.1. Kompiliatoriaus direktyvos

Kompiliatoriaus direktyvos kode aprašomos kaip komentarai, todėl jie yra ignoruojami kompiliatoriuose, kurie jų nepalaiko, bet suprantami kompiliatoriuose, kurie jas palaiko. Direktyvos gali būti naudojamos įvairiems tikslams, pavyzdžiui, paskirstyti kodo blokus per kelias gijas, paskirstyti ciklo iteracijas tarp gijų, paleisti naują lygiagretų regioną, susinchronizuoti lygiagrečių gijų regionų duomenis. Direktyvos sintaksė susideda iš trijų dalių: raktinių žodžių, direktyvos pavadinimo ir direktyvos argumentų. Pavyzdžiui,

```
#pragma omp parallel default(shared) private(var1)
```

kur #pragma omp nurodo, kad tai yra OpenMP direktyva, parallel yra direktyvos pavadinimas, default(shared) ir private(var1) yra direktyvos argumentai – *default(shared)* nusako, kad visi argumentuose nepaminėti kintamieji regione bus bendri (angl. shared), *private(var1)* nusako, kad kintamasis *var1* regione bus privatus kiekvienai gijai.

Iš direktyvų galima sudaryti tam tikras dažnai naudojamas struktūras, kurios įgalina vieną ar kitą funkcionalumą. Jos vadinamos konstruktais. Panagrinėkime pagrindinius – lygiagretumo ir darbo dalijimosi OpenMP konstruktus.

***Parallel* konstruktas**

```
#pragma omp parallel [parametrai]
     kodo blokas
```

Vienai gijai priėjus prie parallel konstrukto, yra sukuriama gijų grupė. Gija, priėjusi direktyvą, tampa pagrindine gija su gijos numeriu 0. Tada visos grupės gijos, įskaitant ir pagrindinę giją, vykdo kodo bloką, esantį iki direktyvos pabaigos. Jeigu bent viena gija nutraukia darbą, visos grupės gijos taip pat nutraukia darbą.

Darbo dalijimosi (angl. worksharing) konstruktai

Paleidus tik parallel konstruktą, kiekviena gija grupėje vykdys kodo regioną. Tai nepagreitins programos veikimo. Tam, kad programos kūrėjas galėtų apibrėžti, kaip darbas regione turi būti paskirstytas tarp skirtingų gijų, yra skirtas darbo dalijimosi konstruktas.

Darbo dalijimosi konstruktas paskirsto kodo regiono vykdymą tarp grupės gijų. Šis konstruktas neturi sinchronizacijos barjero pradžioje, bet jį turi regiono pabaigoje (jeigu nėra nurodyta kitaip). Taip pat jis nesukuria naujų gijų, todėl norint lygiagretumo, jis turi būti lygiagretaus regiono viduje.

Tikriausiai dažniausias konstrukto naudojimo atvejis yra darbo paskirstymas *for* cikle. Tam programos kūrėjui reikia įdėti direktyvą prieš pat for ciklą. Tada pagal direktyvos konfigūraciją ciklo iteracijos paskirstomos kiekvienai gijos grupei – pavyzdžiui, turint 100 iteracijų ciklą ir 5 gijas, galima sukonfigūruoti, kad pirma gija apdorotų iteracijas nuo 1 iki 20, antra gija apdorotų iteracijas nuo 21 iki 40 ir t. t. Žinoma, gijų apdorojamas iteracijas galima apskaičiuoti ir dinamiškai, jeigu algoritmas reikalauja, kad tam tikra gija galėtų apdoroti tik tam tikrą iteraciją.

Kitas būdas paskirstyti darbą yra naudoti sekcijas (angl. sections). Jos leidžia paskirstyti kodo blokus lygiagrečiame regione į dalis, kurių kiekvieną įvykdo tik viena gija. Taip pat galima pasiekti, kad kodo bloką lygiagrečiame regione įvykdytų tik viena gija.

3.1.2. Bibliotekos funkcijos

OpenMP standartas apibrėžia nemažai funkcijų ir procedūrų, skirtų darbui su lygiagretumu. Vienos suteikia prieigą dinamiškai perrašyti ar nuskaityti standartines aplinkos kintamųjų reikšmes, kitos suteikia įrankius valdyti užraktus, trečios įgalina prieigą prie laiko gavimo ir t. t.

Verta paminėti keletą su anksčiau minėtomis sinchronizacijos, užraktų ir lygiagretumo struktūromis susijusių funkcijų ir procedūrų:

1. *void omp_set_num_threads(int num_threads)* – nurodo, kiek gijų turi būti sukurta sekančiame lygiagrečiame regione, jeigu jame nėra nurodytas num_threads argumentas
2. *int omp_get_thread_num()* – grąžina kviečiančiosios gijos numerį grupėje.
3. *int omp_in_parallel()* – grąžino teigiamą sveikąjį skaičių, jeigu kviečiančioji gija yra lygiagrečiame regione.
4. *void omp_init_lock(omp_lock_t * lock)* – inicializuoja OpenMP užraktą.
5. *void omp_destroy_lock(omp_lock_t *lock)* – pakeičia užraktą iš inicializuoto į neinicializuotą

3.1.3. Aplinkos kintamieji

OpenMP standarte yra aprašyta virš 20 aplinkos kintamųjų [OAB18]. Šių kintamųjų keitimas startavus programai (netgi jeigu pati programa juos keičia) yra ignoruojamas, todėl jie visi programos veikimo metu yra konstantos. Norimas jų reikšmes reikėtų priskirti prieš pradedant vykdyti programą.

1. *OMP_STACKSIZE* – nustato aplinkos gijos steko dydį.
2. *OMP_THREAD_LIMIT* – nustato didžiausią gijų grupės gijų kiekį.
3. *OMP_MAX_ACTIVE_LEVELS* – nustato lygį, kiek giliausiai gali būti vienas kitame esančių lygiagrečių regionų.
4. *OMP_SCHEDULE* – nustato lygiagrečių ciklų vykdymo gijų kūrimą. Pavyzdžiui, galima nustatyti, kad lygiagretų ciklą vykdytų 4 gijos. Taip pat galima nustatyti, kad gijų skaičius būtų nustatomas dinamiškai.
5. *OMP_DYNAMIC* – nustato, ar lygiagretaus regiono gijų kiekis grupėje gali būti kintantis.

3.2. Cilk Plus

Cilk Plus yra praplėtinys C ir C++ programavimo kalboms.

Pirmoji versija, pavadinta Cilk, buvo sukurta Masačusetso technologijų institute 1994 metais. Vėliau komercializuota įmonės “Cilk Arts” ir pervadinta į Cilk++. Galiausiai “Cilk Arts” buvo nupirktas Intel ir Intel šią programavimo kalbą pertvarkė su praplėstu C/C++ kalbomis rašytų programų palaikymu ir pervadino į Cilk Plus. Taip pat Cilk Plus palaikymas buvo pridėtas į Intel C++ kompiliatorių.

Pagrindinis Cilk Plus tikslas yra praplėsti C ir C++ kalbas su įrankiais, skirtais implementuoti lygiagretumą. Kaip ir OpenMP, taip ir Cilk Plus lygiagretumas yra paremtas šakok-sujunk (fork-join) lygiagretaus vykdymo modeliu [CSM+15]. Ir panašiai kaip OpenMP, Cilk Plus programas galima rašyti taip, kad tos pačios programos rezultatai būtų vienodi, nesvarbu, ar ji buvo vykdoma nuosekliai ar lygiagrečiai. Šios programavimo kalbos kūrėjai rekomenduoja tai daryti, nes nuoseklias programas lengviau suprasti ir skaityti.

Cilk Plus, priešingai nei OpenMP, užduotis gijoms paskirsto ne su darbo dalijimosi, bet su darbo perskirstymo (angl. work stealing) strategija.

Darbo perskirstymo strategijoje kiekvienas procesoriaus branduolys laiko sukurtų užduočių eilę (angl. task queue). Bet kuriuo metu branduolys gali sukurti naują užduotį (angl. task), kurią galima vykdyti lygiagrečiai su dabartine užduotimi. Tokia užduotis irgi padedama į branduolio užduočių eilę. Kai branduolys nebeturi užduočių savo eilėje, jis gali pažiūrėti į eiles kitame procesoriuje ir „pavogti“ užduotis iš jo eilės apačios (t. y. užduotis, kurias procesorius vykdytų

vėliausiai). Tada gaunasi, kad lygiagrečios užduotys pasisklaido per darbo neturinčius branduolius. O kol visi procesoriaus branduoliai yra apkrauti, nereikia atlikti jokio papildomo užduočių perkėlimo į skirtingus branduolius.

Šią ir kitas lygiagretumo įgyvendinimo strategijas Cilk Plus implementuoja naudojant keletą įrankių:

1. Raktinius žodžius (angl. keywords)
2. Reduktorius (angl. reducers)
3. Masyvų žymėjimą (angl. array notation)
4. SIMD–palaikančias funkcijas (angl. SIMD–Enabled functions)
5. #pragma simd direktyvą (angl. #pragma simd)

3.2.1. Raktiniai žodžiai

Cilk Plus standarte apibrėžti trys raktiniai rodžiai:

1. _Cilk_spawn
2. _Cilk_sync
3. _Cilk_for

Tokie vardai pasirinkti dėl to, kad tenkintų C/C++ kalbų praplėtimo standartus. Tačiau cilk.h failas apibrėžia šiuos vardus trumpiau: cilk_spawn, cilk_syn ir cilk_for. Paprastumo dėlei jie bus naudojami toliau darbe.

Raktažodis cilk_spawn yra rašomas prieš funkcijos kvietimą. Jis pasako, kad ši funkcija gali būti vykdoma lygiagrečiai su funkcijos kvietėju ir kvietėjui nereikia laukti funkcijos rezultatų grąžinimo. Tai reiškia, jog šis raktinis žodis praneša programai apie galimą lygiagretumą ir sukuria naują užduotį, kurios rezultatas yra įvykdyta funkcija su jos rezultatais. Tada Cilk Plus vykdytojas (angl. runtime) gali pasirinkti, ar užduotį vykdyti lygiagrečiai kvietėjui.

Raktažodis cilk_sync funkcijoje nusako, kad visos vaikinės dėl cilk_spawn sukurtos užduotys turi baigti darbą prieš tęsiant kviečiančios funkcijos kodo vykdymą. Verta pažymėti, kad šis raktažodis priverčia laukti užbaigto darbo tik iš vaikinių užduočių. Taip pat cilk_sync yra nutylėtai pridedamas prieš pat kiekvienos, cilk_spawn raktažodį turinčios funkcijos, pabaigą.

Cilk Plus programa įprastai turėtų turėti apie 10*B užduočių, kur B yra mašinos procesoriaus branduolių skaičius. Toks skaičius yra pakankamas užimti kitus branduolius, jeigu vienas branduolys vykdo ilgą užduotį.

Kiekvienam funkcijos kvietimui nėra rekomenduojama kurti naujos užduoties su cilk_spawn, nes užduoties kūrimas kainuoja apie 10 kartų daugiau, negu paprastas funkcijos kvietimas. Didžiąją tos kainos dalį sudaro procesorių darbo perskirstymas [ICP20].

Paskutinis raktažodis cilk_for yra skirtas ciklams su viena nuo kitos nepriklausančiomis iteracijomis įgyvendinti. Jis yra pakaitalas raktažodžiui for.

Sakykime, turime for ciklą:

```
for (int i = 0; i < 16; ++i)
{
     do_something(i);
}
```

kuriame do_something funkcija yra pakankamai greitai įvykdoma bei iteracijos yra visiškai nepriklausomos viena nuo kitos. Iteracijos bus vykdomos viena po kitos nuosekliai, nors jos galėtų būti vykdomos lygiagrečiai. Lygiagretumą galima pridėti ir su cilk_spawn bei cilk_sync:

```
for (int i = 0; i < 16; ++i)
{
     cilk_spawn do_something(i);
}
cilk_sync;
```

Tačiau taip būtų prikurta 16 užduočių, skirtų įvykdyti pakankamai greitai įvykdomai funkcijai, kurias procesorių branduoliai blogiausiu atveju turėtų „pavogti“ 15 kartų. O kaip buvo minėta anksčiau, perskirstymo operacija yra daug lėtesnė, negu funkcijos kvietimas. Be to, trūkumas dar ir tas, jog visas užduotis sukuria tik vienas branduolys ir padeda į savo užduočių eilę, kai idealiausiu atveju užduotys turėtų būti sudėtos į skirtingų branduolių eiles. Be abejo, būtų galima implementuoti algoritmą, kuris į visą tai atkreipia dėmesį ir paskirsto darbą kiek galima lygiau, tačiau tai jau atlieka cilk_for raktažodis. Su juo ciklas atrodytų taip:

```
cilk_for (int i = 0; i < 16; ++i)
{
     do_something(i);
}
```

Šiuo atveju Cilk Plus kompiliatorius ir vykdytojas paskirstys ciklo iteracijas per pusę skirtingiems procesoriams, tada dar kartą per pusę ir taip toliau, iki kol iteracijos yra paskirstomos per visus branduolius, bet tuo pačiu minimizuoja potencialius užduočių perskirstymus iš skirtingų branduolių.

3.2.2. Reduktoriai

Reduktoriai yra duomenų struktūros, skirtos dalintis atmintimi išvengiant lenktynių būsenos (angl. race condition) ir užraktų. Įprastai užraktai yra reikalingi norint naudoti bendrus kintamuosius. Tai sukelia problemų, nes neteisingai išdėstyti užraktai gali privesti prie aklavietės (angl. deadlock), kai gijos laukia viena kitos. Netgi kai užraktai yra sudėti teisingai, jie verčia vienas gijas sustoti ir laukti kitų gijų darbo baigimo. Taip pat dažnai vien su užraktais negalima užtikrinti eiliškumo ir programos rezultatai tampa nenuspėjamais.

Reduktoriai padeda spręsti šias problemas laikydami privačius “žvilgsnius” kiekvienoje gijoje. Tada norint gauti galutinį reduktoriaus rezultatą, reduktoriai sulieja žvilgsnių rezultatus ir

grąžina galutinį rezultatą, kuris yra išlaikęs teisingą eiliškumą. Paprasčiausias būdas tai apibūdinti yra panagrinėti pavyzdį.

Sakykime, jog turime programą, kurioje reikia sudėlioti skaičius nuo 1 iki 10 eilės tvarka į sąrašą. Sąrašas užpildomas lygiagrečiame cilk_for cikle.

```
mutex mtx;
std::list<int> numbers;

cilk_for(int i = 1; i<= 10; ++i)
{
     mtx.lock();
     numbers.push_back(i);
     mtx.unlock();
}
```

Std:list operacijos negarantuoja teisingų rezultatų jas vykdant lygiagrečiai, todėl skaičiaus pridėjimui į sąrašą reikia užraktų. Galutiniame rezultate nėra žinoma, kokia tvarka bus vykdomos ciklo iteracijos, todėl sąraše skaičiai nebūtinai bus pridėti eilės tvarka nuo 1 iki 10.

Tokią programą galima perrašyti su reduktoriumi reducer_list_append, kuris yra skirtas pridėti elementus į sąrašo pabaigą.

```
cilk::reducer<cilk::op_list_append<int>> numbers_reducer;

cilk_for(int i = 1; i<= 10; ++i)
{
     numbers_reducer->push_back(i);
}
const std::list<int> &numbers = numbers_reducer.get_value();
```

Šiuo atveju sąraše numbers elementai yra išsidėstę didėjančiai nuo 1 iki 10.

Cilk Plus bibliotekoje yra paruošti dažniausiai naudojami reduktoriai, skirti sąrašui su galimybe pridėti elementus į pabaigą/pradžią, apskaičiuoti didžiausią, mažiausią reikšmes arba sumą iš elementų rinkinyje ir t. t. Taip pat galima sukurti ir asmeninius reduktorius pagal poreikius.

3.2.3. Vektorizacijos įrankiai

Tam tikrais atvejais yra poreikis atlikti vienodas operacijas visam ar daliai masyvo elementų. Turbūt dažniausiai naudojamas būdas tai padaryti yra su ciklu. Pavyzdžiui:

```
for (int i = 0; i < A.size(); i++)
     A[i] = 10;
```

Toks sprendimas turi trūkumą, nes kompiliatorius nebūtinai supras, kad šį kodą galima vektorizuoti (angl. vectorize). Vektorizacija yra kompiliatoriaus optimizacija, kuri vietoj to, kad sukurtų daug tų pačių operacijų su skirtingais duomenimis, sukuria vieną operaciją su rinkiniu duomenų. Įprastai procesoriai palaiko šią funkciją ir sugeba apdoroti pateiktą rinkinį duomenų lygiagrečiai.

Cilk Plus turi tris įrankius kodo išlygiagretinimui naudojant vektorizaciją.

Pirmasis įrankis yra kalbos konstruktai, kurie vadinami masyvų žymėjimais. Nagrinėtas pavyzdys su jais atrodytų taip:

```
A[0:A.size():1] = 10; arba A[:] = 10;
```

Čia naudojamas operatorius *masyvas[apatinė reikšmė : ilgis : žingsnis]*, kur apatinė reikšmė (pagal nutylėjimą – 0) nurodo, nuo kurios masyvo vietos vykdyti skaičiavimą, ilgis (pagal nutylėjimą – masyvo ilgis) nurodo, keliems elementams nuo apatinės reikšmės vykdyti skaičiavimą ir žingsnis (pagal nutylėjimą yra 1) nurodo, kas kelintam elementui vykdyti skaičiavimą.

Priskyrimą galima atlikti ir kviečiant funkcijas kitam masyvo elementui:

```
A[:] = some_function(B[:]);
```

Šiame pavyzdyje funkcija some_function yra kviečiama kiekvienam B masyvo elementui ir tada priskyrimas yra vektorizuotas. Galima dar labiau optimizuoti kodą – nurodyti, kad funkciją some_function galima iškviesti tiek su vienu argumentu, tiek su argumentų masyvu. Tam reikėtų prieš funkcijos apibrėžimą pridėti žymėjimą __doclspec(vector) arba __attribute__((vector)):

```
__declspec(vector)
int some_function(int nr);
```

Abu žymėjimai pasako, kad kompiliatoriui reikia sugeneruoti atskirą kodą abiem funkcijos kvietimo atvejams – vienam argumentui arba SIMD – palaikančią funkciją, pritaikytą masyvui.

Paskutinis vektorizacijos įrankis Cilk Plus pakete yra #pragma simd direktyva. Ji kompiliatorių priverčia naudoti vektorizaciją ciklui įvykdyti. Direktyva praverčia tais atvejais, kai pats kompiliatorius gali nesugebėti nustatyti, jog ciklas yra tinkamas vektorizacijai. Tokiu atveju virš ciklo užrašius direktyvą, jis yra vektorizuojamas. Taip pat direktyva turi papildomų parametrų, kurie kompiliatoriui padeda sugeneruoti teisingai vektorizuotą kodą.

3.3. Intel Threading Building Blocks

Intel Threading Building Blocks (toliau TBB) yra C++ programavimo kalbos biblioteka, orientuota į kodo išlygiagretinimą kelių branduolių procesoriams. Kaip ir Cilk Plus, ši biblioteka priklauso kompanijai Intel. Tačiau TBB nėra kalbos praplėtimas ar atskira kalba. Ji yra tiesiog biblioteka. Dėl to TBB galima naudoti programose, kurios nebūtinai yra kompiliuojamos su Intel C++ kompiliatoriumi. Biblioteka taip pat veikia visose pagrindinėse operacinėse sistemose.

Žvelgiant iš programos vykdymo pusės, TBB remiasi tais pačiais programavimo šablonais, kaip ir Cilk Plus. Jos abi remiasi užduotimis paremtu lygiagretinimu. Jos abi naudoja darbo perskirstymo strategiją užduotims vykdyti [AAR+10]. Tačiau TBB turi daug gausesnį įrankių kiekį lygiagretumui įgyvendinti. Intel šiuos įrankius skirsto į:

1. Bendruosius lygiagrečiuosius algoritmus (angl. generic parallel algorithms)
2. Lygiagrečias duomenų struktūras (angl. concurrent containers)

3. Priklausomybių ir duomenų vykdymo grafų palaikymą (angl. support for dependency and data flow graphs)
4. Gijos privačios atminties saugyklas (angl. thread local storage)
5. Užduočių planuotoją (angl. task scheduler)
6. Sinchronizacijos primityvus (angl. synchronization primitives)
7. Atminties išskyrėją (angl. memory allocator)
8. Gijas (angl. threads)
9. Lygiagrečiuose skaičiavimuose saugius laikmačius (angl. thread–safe timers)

Gijos privačios atminties saugyklas sudaro dvi klasės. *Combinable* suteikia saugyklą kiekvienos gijos skaičiavimo rezultatams laikyti. Vėliau skaičiavimus gali sujungti į vieną rezultatą. *Enumerable_thread_specific* yra saugykla, kurioje kiekviena gija gauna vieną elementą. Per šią saugyklą galima iteruoti įprastais C++ metodais.

TBB turi sinchronizacijos primityvus atominėms operacijoms bei abipusei atskirčiai (angl. mutual exclusion). Atominės operacijos pasižymi tuo, jog jeigu viena gija vykdo atominę operaciją, kitos gijos operaciją mato vieną, nedalią komandą. Atominės operacijos yra greitesnės už tas pačias operacijas su užraktais, tačiau TBB palaiko limituotą rinkinį šių operacijų – skaitymo, rašymo, priskyrimo, sudėties ir priskyrimo su palyginimu.

