

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Milda Marija Baravykaitė

LYGIAGREČIŲJŲ ALGORITMŲ ŠABLONŲ TYRIMAS IR KŪRIMAS

Daktaro disertacija
Technologijos mokslai, informatikos inžinerija – 07T

Vilnius, 2006

Disertacija rengta 2002 – 2006 metais Vilniaus Gedimino technikos universitete.

Darbo mokslinis vadovas:

prof. habil. dr. Raimondas ČIEGIS (Vilniaus Gedimino technikos universitetas,
technologijos mokslai, informatikos inžinerija – 07T).

Turiny

Ivadas	5
1 Šeimininkas – darbininkai algoritmo šablonas	21
1.1 Ivadas	21
1.2 Šeimininkas – darbininkai algoritmas	22
1.3 Šeimininkas – darbininkai algoritmo šablonas	25
1.3.1 Vartotojo programos lygmuo	26
1.3.2 Vartotojo programos sąsajos lygmuo	28
1.3.3 Šeimininkas – darbininkai algoritmo šablono logikos lygmuo	29
1.3.4 Duomenų perdavimo lygmuo	30
1.3.5 Šablono naudojimas	31
1.4 Šeimininkas – darbininkai algoritmo šablono taikymo pavyzdžiai	31
1.4.1 Pirminių skaičių radimo uždavinys	31
1.4.2 Šeimininkas – darbininkai algoritmo taikymas statinių pa- matų optimizavimui	35
1.5 Skyriaus rezultatai ir išvados	41
2 Šakų ir rėžių algoritmo šablonas	43
2.1 Ivadas	43
2.2 Nuoseklusis šakų ir rėžių algoritmas	45
2.3 Nuosekliojo šakų ir rėžių algoritmo šablonas	49
2.3.1 Uždavinių pavyzdžiai	50
2.4 Lygiagretieji šakų ir rėžių algoritmai	53
2.5 Lygiagrečiųjų šakų ir rėžių algoritmų šablonas	57
2.5.1 Skaičiavimo eksperimentų rezultatai	58
2.5.2 Dinaminis procesorių apkrovos balansavimas	65
2.5.3 Skaičiavimo eksperimentų rezultatai	67
2.6 Juodosios dėžės globaliosios optimizacijos algoritmas	70
2.6.1 Algoritmas	70

2.6.2	Algoritmo taikymas technologiniam uždaviniui	70
2.6.3	Siūlomas modifikuotas juodosios dėžės algoritmas	73
2.7	Skyriaus rezultatai ir išvados	74
Darbo rezultatų apibendrinimas ir išvados		76

Ivadas

Tyrimo objektas

Disertacijos tyrimo objektas – lygiagretieji algoritmai, lygiagretinimo priemonės, lygiagretinimo automatizavimas, algoritmų šablonai, optimizavimo algoritmai ir jų lygiagretinimas.

Temos aktualumas

Daugelyje mokslo sričių susiduriama su uždaviniais, kuriuos išspręsti esamais kompiuteriais truktų nepriimtina ilgai arba neįmanoma dėl tų uždavinių dydžio ir skaičiavimų apimtys. Lygiagrečiais skaičiavimais siejami tikslai – spręsti uždavinius greičiau, arba spręsti didesnius uždavinius. Tačiau tiek lygiagrečiųjų algoritmų sudarymas, tiek lygiagretusis programavimas – sudėtingi procesai, reikalaujantys daug žinių ir įgūdžių. Taupant laiką ir pastangas, kuriami lygiagretinimo įrankiai, lengvinantys įvairių lygiagrečiųjų programų ar metodikų naudojimą.

Šeimininkas – darbininkai algoritmai yra dažnai naudojami ir paplitę įvairiose lygiagrečiųjų skaičiavimų srityse. Lygiagrečiųjų *šeimininkas – darbininkai* algoritmų programų kūrimo automatizavimas palengvina daugelio uždavinių sprendimą. *Šeimininkas – darbininkai* algoritmas efektyvus vykdant tiek homogeniniuose, tiek heterogeniniuose lygiagrečiuosiuose kompiuteriuose. Duomenims tarp procesorių perduoti naudojamos duomenų perdavimo bibliotekos sparčiai vystomos. Nemažai anksčiau kurtų lygiagretinimo įrankių naudoja PVM (*Parallel Virtual Machine*) duomenų perdavimo biblioteką. Išpopuliarėjus MPI (*Message Passing Interface*) duomenų perdavimo bibliotekai aktualu pritaikyti įrankius jos naudojimui.

Daug praktinių uždavinių gali būti formuluojami kaip optimizavimo uždaviniai. Daugelis optimizavimo uždavinių sudėtingi ir reikalaujantys daug skaičiavimo. *Šakų ir rėžių* algoritmo principai taikomi tikslų ir euristinių optimizavimo uždavinių sprendinių radimui. Optimizavimo algoritmų lygiagretinimas taip pat yra sudėtingas uždavinys. Aktualu ruošti įrankius tiek lengvinančius optimizavimo algoritmų taikymą įvairiems uždaviniams, tiek algoritmų lygiagretinimui, tiek naujų algoritmų kūrimui ir testavimui.

Polinių statinių pamatai naudojami vietose, kur namai statomi ant silpnų pag-

rindų. Praktiniuose taikymuose aktualu pamatus suprojektuoti taupiai ir nepažeidžiant saugumo reikalavimų. Todėl polinių pamatų optimizavimo uždavinys yra aktualus, tačiau esant sudėtingoms apkrovoms ir pamatų geometrijoms sunkiai išsprendžiamas. Šio uždavinio sprendimui taikant lygiagretinimo priemones galima gauti sprendinius, artimus optimaliems.

Darbo tikslai

- Praplėsti *šeimininkas – darbininkai* algoritmo šablono funkcionalumą, pritaikant naudoti su įvairiomis duomenų perdavimo bibliotekomis.
- Suformuoti *šakų ir režijų* algoritmo įrankį, lengvinantį eksperimentavimą su nuosekliaisiais algoritmais ir lygiagrečiųjų *šakų ir režijų* algoritmų programų kūrimą.
- Lygiagrečiuosius optimizavimo algoritmus pritaikyti statybos technologijos uždaviniams.

Spręsti uždaviniai

- Suformuoti *šeimininkas – darbininkai* algoritmo šablono lygmenų struktūrą. Pritaikyti šabloną naudoti su MPI duomenų perdavimo biblioteka.
- Sukurti *šakų ir režijų* algoritmo šabloną, pritaikytą:
 - eksperimentavimui su nuosekliaisiais algoritmais;
 - automatiniam lygiagrečiųjų programų kūrimui;
 - naujų algoritmų, paremtų *šakų ir režijų* algoritmo logika, testavimui.
- Polinių statinių pamatų optimizavimui taikyti lygiagrečiuosius optimizavimo algoritmus, generuotus naudojant *šeimininkas – darbininkai* algoritmo šabloną ir lygiagrečiuosius algoritmus, paremtus *šakų ir režijų* algoritmo logika.

Tyrimo metodai

Tyrimų metodai apima algoritmų teoriją, lygiagrečiųjų algoritmų efektyvumo ir sudėtingumo analizę, eksperimentinį tyrimų metodą.

Naudotos C, C++ programavimo kalbos, objektinio programavimo technologija, algoritmų šablonai, PVM ir MPI duomenų perdavimo bibliotekos.

Mokslinis naujumas

Šeimininkas – darbininkai lygiagretinimo priemonė – algoritmo šablonas, suformuotas lygmenų struktūra, paruoštas naudoti su įvairiomis duomenų perdavimo bibliotekomis.

Sukurtas *šakų ir rėžių* algoritmo šablonas, leidžiantis eksperimentuoti su nuosekliaisiais algoritmais, generuoti lygiagrečiųjų algoritmų programas bei kurti naujus algoritmus, paremtus *šakų ir rėžių* algoritmo logika. Pasiūlytas modifikuotas juodosios dėžės globaliosios optimizacijos algoritmas.

Globaliosios optimizacijos lygiagretieji algoritmai taikyti polinių pamatų optimizavimo uždaviniams.

Gautų rezultatų reikšmė

Šeimininkas – darbininkai algoritmo šablono pritaikymas darbui su kitomis duomenų perdavimo bibliotekomis praplečia įrankio naudojimo lygiagretinimo automatizavimui galimybes. Įrankis testuotas homogeninėje ir heterogeninėje aplinkose, gautos efektyviai vykdomos lygiagrečiosios programos.

Globaliosios optimizacijos lygiagrečiųjų algoritmų efektyvumui daro įtaką daugelis veiksnių, automatiniai algoritmų lygiagretinimo įrankiai leidžia nesudėtingai eksperimentuoti ir parinkti tinkamus algoritmų variantus. Šablono rėmuose testuotas modifikuotas juodosios dėžės globaliosios optimizacijos algoritmas.

Polinių pamatų optimizavimo uždaviniams taikyti lygiagretieji globaliosios optimizacijos algoritmai ir gauti tinkami sprendiniai – beveik lygios reakcijos jėgos atramose.

Darbe teikiama:

- *Šeimininkas – darbininkai* algoritmo šablonas, pritaikytas naudoti su PVM ir MPI duomenų perdavimo bibliotekomis.
- Nuoseklusis ir lygiagretusis *šakų ir rėžių* algoritmų šablonas.
- Juodosios dėžės modifikuotas globaliosios optimizacijos algoritmas.

Pagrindiniai darbo rezultatai, skelbti publikacijose:

1. Baravykaitė M., Belevičius R., Čiegis R. One application of the parallelization tool of master – slave algorithms. *Informatica*, 13 (4), 2002, p. 1–12. ISSN 0868-4952 (ISI Master List).
2. Čiegis R., Baravykaitė M., Belevičius R. Parallel global optimization of foundation schemes in civil engineering. *Lecture Notes in Computer Science*, Vol 3732 2006, Applied Parallel Computing: 7th International Confe-

- rence, PARA 2004, Lyngby, Denmark, June 20-23, 2004. Revised Selected Papers. Editors: Jack Dongarra, Kaj Madsen, Jerzy Wasniewski, 2006, p. 305–313. ISSN 0302-9743 (ISI Master List).
3. Baravykaitė M., Šablinskas R. The template programming of parallel algorithms. *Mathematical Modelling and Analysis*, 7(1), 2002, p. 11–20. ISSN 1392-6292.
 4. Baravykaitė M., Čiegis R., Žilinskas J. Template realization of generalized branch and bound algorithm. *Mathematical Modelling and Analysis*, 10(3), 2005, p. 217–236. ISSN 1392-6292.
 5. Baravykaitė M. Lygiagrečiųjų šakų ir režijų algoritmų programų kūrimas. *Lietuvos matematikos rinkinys*, 44, spec. nr., 2004, p. 220–225. ISSN 0132-2818.
 6. Baravykaitė M. Procesorių apkrovos balansavimas lygiagrečiuose šakų ir režijų algoritmuose. *Lietuvos matematikos rinkinys*, 45, spec nr., 2005, p. 119–123. ISSN 0132-2818.
 7. Baravykaitė M. Parallel branch and bound algorithm template. In: Proceedings of 10th international conference Mathematical Modelling and Analysis, 2nd International Conference Computational Methods in Applied Mathematics, June 1-5, 2005, Trakai, Lithuania. Editor: R.Čiegis, 2005, p. 165–170. ISBN 9986-05-924-0.
 8. Baravykaitė M. Lygiagrečiųjų programų automatinio generavimo priemonė šeimininkas – darbininkai algoritmams. Iš: 5-osios Lietuvos jaunųjų mokslininkų konferencijos „Lietuva be mokslo – Lietuva be ateities“ medžiaga: Matematika ir informatika, 2002, p. 32–38. ISBN 9986-05-528-8.
 9. Baravykaitė M. Lygiagretieji šakų ir režijų algoritmai. Iš: 7-osios Lietuvos jaunųjų mokslininkų konferencijos „Lietuva be mokslo – Lietuva be ateities“ medžiaga: Matematika ir informatika, 2004, p. 51—57. ISBN 996-05-724-8.

Konferencijose skaityti pranešimai:

- R. Čiegis, M. Baravykaitė. Implementation of parallel generalized branch and bound template. Workshop on state-of-art in the scientific computing PARA06, Umea, Sweden, June 18-21, 2006.
- M. Baravykaitė, Load balancing for parallel branch and bound algorithms. *Mathematical Modelling and Analysis*, Jurmala, Latvia, June 1-4, 2006.

- M. Baravykaitė, J. Žilinskas. Implementation of parallel optimization algorithms using generalized branch and bound template. Optimal process design, International Networking for Young Scientists, Vilnius February 15-16, 2006.
- M. Baravykaitė. Apkrovimo balansavimas lygiagrečiuose šakų ir rėžių algoritmuose. XLVI Lietuvos matematikų draugijos konferencija, Vilnius, birželio 15-16, 2005.
- M. Baravykaitė. Parallel branch and bound algorithm template. Mathematical Modelling and Analysis, Trakai, June 1-5, 2005.
- R. Čiegis, R. Belevičius, M. Baravykaitė. Parallel global optimization of foundation schemes in civil engineering. Workshop on state-of-art in the scientific computing PARA04, Lyngby, Denmark, June 20-23, 2004.
- M. Baravykaitė. Lygiagrečiųjų šakų ir rėžių algoritmų programų kūrimas. XLV Lietuvos matematikų draugijos konferencija, Kaunas, birželio 17-18, 2004.
- M. Baravykaitė, R. Čiegis, G. Šilko, A. Jakušėvas. Parallelization tools and industrial applications. Mathematical Modelling and Analysis, Jurmala, Latvia, May 27-29, 2004.
- M. Baravykaitė. Lygiagretieji šakų ir rėžių algoritmai, 7-oji Lietuvos jaunųjų mokslininkų konferencija "Lietuva be mokslo – Lietuva be ateities", Vilnius, balandžio 7-8, 2004.
- M. Baravykaitė, R. Belevičius, R. Čiegis. Application of the parallelization tool of Master slave algorithm. Mathematical Modelling and Analysis, Trakai, May 28-31, 2003.
- M. Baravykaitė. Lygiagrečiųjų programų automatinio generavimo priemonė šeimininkas – darbininkai algoritams, 5-oji Lietuvos jaunųjų mokslininkų konferencija "Lietuva be mokslo – Lietuva be ateities", Vilnius, balandžio 16-17, 2002.

Disertacijos struktūra ir apimtis

Disertaciją sudaro įvadas, du skyriai, rezultatų apibendrinimas ir išvados, literatūros sąrašas.

1 skyriuje tiriamas *šeimininkas – darbininkai* algoritmas, šio algoritmo šablonas, jo naudojimas. Pirmajame poskyryje nagrinėjama *šeimininkas – darbininkai* algoritmo reikšmė, antrajame – algoritmas ir jo modifikacijos, trečiajame poskyryje aprašomas *šeimininkas – darbininkai* algoritmo šablonas, jo lygmenų

struktūra. Ketvirtajame poskyryje pateikti algoritmo šablono taikymai pirminių skaičių radimo ir statinių pamatų optimizavimo uždaviniams. Penktajame poskyryje pateikti skyriaus rezultatų apibendrinimai ir išvados.

2 skyriuje nagrinėjami nuoseklūs ir lygiagretieji *šakų ir rėžių* algoritmai. Įvade aptariamas *šakų ir rėžių* algoritmo taikymas, kiti *šakų ir rėžių* algoritmų šablonai. Antrajame poskyryje nagrinėjamas nuoseklusis apibendrintas *šakų ir rėžių* algoritmas, trečiajame poskyryje tiriamas nuosekliojo algoritmo šablonas, pateikiami taikymo kombinatorikos ir globaliosios optimizacijos uždaviniams pavyzdžiai. Ketvirtajame poskyryje aprašomos *šakų ir rėžių* algoritmo lygiagretinimo galimybės. Penktajame poskyryje tiriamas lygiagretusis šablonas, pateikiami skaičiavimo eksperimentų rezultatai, aprašomas dinaminės apkrovos balansavimo modulis. Šeštajame poskyryje nagrinėjamas *šakų ir rėžių* algoritmo principais pagrįstas globaliosios optimizacijos juodosios dėžės algoritmas, jo taikymas statybinės mechanikos uždaviniams, pasiūlytas modifikuotas globaliosios optimizacijos juodosios dėžės algoritmas. Skyriaus pabaigoje pateikiami rezultatų apibendrinimai ir siūlymai.

Darbo rezultatuose ir išvadose pateikiami svarbiausi tyrimų rezultatai.

Literatūros apžvalga

Lygiagrečiųjų algoritmų svarba. Lygiagrečiuoju programavimu siekiame dviejų tikslų: sprendžiami didesni uždaviniai ir/ar uždaviniai sprendžiami greičiau. Abu tikslai yra aktualūs praktiniuose taikymuose. Tačiau lygiagretusis programavimas reikalauja nemažai papildomų žinių, lyginant su nuosekliuoju programavimu.

Lygiagrečiojo algoritmo sudarymas. Bet kuris lygiagretusis algoritmas duotą uždavinį suskaido į smulkesnius uždavinius, kurie gali būti sprendžiami lygiagrečiai, vienu metu skirtinguose procesoriuose.

Kuriant lygiagretųjį algoritmą vykdomi keturi svarbiausi žingsniai [26]:

- Pradinio uždavinio *skaidymas į užduotis* (angl. *partitioning*), kurias galima spręsti lygiagrečiai, vienu metu skirtinguose procesoriuose. Šiame etape siekiama atskleisti algoritmo lygiagretinimo galimybes.
- *Komunikacijų* tarp procesorių nustatymas (angl. *communication*). Šiame etape apibrėžiami skaičiavimams būtini duomenų perdavimai tarp procesorių. Kuo daugiau užduočių sudaroma, tuo daugiau procesorių galima naudoti bei tolygiau paskirstyti apkrovą naudojamiems procesoriams. Ne visos užduotys būna žinomos uždavinio sprendimo pradžioje, jos gali būti dinamiškai generuojamos uždavinio sprendimo metu.

Taip pat kai kurioms užduotims gali būti svarbi jų vykdymo tvarka. Tokiu

atveju reikia užduotis topologiškai surūšiuoti. Tai pirmasis užduočių sprendimo tvarkaraščio sudarymo žingsnis.

- *Užduočių apjungimas* (angl. *agglomeration*). Po to, kai ištiriama galimybė algoritmo užduotis vykdyti lygiagrečiai ir įvertinami tam būtini duomenų perdavimai, ieškoma galimybių gerinti lygiagretųjį algoritmą mažinant duomenų perdavimus apjungiant užduotis į didesnes. Duomenų perdavimai tarp procesorių yra lėtesni lyginant su skaičiavimo operacijų atlikimo greičiu. Duomenų perdavimo greitis priklauso nuo lygiagrečiojo kompiuterio tipo. Užduočių apjungimo žingsnis nėra būtinas, jei algoritmo specifiška yra tokia, kad duomenų perdavimų nėra daug ir jie nežymiai mažina algoritmo efektyvumą.
- *Priskyrimas* (angl. *mapping*). Paskutiniame lygiagrečiojo algoritmo sudarymo etape užduotys yra priskiriamos konkretiems procesoriams, t.y. sudaromas užduočių sprendimo tvarkaraštis.

Tokie algoritmo sudarymo etapai leidžia įgyvendinti daug lygiagrečiojo vykdymo galimybių. Lygiagretieji skaičiavimai ir programos reikalauja iš kūrėjo papildomų žinių ir algoritmo veikimo įvertinimo.

Realizuojant pagrindinius lygiagrečiųjų algoritmų kūrimo žingsnius, lygiagrečias programas galima kurti naudojant kelias strategijas [53].

- *Detalus* (angl. *from scratch*) – programų kūrėjas nurodo žemo lygio instrukcijas, kokias užduotis, kada ir kur vykdyti lygiagrečiai, kada, kaip ir kokius duomenis perduoti tarp procesorių. Kūrėjas turi pats realizuoti visus svarbiausius programos kūrimo žingsnius. Tokiu būdu gaunamos efektyvios programos, nes atsižvelgiama į uždavinio specifišką, tačiau tokioms programoms paruošti reikia sąlyginai daug darbo.
- Tam tikriems lygiagretinimo veiksams naudojame specialias bibliotekas, pavyzdžiui MPI [55] ar PVM [30] bibliotekas duomenims tarp procesorių perduoti. Kūrėjas vis tiek turi įdėti daug pastangų programos kūrimui: aprašyti uždavinio dalinimą, reikiamas komunikacijas, užduočių apjungimą ir priskyrimą. Specifiniai veiksmai, tokie kaip fizinis duomenų perdavimas ar procesorių topologijos įvertinimas, atliekami naudojantis bibliotekomis. Bibliotekos būna paruoštos įvairioms kompiuterių architektūroms, sistemoms, jas galima naudoti su įvairiomis programavimo kalbomis.
- Pusiau automatiškai, kai užtenka nurodyti prašomą informaciją, bet nereikia atlikti konkrečių lygiagretinimo veiksmų. Tokiam programų kūrimui naudojami įrankiai, apibendrinantys dažnai pasitaikančius lygiagrečiųjų skaičiavimų algoritmus, pavyzdžiui, Charm [38], HPC [37].

- Automatinio lygiagretinimo priemonėmis, kai kūrėjui nereikia nurodyti jokios papildomos informacijos, norint gauti lygiagrečiąją programą. Šis metodas paprastai taikomas realizuojant konkrečius lygiagrečiuosius algoritmus, pavyzdžiui, VAST [61], OCREP [45].

Automatinis ar pusiau automatinis lygiagrečiųjų programų kūrimas nėra griežtai atskirti. Kuo daugiau veiksmų turi atlikti ir kuo daugiau informacijos turi suteikti kūrėjas, tuo efektyvesnis tokių programų vykdymas, nes kūrėjas geriau įvertina algoritmo lygiagretinimo galimybes. Tačiau tuo pačiu reikalaujama daugiau lygiarečiojo programavimo įgūdžių ir daugiau laiko, lyginant su automatinio ar pusiau automatinio lygiagrečiųjų programų kūrimu. Naudojant automatinį ar pusiau automatinį lygiagretinimą taupomas laikas ir pastangos, mažinama klaidų tikimybė, tačiau dėl algoritmų apibendrinimų prarandamas efektyvumas.

Lygiagrečiai gali būti vykdomos įvairios programų dalys [53]:

- *Instrukcijos.* Tai labai smulkių dalių vykdymas lygiagrečiai. Paprastai tokio lygio lygiagretumą įgyvendina procesorius.
- *Ciklai, instrukcijų blokai.* Tai smulkių programos dalių vykdymas lygiagrečiai. Tokio tipo lygiagretumą įgyvendina kompiliatoriai (pvz. HPF). Esminis kompiliatorių trūkumas yra tai, kad jie neuniversalūs procesorių atžvilgiu, kiekvienai sistemai kuriamas specifinis kompiliatorius.
- *Funkcijos.* Tai vidutinio dydžio programų dalių lygiagretumo įgyvendinimas. Paprastai šį lygį įgyvendina programuotojas. Vidutinio dydžio dalių lygiagretinimui naudojamos tiek specialios lygiagreta programavimo bibliotekos, tiek ir lygiagretinimo įrankiai.
- *Programos.* Tai yra stambių uždavinio sprendimo algoritmo dalių vykdymas lygiagrečiai. Toks būdas naudojamas specifiniuose taikymuose.

Lygiagrečiųjų skaičiavimų modeliai. Aprašant lygiagrečiųjų skaičiavimų galimybes remiamasi Flyno klasifikacija [25].

- SISD yra skaičiavimo modelis, kai viena instrukcija vykdoma su vienu duomenų bloku. Šis modelis apibrėžia nuoseklius skaičiavimus.
- SIMD yra modelis, kai viena instrukcija vykdoma su daug duomenų blokų. Tai dažnai naudojamas lygiagrečiųjų skaičiavimų duomenų lygiagretumo paradigmos skaičiavimo modelis.
- MISD yra modelis, kai daug instrukcijų vykdoma su vienu duomenų bloku lygiagrečiai.

- MIMD yra skaičiavimo modelis, kai daug instrukcijų vykdoma su daug duomenų blokų vienu metu. Tai plačiai taikomas funkcinio lygiagretumo paradigmos skaičiavimo modelis.

Lygiagretieji kompiuteriai. Lygiagretieji kompiuteriai, kuriuose bus vykdomos programos stipriai daro įtaką tiek lygiagrečiųjų algoritmų, tiek programų kūrimui. Lygiagretieji kompiuteriai skirstomi į tokias grupes:

- vektorinius (ang. *vector*),
- matricinius (ang. *matrix*),
- bendrosios atminties (ang. *shared memory*),
- paskirstytos atminties (ang. *distributed*).

Skiriasi tokių kompiuterių atminties paskirstymo principai, o tai lemia duomenų perdavimo tarp procesorių mechanizmą. Superkompiuteriai ir kompiuterių klasteriai skiriasi duomenų perdavimo greičiais ir procesorių homogeniškumu. Paskirstytos atminties kompiuterių klasei priklauso ne tik specialūs kompiuteriai, bet ir nuoseklieji kompiuteriai, sujungti į vieną sistemą. Tokiu būdu gaunamas heterogeninis lygiagretusis kompiuteris, kurio procesoriai yra ne vienodi. Jei visi lygiagrečiojo kompiuterio procesoriai vienodi, toks kompiuteris vadinamas homogeniniu.

Sparčiai vystosi geografiškai paskirstyta skaičiavimo platforma, sudaryta iš didelio skaičiaus heterogeninių kompiuterių, prie kurių vartotojai jungiasi per tą pačią sąsają – GRID tinklynas [27, 44, 43]. GRID tinklynai plačiai naudojami didelės apimties skaičiavimams atlikti. Dėl geografinio paskirstymo, atsiranda skaičiavimų specifika, pavyzdžiui nėra garantuojama, kad užduotys, paskirtos procesoriams, bus įvykdytos. Dėl to reikia kartoti užduotis, tikrinti gautojo rezultato teisingumą. Specifiką įtakoja ir heterogeninė aplinka bei santykinai lėtas ryšys tarp naudojamų procesorių.

