Intentionally left blank!

[Turn page over, please]

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Aleksandr JAKUŠEV

DIFERENCIALINIŲ LYGČIŲ IR JŲ SISTEMŲ SKAITINIO SPRENDIMO ALGORITMŲ LYGIAGRETINIMO TECHNOLOGIJOS KŪRIMAS, ANALIZĖ IR TAIKYMAI

Daktaro disertacija

Technologijos mokslai, informatikos inžinerija (07T)

VGTU LEIDYKLA TECHNIKA

Vilnius 2007

Disertacija rengta 2003–2007 metais Vilniaus Gedimino technikos universitete.

Darbo vadovas

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

http://leidykla.vgtu.lt
VGTU leidyklos TECHNIKA 1348 mokslo literatūros knyga

ISBN 978-9955-28-213-6

© Jakušev, A., 2007

Žymėjimai

Simboliai

p	Procesų skaičius
p_i	Procesų skaičius i-oje dimensijoje
T_p	Lygiagrečiojo algoritmo vykdymo laikas, kai naudojame p procesų
S_p	Lygiagrečiojo algoritmo *spartinimo* koeficientas
E_p	Algoritmo lygiagretinimo *efektyvumo* koeficientas
DC	Dimensijų skaičius (nuo angl. *Dimension Count*)

Santrumpos

1D	angl. *1-Dimensional* (vienmatis)
2D	angl. *2-Dimensional* (dvimatis)
3D	angl. *3-Dimensional* (trimatis)
BiCGStab	angl. *Bi-Conjugate Gradient-Stable* (dvikryptis jungtinių gradientų stabilus)
BLAS	angl. *Basic Linear Algebra Subprograms* (pagrindiniai tiesinės algebros paprogramiai)
CG	angl. *Conjugate Gradients* (jungtiniai gradientai)
CPU	angl. *Central Processing Unit* (centrinis procesorius)
CSR	angl. *Compressed Sparse Row* (suspausta išretinta eilutė)
EC++	angl. *Embedded C++* (C++ modifikacija)
GHz	Gigahercas, 10^9 Hz
HPF	angl. *High Performance Fortran* (didelio našumo Fortranas)
IŠ	Išraiškų šablonai (angl. *Expression Templates*)
IFD	angl. *Internal Finite Difference* (vidiniai baigtiniai skirtumai)
ILU	angl. *Incomplete LU* (nepilna LU faktorizacija)
IMPES	angl. *IMplicit Pressure, Explicit Saturation* (neišreikštinis slėgis, išreikštinis prisisotinimas)
JG	Jungtiniai gradientai (angl. *Conjugate Gradients*)
KT	Kompiuterinė tomografija (angl. *Computer Tomography*)
LU	angl. *Lower Upper* (LU faktorizacija)
MB	Megabaitas, 2^{20} baitų
MFM	angl. *Multiphase Flow Model* (daugiafazio tekėjimo modelis)
MIMD	angl. *Multiple Instruction stream, Multiple Data stream* (daug komandų srautų, daug duomenų srautų)

