VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Hierarchiniai lygiagretieji šakų ir rėžių algoritmai heterogeninėms skaičiavimų sistemoms

Hierarchical parallel branch-and-bound algorithms for heterogeneous high performance systems

Magistro baigiamasis darbas

Atliko:	Tomas Seniut	(parašas)
Darbo vadovas:	prof. dr. Julius Žilinskas	(parašas)
Recenzentas:	prof. dr. Rimantas Vaicekauskas	(parašas)

Vilnius – 2019

Santrauka

Daug problemų fizikos, ekonomikos ar technikos srityje gali būti formuluojamos kaip globalios ar kombinatorinės optimizacijos uždaviniai. Kadangi globalios bei kombinatorinės optimizacijos uždaviniai priskiriami NP-sunkiems uždaviniams, jie yra labai reiklūs skaičiavimo resursams. Siekiant rasti optimizacijos uždavinių optimalų sprendinį kuo greičiau, naudojami algoritmai protingai perrenkantys galimus sprendinius. Vienas iš tokių algoritmų yra šakų ir rėžių algoritmas. Norint dar pagreitinti sprendinių paiešką šakų ir rėžių algoritmais, jie yra lygiagretinami. Tai reiškia, kad uždavinys išskaidomas į nepriklausomas dalis ir naudojami keli procesorių branduoliai ar keli procesoriai spręsti atskiras uždavinio dalis. Egzistuojantys išlygiagretinti šakų ir rėžių algoritmų įgyvendinimai nepakankamai atsižvelgia į superkompiuterių hierarchiškumą, taigi efektyviai neišnaudoja prieinamų resursų. Šiuo darbu norima ištirti mažai nagrinėtas šakų ir rėžių algoritmo hierarchinio lygiagretinimo galimybes.

Šiam tikslui pasiekti buvo sukurta hierarchiškai lygiagretinanti šakų ir rėžių algoritmus biblioteka. Pasinaudojus sukurta biblioteka įgyvendinti keliaujančio pirklio, kuprinės ir kvadratinio priskyrimo uždaviniai. Hierarchinis lygiagretinimas buvo pasiektas naudojant *OpenMP* ir *MPI* technologijas. *OpenMP* leidžia efektyviai išlygiagretinti procesus, turinčius bendrą atmintį, per kurią vyksta informacijos apsikeitimas. *MPI* skirta lygiagretinimui, kai procesai neturi bendros atminties. Taigi tenka siųsti žinutes į kitus procesus. Šias dvi technologijas galima naudoti kartu. Tai puikiai tinka šiuolaikiniams super-kompiuteriams, nes jie paprastai sudaryti iš daugelio susietų mazgų, kiekviename kurių procesoriai gali turėti kelis branduolius.

Darbe atliktiems skaičiavimams buvo naudotas Paskirstytųjų skaičiavimų tinklas, priklausantis Skaitmeninių tyrimų ir skaičiavimų centrui. Šio tyrimo metu buvo leidžiami tie patys uždaviniai keičiant gijų ir procesų skaičių. Iš rezultatų matyti, kad lygiagretinant tik per bendrą atmintį pasinaudojus *OpenMP* buvo pasiektas beveik tiesinis pagreitėjimas. Lygiagretinimo siunčiant žinutes efektyvumas priklauso nuo sprendžiamo uždavinio. Bendrai *MPI* lygiagretinimo efektyvumas buvo mažesnis nei lygiagretinant su *OpenMP*. Tai rodo, kad hierarchinis lygiagretinimas gali paspartinti šakų ir rėžių algoritmų sprendimą.

Raktiniai žodžiai: Šakų ir rėžių algoritmai, skaičiavimų sistemos, hierarchiniai lygiagretieji algoritmai, hibridinis MPI ir OpenMP lygiagretusis programavimas

Summary

A lot of physics, economics and technical problems can be formulated as global or combinatorial optimization problems. Since global or combinatorial optimization problems are considered NP-complete problems, they are demanding on computing resources. To find solutions for optimization problems as fast as possible algorithms that sensibly enumerate solution space. One of such algorithms is called branch and bound algorithm. To speed up search for solution even more algorithm parallelization is used. This means that problem is divided into parts what are solved independently be separate cores on processor. Existing parallel branch and bound algorithms does not take into account hierarchical structure of supercomputers, thus are not using them efficiently. This work aims to investigate efficiency of hierarchical parallel branch and bound algorithms.

For this purpose library for hierarchical parallelization of branch and bound algorithms was created. Using this library traveling salesman, knapsack and quadratic assignment problem solvers where implemented. Hierarchical parallelization was achieved using *OpenMP* and *MPI* technologies. *OpenMP* allows to parallelize processes using shared memory to share information among processes. *MPI* is used then processes does not have shared memory and need to rely on message passing to communicate with other processes. These technologies can be used together. This fits perfectly for today's supercomputers as usually they consists of several nodes, each of them having multiple cores.

For computations in this for "Paskirstytųjų skaičiavimų tinklas" belonging to "Skaitmeninių tyrimų ir skaičiavimų centras" was used. To study hierarchal parallelization same problems was ran with different number of allocated nodes and cores on supercomputer. Results show that parallelization through shared memory using *OpenMP* achieved near linear speedup. Parallelization through sending messages showed higher dependency on problem being solved. Overall using just *MPI* showed lower speedups than *OpenMP*. In conclusion hierarchal parallelization can be used to speedup solving of branch and bound algorithms.

Keywords: Branch and Bound algorithms, High Performance Systems, Hierarchical parallel algoritms, Hybrid MPI and OpenMP Parallel Programming

Turinys

Įvadas

Daug problemų fizikos, ekonomikos ar technikos srityje gali būti formuluojamos kaip globalios ar kombinatorinės optimizacijos uždaviniai. Sprendžiant šiuos uždavinius yra ieškomi argumentai, x atitinkantys globalius funkcijos ekstremumus, arba įrodoma, kad tokie argumentai neegzistuoja. Kadangi globalios bei kombinatorinės optimizacijos uždaviniai priskiriami NP-sunkiems uždaviniams (nėra žinomas algoritmas, galintis tokias problemas išspręsti per polinominį laiką ar greičiau), jie yra labai reiklūs skaičiavimo resursams. Siekiant rasti optimizacijos uždavinių optimalų sprendinį kuo greičiau, naudojami algoritmai, protingai perrenkantys galimus sprendinius.

Priklausomai nuo tikslo funkcijos iškilumo optimizavimo uždavinius galima spręsti keliais būdais. Jei tikslo funkcija yra iškili tai, kiekvienas lokalus ekstremumas kartu bus ir globalus. Tokiems uždaviniams pakanka rasti lokalų ekstremumą, t.y. kiekviename žingsnyje rasti sprendinį, minimizuojantį ar maksimizuojantį tikslo funkciją. Kai surandamas toks sprendinys uždavinys laikomas išspręstu. Taip negalima elgtis, kai tikslo funkcija nėra iškili, nes lokalių ekstremumų sekimas nenuves prie globalaus ekstremumo. Esant daugybei ekstremumų reikia kitokio tipo optimizavimo algoritmų, kurie galėtų perrinkti visus ekstremumus ir išrinkti geriausią. Vienas iš tokių algoritmų yra šakų ir rėžių algoritmas.

Klasikiniu šakų ir rėžių algoritmo įgyvendinimu siekiama padalinti leistiną sprendinių aibę į poaibius ir įvertinti poaibių viršutinius ar apatinius tikslo funkcijos rėžius. Jei gautas rėžis yra prastesnis nei geriausias žinomas sprendinys, visą poaibį galima atmesti kaip neturintį globalaus ekstremumo. Kiekviena šakų ir rėžių algoritmo iteracija turi tris pagrindinius komponentus: poaibio pasirinkimas, jo padalijimas ir rėžių skaičiavimas. Kiekvienas iš komponentų skiriasi kiekvienam konkrečiam uždaviniui. Taigi, nors ir šakų ir rėžių algoritmai turi bendra formą – konkretus įgyvendinimas skirtingiems uždaviniui stipriai skiriasi. Dėl to ir uždavinio sprendimo spartinimas taikant šį algoritmą stipriai priklauso nuo sprendžiamo uždavinio, blogiausiu atveju tenka perrinkti visus galimus sprendinius.

Norint dar labiau pagreitinti sprendinių paiešką šakų ir rėžių algoritmais jie yra lygiagretinami, t.y. uždavinys išskaidomas į nepriklausomas dalis ir naudojami keli procesorių branduoliai ar keli procesoriai spręsti atskiras uždavinio dalis. Šakų ir rėžių algoritmai yra patogūs lygiagretiems skaičiavimams, nors ir turi sunkumų juos išlygiagretinant. Šakojimosi metu kiekvienas poaibis gali būti generuojamas ir sprendžiamas nepriklausomai nuo prieš tai buvusių ar gretimų poaibių. Sunkumai atsiranda, norint suderinti sprendžiamų poaibių paskirstymą tarp procesų, nes poaibiai yra generuojami ir eliminuojami dinamiškai, tai reiškia, kad nėra iš anksto žinomas uždavinio išskaidymas ir poaibių pasidalijimas tarp procesų turi vykti dinamiškai. Taip pat, procesai turi dalintis tarpiniais rezultatais, norint, kad poaibių eliminavimas vyktų efektyviai. Toks nereguliarus šakų ir rėžių algoritmų elgesys neleidžia atskiriems procesams veikti visiškai nepriklausomai.

Egzistuojantys išlygiagretinti šakų ir rėžių algoritmų įgyvendinimai nepakankamai atsižvelgia į superkompiuterių hierarchiškumą, taigi efektyviai neišnaudoja prieinamų resursų. Dažnai kompiuterių klasterių mazgai yra homogeniški multiprocesoriai, tačiau dėl to, kad klasterių mazgai gali skirtis, bendravimo našumas klasterio viduje ir tarp klasterių yra kitoks, sistema tampa hierarchiška ir nehomogeniška. Šiai dienai galima rasti mazgų, sudarytų iš 128 branduolių, nors mažesnis

skaičius (16 ar 32 branduoliai) yra dažnesnis, o sujungtų mazgų skaičius nėra ribojamas. Didėjant prieinamumui prie superkompiuterių ar kompiuterių klasterių sujungtų sparčiais ryšiais, darosi vis aktualiau efektyviai išnaudoti turimus skaičiavimo resursus.

Nors ir egzistuoja išlygiagretinti šakų ir rėžių algoritmai, juose paprastai naudojamas tik vienas lygiagretinimo būdas. Bendraujama per bendrą atmintį arba pranešimais. Algoritmams vis dar trūksta hierarchiškumo, nėra efektyvu nei vien bendrauti per bendra atmintį, nei vien siuntinėti pranešimus. Galimas hierarchinis algoritmas turėtų gebėti bendrauti ir per bendrą atmintį, ir siunčiant žinutes. Tokių hierarchinių lygiagrečiųjų šakų ir rėžių algoritmų efektyvumas nėra ištirtas. Šiuo darbu ir norima ištirti mažai nagrinėtas šakų ir rėžių algoritmo hierarchinio lygiagretinimo galimybes.

Magistrinio darbo rezultatai buvo pristatyti ir publikuoti „Lietuvos magistrantų informatikos ir IT tyrimai“ konferencijos darbuose[1].

[1] `http://www.journals.vu.lt/proceedings/issue/view/1129`

Darbo tikslas ir uždaviniai

Darbo tikslas – sukurti hierarchinius lygiagrečiuosius šakų ir rėžių algoritmus, efektyviai naudojančius heterogeninių našiųjų kompiuterių resursus.

Pagrindiniai uždaviniai:

1. Nustatyti galimus būdus hierarchiškai išlygiagretinį šakų ir rėžių algoritmą;
2. Ištirti egzistuojančias kompiuterines bibliotekas šakų ir rėžių algoritmams lygiagretinti;
3. Sukurti biblioteką, kuri bus naudojama hierarchiškai išlygiagretinti šakų ir rėžių algoritmus;
4. Pasirinkti kelis optimizavimo uždavinius ir realizuoti jiems skirtus šakų ir rėžių algoritmus;
5. Ištirti hierarchinių lygiagrečiųjų algoritmų spartinimo priklausomybę nuo sprendžiamo uždavinio charakteristikų;

1. Literatūros apžvalga

Šakų ir rėžių (*angl., Branch-and-Bound, BB*) algoritmai yra plačiausiai paplitusi priemonė spręsti NP-sunkius optimizacijos uždavinius. Dėl šių uždavinių natūralaus sudėtingumo tik nedideli uždaviniai gali būti išspręsti, naudojant nuoseklius algoritmo įgyvendinimus per prieinamą laiką. Siekiant spręsti vis sudėtingesnius optimizavimo uždavinius BB algoritmus pradėta lygiagretinti.

Lygiagrečių šakų ir rėžių algoritmų apžvalga galima rasti darbuose [Rou89], [PL90], [TB92], naujesnės apžvalgos yra [GC94] ir [CLR06]. Nors ir praėjo nemažai laiko nuo paskutinės apžvalgos, tačiau lygiagretinimo būdai mažai pasikeitė. Vis dar dauguma darbų, atliekamų lygiagretinant BB algoritmus, yra nehierarchiniai, t.y. negalintys tinkamai išnaudoti hierarchinių skaičiavimo sistemų. Ir kol kas nėra nei vienos bibliotekos, skirtos hierarchiškai lygiagretinti šakų ir rėžių algoritmus. Efektyvus hierarchinių sistemų panaudojimas tampa vis aktualesnis dėl augančio tokios techninės įrangos prieinamumo bei mažėjančių kainų.

Šia apžvalga norima dalinai užpildyti spragą nuo paskutinės BB algoritmų lygiagretinimo apžvalgos akcentuojantis į hierarchinius lygiagrečiuosius algoritmus bei supažindinti skaitytoją su darbo tema ir problematika. Pirma yra aptarti lygiagretieji skaičiavimai ir būdai juos atlikti. Antra, bus trumpai pristatyti nuoseklūs šakų ir rėžių algoritmai, apibrėžiant jų bendrą eigą. Trečia – aptarti šakų ir rėžių algoritmų lygiagretinimo būdai bei jų klasifikacija. Pabaigoje bus apžvelgtos egzistuojančios lygiagrečios šakų ir rėžių algoritmų bibliotekos.

1.1. Paskirstytieji skaičiavimai

Paprasta prasme paskirstytieji skaičiavimai yra vienalaikis kelių kompiuterinių resursų (t.y. procesorių branduolių) naudojimas išspręsti uždavinį. Uždavinys yra padalijamas į atskiras dalis, kurios gali būti sprendžiamos nepriklausomai ir išdalijamos skaičiavimo resursams. Paskirstytųjų skaičiavimo efektyvumas labai priklauso ne tik nuo algoritmo įgyvendinimo, bet ir nuo techninės įrangos, nes tai nulemia galimą bendravimą tarp skaičiavimo komponentų. Yra išskiriami du pagrindiniai būdai, kaip gali bendrauti skaičiavimo vienetai: bendravimas per bendrą atmintį ir bendravimas žinutėmis. Kalbant apie paskirstytuosius skaičiavimus, taip pat, yra svarbu suprasti metrikas, naudojamas matuojant lygiagretinimo efektyvumą.

1.1.1. Bendros atminties sistemos

1.1.1.1. Architektūra

Bendros atminties lygiagretieji kompiuteriai dažnai tarpusavyje labai skiriasi, bet visi turi bendrą savybę. Visi procesoriai turi prieigą prie tos pačios atminties per globalią adresų erdvę. Keli procesoriai gali dirbti nepriklausomai naudodami tuos pačius atminties resursus. Vieno procesoriaus pakeitimai atmintyje yra matomi visų kitų. Istoriškai bendros atminties mašinos skirstomos pagal atminties pasiekimo laiką į dvi grupes[2]: *UMA* (*angl., Uniform Memory Access*) ir *NUMA*

[2] `https://techdifferences.com/difference-between-uma-and-numa.html`

(*angl., Non-Uniform Memory Access*).

UMA mašinos dažniausiai būna įgyvendintos kaip simetrinių procesorių mašinos. Šiose mašinose visi procesoriai yra identiški. Procesoriai turi vienodą prieigą prie atminties ir ją pasiekia per vienodą laiką. Sistema ypatinga tuo, kad visi procesoriai naudoja tą pačią talpyklą (*angl., Cache*), dėl to, bet kokie pakeitimai atmintyje iš karto matomi visiems procesoriams. Kalbant apie *NUMA* sistemas, jos paprastai būna įgyvendintos sujungus kelias simetrinių procesorių sistemas. Kiekviena simetrinių procesorių posistemė gali tiesiogiai pasiekti kitą simetrinių procesorių posistemės atmintį. Dėl to, kad dalis atminties pasiekiama per sujungimą, atminties pasiekimo laikas nėra vienodas visais atvejais. Šiose sistemose talpyklos vientisumas nėra būtinai išlaikomas, tai priklauso nuo techninės įrangos.

Apibendrinant bendros atminties sistemas, jų privalumai yra tai, kad tarp skaičiavimo uždavinio dalių duomenų dalijimasis yra greitas ir vienodas, dėl to, kad atmintis ir procesoriai yra glaudžiai susieti. Kitas teigiamas aspektas yra globalios adresų erdvės teikiamas paprastas ir lengvai suprantamas vartotojams priėjimas prie atminties. Vienas pagrindinių bendros atminties sistemų trūkumų yra prastos sistemos galimybės jos praplėtimui pridedant daugiau procesorių. Didinant procesorių skaičių greitai apkraunamas duomenų judėjimas procesoriaus-atminties keliu. Sistemose, palaikančiose vientisą talpyklą, taip pat, stipriai apkraunamas ir talpyklos valdymas. Kitas trūkumas yra tai, kad programuotojas tampa atsakingas už procesų sinchronizavimą užtikrinant, kad globali atmintis yra prieinama taisyklingai.

1.1.1.2. OpenMP

Viena populiariausių technilogijų bendravimui per bendrą atmintį yra *OpenMP*[3]. *OpenMP* specifikacija yra bendrai apibrėžta grupės pagrindinių techninės ir programinės įrangos gamintojų. Dėl to biblioteka tinka daugumai platformų ir patogi kurti programas, išlygiagretintas naudojantis bendra atmintimi. Biblioteka palaiko programavimo kalbas `C/C++` ir `Fortran` plačiam kompiuterių architektūrų pasirinkimui.

Pagrindiniai *OpenMP* tikslai yra standartizacija, lengvumas, paprastumas naudoti ir portabilumas. *OpenMP* siekia šių tikslų pasiūlydama standartą tinkantį plačiam kompiuterių architektūrų ir operacinių sistemų pasirinkimui. Pasakius *OpenMP* tikslus reikėtų paminėti ir ko ši biblioteka nesiekia. *OpenMP* negarantuoja efektyviausio bendros atminties panaudojimo, tiekėjai taip pat neįsipareigoja identiškai įgyvendinti šį standartą. Karkasas nėra pritaikytas reguliuoti lygiagrečios įvesties/išvesties, programuotojas turi pats užtikrinti sinchronizacija reikiamais atvejais. *OpenMP* pateikiama biblioteka turi paprastą ir paltų rinkinį procesoriaus direktyvų skirtą programuoti bendros atminties sistemas. Lygiagretinimas gali būti stipriai išnaudojamas pritaikius tik 3-4 direktyvas. Direktyvos taip pat leidžia išlygiagretinti programą palaipsniui, skirtingai negu žinutes siunčiančias bibliotekos, kurioms paprastai reikia pilno lygiagretinimo nuo pat pradžių.

Kalbant apie *OpenMP* lygiagretinimo modelį, lygiagretinimas yra pasiekiamas naudojantis tik gijomis. Programos vykdymo gija yra mažiausias darbo vienetas, kuris gali būti įtrauktas į operacinės sistemos planavimą. Giją galima įsivaizduoti kaip paprogramę, kuri gali būti vykdoma

[3] `https://www.openmp.org/`

nepriklausomai. Tačiau gijos vis tik egzistuoja kaip resursai tėviniame procese, taigi nutraukus tėvinį procesą gijos irgi yra nutraukiamos. Paprastai gijų skaičius sutampa su branduolių skaičiumi mašinoje, bet kiekviena programa gali pasirinkti ir kitokį gijų skaičių. *OpenMP* naudoja išreikštinį (ne automatinį) programavimo modelį, tai suteikia programuotojui pilną kontrolę atliekant lygiagretinimą. Dėl šios priežasties programos išlygiagretinimas gali būti paprastas kaip kelių direktyvų įterpimas arba sunkus kaip paprogramių sudarymas su keliais lygiagretinimo lygiais ir daugelio lygių užraktais. *OpenMP* pritaiko šakojimosi-sujungimo (*angl., Fork-Join*) modelį. Tai yra visos *OpenMP* programos prasideda kaip vienos gijos - pradinės gijos - programos. Pradinė gija yra vykdoma iki kol pasiekiamas pirmasis lygiagretinimo instrukcija. Tuo metu įvyksta šakojimasis - pradinė gija sukuria grupę lygiagrečių gijų. Kodas esantis išlygiagretintame bloke yra pradedamas vykdyti lygiagrečiai. Kai visos gijos baigia darbą išlygiagretintame bloke įvyksta gijų sinchronizacija ir jos yra nutraukiamos paliekant tik pradinę giją.

OpenMP susideda iš trijų pagrindinių komponentų. Tai yra kompiliatoriaus direktyvos, bibliotekos metodai ir aplinkos kintamieji. Kompiliatoriaus direktyvos į programos kodą yra įrašomos kaip komentarai ir paprastai yra ignoruojamos kompiliatorių nebent yra nurodoma kitaip kompiliuojant programą. Šios direktyvos gali būti naudojamos daugeliui tikslų, tokių kaip: lygiagrečio regiono sukūrimui, kodo blokų paskirstymui gijoms, ciklo iteracijų paskirstymui, sinchronizacijai. Bibliotekos metodai yra kviečiami, kaip ir bet kurie kiti metodai, atitinkamose programavimo kalbose. *OpenMP* bibliotekos metodai gali būti naudojami nustatyti ir sužinoti gijų skaičių, nustatyti ir užklausti gijų charakteristikas, nustatyti, inicializuoti ar panaikinti užraktus bei nustatyti ir užklausti lygiagretinimo lygius. Taip pat *OpenMP* siūlo ir aplinkos kintamuosius, kuriais galima valdyti lygiagretaus kodo vykdymą. Kintamieji leidžia nustatyti, kaip bus dalijamos ciklo iteracijos, gijų laukimo taisykles ir gijos steko dydį.

