KAUNO TECHNOLOGIJOS UNIVERSITETAS

ELEKTROS IR ELEKTRONIKOS FAKULTETAS

Paulius Dapkus

DUOMENŲ SURINKIMO TINKLAS ULTRAGARSINĖS TOMOGRAFIJOS SPRENDIMAMS

Baigiamasis magistro projektas

Vadovas
Prof. dr. Liudas Mažeika

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS
ELEKTROS IR ELEKTRONIKOS FAKULTETAS
TELEKOMUNIKACIJŲ KATEDRA

DUOMENŲ SURINKIMO TINKLAS ULTRAGARSINĖS TOMOGRAFIJOS SPRENDIMAMS

Baigiamasis magistro projektas
Išmaniosios telekomunikacijų technologijos (kodas 621H64001)

Vadovas
(parašas) Prof. dr. Liudas Mažeika
(data)

Recenzentas
(parašas) _____
(data)

Projektą atliko
(parašas) Paulius Dapkus
(data)

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS

Elektros ir Elektronikos

(Fakultetas)

Paulius Dapkus

(Studento vardas, pavardė)

Išmaniosios telekomunikacijų technologijos (kodas 621H64001)

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto „Duomenų surinkimo tinklas ultragarsinės tomografijos sprendimams“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 16 m. Gegužės 24 d.

Kaunas

Patvirtinu, kad mano **Pauliaus Dapkaus** baigiamasis projektas tema „Duomenų surinkimo tinklas ultragarsinės tomografijos sprendimams“ yra parašytas visiškai savarankiškai, o visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka) *(parašas)*

P. Dapkus. Duomenų surinkimo tinklas ultragarsinės tomografijos sprendimams: Telekomunikacijų inžinerijos magistro baigiamasis projektas / vadovas Prof. dr. Liudas Mažeika; Telekomunikacijų katedra, Elektros ir elektronikos fakultetas, Kauno technologijos universitetas. – Kaunas, 2016. – 81 p.

SANTRAUKA

Šiame darbe yra analizuojama ir optimizuojama sintetinės apertūros fokusavimo technologija, kuri yra plačiai naudojama moksliniuose tyrimuose ir plačioje rinkoje. Kaip ir daugelyje tokio tipo technologijų vienas iš pagrindinių trūkumų yra duomenų apdorojimas. Todėl šiam atvėjui yra pasiūlomas naujas ir greitas duomenų apdorojimo algoritmas, kuris užtikrina spartų sintetinės apertūros fokusavimo technikos generuojamų duomenų srauto apdorojimą ir atvaizdavimą pasitelkiant kombinuotą skaičiavimą - telekomunikacinius tinklus ir grafinio procesoriaus architektūrą. Tokios problematikos sprendimui tiriamos ir vėliau realizuojamos paskirstytų skaičiavimų sistemos.

Šio darbo tikslas yra ištirti esamus greito duomenų surinkimo ir apdorojimo sprendimus telekomunikaciniuose tinkluose ir pasiūlyti tokius, kurie leistų realizuoti ultragarsinės sintetinės apertūros fokusavimo technologiją realiame laike.

Šiame darbe atlikta sintetinės apertūros fokusavimo technikos spartinimo analizė, ištirti ir pasiūlyti paskirstytų skaičiavimų mašinų duomenų apsikeitimo modeliai, kurie funkcionuoja tinklinėje telekomunikacijų struktūroje, pasiūlytas kombinuoto skaičiavimo algoritmo sprendimas, kuris apjungia grafinį procesorių, centrinį procesorių ir duomenų išskirstymą telekomunikaciniame tinkle. Taip pat paskutiniame etape, remiantis atliktais tyrimais, realizuojama sistema.

Reikšminiai žodžiai: sintetinės, apertūros, fokusavimo, technika, paskirstyti, skaičiavimai, grafinėje, kortoje, tinklinėje, infrastruktūroje.

P. Dapkus. Data collection network for ultrasound tomography solutions: Final project of Telecomunications engineering master degree / supervisor Prof. dr. Liudas Mažeika; Department of Telecommunications, Faculty of Telecommunications and Electronics, Kaunas University of Technology. – Kaunas, 2016. – 81 p.

SUMMARY

In this work synthetic aperture focusing technique is analyzed and optimized, it is widely used in research and the broad market. As with many of this type of technology one of the main drawbacks is the data processing. In this case a new and fast data processing algorithm is found, which ensures rapid synthetic aperture focusing technique that generates data traffic processing and imaging using a combined count – telecommunication networks and graphics processor architecture. Solutions to such problems are investigated and later implemented in distributed computing systems.

The main goal is to examine the current speed of data collection and processing solutions for telecommunication networks and propose those, which would allow realization of the ultrasonic synthetic aperture focusing technique in real time.

In this work the synthetic aperture focusing technique acceleration analysis was carried out, investigated and proposed distributed computing machine data exchange models that function in networked telecommunications structure, also proposed combined calculation algorithm solution that brings together the graphic processor, CPU and data breakdown in telecommunication network. At the last stage of research the system was realized.

Keywords: synthetic, aperture, focusing, technique, distribute, calculations, graphic, card, network, infrastructure

TURINYS

Sutrumpinimų sąrašas 7
Įvadas 9
1. Duomenų surinkimo tinklo apžvalga ultragarsinės tomografijos sprendimams 11
1.1 Tinklinis skaičiavimo mašinų metodas 11
1.1.1 Tinklinių skaičiavimo mašinų veikimo principas 12
1.2 Hierarchinis duomenų formatas 13
1.3 Sintetinės apertūros fokusavimo metodas (SAFT) 14
1.4 Linijinės algebros akseleracija 21
1.5 Grafinio procesoriaus sąsaja 24
1.5.1 Heterogeniniai lygiagretūs skaičiavimai 25
1.5.2 Grafinės platformos modelis 26
1.5.3 Virtualūs skaičiavimo branduoliai 26
1.5.4 Grafinio procesoriaus atminties mainai 29
2. MPI modelių duomenų srautų tyrimas 31
2.1 Duomenų srauto modelis MST tinklinėje struktūroje 31
2.1.1 Tyrimo metodika ir parametrai 32
2.1.2 Redukcijos, išsibarstymo ir rinkimo modelis 33
2.1.3 Redukcinės-sklaidos modelis 37
2.1.4 Transliacijos modelis 39
2.1.5 Tinklelio rinkimo redukavimo modelis 42
2.1.6 *Visi visiems* modelis 45
2.2 Eksperimentiniai duomenys ir programinių įrankių pasirinkimas 48
2.3 Sintetinės apertūros fokusavimo algoritmas centrinio ir grafinio procesoriaus lygmenyje 49
2.4 Skaičiavimų paskirstymas tinklo mazguose 55
3. Kombinuoto skaičiavimo realizacija 60
3.1.1 Duomenų paskirstymo ir vaizdų atkūrimo mašina 62
3.1.2 Neapdorotų duomenų struktūrų skirstymas 62
3.1.3 Lygiagretaus skaičiavimo tinklinės mašinos 63
3.1.4 Lyginamieji laikai 63
Išvados 65
Literatūros sąrašas 66
Priedai 70
1. Priedas. Realizuotas SAFT skaičiavimo kodas, pagrindinis ciklinis skaičiavimas 70
2. Priedas. Pagrindinis vykdymo kodas 77
3. Priedas. Konfigūracijos ir eksperimentų vykdymo laiko saugojimas saugojimas 81

Sutrumpinimų sąrašas

GPU	Grafinis procesorinis įtaisas *(angl. Graphical processing unit)*
CPU	Centrinis procesorinis įtaisas *(*angl. *Central processing unit)*
RAM	Laikinoji atmintis *(*angl. *Random Access Memory)*
FPGA	Programuojama logika *(angl. Field-programmable gate array)*
ASIC	Specialios integracijos lustas *(angl. Application-specific integrated circuit)*
MPI	Žinučių apsikeitimo sąsaja *(angl. Message passing interface)*
SAFT	Sintetinės apertūros fokusavimo technika *(angl. Synthetic aperture focusing technique)*
FMC	Pilnutinės matricos gavimas *(angl. Full matrix capture)*
HDF5	Hierarchinis duomenų formatas *(angl. Hierarchical data format 5)*
API	Aplikacijų programavimo sąsaja *(angl. Application programming interface)*
SPMD	Viena programa, daug duomenų *(angl. Single program, multiple data)*
SIMD	Viena instrukcija, daug duomenų *(angl. Single instruction, multiple data)*
CSRC	Pridedamas šaltinis *(*angl. *Contributing source)*
DCT	Diskretinė kosinuso transformacija *(*angl. *Discrete cosine transform)*
DFT	Diskretinė Fourier transformacija *(*angl. *Discrete Fourier transform)*
GLBP	*Gateway* apkrovos balansavimo protokolas *(*angl. *Gateway Load Balancing Protocol)*
HD	Didelė raiška *(*angl. *High-definition)*
IP	Interneto protokolas *(*angl. *Internet Protocol)*
IPv4	Interneto protokolo ketvirtoji versija *(*angl. *Internet Protocol version 4)*
IPv6	Interneto protokolo šeštoji versija *(*angl. *Internet Protocol version 6)*
ITU-T	Tarptautinė telekomunikacijų sąjunga *(*angl. *International Telecommunication Union)*
JPEG	Fotografinių vaizdų išsaugojimo formatas *(*angl. *Joint Photographic Experts Group*
LAN	Vietinis multicast rezoliucijos vardas *(*angl. *Link Local Multicast Name Resolution)*
MAC	Multicast srities vardų struktūra (angl. *Multicast Domain Name System)*
OS	Operacinė sistema *(*angl. *Operating system)*
OSI	Srities vardų struktūra (angl. *Open Systems Interconnection)*
SSH	Saugaus prisijungimo protokolas *(*angl. *Secure shell)*
TCP	Persiuntimo kontrolės protokolas *(*angl. *Transmission Control Protocol)*

UDP	Vartotojų datagramų protokolas *(*angl. *User Datagram Protocol)*
MST	Mažiausios aprėpties medis *(*angl. *Minimum spanning tree)*
Qt	Integruota rogramavimo sąsaja (angl. Integrated Development Environment)
FMC	Pilnutinės matricos priėmimas *(angl. Full matrix capture)*

Įvadas

Nustatyti medžiagos savitas charakteristikas ar tridimensinį/dvidimensinį medžiagos grafinį išsidėstymą yra sudėtingas ir ilgas kelias, kuris reikalauja ypatingo tikslumo. Tokių problemų sprendimui yra naudojamos įvairios ultragarsinės tomografinio skenavimo technologijos, kuriami įvairūs metodai, kurie pasižymi tikslumu ir skenavimo greičiu. Vieni iš metodų yra neardomųjų bandymų tyrimai, kurie vaidina itin svarbią rolę inžinerinių struktūrų gamyboje ir aptarnavime [1—3]. Ultragarsiniai tomografiniai bandymai yra vieni iš labiausiai paplitusių neardomųjų bandymų tyrimų gamoje. Šie tyrimai leidžia nustatyti tūrinius pažeidimus. Todėl taip galima rasti struktūrų trūkumus, kurie yra ant paviršiaus, labai arti jo ir tuos, kurie yra struktūros viduje.

Vienas iš didesnių ultragarsinių tomografinių matavimų trūkumų yra santykinai dideli ultragarsinių bangų signalai, kurie būna kelių milimetrų eilės. Tai apriboja atkuriamų vaizdų rezoliucijos parametrus, taip darant įtaką norint nustatyti itin mažo dydžio defektus. Šiuos apribojimus galima panaikinti naudojant stipriai koncentruotas ultragarsines bangas, akustines linzes, fazuotas gardeles arba analizuojant neapdorotus ultragarsinius vaizdus, naudojant sintetinės apertūros fokusavimo technologiją *(angl. Synthetic aperture focusing technique (SAFT)).*

Šiame darbe yra analizuojama ir optimizuojama sintetinės apertūros fokusavimo technologija *(angl. Synthetic aperture focusing technique (SAFT)),* kuri yra dažniausiai naudojama moksliniuose tyrimuose ir rinkoje. Analizuojamoje technologijoje ketinama rasti trūkumų, kurie kaip ir daugelyje tokio tipo technologijų yra duomenų surinkimas ir apdorojimas. Tai atsitinka dėl to, nes skenuojamojo objekto vietos skaičiavimas reikalauja didelės duomenų pralaidos apdorojimo mašinai. Čia į pagalbą galima panaudoti įvairios paskirties telekomunikacinius tinklus ir tinkle sujungtas mašinas, kurios skaičiavimus gali atlikti lygiagrečiai. Sumodeliavus skaičiavimo mašiną, kuri gebėtų veikti tinkle, būtų galima sudaryti naujas skenavimo galimybes, kurios nulemtų atvaizdo kokybę atliekant aukštos rezoliucijos nuotraukas. Šiam atvėjui yra randamas naujas, optimizuotas ir greitas duomenų apdorojimo algoritmas, kuris užtikrintų spartų sintetinės apertūros fokusavimo technikos generuojamų duomenų srauto apdorojimą ir atvaizdavimą pasitelkiant kombinuotą skaičiavimą - telekomunikacinius tinklus ir grafines kortas. Tokios problematikos sprendimui tiriamos ir vėliau realizuojamos paskirstytų skaičiavimų sistemos.

Šio darbo tikslas yra ištirti esamus greito duomenų surinkimo ir apdorojimo sprendimus telekomunikaciniuose tinkluose ir pasiūlyti tokius, kurie leistų realizuoti ultragarsinės sintetinės apertūros fokusavimo technologiją realiame laike.

Tam, kad būtų pasiektas šis darbo tikslas, reikia atlikti šiuos uždavinius:

1. Atlikti sintetinės apertūros fokusavimo technikos analizę;
2. Ištirti ir pasiūlyti tinkamiausią ir optimaliausią paskirstytų skaičiavimų mašinų duomenų apsikeitimo modelį tinklinėje telekomunikacijų struktūroje;
3. Pasiūlyti kombinuoto skaičiavimo algoritmo sprendimą, apjungiant grafinį procesorių, centrinį procesorių ir duomenų išskirstymą telekomunikaciniame tinkle;
4. Realizuoti sistemą, remiantis atliktais tyrimais.

Darbas yra padalintas į tris dalis: 1) apžvelgiamos tinklinio skaičiavimo, SAFT ir programinio spartinimo technologijos; 2) tiriami telekomunikacinio tinklo duomenų paskirstymo metodai, pasiūlomi nauji modeliai ir optimizuojama sintetinės apertūros fokusavimo technologija anksčiau minėtam atvėjui, 3) remiantis pasiūlytais modeliais, realizuojama sintetinės apertūros fokusavimo technologija paskirstymui telekomunikaciniame tinkle.

1. Duomenų surinkimo tinklo apžvalga ultragarsinės tomografijos sprendimams

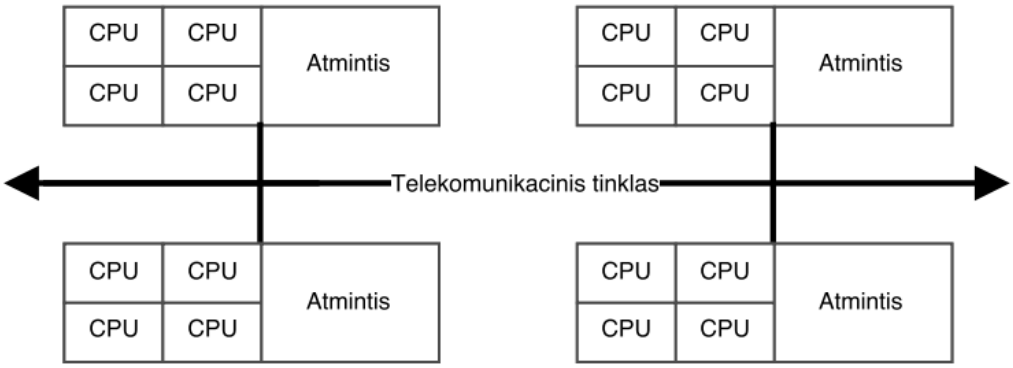
Šioje dalyje apžvelgiama bendrinė tinklinė tomografinė sistema, jos struktūros ir svarba. Analizuojami sprendimo metodai, atitinkamos priemonės norint išanalizuoti, ištirti ir pasiūlyti spartos atžvilgiu greitą tinklinį skaičiavimo modelį. Taip pat atlikti duomenų surinkimo ir apdorojimo sistemą. Įvykdžius apžvalgą sudaromi uždaviniai.

1.1 Tinklinis skaičiavimo mašinų metodas

Vienas iš iškelto tikslo sprendimų galėtų būti taikymas tinklines skaičiavimo mašinas. Tokiu metodu rekursyvūs SAFT ir ne tik skaičiavimai gali būti išskaidomi mašinų segmentams ir toliau surenkami.

Norint dalintis procesų skaičiavimo rezultatais tinkle, reikalinga didelės greitaveikos programinė biblioteka, viena iš jų yra atviro kodo *(angl. open Source)* programinė biblioteka MPI.

Augant lygiagrečių procesų ir paskirstytos atminties tinklams, išaugo poreikis turėti standartą, kuris būtų standartizuotas. Taip atsirado MPI standartas (1994 m.).



1.1 pav. Didelio našumo pranešimų perdavimo programinės bibliotekos (MPI) veikimo principas (tinklo infrastruktūra yra lanksti ir galimos šios technologijos Gigabit Ethernet,10 Gigabit Ethernet, InfiniBand, Myrinet, Quadric) [22]

Kiekvienas fizinis procesorius turi tik savo nuosavą atmintį ir sudaro atskirą tinklo mazgą, mazgai yra sujungiami tarpusavyje tinklu *(žr. 1.1 pav).* Jie gali būti sujungiami naudojant standartinius vietinės prieigos tinklus (pvz., *Gigabit Ethernet, 10 Gigabit Ethernet, InfiniBand, Myrinet, Quadric*), taip pat tinklas gali veikti įvairiose topologijose [14-19]. Standartas leidžia neadresuoti atminties tarp segmentų, kiekvienas skaičiavimo procesorius adresuoja tik savo fizinę atmintį.

Paskirstytos atminties programavimo įrankis teikia programuotojui tokias priemones:

- Paskirstyti skaičiavimus tarp tinkle sujungtų mašinų procesų [14];
- Organizuoti paskirstytų skaičiavimų duomenų mainus t.y. duomenys siunčiami įvairias modeliais [14].

MPI modelis šioje sistemoje yra išreiškiamas kaip paskirstytos atminties modelis ir jį naudojančios programavimo priemonės tokios kaip C/C++ programavimo kalba. Pagal šį komunikavimą, kiekvienas procesas turi tik savo vidinius lokaliuosius kintamuosius, kurie yra to proceso lokalioje atmintyje. Standarte nėra jokių bendrai įvardijamų kintamųjų t.y. nėra tarp procesorinės bendrųjų kintamųjų keitimo *(angl. data race)* problemos. Kai kiekvienas tinklo segmento procesas dirba tik jam priskirta atmintimi (fizinė RAM atmintis), tai jo atliekami atminties pakeitimai (pvz., kintamųjų reikšmių pakeitimai) nėra veikiami kitų procesų atminties regionų (pvz., jeigu tinklo segmentai turi tokius pat kintamuosius, tai jie vienas kitam įtakos neturi) [14-18]. Standartas taip pat nesuderina spartinančiųjų atmintinių (angl. Cache coherency), o tai yra didžiulė problema tokio tipo sistemose. Kai vienam iš procesų prireikia duomenų iš kito proceso, tuomet tai yra programinio modelio uždavinys. Duomenų kaita privalo būti programuojama t.y. kada ir kaip duomenys bus siunčiami ir gaunami. Sinchronizacija tarp lygiagrečiųjų procesų segmentų irgi yra programinio modelio dalis.

Įžvelgiami šie programinės bibliotekos privalumai:

1. Standartizuotas protokolas C/C++ ir fortran kalbos, kurios pakeitė kitus paskirstytos atminties programavimo įrankius ir bibliotekas [14-29];
2. Atviro kodo *(angl. OpenSource)* ir portabilumo privalumai. Standartą realizuojančios bibliotekos yra nemokamos (BSD licencija ar LGPLv2-LGPLv3) ir egzistuoja visose platformose. Todėl be didelių suderinamumo pakeitimų sistema gali būti perkelta iš vienos sisteminės architektūros į kitą sisteminę architektūrą [14-29];
3. Bibliotekos našumas. Programų kompiliavimas galimas specializuotų ir nestandartinių kompiliatorių pagalba, optimizuotomis atitinkamose platformose, o tai leidžia gerinti lygiagrečiųjų programų efektyvumą;
4. Funkcionalumas. Funkcijų gausa leidžia realizuoti ne tik bazines duomenų persiuntimo operacijas [1], bet ir sudėtingus tinklinius, kompleksinius, grupinius duomenų mainus. Reikia paminėti, kad šiuo metu jau yra sukurta nemažai įvairių aukštesnio lygio lygiagrečiųjų modelių, kurie remiasi išlygiagretinimu tinkle [14-29].

1.1.1 Tinklinių skaičiavimo mašinų veikimo principas

Lygiagreti programa rašoma C / C++ / Fortran kalba, naudojanti MPI funkcijas duomenų

mainams. Pagrindinės panaudojamos funkcijos/modeliai yra šie:

- Redukcinės, išskirstymo, rinkimo;
- Redukcinės – sklaidos;
- Transliacijos;
- Visi visiems;
- Tinklelio rinkimo-redukavimo.

Programa yra kompiliuojama panaudojant kompiliatorius su MPI (mpic++) arba nestandartiniais kompiliatoriais. Toliau yra sudaromas SSH traktų generavimas ir vykdomasis mašinų failas, kuris yra skaitomas pasirinktuose procesoriuose (branduoliuose). Programos paleidimas vykdomas naudojant MPI užduočių atlikimo aplinką (mpirun). Taigi, kiekvienas iš paleistų lygiagrečiųjų procesų vykdo tą patį programinį kodą tik su tam tikrais duomenimis. Duomenų apdirbimas yra vykdomas modelių pagalba. Visi veikiantys procesai gauna savo unikalų numerį – ID – *(angl. rank),* kurį kiekvienas procesas gali sužinoti vykdant MPI instrukcijas. Pagal ID numerį, procesai nustato (modelių pagalba) ir atlieka savo darbo dalį, naudodami jiems priskiriamą duomenų dalį. Toks lygiagrečiųjų skaičiavimų atlikimo būdas vadinamas - SPMD (Single Program, Multiple Data) [2].

Apibendrinus, metodas naudojamas paskirstyti rekursyvinius skaičiavimus yra gana senas, bet plačiai naudojamas ir jis yra vienas iš greičiausių atviro kodo *(angl. Open Source)* bibliotekų. Panaudojimas vykdomas gerai žinomomis programavimo kalbomis ir yra lengvai integruojamas į naujas architektūras.

1.2 Hierarchinis duomenų formatas

Pagrindinis šio formato tikslas yra atlikti neapdorotų duomenų spaudą tinklo segmentuose. Taigi toks metodas, surinkęs neapdorotus duomenis paskirstymo mašinoje, juos spaudžia ir paskirsto tinkle, o taip sutaupomas persiunčiamas duomenų kiekis tarp mašinų.

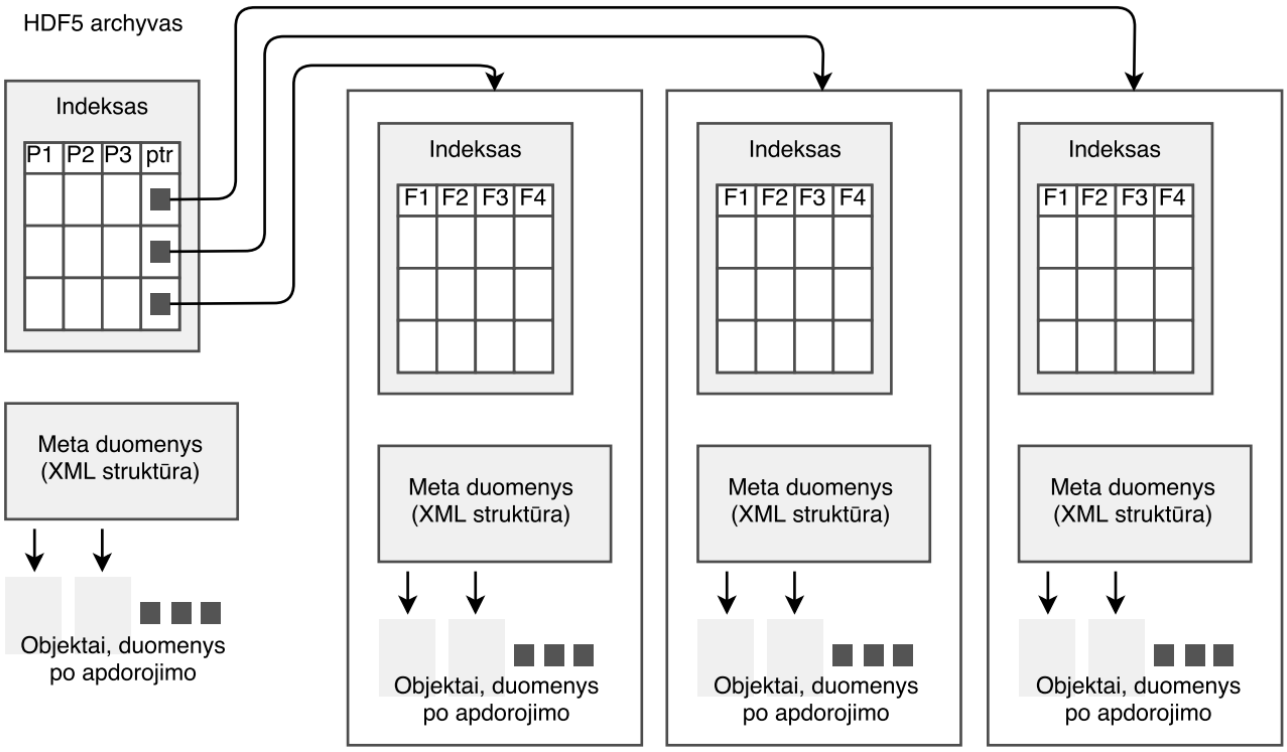
Hierarchinis duomenų formatas – HDF5 duomenų spaudimą ir indeksavimą gali atlikti lygiagrečiai paskirstydamas duomenis. HDF5 yra duomenų modelis [10-13], tai pusiau duomenų bazė, kuri saugo ir valdo duomenis. Šis standartas palaiko įvairių tipų duomenų standartus *(angl. Datatypes)* [10-13], kurie pateikti lentelėje *(žr. 1.1 lent.).* Šie duomenys gali būti atpažįstami HDF5 vidinių valdymo instrukcijų pagalba.

Lentelė 1.1. Hierarchinio duomenų formato tipai [10-13]

Tipas	Aprašymas
H5T_IEEE_F64LE	Aštuonių baitų, *little-endian* tipas, IEEE slankiojantis kablelis
H5T_IEEE_F32BE	Keturių baitų, *big-endian* tipas, IEEE slankiojantis kablelis
H5T_STD_I32LE	Keturių baitų, *little-endian* tipas, dvilypis sveikasis skaičius (*angl. signed complement integer*)
H5T_STD_U16BE	Dviejų baitų, *big-endian* tipas, teigiamos reikšmės sveikasis skaičius *(angl. unsigned integer)*
H5T_C_S1	Vienas baitas, nulinio prefikso simbolių masyvas su aštuonių bitų simboliais.
H5T_INTEL_B64	Aštuonių baitų laukas *Intel CPU* architektūrai
H5T_CRAY_F64	Aštuonių baitų laukas *Cray* slankiojančio kablelio skaičius
H5T_STD_ROBJ	Rodyklė į visą *HDF5* objektą failinėje struktūroje

Bazinis ir pagrindinis veikimo principas pateiktas paveiksle *(žr. 1.2 pav).*

Pagrindinė HDF5 paskirtis yra lankščiai ir efektyviai išspręsti didelės apimties duomenų saugojimo problemas [10-13]. Šis formatas yra plačiai naudojamas duomenų centruose, didelės spartos klasterinėse mašinose, taip pat fizikinių ir matematinių procesų skaičiavimų modeliavimui. Programinė bibloteka yra atviro kodo *(angl. Open source).*



1.2 pav. Hierarchinio duomenų formato veikimo struktūra

1.3 Sintetinės apertūros fokusavimo metodas (SAFT)

Sintetinės apurturos fokusavimo metodas yra vienas iš ultragarsinių neardomųjų matavimo

būdų, kuris dėka naudojamo signalų apdorojimo pasižymi padidintu lyginant su kitais metodais erdviniu skiriamumu. Jo esmė yra tame, kad ultragarsinės bangos virtualiai, tai yra po signalų matavimo, yra sufokusuojamos norimame tiriamo objekto taške. Atlikus tokią operacija visiems tiriamos objekto srities taškams (su pasirinktu žingsniu) gaunamas sąlyginai ryškus neveinalytiškumų (defektų) esančių objekte vaizdas. Keičiant parametrus galima gauti tipinius ultragarsiniams nerdomeisiems bandymams B-scan ar C-scan vaizdus. Gaunamas erdvinis skiriamumas yra sietinas su nudojamų ultragarsinių bangų ilgiu. Kadangi prieš SAFT apdorojimą galima pasirinkti bet kokias objekto sritis (aišku tik tose ribose kur buvio atliekami matavimai) šis metodas po duomemų sukaupimo leidžia pakanakamai detaliai tiri objektą. Lyginanat su kitais ultragarsiniai neradomaisiais metodais, pavyzdžio panaudojant fazuotas ultaragarsines gardeles SAFT budų gaunamas skiriamumas yra geresnis. Metodo trūkumas, kad didelį objektą gaunamas didelis signalų kiekis ir SAFT apdorojimas formuojant vaizdus gali užtrukti ilgai. Todėl šiame darbe ir tiriami metodai kaip galima būtų kaupiant duomenis lygiagrečiai vykdyti ir apdorojimą, tuo būdu iš esmės sutrumpinti vaizdū gavimom laiką.

Kiti neardomiejo metodai, pasižymintys aukštu erdviniu skiraumu naudoja *CT* skenavimo technologiją, kuri paremta rentgeno spidulių principu. Skenavimo pagalba gaunami didelio ryškumo tomografiniai vaizdai. Šis būdas labai sudėtingas, be to rengneno spindulių pavojingumas riboja platų jo panaudojimą. Šie metodai nebus nagrinėjami šio darbo kontekste.