Kai atominių operacijų nepakanka, naudojami muteksai (angl. mutex) ir užraktai. Muteksai bibliotekoje yra kelių rūšių, tačiau jie visi yra abstrakcijos virš p-gijų muteksų (Linux aplinkose) ir CRITICAL_SECTION sekcijų (Windows aplinkose).

Būna atvejų, kai reikia apskaičiuoti, kiek laiko programa jau yra vykdoma. Tai ne visada yra paprastas uždavinys. Operacinės sistemos pateikiami metodai gali grąžinti netikslius rezultatus skirtingose gijose, nes pačios mašinos gijų laikrodžiai gali būti nesusisinchronizavę. Tokiai problemai spręsti TBB yra įgyvendintas lygiagrečiuose skaičiavimuose saugus laikmatis. Jis sugeba grąžinti tikslų rezultatą, kiek laiko praėjo nuo programos vykdymo pradžios. Tada gautą laiką galima palyginti su kitu laiku ir paskaičiuoti jų skirtumą. Tai gali būti pravartu skaičiuojant tam tikro kodo bloko vykdymo trukmę.

3.3.1. Bendrieji lygiagretūs algoritmai

Intel TBB pasižymi tuo, jog turi gausų kiekį konstruktų, skirtų įvairiems lygiagretaus programavimo šablonams ir algoritmams padengti.

Be parallel_for, parallel_do ir parallel_while, kurie yra atitinkamai for, do ir while ciklų lygiagretiems skaičiavimams pritaikytos versijos, bibliotekoje taip pat pridėti parallel_reduce, parallel_scan, parallel_pipeline ir parallel_sort konstruktai.

Konstruktas parallel_reduce yra skirtas redukcijos (angl reduction) programavimo šablonui. Šablonas sujungia kiekvieną kolekcijos elementą į vieną elementą naudojant tam tikrą sujungimo funkciją [MRR12]. Kaip vieną iš dažnai pasitaikančių šablono naudojimo pavyzdžių galima paminėti visų rinkinio elementų sudėtis.

Konstruktas parallel_scan yra skirtas skenavimo (angl. scan) lygiagretaus programavimo šablonui. Šablonas veikia panašiai kaip ir redukcija. Jo galutinis rezultatas irgi yra vienas elementas, kuris apskaičiuojamas sujungiant visus kolekcijos elementus. Skirtumas tik tas, jog šiuo atveju kartu su galutiniu elementu kartu reikalaujama apskaičiuoti ir prieš tai buvusią kolekcijos redukciją. Šablonas dažnai pritaikomas kompiliatoriuose išlygiagretinti rekursyvioms funkcijoms [KH16].

Konstruktas parallel_pipeline yra skirtas spręsti problemai, kai duomenys turi eiti per eilę filtrų. Vieni filtro rezultatai yra sekančio filtro įeities parametrai. Priklausomai nuo filtro, jis gali dirbti su tam tikrais bendrais kintamaisiais, priklausyti nuo kitų išorinių veiksnių ar panašiai. Tada filtras gali būti vykdomas tik nuosekliai. Tačiau, jeigu filtras nėra taip sulimituotas, tai galima nurodyti per parametrą ir filtras gali būti išlygiagretintas.

Konstruktas parallel_sort yra skirtas surikiuoti sąrašo elementams pagal duotą palyginimo operatorių. Kaip ir kiti algoritmai, jis irgi bando išlygiagretinti rikiavimą. Rikiavimas yra nestabilus (angl. unstable sort) – elementai su vienodais lyginimo raktais gali neišsaugoti tokio paties eiliškumo, kokį turėjo prieš pradedant rikiuoti. Parallel_sort privalumas virš įprasto rikiavimo iš std bibliotekos yra tas, jog įprasto rikiavimo sudėtingumas yra $O(n \log n)$, kur n – sąrašo elementų kiekis, o parallel_sort sudėtingumas artėja prie $O(n)$ priklausomai nuo mašinos procesorių branduolių kiekio [Rei07].

3.3.2. Lygiagrečios duomenų struktūros

Vienas iš trūkumų dirbant su STL bibliotekos konteinerio tipo duomenų struktūromis yra tai, kad jos nėra saugios lygiagrečiame kontekste. Bandymai keisti jų laikomus duomenis lygiagrečiai gali baigtis sugadintais duomenimis konteineryje. Dėl to duomenų keitimui paprastai reikia naudoti užraktus, kad tik viena gija keistų jų duomenis vienu metu. Tai prideda papildomo kodo, kuris gali turėti klaidų bei blokuoja gijas laukiant, kol kitos gijos baigs darbą.

TBB biblioteka turi nemažai konteinerių, kurie atitinka konteinerius iš STL bibliotekos, tačiau yra optimizuoti lygiagrečiam veikimui. Didžioji konteinerių metodų dalis ir naudojimas yra panašūs ar vienodi, kaip ir STL konteineriai. Likusi dalis gali skirtis, nes pagal struktūrą nėra tinkami lygiagrečioje aplinkoje.

Vis dėlto, TBB bibliotekos konteinerių implementacijos gali turėti užraktų. Tačiau užraktai yra optimizuoti taip, kad skirtingi užraktai rakintų skirtingas konteinerio dalis, tada daugiau negu viena gija gali dirbti su konteineriu ir nesiblokuoti, jeigu vienu metu jos rakina skirtingas dalis.

Deja, lygiagrečiam naudojimui pritaikyti konteineriai įgauna papildomai vykdymo kodo. Todėl jų metodai gali būti lėtesni už konteinerius iš STL bibliotekos.

3.3.3. Atminties išskyrėjas

Atminties išskyrimas yra viena iš pagrindinių programavimo užduočių. Taip pat ją įgyvendinti lygiagrečiuose skaičiavimuose yra vienas iš sudėtingesnių darbų. Taip yra dėl dviejų priežasčių.

Pirmoji problema su nuosekliems skaičiavimams pritaikytais atminties išskyrėjais yra ta, jog atminties išskyrėjas naudojasi globaliu užraktu kiekvienam išskyrimui iš vieno globalaus atminties heap‘o. Dėl to kiekviena gija, kurioje išskiriama atmintis, turi laukti savo eilės. Dėl to programose su dažnais atminties išskyrimais gali būti situacijų, kuomet jos vis labiau lėtėja didėjant branduolių skaičiui.

Kita problema lygiagrečiuose skaičiavimuose yra netikras bendrinimas (angl. false sharing). Jis pasirodo, kai keletas gijų naudojasi atminties vietomis, kurios yra greta viena kitos. Procesoriaus branduoliai atmintį išsiima ir laiko blokais, vadinamais podėlio linijomis (angl. cache lines). Jeigu skirtingų gijų atmintis patenka į tą pačią podėlio liniją, po vienos gijos atminties keitimo, branduolys turi išsaugoti ir kitos gijos duomenis iš podėlio į globalią atmintį. Dėl to optimaliu scenarijumi podėlio linijoje turėtų būti tik vienai gijai priklausanti atmintis.

TBB bibliotekoje yra du atminties išskyrėjai šioms problemoms spręsti – *scalable_allocator* ir *cache_aligned_allocator*. Abu išskyrėjai remiasi algoritmais, kurie globalią atmintį paskirsto į dalis ir tas dalis išskiria skirtingoms gijoms. Cache_aligned_allocator išskyrėjas papildomai apsaugo ir nuo netikro bendrinimo. Jis tai įgyvendina atmintį išskirdamas tokiais atminties blokais, kurių dydis sutampa su podėlio eilutės dydžiu.

Verta paminėti, kad TBB atminties išskyrėjų funkcijos turi papildomą kainą. Kadangi jų išskyrimai remiasi globalios atminties vietos padalijimu ir didesnės atminties bloko išskyrimu, negu iš tikro reikia duomenims laikyti, atmintis nebebūna taip efektyviai išnaudojama, kaip yra įprastuose išskyrėjuose.

3.4. Google Go

Google Go (dar vadinama Golang) yra programavimo kalba. Ji buvo pradėta kurti kompanijoje Google 2007-ais metais ir išleista 2012-ais metais. Kurdami Go jos kūrėjai siekė kelių tikslų:

- Go kalba parašytą programą turi būti galimybė plėsti, joje esant daug priklausomybių ir prie programos dirbant didelėms komandoms
- Kalba turi būti panaši į C, nes Google programuotojai yra labiausiai pratę prie C kalba paremtų kalbų (C, C++, Java)
- Kalba turi būti moderni. Trūkumas, kurį bandoma išspręsti, yra tai, kad C, C++ ir iš dalies Java yra gana senos kalbos, kurios buvo sukurtos prieš kelių branduolių procesorių, interneto ir internetinių programų išpopuliarėjimą. Dėl to šios kalbos labiau buvo pritaikytos prie atsiradusių poreikių, o ne kurtos su mintimi apie juos [Pik12]

Siekiant šių tikslų buvo sukurta procedūrinė programavimo kalba, kurios sintaksė primena C kalbos sintaksę. Ji taip pat yra statinė, kompiliuojama, naudoja tik 25 raktinius kalbos žodžius, turi į kalbą integruotus šiukšlių surinkimą (angl. garbage collection), refleksiją, interface'ų tipus ir lygiagretumo valdymo įrankius.

3.4.1. Go paprogramės ir kanalai

Go požiūris į lygiagretumą geriausiai apibūdinamas dažnai šios kalbos bendruomenėje cituojama vieno iš kalbos kūrėjų Rob Pike posakiu, kad programos lygiagrečiai dirbantys komponentai turėtų "ne bendrauti dalindamiesi atmintimi, bet dalintis atmintimi bendraudami" [Pik15]. Bendravimas dalinantis atmintimi kalba apie atvejus, kuomet kelios gijos turi prieigą prie vienos atminties vietos (pavyzdžiui, bendro kintamojo), kurį pasiekia per užraktą. Įprastai Go kalboje programos kūrėjui naudoti užraktų nereikia (tačiau biblioteka jiems yra). Vietoj to lygiagretumą kalboje rekomenduojama įgyvendinti dviem kalbos konstruktais: go paprogramėmis (angl. goroutines) ir kanalais (angl. channels).

Go paprogramė yra funkcija, kuri gali būti vykdoma lygiagrečiai su kitomis funkcijomis. Jai sukurti naudojamas kalbos raktinis žodis *go*, po kurio eina funkcijos iškvietimas. Pavyzdžiui, *go f(0)*. Jeigu f(0) būtų kviečiama be raktinio žodžio go, funkcija būtų kviečiama ir iš karto vykdoma. Su raktiniu žodžiu go sukuriama go paprogramė, kuri funkciją f(0) gali pradėti vykdyti iš karto kitoje gijoje arba laukti, kol esama gija užsiblokuos kitoje go paprogramėje ir tada vykdyti funkciją. Esminis faktas yra tai, kad einamoji go paprogramė nelaukia naujos paprogramės darbo vykdymo pabaigos ir sukūrusi naują go paprogramę – eina prie sekančios operacijos [Cox17].

Kanalai yra struktūra, skirta go paprogramėms dalintis duomenimis. Kanalai leidžia vienai go paprogramei persiųsti duomenis į kitą go paprogramę. Kanalas turi dvi pagrindines operacijas – siuntimo (angl. send) ir gavimo (angl. receive), kurios ir yra naudojamos bendravimui kanalu.

Siuntimo operacija nusiunčia reikšmę į kanalą, o gavimo operacija ištraukia reikšmę iš kanalo [DK15]. Kanalas sukuriamas kviečiant funkciją make:

```
ch := make(chan type[, capacity])
```

kur *type* yra tipas duomenų, kurie bus laikomi kanale, capacity yra kanalo talpa, nusakanti, kiek elementų gali būti laikoma kanale, o ch yra rodyklė į kanalą, su kuria go paprogramės gali pasiekti kanalą.

Jeigu talpa yra nenurodyta, tada kanalas yra neturintis buferio (angl. unbuffered). Siuntimo operacija į tokį kanalą blokuoja siuntusią go paprogramę iki tol, kol kita go paprogramė iškvies gavimo operaciją. Po jos reikšmė yra perduota ir abi go paprogramės gali tęsti darbą. Jeigu gavimo operacija buvo iškviesta pirmiau, duomenų laukianti go paprogramė užsiblokuotų iki kol kita go paprogramė iškviestų siuntimo operaciją tame kanale.

Jeigu talpa nurodyta, tada kanalas turi buferį (angl. buffered), į kurį telpa nurodytas kiekis elementų. Nauji elementai siunčiami į kanalo galą ir gaunami iš kanalo pradžios, kaip eilės duomenų struktūroje. Į tokį kanalą siunčianti go paprogramė blokuojasi iki kol elementas įdedamas kanalą. Elementui prisidėjus į kanalą, siuntusi go paprogramė atsiblokuoja. Jeigu kanalas jau yra pilnai pripildytas elementais, tada siunčianti go paprogramė blokuojasi iki kol kita paprogramė iškviečia gavimo operaciją iš to kanalo.

Šios operacijos tarp kanalų priverčia siunčiančias ir gaunančias go paprogrames susisinchronizuoti. Jie gali būti naudojami pranešti apie tam tikrą įvykį (pvz, kanalas gali priimti vieną loginio tipo elementą) arba dalintis bendrais duomenimis. Kanalai taip pat gali būti uždaromi su funkcija *close(channel)*. Ši funkcija naudinga, kai norima, kad go paprogramės nebelauktų kanalo siuntimo ar gavimo operacijų.

3.4.2. Žemo lygio atminties sinchronizavimas

Atvejams, kai dėl tam tikrų priežasčių kanalai nėra tinkamas sprendimas dalytis ir sinchronizuotis tarp go paprogramių, galima naudoti žemo lygio atminties sinchronizavimo įrankius iš *sync* paketo.

Paketą sudaro 8 tipai su tam tikromis funkcijomis, skirtomis dirbti su atitinkamais tipais.

Tipas *WaitGroup* yra skirtas laukti, kol kiekviena go paprogramė rinkinyje baigs darbą. Jo veikimas remiasi tuo, kad pagrindinė go paprogramė kviečia funckiją *Add* su go paprogramių skaičiumi ir vėliau gali kviesti funkciją *Wait*. Ši funkcija blokuoja pagrindinę go paprogramę iki kol visos pridėtos go paprogramės iškviečia funkciją *Done*, kuri pasako, jog go paprogramė baigė laukiamą darbą.

Tipas *Once* naudojamas, kai norima, jog tam tikras kodo blokas būtų vykdomas tik vieną kartą. Jis gali būti panaudojamas, kai kelios skirtingos go paprogramės gali kviesti tą patį kodo

bloką, bet norima, kad jį įvykdytų tik pirmoji kviečiančioji go paprogramė, o kitos kodo bloko vykdymą praleistų.

Tipas *Locker* realizuoja užraktą, kuris turi dvi funkcijas – rakinimo ir atrakinimo.

Duomenų struktūra *Pool* yra skirta laikyti laikiniems objektams. Bet kuris elementas šioje struktūroje gali būti bet kada ištrintas be jokio pranešimo. *Pool* naudojamas išlaikyti tuo metu nereikalingus objektus, kurių gali reikėti vėliau. Taip sutaupoma laiko atminties išskyrimui ir sumažinamas šiukšlių surinkėjo darbas.

Darbui su muteksais yra realizuoti *Mutex* ir *RWMutex* tipai. Juos galima užrakinti ir atrakinti su atitinkamomis funkcijomis. *RWMutex* papildomai turi atskiras operacijas skaitymo bei rašymo rakinimui ir atrakinimui.

Tipas *Cond* yra skirtas go paprogramėms laukti arba pranešti, kad įvyko tam tikras įvykis. Tai įgyvendinama su funkcijomis, skirtomis atblokuoti visoms arba vienai laukiančiai go paprogramei, taip pat su funkcija, kuri užblokuoja go paprogramę iki kol ji bus atblokuota per to paties *Cond* tipo atblokavimo funkciją.

Paskutinis tipas *sync* pakete yra vadinamas *Map*. Jis yra paprasto Go kalbos duomenų tipo *map* atmaina, kuri yra optimizuota, kai elementai su vienodu raktu yra įdedami tik kartą ir skaitomi daug kartų (pavyzdžiui, implementuoti podėliams) arba kai kelios go paprogramės skaito, rašo ar perrašo elementus su skirtingais raktais. Tokiais atvejais ši duomenų struktūra veikia greičiau, negu standartinė Go kalbos *map* struktūra, rakinama su muteksais.

4. Matavimų specifikacija

4.1. Aplinka

Darbe nagrinėjamos užduotys yra matuojamos stacionariame kompiuteryje.

Kompiuteris naudoja Intel kompanijos procesorių ir turi tokius parametrus:

- MSI B360 GAMING PLUS pagrindinė plokštė
- Intel® Core™ i7 9700 Coffee Lake architektūros procesorius, turintis 8 branduolius su 3 GHz taktiniu dažniu
- 16 GB DDR4 2400MHz CL 12 1.5V operatyvinė atmintis
- Samsung SSD 850 PRO 256GB kietasis diskas
- Microsoft Windows 10 Pro (10.0.18362 Build 18362x64 versija)

Visi matuojami uždaviniai buvo implementuoti šio darbo autoriaus ir visas išeities kodas yra patalpintas į viešą kodo repozitoriją, kuri yra pasiekiama adresu: https://bitbucket.org/fgauris/mif-parallel-programming/src.

4.2. Kompiliatoriai ir optimizacijos

Visų su C++ kalba rašytų modelių programos yra sukompiliuotos su Intel C++ kompiliatoriaus 19.0 versija. Google Go modelio programos yra sukompiliuotos su Google Go kompiliatoriaus 1.14 versija. Abi šios versijos pasirinktos todėl, kad darbo rašymo metu jos buvo naujausios išleistos stabilios versijos. Intel C++ kompiliatorius pasirinktas dėl to, jog jis yra vienas populiariausių kompiliatorių rinkoje ir du iš nagrinėjamų išlygiagretinimo modelių – Cilk Plus ir TBB priklauso būtent Intel kompanijai, todėl jie yra pirmos klasės įrankiai šiame kompiliatoriuje.

Abu naudoti kompiliatoriai palaiko kompiliatoriaus optimizacijas. Prieš matuojant galutinius rezultatus, kiekvienas uždavinys buvo patikrintas, kaip greitai jis veikia su kompiliatoriaus optimizacijomis ir be jų.

Naudojant kompiliatoriaus optimizacijas, visų uždavinių vykdymo laikas būdavo trumpesnis, lyginant su išjungtomis optimizacijomis, tačiau santykiniai rezultatai dėl to nekito. Tai yra, jeigu vienas modelis būdavo efektyviausias išjungus optimizacijas, jis būdavo efektyviausias ir su optimizacijomis. Vienintelė išimtis buvo matricų daugyba (plačiau apie tai 4.1.2 skyriuje), kurios OpenMP matricų daugybos optimizacija Qopt-matmul įtakojo ir santykines reikšmes. Dėl to matricų daugybos uždavinio rezultatai pateikiami su išjungta Qopt-matmul optimizacija.

Kalbant apie kitas optimizacijas, visuose uždaviniuose naudojamas O2 optimizacijos lygis (orientuotas į programos greitį), įjungtas OpenMP palaikymas (/Qopenmp) ir išjungtas automatinis ciklų išlygiagretinimas (/Qparallel).

4.3. Rezultatų skaičiavimas

Kiekvienas uždavinys buvo matuojamas su trimis skirtingais duomenų rinkiniais. Duomenys parinkti taip, kad programos vidutinės trukmės svyruotų tarp kelių milisekundžių, kelių šimtų milisekundžių ir daugiau nei kelių sekundžių. Tokiu būdu buvo galima pamatyti, kaip nagrinėjami išlygiagretinimo modeliai elgiasi su skirtingo dydžio ir trukmės užduotimis.

Kiekvienas uždavinys buvo matuotas ant skirtingo kiekio procesoriaus branduolių. Matuota pradedant vienu branduoliu ir baigiant 8 branduoliais.

Pirmiausiai buvo matuojama uždavinio nuosekli versija. Ji taip pat buvo matuojama ant skirtingų kiekių branduolių, bet kaip ir buvo galima tikėtis, nuosekliosios uždavinių versijos veikė beveik vienodais greičiais, su kelių procentų nuokrypiu, nepriklausomai nuo turimų branduolių kiekių. Todėl tolimesniems skaičiavimams bus naudojamos nuoseklių versijų trukmės ant vieno branduolio.

Toliau buvo einama prie išlygiagretintų uždavinio versijų, kurioms buvo išmatuotos kiekvienos branduolių kiekio ir duomenų rinkinio poros vykdymo trukmės, kurios parodo kiek laiko reikėjo programai baigti darbą turint n branduolių. Visų apskaičiuotų trukmių grafikai yra pateikiami šio darbo prieduose.

Išmatavus vykdymo trukmes nuoseklioje ir išlygiagretintose versijose, buvo skaičiuojami kiekvienos branduolių kiekio ir duomenų rinkinio poros spartinimai. Jie parodo programos pagreitėjimą keičiantis branduolių kiekiui. N branduolių spartinimas (žym. S_n) apskaičiuojamas pagal formulę [HP12]:

$$S_n = \frac{T_{nuoseklus}}{T_{lygiagretus}(n)}$$

kur $T_{nuoseklus}$ – nuoseklaus sprendimo trukmė, $T_{lygiagretus}$ (n) – lygiagretaus užduoties sprendimo su n branduolių trukmė.