MINRES	angl. *MINimal RESiduals* (mažiausių netikčių)
MISD	angl. *Multiple Instruction stream, Single Data stream* (daug komandų srautų, vienas duomenų srautas)
MM	Modifikuojanti matrica (angl. *preconditioner*)
MPI	angl. *Message Passing Interface* (duomenų persiuntimo sąsaja)
MR	angl. *Minimal Residuals* (mažiausių netikčių)
MSD	angl. *Modified Steepest Descent* (modifikuotas didžiausio nuolydžio)
NUMA	angl. *Non-Uniform Memory Access* (netolygiai pasiekiama atmintis)
OO	Objektiškai orientuotas (angl. *Object-Oriented*)
OOP	Objektiškai orientuotas programavimas (angl. *Object-Oriented Programming*)
PC	angl. *Personal Computer* (asmeninis kompiuteris)
PCG	angl. *Preconditioned Conjugate Gradient* (jungtiniai gradientai su modifikuojančia matrica)
PVM	angl. *Parallel Virtual Machine* (lygiagrečioji virtualioji mašina)
RAII	angl. *Resource Acquisition Is Initialization* (resursų gavimas yra iniciacija)
RAM	angl. *Random Access Memory* (laisvosios kreipties atmintis)
SD	angl. *Steepest Descent* (didžiausio nuolydžio)
SIMD	angl. *Single Instruction stream, Multiple Data stream* (vienas komandų srautas, daug duomenų srautų)
SISD	angl. *Single Instruction stream, Single Data stream* (vienas komandų srautas, vienas duomenų srautas)
SMP	angl. *Symmetric Multi Processor* (simetrinis daugiaprocesorinis)
SRSD	angl. *Sub-Relaxation Steepest Descent* (didžiausio nuolydžio metodas su nepilna relaksacija)
STL	angl. *Standard Template Library* (standartinė šablonų biblioteka)
TSGD	angl. *Two Step Gradient Descent* (dvipakopis gradientinio nuolydžio)
UMA	angl. *Uniform Memory Access* (tolygiai pasiekiama atmintis)
UPC	angl. *Unified Parallel C* (suvienytas lygiagretusis C)

Turinys

Žymėjimai **iii**

Įvadas **1**

1. Tiesinės algebros algoritmų lygiagretinimas **9**
1.1. Įvadas 9
1.2. Diferencialinių lygčių ir jų sistemų sprendimo įrankių lygiagretinimas 10
1.2.1. Egzistuojantys įrankiai ir jų lygiagretinimo metodai 11
1.2.2. Egzistuojančių įrankių lygiagretinimo metodų analizė 15
1.3. Lygiagretinimo technologijų apžvalga 17
1.3.1. Algoritmų lygiagretinimas 17
1.3.2. Lygiagretinimo standartai ir specifinės programavimo kalbos 20
1.3.3. Egzistuojančių sprendimų, palengvinančių algoritmų lygiagretinimo procesą, apžvalga 25
1.3.4. Lygiagrečiojo programavimo standartų ir bibliotekų analizė . 27
1.4. Pirmojo skyriaus rezultatai ir išvados 29

2. Skaičiavimo matematikos algoritmų lygiagretinimo technologija **31**
2.1. Įvadas 31
2.2. Naujos lygiagretinimo technologijos principai 32
2.2.1. Technologijos pagrindinės idėjos 32
2.2.2. Technologijos detalizavimas 33

2.3. Reikalavimai, keliami technologijos realizacijai 42
2.4. Technologijos realizacija 44
2.4.1. Vartotojo sąsajos lygmuo 45
2.4.2. Nuoseklusis bibliotekos panaudojimo pavyzdys 61
2.4.3. Vidinės logikos lygmuo 64
2.4.4. Lygiagrečioji bibliotekos versija 70
2.4.5. Lygiagrečiųjų duomenų algoritmų lygiagretinimas taikant biblioteką 74
2.5. Realizacijos efektyvumo analizė 75
2.5.1. Masyvo elemento pasiekimo greičio įvertinimas 75
2.5.2. Išraiškų šablonų veikimo greičio įvertinimas 77
2.6. Technologijos tolesnio vystymo galimybės 79
2.6.1. Nestruktūrinių tinklų palaikymas 79
2.6.2. Bibliotekos optimizavimas SMP tipo kompiuteriams 81
2.6.3. Statinis krūvio balansavimas 83
2.7. Antrojo skyriaus rezultatai ir išvados 87