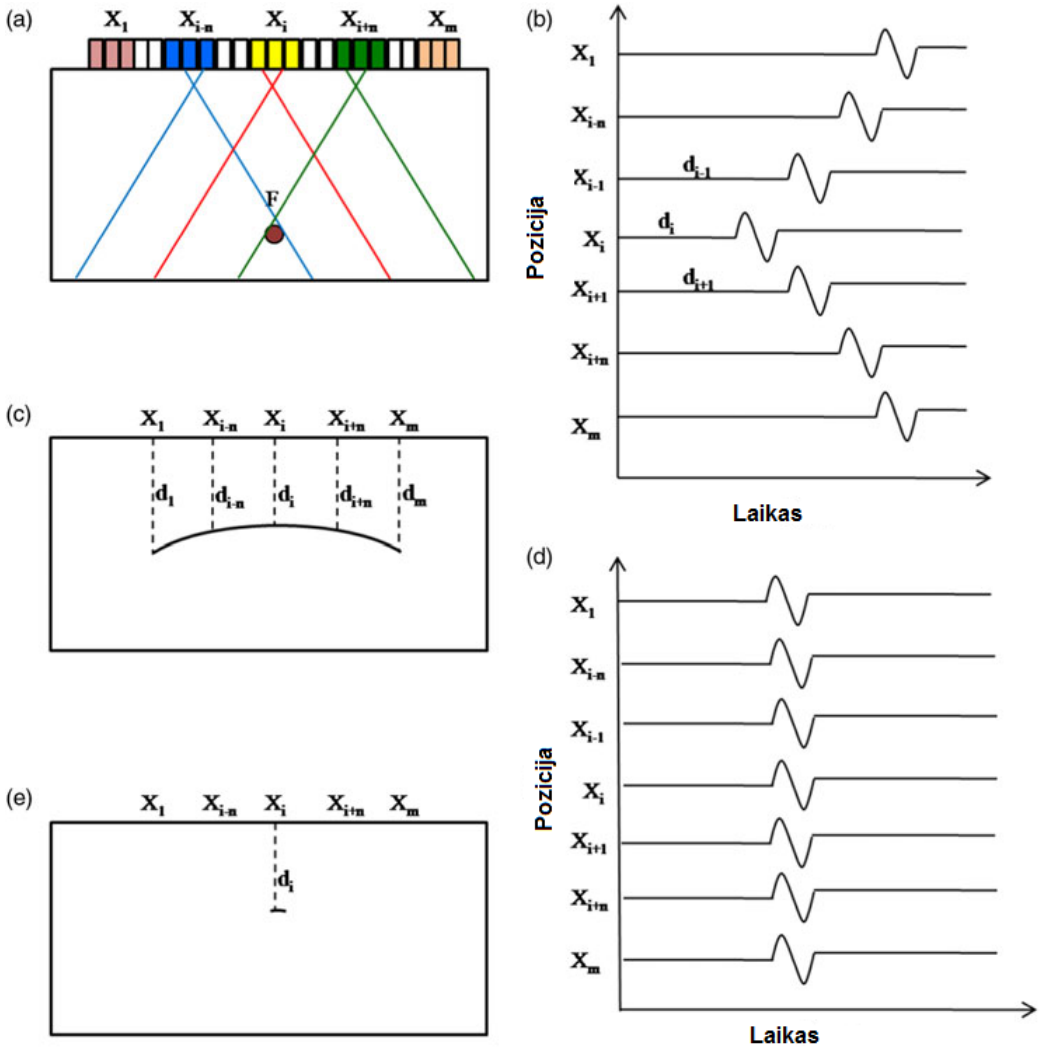
Apžvelgiamos dvi pagrindinės SAFT metodikos, tai SAFT naudojantis sukauptą duomenų masyvą ir paprasta SAFT metodas naudojant fazuotas gardeles.

Integruotas priėjimas panaudojant fazuotas gardeles ir SAFT yra paremtas B-scan vaizdo apdorojimu. Pagal šį metodą priimama, kad kiekvienas išsignalų yra gautas iš vieno gardelės elemento, spinduliuojančio besiskleidžias ar difraguojančias bangas. Toks gardelės elementas dėl sąlyginai mažų matmenų turi labai plačia kryptingumo diagramą. SAFT metodo efektyvumas ar erdivinios skiriamumas labai priklauso nuo tiek nuo ultagarsinių bangų šaltinio kryptingumo diagramos pločio tiek nuo atspindėto signalo kryptingumo. Kitai žodiais tariant atkurtų vaizdų ryškumas bus juo didesni juo didesniu kampu bus matomas objekte tiriamas nevienalytiškumas. Tyrimams mažame gylyje dažnai naudojama ne visa gardele bet tik dalimi elementų [2].

Skirtumas tarp įprasto fazuotos gardelės darbo ir SAFT yra tame kad įprasto režimo atveju gardelė fokusuojama tam tikrame gylyje, ne visoje srityje. Todėl ganamų vaizdų raiška taip pat labai netolygi. Tačiau šio metodo proivalumas, kad matavimai mvykdomi faktiškai reliame laike. SAFT atveju vaizdai gaunami po signalų sukaupimo, tačiau tuo atveju analizė gali būti ženkliai detalesnė. [1].

Paveiksle schematiškai pateikta SAFT defektų detektavimo metodologija naudojant

ultragarsines fazuotas gardeles*(žr. 1.3 pav.).*



1.3 pav. SAFT metodologija naudojant fzuotas gardeles: (a) duomenų įgijimas; (b) neapdoroti signalai, (c) stilizuotas B-scan vaizdas preiš apdorojimą; (d) signalai sustumdyti laike pagal SAFT (e) stilizuotas B-scan vaizdas po apdorojimą [1]

Linijinė gardelė su *L* elementų skaičiumi yra pateiktas kaip pavyzdys. Tegu *k* būna aktyvių elementų skaičius, skirtas fokusavimui. Fokusavimas atliekamas 2 mm nuo paviršiaus[1]. Gardelei iš 64 elementų ir 0.6 mm žingsnio (jei apertūra iš šešių aktyvių elementų yra panaudojama fazuoto masyvo SAFT), iš gautų rezultatų galima matyti, jog B-tipo vaizdai sukurti elektroninio skenavimo metu turės signalus iš 59 aktyvių apertūrų, kurios yra išskirstytos po 0.6 mm. Sakykime, pateikiamas taškinis reflektorius *F (žr. 1.3 pav),* kuris yra tiksliai po aktyvia apertūra *Xi* (*i* yra sveikas skaičius, kuris yra $1 < i < m$), kuriai yra vykdoma SAFT procedūra. Tegu *di* būna pažeidimo *F* atstumas nuo paviršiaus [3], kur *Xi* yra fiksuota. Pažeidimas *F* yra taip pat detektuojamas gretimųjų apertūrų *Xi-n, ..., Xi-2,Xi+1Xi+2,...,Xi+n*, kur n yra sveikasis skaičius, kaip $1 \leq (i-n)$ *ir* $(i+n) \leq m$. Atskyrimas tarp apertūrų *Xi* ir *Xi-n* yra išreiškiamas iš $s = n \times p$. Tegu *dn* išraiška išreiškiama pažeidimo *F* atstumo nuo tarpinės apertūros *Xi-n*. Atstumas *dn* gali būti išreikštas[1]:

$$t_n = 2 * \frac{[d_i^2 + (np^2)]^{\frac{1}{2}}}{V}, \quad (1)$$

, kur *V* yra ultragarsinių bangų greitis bandinyje.

Kadangi *dn* > *di* (arba *tn* > *ti*, kur *t* yra laikas, kurį keliauja banga iki defekto ir atgal), signalas nuo pažeidimo *F* pasirodo didesniame atstume arba sklidimo laikas nuo apertūros, kuri yra bet kurioje apertūros *X*i pusėje[1-3]. Schematiškai tai parodyta paveikslo *b* dalyje *(žr. 1.3 pav).* Dėl bangų difrakcijos efekto [3], pažeidimas taške *F* padidins virtualų signalą stilizuotame B-tipo vaizde, kuris pateiktas paveikslo *c* dalyje *(žr. 1.3 pav).*

Aktyvus apertūros dydis gali būti išreikštas kaip *A = k*x*p*, kur p yra žingsnio elementas [3]. Ši apertūra yra valdoma elektroniškai per visą masyvo ilgį ir sugeneruojant B-skenavimo vaizdą. Visas aktyvių apertūrų skaičius (m), panaudotas šio vaizdo atkūrimui yra:

$$m = (L - k) + 1. \quad (2)$$

Vykdant SAFT procesus, iš pažeidimo F signalas, priimtas visų apertūrų, yra laikiškai pastumiamas per ts, kaip *ts = tn – ti* [3]. Laiko postūmis leidžia sugeneruoti A-tipo signalą iš individualių apertūrų pateiktų paveiksle *(žr. 1.3 pav. d dalis).* Visi A-tipo signalai yra susumuojami ir suvidurkinami, o A-tipo signalų integravimo rezultatų aplitudė yra atvaizduojama B-tipo vaizde [6]. Vykdant sumavimo procesus, yra svarbi A-tiposignalų interferencija, kuri padidina signalo amplitudę pažeidimo *F* vietoje. Kuomet panašios laiko postūmio, sumavimo ir vidurkinimo procedūros yra atliekamos taškui, kuriame nėra pažeidimo F, sumuojamų signalų fazės nesutampa ir inegralinio signalo amplitudė sumažėja. Šios interferencijos reazultate gaunamas virtualaus fokusavimo efektas [6]. Galutinis rezultatas po SAFT apdorojimo yra pateiktas paveiksle *(žr. 1.3 pav. e dalis).*

Linijinė gardelė, kurio dažnis 10MHz, turint elementų su 0.3mm žingsniu, reikia fokusuoti 2mm gylyje [3]. Tada aktyvios apertūros parinkimui artimos zonis ilgis (N), kuris yra didesnis nei reikalaujamas židinio ilgis (F) [3]. Linijiniai gardeliai dirbant dažniu *f*, žingsnį *p* ir ortoganaliuosius matmenis *b*, kur *k* elementų skaičius artimo zonos ilgis *N* skaičiuojamss pagal [3]:

$$N = \frac{cA^2 f}{4V}, \quad (3)$$

čia *c* – konstanta, priklausant nuo kraštinių santykio (kraštinių santykis = *A/b*, *c* ≈1 kraštinių santykiui < 0.5), A yra aktyvioji apertūra *(k x p)* ir *V* yra garso greitis medžiagoje [3].

10 MHz, 0.3 mm žingsnio, 5 mm stačiakampiai linijoiniai gardeliai, kada aštuoni aktyvūs elementai yra naudojami aliuminiui tirtio (a = 2.4mm, c = 1, V = 6.3 mm/μs), *N* reikšmė tampa 2.3 mm, kuri tenkina *N* ir 5 Mhz, 0.6mm žingsnio 10 mm ortogonalių matmenų tiesiniam masyvui,

aktyvių elementų skaičius [2]. Tokią metodiką tenkinantis kreiterijų siekia 6 [2].

Vienas iš programinių įrankių skirtų darbui su fazuotomis gardelėmis įvairiose aplinkose yra CIVA 11 [7,8]. CIVA yra daugiafunkcinis ultragrsinių neardomųjų bandymų metodų modeliavimo paketas, sukurtas CEA, Prancūzijoje.

Ultragarsinis CIVA modulis gali skaičiuoti ultragarsinių bangų lauką pasirinktoms medžiagoms ar jutiklio konfigūracijai ir tuo pat metu gali būti panaudotas nustatyti defekto atsaką [7,8].Modliavimai gali būti atliekami tipinias nardomiesiems bandymams parametrams, pvz.: bangos dažnis 10 MHz, žingsnis 0.6 mm, 128 elementai tiesiniame jutiklių masyve [7].

Autorių teigimu, modeliavimo rezultatai parodo ultragrasinio spindulio profilį [7]. Kuomet aštuoni aktyvūs elementai yra panaudoti be židinio efekto, garso banga yra gerai nukreipta į artimojo lauko tašką (maždaug 3mm nuo paviršiaus [7]) ir dėl to jis ima diverguoti. Spindulio divergencijos kampas yra maždaug ±10º nuo paviršiaus normaliosios padėties. Tais atvejais, kada šie aštuoni elementai yra panaudojami fokusuoti garso bangą 2 mm atstume, divergencinė garso banga yra už fokusinio taško, kuris yra gyvybiškai svarbus SAFT procedūrų vykdymui. Maksimalaus garso spaudimo taškas taip pat priartėja prie skenavimo paviršiaus. Spindulio divergencijos kampas šiuo atveju yra maždaug ±20º [7].

Su konvenciškai fazuotu gardele, maksimalus gylis iki kurio garso banga gali būti fokusuojama yra ribojama artimojo atstumo ilgio [3], kuris yra nustatomas pagal aktyvios apertūros dydį duotajam tiesiniam jutikliui.

Dirbant su fazuota ultragarsine gardele ir naudojant SAFT faktiškai nėra problemų dėl fokusavimo gyliui [1], todėl to galima naudoti ženkliai mažesnę aktyvią apertūrą, lyginant su įprastu fazuotos gardelės darbu. Kada lyginama su FMC, fazuoto masyvo SAFT pateikia atskirus privalumus, duomenų dydžio atžvilgiu [1].

Pavyzdžiui, kada 128 elementai yra panaudoti ir duomenys yra priimami iš 2440 duomenų taškų (skirtas 1 baitas kiekvienam), tada panaudojant FMC, rinkmenos dydis galėtų būti arti 39 MB (128 x 128 x 2440) [3].

Panaudojant šešis aktyvius fazuoto masyvo SAFT elementus, rinkmenos dydis identiškomis sąlygomis būtų 0.29 MB (128 x 2440), o tai yra ženkliai mažiau nei FMC technikos atveju [3].

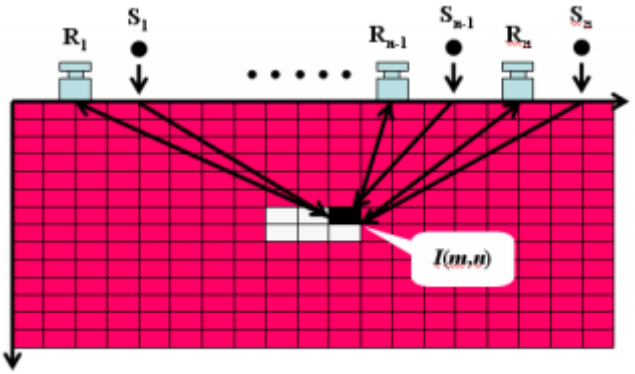
Lyginant su FMC galima pasiekti rezultatus, leidžiančius lyginti PA-SAFT, įgaubtai fazuotai gardeliai, kur reikia didelio skaičiaus aktyvių elementų (skirtų siųsti ir priimti garsinius spindulius) [2]. Galima sakyti, kad efektyvus sintetinio fokusavimo panaudojimas užtikrina geresnę horizontalųjį skiriamumą.

Autoriaus eksperimentai [2] parodo, kad fazuotos apertūros SAFT nėra labai efektyvus neefektyvus ieškant trūkių viršųnių. Ši technologija gali būti panaudojama tiksliam plyšiuose esančių defektų įvertinimui [2].

Bendrai sintetinės apertūros fokusavimo technika buvo sukurta siekiant įveikti apribojimus, kurie pasireiškia gamybinėse sąlygose ir kontroliuojant dideles apertūras padidinat grafinių atvaizdų rezoliuciją [1,9] *(žr. 1.4 pav.).* Fundamentali SAFT matematinė išraiska pateikta žemiau [1]:

$$I(m,n) = \frac{1}{N}\sum_{i=1}^{N} T_i(t_i), \quad t_i = \frac{|S_i G(m,n)|+|G(m,n)R_i|}{C_P}, \qquad (4)$$

kur *G(m,n)* - tinklas iš *m*-tūjų eilučių ir *n*-tūjų stulpelių, *N* - skaičiavimų atlikimų skaičius, C_P išilginės bangos greičio daugiklis.



1.4 pav. Priimimo ir išsiuntimo schema. Operacijų nustatymo objekte panaudojant SAFT [1]

SAFT yra po-procedūrinė vaizdų atkūrimo technika [1], kuri skirta priimti sufokusuotą garsinį spindulį medžiagoje nenaudojant akustinių lęšių. Panaudojant SAFT, kiekvienai jutiklio pozicijai, neapdoroti duomenys, kuriuose garso banga yra išsklaidyta defekto yra sąlygotos diverguojančiojos ultragarsinio spindulio [3]. Vykdant rekonstravimo procesą, garso bangos yra pastumtos laike arba analizuojamos kartu su išsklaidymo šaltiniu ir suvidurkinamos panaudojant bangų daugybos formules [3]. Signalo amplitudė defekto vietoje yra ženkliai padidėja, vykdant vidurkinimo procesus, o laikinio postūmio metu kiekviename momentiniame postūmyje atitinkamoje fazėje yra surenkamos interferuotos garso bangos.

Lyginant su kitomis vietomis, prasta signalo fazės koreliacija sukelia likviduojančiąją garso bangų interferenciją, taip ženkliai sumažinant signalo amplitudę [3,6]. Ši technika pasižymi aukšta šonine rezoliucija ir pagerina ultragarsinių vaizdų kontrastą eliminuojant išorinius triukšmus arba foninio sklaidymo triukšmus, gautus iš šiurkščių paviršiaus struktūrų.

SAFT metodo efektyvumas priklauso nuo keleto veiksnių:

1. Diverguojančios garso bangos kampo [3];

2. Žingsnio dydžio, kuriame duomenys yra tinkami apdorojimui [3].

SAFT apdorojimui, defektas turi būti artimojoje zonoje, kuri yra nusakoma atstumu nuo jutiklio paviršiaus iki kur garso banga sklinda. Dideli garso bangos skleistės reikalavimai SAFT metodui yra realizuojami panaudojant mažesnio elemento keitiklius [5]. Mažesnis elementas taip pat reiškia trumpesnį artimos zonos ilgį, vadinasi, yra labai mažas medžiagos sluoksnis iki tiriamo paviršiaus, kur SAFT metodika nėra efektyvi.

Sintetinis fokusavimas padidina šoninę rezoliuciją, kuri tampa teoriškai tokia pat kaip ir tiriamos medžiagos gylis. Signalo ir triukšmo santykis yra pagerintas vykdant šį procesą [2]. Šoninė rezoliucija, kuri teoriškai gali būti pasiekta su SAFT yra D/2, kur D yra keitiklio aktyvios zonos diametras. Tačiau, praktikoje yra mažiausio kristalo riba, kuri gali būti pagaminta reikiamam dažniui. Tai nustato praktiškai pasiekiamą fokusavimą panaudojant SAFT [4].

Sintetinės apertūros fokusavimo technika yra panaudojama kuriant vaizdus iš ultragarsinių skenavimo duomenų. SAFT tradiciškai naudojamas vaizdų atkūrimui vienalytėje terpėje *(angl. Single medium),* bet neseniai pasirodęs fazinio postūmio migracijos *(angl. Phase shift migration – PSM)* algoritmas išplėtė SAFT panaudojimo galimybes iki daugiasluoksnių struktūrų [6].

Straipsnyje pristatomas panašus fokusavimo algoritmas, kuris yra pavadintas daugiasluoksniu Omega-K (MULOK). Jis apjungia PSM ir W-K algoritmą, todėl galima efektyviau vykdyti daugiasluoksnį skenavimą [6]. Asimptotinis išsibarstymas MULOK atveju yra mažesnis lyginant su PSM ir tai patvirtina proceso vykdymo laikų analizė. Panaudojant eksperimentinius duomenis, gautus iš daugiasluoksnės struktūros, parodoma, kad lyginant vaizdo kokybę, iš esmės nėra skirtumo tarp dviejų naudojamų algoritmų [6].

Straipsnyje [5] rašoma apie defektų detekciją po lygiu arba banguotu paviršiumi – panaudojamas *(angl. Time Reversal Mirror)* metodas. Technika pagrįsta *(angl. Time Recersal)* ultragarsinių laukų koncepcija, kuri apima fazės ir amplitudės informaciją, ateinančią iš defekto [5]. Ši technika yra save adaptuojanti, o tai leidžia nustatyti defektą kietoje medžiagoje. Stipriai išsibarsčiusioje medžiagoje, SAFT metodas leidžia mažinti struktūrinį triukšmą.

Rezultatai [5] gaunami su 121 *(angl. Time reversal)* veidrodžiu tiriant titato ir duraliuminio pavyzdžius. Autoriai demonstruoja Time reversal metodikos galimybę, kuri leidžia kompensuoti netolygumus, esančius skystis-kieta medžiaga terpėse, esant skirtingiem jų matmenims, detektuojant smulkius defektus.

Išvadose [5] pateikiama, kad TRM sudaro tinkamas sąlygas pagerinant pavyzdžių tyrimus su dideliu ultragarsiniu šlakiniu triukšmo lygiu. Visi eksperimentai rodo TRM efektyvumą siekiant detektuoti defektą komponente, kuris yra lygus arba banguotas. Panaudojant šią techniką, yra

pagerinamas tyrimo signalo-triukšmo santykis:

1. TRM procesas yra save adaptuojantis, kuris leidžia kurti sufokusuotą bangą, sutampančią su defekto forma, medžiaga, kuria sklinda banga ir jų (medžiagos ir veidrodžio) geometrija [5].

2. TRM sumažina struktūrinio triukšmo amplitudę. Dėl informacijos praradimo atliekant bangos skleidimą, procesas negali būti atliekamas mikrostruktūroje, kurios matmetys yra maži, lyginant su bangos ilgiu [5].

Pirmieji eksperimentai [5] buvo vykdyti su TRM veidrodžio prototipu, veikiančiu 5MHz dažniu, kuris leidžia detektuoti 0,4mm diametro plokščią skylę, esančią 65mm gylyje, triukšmingame pažeistame titano komponente, signalo-triukšmo santykiui esant 30dB.

Kitame eksperimente [5] demonstruojama galimybė panaudoti TRM mažų defektų detekcijai. Panaudojant TRM, C-skenavimo vaizdai gali būti realizuoti realiame laike.

Sistemoje autoriai gauna visišką TRM seką (trijų iteracijų procese) ir vykdo signalo apdorojimą [5]. Iš viso procedūra užtrunka 40ms. Straipsnyje [5] pabrėžiama, kad rezultatai gali būti pritaikyti pramonėje.

1.4 Linijinės algebros akseleracija

Siekiant užtikrinti greitaveiką, yra svarbu atsižvelgti į programinę skaičiavimo biblioteką. Kadangi standartinės matematinės bibliotekos nėra optimizuotos didelių duomenų masyvams, tam yra naudojamos atskiros programinės bibliotekos, kurios yra optimizuotos dideliems masyvams, o taip pat suderintos su tokiomis sąsajomis kaip:

- *OpenCL*. Grafinio procesoriaus programinė biblioteka;
- MPI. Didelio našumo pranešimų perdavimo programinė biblioteka;

Tam tikslui buvo analizuotos šios linijinės matematinės algebros programinės bibliotekos:

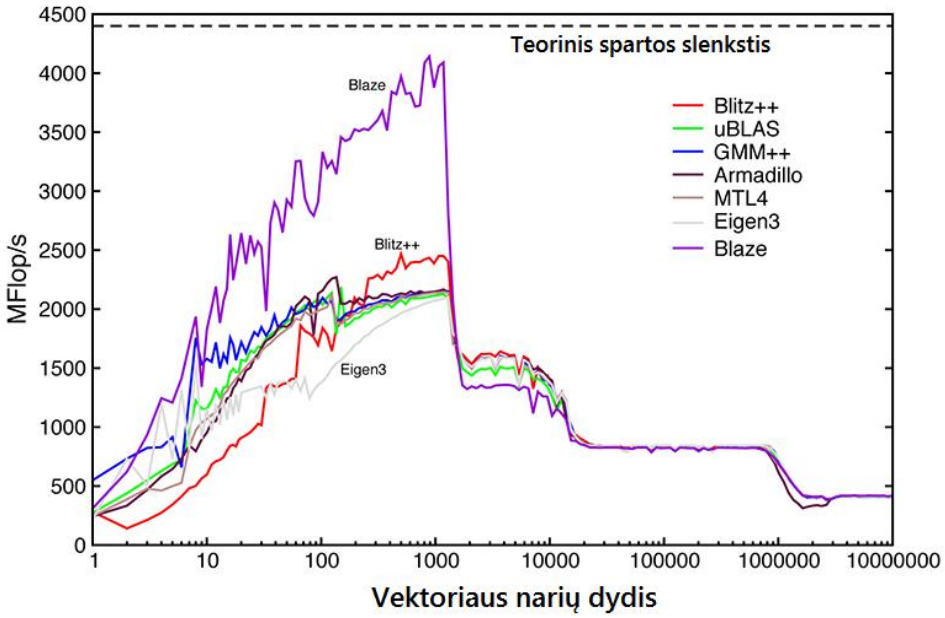
- *Armadillo;*
- *MAGMA;*
- *Intel MKL;*
- *Blaze-lib;*
- *Blitz++;*
- *Boost uBLAS;*
- *GMM++;*
- *MTL4;*
- *Eigen3;*

Dauguma jų atlieka kompleksinius matricinius, vektorinius veiksmus, rūšiavimus [31]. Bibliotekos skiriasi savo skaičiavimo sparta, funkcijų gausa [30] ir pritaikomumu.

Bibliotekos kūrėjų atliktas programinių paketų tyrimas [30], kuriame atskiros bibliotekos yra lyginamos tarpusavyje. Tyrimas susidėjo iš tankių vektorinių sudėčių ir daugybos. Toliau pateikiamas aprašytas programinis tankaus vektoriaus sudėties testavimo kodas [30, 31]:

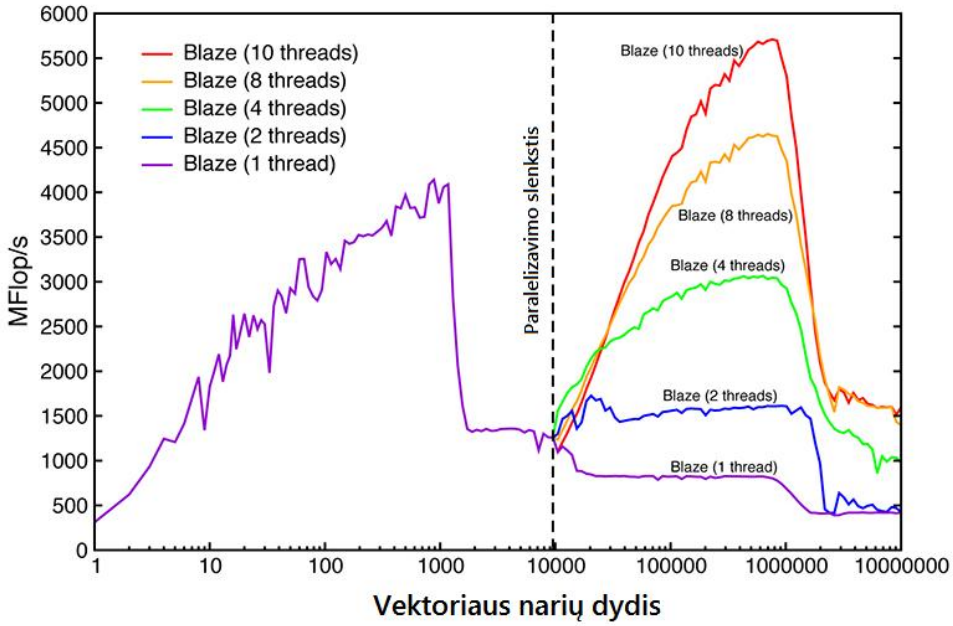
```
blaze::DynamicVector<double> a( N ), b( N ), c( N ); // slankiojancio kablelio kintamasis
// ..... Vektoriaus inicializacija ir kintamųjų suteikimas
c = a + b; // tankaus vektoriaus sudėtis
```

Gauti programinių bibliotekų lyginamieji grafiniai rezultatai [30, 31], kurie pateikiami paveikslėlyje *(žr. 1.5 pav).*



1.5 pav. Tankaus vektoriaus narių dydžio skaičiavimo sparta naudojant matematinę sudėtį [30]

Naudojant *Blaze-lib* integruotą virtualių procesorių galimybę, galimas ženklus procesų paspartinimas *(žr. 1.6 pav) [30, 31]*.

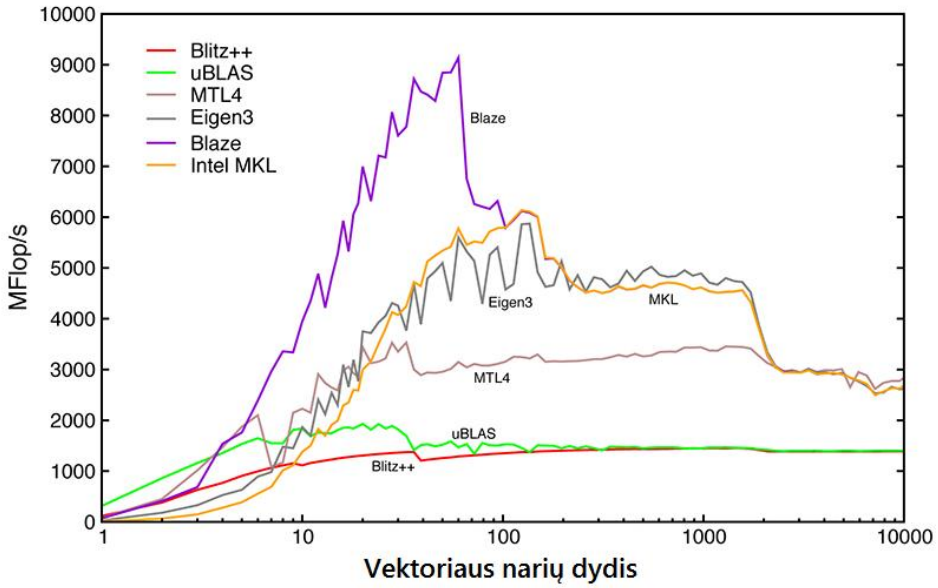


1.6 pav. Tankaus vektoriaus narių dydžio skaičiavimo sparta naudojant matematinę sudėtį. Blaze-lib daugiabranduolinio procesoriaus išnaudojimo galimybės [30]

Pateikiamas [30, 31] aprašytas programinis tankaus vektoriaus daugybos su dinaminės matricos testavimo kodas:

```
blaze::DynamicMatrix<double,rowMajor> A( N, N ); // slankiojancio kablelio kintamasis
blaze::DynamicVector<double> a, b; // slankiojancio kablelio kintamasis
// ..... Vektoriaus inicializacija ir kintamųjų suteikimas
b = A * a; // tankaus vektoriaus daugyba
```

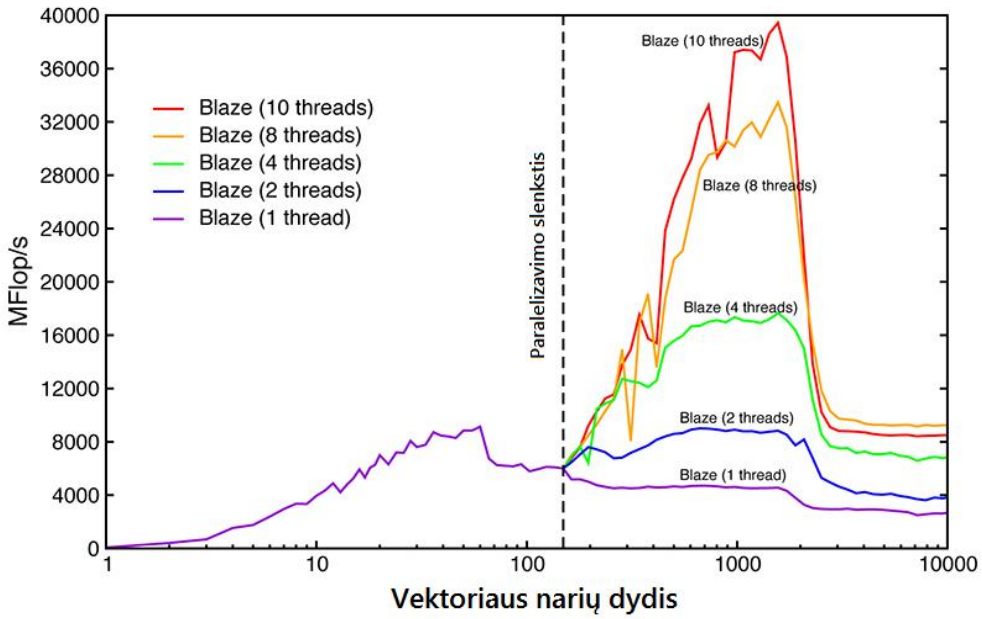
Gauti grafiniai rezultatai pateikiami paveikslėlyje *(žr. 1.7 pav).*



1.7 pav. Tankaus vektoriaus ir dinaminės matricos narių dydžio skaičiavimo sparta naudojant

matematinę sudėtį [30]

Naudojant *Blaze-lib* integruotą virtualių procesorių galimybę, galimas ženklus procesų paspartinimas *(žr. 1.8 pav) [30, 31]*.