Pateikus spartinimus, kiekvienam duomenų rinkiniui ir išlygiagretinimo metodui yra pateikiami maksimalūs spartinimai lyginant su nuosekliu sprendimu.

Maksimalus spartinimas apskaičiuojamas pagal formulę:

$$S_{vid} = Max(S_1, S_2, \dots, S_n)$$

kur n – branduolių kiekis, S_n – uždavinio, spręsto su n branduolių, spartinimas.

Apskaičiavus spartinimus, pateikiama apibendrinta visų uždavinių rezultatų diagrama.

5. Uždaviniai

2006-aisiais metais Berklio Kalifornijos universiteto mokslininkai išleido straipsnį „The Landscape of Parallel Computing Research: A View from Berkeley“ [ABC+06], kuriame jie nagrinėjo procesorius su keliais branduoliais. Straipsnyje buvo išskirta 13 uždavinių kategorijų (straipsnyje vadinamų nykštukais), kurie reprezentuoja skirtingas lygiagretaus programavimo sritis. Iš šių kategorijų išryškėja uždaviniai ar šablonai, kurie gali būti panaudojami lygiagretaus programavimo modeliams įvertinti ir palyginti.

Šiame skyriuje yra nagrinėjamas tam tikras poaibis uždavinių, kurie priklauso skirtingoms uždavinių kategorijoms. Poaibis buvo parinktas taip, kad jo uždaviniai pasižymėtų kuo dažniau sutinkamomis ir aktualiomis programavimo struktūromis bei būtų plačiai žinomi ir turėtų trumpus bei aiškius reikalavimus.

5.1. Matricų daugyba

Matrica yra stačiakampė elementų lentelė. Matricos yra naudojamos spręsti tiesinių lygčių sistemoms, taip pat atliekant tiesines transformacijas. Daugelis modeliavimo/simuliacijos uždavinių, aprašomų diferencialinėmis lygtimis ir sprendžiamų skaitiniais metodais, operuoja matricomis.

Viena iš matricų operacijų yra matricų daugyba. Šią operaciją galima priskirti tiek prie tankios tiesinės algebros (angl. dense linear algebra) kategorijos, tiek prie retos tiesinės algebros (sparse linear algebra) kategorijos pagal tai, kokio tipo matricos dauginamos.

Kai daugybos apskaičiavimo algoritmas dirba su retomis (angl. compressed) formato matricomis, tada jis priskiriamas retos tiesinės algebros kategorijai. Retomis arba suspaustomis matricomis vadinamos tos, kurios turi didžiąją dalį nulinių elementų. Kai algoritmas dirba su tankiomis matricomis – jis priklauso tankios tiesinės algebros kategorijai. Kadangi abiem atvejais algoritmai būtų gana panašūs, darbe bus nagrinėjamos tankios matricos.

Bendruoju atveju dauginti galima suderintas matricas, t. y. galima dauginti matricas A ir B, jeigu B matricos eilučių skaičius sutampa su A matricos stulpelių skaičiumi. Jų sandauga yra matrica [AB], kurios eilučių skaičius sutampa su A eilučių skaičiumi ir stulpelių skaičius sutampa su B stulpelių skaičiumi. O kiekvienas matricos [AB] elementas C_{ij} yra apskaičiuojamas pagal formulę:

$$[\mathbf{AB}]_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \cdots + a_{i,n}b_{n,j} = \sum_{r=1}^{n} a_{i,r}b_{r,j},$$

5.1.1. Realizacijos

Klasikinis nuoseklusis matricų daugybos algoritmo pseudo kodas, kuriuo remiasi visų modelių implementacija, yra toks:

```
For i from 1 to n:
     For j from 1 to p
          Sum = 0
          For k from 1 to m
               Sum = Sum + Ajk * Bkj
          Cij = Sum
```

Čia n yra A matricos eilučių skaičius, p – A matricos stulpelių skaičius ir B matricos eilučių skaičius, m – B matricos stulpelių skaičius.

Kiekvienos 1..n ir 1..p ciklų iteracijos metu tam tikras rezultatų matricos C gauna apskaičiuotą reikšmę. Kadangi iteracijos skaičiuoja reikšmes skirtingiems laukeliams, abiejų ciklų iteracijos gali būti vykdomos lygiagrečiai ir nepriklausomai.

OpenMP realizacijoje galima naudoti kompiliatoriaus direktyvą, kuri nurodo, kad galima išlygiagretinti for ciklų iteracijas per skirtingus branduolius. Su šia direktyva realizacija yra tokia:

```
#pragma omp parallel for
for(int i = 0; i < m; ++i)
     #pragma omp parallel for
     for(int j = 0; j < p; ++j)
          Cij laukelio apskaičiavimas
```

Cilk Plus irgi turi specialiai for ciklų išlygiagretinimui egzistuojantį raktinį žodį cilk_for. Su juo daugybos realizacija atrodytų štai taip:

```
cilk_for(int i = 0; i < m; ++i)
     cilk_for(int j = 0; j < p; ++j)
          Cij laukelio apskaičiavimas
```

Cilk_for raktinis žodis nurodo, kad ciklų iteracijas reikia paskirstyti blokais per procesoriaus branduolius.

Intel TBB irgi yra ne išimtis, kai kalbama apie įrankius for ciklų išlygiagretinimui. Lygiagretiems for ciklams TBB turi specialų konstruktą parallel_for, kuris kaip ir kitų išlygiagretinimo modelių atveju išdalija iteracijas ar jų blokus vykdyti skirtingiems procesoriaus branduoliams. Intel TBB realizacija yra įgyvendinta tokiu būdu:

```
parallel_for(size_t(0), size_t(m), [&](int i){
     parallel_for (size_t(0), size_t(p), [&](int j){
          Cij laukelio apskaičiavimas
     });
});
```

Go kalbos atveju specialaus konstrukto būtent lygiagrečiam for ciklui nėra. Vietoje to yra naudojamas go paprogramės ir WaitGroup tipas, todėl realizacija yra kiek ilgesnė:

```
var wg sync.WaitGroup
wg.Add(m*p)
for i := 0; i < m; i++ {
     go func(i int) {
          for j := 0; j < p; j++ {
               go func(i, j int){
                    Cij laukelio apskaičiavimas
                    wg.Done()
               }(i,j)
          }
     }(i)
}
```

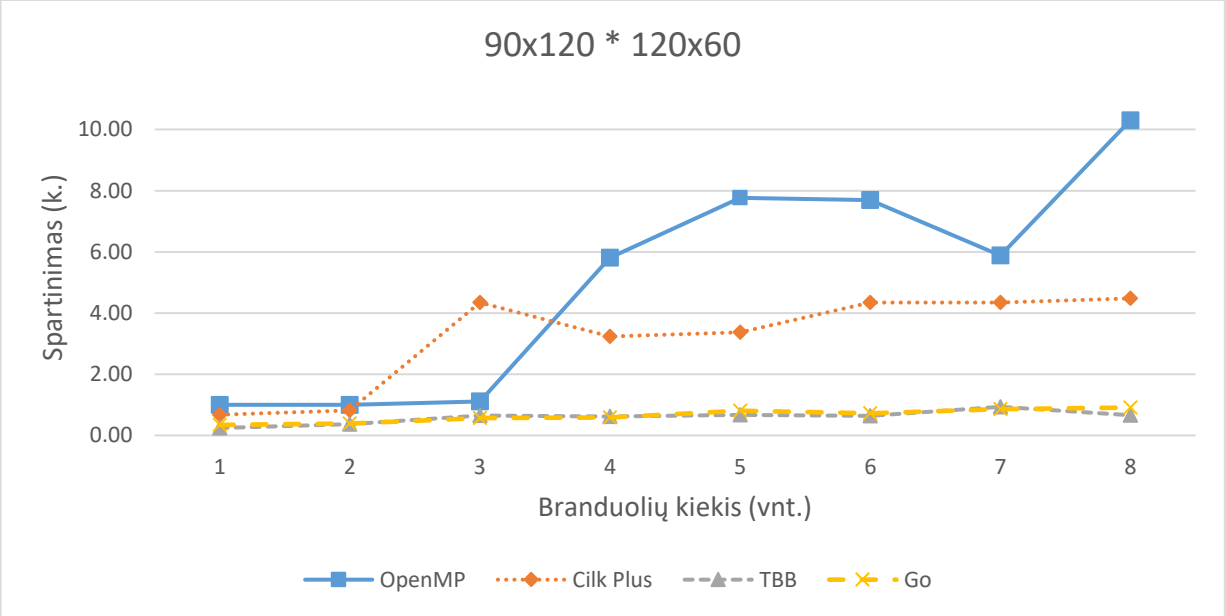
```
wg.Wait()
```

Ši realizacija kiekvienoje 0..m-1 iteracijoje sukuria po m naujų go paprogramių, kurios kiekviena sukuria po dar p naujų go paprogramių. Taip gaunama m*p go paprogramių, kurios gali lygiagrečiai skaičiuoti C matricos laukelių reikšmes. Tipo WaitGroup kintamasis wg yra naudojamas visų go paprogramių sinchronizacijai. Pradžioje su metodu wg.Add nurodoma, kad laukimo grupė turės laukti m*p go paprogramių, o su metodu Wait nurodoma, kad negalima toliau vykdyti sekančios eilutės, kol visi laukimo grupės nariai nebaigė darbo.

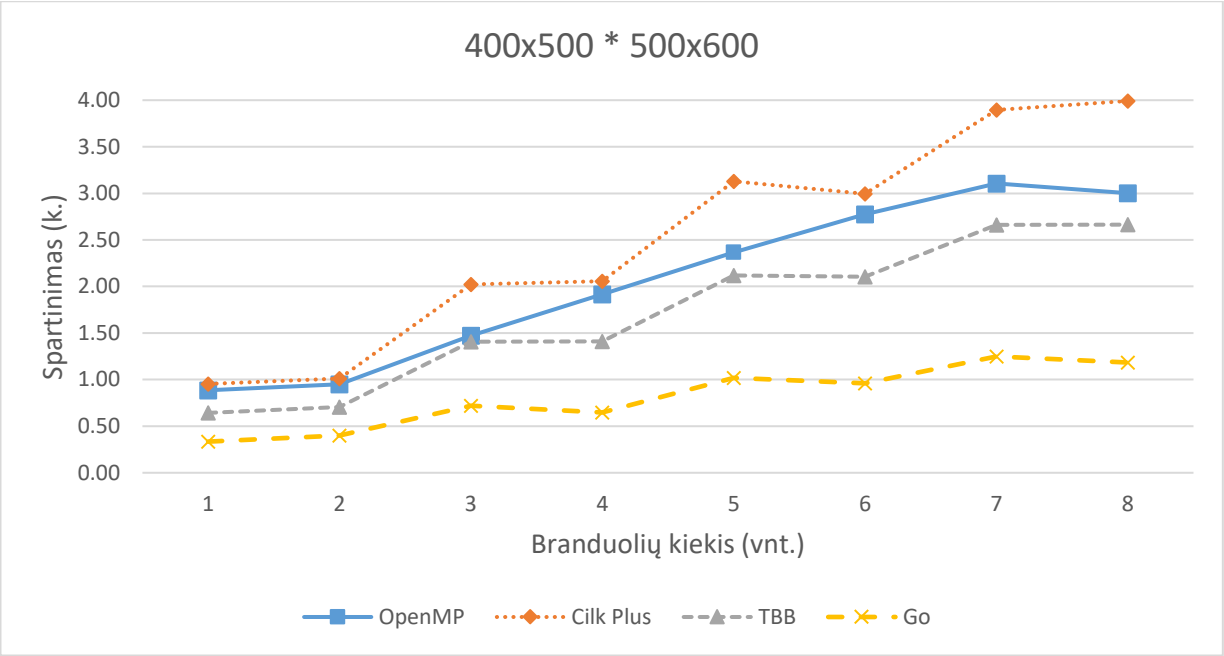
5.1.2. Rezultatai

Matricų daugybos matavimai buvo atliekami su trimis skirtingų dydžių matricų daugybomis. Pirmuoju atveju buvo dauginamos mažos 90x120 ir 120x60 dydžio matricos. Antruoju atveju buvo dauginamos vidutinės 400x500 ir 500x600 dydžio matricos. Trečiuoju atveju buvo dauginamos didesnės 900x1100 ir 1100x1000 dydžio matricos. Kiekviena daugyba buvo atliekama po lygiai 100 kartų ir skaičiuojama, kiek vidutiniškai laiko užtruko viena daugyba. Prieduose 1, 2 ir 3 pateikiamos vidutinės daugybų trukmės milisekundėmis.

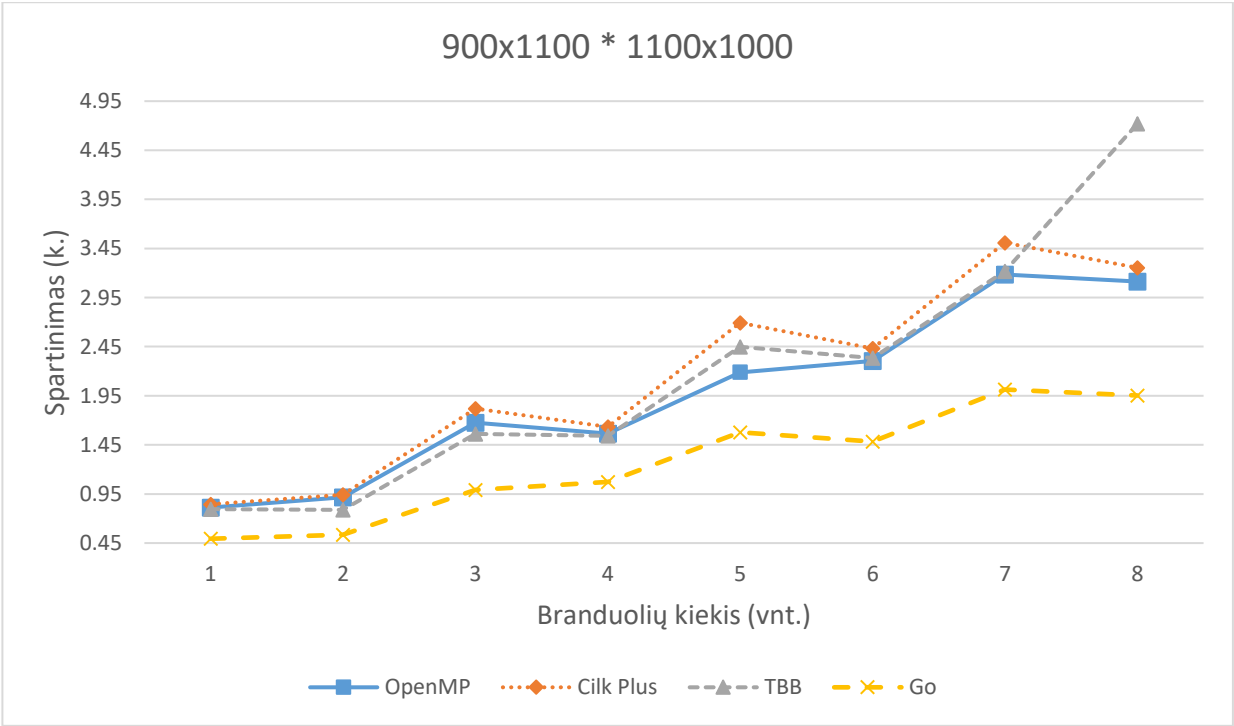
Žemiau pateikiamos spartinimų diagramos:



5 pav. Matricų (90x120 * 120x60) daugybos spartinimai lyginant su nuoseklia versija



6 pav. Matricų (400x500 * 500x600) daugybos spartinimai lyginant su nuoseklia versija



7 pav. Matricų (900x1100 * 1100x1000) daugybos spartinimai lyginant su nuoseklia versija

	OpenMP	**Cilk Plus**	**TBB**	**Go**
90x120 * 120x60	**10.30**	4.48	0.93	0.91
400x500 * 500x600	3.11	**3.99**	2.66	1.25
900x1100*1100x1000	3.18	3.51	**4.72**	2.01

1 lentelė. Maksimalūs matricų daugybos spartinimai lyginant su nuoseklia versija

Žvelgiant į spartinimus su vienu ir dviem branduoliais, galima pastebėti, jog spartinimas buvo žemiau vieneto, o tai reiškia, kad nuosekliai dauginti matricas buvo greičiau. Didėjant branduolių kiekiui, su mažomis matricomis, TBB ir Go versijos vis tiek skaičiavo lėčiau negu nuosekli versija. Vidutiniu matricu atveju, didėjant branduolių kiekiui, visi išlygiagretinimo modeliai pradėjo lenkti nuosekliąją versiją, o su didžiausiomis matricomis, jau ties trimis branduoliais, visi modeliai buvo spartesni.

Iš visų išlygiagretinimo modelių lėčiausias buvo Go beveik visais atvejais. To buvo galima tikėtis, kadangi jis neturėjo į pačią kalbą integruotų for ciklo išlygiagretinimo įrankių, priešingai nei kiti modeliai. Dauginant vieno branduolio aplinkoje, Go versijos sprendimai trukdavo net kelis kartus ilgiau už kitus modelius. Tačiau įdomu tai, kad žvelgiant į žvelgiant į didžiausių matricų trukmių diagramą (3 pried.), galima matyti, jog daugėjant branduolių kiekiui, su didžiausiomis matricomis, Go pradėjo mažinti atskyrą iki kitų modelių.

Vis dėlto, lygiant su nuoseklia versija, didžiausių matricų atveju, maksimalus Go spartinimas yra tik 2.01 karto. O vos 0.91 ir 1.25 karto spartinimai mažesnėms matricoms rodo, kad ši versija iš tiesų dažnai veikė lėčiau už nuoseklų sprendimą.

OpenMP versija, žvelgiant į trukmių ir spartinimo diagramas, visais nagrinėtais atvejais nusileisdavo tik Cilk Plus versijai arba būdavo greitesnė už visus kitus modelius. Su mažiausiomis matricomis OpenMP maksumalus spartinimas viršijo net 10 kartų, o su vidutinėmis ir didžiausiomis – apie 3 kartus.

Taip pat verta paminėti OpenMP optimizaciją, kur pažymėjus ciklą su parallel for direktyva, kompiliatorius sugeba nustatyti, ar pažymėtas išlygiagretinti ciklas yra matricų daugybos ciklas. Jeigu nustatoma, kad ciklai yra skirti matricų daugybai, tada parašytas kodas yra pakeičiamas kvietimais į Intel matricų daugybos biblioteką matmul [IC20]. Ši optimizacija galioja tik OpenMP modeliui. Neišjungus optimizacijos, pavyzdžiui, nagrinėtas vidutines matricas OpenMP versija sudaugindavo vidutiniškai nuo trijų iki penkių kartų greičiau. Tai būtų daug kartų pagerinę OpenMP matricų daugybos rezultatus, tačiau šio skyriaus tikslas yra palyginti pasirinktus išlygiagretinimo modelius iteruojant per for ciklus. Matricų daugybos yra uždavinys, kuris padeda iliustruoti ilgų ciklų aktualumą realiame pasaulyje. Dėl to rezultatuose pateikiami OpenMP įverčiai be šios optimizacijos.

Abu Intel kompanijos modeliai – Cilk Plus ir TBB pateikė panašius spartinimus su didžiausiomis matricomis. Su vidutinėmis matricomis Cilk Plus buvo visais atvejais greitesnis.

Žiūrint į maksimalius spartinimus, maksimalus TBB spartinimas su mažiausiomis matricomis buvo tik 0.93 karto ir tai buvo prasčiausias spartinimas iš visų išmatuotų C++ modelių. Tačiau dauginant didžiausias matricas, maksimalus TBB spartinimas buvo 4.72 karto.

Tai reiškia, kad skaičiuojant didesnes matricas, TBB iš tikro didinant branduolių skaičių, spartėjo efektyviausiai iš visų modelių.

Iš šių matavimų galima daryti išvadą, kad esant situacijai, kai turimas ciklas ar vienas kitame esantys ciklai, kurių iteracijos nepriklauso viena nuo kitos, visi nagrinėti C++ kalbos išlygiagretinimo įrankiai pasižymi labai panašiais efektyvumais. Tai ypač ryškiai matoma su didesniu iteracijų kiekiu.

Su dideliu kiekiu iteracijų greičiausiai spartėjo Intel TBB biblioteka. Su mažu kiekiu iteracijų – OpenMP spartėjo ypač greitai. Taigi, su mažu iteracijų kiekiu ir mažu branduolių kiekiu nuoseklus sprendimas dažnu atveju eikvoja mažiausiai resursų ir yra pakankamai greitas. Didėjant branduolių kiekiui, Cilk Plus ir ypač OpenMP spartinimai yra žymiai geriau pastebimi.

5.2. Išilginio perteklumo tikrinimas

Viena iš informatikoje ir telekomunikacijose esančių problemų yra skaitmeninių duomenų siuntimas per nepatikimus komunikacijos kanalus. Nepatikimi komunikacijos kanalai yra tos duomenų perdavimo priemonės, kurios negali garantuoti, kad duomenys bus visada nusiųsti būtent tokie, kokie buvo išsiųsti.