Lygiagrečiųjų algoritmų analizė. Lygiagrečiųjų programų ir algoritmų efektyvumą įvertiname spartinimo ir efektyvumo koeficientais [26]. Spartinimo koeficientu vadinamas santykis

$$S_p = \frac{T_0}{T_p}, \quad (1)$$

kur T_0 - greičiausio nuosekliojo algoritmo vykdymo laikas, T_p - lygiagrečiojo algoritmo vykdymo laikas, naudojant p procesorių. Šis dydis parodo pagreitėjimą, kuris pasiekiamas vykdant lygiagretųjį algoritmą naudojant p procesorių. Idealiu atveju S_p gali didėti tiesiškai. Šis koeficientas skaičiuojamas ir lygiagrečiųjų programų įvertinimui, tada naudojamas ilgiausiai užtrukusio procesoriaus programos vykdymo laikas.

Lygiagrečiojo algoritmo efektyvumo koeficientas

$$E_p = \frac{S_p}{p}, \quad (2)$$

parodo, kaip efektyviai naudojami p procesoriai. Idealiu atveju E_p artėja prie vieneto. Praktiškai, tai sunkiai pasiekama dėl papildomų lygiagrečiųjų programų vykdymo kaštų, pavyzdžiui, lygiagrečiųjų skaičiavimų inicializavimo, duomenų tarp procesorių perdavimo, netolygaus užduočių paskirstymo.

Algoritmo izoefektyvumo funkcija susieja papildomus lygiagrečiųjų skaičiavimų kaštus ir naudojamą procesorių skaičių. Ji naudojama tiriant algoritmo plečiamumą ir parodo, kaip turi didėti uždavins, kad taip pat efektyviai naudotume vis daugiau procesorių [12].

Spartinimo koeficiento skaičiavimui naudojamas ilgiausiai užtrukusio procesoriaus darbo laikas, todėl lygiagrečiųjų skaičiavimų efektyvumą stipriai įtakoja užduočių paskirstymo tolygumas – procesorių apkrovos balansas. Apkrovos balansavimui skiriama daug dėmesio paskirstytos atminties lygiagrečiuosiuose kompiuteriuose. Apkrovos balansavimas – procesas, kurio metu sprendžiama, kaip ir kokiems procesoriams paskirstyti procesus, kad visas uždavins būtų išspręstas greičiau, efektyviau ar taupiau [63]. Dažniausiai stengiamasi sutrumpinti skaičiavimo laiką. Teoriškai siekiama, kad uždavinio vykdymo metu procesorių apkrova būtų vienoda. Vykdam optimizavimo algoritmus praktiškai dažnai pakanka, kad nebūtų nevykdančių užduočių procesorių tuo metu, kai kiti yra stipriai apkrauti. Jei visos užduotys žinomos prieš pradedant spręsti uždavinį ir naudojama duomenų lygiagretumo paradigma, tai procesorių apkrova tiesiogiai proporcinga jiems skirtam duomenų kiekiui. Tačiau tai gali būti neefektyvu kitoms lygiagretumo paradigmoms, ar kai užduotys generuojamos dinamiškai programos vykdymo metu.

Apkrovos balansavimas skirstomas į tokias rūšis [65]:

- *Statiškas balansavimas* – procesorių apkrova balansuojama prieš pradedant vykdyti algoritmą. Statiškas apkrovos balansavimas naudojamas tuo atveju, kai galima iš anksto numatyti ir apskaičiuoti procesorių apkrovą ir su tuo susijusį duomenų kiekį. Tai sunkiai pasiekama NP sudėtingumo uždaviniais. Vykdam šį apkrovos balansavimą, sprendžiamas optimizavimo uždavinys ir sudaromas apkrovos paskirstymo algoritmas. Tokiu būdu gaunamas optimalus užduočių paskirstymas. Dažnai pasitelkiamos euristikos, kurios reikalauja mažiau skaičiavimų ir konstruoja beveik optimalius apkrovos paskirstymus. Stengiamasi supaprastinti šį uždavinį, kad skaičiavimo laikas būtų kuo trumpesnis. Paprasčiausias tokio tipo balansavimas taikomas duomenų lygiagretumo paradigmoje – pradinių duomenų tolygus dalinimas procesoriams.
- *Dinaminis balansavimas* – procesorių apkrova balansuojama algoritmo vykdymo metu. Dinaminis balansavimas naudojamas tada, kai tik vykdymo

metu galima įvertinti procesorių apkrovą ir balansavimo reikalingumą. Užduotys perduodamos iš labiau apkrauto procesoriaus mažiau apkrautiems. Dinaminis apkrovos balansavimas lankstesnis ir plačiau taikomas už statinį apkrovos balansavimą, tačiau jam įgyvendinti reikalingi sudėtingesni algoritmai. Algoritmo vykdymo metu atliekami didesni papildomi skaičiavimai ir duomenų perdavimai.

Toliau plačiau panagrinėsime dinaminį apkrovos balansavimą. Jo įgyvendinimui būtini šie žingsniai [65]:

1. Apkrovos matavimas procesoriuose.
2. Apkrovos informacijos pasikeitimo inicijavimas.
3. Balansavimo inicijavimas.
4. Balansavimo operacija.

Naudojant dinaminį apkrovos balansavimą, skaičiavimų pabaigos nustatymas tampa sudėtingesnis ir aptariamas kartu su dinaminio apkrovos balansavimu.

Balansavimo operacija remiasi apkrovos informacija procesoriuose. Todėl *apkrovos matavimas* yra labai svarbus. Apkrovos matas yra neneigiama reikšmė, kuri lygi nuliui, kai procesorius nėra apkrautas ir kuri didėja, kai didėja procesoriaus apkrova. Apkrova skaičiuojama dažnai, todėl apkrovos mato skaičiavimai turi būti santykinai trumpi. Kita vertus šis matas turi gerai atspindėti procesoriaus apkrovą.

Apkrovos informacijos apsikeitimo inicijavimas nusako, kada procesai keičiasi apkrovos informacija, sprendžia ar reikia pradėti apkrovos balansavimo operaciją. Efektyvu būtų visada turėti naujausią informaciją apie procesorių apkrovą. Tačiau apkrovos informacijos apsikeitimas yra susijęs su papildomais duomenų perdavimais tarp procesorių, kurie mažina lygiagrečiojo algoritmo efektyvumą. Kita vertus, reikia garantuoti, kad reikiamu momentu, procesoriai turės adekvatų kitų procesorių apkrovos vaizdą, tik tada priimti balansavimo sprendimai bus efektyvūs. Apkrovos informacijos pasikeitimą galima inicijuoti [65] įvairiais momentais:

- Pagal pareikalavimą, t.y. procesoriai apsikeičia apkrovos informacija, kai nustatoma, kad reikalingas apkrovos balansavimas ir reikia tirti kitų procesorių apkrovą. Tokios strategijos privalumas, kad apkrovos informacija keičiamasi retai.
- Periodiškai – informacijos apsikeitimas vyksta periodiškai, nepriklausomai, nuo to, ar gauta informacija yra naudojama. Šiuo atveju apkrovos informacija keičiamasi dažniai.

- Pasikeitus apkrovos būsenai, t.y. procesoriai apsikeičia informacija, kai jų apkrovos būsena pasikeičia nustatytu dydžiu arba viršija nustatytas ribas. Tokio informacijos pasikeitimo dažnumas yra tarpinis tarp periodinio ir pagal pareikalavimą apkrovos pasikeitimo inicijavimo taisyklių.

Balansavimo inicijavimo žingsnis apima gautos apkrovos informacijos apdorojimą ir balansavimo partnerių pasirinkimą. Dinaminis apkrovos balansavimas – sudėtinga procedūra, reikalaujanti papildomų skaičiavimų ir duomenų perdavimų, dėl to ne visada verta ją atlikti. Tegul T_1 – laikas, kurį užtruktų labiausiai apkrautas procesorius vykdydamas užduotis, nenaudodamas balansavimo; T_2 – laikas, kurį užtruktų labiausiai apkrautas procesorius vykdydamas užduotis, naudodamas balansavimą; T_3 – užduočių paruošimo perdavimui laikas, T_4 – užduočių perdavimo laikas. Balansavimą verta vykdyti, kai

$$T_1 \geq T_2 + T_3 + T_4. \quad (3)$$

Balansavimo inicijavimas skirstomas į tokias grupes [65]:

1. Siuntėjo inicijuojamą, kai balansavimą inicijuoja procesorius, kurio apkrova per didelė lyginant su kitais.
2. Gavėjo inicijuojamą, kai balansavimą inicijuoja procesorius, kurio apkrova per maža lyginant su kitais procesoriais.
3. Simetrinį, apjungiantį siuntėjo ir gavėjo inicijuojamus balansavimo algoritmus. Balansavimo operaciją gali inicijuoti tiek per daug, tiek ir per mažai apkrauti procesoriai.
4. Periodinį, kai balansavimo operacija vykdoma periodiškai.

Siuntėjo ir gavėjo inicijuojamiems balansavimams, reikia nustatyti „kritinę“ procesorių apkrovos būseną, t.y. apatinius ir viršutinius leistinus apkrovos režius. Siuntėjo inicijuoti algoritmai efektyvesni mažai apkrautose sistemose, o gavėjo inicijuoti – stipriau apkrautose sistemose [63]. Simetrinis inicijavimas tinkamas vykdant lygiagrečiuosius optimizavimo algoritmus. Periodinis balansavimo inicijavimas artimas apkrovos perskirstymo metodams, taikomiems skaičiavimams naudojant duomenų lygiagretumo paradigmą.

Priėmus *balansavimo inicijavimo* sprendimą, pradedama vykdyti *balansavimo operacija*, kuri apibrėžiama tokiomis svarbiausiomis taisyklėmis [65]:

- Balansavimo partnerio pasirinkimo taisykle, kuria pasirenkami procesoriai, su kuriais vyks balansavimo operacija.
- Užduočių perskirstymo taisykle, kuri nurodo, kaip paskirstyti perduodamas užduotis tarp pasirinktų procesorių. Pavyzdžiui, nurodoma santykinė dalis užduočių, kurios turi būti perduotos kitiems procesoriams.

- Perskirstomų užduočių pasirinkimo taisykle, kuri nusako, kurias užduotis perskirstyti.
- Užduočių perdavimo taisykle, t.y. pasirinktų užduočių persiuntimas pasirinktiems procesoriams.

Balansavimo operacijos taisyklės priklauso tiek nuo sprendžiamo uždavinio, tiek ir nuo lygiagrečiojo kompiuterio, kuriame bus vykdoma programa. Šios taisyklės apibrėžia daug balansavimo galimybių. Kai nėra siekiama suvienodinti procesorių apkrovą, o tik norima garantuoti, kad procesoriai neliktų be darbo, kai kiti procesoriai dar turi daug užduočių, balansavimo taisyklės supaprastėja.

Pagal tai, kaip ir kokiuose procesoriuose vykdomi dinaminio balansavimo žingsniai, algoritmus klasifikuojame į tokias grupes:

- Centralizuotus (globaliuosius), kai visus balansavimo sprendimus priima vienas procesorius.
- Lokaliuosius, kai balansavimo sprendimų priėmimas paskirstomas procesoriams.
- Kombinuotus, kai procesoriai paskirstomi grupėmis, kurių viduje vykdomas globalusis, o tarp grupių – lokalusis balansavimo algoritmas.

Balansavimo algoritmuose siekiame kompromiso tarp papildomų algoritmo kaštų, atsirandančių dėl duomenų perdavimų ir balansavimo skaičiavimų, ir turimos informacijos apie sistemos apkrovimą tikslumo. Centralizuoti arba globalieji algoritmai suteikia tiksliausią informaciją apie sistemos apkrovą, tačiau centralizuotas sprendimų priėmimas riboja naudojamų procesorių skaičių. Lokalieji balansavimo algoritmai renka apkrovos informaciją tik iš kaimyninių procesorių, todėl pilnas procesorių apkrovos balansas gali būti pasiektas tik iteracijomis. Tačiau praktiškai jie populiarūs, nes dažnai užtenka tik pakankamai gero apkrovos balansavimo. Kombinuoti algoritmai apjungia globaliųjų ir lokaliųjų apkrovos balansavimo algoritmų privalumus, tačiau juos sudėtinga realizuoti.

Lokalieji apkrovos balansavimo algoritmai gali remtis įvairiais principais. [66] darbe pateikiama tokia algoritmų klasifikacija:

1. Deterministiniai algoritmai
 - (a) difuziniai,
 - (b) dimensijos pasikeitimo,
 - (c) gradientiniai.
2. Stochastiniai algoritmai
 - (a) atsitiktinio priskyrimo,

(b) paremti fiziniais modeliais.

Deterministiniai ar stochastiniai algoritmai skirstomi pagal užduočių perskirstymo taisyklę. Deterministiniams metodams pagal apibrėžtus parametrus ir taisykles nustatoma, kuris kaimynas ir kiek užduočių perduos. Stochastiniuose metoduose įvedamas atsitiktinis parametras, kad apkrova būtų geriau paskirstyta.

Difuziniai ir dimensijos pasikeitimo metodai yra panašūs: jie abu tiria visų artimiausių kaimynų apkrovą kiekviename iteracijos žingsnyje. Difuziniuose metoduose apkrova balansuojama su visais kaimynais. Procesorius gali perduoti dalį užduočių vienam ar keliems kaimynams ir tuo pačiu metu gauti užduočių iš kitų kaimynų. Difuzinių metodų aprašymui naudojamos difuzijos diferencialinės lygtys [18]. Dimensijos pasikeitimo metoduose viename iteracijos žingsnyje apkrova balansuojama su vienu kaimynu. Gradientiniuose metoduose apkrova visada keičia mažiausiai apkrauto procesoriaus kryptimi.

Stochastinio dinaminio balansavimo atsitiktinio priskyrimo metode užduotis atsitiktinai priskiriama kuriam nors, dažniausiai kaimyniniam, procesui. Šis procesas, gavęs naują užduotį, gali ją spręsti ar perduoti kitam irgi atsitiktinai pasirinktam procesui. Kitas būdas yra fiziniais modeliais paremtas balansavimas, kai apkrova balansuojama remiantis fizinių procesų modeliais. Pavyzdžiui naudojamas fiktyvios temperatūros pasiskirstymo algoritmas (angl. *simulated annealing*) [65].

Taikant dinaminį apkrovos balansavimo algoritmą tampa sudėtinga nustatyti skaičiavimų pabaigą. Kai naudojamas dinaminis apkrovos balansavimas, skaičiavimų pabaiga nustatoma pagal tokius kriterijus [64]:

1. Visi procesoriai baigė jiems skirtą darbą – pasiekta lokali pabaigos sąlyga.
2. Nėra tarp procesorių perduodamų duomenų.

Pirmoji sąlyga nesunkiai nustatoma, jos realizavimas priklauso nuo sprendžiamo uždavinio.

Antrosios sąlygos nustatymas yra sudėtingesnis. Globaliesiems apkrovos balansavimo algoritmams pabaiga nustatoma, kai šeimininko procesas nebeturi nepaskirstytų užduočių o visi darbininkų procesai paprašo darbo. Lokaliesiems dinaminiam apkrovos balansavimo algoritmams siūloma keletas skaičiavimo pabaigos nustatymo algoritmų:

- patvirtinimo pranešimų (angl. *message acknowledgements*),
- žiedo,
- dvigubo žiedo,
- medžio,

- pastovios energijos.

Šie algoritmai išsamiai aprašyti [64].

Apkrovos balansavimo procesui automatizuoti kuriami specialūs įrankiai skirti apkrovos balansavimui: Athapascan [2], Zoltan[7], DLB [46].

Lygiagrečiųjų algoritmų šablonai. Algoritmų šablonai (angl. *algorithm templates*) apibrėžti apie 1990-uosius metus [33], neprarado aktualumo ir dabar. Algoritmų šablonais atskiriamos pastovioji algoritmo dalis ir kintanti, nuo uždavinio priklausanti sprendimo dalis. Algoritmų šablonai realizuojami programiškai ir apjungiami į įrankius. Bendruoju atveju algoritmų šablonai taikomi tiek nuosekliams, tiek lygiagretiesiems algoritmams. Lygiagrečiųjų algoritmų šablonai turėtų apibrėžti visus keturis lygiagrečiųjų algoritmų kūrimo žingsnius [33]. Šablonai gali būti sukurti įvairiems lygiagretiesiems kompiuteriams, operacinėms sistemoms, lygiagretumo tipams ar paradigmoms. Kitaip negu tradiciniame programavime, naudojant šablonus vartotojui nereikia mokėti naudotis lygiagrečiojo programavimo priemonėmis. Jam užteka atpažinti savo programos tipą, pasirinkti tinkamą šablono ir pagal šablono reikalavimus į nurodytas vietas įterpti savo programos dalis.

Reikalavimai šablonui [54]:

1. *Specifikos atskyrimas.* Uždavinio kintančios dalies atskyrimas nuo šablono.
2. *Hierarchiškumas.* Šablonai turi turėti galimybę naudoti kitus šablonus.
3. *Nepriklausomumas nuo platformos.* Šablono galima vykdyti įvairiuose kompiuteriuose.
4. *Plečiamumas, jungimas į rinkinius.* Šablonus galima praplėsti naujais algoritmais ar dalimis, juos galima jungti į rinkinius.
5. *Atviros sistemos koncepcija.* Vartotojui turi būti palikta galimybė pačiam modifikuoti, papildyti šablono.
6. *Programų korektiškumas.* Garantuojama, kad šablone įdiegti algoritmai veikia korektiškai.
7. *Esamų programavimo kalbų naudojimas.* Šablonai turi remtis paplitusiomis programavimo kalbomis ar priemonėmis.
8. *Efektyvumas.* Garantuojama, kad programos efektyviai realizuoja algoritmus.
9. *Palaikymo programos ir naudojumas.* Šablonai turėtų vystytis ir plėstis, siekiama, kad jie būtų populiarūs ir naudojami.

Eksperimentinė aplinka. Disertacijoje pateikti eksperimentai buvo atlikti naudojant:

- IBM RS/6000 septynių kompiuterių (200 MHz procesoriais ir 96 MB DRAM) klasterį.
- VGTU klasterį *vilkas.vtu.lt* - kurį šiuo metu sudaro 20 vienprocesorinių Intel Pentium 4 (3,2 GHz procesoriai ir 1 GB DDRAM) ir 10 dviprosesorinių Intel Pentium III (1,4 GHz ir 1 GB DDRAM) SMP technologijos personalinių kompiuterių, sujungtų Gigabit Ethernet technologija į lokalų tinklą.
- IDRIS superkompiuterių centro IBM Power4 kompiuterį *Zahir* iš viso su 1024 procesorių.

1 skyrius

Šeimininkas – darbininkai algoritmo šablonas

Šiame skyriuje nagrinėsime algoritmų lygiagretinimo priemonę *Šeimininkas – darbininkai* algoritmo šabloną. 1.2 poskyryje pateikiamas *šeimininkas – darbininkai* algoritmas. 1.3 poskyryje pateiktas algoritmo lygiagretinimo priemonės aprašymas. 1.4 poskyryje pateikti skaičiavimo eksperimentų rezultatai.

1.1 Įvadas

Nagrinėsime plačiai naudojamą *šeimininkas – darbininkai* (angl. *Master – slave*) algoritmą. Nors jis gali būti naudojamas ir nuosekliuose skaičiavimuose, šiame skyriuje nagrinėsime lygiagretųjį *šeimininkas – darbininkai* algoritmą ir šio algoritmo programų lygiagretinimo priemonę. Tai viena iš galimų MIMD skaičiavimo modelio realizacijų, nes algoritmas apibrėžia dvi funkcijas, atliekančias skirtingas operacijas su skirtingais duomenimis. Algoritmas taip pat gali būti taikomas kaip globalusis dinaminis apkrovos balansavimo algoritmas.

Algoritme apibrėžiami dviejų tipų procesai – šeimininkas ir darbininkai. Paprastai šeimininko procesas yra vienas, o darbininkų – daug. Jie gali veikti skirtinguose procesoriuose, tačiau šeimininko procesas gali veikti viename procesoriuje kartu su darbininko procesu. Šeimininkas centralizuotai valdo užduočių, reikalingų išspręsti visą uždavinį, sąrašą ir skirsto jas darbininkams. Darbininkai skaičiuoja atsiųstas užduotis ir grąžina šeimininkui rezultatą. Tokią logiką galima vykdyti tiek bendrosios, tiek paskirstytos atminties lygiagrečiuosiuose kompiuteriuose. Toliau šiame skyriuje bus kalbama apie algoritmo vykdymą paskirstytos atminties lygiagrečiuose kompiuteriuose.

Šeimininkas – darbininkai algoritmas yra efektyvus heterogeninėje aplinkoje. Dėl šios priežasties jis dažnai taikomas skaičiavimams GRID tinklyne [49], kuriami specialūs įrankiai [35]. Algoritmo principai taikomi komunikacijos pro-

tokuluose, taip pat gali būti naudojami ir jungiami su kitais algoritmais.

Šeimininkas – darbininkai algoritmas apibrėžia bendrus principus, kaip organizuojamas lygiagretusis programos vykdymas. Šie principai gali būti taikomi įvairiems uždaviniams. Todėl galima formuoti *šeimininkas – darbininkai* algoritmo šabloną. Skyriuje aptariamas dr. R.Šablinsko sukurtas *šeimininkas – darbininkai* algoritmo šablono tolesnis vystymas, testavimas ir naudojimas statinių pamatų optimizavimo uždavinio sprendimui.

1.2 Šeimininkas – darbininkai algoritmas

Šeimininkas – darbininkai algoritmą turi vykdyti du ar daugiau procesų. Algoritmas naudoja komunikacijos modelį *vienas su daug*, t.y. šeimininkas keičiasi duomenimis su kitais procesais – darbininkais. Darbininkai tarp savęs informacija nesikeičia.

Tik vienas procesas – šeimininkas – sprendimo pradžioje skaito įvedamus duomenis, suformuoja užduočių, reikalingų uždaviniui išspręsti, sąrašą, paleidžia vykdyti darbininkų procesus, ima užduotis iš užduočių sąrašo ir skirsto jas kitiems procesams – darbininkams. Darbininkai sprendžia gautas užduotis, o rezultatą grąžina šeimininkui. Nors skirtingi darbininkai gali vykdyti ir skirtingus algoritmus, dažniausiai praktikoje pasitaiko, kad visi darbininkai vykdo tą patį algoritmą.

Šeimininkas – darbininkai algoritmo lygiagretinimo žingsniai:

- *Skaidymas į užduotis.* Formuojamas nepriklausomų užduočių sąrašas. Konkretiam uždaviniui tai apibrėžiama specifiškai, tačiau norint naudoti *šeimininkas – darbininkai* algoritmą, reikia, kad visas sąrašo užduotis būtų galima vykdyti bet kuria tvarka ir nepriklausomai nuo kitų užduočių. Bendresniu atveju užduotys gali būti nevienodo sudėtingumo ir darbų sąrašas gali būti generuojamas dinamiškai, atsižvelgiant į jau atliktas užduotis.
- *Komunikacijų nustatymas.* Komunikacijų modelis yra griežtai apibrėžtas: duomenimis keičiasi tik šeimininkas su darbininkais. Darbininkai tarpusavyje duomenimis nesikeičia. Tai apriboja *šeimininkas – darbininkai* algoritmu sprendžiamų uždavinių klasę. Jis netinkamas uždaviniams, kurių sprendimo eigoje užduotims spręsti reikia informacijos apie kaimyninių užduočių sprendimo rezultatus, pavyzdžiui, iteraciniams tiesinės algebros algoritmams.
- *Užduočių apjungimas.* Sąrašo užduotys gali būti jungiamos į didesnes. Jei šis žingsnis reikalingas, užduotys jungiamos formuojant užduočių sąrašą. Šiam algoritmui šis žingsnis nėra būtinas ir toliau nebus detalizuojamas.

- *Priskyrimas*. Sąrašo užduotį šeimininkas paskiria vykdyti laisvam darbininko procesui. Šiame žingsnyje galima atsižvelgti į papildomas sąlygas, pavyzdžiui, procesoriaus, kuriame vykdomas darbininko procesas, atminties dydį, spartą, procesorių jungimo topologiją.

Šeimininkas – darbininkai algoritmui būtini tokie žingsniai:

1. Šeimininko funkcija, nuskaitanti pradinis duomenis, suformuojanti užduočių sąrašą.
2. Darbininko funkcija, atliekanti vienos užduoties skaičiavimus.
3. Šeimininko funkcija, apdorojanti iš darbininko gautus užduoties sprendimo rezultatus.

1.1-A ir 1.1-B algoritmuose pateikiamos paprasčiausio *šeimininkas – darbininkai* algoritmo realizacijos.

Algoritmas 1.1-A *Šeimininkas – darbininkai algoritmas. Šeimininko dalis.*

1. Suformuoti užduočių sąrašą $P_i, i = 1, 2, \dots, M$.
2. Kiekvienam darbininkui nusiųsti po vieną užduotį.
3. **while** (sugrąžinta mažiau, negu M rezultatų) **do**
4. gauti užduoties rezultatą iš j -tojo darbininko;
6. pridėti gautą rezultatą prie bendrojo rezultato;
- if** (užduočių sąrašas nėra tuščias) **then**
7. siųsti kitą užduotį P_k j -tajam darbininkui.
- else**
8. siųsti j -tajam darbininkui pabaigos pranešimą.
- end if**
- end do**

Algoritmas 1.1-B *Šeimininkas – darbininkai algoritmas. Darbininko dalis.*

1. **while** (negautas pranešimas apie pabaigą) **do**
2. gauti užduotį P_k iš šeimininko;
3. išspręsti užduotį P_k ;
4. nusiųsti rezultatą šeimininkui;
- end do**

Kadangi užduočių sąrašas tvarkomas dinamiškai, algoritmas yra efektyvus ir tada, kai užduotys yra nevienodo sudėtingumo ar algoritmas vykdomas heterogeniniame lygiagrečiajame kompiuteryje. Greitesnis procesorius gauna daugiau

užduočių iš šeimininko proceso. Tokiu būdu *šeimininkas – darbininkai* algoritme realizuojamas centralizuotas dinaminis apkrovos balansavimo algoritmas.

Ne visada įmanoma paruošti vieną užduočių sąrašą, iš kurio skirstomos užduotys visiems darbininkams. Kartais paruošiamas pradinis užduočių sąrašas, iš kurio generuojamas kitas užduočių sąrašas ir taip toliau. Kiekvienas užduočių sąrašas gali būti sudaromas tik po to, kai išsprendžiamos visos prieš tai buvusio sąrašo užduotys. Pavyzdžiui, tokia situacija būdinga adaptyvaus integravimo uždaviniams [13].