3. Lygiagretinimo technologijos taikymai **91**
3.1. Įvadas 91
3.2. Lygiagretusis įrankis, skirtas tekėjimams poringose terpėse modeliuoti 92
3.2.1. Įvadas 92
3.2.2. Nuoseklioji įrankio versija 93
3.2.3. Lygiagrečioji, patobulinta įrankio versija 104
3.3. Vaizdų filtravimo uždaviniai 108
3.3.1. Įvadas 108
3.3.2. Netiesiniai difuziniai filtrai 110
3.3.3. Baigtinių skirtumų aproksimacija 110
3.3.4. Algoritmų lygiagretinimo efektyvumo įvertinimas 112
3.3.5. Kompiuterinės tomografijos vaizdų apdorojimas 119
3.4. Variacinių iteracinių tiesinių lygčių sistemų sprendimo metodų lygiagretinimas 120
3.4.1. Vienažingsniai variaciniai iteraciniai metodai 121
3.4.2. Įvairios standartinių metodų modifikacijos 123
3.4.3. Dvipakopio gradientinio nuolydžio metodo lygiagretinimo analizė 127
3.4.4. Skaitiniai lygiagretinimo eksperimentai 131
3.5. Trečiojo skyriaus rezultatai ir išvados 133

Darbo rezultatų apibendrinimas ir išvados **135**

Literatūros sąrašas **139**

Priedai
1 priedas. Lygiagretinimo technologijos realizacija CD
2 priedas. Masyvų ir vektorių regresiniai tikrinimai CD
3 priedas. Matricų regresiniai tikrinimai CD
4 priedas. Bibliotekos greičio tikrinimo kodas CD
5 priedas. Difuziniai filtrai CD
6 priedas. Variaciniai iteraciniai tiesinių lygčių sistemų sprendimo algoritmai CD

Įvadas

Problemos formulavimas

Lygiagrečiųjų algoritmų kūrimo ir realizacijos svarba yra gerai žinoma. Tačiau tai yra sudėtingas procesas, reikalaujantis iš programuotojo daug specifinių žinių ir patirties. Todėl nuolat kuriamos naujos technologijos, kurių tikslas yra palengvinti šį procesą. Dažnai kuo technologija yra universalesnė, kuo daugiau faktorių ji įskaito, tuo sudėtingiau ją taikyti. Taip pat gali nukentėti technologijos efektyvumas, nes pirmenybė teikiama teisingam veikimui visais atvejais, kurių yra labai daug. Ir atvirkščiai, siaurinant sprendžiamų uždavinių ratą, įvedami nauji apribojimai ir reikalavimai, kuriuos panaudojus galima tikėtis gauti efektyvesnį nagrinėjamų uždavinių sprendimą. Todėl vienas iš būdų lengvinti lygiagretinimą yra kurti *specifinių* uždavinių klasių lygiagretinimo technologijas.

Diferencialinių lygčių ir jų sistemų sprendimo algoritmai sudaro tokią klasę. Atliekant tikslius skaičiavimus, tokio tipo uždaviniai reikalauja daug atminties ir procesorių resursų, todėl jų lygiagretinimas yra ypač aktualus. Nors bendri diferencialinių lygčių sprendimo etapai lieka tie patys (diskretizacija, linearizacija, tiesinių lygčių sistemų sprendimas ir kiti), egzistuoja daug jų variantų, todėl nuolat atsiranda nauji diferencialinių lygčių ir jų sistemų sprendimo įrankiai. Deja, tokio tipo įrankiams lygiagretinti naudojamos bendresnės lygiagretinimo technologijos, todėl įrankių lygiagretinimas arba realizuojamas žemame lygyje ir negali būti pritaikytas kitur, arba nepasižyi visu norimu funkcionalumu.

Šiame darbe siekiama sukurti lygiagretinimo technologiją, skirta būtent diferencialinių lygčių sprendimo algoritmų lygiagretinimui (nors, aišku, tuo jos taikymai gali neapsiriboti). Analizuojant efektyvias, tačiau žemame lygyje atliktas populiarių diferencialinių lygčių sprendimo paketų lygiagretinimo realizacijas, bus siekiama surasti šios klasės uždavinių lygiagretinimo reikalavimus ir sukurti technologija, kuri juos atitiktų. Sukurus technologiją, bus siekiama ją išbandyti lygiagretinant šios klasės uždavinius.