Šią problemą sprendžia klaidų aptikimo ir taisymo algoritmai. Klaidų aptikimo algoritmai leidžia nustatyti, ar siuntimo metu duomenų srautas pakito, o tuo tarpu klaidų taisymo algoritmai tam tikrais atvejais leidžia iš atsiųstų su klaidomis duomenų atstatyti išsiųstus duomenis.

Yra daug skirtingų klaidų aptikimo algoritmų. Dažniausiai jie yra realizuojami su tam tikra duomenų maišos funkcija. Konkrečiau – algoritmas prideda fiksuoto ilgio maišos funkcijos rezultatą prie tam tikro siunčiamo duomenų kiekio ir tada visa tai siunčia. Duomenų gavėjas, gavęs duomenis ir žinodamas algoritmo realizaciją, gali nustatyti, kuri duomenų dalis buvo originali žinutė, o kuri – klaidų tikrinimo dalis. Tada jis pats perskaičiuoja maišos funkcijos rezultatą iš gautų originalių duomenų ir palyginęs su atsiųstųjų rezultatu, nustato, ar originaliuose duomenyse atsirado klaidų siuntimo metu.

Išilginio perteklumo tikrinimas (angl. longitudinal redundancy check), dar trumpinamas LRC, yra klaidų aptikimo algoritmas. Šis klaidų tikrinimas duomenų srautą padalija į vienodo dydžio blokus, įprastai – po 4 baitus ir prie jų prideda tikrinimo baitą. Tikrinimo baitas apskaičiuojamas atlikus XOR operaciją visiems bloko baitams. Šitoks algoritmas yra aprašytas ISO/IEC 7816 standarte, SLIP (angl. Serial line internet protocol) protokolui [FDC84].

Šis algoritmas yra vienas iš ciklinio klaidų tikrinimo (angl. cyclic redundancy check) versijų ir jos abi yra priskiriamos kombinacinės logikos (angl. combination logic) lygiagrečių uždavinių grupei [MMR+16].

5.2.1. Realizacijos

Sprendžiamas uždavinys, kuris bus matuojamas šiame skyriuje, yra toks: turimas tam tikro dydžio duomenų srautas, kurį reikia paversti į duomenų srautą su tikrinimo baitais.

Bendru atveju, kuriuo remsis visų modelių realizacijos, algoritmas apskaičiuoti srauto dalis ir tikrinimo baitą, realizuojamas taip:

```
For each part in byte stream
     For each block in part
          apskaičiuoti bloko lrc
          pridėti lrc prie bloko pabaigos
```

kur bloko lrc apskaičiuojamas pagal algoritmą:

```
byte lrc = 0
For each byte in block
     lrc = lrc XOR byte
```

Čia *part* yra duomenų srauto dalis, *block* – 4 baitų ilgio duomenų srauto dalies baitų blokas, *lrc* yra tikrinimo baitas.

Šio uždavinio sprendimui duomenų srautas bus padalintas į tiek lygių dalių, kiek mašina turi procesoriaus branduolių. Jeigu duomenų srautas yra 2000 baitų ilgio ir mašina turi 4 branduolius, tada srautas bus padalintas į keturias 500 baitų ilgio dalis.

Taigi, nuosekliuoju atveju algoritmas eis per visas duomenų srauto dalis nuosekliai, eidamas per jas, jis eis per visus duomenų srauto blokus ir eidamas per juos, eis per kiekvieną bloko baitą ir jam naudos XOR operaciją su prieš tai buvusių bloko baitų XOR operacijų rezultatu. Perėjus per visą bloką, prie bloko pabaigos bus pridėtas gautasis baitas. Perėjus per visus duomenis, kiekvienas blokas turės savo tikrinimo baitą, einantį po bloko.

Lygiagrečiu atveju algoritmas nagrinėjamam modeliui nurodys, kad kiekvieną duomenų srauto dalį reikia vykdyti lygiagrečiai. Kadangi dalių bus tiek, kiek procesorius turi branduolių, modeliai turės galimybę kiekvieną dalį paskirstyti ant skirtingų branduolių. Taip pat tuo pačiu bus testuojama, kaip greitai veikia nagrinėjami modeliai, kai turimas nedidelis kiekis lygiagrečių užduočių, palyginus su matricų daugybos atveju, kur buvo daugybė lygiagrečių užduočių.

OpenMP realizacijoje sprendimas atrodo taip:

```
#pragma omp parallel
     #pragma omp single
     {
          for (int i = 0; i < partsLength; i++)
          {
               #pragma omp task
               encodePart(parts[i]);
          }
          #pragma omp taskwait
     }
```

Čia yra naudojamos keturios kompiliatoriaus direktyvos: Parallel, single, task ir taskwait.

Parallel direktyva nurodo, jog po jos einantį kodo bloką turi vykdyti visos gijos. Ji reikalinga tam, kad pradėtų lygiagretumą. Single direktyva apriboja, kad kodo bloką vienu metu

vykdytų tik viena direktyva, nes nėra poreikio, kad tą patį bloką vykdytų kiekviena gija. Tada einama per visas duomenų srauto dalis ir kiekvienai jų apdoroti sukuriama lygiagreti užduotis su direktyva task. Sukūrus visas užduotis, pasiekiama direktyva taskwait, kuri nurodo, jog sekanti kodo komanda turės būti vykdoma tik kai visos sukurtos užduotys baigs darbą. Taigi, direktyva taskwait yra skirta sinchronizuotis visoms lygiagrečioms gijoms. Po sinchronizacijos srautas bus apdorotas ir turės pridėtus tikrinimo baitus.

Cilk Plus realizacija analizuojamam algoritmui prašo mažiau kodo:

```
for (int i = 0; i < partsLength; i++)
     cilk_spawn encodePart(parts[i]);
cilk_sync;
```

Šiame algoritme remiamasi cilk_spawn ir cilk_sync raktiniais žodžiais. Kaip ir nuosekliuoju atveju, taip ir šiuo, per duomenų srauto dalis yra einama tiesiškai. Einant tiesiškai kiekviena iteracija sukuria naują lygiagrečią cilk užduotį panaudojant raktinį žodį cilk_spawn. Todėl šiuo atveju metodas encodePart bus vykdomas kaip cilk užduotis. Praėjus visas iteracijas prieinamas raktinis žodis cilk_sync. Jis nurodo, kad sekanti kodo komanda turi būti vykdoma tik kai visos užduotys bus pilnai įvykdytos. Užduotims baigus darbą, srautas bus papildytas tikrinimo baitais.

Intel TBB sprendime fokusuojamasi į specialią duomenų struktūrą, vadinamą task_group. Ši struktūra yra TBB užduočių konteineris, kuris turi metodus darbui su lygiagrečių užduočių rinkiniu. Nagrinėjamoje užduotyje bus naudojami du metodai. Metodas run sukuria naują užduotį pagal parametruose paduotą funkciją, ją prideda į konteinerį ir iš karto baigia metodo run darbą, nelaukdamas užduoties pabaigos. Taigi, run metodas leidžia startuoti duomenų srauto dalies tikrinimo baitų apdorojimo užduočiai. Visų užduočių laukimui yra kitas task_group struktūros metodas – wait. Iškvietus šį metodą, jis baigsis tada, kai visos konteineryje esančios užduotys baigs darbą. Intel TBB sprendimo realizacija pateikta žemiau:

```
task_group g;
for (int i = 0; i < partsLength; i++)
{
     Unsigned char* part = parts[i];
     g.run([&, part] {
          encodePart(part);
     });
}
g.wait();
```

Paskutinis įrankis, su kuriuo yra įgyvendinta išilginio perteklumo tikrinimo užduotis, yra Google Go programavimo kalba. Užduotyje kiekvienos duomenų srauto dalies apdorojimui yra sukuriama go paprogramė. Visos paprogramės yra sudedamos į konteinerio stiliaus duomenų struktūrą WaitGroup ir surinkus visas go paprogrames, yra laukiama, kol jos baigs vykdytis naudojant laukimui skirtą WaitGroup tipo metodą – Wait. Algoritmas realizuojamas tokiu būdu:

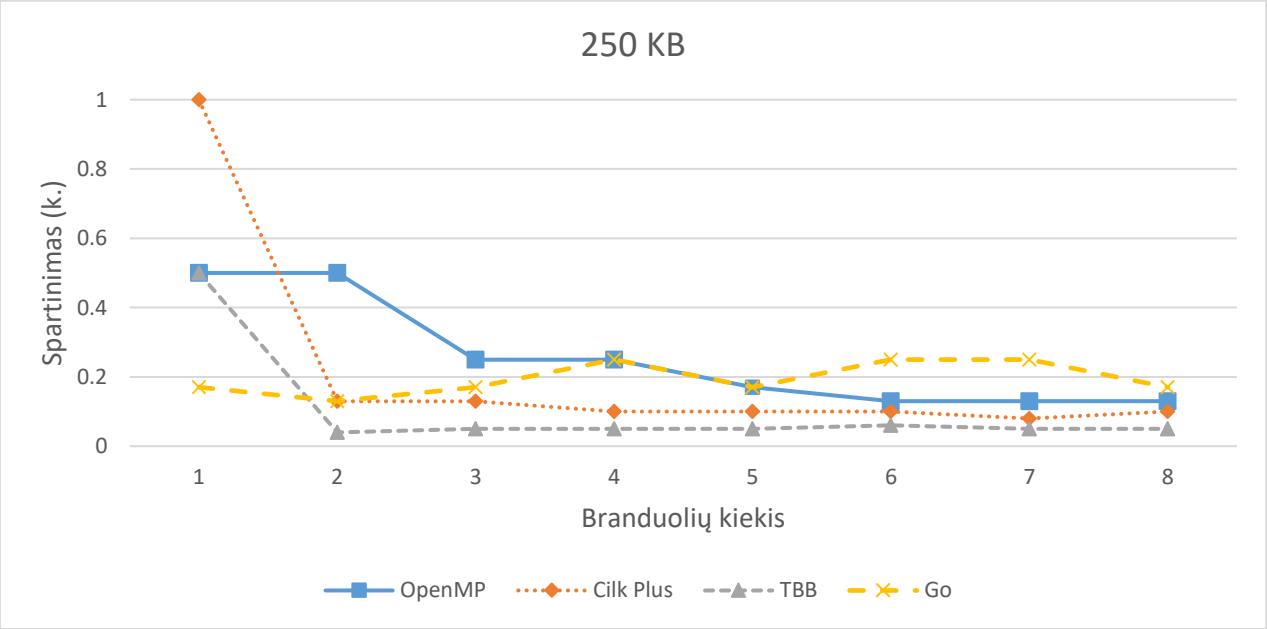
```
var wg sync.WaitGroup
wg.Add(partsLength)
```

```
for i := 0; i < partsLength; i++ {
     go func (i int){
          encodePart(parts[i])
          wg.Done()
     }(i)
}
wg.Wait()
```

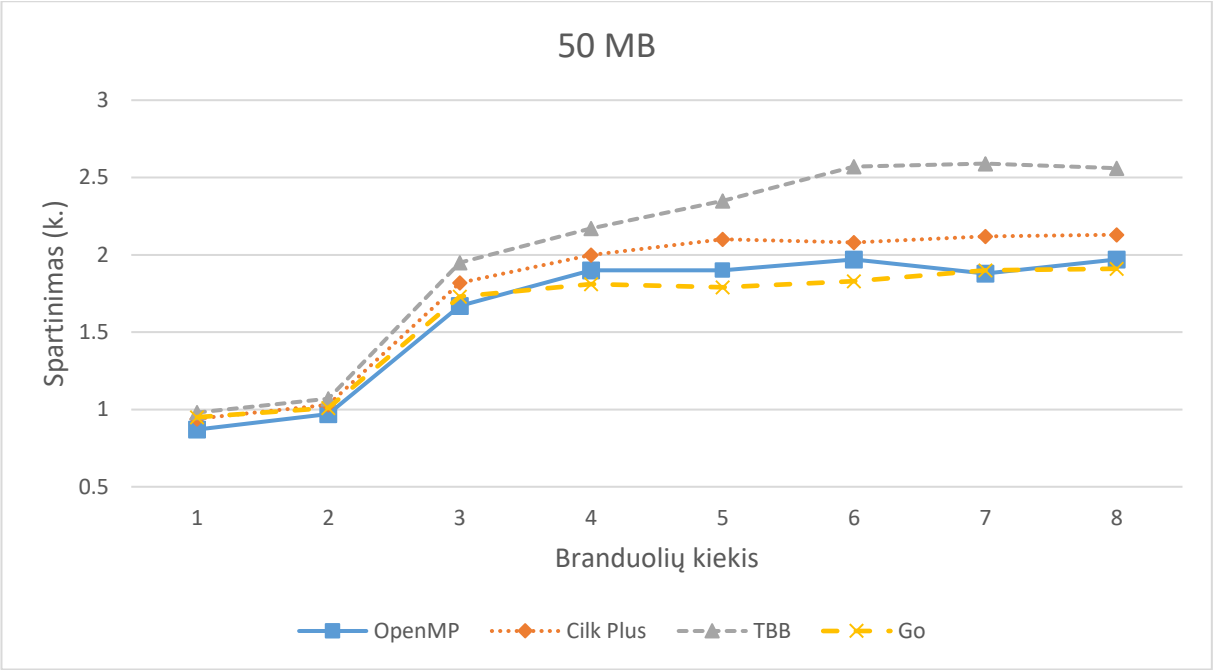
5.2.2. Rezultatai

Duomenų srauto papildymas išilginio perteklumo tikrinimo baitais buvo atliekamas su trimis skirtingų dydžių duomenų srautais. Pirmajame matavime buvo apdorojamas 250 KB duomenų srautas norint išmatuoti, kaip veikia mažo dydžio duomenų srautas. Antruoju atveju buvo atliekami matavimai su 50 MB duomenų srautu. Ir trečiuoju matavimu buvo matuojama, kaip apdorojamas 1 250 MB duomenų srautas.

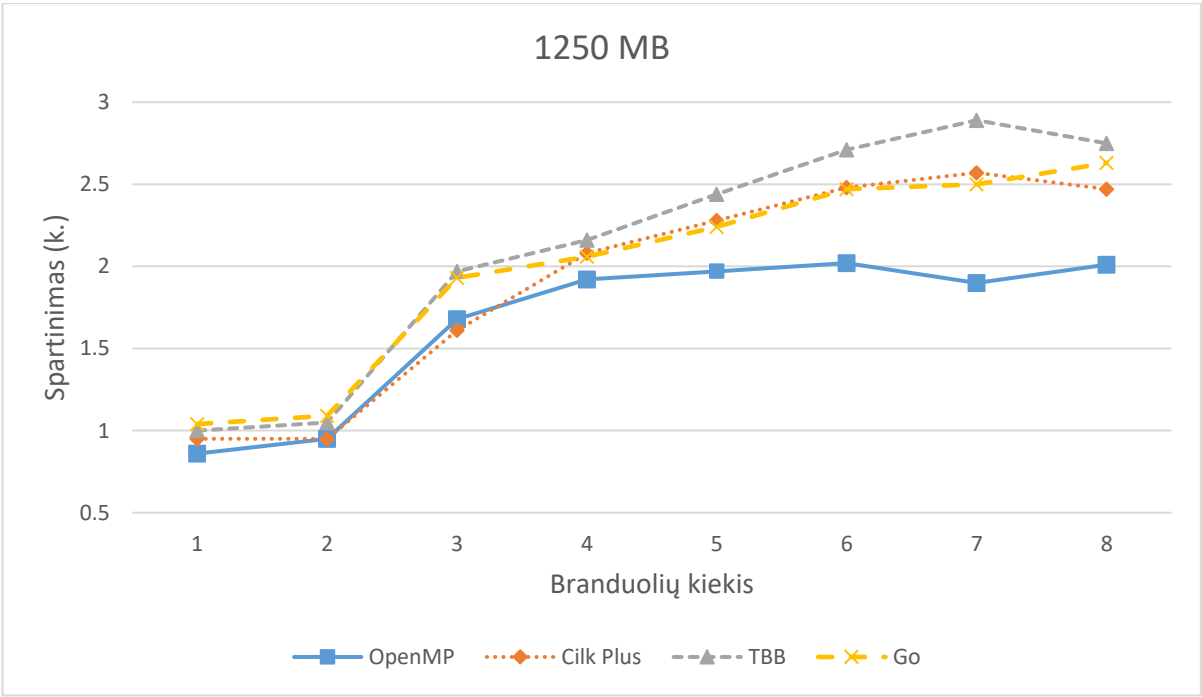
Gautos trukmių diagramos pateiktos 4, 5, 6 prieduose, o spartinimai pateikiami žemiau:



8 pav. 250 KB išilginio perteklumo tikrinimo spartinimai lyginant su nuoseklia versija



9 pav. 50 MB išilginio perteklumo tikrinimo spartinimai lyginant su nuoseklia versija



10 pav. 1250 MB išilginio perteklumo tikrinimo spartinimai lyginant su nuoseklia versija

	OpenMP	**Cilk Plus**	**TBB**	**Go**
250 KB	0.5	**1**	0.5	0.25
50 MB	1.97	2.13	**2.59**	1.91
1 250 MB	2.02	2.57	**2.89**	2.63

2 lentelė. Maksimalūs išilginio perteklumo tikrinimo spartinimai lyginant su nuoseklia versija

Nuoseklusis algoritmas veikė efektyviausiai su bet kokiu branduolių skaičiumi, kai buvo skaičiuojamas 250 KB duomenų srautas. Tai rodo spartinimo diagrama, kurioje matoma, kad

spartėjimai laikėsi žemiau arba ties vieneto riba. Skaičiuojant 50 MB srautą, jau ties dviem branduoliais visi kiti modeliai pradėjo jį lenkti beveik du kartus. Identiška situacija buvo ir apdorojant 1250 MB srautą.

Žvelgiant į 250 KB duomenų srautą, kiekvienas modelis beveik visada efektyviausiai veikė su vienu branduoliu. To buvo galima tikėtis, nes tokiu atveju, programos naudodavo tik vieną lygiagrečią užduotį. Augant užduočių skaičiui ir dirbant su tokiu mažu srautu, išryškėja tai, kad didžioji dalis darbo tekdavo išlygiagretinimo valdymui, o ne srauto apdorojimui, dėl to visi C++ modeliai lėtėjo didinant užduočių skaičių. Go versija – išsiskiria, nes jos veikimas išliko panašus nepriklausomai nuo gijų ir branduolių skaičiaus. Taip pat išsiskiria ir TBB versija, kadangi jos veikimas lėtėjo ypač stipriai, didinant branduolių kiekį. Tai matome iš žemiausio spartėjimo koeficiento, kuris buvo prastesnis už bet kurį kitą nagrinėtą modelį šiame sraute.

Atliekant išilginį perteklumo tikrinimą su 50 MB duomenų srautu, užduoties išlygiagretinimas greitino sprendimus. Visi nagrinėjami modeliai greitėjo panašiu greičiu. Didžiausias spartinimas pastebimas ties trimis branduoliais. Toliau didinant branduolių kiekį, spartinimas sumažėjo, tačiau išliko teigiamas. Šiuo atveju greičiausiai spartėjo ir greičiausiai veikė TBB versija, toliau ėjo visi kiti modeliai, kurių spartinimai skyrėsi vos per kelias šimtąsias.

Apdorojant didžiausią nagrinėtą – 1250 MB srautą, spartinimai buvo labai panašūs kaip ir su 50 MB srautu. Čia greičiausiai veikė ir didžiausią spartinimą turėjo taip pat TBB bibliotekos sprendimas. Jo maksimalus spartinimas siekė 2.89 karto. Go versijos spartinimas buvo mažesnis – 2.63 karto, o Cilk Plus versijos – 2.57 karto. Žvelgiant į užduoties įvykdymo spartinimus su daugiau nei 4 branduoliais, OpenMP veikė lėčiausiai ir jos maksimalus spartinimas buvo 2.02 karto lyginant su nuoseklia versija.

Apibendrinant, skyriuje buvo sprendžiama užduotis, kurioje buvo kuriamas nedidelis kiekis ilgai dirbančių lygiagrečių užduočių. Atlikus matavimus, iš matavimų buvo gautos kelios išvados.

Pirmoji būtų tai, kad pridedant išilginio perteklumo tikrinimo baitus mažiems duomenų srautams vis dėlto geriausia yra naudotis nuosekliuoju algoritmu. Antra, dirbant su didesniais duomenimis, išlygiagretinimas darbą paspartina kelis kartus, o Intel TBB su tokiu uždaviniu tvarkosi efektyviausiai. Ir galiausiai, sprendžiant uždavinį su nedideliu lygiagrečių darbų kiekiu ir panašiais sprendimo metodais, Google Go kalba, pasirodo, gali būti tokia pati efektyvi, kaip ir C++ kalba bei jos modeliai.

5.3. Quicksort rikiavimas

Yra ne vienas būdas surikiuoti duomenis tam tikra eilės tvarka. Skirtingi rikiavimo algoritmai pasižymi skirtingais užduoties sprendimo būdais bei skirtingais rikiavimo greičiais. Taip pat vieni algoritmai yra labiau išlygiagretinami už kitus.

Vienas iš greičiausių ir labiausiai išlygiagretinamų rikiavimo algoritmų yra Quicksort rikiavimas. Jis buvo sukurtas 1959 metais ir vis dar yra plačiai naudojamas, nes yra greitesnis už daugumą kitų rikiavimo algoritmų [Ski08].