1.8 pav. Tankaus vektoriaus narių dydžio ir dinaminės matricos skaičiavimo sparta naudojant matematinę sudėtį. Blaze-lib daugiabranduolinio procesoriaus išnaudojimo galimybės [30]

1.5 Grafinio procesoriaus sąsaja

Modeliuojant duomenų apdorojimo sistemą, kurios skaičiavimo moduliai būtų nepriklausomi nuo aparatinės dalies t.y. grafinių kortų yra viena iš pagrindinių programinių bibliotekų - *OpenCL*. Labiausiai išplėtotos programinės bibliotekos:

- *CUDA*. Lygiagrečių skaičiavimų biblioteka, skirta Nvidia grafinėms plokštėms;
- *OpenCL*. Lygiagrečių skaičiavimų biblioteka, skirta tiek AMD grafinių kortų plokštėms, tiek Nvidia.

CUDA biblioteka turi nedidelį pranašumą lyginant su *OpenCL* biblioteka *(žr. 1.2 lent)*, bet turi ir didelį trūkumą, kadangi *CUDA* palaiko tik *Nvidia* gaminamas plokštes, o AMD plokštės nėra palaikomos [4]. Atsižvelgiant į tai, kad buvo siekiama, jog sistema būtų nepriklausoma nuo procesorių gamintojų, buvo analizuojama OpenCL biblioteka.

OpenCL yra atvira ir nemokama lygiagreti skaičiavimo API [5], sukurta tam, jog skirtingiems grafiniams procesoriams ir kitiems procesoriams būtų galima dirbti kartu su CPU, ir taip tokiu būdu sukuriant papildomą skaičiavimo galią. *OpenCL* 1.0 standartas buvo išleistas 2008 metais gruodžio 8 dieną [5], nepriklausomo standartų konsorciumo.

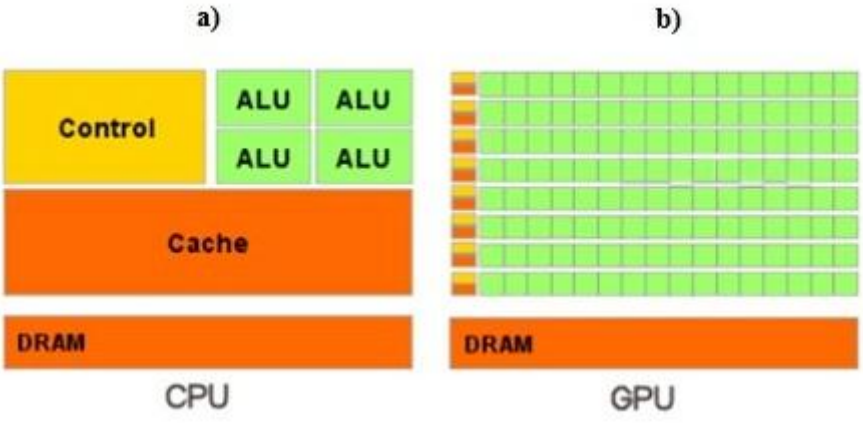
Lentelė 1.2. CUDA ir OpenCL algoritmų praktinis palyginimas [5]

Qubitai	Kernelio skaičiavimo trukmė				Duomenų persiuntimo trukmė			
	CUDA		OpenCL		CUDA		OpenCL	
	Vidurkis	stdev	Vidurkis	stdev	Vidurkis	stdev	Vidurkis	stdev
8	1.96	0.027	2.23	0.004	0.009	0.007	0.011	0.007
16	3.85	0.006	4.73	0.013	0.015	0.001	0.023	0.008
32	7.65	0.007	9.01	0.012	0.025	0.01	0.039	0.01
48	13.68	0.015	19.8	0.007	0.061	0.01	0.086	0.008
72	25.94	0.036	42.17	0.085	0.106	0.006	0.146	0.01
96	61.1	0.065	71.99	0.055	0.215	0.009	0.294	0.011
128	100.76	0.527	113.54	0.761	0.306	0.01	0.417	0.007

Kūrėjai siekia padalinti skaičiavimo problemas skirtingoms procesorių sistemoms. Taip sudaroma galimybė naudoti GPU kaip papildomą skaičiavimų įrenginį kartu su CPU. Heterogeninio skaičiavimo modelio apribojimai yra tuomet, kada programuotojai gali rinktis tik standartines programavimo kalbas, taip apribojant jų galimybę naudoti kitas atviras kalbas. Pavyzdžiui, ribojant *Nvidia CUDA* programinės bibliotekos aparatinės įrangos pasirinkimą, todėl norint eksplotuoti aplikaciją kitoje sistemoje, tenka pertvarkyti visą sistemą. *OpenCL* to nereikia.

1.5.1 Heterogeniniai lygiagretūs skaičiavimai

Heterogeniniai skaičiavimai apima įvairių tipų skaičiavimo įrenginių panaudojimą. Skaičiavimo įrenginys gali būti bendros paskirties apdorojimo įrenginys, pavyzdžiui, CPU, grafinis procesorius arba specialios paskirties procesoriai, tokie kaip FPGA. Seniau, dauguma kompiuterinių aplikacijų naudojo jau procesoriuose įdiegtas technologijas. Modernėjant programinei įrangai, atsiranda reikalavimas bendrauti su atskiromis sistemomis (pvz. garso/vaizdo apdorojimo, tinklo įranga ir kt.), net pažangi CPU technologija nėra stipri šiam našumo nepakankamumui. Tam jog būtų pasiekta didesnė greitaveika ir sistemą būtų galima integruoti į heterogeninę architektūrą yra naudojama specializuota aparatinė įranga.



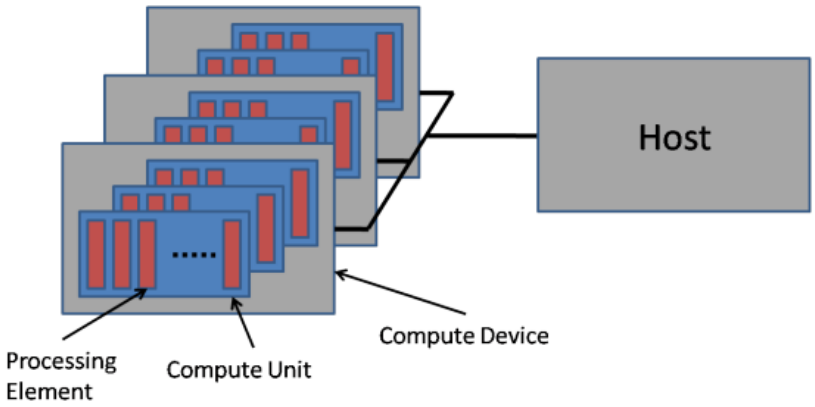
1.9 pav. Skirtingų procesorinių architektūrų palygimas, a) dalis centrinio procesoriaus

architektūra ir b) grafinio procesoriaus architektūra [5]

Skirtingų tipų skaičiavimo įrenginiai leidžia programinės įrangos kūrėjui parinkti labiausiai tinkamą apdorojimo įrenginį, kuris našiau atliks nurodytas užduotis *(žr. 1.9 pav).*

1.5.2 Grafinės platformos modelis

OpenCL platformos modelis yra priskirtas kaip atskira sistema, kuri yra prijungta prie vienos arba keletos *OpenCL* palaikančių įrenginių. Paveikslėlis *(žr. 1.10 pav)* rodo platformos modelį. Galima matyti, kad vienas pagrindinis įtaisas turi keletą lygiagrečiai sujungtų skaičiavimo įtaisų (grafinių procesorinių plokščių).



1.10 pav. Grafinės skaičiavimo platformos modelis [5]

Kiekvienas skaičiavimo įtaisas turi po atskirus duomenų apdorojimo ir skaičiavimo elementus *(žr 1.11 pav).*

Šiuo atveju, pagrindinis įtaisas yra motininis lustas, kuris turi standartinį CPU, veikiantį *Linux* operacinės sistemos pagrindu. Skaičiavimo įtaisai traktuojami kaip atskiri GPU, DSP ar multiprocesoriniai CPU moduliniai lustai. *OpenCL* palaikantys įrenginai susideda iš vieno ar daugiau skaičiavimo elementų (atskirų virtualių skaičiavimo procesorių) *(angl. Streming Processors).* Skaičiavimo elementai vykdo SIMD instrukcijas arba SPMD instrukcijas. Tipiškai SPMD instrukcijos vykdomos per bendros paskirties įrenginius, tokius kaip monolitinius procesorinius CPU *(angl. Processing Unit) [5].* Priešingai SIMD instrukcijos reikalauja vektorinės procesorinės sistemos, tokios kaip grafinis procesorius *(Angl. Graphical computing unit)* arba virtualios vektorinės praplėtimo galimybės CPU viduje [5].

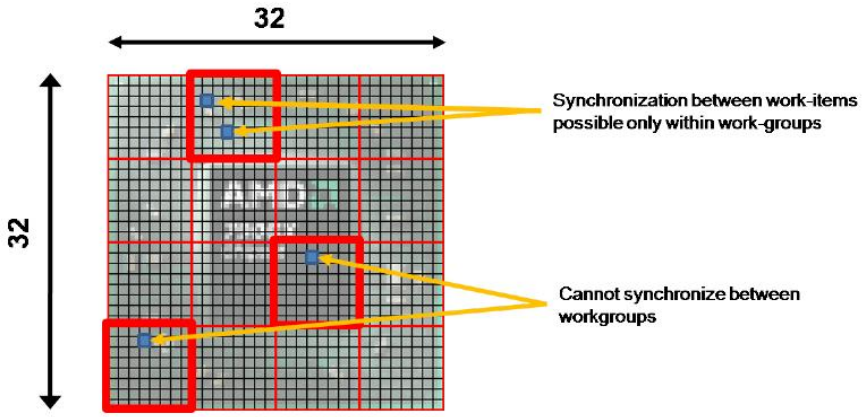
1.5.3 Virtualūs skaičiavimo branduoliai

OpenCL teikia lygiagrečius skaičiavimus nustatant problemą į *N*-dimensijos indekso vietą. Kai branduoliui *(angl. kernel)* reikia vykdyti programinius skaičiavimus, indekso vieta yra nustatoma. Kiekvienas nepriklausomas vykdymo elementas indekso vietoje vadinamas darbiniu

elementu. Kiekvienas darbinis elementas vykdo tokią pačią branduolio *(angl. kernel)* funkciją, bet su skirtingais duomenimis. Kai branduolio *(angl. kernel)* komandos yra sudedamos į komandų eilę, indekso vieta privalo būti nustatyta, tam kad būtų galima sekti visų darbinių elementų kiekį reikalingą komandų vykdymui [4][5]. *N*-dimensijos indekso vieta gali būti *N=1,2* arba 3. Skaičiuojant tiesišką duomenų masyvą *N* būtų *N=1*; vykdant nuotraukos atvaizdavimą *N=2*; ir atkuriant 3D vaizdus *N=3*.

Skaičiuojant 1024x1024 paveikslėlį tai būtų atliekama tokiu būdu: globali indekso vieta apima dviejų dimensijų erdvę 1024x1024, kurią sudaro vienas darbinis elementas vienam pikseliui, bendroje sumoje 1,048,576 įvykdymų. Su tokia indekso vieta kiekvienas darbinis elementas turi unikalų globalų ID. Darbinio elemento pikselio *x=30, y=22* globalus ID bus (30,22) [5].

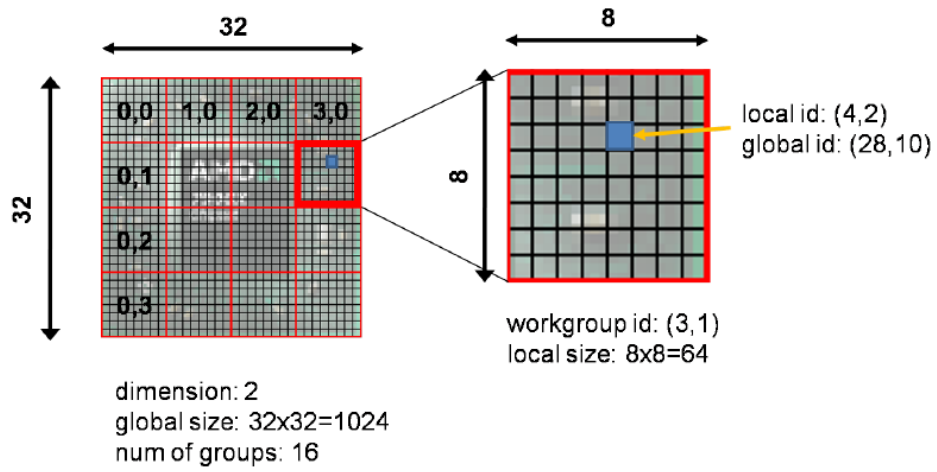
OpenCL taip pat leidžia grupuoti darbinius elementus, taip sukuriant darbinę grupę [5]. Tai galima matyti paveiksle *(žr 1.11 pav).* Kiekvienos darbinės grupės dydis yra nustatomas pagal jos lokalią indekso vietą. Visi darbiniai elementai toje pačioje darbinėje grupėje yra vykdomi tame pačiame įtaise. Priežastis dėl ko vykdymas vyksta viename įtaise yra ta, kad darbiniams elementams leidžiama dalintis lokalia atmintimi ir ją sinchronizuoti.



1.11 pav. Darbinių elementų grupavimas į darbines grupes [5]

Kitas pavyzdys *(žr 1.11 pav)* rodo dviejų dimensijų paveikslėlį su globaliu dydžiu 1024 (32x32). Indekso vieta padalinama į 16 darbinių grupių. Išryškinta darbinė grupė turi ID (3,1) ir jos lokalus dydis yra 64 (8x8).

Išryškintas darbinis elementas toje darbinėje grupėje turi lokalų ID (4,2), bet taip pat gali būti pasiekiamas su globaliu ID (28,10) [5].



1.12 pav. Darbinės grupės pavyzdys [5]

Sekantis pavyzdys iliustruoja kaip kernelis vykdomas pačiame *OpenCL*. Šiame pavyzdyje kiekvienas elementas yra pakeltas kvadratu tiesiškame masyve. Skaliarinei funkcijai reikia paprasto *for* ciklo iteravimo per elementus masyve ir pakeliant juos kvadratu:

```
void kvadratas (int skaitliukas, const float *mas, float *rez){
int i;
for(i = 0; i < skaitliukas; i++)
        rez[i] = mas[i] * mas[i];
}
```

Duomenų lygiagretavimui reikia nuskaityti elementą iš masyvo lygiagrečiu būdu ir viską išvesti į išėjimą. Verta paminėti, kad sekantis kodas neturi *for* ciklo: šis kodas paprastai skaito indekso reikšmes tam tikram branduoliui *(angl. kernel)* paprašius. Vykdomas skaičiavimas ir išvedamos reikšmės į išėjimą:

```
kernel void dp_kvadratas (global const float *mas, global float *rez){
int id = gauti_globalu_id(0);
rez[id] = mas[id] * mas[id];
}
```

OpenCL vykdymo modelis palaiko dvi branduolio *(angl. kernel)* kategorijas: *OpenCL* branduoliai ir paprasti branduoliai.

OpenCL branduoliai programuojami *OpenCL C* kalba ir kompiliuojami *OpenCL* kompiliatoriumi. Visi įtaisai, kurie naudoja *OpenCL* palaiko *OpenCL* branduolio vykdymą [5].

Paprasti branduoliai yra išplėstiniai branduoliai, kurių pagalba galima nustatyti specialias funkcijas programiniame kode arba eksportuoti iš bibliotekos tam tikram poreikiui. *OpenCL* API įtrauktos funkcijos, kurios parodo ar paprastas branduolys *(angl. kernel)* bus suderinamas su *OpenCL* branduoliu [5].

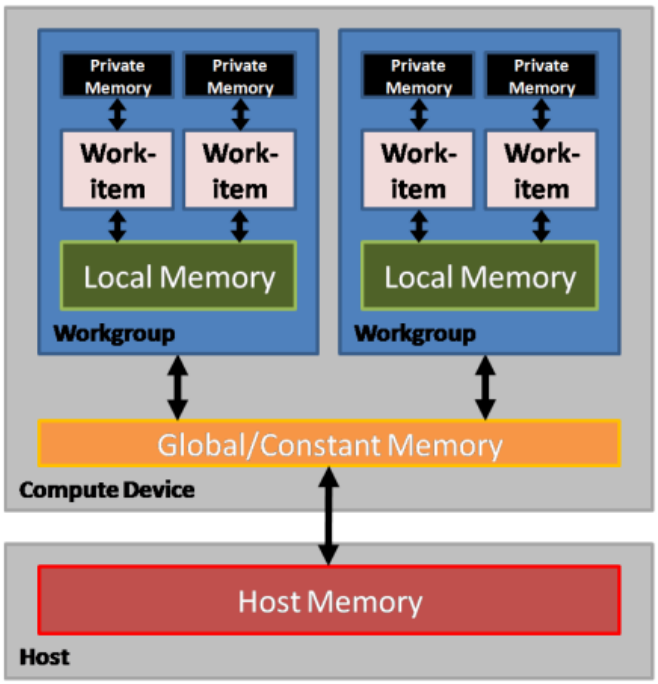
Jei naudojami paprasti branduoliai, projektuojant sistemą programuotojai turėtų susirūpinti dėl jų neveikimo su kitais *OpenCL* įtaisais.

1.5.4 Grafinio procesoriaus atminties mainai

Kadangi bendros atminties adresų erdvė yra nepasiekiama pagrindiniuose motininiuose įtaisuose ir *OpenCL* prietaisuose. *OpenCL* atminties modelis vykdant branduoliui *(angl. kernel)* apibrėžia keturis atminties regionus, kurie yra prieinami darbo elementams. Tolesniame paveikslėlyje *(žr 1.13 pav)* parodyti atminties regionai prieinami pagrindiniuose motininiuose įtaisuose ir kompiuterizuotuose prietaisuose.

Globali atmintis yra atminties regionas, kuriame visi darbiniai elementai ir darbo grupės turi skaitymo ir rašymo prieigą tiek kompiuterizuotuose prietaisuose, tiek pagrindiniuose motininiuose įtaisuose [5]. Šis atminties regionas gali būti skiriamas tik veikimo laiko metu pagrindinio motininio įtaiso.

Pastovi atmintis yra globalios atminties regionas, kuris išlieka pastovus per visą branduolio vykdymą. Darbiniai elementai šiame regione turi tik skaitymo prieigą. Pagrindiniams motininiams įtaisiams leidžiama skaitymo ir rašymo prieiga.



1.13 pav. Atminties kaita tarp pagrindinės plokštės ir grafinio procesoriaus atminties [5]

Vietinė atmintis yra atminties regionas naudojamas darbo elemetų duomenų dalijimuisi darbo grupėje. Visi darbo elementai toje pačioje darbo grupėje turi tiek skaitymo, tiek rašymo prieigą [5].

Privati atmintis yra regionas, kuris yra prieinamas tik vienam darbo elementui.

Daugeliu atveju, pagrindinio motinio įtaiso atmintis ir kompiuterizuoto prietaiso atmintis yra nepriklausomos viena nuo kitos. Taigi, atminties valdymas turi būti aiškūs norint, kad būtų leidžiama dalintis duomenimis tarp pagrindinio motininio įtaiso ir kompiuterizuoto įrenginio. Tai reiškia, kad duomenys turi būti aiškiai perkelti iš pagrindinio motininio įtaiso atminties į globalią atmintį, į vietinė atmintį ir atgal [5]. Šis procesas veikia įtraukiant skaitymo/rašymo komandas į komandų eilę. Komandos pateiktos eilėje, gali būti blokuojančios arba ne blokuojančios. Blokavimas reiškia, kad pagrindinio motininio įtaiso atminties komanda laukia, kol atminties operacija yra baigta prieš tęsiant toliau. Ne blokavimas reiškia, kad pagrindinis motininis įtaisas tiesiog kelia komandą į eilę ir tęsia toliau, o ne laukia, kol atminties operacija yra baigta.

Uždaviniai

Tam, kad būtų pasiektas šis darbo tikslas, reikia atlikti šiuos uždavinius:

1. Atlikti sintetinės apertūros fokusavimo technikos analizę;
2. Ištirti ir pasiūlyti tinkamiausią ir optimaliausią paskirstytų skaičiavimų mašinų duomenų apsikeitimo modelį tinklinėje telekomunikacijų struktūroje;
3. Pasiūlyti kombinuoto skaičiavimo algoritmo sprendimą, apjungiant grafinį procesorių, centrinį procesorių ir duomenų išskirstymą telekomunikaciniame tinkle;
4. Realizuoti sistemą, remiantis atliktais tyrimais.

2. MPI modelių duomenų srautų tyrimas

Šioje dalyje atliekami tinklo srautų paskirstymo tyrimai panaudojant MST ir skaičiavimo mašinų skaičiavimo gebos indeksą. Šie tyrimai yra atliekami tam, kad būtu išsiaiškinti skirtingų modelių duomenų srautai, o taip pat išsiaiškinti pasiūlytų naujų modelių duomenų srautų ribas. Toliau remiantis šio darbo atliktais tyrimo rezultatais sudaromas SAFT metodo skaičiavimo algoritmas, kuris optimizuojamas keliais nuosekliais etapais. Tyrimų seka:

- Pateikiami duomenų srauto modeliai redukcijai, redukcinės sklaidos, transliacijos *(angl. Broadcast),* tinklelio rinkimo-redukavimo ir visi visiems tinklo segmentams.
- SAFT bazinio metodo skaičiavimo interpretacijos eksperimentinėje *Matlab* posistemėje. Programiškai interpretuojamas SAFT algoritmas ir pateikiamos standartinio algoritmo spartos optimizacijos;
- Remiantis optimizacijomis ankstesniame etape SAFT algoritmas suprogramuojamas POSIX sistemos architektūroje panaudojant vieną virtualų procesorinį branduolį. Realizacijai pasirenkama C++ kalba;
- Atliekami pagrindinių SAFT rekursyvių skaičiavimų lygiagretinimo tyrimai ir realizuojamas lygiagretus programinis kodas. Lygiagretinimas atliekamas centrinio procesoriaus architektūroje;
- Pagrindinių SAFT rekursyvių skaičiavimų lygiagretinimas grafinio procesoriaus architektūroje;
- Sumodeliuojamas naujas tinklinis SAFT algoritmas, kuris paskirsto duomenis atskiroms skaičiavimo mašinoms;
- Remiantis atliktais tyrimais ir realizacijomis atliekamas kombinuoto skaičiavimo tyrimas;

2.1 Duomenų srauto modelis MST tinklinėje struktūroje

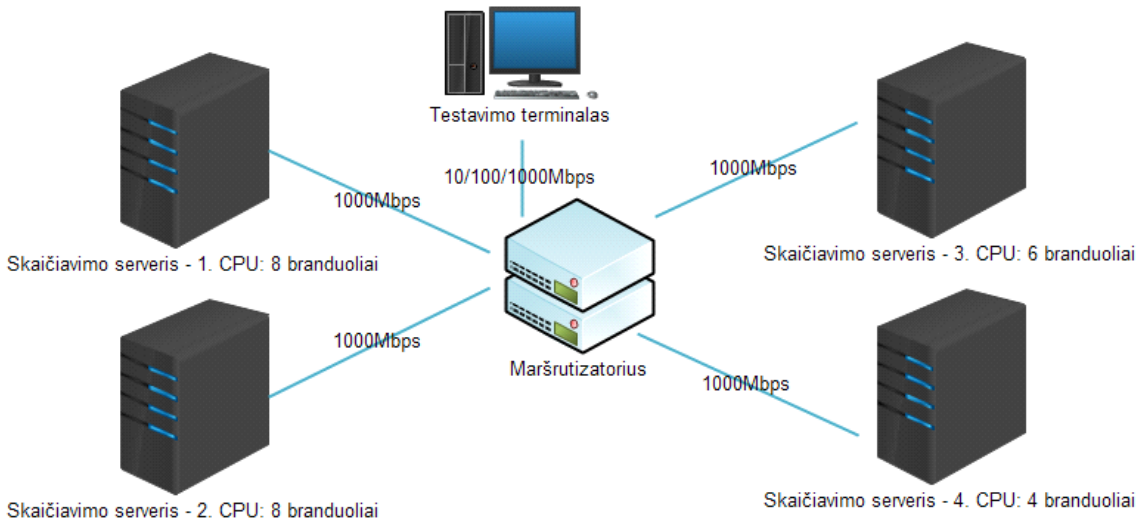
Šio tyrimo tikslas yra išanalizuoti skirtingų MPI modelių duomenų srautus. Išsiaiškinti ir pasiūlyti MPI modelių modifikacijas palyginat duomenų srautus tarp senų MPI modelių su naujų. MPI modeliams aprašyti šiame tyrime sudarytos kelios prielaidos, kurių pagalba aprašomos lygiagrečių skaičiavimų formuluotės. Sudaroma tyrimo metodika, tiriami standartiniai MPI modeliai su naujai siūlomais modeliais. Modeliai yra skirti duomenų apsikeitimui tarp atskirų tinklo segmentų. Eksperimentui panaudojami šie standartiniai MPI modeliai:

- Redukcijos, išsibarstymo ir rinkimo;

- Redukcinės-sklaidos;
- Transliacijos;
- Tinklelio rinkimo-redukavimo;
- Visi visiems.

2.1.1 Tyrimo metodika ir parametrai

Tyrimui panaudojamų duomenų skirstymas ir skaičiavimas tinkle susideda iš sujungtų tinklinių mazgų, kurie yra sujungiami komutatoriaus pagalba *(žr 2.1 pav.)*. Mazgai apibrėžiami kaip dvipusiu ryšiu veikiančios tinklinės mašinos, o tai reiškia, kad duomenys gali būti perduodami tuo pačiu metu dvejomis kryptimis. Laikas, per kurį siunčiami duomenys tarp keletos tinklo segmentų yra nepriklausomas nuo atstumo tarp segmentų ir gali būti nusakomas kaip $\beta n + \alpha$, kur α yra periodinis išsiuntimo laikas, kuris nėra priklausomas nuo išsiunčiamo duomenų kiekio, o β išreiškiamas kaip vieno baito išsiuntimo laikas. Persiunčiamas duomenų kiekis yra išreiškiamas n.



2.1 pav. Eksperimento testavimo tinklas. Tinklas susideda iš keturių skaičiavimo mašinų, testavimo terminalo ir tinklo infrastruktūros

Tyrimo atlikimui suprogramuota eksperimentinė MPI terpė, kurioje analizuojami MPI modeliai. Naudojama MPI terpė remiasi *C/C++ Qt* struktūra *(angl. framework).* Suprogramuota eksperimentinė terpė remiasi išeities kodo *(angl. source code)* modelių modifikacijomis (modifikuojami MPI modeliai) šiose funkcijose: redukcijos, redukcinės-sklaidos, transliacijos *(angl. broadcast),* tinklelio rinkimo-redukavimo ir „visi visiems“. Modifikacijos atliekamos *C++* programavimo kalba, keičiant modelių funkcionalą. Modelių testavimui panaudoti keturi POSIX serveriai *(žr. 2.1 pav.),* kurių skaičiavimo geba skirtinga ir charakterizuojama taip:

$$T_{cpu\_koef} = \begin{bmatrix} mašina\_0 & mašina\_2 \\ mašina\_1 & mašina\_3 \end{bmatrix}. \quad (5)$$

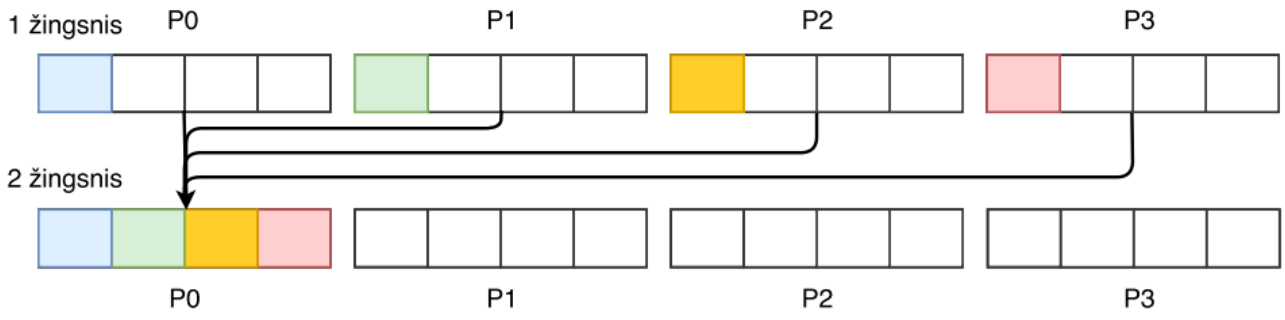
Koeficientų matrica T_{cpu_koef} apskaičiuojama vykdant mašininį skaičiavimo kodą. Kodas vykdo statinio vektoriaus (405000000 narių) perrūšiavimą į trimatę matricą ir kiekvienas matricos (450x450x2000 – 386MB duomenų) narys yra keliamas kvadratu, tai yra daroma su trijomis matricomis. Gavus tris trimates matricas jos yra sudauginamos panariui. Tokie skaičiavimai yra atliekami nepriklausomai nuo procesorinės architektūros ir skaičiuoja kodo vykdymo laiką, kuris vėliau naudojamas sudaryti koeficientų matricą. Eksperimentui sudaryti mašinų (POSIX serverių) skaičiavimo gebos koeficientai, kurie toliau naudojami norint tirti modelius pateikiami (6) išraiškoje:

$$T_{cpu\_koef} = \begin{bmatrix} 0 & 0{,}091 \\ 0{,}334 & 0{,}286 \end{bmatrix} \quad (6)$$

Koeficientų skaičiavimui, procesas turintis mažiausią vykdymo laiką yra naudojamas kaip pavyzdinis, pagal kurį sekantys procesai yra normuojami.