1.1.2. Paskirstytosios atminties sistemos

1.1.2.1. Architektūra

Kaip ir su bendros atminties sistemomis, paskirstytosios atminties sistemos tarpusavyje stipriai skiriasi, bet turi jas vienijančių savybių. Paskirstytosios atminties sistemoms reikia tinklo, kuriuo galima būtų sudaryti ryšį tarp procesorių. Tinklo tipas naudojamas sujungimui gali būti įvairaus pobūdžio, net kažkas paprasto kaip internetas. Paskirstytosios atminties atveju, kiekvienas procesorius turi savo atmintį. Vieno procesoriaus atminties adresai nėra susieti su kito procesoriaus atminties adresais, taigi globali adresų erdvė neegzistuoja. Kadangi kiekvienas procesorius turi savo atmintį – jis gali veikti laisvai prieinant turimą atmintį. Pakeitimai lokalioje procesoriaus atmintyje nedaro jokios įtakos kitiems procesoriams. Sąvoka talpyklos vientisumas neegzistuoja. Kai procesoriui prireikia duomenų iš kito procesoriaus, paprastai, tai programuotojo darbas apibrėžti kaip ir kada bus apsikeitinėjama duomenimis. Sinchronizacija tarp procesorių taip pat tampa programuotojo atsakomybė.

Kalbant apie paskirstytųjų sistemų privalumus ir trūkumus verta paminėti, kad tokios sistemos yra lengviau praplečiamos, nes galima pasiekti didelio masto lygiagretinimą, naudojantis paprasta technine įranga ir kompiuterių tinklais. Šiose sistemose atskiri procesoriai gali netrukdomai

dirbti su atmintimi ir be papildomo valdymo kaštų, susijusių su talpyklos vientisumo palaikymu. Sistemoje lengva plėsti turimus resursus. Tiek atmintis, tiek procesoriai gali būti prijungti prie egzistuojančios sistemos proporcingai auginant jos pajėgumą. Pagrindiniai trūkumai paskirstytosios atminties sistemose kyla iš poreikio programuotojams patiems būti atsakingiems už daugumą detalių, susijusių su bendravimu tarp procesorių. Iš to kyla sunkumai pernešti duomenų struktūras, egzistuojančias procesoriaus atmintyje kitiems procesoriams. Šiose sistemose stipriai pasireiškia nevienodas atminties pasiekimo laikas, nors duomenys esantys lokalioje atmintyje yra pasiekiami greitai, gauti reikiamus duomenims iš kito procesoriaus gali užtrukti sąlyginai labai ilgai.

1.1.2.2. MPI

MPI[4] (*angl., Message Passing Interface*) yra žinučių siuntimo bibliotekos standartas, sukurtas *MPI* forumo susitarimu, kuriame dalyvauja daugiau nei 40 organizacijų, priklausančių tiekėjams, mokslininkams bei programuotojams. Šio forumo tikslas buvo sukurti lankstų, našų ir pritaikomą daugeliui platformų standartą, skirtą naudoti, kuriant žinutes tarp procesorių siunčiančias programas. Šiam tikslui *MPI* yra pirmas standartizuotas ir nuo tiekėjų nepriklausantis standartas. Nors *MPI* ir nėra IEEE ar ISO standartas, tačiau tai tapo industrijos standartu, rašant žinutes tarp procesorių siunčiančias programas didelio našumo skaičiavimo platformose.

Vis dėl to *MPI* yra tik specifikacija kūrėjams ir naudotojams, kaip turi atrodyti žinutes tarp procesorių siunčianti biblioteka, tai nėra pati biblioteka. *MPI* pagrinde aprašo lygiagretaus programavimo modelį siunčiant žinutes: duomenys yra pernešami iš vieno procesoriaus adresų erdvės į kito procesoriaus adresų erdvę veiksmo metu, kuriuo abu procesoriai turi aktyviai siųsti ir klausytis žinutes. *MPI* standartas per savo gyvenimo metus turėjo kelias peržiūras ir šiuo metu naujausios versijos yra *MPI*-3.x. Realūs *MPI* įgyvendinimai gali skirtis naudojamomis standarto versijomis ir funkcionalumu, kurį palaiko, taigi, programuotojams reikia turėti omenyje šiuos skirtumus.

Originaliai *MPI* buvo sukurta paskirstytosioms atminties sistemoms, kurios *MPI* išleidimo metais stipriai populiarėjo. Architektūrinėmis tendencijomis keičiantis, buvo pradėta jungti bendros atminties procesorius į tinklus taip sukuriant hibridines paskirstytosios / bendros atminties sistemas. Nuo to laiko *MPI* įgyvendinimai buvo pritaikytos veikti su abiem atminties tipais. Įgyvendinimai taip pat buvo išvystyti palaikyti įvairias mazgų jungtis ir protokolus. Šiandien *MPI* veikia praktiškai su visomis platformomis ir tinklais. Bet programavimo modelis vis tiek aiškiai išlieka paskirstytosios atminties, nepriklausomai nuo mašinos fizinės architektūros.

Nors *MPI* programavimo sąsaja buvo standartizuota, kiekvienas įgyvendinimas skiriasi palaikoma *MPI* versija ar funkcionalumu. Taip *MPI* programos gali būti kompiliuojamos ir leidžiamos skirtingai ant skirtingų platformų. Vienas populiariausių *MPI* įgyvendinimų taip pat naudojamų Matematikos ir informatikos fakulteto Paskirstytųjų skaičiavimų centre yra *OpenMPI*. *OpenMPI* yra atviro kodo *MPI* standarto įgyvendinimas, sukurtas ir palaikomas konsorciumo akademinių ir industrinių partnerių. *OpenMPI* yra prieinama visiems Linux klasteriams.

[4] `https://www.mpi-forum.org/`

1.1.3. Metrikos

Išlygiagretintomis programomis paprastai siekiama dviejų tikslų. Pirma, tai našumo – galimybės sumažinti uždavinio sprendimo laiką proporcingai padidinus prieinamų resursų kiekį. Antra – mastelio keitimo – galimybės padidinti našumą, kai išauga uždavinio sudėtingumas ar dydis. Pagrindiniai faktoriai, ribojantys šiuos tikslus yra architektūriniai ir algoritminiai apribojimai. Architektūriniai apribojimai susideda iš latentinio laiko ir duomenų srauto, atminties talpos ir duomenų koherentiškumo. Pagrindiniai algoritminiai apribojimai yra nuoseklus kodas, komunikacijos ar sinchronizacijos dažnumas ir prastas užduoties vykdymo planavimas.

Kalbant apie našumo metrikas, galima išskirti pastarąsias kaip: naudojamas matuoti procesorių našumą ir programos našumą. Šiame darbe didžiąją dalimi yra susitelkiama į programos našumo metrikas. Aplikacijos našumas paprastai tiriamas lyginant įvairaus dydžio uždavinius ar metrikas, gautas leidžiant programą, naudojantis keliais procesoriais ir vienu procesoriumi. Geriausiai žinomos metrikos yra:

- Spartinimas (*angl., speedup*) – greitumo matas. Tai yra santykis tarp nuoseklaus ir lygiagretaus algoritmo vykdymo laiko:
$$S(p) = \frac{T(1)}{T(p)}.$$
- Našumas (*angl., efficiency*) - kompiuterinių resursų panaudojimo matas. Tai yra santykis spartinimo ir procesorių skaičiaus:
$$E(p) = \frac{S(p)}{p}.$$
- Dubliavimas (*angl., redundancy*) – padidėjusio skaičiavimo pajėgumo reikalingumo su išaugusių procesorių skaičiumi matas. Tai yra santykis operacijų skaičiaus, kurį turi atlikti lygiagretusis ir nuoseklusis algoritmai:
$$R(p) = \frac{O(1)}{O(p)}.$$
- Utilizacija (*angl., utilization*) – gero skaičiavimo pajėgumų išnaudojimo matas. Tai yra sandauga tarp našumo ir dubliavimosi:
$$U(p) = R(p) \times E(p).$$
- Kokybė (*angl., quality*) – lygiagretinimo naudojimo aktualumo matas:
$$Q(p) = \frac{S(p) \times E(p)}{R(p)}.$$

Be metrikų yra ir taisyklių, padedančių įvertinti išlygiagretintų programų potencialą, iš kurių populiariausia yra:

- Amdahl'o taisyklė (*angl., Amdahl law*) – gali būti naudojama nustatyti ribą maksimaliam spartinimui, kurią gali pasiekti programa naudojant p procesorių. Taisyklės išraiška:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}},$$

  čia f yra skaičiavimo dalis, kurios negalima išlygiagretinti.

Apibendrinant šią temą galima pasakyti, kad programos efektyvumas yra funkcija, kuri mažėja, augant procesorių skaičiui, ir paprastai didėja, augant uždavinio dydžiui. O programa vadinama nepriklausomo nuo mastelio, kai efektyvumas yra išlaikomas didinant procesorių skaičių ir uždavinio dydį. Nepriklausomumas nuo mastelio rodo gebėjimą efektyviai išnaudoti turimus resursus.

1.2. Nuoseklieji šakų ir rėžių algoritmai

Šiame skyriuje bus trumpai apibūdinti pagrindiniai šakų ir rėžių algoritmų komponentai. Kombinatorinį optimizacijos uždavinį galima spręsti metodais, pilnai perrenkančiais sprendinių aibę. Sprendžiant uždavinį $P : Z(P) = min_{x \in S} f(x)$, f yra tikrinių verčių funkcija, o S tikrinių verčių vektoriaus V poaibis. Yra laikoma, kad P gali būti išspręsta perrenkant baigtinį skaičių taškų (nebūtinai žinomų iš anksto), esančių S, ir kad uždavinys yra NP-sunkus, tai reiškia, kad nėra žinomas algoritmas, galintis išspręsti šį uždavinį per polinominį V dimensijos atžvilgiu laiką ar greičiau. Taip pat laikoma, kad arba uždavinys neturi sprendinių ($S = \emptyset$) arba turi optimalų sprendinį ($Z(P) > -\infty$). Problema yra intensyvi skaičiavimais, o skaldyk ir valdyk sprendimo strategija uždavinio nepalengvina.

BB sprendimo būdas susidaro iš netiesioginio perrinkimo S patikrinant tik galimų sprendinių poaibį. Kiti sprendiniai, kurie negali vesti prie galimo arba optimalaus sprendinio, yra atmetami.

Sprendinių perrinkimas susideda iš BB medžio konstravimo, kur kiekvienas mazgas yra poaibis sprendžiamo uždavinio sprendinių aibės. Medžio dydis, sugeneruotų mazgų skaičius, yra tiesiogiai susijęs su pasirinkta medžio generavimo strategija.

Šakų ir rėžių algoritmai gali būti apibendrinti į šias dalis:

Paieškos medžio konstravimas

- Šakojimosi sistema dalijanti V į vis mažesnius ir mažesnius poaibius siekiant gauti uždavinį, kurį galima lengvai išspręsti. BB medžio mazgai reprezentuoja sprendinių poaibius, medžio briaunos – sąryšius tarp medžio tėvų ir vaikų sugeneruotų šakojimosi metu, o medžio lapai – galimus sprendinius ar atmestus poaibius.

- Paieškos ar tyrinėjimo strategija, pasirenkanti vieną iš dar neapdorotų mazgų pagal iš anksto pasirinkus prioritetus. Paprastai naudojamas pasirinkimo strategijos, pagrįstos mazgo gyliu medyje (*angl., Depth-First*) arba pagal numatytą mazgo gebėjimą turėti gerą sprendinį (*angl., Best-First*).

Šakojimasis

- Rėžių funkcija, apskaičiuojanti apatinį rėžį mazgui, sukurtam po šakojimosi.
- Tyrinėjimo intervalas, apribojantis pasirenkamus mazgus. Tik tie mazgai, kurių apatinis rėžis yra geresnis nei žinomas viršutinis rėžis yra tyrinėjami, kiti mazgai yra atmetami. Viršutinis rėžis gali būti pateiktas nuo pradžių, kaip jau žinomas uždavinio sprendinys arba gautas kitais būdais pvz., apskaičiuotas euristiniais metodais. Viršutinis rėžis vis keičiasi, randant geresnius sprendinius.

Stabdymo sąlyga

- Sąlyga nurodo, kada uždavinys išspręstas ir optimalus sprendinys rastas. Tai atsitinka kai visi mazgai būna išspręsti arba eliminuoti. Ši dalis yra ganėtinai paprasta kuriant nuoseklius algoritmus, bet tampa sudėtingesnė keliems procesams tyrinėjant medį. Šiuo atveju procesas turi nebaigti darbo, kol kitas procesas dar dirba.

Turimam uždaviniui išspręsti galima naudoti skirtingas medžio konstravimo ir šakojimosi operacijas efektyviau arba ne sprendžiančias kokretų uždavinį. Be to, šakojimosi ir medžio konstravimo operacijos gali būti priklausomos tarpusavyje ir (ar) priklausomos nuo proceso istorijos, t.y. konkrečios veiksmų sekos, kuria buvo atliktos operacijos. Laikas, reikalingas atlikti kiekvienai iš operacijų stipriai priklauso nuo konkretaus uždavinio.

Sprendžiant uždavinį stengiamasi kuo greičiau gauti siaurus rėžius. Siauri rėžiai, ypatingai geras viršutinis rėžis, leidžia sumažinti sugeneruojamų po-uždavinių skaičių atmesdamas visus poaibius, neturinčius optimalaus sprendinio. Tuo būdu yra sumažinamas darbas, reikalingas išspręsti uždavinį.

Paieškos medžio konstravimo strategijos yra nemažiau svarbios BB algoritmams. „Pirma geriausias“ strategijoje geriausias žinomas viršutinis rėžis laikomas Z_{best} kintamajame tam, kad galima būtų greitai testuoti apatinius rėžius. Taip pat, atmintyje yra laikomas sąrašas L įvertintų, bet vis dar neištirtų, uždavinių. Pradžioje Z_{best} pasirenkamas lygus begalybei ir originalus uždavinys padedamas į sąrašą. Kiekviename žingsnyje, jei L nėra tuščias, pasirenkamas po-uždavinys Q, priklausantis L, turintis mažiausią apatinį rėžį. Jei po-uždavinys nėra eliminuojamas, jis skaidomas pagal šakojimosi operaciją. Tada kiekvienas naujai sugeneruotas po-uždavinys yra įvertinamas ir, jei kažkurio viršutinis rėžis yra geresnis nei Z_{best}, Z_{best} yra atnaujinamas. Likę po-uždaviniai yra testuojami, ar nereikia jų atmesti, ir, jei ne, yra įtraukiami į sąrašą L. Šie žingsniai yra kartojami, kol L tampa tuščias. Ši medžio konstravimo strategija (pirma geriausias) yra efektyviausia laikyti sąrašą *heap* duomenų struktūroje. Pagrindinis šios strategijos privalumas yra tai, kad ši strategija yra optimali išskaidytų po-uždavinių skaičiaus atžvilgiu, kai rėžių ir šakojimosi operacijos nepriklauso nuo proceso istorijos. Trūkumas – gali prireikti didelių atminties resursų laikyti sąrašui L.

„Pirma gylis“ yra kita strategija paieškos medžiui konstruoti. Šiuo atveju sąraše L yra neįvertinti ir netyrinėti po-uždaviniai. Kaip ir „pirma geriausias“ strategijoje pradžioje Z_{best} taip pat prilyginimas begalybei ir originalus uždavinys yra išskaidomas pagal šakojimosi procedūrą. Kiekviename žingsnyje, jei buvęs po-uždavinys buvo išskaidytas, visi nauji po-uždaviniai yra dedami į

sąrašą L, išskyrus vieną po-uždavinį, kuris yra įvertinamas ir ištyrinėjamas. Jei buvęs po-uždavinys buvo eliminuotas, tada pasirenkama po-uždavinys iš sąrašo L, kuris buvo pridėtas paskutinis. Ir tolesni žingsniai pereina prie šios po-uždavinio įvertinimo ir ištyrinėjimo. Kiekvieną kartą, kai po-uždavinys yra įvertinamas, Z_{best} yra atnaujinamas, jei to reikia. Žingsniai atliekami tol, kol L tampa tuščias. Steko duomenų tipo struktūra geriausiai tinka laikyti „pirma gylis“ strategijos generuojama po-uždavinių sąrašą L. Šis metodas turi tris pagrindinius privalumus. Pirma, iš visų paieškos medžių konstravimo strategijų ši minimizuoja atminties reikalavimus. Antra, kai po-uždavinys nėra eliminuojamas, nemaža dalis informacijos, sugeneruotos per paskutinę rėžių tikrinimo operaciją, gali būti tiesiogiai prieinama sekančiai operacijai. Trečia, galimi sprendiniai randami greičiau, nei kitomis strategijomis. Tai ypač svarbu, kai viršutiniai rėžiai yra skaičiuojami tik po-uždaviniams atitinkantiems paieškos medžio lapus. Šio metodo trūkumas yra, kad jis gali sugeneruoti labai daug po-uždavinių. Šis trūkumas gali būti sumažinamas ankstyvosiose sprendimo stadijose randant gerą viršutinį rėžį, taip sustiprinant apatinio rėžio testą, ko pasekoje bus sugeneruota mažiau po-uždavinių.

Viršuje pateiktos strategijos yra populiariausios, yra ir kitos galimos paieškos medžio konstravimo strategijos bei variacijos. Platesnę strategijų apžvalgą galima rasti [Iba87].

1.3. Lygiagretieji šakų ir rėžių algoritmai

Kompiuterių architektūra daro stiprią įtaką lygiagrečiųjų BB algoritmų kūrimui. Taigi, prieš aptariant lygiagrečiųjų BB algoritmų našumą ir kūrimą, bus aptartas lygiagretumas techninės įrangos lygmenyje. Tai ne tas pats kaip lygiagretumas programinės įrangos lygmenyje, kur keli procesai gali simuliuoti lygiagretumą pasidalindami to paties procesoriaus resursais.

1.3.1. Lygiagretumas techninėje įrangoje

Šiame darbe kalbama tik apie lygiagrečias kompiuterių architektūras, sudarytas pagal srauto valdymo (*flow-control*) modelį, t.y. kiekvienas procesorius sistemoje vykdo komandas eilės tvarka, nustatyta valdymo įrenginio. Kiti žinomi modeliai yra duomenų srauto (*data-flow*) modelis, kuriame procesorius atlieka operacijas pagal įvedimo duomenų prieinamumą ir užklausos srauto (*demand-flow*) modelis, kuriame procesoriai atlieka operacijas eilės tvarka, nustatyta iš reikalavimų duomenims. Srauto valdymo modeliai, turintys tik vieną valdymo įrenginį, priklauso SIMD (viena komanda daug duomenų, *single instruction multiple data*) kategorijai, modeliai, turintys daugiau valdymo įrenginių (paprastai po vieną kiekvienam procesoriui), priklauso MIMD (daug komandų daug duomenų, *multiple instructions multiple data*) kategorijai.

Sinchronizacijos būvimas reiškia globalų laikrodį, sinchronizuojantį procesorių darbą. Kai yra tiktai vienas laikrodis, sistema vadinama sinchronine, jei yra keli laikrodžiai (paprastai vienas procesoriui), sistema vadinama asinchronine. SIMD architektūros yra sinchroninės pagal apibrėžimą, MIMD architektūros paprastai būna asinchroninės.

Sistemos grūdėtumas nurodo, kokio dydžio duomenys bus apdorojami procesorių. Smulkiagrūdėse sistemose procesoriai dirba tik su smulkiais įėjimo vektoriais ar skaliariniais dydžiais.

Stambiagrūdėse sistemose procesoriai dirba su dideliais duomenų kiekiais.

Esanti komunikacija nurodo, kaip tarpusavyje bendrauja procesoriai. Yra du pagrindiniai būdai bendrauti procesoriams. Pirmasis yra bendrauti per bendrą atmintį (bendros atminties sistemos), t.y. skirtingi procesoriai nuskaito ir įrašo duomenis į tą patį atminties regioną. Antrasis būdas yra bendravimas žinutėmis (žinučių siuntimo sistemos), kai kiekvienas procesorius siunčia informaciją ar užklausą informacijai tiesiai į kitą procesorių. Bendros atminties sistemos dar yra skirstomos, į turinčias bendrą fizinę atmintį ar mechanizmą, leidžiantį iš kiekvieno procesoriaus prieiti, bet kurią atminties vietą, jos vadinamos atitinkamai stipriai ir silpnai sujungtos. Žinutes perdavinėjančios sistemos gali būti skirstomos pagal tinklo sujungimo topologiją, kuri apibūdina kaip yra sujungti procesoriai tarpusavyje. Dažniausios topologijos yra žiedo, medžio, tinklelio ir hiperkūbo (daugiau detalių galima rasti [BT89]).

Galiausiai yra svarbus procesorių skaičius. Masiškai lygiagrečios sistemos sudarytos iš procesorių skaičiaus, esančio dešimties tūkstančių eilės dydžio. Smulkiagrūdės sistemos paprastai būna masiškai lygiagrečios, tuo tarpu stambiagrūdės sistemos paprastai nėra masiškai lygiagrečios. Verta pastebėti, kad stipriai sujungtos atminties sistemos turi ribotą procesorių skaičių, paprastai ne daugiau 20, dėl sudėtingumo sudarant visiems procesoriams vienu metu prieinamą atmintį neturinčios našumo ydų.