Quicksort rikiavimas remiasi elementų palyginimu, todėl jis gali būti naudojamas bet kokiam elementų rinkiniui, kuriam turint du bet kokius elementus, galima nustatyti, kuris iš jų yra didesnis.

Patį rikiavimą galima apibūdinti kaip „skaldyk ir valdyk“ metodą, kurio metu algoritmas suranda elementą, vadinamą ašimi (angl pivot). Tada pagal ašies elemento reikšmę rikiavimas padalija rinkinį į dvi dalis taip, kad kairėje esantys elementai yra mažesni už ašinį elementą, o dešinėje – didesni. Tada rekursyviai kviečiamas Quicksort rikiavimas kairei pusei ir tada rekursyviai kviečiamas Quicksort rikiavimas dešinei pusei. Šiuo būdu algoritmui baigus darbą, elementų rinkinys būna surikiuotas.

Šis rikiavimas nėra stabilus. Tai reiškia, kad vienodi elementai nebūtinai išlaiko tokį patį eiliškumą, kurį turėjo prieš rinkinio surikiavimą. Tačiau Quicksort yra labai mažai atminties reikalaujantis rikiavimas, nes gali rikiuoti elementus paties sąrašo viduje. Taigi, jam nereikia atminties papildomam sąrašui.

Kalbant apie greitį, blogiausiu atveju Quicksort reikalauja O (n^2) palyginimų išrikiuoti n elementų, o vidutiniškai – O (n log n) palyginimų.

Pagrindinė priežastis, kodėl šiame darbe pasirinktas spręsti būtent Quicksort rikiavimo algoritmas, yra tai, jog jį galima išlygiagretinti beveik nenukrypstant nuo nuosekliosios jo versijos. Antroji priežastis yra tai, jog Quicksort priklauso grafo perėjimo (angl. graph traversal) lygiagrečių uždavinių grupei [God17], kuri darbe dar nebuvo nagrinėta.

5.3.1. Realizacijos

Šiame skyriuje yra matuojamas uždavinys su tokia sąlyga: turimas tam tikro dydžio sveikųjų skaičių masyvas, kurio skaičius reikia surikiuoti didėjimo tvarka.

Bendru atveju Quicksort algoritmas išrikiuoti elementus masyve yra realizuojamas tokiu būdu:

```
Function Quicksort(A, low, high)
     If low < high
          p = Partition(A, low, high)
          Quicksort(A, low, p - 1)
          Quicksort(A, p+1, high)
```

Kur *A* yra elementų masyvas, *low* yra rikiuojama masyvo dalies pradžia, *high* – masyvo dalies pabaiga, o funkcija *Partition* yra skirta pasirinkti masyvo dalies ašį ir pagal ją išdėstyti elementus. Skirtingos realizacijos remiasi skirtingais ašies pasirinkimo būdais. Paprastumo dėlei darbe naudojamas Nico Lamuto [Ben99] sukurtas ašies pasirinkimo metodas, kuriame ašis yra paskutinis dalies elementas. Toks ašies pasirinkimas yra lengvai realizuojamas ir suprantamas. Visa funkcija tokiu atveju atrodo taip:

```
Function Partition(A, low, high)
     pivot = A[high]
     i = low
     For j = low to high
          If A[j] < pivot
               sukeisti A[i] ir A[j] vietomis
               i = i + 1
     sukeisti A[i] ir A[high] vietomis
     return i
```

Visam masyvui išrikiuoti galima aprašyti papildomą funkciją:

```
Function Quicksort(A)
     return Quicksort(A, 0, length(A) – 1)
```

Vykdant šį algoritmą tiesiškai, jis rikiuos elementus nuo kairės pusės ir eis link dešinės pusės elementų. Algoritmo išlygiagretinimas orientuosis į tai, kad atskiras masyvo dalis galima rikiuoti lygiagrečiai. Taigi, rekursyvius Quicksort kvietimus būtų galima vykdyti lygiagrečiai.

Buvo išbandyti keturi būdai išlygiagretinti algoritmą. Pirmiausiai buvo bandyta kiekvieną rekursyvų kvietimą kviesti lygiagrečiai, tada bandyta kviesti tik vieną kvietimą lygiagrečiai, o kitą – nuosekliai. Geresnius rezultatus pateikė kairės dalies rikiavimo kvietimas lygiagrečiai, negu kad abiejų dalių kvietimas lygiagrečiai, todėl galutiniame sprendime nagrinėjamas būtent tik kairės dalies kvietimas lygiagrečiai. Taip pat, testuojant mažus masyvus, visiškai nuoseklusis variantas veikė kur kas greičiau, todėl buvo pridėta papildoma optimizacija – kai rikiuojamos dalies dydis yra mažesnis, negu 100 elementų, tada dalies rikiavimas kviečiamas nuosekliai, o ne lygiagrečiai.

OpenMP atveju reikia modifikacijų dviejose vietose. Pirmiausiai reikia pradinį rikiavimo metodą apglausti su išlygiagretumo direktyvomis parallel ir single:

```
#pragma omp parallel
     #pragma omp single no wait
          quicksort(array, 0, arrayLength - 1);
```

Pasibaigus parallel direktyvai, visos gijos susisinchronizuoja. Kita direktyva single turi no wait parametrą, kuris nurodo, jog nereikia laukti lygiagrečių užduočių įvykdymo, kad būtų vykdoma sekanti operacija. Antroji modifikacija yra paleisti kairiąją dalį kaip lygiagrečią užduotį, jeigu jos elementų kiekis viršija apibrėžtą kiekį:

```
if (high – low < cutOff)
     quicksort(array, low, pivot – 1);
else
{
#pragma omp task
```

```
        quicksort(array, low, pivot – 1);
    }
    quicksort(array, pivot +1, high);
```

Kitų modelių realizacijos veikia tokiu pačiu principu. Cilk Plus atveju pradinis metodas atrodytų taip:

```
quicksort(array, 0, arrayLength - 1);
cilk_sync;
```

Čia reikia tik raktažodžio cilk_sync, kad visos cilk užduotys susisinchronizuotų prieš baigiant rikiavimą. O rekursyvūs kvietimai modelyje įgyvendinami cilk_spawn raktažodžiu:

```
if (high – low < cutOff)
     quicksort(array, low, pivot – 1);
else
     cilk_spawn quicksort(array, low, pivot – 1);
quicksort(array, pivot +1, high);
```

Intel Tbb modelyje visų dalių rikiavimo užduočių sinchronizacijai naudojama išilginiame klaidų tikrinime naudota task_group duomenų struktūra. Su ja įgyvendinimas yra štai toks:

```
task_group g;
quicksort(array, 0, arrayLength – 1, g);
g.wait();
```

O rekursyvus kvietimas kairei masyvo daliai kviečiamas užduočių grupei kaip nauja užduotis:

```
if (high – low < cutOff)
     quicksort(array, low, pivot – 1, g);
else
     g.run([&, low, pivot] {
          quicksort(array, low, pivot – 1, g);
     });
quicksort(array, pivot +1, high);
```

Analogiškai TBB versijai Google Go kalbos realizacija orientuojasi į sync.WaitGroup duomenų struktūrą pradiniam rikiavimo metodui aprašyti:

```
var g sync.WaitGroup
quicksort(array, 0, len(array) – 1, g)
g.Wait()
```

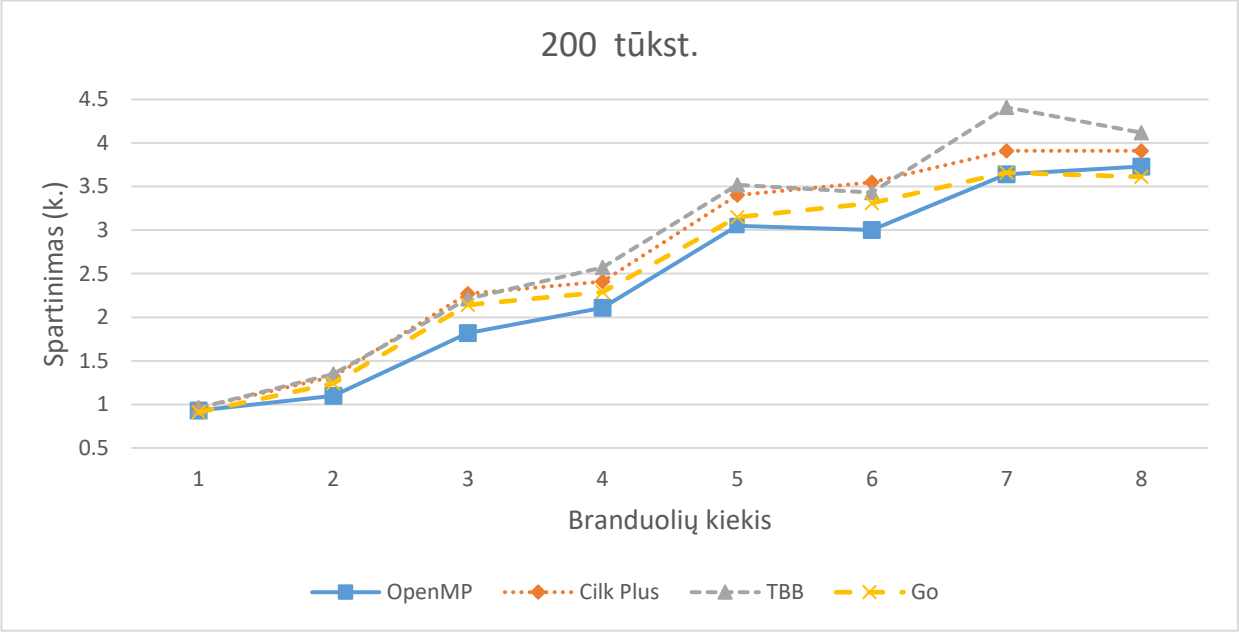
O rekursyvus kvietimas realizuojamas sukuriant po naują go paprogramę rekursyviam kvietimui:

```
if (high – low < cutOff)
     quicksort(array, low, pivot – 1, g)
else {
     g.Add(1)
     go func(array []int, low int, pivot int, g sync.WaitGroup) {
          quicksort(array, low, pivot – 1, g)
          g.Done()
     }(array, low, pivot, g)
}
quicksort(array, pivot +1, high)
```

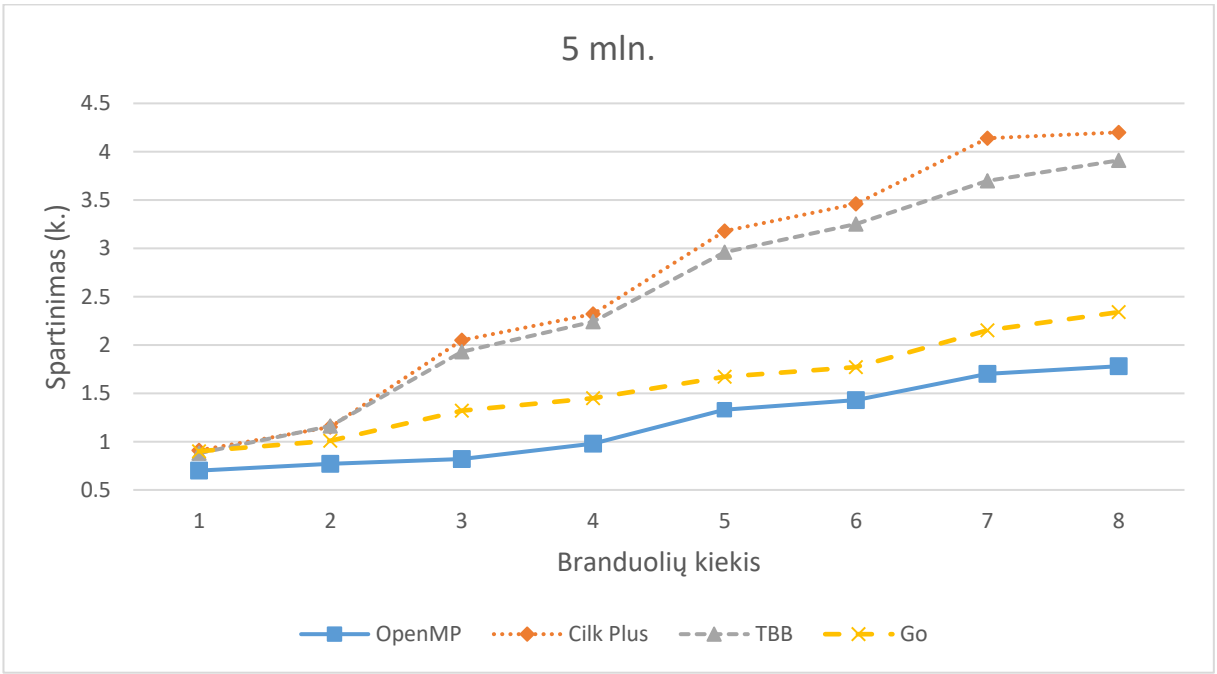
5.3.2. Rezultatai

Quicksort rikiavimas buvo matuojamas su trijų dydžių sveikųjų skaičių sąrašais. Pirmasis sąrašas buvo 500 tūkstančių elementų ilgo, antrasis – 5 milijonų elementų ilgio ir trečiasis – 20 milijonų elementų ilgio. Kiekvienas sąrašas buvo rikiuojamas 100 kartų ir imamas jų vidurkis.

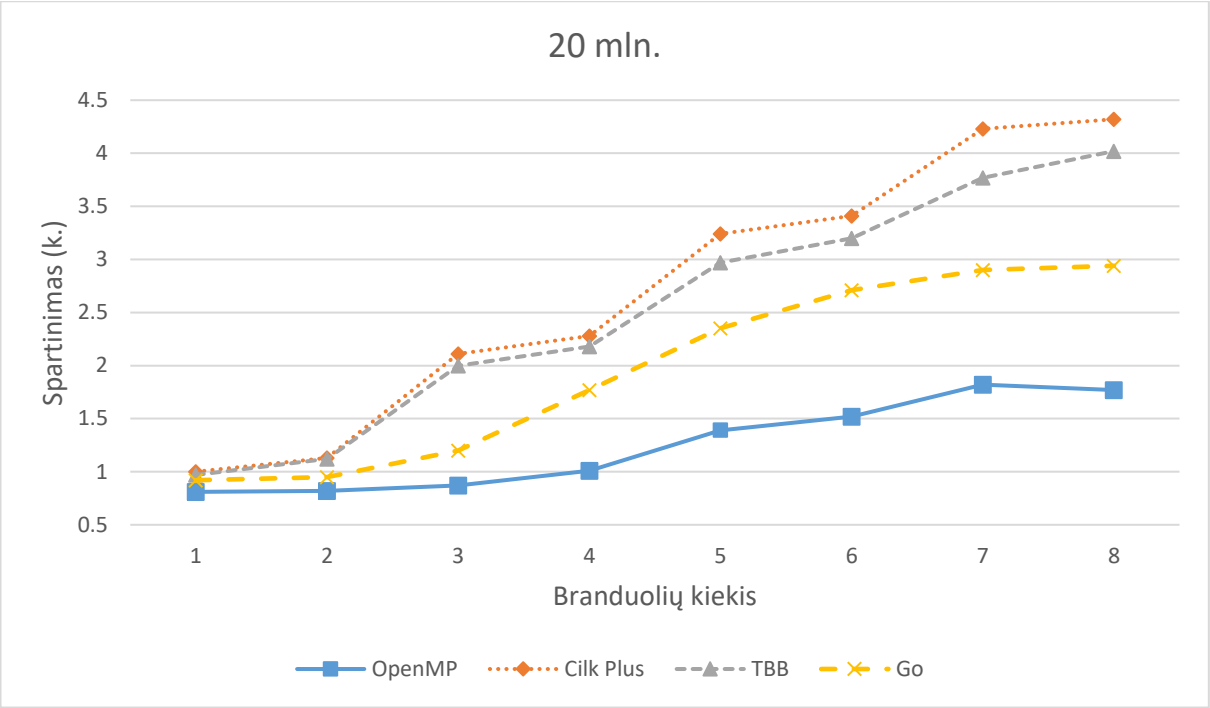
Gauti matavimo rezultatai pateikti 7,8 ir 9 prieduose bei žemiau:



11 pav. 200 tūkst. elem. sąrašo quicksort rikiavimo spartinimai lyginant su nuoseklia versija



12 pav. 5 milijonų elem. sąrašo quicksort rikiavimo spartinimai lyginant su nuoseklia versija



13 pav. 20 milijonų elem. sąrašo quicksort spartinimai lyginant su nuoseklia versija

	OpenMP	Cilk Plus	TBB	Go
200 K	3.73	3.91	**4.41**	3.66
5 000 K	1.78	**4.20**	3.91	2.34
20 000 K	1.82	**4.32**	4.02	2.94

3 lentelė. Maksimalūs quicksort rikiavimo spartinimai lyginant su nuoseklia versija

Rikiuojant 200 tūkst. elementų sąrašą, visi išlygiagretinimo modeliai veikė beveik identiškais greičiais. Kiek lėtesnis buvo tik OpenMP, kurio spartinimas beveik visais matuotais atvejais buvo žemiausias. Su dauguma branduolių kiekiu, labiausiai spartėjo Intel TBB bibliotekos versija.

Rikiuojant 5 mln. ir 20 mln. sąrašus, greičiausiai rikiavo Cilk Plus modelis. Jo maksimalus spartinimas su 5 mln. sąrašu buvo 4.2 karto, o su 20 mln. sąrašu – net 4.32 karto.

Vos keliais procentais lėčiau rikiavo TBB modelis. Jo maksimalus užfiksuotas spartinimas su didžiausiu sąrašu taip pat viršijo 4 kartų ribą. Taip pat verta pastebėti, kad spartėjimai didėjant branduolių kiekiui augo beveik tiesiškai.

Tuo tarpu OpenMP rikiavimo trukmės su 5 ir 20 mln. elementų sąrašais buvo prasčiausios iš visų išlygiagretinimo modelių. Abiem sąrašams jis greičiau veikė už nuoseklų rikiavimą tik su daugiau nei 4 branduoliais, o pagal maksimalų spartinimą su nuosekliu sprendimu buvo greitesnis atitinkamai tik 1.78 ir 1.82 karto.

Apibendrinant rezultatus galima teigti, kad sprendžiant Quicksort rikiavimo uždavinį, kuriame rekursyviai buvo kuriamos lygiagrečios užduotys, efektyviausiai su tuo dorojosi Intel

Thread Building Blocks ir Cilk Plus išlygiagretinimo įrankiai. Jie gana ryškiai lenkė kitus matuotus įrankius.

5.4. Skaičių skaičiavimas su „MapReduce“ šablonu

MapReduce yra programavimo šablonas, skirtas lygiagrečiai apdoroti didelius duomenų kiekius. Šiuo šablonu paremta programa yra sudaryta iš dviejų pagrindinių funkcijų – map ir reduce.

Map funkcijos darbas yra priimti duomenų rinkinį, jį perrikiuoti, išfiltruoti ar pakeisti rinkinio elementų struktūrą. Tada toks perdirbtas rinkinys ar keli rinkiniai yra siunčiami į reduce funkciją. Reduce funkcija rinkinį ar rinkinius suagreguoja į vieną duomenų rezultatą.

Šiuo šablonu sprendžiami uždaviniai yra priskiriami to paties pavadinimo – MapReduce lygiagrečių uždavinių grupei.

5.4.1. Realizacijos

MapReduce grupės matavimui yra sprendžiamas toks uždavinys:

Turime sveikųjų skaičių rinkinį. Reikia rasti, kiek kartų jame pasikartoja kiekvienas skaičius.

Pavyzdžiui, turime skaičių rinkinį: 10, 6, 3, 6, 5.

Jo atsakymas būtų, kad 6 pasikartoja 2 kartus, 10, 3 ir 5 pasikartoja po 1 kartą.

Be abejo, tokį uždavinį spręsti galima įvairiais būdais, bet šiame darbe jis bus sprendžiamas remiantis MapReduce šablonu.

Žvelgiant į sprendimą bendru atveju, pradinis skaičių rinkinys yra padalijamas į mažesnio dydžio skaičių rinkinius, konkrečiau – pradinis rinkinys dalijamas į 3000 elementų rinkinius, kurių kiekvienam padalytam rinkiniui map funkcija suskaičiuoja, kiek kartų kiekvienas skaičius pasikartoja tame rinkinyje. Tada, kai visi rinkiniai yra apskaičiuoti, gautiems suskaičiuotiems atskirtų rinkinių skaičiams kviečiama reduce funkcija, kuri susumuoja visų rinkinių skaičių pasikartojimus ir grąžina galutinį rezultatą – skaičius ir jų pasikartojimų kiekį pradiniame rinkinyje.

Sprendžiant užduotį nuosekliai, kiekvienas rinkinys apdorojamas su map funkcija paeiliui ir visi map funkcijos rezultatai dedami į vieną sąrašą. Užduoties išlygiagretinimui yra orientuojamasi būtent į šią programos dalį. Lygiagrečiose versijose kiekvienas rinkinys yra apdorojamas lygiagrečiai ir jo rezultatas talpinamas į bendrą duomenų struktūrą. Nuosekliojoje versijoje naudojama vektoriaus duomenų struktūra iš C++ std bibliotekos. Visa realizacija aprašoma tokiu kodo bloku:

```
Int chunkSize = chunks.size();
vector<map<int, int>> mapResults;
mapResults.resize(chunkSize);
```

```
for (int i = 0; i < chunkSize; i++) {
     map<int, int> mapResult = map(chunks[i]);
     mapResults.push_back(mapResult);
}
return reduce(mapResults);
```

Lygiagrečiose versijose skirtingai realizuojama ciklo dalis, kurioje yra naudojamos atitinkamame modelyje esančios duomenų struktūros, pritaikytos aprašytam scenarijui.