2.1.2 Redukcijos, išsibarstymo ir rinkimo modelis

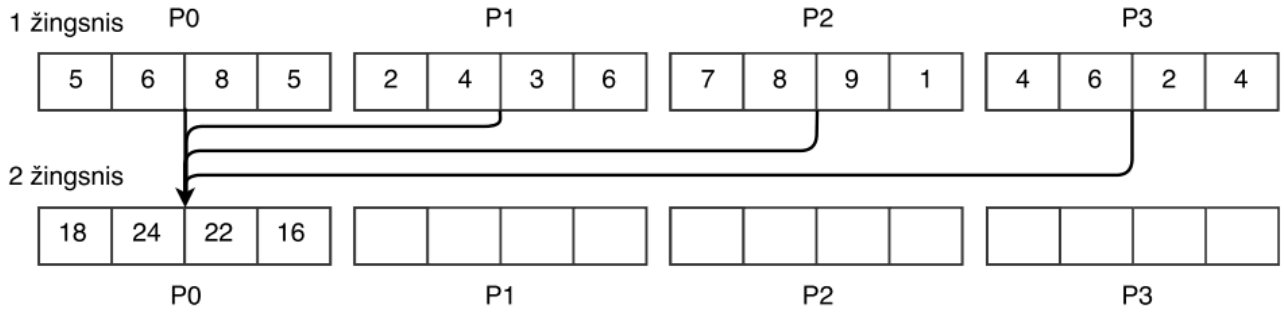
Šie modeliai yra vieni iš paprasčiausių ir atlieka pagrindines funkcijas, kurias naudoja žemiau aprašomi modleiai. Čia aprašomos trys procedūros: išsibarstymo, rinkimo ir redukcijos, kadangi veikimo modelių principai yra panašūs ir vienas kitą papildantys. Išsibarstymo *(angl. MPI_Scatter)* išbarsto duomenis nuo šakninių *(angl. root)* procesų visiems procesams duomenų perdavimo posistemei, tuo tarpu rinkimo *(angl. MPI_Gather)* procesas surenka duomenis iš visų procesų perdavimo posistemių ir perduoda informaciją šakniniam *(angl. root)* procesui *(žr. 2.2 pav).*



2.2 pav. Tinklinis MPI bendravimas. Rinkimo veikimo principas, pirmajame žingsnyje pavaizduoti keturi tinklo segmentai P0, P1, P2 ir P3, kurie atminties masyve (spalvoti kvadratai) talpina duomenis, antrajame žingsnyje duomenys yra surenkami P0 šakninio segmento

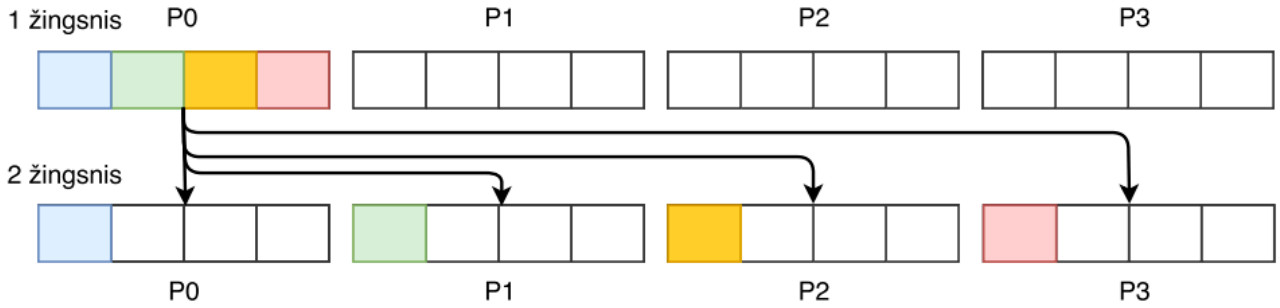
Redukcijos *(angl. MPI_Reduce)* procesas įvykdo duomenų redukcijos operacijas (optimizuojami sudėtiniai struktūros elementai) iš visų agreguotų duomenų, duomenis išsaugant

šakniniame procese arba procese iš kurio yra vykdoma (*mpirun*) programa *(žr. 2.3 pav).*



2.3 pav. Tinklinis bendravimas. Redukavimo veikimo principas, pirmajame žingsnyje pavaizduoti keturi tinklo segmentai P0, P1, P2 ir P3, kurie atminties masyve talpina duomenis (skaičiai), antrajame žingsnyje duomenys iš kiekvieno masyvo yra redukuojami (sudedami) P0 šakninio segmento

Toliau nagrinėjamas išsibarstymo *(angl. MPI_Scatter)* modelis, grafinė išsibarstymo struktūra pateikta paveiksle *(žr. 2.4 pav).*



2.4 pav. Tinklinis bendravimas. Išsibarstymo veikimo principas, pirmajame žingsnyje pavaizduotas vienas tinklo segmentas P0, kuris savo atminties masyvuose (spalvoti kvadratai) talpina duomenis, antrajame žingsnyje duomenys yra išskirstomi P0, P1, P2 ir P3 segmentams

Šakninis procesas P0 pirmuoju žingsniu siunčia pirmuosius masyvo elementus kitiems tinklo segmentams, taip priskiriant visus reikšminius elementus aktyviesiems procesams. Kiekvienas procesas arba tinklo segmentas saugo gautus duomenis masyve (*angl. recvbuf*). Šis tiesinis modelis naudojamas MPI programinėje bibliotekoje.

Jei p yra procesų numeris ir n yra visas duomenų dydis, kuris bus išskirstymas iš šakninio į kitus procesus, jo apdorojimo operacijos laikas nusakomas (7) lygtimi [14]:

$$T = (p-1)\alpha + \frac{p-1}{p} n\beta. \qquad (7)$$

Ilgiems pranešimams (nuo 40000 iki 16000000 bitų), šis modelis yra tinkamas, nes vėlavimo laikas gali būti kompensuojamas duomenų apdorojimo trukme, tačiau nėra tinkamas

optimizavimui (narys $\frac{p-1}{p}n\beta$ nėra toliau optimizuojamas ir apsprendžia greitaveikos ribas [14]). Trumpiems pranešimams (nuo 40 iki 20000 bitų), modelis naudoja mažiau nei *p-1* žingsnių kiekį, todėl, pavyzdžiui, $log_2 p$ procedūros panaudojimas yra našesnis.

Modelis paremtas minimalios paieškos medžio algoritmu. Pirmajame žingsnyje šakninis mazgas siunčia n/2 duomenų kiekį procesui (šakninis procesas + *p/2*) su sąlyga, kad šakninis mazgas yra ne nulinis. Kiekvienas iš šių procesų elgiasi kaip šakninis savo aktyvioje aplinkoje ir rekursiškai dalinasi duomenimis. Šis bendravimas užima $\log_2 p$ žingsnių. Duomenų perdavimo spartos kanalas tokiu atveju turi būti $\frac{n}{2}\beta$ pirmame žingsnyje, $\frac{n}{2^2}\beta$ sekančiame žingsnyje ir analogiškai sekantiems žingsniams, iki $\frac{n}{2^{log_2 p}}\beta$. Taip pat sumuojamas procesorių skaičiavimo koeficientas. Visas proceso užimamas laikas skaičiuojamas (8) formule:

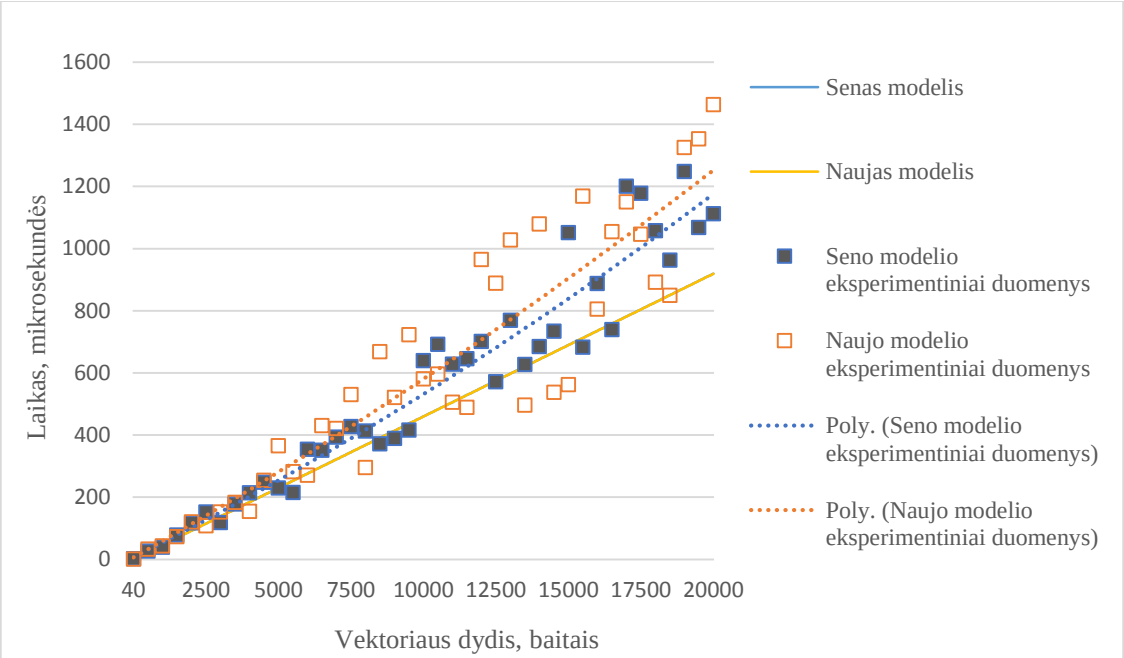
$$T = (log_2 p)\alpha + \frac{p-1}{p}n\beta + \beta \sum_i \sum_j T_{CPU\_koef} \qquad (8)$$

Lyginant su tiesiniu modeliu (7), trakto vėlinimas modeliu (8) gali būti mažesnis, kadangi šakninis mazgas duomenis siunčia n/2 metodu. Tai įvertinus, galima teigti, kad modelis (8) su MST algoritmu yra labiausiai tinkamas įvairių ilgių pranešimų apdorojimui. Modelio (8) trūkumas yra reikiamybė turėti laikinų duomenų saugojimo buferį sistemos vidiniams mazgams. Maksimali buferio užimama vieta yra *n/2*. Tiesinio (7) modelio principas nereikalauja papildomo laikinųjų duomenų saugojimo vidiniuose mazguose.

Duomenų rinkimui ir mažinimui, MPI naudojamas modelis (7), savo principais panašus į siūlomą (8).

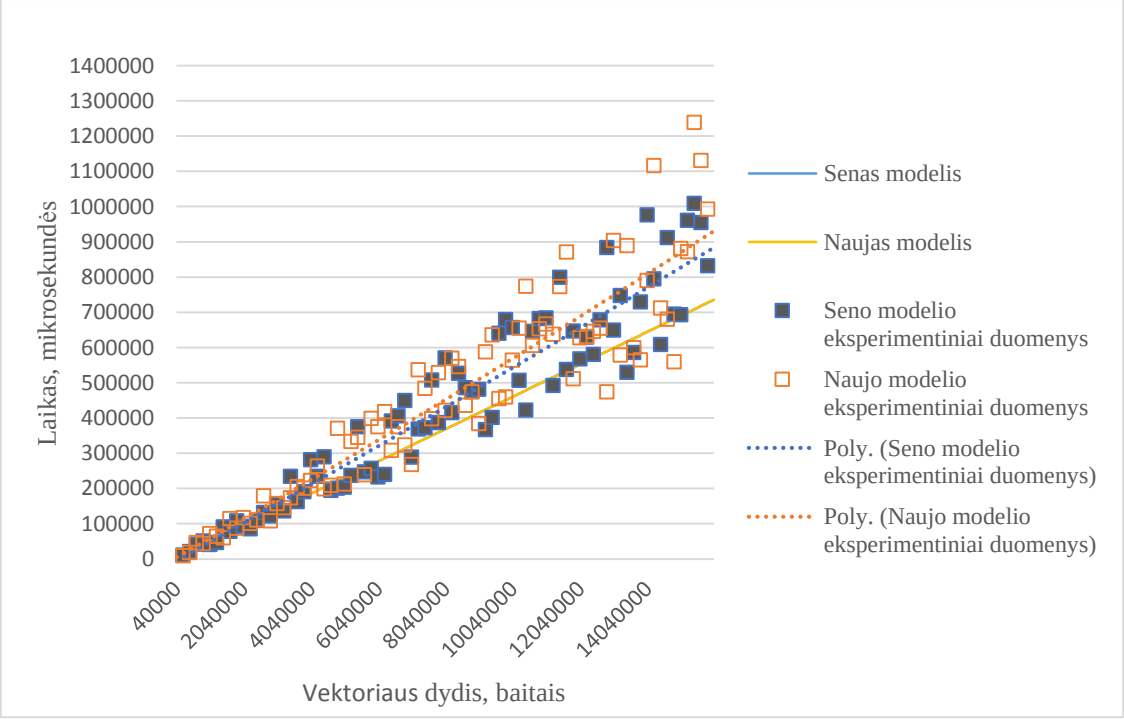
Paveiksluose pateikti išbarstymo (duomenys išsiunčiami tinklo segmentams) našumo grafikai, lyginant seną (7) ir naują (8) modelius *(žr. 2.5-2.6 pav.)*. Pateikiami teorinių modelių ir eksperimentiniai kitimai realiose sąlygose. Tyrimo atlikimui modelis realizuotas tinklo dėtuvės *(angl. stack)* lygyje pagal (7-8) formules. Abu paveikslai *(žr. 2.5 ir 2.6 pav.)* aproksimuoti antro laipsnio polinomu. Duomenys paveikslėliuose yra surinkti remiantis anksčiau kalbėta metodika *(žr. 2.1.1 skyrių)*.

Peteikta grafinė priklausomybė (*žr. 2.5 ir 2.6 pav.)*), kuri parodo duomenų masyvo dydžio priklausomybę išskaidant pranešimų dydžius į : trumpus (nuo 40 iki 20000 bitų) ir ilgus (nuo 40000 iki 16000000 bitų) panaudojant 26 procesų klasterį, kuriame yra keturios fizinės mašinos su paskirstytu branduolių kiekiu (pirma mašina – 8 branduoliai, antra mašina – 8 branduoliai, trečia mašina – 4 branduoliai ir ketvirta mašina – 6 branduoliai). Paveiksluose pateiktas *(žr. 2.5-2.6 pav.)* tinklelio gavimo funkcijos realizacijos našumas, lyginant naują (8) ir seną (7) modelį.



2.5 pav. Redukcijos modelių palyginimai, lyginamas senas modelis (7) su nauju (8). Naudojami trumpi pranešimai (nuo 40 iki 20000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu. Naujo (8) ir seno (7) modelių teorinės kreivės viena kitą perdengia

Trumpiems pranešimams, panaudojant MST ir mašinų koeficientų matricą, našumo augimas, remiantis senu modeliu (7) yra prastesnis lyginat su siūlomu modeliu (8), to priežastis yra $\log_2 p$ funkcijos panaudojimas, lyginant su $p-1$ funkcija. Dėl tos pačios priežasties, našumo praradimas vyksta taip pat ir ilgiems pranešimams *(žr. 2.6 pav.)*.



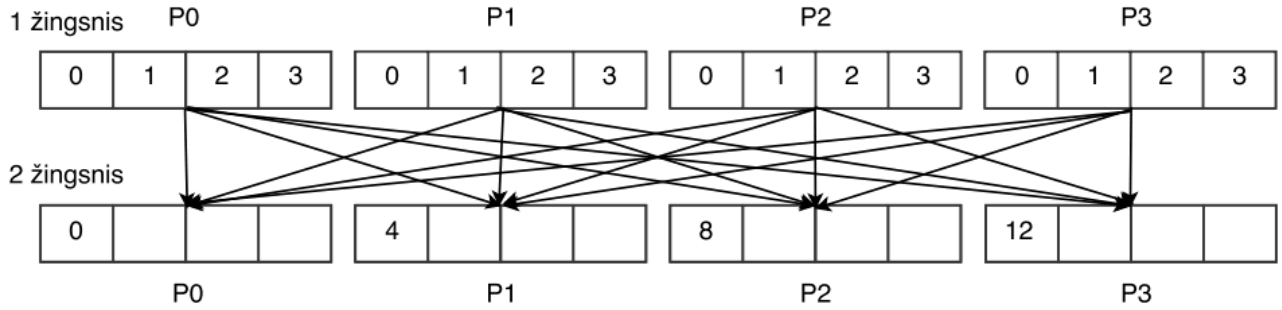
2.6 pav. Redukcijos modelių palyginimai, lyginamas senas modelis (7) su nauju (8). Naudojami

ilgi pranešimai (nuo 40000 iki 16000000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu. Naujo (8) ir seno (7) modelių teorinės kreivės viena kitą perdengia

Pranešimo dydžiui augant, greitaveika užima didesnę rolę suminiame informacijos apdorojimo kontekste ir našumo skirtumas tarp dviejų modelių susiaurėja dėl artimų greitaveikos reikalavimų duomenų mainų procedūroms vykdyti. Tam tikriems pranešimų dydžiams, tiesioginiai modeliai veikia geriau už MST, tačiau šiuo metu nėra pakankamai išaiškintos šio nuokrypio priežastys. Manoma, kad tam tikros konstantos neatsispindi matematinėje funkcijoje arba yra atitinkami skirtumai ryšio trakto laikiniame paskirstyme tarp dviejų modelių.

2.1.3 Redukcinės-sklaidos modelis

Redukcinė-sklaida *(angl. Reduce-scatter)* yra viena iš alternatyvų galima duomenų redukcijai. Tai gali būti naudojama tuo metu kai norima atlikti *(angl. broadcast)* duomenų redukciją kekviename tinklo segmente, o tai leidžia kiekviename segmente turėti redukcijos rezultatą. Tokia metodika geba sutaupyti tinklo apkrautumą, kadangi nebereikia po redukcijos išsiųsti papildomų duomenų tinklo segmentams. Vietoje duomenų saugojimo šakniniame mazge jie išsiunčiami visiems procesams. Tai yra redukcijos *(angl. MPI_Reduce)* iškvietimo ekikvalentas, kurį vykdo išbarstymo *(angl. MPI_Scatter)* funkcija. Redukcinė sklaida yra apibrėžiama paveiksle *(žr. 2.7 pav.).*



2.7 pav. Tinklinis bendravimas. Redukcinės-sklaidos veikimo principas, pirmajame žingsnyje pavaizduoti keturi tinklo segmentai P0, P1, P2 ir P3, kurie atminties masyve (skaičiai) talpina duomenis, antrajame žingsnyje duomenys yra redukuojami kiekvieno (P0, P1, P2 ir P3) tinklo segmento

Redukcinė sklaida atlieka elementinę redukciją elementų vektoriuose, siuntimo masyve. Rezultatų vektorius yra padalinamas į *p* disjunktyvius segmentus. Segmentas savyje laiko gautus elementus. *I*-tasis segmentas yra siunčiamas procesui ir saugomas gavimo masyve, kuris aprašomas duomenų tipo *(angl. datatype)* funkcijose.

MPI standarte [14] realizuotos redukcinės-sklaidos funkcijos, atliekant MST redukavimą iki nulinės eilės, po to vykdant tiesinę išbarstymo funkciją. Tai užima $\log_2 p + p - 1$ žingsnių ir

reikalauja $2n\beta(\frac{p-1}{p})$ greitaveikos (9) [14]:

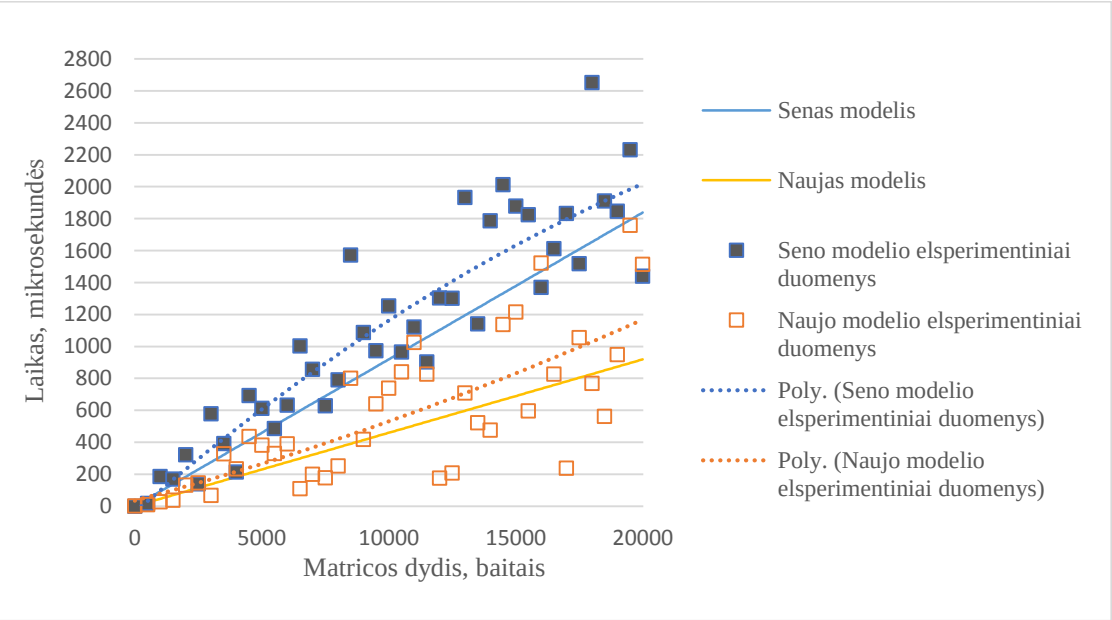
$$T = (log_2 p + p - 1)\alpha + \frac{p-1}{p} n\beta. \qquad (9)$$

Naujas modelis (10) veikia geriau abiem atvėjais: ilgiems (nuo 40000 iki 16000000 bitų) ir trumpiems (nuo 40 iki 20000 bitų) pranešimams. Trumpiems pranešimams, naudojamas rekursyvinis dubliavimo modelis, kuris užima $\log_2 p$ žingsnių. Pirmoje iteracijoje, procesai keičiasi $n - \frac{n}{p}$ duomenų kiekiu, antroje, $n - 2\frac{n}{p}$ duomenų kiekiu, trečioje $n - 2^2\frac{n}{p}$ duomenų kiekiu ir analogiškai kitose iteracijose. Taip pat sumuojamas procesorių skaičiavimo koeficientas. Laikas, užimamas vykdyti šį modelį, aprašomas (10) formule:

$$T = log_2 p\alpha + \left(log_2 p - \frac{p-1}{p}\right) n\beta + \beta \sum_i \sum_j T_{CPU\_koef} \qquad (10)$$

Pranešimų dydžiui augant, mažam procesų kiekiui, $log_2 p$ žingsnių efektas greitai dingsta. Dideliems pranešimų dydžiais, efektas imant $log_2 p$ žingsnių yra didesnis todėl galima sakyti, kad modelis yra našesnis didesniems pranešimų dydžiams. Atlikti tyrimai dviejų dydžių pranešimuose: trumpuose (nuo 40 iki 20000 bitų) ir ilguose (nuo 40000 iki 16000000 bitų) panaudojant 26 procesus klasteryje, kuriame yra keturios fizinės mašinos su paskirstytu branduolių kiekiu (pirma mašina – 8 branduoliai, antra mašina – 8 branduoliai, trečia mašina – 4 branduoliai ir ketvirta mašina – 6 branduoliai).

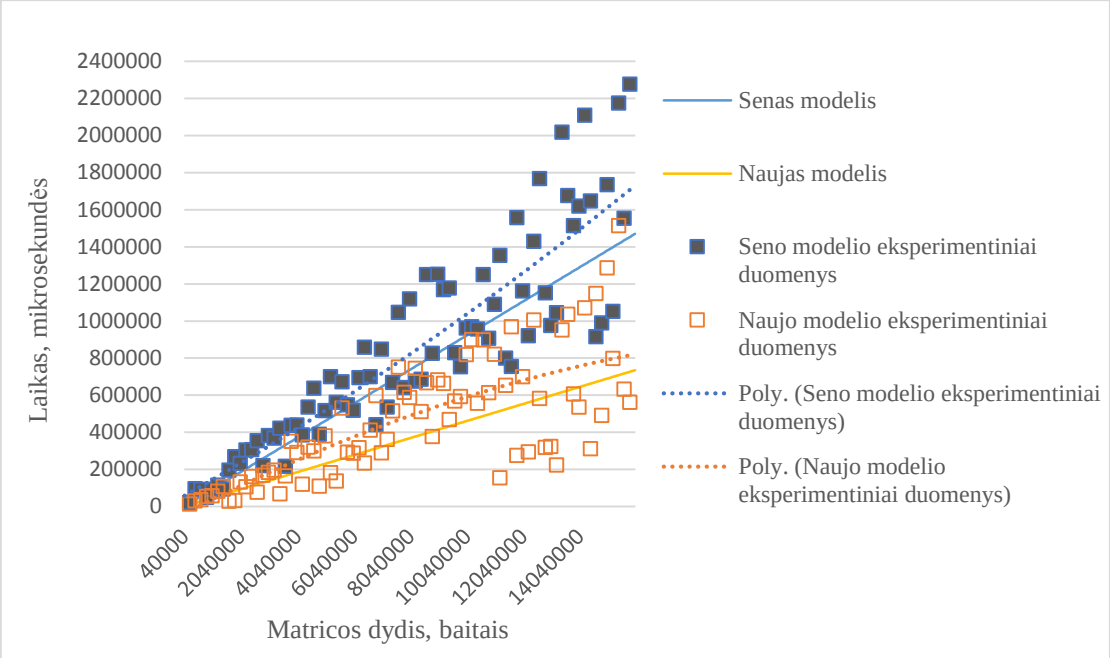
Paveiksle pateiktas redukcinės-sklaidos modelio našumo palyginimas tarp seno (9) ir naujo modelio (10) *(žr. 2.8 pav.).*



2.8 pav. Redukcinės-sklaidos modelių palyginimai, lyginamas senas modelis (9) su nauju (10). Naudojami trumpi pranešimai (nuo 40 iki 20000 bitų), eksperimentiniai duomenys aproksimuoti

antro laipsnio polinomu

Trumpiems ir ilgiems pranešimams, matoma aiški nauda $log_2 p$ lyginant su $(log_2 p + p - 1)$ žingsniais. Abu paveikslai *(žr. 2.8 ir 2.9 pav.)* aproksimuoti antro laipsnio polinomu. Atliekant tyrimą ilgiems pranešimams realiomis salygomis, naujo modelio (10) greitaveika yra geresnė didinat pranešimų kiekį, lyginant su senu modeliu (9).



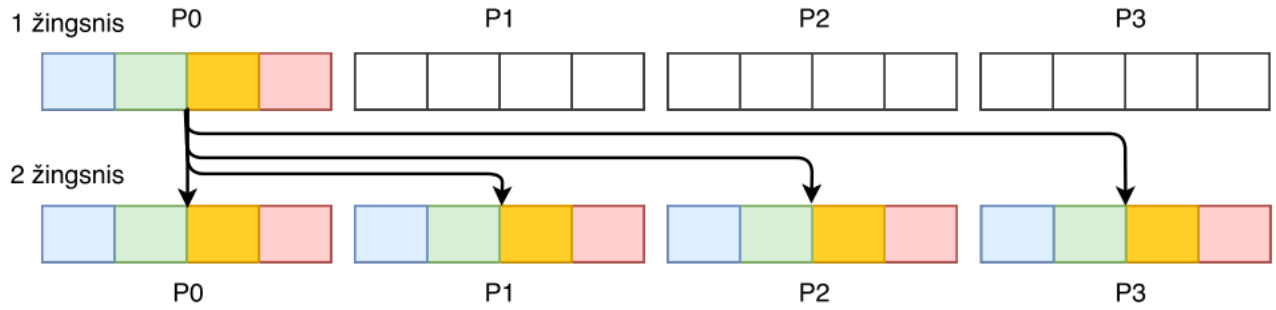
2.9 pav. Redukcinės-sklaidos modelių palyginimai, lyginamas senas modelis (9) su nauju (10). Naudojami ilgi pranešimai (nuo 40000 iki 16000000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu

Paveiksle matoma *(žr. 2.9 pav.),* kad didėjant pranešimų kiekiui nuo 9,57Mbit matomas spartos didėjimas, naujo modelio (10) aproksimacinė linija ties minėtu duomenų kiekiu krypsta žemyn.

2.1.4 Transliacijos modelis

Transliavimas *(angl. broadcast)* yra dažnai naudojamas ir itin svarbi duomenų paskirstymo procedūrų dalis. Transliacija naudojama kai turimus duomenis reikia paskirstyti tinkle. Tai vienas iš paprastesnių paskirstymo modelių, pavyzdžiui, lyginat su redukcija ar redukcijos-sklaidos modeliais. Schematinis funkcijos pavyzdys pateiktas paveiksle *(žr. 2.10 pav.)*.

Transliacijos modelis *(angl. broadcast)* transliuoja pranešimus iš procesų, turinčių šakninį prioritetą visiems kitiems grupės procesams ar segmentams. Dažnai transliavimui standartiniame MPI naudojamas MST algoritmas [14].



2.10 pav. Tinklinis bendravimas. Transliacijos veikimo principas, pirmajame žingsnyje pavaizduoti keturi tinklo segmentai P0, P1, P2 ir P3 iš kurių vienas P0 turi užpildytą atminties duomenų masyvą (spalvoti kvadratai), antrajame žingsnyje duomenys yra išsiunčiami taip, kad kiekvienas (P0, P1, P2, P3) segmentas turėtų tuos pačius duomenis kaip šakninis segmentas P0

Pirmoje iteracijoje, šakninis procesas siunčia duomenis procesui (šakninis + $\frac{p}{2}$). Kiekvienas iš šių procesų elgiasi kaip šakninis su savo pomedžių *(angl. subtree)* ir rekursiškai kartoja šį algoritmą. Komunikavimas užima iš viso $\log_2 p$ žingsnius. Sunaudota greitaveika kiekvienam žingsniui yra $n\beta$, kur n yra duomenų kiekis [14]. Šio modelio vykdymo trukmė aprašoma (11) formule [14]:

$$T = (log_2 p)(\alpha + n\beta). \qquad (11)$$

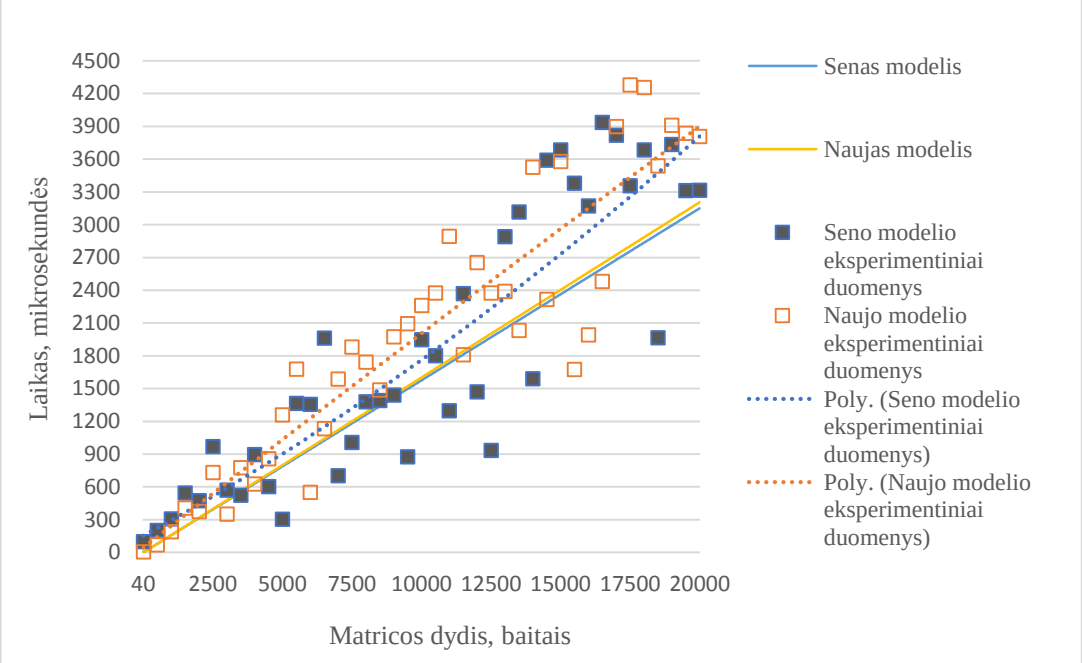
Šis modelis yra naudojamas MPI standartuose. Jis yra nenašus trumpiems pranešimams dėl MST duomenų skaidymo. Ilgiems pranešimams, šis modelis yra geresnis, kur MST vėlinimas gali būti ignoruojamas. Vadinasi, pranešimas, kuris bus transliuojamas, pirmiausia padalijamas ir išskirstomas procesams, panašiai, kaip ir išsibarstymo *(angl. MPI_Scatter)* funkcijoje. Duomenys yra išskirstomi ir surenkami atgal į procesus.