Vykdamas 1.1 algoritmą heterogeniniame kompiuteryje gali atsitikti taip, kad sunkiausia užduotis atiteks lėčiausiam darbininkui. Tada uždavinys bus sprendžiamas neefektyviai, nes visi procesai lauks, kol lėčiausias darbininkas išspręs savo užduotį. Tokiu atveju galima modifikuoti šeimininko vykdomą algoritmo dalį, kad nebaigtos spręsti užduotys būtų kartojamos. 1.2 algoritme pateikiamas modifikuoto *šeimininkas – darbininkai* algoritmo šeimininko dalis suformuluota [48]. Darbininko dalis lieka tokia pati, kaip ir 1.1-B algoritme.

Algoritmas 1.2 *Modifikuotas šeimininkas – darbininkai algoritmas.*
Šeimininko dalis.

1. **while**(galima suformuoti naują užduočių sąrašą P^i) **do**
2. Sudaryti užduočių sąrašą P^i ;
3. Kiekvienam darbininkui nusiųsti po vieną užduotį iš P^i ;
4. **while** (nepasiekta sąrašo pabaigos sąlyga) **do**
5. gauti užduoties rezultatą iš j -tojo darbininko;
6. pridėti gautą rezultatą prie bendrojo rezultato;
7. **if** (neįvykdytų užduočių sąrašas netuščias) **then**
8. siųsti neįvykdytą užduotį P_k^i j -tajam darbininkui.
- end if**
- end do**
- end do**
9. Nusiųsti visiems darbininkams pabaigos pranešimą

Toks modifikuotas algoritmas leidžia naudoti kelis užduočių sąrašus. Kiekvieno sąrašo pabaigos požymis gali būti skirtingas. 8 žingsnyje siunčiama neįvykdyta užduotis rodo, kad užduotys gali būti kartojamos. Tai aktualu, kai *šeimininkas – darbininkai* algoritmas vykdomas nestabilioje sistemoje ir gali nutrūkti ryšys su darbininku arba kai procesorių skaičiavimo sparta yra labai skirtinga. Jei 1 žingsnyje formuojamas tik vienas užduočių sąrašas, o 8 žingsnyje neleidžiama kartoti išsiųstų užduočių, tai gauname 1.1 algoritmą.

Šeimininkas – darbininkai algoritmus galima taikyti įvairiems uždaviniams spręsti, pavyzdžiui, DES kodavimo, adaptyvaus integravimo ar pirminių skaičių radimo uždaviniams [48].

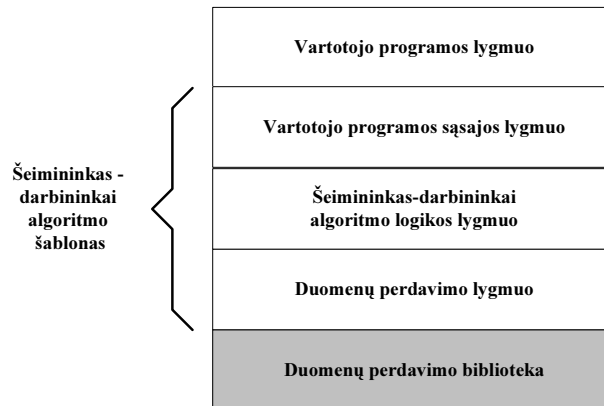
Šeimininkas – darbininkai logika negali būti taikoma kaip apvalkalas visiems uždaviniams. Algoritmo taikymą riboja griežtas komunikacijų modelis.

Siekiant, kad lygiagretusis algoritmas būtų efektyvus, reikia, kad užduoties skaičiavimo laikas būtų žymiai ilgesnis už duomenų perdavimo laiką. Taip pat, siekiant geresnio apkrovos balansavimo, užduočių turi būti daug daugiau negu darbininkų. *Šeimininkas – darbininkai* algoritmo efektyvumas gali blogėti didėjant darbininkų skaičiui. Kada šeimininko procesas bus apkrautas darbu ir darbininkų procesai turės laukti, kol gaus naują užduotį. Tai pagrindinis šio algoritmo trūkumas. Norint naudoti daug procesorių, algoritmas modifikuojamas: formuojami keli procesorių klasteriai, kuriuose vykdomas *šeimininkas – darbininkai* algoritmas. Šeimininkai tarpusavyje duomenimis gali keistis ir kitais komunikacijos modeliais.

1.3 Šeimininkas – darbininkai algoritmo šablonas

Aptariamas pusiau automatinis *šeimininkas – darbininkai* algoritmo lygiagretinimo įrankis. Įrankyje C programavimo kalba realizuotas 1.2 algoritmo šablonas, lygiagretumas vykdomas pagrindinių algoritmo funkcijų lygmenyje. Pradinis šablono variantas sukurtas dr. R.Šablinsko ir aprašytas [48], įrankis toliau vystomas.

Algoritmo šablonas skirtas automatiniam lygiagretinimui, todėl siekiama, kad vartotojui nereikėtų programuoti lygiagrečiųjų algoritmų. Įdiegta algoritmo logika gali būti pritaikyta įvairiems uždaviniams. Vartotojas ruošia nuosekliąją programą, laikydamasis tam tikrų reikalavimų. Iš šios nuosekliosios programos, naudojant šabloną, gaunama lygiagrečioji programa. Norint paslėpti nuo vartotojo lygiagrečiųjų algoritmų programavimą ir siekiant šablono universalumo, įrankis suformuotas lygmenimis (žr. 1.1 pav.).



1.1 pav. Šeimininkas – darbininkai algoritmo šablono lygmenų struktūra

Šeimininkas – darbininkai algoritmo šablono lygmenys:

- vartotojo programos lygmuo,
- vartotojo programos sąsajos lygmuo,
- *šeimininkas – darbininkai* algoritmo logikos lygmuo,
- duomenų perdavimo lygmuo.

Tokiu būdu griežtai atskirta vartotojo specifikuojama uždavinio dalis nuo algoritmo šablono. Dėl tokios įrankio struktūros jį galima naudoti su įvairiomis duomenų perdavimo bibliotekomis. Šiuo metu realizuotos versijos naudoja PVM ir MPI duomenų perdavimo bibliotekas. Naudojama standartinė C kalba leidžia įrankį kompiliuoti įvairiose sistemose įvairiais kompiliatoriais.

Įrankis pateikiamas kaip biblioteka, kompiliavimui automatizuoti paruoštas Makefile failas. Vartotojas gali pasirinkti 1.1 ar 1.2 algoritmus. Paprasčiausiu atveju lygiagrečioji programa gaunama vienu žingsniu. Sudėtingesniais atvejais įrankį galima pritaikyti specifiniams poreikiams.

Dabar aptarsime kiekvieną įrankio lygmenį. Detalizuosime tik paprasčiausią *šeimininkas – darbininkai* 1.1 algoritmą.

1.3.1 Vartotojo programos lygmuo

Nuosekloji vartotojo programa yra kuriamos lygiagrečiosios programos ir pertvarkymų, reikalingų lygiagretinimui, pagrindas. Todėl ištestuota nuosekloji programa yra pertvarkoma, kad atitiktų *šeimininkas – darbininkai* algoritmo šablono reikalavimus. Vartotojo programa turi būti sudaryta iš tokių dalių:

1. Duomenų inicializavimo dalis.
2. Skaičiavimų inicializavimo dalis.
3. Skaičiavimo ciklas, naudojantis tokias procedūras:
 - (a) Paimk neįvykdytą užduotį t_i iš užduočių sąrašo.
 - (b) Įvykdyk užduotį t_i ir grąžink rezultatą r_i .
 - (c) Pridėk rezultatą r_i prie galutinių rezultatų.
4. Rezultatų išvedimo dalis.

Šios dalys realizuojamos pagrindinėje programoje:

Programa t_main.c.

```
struct t_init_data id;
struct t_data      t;
struct t_result    res;

M_prepare_job_pool(&id)
if ( ! S_initialize(id) ) return 0;
while ( M_take_piece_from_pool(&t))
{
    S_compute(t, &res);
    M_add_to_result(res);
}
M_print_result();
```

Raidės *M* ir *S* procedūrų pavadinimuose atskiria šeimininko (*M*) ir darbininko (*S*) procesuose vykdomas procedūras.

*M_prepare_job_pool(t_init_data *id)* procedūroje suformuojamas šeimininko procesas ir vykdomos pradinės operacijos: šeimininkas skaito pradinis duomenis, suformuoja darbų sąrašą. Ši procedūra kviečiama vieną kartą programos vykdymo metu. Procedūra suformuoja *t_init_data* struktūrą, apibrėžiančią visus duomenis, reikalingus darbininko proceso inicializavimui. Šį parametą įrankis perduoda darbininkų procesams.

S_initialize(t_init_data id) procedūra sukuria darbininkų procesus, vykdo pradinį inicializavimą, pavyzdžiui išskiria dinaminę atmintį, priskiria pradines kintamųjų reikšmes. Procedūra yra kviečiama vieną kartą kiekvienam darbininko procesui, jai perduodama *M_prepare_job_pool(t_init_data *id)* procedūroje paruošta *t_init_data* struktūra.

*M_take_piece_from_pool(t_data *t)* procedūroje šeimininko procesas paima dar neįvykdytą užduotį iš darbų sąrašo ir suformuoja *t_data* struktūrą, visiškai aprašančią vieną užduotį. Šie duomenys bus persiunčiami darbininkui. Procedūra yra vykdoma cikle, kiekvieną kartą, kai reikia siųsti naują užduotį darbininkui. Darbų sąrašas gali būti paruoštas išreikštiniu būdu kaip fiksuotas užduočių skaičius arba gali būti apibūdinamas tam tikru kriterijumi, pavyzdžiui, pasiektu skaičiavimų tikslumu.

*S_compute(t_data d, t_result *res)* procedūra sprendžia užduotį, gautą iš šeimininko. Jai perduodama *t_data* struktūra. Skaičiavimams pasibaigus formuojama rezultatų struktūra *t_result*, kuri bus perduota šeimininkui.

M_add_to_result(t_result res) procedūroje šeimininkas apdoroja iš darbininko gautą rezultatą *t_result*. Paprasčiausias apdorojimo pavyzdys – gautų rezultatų sumavimas prie esamo rezultato.

M_print_result() funkcija išveda viso uždavinio sprendimo rezultatą.

Nuosekliojoje programoje vartotojas turi surašyti šeimininko ir darbininko funkcijas į atskirus modulius – i_master.c ir i_slave.c. Siūloma vengti naudoti globaliuosius kintamuosius. Duomenys turi būti suformuojami proceso viduje, arba perduodami funkcijomis, naudojant vieną iš trijų duomenų struktūrų, vartotojo aprašytą i_defs.h faile:

1. struct t_init_data id – pradinių duomenų struktūra, suformuojama M_prepare_job_pool(t_init_data *id) procedūros ir naudojama S_initialize(t_init_data id) procedūroje.
2. struct t_data d – vieną užduotį aprašantis duomenys, formuojami M_take_piece_from_pool(t_data *d) ir naudojami S_compute(t_data d, t_result *res).
3. struct t_result res – vienos užduoties rezultatų duomenys, formuojami S_compute(t_data d, t_result *res) procedūros, o apdorojami M_add_to_result(t_result res) procedūros.

Šablono naudojamų struktūrų elementai gali būti visų standartinių C kalbos kintamųjų tipų: char, int, long, float, double ir kiti, taip pat masyvai ir struktūros. Jei duomenų perdavimui naudojamos struktūros, jos turi būti anksčiau aprašytos tame pačiame i_defs.h faile. Failą i_defs.h analizuoja specialus šablono modulis, kuris paruošia duomenis perdavimui.

Testavimas. Pertvarkius nuosekliąją programą, kad ji tenkintų visus šablono reikalavimus, vartotojui rekomenduojama patikrinti nuosekliojo programos varianto teisingumą.

1.3.2 Vartotojo programos sąsajos lygmuo

Šio lygmens tikslas – sujungti vartotojo lygmens procedūras su *šeimininkas – darbininkai* algoritmo šablono procedūromis. Vartotojo sąsajos lygmuo susideda iš vartotojo duomenų paruošimo modulio ir šešių procedūrų:

- u_master_prepare_job_pool;
- u_master_take_piece_from_pool;
- u_master_add_result_to_total;
- u_master_print_result;
- u_slave_initialize;
- u_slave_compute.

Šios procedūros kviečia vartotojo aprašytas procedūras ir konvertuoja vartotojo naudojamus kintamuosius į apibendrintus kintamuosius, naudojamus šablone.

Vartotojo duomenų paruošimo modulis. Šablonas papildomai ruošia tarp procesorių perduodamus duomenis: atskirų tipų kintamieji yra surašomi į atskirus masyvus, kurie pakuojami ir persiunčiami tarp procesų naudojant duomenų perdavimo bibliotekų funkcijas. Po perdavimų, masyvai vėl transformuojami į vartotojo programoje naudotus kintamuosius. Todėl įrankis gali būti naudojamas su įvairiomis duomenų perdavimo bibliotekomis.

Tam, kad vartotojas galėtų naudoti jam patogius kintamųjų vardus ir konstrukcijas, jų skirstymą į duomenų tipo masyvus atlieka įrankis. Vartotojo duomenų paruošimo modulis analizuoja vartotojo duomenų struktūras, apibrėžtas vartotojo lygmenyje faile `i_defs.h` ir ruošia duomenų transformacijos funkcijas. Jis atpažįsta pagrindines C kalbos konstrukcijas, standartinių tipų kelių dimensijų masyvus ir struktūras. Vartotojo duomenų paruošimo modulis analizuoja kiekvieną kintamąjį vartotojo duomenų struktūroje, priskiria jį vienam iš duomenų tipo masyvų. Šios funkcijos vėliau kompiliuojamos su algoritmo šablonu.

1.3.3 Šeimininkas – darbininkai algoritmo šablono logikos lygmuo

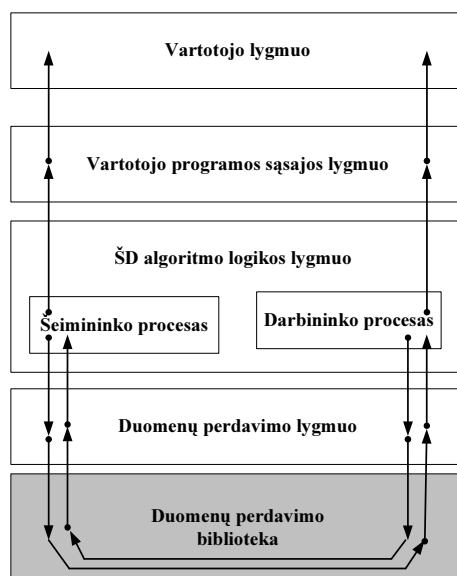
Pagrindinis šio lygmens tikslas yra sukurti šeimininko ir darbininko procesus, organizuoti darbų paskirstymą tarp procesų ir duomenų perdavimus. Algoritmo logikos lygmuo jungia vartotojo sąsajos ir duomenų perdavimo lygmenis (žiūrėti 1.2 pav.).

Šis lygmuo susideda iš dviejų procedūrų: Master ir Slave. Procedūros yra gana paprastos, jas sudaro veiksmų ciklas. Šeimininko proceso veiksmams apibrėžiami pagrindinių algoritmo žingsnių:

- `ACT_INIT_SLAVES` ruošiami duomenys darbininko proceso inicializavimui.
- `ACT_SEND_ALL` duomenys siunčiami visiems darbininkams.
- `ACT_RECEIVE_RES` priimami darbininko siunčiami skaičiavimų rezultatai.
- `ACT_SEND_TASK` siunčiama užduotis darbininkui.
- `ACT_SEND_FINISH` siunčiamas skaičiavimų pabaigos požymis.
- `ACT_FINISH` baigiamas šeimininko proceso veiksmų ciklas.

Darbininko procesas vykdo veiksmus priklausomai nuo gautų duomenų tipo:

- `TAG_INIT` tipas reiškia, kad priimami duomenys darbininko proceso inicializavimui.



1.2 pav. Šeimininkas – darbininkai algoritmo šablono logikos lygmens operacijos

- TAG_PIECEINFO tipas reiškia, kad priimami užduoties duomenys. Vykdomi skaičiavimai, ruošiamas ir siunčiamas šeimininkui rezultatas.
- TAG_HALT tipas reiškia skaičiavimų pabaigą.

Šiame lygmenyje numatytas modifikuoto 1.2 algoritmo vykdymas, įdiegtas klaidų tikrinimo mechanizmas: užduotys gali būti pakartotinai vykdomos, jei dėl kažkokių priežasčių darbininkas neatsiunčia rezultato.

1.3.4 Duomenų perdavimo lygmuo

Duomenų perdavimo lygmuo yra sąsajos lygmuo, kurio tikslas – apibendrinti duomenų perdavimo bibliotekos procedūras. Šiame lygmenyje aprašytos naudojamų tipų masyvų pakavimo ir išpakavimo procedūros, siunčiamų pranešimų tikrinimo procedūros. Šio lygmens funkcijos yra kviečiamos iš *šeimininkas – darbininkai* logikos lygmens procedūrų.

Paruoštos atskiros sąsajos MPI ar PVM duomenų perdavimo bibliotekų naudojimui. Modifikuojant duomenų perdavimo lygmens procedūras, *šeimininkas – darbininkai* algoritmo šabloną galima pritaikyti ir kitoms duomenų perdavimo bibliotekoms.

1.3.5 Šablono naudojimas

Lygiagrečiosios programos kompiliavimui yra paruošti vykdomieji failai (Makefile), kurie leidžia vienu žingsniu sujungti šabloną su vartotojo programa. Juose atliekami tokie veiksmai:

1. Kuriama vartotojo `i_master.c`, `i_slave.c` ir `i_defs.h` sąsaja.
2. Kompiliuojamas duomenų transformacijos modulis.
3. Analizuojamos `i_defs.h` faile aprašytos struktūros ir kuriamos duomenų transformacijos funkcijos.
4. Kompiliuojamas algoritmo šablonas, kartu su vartotojo programos moduliais ir duomenų transformacijos funkcijomis.

Paruoštus Makefile failus galima modifikuoti, pritaikant šabloną naudojamai sistemai, kompiliatoriams ir specifiniams poreikiams. Šablono failai nekomerciniam naudojimui gali būti rasti adresu <http://mmb.vtu.lt/MST3/>.

1.4 Šeimininkas – darbininkai algoritmo šablono taikymo pavyzdžiai

1.4.1 Pirminių skaičių radimo uždavinys

Natūralieji skaičiai, kurie dalinasi tik iš vieneto arba savęs, vadinami pirminiais skaičiais. Šiais skaičiais domėtasi nuo seno, tačiau XX amžiuje atsirado realūs taikymai: viešojo rakto (asimentrinės) kodavimo sistemos naudoja fundamentaliąją aritmetikos teoremą, teigiančią, kad visus sveikus skaičius galima užrašyti kaip pirminių skaičių sandaugą, ir šis išskaidymas yra vienintelis. Pirminiai skaičiai taip pat naudojami indeksavimui, pseudoatsitiktiniams skaičiams generuoti.

Galima išskirti du pirminių skaičių radimo uždavinius:

1. Pirminių skaičių tikrinimas – kai tikrinama, ar skaičius yra pirminis. Tokiu būdu ieškoma ypač didelių pirminių skaičių. Naudojami įvairūs testai: Fermat, Lucas-Lehmer ir kiti.
2. Pirminių skaičių, mažesnių už nurodytą kintamąjį, skaičiaus radimas. Šio uždavinio sprendimo algoritmai buvo naudoti *šeimininkas – darbininkai* algoritmo šablono testavimui.

Pirminių skaičių radimo uždavinį sprendėme dviem algoritmais: dalinimo ir Eratosteno rėčio. Ieškoma kiek yra pirminių skaičių, ne didesnių už M .

Algoritmas 1.3 Pirminių skaičių radimo dalinimo algoritmas

1. $P = \{2\}, \quad n = 1,$
2. **for all** ($3 \leq j \leq M$) **do**
3. **if** (j dalinasi iš kurio nors pirminio ne didesnio už $\sqrt{j}$) **then**
4. j nėra pirminis;
5. **else**
6. $P := P \cup \{j\}, \quad n := n + 1.$
7. **end if**
8. **end do**

Algoritmas 1.4 Eratosteno rėčio pirminių skaičių radimo algoritmas

1. Išskiriamas M dydžio masyvas T . Visiems jo elementams priskiriamas požymis, kad skaičius yra pirminis $T(i) = 1, i = 1, \dots, M, \quad n = 0.$
2. **for all** ($2 \leq j \leq \sqrt{M}$) **do**
3. **if** ($T(j) = 1$) **then**
4. $P := P \cup \{j\}, \quad n := n + 1;$
5. Masyvo elementams nuo $2j$ iki M su žingsniu j priskiriamas požymis, kad skaičius nėra pirminis: $P(i) = 0, i = 2j, \dots, kj \leq M$
6. **end if**
7. **end for**
8. **for all** ($\sqrt{M} \leq j \leq M$) **do**
9. **if** ($P(j) = 1$) **do**
10. $P := P \cup \{j\}, \quad n := n + 1.$
11. **end if**
12. **end for**

Šie algoritmai nesunkiai realizuojami naudojant *šeimininkas – darbininkai* algoritmo logiką. Pateikiame vieną *šeimininkas – darbininkai* algoritmo šablono realizaciją. Siekiant didesnio efektyvumo, Eratosteno rėčio algoritmas yra modifikuojamas įvedant papildoma parametą – langą, kuris nurodo, jog pirminių bus ieškoma ne visame duotame intervale, o tik fiksuoto ilgio jo dalyje – „lange“. Suradus pirminius lango viduje, imamas kitas „langas“.

Užduočių sąrašas. Užduočių sąrašą sudarome dalindami intervalą $[\sqrt{M}, \dots, M]$ į k dalių:

$$\left\{ t_i = (\sqrt{M} + m(i-1), \sqrt{M} + mi], i = 1, 2, \dots, k \right\}, \quad m = \left\lceil \frac{M - \sqrt{M}}{k} \right\rceil.$$

Pirminių skaičių paieška bet kuriame intervale t_i gali būti vykdoma nepriklausomai nuo paieškos kituose intervaluose. Užduočių sąrašas nėra saugomas, o generuojamas programos vykdymo metu.

Algoritmas. `M_prepare_job_pool` procedūra nuskaitys pradinis duomenis (intervalą, kuriame reikia rasti kiek yra pirminių skaičių $[2, \dots, M]$, vienos užduoties dydį K) iš failo ir paruošia pradinį duomenų struktūrą

`struct t_init_data{ long M, K }.` Taip pat randama, kiek yra pirminių skaičių intervale $[0, \dots, \sqrt{M}]$. `S_initialize` procedūra išskiria darbininkų duomenų struktūras ir taip pat skaičiuoja pirminius skaičius $[0, \dots, \sqrt{M}]$ intervale.

`M_take_piece_from_pool` procedūra seka, kuris intervalas yra tiriamas ir praneša, kai paskutinė užduotis yra išsiųsta. Ji formuoja užduotį apibūdinančią struktūrą – intervalo t_i pradžią: `struct t_data{ long i }.`

`S_calculate` procedūra ieško pirminių skaičių gautame intervale. Intervale rastų pirminių skaičių kiekis grąžinamas struktūroje `struct t_result{ long plu }.` Eksperimentuose vykdyti dalinimo ir Eratosteno rėčio algoritmai.

`M_add_to_result` procedūra sumuoja atsiųstus rastus pirminių skaičių kiekius.

Naudojantis *šeimininkas – darbininkai* algoritmo šablonu, gautos lygiagrečiosios programos. Jos vykdytos IBM RS/6000 kompiuterių klasteryje ir VGTU klasteryje *vilkas.vtu.lt*. $[\sqrt{M}, \dots, M]$ intervalą reikia skaidyti taip, kad dalių skaičius K būtų daug didesnis už naudojamų procesorių skaičių $K > p$. Kuo didesnė K reikšmė, tuo geresnį procesorių apkrovos balansą pasiekiamo. Kita vertus, kiekviena užduotis t_i turi būti pakankamai didelė, kad būtų gautas didelis algoritmo efektyvumas, t.y. duomenų perdavimo kaštai išliktų pakankamai maži, o užduoties skaičiavimas užtruktų bent kelias procesoriaus darbo sekundes. Visuose skaičiavimuose naudojome $K = 100$. Atliktų skaičiavimų IBM RS/6000 klasteryje, naudojant PVM duomenų perdavimo biblioteką rezultatai pateikti 1.1 ir 1.2 lentelėse. Rezultatai ir įrankio aprašymas skelbti [70, 75].

1.1 lentelė. Pirminių skaičių radimas Eratosteno rėčiu $[2; 10^9]$ intervale. PVM, IBM RS/6000

Procesoriai, p	Laikas, T_p	S_p	E_p
1	834,37	1,00	1,00
2	455,51	1,83	0,91
3	296,96	2,81	0,94
4	221,03	3,77	0,94
5	178,66	4,67	0,93
6	153,02	5,45	0,91
7	129,66	6,43	0,92

Iš rezultatų matyti, kad įrankio generuota lygiagrečioji programa vykdoma efektyviai. Eratosteno rėčio algoritmo vykdymo efektyvumas mažesnis, nes šio algoritmo skaičiavimai atliekami greičiau ir papildomi programos kaštai – duomenų perdavimas – stipriau įtakoja bendrą efektyvumą.

1.2 lentelė. Pirminių skaičių radimas dalinimo algoritmu $[2; 10^8]$ intervale, PVM, IBM RS/6000

Procesoriai, p	Laikas, T_p	S_p	E_p
1	1492,42	1,00	1,00
2	753,75	1,98	0,99
3	523,66	2,85	0,95
4	401,18	3,72	0,93
5	317,54	4,70	0,94
6	253,81	5,88	0,98
7	213,81	6,98	0,99

Eratosteno rėčio algoritmo programos buvo testuotos ir homogeniniame lygiagrečiajame kompiuteryje – VGTU klasteryje *vilkas.vtu.lt*. 1.3 lentelėje pateikti rezultatai gauti vykdant programą dviprosesoriniuose kompiuteriuose, naudojant po vieną procesorių. 1.4 lentelėje pateikti rezultatai gauti vykdant Eratosteno rėčio algoritmo programą vienprocesoriniuose mazguose.