OpenMP ciklo realizacija įgyvendinama tokiu būdu:

```
vector<map<int, int>> mapResults;
mapResults.resize(chunkSize);
#pragma omp parallel for
     for (int i = 0; i < chunkSize; i++) {
          map<int, int> mapResult = map(chunks[i]);
          #pragma omp critical
               mapResults.push_back(mapResult);
     }
```

Čia naudojama for ciklo išlygiagretinimo direktyva parallel for, kuri paskirsto ciklo blogus skirtingoms gijoms. Ciklo viduje yra naudojama direktyva critical. Ji nurodo, jog po jos einanti komanda įdėti map funkcijos rezultatą į sąrašą, yra kritinė sekcija, todėl turi būti vykdoma vienos gijos vienu metu. Tokiu būdu išvengiama, kad dvi ar daugiau gijų mėgintų dėti elementą į vektorių tuo pačiu metu.

Cilk Plus realizacija:

```
cilk::reducer< cilk::op_list_append<map<int, int>> > mapResults;
cilk_for(int i = 0; i < chunkSize; i++) {
     map<int, int> mapResult = map(chunks[i]);
     mapResults->push_back(mapResult);
}
return reduce(toVector(mapResults));
```

Cikl lygiagretinimui naudojamas jau nagrinėtas cilk_for raktinis žodis. Kiek įdomesnė duomenų struktūra yra reducer, kuri yra saugi dirbti su daugiau nei viena gija vienu metu. Šiame uždavinyje naudojama op_list_append reducer struktūra, kuri sukuria sąrašą, į kurį galima pridėti elementus keliose gijose vienu metu. Dėl to nėra poreikio turėti kritinės sekcijos, kaip OpenMP atveju. Vienintelis trūkumas, jog reduce funkcija tikisi vektoriaus kaip įvesties parametro, todėl reikia papildomai perkelti elementus iš cilk reducer į vektorių. Tačiau palyginus poreikį turėti kritinę sekciją ir elementų perkėlimą, elementų perkėlimas yra įvykdomas greičiau nei gijų sinchronizavimasis.

Intel Tbb realizacija nedaug skiriasi nuo Cilk Plus realizacijos:

```
concurrent_vector< map<int, int> > mapResults;
mapResults.resize(chunkSize);
tbb::parallel_for(size_t(0), size_t(chunkSize), [&](int i) {
          map<int, int> mapResult = MapReduceHelper::map(chunks[i]);
     mapResults.push_back(mapResult);
});
return reduce(toVector(mapResults));
```

Šio modelio atveju for ciklo išlygiagretinimui naudojamas parallel_for konstruktas. Iteracijos jame taip pat nėra blokuojamos dėl sinchronizacijos. Vietoje sinchronizacijos naudojama lygiagrečiam darbui pritaikyta concurrent_vector duomenų struktūra. Sudėjus visus

rinkinių rezultatus į ją, kviečiama funkcija toVector perversti concurrent_vector į vector ir tada kviečiama reduce funkcija.

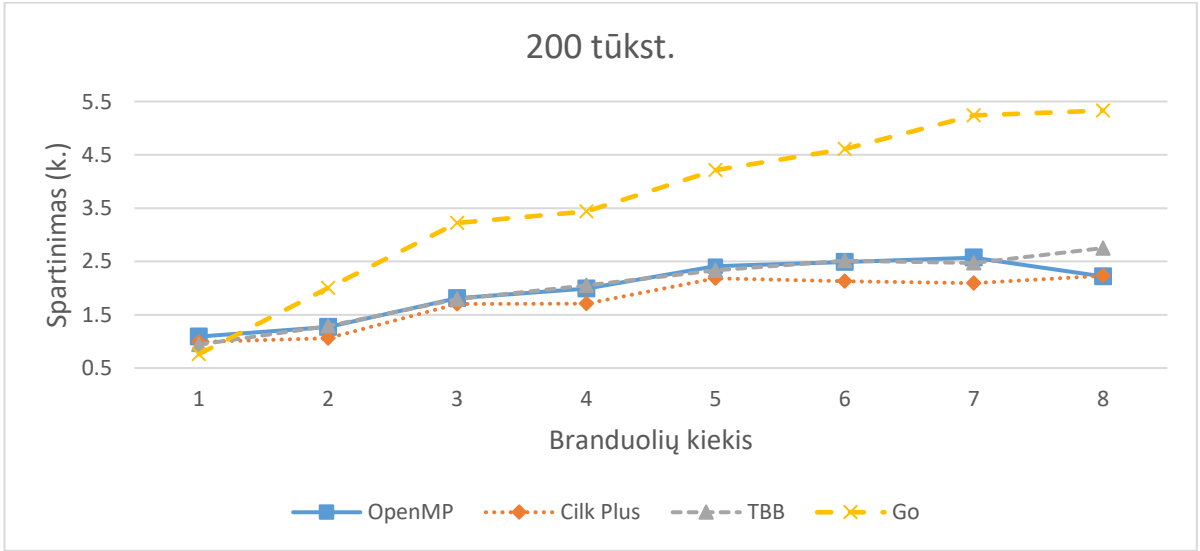
Google Go kalbos versija:

```
var wg sync.WaitGroup
wg.Add(chunkSize)
channel := make(chan map[int]int, chunkSize)
for chunk := chunks.Front(); chunk != nil; chunk = chunk.Next() {
     go func (chunk []int) {
          mapResult := map(chunk)
          channel <- mapResult
          wg.Done()
     }(chunk.Value.([]int))
}
wg.Wait()
close(channel)
return reduce(toVector(mapResults))
```

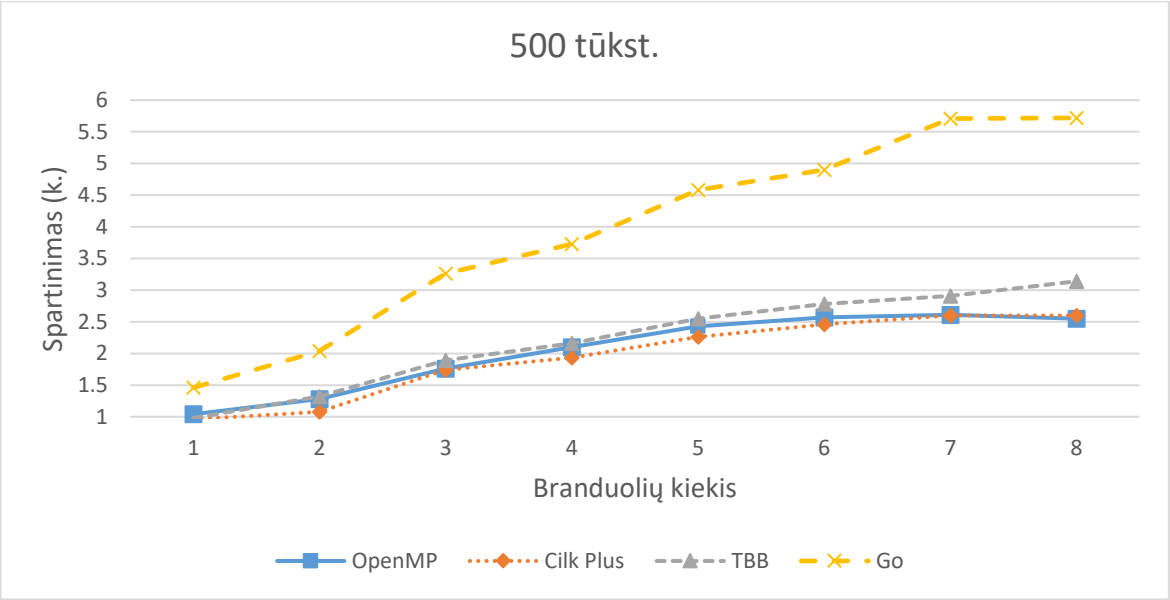
Kaip ir ankstesnėse programose, kiekvienai ciklo iteracijai kuriama go paprogramė. Tačiau šioje programoje iteracijų rezultatų surinkimui naudojama tam skirta duomenų struktūra – kanalas (angl. channel). Čia ji leidžia iš lygiagrečių užduočių surinkti map funkcijos rezultatus ir tada juos perduoti į reduce funkciją. Išlaikant C++ kalbos modelių struktūrą, čia kanalas irgi perverčiamas į list.List duomenų struktūrą (alternatyva C++ vektoriaus struktūrai), kuri perduodama reduce funkcijai.

5.4.2. Rezultatai

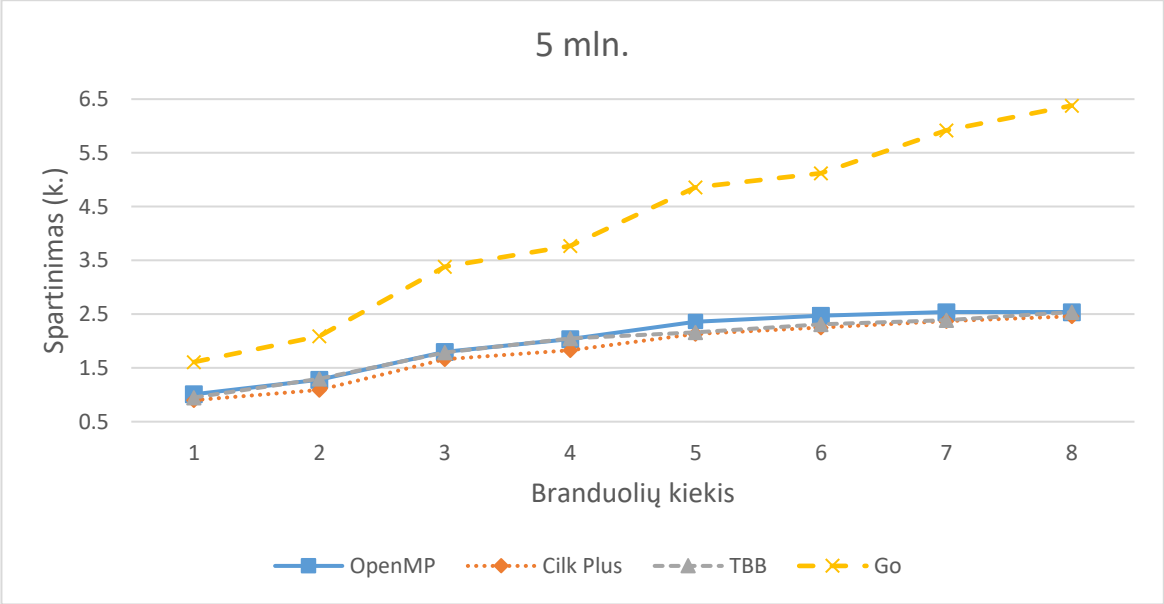
Skaičių skaičiavimo uždavinys yra matuojamas su trimis skirtingo dydžio skaičių rinkiniais: 200 tūkst., 500 tūkst. ir 5 mln. Kiekvienam rinkiniui skaičiavimai buvo leidžiami 100 kartų ir imamas jų vidurkis. Programų vykdymų trukmės pateiktos 10, 11 ir 12 prieduose. Gauti spartinimai pateikiami žemiau:



14 pav. 200 K MapReduce skaičių skaičiavimo spartinimai lyginant su nuoseklia versija



15 pav. 500 K MapReduce skaičių skaičiavimo spartinimai lyginant su nuoseklia versija



16 pav. 5 000 K MapReduce skaičių skaičiavimo spartinimai lyginant su nuoseklia versija

	OpenMP	**Cilk Plus**	**TBB**	**Go**
200 K	2.57	2.23	2.75	**5.33**
500 K	2.61	2.6	3.14	**5.72**
5 000 K	2.54	2.46	2.54	**6.38**

4 lentelė. Maksimalūs MapReduce skaičių skaičiavimo spartinimai lyginant su nuoseklia versija

Žvelgiant į programų vykdymų trukmių rezultatus nesunku pastebėti, jog greičiausiai skaičiavimus atliko Google Go versija. Ji lenkė visus kitus matuotus modelius visuose skaičių rinkiniuose. Vienas iš dažniausiai minimų minėtosios kalbos privalumų yra kanalų efektyvumas. Ir pagal gautus rezultatus galima teigti, kad kanalai yra tikrai efektyvus įrankis dalijantis atmintimi tarp lygiagrečių užduočių.

Toliau rikiavosi visi kiti C++ modeliai. Jie spartėjo beveik identiškai ir visų jų maksimalūs spartinimai svyravo tarp dviejų ir trijų kartų. TBB biblioteka buvo nežymiai greitesnė skaičiuojant su mažesniais ir vidutiniais skaičių rinkiniais. Su didžiausiais skaičiuotais rinkiniais, OpenMP spartėjo labiausiai ir turėjo didžiausią maksimalų spartėjimą tarp C++ modelių.

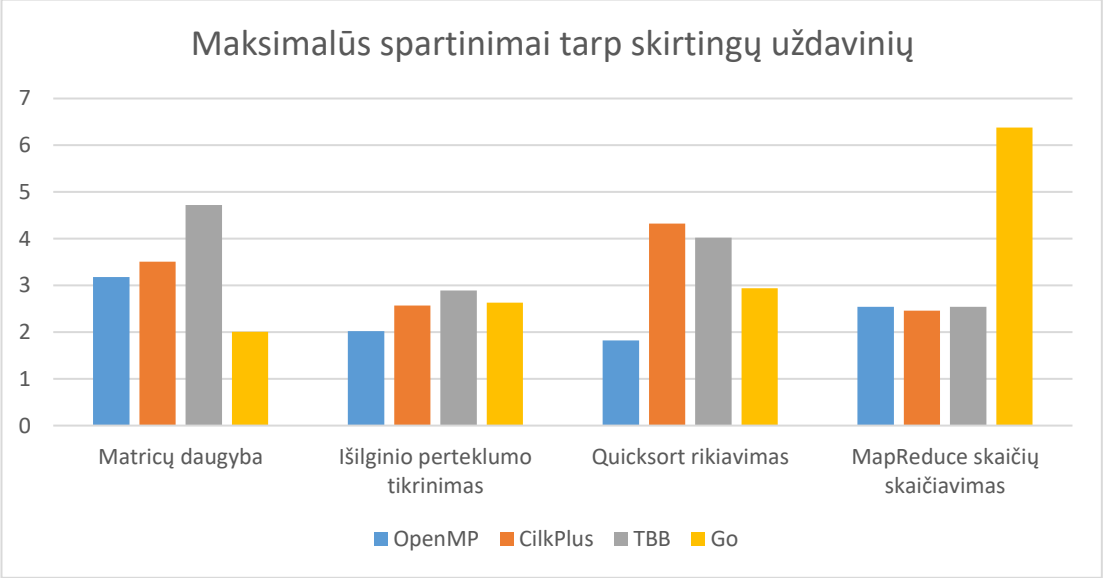
Apibendrinant uždavinio rezultatus, galima daryti išvadą, kad sprendžiant uždavinį, kuriame lygiagrečios užduotys turi dalytis atmintimi, Go turi labai efektyvius įrankius tokiai užduočiai spręsti. Atitinkami įrankiai nagrinėjamuose C++ modeliuose nebuvo tokie efektyvūs lyginant su Go versija.

Lyginant C++ modelius vieną su kitu, nė vienas modelis neturėjo ryškaus pranašumo prieš kitus, todėl negalime išskirti kažkurio vieno modelio.

5.5. Matavimų rezultatų apibendrinimas

Atliekant matavimus skirtingiems uždaviniams buvo gauti spartinimai kiekvienam uždaviniui. Analizuotos spartinimų diagramos leidžia įvertinti kaip skirtingi modeliai lyginasi tame pačiame uždavinyje. Tačiau sunku įvertinti kaip skirtingi modeliai spartėja tarp skirtingų uždavinių.

Sprendžiant šią problemą, kad būtų galima išvysti bendresnį vaizdą, pateikiama stulpelinė diagrama (17 pav.). Joje pateikiami maksimalūs spartinimai kiekviename uždavinyje lyginant su nuoseklia versija. Spartinimai imti iš kiekvieno uždavinio didžiausių matuotų duomenų.



17 pav. Maksimalūs spartinimai tarp skirtingų uždavinių

Žvelgiant į diagramą, pirmiausiai galima pastebėti, kad sprendžiant MapReduce uždavinį, Google Go versija spartėjo įpatingai greitai ir užfiksavo didžiausią spartėjimą tarp visų užduočių. Tačiau kituose uždaviniuose jos spartėjimas atsilikdavo nuo C++ modelių.

Neskaitant Google Go versijos MapReduce skaičių skaičiavimo uždavinyje, Intel TBB visuose uždaviniuose spartėjo labiausiai arba buvo antras labiausiai spartėjantis modelis. Iš šios

diagramos galima teigti, kad Intel TBB biblioteka būtų buvęs geras pasirinkimas spręsti bet kuriai iš šių uždavinių kategorijų.

Cilk Plus ne daug atsiliko nuo TBB modelio. Jį net lenkė rekursyviai kuriant lygiagrečias užduotis. Didžiausias atsilikimas matomas dirbant su išlygiagretintais for ciklais.

OpenMP lygiavosi į kitus C++ modelius tik MapReduce uždavinyje, kuriame jos maksimalus pagreitėjimas sutapo su TBB. Kituose matuotuose uždaviniuose OpenMP spartėjimas buvo lėčiausias. Tai ypač pastebima Quicksort uždavinyje, kur kiti nagrinėti modeliai jį lenkė bent jau pusantro karto.

6. Rezultatai ir išvados

Šiame darbe buvo nagrinėjami bendrosios atminties lygiagretaus programavimo modeliai.

Pirmojoje darbo dalyje buvo analizuojami keturi išlygiagretinimo modeliai – OpenMP, Cilk Plus, Intel TBB ir Google Go. Analizės metu buvo išanalizuota, kaip veikia šie modeliai, kokiomis idėjomis remiasi ir kokius įrankius suteikia programų kūrėjams.

Antrojoje darbo dalyje buvo remiamasi Berklio Kalifornijos universiteto darbu „The Landscape of Parallel Computing Research: A View from Berkeley“. Iš šio darbo buvo išrinktos keturios kategorijos išlygiagretinamų uždavinių, tinkamų išlygiagretinimo modelių palyginimui.

Darbo metu buvo gauti tokie pagrindiniai rezultatai:

1. Išanalizuota, kaip veikia ir kuo pasižymi bendrosios atminties lygiagrečiųjų kompiuterių architektūra.
2. Detaliai išanalizuoti pasirinktieji bendrosios atminties išlygiagretinimo modeliai, jų programavimo šablonai ir struktūros.
3. Realizuoti skirtingų kategorijų uždaviniai, kurie gali būti išlygiagretinti naudojant pasirinktus modelius.
4. Išmatuoti ir palyginti uždavinių spartinimai ir vykdymo trukmės su skirtingais branduolių kiekiais bei skirtingais modeliais.

Apibendrinus gautus rezultatus galima daryti tokias išvadas:

1. Galima teigti, kad programos išlygiagretinimo problema yra aktuali, nes vien C/C++ kalboms skirtų modelių yra daugiau nei keli.
2. Skirtingi modeliai remiasi tomis pačiomis lygiagretaus programavimo idėjomis ir šablonais (šakok-junk, užduočių kūrimas ir pan.), tačiau kiekvienas nagrinėtas modelis turėjo jam būdingų savybių ar optimizacijų skirtingiems atvejams.
3. Lygiagretinant for ciklą su nepriklausomomis iteracijomis, OpenMP, Cilk Plus ir TBB modeliai pasižymi gana panašiu efektyvumu.
4. Google Go kalboje nėra specialių įrankių, kurie ypač lengvai leidžia išlygiagretinti ciklą, priešingai negu C++ nagrinėtuose įrankiuose.
5. Kuriant ir vykdant mažą kiekį lygiagrečių užduočių (angl. parallel tasks) ilgiems skaičiavimams, Intel TBB versijos vykdymo laikas buvo beveik 15 procentų trumpesnis už kitas versijas.
6. Rekursyviai kuriant didelį kiekį lygiagrečių užduočių, Cilk Plus veikė sparčiausiai. OpenMP versija buvo gana ryškiai lėčiausia iš visų išlygiagretintų versijų.
7. Naudojant lygiagrečias užduotis ir lygiagretumą palaikančius konteinerius, Go kalbos versija demonstravo apie dvigubai didesnį spartėjimą už kitus modelius. C++ kalbos instrumentai spartėjo beveik identišku tempu lyginant juos tarpusavyje.

Apibendrinant rezultatus ir išvadas, skirtinguose uždavinių tipuose modeliai pasižymėjo skirtingais efektyvumais. Vienu atveju buvo efektyvesnis vienas modelis, kitu atveju – kitas. Dėl to, prieš renkantis išlygiagretinimo įrankį, verta pagalvoti, kokia problema bus sprendžiama, kaip tikėtina atrodys kodo struktūra ir pagal tai pasirinkti, kurį lygiagretaus programavimo įrankį naudoti.