Šis metodas gali būti naudojamas dėl gerų našumo rezultatų komutuojamame tinkle. Modelio vykdymo laikas apibrėžiamas (12) formule:

$$T = 2log_2 p\alpha + 2\frac{p-1}{p}n\beta + \beta \sum_i \sum_j T_{CPU\_koef} \qquad (12)$$

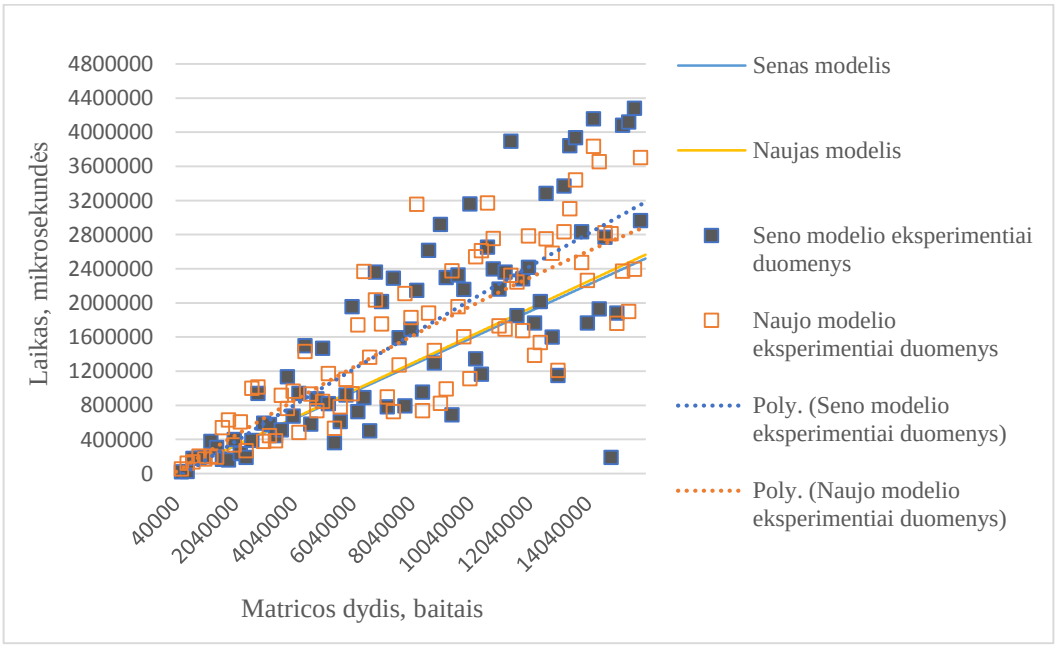
Lyginant šį modelį (12) su senuoju modeliu (11), manoma, kad ilgiems pranešimams, kur vėlavimas yra nereikšmingas ir kuomet $log_2 p$ > 2 (arba p >4), modelis gali būti našus *(žr. 2.12 pav.)*. Maksimalus našumo gerinimas, lyginant su senuoju modeliu (11) gali būti iki $\frac{log_2 p}{2}$. Kitaip tariant, kuo didesnis procesų skaičius, tuo didesnis našumo pagerėjimas.

Galima galvoti, kad ilgiems pranešimams, turintiems penkis ar daugiau procesų naujas modelis gali tikti. Reikėtų atsižvelgti modeliuojant SAFT algoritmą pasirinkus transliavimo modelį, gali reikėti pasirinkti skyros liniją tarp trumpų ir ilgų pranešimų.



2.11 pav. Transliacijos modelių palyginimai, lyginamas senas modelis (11) su nauju (12). Naudojami trumpi pranešimai (nuo 40 iki 20000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu

Paveiksle *(žr. 2.12 pav.).* pateiktas seno (11) ir naujo modelio (12) našumo palyginimas. Demonstruojamas našumas tik ilgiems pranešimams *(žr. 2.12 pav.).* Trumpiems pranešimams *(žr. 2.11 pav.)* naudingiau yra naudoti seną modelį (11).



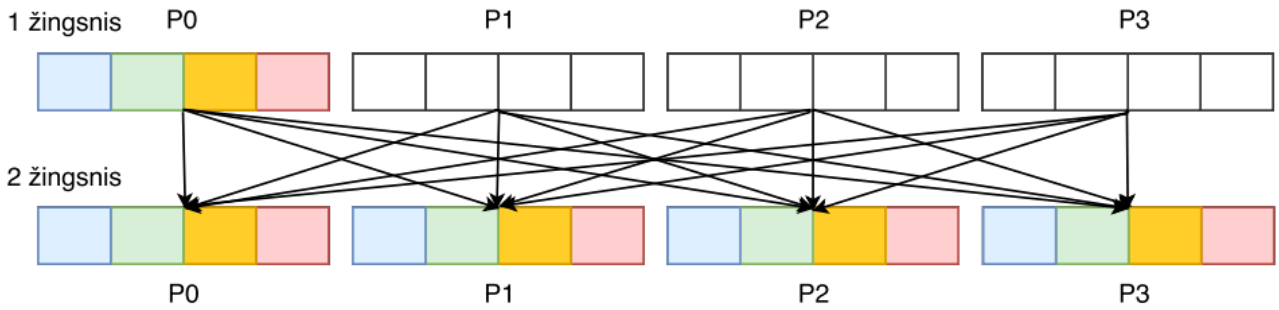
2.12 pav. Transliacijos modelių palyginimai, lyginamas senas modelis (11) su nauju (12). Naudojami ilgi pranešimai (nuo 40000 iki 16000000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu

Atlikti tyrimai dviejų dydžių pranešimuose trumpuose (nuo 40 iki 20000 bitų) ir ilguose (nuo 40000 iki 16000000 bitų) panaudojant 26 procesus klasteryje, kuriame yra keturios fizinės mašinos su paskirstytų branduolių kiekiu (pirma mašina – 8 branduoliai, antra mašina – 8 branduoliai, trečia mašina – 4 branduoliai ir ketvirta mašina – 6 branduoliai). Paveiksluose pateikti (žr. 2.11 pav. ir 2.12 pav.) transliacijos funkcijos realizacijos našumas, lyginant naują ir seną modelį. Abu paveikslai (žr. 2.11 ir 2.12 pav.) aproksimuoti antro laipsnio polinomu.

Kitas ženklus rezultatas yra naujo modelio (12) plečiamumas ilgiems pranešimams: modelio vykdymo laikas mažėja didinant pranešimų skaičių. Tai matoma kai pranešimo ilgis siekia 10Mbit ir daugiau pranešimų *(žr. 2.12 pav.)*.

2.1.5 Tinklelio rinkimo redukavimo modelis

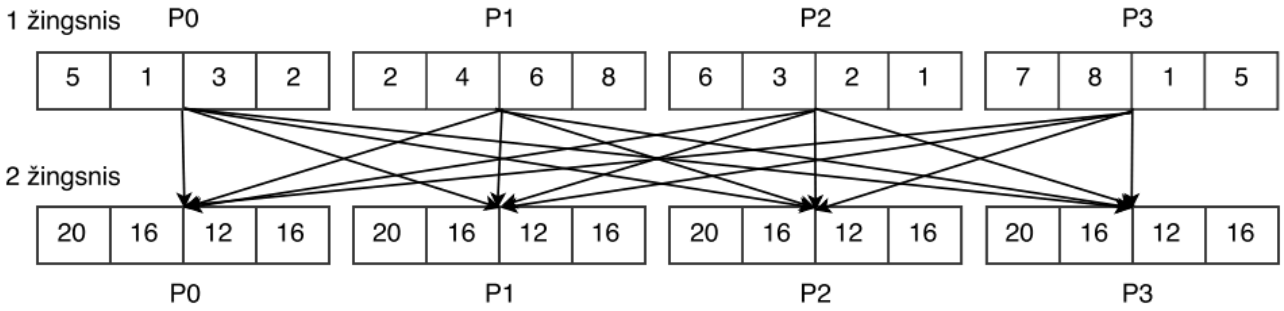
Tinklelio rinkimas ir redukavimas yra panašūs į duomenų rinkimo ir duomenų redukcijos funkcijas, išskyrus tai, jog visi procesai gauna rezultatus (kitais atvejais rezultatus gauna tik šakninis mazgas). Schematinis tinklelio rinkimo *(angl. MPI_Allgather)* funkcijos išpildymas pateiktas paveiksle *(žr. 2.13 pav.)*.



2.13 pav. Tinklinis bendravimas. Tinklelio rinkimo veikimo principas, pirmajame žingsnyje pavaizduoti keturi tinklo segmentai P0, P1, P2 ir P3 iš kurių vienas P0 turi užpildytą atminties duomenų masyvą (spalvoti kvadratai), antrajame žingsnyje duomenys yra išsiunčiami taip, kad kiekvienas (P0, P1, P2, P3) segmentas turėtų tuos pačius duomenis kaip šakninis segmentas P0

Taigi toks rinkimas (žr. 2.13 pav.) veikia taip pat kaip ir transliacijos metodas tik tinklelio rinkimo redukavimo modelis turi papildomą funkcija kuri atlieka redukcija. Schematinis tinklelio redukavimo *(angl. MPI_Reduce)* funkcijos išpildymas pateiktas paveiksle *(žr. 2.14 pav.)*.

Vienas tinklelio rinkimo funkcijos realizavimo būdų yra funkcijos realizavimas *0*-tajam procesui ir tada vykdyti transliaciją *(angl. MPI_Bcast)*.



2.14 pav. Tinklinis bendravimas. Tinklelio redukavimo veikimo principas, pirmajame žingsnyje pavaizduoti keturi tinklo segmentai P0, P1, P2 ir P3, kurie atminties masyve (skaičiai) talpina duomenis, antrajame žingsnyje duomenys iš kiekvieno segmento (pirmame žingsnyje) yra redukuojami (P0, P1, P2 ir P3) tinklo segmentų

Nuo antrosios iteracijos žingsnio, kiekvienas procesas persiunčia duomenis kitam procesui ir gauna praėjusio žingsnio informaciją iš prieš tai siųstam procesui. Visas modelis vykdomas per *p-1* žingsnių, tam reikalinga kiekvienam žigsniui $\frac{n}{p}\beta$, kur *n* yra visas duomenų dydis, kuris turi būti gaunamas bet kokio proceso iš visų kitų procesų. Šios procedūros sunaudojamas laikas aprašomas (13) lygtimi [14]:

$$T = (p-1)\alpha + \frac{p-1}{p}n\beta. \qquad (13)$$

Pažymėtina, kad procedūros greitaveika negali būti apribota, nes kiekvienas procesas turi gauti *n/p* duomenis iš kitų *p-1* mazgų. Vėlinimo laikas gali būti sumažintas, jeigu naudojamas logaritminis modelis (14), kuris naudoja $\log_2 p$ žingsnių skaičius. Šis modelis gali būti vadinamas kaip, rekursyviniu dubliavimu.

Rekursyvinio dubliavimo modelyje (14), procesai keičiasi duomenimis su kiekvienu kitu $\log_2 p$ žingsniu. Modelio pseudo programinis pavyzdys galėtų būti toks:

```
kaukė=1
ciklas (kaukė < procesų kiekis)
{
   galutinis_taškas = prioritetas ^ kaukė
   duomenų apsikeitimo su visais segmentais instrukcija
   kaukė <<= 1
}
```

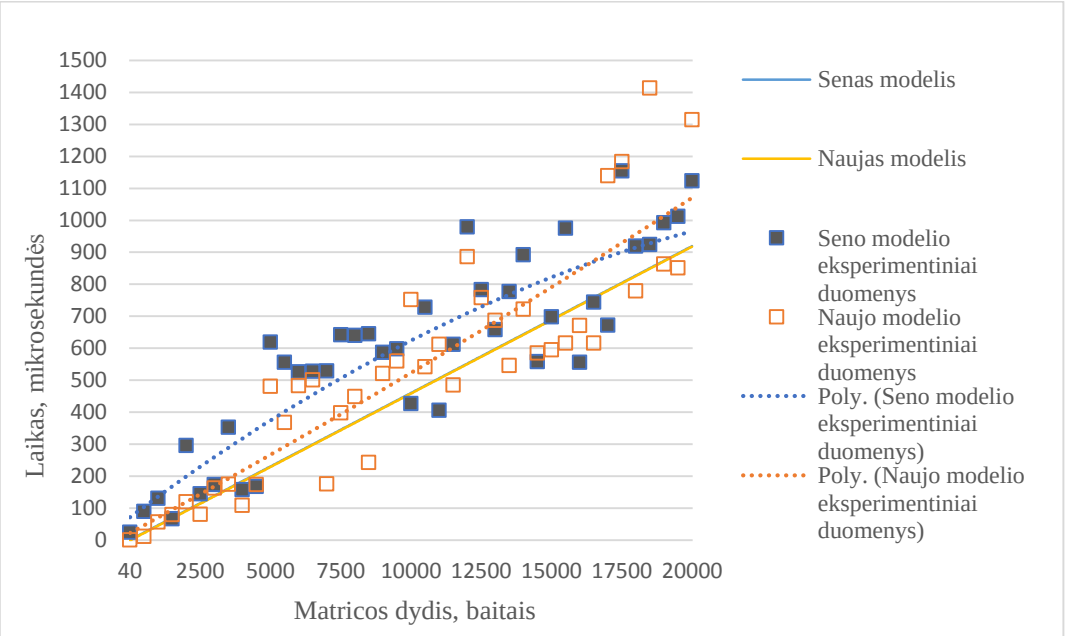
Pirmame rekursyvinio dubliavimo žingsnyje, kiekvienas procesas keičiasi duomenimis su nuotoliniu procesu, kurio pirmenybė nustatoma XOR funkcija. Šiam procesui suteikiama vieneto reikšmė (vadinama „kauke“). „Kaukei” atliekamas postūmis per vieną bitą, kol visi procedūros žingsniai yra įvykdyti. Antrajame žingsnyje kiekvienas procesas keičiasi duomenims, kuriuos jis

turi (įskaitant duomenis iš praeito žingsnio). Antro laipsnio procesams, $\log_2 p$ žingsniuose, kiekvienas procesas gauna kitų procesų paskirstytus duomenis. Reikalinga greitaveika šiai procedūrai $\frac{n}{p}\beta$ pirmajame žingsnyje, $2\frac{n}{p}\beta$ antrajame žingsnyje ir analogiškai sekančiuose žingsniuose, iki paskutiniojo žingsnio $\frac{2^{log_2 p-1}n}{p}\beta$. Laikas, trunkantis vykdant šį algoritmą, apibrėžiamas (14) lygtimi:

$$T = log_2 p\alpha + \frac{p-1}{p}n\beta + \beta\sum_i\sum_j T_{CPU\_koef}. \qquad (14)$$

Kiekviename rekursyvinio dubliavimo žingsnyje (14), užtikrinama, kad jei esamas pomedis *(angl. „subroot“)* nėra antro laipsnio, visi procesai gauna duomenis, kuriuos jie būtų gavę ir tuo atveju, jei pomedis *(angl. subroot)* būtų antrojo laipsnio. Belaipsniai žingsniai yra koreguojami ir logaritmiškai, siekiant minimizuoti žingsnių skaičių ir sumažinti vėlavimą. Visas žingsių skaičius belaipsnėms funkcjoms yra apribotas funkcijos $2\log_2 p$.

Rekursyvinio dubliavimo modelis (14) neturi perteklinių duomenų ir naudoja logaritminį žingsių skaičių, todėl šis modelis gali būti panaudojamas tiek trumpuose (nuo 40 iki 20000 bitų), tiek ilguose (nuo 40000 iki 16000000 bitų) pranešimuose.

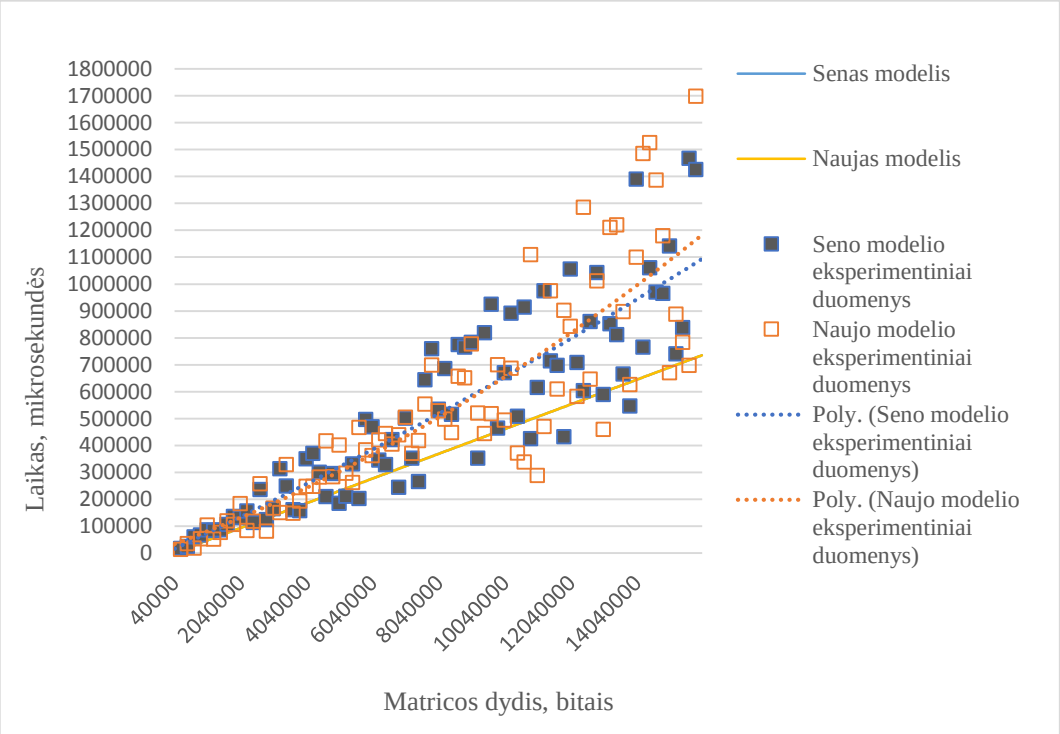


2.15 pav. Tinklelio rinkimo modelių palyginimai, lyginamas senas modelis (13) su nauju (14). Naudojami trumpi pranešimai (nuo 40 iki 20000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu. . Naujo (14) ir seno (13) modelių teorinės kreivės viena kitą perdengia

Atlikti tyrimai dviejų dydžių pranešimuose *(žr. 2.15 pav. ir 2.16 pav.)*: trumpuose (nuo 40 iki 20000 bitų) ir ilguose (nuo 40000 iki 16000000 bitų) pranešimuose panaudojant 26 procesus klasteryje, kuriame yra keturios fizinės mašinos su paskirstytu branduolių kiekiu (pirma mašina – 8

branduoliai, antra mašina – 8 branduoliai, tečia mašina – 4 branduuoliai ir ketvirta mašina – 6 branduoliai). Paveiksluose pateikti *(žr. 2.15 pav. ir 2.16 pav.)* tinklelio gavimo funkcijos realizacijos našumas, lyginant naują ir seną modelį.

Trumpiems pranešimams (nuo 40 iki 20000 bitų), kur vėlavimas dominuoja visą laiką, našumo pagerėjimas yra sukeliamas naujo modelio, lyginant jį su senuoju modeliu. Pranešimo dydžiui augant, vėlavimas turi mažesnę įtaką. Laikas yra dominuojamas greitaveikos kaina, kuri abiems modeliams (13 ir 14) yra reikalinga tokia pati. Galima būtų tikėtis, kad modelio našumas gali būti panašus ir dideliems pranešimams.



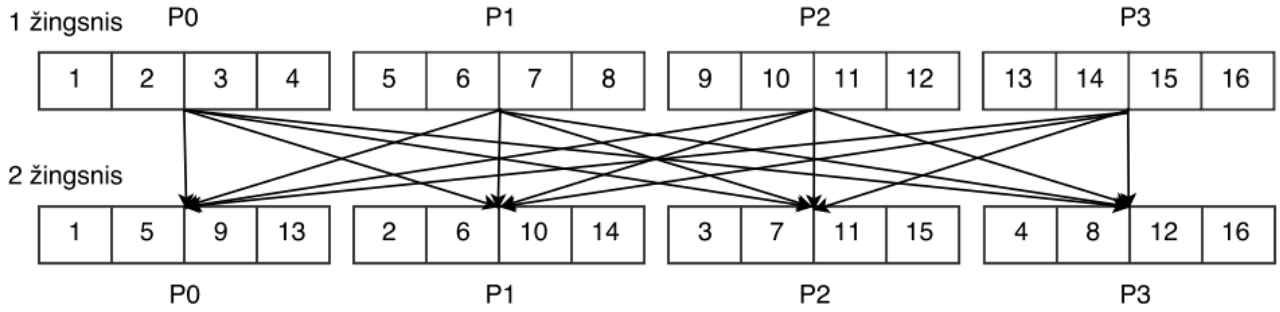
2.16 pav. Tinklelio rinkimo modelių palyginimai, lyginamas senas modelis (13) su nauju (14). Naudojami ilgi pranešimai (nuo 40000 iki 16000000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu. Naujo (14) ir seno (13) modelių teorinės kreivės viena kitą perdengia

Ilgų pranešimų rezultatai parodo *(žr. 2.16 pav.)* kad naujasis modelis (13) yra prastesnis už senąjį (14) ilguose pranešimuose. Apibendrinant galima teigti, kad komunikacijos laikinis paskirstymas rekursyvinio dubliavimo metu yra reikšminga dedamoji, siekiant optimizuoti pranešimų apdorojimo trukmes ilgiems pranešimams.

2.1.6 *Visi visiems* modelis

Visi visiems (angl. MPI_Alltoall) komunikavimas vykdomas tradicinė forma, kuri surenka lygiagrečiai paskirstytus duomenis matricos perkėlimui, masyvo paskirstymams ar kitoms

aktualioms procedūroms. Tai yra atitinkama duomenų perdavimo struktūra, kuomet duomenys iš visų turimų procesų persiunčiami visiems likusiems procesams ar segmentams. *Visi visiems* realizacija pateikta paveiksle *(žr. 2.17 pav.).*



2.17 pav. Tinklinis bendravimas. Tinklelio visi visiems veikimo principas, pirmajame žingsnyje pavaizduoti keturi tinklo segmentai P0, P1, P2 ir P3, kurie atminties masyve (skaičiai) talpina duomenis, antrajame žingsnyje duomenys iš kiekvieno segmento (pirmame žingsnyje) yra priimami (P0, P1, P2 ir P3) tinklo segmentų

Kiekvienas procesas siunčia elementus visiems procesams ar segmentams. Duomenys siunčiami kiekvienam procesui yra atskiri, paimti iš masyvo pagal numatytą proceso laipsnį. Taigi pavyzdžiui pirmasis atminties blokas išsiųstas iš proceso ar segmento *P0* (žr. 2.17 pav.)) yra gaunamas proceso *P0* (antrajame žingnyje) ir patalpintas *P0* (antrajame žingnyje) pirmame masyvo bloke, toliau seka tokie patys žingsniai iki tol kol visi masyvų duomenys yra padalinami.

Pagrindinis standartinio modelio minusas yra tai, kad pranešimai siunčiami be tvarkos. Visi pranešimai siunčiami *0*-iniam procesui, tuomet pirmam procesui ir analogiškai visiems likusiems procesams, dėl to sudaromas mazgo perkrovimas *(angl. botleneck).*

Geresnis to sprendimas yra realizuoti visi visems serijas, duomenis perduodant procesų poroms. Toks algoritmas gali būti įgyvendintas per $p-1$ žingsnį. Kiekviename žingsnyje, kiekvienas procesas keičiasi duomenimis su nuotoliniu mazgu, kurio laipsnis yra išskaičiuojamas XOR logikos pagalba. Šis algoritmas veikia tik su procesais, kurių numeris skaičiuojamas antruoju laipsniu.

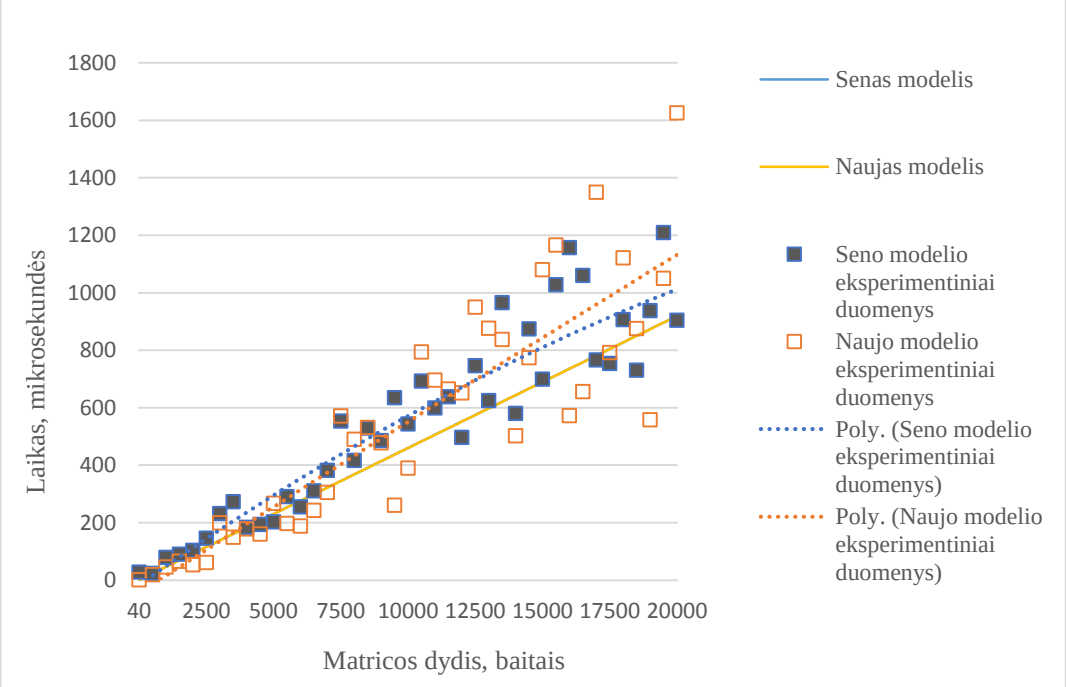
Jeigu *n* yra visas duomenų kiekis, kuris vieno proceso turi būti išsiųstas visiems kitiems procesams, laikas, užimamas šio standartinio MPI modelio yra nusakomas (15) lygtimi [14]:

$$T = (p-1)\alpha + \frac{p-1}{p} n\beta. \qquad (15)$$

Tai yra geras modelis (15) ilgiems pranešimams, nes jis vartoja ne daugiau pralaidumo nei reikia. Trumpų pranešimų apdorojimui yra vietos optimizuoti šį principą, jeigu naudojamas logaritminis žingsnių skaičius (16):

$$T = (log_2 p)\alpha + np\beta + \beta \sum_i \sum_j T_{CPU_{koef.}} \quad (16)$$

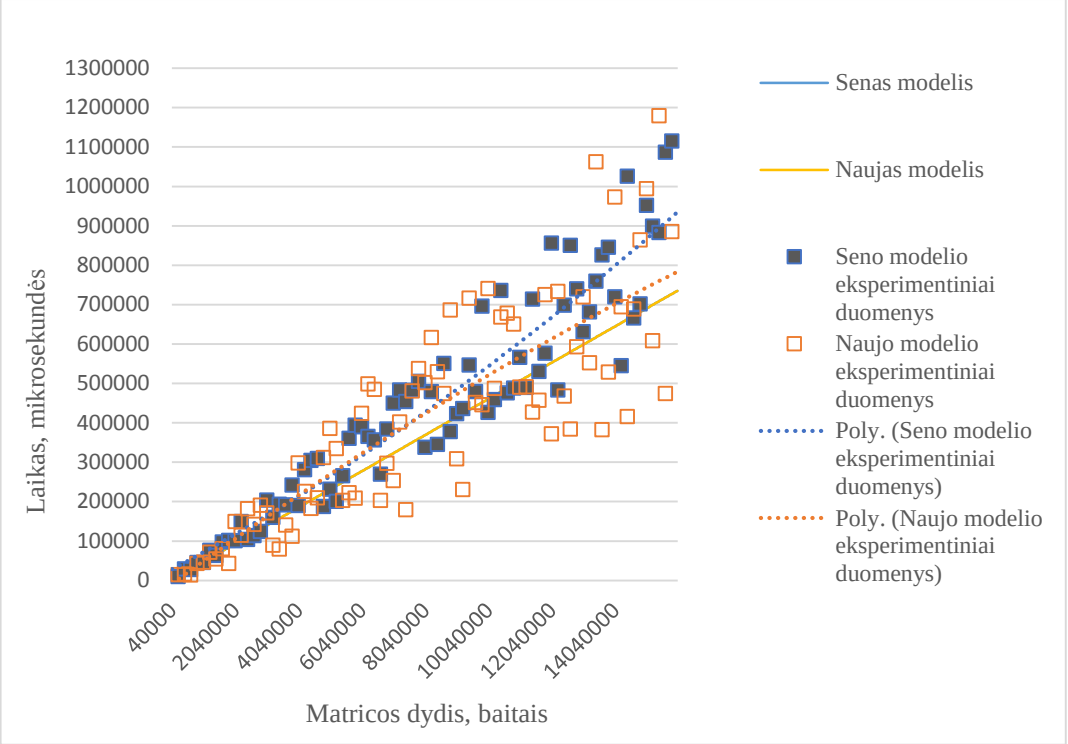
Proceso poravimas kiekviename žingsnyje yra atliekamas tokiu pačiu būdu kaip ir tinklelio gavimo modelyje (13-14). Pirmajame žingsnyje, kiekvienas procesas siunčia visus duomenis, kuriuos turi į gavėjo procesą (įskaitant duomenis, skirtus kitiems procesams). Antrame žingsnyje, kiekvienas procesas siunčia savo duomenis kitiems procesams, kaip ir pirmajame žingsnyje. Ši procesūra trunka iki $\log_2 p$ žingsnių. Pabaigoje, visi procesai turi visus duomenis, kuriuos turėjo visi procesai pradžioje. Kiekvienas procesas tada pasirenka duomenis, kurie jam reikalingi ir eliminuoja likusius. Greitaveika, kuri reikalinga pirmajame žigsnyje yra *nβ*, sekančiame žingsnyje reikalinga yra *2nβ* ir taip iki $2^{log_2 p-1} n\beta$ paskutiniojo žingsnio. Šio modelio vykdymo trukmė yra nusakoma (16) lygtimi.



2.18 pav. Visi visiems modelių palyginimai, lyginamas senas modelis (15) su nauju (16). Naudojami trumpi pranešimai (nuo 40 iki 20000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu. Naujo (16) ir seno (15) modelių teorinės kreivės viena kitą perdengia

Tačiau šis principas naudoja daugiau pralaidumo, lyginant su senuoju modeliu, todėl jis labiau gali būti labiau priskiriamas ilgiems pranešimams *(žr. 2.19 pav.)*.