Tyrimo objektas

Disertacijos tyrimo objektas – lygiagretieji algoritmai, lygiagretinimo priemonės, lygiagretinimo automatizavimas, efektyvus skaitinis programavimas C++ kalba, sudėtingų technologinių procesų modeliavimas naudojant lygiagrečiuosius algoritmus.

Darbo aktualumas

Programų ir algoritmų lygiagretinimo nauda yra akivaizdi – naudojant lygiagrečiuosius kompiuterius galima išspręsti didesnius uždavinius ir tai padaryti greičiau. Tai labai svarbu šiais laikais, kai formuluojamų uždavinių apimtys pralenkia moderniausių kompiuterių pajėgumus. Lygiagrečiai spręsti uždavinius yra ir ekonomiškai naudinga, nes sujungti N kompiuterių yra pigiau, negu pagaminti N kartų galingesnį kompiuterį. Kartais tam net nereikia įsigyti naujų kompiuterių, nes užtenka panaudoti esamus resursus.

Negana to, kompiuterių architektūros vystymosi tendencijos rodo, kad ateityje ir paprasti (nespecializuoti) asmeniniai kompiuteriai gali tapti lygiagretūs [77]. Jau dabar kelių branduolių (angl. *multicore*) kompiuteriai, kurie turi iki 4 procesorių, yra įprastas reiškinys. Kai kurie specializuoti kompiuteriai turi iki 32 CPU, naudojančių tą pačią atmintį. Šiuo metu vystoma nauja *manycore* architektūra, kuri leistų sujungti šimtus ir tūkstančius CPU vienoje mikroschemoje.

Tačiau programų, pritaikytų lygiagretiesiems skaičiavimams, kūrimas yra nelengvas procesas, reikalaujantis iš vartotojo daug specifinių žinių ir įgūdžių. Tai stabdo lygiagretinimo plėtrą. Todėl priemonių, palengvinančių lygiagrečiųjų programų rašymą eiliniam vartotojui, kūrimas yra labai svarbi užduotis.

C++ kalba yra C kalbos vystymo rezultatas. Ji turi daug galimybių, leidžiančių daugelį algoritmų realizuoti aiškiau ir paprasčiau, negu su C, gauti lankstesnį kodą. Deja, plačios C++ kalbos galimybės reikalauja iš programuotojo daugiau pastangų, norint įvertinti jų pridėtinius kaštus ir išmokti visas galimybes tinka-

mai taikyti. Viena iš netinkamo taikymo pasekmių – lėčiau veikiantis ir (arba) daugiau atminties naudojantis kodas. Tai stabdo šios kalbos plitimą sprendžiant skaičiavimo matematikos ir matematinio modeliavimo uždavinius. Panašiai kaip ir lygiagrečiųjų programų kūrimo atveju didelę pagalbą taikant C++ skaičiavimo uždavinių sprendimui suteikia specialios bibliotekos, leidžiančios vartotojui naudotis visais C++ kalbos privalumais ir neprarasti efektyvumo.

Pastaruoju metu daugiafaziam tekėjimui poringose terpėse skiriama labai daug dėmesio, nes egzistuoja daugybė uždavinių, kur sutinkami šitie reiškiniai. Galima nurodyti tokių pavyzdžių, pvz., poringų medžiagų (pvz., medienos) džiovinimas arba prisotinimas, žemės užteršimo modeliavimas ir jo valymas, naftos gavyba, radioaktyviųjų atliekų saugojimas, popieriaus gamyba, biologija (įvairių audinių veikla), filtrų konstravimas ir kiti uždaviniai.

Vaizdų glodinimas gali būti daug kur pritaikomas, pavyzdžiui, triukšmų šalinimas arba kraštų detekcija. Tai yra svarbu tokiose srityse, kaip kompiuterinė tomografija arba kompiuterinis regėjimas (angl. *machine vision*).