7. Šaltinių sąrašas

[AAR+10] Alexei Alexandrov, Douglas Armstrong, Hrabri Rajic, Michael Voss, Donald Hayes. High-Level Performance Modeling of Task-Based Algorithms. 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010

[ABC+06] Krste Asanovíc, Rastislav Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The Landscape of Parallel Computing Research: A View from Berkeley, EECS Department University of California, Berkeley, 2006

[Ben99] Jon Bentley (1999). Programming Pearls. Addison-Wesley Professional, 1999.

[CJP08] Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP. Portable Shared Memory Parallel Programming. The MIT Press, 2008, pp. 28

[Cox17] Katherine Cox-Buday. Concurrency in Go: Tools and Techniques for Developers. O'Reilly Media, Inc., 2017. Pp. 70 – 107

[CSM+15] Michael Coblenz, Robert Seacord, Brad Myers, Joshua Sunshine, Jonathan Aldrich. A Course-Based Usability Analysis of Cilk Plus and OpenMP. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2015

[DK15] Alan A. A. Donovan, Brian W. Kernighan. The Go Programming Language, Addison-Wesley Professional, 2015, pp. 225 – 234

[Dun90] Ralph Duncan. A Survev of J Parallel Computer Architectures. IEEE Computer magazine, 1990, pp 11 – 13

[FDC84] David J. Farber, Gary S. Delp, Thomas M. Conte. A Thinwire Protocol for connecting personal computers to the Internet. IETF, 1984.

[God17] Patrick Goddijn. The applicability of Divide-and-Conquer: Graph Traversal in parallel on computational grid. University of Amstredam, 2017, pp 13

[HJ16] Kai Hwang, Naresh Jotwani. Advanced Computer Architecture, 3e. McGraw-Hill Education, 2016, pp. 20 – 30

[ICP20] Intel Corporation. Cilk Plus documentation. 2020. Prieiga per interneta: <https://www.cilkplus.org/tutorial-cilk-plus-keywords>

[IC20] Intel Corporation. Intel C++ Compiler documentation. 2020. Prieiga per internetą: <https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-qopt-matmul-qopt-matmul>

[KH16] David B. Kirk, Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2016, pp. 175 – 196

[MMR+16] Evgeniy Mytsko, Andrey Malchukov, Svetlana Ryzova, Valeiy Kim. A study of hardware implementations of the CRC computation algorithms. EDP Sciences, 2016

[MRR12] Michael McCool, Arch D. Robinson, James Reinders. Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, 2012

[OAB18] OpenMP Application Programming Interface 5.0, OpenMP Architecture Review Board, 2018

[PBF10] Artur Podobas and Mats Brorsson, Karl-Filip Faxén. A Comparison of some recent Task-based Parallel Programming Models. 2010. Prieiga per internetą: <http://soda.swedish-ict.se/3869/1/wool-comparison-final.pdf>

[Pik12] Rob Pike. Go at Google: Language Design in the Service of Software Engineering, 2012. Prieiga per internetą: < https://talks.golang.org/2012/splash.article>

[Pik15] Rob Pike. Go Proverbs, Gopherfest, 2015

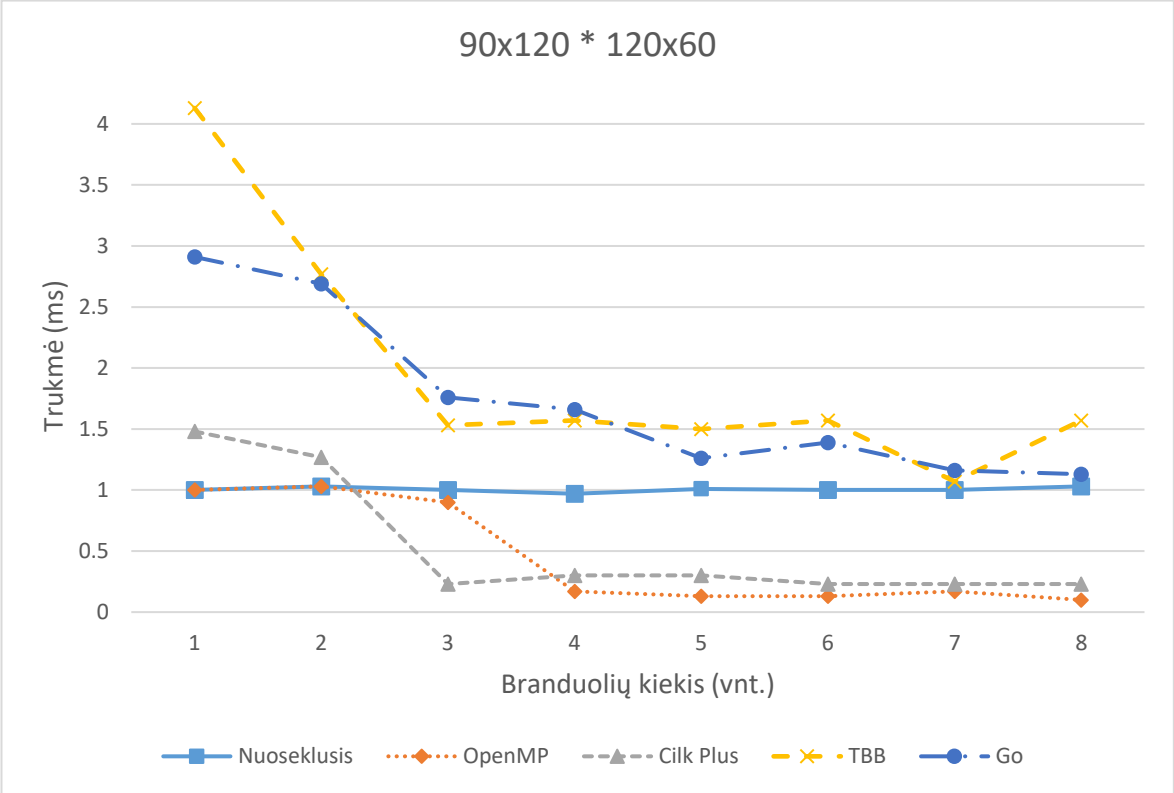
[Rei07] James Reinders. Intel Threading Building Blocks. O'Reilly Media, 2007, pp. 65 – 80

[Ski08] Steven Sol Skiena. The Algorithm Design Manual. Springer, 2008, pp 125-130

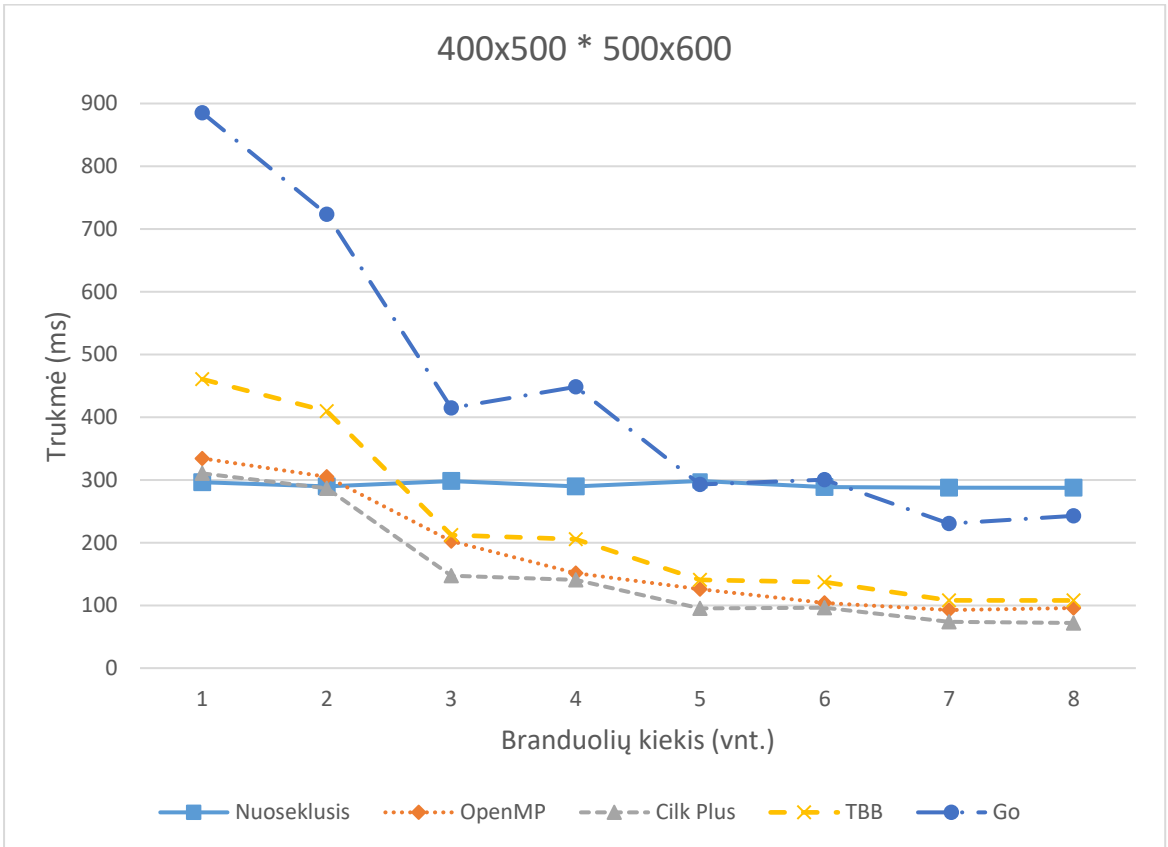
[Sum12] Mark Summerfield. Programming in Go. Addison-Wesley, 2012, pp. 205

[HP12] John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitive Approach. Waltham, Morgan Kaufmann, 2012, pp. 46 – 47

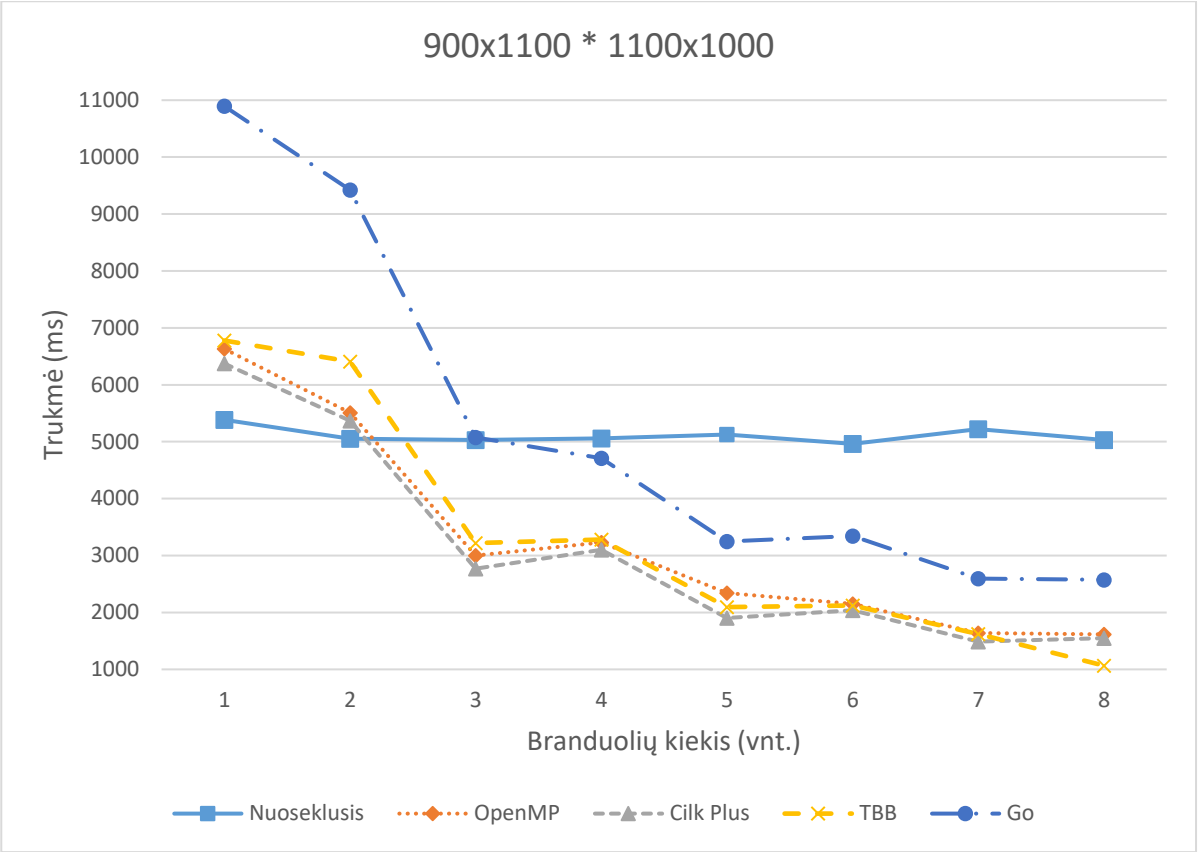
A. Priedai



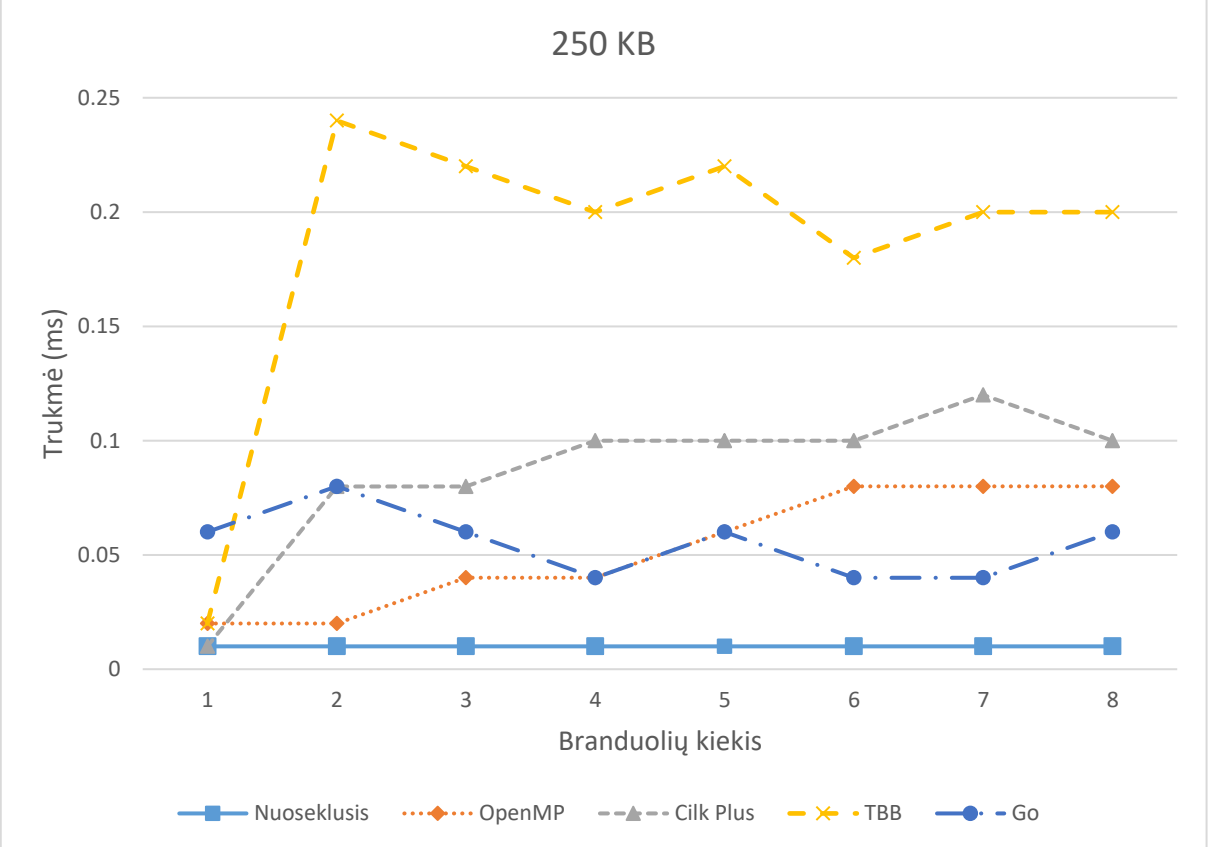
1 pried. Matricų (90x120 * 120x60) daugybos trukmės



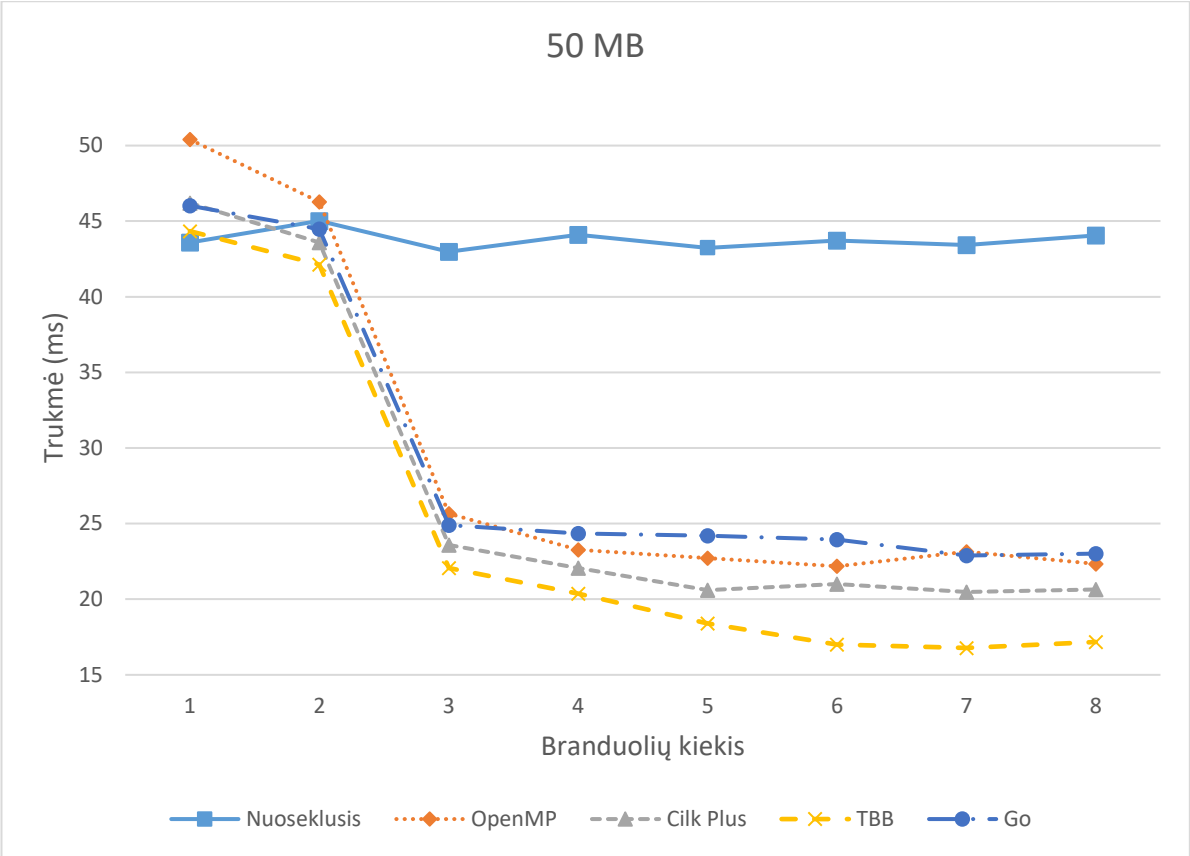
2 pried. Matricų (400x500 * 500x600) daugybos trukmės



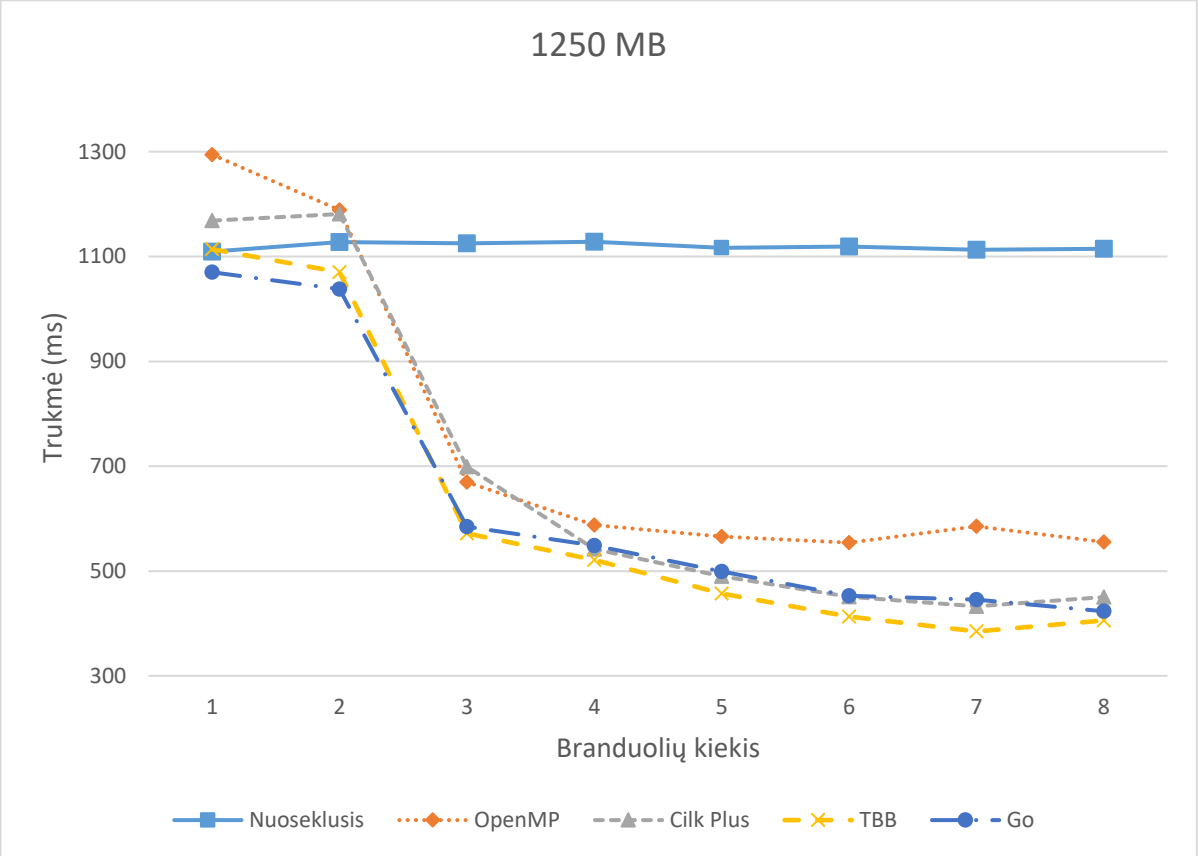
3 pried. Matricų (900x1100 * 1100x1000) daugybos trukmės