Programinės įrangos lygmenyse šie padalijimai nėra tokie griežti kaip techninės įrangos lygmenyje, nes daugelį savybių galima imituoti pasinaudojant atitinkamomis programinės įrangos mechanizmais. Nors našumas tokių imitavimų paprastai yra mažas. Pavyzdžiui, MIMD sistema gali imituoti SIMD sistemą arba asinchroninė sistema – sinchroninę. Likusioje apžvalgos dalyje kalbant apie lygiagretinimą bus daugiau turimą omenyje programinės įrangos lygmenį, nors yra stipri sąsaja tarp lygiagrečių algoritmų sampratos programinės įrangos lygmenyje ir jų įgyvendinimo lygiagrečiose architektūrose.

1.3.2. Šakų ir rėžių algoritmų lygiagretinimo būdai

Yra du pagrindiai būdai išlygiagretinti šakų ir rėžių algoritmus. Lygiagretinimas mazgų pagrindu naudojamas, norint išlygiagretinti konkrečias operacijas šakojant ir tiriant po-uždavinį, pavyzdžiui, atliekant rėžių tikrinimo operaciją lygiagrečiai. Toks lygiagretinimas nedaro įtakos šakų ir rėžių algoritmo struktūrai ir tik ribotai paspartina jo veikimą. Toks lygiagretinimo įgyvendinimas paprastai būna savitas kiekvienam uždaviniui. Lygiagretinimas medžio sudarymo pagrindu, naudojamas siekiant konstruoti BB medį lygiagrečiai sprendžiant kelis po-uždavinius vienu metu. Toks lygiagretinimas gali paveikti pačio algoritmo struktūrą ir paprastai pasiekia didesnį spartinimą.

Šios lygiagretinimo rūšys gali būti tarpusavyje jungiamos nuosekliai ar hierarchiškai. Nuoseklus BB lygiagretinimo rūšių sujungimas įgyvendintas [PM90] darbe, kur pagrindinio mazgo konstravimas vyko naudojant išlygiagretintą rėžių skaičiavimo operaciją, o likusiam uždaviniui lygiagrečiai konstravo medį. Hierarchinio atvejo pavyzdys yra [MP93] darbas, kuriame keli BB medžiai yra konstruojami lygiagrečiai, kiekvieno iš tų medžio sudarymas yra išlygiagretintas ir kiekvienas iš po-uždavinių yra sprendžiamas lygiagrečiai. Paprastai BB algoritmai naudoja tik vieną lygiagretinimo būdą.

Literatūroje dažniausiai sutinkamas lygiagretinimo būdas medžio sudarymo pagrindu. Šiam būdui tinka įgyvendinimas stambiagrūdėms asinchroninėms MIMD sistemoms, nors yra darbų, įgyvendinančių lygiagretinimą medžio sudarymo pagrindu ir ant smulkiagrūdžių sinchroninių SIMD sistemų ([KT88], [DFR90]). Tačiau smulkiagrūdės sistemos yra tinkamos tik algoritmams, reikalaujantiems nedaug atminties, ypač atliekant rėžių tikrinimo operaciją. SIMD atveju atliekamos instrukcijos turi būti vienodos visiems procesoriams, kas nėra labai tinkama BB algoritmams, nes rėžių skaičiavimo operacija gali skirtis kiekvienam po-uždaviniui ([KT88]). Smulkiagrūdės ir SIMD architektūros parodė, kad visų procesorių sinchronizacijos valdymas visiems procesoriams yra per daug brangus algoritmo našumo atžvilgiu.

1.3.3. Lygiagrečiųjų šakų ir rėžių algoritmų klasifikacija

Šiame darbe aptariama lygiagrečiųjų BB algoritmų klasifikacija, pasiūlyta [GC94] darbe. Norint suklasifikuoti lygiagretinimą medžio sudarymo pagrindu, asinchroninės MIMD sistemoms, pirma, reikia atskirti sinchroninius ir asinchroninius algoritmus. Sinchroninis algoritmas yra padalintas į fazes taip, kad kiekvienos fazės metu procesoriai dirba nepriklausomai, o procesorių bendravimas vyksta tik po fazės, procesai turi tapti sinchronizuoti prieš pasikeisdami informacija. Dar galima išskirti griežtai ir silpnai sinchronizuotus procesus. Pirmu atveju, komunikacijos protokolas (kuri informacija ir kur siunčiama) nesikeičia su kiekvienu programos vykdymu. Tokie algoritmai elgiasi deterministiškai t.y. kad su kiekvienu to paties uždavinio sprendimu, uždavinys seks ta pačia eiga. Antru atveju, algoritmas gali nesilaikyti identiškos bendravimo sekos sprendžiant tą patį uždavinį, kai, pavyzdžiui, komunikacijos protokolas priklauso nuo informacijos, žinomos tik programos veikimo metu. Kai vykdomas asinchroninis algoritmas, bendravimas gali vykti bet kuriuo metu ir yra nenuspėjamas. Dėl to tokie algoritmai pasižymi nedeterministiniu elgesiu.

Antras parametras naudojamas klasifikavime yra susijęs su darbų sąrašo, kuriame procesai laiko ir ieško sugeneruotų, bet dar neištirtų po-uždavinių, vieta atmintyje. Paprastai procesas ieškodamas darbo paima uždavinį iš darbų sąrašo ir įvertina ar ištiria jį. Kai procesas baigia nagrinėti uždavinį, sugeneruoti uždaviniai yra padedami į kitą ar tą patį darbų sąrašą. Procesas, taip pat, gali atlikti veiksmus nepriklausomai nuo egzistuojančių darbų sąrašų. Pavyzdžiui, naujai sugeneruotas po-uždavinys gali būti įvertintas ir ištirtas be jo įrašymo ir pasiėmimo iš darbų sąrašo. Klasifikacijoje yra atskiriami vieno ir kelių darbų sąrašų algoritmai.

Kai yra tik vienas darbų sąrašas, jis yra saugomas vienoje atminties vietoje. Taip paprastai būna įgyvendinti nuoseklūs BB algoritmai, kai visi darbai yra saugomi viename sąraše. Verta pastebėti, kad darbų sąrašas gali būti padalintas į dvi atskiras dalis ([MP89], [PM90]): viena dalis saugoja neįvertintus ir neištirtus po-uždavinius, o kita įvertintus, bet neištirtus po-uždavinius. Vieno darbų sąrašo algoritmai gali būti įgyvendinti bendros atminties sistemose. Žinutes siunčiančiuose architektūrose vieną darbų sąrašą galima įgyvendinti pasinaudojus šeimininko-darbininko (*master-worker*, MW) paradigma, kai vienas procesas vadinamas šeimininku prižiūri darbų sąrašą ir siunčia darbus kitiems procesams, darbininkams, kurie atlieka darbus ir siunčia rezultatus atgal šeimininkui.

Daugelio darbų sąrašo algoritmuose yra kelios vietos kur procesai gali rasti darbų sąrašus.

Kelios struktūrinės schemos yra galimos, iš kurių trys populiariausios yra: atskiras, grupinis ir maišytas darbų sąrašas. Naudojant atskirą darbo sąrašo schemą kiekvienas procesas turi savo darbų sąrašą. Grupinėje schemoje procesai yra suskirstyti į grupes, ir kiekviena grupė turi savo darbų sąrašą. Atskirą schemą gali laikyti grupinės schemos atveju, kai grupės sudarytos tik iš vieno proceso. Maišytoje schemoje kiekvienas procesas turi savo darbų sąrašą, bet, taip pat, egzistuoja ir bendras darbų sąrašas visai grupei.

Apibendrinant galima suklasifikuoti visus BB algoritmus skirtus MIMD architektūroms į keturias grupes: sinchroninius vieno darbų sąrašo (*Synchronous Single Pool*, *SSP*), asinchroninius vieno darbų sąrašo (*Asynchronous Single Pool*, *ASP*), sinchroninius daugelio darbų sąrašų (*Synchronous Multiple Pool*, *SMP*) ir asinchroninius daugelio darbų sąrašų (*Asynchronous Multiple Pool*, *AMP*) algoritmus.

1.3.4. Lygiagrečiųjų šakų ir rėžių algoritmų problemos

Viena iš problemų yra darbų paskirstymas pradedant uždavinio sprendimą. Tuo metu egzistuoja tik vienas pradinis mazgas, BB medžio šaknis, prieinamas visiems procesams. Kadangi šakojimosi operacija sukuria tik ribotą skaičių po-uždavinių, egzistuoja startinė algoritmo eigos fazė, kada lygiagretinimas nėra pilnai išnaudojamas. Šios fazės sunku išvengti ir dėl to, kad priskirti darbus procesams vos tik darbams atsiradus ne visada yra geras sprendimas. Šį atvejį pavaizduojantis pavyzdys: nuoseklusis algoritmas naudojantis „pirma gylis“ paiešką ir skaidantis uždavinį į du po-uždavinius. Dažnai skaidymą stengiamasi įgyvendinti taip, kad tik vienas iš po-uždavinių turėtų didelę tikimybę vesti prie optimalaus sprendinio. „Pirma gylis“ paieška pasirinktų geresnį po-uždavinį, kito po-uždavinio įvertinimas ir ištyrimas būtų paliktas vėlesniam laikui, kai pasirinkto po-uždavinio po-medis būtų pilnai ištirtas. Jei ištirtas po-medis turėjo optimalų ar gerą sprendinį, egzistuoja tikimybė, kad kitas po-medis bus iš karto atmestas. Jei tokia pat šakojimosi procedūra bus panaudota lygiagrečiojo algoritmo atveju ir procesai ims uždavinius iš karto jiems tapūs prieinamiems, abiejų po-uždavinių po-medžiai bus tiriami vienu metu. Taigi po-medis prasidedantis po-uždaviniu, kurio nuoseklusis algoritmas būtų nesprendęs, bus sprendžiamas, ko padarinys yra daugiau atlikto darbo lyginant su nuosekliuoju algoritmu. Tai vadinama žalinga anomalija ([LW84]), kai lygiagretusis algoritmas reikalauja daugiau darbo išspręsti tą patį uždavinį nei nuoseklusis algoritmas. Taigi, siekiamumas yra visiems procesams duoti darbo kuo anksčiau tuo pačiu metu, išvengiant duoti po-uždavinių, turinčių mažą tikimybę vesti prie optimalaus sprendinio.

Kelios strategijos gali būti naudojamos šiai problemai spręsti: 1) originalų uždavinį duoti vienam procesui ir palaipsniui perduoti sugeneruotas problemas kitiems procesams, 2) sprendimo pradžioje generuoti daugiau po-uždavinių nei paprastai taikant kitokią šakojimosi operaciją, 3) atlikti nuoseklųjį BB algoritmą iki kol sugeneruotų po-uždavinių skaičius tampa pakankamas visiems procesams, 4) atlikti nuoseklųjį to paties medžio konstravimą visiems procesams iki kol po-uždavinių skaičius pasidaro pakankamas ir kiekvienas procesas išsirenka atskirus po-uždavinius. Pirmi trys pasiūlymai gali būti naudojami visiems algoritmams, o paskutinis paprastai naudojamas daugelio darbo sąrašų algoritmuose. Konkrečios strategijos pasirinkimas priklauso nuo jo tinkamumo uždaviniui bei programinei įrangai.

Kitas po pradinio darbų paskirstymo svarbus dalykas yra tinkama strategija tolesniam darbų skirstymui ir dalijimui tarp visų procesų. Šiuo atveju siekiama, kad darbo krūvis būtų panašus visiems procesams ir kad jie gautų tik uždavinius, turinčius didelę tikimybę turėti optimalų sprendinį, norint išvengti atvejų, kai nuoseklusis algoritmas atliktų mažiau darbo nei lygiagretusis sprendžiant tą patį uždavinį. Pasiekti šiuos rezultatus yra lengviausia vieną darbų sąrašą naudojantiems algoritmams. Daugelių darbų sąrašų algoritmuose situacija daug sudėtingesnė. Vienas iš sprendimo būdų tinkamai suvaldyti daugelį darbų sąrašų yra dinamiškai kurti procesus, kurie paimtų dalį darbų iš darbų sąrašo ([JS89], [HCH+13a]). Dažniau pasitaikanti situacija yra, kai turimas fiksuotas procesų skaičius ir pastariesiams leisti keistis darbais tarpusavyje. Šiuo atveju galima išskirti dinaminius ir statinius paskirstymus. Statinio paskirstymo atveju, procesai apsikeičia ar dalinasi po-uždaviniais programos paleidimo metu apibrėžtais kriterijais, paprastai būna darbų paskirstymas pačioje uždavinio sprendimo pradžioje. Šis paskirstymas daugiausia naudojamas maišytus darbų sąrašus turintiems algoritmams, kur globalus darbų sąrašas naudojamas sekti darbams, o kiekvienas procesas atlieka nuoseklųjį BB algoritmą (naudodamas savo lokalų darbų sąrašą) imdamas uždavinius iš globalaus darbų sąrašo, kai lokalus darbų sąrašas tampa tuščias. Statinio paskirstymo atveju nėra jokio darbų apsikeitimo tarp lokalių darbų sąrašų. Dinaminis paskirstymas leidžia dalintis darbais, esančiais lokaliame darbų sąraše. Sprendimas, kaip atlikti paskirstymus, paprastai yra priimamas atitinkamo lokalaus darbų sąrašo proceso. [GC94] išskiria tris strategijas dinaminiam darbų paskirstymui:

1. **Strategija su prašymu**. Šiuo būdu procesas su beveik tuščiu darbų sąrašu siunčia prašymą kitam procesui. Prašymas gali būti priimtas ir atsiųsta dalis kito proceso darbo sąrašo arba atmestas ir procesas gali priimti sprendimą siųsti prašymą kitam procesui. Jei gavęs prašymą procesas nusprendžia jį priimti, jam reikia nuspręsti, kiek ir kuriuos uždavinius siųsti iš lokalaus darbų sąrašo.

2. **Strategija be prašymo**. Šiuo būdu procesai gali nuspręsti pasidalinti savo darbų sąrašais nesulaukus prašymo iš kito proceso. Prieš siunčiant uždavinius kitiems procesams, reikia nuspręsti, kaip dažnai juos siųsti, kam siųsti uždavinius, ir kaip nuspręsti, kuriam procesui siųsti.

3. **Maišyta strategija**. Šitas būdas apjungia prieš tai buvusius. Procesai gali dalintis uždaviniais iš darbų sąrašo neprašomi ir siųsti prašymus, kai patiems trūksta darbo.

Dar viena problema lygiagretinant BB algoritmus yra susijusi su šakų atmetimu po apatinio rėžio testo. Apatinio rėžio testui atlikti, procesui reikia žinoti viršutinį rėžį sprendžiamam uždaviniui. Siekiama, kad visi procesai visada turėtų pačius naujausius viršutinius rėžius, kitaip apatinio rėžio testas nebus toks efektyvus. Taigi reikia dalintis viršutiniais rėžiais tarp procesų. Būdai pasidalinti viršutinį rėžį tarp procesų stipriai priklauso nuo konkretaus sprendžiamo uždavinio ir techninės įrangos. Asinchroniniams algoritmams, veikiantiems bendros atminties sistemose, geriausia dalintis viršutiniu rėžiu laikant kintamąjį prieinamą visiems procesams bendroje atmintyje. Kai vienas iš procesų randa geresnį viršutinį rėžį, jam tereikia atnaujinti visiems prieinamą kintamąji saugantį

viršutinį rėžį. Asinchroniniuose algoritmuose, veikiančiuose žinutes perdavinėjančiose sistemose, kiekvienas procesas gali turėti savo kintamąjį, laikantį geriausią viršutinį rėžį. Kai procesas randa geresnį rėžį, jis perduodą šią informaciją kitiems procesams išsiųsdamas žinutes. Paprastai informacija yra perduodama iškart visiems procesams, tačiau perdavinėjimas turi būti nutrauktas, jei kitas procesas pradeda perduoti dar geresnį rėžį. Tokių perdavimų našumas priklauso nuo naudojamos sistemos topografijos.

Sprendimo užbaigimo aptikimas yra dar viena problema. Ji yra triviali algoritmams, naudojantiems vieną darbų sąrašą ir SMP algoritmams. Tikroji problema yra aptikti sprendimo užbaigimą AMP tipo algoritmuose. Sąlyga, kad visi darbų sąrašai yra tušti, nėra pakankama paskelbti užbaigimą, nes keli procesai vis dar gali dirbti ir, baigus skaičiavimus, papildyti darbų sąrašą, arba žinutės su darbais dar gali keliauti tarp procesų. Ši problema nėra išskirtinė BB algoritmams, o yra būdinga visoms paskirstytosioms sistemos be centrinio valdančio įrenginio. Metodai, skirti aptikti užbaigimą, tokiais atvejais yra tiriami [DS80].

1.4. Egzistuojančios lygiagrečiųjų šakų ir rėžių algoritmų bibliotekos

Norint įgyvendinti lygiagretųjį šakų ir rėžių algoritmą, galima pasinaudoti arba esama biblioteka arba įgyvendinti algoritmą pačiam. Bibliotekos skirtos šakų ir rėžių algoritmų lygiagretinimui turi privalumą prieš individualų algoritmo įgyvendinimą pagal poreikį. Bibliotekos standartizuoja algoritmus, taip jie tampa labiau prieinami. Svarbiausia, su bibliotekos sukurti algoritmai leidžia vienareikšmiškai lyginti spartinimą tarpusavyje ir rezultatai lengviau gali būti pakartoti. Gebėjimas pakartoti rezultatus leistų keliems skirtingiems autoriams tobūlinti jau esamus lygiagrečiųjų šakų ir rėžių algoritmų įgyvendinimus. Aišku, bibliotekų naudojimas turi ir trūkumų. Pateikdamos šabloną bibliotekos dažnai apriboja galimybę sukurti algoritmą geriausiai atitinkantį konkretų uždavinį ir techninę įrangą. Dėl to gali atsitikti taip, kad biblioteka gali būti pritaikoma tik ribotam optimizacijos uždavinių skaičiui ir neleistų išnaudoti visų, ypač techninių, prieinamų galimybių. Taigi, pasinaudojus bibliotekomis algoritmai paprastai būna mažesnio našumo.

Šio skyriaus siekis yra ištirti esamų šakų ir rėžių algoritmo lygiagretinimo bibliotekų galimybes ir jų tinkamumą hierarchiniam lygiagretinimui. Patikrinus literatūrą buvo rastos bibliotekos: PPBB [TP96], Zram [BMF+99], SYMPHONY [RG05], PICO [EPH01], PeBBL [EHP15], BCP [Sal02], ALPS/BiCePS [XRL+05], Bob++ [DLC+06], Mallba [AAB+02]. Apžvelgiant pateiktas bibliotekas dauguma jų nepavyko ištestuoti. Šiame skyrelyje bus trumpai aptartos šios bibliotekos.

PPBB[5] kaip biblioteka siekė šių tikslų: 1) šakų ir rėžių algoritmo sąsajos atskyrimas nuo komunikacijos procesų tam, kad šakų ir rėžių algoritmas būtų nepriklausomas nuo apkrovos balansavimo ir komunikacijos būdų. 2) Apkrovos balansavimo sąsajos atskyrimas nuo komunikacijos būdų ir šakų, ir rėžių algoritmo, kad būtų prieinami keli apkrovos balansavimo būdai. 3) Pateikti prioritetinės eilės valdymo funkcijas įvairiems šakų ir rėžių algoritmo dalims. 4) Pateikti funkcionalumą stebėti lygiagrečius procesus. 5) Pateikti nuo sistemos priklausantį komunikacijos būdą, į kurį įeitų žinučių siuntimas, baigimo aptikimas, buferio valdymas. Deja ši biblioteka yra sena, straipsniai aprašantys šią biblioteką išleisti 1995 metais, o nuo 1996 metų joje nebuvo pakeitimų. Bandant

[5] `http://www2.cs.uni-paderborn.de/cs/ag-monien/SOFTWARE/PPBB/documentation.html`

kompiliuoti biblioteką iškilo nemažai klaidų. Nėra paruoštos nuo sistemos nepriklausančios konfigūracijos, todėl visus reikiamus parametrus reikia sudėlioti pačiam ir net tai atlikus vis tiek kyla problemų kompiliuojant kodą.

PICO[6] stengiasi pateikti hierarchiškai organizuotą funkcionalumo rinkinį, kurį vartotojas gali kombinuoti ir praplėsti savo programomis. PICO sudaryta kaip `C++` klasių biblioteka, taip kūrėjai norėjo suteikti daugiau lankstumo ir laisvės nei atskiros funkcijos. Karkasas sudarytas iš dviejų sluoksnių *nuoseklaus* ir *paralelaus*. *Nuoseklus* sluoksnis laiko apibrėžimus šakų ir rėžių algoritmo objektams. Šios bibliotekos naudotojai negalvojantys apie lygiagretinimą ar tiesiog ankstyvose algoritmo įgyvendinimo stadijose gali naudotis šiuo sluoksniu rašydami ir testuodami savo programas įprastoje nuoseklioje aplinkoje. *Paralelus* sluoksnis leidžia *nuosekliame* sluoksnyje esamą programą paversti į lygiagrečiai vykdomą, tam reikia tik įgyvendinti papildomas kelias klases, kurios aprašys kaip reikia supakuoti ir išpakuoti objektus, kurie bus siunčiami tarp procesų. Tuo būdu visos programos gali naudotis *paralelaus* sluoksnio pilnu funkcionalumu. Su šia biblioteka nebuvo tęsiamas darbas, nes nepavyko rasti viešai prieinamo bibliotekos kodo. Taip galėtų būti dėl to, kad ji nebėra palaikoma. Ją aprašantis straipsnis yra išėjęs 2000 metais.

PeBBL[7] yra naujesnis PICO kūrėjų darbas. PeBBL buvo gautas padalinus PICO ir praplėtus jos dalį atsakingą už bendrą šakų ir rėžių algoritmų įgyvendinimą. Dėl šios priežasties PeBBL pagrindinės struktūros idėjos nėra pasikeitusios nuo PICO. Kaip ir PICO ši biblioteka taip pat nebuvo naudota tolimesniems darbams, nes nebuvo rastas viešai prieinamas bibliotekos kodas, nors šis darbas ir yra daug naujesnis. Straipsnis apie šią biblioteką pasirodė 2013 metais.