Tiesinių lygčių sistemos sprendimas yra daugybės modeliavimo uždavinių, kur modeliavimas atliekamas, sprendžiant diferencialinę lygtį arba lygčių sistemas, svarbi sudedamoji dalis. Tiesinių lygčių sistemų sprendimas tokiuose uždaviniuose gali sudaryti iki 80 % viso skaičiavimo laiko. Todėl efektyvūs tiesinių lygčių sprendimo metodai ir jų lygiagrečiosios versijos yra labai svarbios daugelio uždavinių modeliavimui.

Darbo tikslai

1. Sukurti technologiją, kurią bus galima taikyti lygiagretinant diskrečiuosius modelius, gaunamus po parabolinių ir hiperbolinių lygčių bei jų sistemų aproksimavimo baigtinių skirtumų, baigtinių tūrių ar baigtinių elementų metodais.
2. Parengti įrankių ir metodinių nurodymų visumą, kuri leistų vartotojui *nesudėtingai* naudoti sukurtą technologiją lygiagretinant įvairius uždavinius ir išbandyti ją.
3. Naudojant sukurtą technologiją, realizuoti lygiagrečiuosius algoritmus, skirtus modeliuoti sudėtingus technologinius procesus – daugiafazį skysčių tekėjimą poringose terpėse, vaizdų glodinimą, taip pat tiesinių lygčių sistemų iteracinių sprendimo algoritmų lygiagrečiąsias versijas.

Spręsti uždaviniai

1. Atlikti egzistuojančių diferencialinių lygčių sprendimo įrankių lygiagretinimo analizę, siekiant nustatyti reikalavimus, keliamus tokio tipo uždavinių lygiagretinimui.
2. Atlikti egzistuojančių lygiagretinimo modelių ir jais paremtų lygiagretinimo standartų bei įrankių analizę.
3. Remiantis atlikta analize, pasiūlyti naujos lygiagretinimo technologijos principus. Nauja technologija, išlaikant egzistuojančių technologijų privalumus ir pašalinant jų trūkumus, turi leisti efektyviau lygiagretinti diferencialinių lygčių sprendimo algoritmus.
4. Realizuoti pasiūlytą technologiją skaičiavimo matematikos objektų bibliotekos pavidalu. Taikant šią biblioteką turi būti galima efektyviai realizuoti ir pusiau automatiškai lygiagretinti diferencialinių lygčių sprendimo algoritmus.
5. Remiantis šia technologija sukurti naują procesų poringose terpėse modeliavimo paketą, kuris galėtų panaudoti tiek nuosekliuosius, tiek ir lygiagrečiųjų kompiuterių resursus, realizuoti vaizdų filtravimo ir iteracinių tiesinių lygčių sistemų sprendimo algoritmų lygiagrečiasias versijas.

Tyrimo metodai

Tyrimo metodai apima skaitinius metodus, lygiagrečiųjų algoritmų efektyvumo ir sudėtingumo analizę, įvairių lygiagretinimo priemonių lyginamąją analizę, eksperimentinį tyrimų metodą.

Taikytos C++ programavimo kalba, objektinio programavimo ir šabloninio metaprogramavimo (angl. *template metaprogramming*) technologijos, MPI duomenų perdavimo standartą realizuojančios bibliotekos.

Mokslinis naujumas

Sukurta nauja algoritmų lygiagretinimo technologija, kuri gali būti taikoma diskrečiųjų modelių, gaunamų po diferencialinių lygčių bei jų sistemų aproksimavimo, pusiau automatiniam lygiagretinimui. Ši technologija pasižymi tokiomis savybėmis:

- Ji naudoja egzistuojančių lygiagrečiųjų duomenų ir globalios atminties lygiagretinimo modelių elementus.
- Palyginti su lygiagrečiųjų duomenų modeliu, ji leidžia spręsti *platesnę* uždavinių klasę.
- Palyginti su globalios atminties modeliu, naujos technologijos apribojimai leidžia realizuoti *efektyvesnį* duomenų apsikeitimą. Taip pat užtikrinama pusiau automatinio lygiagretinimo galimybė.