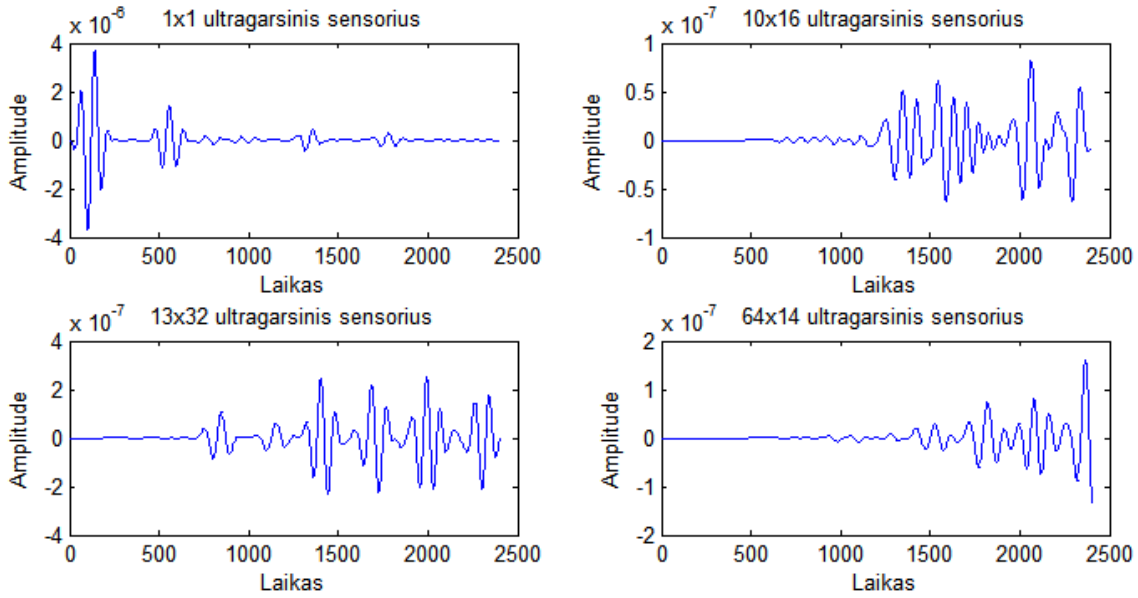
Didėjant pranešimų kiekiui nuo 9,57 Mbit, matomas nedidelis spartos didėjimas ilguose pranešimuose *(žr. 2.19 pav).* Spartos augimas matomas iki 1,329 karto.



2.19 pav. Visi visiems modelių palyginimai, lyginamas senas modelis (15) su nauju (16). Naudojami ilgi pranešimai (nuo 40000 iki 16000000 bitų), eksperimentiniai duomenys aproksimuoti antro laipsnio polinomu. Naujo (16) ir seno (15) modelių teorinės kreivės viena kitą perdengia

2.2 Eksperimentiniai duomenys ir programinių įrankių pasirinkimas

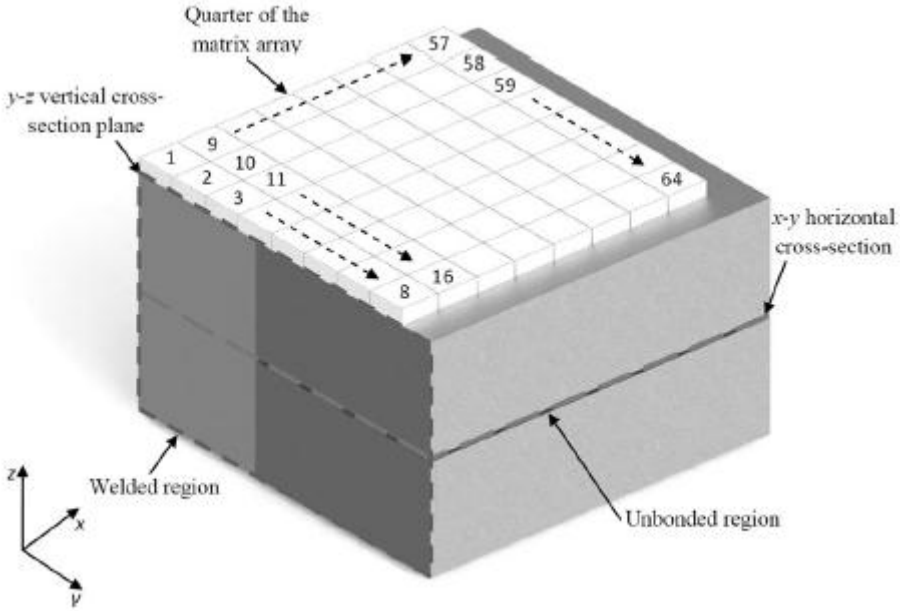
Duomenų kiekis priklauso nuo ultragarsinių sensorių kiekio ir tipo. Šio darbo metu buvo testuojama su 64x64 ultragarsinių sensorių masyvo duomenimis [39].



2.20 pav. Ultragarsinių sensorių aplitudės pokytis per laiką

Naudotas algoritmas optimizuotas šio tipo sensorių duomenims. Ultragarsinių tomografinių SAFT signalų laikinė diagrama pateikta paveikslėlyje (žr. 2.20 pav). Signalai buvo išgaunami *Matlab* posistemėje panaudojant straipsnio autorių eksperimentinius duomenis [39], kurių ilgiai yra 64x64x2402 ir pateikiami *Matlab* kodavimo pavidale.

Eksperimentinis bandinys susideda iš dviejų suvirintų metalo plokščių, kurios yra pateikiamos paveiksle (žr. 2.21 pav.) [39].



2.21 pav. Tiriamasis bandinys susidedantis iš dviejų suvirintų plokščių

Atliekant simuliaciją [39], buvo dedamas matricos masyvas tiesiai ant suvirinto lakštinio paviršiaus be jokios sukabinimo laikmenos *(žr. 2.21 pav.) [39]*. Ketvirtis masyvo buvo 4,5 mm x 4,5 mm, o kiekvienas ultragarsinis elementas turėjo 0.5625 mm x 0.5625 mm atstumus[39]. Buvo daroma prielaida, kad tarpų tarp elementų nėra [39]. Modeliavimas buvo atliekamas naudojant ABAQUS versiją 6,10-1 [39].

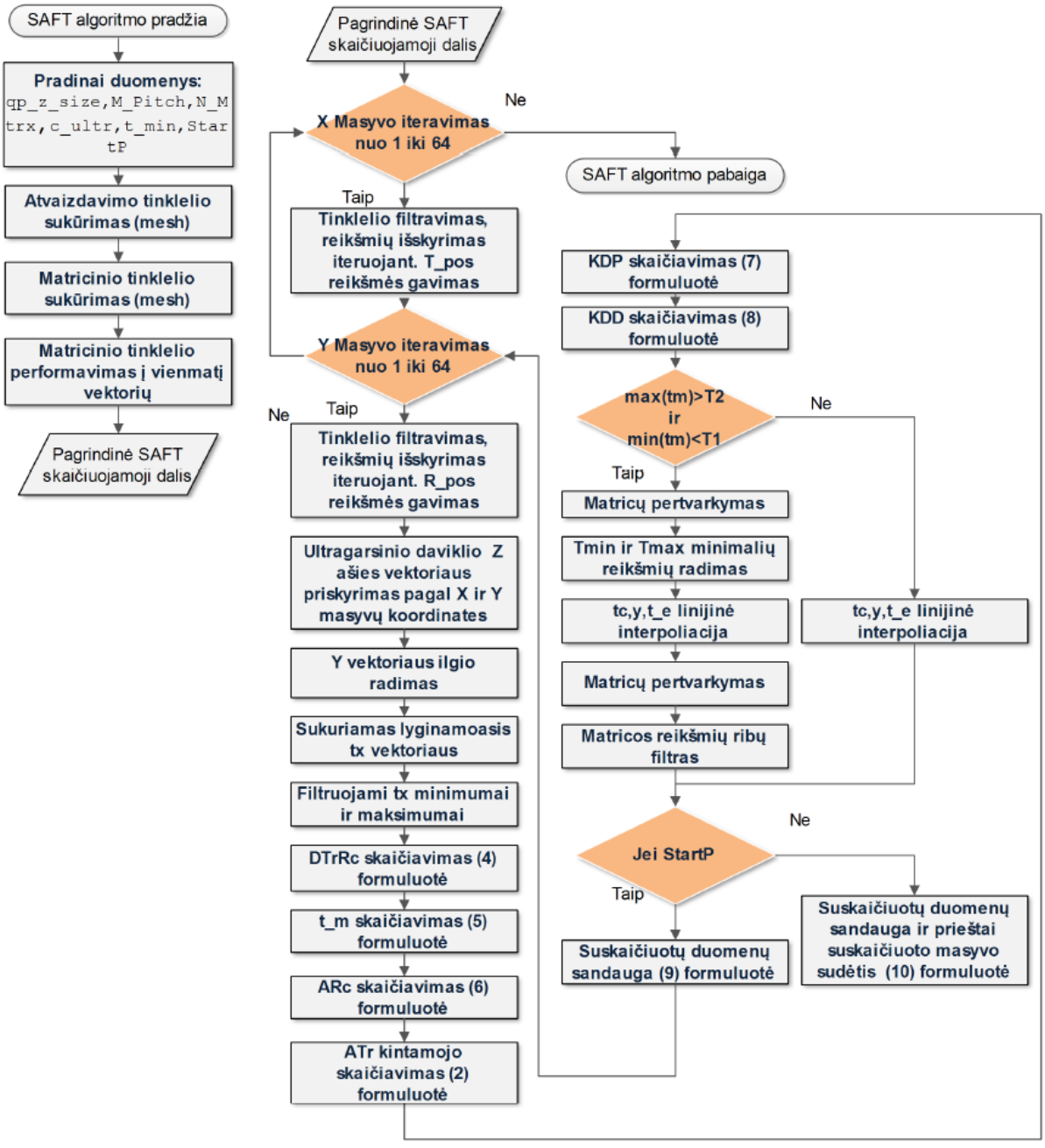
Duomenų saugojimui ir operacijoms tarpusavyje (matricų ir vektorių skaičiavimai), naudojama algebrinių skaičiavimų biblioteka *Blaze,* taip pat modifikuotas kompleksinis sintetinės apertūros fokusavimo algoritmas. Didinant sistemos našumą, naudojama paskirstytų duomenų persiuntimo biblioteka *MPI,* dėl atviro kodo galimybių, plataus pritaikomumo (trecių šalių optimizacijų pritaikomumo) ir didelės spartos. Pasitelkiant šiuos modulius, tokiu būdu skaičiavimai atliekami tinkliniame klasteryje ir tokiu būdu tinkle išskaidomi duomenų elementai.

2.3 Sintetinės apertūros fokusavimo algoritmas centrinio ir grafinio procesoriaus lygmenyje

Atlikus SAFT algoritmo analizę pateikiamas bazinis algoritmas ir algoritmų realizacijos,

kurios apima centrinio processoriaus architektūrą. Pirmu atvėju atliktos pagrindinės SAFT algoritmo optimizacijos, toliau seka realizacijos viename branduolyje ir keliuose.

Pirmosios įžvalgos atliktos algoritmo struktūroje, stengtasi išsiaiškinti kekvieną algoritmo elementą ir taip eliminuoti ar modifikuoti SAFT skaičiavimo elementus taip, kad algoritmo eiga vyktų sparčiau. SAFT algoritmas, pateiktas *(žr. 2.22 pav).*

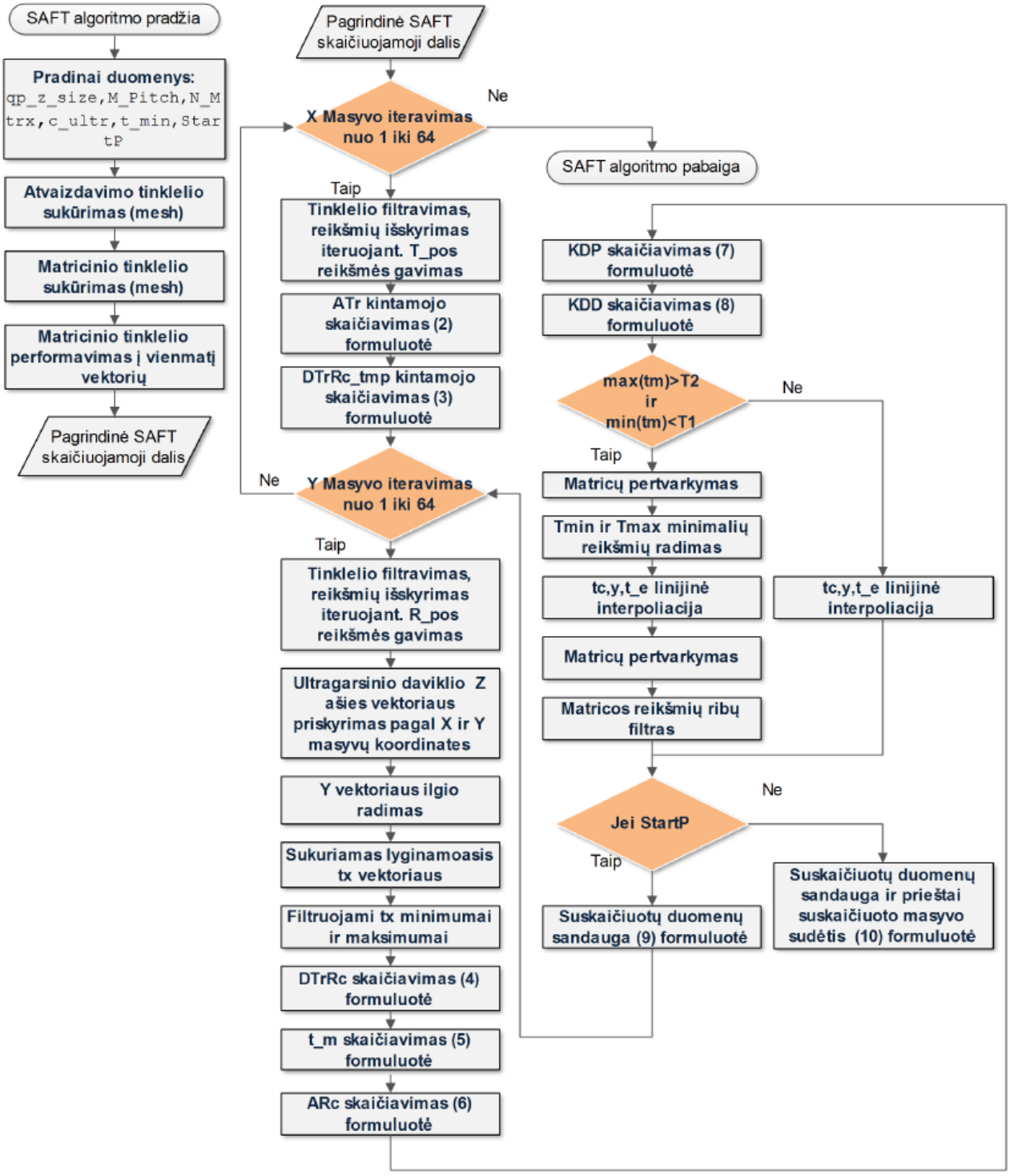


2.22 pav. Pirminis SAFT algoritmas vieno brandolio procesorinei archtektūrai

Siekiant išsiaškinti ir optimizuoti SAFT algoritmą pasirinkta *Matlab* posistemė. Algoritmas pagrįstas matriciniais perrinkimo skaičiavimais. Šio algoritmo pagrindinė optimizacija įžvelgiama ciklinėse dalyse, kurios yra monolitinio tipo. Pagrindinės tolimesnės optimizacijos gali būti atliekamos keičiant ciklines SAFT (*žr. 2.23 pav. masyvų iteracijas*) dalis į *GPU kernel*

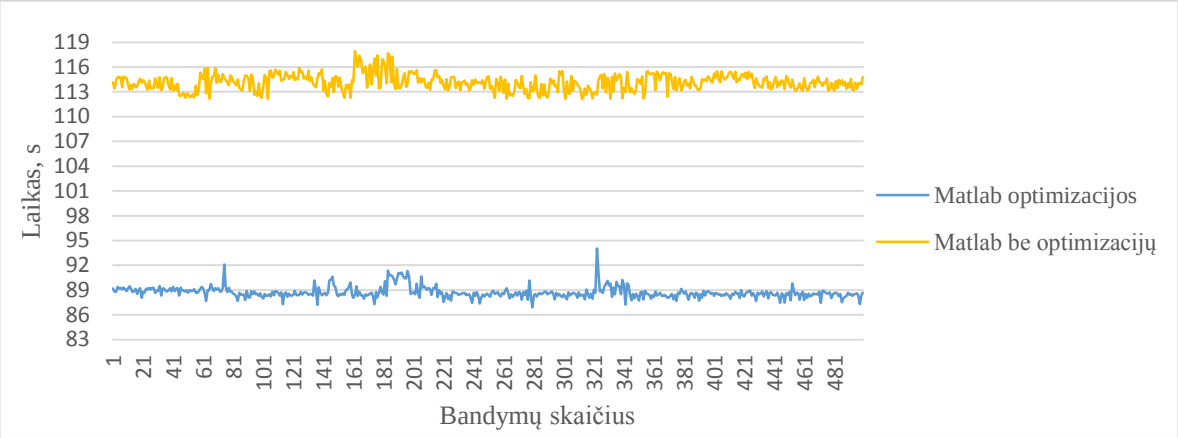
skaičiuojamąsias dalis arba paskirstant ciklines dalis atskiroms skaičiavimo mašinoms.

Darbo metu pastebėtos nereikalingos skaičiavimų iteracijos *(žr. 2.22 pav. ATr bloką).* Taip pat pastebėti skaičiavimai *(žr. 2.22 pav. DTrRc bloką),* kurie atliekami galiniame cikle ir yra kas kart perskaičiuojami. Tam buvo pašalinami nereikalingi SAFT elementai (tinklelio filtravimas, reikšmių išskyrimas, Atr ir DTrRc skaičiavimai *(žr. 2.22 ir 2.23 pav.))*, rezultatas matomas paveiksle *(žr. 2.22 pav).*



2.23 pav. Optimizuotas SAFT algoritmas vieno brandolio procesorinei archtektūrai

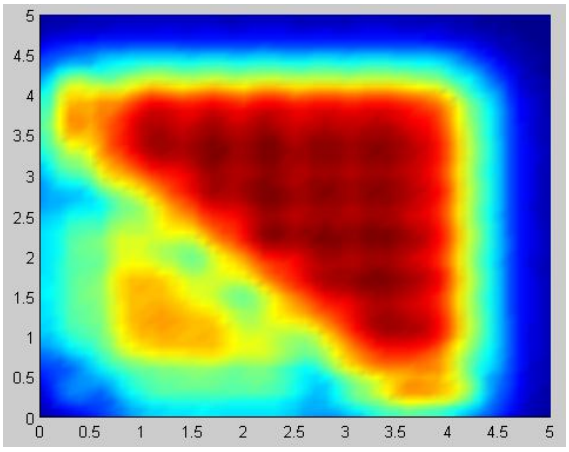
Rezultatų palyginimui buvo sukurtas simuliacijos laiko matavimo programa, kuri įsimena kodo vykdymo pradžios laiką ir kodo vykdymo baigties laiką. Tokiu metodu gaunamas laiko skirtumas, kuris lygus simuliacijos sugaištam laikui.



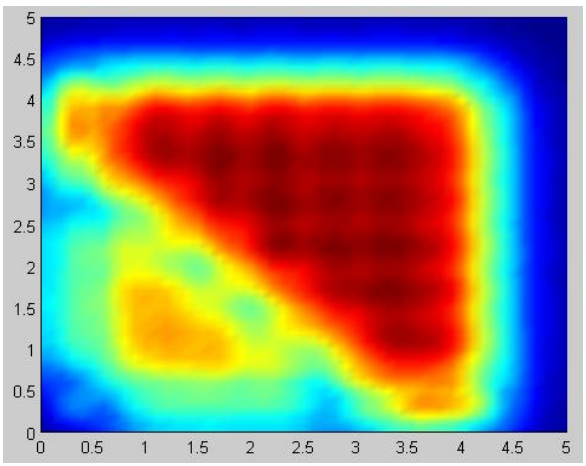
2.24 pav. Algoritmų veikimo laikai, lyginami Matlab be optimizacijų ir su optimizacijomis algoritmai. Testo atlikimui panaudoti penki šimtai skaičiavimo taškų

Optimizuoto ir neoptimizuoto SAFT algoritmo skaičiavimo laikai pateikti paveiksle *(žr. 2.24 pav.)* ir lentelėje *(žr. 2.1 lent)*.

Atlikus algoritmų palyginimą, matoma, kad C-skenavimo bandiniai nesiskiria ir yra grafiškai identiški *(žr. 2.25 ir 2.26 pav.)*.



2.26 pav. C-skenavimo tiriamojo bandinio plokštelių suvirinimo atvaizdavimas (sustiprintu bendro fokusavimo metodu). Pirminis algoritmas



2.25 pav C-skenavimo tiriamojo bandinio plokštelių suvirinimo atvaizdavimas(sustiprintu bendro fokusavimo metodu). Optimizuotas algoritmas

Skaičiavimai buvo atliekami tokių parametrų mašinoje:

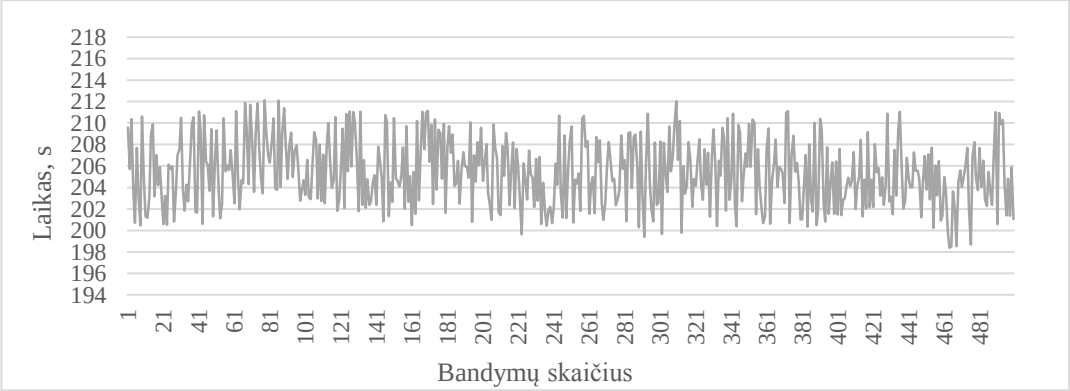
- *Centrinis procesorius*: AMD FX8350 @ 4200 Mhz

- *Darbinė atmintis*: 16 Gb, 2100 MHz
- *Motininė plokštė*: M5A99FX PRO R2.0
    - Lentelė 2.1. Algoritmų veikimo laikai, lyginami *Matlab* be optimizacijų ir su optimizacijomis

SAFT algoritmas	Skaičiavimo laikas, s
Neoptimizuotas	114,003
Optimizuotas	88,706

Stebimas 22,28 % našesnis duomenų apdorojimas, lyginant su pirminiu algoritmu, nenaudojant GPU skaičiavimo galios.

Sekantis žingsnis yra sudaryti ir realizuoti skaičiavimo algoritmą, kuris SAFT algoritmu *(žr. 2.23 pav)* skaičiuoja vieno branduolio *(angl. core)* terpėje. Tikslesniems rezultatams išgauti tiriamas algoritmas buvo leidžiamas 500 kartų ir taip gautas vidutinis skaičiavimo laikas. Laikai pateikiami paveiksle *(žr. 2.27 pav.)*.



2.27 pav. Sintetinės apertūros fokusavimo technikos algoritmo veikimo laikai vieno branduolio procesorinėje architektūroje. Testo atlikimui panaudoti penki šimtai skaičiavimo taškų

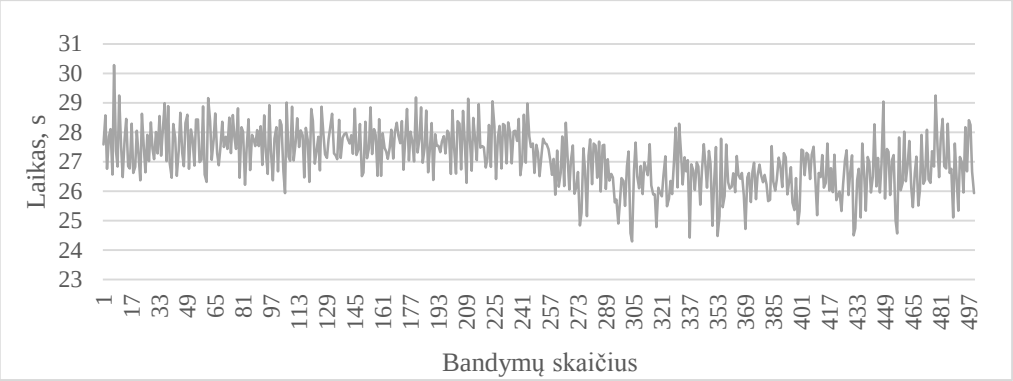
Skaičiavimo laikas yra nemažas (vidutinis skaičiavimo laikas 205,394 sekundės), kadangi skaičiavimas atliekamas vieno branduolio pagalba. Paveiksle *(žr. 2.27 pav.)* matoma, kad skaičiavimo laikai yra nepastovūs ir tai gali būti todėl, kad skaičiavimo mašinoje veikė atskiri *Linux* servisai, kuriems reikalingi resursai. Tyrimui panaudota spartinančioji linijininė algebra armadillo ir Qt C++ terpė su specifinėmis panaudoto procesoriaus instrukcijomis (*QMAKE_CXXFLAGS_RELEASE *= -Ofast -funroll-loops -fomit-frame-pointer -ffast-math -march=native -mtune=native -flto -fwhole-program* ir *QMAKE_CXXFLAGS += -ftree-vectorize -march=native -mtune=native -ftree-vectorize -ftree-vectorizer-verbose=1*).

Atlikus vieno branduolio realizacinį skaičiavimą, toliau seka lygiagretinimas centrinio

procesoriaus branduolių lygyje, tai atliekama OpenMP lygiagrečių paskirstytų skaičiavimų bibliotekos pagalba, pritaikant šias OpenMP direktyvas:

- Centrinio procesoriaus suminio branduolių kiekio išskyrimas. Programinė OpenMP direktyva *(omp_set_num_threads(8));*
- Neapdorotų duomenų iteracijų skaidymas per centrinio procesoriaus branduolius *(angl. cores).* Skirstymo direktyva su parametrais (*#pragma omp parallel for private(i, j, k) shared(YIsum, YI, ARc, t_m, KDD, KDP, DTrRc, ATr, T_pos, R_pos))*
- Apskaičiuoti duomenys renkami *(#pragma omp*) direktyvos pagalba.

Tikslesniems rezultatams išgauti tiriamas algoritmas buvo leidžiamas 500 kartų ir taip gautas vidutinis skaičiavimo laikas. Laikai pateikiami paveiksle *(žr. 2.28 pav.).*



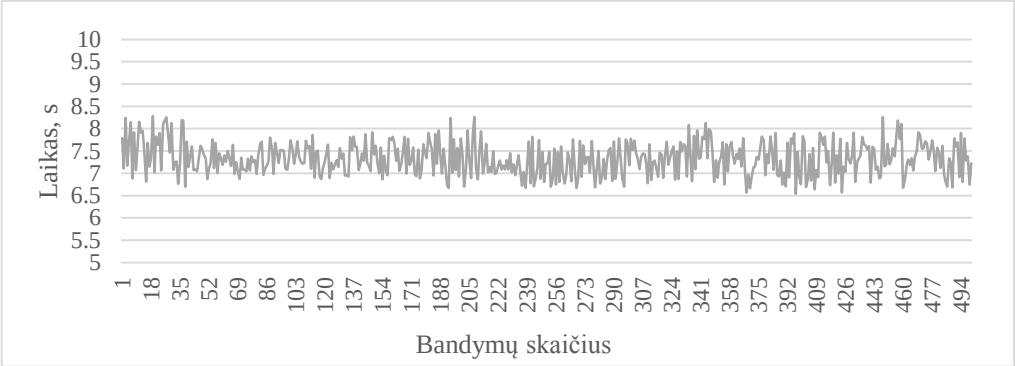
2.28 pav. Sintetinės apertūros fokusavimo technikos algoritmo veikimo laikai kelių (8) branduolių procesorinėje architektūroje. Testo atlikimui panaudoti penki šimtai skaičiavimo taškų

Paveiksle *(žr. 2.28 pav.)* matoma, kad skaičiavimo laikai yra nepastovūs ir tai gali būti todėl, kad skaičiavimo mašinoje veikė atskiri *Linux* servisai, kuriems reikalingi resursai. Skaičiavimo laikas lygiagretinus algoritmą yra 7,57 kartus našesnis, lyginat su vieno branduolio realizacija, panaudojant 8 branduolius. Idealiu atvėju galima būtu skaityti, kad optimizacija turėtų būti aštuonis kartus spartesnė, bet kadangi tarp branduoliniame bendravimo lygmenyje yra vykdomos paskirstymo operacijos, šis laikas realiomis sąlygomis negali būti prilygintas. Vidutinis skaičiavimo laikas yra 27,107 sekundės.

Paskutinis ir svarbiausias sintetinės apertūros fokusavimo technikos optimizavimas galimas panaudojant grafinį procesorių. Šiuo atvėju skaičiavimų lygiagretinimas atliekamas grafinės kortos lygmenyje, nenaudojant centrinio procesoriaus.

Sudarytas ir realizuotas skaičiavimo algoritmas, kuris geba skirstyti skaičiavimo elementus per grafinio procesoriaus branduolius *(angl. kernels).* Tiriamas algoritmas buvo leidžiamas 500 kartų ir taip gautas vidutinis skaičiavimo laikas. Laikai pateikiami paveiksle *(žr. 2.29 pav.).*

Paveiksle *(žr. 2.29 pav.)* stebimas didelis našumas, lyginant su centrinio procesoriaus skaičiavimo geba, tai vyksta dėl grafinio procesoriaus architektūros ir didelio procesorių *(angl. kernels)* kiekio.



2.29 pav. Sintetinės apertūros fokusavimo technikos algoritmo skaičiavimo laikai grafinio procesoriaus architektūroje. Testo atlikimui panaudoti penki šimtai skaičiavimo taškų

Šiuo atvėju *(žr. 2.29 pav.)* naudojama AMD grafinė korta R9 290X su slankiojančio kablelio skaičiuojamaja geba 5632 Gflop (FP32) ir 704 (FP64). Grafinio procesoriaus skaičiavimų vidutinis laikas yra 7,334 (64 grafiniai branduoliai). Palyginus su centrinio procesoriaus vienu branduoliu, grafinio procesoriaus vienas branduolys yra 2,28 karto lėtesnis, bet kadangi grafinis procesorius šiuo atvėju turi 64 branduolius, skaičiavimo laikas vykdomas greičiau nei centrinio procesoriaus atėju, kuris yra apribotas aštuoniais branduoliais.