1.3 lentelė. Pirminių skaičių radimas Eratosteno rėčiu $[2; 10^9]$ intervale, PVM, vilkas.vtu.lt

Procesoriai, p	Laikas, T_p	S_p	E_p
1	61,59	1,00	1,00
2	30,99	1,99	0,99
3	20,77	2,97	0,99
4	15,57	3,96	0,99
6	10,39	5,93	0,99
8	7,79	7,91	0,99
10	6,22	9,90	0,99

Skaičiavimai VGTU klasteryje *vilkas.vtu.lt* atliekami greičiau, negu RS/6000 klasteryje, lygiagrečiųjų programų efektyvumas nemažėja.

Rezultatai, naudojant MPI duomenų perdavimo biblioteką, pateikti 1.5 lentelėje.

Visų eksperimentų rezultatai rodo, kad *šeimininkas – darbininkai* algoritmo programų pagreitinėjimas beveik tiesinis, o efektyvumas artimas vienetui. Taip pat parodyta, kad efektyviai vykdomos lygiagrečiosios programos gaunamos tiek naudojant MPI, tiek PVM duomenų perdavimo bibliotekas.

1.4 lentelė. Pirminių skaičių radimas Eratosteno rėčiu $[2; 10^9]$ intervale, PVM vilkas.vtu.lt

Procesoriai, p	Laikas, T_p	S_p	E_p
1	37,56	1,00	1,00
2	19,11	1,97	0,98
4	9,60	3,91	0,98
6	6,43	5,84	0,97
8	4,82	7,79	0,97
10	3,87	9,71	0,97
12	3,22	11,66	0,97
15	2,58	14,56	0,97

1.5 lentelė. Pirminių skaičių radimas Eratosteno rėčiu $[2; 10^9]$ intervale, MPI, vilkas.vtu.lt

Procesoriai, p	Laikas, T_p	S_p	E_p
1	37,56	1,00	1,00
2	19,05	1,97	0,98
3	12,69	2,93	0,98
4	9,43	3,98	0,99
6	6,28	5,97	0,99
8	4,72	7,94	0,99
10	3,80	9,87	0,98
12	3,15	11,92	0,99
15	2,51	14,90	0,99

1.4.2 Šeimininkas – darbininkai algoritmo taikymas statinių pamatų optimizavimui

Pateiksime *šeimininkas – darbininkai* algoritmo šablono taikymą statybinės mechanikos uždavinio sprendimui.

Uždavinio formulavimas

Aptariamas rostverkinio tipo statinių pamatų optimizavimo uždavinys. Tokie pamatai yra sudaryti iš atskirų sijų, paremtų į pagrindą – gruntą sukaltais poliais, sijos taip pat gali gulėti ant kitų sijų. Statinių projektavimo metu visos statinio dalys turi būti suprojektuotos kiek galima taupiau, nepažeidžiant saugumo ir eksploatacijos sąlygų. Optimaliai suprojektuotuose rostverkiniuose pamatuose poliai yra išdėstyti taip, kad atraminės reakcijos juose yra lygios ir neviršija leistinosios reakcijos. Praktiškai tai yra sunkiai pasiekama, jei projektuotojas įveda

„nejudamas atramas“, kurios turi išlaikyti savo poziciją ir nedalyvauja optimizavimo procese. Visos apkrovos (koncentruotos jėgos, tiesiškai paskirstyti slėgiai, lenkimo momentai) yra fiksuotos. Optimizavimo tikslų siekiama keičiant polių standžius, ilgius, polių pozicijas sijose. Be abejo, technologiniais sumetimais patogiausia naudoti vienodus polius, o keisti tik jų pozicijas. Globalaus minimumo sprendinys yra svarbus, tačiau praktinio taikymo uždaviniuose dažnai tenkinamasi pakankamai geru sprendiniu, kuris įvertinamas inžineriškai, bei gali ir nebūti globalusis minimumas.

Matematinis modelis

Sprendžiamas optimizavimo uždavinys:

$$\min_{s.t. x \in D} P(x), \quad (1.1)$$

čia P – tikslo funkcija, D – galimas struktūros pavidalas, kurį apibrėžia duotų atramų tipas, skaičius ir išdėstymas, struktūra. P yra apibrėžta, kaip didžiausias skirtumas tarp vertikalios atramos reakcijos atramos vietoje ir leistinos reakcijos į atramą. Taip galima gauti skirtingas reakcijas atramose skirtingose sijose ar net nurodytoje atramoje toje pačioje sijoje:

$$P(x) = \max_{1 \leq i \leq N_a} |R_i - f_i R_{allowable}|, \quad (1.2)$$

čia N_a yra atramų skaičius, $R_{allowable}$ – leistinoji reakcija, f_i – šios reakcijos daugiklis, R_i – atraminė jėga (reakcijos jėga) kiekvienoje atramoje.

Toliau uždavinys yra suvedamas į minimizacijos uždavinį, su apribojimais, laikant P_{max} nežinomu parametru. P_{max} apriboja P dydžius visur struktūroje ir visiems apkrovos atvejams, kai keičiama atramos i padėtis sijyne x dydžiu Δx_i :

$$P(x) + \sum_{i=1}^{N_s} \frac{\partial P(x)}{\partial x_i} \Delta x_i - P_{max} \leq 0, \quad (1.3)$$

visai struktūrinei erdvei x . Sijos ilgis optimizavimo metu nekinta:

$$L(x) + \sum_{i=1}^{N_s} \frac{\partial L(x)}{\partial x_i} \Delta x_i - L_0 \leq 0, \quad (1.4)$$

čia L_0 yra pradinis modelio ilgis. Parametrus x_i , nuo kurių priklauso struktūros forma, galima pasirinkti keliais būdais. Šiuo atveju x_i yra visų pasirinkto rinkinio atramų koordinatės baigtinių elementų tinklelio mazguose. Uždavinys yra netiesinis, todėl jis sprendžiamas iteraciniu algoritmu. Kiekvienoje iteracijoje esama forma yra keičiama į geresnę kaimyninę formą.

Pamatų optimizavimo uždavinys buvo sprendžiamas [5] aprašytais nuosekliaisiais algoritmais, reikalavusiais trijų žingsnių:

- baigtinių elementų analizės;
- jautrumo analizės atsižvelgiant į konstrukcinius parametrus;
- optimizavimo.

Poskyryje aprašomas globaliosios optimizacijos algoritmo taikymas statinių pamatų optimizavimui ir jo lygiagrelinimas.

Baigtinių elementų analizė

Polius jungianti sija aproksimuojama lenkiamo strypo baigtiniu elementu. Polius, remiantis sija, idealizuojamas atramine reakcija. Naudojamas dviejų mazgų su dviem laisvės laipsniais mazgo elementas [4]. Mazgo laisvės laipsniai yra:

$$\mathbf{u} = \{\omega_i, \theta_i, \omega_j, \theta_j\}^T, \quad (1.5)$$

čia ω_k ir θ_k , $k = i, j$ yra atitinkamai poslinkiai ir posūkiai. Interpoliavimo funkcijos visiems laisvės laipsniams:

$$[N] = \begin{Bmatrix} 1 - \frac{3x^2}{L^2} + \frac{2x^3}{L^3} \\ x - \frac{2x^2}{L} + \frac{x^3}{L^2} \\ \frac{3x^2}{L^2} - \frac{2x^3}{L^3} \\ -\frac{x^2}{L} + \frac{x^3}{L^2} \end{Bmatrix}^T, \quad (1.6)$$

čia L – elemento ilgis. Lenkimo momentai mazguose, yra susiję su poslinkiais:

$$M = -EI w_{i,xx} = EI \sum_i N_{i,xx} u_i. \quad (1.7)$$

Po to, kai randami mazginiai poslinkiai, reakcijos jėgas galima rasti pagal

$$R_i = \sum_j K_{ij} u_j^a, \quad (1.8)$$

čia K yra baigtinių elementų rinkinio standumo matrica.

[5, 6] aprašytuose optimizavimo algoritmuose visų pamatų optimizavimo pagrindas yra vienos atskiros sijos optimizavimas. Pamatai yra suskaidomi į atskiras sijas. Vadinamos „viršutinės sijos“ guli ant „apatinių sijų“. Pradžioje, visos sijos yra analizuojamos ir optimizuojamos atskirai. Sujungimai ir viršutinių bei apatinių sijų persidengimai yra idealizuojami kaip nejudinamos atramos viršutinėms sijoms

ir koncentruotos jėgos apatinėms. Kadangi tokios tariamos reakcijos/koncentruotos jėgos priklauso nuo atramų išdėstymo, visi skaičiavimai yra iteraciniai.

Optimizavimui reikalingas reakcijos jėgų jautrumas skaičiuojamas:

$$\frac{\partial \mathbf{R}}{\partial x_k} = \frac{\partial \mathbf{K}^a}{\partial x_k} \mathbf{u}^a + \mathbf{K}^a \frac{\partial \mathbf{u}^a}{\partial x_k}, \quad (1.9)$$

čia viršutinis indeksas a žymi ansamblį. Mazginiai poslinkiai yra gaunami sprendžiant lygtį:

$$\mathbf{K}^a \frac{\partial \mathbf{u}^a}{\partial x_k} = \bar{\mathbf{P}}^a, \quad (1.10)$$

čia $\bar{\mathbf{P}}^a$ žymi pseudo-apkrovos vektorius.

$$\bar{\mathbf{P}}^a = \frac{\partial \mathbf{P}^a}{\partial x_k} - \frac{\partial \mathbf{K}^a}{\partial x_k} \mathbf{u}^a \quad (1.11)$$

čia $\mathbf{P}^a$ yra visos struktūros vektorius.

Išvestinių ansamblio matrica $\frac{\partial \mathbf{K}^a}{\partial x_k}$ formuojama iš atskirų elementų matricų išvestinių i -tojo ar j -ojo mazgo koordinatų atžvilgiu, priklausomai nuo to, kuriuo baigtinio elemento mazgu yra k -tasis ansamblio mazgas. Jei ansamblio k -tasis mazgas elementui iš viso nepriklauso, elemento išvestinės standumo matrica yra nulinė.

[5, 6] taikytą optimizavimo algoritmą sudaro vienos sijos optimizavimo procedūra ir viso sijyno optimizavimas.

Vienos sijos optimizavimo procedūra. $\mathbf{T}$ žymi dabartinį projektavimo kintamųjų būvį (polių pozicijas). Priklausomai nuo esamų projektavimo ribojimų ir kitų sąlygų, yra parenkami du absoliučių jų kitimo ribų rinkiniai: maksimumas ir minimumas. Projektuojami kintamieji negali viršyti šių ribojimų. Absoliutūs ribojimai gali kisti, priklausomai nuo kintamojo, tačiau maksimalus absoliutaus poslinkio ribojimas turi būti ne neigiamas, o minimalus – ne teigiamas.

$$\mathbf{T}^{\min} \leq \mathbf{T} \leq \mathbf{T}^{\max}, \quad \mathbf{T}^{\min} \leq 0, \mathbf{T}^{\max} \geq 0. \quad (1.12)$$

Pirmuoju sprendimo žingsniu, esamas kintamojo statusas yra $\mathbf{T} = 0$. Toliau, nurodomos kintamojo vienos iteracijos poslinkio ribos, vėl maksimumas ir minimumas. Šios ribos gali kisti priklausomai nuo kintamojo ir turi būti kontroliuojamos, kad uždavinį būtų galima spręsti kaip tiesinį. Bendru atveju, poslinkio ribos turi mažėti, artėjant prie optimalaus sprendinio. Aproximacijos tikslumas turi būti didesnis, artėjant prie optimalaus sprendinio, nes padalinimai smulkėja ir gali būti panaikinti aproximacijos paklaidų. Iteracijos poslinkių ribos gali būti suderintos specialiomis programos procedūromis, todėl:

$$\Delta \mathbf{T}^{\min} \leq \Delta \mathbf{T} \leq \Delta \mathbf{T}^{\max}. \quad (1.13)$$

Nurodomi du tarpinių, visada teigiamų, kintamųjų rinkiniai $\Delta \mathbf{T}^+$ ir $\Delta \tilde{\mathbf{T}}$. Siekiant patenkinti Simplekso procedūros [11] reikalavimus, gaunama lygybė:

$$\Delta \mathbf{T}^+ + \Delta \tilde{\mathbf{T}} = \Delta \mathbf{T}^{\max} - \Delta \mathbf{T}^{\min}. \quad (1.14)$$

Taigi suformuluojamas toks tiesinio programavimo uždavinys:

$$\min_{s.t. x \in D} P_{max}, \quad (1.15)$$

čia D yra apibrėžta tokiais apribojimais:

$$\mathbf{P} + \frac{\partial \mathbf{P}}{\partial \mathbf{T}} \Delta \mathbf{T} - \mathbf{I} P_{max} \leq 0, \quad (1.16)$$

$$\Delta \mathbf{T}^+ + \Delta \tilde{\mathbf{T}} = \Delta \mathbf{T}^{\max} - \Delta \mathbf{T}^{\min}, \quad (1.17)$$

$$L + \frac{\partial \mathbf{L}^e}{\partial \mathbf{T}} = L_0. \quad (1.18)$$

Sijyno optimizacijos procedūra. Pirmiausia atskiriamos „viršutinės“ ir „apatinės“ sijos. Toliau kiekviena sija yra optimizuojama cikle, nes vienos sijos atramų padėtys schemoje įtakoja reakcijų pasiskirstymą kitose sijose:

Algoritmas 1.5 *Sijyno optimizavimo algoritmas.*

1. Struktūros inicializavimas.
Nustatomas standumas fiktyviose nejudamose viršutinių sijų atramose imituojant sujungimus su apatinėmis sijomis.
2. Stop := false
3. **while** (Stop = false) **do**
4. Optimizuoti viršutines sijas naudojantis paskutinėje iteracijoje apibrėžtais fiktyvių nejudamų atramų standumais.
5. Optimizuoti apatines sijas, specifines apkrovas papildant iš viršutinių sijų atsiradusiomis koncentruotomis jėgomis
6. **if** (viršutinės ir apatinės sijų standumai sujungimuose sutampa) **then**
7. Stop := true.
8. **end if**
9. **end do**
8. Rezultatai filtruojami ir išskiriamos sijų sujungimuose sutampančios atramos

Algoritmas pradedamas vykdyti nuo pradinės sijyno schemos, kuri yra artima optimaliai. Tokią schemą pasiūlo ekspertinė sistema. Daugeliu atveju tai garantuoja globaliojo optimalaus sprendinio radimą.

Globalioji optimizacija

Lygiagretiems skaičiavimams, projektavimo kintamaisiais pasirinktos atramų padėties, t.y. jų koordinatės sijoje. Visas sijynas transformuojamas į vienmatę siją, todėl x reiškia atramos padėtį tokioje abstrakčioje sijoje. Išsprendus uždavinį, x transformuojamas į koordinates plokštumoje. Reakcija skaičiuojama naudojant baigtinių elementų analizę.

Globaliajai optimizacijai naudojamas atsitiktinės paieškos metodas: tiriamoje srityje generuojami taškai – atsitiktinės atramų pozicijos. Toliau vykdomas lokalus optimizavimo algoritmas ir išrenkama geriausia atramų išdėstymo schema. Visi uždaviniai gali būti sprendžiami nepriklausomai nuo kitų. Dėl to juos galima spręsti lygiagrečiai, taikant *šeimininkas – darbininkai* algoritmą. Užduočių sąrašas gaunamas pasirenkant 50 skirtingų pradinių judamų atramų pasiskirstymų.

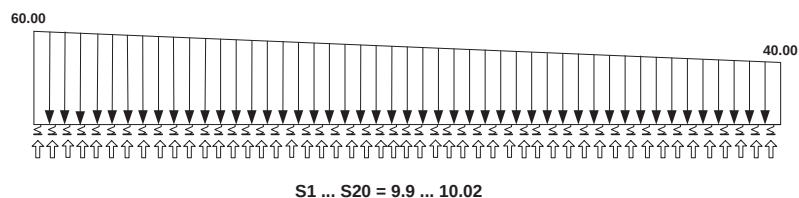
`M_Prepere_job_pool` kviečia procedūrą, formuojančią pradinių atramų pasiskirstymo variantus. `S_initialize` išskiria dinaminę atmintį, reikalingą skaičiavimams. `M_take_piece_from_pool` ima vieną netirtą pradinių apkrovų paskirstymą ir suformuoja užduotį darbininkui. Darbininko funkcija `S_compute` vykdo nuoseklų optimizavimo algoritmą ir grąžina gautas reakcijas, o `M_add_to_result` išrenka geriausią sprendinį. Pabaigoje išvedamas geriausias sprendinys.

Pastebima, kad kiekvienos užduoties skaičiavimo kaštai gali būti labai skirtingi, todėl išankstinis užduočių paskirstymas tarp procesorių gali duoti labai neefektyvų lygiagretųjį algoritmą. *Šeimininkas – darbininkai* algoritmas uždavinius išdalina dinamiškai programos vykdymo metu, taip galima pasiekti gerą lygiagrečiosios programos efektyvumą. Sijyno optimizacijos uždavinys tinka *šeimininkas – darbininkai* įrankio testavimui. Šis globaliosios optimizacijos metodas yra labai paprastas, bet negalima garantuoti, kad sprendinys yra arti globaliojo minimumo. Algoritmo efektyvumas priklauso nuo adaptyviai pasirinktų pradinių artinių, bandoma paskirstyti nagrinėjamus taškus netolygiai tiriamame regione didesniu tankumu labiau tikėtinose srityse.

Eksperimentas

Skaičiavimo eksperimentas atliktas optimizuojant polių pozicijas, kai pamatus sudaro viena sija, apkrauta trapecijos pavidalo paskirstyta apkrova, kurios intensyvumas kinta nuo 60 iki 40. Skaičiavimų pradžioje užduodama didžiausia leistinoji reakcija $R_{allowable} = 10, 20$, vertikalus standumas $k_{spring} = 10^5$.

Šiame pavyzdyje nėra nurodyta nejudamų atramų, todėl visas atramas programa gali judinti. Programa generuoja skaičiavimo schemą su 101 mazgu ir 100 baigtinių elementų ir tik 50 atramų. 1.3 paveikslėlyje pateikiamas gautas polių išdėstymas. Reakcijos atramose nėra vienodos, kinta nuo 9,9 iki 10,02, nes programa baigia optimizavimo algoritmą, kai leistinoji reakcija yra neviršyta visose atramose.



1.3 pav. Šeimininkas – darbininkai algoritmu apskaičiuotos polių padėtys

1.6 lentelėje pateikiami rezultatai, iliustruojantys gautos lygiagrečiosios programos efektyvumą. Skaičiavimai atlikti RS/6000 kompiuterių klasteryje.

1.6 lentelė. Pamatų optimizavimo uždavinys, RS/6000

Procesoriai, p	Laikas, T_p	S_p	E_p
1	365,1	1,00	1,000
2	188,8	1,93	0,967
3	126,6	2,88	0,961
4	96,5	3,78	0,946
5	78,5	4,65	0,930
6	65,4	5,58	0,930
7	56,3	6,48	0,926

Iš rezultatų matyti, kad lygiagretusis algoritmas, generuotas naudojant *šeimininkas – darbininkai* šabloną, efektyviai naudoja kompiuterinius resursus, spartinio koeficientas auga tiesiškai: pridendant daugiau procesorių, efektyvumo koeficientas artimas vienetui. Skaičiavimų rezultatai skelbti [68], panašūs rezultatai buvo gauti ir naudojant inžinerinius metodus.

1.5 Skyriaus rezultatai ir išvados

1. *Šeimininkas – darbininkai* šabloną papildžius modulių, kuris vartotojo naudojamas duomenų struktūras konvertuoja į struktūras, naudojamas šablono duomenims tarp procesorių perduoti, supaprastėjo vartotojo programos paruošimas. Vartotojas gali naudoti jam patogias duomenų struktūras net ir tarp procesų perduodamiems duomenims.
2. Suformuota šablono lygmenų struktūra leidžia paruošti įrankį skirtingų duomenų perdavimo tarp procesorių bibliotekų naudojimui. Įrankis testuotas su PVM ir MPI duomenų perdavimo bibliotekomis.

Naudojant šabloną gaunama lygiagrečioji programa yra efektyvi. Tiek naudojant MPI, tiek PVM duomenų perdavimo bibliotekas, gautų lygiagrečiųjų programų efektyvumas yra didesnis už 0,9 homogeniniame lygiagrečiajame kompiuteryje.

3. Globaliosios optimizacijos algoritmas naudotas rostverkinio tipo pamatų optimizacijai. Lygiagrečiojo algoritmo, generuoto *šeimininkas – darbininkai* lygiagretinimo įrankiu, pagreitėjimas net ir heterogeninėje aplinkoje yra proporcingas naudotų procesorių skaičiui. Skaičiavimo eksperimento metu nebuvo gautas geresnis sprendinys, negu rastas anksčiau, kai pradinis atramų išdėstymas buvo generuojamas ekspertinės programos. Tačiau pasiūlytu algoritmu galima ištirti didelį skaičių pradinių paskirstymų, taip didinant tikimybę rasti geresnį sprendinį.

2 skyrius

Šakų ir režių algoritmo šablonas

Šiame skyriuje nagrinėjamas *šakų ir režių* algoritmas, jo lygiagretinimo galimybės ir siūloma *šakų ir režių* algoritmo lygiagretinimo priemonė.

2.2 poskyryje analizuojamas apibendrintas nuoseklusis *šakų ir režių* algoritmas, 2.3 poskyryje aprašytas nuosekliojo algoritmo šablonas. 2.4 poskyryje tiriami lygiagretieji *šakų ir režių* algoritmai. 2.5 poskyryje nagrinėjamas lygiagrečiojo algoritmo šablonas, pateikti eksperimentų rezultatai. 2.6 poskyryje pateikiamas juodosios dėžės globaliosios optimizacijos algoritmo taikymas statinių pamatų optimizavimo uždaviniui ir modifikuoto algoritmo testavimas, naudojant šabloną.

2.1 Įvadas

Daug inžinerinių, fizikos, ekonomikos ir kitų sričių uždavinių gali būti formuluojami kaip optimizavimo uždaviniai, kur reikia rasti mažiausią tiriamos funkcijos reikšmę. Matematiškai uždavinį galima formuluoti taip:

$$f^* = \min_{X \in D} f(X), \quad (2.1)$$

čia $f(X)$ – tiriamą funkciją, X – minimizavimo kintamieji, D – paieškos sritis. Paprastai siekiama surasti ne tik minimalią funkcijos reikšmę, bet ir argumentus, su kuriais ši reikšmė yra pasiekama $X^* : f(X^*) = f^*$.

Optimizavimo uždaviniai būna didelės dimensijos, reikalauja daug skaičiavimų, bei kompiuterinių resursų. Yra kuriami nauji algoritmai, dažniausiai euristiniai, spartinantys tokių uždavinių sprendimą. Kita vystymosi kryptis – populiarių tikslių algoritmų lygiagretinimas.

Tiriamas *šakų ir režių* (angl. *branch-and-bound*) metodas, kuris nusako algoritmų, ieškančių tikslaus optimizavimo uždavinio sprendinio, principus. Konkrečiam uždaviniui šie principai įgyvendinami skirtingai [14]. Dėl šios savybės galima formuoti *šakų ir režių* algoritmo šabloną. *Šakų ir režių* metodu remiasi

daug optimizavimo algoritmų, pavyzdžiui, šakų ir atmetimo (angl. *branch-and-cut*), alfa-beta genėjimo (angl. *alpha-beta pruning*). Toliau skyriuje bus kalbama apie *šakų ir režijų* algoritmą ir jo modifikacijas.

Šio darbo tikslas – sukurti algoritmo šabloną, leidžiantį vartotojui eksperimentuoti su nuosekliaisiais algoritmais ir supaprastinantį lygiagrečiųjų *šakų ir režijų* programų kūrimą. Įrankis taip pat palengvina naujų algoritmų, paremtų *šakų ir režijų* algoritmo logika, realizavimą.

Yra sukurta nemažai optimizavimo algoritmų lygiagretinimo įrankių. Panašios idėjos įdiegtos tokiuose įrankiuose ir programų paketuose:

- PICO [22] – objektiškai orientuota *šakų ir režijų* algoritmų lygiagretinimo sistema. Tai C++ sukurta sistema, naudojanti MPI duomenų perdavimo biblioteką ir komercinius CPLEX, XPRESS-MP paketus. Nuoseklusis ir lygiagretusis šio paketo lygmenys yra atskirti. Sistema reikalauja daug papildomos informacijos apie uždavinį, detalių, tačiau taip gaunama lanksti ir apibendrinta sistema. Lygiagrečiame algoritme procesai grupuojami į klasterius, kiekviename klasteryje vykdomas *šakų ir režijų* algoritmas. Lygiagretinimas vykdomas naudojant *šeimininkas – darbininkai* algoritmo logiką. Šis komercinis produktas 2000 metais sukurtas Sandia laboratorijoje (JAV), toliau plečiamas ir taikomas.
- PPBB-Lib [59] biblioteka, paruošta C kalba, naudoja PVM duomenų perdavimo biblioteką. Bibliotekoje yra funkcijos algoritmui lygiagretinti ir balansuoti procesorių apkrovą. Lygiagretinama į nuosekliają programą, paruoštą pagal įrankio reikalavimus, įterpianč kreipinius į PPBB-Lib biblioteką ir pritaikius duomenų struktūras. Sukurtas 1996 metais Padeborn universitete, Vokietijoje.
- PUBB C/C++ kalbos biblioteka *šakų ir režijų* algoritmo lygiagretinimui, duomenų perdavimui naudoja PVM biblioteką. Atskirti vartotojo ir įrankio lygmenys, programas galima vykdyti ir nuosekliai. Lygiagrečiai programai, vartotojas turi papildomai paruošti duomenis taip, kad juos būtų galima perduoti PVM funkcijomis. Lygiagretinama naudojant *šeimininkas – darbininkai* arba duomenų lygiagretinimo paradigmas. Pradinis įrankio variantas sukurtas 1999 metais Japonijoje [50], 2003 metais paskelbtos jo modifikacijos [51].
- MALLBA šablonų biblioteka, skirta *šakų ir režijų* algoritmo lygiagretinimui sprendžiant kombinatorinius uždavinius. Tai objektiškai orientuotos C++ programos, naudojančios MPI. Įrankį sudaro keletas lygiagrečiųjų algoritmų šablonų. Įrankis pradėtas kurti 2001 metais Barselonoje [1].
- BOB įrankis *šakų ir režijų* metodika paremtų algoritmų lygiagretinimui. 1995 pristatyta įrankio versija C kalba, duomenų perdavimui naudojanti PVM

[17]. Vėliau įrankis praplėstas naudojant C++ ir MPI duomenų perdavimo biblioteką. Šiuo metu į BOB diegiami nauji algoritmai [16], lygiagretinimo dalis vystoma su specialia procesorių apkrovos tvarkymo sistema Athapascan [10]. Įrankis sukurtas PRiSM laboratorijoje, Prancūzijoje.