BCP[8] (*angl., Branch and Cut and Price*) autoriai stengėsi apriboti šios bibliotekos apimtį, kad nauji vartotojai galėtų greičiau pradėti rašyti programas. Iš vartotojo pusės buvo stengtasi paslėpti bibliotekos įgyvendinimo detales ir konfigūracijas, kurių nereikia paprastam programos įgyvendinimui. BCP kaip biblioteka yra kolekcija klasių ir metodų, skirtų valdyti medžių perrinkimą, rėžius ir kitus kintamuosius. Pats BCP nemoka spręsti LP (*angl., Linear Programming*) problemų ir dėl to naudoja OSI sprendėją, kurį galima rasti nuorodoje: https://projects.coin-or.org/Osi/ . Osi yra naudojamas per atskirą sąsają, todėl neturi būti sunku jį sukeisti su kitu LP sprendėju. BCP biblioteka sprendžia tik su minimizacijos problemas. Ši biblioteka taip pat nebuvo ištestuota. Ją pavyko sukompiliuoti ir rasti tam tikrų pavydžių, tačiau trūko konkretesnių šakų ir rėžių algoritmo pavyzdžių. Trūko ir nuoseklios bei plačios dokumentacijos. Dėl to nepavyko įgyvendinti uždavinių pasinaudojus šia biblioteka.

ALPS[9] yra CHiPPS dalis. CHiPPS sudaro trys dalys iš kurių ALPS yra pagrindinė, kuri atsakinga už bendrą funkcionalumą, o konkretesnius uždaviniui specifinius dalykus palieka BiCePS ir BLIS karkasams. BiCePS (*angl., Branch, Constrain and Price Software*) siūlo pagrindą vadinamiems *Branch*, *Constrain* ir *Price* algoritmams. BiCePS pagrinde yra duomenų valdymo sluoksnis, nes ši bibliotekos dalis paprastai naudojama relaksacijos pagrindu sudarytiems šakų ir rėžių algoritmams. Tokie algoritmai paprastai turi sudėtingas ir dideles duomenų struktūras taigi efektyvus

[6] `https://www.sciencedirect.com/science/article/pii/S1570579X01800148`
[7] `http://www.optimization-online.org/DB\_HTML/2013/10/4077.html`
[8] `https://projects.coin-or.org/Bcp`
[9] `https://projects.coin-or.org/CHiPPS`

duomenų pernešimas yra būtinas. BLIS (*angl., BiCePS Linear Integer Solver*) karkasas atsakingas už vadinamas *mixed integer linear programs*, t.y. problemos kai dalis ar visi kintamieji gali būti tik sveikieji skaičiai. BLIS pagrinde siūlo būdus saugoti ir dalintis informacija apie rėžius, sprendinius ir mazgais. Grįžtant prie pagrindinės dalies, ALPS, jos architektūra pagrįsta idėja, kad visa informacija, kuri yra dinamiškai generuojama gali būti saugoma ir pernešama autorių vadinamuose žinių grupėse (*angl., Knowledge Pools, KPs*). Taigi ALPS pagrinde susideda ir trijų klasių skirtų KPs valdymui ir apibūdinimui, o specializuotų paieškos medžių įgyvendinimas yra tiesiog naujų klasių sudarymas paveldint iš pagrindinių. ALPS biblioteka greičiausiai neseniai neteko palaikymo, iki 2017 metų pradžios ji periodiškai susilaukdavo atnaujinimų, kurių nebėra nuo to laiko. Biblioteka irgi buvo netestuota, nors ir pavyko ją sukompiliuoti ir rasti keliaujančio pirklio uždavinio pavyzdį, tačiau pavyzdys nepasileido. Pateiktas pavyzdys neveikė paleidžiant pagal instrukcijas, o priežasties rasti nepavyko.

Bob++[10] yra karkasas skirtas įgyvendinti tiek lygiagrečiuosius, tiek nuosekliuosius algoritmus. Karkasas leidžia įgyvendinti ne tik šakų ir rėžių algoritmus, bet ir skaldyk, ir valdyk bei dinaminio programavimo uždavinius. Bob++ pagrįstas globalios pirmumo sekos idėja po-uždaviniams, kuri paslepia duomenų struktūras ir apkrovos balansavimo priemones. Karkasas stengiasi minimizuoti darbą, reikalingą vartotojui įgyvendinti bendrą algoritmų funkcionalumą, ir leidžia koncentruotis ties konkrečiu uždaviniu. Bob++ pateikia visas klases ir metodus reikalingus įgyvendinti algoritmams ir visi jie yra laisvai matomi vartotojui. Norint įgyvendinti programą, pakanka perrašyti tik 4 metodus, kurie yra priklausomi nuo konkretaus uždavinio. Bob++ biblioteka irgi nebuvo ištestuota. Biblioteka jau yra ganėtinai senoka, paskutiniai straipsniai išėjo 2006 metais. Pagrindinė bėda ją bandant panaudoti buvo tai, kad bibliotekos internetiniame portale neveikė didžioji dalis nuorodų, tarp jų ir nuorodą į `git` repozitorija, taigi nepavyko gauti pilno bibliotekos kodo. Buvo rasti tik padriki kodo failai, be jokių kompiliavimo instrukcijų ar dokumentacijos.

Mallba[11] projektas stengėsi sukurti bibliotekos griaučius kombinatoriniams optimizacijos uždaviniams spręsti, kurie galėtų susitvarkyti su uždavinio lygiagretinimu vartotojui paprastu būdu, kuris taip pat būtų efektyvus. Pagrindinės Mallba savybės yra 1) vientisumas pateikiamų griaučių, nes visi buvo kurti tais pačiais principais; 2) gebėjimas keisti algoritmų vykdymą tarp lygiagretaus ir nuoseklaus. Vartotojams pakanka įgyvendinti nuoseklų algoritmą, kad galima būtų vykdyti jį lygiagrečiai; 3) tinkamumas vartoti su plačiu pasirinkimu skaičiavimo mašinų; 4) lanksti ir lengvai praplečiama sistemos architektūrą. Naujus griaučius nesunkiai galima pridėti, o senus praplėsti. Mallba griaučiai yra sukurti, remiantis interesų tarp konkrečios sprendžiamos problemos ir bendrų metodų jai išspręsti atskyrimo. Griaučius galima įsivaizduoti kaip bendrus šablonus, kuriuos reikia užpildyti problemos specifiką, norint ją išspręsti. Nors vartotojas ir turi patiekti savybes reikalingas uždaviniui išspręsti informacija kaip reikia lygiagretinti uždavinį yra paslėptą griaučiuose, taigi vartotojui nereikia vargintis su lygiagretinimo klausimais. Ši biblioteka buvo neištirta, nes jos nepavyko sukompiliuoti. Biblioteka yra ganėtinai sena, paskutiniai atnaujinimais jai buvo 2006 metais, tačiau pagrindas yra ir senesnis. Kompiliuojat kilo namažai klaidų, tačiau net ir jas sutvarkius biblioteka tinkamai nepasileido.

[10] `http://www.prism.uvsq.fr/~blec/bobpp/`

[11] `http://neo.lcc.uma.es/mallba/easy-mallba/index.html`

Zram[12] yra biblioteka skirta lygiagretiems paieškos algoritmams. Bibliotekos architektūra yra sluoksniuota, susidedanti iš 4 sluoksnių. *Techninės įrangos* sluoksnis paslepia visas priklausomybes nuo mašinos architektūros. Norint naudotis Zram reikia, kad mašina palaikytų bendravimą tarp procesorių žinučių siuntimu. Bendra atmintis tarp procesorių nėra reikalinga. Biblioteka nenaudoja jokios informacijos apie sistemos topologiją. *Servisų modulio* sluoksnis atsakingas už aukštesnio lygio funkcijas, jau susijusiais su bendrais paieškos algoritmų poreikiais tokiais, kaip baigimo aptikimas, dinaminis apkrovos paskirstymas ir grįžimo punktų sudarymas. *Paieškos variklių* sluoksnyje laikomi paieškos algoritmai tokie, kaip šakų ir rėžių, atbulinės paieškos algoritmai ir duomenų struktūros reikalingos jų įgyvendinimui. Ir paskutinis, sluoksnis atsakingas už konkrečių algoritmų įgyvendinimą. Su šiuo sluoksniu ir tenka vartotojui daugiausia dirbti. Paprastai aplikacijos sluoksnyje nėra jokio išreikšto lygiagretinimo. Šiai bibliotekai taip pat nebuvo atlikti tolesni tyrimai. Nors biblioteka nėra nauja, ją pavyko paleisti ir sukompiliuoti. Tarp pavyzdžių buvo rastas bendras šakų ir rėžių algoritmo įgyvendinimas, nesprendžiantis konkretaus uždavinio. Pasinaudojus šiuo pavyzdžiu pavyko įgyvendinti keliaujančio pirklio uždavinį. Tačiau leidžiant uždavinį ant PST pastebėta, kad praktiškai visą darbą atlieka vienas procesas. Skirtingos konfigūracijos mazgų skaičiui to nepakeitė, ir iš algoritmo pusės irgi nebuvo aišku, kodėl lygiagretinimas nevyksta tinkamai. Dėl šios priežasties negalima buvo toliau tirti bibliotekos.

SYMPHONY[13] yra biblioteka skirta *branch and cut* tipo uždaviniams spręsti. Biblioteka siekia supaprastinti tokių algoritmų įgyvendinimą, pateikiant bendruosius algoritmo metodus ir vartotojui paliekant tik uždavinio specifikos įgyvendinimą. SYMPHONY buvo sukurta su dviem pagrindiniais tikslais, tai yra efektyvumas ir naudojimo lengvumas. Biblioteka siekia efektyvumo iš nepriekaištingu bendro algoritmo įgyvendinimu, o naudojimo lengvumo iš apgalvotos ir vientisos vartotojo sąsajos. Biblioteka pilnai įgyvendinta `C` programavimo kalba, kurioje nėra klasių tai vartotojams tenka pateikti trūkstamus metodus su uždavinio specifika. Šia biblioteka pavyko sukompiliuoti ir rasti tinkamą pavyzdį šakų ir rėžių algoritmui. Pagrindinis ir svarbus šios bibliotekos apribojimas yra tas, kad ji skirta tik bendros atminties sistemoms, taigi, masinio lygiagretinimo testuoti šiai bibliotekai negalima. Nagrinėti kaip šią biblioteką galima būtų panaudoti sukurti hierarchinius lygiagretinimo algoritmus buvo sunku. Biblioteka labai didelė, galinti spręsti pačius įvairiausiu uždavinius. Dėl to labai sunku suprasti jos struktūrą ar praplėtimo taškus. Kadangi nebuvo rasta, kaip šios bibliotekos lygiagretinimą galima būtų praplėsti ir paskirstytosios atminties sistemomis, biblioteka buvo naudojama tolesniam darbui.

Apžvelgus visas bibliotekas matyti, kad nei viena iš jų negali atlikti hierarchinio lygiagretinimo. Hierarchinis lygiagretinimas galėtų efektyviau panaudoti turimus skaičiavimų resursus. Verta patyrinėti hierarchinio lygiagretinimo galimybes labiau. Dėl to šio darbo metu buvo sukurtas hierarchinis lygiagretusis šakų ir rėžių algoritmas ir buvo pradėtas tirti jo lygiagretinimo efektyvumas.

[12] `http://www.cs.unb.ca/~bremner/software/zram/`
[13] `https://projects.coin-or.org/SYMPHONY`

2. Teorinė dalis

Šioje dalyje apžvelgiamos temos tiesiogiai susijusios su atliktu darbu tyrimo metu. Pirma, bus pristatyti kitų autorių įgyvendinti hierarchiniai šakų ir rėžių algoritmai. Toliau bus pristatyti darbo metu įgyvendinti uždaviniai: keliaujančio pirklio, kuprinės ir kvadratinio priskyrimo uždavinius, su kuriais buvo įgyvendinti šakų ir rėžių algoritmai. Pabaigoje bus trumpai aptarti šakų ir rėžių algoritmai, bei kokie apatinių rėžių skaičiavimo metodai buvo panaudoti įgyvendintuose uždaviniuose.

2.1. Hierarchiniai lygiagretieji šakų ir rėžių algoritmai

Kadangi hierarchiniai lygiagretieji šakų ir rėžių algoritmai skirti tinkamai išnaudoti heterogenines skaičiavimų mašinas, bendru atveju hierarchinis lygiagretusis šakų ir rėžių algoritmas priklauso nuo skaičiavimo mašinos architektūros ir nuo konkretaus uždavinio. Mašinos architektūra nulemia kiekvieno procesoriaus taktinį dažnį, jų skaičių mazge, bendravimo efektyvumą mazge, prieinamą atmintį, mazgų skaičių, mazgų sujungimo topologiją ir bendravimo efektyvumą tarp mazgų. Konkretus šakų ir rėžių optimizavimo uždavinys gali turėti skirtingas skaičiavimo charakteristikas: rėžių skaičiavimo ir medžių šakojimosi operacijos gali būti reiklios skaičiavimams arba paprastos, šakojimasis gali generuoti daug apylygiai gerų po-uždavinių (arba mažai), viršutinis rėžis gali būti atnaujinamas dažnai (ar retai). Tai nulemia reikalingą po-uždavinių grūdėtumą, kad procesoriai nepraleistų per daug laiko atlikdami vieną operaciją arba negaištų laiko laukdami naujo po-uždavinio. Efektyvus bendravimas tarp mazgų, norint perduoti viršutinį rėžį ar pasidalinti uždaviniais, taip pat yra svarbus dalykas, nes būtent tai paprastai yra lėčiausia grandis. Prieinamas atminties kiekis mazge gali apriboti galimybes naudoti „pirma geriausias“ medžio generavimo strategiją esant daug po-uždavinių. Šie dalykai yra svarbūs, kuriant lygiagrečiuosius ir ypač hierarchinius lygiagrečiuosius šakų ir rėžių algoritmus, nes hierarchiniai algoritmai labiau priklauso nuo kompiuterio architektūros.

Esamus lygiagrečiųjų algoritmų įgyvendinimus galima padalinti į dvi grupes: šeimininko-darbininko ir decentralizuoti algoritmus. Šeimininko-darbininko atveju egzistuoja šeimininkas ar šeimininkų hierarchija, kuri atsakinga už darbų paskirstymą darbininkams. Visas bendravimas, darbo paprašymas ir naujo viršutinio rėžio paskleidimas, vyksta perduodant informaciją šeimininkams, kurie ją perduodą savo darbininkams (ir kitiems šeimininkams). Naudojant tik vieną šeimininką, sukelia problemų lygiagretinant algoritmus masiškai lygiagrečiose sistemose, nes vienas šeimininkas negali spėti paskirstyti darbus visiem darbininkams. Todėl masiškai lygiagrečiuose sistemose, kurios paprastai būna hierarchinės, tinka naudoti šeimininkų hierarchija (paprastai nestambesnė nei skaičiavimo mašinos mazgų hierarchija), kur pagrindinis šeimininkas duoda uždavinius žemesnio rango šeimininkams, kurie paprastai tą uždavinį suskaido į smulkesnius po-uždavinius ir siunčia juos darbininkams ar dar žemesnio rango šeimininkams. Šiuo būdu yra išvengiamas šeimininko(-ų) apkrovimas, nes darbas yra pasidalinamas. Šio metodo trūkumas yra, kad dalis procesorių (šeimininkai) praleidžia savo laiką atlikdami administracinius uždavinius, o ne spręsdami optimizavimo uždavinį. Decentralizuoto algoritmo atveju, nėra išskirto procesoriaus atliekančio

šeimininko darbą. Kiekvienas mazgas palaiko savo darbų sąrašą ir gali juo dalintis su kitais mazgais. Dalijimasis darbais mazgo viduje paprastai vyksta per bendrą atmintį ir yra efektyvus. Šio metodo privalumas yra tas, kad kiekvienas procesorius gali skirti savo laiką spręsti optimizacijos uždaviniui. Tačiau nesant koordinuoto bendravimo tai gali išaugti į trūkumą, nes prastai organizuotas bendravimas gali tapti labai neefektyvus ypač daug bendraujant tarp mazgų, kur bendravimas yra labai lėtas. Decentralizuotas metodas nepateikia atsakymo, kaip vyksta bendravimas tarp mazgų, t.y. kada ir kas kreipiasi į kitus mazgus, norint paprašyti ar nusiųsti darbo, kaip išsirinkti mazgus, į kuriuos kreiptis, ir kas turi atsakyti į tokias užklausas. Norint algoritmui veikti sklandžiai, šie klausimai turi būti apgalvoti ir ištirti.

Toliau aptarti keli įgyvendinti lygiagretieji šakų ir rėžių algoritmai. Aptarinėjami algoritmai turi vieną trūkumą. Jie visi sukurti autorių reikmėms, dėl to algoritmai yra pritaikyti konkrečiam uždaviniui ir įgyvendinimas nėra prieinamas viešai. Tai apsunkina lygiagrečiųjų šakų ir rėžių algoritmų tyrimus, nes rezultatus sunku tiksliai atkartoti.

2.1.1. Šeimininko-darbininko algoritmų įgyvendinimai

Visi hierarchiniai lygiagretieji BB algoritmai yra įgyvendinti šeimininko-darbininko pagrindu. Pagrindiniai darbai atlikti tiriant šios algoritmus yra [BMT12a], [BMT12b], [AFO06b], [ANF03], [HCP+13b] ir [HCH+13b]. Toliau bus detaliau aptarti šie darbai.

2.1.1.1. Aida et al.

Pirmieji darbai aptarti šiame skyriuje bus autoriaus Aida et al.. Kadangi jo du darbai yra tęsinys vienas kito, bus aptartas tik naujausias [AFO06a] darbas.

Autorių pasiūlytas šakų ir rėžių algoritmas yra išlygiagretintas, pasinaudojus hierarchine šeimininko-darbininko paradigma, siekiant išvengti našumo mažėjimo augant kompiuterių tinklams naudojantis paprastu šeimininko-darbininko paradigmos įgyvendinimu. Darbe vienas šeimininkas valdo kelis procesų rinkinius, kurie susideda iš vieno žemesnio rango šeimininko ir kelių darbininkų. Darbų pasidalijimas vyksta dviem būdais: dalijimasis iš šeimininko žemesnio rango šeimininkams ir iš žemesnio rango šeimininkų darbininkams. Skaičiavimų rezultatai surenkami atbuline tvarka. Autoriai įvardina du privalumus, jų sukurto hierarchinio šeimininko-darbininko paradigmos įgyvendinimo prieš paprastą šeimininko-darbininko paradigmos įgyvendinamą. Pirmas privalumas yra sumažinti bendravimo valdymui reikalingi resursai sugrupuojant šeimininkus ir su jais bendraujančius darbininkus pagal jų susiejimą techninės įrangos lygmenyje. Antras privalumas yra išvengti našumo sumažėjimo, atsirandančio dėl vieno šeimininko ribotų galimybių apdoroti visas ateinančias užklausas iš didelio skaičiaus darbininkų. Autorių hierarchinis BB algoritmas atlieka skaičiavimus tokiu būdu. Žemesnio rango šeimininko ir darbininkų rinkinys dirba prie vieno po-medžio iš sugeneruoto paieškos medžio. Žemesnio rango šeimininkas siunčia po-uždavinius darbininkams ir gauna skaičiavimo rezultatus iš jų. Šeimininkas atlieka darbų paskirstymą žemesnio rango šeimininkams ir skelbia geriausią žinomą viršutinį rėžį žemesnio rango šeimininkams jį persiųsdamas. Geriausio viršutinio rėžio skelbimas yra svarbus, norint paspartinti uždavinio sprendimą, nes skatina po-medžių eliminavimą. Kiekvienas žemesnio rango šeimininkas seka ne-

įvertintus po-uždavinius iš sprendžiamo po-medžio eilėje ir geriausią viršutinį rėžį, kurį siunčia savo darbininkams, o jam atsinaujinus ir šeimininkui. Darbininkai gavę uždavinį iš šeimininkų atlieka šakojimosi procedūrą, įvertina viršutinius ir apatinius rėžius ir eliminuoja po-medžius, kurie atmetami dėl apatinio rėžio testo. Galiausiai darbininkas grąžina darbo rezultatus šeimininkui. Darbo rezultatai susideda iš suskaičiuotų viršutinių ir apatinių rėžių, ir sugeneruotų po-uždavinių. Šeimininko procesas reguliariai siunčia užklausas žemesnio rango šeimininkams sužinoti apie jų darbo statusą. Darbo statuso informaciją sudaro skaičius, neįvertintų po-uždavinių, ir viršutinis rėžis. Kai neįvertintų po-uždavinių skaičius nėra pakankamai subalansuotas tarp žemesnio rango šeimininkų, šeimininko procesas perkelia po-uždavinius iš daugiau jų turinčių šeimininkų į mažiau. Kai šeimininko procesas aptinka geresnį nei turimą viršutinį rėžį, jis jį atnaujina savo procese ir skelbia atnaujintą rėžį kitiems žemesnio rango šeimininkams. Galiausiai šeimininko procesas stabdo uždavinio sprendimą, kai pasiekiamos atitinkamos sąlygos.