2.4 Skaičiavimų paskirstymas tinklo mazguose

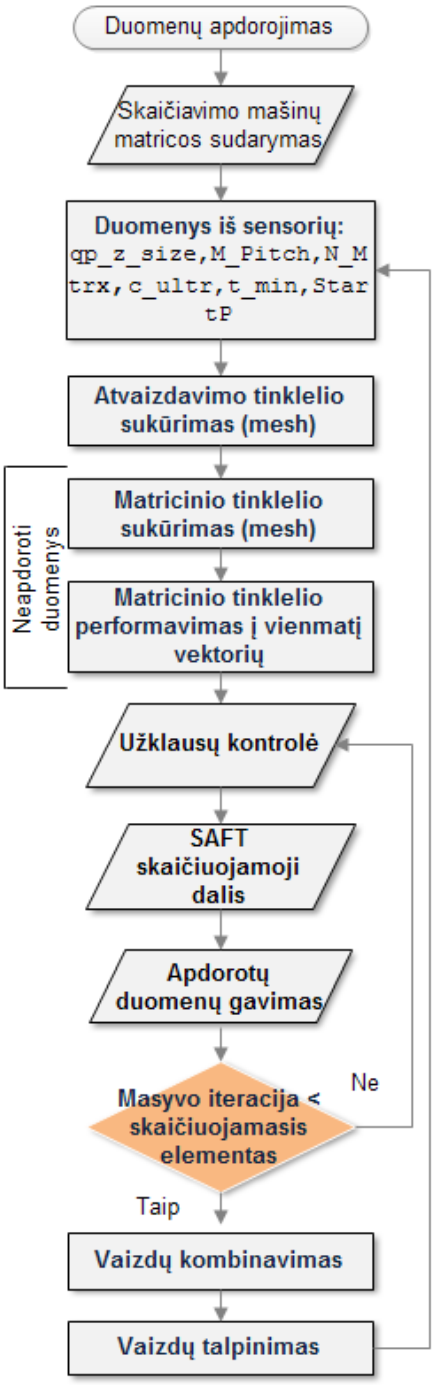
Modifikuotas ir sumodeliuotas tinklinis sintetinės apertūros fokusavimo technikos algorimas apima šiuos elementus ir standartus:

- Linijų (atskirų skaičiavimo modulių) aptikimas. Algoritmas atlieka priskirtų prievadų paiešką. Algoritmui radus priskirtus sisteminius prievadus tikrina antraštes, kuriose pažymima SAFT sistema;
- Linijų (atskirų skaičiavimo modulių) balansavimu, t.y. užklausiamos linijų apkrautumas ir nustatoma reali galima skaičiuojamoji galia. Šiuo atveju atliekamos užklausos kiekvienai priskirtai skaičiavimo mašinai. Linijos gavusios užklausas išsiunčia atsaką pagrindinei paskirstymų mašinai su esama skaičiuojamąja geba;
- Neapdorotų *raw* (iš ultragarsinių sensorių) duomenų skaidymas į aukštos spartos HDF standartą *(žr. 1.2 potėmė);*
- Skaičiavimo mašinų užklausų priėmimas ir prioritizavimas, priklausomai nuo skaičiavimo mašinos aparatinės dalies resursų ir realios galimos skaičiuojamosios

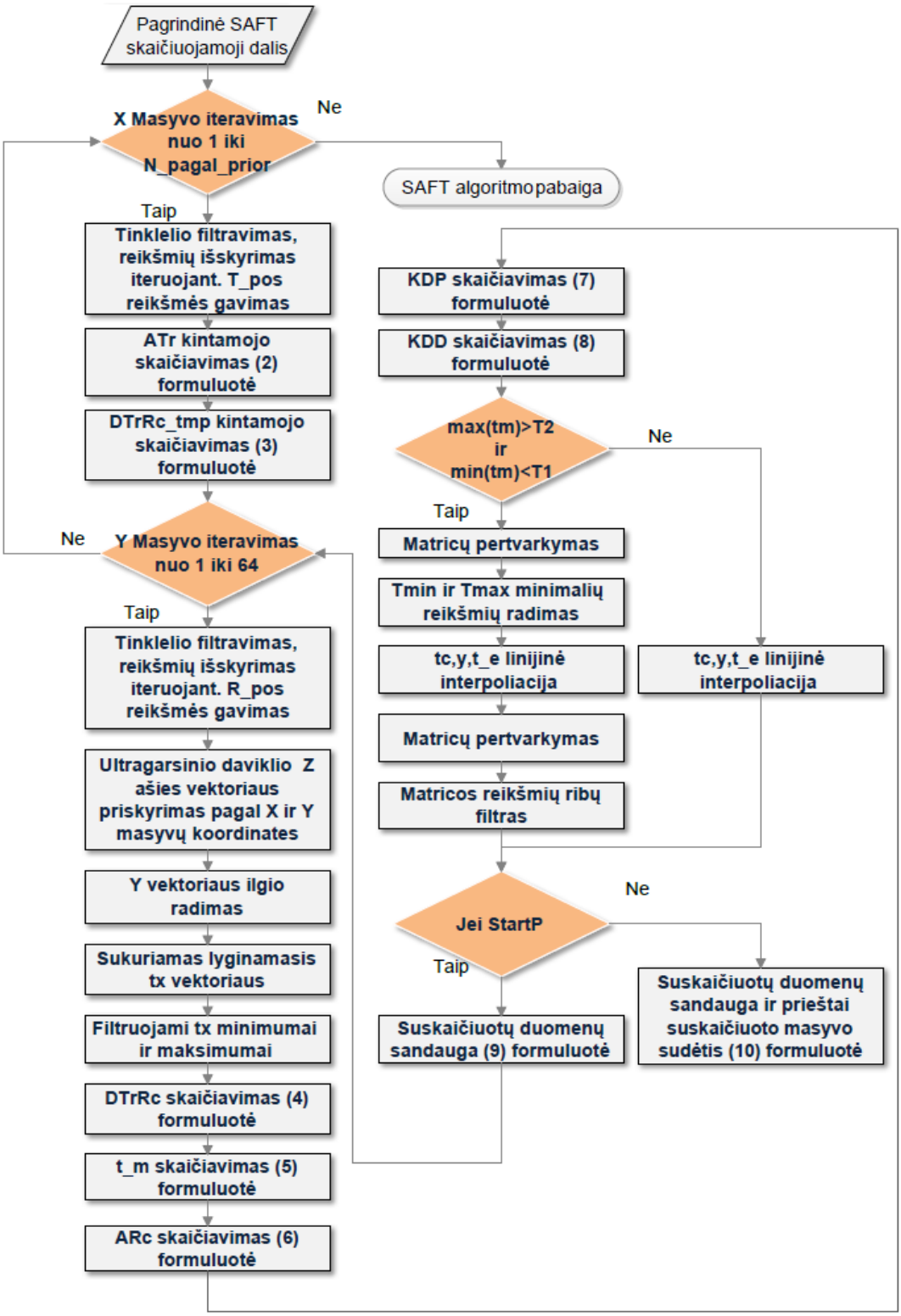
gebos;

- Vaizdų generavimo. Baigus vieno kadro skaičiavimus sukuriama atskira vykdymo paprogramė, kuri sukombinuoja einamąjį kadrą ir perduoda statiniam talpinimui tinkle. Šioje dalyje galimai panaudojami atviro kodo *(angl. OpenSource)* servisai su bazine konfigūracija: *Apache2*, *Nginx*.

Pagrindinis paskirstytų skaičiavimų SAFT algoritmas pateiktas paveiksle *(žr. 2.30 pav)*, jo sekamieji algoritmai pateikti atskiruose paveiksluose *(žr. 2.31-2.33 pav)*.



2.30 pav. Pagrindinis valdymo, užklausų priėmimo ir SAFT skaičiavimo algoritmas



2.31 pav. SAFT algoritmas modifikuotas paskirstytų skaičiavimo mašinai

Sekant paeiliui, algoritmo pagrindas sudarytas iš atskirų modulinių algoritmų. Kiekvienas gali būti lygiagretinamas CPU architektūroje, efektyviausiai jeigu paruošiamieji skaičiavimo vektoriai ir matricos, išskyrus SAFT skaičiavimo algoritmas (SAFT skaičiuojamoji dalis) *(žr. 2.29 pav.)*, funkcionuotų CPU sluoksnyje. Visus kitus ilgai trunkančius skaičiavimus, tokius kaip pagrindinis SAFT algoritmas perkeliamas į GPU apdorojimo dalį. Sukomponuota tokia architektūra

pasižymėtų našia skaičiuojamąja galia ir aukštu aparatūrinės įrangos panaudojimu.

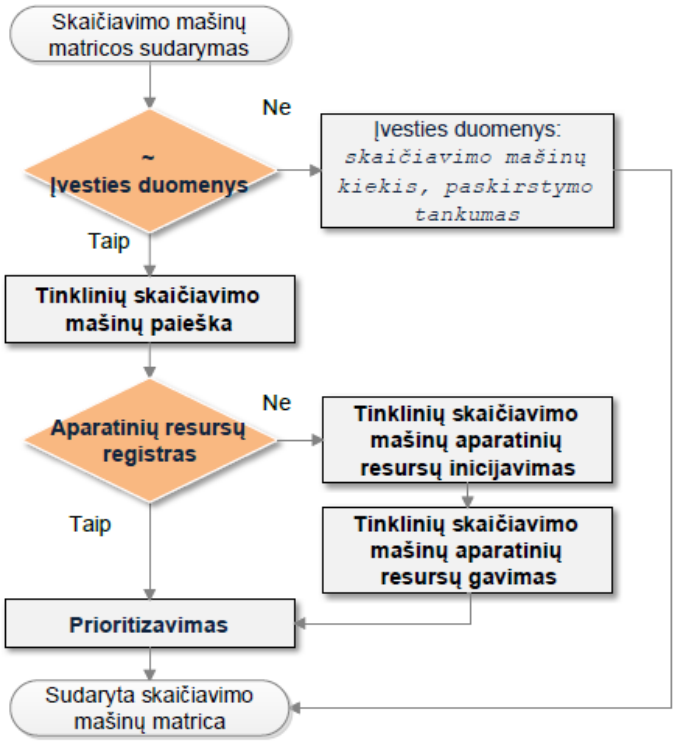
Algoritmo dalis, kuri atsakinga už SAFT duomenų rušiavimą ir skaičiavimą pateikta paveiksle *(žr. 2.29 pav.)*.

SAFT algoritmo pagrindinė skaičiuojamoji dalis ženkliai nesikeičia *(žr 2.22 pav. ir 2.30 pav.)*, kadangi duomenys yra skaidomi skirtingoms skaičiavimo mašinoms, paliekant tą patį skaičiavimo modelį.

Pagrindinis pakeitimas atliekamas pirminiame cikliniame laipsnyje *(žr 2.30 pav)*. Jis gali skirtis, priklausomai nuo mašinos darbingumo prioriteto.

Vienas iš svarbių šios sitemos elementų yra duomenų paskirstymas, priklausomai nuo skaičiuojamųjų mašinų resursinių parametrų. Tai leidžia balansuoti sistemos apkrautumą išlošiant skaičiavimo efektyvumą.

Algoritmo dalis, kuri atsakinga už duomenų paskirstymą ir balansavimą pateikta paveiksle *(žr. 2.30 pav.)*



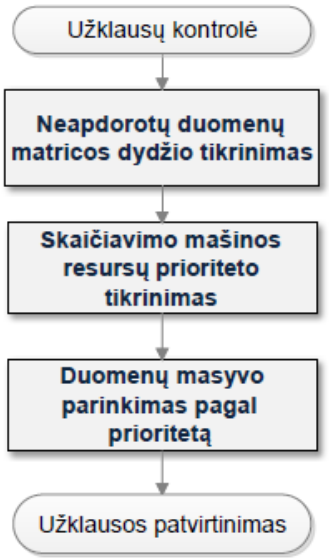
2.32 pav. Resursų balansavimo algoritmas

Kadangi sistemos išpildymo atvėju sistema privalo funkciuonuoti tinkle, todėl turi vykti bendravimas tarp atskirų skaičiavimo mašinų. Šiuo atveju yra sudaromos MPI jungtys SSH metodika su kitomis MPI mašiniomis, panaudojant modifikuotus MPI skaidymo ir priėmimo algoritmus. Jungtys sudaromos priskirtais statiniais prievadais, sisteminiais vardais (*angl.*

hostnames) ir autetifikaciniais SSH RSA raktais. Minėti parametrai nustatomi inicijuojant skaičiavimo algoritmą.

Bendravimui tarp atskirų mašinų sudarytas algoritmas *(žr. 2.31 pav.),* kuris:

- Atlieka instrukcines užklausas į nepdorotų HDF duomenų failinę duombazę, kuri laiko neapdorotus *(angl. raw)* SAFT duomenis;
- Prioriteto palyginimai. Aukštesnio prioriteto skaičiavimo mašina gali perimti didesnio dydžio masyvą, taip tolygiai išlaikant skaičiavimo laikus tarp mašinų ir mašinų skaičiavimo elementų (grafinio procesoriaus branduolių *(angl. kernel)* ir centrinio procesoriaus*);*
- Duomenų masyvo parinkimas pagal prioritetą iš HDF failinės duomabzės.



2.33 pav. Užklausų kontrolė

3. Kombinuoto skaičiavimo realizacija

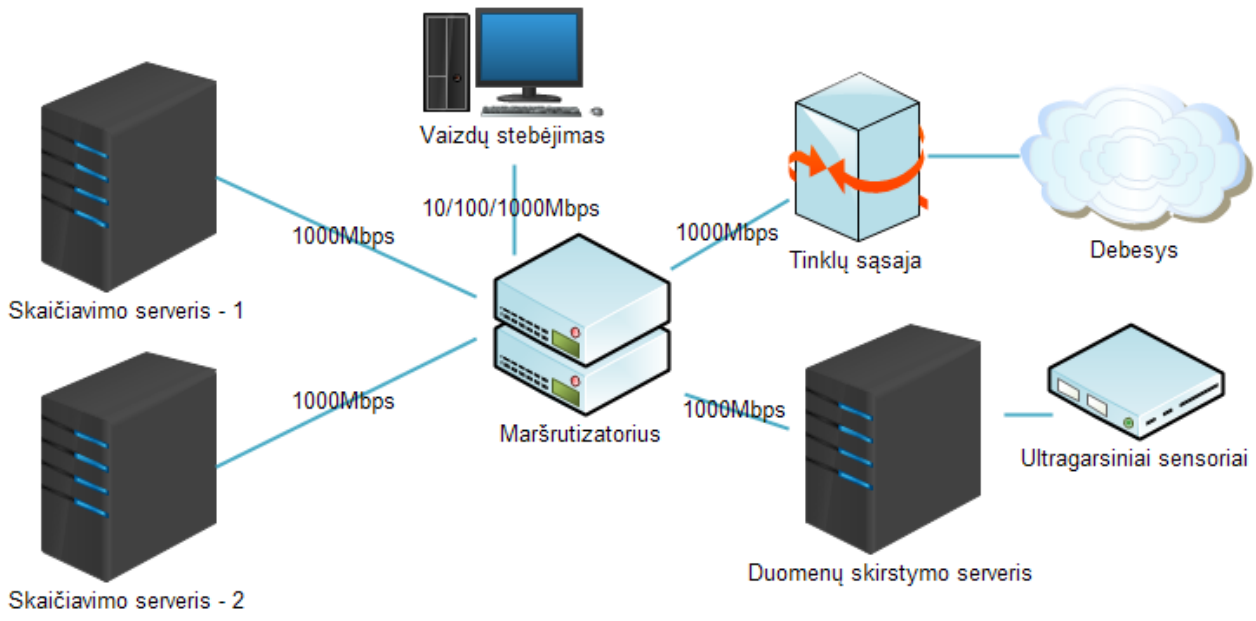
Realizacija vykdoma remiantis atliktų tyrimų rezultatais. Pagrindinės sintetinės apertūros fokusavimo technikos algoritmo skaičiavimo mašinos dedamieji komponentai yra hibridinis skaičiavimas, išskaidant rekursinius algoritmus ir nepriklausomus vienas kitam lygiagrečius skaičiavimus:

- Duomenų surinkimas, matricinių tinklelių paruošimas, skirstymas ir surinkimas tinkle - centrinis procesorius;
- Pagrindiniai SAFT skaičiavimo ciklai – grafinis procesorius.

Eksperimentavimo ir realizavimo platformai pasirinkti du identiški UNIX serveriai su šiais parametrais:

- *Centrinis procesorius*: AMD FX8350 @ 4200 Mhz, 8 branduoliai;
- *Darbinė atmintis*: 16 Gb, 2100 MHz, dvi atminties plokštės;
- *Motininė plokštė*: M5A99FX PRO R2.0;
- *Grafinė korta*: AMD R9 290X;
- *Tinklo korta*: Intel Gigabit CT PCI-E EXPI9301CTBLK.

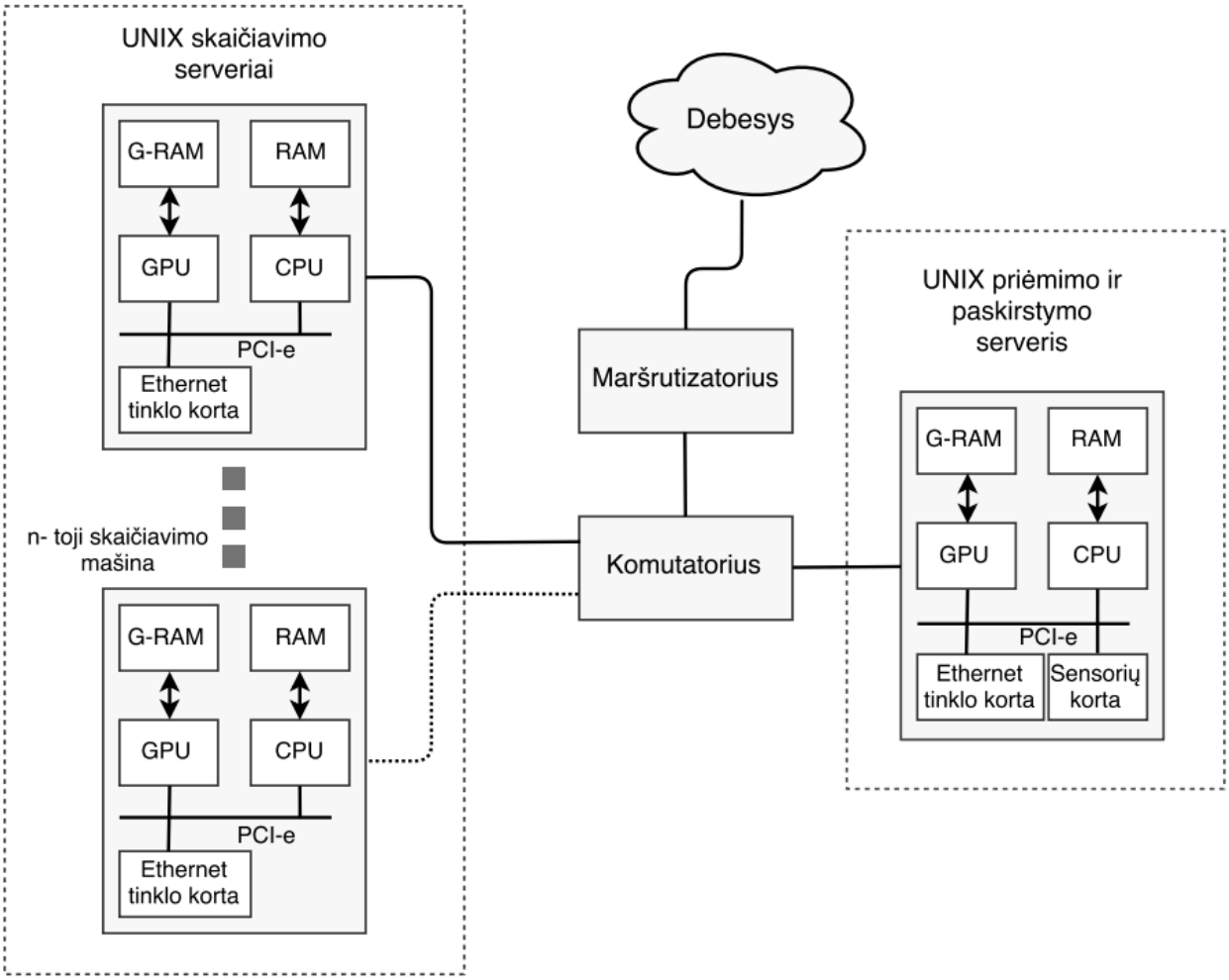
Realizacijai naudojamas tinklas *(žr. 3.1 pav.)* yra standartinis aukštos/vidutinės *(angl. gigabit)* pralaidos vietinis tinklas, kuris turi skaičiavimų paskirstymo išskaidymą į debesis. Šiame realizavime debesys yra vaizduojami kaip galimybė apjungti nutolusias mašinas tinkle.



3.1 pav. Realizacijos tinklas. Siūlomo ultragarsinės tomografijos duomenų surinkimo tinklo

struktūra susideda iš dviejų POSIX serverių ir vietinės tinklo infrastruktūros

Bendrinė sistemos struktūra pateikta paveiksle *(žr. 3.2 pav.).* Struktūra apsprendžia bendrinius sudedamųjų dalių blokus.



3.2 pav. Bendrinė skaičiavimo sistemos struktūra

Skaičiavimo sistemos struktūrą sudaro šie elementai:

- *Ultragarsinių sensorių masyvas.* Siunčia ir priima ultragarsines bangas pagal SAFT metodiką;
- *Analogas kodas keitiklis.* Analoginio signalo vertimas į skaitmeninę formą (ši funkcija yra priklausoma nuo ultragarsinių sensorių, tam tikrais atvejais ši opcija nebereikalinga);
- *Paskirstymo mašina.* Apsprendžia skaičiuojamąją gebą, matricinių neapdorotų duomenų paskirstymo funkciją, realaus laiko vaizdų rekonstravimą ir valdymą;
- *Skaičiavimo mašinų klasteris.* Pagrindinis skaičiavimo masyvas. Atliekami kompleksiniai *SAFT* algoritminiai skaičiavimai, duomenų grąžinimas į paskirstymo mašiną vykdomas per SSH komunikavimo metodiką;

- *Tinklo komutatorius.* Įrenginių komutavimo blokas;

3.1.1 Duomenų paskirstymo ir vaizdų atkūrimo mašina

Paskirstymo mašina yra pagrindinis valdomasis tinklo segmentas, kuris atlieka skaičiavimų balansavimą ir pagrindinį skaičiavimų paskirstymą tinklinėms mašinoms. Bendrai paskirstymo mašina apsprendžia šiuos punktus:

- Duomenų surinkimą iš ultragarsinių sensorinių masyvų;
- Ultragarsinių sensorių duomenų matricos išskaidymą į atskirus segmentus tolesniam apdorojimui ir paskirstymui;
- Duomenų skaidymą hierarchinio duomenų standarto metodu;
- Paskirstytų skaičiavimų darbo režimo palaikymą;
- Tinklinių skaičiavimo mašinų skaičiavimo galios įvertinimą;
- Tinklinių skaičiavimo mašinų veiklos stebėseną.

Paskirstymo mašina remiasi POSIX IEEE 1003 architektūra. Sistemos lankstumui tai gali būti bendros priegos kompiuteris, kuris privalo tenkinti minimalius parametrus:

- x86-64 (AMD64) procesoriaus architektūros palaikymą;
- Mažiausiai *4 Gb* darbinės atminties *(angl. RAM);*
- 20 Gb laisvos kietojo disko atminties *(angl. HDD);*
- Atskiro grafinio procesoriaus posistemė *(angl. GPU),* palaikanti *OpenCL* programinę biblioteką;
- RS232 prievado *(angl. Port)* palaikymą;

Siekiant paprastumo ir pritaikomumo, tomografinių vaizdų stebėjimas vykdomas per WEB naršykles, kurių pagalba stebimos amplitudės gilumas išreikštas spalvomis.

Vaizdų persiuntimas vykdomas iš paskirstymo mašinos, kurioje veikia Apache2 servisas su mysql duomenų baze. Vaizdų generavimui panaudojama laisvai prieinama *(angl. Open source)* Codeflow programinė biblioteka, paremta JavaScript programine kalba.

3.1.2 Neapdorotų duomenų struktūrų skirstymas

Duomenų priėmimas vykdomas per paskirstymo mašiną. Duomenys priimami atskiro prievado pagalba. Duomenų formatas yra nekoduotas ir nesuspaustas. Tokio formato duomenis perduoti tinklu yra neefektyvu ir nepraktiška. Todėl yra spaudžiami ir indeksuojami hierarchinio duomenų formato pagalba. Tai vykdoma programiniu būdu *(žr. 2.1 Hierarchinis duomenų formatas potemė).*

Siekiant efektyviai išnaudoti klasterines skaičiavimo mašinas, reikalingas dinamiškai

prisitaikantis paskirstymo algoritmas, kuris priklausomai nuo homogeninio tinklo parametrų yra konfigūruojamas per vartotojo sąsają.

3.1.3 Lygiagretaus skaičiavimo tinklinės mašinos

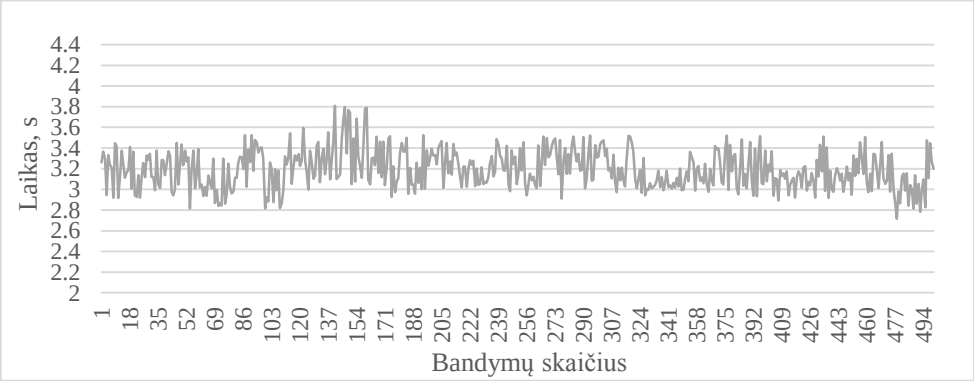
Šių mašinų paskirtis yra vykdyti kodą, kuris komunikuoja su paskirstymo mašina ir atlieka priskirtų matricų skaičiavimus. Komunikacijai naudojamas *Gigabit* pralaidos standartas, užtikrinant nedidelius tinklo vėlavimus tarp skaičiavimo įrenginių.

Skaičiavimų įrenginių kiekis yra ribojamas tinklo įrangos standartų ir atstumų tarp tinklo įrangos.

Skaičiavimo klasterinės mašinos programinis algoritmas skiriasi nuo paskirstymo mašinos, jis neturi keletos valdymo funkcijų. Veikia kaip belaisvis *(angl. Slave).*

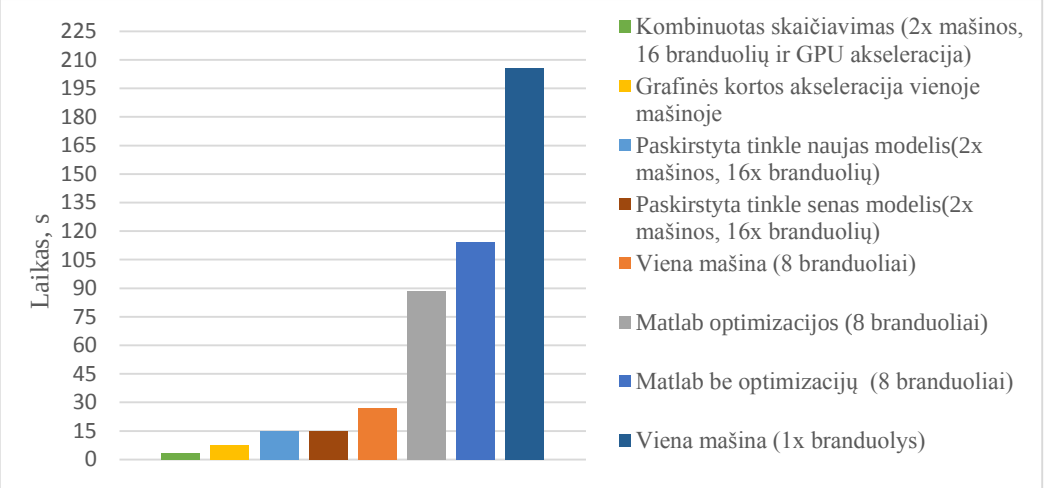
3.1.4 Lyginamieji laikai

Realizuotos skaičiavimo sistemos laikai pavaizduojami paveiksle *(žr. 3.3 pav.).* Realizacija įgivendinta panaudojant kombinuotą skaičiavimą telekomunikacijų tinkle. Tiriamas algoritmas buvo leidžiamas 500 kartų, tokiu būdu gautas vidutinis skaičiavimo laikas.



3.3 pav. Kombinuoto (skaičiavimų paskirstymas telekomunikaciniame tinkle apjungiant centrinį procesorių ir grafinę kortą skaičiavimams) skaičiavimo laikai. Testo atlikimui panaudoti penki šimtai skaičiavimo taškų

Sukurtos kombinuotos sistemos skaičiuojamoji geba paskirstant tarp dviejų tinklo segmentų yra 4,65 karto našesnė lyginat be grafinės kortos. Paveiksle *(žr. 3.4 pav.)* pateikiamas grafinis laikų išsibarstymas realizuojant skirtinguose skaičiavimo platformose.



3.4 pav. Grafinis laikų išsibarstymas realizuojant skirtinguose skaičiavimo platformose

Skaitiniu formatu pateikiami *(žr. 3.1 Lent.)* apibendrinti skaičiavimo laikai realizuojant skirtinguose skaičiavimo platformose.

Lentelė 3.1. Skaičiavimo vidutiniai laikai pateikiami sekundėmis (skaičiavimų iteracijų kiekis – 500 kartų)

Matlab be optimizacijų (8 branduoliai)	114,149
Matlab optimizacijos (8 branduoliai)	88,706
Viena mašina (1x braduolių)	205,395
Viena mašina (8x branduoliai)	27,108
Tinklinis skaičiavimas senu modeliu (2x mašinos, 16x branduolių)	14,788
Tinklinis skaičiavimas nauju modeliu (2x mašinos, 16x branduolių)	14,752
GPU viena mašina	7,335
Kombinuotas skaičiavimas(2x mašinos, 16 branduolių ir GPU akseleracija)	3,193

Bendru atveju kombinavimo metodika pasiūlyti metodai spartino skaičiavimus 35,74 kartais panaudojant dvi paskirstytų skaičiavimų tinklines mašinas.