2.2 Nuoseklusis šakų ir rėžių algoritmas

Šakų ir rėžių algoritmas tinka tiek minimizavimo, tiek maksimizavimo uždaviniams spręsti. Toliau bus aptariamas *šakų ir rėžių* algoritmo taikymas minimizavimo uždaviniams. Algoritmo idėja – nustatyti sprendinių sritis, kuriose tikrai nebus optimalaus uždavinio sprendinio ir tų sričių toliau netirti. Jei tai pavyksta, uždavinys išsprendžiamas greičiau, jei ne – algoritmas veikia kaip pilno perrinkimo algoritmas. Sritys, kuriose negali būti optimalaus sprendinio, nustatomos pradinį uždavinį skaidant į mažesnius, įvedant papildomus ribojimus sprendiniui taip, kad mažesnę uždavinį būtų lengviau išspręsti arba įvertinti. Skaidymą galima vadinti algoritmo šakos žingsniu. Įvertinimui skaičiuojami funkcijos reikšmių toje srityje rėžiai, apatinis ir viršutinis. Kuo tiksliau skaičiuojamas rėžis, tuo greičiau bus atmetamos neperspektyvios sritys. Jeigu užduoties srityje apatinis minimizuojamos funkcijos reikšmių rėžis yra didesnis už žinomą viršutinį minimalios reikšmės rėžį, vadinasi, toliau šios srities galima netirti, nes ten optimalaus sprendinio nebus. Viršutiniu rėžiu dažnai imama mažiausia tuo metu žinoma funkcijos reikšmė. Dar neištirtos sritys saugomos sąrašė. Tikslus sprendinys randamas, kai nebelieka neištirtų sričių. Bet kuriuo sprendimo momentu yra žinomas geriausias iki to laiko rastas sprendinys ir dar neištirtos sprendinių sritys. Dėl to algoritmas gali būti naudojamas kaip euristika, pabaigos požymiu laikant algoritmo vykdymo laiką.

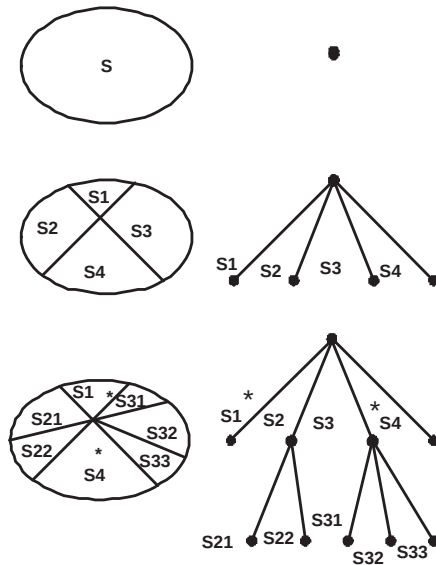
Algoritmas taikomas tiek kombinatorikos uždaviniams, kur paieškos sritis yra diskreti, tiek globaliosios optimizacijos uždaviniams, kur tiriamą funkciją yra netiesinė tolydaus kintamojo funkcija. Šiuo atveju, uždavinys sprendžiamas duotu tikslumu.

Patogu *šakų ir rėžių* algoritmą vaizduoti kaip paieškos medį, kur mazgas žymi uždavinį, mazgo vaikai – šio uždavinio skaidymą (žr. 2.1 paveikslėlį). Žvaigždute pažymėtose srityse nustatyta, kad nebus optimalaus sprendinio.

Pagrindiniai algoritmo žingsniai:

1. *Srities pasirinkimas*. Srities pasirinkimas nepriklauso nuo sprendžiamo uždavinio. Reikia apibrėžti taisyklę, kokia tvarka bus tiriamos dar neištirtos sritys. Populiariausios taisyklės yra tokios:

- (a) *Geriausio rėžio taisyklė* (angl. *Best first search*, BFS). Pasirenkama sritis su mažiausia rėžio reikšme. Neištirtų užduočių sąrašas gali būti



2.1 pav. Paieškos medis

formuojamas naudojant piramidės arba prioritetinės eilės duomenų struktūras.

- (b) *Naujausio režio taisyklė* (angl. *Last first search, LFS*). Pasirenkama vėliausiai suformuota sritis. Mazgas su žemiausiu lygmeniu medyje yra dalinamas ir tiriamas toliau. Ši taisyklė įgyvendinama FILO duomenų struktūromis, pavyzdžiui, steku.
- (c) *Apžvalgos taisyklė* arba paieška į plotį (angl. *Breadth first search, BrFS*). Pasirenkama anksčiausiai suformuotos sritys. Ši taisyklė įgyvendinama FIFO struktūromis, pavyzdžiui, eile.
- (d) *Pagerintas pasirinkimas* (angl. *Improved selection*). Jis yra paremtas euristiniu [39, 15] arba tikimybinio [21] kriterijumi. Sąrašas gali būti formuojamas naudojant piramidę arba prioritetinę eilę.

Galimos ir šių strategijų kombinacijos, kitais parametrais paremtos užduočių parinkimo taisyklės. Srities parinkimo tvarka įtakoja algoritmo efektyvumą sprendžiant konkretų uždavinį.

2. *Šakos žingsnis*. Šakos žingsnyje tyrimų sritis dalinama į smulkesnes dalis. Šis žingsnis priklauso nuo sprendžiamo uždavinio tipo. Dažnai dalinimas realizuojamas įvedant papildomus ribojimus kintamiesiems arba priskiriant jiems reikšmes (kombinatoriniame optimizavime). Pavyzdžiui, globaliajame optimizavime sritį galima dalinti į hiper-stačiakampius, simpleksus ir kt. Šakos žingsnyje vieną užduotį daliname į dvi ar daugiau užduočių. Konst-

ruojant dalinimą, tiriamas konvergavimas: reikalaujama, kad užduoties dydis mažėtų ir sprendinių skaičius užduotyje po padalinių būtų baigtinis. Vengiama persidengiančių dalinių, kai sprendinys gali priklausyti kelioms sritims. Skaičiavimų pradžioje turi būti suformuota pradinė sritis arba padalinimas – atliktas srities padengimas.

3. *Rėžio skaičiavimas.* Rėžio taisyklės apibrėžia, kaip rasti minimizuojamos funkcijos minimalių reikšmių viršutinį $UB(D)$ ir srities apatinį $LB(I)$ rėžius. Čia D – visa uždavinio sritis, I – pasirinkta tiriamą sritis. Bendruoju atveju, rėžio skaičiavimas priklauso nuo sprendžiamo uždavinio, tačiau rėžio reikšmės turi artėti prie optimalaus sprendinio smulkinant skaidymą. Riboję sprendiniui apatinio ir viršutinio rėžio reikšmės sutampa su optimalia reikšme. Viršutiniu tiriamos srities D rėžiu $UB(D)$ gali būti laikoma mažiausia žinoma funkcijos reikšmė. Globaliajame optimizavime apatinis rėžis $LB(I)$ tiriamojame srityje I gali būti nustatomas naudojant Lipšico konstantas arba intervalinę aritmetiką.

Šakų ir rėžių algoritmams svarbus ir pradinis sprendinio inicijavimas: kuo tikslesne reikšme inicijuojamas sprendinys skaičiavimų pradžioje, tuo greičiau yra atmetamos neperspektyvios sritys, todėl dažnai pradinis sprendinys inicijuojamas nesudėtinga euristika apskaičiuotu sprendiniu.

Globaliajame optimizavime sprendinys gaunamas tam tikru tikslumu, kai sritis po skaidymo tampa pakankamai maža, arba skirtumas tarp rėžių tampa mažesnis už duotą tikslumą. Kombinatoriniuose uždaviniuose sprendinys apibūdinamas individualiai, juos galima interpretuoti kaip suformuoto paieškos medžio lapus.

2.1 pateikiame apibendrintą *šakų ir rėžių* algoritmą. Čia L – sričių sąrašas, S — sprendinys, $UB(D)$ ir $LB(I_i)$ reiškia viršutinį ir apatinį minimizuojamos funkcijos reikšmės rėžį visoje srityje D arba jos dalyje I_i .

Galimos nekantri ir tingi algoritmo strategijos, kurios apibrėžia, kokia tvarka vykdomi pagrindiniai algoritmo šakos ir rėžio žingsniai. 2.2 pateikiamas tingios strategijos apibendrintas *šakų ir rėžių* algoritmas.

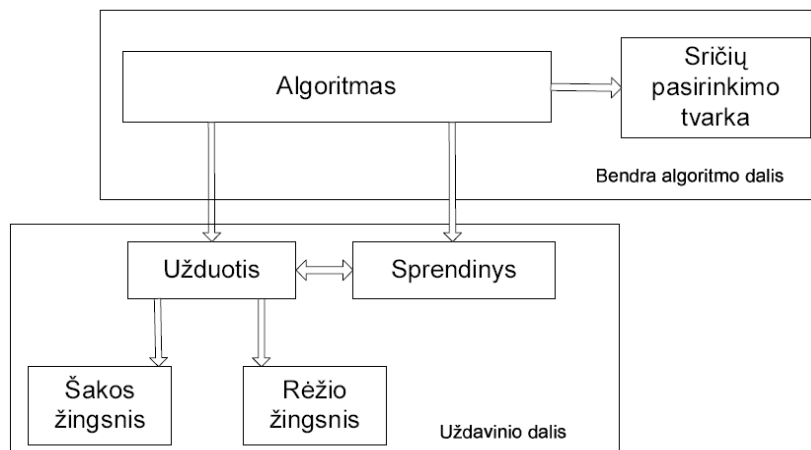
Algoritmas 2.1 *Apibendrintas šakų ir rėžių algoritmas. Nekantri strategija*

1. Pradinę sritį D daliname į $L = \{L_j | D \subseteq \bigcup L_j, j = \overline{1, m}\}$. Naudodami padengimo taisyklę,
2. $S = \emptyset, UB(D) = \infty$.
3. **while** (sričių sąrašas nėra tuščias $L \neq \emptyset$) **do**
4. Pasirinkti $I \in L$ naudojant pasirinkimo taisyklę, išimti I iš L .
5. **if** ($LB(I) < UB(D) + \epsilon$) **then**
6. Skaidyti I į p dalių I_j naudojant šakos taisyklę.
7. **for all** ($I_j, j = \overline{1, p}$) **do**
8. Rasti $UB(I_j \cap D)$ ir $LB(I_j)$ naudojant rėžio taisyklę.
9. $UB(D) = \min(UB(D), UB(I_j \cap D))$.
10. **if** ($LB(I_j) < UB(D) + \epsilon$) **then**
11. **if** (I_j yra sprendinys) **then** $S = I_j$.
12. **else** $L = \{L, I_j\}$.
- end if**
- end if**
- end for**
- end if**
- end while**

Algoritmas 2.2 *Apibendrintas šakų ir rėžių algoritmas. Tingi strategija*

1. Pradinę sritį D daliname į $L = \{L_j | D \subseteq \bigcup L_j, j = \overline{1, m}\}$. Naudodami padengimo taisyklę,
2. $S = \emptyset, UB(D) = \infty$.
3. **while** (sričių sąrašas nėra tuščias $L \neq \emptyset$) **do**
4. Pasirinkti $I \in L$ naudojant pasirinkimo taisyklę, išimti I iš L .
5. Rasti $UB(I \cap D)$ ir $LB(I)$ naudojant rėžio taisyklę.
6. $UB(D) = \min(UB(D), UB(I \cap D))$.
7. **if** ($LB(I) < UB(D) + \epsilon$) **then**
8. **if** (I yra sprendinys) **then** $S = I$.
9. **else**
10. Skaidyti I į p dalių I_j naudojant šakos taisyklę.
11. **for all** ($I_j, j = \overline{1, p}$) **do**
12. $L = \{L, I_j\}$.
- end for**
- end if**
- end if**
- end while**

Šių strategijų taikymo efektyvumas priklauso nuo sprendžiamo uždavinio. Naudojant nekantrią strategiją, režiai yra skaičiuojami prieš įterpiant neištirtą uždutį į sąrašą, taip garantuojant, kad sąraše nebus užduočių, kurios jau jų sudarymo momentu buvo neperspektyvios. Tingi strategija atideda režio skaičiavimą, tokiu būdu užduotys atmetamos prieš šakos žingsnį.



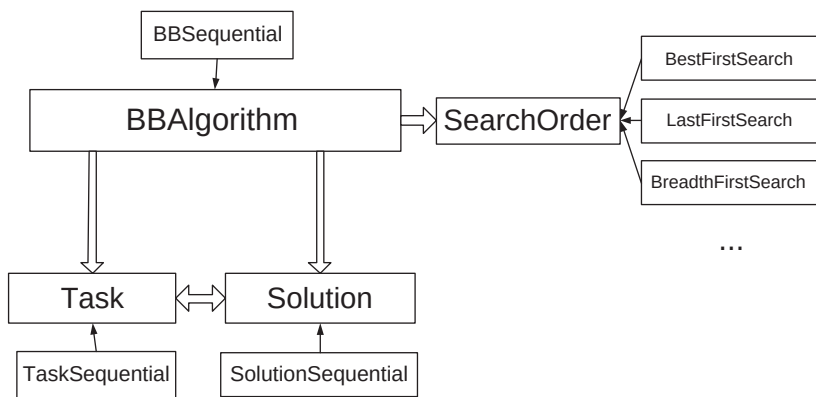
2.2 pav. Šakų ir režių algoritmo schema

Šakų ir režių metodo apibrėžimas yra bendras įvairiems optimizavimo uždaviniams, dėl to, atskyrus šias dalis galima formuoti algoritmo šabloną. Algoritmas, tiriamos srities parinkimo taisyklės gali būti taikomi įvairiems optimizavimo uždaviniams. Rėžio ar šakos žingsniai gali būti apibendrinti tam tikroms klasėms sprendžiamų uždavinių, pavyzdžiui, Lipšico funkcijų optimizavimui.

2.3 Nuosekliojo šakų ir režių algoritmo šablonas

Šakų ir režių algoritmo šablono idėjos aprašytos [9], jos naudotos daugelio įrankių kūrime. Remiantis 2.2 schema sukurtas *šakų ir režių* algoritmo šablonas. Jame įdiegtas apibendrintas algoritmas, kuris atlieka veiksmus su objektais *Užduotis* ir *Sprendinys*. Šie objektai priklauso nuo sprendžiamo uždavinio ir yra sukuriami šablono vartotojo. Algoritmo šablone realizuotos trys pasirinkimo taisyklės: geriausio režio, naujausio režio ir apžvalgos. Tokiu būdu kurdamas nuosekliąją programą vartotojas gali testuoti, kuris eilės tvarkymo algoritmas geriausiai tinka konkrečiam uždaviniui.

Šablone siekiama sujungti įvairius *šakų ir režių* algoritmų variantus ir leisti vartotojui nesunkiai eksperimentuoti su nuosekliais algoritmais. Į šabloną įtraukti standartiniai metodai, paliekant galimybę šabloną praplėsti. 2.3 paveikslėlyje pateikiama *šakų ir režių* algoritmo šablono C++ klasių schema.



2.3 pav. Šakų ir rėžių nuosekliųjų algoritmų šablono klasių schema

BBAAlgorithm – tai apibendrinta *šakų ir rėžių* algoritmų klasė. Naudojant objektinio programavimo paveldimumo technologiją, galima gauti įvairius *šakų ir rėžių* principu besiremiančius algoritmus. Eksperimentuose naudotas nekantriosios strategijos 2.1 algoritmas.

Task – užduoties klasė, kurioje turi būti apibrėžti užduoties arba tyrimų srities inicializavimas, srities dalinimas (šakos žingsnis), sprendinių toje srityje rėžio skaičiavimas, taip pat metodas parodantis, ar šioje užduotyje suformuojamas sprendinys.

Solution klasėje apibrėžiamas sprendžiamo uždavinio sprendinys.

SearchOrder – užduočių tyrimo tvarkos klasė, realizuojamos parinkimo taisyklės. Užduočių tyrimo tvarka gali stipriai įtakoti algoritmo vykdymo efektyvumą. Apibrėžiami įterpimo į sąrašą, paėmimo iš sąrašo metodai, taip pat metodas nustatyti, kad sąrašas tuščias.

Dėl tokios klasių schemos, įrankiu galima ne tik spręsti uždavinius naudojant *šakų ir rėžių* algoritmą, testuoti įvairias užduočių parinkimo taisykles, bet ir testuoti naujus algoritmus.

2.3.1 Uždavinių pavyzdžiai

Šakų ir rėžių algoritmų šablonas gali būti taikomas tiek kombinatoriniams, tiek globaliosios optimizacijos algoritmams. Įrankį testavome sprenddami šių klasių uždavinius.

Keliaujančio pirklio uždavinys – aplankyti visus duotus miestus taip, kad viso maršruto svoris būtų mažiausias. Maršruto svoriu gali būti įvardintas jo ilgis, kaina, trukmė. Kiekvienas miestas aplankomas tik vieną kartą ir maršrutas turi baigtis tame pačiame mieste, iš kurio prasidėjo. Uždavinys vadinamas simetriniu, jei kelio svoris nepriklauso nuo judėjimo krypties.

Duotas grafas $G = \{V, E\}$ su n viršūnių $V = \{v_i, i = 1, \dots, n\}$ ir m briaunų $E = \{e_{ij} = (v_i, v_j)\}$, $|E| = m$, briaunos svoris $W(e_{ij})$ apibrėžia kelio svorį

$c_{ij} = W(e_{ij})$ kiekvienai briaunai $e(i, j) \in E$

$$\min \sum_{i,j} c_{ij} x_{ij}, \quad (2.2)$$

su sąlyga

$$\sum_{j:(i,j) \in A} x_{ij} = 1, i = 0, 1, \dots, n-1 \quad (2.3)$$

$$\sum_{i:(i,j) \in A} x_{ij} = 1, i = 0, 1, \dots, n-1 \quad (2.4)$$

$$x_{ij} \in \{0, 1\}, \quad i, j \in \{0 \dots n-1\}. \quad (2.5)$$

Tai NP-sudėtingumo kombinatorinio optimizavimo uždavinys, kuris sprendžiamas šiais būdais:

- Tiksliais algoritmais. Šiais algoritmais randamas tikslus uždavinio sprendinys, tačiau skaičiavimų apimtis eksponentiškai didėja, didėjant miestų skaičiui. *Šakų ir rėžių* algoritmas yra tikslaus algoritmo pavyzdys.
- Apytiksliais algoritmais, kuriais randamas artimas optimaliam sprendinys ir įvertinama jo kokybė.
- Euristiniai algoritmai, reikalaujantys mažiausiai skaičiavimų, tačiau nėra garantuojama, kad jų sprendiniai yra artimi optimaliam. Tai lokalsios paieškos algoritmai: artimiausio kaimyno, fiktyvios temperatūros paskirstymo (angl. *simulated annealing*).

Sprendimo metodų realizacijos ir uždavinio taikymas praktikoje plačiau aptariami [60, 57]. Aptarsime realizuoto algoritmo pagrindines detales.

Skaičiavimo eksperimentuose $n = 20$. Pradinis srities padengimas – pasirenkamas pradinis miestas. Šakos žingsnis – pasirenkamas kitas miestas, į kurį einama eiliniame žingsnyje, apatinis rėžis – dabartinis kelio ilgis, sprendinys, kai į maršrutą įtraukti visi miestai.

Naudojant šablona, keliaujančio pirklio uždavinys geriausio rėžio taisykle išsprendžiamas ištyrus 50851 užduotis, naujausios ribos taisykle išsprendžiamas ištyrus 99278 užduotis, apžvalgos taisykle – 348990 užduotis.

Lipšico funkcijos minimizavimo uždavinys.

Tai globaliosios optimizacijos uždavinys. Tegul f yra Lipšico funkcija, $x \in D \subset \mathbb{R}^2$, tenkinanti nelygybę

$$|f(x_1) - f(x_2)| \leq L\|x_1 - x_2\| \quad \text{visiems } x_1, x_2 \in D, \quad (2.6)$$

čia L yra Lipšico konstanta. Tegul D yra stačiakampis $(a_1, b_1) \times (a_2, b_2)$. Šakos žingsnyje stačiakampis dalinamas į du, padalinant ilgiausią kraštinę į dvi lygias dalis. Funkcijos $f(x)$, $x \in P$ apatinis rėžis μ skaičiuojamas taip:

$$\mu(P) = f(m) - \frac{1}{2}L\delta(P), \quad (2.7)$$

čia m – kraštinės vidurio taškai. $\delta(P)$ yra P diametras:

$$\delta(P) = \left(\sum_{j=1}^2 (b_j - a_j)^2 \right)^{1/2}. \quad (2.8)$$

Tirtos įvairios funkcijos iš [36]:

$$f_1(x) = \frac{1}{2}x_1^2 - 9x_1 + 20 + \frac{1}{2}x_2^2 - 9x_2 + 20, D = [0, 10] \times [0, 10]. \quad (2.9)$$

Šekelio funkcija, kur n - dimensija, m - minimumų skaičius:

$$f(x) = \sum_{i=1}^m \frac{1}{c_i + \sum_{j=1}^n (x_j - a_{ji})^2} \quad (2.10)$$

kai $n = 2$ ir $m = 2$:

$$f_2(x) = \frac{-1}{((x_1 - 4)^2 + (x_2 - 4)^2 + 0, 7)} - \frac{1}{((x_1 - 2, 5)^2 + (x_2 - 3, 8)^2 + 0, 73)}, \quad (2.11)$$

kuri buvo minimizuojama srityje $D = [0, 10]^2$, tikslumu 0,0001. Naudojant geriausio rėžio taisyklę uždavinys išspręstas ištyrus 788 užduotis, kai naudojama naujausios ribos taisyklė – 694053, o kai apžvalgos taisyklė – 3327 užduotis.

Lipšico konstanta praktiniams uždaviniams dažnai nėra žinoma.

Šablone įdiegti apibendrinto rėžio ir šakos žingsnių skaičiavimo algoritmus, kai sritis yra padengiama stačiakampiais ar simpleksais. Tai leidžia įrankį naudoti ir naujiems algoritmams bandyti.

2.4 Lygiagretieji šakų ir rėžių algoritmai

Išskiriami įvairūs lygiagrečiųjų *šakų ir rėžių* algoritmų būdai: pagal tai, kuri algoritmo dalis vykdoma lygiagrečiai, kiek neištirtų užduočių sąrašų kuriama, ar procesoriai apsieičia duomenimis, dirba sinchronizuotai ar asinchroniškai.

[31] aprašyti trys pagrindiniai būdai kurti lygiagretųjį *šakų ir rėžių* algoritmą:

- *1 tipo lygiagretumas*. Tai lygiagretumas vykdamas operacijas su kiekviena užduotimi atskirai. Pavyzdžiui, tiriamos funkcijos reikšmę skaičiuoti ar rėžio žingsnį vykdyti lygiagrečiai. Tokio tipo lygiagretumas neturi įtakos bendrai *šakų ir rėžių* algoritmo struktūrai ir priklauso tik nuo duoto uždavinio.
- Vykdamas *2 tipo lygiagretumą šakų ir rėžių* paieškos medis konstruojamas lygiagrečiai, t.y. lygiagrečiai vykdomi veiksmai su keliomis užduotimis. Tokio tipo lygiagretumas gali įtakoti algoritmo vykdymą ir užduočių tyrimo tvarką. Algoritmas gali būti sinchronizuotas arba ne sinchronizuotas, turėti vieną ar kelias neištirtų užduočių eiles.
- *3 tipo lygiagretumas* reiškia, kad keli *šakų ir rėžių* paieškos medžiai yra kuriami vienu metu. Šiuos medžius charakterizuoja skirtingos operacijos (rėžio, šakos, atmetimo, pasirinkimo iš eilės). Rezultatai, gauti konstruojant vieną medį gali būti panaudoti konstruojant kitą.

Plačiau aptarsime 2 tipo lygiagrečiuosius *šakų ir rėžių* algoritmus, kurių pagrindu kuriamas algoritmo šablonas. *Šakų ir rėžių* algoritmo lygiagretinimo žingsnių bendri principai yra šie:

- *Skaidymas į užduotis*. Bet kurios srities tyrimas gali būti vykdomas nepriklausomai nuo kitų. Dėl to skaidymas gali būti atliekamas pradinę sritį dalinant į dalis arba vykdamas šakos žingsnį.
- *Komunikacijų nustatymas*. Komunikacijų modelis nėra griežtai apibrėžtas. Tačiau lygiagrečiuosiuose algoritmuose duomenų perdavimų reikia skaičiavimų inicializavimui ir galutinio rezultato išrinkimui. Vykdamas nuoseklųjį algoritmą, naudojami rezultatai, gauti tiriant kitas sritis. Konstruojant analogišką nuosekliajam lygiagretųjį algoritmą, reikia realizuoti reikalingų duomenų pasikeitimus. Galimi algoritmai ir be duomenų pasikeitimų, kai nenaudojama informacija iš kitų procesorių. Komunikacijos gali būti tiek sinchronizuotos, tiek asinchroninės.
- *Užduočių apjungimas*. Vykdamas *šakų ir rėžių* algoritmą konstruojamo medžio mazgai tėvai apjungia vaikus. Tokiu būdu galima apjungti užduotis į didesnes. Norint įvertinti apjungtų užduočių dydį, reikalingas nuo uždavinio priklausantis tyrimas.

- *Priskyrimas*. Galimi keli užduočių priskyrimo modeliai: naudojant centralizuotą, decentralizuotą arba mišrų užduočių sąrašus.

[47] aprašytas 2 tipo lygiagrečiųjų algoritmų skirstymas į

- *Horizontaliuosius*, kai vienas procesorius tvarko užduočių sąrašą ir skirsto jį kitiems procesoriams, kurie atlieka šakos ir režio žingsnius. Toks algoritmas gal būti realizuotas naudojant *šeimininkas – darbininkai* algoritmą.
- *Vertikaliuosius*, kai pradinis uždavinys padalinamas visiems procesoriams, o šie toliau vykdo *šakų ir režių* algoritmą jiems paskirtose dalyse. Tokio algoritmo realizavimas panašus į duomenų lygiagretumo SIMD algoritmus.