Autorių darbe yra įgyvendinta adaptyvi uždavinių grūdėtumo kontrolė. Pasiūlytame algoritme uždavinio grūdėtumas tiesiogiai priklauso nuo esamo po-medžio gylio. Skaičiavimo našumas kiekvienoje žemesnio rango šeimininkų ir darbininkų grupėje priklauso nuo bendravimui tenkančia darbo dalimi ir turėjimo naujausią viršutinį rėžį. Iš vienos pusės, mažinant bendravimą tarp procesų daugiau laiko bus praleista sprendžiant uždavinį, iš kitos pusės, retas viršutinio rėžio atnaujinimas gali nulemti mažiau eliminuotų po-medžių ir taip daugiau darbo. Jei darbininkas randa geresnį viršutinį rėžį, greitas jo paskelbimas šeimininkui ir taip kitiems darbininkams padėtų jiems efektyviau atlikti po-medžių eliminavimą. Reikia siekti kompromiso tarp bendravimo mažinimo ir viršutinio rėžio skelbimo dažnio. Norint sumažinti bendravimą riekia, kad užduočių grūdėtumas būtų didelis, o siekant dažno viršutinio rėžio dalijimosi, reikia mažo užduočių grūdėtumo. Pasiūlytas autorių algoritmas automatiškai reguliuoja užduoties grūdėtumą siekiant sumažinti bendravimą tarp procesų ir efektyviai perdavinėti viršutinį rėžį. Algoritmo idėja yra sumažinti užduoties grūdėtumą, kad bendravimas tarp procesų vyktų dažniau, kai galima tikėtis didelio naujo geresnio viršutinio rėžio poveikio ar kai skirtumas tarp naujo ir dabartinio viršutinio rėžio yra didelis. Padidinti užduoties grūdėtumą, kad bendravimas tarp procesų vyktų rečiau, kai nesitikima didelio poveikio iš naujo viršutinio rėžio arba kai skirtumas tarp naujo ir esamo viršutinio rėžio yra mažas. Kai tik šeimininkas gauna naują viršutinį rėžį iš darbininko, jis yra palyginamas su esamu ir, jei yra geresnis, suskaičiuojamas skirtumas tarp jų. Jei skirtumas yra mažesnis už tam tikrą slenkstinį dydį, šeimininkas kitą kartą siųsdamas užduotį darbininkui padidina jos grūdėtumą ar nurodo darbininkui generuoti gilesnius po-medžius. Jei skirtumas yra didesnis, viskas daroma atvirkščiai. Autoriai siūlo slenkstį, θ, apsibrėžti kaip $\theta = a \times \Delta Z$, kur ΔZ yra skirtumas tarp viršutinių rėžiu kai algoritmas pirmą kartą juos atnaujina, nes tai paprastai būna didžiausiai skirtumas, o a $(0 \leq a \leq 1)$ yra laisvai pasirenkamas parametras.

Autoriai savo algoritmą testavo naudojant kompiuterių tinklą paskirstytą po keturis Japonijos miestus, sudarytą iš keturių kompiuterių klasterių. Rezultatams gauti buvo spendžiamas dvilinijinės matricos nelygybės tikrinių verčių uždavinys (*angl., Bilinear Matrix Inequality Eigenvalue Problem, BMI-EP*). BMI-EP yra žinomas kaip bendro pobūdžio uždavinys svarbus analizei bei kūrimui grįžtamojo ryšio sistemoms, naudojamoms daugelyje pramonės taikymo sričių, tokių kaip

roboto rankų valdymui ar sraigtasparnio padėties valdymui. Autoriai testavo algoritmą sprendžiant skirtingo dydžio uždavinius, taip pat, jas sprendžiant neišlygiagretinus bei išlygiagretinus naudojant tik vieną kompiuterių klasterį panaudojus standartinę šeimininko-darbininko paradigmą. Didžiausias eksperimentas buvo leidžiamas naudojant 412 procesorių branduolių, leidžiant tik vieną kompiuterio klasterį buvo naudojami 73 branduoliai. Autoriai gavo spartinimus 28-51 ir 88-126 karto sprendžiant uždavinius su vienu kompiuterių klasteriu ir visais keturiais atitinkamai lyginant su nuosekliuoju algoritmu. Atsižvelgiant į procesorių skaičių, pasiektas tik 30.6% spartinimo našumas naudojant visus keturis kompiuterių klasterius, o naudojant tik vieną kompiuterio klasterį su standartine šeimininko-darbininko paradigma spartinimo našumas buvo 69.9%. Autoriai parodo, kad nemaža dalis algoritmo veikimo užtrunka algoritmo užbaigimui (apie 30% viso algoritmo veikimo laiko) nors rezultatai tuo metu jau yra prieinami. Tiriant darbų paskirstymo efektyvumą buvo naudojama darbų paskirstymo procedūra, kuri stengėsi skirti mažesnio rango šeimininkams darbus proporcingai pagal matuojamą jų klasterių efektyvumą. Taip pat, radus laisvą mažnesnio rango šeimininką, šeimininko procesas paimdavo nespręstus uždavinius iš kito žemesnio rango šeimininko ir perskirstydavo juos. Laisvas šeimininkas yra laikomas tada, kai jo darbų eilėje nėra nė vieno uždavinio. Nors laisvo laukimo laikai nėra didesni nei 8% nuo bendro uždavinio sprendimo laiko, jie nėra tolygūs ir gali skirtis tarp kompiuterių klasterių 2 kartus. Automatinis užduočių grūdėtumo parinkimas buvo patikrintas palyginus rankiniu būdu pasirinktą grūdėtumą su automatiniu algoritmu. Rankiniu būdu parinkimas vyko sprendžiant tą patį uždavinį kelis kartus su skirtingais grūdėtumo nustatymais ir pabaigoje išrinkus geriausią. Automatinio grūdėtumo nustatymo algoritmui buvo empiriškai parinktas parametras $a = 0{,}5$, autorius šį skaičių nurodo kaip reikalaujančio daugiau tyrimų. Šio eksperimento rezultatai parodė, kad automatinis grūdėtumo parinkimo algoritmas pasirodo neprasčiau negu rankinis grūdėtumo nustatymas. Kaip tolesnius darbus autoriai nurodo darbų paskirstymo algoritmo detalesnį testavimą ir skirtingus jo variantus, nes autoriams buvo sunku gauti pakartotinus darbo paskirstymo rezultatus, taip pat detalesnis tyrimas dėl laisvo parametro a parinkimo yra reikalingas norint pagerintį automatinį grūdėtumo parinkimą.

2.1.1.2. Bendjoudi et al.

Sekantis aptartas darbas yra autorių Bendjoudi et al. darbas. Autoriai yra atlikę du darbus ([BMT12a], [BMT12b]) apie hierarchinius lygiagrečiuosius šakų ir rėžių algoritmus, vėlgi šie darbai yra tęsinys vienas kito. Šiame skyriuje bus aptartas tik naujesnis [BMT12b] darbas.

Autoriai siūlo adaptyvų hierarchinį šeimininko-darbininko (*angl., adaptive hierarchical master–worker, AHMW*) modelį. Šiuo modeliu sukurtas algoritmas galintis prisitaikyti prie kintančių kompiuterio tinklo resursų ir atsigauti po atskirų komponentų sutrikimų. Kaip ir Aida et al. darbuose autoriai, taip pat, išskiria tris skirtingas roles savo hierarchiniame lygiagrečiajame šakų ir rėžių algoritme ir tris sąryšius. Galimi sąryšiai yra: tėvo - kiekvieno proceso tėvas yra procesas, kuris jį sukūrė ir jam vadovauja, vaiko - proceso vaikai yra procesai, kuriuos jis sukūrė ir jiems vadovauja, kolegos - tai yra visi kaimyniniai procesai išskyrus tėvą ir kitų tėvų vaikus. Autorių išskirtos rolės vadinamos super-šeimininkas, šeimininkas, darbininkas. Super-šeimininkas yra tik vienas visoje sistemoje. Jo darbai yra įsisavinti naujos kompiuterinius resursus patekusius

į kompiuterių tinklą, suskaidyti pradinę užduotį ir gautus po-uždavinius padalinti jo valdomiems darbininkais arba šeimininkais, surinkti darbų rezultatus, perskirstyti sutrikusių darbininkų darbą ir pradėti konstruoti visą sistemos hierarchiją. Šeimininko procesas yra tarpiniai procesas tarp super-šeimininko ir darbininkų. Skirtingai nei įprastuose šeimininko-darbininko algoritmuose, autoriai nesiūlo jokių apribojimų šiai rolei. Ši rolė gali atlikti visus darbus kaip ir super-šeimininkas išskyrus įsisavinti naujus kompiuterinius resursus. Papildomai šiai rolei reikia gauti užduotis iš proceso tėvo, kad galėtų dalyvauti užduoties vykdyme, taip pat šeimininko procesas persiuntinėja ateinančias ir išeinančias žinutes tarp per jį susietų procesų ir aptinka sutrikusios procesus tarp savo vaikų. Darbininko procesas taip pat skiriasi nuo standartinio šeimininko-darbininko paradigmos. Be užduoties skaičiavimų darbininko procesas dar gali dalyvauti kurdamas sistemos hierarchiją ir pereiti prie užduočių skaidymo besiruošdamas tapti šeimininkų, kai atsiranda jam valdomų kompiuterinių resursų.

AMHW yra daugiasluoksnis hierarchinis šeimininko-darbininko tipo algoritmas sudarytas iš daugelio šeimininko-darbininko posistemių. Kiekviena šeimininko-darbininko posistemė veikia kaip standartinis šeimininko-darbininko algoritmas, kur vienas šeimininkas valdo kelis darbininkus. Kiekviena posistemė yra įgyvendinta pasinaudojus *P2PBB* karkasu ([BMT08]), kur kiekvienas šeimininkas valdo kintančią ir bendraujančią darbininkų grupę. Dėl to hierarchija generuojama AMHW algoritmo yra daugiasluoksnė, dinaminė ir besikeičianti laikui bėgant priklausomai nuo naujų skaičiavimo resursų atsiradimo. Šiame modelyje vienas šeimininkas gali turėti ir šeimininkus ir darbininkus kaip savo vaikus. Lankstumas skirtingų procesų valdyme yra viena pagrindinių AMHW savybių. Mazgo rolė hierarchijoje nėra iš anksto nuspręsta, bet įgyjama hierarchiškai. Pradžioje visi naujai sukurti mazgai būna darbininkais, bet tampa šeimininkais vos tik gauna procesų valdyti. Šeimininkas, taip pat, gali virsti darbininku, kai nebeturi procesų, kuriuos valdo. Koleginiai procesai vienas kitą visada laiko paprastais darbininkais. Hierarchinės sistemos plotis ir gylis yra dinaminiai ir priklauso nuo įgytų kompiuterinių mazgų skaičiaus ir grupių formuojančių šeimininko-darbininko posistemes dydžio. Hierarchijos plotis nusako kiek užduočių gali būti vykdomos lygiagrečiai, o gylis nusako užduočių grūdėtumą. Kuo platesnė ir gilesnė sistemos hierarchija, tuo daugiau užduočių galima vykdyti vienu metu ir, į tuo smulkesnes po-užduotis jos bus skaidomos. Ideali konfigūracija ir būtų plati ir gili sistema, tačiau reikia atsižvelgti į šeimininko pajėgumą priimti naujus darbininkus ir leistinos darbininkui perduoti užduoties grūdėtumą. Pirmu atveju šeimininkai tampa per daug apkrauti darbo koordinuodami darbininkus ir gali nespėti efektyviai atlikti savo pareigų. Antru atveju darbininkai gali pradėti spręsti daug smulkių, greitai sprendžiamų problemų ir šeimininkams teks dažnai perdavinėti darbininkams naujus uždavinius, kas gali perkrauti šeimininkus. Pačioje sprendimo pradžioje super-šeimininkas skaido pradinį uždavinį į daugelį po-uždavinių ir perduoda juos šeimininkams, kurie savo ruožtu skaido juos į dar mažesnius po-uždavinius ir išsaugo savo darbų sąraše. Skirtingai nei standartiniuose šeimininko-darbininko algoritmuose užduočių skaidymas, taip pat, vyksta paskirstytai. Tai turi kelis privalumus: 1) super-šeimininkas nėra apkrautas ir turi daugiau laiko užsiimti kitais darbais, tokiais kaip derinti hierarchinę struktūrą, surinkti darbų rezultatus, vesti norimas statistikas ir kitką, 2) paskirstytas užduočių skaidymas sumažina laukimo laiką, nes jos greičiau suskaidomos, 3)

smulkiagrūdžios užduotys yra greičiau pasiekiamos, nei su standartiniais šeimininko-darbininko algoritmais, tai leidžia greičiau įtraukti visus darbininkus į uždavinio sprendimą ir sumažiną galimybę įvykiui, kai darbininkai neturi pakankamai smulkių užduočių. Kiekvienas šeimininkas turi savo darbų sąrašą, kuriame yra dar neįvertinti darbai. Šis sąrašas sudaromas, kai yra išskaidomas pirmas uždavinys gautas iš tėvo. Kai šeimininkas gauna užklausa darbui iš vieno iš savo vaikų jis pasirenką uždavinį iš savo darbų sąrašo ir persiunčia jį darbininkui. Jei šeimininkas neturi darbų, darbininkas yra priverstas laukti. Procesai toje pačioje šeimininko-darbininko posistemėje bendrauja be jokių tarpininkų, t.y., tiek šeimininkas tiek visi darbininkai mato vienas kitą. Darbininkai iš atskirų šeimininko-darbininko posistemių nemato vienas kito ir turi bendrauti per kitus procesus. Taigi bendravimas gali būti horizontalus (tarp kolegų) ir vertikalus (su tėvu arba vaikais). Subalansuotoje hierarchijoje perduoti informacija iš vieno darbininko, bet kuriam kitam reikia padaryti ne daugiau nei $2 \times \log_k(n) - 1$ šuolių, kur k yra kiekvienos grupės dydis, o n yra sukonstruoto super-šeimininkų – šeimininkų – darbininkų medžio gylis. Turint tokią hierarchinę struktūrą darbo pabaigos aptikimas nėra trivialus dalykas. Super-šeimininkas nemato visų procesų, todėl negali žinoti viso uždavinio statuso, o atskiri procesai turi tik lokalų ir savo tiesioginių vaikų vaizdą. Taigi darbo užbaigimo aptikimą turi atlikti kiekvienas šeimininkas atskirai ir propaguoti jį savo tėvui. Kai visi proceso vaikai neturi darbo ir proceso darbų sąrašas yra tuščias, jis skelbia apie savo užbaigimą tėvui. Kai visi super-šeimininko vaikai skelbia darbų užbaigimą, uždavinys laikomas išspręstu.

Algoritmo testavime autoriai naudojo cecho srauto uždavinį (*angl., Flow-Shop problem, FSP*). Uždavinys paprastai formuluojamas kaip n darbų planavimas ant m mašinų, taip kad mašina vienu metu gali dirbti tik prie vieno darbo ir visi darbai turi pereiti visas mašinas tą pačia tvarka. Minimizuoti siekiama tikslo funkcija yra darbų atlikimo laikas. Testai buvo atlikti naudojant Grid'5000[14] kompiuterių klasterį, kuriame buvo panaudota apie 2000 procesorių. AHMW našumas yra lyginamas su vieno sluoksnio hierarchiniu šeimininko-darbininko algoritmu (1-HMW) ir paprastu šeimininko-darbininko algoritmu (MW). Pirmieji jų rezultatai rodo, kad autorių pasiūlytas AHMW algoritmas startuoja daug greičiau nei likusieji. Startuoti visiems 1500 testuojamiems mazgams prireikė 70 sekundžių naudojant AHMW (grupės dydis lygus 10), 900 sekundžių - 1-HMW ir 1700 sekundžių su MW algoritmu. Autoriai pateikia šeimininkų apkrovimą 10 minučių intervalais, iš kurių matyti, kad AHMW atveju vidutinis apkrovimas yra $\sim$10% kai 1-HMW ir MW yra $\sim$20% ir $\sim$90% atitinkamai. Tiriant algoritmo našumą buvo naudojami Taillard'o uždaviniai [Tai93]. Viename iš testų buvo leidžiami keli uždaviniai ir žiūrimą, kuris algoritmas per 10 minučių gaus geriausią rezultatą, nugalėtojas visais 10 atveju buvo autorių sukurtas AHMW algoritmas. Tai aiškinama tuo, kad AHMW atlieka paskirstytą medžio šakojimą leidžiantį greičiau pasiekti smulkesnius použdavinius, kuriuos gali ištirti ir įvertinti darbininkai. Kitų algoritmų atveju reikia daugiau laiko, kol užduotys pasidarys pakankamai smulkios, taigi darbuotojai praleidžia daugiau laiko laukdami. Sekantis autorių bandymas buvo įvertinimas dalies laiko, kurį procesai praleidžia dirbdami ir kiek laiko praleidžia tuščiai. Dalį laiko, kurį procesai praleidžia dirbdami autoriai apibrėžė, kaip našumą ir prilygino vykdymo laiko santykiu su vykdymo ir tuščiai praleisto laiko suma. Tuščiai

[14] `http://www.grid5000.fr`

praleistas laikas yra laikas, kurį procesai skyrė bendravimui ir meistro užduočių skaidymo laiką. Eksperimentui buvo sprendžiami 10 Taillard'o uždavinių. Gauti rezultatai rodo, kad AHMW efektyvumas siekia 99.02%, kai vidutinis 1-HMW algoritmo efektyvumas siekia 92.63%, o paprasto MW - 83.43%. Tai rodo, kad AHMW algoritmo darbininkai praleidžia 99% savo laiko spręsdami užduotį ir tik 1% bendraudami ir laukdami užduočių. Pažiūrėjus į tai iš šeimininkų pusės tik 1% jų laiko buvo praleistas užduočiai skaidyti ir perdavinėti darbininkams. Ateityje autoriai nori pagerinti procesų sutrikimų toleravimą ir ištestuoti savo algoritmą su didesniais ir plačiau naudojamis kompiuterinių tinklais, kuriuose būna ypač dažni mazgų sutrikimai.

2.1.1.3. Herrera et al.

Kadangi abu Herrera et al. darbai irgi yra tęstiniai, šiame skyriuje aptartas bus tiktai naujesnis [HCH+13b] darbas.

Autorių siūlomas hierarchinis lygiagretusis šakų ir rėžių algoritmas neturi tokios griežtos struktūros kaip prieš tai aptarti algoritmai. Bendravimas tarp mazgų vyksta naudojantis *MPI*. Kiekviename mazge yra startuojamas vienas *MPI* procesas, kuris esant poreikiui ir laisviems procesams mazge sukuria naują giją, naudojant *Pthreads*, su kuria pasidalina darbu. Esant pakankamai mazgų yra, kuriama *MPI* procesų hierarchiją. Kiekvienas *MPI* procesas darbininkas informuoja savo šeimininką, kai viena iš gijų pasibaigia. Tai turi užtikrinti, kad tarp bendravimo padaromas atitinkamas darbo kiekis. Darbo paskirstymas yra pradedamas proceso, kuriam ištuštėjo darbo sąrašas, siunčiant žinutę savo šeimininkui, kurio prašoma darbo. Tada šeimininkas prašo daugiausiai darbų turinčio proceso (donoro) persiųsti dalį darbų prašiusiajam procesui. Šiuo mechanizmu darbo migravimas yra globalus. Donoro procesas išsirenka labiausiai apkrautą savo giją, sustabdo jos veiklą ir siunčia visą gijos informaciją. Proceso apkrova vertinama kaip uždavinių skaičius laukiantis būti įvertintas. Algoritmas startuoja su vienu *MPI* procesu mazge, kiekvienas *MPI* procesas skaido pradinį uždavinį ir pasirenka po-uždavinį tolimesnei eigai. Patys *MPI* procesai užduočių sprendime tiesiogiai nedalyvauja. Jų paskirtis yra paskirstyti darbą ir sukurti gijas darbui atlikti bei surinkti darbo rezultatus. *MPI* procesai paprastai sukuria tiek gijų, kiek yra uždavinių ir laisvų procesų gijoms kurti. Geriausią viršutinį rėžį, taip pat, perdavinėja *MPI* procesai ir taip daro iš karto kai, kuri nors gija raportuoja geresnį nei žinomą viršutinį rėžį. *MPI* naudojamas bendravimui tarp mazgų, mazgo darbas vyksta kuriant naujas gijas su *Pthreads* ir bendraujant per bendrą atmintį. Naujai sukurtos gijos gauną pusę darbo sąrašo vienetų iš savo tėvinio proceso. Radus naują geresnį viršutinį rėžį jis yra atnaujinamas bendroje atmintyje, toks atnaujinimas iš karto matomas visoms gijoms mazge ir nereikalauja žinučių siuntinėjimo. Skaičius gijų, kurias galima sukurti atitinką branduolių skaičių mazge. Toks gijų modelis leidžia algoritmui naudoti kompiuterinius resursus pagal jų poreikį. Kiekviena gija valdo savo darbų sąrašą. Skirstant darbus tarp gijų yra atsižvelgiama į darbų prioritetus, šitaip stengiamasi išlaikyti balansą tarp gijų.

Testuojant algoritmą buvo naudojami uždaviniai iš [PŽ06] darbo. Pirmieji autorių testai buvo skirti įvertinti konstruojamo BB medžio šakų pasirinkimo taisyklę. Buvo testuojamos pirma gylis ir pirma geriausias iš giliausiu (hibridinė strategija). Hibridinė strategiją sukonstruoja medį, turintį daug daugiau saugomų elementų nei pirma gylis strategija ($10^4 - 10^5$ karto daugiau pagal

eksperimento rezultatus). Toks didelis saugomų elementų skaičius sulėtina skaičiavimą, hibridinei strategijai prireikė vidutiniškai 40% daugiau laiko išspręsti uždavinius. Išspręstų po-uždavinių skaičius abiem atvejais vienodas. Autoriai pastebėjo, kad pirma gylis strategijos atveju lygiagretinimo našumas geriausia išsilaiko nei hibridinės strategijos atveju. Pirma gylis strategija pasiekia beveik tiesinį spartinimą. Šis rezultatas taip pat siejamas su naudojamu mažesniu atminties kiekiu, nes išspręstas po-uždavinių skaičius nepriklausė nuo pasirinktos strategijos. Autoriai taip pat įvertino procesų santykinį apkrovimą disbalansą, kurį apibrėžė kaip

$$RLI = 1 - \frac{W_{tot}}{pW_{max}}$$

, kur p yra *MPI* procesų skaičius, W_{tot} - kiekis skaičiavimų atliktų visų procesų ir W_{max} - kiekis skaičiavimų atliktų daugiausiai dirbusio proceso. RLI vertės arti nulio rodo gerai subalansuotą procesą. Eksperimentai su 2, 4 ir 8 *MPI* procesais rodo, kad RLI neturi nuspėjamo elgesio ir stipriai priklauso nuo pradinio užduoties paskirstymo tarp procesų. Sekantis eksperimentas buvo skirtas įvertinti viršutinio rėžio transliavimo ir užduočių dalijimosi tarp *MPI* procesų naudą. Rezultatai rodo, kad RLI stipriai sumažėjo (mažiau nei 7.3%), nors lygiagretinimo našumas beveik nepakito. Tolimesniems darbams autoriai nori sukurti ir ištirti BB medžio šakos pasirinkimo taisyklę, kuri reikalautų mažai atminties saugoti medžio elementus ir turėtų mažas paieškos valdymo išlaidas.