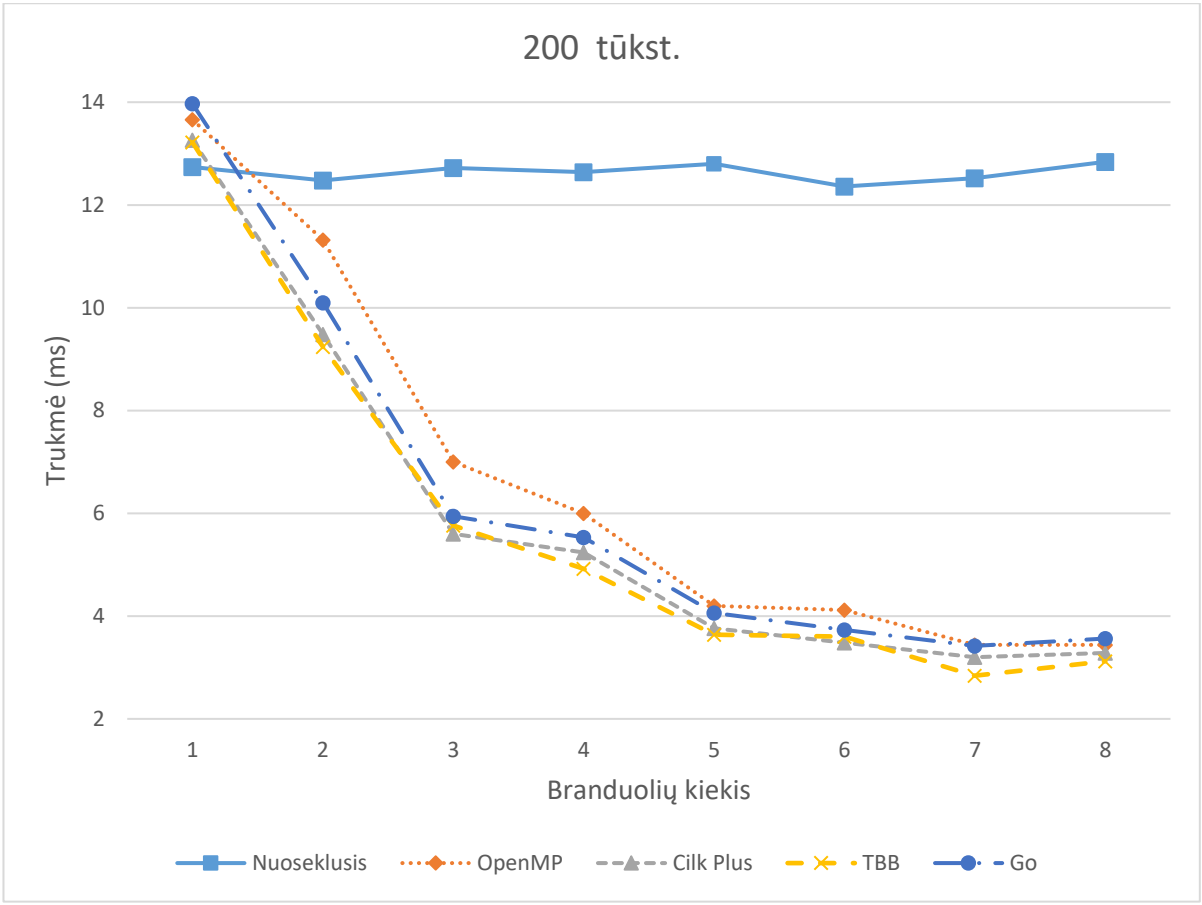
4 pried. 250 KB išilginio klaidų tikrinimo trukmės



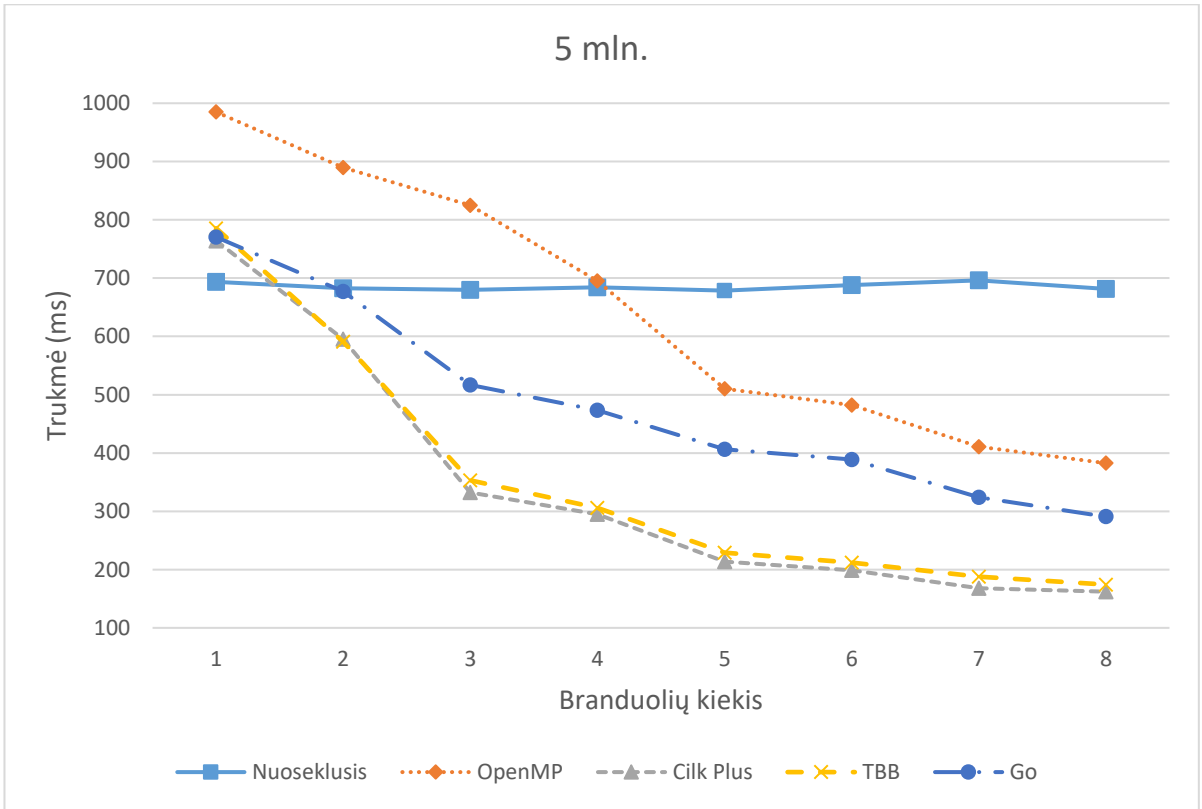
5 pried. 50 MB išilginio klaidų tikrinimo trukmės



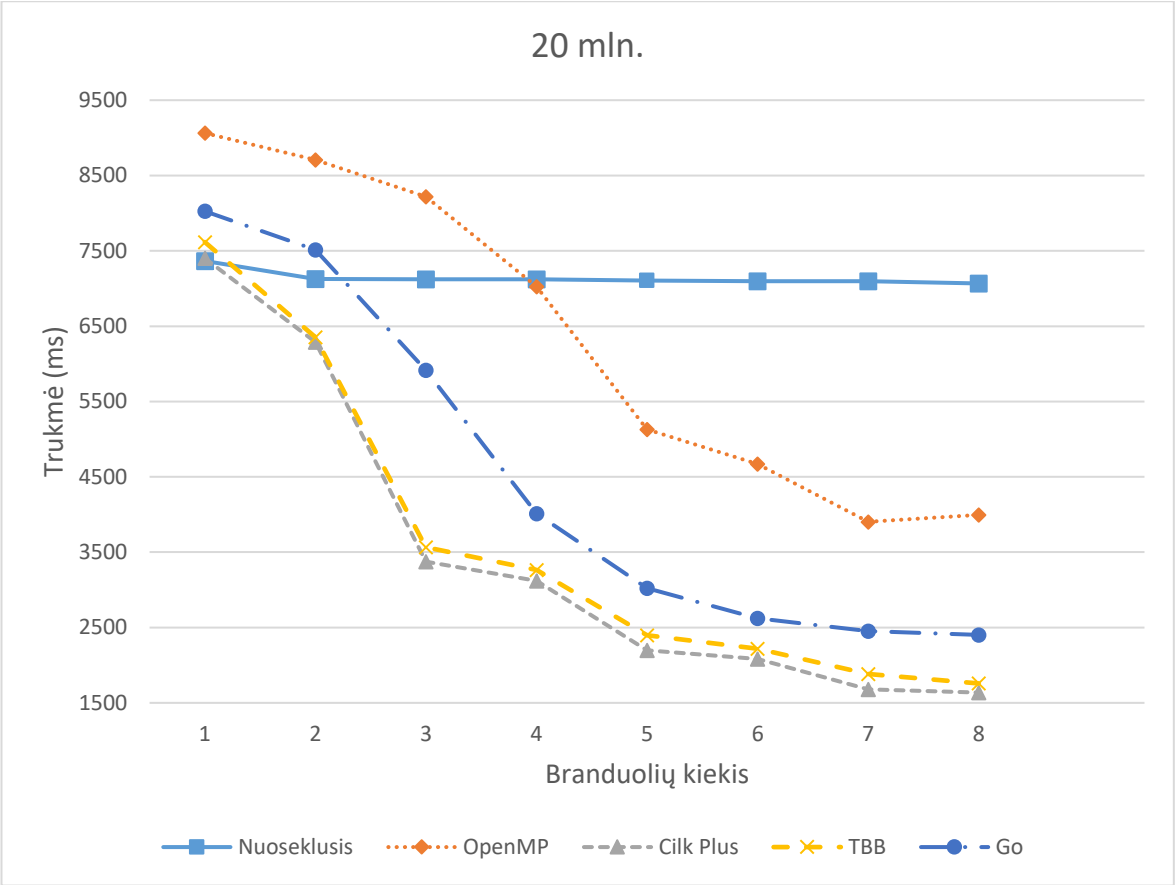
6 pried. 1250 MB išilginio klaidų tikrinimo trukmės



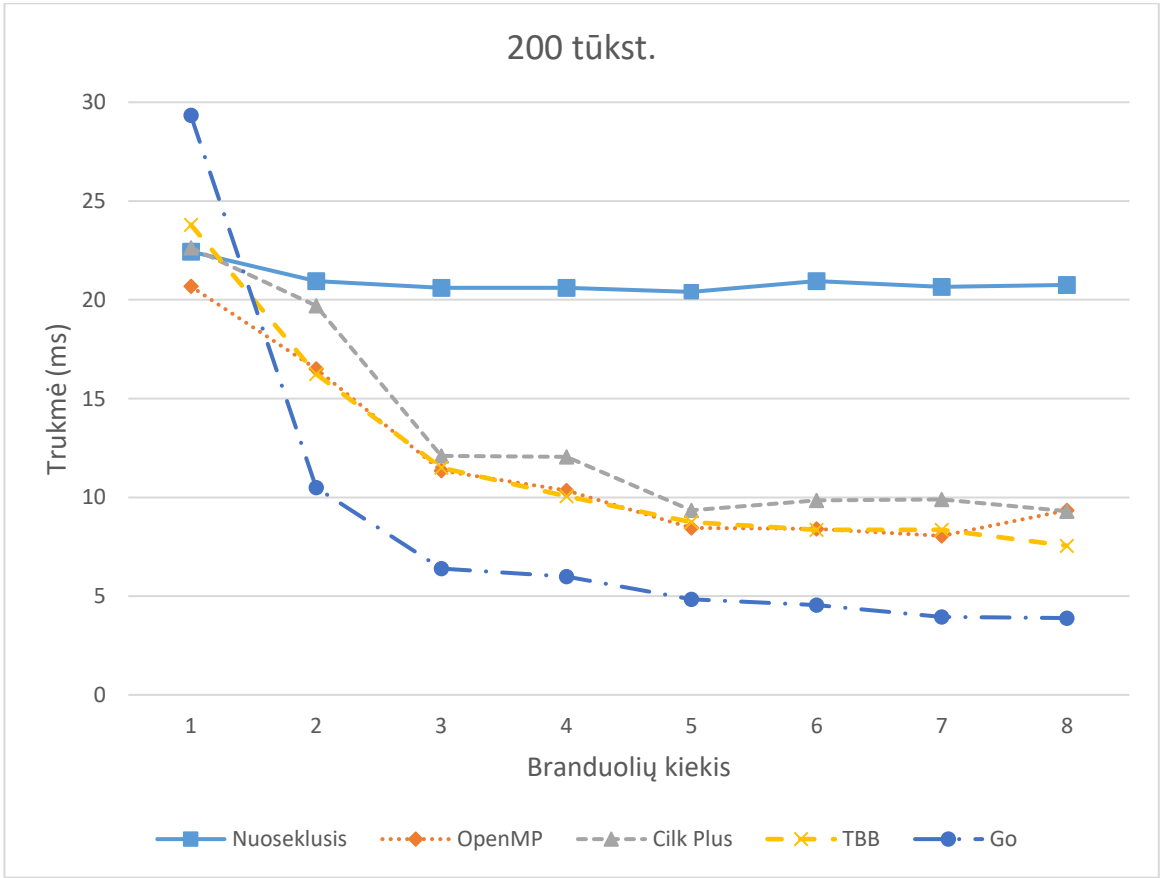
7 pried. 200 tūkst. elementų sąrašo quicksort rikiavimo trukmės



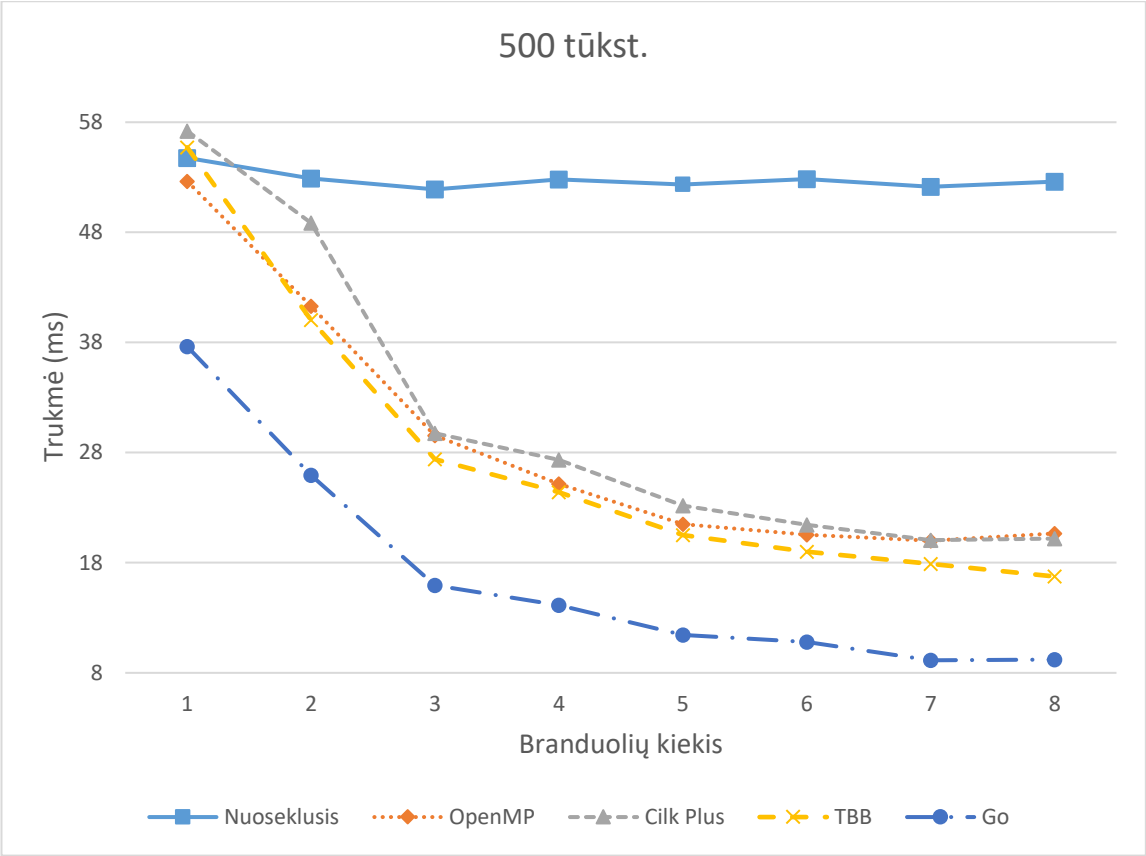
8 pried. 5 milijonų elementų sąrašo quicksort rikiavimo trukmės



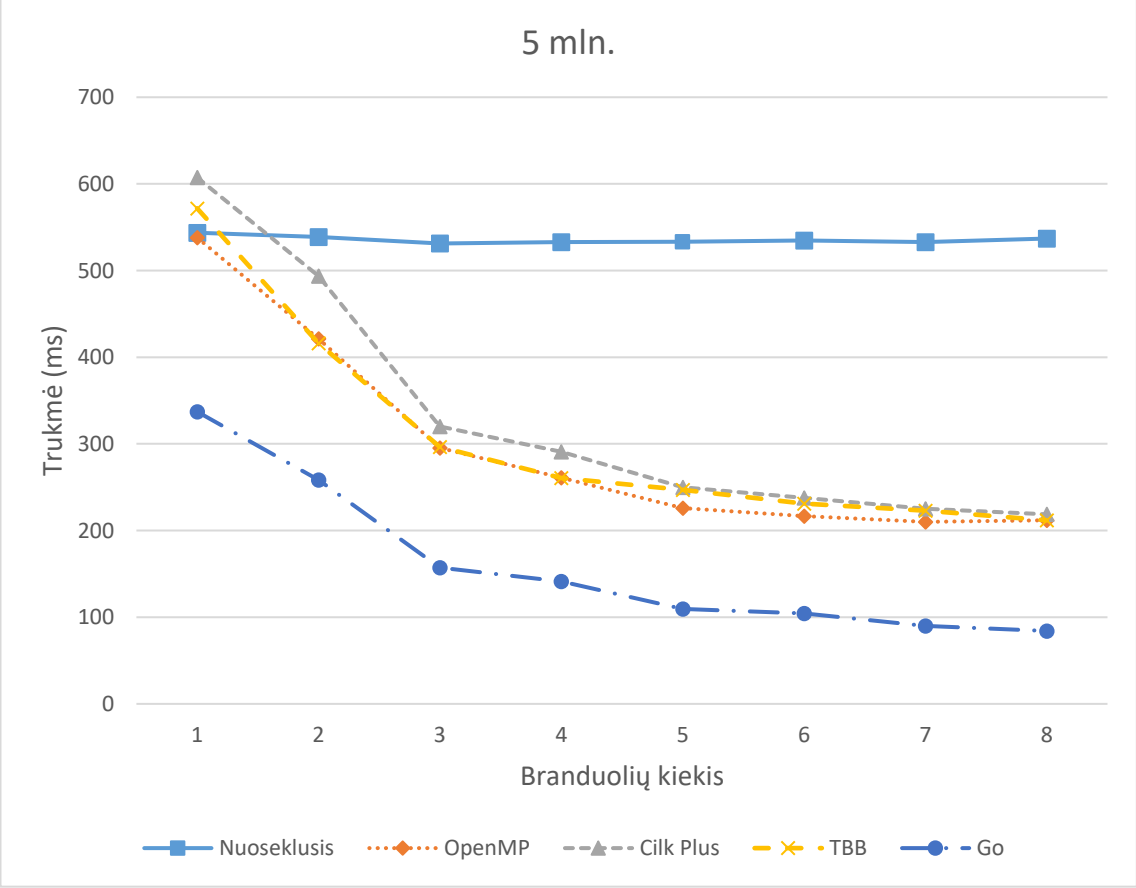
9 pried. 20 milijonų elementų sąrašo quicksort rikiavimo trukmės



10 pried. 200 K MapReduce skaičių skaičiavimo trukmės



11 pried. 500 K MapReduce skaičių skaičiavimo trukmės



12 pried. 5 000 K MapReduce skaičių skaičiavimo trukmės