2 tipo lygiagretusis *šakų ir režių* algoritmas, naudojantis vieną užduočių sąrašą gali būti realizuotas vykdant *šeimininkas – darbininkai* algoritmą komunikacijoms ir užduočių paskirstymui. *Šakų ir režių* algoritmo neištirtų užduočių sąrašas yra laikomas darbų sąrašu. Šeimininko procesas ima užduotį iš sąrašo pagal pasirinkimo taisyklę ir siunčią ją šiuo metu neturinčiam darbo darbininko procesui. Darbininko procesas apskaičiuoja režius ir šakos žingsnį. Tada siunčia naujas užduotis atgal šeimininko procesui, kartu su nauju viršutiniu režiu, jei toks rastas. Šeimininko procesas vykdo 2.3 A algoritmą, darbininko procesas – 2.3 B.

Tokio algoritmo vykdymo metu visos generuojamos užduotys yra viename procese, tvarkomos ta pačia eile, kaip ir nuosekliojo algoritmo atveju. Visiems procesams žinomas mažiausias viršutinis režis. Tačiau algoritmo vykdymui reikia daug duomenų perdavimų, be to naudojant daugiau procesorių, šeimininko procesas tampa stipriai apkrautas.

Algoritmas 2.3 A *Apibendrintas šakų ir režijų algoritmas. Šeimininko dalis*

1. Padalinti pradinę sritį D į $L = \{L_j | D \subseteq \bigcup L_j, j = \overline{1, m}\}$ naudojant padengimo taisyklę.
2. $S = \emptyset, UB(D) = \infty$.
3. Sukurti laisvų Darbininkų sąrašą $SL = \{S_k, k = \overline{1, n}\}$
4. **for all** ($SL_k, k = \overline{1, n}$) **do**
5. Pasirinkti $I \in L$ naudojant pasirinkimo taisyklę, išimti I iš L .
6. Pasirinkti laisvą procesą S_k , išimti S_k iš SL
7. Siųsti $I, UB(D)$ procesui S_k
8. **end for**
9. **while** (sričių sąrašas nėra tuščias $L \neq \emptyset$ ir ne visi Darbininko procesai yra laisvi) **do**
10. Gauti $\{I_j, LB(I_j), UB(I_j)\}, j = 1, \dots, J$ iš Darbininko proceso S_k
11. Įterpti Darbininko procesą S_k į laisvų Darbininko procesų sąrašą SL
12. **for all** ($I_j, j = \overline{1, J}$) **do**
13. $UB(D) = \min(UB(D), UB(I_j \cap D))$.
14. **if** ($LB(I_j) < UB(D) + \epsilon$) **then**
15. **if** (I_j yra sprendinys **then**) $S = I_j$.
16. **else** $L := \{L, I_j\}$.
17. **end if**
18. **end for**
19. **for** (visiems darbininkams S_k iš SL) **do**
20. **if** (sričių sąrašas nėra tuščias $L \neq \emptyset$) **then**
21. Pasirinkti $I \in L$ naudojant pasirinkimo taisyklę, išimti I iš L .
22. **if** ($LB(I) < UB(D) + \epsilon$) **then**
23. išimti S_k iš SL
24. siųsti $\{I, UB(D)\}$ procesui S_k
25. **end if**
26. **end if**
27. **end for**
28. **end while**
29. Siųsti *Pabaigos* požymį visiems Darbininko procesams

Algoritmas 2.3 B *Apibendrintas šakų ir režijų algoritmas. Darbininko dalis*

1. **while** (negautas *Pabaigos* požymis) **do**
2. Gauti $\{I, UB(D)\}$ iš Šeimininko proceso
3. Skaidyti I į p dalių I_j naudojant šakos taisyklę
4. Rasti $UB(I_j \cap D)$ ir $LB(I_j)$ naudojant režio taisyklę.
5. **if** ($LB(I_j) < UB(D) + \epsilon$) **then**
6. įterpti $I_j, LB(I_j), UB(I_j \cap D)$ į perduodamą pranešimą M
7. **end if**
8. Siųsti M Šeimininko procesui
9. **end while**

Detaliau aptarsime *šakų ir režijų* lygiagretinimo įrankyje įdiegtus lygiagrečiuosius algoritmus.

Įdiegti srities padalinimo su paskirstytu sričių sąrašu algoritmai. Vykdamas vertikalųjį 2 tipo lygiagretųjį algoritmą, arba algoritmą su pradinės srities padalinimu, galima sumažinti duomenų perdavimų bei naudotis savybe, kad užduotis apjungia tėvo mazgas paieškos medyje.

Išankstinis sprendinių srities padalinimas. Pradinė sritis yra dalinama į daug smulkesnių sričių, kurios yra priskiriamos procesoriams, ir vykdomas 2.1 algoritmas. Po pradinio padalinimo atsiradusių užduočių turėtų būti tiek, kiek naudojama procesorių. Smulkesnės sritys yra generuojamos iš padalintųjų algoritmo vykdymo metu vykdamas šakos žingsnį. Vėliau generuotos užduotys nėra perskirstomos. Duomenų perdavimų reikia pradiniam sričių padalinimui ir galutiniam rezultatų surinkimui. Tokiu būdu gaunamas paprastas algoritmas su statiskai paskirstytu užduočių sąrašu (angl. SJP, *Static distribution of job pool*).

Atsitiktinis tiriamos srities paskirstymas. *Šakų ir režijų* algoritmu generuojamos užduotys gali būti nevienodo sudėtingumo. Sudėtingumas šiuo atveju apibrėžiamas, kaip greitai sritis atmetama, jei joje nėra optimalaus sprendinio ar kiek užduočių iš jos vėliau generuojama. Siekiant gauti tolygų procesorių apkrovos paskirstymą tarp procesorių, pradinė tyrimų sritis yra dalinama į žymiai didesnių skaičių užduočių, negu yra procesorių. Sritis yra atsitiktinai padalinama tarp procesorių. Tikimasi, kad tokiu būdu kiekvienas procesorius gaus apytiksliai vienodą skaičių „sunkių“ ir „lengvų“ užduočių. Algoritmas vadinsis RJP (angl. *Random distribution of job pool*) – lygiagretusis *šakų ir režijų* algoritmas su atsitiktinai paskirstytu užduočių sąrašu. Panaši modifikacija buvo naudojama lygiagrečiuosiuose integravimo algoritmuose [13]. Sritis galima paskirstyti ir cikliška ar kitais algoritmais.

Mažiausio lokalaus minimumo funkcijos reikšmės pasikeitimas. Sritis yra eliminuojama iš tolesnių tyrimų lyginant apatinį režį su viršutiniu viso uždavinio režiu. Geriausia žinoma funkcijos reikšmė gali būti naudojama kaip viršutinis režis. Anksčiau apibrėžtuose lygiagrečiuose algoritmuose procesoriai turi tik jiems priskirtose srityse rastų minimumų reikšmes. Kai kuriose situacijose tai gali įtakoti lėtesnį sričių atmetimą. Procesoriai gali keistis rastais minimumais ir taip pagerinti algoritmo kokybę. Kai procesorius randa naują, geresnę, minimumo reikšmę jis siunčia visiems kitiems procesoriams. Norint nestabdyti vykdomų *šakų ir režijų* algoritmo skaičiavimų, tai vykdoma asinchroniškai. Ši algoritmo modifikacija vadiname SJP SE ir RJP SE, priklausomai nuo to, kaip dalinamas pradinis užduočių sąrašas.

Lygiagrečiųjų optimizavimo algoritmų specifika

Lygiagretieji optimizavimo algoritmai turi nereguliariai kintančią paieškos sritį. Reikia paminėti, kad dėl srities padalinimo jos tyrimo tvarka skiriasi algoritmą

vykdant nuosekliai ir lygiagrečiai. Sritys, anksti eliminuotos vykdant algoritmą nuosekliai, gali ilgiau likti neišmestos iš tyrimų vykdant algoritmą lygiagrečiai, ar atvirkščiai. Dėl to lygiagrečiajame algoritme gali išaugti iš viso ištirtų sričių skaičius.

Apibrėžiamos situacijos, būdingos lygiagretiesiems optimizavimo uždaviniams, vadinamos anomalijomis [58]:

- *nuostolinga anomalija*, kai pagreitėjimas naudojant p procesorių yra mažesnis už vieneta, nes ištiriama daugiau užduočių negu nuosekliauoju algoritmu.
- *lėtėjimo anomalija*, kai pagreitėjimas yra didesnis už vieneta, bet mažesnis už procesorių skaičių. Ištiriama daugiau užduočių.
- *pagreitėjimo anomalija*, kai pagreitėjimas yra didesnis už naudojamą procesorių skaičių. Tokia situacija susiklosto, kai uždaviniui išspręsti lygiagrečiai reikia tirti mažiau užduočių.

Dėl neapibrėžtai kintančios tyrimų srities sunku įvertinti lygiagrečiojo algoritmo kokybę. Siūloma skaičiuoti algoritmo pertekliško koeficientą (angl. *Search overhead factor*)

$$SOF = \frac{W_p}{W}, \quad (2.12)$$

čia W_p – darbo apimtis sprendžiant uždavinį p procesorių, W – nuosekliojo algoritmo darbo apimtis [34]. Darbo apimtį siūloma matuoti tirtų užduočių skaičiumi, darant prielaidą, kad kiekvienai užduočiai ištirti reikia tiek pat laiko. Daugelio lygiagrečiųjų *šakų ir rėžių* algoritmų pertekliško koeficientas yra didesnis už vieneta [34].

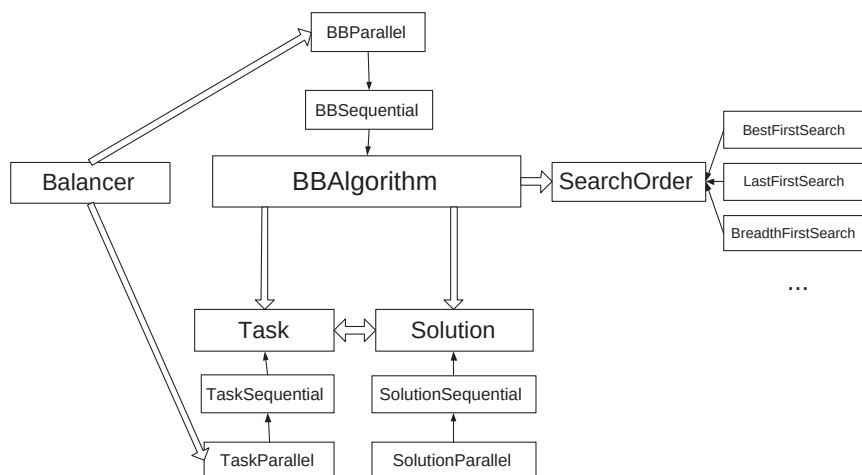
Atmetus prielaidą apie vienodą užduočių sudėtingumą, reikia įvertinti, kiek perspektyvi yra užduotis. Tokiu atveju naudojamas ne tik užduočių skaičius, bet ir užduočių rėžis ir užduočių vieta tyrimo eilėje [65]. Sudėtingesnis apkrovos mato vertinimas gali priklausyti nuo sprendžiamo uždavinio ir turi būti įdiegtas šablono vartotojo.

2.5 Lygiagrečiųjų šakų ir rėžių algoritmų šablonas

Lygiagretusis *šakų ir rėžių* algoritmo šablono variantas remiasi schema, parodyta 2.4 paveikslėlyje. Čia *Balancer* – 2.5.2 poskyryje aptariamas dinaminio apkrovos balansavimo modulis. Paruošus nuosekliają programą, galima nesunkiai ją išlygiagretinti, pasirinkus vieną iš pateiktų 2 tipo lygiagrečiųjų algoritmų: SJP, RJP, SJP SE, RJP SE ar algoritmus su dinaminio apkrovos balansavimu.

Duomenų perdavimui naudojama MPI biblioteka.

Naudojant SJP SE ir RJP SE algoritmus tarp procesorių yra persiunčiami *Solution* objektai, todėl vartotojui reikia papildomai aprašyti sprendinio pakavimo



2.4 pav. Lygiagrečiojo šakų ir rėžių algoritmo šablono schema

ir išpakavimo metodus. Šių metodų realizavimui nereikia papildomų MPI duomenų perdavimo bibliotekos žinių. Užtenka objekto duomenis perkelti į nurodyto tipo masyvus. Naudojant algoritmus su dinaminio apkrovos balansavimu reikia paruošti *Task* objektų pakavimo ir išpakavimo metodus.

Šablonas padeda parinkti geriausią lygiagretųjį algoritmą ir sričių pasirinkimo taisykles turimam uždaviniui. Paruošus Makefile nesunku pasirinkti įvairias algoritmų ir sričių pasirinkimo taisyklių kombinacijas. Įrankis testuotas įvairiuose lygiagrečiuose kompiuteriuose, įrankį galima plėsti.

Šakų ir rėžių algoritmo šablona galima rasti adresu <http://mmb.vtu.lt/BBT/>.

2.5.1 Skaičiavimo eksperimentų rezultatai

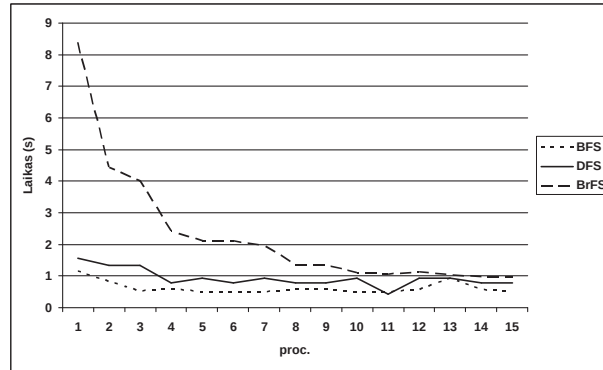
Eksperimentuose sprendėme 2.3.1 poskyryje aprašytus uždavinius. Eksperimentuose tirta:

- Įdiegtų algoritmų vykdymas su įvairiomis sričių pasirinkimo taisyklėmis.
- Kombinatorinių ir globaliosios optimizacijos uždavinių sprendimas įvairiais lygiagrečiaisiais algoritmais, SOF kitimas, procesorių apkrovos disbalansas.

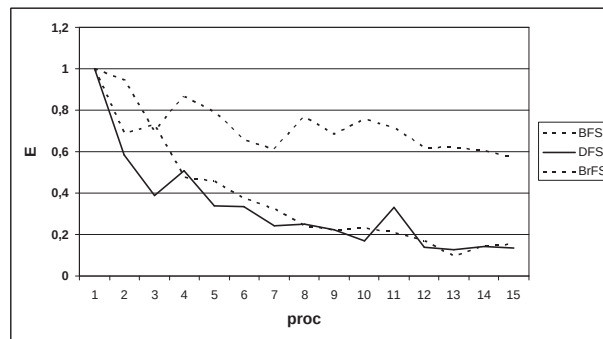
2.5 ir 2.6 paveikslėliuose pateikiami Lipšico $f_1(x)$ funkcijos minimizavimo uždavinio sprendimo SJP algoritmu, naudojant įvairias srities pasirinkimo taisykles, skaičiavimo vykdymo laiko (sekundėmis) ir efektyvumo grafikai.

Rezultatai rodo, kad naudojant apžvalgos taisyklę, lygiagrečiojo algoritmo efektyvumas didesnis palyginus su kitų paieškos taisyklių lygiagrečiųjų algoritmų

rezultatus, nors nuoseklesiems skaičiavimams ši taisyklė duoda blogiausią rezultatą. Eksperimentuose toliau tirsime geriausiai tinkamą nuoseklesiems skaičiavimams geriausio režio taisyklę ir SJP algoritmu efektyviausiai lygiagretinamą apžvalgos taisyklę.

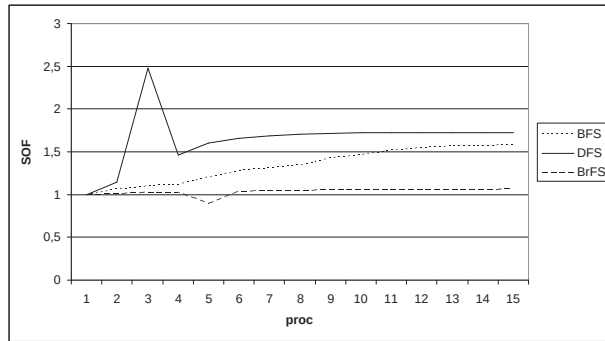


2.5 pav. $f_1(x)$ funkcijos minimizavimo, naudojant SJP algoritmą ir įvairias eiles skaičiavimo laikas klasteryje vilkas.vtu.lt

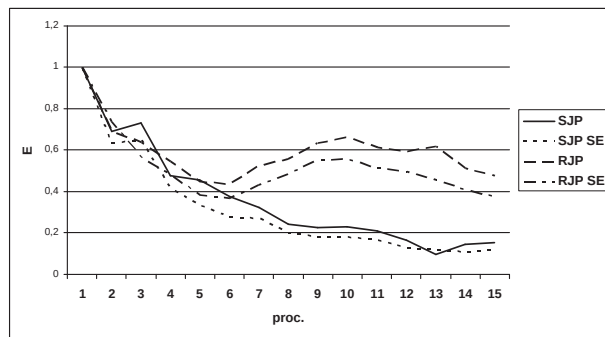


2.6 pav. $f_1(x)$ funkcijos minimizavimo, naudojant SJP algoritmą ir įvairias eiles efektyvumas klasteryje vilkas.vtu.lt

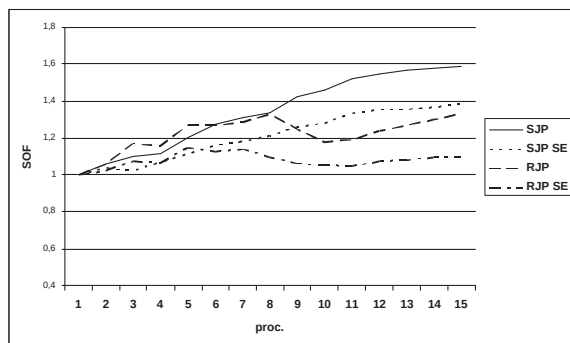
2.7 pav. pavaizduotas SOF parametras paaiškina 2.6 pav. pateiktus rezultatus: naudojant geriausio režio ir naujausio režio taisykles išauga uždavinio dydis, todėl algoritmas tampa neefektyviu. SOF faktorių galima mažinti lygiagrečiaisiais algoritmais su mažiausio lokalaus minimumo funkcijos reikšmės pasikeitimais. Ši faktorių taip pat keis ir algoritmai kitaip padalinantys pradinę tyrimų sritį procesoriams – RJP algoritmai.



2.7 pav. $f_1(x)$ funkcijos minimizavimo, naudojant SJP algoritmą ir įvairias eiles SOF faktorius klasteryje vilkas.vtu.lt

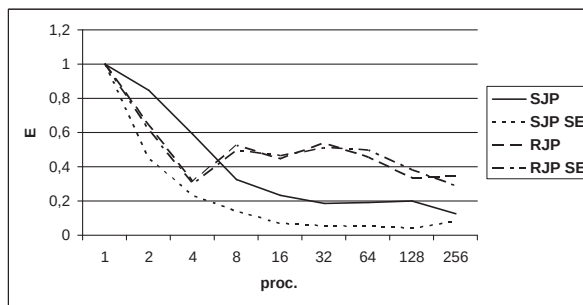


2.8 pav. $f_1(x)$ funkcijos minimizavimo efektyvumas gautas naudojant įvairius algoritmus ir geriausio režio taisyklę klasteryje vilkas.vtu.lt



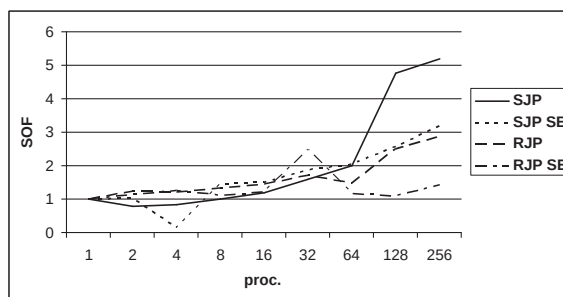
2.9 pav. $f_1(x)$ funkcijos minimizavimo SOF faktorius gautas naudojant įvairius algoritmus ir geriausio režio taisyklę klasteryje vilkas.vtu.lt

2.8 ir 2.9 paveikslėliuose pateikti rezultatai, gauti tiriant įvairių algoritmų vykdymą su ta pačia geriausio režio eilės tvarkymo taisykle. Rezultatai rodo, kad



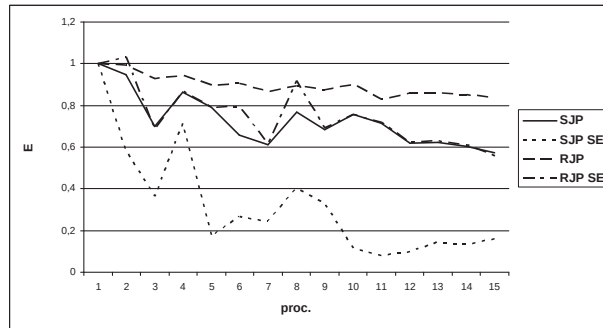
2.10 pav. $f_1(x)$ funkcijos minimizavimo efektyvumas gautas naudojant įvairius algoritmus ir geriausio režio taisyklę IDRIS centre

šiai funkcijai algoritmai su geriausio sprendinio apsikeitimu pagerina efektyvumą, nes sumažina tyrimų srities išaugimą, kuri rodo SOF faktorius. Naudojant RJP SE algoritmą SOF faktorius išauga nežymiai, tačiau algoritmo vykdymo efektyvumas yra mažesnis už RJP algoritmo dėl išaugančio procesorių apkrovos disbalanso.

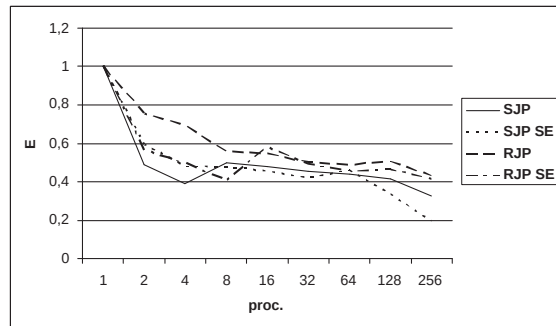


2.11 pav. $f_1(x)$ funkcijos minimizavimo SOF faktorius gautas naudojant įvairius algoritmus ir geriausio režio taisyklę IDRIS centre

2.10 ir 2.11 paveikslėlyje pateikiami rezultatai gauti IDRIS superkompiuterių centre. Eksperimentais tirta algoritmų vykdymas naudojant daugiau procesorių taip pat ir įrankio tinkamumas naudoti kitose sistemose. Vykdymo rezultatai išlieka panašūs ir naudojant daugiau procesorių: RJP algoritmų efektyvumas išlieka geresnis, naudojant daugiau procesorių.



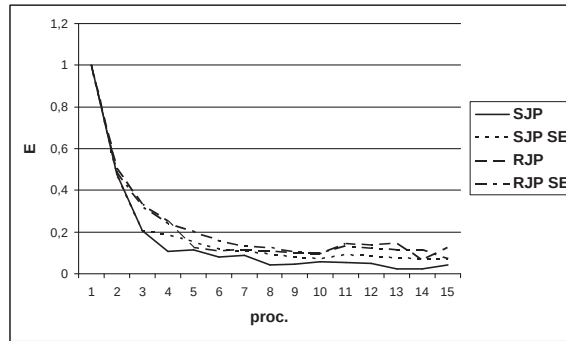
2.12 pav. $f_1(x)$ funkcijos minimizavimo efektyvumas gautas naudojant įvairius algoritmus ir apžvalgos taisyklę klasteryje vilkas.vtu.lt



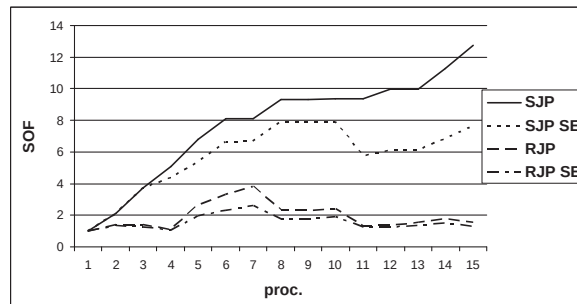
2.13 pav. $f_1(x)$ funkcijos minimizavimo efektyvumas gautas naudojant įvairius algoritmus ir apžvalgos taisyklę IDRIS centre

2.12 ir 2.13 paveikslėliuose pateikiami $f_1(x)$ funkcijos minimizavimo lygia-grečiųjų programų efektyvumas naudojant apžvalgos taisyklę. Efektyvumas iš-lieka didžiausias naudojant RJP algoritmą. Šiuo atveju algoritmai su geriausio sprendinio apsikeitimu nėra efektyvesni. Verta naudoti algoritmus su atsitiktiniu tiriamos srities paskirstymu.

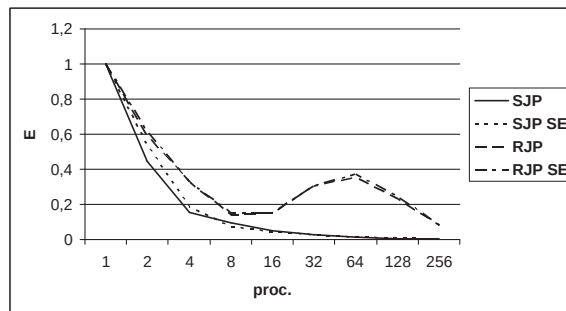
$f_2(x)$ Šekel funkcijos eksperimentų rezultatai pateikti 2.14 ir 2.15 paveiks-lėliuose. Šiai funkcijai visų algoritmų vykdymas nėra efektyvus. Matyti, kad algoritmai su sprendinio pasikeitimu sumažina SOF, kuris kitu atveju išauga labai stipriai, tačiau programų efektyvumas lieka mažas dėl procesorių apkrovos disba-lanso. Panašūs rezultatai gauti ir IDRIS superkompiuteryje, pateikti 2.16 ir 2.17 paveikslėliuose.