2.2. Šakų ir rėžių algoritmas

Pereinant prie aptartų uždavinių sprendimo būdų galima kalbėti apie tikslius ir apytikslius metodus. Šiame darbe siekiama tikslių šių uždavinių sprendimų. Tam, darbe, naudojamas šakų ir rėžių algoritmas, kuris efektyviai perrenka galimus sprendinius. Šiame skyriuje bus plačiau aptarta bendra šakų ir rėžių algoritmų formą ir įgyvendinimas.

Šakų ir rėžių algoritmo sprendimo būdas susidaro iš netiesioginio sprendinių aibės perrinkimo patikrinant tik galimų sprendinių poaibius. Sprendiniai, kurie negali vesti prie galimo arba optimalaus sprendinio, yra atmetami. Sprendinių perrinkimas vyksta tiriant paieškos medį, kur kiekvienas mazgas yra poaibis sprendžiamo uždavinio sprendinių aibės. Medžio dydis, t.y. sugeneruotų mazgų skaičius, yra tiesiogiai susijęs su pasirinkta medžio generavimo strategija. Medžio generavimui svarbiausios dalys yra šakojimosi procedūra, genėjimas ir tolesnės šakos pasirinkimas. Šakojimusi vadinama procedūra, kuri iš gauto mazgo sugeneruoja bent vieną papildomą mazgą. Genėjimas yra procedūra, kurios metu yra patikrinama, ar turimas mazgas gali vesti prie optimalaus sprendinio ir jei negali, to mazgo pašalinimas. Mazgas tikrinamas prie kokio jis sprendinio gali nuvesti skaičiuojant mazgo apatinį rėžį, kuris nurodo geriausią sprendinį, kurį galima rasti po-medyje iš šio mazgo. Rėžių skaičiavimo algoritmai paprastai yra specifiniai kiekvienai sprendžiamų uždavinių grupei ir turi užtikrinti, kad pateiktas rėžis yra ne blogesnis nei geriausias sprendinys toje šakoje. Paskutinė medžio konstravimo procedūra – tolesnės šakos pasirinkimas, parenkama pagal poreikį, nes yra kelios populiarios strategijos. Populiariausios yra pirma-gylis, pirma-geriausias ar hibridinės. Šios trys procedūros apsprendžia medžio formą ir tuo būdu algoritmo efektyvumą. Šakų ir rėžių algoritmo pseudokodas yra aprašytas 1 algoritme.

Algoritmas 1 Šakų ir rėžių algoritmo pseudokodas.

```
mazgai = {uždavinys}
sprendinys = NULL
while mazgai IS NOT EMPTY do
    mazgas = pasirinkimas(mazgai)
    for all vaikas IN šakojimas(mazgas) do
        if apatinisRėžis(vaikas) WORSE THAN sprendinys then
            atmesti(vaikas)
        else if yraSprendinys(vaikas) and vaikas BETTER THAN sprendinys then
            sprendinys = vaikas
        else
            mazgai = mazgai ∪ {vaikas}
        end if
    end for
end while
```

Paprastą šakų ir rėžių algoritmo įgyvendinimą uždaviniui galima įsivaizduoti kaip medžio konstravimą, kur kiekviena šaka iš esamos būsenos yra atliktas pasirinkimas į kitas galimas būsenas. Rėžių skaičiavimas vyksta kiekviename mazge nustatant, kokį geriausią sprendinį galima rasti pratęsus kelionę šia kryptimi. Ši informacija vadinama apatiniu rėžiu. Rėžių skaičiavimas yra svarbus, nes tai leidžia nuspręsti ar apkarpyti medžio šakas sužinojus, kad jos neveda prie optimalaus sprendinio. Apkarpymas vyksta lyginant geriausią žinomą sprendinį visam uždaviniui, vadinama viršutiniu rėžiu, su apatiniu rėžiu. Jei apatinis rėžis yra prastesnis nei viršutinis, rėžis šaka yra atmetama. Atliekant tyrimus šiame darbe pradinis viršutinis rėžis buvo nustatomas euristiniais metodais. Algoritmo eigoje viršutinis rėžis yra atnaujinamas radus geresnį sprendinį. Uždavinio sprendinys tampa paskutiniu rastu viršutiniu rėžiu.

2.2.1. Darbo metu įgyvendinti uždaviniai

Šiame darbe buvo sprendžiami trys uždaviniai, kurie šiame skyriuje bus aptarti plačiau.

2.2.1.1. Keliaujančio pirklio uždavinys

Keliaujančio pirklio uždavinį (TSP) [ABC+07] pradėjo tyrinėti airių matematikas Sir William Rowan Hamilton ir anglų matematikas Thomas Penyngton Kirkman. Manoma, kad pirmoji bendra TSP formą buvo pateikta Kalr Menger. Šiais laikas TSP apibrėžiamas taip:

> Turint n miestų aibę ir kelionės kainą (ar atstumą) tarp kiekvienos galimos poros, keliaujančio pirklio uždavinys yra surasti geriausią galimą būdą aplankyti visus miestus ir grįžti į pradinę padėtį taip, kad kelionės kaina (ar atstumas) būtų minimalus.

TSP uždaviniai paprastai išskiriami į tris grupes. Simetrinis keliaujančio pirklio uždavinys, asimetrinis keliaujančio pirklio uždavinys ir kelių keliaujančių pirklių uždavinys. Šiame darbe buvo įgyvendintas simetrinis keliaujančio pirklio uždavinys, tai yra paprasčiausia uždavinio forma. Tai, kad uždavinys simetrinis reiškia, kad visai miestų aibei kelionės kaina tarp miestų nepriklauso

nuo kelionės krypties, t.y. $d_{rs} = d_{sr}$, kur d_{ij} yra kelionės kaina iš miesto i į miestą j. Jei kaina matuojama atstumu, tai uždavinys tampa trumpiausio maršruto paieška. Tada paprastai miestai pateikiami kaip koordinatės (x_i,y_i), o d_{ij} yra Euklidinis atstumas tarp miestų.

Keliaujančio pirklio uždavinio formuluotė įgavo savo formą dėl istorinių priežasčių. Šis uždavinys turi ir platesnes pritaikymo galimybes. Dažniausiai minimi pritaikymai yra:

- Elektrinių plokščių gręžimas. Sujungiant laidininką viename sluoksnyje su laidininku esančiu kitame sluoksnyje arba, įstatant į plokštes elementų kojeles, skylės turi būti išgręžtos plokštėje. Reikiamos skylės taip pat gali būti skirtingų dydžių. Taigi mašinos galvutė, gręžianti skyles, gali kelis kartus keisti gręžimo įrangą. Galvučių keitimo operacija yra ganėtinai brangi laiko atžvilgiu, todėl pirma išgręžiamos to paties dydžio skylės. Šis darbas gali būti traktuojamas kaip keletas keliaujančio pirklio uždavinių, kuriame siekiama rasti minimalų galvutės kelionės laiką išgręžiant visas skyles.

- Rentgeno spindulių kristalografija. Kristalografija yra mokslas apie kristalų struktūros analizę. Difraktometras yra naudojamas gauti informaciją apie kristalų struktūrą. Ši informacija yra renkama matuojant rentgeno atspindžio spinduliuotės intensyvumą iš įvairių pozicijų. Eksperimento kokybė nepriklauso nuo matavimo eilės tvarkos, tačiau tai gali daryti stiprią įtaką eksperimento trukmei. Nors pats matavimas gali būti atliktas greitai, prireikia laiko perkelti aparatą į kitą poziciją. Kadangi kai kuriems eksperimentams gali reikėti tikrinti šimtus tūkstančių pozicijų, tai užtrunka nemažai laiko. Aparato pozicija yra valdoma motorais, o kompiuteriu galima labai tiksliai suskaičiuoti laiką reikalingą kiekvienos pozicijos pakeitimui. Taigi tai tampa keliaujančio pirklio uždaviniu, kuriam reikia rasti kelią, kuriuo galima greičiausiai apeiti visus matavimo taškus.

- Užsakymo sudėjimas sandėliuose. Atvykus naujoms prekėms į sandėlį galima pradėti ruošti atitinkamus užsakymus. Tam reikia surinkti visas užsakymo prekės iš sandėlio ir išsiųsti jas klientui. Sąsaja su keliaujančio pirklio uždaviniu yra gan akivaizdi. Prekės yra sudėtos skirtingose sandėlio vietose ir reikia rasti greičiausią kelią, kuriuo galima surinkti visas užsakymo prekes.

2.2.1.2. 0-1 kuprinės uždavinys

Kuprinės uždavinio [KPP+04] tyrinėjimai yra minimi jau 1897 metais. Šio uždavinio pavadinimas kilęs iš matematiko Tobias Dantzig darbų, o šiais laikais formuluojamas:

Turint *n* daiktų aibę, iš kurių kiekvienas turi vertę ir svorį, kuprinės uždavinys yra nustatyti poaibį daiktų, neviršijantį tam tikro svorio ir turinčios didžiausią galimą vertę.

Ši formuluotė atitinka 0-1 kuprinės uždavinį, kuris ir buvo įgyvendintas šiame darbe. Ši formuluotė skiriasi nuo kitų tuo, kad leidžia turėti tik vieną daikto kopija, kitokios formuluotės gali leisti paimti tuos pačius daiktus ribotą ar neribotą skaičių kartų.

Kaip ir keliaujančio pirklio uždavinys, kuprinės uždavinys turi platesnes pritaikymo galimybės, nei akivaizdu iš formuluotės. Vieni labiausiai paplitusių pritaikymų:

- Testų, su pertekliniu uždavinių skaičiumi, vertinimas. Tai vienas iš pirmųjų šio uždavinio pritaikymų. Testų laikytojams pateikiamas testas, kur kiekviena užduotis verta skirtingo balų skaičiaus, o bendra balų suma didesnė, nei reikia maksimaliam įvertimui. Testų laikytojams atlikus visus uždavinius geriausiai pagal savo galimybes, naudojamas kuprinės uždavinys rasti atliktų uždavinių poaibį su kuriuo maksimali užduočių balų suma atitinka leidžiamą, o studentų surinkti taškai maksimizuoti.

- Investicijų portfelio optimizavimas. Optimizuojat investicijas labiausiai ribotas resursas būna biudžetas. Šiuo atveju galima pirkti daugiau nei vieną akciją, o papildomos akcijos gali mažinti grąžą. Esant didesniems biudžetams aktualu juos išnaudoti efektyviai, todėl sprendžiamas kuprinės uždavinys padeda tai pasiekti.

- Planavimas. Dauguma planavimo uždavinių gali būti formuluojami kaip kuprinės uždavinys. Šiuo atveju laikas būna ribojantis resursas. Tai yra labai aktualu atvejais kai eikvojamas laikas atitinka didelius kaštus. Norint efektyviai išnaudoti prieinamus resursus galima spręsti kuprinės uždavinį.

2.2.1.3. Kvadratinio priskyrimo uždavinys

Kvadratinio priskyrimo uždavinys [OEC+98] buvo pristatyta 1957 metais ekonomikų Tjalling C. Koopmans ir Martin Beckmann siekiant modeliuoti gamyklų išdėstymo problemą. Šiais laikais uždavinys formuluojamas šitaip:

> Yra n gamyklų ir n lokacijų. Tarp kiekvienos lokacijų poros pateikiamas atstumas tarp jų, o kiekvieną gamyklų pora turi svorį ar tėkmę (t.y. kiekis resursų pervežamų tarp gamyklų). Kvadratinio priskyrimo uždavinys yra priskirti visas gamyklas lokacijoms taip, kad atstumų ir tėkmės sandaugos suma buvo minimizuota.

Kvadratinio priskyrimo problemos formuluotė yra ganėtinai bendra, netgi keliaujančio pirklio uždavinys gali būti laikomas specialiu kvadratinio priskyrimo atveju, kai tėkme tarp gamyklų sudaro žiedinę struktūrą ir yra pastovios vertės.

Nors originali uždavinio formuluotė ir kalba apie gamyklų išdėstymą, šio uždavinio rezultatai yra plačiai pritaikomi. Keletas pritaikymo pavyzdžių:

- Klaviatūros išdėstymas. Klaviatūros mygtukų išdėstymas gali būti formuluojamas kaip kvadratinio priskyrimo uždavinys. Turint simbolių aibę, kurią reikia išdėstyti klaviatūroje, ir kiekvienos simbolių poros pasitaikymo tikimybę galima spręsti kvadratinio priskyrimo uždavinį, kurio rezultatas pateiks optimalų klaviatūros išdėstymą.

- Oro uosto vartų priskyrimas. Atskridus keleiviams ar vykstant į skrydį susidaro žmonių srautas iki ar iš skrydžio vartų. Oro uostai yra linkę sumažinti žmonių srautą oro uoste, todėl gali spręsti kvadratinio priskyrimo uždavinį tam atlikti. Be žmonių srauto oro uostams gali būti svarbus ir lagaminų srautas.

- Elektrinių komponentų išdėstymas schemoje. Tarp komponentų reikalinga jungianti medžiaga gali būti laikoma kaip tėkmė. Tada komponentų išdėstymas schemoje tampa kvadratinio priskyrimo uždaviniu, siekiančiu minimizuoti reikalingos jungiančios medžiagos kiekį.

2.2.2. Rėžių skaičiavimai

Kaip buvo minėta, rėžių skaičiavimo algoritmai skiriasi kiekvienai uždavinių grupei ir net tai pačiai grupei gali būti keli būdai apskaičiuoti rėžius. Šakų ir rėžių algoritmai priklauso nuo efektyvaus rėžių skaičiavimo, o nesant informacijai apie rėžius, šakų ir rėžių algoritmai virsta paprasčiausia išsamia paieška. Tam tikra pusiausvyra yra reikalinga pasirenkant apatinio rėžio skaičiavimo algoritmą, mat algoritmai, duodantys tikslesnius rėžius, paprastai reikalauja daugiau skaičiavimų. Sprendžiant didesnius uždavinius papildomas laikas skaičiuojant tikslesnius rėžius atsiperka. Dėl jų sumažėja paieškos medžiai, nes yra anksčiau atmetamos šakos, neturinčios optimalaus sprendinio. Šiame poskyryje bus trumpai aptarti apatinio rėžio skaičiavimo metodai darbe įgyvendintiems uždaviniams.

2.2.2.1. Keliaujančio pirklio uždavinio apatinis rėžis

Šiame darbe naudojamas apatinio rėžio skaičiavimo algoritmas keliaujančio pirklio uždaviniui yra gana paprastas. Skaičiuojant rėžį tam tikroje būsenoje pirma yra atmetami visi jau aplankyti miestai, o kiekvienam likusiam miestui randami du artimiausi miestai, atstumai iki jų susumuojami. Gautos sumos kiekvienam miestui yra vėl susumuojamos ir rezultatas dalijamas iš dviejų. Algoritmo eiga pseudokodu aprašyta 2 algoritme. Taip gaunamas įvertinimas geriausio kelio aplankyti likusius miestus. Toks algoritmas efektyviai apkarpo šakas ir nėra ypatingai reiklus skaičiavimams.

Algoritmas 2 Pseudokodas trumpiausiam keliui tarp miestų sužinoti.

```
keliasTarpMiestų = 0
for all miestas IN miestai do
    kaimynas1 = artimiausiasMiestas(miestas, miestai)
    kaimynas2 = antrasArtimiausiasMiestas(miestas, miestai)

    keliasTarpMiestų += atstumas(kaimynas1, miestas) + atstumas(kaimynas2, miestas)
end for
return keliasTarpMiestų / 2
```

2.2.2.2. 0-1 kuprinės uždavinio apatinis rėžis

Darbe naudotas apatinio rėžio skaičiavimo algoritmas kuprinės uždaviniui yra ir efektyvus, ir paprastas. Ji taikant daiktai turi būti išrikiuoti pagal savo vertės ir svorio santykį. Algoritmo eiga aprašyta 3 algoritme. Pirma vertingiausi daiktai už svorio vienetą yra sukraunami į kuprinę kol daugiau negali tilpti kitas daiktas. Tada paimamas sekantis daiktas eilėje ir pridedama tik ta dalis, kuri dar gali tilpti į kuprinę. Žvelgiant iš uždavinio sąlygos pusės tai nėra teisinga, nes negalima daiktų skaldyti į dalis, tačiau skaičiuojant apatinį rėžį svarbu tik geriausio galimo rezultato įvertinimas. Šis algoritmas greitai ir efektyviai nustato apatinį rėžį.

Algoritmas 3 Pseudokodas maksimaliai kuprinės vertei sužinoti.

```
kuprinė = { }
for all daiktas IN išrikiuoti(daiktai) do
    if arTelpaKuprinėje(daiktas) then
        kuprinė = kuprinė ∪ {daiktas}
    else
        kuprinė = kuprinė ∪ { telpantiDalis(daiktas)}
        BREAK
    end if
end for
return esamaVertė(kuprinė)
```

2.2.2.3. Kvadratinio priskyrimo uždavinio apatinis rėžis

Sprendžiant kvadratinio priskyrimo uždavinį nebuvo naudojamos specialus rėžių skaičiavimo būdas. Apatiniu rėžiu yra laikoma esamos konfigūracijos kaina. Taikant šį rėžį po-medžiai yra atmetami tada kai jų esama konfigūracijos kaina tampa didesnė nei geriausias žinomas viršutinis rėžis.

3. Praktinė dalis

Šiame skyriuje bus pristatytas darbo metu sukurtas hierarchinis lygiagretusis šakų ir rėžių algoritmas, bei jo lygiagretinimo rezultatai. Tačiau pirma bus aptartas Paskirstytųjų skaičiavimų tinklas, kuriuo naudojantis buvo gauti šio darbo rezultatai.

3.1. Paskirstytų skaičiavimų tinklas (PST)

Šiame darbe atliktiems skaičiavimams buvo naudotas Paskirstytųjų skaičiavimų tinklas[15], toliau PST, priklausantis Skaitmeninių tyrimų ir skaičiavimų centrui. PST yra specialiai paruoštas kompiuterių tinklas su galimybė vykdyti programas, naudojančias daugelį mazgų, leidžiant joms efektyviai apsikeisti duomenimis. 1 lentelėje yra pateikta PST superkompiuterio telkinių pagrindinės techninės charakteristikos.

1 lentelė. PST superkompiuterio telkinių techninės charakteristikos.

Pavadinimas	Mazgai	Procesoriai	RAM	Tinklas
alpha	26	2 x Intel Xeon X5650 2.66GHz	24GB	2x1Gbit/s ethernet
beta	42	2 x Intel Xeon X5650 2.66GHz	24GB	1Gbit/s,4xDDR(20Gbit/s) infiniband
gamma	3	4 x 4 x Intel Xeon X7550 2GHz	256GB	1Gbit/s,4xQDR(40Gbit/s) infiniband

PST gali naudotis visi registruoti VU MIF kompiuterio tinklo vartotojai. Be papildomos registracijos PST (eilėje *short*) gali naudotis ir esami VU MIF tinklo vartotojai. Norint gauti pilną prieigą reikia papildomos registracijos, kurią atlieka STSC darbuotojai. PST yra instaliuota Debian operacinė sistema su daugeliu paketų skaičiavimams, iš kurių aktualiausi šiam darbui yra `gcc`, *OpenMP* ir *OpenMPI*, taigi PST turi tai, ko reikia šiam darbui atlikti.

Užduočių vykdymo valdymui PST naudoja *Slurm*. *Slurm* yra atviro kodo, nuo klaidų atsistatanti ir prie klasterio prisitaikanti darbų valdymo ir planavimo sistema. Kaip klasterio darbo krūvio valdymo sistema *Slurm* atlieka tris pagrindines užduotis. Pirma, sistema skiria vartotojams išskirtinį ar ne priėjimą prie skaičiavimo resursų tam tikram laikui, kad vartotojai galėtų atlikti užduotis. Antra, sistema suteikia karkasą paleisti, vykdyti ir stebėti atliekamas užduotis. Trečia, sistema valdo laukiančių užduočių eilę pagal atitinkamus prioritetus ir poreikius. *Slurm* pateikia daug galimybių, paleidžiamų darbų vykdymo konfigūracijoms, iš kurių šiame darbe svarbiausios, aišku, yra mazgų skaičiaus ir naudojamų procesorių juose skaičius.

3.2. Sukurtas hierarchinis lygiagretusis šakų ir rėžių algoritmas

Neradus bibliotekos tinkamos hierarchiniam lygiagretinimui šakų ir rėžių algoritmo įgyvendinimui buvo pasirinkta parašyti savo algoritmą. Šis darbas buvo pradėtas nuo kelių uždavinių sprendimo įgyvendinimo pasianduojus šakų ir rėžių algoritmu ir jų hierarchinio lygiagretinimo. Tai buvo keliaujančio pirklio, kuprinės ir kvadratinio priskyrimo uždaviniai. Hierarchiškai išlygiagretinus kiekvieną iš uždavinių buvo pradėta kurti hierarchinio lygiagretinimo biblioteka. Pirma,

[15] `https://mif.vu.lt/cluster/`

sukūrus atskirus uždavinių įgyvendinimus, leido geriau suprasti, kas yra bendro tarp skirtingų šakų ir rėžių algoritmų uždavinių ir kuo galėtų pasirūpinti kuriama biblioteka.