Išvados

1. Darbo metu optimizuotas, duomenų apdorojimo algoritmas, kuris užtikrina spartesnį sintetinės apertūros fokusavimo technologijos generuojamų duomenų srauto apdorojimą ir atvaizdavimą. Taip pat nuspręsta sistemos architektūra.
2. Išsiaiškinta jog naudojant monolitinę architektūra nėra tolygiai paskirstomos sintetinės apertūros fokusavimo užduotys dėl neoptimizuoto *Matlab* kodo. Todėl simuliavimas trunka 114,149 sekundes. Analizuojant sintetinės apertūros fokusavimo algoritmą, ciklinėse iteracijose pastebėti nereikalingi skaičiavimai, kurie prailgina simuliacijos laiką. Gauti rezultatai leido nekeičiant galutinio rezultato 22,28% atlikti simuliaciją greičiau.
3. Duomenų apsikeitimo tarp skirtingų telekomunikacinių mašinų tyrimai parodo, kad naujai siūlomi modeliai tam tikrais atvėjais yra pranašesni panaudojant MST ir mašinų koeficientų matricą. Redukcijos, išsibarstymo ir rinkimo modelis veikė prasčiausiai. Redukcinės-sklaidos pasiūlytas modelis veikia 1,829 kartų greičiau trumpose žinutėse ir ilgose žinutėse 1,684 karto nei senasis modelis. Transliacijos modelio vykdymo laikas mažėja didinant pranešimų skaičių ir pasiekiamas iki 1,223 karto sparta. Tai labiausiai matoma kai pranešimo ilgis siekia 10Mb ir daugiau. Tinklelio rinkimo ir redukavimo modelis turi pranašumą tik trumpuose pranešimuose, spartos pagreitėjimas siekia iki 1,347 kartų. Paskutinis tirtas visi visiems modelis rodo spartos didėjimą ilguose pranešimuose didėjant pranešimų kiekiui nuo 9,57Mb, spartos augimas matomas iki 1,329 karto;
4. Pasiūlytas kombinuoto skaičiavimo algoritmas, kuris remiasi skaičiavimų išskirstymu telekomunikaciniams mazgams. Algoritmas paremtas mašinų spartos matricos sudarymo metodika, tokiu atveju duomenų išsiuntimo kiekis yra balansuojamas tinklo segmentams. Manoma, kad mažiau nei 16 SAFT sensorių kiekiui nėra prasmės skaidyti tinklinėje infrastruktūroje panaudojant šiuolaikinius serverių resursus.
5. Remiantis atliktais tyrimais buvo atliekama realizacija panaudojant sparčiausią modelį (redukcinės-sklaidos), pritaikant sintetinės apertūros fokusavimo algoritmą tinklinei paskirstytų skaičiavimų mašinai bei panaudojant grafinį procesorių. Šios realizacijos metu sukurta kombinuota sistema veikianti tinklinėje telekomunikacinėje infrastruktūroje.

Literatūros sąrašas

1. Tong Jian-Hua, Liao Shu-Tao: Using the Synthetic Aperture Focusing Technique with Elastic Wave Method to Detect the Defects inside Concrete Structures. 2014: konferencijos medžiaga.-Prague, Czech Republic.
2. Lui A. (2012). Development of an Ultrasonic Linear Phased Array System for Real-time Quality Monitoring of Resistance Spot Welds. Master Thesis. National Library of Canada.
3. Bazulin E. G. 2013. On the Possibility of Using the Maximum Entropy Method in Ultrasonic Nondestructive Testing for Scatterer Visualization from a Set of Echo Signals. Acoust. Phys. 59 (2), 210–227.
4. Jeong J. S. 2014. Dual-Element Transducer with Phase-Inversion for Wide Depth of Field in High-Frequency Ultrasound Imaging. Sensors, 14: 14278-14288; doi:10.3390/s140814278
5. Chakroun N., Fink M., Wu F. 1995. Time reversal processing in ultrasonic nondestructive testing IEEE Trans. Ultrason. Ferroelectr. Freq. Control. 42, 1087–98
6. Skjelvareid, Martin H., et al. 2011. Synthetic aperture focusing of ultrasonic data from multilayered media using an omega-k algorithm. Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on 58.5 (2011): 1037-1048.
7. Lane C. 2014. The Development of a 2D Ultrasonic Array Inspection for Single Crystal Turbine Blades. United Kingdom. Springer. 116 p.
8. Calmon P, Mahaut S, Chatillon S, Raillon R. 2006. CIVA: an expertise platform for simulation and processing NDT data. Ultrasonics, 44: 975 – 979.
9. Cochran. S. 2006. Fundamentals of ultrasonic phased arrays. Insight, 48(4): 212-217
10. Yan M., Mcgrath R. E., Folk M. 2004. Performance Study of HDF5-WRF IO modules, WRF Workshop.
11. The HDF Group. HDF5 File Format Specification Version 3.0. 2016. Žiūrėta: 2016 01 10. Priega per internetą: < https://www.hdfgroup.org/HDF5/doc/H5.format.html>
12. Michelson D. B., Lewandowski R., Szewczykowski M., Beekhuis H. 2014. EUMETNET OPERA weather radar information model for implementation with the HDF5 file format. The EUMETNET OPERA weather radar information model presented in UML diagrams, 1-38.
13. Howison M., Koziol Q., Knaak D., Mainzer J., Shalf J. 2010. Tuning HDF5 for Lustre File Systems, in Proceedings of 2010 Workshop on Interfaces and Abstractions for Scientific Data Storage „IASDS10“, LBNL-4803E.

14. Geist A., Gropp W., Huss-Lederman S., Lumsdaine A., Lusk E., Saphir W., Skjellum T., Snir M. MPI-2 Extensions to the MessagePassing Interface. Message Passing Interface Forum. 1997. – 376 p.
15. Träff J. L., Gropp W., Thakur R. 2007. Self-Consistent MPI Performance Requirements, in Proceedings of the 14th European PVM/MPI Users' Group Meeting „Euro PVM/MPI 2007", 36-45.
16. Byna S., Gropp W., Sun X., Thakur R. 2003. Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost, in Proceedings of the IEEE Int'l Conference on Cluster Computing „Cluster 2003", 412-419.
17. Thakur R., Rabenseifner R., Gropp W. 2005. Optimization of Collective Communication Operations in MPICH. Int'l Journal of High Performance Computing Applications, (19)1:49-66.
18. Gropp W., Thakur R. 2007. Revealing the Performance of MPI RMA Implementations, in Proceedings of the 14th European PVM/MPI Users' Group Meeting „Euro PVM/MPI 2007". 272-280.
19. Pješivac-Grbovic J., Angskun T., Bosilca G., Fagg G. E., Gabriel E., Dongarra J. J. 2005. Performance analysis of mpi collective operations, in Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium „Proceedings 2005".
20. Faraj A., Yuan X., Patarasuk P. 2007. A Message Scheduling Scheme for All-to-all Personalized Communication on Ethernet Switched Clusters. IEEE Transactions on Parallel and Distributed Systems, 18(2):264-276.
21. Hasanov K., Lastovetsky A., 2015. Hierarchical Optimization of MPI Reduce Algorithms, in in Proceedings of Parallel Computing Technologies: 13th International Conference „PACT 2015", 21-34.
22. Silva P., Silva J. G. 1999. Implementing MPI-2 Extended Collective Operations, in Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 125-132.
23. Hofmann M., Runger G. 2008. MPI Reduction Operations for Sparse Floating-point Data , in Conference: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting, 7-10.
24. Balaji P., Kimpe D. 2013. On the reproducibility of MPI reduction operations, in Proceedings - 2013 IEEE International Conference on High Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing „EUC 2013", 407-414.

25. Dongarra J., Walker D., Lusk E., Knighten B., Snir M., Gropp W., Geist A., Otto S., Hempel R., Cownie J., Skjelumn T., Clarke L., Littlefield R., Sears M., Huss-Lederman S. 2015. Message Passing Interface Forum „ MPI: A Message-Passing Interface Standard Version 3.1", 1-868.
26. Alamasi G., Archer C. J., Erway C. C., Heidelberger P., Martorell X., Moreira J. E., Steinmacher-Burow B., Zheng Y. 2005. Optimization of MPI collective communication on BlueGene/L systems, in Proceedings of the 19th Annual International Conference on Supercomputing, „ICS 2005", 253-262.
27. Woodall T. S., Graham R. L., Castain R. H., Daniel D. J., Sukalski M. W., Fagg G. E., Gabriel E., Bosilca G., Angskun T., Dongarra J. J., Squyres J. M., Sahay M., Kambadur P., Barrett B., Lumsdaine A. 2004. Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations, in Proceedings, 11th European PVM/MPI Users' Group Meeting „Euro PVM/MPI 2004**",** 105-111
28. Graham R. L., Bosilca G., Pješivac-Grbovic J. 2007. A Comparison of Application Performance Using Open MPI and Cray MPI, in Conference of Cray Users Group „CUG'07", 1-10.
29. Barrett B. W., Brightwell R., Squyres J. M., Lumsdaine A. 2006. Implementation of open mpi on the cray xt3, in Proceedings of 46th CUG Conference „CUG Summit 2006".
30. Iglberger K., Hager G., Godenschwager C., Scharpff T. The Blaze library. 2016. Žiūrėta: 2016 02 10. Prieiga per internetą: <https://bitbucket.org/blaze-lib/blaze/>
31. Iglberger K., Hager G., Treibig J., R¨ude U. 2012. Expression Templates Revisited: A Performance Analysis of the Current ET Methodology. SIAM Journal on Scientific Computing 34(2), 42-69 p. DOI: 10.1137/110830125
32. Advanced Micro Devices Inc. Introduction to OpenCL™ Programming. Training Guide.
33. Karimi K., Dickson N. G., Hamze F. A Performance Comparison of CUDA and OpenCL. arXiv:1005.2581v3. 2010. - P.12.
34. Kindratenko V., Enos J., Shi G., Showerman M., Arnold G., Stone J., Phillips J., Hwu W. 2009. GPU Clusters for HighPerformance Computing, in Proceedings of IEEE International Conference on Cluster Computing, Workshop on Parallel Programming on Accelerator Clusters.
35. Showerman M., Enos J., Pant A., Kindratenko V., Steffen C., Pennington R., Hwu W. 2009. QP: A Heterogeneous MultiAccelerator Cluster, In Proceedings of 10th LCI International Conference on High-Performance Clustered Computing – LCI'09.
36. Takizawa H., Kobayashi H. 2006. Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. J. Supercomput., 36, 219--234.

37. Noaje G., Jaillet C., Krajecki M. 2011. Source-to-Source Code Translator: OpenMP C to CUDA, in Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications, 512-519.
38. Asaaf R. 2014. Multi-Threaded Automatic Integration Using OpenMP and CUDA. Master Thesis. Western Michigan University, USA.
39. Jasiūnienė E., Samaitis V., Mažeika L., Sanderson R. 2015. 3D ultrasonic non-destructive evaluation of spot welds using an enhanced total focusing method. Journal of materials engineering and performance. New York: Springer, 24(2): 825-831.

Priedai

1. Priedas. Realizuotas SAFT skaičiavimo kodas, pagrindinis ciklinis skaičiavimas

```
/*
 * mpi run config:
 * /srv/homes/paulius_mpi
 * ssh ubuntu@hmpi1.d.vkc.xyz
 * mpirun -H hmpi0,hmpi1 ./saftex
 *
 * Paulius Dapkus, KTU VKC. paulius.dapkus@ktu.lt or paulius.dapkus@gmail.com
 *
 * */

#include "declarable.h"
#include "saft.h"
#include "QDebug"
#include <time.h>
#include "readmatrixdata.h"
#include "H5Cpp.h"

#include <omp.h>
#include "mpi.h"
//#include <clBLAS.h>
#include <armadillo>

// * * * * * * * * * * * * * *
// SAFT declarable
const int qp_z_size = 2402;
const float M_Pitch = 0.5625;
const int N_Mtrx = 8;

const float c_ultr = 5.9655;
const float dt = 0.000833333;//1/1200;
const float t_min = -0.0840;
int StartP = 1;
const int NNN = N_Mtrx*N_Mtrx;

// For exec time calc
clock_t _begin, _end;
double _time_spent;

long elapsed = 0;
static timeval t0,t1;
static timeval f_test_0, f_test_1;
static timeval f_test_0_per_core, f_test_1_per_core;
int m_size = 64*5;
rowvec timeVals(m_size);
int counter = 0;
/* MESURE TIME OF LINE
gettimeofday(&f_test_0,NULL); // for tests mesuring time
```

```
// 0.000538684 [s] su (51x51x61)
gettimeofday(&f_test_1,NULL);
elapsed = ((f_test_1.tv_sec-f_test_0.tv_sec)*1000000 + f_test_1.tv_usec-f_test_0.tv_usec);
timeVals(counter)= elapsed / 1e6;
counter++;
if(counter == m_size)
   kBsc1 = 80;/**/
Saft::Saft()
{}
double Saft::calculateSaft(int myid, int mystart, int myend)
{
   if (myid == 0){
      cout << "Armadillo version: " << arma_version::as_string() << endl;
   }
   int kBsc1;
   _begin = clock(); // For exec time calc
   gettimeofday(&f_test_0,NULL); // for tests mesuring time
   gettimeofday(&t0,NULL);
   // Get saft data from HDF5 dataset
   ReadMatrixData *readMatrix = new ReadMatrixData();
   cube rawSaftData = readMatrix->getMatrix();
   // 2D meshGrid  @@@@@@@ - @@@@@@ C1
   int X_Matrix_size = 64;
   int Y_Matrix_size = 64;
   rowvec X_Matrix(X_Matrix_size); X_Matrix.fill(0);
   rowvec Y_Matrix(Y_Matrix_size); Y_Matrix.fill(0);
   int l=1;
   // Construct X vector
   for(int a=0; a<X_Matrix_size; a++){
      X_Matrix(a) = l * M_Pitch - M_Pitch/2;
      l++;
      if(l > N_Mtrx) l=1;
   }
   // Construct Y vector
   int iter=2; l=1;
   for(int a=0; a<Y_Matrix_size; a++){
      Y_Matrix(a) = l * M_Pitch - M_Pitch/2;

      if(iter > N_Mtrx){
         l++;
         iter=1;
      }
      iter++;
   }
   // Plot mesh ranges @@@@@@ - @@@@@@ C2
   // Maximum meshGrid value
   float max_x_range = 5;
   float max_y_range = 5;
   float max_z_range = 3.5;

   // Minimum meshGrid value
   float min_x_range = 0;
```

```
float min_y_range = 0;
float min_z_range = 0.5;

// meshGrid step
float x_divider = 0.1; //0.1 ----- 0.15 visuose nepraranda kokybes
float y_divider = 0.1; //0.1
float z_divider = 0.05;/**/ // 0.05

// For constructing meshGrid
int i, j, k;
int i_t = (max_x_range-min_x_range)/x_divider;
int j_t = (max_y_range-min_y_range)/y_divider;
int k_t = (max_z_range-min_z_range)/z_divider;

static int i_t_full = i_t+1;
static int j_t_full = j_t+1;
static int k_t_full = k_t+1;

//qDebug()<<"Using matrix sizes: "<<i_t_full<<"x"<<j_t_full<<"x"<<k_t_full;

cube x_meshGrid( i_t_full, j_t_full, k_t_full);
cube y_meshGrid( i_t_full, j_t_full, k_t_full);
cube z_meshGrid( i_t_full, j_t_full, k_t_full);

// Building X meshGrid
float _min_x_range = min_x_range;
for (k = 0; k <= k_t; k++){
   for (i = 0; i <= i_t; i++){
      for (j = 0; j <= j_t; j++){
          x_meshGrid(i, j, k) = _min_x_range;
          _min_x_range = _min_x_range + x_divider;
       }
       _min_x_range=min_x_range;
   }
}/**/

// Building Y meshGrid
float _min_y_range = min_y_range;
for (j = 0; j <= j_t; j++){
   for (k = 0; k <= k_t; k++){
       for (i = 0; i <= i_t; i++){
          y_meshGrid(i, j, k) = _min_y_range;
          _min_y_range = _min_y_range + y_divider;
       }
       _min_y_range=min_y_range;
   }
}/**/
// Building Z meshGrid
float _min_z_range = min_z_range;
for (j = 0; j <= j_t; j++){
   for (i = 0; i <= i_t; i++){
       for (k = 0; k <= k_t; k++){
```

```
                z_meshGrid(i, j, k) = _min_z_range;
                _min_z_range = _min_z_range + z_divider;
              }
              _min_z_range=min_z_range;
            }
          }/**/

          // Plot mesh sizes @@@@@ - @@@@@ C3
          /*const int Nx=x_meshGrid.n_cols;
          const int Ny=y_meshGrid.n_rows;
          const int Nz=z_meshGrid.n_slices;/**/
          cube YIsum(i_t_full, j_t_full, k_t_full);
          cube YIWork(i_t_full, j_t_full, k_t_full);
          YIWork.fill(0);
          YIsum.fill(0);
          //int check = 0, check2 = 0;

        #ifdef DEBUG
          gettimeofday(&f_test_1,NULL);
          elapsed = (f_test_1.tv_sec-f_test_0.tv_sec)*1000000 + f_test_1.tv_usec-f_test_0.tv_usec;
          cout << " * Prepare matrixes exec time: " << elapsed / 1e6 << endl;

          qDebug()<<"Max system threads: "<<omp_get_max_threads();
        #endif

          gettimeofday(&f_test_0_per_core,NULL); // test time per core

          // Main SAFT cycle
        #ifdef USE_OPEN_MP
          //omp_set_dynamic(0);
          omp_set_num_threads(WORKING_MP_THREADS);
          #pragma omp parallel for private(i, j, k) //reduction(+:check) //shared(YIsum, YI, ARc,
t_m, KDD, KDP, DTrRc, ATr, T_pos, R_pos)// reduction(+:klo)  //shared(klo, klo2)  < -----isimk
T_pos ir R_pos kad veiktu. Jei nori, kad veiktu klo++ reik surinkt visus globalius parametrus ir
susert i private
        #endif
          for(kBsc1=mystart; kBsc1<myend; kBsc1++){

            cube ATr  (i_t_full, j_t_full, k_t_full);
            cube DTrRc(i_t_full, j_t_full, k_t_full);
            cube KDP  (i_t_full, j_t_full, k_t_full);
            cube KDD  (i_t_full, j_t_full, k_t_full);

            cube t_m  (i_t_full, j_t_full, k_t_full);
            cube ARc  (i_t_full, j_t_full, k_t_full);
            cube YI   (i_t_full, j_t_full, k_t_full);

            rowvec T_pos(3);
            rowvec R_pos(3);

            qDebug()<<"Running element: "<<kBsc1;
```

```
            // T_pos @@@@@ - @@@@@ C4
            T_pos(0) = X_Matrix(kBsc1);
            T_pos(1) = Y_Matrix(kBsc1);
            T_pos(2) = 0;

            // @@@@@ - @@@@@ C5
            for (k = 0; k <= k_t; k++)
               for (i = 0; i <= i_t; i++)
                  for (j = 0; j <= j_t; j++)
                     // ATr=atan2( sqrt( (X-T_pos(1)).^2 + (Y-T_pos(2)).^2 ), (Z-T_pos(3)));
                     ATr(i, j, k) = atan2(sqrt(pow(x_meshGrid(i, j, k) - T_pos(0), 2) +
pow(y_meshGrid(i, j, k) - T_pos(1), 2)), z_meshGrid(i, j, k) - T_pos(2));

            //#pragma omp for
            for(int kBsc2=0; kBsc2<NNN; kBsc2++){
               // R_pos @@@@@ - @@@@@ C6
               R_pos(0) = X_Matrix(kBsc2);
               R_pos(1) = Y_Matrix(kBsc2);
               R_pos(2) = 0;

               // 6.71562e-06 [s] su (51x51x61)
               // @@@@@ - @@@@@ C6++
               vec y = vectorise(rawSaftData( span(0, qp_z_size-1), span(kBsc1,kBsc1),
span(kBsc2, kBsc2)));

               // 3.05625e-06 [s] su (51x51x61)
               // @@@@@ - @@@@@ C7
               rowvec tx(qp_z_size);
               for(int u=0; u<qp_z_size; u++ )
                  tx(u) = u * dt + t_min;

               // @@@@@ - @@@@@ C7++
               float T1 = tx(0); // First tx element
               float T2 = tx(qp_z_size-1); // Last tx element

               // 0.00164284 [s] su (51x51x61)
               DTrRc = sqrt(square(x_meshGrid- T_pos(0)) + square(y_meshGrid - T_pos(1)) +
square(z_meshGrid - T_pos(2))) +
                     sqrt(square(x_meshGrid- R_pos(0)) + square(y_meshGrid - R_pos(1)) +
square(z_meshGrid - R_pos(2)));

               // 0.000251563 [s] su (51x51x61)
               t_m = DTrRc/c_ultr;

               // 0.00715759 [s] su (51x51x61)
               // @@@@@ - @@@@@ C8
               for (k = 0; k <= k_t; k++)
                  for (i = 0; i <= i_t; i++)
                     for (j = 0; j <= j_t; j++)
                        // ARc=atan2( sqrt( (X-R_pos(1)).^2 + (Y-R_pos(2)).^2),(Z-R_pos(3)));
                        ARc(i, j, k) = atan2(sqrt(pow(x_meshGrid(i, j, k) - R_pos(0),2) +
pow(y_meshGrid(i, j, k) - R_pos(1),2)), z_meshGrid(i, j, k) - R_pos(2)); // cube size -> (51, 51, 61)
```

```
                // 0.00583497 [s] su (51x51x61)
                // @@@@@@ - @@@@@@ C9
                //KDP=exp( -10 * ATr.^2 ).*exp( -10 * ARc.^2);
                KDP= exp( -10 * square(ATr) ) % exp( -10 * square(ARc));

                // 0.000309422 [s] su (51x51x61)
                // @@@@@@ - @@@@@@ C10
                KDD=1/(DTrRc + 0.2);

                // 0.000538781 [s] su (51x51x61)
                // @@@@@@ - @@@@@@ C12
                // Perkeltas pries if, kadangi naudojamas ir if'e ir else'e
                rowvec t_e(k_t_full * i_t_full * j_t_full);  // Galima gal but optimizuot, kad nereiketu is 3D matricos perrusiuot i vektoriu
                {
                   int it=0;
                   for (k = 0; k < k_t_full; k++)
                      for (i = 0; i < i_t_full; i++)
                        for (j = 0; j < j_t_full; j++){
                            t_e(it) = t_m(i,j,k);
                            it++;
                        }
                }/**/
                //colvec t_e(k_t_full * i_t_full * j_t_full); // skaiciuoja ilgiau
                //t_e = vectorise(t_m); // skaiciuoja ilgiau


                // @@@@@@ - @@@@@@ C11
                //if (max(max(max(t_m)))>T2)||(min(min(min(t_m)))<T1),
        #ifdef USE_OPEN_MP
                #pragma omp flush (T2, T1)
        #endif
                if( t_m.max() > T2 || t_m.min() < T1){
                   // @@@@@@ - @@@@@@ C13
                   rowvec Tmax = ((t_e-T2)-abs((t_e-T2)))/2+T2; // in matlab code t_e ---> Tmin

                   // @@@@@@ - @@@@@@ C14
                   uvec K2 = find(Tmax == T2);

                   // @@@@@@ - @@@@@@ C15
                   rowvec YIe;
                   interp1(tx, y, t_e, YIe, "linear");//nearest");    // "linear");  // faster than "linear" is "*linear"

                   // @@@@@@ - @@@@@@ C16
                   YIe(K2).fill(0);

                   // 0.000541869 [s] su (51x51x61)
                   // @@@@@@ - @@@@@@ C17
                   int it=0;
                   for (k = 0; k < k_t_full; k++)
```

```
                    for (i = 0; i < i_t_full; i++)
                      for (j = 0; j < j_t_full; j++){
                          YI(i,j,k) = YIe(it);
                          it++;
                      }/**/
                  //YI = reshape(YIe, k_t_full * i_t_full * j_t_full);
               }
               else{
                  // @@@@@ - @@@@@ C18
                  rowvec YIea;
                  interp1(tx, y, t_e, YIea, "linear");//"nearest");//, "linear");  // faster than "linear" is
"*linear"

                  // 0.000534594 [s] su (51x51x61)
                  int it=0;
                  for (k = 0; k < k_t_full; k++)
                     for (i = 0; i < i_t_full; i++)
                        for (j = 0; j < j_t_full; j++){
                           YI(i,j,k) = YIea(it);
                           it++;
                        }
               }
               // 0.000538684 [s] su (51x51x61)
        #ifdef USE_OPEN_MP
               #pragma omp critical // NERA BUTINA
               {
                  YIsum=YIsum+YI % KDP % KDD; // % <--- daugyba
               }
        #else
               //YIsum=YIsum+YI % KDP % KDD; // % <--- daugyba
               YIWork = YIWork + YI % KDP % KDD;
        #endif
               /*if (StartP != 1)
                  YIsum=YIsum+YI % KDP % KDD;
               else{
                  YIsum=YI % KDP % KDD; // % <--- daugyba
                  StartP=0;
               }*/
            }
            MPI_Barrier(MPI_COMM_WORLD);
         }/**/

         /*cube YIsum(x_dim, y_dim, z_dim);
         cube YIWork(x_dim, y_dim, z_dim);
         YIWork.fill(0);
         YIsum.fill(0);*/

         /* MPI_Reduce(YIWork.memptr(),
                  YIsum.memptr(),
                  YIWork.size(),
                  MPI_DOUBLE,
                  MPI_SUM,
```

```
                8,
                MPI_COMM_WORLD);/**/

          /* if (myid == 8 || myid == 0) {
              YIsum.save("arma_final4.h5",hdf5_binary);
           }*/

           gettimeofday(&f_test_1_per_core,NULL);
           elapsed = (f_test_1_per_core.tv_sec-f_test_0_per_core.tv_sec)*1000000 +
f_test_1_per_core.tv_usec-f_test_0_per_core.tv_usec;

           qDebug()<<"Exec per CPU: "<<elapsed / 1e6 << ", with CPU id of: "<<myid<<", CPU
cores load: "<<myend-mystart;


        #ifdef DEBUG
           qDebug()<<"first check(2765): "<<check<<", second check(1331): "<<check2;    // first
iter:  2765 , second:  1331
        #endif

        #ifdef TEST_V_M
           //timeVals.print("----> ");
           //qDebug()<<timeVals(2);
           double x = sum(timeVals)/(m_size);
           qDebug()<<x;
        #endif

          // if (myid == 0){
              //gettimeofday(&t1, NULL);
              //elapsed = (t1.tv_sec - t0.tv_sec)*1000000 + t1.tv_usec - t0.tv_usec;
             /* _end = clock();
              _time_spent = (double)(_end - _begin) / CLOCKS_PER_SEC;
              cout << " * Cpu  exec time: " << _time_spent << endl;
              cout << " * Real exec time: " << elapsed / 1e6 << endl;
              cout.flush();/**/

              //YIsum.save("/tmp/arma_final2.h5",hdf5_binary);

              //qDebug()<<" * Saft done";
          // }


           delete readMatrix;

           return elapsed / 1e6;
        }
```

2. Priedas. Pagrindinis vykdymo kodas

```
        /*
         * Paulius Dapkus, KTU VKC.
         * paulius.dapkus@ktu.lt or paulius.dapkus@gmail.com
```

```
 *
 */
#include <QCoreApplication>

#include <time.h>
//#include <declarable.h>
#include <iostream>
#include <string>
#include <readmatrixdata.h>
#include "fileops.h"
#include <QDebug>

#include "saft.h"

#include <QThread>

#include <mpi.h>

// 51*51*61  * 8 /1024/1024 ----- MB from matrix
// /home/ubuntu/mpi/saftex/buildas
// mpirun -hostfile ~/mpihosts_s ./saftex testas 15

/* --- mpihosts_s
hmpi0 slots=2
hmpi1 slots=8
hmpi2 slots=6
 */

// For exec time calc
clock_t begin, end;
double time_spent;
long el = 0;
static timeval t0,t1;

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    int myid,numproc;
    int nr;
    int mystart,myend;

    QString logName = "";
    int runTime = 0;
    logName = argv[1]; // test name in log file
    runTime = atoi(argv[2]); // run time amount

    if(logName == "")
        logName = "No Log Name";

    if(runTime == 0)
        runTime = 1;
```

```
        begin = clock(); // For exec time calc
        gettimeofday(&t0,NULL);

        //ReadMatrixData *readM = new ReadMatrixData();
        //cube w = readM->getMatrix();

        int required = MPI_THREAD_SERIALIZED;// MPI_THREAD_SINGLE
//MPI_THREAD_FUNNELED; // MPI_THREAD_MULTIPLE; //MPI_THREAD_SERIALIZED;
        int provided; // Provided level of MPI threading support

        MPI_Init_thread(NULL, NULL, required, &provided);

        // get myid and # of processors
        MPI_Comm_size(MPI_COMM_WORLD,&numproc);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);

        char hostname[256];
        gethostname(hostname, sizeof(hostname));

        qDebug()<<myid<<"******************";

        int NNN = 64;

        // get # of terms
        nr = NNN; //atoi(argv[1]);
        MPI_Barrier(MPI_COMM_WORLD);
        if (myid == 8 || myid == 1){
           printf("Array size: %d\n",nr);
           //qDebug()<<"My id: "<<myid;
           qDebug()<<"Num of process: "<<numproc;
        }

        // Divide loop for procesors
        mystart = (nr / numproc) * myid;
        if (nr % numproc > myid){
           mystart += myid;
           myend = mystart + (nr / numproc) + 1;
        }else{
           mystart += nr % numproc;
           myend = mystart + ( nr / numproc );
        }
        MPI_Barrier(MPI_COMM_WORLD);
        printf("CPU%d-%s %d ~ %d \n",myid,hostname,mystart,myend);

        FileOps *writeTimes = new FileOps;
        if (myid == 8 || myid == 1){
           writeTimes->addText("\n- - - - - ");
           writeTimes->addText("- Name: " + logName);
           writeTimes->addText("- - - - - ");
        }
```

```
    double thisTestTime= 0;
    double testTimeSum= 0;

    for(int a=0; a<runTime; a++){
        Saft *saft = new Saft();
        thisTestTime = saft->calculateSaft(myid, mystart, myend);
        delete saft;

        MPI_Barrier(MPI_COMM_WORLD);
        testTimeSum += thisTestTime;
        if (myid == 8 || myid == 1){
            qDebug()<<"This time: "<<thisTestTime;
            qDebug()<<"Sum: "<<testTimeSum;
            qDebug()<<"Run Time: "<<a+1<<" of"<<runTime;
            qDebug()<<"Curr time: "<<testTimeSum/(a+1);

            writeTimes->writeCalcTime(thisTestTime);
        }
        QThread::sleep(5);
        if(myid == 8 || myid == 1){
            qDebug()<<"-----------------------------------------------";
        }
    }
   if (myid == 8 || myid == 1){
        qDebug()<<"Final runtime: "<<testTimeSum/runTime;
        writeTimes->addText("- - AVG - - ");
        writeTimes->writeCalcTime(testTimeSum/runTime);
        writeTimes->addEndLine();
    }

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();

    if (myid == 8 || myid == 1){
        gettimeofday(&t1,NULL);
        el = (t1.tv_sec-t0.tv_sec)*1000000 + t1.tv_usec-t0.tv_usec;
        end = clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        cout << " * Main program cpu exec time: " << time_spent << endl;
        cout << " * Real exec time: " << el / 1e6 << endl;
        cout << " * Main done" << endl;
    }

    return 0;
    //return a.exec();
}
```

3. Priedas. Konfigūracijos ir eksperimentų vykdymo laiko saugojimas saugojimas

```
/*
 * Paulius Dapkus, KTU VKC.
 * paulius.dapkus@ktu.lt or paulius.dapkus@gmail.com
 *
 */
#include "fileops.h"
#include <QFile>
#include <QTextStream>
#include <declarable.h>

FileOps::FileOps()
{

}

int FileOps::writeCalcTime(double time)
{
   QFile file(execTimeFileName);
   if (file.open(QIODevice::Append)) {
     QTextStream stream(&file);
     stream << time << endl;
   }
   file.close();
}

int FileOps::addEndLine()
{
   addText(QString("- - END - - "));
}

int FileOps::addText(QString text)
{
   QFile file(execTimeFileName);
   if (file.open(QIODevice::Append)) {
     QTextStream stream(&file);
     stream << text << endl;
   }
   file.close();
}
```