2.14 pav. $f_2(x)$ funkcijos įvairių lygiagrečiųjų algoritmų vykdymo efektyvumas naudojant geriausio režio taisyklę klasteryje vilkas.vtu.lt

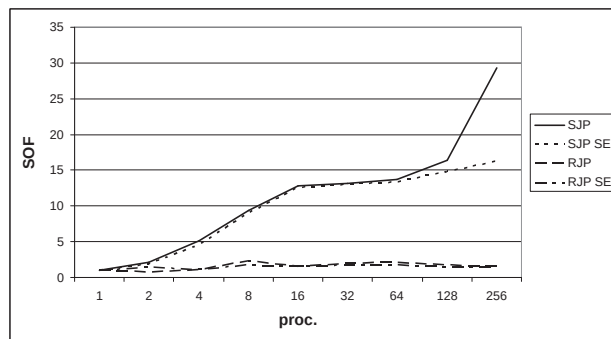


2.15 pav. $f_2(x)$ funkcijos įvairių lygiagrečiųjų algoritmų vykdymo SOF faktorius naudojant geriausio režio taisyklę klasteryje vilkas.vtu.lt

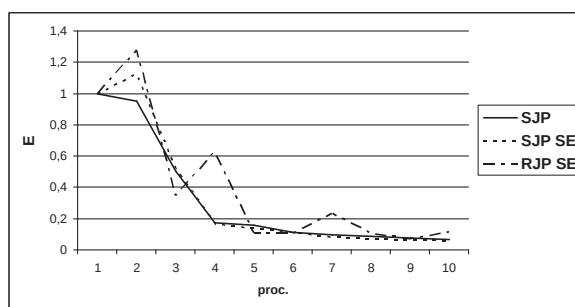


2.16 pav. $f_2(x)$ funkcijos įvairių lygiagrečiųjų algoritmų vykdymo efektyvumas naudojant geriausio režio taisyklę IDRIS centre

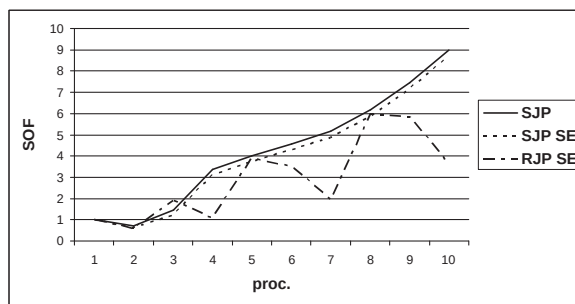
Eksperimentai atlikti ir sprendžiant TSP uždavinį naudojant geriausio režio taisyklę. Eksperimentai vykdyti klasteryje vilkas.vtu.lt ir superkompiuterių centre IDRIS.



2.17 pav. $f_2(x)$ funkcijos įvairių lygiagrečiųjų algoritmų vykdymo SOF faktorius naudojant geriausio režio taisyklę IDRIS centre

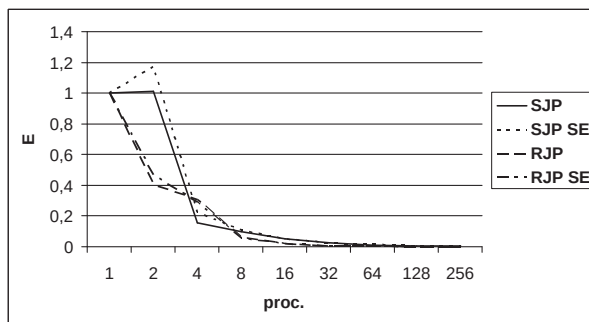


2.18 pav. TSP įvairių lygiagrečiųjų algoritmų vykdymo efektyvumas naudojant geriausio režio taisyklę klasteryje vilkas.vtu.lt

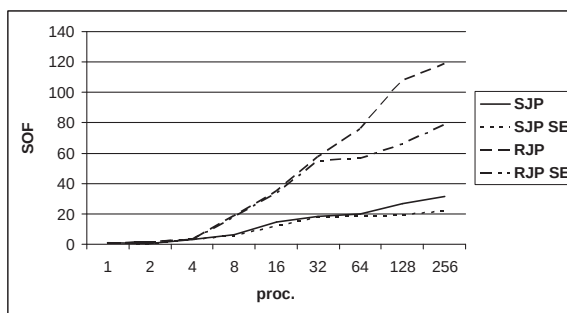


2.19 pav. TSP įvairių lygiagrečiųjų algoritmų vykdymo SOF faktorius naudojant geriausio režio taisyklę klasteryje vilkas.vtu.lt

Panašūs rezultatai gauti ir tiriant keliaujančio pirklio uždavinį: stipriai išauga SOF faktorius, dėl to labai krenta lygiagrečiųjų programų efektyvumas. Grafikai parodyti 2.18, 2.19 ir 2.20, 2.21 paveikslėliuose. Šakų ir režijų algoritmo šablono testavimo rezultatai skalbti [71, 72, 74, 76].



2.20 pav. TSP funkcijos įvairių lygiagrečiųjų algoritmų vykdymo efektyvumas naudojant geriausio režio taisyklę klasteryje IDRIS



2.21 pav. TSP funkcijos įvairių lygiagrečiųjų algoritmų vykdymo SOF faktorius naudojant geriausio režio taisyklę klasteryje IDRIS

2.5.2 Dinaminis procesorių apkrovos balansavimas

Šakų ir režijų algoritmo šablone įdiegti lygiagretieji algoritmai remiasi pradinės tyrimo srities padalinimu tarp procesorių. Vienas iš šių algoritmų efektyvumą mažinančių veiksnių yra procesorių apkrovos disbalansas. *Šakų ir režijų* algoritme užduotys generuojamos dinamiškai algoritmo vykdymo metu. Dėl to sprendimo pradžioje padalinant užduotis nėra žinoma, kiek sričių bus generuota iš pradinių ir galimas atvejis, kad kai kurie procesoriai generuos daugiau užduočių už kitus, dėl to dirbs ilgiau ir lygiagretusis algoritmas bus neefektyvus.

Duomenų lygiagretumo atveju, kai procesoriaus apkrova tiesiogiai susijusi su gautų duomenų skaičiumi, apkrovai balansuoti naudojamas statinis balansavimas. Tačiau *šakų ir režijų* algoritams statinis balansavimas neduoda gerų rezultatų. Norint didinti šablone įdiegtų algoritmų efektyvumą, siekiama, kad pabaigęs tirti algoritmo pradžioje gautas užduotis, procesorius gautų užduočių iš labiau apkrautų procesorių, jei tokių yra.

Kuriant balansavimo modulį, siekiama leisti vartotojui išbandyti įvairius balansavimo algoritmus, todėl įdiegti dažniausiai naudojami algoritmai bei palikta galimybė išplėsti šabloną.

Įdiegtas lokalus, gavėjo inicijuotas apkrovos balansavimo algoritmas. Siekiama ne suvienodinti apkrovą visuose procesuose, bet garantuoti, kad procesoriai nebus be darbo tuo metu, kai kiti yra stipriai apkrauti.

Šablone naudojamas paprasčiausias *apkrovos matas šakų ir rėžių* algoritmams – dar neištirtų užduočių skaičius [34]. Tačiau šis matas nėra tikslus. Reikėtų taip pat bandyti įvertinti, kiek perspektyvi yra užduotis. Tokiu atveju naudojamas ne tik užduočių skaičius, bet ir užduočių rėžis ir užduočių vieta neištirtų užduočių sąrašė [65]. Sudėtingesnis apkrovos mato vertinimas gali priklausyti nuo sprendžiamo uždavinio ir turi būti įdiegtas šablono vartotojo.

Apkrovos informacijos apsiikeitimo inicijavimas apibrėžia, kad apkrovos informacijos pasikeitimą inicijuoja procesorius, kuris nebeturi neištirtų užduočių. Šią taisyklę galima apibendrinti, kai balansavimo informacijos prašo procesorius, kurio apkrova tampa mažesnė už nurodytą lygį. Dėl optimizavimo uždavinio specifikos, net ir esant mažai neištirtų užduočių, negalima garantuoti, kad procesorius netaps stipriai apkrautas jam tiriant užduotis. Dėl to nuspręsta, kad balansavimo informacijos surinkimas inicijuojamas, kai vienas iš procesorių neturi neištirtų užduočių.

Balansavimo operacijos inicijavimo žingsnis apima gautos apkrovos informacijos apdorojimą ir balansavimo partnerių pasirinkimą. Įdiegta taisyklė, kad balansavimas inicijuojamas, jei bent iš vieno procesoriaus gauta informacija rodo, kad jo apkrova didesnė už nurodytą užduočių skaičių. Šis skaičius priklauso nuo sprendžiamo uždavinio. Apkrovos informacijos pasikeitimas vykdomas asinchroniškai, balansavimo inicijavimo sprendimą priima balansavimo informaciją surinkęs procesorius. Įvertinama, kad siunčiama apkrovos informacija gali ateiti pavėluotai, ir gali keistis laikui bėgant. Nepavykus susisiekti su labiausiai į partnerius tinkančiu (stipriausiai apkrautu) procesoriumi, bandoma vykdyti balansavimą su kitais tinkamais partneriais. Tokiu atveju nereikia rinkti informacijos iš naujo. Tačiau apkrovos informacija galėjo pasenti per tą laiką. Procesorius, į kurį kreipiamasi, gali nesutikti vykdyti balansavimo operacijos, jei jo apkrova pasikeitė po to, kai, jis siuntė apkrovos matą.

Balansavimo operacija vykdoma sinchronizuotai tarp dviejų procesorių: užduočių siuntėjo ir užduočių gavėjo. Užduočių gavėjas praneša užduočių siuntėjui, kad šis persiųstų užduotis. Užduočių siuntėjas laikinai sustabdo *šakų ir rėžių* algoritmo vykdymą ir pagal *užduočių paskirstymo* ir *užduočių pasirinkimo* taisykles ruošia užduotis persiuntimui. Tyrimai rodo, kad pusės užduočių perdavimas nėra efektyvus, o nusiuntus netinkamas užduotis, balansavimo operacija gali neduoti norimo rezultato [65]. Užduotys imamos iš eilės, remiantis eilės tvarkymo taisykle. Nurodomas skaičius, kiek užduočių perduoti, arba santykinis dydis, kokią dalį visos eilės siųsti. Tokiu būdu persiunčiamos potencialiai perspektyvios

užduotys, iš kurių su didele tikimybe bus generuojama daug neištirtų sričių. Yra siūlomi algoritmai, kurie skirti subalansuoti „sunkias“ ir „lengvas“ užduotis [65]. Tačiau šiuo atveju siekiama ne subalansuoti apkrovą, o parūpinti darbo, todėl pasirinkta taisyklė yra tinkama. Įvykdžius šias taisykles, užduotys yra paruošiamos perdavimui, o po perdavimo abiejuose procesoriuose toliau vykdomas *šakų ir rėžių* algoritmas.

Įdiegus dinaminį apkrovos balansavimo algoritmą tapo sudėtinga nustatyti skaičiavimų pabaigą. Yra nemažai siūlymų, kaip greičiausiai nustatyti, kada nebeieškoti balansavimo partnerių. Šablone įdiegtas žiedo pabaigos nustatymo algoritmas. Nustatytas lokalus skaičiavimų pabaigos požymis: nebeliko neištirtų užduočių ir nepavyko apkrovos balansavimo operacija, nes iš visų kaimynų gautas požymis, kad jie neturi darbo. Pirmasis procesorius siunčia pabaigos pranešimą procesoriui su vienetu didesniu numeriu. Kiti procesoriai siunčia pranešimą toliau, tik jei patys yra pasiekę lokalų pabaigos požymį ir yra gavę pabaigos pranešimą. Kai pirmasis procesorius gauna pabaigos pranešimą, vadinasi visi procesoriai baigė darbą. Šablonas gali būti praplėstas kitomis pabaigos nustatymo taisyklėmis, aprašytomis [34, 64].

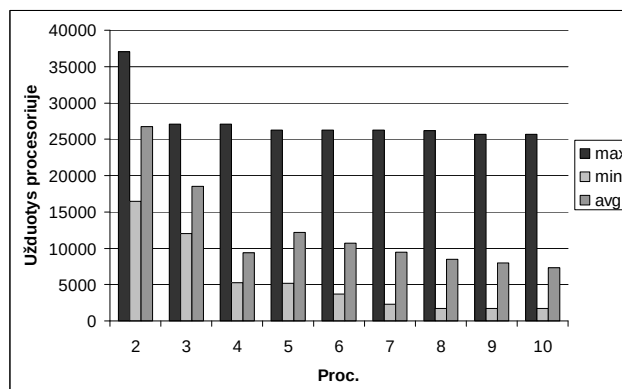
Įdiegus dinaminį apkrovos balansavimą, įrankis papildytas šiomis klasėmis:

- *Communication*. Klasė skirta apibendrinti globaliosios komunikacijos operacijas *šakų ir rėžių* algoritme. Ji naudojama tiek sprendiniams tarp procesorių pasikeisti, tiek apkrovos balansavimo operacijoms.
- *Balancer*. Apkrovos balansavimo algoritmų klasė, apjungianti keletą difuzinio apkrovos balansavimo algoritmų.
- *Sol_exchange*. Klasė, aprašanti sprendinių tarp procesorių pasikeitimą.
- *Termination*. Klasė, kurioje aprašytas pabaigos nustatymo algoritmas.

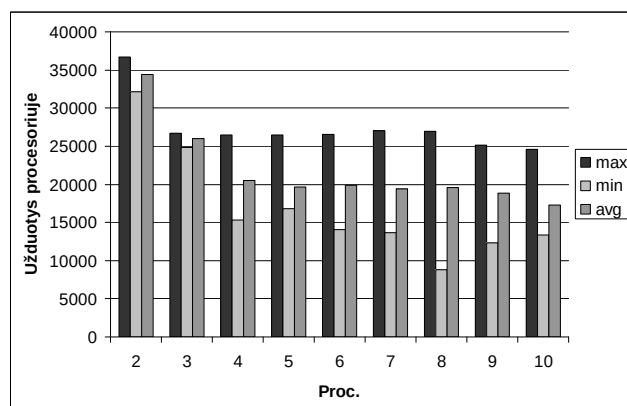
Dėl tokios struktūros įrankį galima nesunkiai praplėsti tiek kitais dinaminio balansavimo, tiek pabaigos nustatymo algoritmais.

2.5.3 Skaičiavimo eksperimentų rezultatai

Naudojant apkrovos balansavimo modulį, galima pasirinkti, kaip skirstyti užduotis: po vieną iš pradžių, po kelias iš pradžių, dalis sąrašo. Pateiktuose eksperimentų rezultatuose siekta parodyti, kaip pavyksta sumažinti procesorių apkrovos disbalansą persiuntinėjant po vieną užduotį. Vykdam tokį algoritmą, keičiasi užduočių tyrimo tvarka, dėl to gali pasikeisti ir visos ištirtų užduočių skaičius.

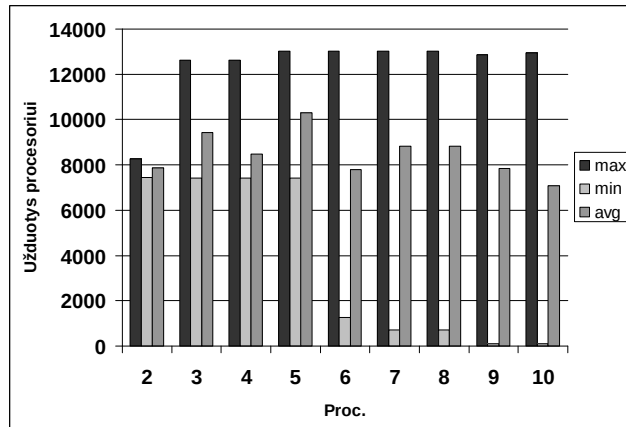


2.22 pav. $f_1(x)$ apkrovos disbalansas nenaudojant apkrovos balansavimo. (Geriausio režio taisyklė, vilkas.vtu.lt)

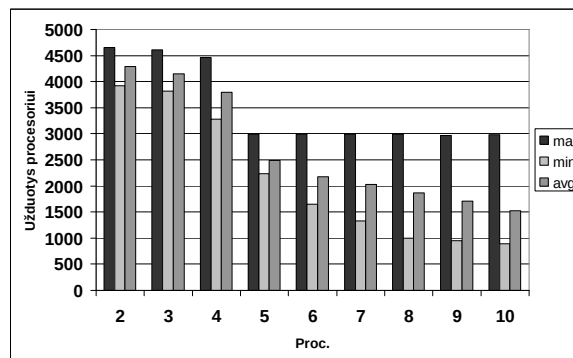


2.23 pav. $f_1(x)$ apkrovos disbalansas naudojant apkrovos balansavimą. (Geriausio režio taisyklė, vilkas.vtu.lt)

2.22 ir 2.24 parodytas procesorių apkrovos disbalansas sprendžiant $f_1(x)$ ir $f_2(x)$ uždavinius RJP SE algoritmu ir naudojant geriausio režio taisyklę. Pavaizduoti daugiausia, mažiausiai ir vidutiniškai procesoriuje ištirtų užduočių skaičius. Matyti, kad naudojant daugiau procesorių, disbalansas išauga ir tai mažina algoritmo efektyvumą. Tačiau naudojant paprastus dinaminio apkrovos balansavimo algoritmus pavyksta sumažinti disbalansą (žiūr. 2.23 pav.).



2.24 pav. $f_2(x)$ apkrovos disbalansas nenaudojant apkrovos balansavimo. (Geriausio režio taisyklė, vilkas.vtu.lt)



2.25 pav. $f_2(x)$ apkrovos disbalansas naudojant apkrovos balansavimą. (Geriausio režio taisyklė, vilkas.vtu.lt)

Perskirstant užduotis tarp procesorių keičiasi jų tyrimo tvarka todėl gali keistis ir užduočių, reikalingų uždaviniui išspręsti, skaičius. 2.25 paveikslėlyje parodyta, kaip naudojant dinaminį apkrovos balansavimą sumažėjo ne tik disbalansas, bet ir visas ištirtų užduočių skaičius. *Šakų ir režijų* algoritmo šablone įdiegtas dinaminis apkrovos balansavimo modulis aprašytas [73].

2.6 Juodosios dėžės globaliosios optimizacijos algoritmas

Šakų ir rėžių metodika gali būti naudojama realizuojant daug globaliosios optimizacijos algoritmų. Poskyryje pateikiamas globaliosios optimizacijos juodosios dėžės algoritmas, suformuluotas [67]. Jis panaudotas sprendžiant statinių pamatų optimizavimo uždavinį. Po to pateikiamas sukurtas ir realizuotas juodosios dėžės algoritmas.

2.6.1 Algoritmas

Juodosios dėžės algoritmais galima vadinti labai plačią klasę algoritmų, kai akcentuojamas jų taikymas. Vartotojams nėra įdomu, kaip vykdomi algoritmai, juos domina tik įvedami duomenys ir gaunami rezultatai. Šis pavadinimas taikomas panašioms sistemoms inžinerijoje, fizikoje.

Globaliosios optimizacijos juodosios dėžės algoritmas, sukurtas dr. J.Žilinsko, gali būti aprašytas kaip lokaliųjų paieškų tvarkymo strategija, ieškant globalaus sprendinio. Naudojant šį algoritmą galima tirti funkciją, kurios analizinė išraiška nėra žinoma. Skaičiavimų eigoje užtenka skaičiuoti funkcijos reikšmes duotuose taškuose. Apatinis rėžis skaičiuojamas tik apytiksliai, nes nėra žinoma funkcijos išraiška. Taigi, negalima garantuoti, kad randame tikslųjį sprendinį. Tačiau realaus taikymo uždaviniams užtenka rasti pakankamai gerą sprendinį. Skaičiavimai stabdomi po nurodyto laiko, taigi lygiagrečiuoju algoritmu dažniausiai siekiama išspręsti kuo didesnę uždavinį, bet ne išspręsti jį greičiau.

Pasirinkus neištirtą sritį, pasirinktuose taškuose skaičiuojamos funkcijos reikšmės, skaičiuojami funkcijos gradientai šiuose taškuose ir tiriamas funkcijos monotoniškumas. Jei randamas lokalis minimumas, išskiriama sritis apie minimumą ir ji atmetama iš tolesnių paieškų. Formuojami du neištirtų sričių sąrašai, viename saugomos dar neištirtos sritys, kitame – atmestos. Atmestųjų sričių sąrašas vėliau gali būti analizuojamas. Jei srityje randami keli lokalieji minimumai, sritis yra dalinama. Jei srityje nerandama lokaliųjų minimumų, mažinama srities dimensija. Algoritmas išsamiai aprašytas [67]. Algoritmas lygiagretinamas pradinę tyrimų sritį skaidant ir padalinant procesoriams.

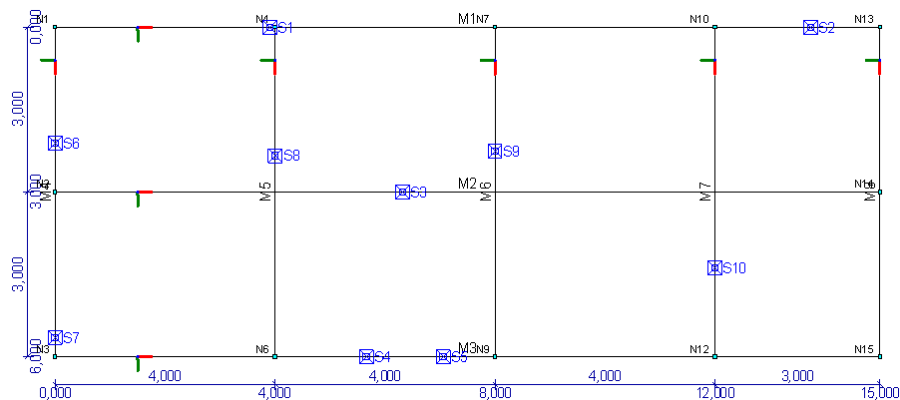
2.6.2 Algoritmo taikymas technologiniam uždaviniui

Algoritmas taikytas statinių pamatų optimizavimo uždaviniui, aprašytam 1.4.2 skyriuje. Šis algoritmas intelektualiau apdoroja atliekamas lokaliasias paieškas, todėl galima spręsti sudėtingesnės geometrijos ir apkrovų rostverkinio tipo pamatų optimizavimo uždavinius.

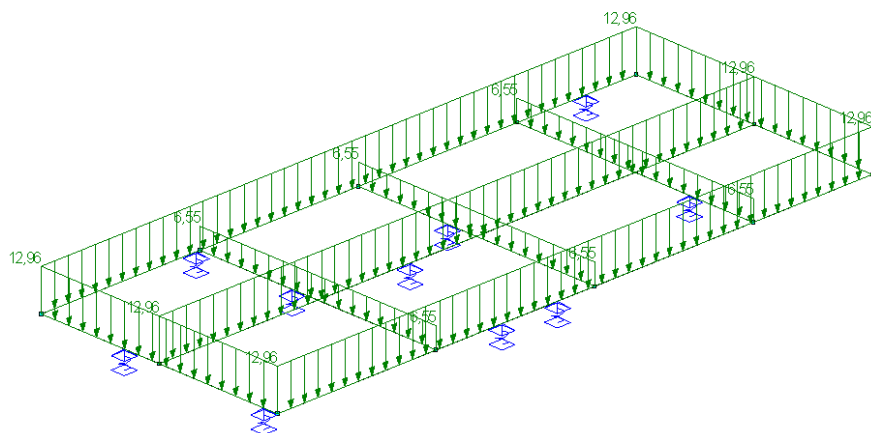
Spręsti du uždaviniai:

1 uždavinys. Stačiakampio formos rostverkinio tipo pamatai yra apkraunami dviem paskirstytomis vertikaliomis apkrovomis. Minimalus leistinas atstumas tarp atramų yra 0,210, vertikalus standumas $k_{spring} = 10^{15}$. Atramų skaičius lygus 10. Po 1 valandos skaičiavimų klasteryje *vilkas.vtu.lt* naudojant 10 Pentium III procesorių, gautas atramų paskirstymas parodytas 2.26 ir 2.27 paveikslėliuose. Gautos reakcijos jėgos yra tokios:

–206, 3; –170, 6; –204, 5; –189, 0; –94, 96;
– 196, 8; –168, 6; –207, 5; –192, 1; –207, 4.



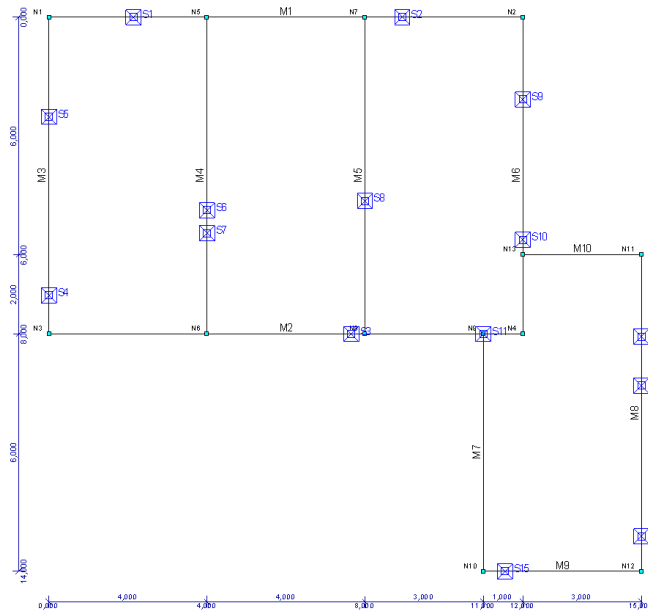
2.26 pav. 1 uždavinio geometrija ir apkrovos



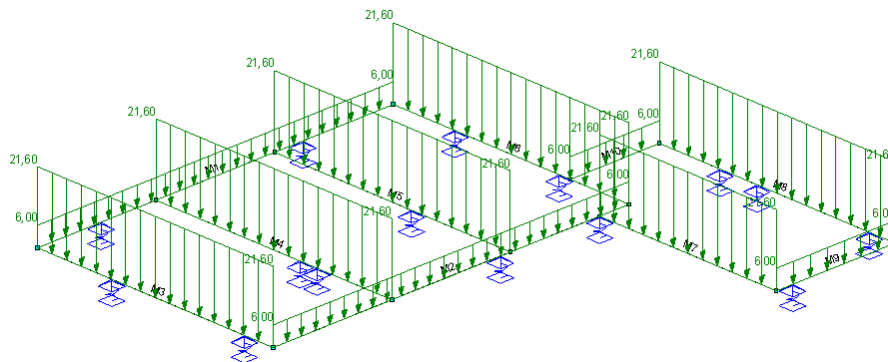
2.27 pav. 1 uždavinio optimizuotas atramų išdėstymas

2 uždavinys. Tiriami du stačiakampiai rėmai, apkrauti išskirstyta apkrova. Atramų skaičius yra 15, $k_{spring} = 10^{10}$ minimalus leistinas atstumas tarp atramų yra 0,10. Po 5 valandų skaičiavimų klasteryje *vilkas.vtu.lt* naudojant 10 procesorių gautas atramų paskirstymas parodytas 2.28 ir 2.29 paveikslėliuose. Reakcijos reikšmės yra tokios:

–156, 4; –158, 1; 147, 4; –161, 2; –153, 6; –118, 8; –130, 6;
– 161, 3; –161, 5; –139, 7; –161, 2; –144, 0; –94, 37; –121, 1; –136, 4.