Šis skyrius bus pradėtas paaiškinant, kaip buvo atliktas lygiagretinimas per bendrą atmintį ir siunčiant žinutes. Tada bus apžvelgta sukurta biblioteka, kokį funkcionalumą ji teikia ir kokias funkcijas reikia vartotojui įgyvendinti, norint įgyvendinti savo algorimtą. Paskutiniame skyrialyje parodomi lygiagretinimo rezultatai. Rezultatai gauti pasinaudojus sukurta biblioteka ir PST išspręsti tuos pačius tris uždavinius.

3.2.1. Lygiagretinimas per bendrą atmintį

Darbe lygiagretinimui per bendra atmintį naudojama *OpenMP* biblioteka. Darbams paskirstyti tarp gijų buvo naudojamas bendras darbų sąrašas visoms gijoms. Dėl šios priežasties šakų ir rėžių algoritmo įgyvendinimas dėl lygiagretinimo beveik nesikeičia.

Didžiausias pokytis nuo nuoseklaus algoritmo įvyksta duomenų struktūroje naudojamoje darbų sąrašui saugoti. Šiame darbe darbų sąrašui saugoti naudojamas paprastas stekas. Lygiagretinimo atveju, stekas turi užtikrinti taisyklingą veikimą įdėdant ir išimant uždavinius kelioms gijoms vienu metu. Tai buvo padaryta pasinaudojus užraktais (*angl., locking*). Kiekvienos įdėjimo ir išėmimo operacijos metu gija pasiima užraktą ir paleidžia jį tik tada, kai pabaigia operaciją. Svarbu, kad užraktas kartu rakina ir išėmimo ir įdėjimo operaciją. Tai yra svarbu, nes stekas tiek elemento išėjimo tiek elemento įdėjimo operacijos metu dirba su viršutinio steko elementu. Nerakinant abiejų operacijų vienu metu gali leisti įvykti situacijai kai skirtingos gijos turi savo steko versijas.

Kita svarbi paveikta algoritmo vieta yra geriausio žinomo sprendinio atnaujinimas. Esant daugeliui gijų jos gali aptikti geresnį sprendinį vienu metu ir bandyti jį atnaujinti. Tai nėra gerai, nes veda prie neprognozuojamų rezultatų. Dėl šios priežasties geresnio sprendinio aptikimas taip pat turi būti rakinamas. Tam pakanka *OpenMP* direktyvos `#pragma omp critical`, kuri nurodo, kad tik viena gija vienu metu gali vykdyti kodą nurodytame bloke. Paprastu atveju šis rakinimas bus atliekamas kiekvienam naujam sprendiniui tikrinant ar jie nėra geresnis už esamą. Kadangi rakinimas yra ganėtinai brangi operacija, tiek rakto gavimas, tiek kitų procesorių laukimas, verta tai atlikti kuo rečiau. Vienas iš būdų tai padaryti yra dvigubas patikrinimas ar dabartinis sprendinys yra geresnis nei žinomas. Pirma, patikrinimas atliekamas be rakinimo, ir jei dabartinis sprendinys nėra geresnis jis tiesiog atmetamas. Tačiau jei dabartinis sprendinys yra geresnis tada atliekamas rakinimas ir išnaujo tikrinima ar dabartinis sprendinys yra geresnis. Jei ir antrą kartą dabartinis sprendinys yra geresnis - yra atnaujinamas uždavinio viršutinis rėžis. Šitaip galima minimaliais kaštais užtikrinti taisyklingą geriausio žinomo sprendinio atnaujinimą daugelio gijų aplinkoje.

Išlygiagretinus algoritmą iškyla klausimas, kurio nebuvo nuoseklioje versijoje. Kada gijoms baigti darbą? Nuoseklaus algoritmo atveju viena gija ims ir dės uždavinius į darbų sąrašą, taigi jei sąraše nieko nėra, tai uždavinys yra baigtas. Tačiau išlygiagretintu atveju, kai daugelis gijų dirba su tuo pačiu darbų sąrašu, tai nėra taip trivialu. Jei viena gija randa tuščią darbų sąrašą, tai dar nereiškia, kad kita gija jo nepapildys. Dėl to atsiranda darbo pabaigos aptikimo poreikis. Šiame darbe naudojamas paprastas metodas, kai kiekviena gija pasižymi kada randa darbo ir nuima pasižymėjimą kai darbo neberanda. Tokiu būdu kai gija neranda darbo, bet nors viena kita gija

yra pasižymėjusi, kad dirba, uždavinio sprendimas nėra baigiamas ir gija toliau bandoma ieškoti darbo bendrame darbų sąraše. Kai darbų sąrašas bus papildytas uždavinys bus paimtas pirmosios gijos paprašiusios uždavinio. Jei visos gija neras darbo ir nei viena kita gija nėra pasižymėjusi, kad šiuo metu dirba vadinasi uždavinys yra baigtas ir visos gijos gali nutraukti darbą. Toks darbo pabaigos aptikimo trūkumas yra, kad pačioje uždavinio pradžioje ar jo pabaigoje, kai darbų sąrašas yra ganėtinai tuščias, galimas gijų tuščias sukimasis laukiant darbų. Gijos neturinčios darbo vis bandys nesėkmingai rasti uždavinį, kol arba jis atsiras arba uždavinys bus išspręstas.

3.2.2. Lygiagretinimas siunčiant žinutes

Paskirstytosios atminties sistemose toks lygiagretinimo modelis kaip su *OpenMP* nėra galimas, todėl tenka imtis kitokių veiksmų. *MPI* neturi direktyvų, lygiagretinimas yra valdomas *MPI* bibliotekos teikiamomis funkcijomis. Dėl to visas lygiagretinimas turi būti pilnai išreikštas programuotojo. Todėl paprastai lygiagretinimas su *MPI* yra sudėtingesnis nei su *OpenMP*, programuotojas turi pasirūpinti visomis lygiagretinimo detalėmis. Šiame darbe lygiagretinimas siunčiant žinutes buvo įgyvendintas be dinaminio balansavimo, t.y. sprendimo eigoje procesai tarpusavyje nesidalina uždaviniais. To didžiausias trūkumas yra, tai kad algoritmo veikimo laikas bus lygus ilgiausiai veikiančio proceso laikui. Aišku vieniems procesams baigus darbą laukti kitų nėra efektyvus resursų panaudojimas.

Vienas didžiausių pakeitimų dėl lygiagretinimo siunčiant žinutes yra atliktas pradėdant problemos sprendimą. Lygiagretinant algoritmą vienas pirmų kylančių klausimų yra kaip tolygiai paskirstyti pradinius darbus procesams. Tai tampa svarbiau, nes nėra dinaminio balansavimo. Vienas iš būdų yra sugeneruoti tam tikrą po-uždavinių skaičių ir suskirstyti juos procesams. Iki šiol naudojama pirma-gylis medžio šakojimosi strategija čia nelabai tinka. Nors iš anksto žinoti po-uždavinio sprendimo laiko negalima, galima tikėtis, kad giliau sugeneruoti po-uždaviniai bus išsprendžiami greičiau. Dėl to tolygiai paskirstant darbus norisi procesams dalinti to paties gylio po-uždavinius. Tam algoritmo pradžioje yra naudojama kitokia uždavinio medžio šakojimo strategija. Šio lygiagretinimo atveju algoritmas pradedamas nuo pirma-plotis strategijos, kuri stengiasi sugeneruoti visus aukštesnio lygio uždavinius prieš pereinant giliau. Ši strategija yra vykdoma iki kol yra pasiekiamas pakankamas, pasirenkamas kiekis po-uždavinių pradėti standartinį uždavinio sprendimo algoritmą. Kai turima pakankamai po-uždaviniu jie yra paeiliui įdedami į atitinkamo proceso darbų sąrašą. Tai galima, nes pagal *MPI* paleidus programą nuo pat programos pradžių visi procesai vykdo esanti kodą ir tik naudojant *MPI* bibliotekos funkcijas galima pakeisti procesų eiga. Procesai užpildo darbų sąrašą paprasčiausiai pasiimant tik tą po-uždavinį, kurio eilės numerio modulis atitinka proceso numerį. Toks pradinio darbo paskirstymas leidžia efektyviai tęsti darbą gijomis esančiomis procesuose.

Čia taip pat yra svarbu atnaujinti geriausią žinomą apatinį rėžį. Tačiau procesams neturint bendros atminties apie naują rėžį jie turi sužinoti laukdami žinutės. Įgyvendinta procedūra yra ganėtinai paprasta. Jei gija randa geresnį apatinį rėžį, ji patikrina ar nėra atkeliaujančios informacijos apie geresnius rėžius iš kitų procesų. Jei randa žinutę ir atėjęs rėžis yra geresnis nei tuo metu žinomas, pastarasis rėžis yra atnaujinamas. Sekantis gijos žingsnis yra vėl patikrinti ar jos rastas rėžis

yra geresnis nei žinomas ir jei taip, jį atnaujinti ir išsiųsti žinutę visiems procesams apie naujesnį rėžį. Nors problemos sprendinys paprastai būna tam tikras masyvas atitinkantis resursų priskyrimą (pvz.: keliaujančio pirklio uždavinio atveju tai yra miestų aplankymo tvarka), siunčiant žinutę pakanką tik sprendinio gerumo indikacijos (pvz: keliaujančio pirklio uždavinio atveju sprendinio kelio ilgis) t.y. tik viršutinio rėžio, be kitos informacijos medžio mazge. Siunčiant tik rėžį yra minimizuojamos žinučių dydis. Tačiau tai iškelia problemą, kad tikrąjį pilną sprendinį žinos tik jį radęs procesas, o kiti procesai turės tik to sprendinio indikaciją. Tai sukelia sinchronizacijos poreikį procesams baigus darbą. Kiekvienas procesas seka turimo apatinio rėžio šaltinį (proceso numerį). Atėjus sinchronizacijos laikui, pirma, procesai priima žinutes, kurių dar nebuvo priėmė. Po to procesai surenka informaciją apie uždavinio sprendinio šaltinį. Tada šaltinis transliuoja kitiems procesams uždavinio sprendinį. Tai atlikus visi procesai turi teisingą atsakymą, kuris gali būti pateikiamas vartotojams.

3.2.3. Sukurta hierarchinio lygiagretinimo biblioteka

Kaip jau buvo minėta, hierarchinio lygiagretinimo biblioteka šakų ir rėžių algoritmams buvo pradėta nuo atskirų uždavinių įgyvendinimo. Buvo įgyvendinti ir hierarchiškai išlygiagretinti keliaujančio pirklio, kuprinės ir kvadratinio priskyrimo uždaviniai. Matant visus įgyvendinimus buvo matyti kokį funkcionalumą galima iškelti į biblioteką ir ką reikėtų palikti kiekvienam uždavinių įgyvendinimui. Turint omeny, kad ši biblioteka turi teigti hierarchinį šakų ir rėžių algoritmų lygiagretinimą, lygiagretinimas yra esminis bibliotekos funkcionalumas. Kuriant biblioteka buvo išskirtos trys grupės funkcionalumo, kurį gali ar turi aprašyti vartotojas. Tai yra darbų sąrašo įgyvendinimas tiek statinio balansavimo metu, tiek sprendimo metu, nuo konkretaus uždavinio priklausantys šakų ir rėžių algoritmo metodai, metodus nurodančius duomenis persiunčiant informaciją į kitus procesus. Taigi norint hierarchiškai išlygiagretinti šakų ir rėžių algoritmą nereikia rūpintis lygiagretinimo detalėmis. Įgyvendinus biblioteką, kartu įgyvendinti ir tie patys tris uždaviniai, šį kartą jau pasinaudojus biblioteka. Biblioteka galima rasti `GitHub`[16], ji pateikia su įgyvendintais uždaviniais kaip pavyzdžiais.

Darbų sąrašas yra vienas pagrindinių šakų ir rėžių algoritmo įgyvendinimo aspektų. Yra daug pasirinkimų kokias duomenų struktūras galima naudoti saugant darbus atlikimui, šiuo klausimu įvairūs autoriai atliko nemažai tyrimų. Dėl to bibliotekos vartotojas turi pasirinkimą pateikti savo duomenų struktūrą darbams saugoti. Tačiau biblioteka pateikia ir savo duomenų struktūras, kuriomis galima naudotis neįgyvendinant savo darbų sąrašų. Įgyvendintoje bibliotekoje naudojami du darbų sąrašai. Vienas naudojamas atliekant statinį balansavimą, kitas likusią sprendimo dalį. Šiuo atskyrimu leidžiama bibliotekos vartotojui naudoti kitokia medžio tyrinėjimo strategija. Tai gali būti norima, dėl to, kad statiniu balansavimu paprastai siekiama tolygiai padalinti darbus tarp procesų. Tam tinka paieškos medžio tyrinėjimo strategijos medį tiriančios į plotį. Tuo tarpų ieškant sprendinio paprastai norima išvengti medžio tyrinėjimo į plotį. Bibliotekos vartotojui taip pat reikėtų turėti omenyje, kad darbų sąrašas bus naudojamas daugiagijiškoje aplinkoje. Taigi tiek darbų išėmimas tiek įdėjimas turi būti saugiau operacijos tokioje aplinkoje. Svarbu paminėti ir tai, kad

[16] `https://github.com/TomasSeniut/HPBB`

darbų sąrašas turi pasirūpinti ir darbų pabaigos aptikimu, t.y. darbų sąrašas neturi nurodyti, kad jis yra tuščias, jei jame dar gali atsirasti darbų.

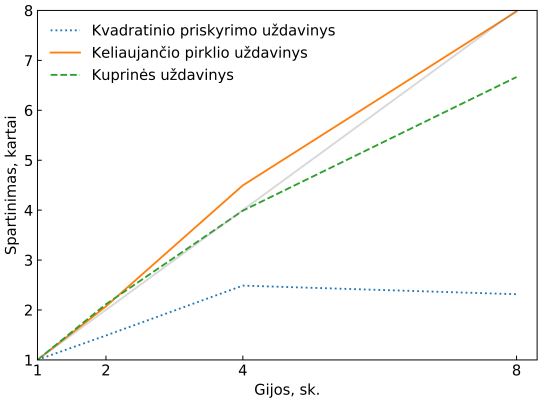
Kalbant apie konkretaus uždavinio specifiką, vartotojui reikia pateikti penkias funkcijas. Šios funkcijos apibrėžia šakų ir rėžių algoritmo darbą. Šių funkcijų pasirinkimui nemažą įtaką darė aptarta **Zram** biblioteka, todėl pasirinktos funkcijos turi nemažai tarpusavio panašumų. Vartotojui nereikia rūpintis tuo, kad funkcijos yra vykdomos daugiagijiškoje aplinkoje, nes kiekviena funkcija dirbs tik su savo turimu uždaviniu, o sinchronizacija tarp gijų pasirūpina biblioteka. Tarp funkcijų viena yra labiau techninė, nei tiesiogiai susijusi su šakų ir rėžių algoritmu. Vartotojas, jei to reikia, turi nurodyti kaip pašalinti tyrinėjamo medžio mazgą iš skaičiavimo mašinos atminties. Tai svarbu, nes sprendžiami uždaviniai sugeneruoja pakankamai mazgų, kad kiltų problemų su laisva atmintimi jų nešalinant. Viena iš tiesiogiai susijusių su šakų ir rėžių algoritmo sprendimu funkcijų yra po-uždavinių generavimas iš turimo uždavinio. Čia vartotojas turi nurodyti, kaip generuoti po-uždavinius, tačiau neturi rūpintis ar sugeneruotus po-uždavinius reikia atmesti, ar jie yra geresni sprendinai. Sugeneravus po-uždavinius reikia pasakyti, ar gautas po-uždavinys yra galimas sprendinys. Jei tai yra galimas sprendinys, biblioteka tikrins, ar jis yra geresnis nei žinomas geriausias sprendinys. Tai atliks kita funkcija, kurią turi pateikti vartotojas. Ši funkcija kaip parametrus gaus dabartinį sprendinį ir geriausią žinomą sprendinį ir turės atsakyti į klausimą, ar naujai rastas sprendinys yra geresnis, jei taip, biblioteka pasirūpins juo atnaujinimu. Jei konkretus po-uždavinys nėra galimas sprendinys, tada biblioteka tikrins, ar jai nereikia po-uždavinio atmesti. Šiam patikrinimui taip pat reikalinga vartotojo pateikta funkcija. Šių funkcijų pateikimas leidžia bibliotekai vykdyti šakų ir rėžių algoritmą.

Kita grupė funkcijų, kurias reikia pateikti vartotojui yra susijusi su informacijos perdavimu tarp procesų. Tam reikia keturių funkcijų. Nors bibliotekoje nėra įgyvendintas dinaminis balansavimas, procesai vis tiek keičiasi informacija perduodant viršutinį rėžį, o sprendimo pabaigoje perduoda patį sprendinį. Trys iš šių funkcijų reikalingos nurodyti duomenys, kurie bus siunčiami ir gaunami žinute. Vartotojui reikia nurodyti iš turimo tyrinėjamo medžio mazgo, kurią informaciją persiųsti kaip rėžį, ką kaip patį sprendinį. Be to, bibliotekai reikia pateikti ir gaunamo rėžio tipą bei atminties vietą, kuriose saugoti rėžį. Paskutinė funkcija reikalingą pasakyti ar reikia atnaujinti geriausią žinomą sprendinį procese su informacija, gauta iš kito proceso. Siunčiamos informacijos atskyrimas padarytas norint sumažinti komunikacijos dydį tarp procesų. Dėl šios priežasties vartotojui tenka įgyvendinti daugiau funkcijų, nei pakanka tiesiog persiųsti visą uždavinį.

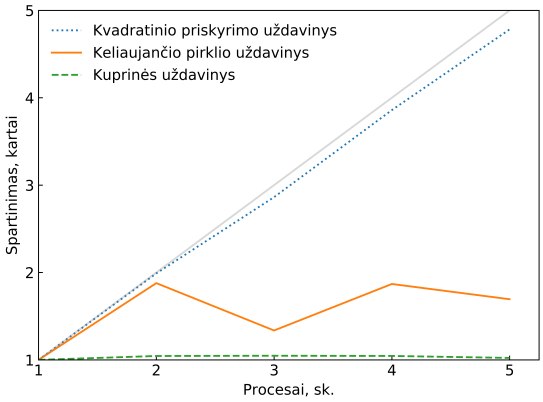
3.2.4. Lygiagretinimo rezultatai

Šiame skyriuje pateikti rezultatai buvo gauti pasinaudojus PST, kuriuo pasinaudojant buvo atlikti skaičiavimai. Visi uždaviniai su ta pačia konfigūracija buvo spręsti 5 kartų. Surinkti sprendimų laikai kiekvienai konfigūracijai buvo išvalyti išėmus laikus besiskiriančius nuo kitų kelis kartus. Tai nebuvo daugiau nei 2 iš 5 gautų sprendimo laikų.

Rezultatai gauti lygiagretinant per bendrą atmintį naudojant vieną darbų sąrašą pateikti 1 pav., juose matyti spartinimo priklausomybė nuo gijų skaičiaus įgyvendintiems uždaviniams. Iš paveikslo matyti, kad du uždaviniai pasiekė tiesinį ar beveik tiesinį spartinimą naudojant iki 8 gijų.



1 pav. Įgyvendintų uždavinių spartinimo priklausomybė nuo gijų skaičiaus. Lygiagretinimas atliktas naudojant *OpenMP*. Pilka linija žymi tiesinį spartinimą.



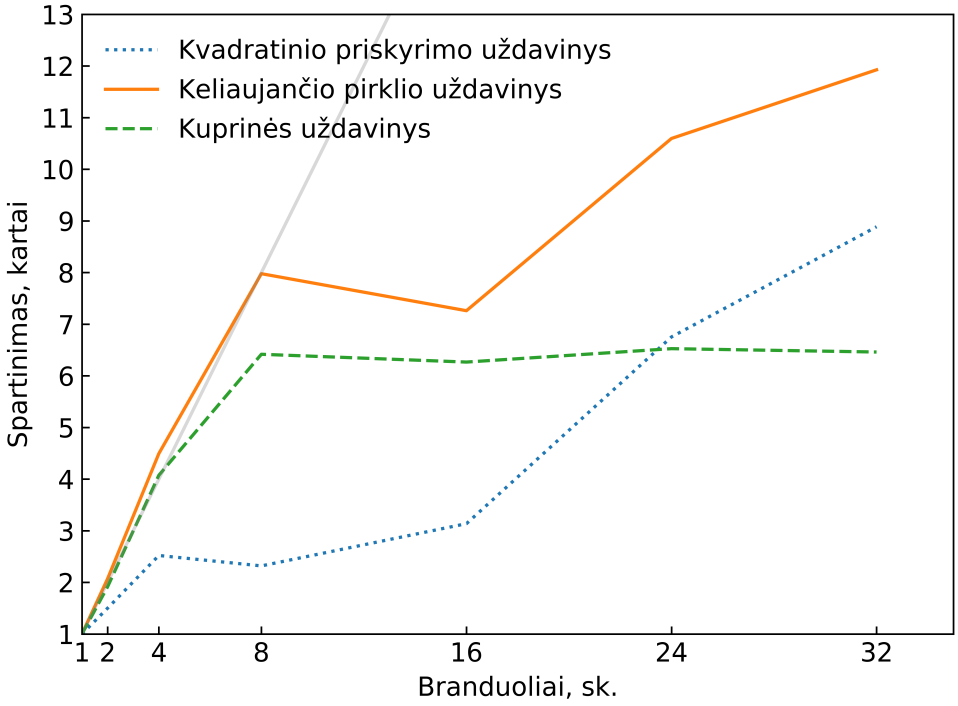
2 pav. Įgyvendintų uždavinių spartinimo priklausomybė nuo gijų skaičiaus. Lygiagretinimas atliktas naudojant *MPI*. Pilka linija žymi tiesinį spartinimą.