2.28 pav. 2 uždavinio geometrija ir apkrovos



2.29 pav. 2 uždavinio optimizuotas atramų išdėstymas

Pastebėsime, kad abiejų uždavinių sprendiniuose apkrovų reakcijos poliuose skiriasi nedaug. Inžineriniais skaičiavimais tokie rezultatai nebuvo gauti. Rezultatai skelbti [69].

2.6.3 Siūlomas modifikuotas juodosios dėžės algoritmas

Šiame poskyryje aprašomas modifikuotas algoritmas, kuris irgi naudoja *šakų ir rėžių* metodą. Jis realizuotas laikantis *šakų ir rėžių* algoritmo šablono taisyklių, todėl jį testavome ir optimizavome naudodami sukurta įrankį. Siekiame sukurti euristiką, kuria sprendžiant globaliosios optimizacijos uždavinius, nereikėtų papildomų žinių apie tiriamą funkciją (tokių kaip funkcijos gradientas arba Lipšico konstantos), o užtektų tik skaičiuoti funkcijos reikšmes. Algoritme vystome [67] aprašytą algoritmą.

Sprendžiant pastatų pamatų optimizavimo uždavinius pastebėta, kad naudotas juodosios dėžės algoritmas skaičiavimų eigoje lėtai atmeta neperspektyvias sritis, dėl to skaičiavimai trunka ilgai ir ne visada yra efektyvūs. Siūlomoje algoritmo modifikacijoje siekta greičiau atmetinėti neperspektyvias tyrimo sritis. Apibrėžiame pagrindinius *šakų ir rėžių* metodo žingsnius:

- *Rėžio žingsnis.* Srityje D pasirenkame M taškų, kurie yra naudojami kaip pradiniai taškai Simplekso [11] metodo lokaliųjų minimumų paieškos procedūroje. Išskiriami atvejai:
 - *Nerasta lokaliųjų minimumų.* Sritis turi būti atmesta, jai priskiriamas didelis apatinis rėžis LB .
 - *Rastas vienas lokalusis minimumas.* Jei rasta reikšmė mažesnė už žinomą viršutinį rėžį UB , atnaujinamas viršutinis rėžis UB , o lokalusis sprendinys išsaugomas. Sritis turi būti atmesta iš tolesnių skaičiavimų, dėl to jai priskiriamas didelis apatinis rėžis.
 - *Rasti keli lokalieji minimumai.* Skaičiuojamas apatinis funkcijos reikšmių rėžis

$$MinLM = C(MaxLM - MinLM), \quad (2.13)$$

čia $MinLM$ – mažiausia iš rastų lokaliųjų minimumų reikšmių, $MaxLM$ – didžiausia iš rastųjų minimumų reikšmių, C – metodo parametras.

- *Šakos žingsnyje* sritis dalinama į dvi dalis. Jei ši sritis yra stačiakampis, tai dalinama pusiau ilgiausia kraštinė.

Pradinę sritį padalinus nurodytą skaičių, ji toliau nedalinama ir laikoma sprendiniu. Kiekvienoje srityje saugome surūšiuotus jau rastus lokaliuosius minimumus.

Jei tokių reikšmių yra daug, saugomos tik K reikšmių. Tokiu būdu tikrinama, ar tiriamose srityse randami nauji minimumo taškai, ar patikslinami jau rastieji.

Algoritmas buvo testuojamas sprendžiant pasirinktas skirtingų sudėtingumų, įvairių dimensijų funkcijas iš [41]. 2.1 lentelėje pateikiamos funkcijos, jų dimensija, lokaliųjų ir globaliųjų minimumų skaičius. Kelių globaliųjų minimumų atveju, algoritmu randamas vienas iš minimumų.

2.1 lentelė. Juodosios dėžės modifikuoto algoritmo testavimo funkcijos

Funkcija	Dimensija	Lokaliųjų minimumų skaičius	Globaliųjų minimumų skaičius
Rosenbrock	2	1	1
McCormic	2	1	1
Box and Betts	3	1	1
Paviani	10	1	1
Gen. Rosenbrock	30	1	1
Gold and Price	2	4	1
Shekel 5	4	5	1
Shekel 7	4	7	1
Shekel 10	4	10	1
Levy 4	4	71000	1
Levy 5	5	10^5	1
Levy 6	6	10^6	1
Levy 7	7	10^8	1
Griewank	10	10^3	1
Six Hump Camel	2	6	2
Branini	2	23	5
Shubert	2	400	9
Hansen	2	760	9

Uždaviniai su vienu ar keliais lokaliaisiais minimumais išsprendžiami labai greitai. Algoritmas ilgiau sprendžia uždavinius su labai daug lokaliųjų minimumų, pavyzdžiui Levy uždavinius. Tačiau spręstiems uždaviniams algoritmas yra stabilus. Nepavyko išspręsti tik labai didelės dimensijos Generalized Rosenbrock uždavinio.

2.7 Skyriaus rezultatai ir išvados

1. Išanalizavus nuoseklųjį *šakų ir režių* algoritmą, paruoštas nuoseklusis algoritmo šablonas. Vartotojas realizuoja tik nuo uždavinio priklausančias sprendimo dalis. Naudojant šabloną galima lengvai eksperimentuoti su

įvairiomis srities parinkimo strategijomis. Išnagrinėjus lygiagrečiuosius šakų ir režijų algoritmus, paruoštas lygiagretusis algoritmo šablonas. Jis remiasi nuosekliuoju šablonu. Vartotojui, norint gauti lygiagrečiąją programą, nereikia pačiam realizuoti lygiagrečiųjų algoritmų, užtenka pasirinkti vieną iš šablone įdiegtų.

2. Įrankiams testuoti realizuoti keliaujančio pirklio ir Lipšico funkcijų minimizavimo uždaviniai. Skaičiavimo eksperimentai parodė, kad Lipšico funkcijai naudojant apžvalgos srities parinkimo taisyklę pasiekiamas lygiagrečiosios programos efektyvumas 0,6 (pagreitėjimas iki 8,5 karto naudojant 15 procesorių). IDRIS superkompiuterių centre, su 256 procesoriais pasiektas 156 kartų pagreitėjimas.
3. Nustatyta, kad lygiagrečiosios *šakų ir režijų* programos, paremtos duomenų srities padalinimu tarp procesorių efektyvumą mažina du faktoriai:
 - Tyrimų srities išaugimas. Jį matuojame SOF faktoriumi. Šis faktorius mažėja, kai procesoriai apsieičia rasta geriausiais sprendiniais.
 - Procesorių apkrovos disbalansas. Procesorių apkrovos disbalansą mažina statinis arba dinaminis apkrovos balansavimo algoritmai. Kadangi optimizavimo uždaviniuose sritys yra generuojamos dinamiškai, dinaminiai apkrovos balansavimo algoritmai duoda geresnius rezultatus. Įrankyje realizuoti difuziniai dinaminio balansavimo algoritmai sumažina skirtumus tarp labiausiai ir mažiausiai apkrautų procesorių.
4. Juodosios dėžės optimizavimo algoritmu buvo spęstas statinių pamatų optimizavimo uždavinys. Sudėtingos geometrijos atvejams buvo gauti geri rezultatai – optimizavus atramų pozicijas gautos beveik vienodos reakcijos.
5. Sukurtas modifikuotas juodosios dėžės globaliosios optimizacijos algoritmas. Jis formuluotas naudojant *šakų ir režijų* metodiką, jo realizavimui naudotas algoritmo šablonas. Algoritme atsisakyta gradiento skaičiavimo, greičiau atmetamos neperspektyvios sritys. Algoritmas sėkmingai testuotas naudojant standartinius testus.

Darbo rezultatų apibendrinimas ir išvados

1. Modifikuotas *šeimininkas – darbininkai* algoritmo šablonas. Suformuota lygmenų sistema:

- Vartotojo programos;
- Vartotojo programos sąsajos;
- *Šeimininkas – darbininkai* algoritmo logikos;
- Duomenų perdavimo.

Šablono lygmenų struktūra leidžia įrankį naudoti su įvairiomis duomenų perdavimo bibliotekomis. Šablonas paruoštas naudoti su PVM ir MPI duomenų perdavimo bibliotekomis. Atlikus eksperimentus, parodyta, kad naudojant abi bibliotekas automatiškai generuojamos efektyviai vykdomos lygiagrečiosios programos.

2. Polinių statinių pamatų optimizavimo uždaviniui pritaikius *šeimininkas – darbininkai* algoritmą, nustatyta, kad šis paprastas lygiagretusis algoritmas yra efektyvus. Nesudėtingos geometrijos uždaviniui gauti geri rezultatai: polių išdėstymo schema. Tačiau panašūs rezultatai buvo gauti ir eksperimentiniais metodais. Taikytas lygiagretusis algoritmas leidžia efektyviai išnaudoti lygiagrečiųjų skaičiavimų galimybes.
3. Išnagrinėtas apibendrinto *šakų ir rėžių* algoritmo šablono sudarymas. Atlikta *šakų ir rėžių* algoritmų lygiagretinimo analizė. Atskyrus kintamas, nuo sprendžiamo uždavinio priklausančias, dalis nuo nekintamų algoritmo dalių, sukurtas nuoseklusis *šakų ir rėžių* algoritmo šablonas, pritaikytas kombinatorikos ir globaliosios optimizacijos uždavinių sprendimui. Naudojant šablona, vartotojas gali nesunkiai eksperimentuoti su įvairiais nuosekliaisiais *šakų ir rėžių* algoritmais ir eksperimentiškai tirti algoritmo tinkamumą sprendžiamam uždaviniui.

4. Remiantis nuosekliuoju *šakų ir rėžių* algoritmo šablonu, sukurtas lygiagrečiasis *šakų ir rėžių* algoritmų šablonas. Šablono struktūra leidžia automatiškai generuoti įvairių lygiagrečiųjų *šakų ir rėžių* algoritmų programas, naudojantis nuosekliomis vartotojo programomis. Šablone įdiegti lygiagretieji algoritmai, paremti tyrimų srities padalinimu tarp procesorių. Šablono struktūra pritaikyta eksperimentiškai parinkti tinkamus algoritmus ar jų modifikacijas sprendžiamiems uždaviniams.

Nustatyta, kad šablone įdiegtų lygiagrečiųjų *šakų ir rėžių* algoritmų efektyvumą mažina išaugantis SOF faktorius ir procesorių apkrovos disbalansas. Siūloma SOF faktorių mažinti procesoriams keičiantis informacija apie geriausius žinomus lokaliuosius sprendinius. Procesorių apkrovos balansavimui sukurtas dinaminio apkrovos balansavimo modulis.

5. Polinių statinių pamatų optimizavimo uždaviniams taikytas juodosios dėžės globaliosios optimizacijos algoritmas. Sudėtingiems atvejams gautas pakankamai geras sprendinys. *Šakų ir rėžių* algoritmo šablono rėmuose pasiūlytas ir ištirtas modifikuotas juodosios dėžės globaliosios optimizacijos algoritmas, nereikalaujantis papildomos informacijos apie tiriamas funkcijas. Tai leidžia *šakų ir rėžių* algoritmo šabloną plačiau naudoti praktinių uždavinių sprendimui.

Literatūra

- [1] Alba E., Almeida F., Blesa M., Cabeza J., Cotta C., Diaz M., Dorta I., Gabarro J., Leon C., Luna J., Moreno L., Pablos C., Petit J., Rojas A., Xhafa F. 2002 MALLBA: A Library of Skeletons for Combinatorial Optimisation. *Lecture Notes in Computer Science* 2400, 8th International Euro-Par Conference (EuroPar 2002), Paderborn, Germany, August 2002, p. 927–932.
- [2] Cavalheiro G., Galilee F., Roch J-L. Athapascan-1: Parallel Programming with Asynchronous Tasks. In: Yale Multithreaded Programming Workshop, Yale, USA, June 1998, internetinė prieiga <http://www-id.imag.fr/Logiciels/ath1/>, [2005.07.15].
- [3] Bader D.A., Hart W.E., Phillips C.A., Greenberg H.J. Parallel Algorithm Design for Branch and Bound. In: Tutorials on Emerging Methodologies and Applications in Operations Research, Chapter 5. *Kluwer Academic Press*, 2004, p. 1–44.
- [4] Barauskas R., Belevičius R., Kačianauskas R., Baigtinių elementų metodo pagrindai, Vilnius, *Technika*, 2004, P 612.
- [5] Belevičius R., Valentinavičius S., Optimizacijos uždaviniai statinių pamatams. *Lietuvos matematikos rinkinys*, 40 spec. nr., 2000, p. 365–372.
- [6] Belevičius R., Valentinavičius S., Michnevič E. Multilevel optimization of grillages. *Journal Civil Engineering and Management (Statyba)*, VIII(2), 2002, p. 98–103.
- [7] Boman E., et. all. Zoltan: data-management for parallel applications. Users Guide. Internetinė prieiga <http://www.cs.sandia.gov/Zoltan/>, [2005.09.21].
- [8] Briat J., Ginzburg I., and Pasin M., Plateau B. Athapascan Runtime: Efficiency for Irregular Problems. In: Proceedings of the Europar'97 Conference. Passau, Germany, Aug. 1997, Springer Verlag, 1997, p. 590–599.

- [9] de Bruin A., Kinderwater G.A.P., Trienekens H.W.J.M., van der Goot R.A., van Ginkel W. An object oriented approach to generic branch and bound. Technical Report 1996. EUR-FEW-CS-96-10. Erasmus University, Rotterdam, The Netherlands
- [10] Carissimi A., Pasin M. Athapascan: An Experience on Mixing MPI Communications and Threads. *Lecture Notes in Computer Science* 1497, Proceedings of the 5th European PVM/MPI User's Group Meeting. Liverpool, UK, Sept., 1998, p. 137–144.
- [11] Čiegis, R., Būda V. Skaičiuojamoji matematika. Vilnius, *TEV*, 1997, P 238.
- [12] Čiegis R. Lygiagretieji algoritmai ir tinklinės technologijos. Vilnius, *Technika*, 2005, P 322.
- [13] Čiegis R., Šablinskas R., Wasniewski J. Hyper-rectangle distribution algorithm for parallel multidimensional numerical integration. *Lecture Notes in Computer Science* 1697. Recent advances in PVM and MPI, 6th European PVM/MPI user's group meeting, Barcelona, Spain, 1999, p. 275–282.
- [14] Clausen J. Parallel Search-Based Methods in Optimization. *Lecture Notes in Computer Science*. 1184. Applied Parallel Computing – Industrial Computation and Optimization, Proceedings of PARA96. 1996, p. 176–185.
- [15] Csendes T. Generalized subinterval selection criteria for interval global optimization, *Numerical Algorithms* 37, 2004, p. 93–100.
- [16] Cun B.L. Bob++ library illustrated by VRP. In: European Operational Research Conference (EURO'2001), Rotterdam, 2001, p. 157.
- [17] Cun B., Roucairol C. BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms (User Guide of BOB V1.0). Technical Report, Université de Versailles – Laboratoire PRiSM 1995.
- [18] Cybenko G. Dynamic load balancing for distributed memory multiprocessors. *Parallel Distributed Computing* 7(2), 1989, p. 279–301.
- [19] Dijkstra E., Scholten C.S. Termination detection for diffusing computations. *Information Processing Letters* 11, 1980, p. 1–4.
- [20] Duncan R. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2), 1990, p. 5–16.
- [21] Dür M., Stix V. Probabilistic subproblem selection in branch-and-bound algorithms. *Journal of Computational and Applied Mathematics* 182, 1980, p. 67–80.

- [22] Eckstein J., Phillips C.A., Hart W.E. PICO: An Object-Oriented Framework for Parallel Branch and Bound, RUTCOR Research Report. 40-2000. Rutgers University, Piscataway, NJ, 2000.
- [23] Floudas C.A. editor. Deterministic Global Optimization: Theory, Methods and Applications. *Nonconvex Optimization and its Applications* 37. Kluwer Academic Publishers, 2000, P 597.
- [24] Floudas C.A., Akrotirianakis I.G., Caratzoulas S., Meyer C.A., Kallrath J. Global optimization in the 21st century: Advances and challenges. *Computers & Chemical Engineering* 29, 2005, p. 1185–1202.
- [25] Flynn M.J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computing* C-21(9), 1972, p. 948–960.
- [26] Foster I. Designing and building parallel programs. *Addison-Wesley publishing Company*. 1994, P 430.
- [27] Foster I., Kesselman C., Tuecke S. The anatomy of the grid: enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3), 2001, p. 200–222.
- [28] Gautier T., Hamidi H.R. Automatic Re-scheduling of Dependencies in a RPC-based Grid. In: *Proceedings of International Conference on Supercomputing (ICS 2004)*, Saint Malo, France, 2004, p. 89–94.
- [29] Gautier T., Hamidi H.R. High Performance Composition of Services with Data Dependencies on a Computational Grid. In: *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications, PDPTA '04*, June 21-24, 2004, Las Vegas, Nevada, USA, 2004, p. 801–805.
- [30] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V. PVM: Parallel Virtual Machine. *The MIT Press*, Cambridge. 1993, P 298.
- [31] Gendron B., Crainic T.G. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, 42(6), 1994, p. 1042–1066.
- [32] Gonzalez J.R., Leon C., Casiano Rodriguez C. An Asynchronous Branch and Bound Skeleton for Heterogeneous Clusters. *Lecture Notes in Computer Science* 3241, Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19 - 22, 2004. *Proceedings*, 2004, p. 191–198.

- [33] Goswami D., Singh A., Preiss B. From Design Patterns to Parallel Architecture Skeletons. *Journal of Parallel and Distributed Computing*, 62(4), 2002, p. 669–695.
- [34] Grama A., Gupta A., Karypis G., Kumar V. Introduction to Parallel Computing, *Addison Wesley*, 2003, P 656.
- [35] Goux J.P, Kulkarni S., Linderroth J., Yoder M. An enabling framework for master-worker applications on the computational grid. In: Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), Pittsburgh, Pennsylvania, 2000 August 15, 2000, p. 43–50.
- [36] Horst R., Pardalos P.M., Thoai N.V. Introduction to Global Optimization. 2nd. *Kluwer Academic Publishers*, 2001, P 368.
- [37] International Business Machines Corporation XL High Performance Fortran for AIX Language Reference and User's Guide, Version 1 Release 3, IBM, 1997.
- [38] Kale L.V., Ramkumar B., Sinha A.B., Gursoy A. The Charm parallel programming language and system: Part I — description of language features. Technical Report, Parallel Programming Lab., Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1994.
- [39] Kreinovich V., Csendes T. Theoretical justification of a heuristic subbox selection criterion for interval global optimization, *Central European Journal of Operations Research*, 9(3), 2001, p. 255–265.
- [40] Lawler E.W., Lenstra J.K., Rinnooy Kan A., Smoys D.B. The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. *Wiley Series in Discrete Mathematics and Optimization*. John Wiley & Sons, 1985. P 476.
- [41] Madsen K., Žilinskas J. Test functions for global optimization, 1999. Internetinė prieiga <http://www.imm.dtu.dk/km/GlobalOpt/testex/>, [2005.10.15].
- [42] Mattern F. Algorithms for distributed termination detection. *Distributed Computing* 2(3), 1987, p. 161–175.
- [43] Mažeika D., Lapienis S. Diegiamas paskirstytų skaičiavimų tinklas. *Mokslo Lietuva* 2(314), 2005, p. 3.
- [44] Nmeth Z., Sunderam V., Characterizing Grids: Attributes, Definitions, and Formalisms, *Journal of Grid Computing*, 1, 2003, p. 9–23.

- [45] OCREP: An automatic parallelization tool for synchronous languages, 1998. Internetinė prieiga <http://www.inrialpes.fr/pop-art/people/girault/Ocrep/>, [2004.05.03].
- [46] Payli R.U, Yilmaz E., Ecer A., Akay H.U., Chien S. DLB – a dynamic load balancing tool for grid computing. In: Parallel CFD'04 Conference, May 24-27, 2004, Canary Island, Spain, 2005, p. 387–394.
- [47] Roucairol C. Parallel branch and bound algorithms: An Overview. In: Proc. of the Int. Workshop on Parallel and Distributed Algorithms, Gers, France, 1988, p. 153–163.
- [48] Šablinskas R. Lygiagrečiųjų algoritmų tyrimas paskirstytos atminties lygiagretiesiems kompiuteriams. *Vytauto didžiojo universitetas*. 1999.
- [49] Shao G., Berman F., Wolski R. Master/Slave computing on the Grid. In: 9th Heterogeneous Computing Workshop, 2000, p. 3–16.
- [50] Shianno Y., Fujier T. PUBB (Parallelization Utility for Branch-and-Bound algorithms). User Manual, Version 1.0, 1999.
- [51] Shinano Y., Fujie T., Kounoike Y. Effectiveness of Parallelizing the ILOG–CPLEX Mixed Integer Optimizer in the PUBB2 Framework. *Lecture Notes in Computer Science*, 2790, Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference Klagenfurt, Austria, August 26-29, 2003 Proceedings, 2004, p. 451–460.
- [52] Shinano Y., Fujie T., Ikebe Y., Hirabayashi R. Solving the Maximum Clique Problem Using PUBB. In: IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium, 1998, p. 326–332.
- [53] Silva L., Buyya R. Parallel Programming Models and Paradigms, Chapter 1 in *High Performance Cluster Computing: Programming and Applications*. Prentice Hall PTR, NJ, USA, 1999.
- [54] Singh A., Schaeffer J., Szafron D. Views on template-based parallel programming. In: CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research. 1996, p.35–47.
- [55] Snir M., Otto S., Huss-Lederman S., Walket D., Dongarra J. MPI: The Complete Reference. *The MIT Press*. 1996, P 350.
- [56] Törn A., Žilinskas A. Global Optimization. *Lecture Notes in Computer Science* 350, 1989, P 252.

- [57] Traveling Salesman Problem. Internetinė prieiga <http://www.tsp.gatech.edu/index.html>, [2003.03.15].
- [58] Trienekens H. Parallel branch and bound and anomalies. Report EUR-CS-89-01, Department of Computer Science, Erasmus University, Rotterdam, 1989.
- [59] Tschoke S., Polzer T. Portable Parallel Branch-and-Bound Library PPBB-Lib. User Manual. Version 2.0. Department of Computer Science, University of Paderborn, 1996.
- [60] TSPLIB. Internetinė prieiga <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>, [2003.03.15].
- [61] VAST auto-parallelization, auto-vectorization tools, 2003. Internetinė prieiga: <http://www.absoft.com/Products/Libraries/vast.html>, [2004.10.15].
- [62] Watts J., Taylor S. A Practical approach to Dynamic Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3), 1998, p. 235–248.
- [63] Willebeek M.H., Reeves A.P. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(9), 1993, p. 979–993.
- [64] Wilkinson B., Allen M. Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition. *Prentice Hall*, 2005, P 431.
- [65] Xu C., Lau F. Load Balancing in Parallel Computers. Theory and Practice. *Kluwer Academic Publishers*. 1997, P 203.
- [66] Xu C., Lau F. Iterative Dynamic Load Balancing in Multicomputers. *Journal of Operational Research Society*, 45(7), 1994, p. 786–796.
- [67] Žilinskas J. Black box global optimization: covering methods and their parallelization. *Kaunas University of Technology*, 2002.

Autorės publikacijos

- [68] Baravykaitė M., Belevičius R., Čiegis R. One application of the parallelization tool of master-slave algorithms. *Informatica*, 13 (4), 2002, p. 1–12.
- [69] Čiegis R., Baravykaitė M., Belevičius R. Parallel global optimization of foundation schemes in civil engineering. *Lecture Notes in Computer*

Science, Vol 3732 2006, Applied Parallel Computing: 7th International Conference, PARA 2004, Lyngby, Denmark, June 20-23, 2004. Revised Selected Papers. Editors: Jack Dongarra, Kaj Madsen, Jerzy Wasniewski 2006, p. 305–313.

- [70] Baravykaitė M., Šablinskas R. The template programming of parallel algorithms. *Mathematical modelling and analysis*, 7(1), 2002, p. 11–20.
- [71] Baravykaitė M., Čiegis R., Žilinskas J. Template realization of generalized branch and bound algorithm. *Mathematical modelling and analysis* 10(3), 2005, p. 217–236. .
- [72] Baravykaitė M. Lygiagrečiųjų šakų ir režijų algoritmų programų kūrimas. *Lietuvos matematikos rinkinys*, 44, spec. nr., 2004, p. 220–225.
- [73] Baravykaitė M. Procesorių apkrovos balansavimas lygiagrečiuose šakų ir režijų algoritmuose. *Lietuvos matematikos rinkinys*, 45, spec nr., 2005, p. 119–123.
- [74] Baravykaitė M. Parallel branch and bound algorithm template. *Proceedings of 10th international conference Mathematical Modelling and Analysis, 2nd International Conference Computational Methods in Applied Mathematics*, June 1–5, 2005, Trakai, Lithuania, 2005, p. 165–170.
- [75] Baravykaitė M. Lygiagrečiųjų programų automatinio generavimo priemonė šeimininkas – darbininkai algoritams. 5-osios Lietuvos jaunųjų mokslininkų konferencijos „Lietuva be mokslo – Lietuva be ateities“ medžiaga: Matematika ir informatika, 2002, p. 32–38.
- [76] Baravykaitė M. Lygiagretieji šakų ir režijų algoritmai. 7-osios Lietuvos jaunųjų mokslininkų konferencijos „Lietuva be mokslo – Lietuva be ateities“ medžiaga: Matematika ir informatika, 2004, p. 51–57.