Sprendžiant kvadratinio priskyrimo uždavinį pasiektas spartinimas tik iki 2.5 karto. Rezultatai lygiagretinant siunčiant žinutes tarp procesų naudojant atskiros darbų sąrašus procesams pateikti 2 pav. Gaunant rezultatus naudota iki 5 procesų, rezultatuose matyti gerokai prastesnis uždavinių spartinimas išskyrus kvadratinio priskyrimo uždavinį.

Tiriant spartinimą siunčiant žinutes patikrinta spartinimo priklausomybei daroma įtaka procesų būvimo tame pačiame ar atskiruose superkompiuterio mazguose. Skirtumas tarp spartinimo tame pačiame ir atskiruose mazguose yra nereikšmingas, tai rodo, kad uždavinių sprendimo laikas nėra ribojamas informacijos apsikeitimo greičiu tarp procesų.

Kvadratinio priskyrimo uždavinio spartinimas rodo atvirkščias priklausomybes nei galima būtų tikėtis. Lygiagretinant pasinaudojus tik *MPI* su darbu sąrašu kiekvienam procesui pasiekia beveik tiesinį spartinimą, tuo tarpu naudojant tik *OpenMP* su vienu darbų sąrašu tarp gijų pasiektas tik 2.5 kartų spartinimas esant 4 gijoms. Iš 2 ir 1 pav. galima spręsti, kad lygiagretinimas naudojantis bendru darbu sąrašu reikalauja per daug sinchronizacijos tarp gijų. *MPI* atveju, kiekvienam procesui turint savo darbų sąrašą, procesai nėra stabdomi sinchronizacijos. Taip pat *MPI* atveju kiekvienas procesas sprendžiant uždavinį apdoroja labai panašų mazgų skaičių (iki 1% skirtumas apdorotame mazgų skaičiuje), tai rodo labai tolygiai paskirstytus paieškos po-medžius tarp procesų. Šiuos pastebėjimus galima paaiškinti tuo, kad kvadratinio priskyrimo uždavinys naudoja neefektyvų apatinio rėžio skaičiavimo algoritmą. Dėl paprasto apatinio rėžio skaičiavimo paieškos po-medžiai yra atmetami vėlai. Dėl to darbai yra ganėtinai lygiai padalinami darbai tarp procesų. Neefektyvi apatinio rėžio skaičiavimo procedūra, nereikalauja daug skaičiavimų, todėl gijos greičiau apdoroja mazgus ir daugiau savo laiko praleidžia naudodamos bendrą darbų sąrašą, kuriam reikia sinchronizacijos. Dėl šių priežasčių lygiagretinant uždavinius, kurių paieškos medžiai padalinami tolygiai, labiau tinka atskiri darbų sąrašai.

Keliaujačio pirklio uždavinys rodo 7.9 kartų spartinimą naudojant bendrą darbų sąrašą pasinaudojus *OpenMP* ir 1.9 kartų spartinimą naudojantis atskirais darbų sąrašais pasinaudojus *MPI*. Turint vieną darbų sąrašą pasiektas daug geresnis spartinimas nei naudojant kelis darbų sąrašus. Gautos priklausomybes yra tai ko ir galima buvo tiktėtis iš lygiagretinimo naudojant *OpenMP* ir



3 pav. Įgyvendinto algoritmo spartinimas pasitelkiant *OpenMP* kartu su *MPI* naudojant skirtingą branduolių skaičių. Lygiagretinimui iki 8 branduolių buvo naudojama tik *OpenMP*. Toliau buvo prijungiami *MPI* procesai, kurių kiekvienas turi 8 gijas.

MPI.

Kuprinės uždavinys rodo 6.7 kartų spartinimą naudojantis *OpenMP* su vienu darbų sąrašu ir tik 1.05 kartų spartinimą naudojantis *MPI* su daugeliu darbų sąrašų. Nors naudojantis vienu darbų sąrašu pasiekiamas labai geras spartinimas, naudojant kelis darbų sąrašus spartinimo beveik nėra. Taip yra dėl to, kad paieškos po-medžiai nėra padalinami tolygiai tarp procesų. Po-medžiai vedantys prie optimalaus sprendinio yra giliai paieškos medyje, o kiti po-medžiai yra greitai atmetami. Statinis balansavimas nepasiekia reikiamo paieškos po-medžio gylio, dėl to po-medis vedantis prie optimalaus sprendinio atitenka vienam procesui. Dėl to statinis darbų balansavimas nėra pakankamas, paskirstyti darbus procesams, šioje situacijoje. Šiam uždaviniui taip atsitinka kai daiktų svoris ir vertė yra nekoreliuoti, tada uždavinio sprendimo pradžioje skaičiuojami apatiniai rėžiai efektyviai atmeta paieškos po-medžius. Dėl šių priežasčių keli darbų sąrašai esant labai netolygiam darbų paskirstymui tarp procesų nėra tinkamas pasirinkimas.

Kartu naudojant *OpenMP* ir *MPI*, kai kiekvienas procesas turi savo darbų sąrašą, o gijos procese dalinasi vienu darbų sąrašu, spartinimo rezultatai yra pateikti 3 pav.. Paveiksle iki 8 branduolių naudojama tik *OpenMP*, o toliau pridedami *MPI* procesai, kiekvienas turintis 8 gijas. Keliaujančio pirklio uždavinys pasiekė 12 kartų spartinimą naudojant 4 procesus, kurių kiekvienas turi po 8 gijas. Kvadratinio priskyrimo uždavinys su tuo pačiu gijų ir procesų skaičiumi pasiekė 8.9 kartų spartinimą. Kuprinės uždavinys pasiekė 6.5 kartų spartinimą, tačiau spartinimas nepriklausė nuo procesų skaičiaus. Iš rezultatų matyti, kad reikiamomis sąlygomis hierarchinis lygiagretinimas gali pasiekti didesnį spartinimą nei *OpenMP* ir *MPI* atskirai.

3.2.5. Rezultatų palyginimas

Populiarėjant hibridiniui lygiagretinimui daugėja atliktų darbų ir tyrimų šioje sritije. Todėl gerai yra apžvelgti, kitus darbus atliktus hibridiškai lygiagretinančius algoritmus minimus šiame darbe. Nors literatūroje nebuvo rasti hierarchiškai išlygiagretinti šakų ir rėžių algoritmai sprendžiantys tuos pačius uždavinius, šiame skyriuje bus aptarti panašiausi darbai. Jų yra du – [HCP+13a] ir [QSC+12].

Pirmajame darbe [HCP+13a] naudojamos *MPI* ir *Pthreads* technologijos. Darbe pasinaudojus šakų ir rėžių algoritmais sprendžiamas globalios optimizacijos uždavinys. Autoriai palygina *MPI*, *Pthreads* ir hibridiniu lygiagretinimu pasiektą spartinimą. *MPI* lygiagretinimo atveju yra naudojamas statinis balansavimas, turint vieną darbų sąrašą kiekvienam procesui, o viršutinias rėžiais nėra dalinamasi. *Pthreads* lygiagretinimo atveju taip pat yra naudojami keli darbų sąrašai. Gija gali sukurti naują giją ir duoti pusę savo turimų darbų. Šiuo atveju kiekviena gija turės savo darbų sąrašą, o viršutinis rėžis laikomas kintamajame, prieinamu visoms gijoms. Hibridiniame modelyje darbas pradedamas su viena gija procesui, augant darbų skaičiui kuriamos naujos gijos. Autorių rezultatuose matyti, kad statinis balansavimas tolygiai paskirsto darbus tarp procesų. Nors esant daugiau procesų darbai yra paskirstomi mažiau tolygiai, autoriai nurodo, kad turint net 32 procesus apkrovos skirtumas tarp procesų nebuvo didesnis nei 30%. Autorių rezultatai, rodo, gerus, beveik tiesinius spartinimus. Tai sutampa su šiame darbe gautais rezultatais, kai tolygiai paskirčius uždavinį tarp procesų gaunami beveik tiesiniai spartinimai naudojant kelis darbų sąrašus.

Kitame darbe [QSC+12] naudojamos *MPI* ir *OpenMP* technologijos sprendžiant kuprinės uždavinį, tačiau sprendimas naudoją kitą, ne šakų ir rėžių, algoritmą. Darbe naudojamas dinaminio programavimo algortimas išspręsti krepšelio uždaviniui. Kadangi dinaminio programavimo algoritmai yra už šio darbo srities, nebus plačiau aptartas autorių algorimo įgyvendinimas. *MPI* darbe naudojamas procesams spręsti atskiras sritis į kurias padalintas uždavinys. Prijungus *OpenMP* išlaikomas tas pats branduolių skaičius sumažinant procesų skaičių. *OpenMP* naudojamas išlygiagretinti *for* ciklą kiekviename procese. Iš autorių rezultatų matyti, kad pereinant nuo *MPI*prie hibridinio *OpenMP* ir *MPI* sprendimo pasiekiamas 2 kartų spartinimas.

Rezultatai ir išvados

Šiame darbe buvo padaryti trys svarbūs dalykai. Apžvelgtos egzistuojančios šakų ir rėžių algoritmus lygiagretinančios bibliotekos, įgyvendinti hierarchiškai išlygiagretinti keli kombinatoriniai uždaviniai šakų ir rėžių algoritmu, bei įgyvendintas algoritmas ištirtas pasinaudojus PST bei gauta spartinimo priklausomybė nuo naudojamų branduolių skaičiaus.

Iš viso darbe apžvelgtos 9 šakų ir rėžių algoritmą lygiagretinančios bibliotekos. Dvejos ištirtos ir eksperimentiškai sprendžiant keliaujančio pirklio uždavinį. Abi šios bibliotekos turėjo trūkumų. Pirmoji, `Zram`, buvo skirta būtent šakų ir rėžių algoritmams. Biblioteka pateikė bendrą šakų ir rėžių algoritmo įgyvendinimą ir lygiagretinimą, o uždavinio specifiką reikėjo įgyvendinti pačiam. Dėl to buvo įdomu panagrinėti bibliotekos architektūrą ir vartotojo naudojimo būdą. Tačiau biblioteka nelygiagretino uždavinių kaip turėtų, biblioteka veikė tik nuosekliuoju rėžimu. Antroji biblioteka, `SYMPHONY`, buvo labiau bendro pobūdžio. Biblioteka siūlo pačių įvairiausių algoritmų ir uždavinių sprendimą. Dėl to biblioteka labai didelė, ilgas diegimas, ir ganėtinai painus jos veikimas. Lygiagretinimas joje atliktas naudojantis tik *OpenMP*, pati biblioteka buvo sudaryta iš per didelio skaičiaus komponentų, kad galima būtų galvoti apie jos lygiagretinimo praplėtimą naudojant *MPI*. Taigi, apžvelgtos bibliotekos neįgyvendina hierarchinio lygiagretinimo.

Neradus tinkamos bibliotekos buvo nuspręsta įgyvendinti savo šakų ir rėžių algoritmus ir iš jų sukurti biblioteką. Tam buvo pasirinkti trys uždaviniai tinkantys šakų ir rėžių algoritmams: keliaujančio pirklio, 0-1 kuprinės ir kvadratinio priskyrimo. Pradžioje šie uždaviniai įgyvendinti atskirai nedarant jokios bibliotekos. Norėta pamatyti atskirų algoritmų įgyvendinimą ir iš jų nuspręsti, kokia turi būti bibliotekos forma ir tik tada sukurti biblioteką. Sukūrus hierarchiškai lygiagretinančią šakų ir rėžių algoritmus biblioteką, su ja įgyvendinti ir minėti uždaviniai. Hierarchinis lygiagretinimas pasiektas naudojant *OpenMP* lygiagretinimui per bendrą atmintį ir *MPI* – žinučių siuntimui. Lygiagretinimui per bendrą atminti įgyvendinti buvo naudotas bendras darbų sąrašas visoms gijoms. Lygiagretinant siunčiant žinutes, kiekvienas procesas turi savo darbų sąrašą. Uždavinio sprendimo pradžioje atliekamas statinis darbų balansavimas, o eigoje procesai dalinasi rastais sprendiniais. Taigi, buvo sukurta hierarchiškai išlygiagretinti šakų ir rėžių algoritmus biblioteka.

Sprendžiant uždavinius pasinaudojus PST buvo gautos uždavinių spartinimo priklausomybės. Kalbant apie lygiagretinimą su *OpenMP*, naudojantis vienu darbų sąrašu, uždaviniai pasiekė 2.5, 7.9 ir 6.7 kartų spartinimą kvadratinio priskyrimo, keliaujančio pirklio ir kuprinės uždaviniams atitinkamai. Lygiagretinant su *MPI*, pasinaudojant daugelių darbų sąrašu, pasiekti spartinimai buvo mažesni, 4.8, 1.9 ir 1.05 kartų kvadratinio priskyrimo, keliaujančio pirklio ir kuprinės uždaviniams atitinkamai. Iš rezultatų galima spręsti, kad keli darbų sąrašai yra tinkami tik, kai statinis balansavimas gali tolygiai padalinti darbus tarp procesų, kitu atveju keli darbų sąrašai neduos tinkamų rezultatų. Lygiagretinimas naudojantis bendru darbų sąrašu pasiekia didesnį spartinimą ir yra ypač tinkamas, kai gijos praleidžia didžiąją laiko dalį tyrinėdamos paieškos medį ir tik mažą dalį – sąveikaudamos su bendru darbų sąrašu.

Tiek lygiagretinimas per bendrą atmintį, tiek lygiagretinimas siunčiant žinutes turi privalumų ir trūkumų. Lygiagretinimas bendra atmintimi pasiekia geresnį spartinimą, tačiau yra apribotas

branduolių skaičiaus procesoriuje. Lygiagretinimas siunčiant žinutes gali pasiekti didesnius lygiagretinimo mastus, tačiau geras spartinimas yra pasiekiamas sunkiau. Hierarchiniu lygiagretinimu, naudojantis ir lygiagretinimą per bendrą atmintį ir lygiagretinimą siunčiant žinutes, siekiama sujungti abiejų būdų privalumus. Darbo rezultatai parodė, kad kartu naudojamas lygiagretinimas per bendrą atmintį ir siunčiant žinutes leidžia išnaudoti daugiau prieinamų branduolių neprarandant lygiagretinimo efektyvumo.

Literatūra

[AAB+02] Enrique Alba, Francisco Almeida, M Blesa, J Cabeza ir k.t. Mallba: a library of skeletons for combinatorial optimisation. *Euro-Par 2002 Parallel Processing*:63–73,

[ABC+07] David L. Applegate, Robert E. Bixby, Vasek Chvatal ir William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, ISBN: 0691129932, 9780691129938.

[AFO06a] Kento Aida, Yoshiaki Futakata ir Tomotaka Osumi. Parallel Branch and Bound Algorithm with the Hierarchical Master-Worker Paradigm on the Grid. 47(Acs 15),

[AFO06b] Kento Aida, Yoshiaki Futakata ir Tomotaka Osumi. Parallel branch and bound algorithm with the hierarchical master-worker paradigm on the grid. *IPSJ Digital Courier*, 2:584–597,

[ANF03] Kento Aida, Wataru Natsume ir Yoshiaki Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, p.p. 156–163. IEEE,

[BMF+99] Adrian Brüngger, Ambros Marzetta, Komei Fukuda ir Jurg Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 90:45–63,

[BMT08] Ahcene Bendjoudi, Nouredine Melab ir El-Ghazali Talbi. P2p design and implementation of a parallel branch and bound algorithm for grids. *International Journal of Grid and Utility Computing*, 1(2):159–168,

[BMT12a] Ahcene Bendjoudi, Nordine Melab ir El-Ghazali Talbi. An adaptive hierarchical master–worker (ahmw) framework for grids—application to b&b algorithms. *Journal of Parallel and Distributed Computing*, 72(2):120–131,

[BMT12b] Ahcène Bendjoudi, Nouredine Melab ir E-G Talbi. Hierarchical branch and bound algorithm for computational grids. *Future Generation Computer Systems*, 28(8):1168–1176,

[BT89] Dimitri P Bertsekas ir John N Tsitsiklis. *Parallel and distributed computation: numerical methods*, tom. 23. Prentice hall Englewood Cliffs, NJ,

[CLR06] Teodor Gabriel Crainic, Bertrand Le Cun ir Catherine Roucairol. Parallel branch-and-bound algorithms. *Parallel combinatorial optimization*:1–28,

[DFR90] Frank Dehne, Afonso G Ferreira ir Andrew Rau-Chaplin. Parallel branch and bound on fine-grained hypercube multiprocessors. *Parallel Computing*, 15(1-3):201–209,

[DLC+06] A Djerrah, Bertrand Le Cun, V-D Cung ir Catherine Roucairol. Bob++: framework for solving optimization problems with branch-and-bound methods. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, p.p. 369–370. IEEE,

[DS80] Edsger W Dijkstra ir Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4,

[EHP15] Jonathan Eckstein, William E Hart ir Cynthia A Phillips. Pebbl: an object-oriented framework for scalable parallel branch and bound. *Mathematical Programming Computation*, 7(4):429–469,

[EPH01] Jonathan Eckstein, Cynthia A Phillips ir William E Hart. Pico: an object-oriented framework for parallel branch and bound. *Studies in Computational Mathematics*, 8:219–265,

[GC94] B. Gendron ir T. G. Crainic. Parallel Branch-and-Branch Algorithms: Survey and Synthesis, DOI: 10.1287/opre.42.6.1042.

[HCH+13a] Juan F R Herrera, Leocadio G. Casado, Eligius M T Hendrix, Remigijus Paulavicius ir Julius Zilinskas. Dynamic and hierarchical load-balancing techniques applied to parallel branch-and-bound methods. *Proceedings - 2013 8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2013*:497–502, DOI: 10.1109/3PGCIC.2013.85.

[HCH+13b] Juan FR Herrera, Leocadio G Casado, Eligius MT Hendrix, Remigijus Paulavicius ir Julius Zilinskas. Dynamic and hierarchical load-balancing techniques applied to parallel branch-and-bound methods. *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, p.p. 497–502. IEEE,

[HCP+13a] Juan F R Herrera, Leocadio G. Casado, Remigijus Paulavicius, Julius Ilinskas ir Eligius M T Hendrix. On a hybrid MPI-Pthread approach for simplicial branch-And-bound. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*:1764–1770, DOI: 10.1109/IPDPSW.2013.178.

[HCP+13b] Juan FR Herrera, Leocadio G Casado, Remigijus Paulavicius, Julius Zilinskas ir Eligius MT Hendrix. On a hybrid mpi-pthread approach for simplicial branch-and-bound. *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, p.p. 1764–1770. IEEE,

[Iba87] Toshihide Ibaraki. *Enumerative approaches to combinatorial optimization*. Baltzer,

[JS89] JM Jansen ir FW Sijstermans. Parallel branch-and-bound algorithms. *Future Generation Computer Systems*, 4(4):271–279,

[KPP+04] H. Kellerer, H.K.U.P.D. Pisinger, U. Pferschy ir D. Pisinger. *Knapsack Problems*. Springer Nature Book Archives Millennium. Springer, ISBN: 9783540402862. URL: https://books.google.lt/books?id=u5DB7gck08YC.

[KT88] GAP Kindervater ir HWJM Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research*, 33(1):65–81,

[LW84] Guo-Jie Li ir Benjamin W Wah. Computational efficiency of parallel approximate branch-and-bound algorithms. *International Conference on Parallel Processing*, p.p. 473–480,

[MP89] DL Miller ir JF Pekny. Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem. *Operations Research Letters*, 8(3):129–135,

[MP93] Donald L Miller ir Joseph F Pekny. Commentary—the role of performance metrics for parallel mathematical programming algorithms. *ORSA Journal on Computing*, 5(1):26–28,

[OEC+98] Diskrete Optimierung, Rainer E. Burkard Er, A C Ela, Rainer Burkard, Eranda Dragoti-Cela, Panos Pardalos ir Leonidas Pitsoulis. The quadratic assignment problem. *Handbook of Combinatorial Optimization*, DOI: 10.1007/978-1-4613-0303-9_27.

[PL90] Panos Pardalos ir XO LI. Parallel branch-and-bound algorithms for combinatorial optimization. *Supercomputer*, 7(5):23–30,

[PM90] Joseph F Pekny ir Donald L Miller. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. *Proceedings of the 1990 ACM annual conference on Cooperation*, p.p. 56–62. ACM,

[PŽ06] Remigijus Paulavičius ir Julius Žilinskas. Analysis of different norms and corresponding lipschitz constants for global optimization. *Technological and Economic Development of Economy*, 12(4):301–306,

[QSC+12] L. Quan, W. Shen, J. Cui ir D. Geng. Application of 0–1 knapsack mpi + openmp hybrid programming algorithm at mh method. *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*, p.p. 2452–2456, DOI: 10.1109/FSKD.2012.6234092.

[RG05] Ted K Ralphs ir Menal Güzelsoy. The symphony callable library for mixed integer programming. *The next wave in computing, optimization, and decision technologies*:61–76,

[Rou89] Catherine Roucairol. Parallel Branch and Bound Algorithms an overview. *Parallel Combinatorial Optimization*, 1(1):1–28,

[Sal02] Matthew J Saltzman. Coin-or: an open-source library for optimization. *Programming languages and systems in computational economics and finance*, p.p. 3–32. Springer,

[Tai93] Eric Taillard. Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285,

[TB92] HWJM Trienekens ir A Bruin. Towards a taxonomy of parallel branch and bound algorithms:1–23, URL: http://repub.eur.nl/handle/1491.

[TP96] Stefan Tschoke ir Thomas Polzer. Portable parallel branch and bound library user manual: library version 2.0,

[XRL+05] Yan Xu, Ted K Ralphs, Laszlo Ladányi ir Matthew J Saltzman. Alps: a framework for implementing parallel tree search algorithms. *The next wave in computing, optimization, and decision technologies*, p.p. 319–334. Springer,