Technologijos *realizacija* leidžia C++ kalba kurti tiek efektyvius, tiek aiškius ir pusiau automatiškai lygiagretinamus algoritmus.

Taikant sukurtą technologiją realizuoti tokių sudėtingų uždavinių lygiagretieji algoritmai:

- Įrankis, skirtas skysčių tekėjimams poringose terpėse modeliuoti. Taip pat, be lygiagretinimo galimybės, nauja įrankio versija pasižymi kitomis savybėmis, tokiomis kaip naujų matematinių modelių ir lygčių sprendimo metodų palaikymas, kas leidžia išplėsti modeliuojamų uždavinių klasę.
- Medicininių tomogramų vaizdų glodinimas taikant netiesinius difuzinius filtrus.
- Iteraciniai tiesinių lygčių sistemų sprendimo metodai.

Gautų rezultatų reikšmė

Pasiūlyta technologija leidžia kurti efektyvesnes skaičiavimo matematikos algoritmų lygiagretinimo priemones, tiek vartotojo sąsajos, tiek lygiagretinimo efektyvumo prasme, lyginant su šiuo metu paplitusiomis lygiagrečiųjų duomenų bei globalios atminties lygiagretinimo modeliais.

Pasiūlyta technologija realizuota skaičiavimo matematikos objektų bibliotekos pavidalu. Ši biblioteka leidžia vartotojui patogiai ir efektyviai realizuoti lygiagrečiųjų duomenų algoritmus naudojant C++ kalbą. Tokiu būdu realizuotas algoritmas gali būti *pusiau automatiškai* išlygiagretintas. Realizacijoje naudojami tik plačiai paplitę standartai (C++, MPI), todėl naudojant biblioteką sukurtos programos yra lengvai pernešamos į kitas platformas.

Naudojant biblioteką sukurtos aktualių uždavinių algoritmų lygiagrečios realizacijos. Išlygiagretintas skysčių tekėjimo poringose terpėse modeliavimo įrankis ir praplėstas jo funkcionalumas. Taip pat sukurtos lygiagrečiosios vaizdų filtravimo (taikomas nustatant insulto sritis žmonių smegenyse) ir iteracinių tiesinių lygčių sistemų sprendimo algoritmų versijos. Dabar šitie uždaviniai gali būti sprendžiami didesnės apimties ir greičiau.

Ryšys su mokslinėmis programomis

Sukurta nauja skaitinių diferencialinių lygčių sprendimo algoritmų lygiagretinimo technologija ir nagrinėjami uždaviniai, susiję su tarptautiniais EUREKA projektais:

E!2623 OPTPAPER. Šio projekto tikslas – energijos vartojimo optimizavimas popieriaus pramonėje.

E!2981 CTBSTROKE. Šio projekto tikslas – sukurti algoritmus ischeminio insulto sritims nustatyti galvos smegenų kompiuterinės tomografijos vaizduose.

Ginamieji disertacijos teiginiai

- Nauja diferencialinių lygčių sprendimo įrankių pusiau automatinio lygiagretinimo technologija, jungianti lygiagrečiųjų duomenų ir globalios atminties lygiagrečiojo programavimo modelių elementus.
- Technologijos realizacija matematinių objektų bibliotekos pavidalu, naudojant C++ ir MPI.
- Taikomųjų uždavinių (daugiafazis skysčių tekėjimas poringose terpėse, vaizdų glodinimas netiesiniais difuziniais filtrais, variaciniai iteraciniai tiesinių lygčių sprendimo metodai) algoritmų lygiagrečiosios versijos, gautos taikant pasiūlytą lygiagretinimo technologiją.

Darbo aprobavimas

Su disertacijos tema susiję rezultatai paskelbti 12 straipsniuose, iš kurių vienas žurnale, įtrauktame į *ISI Master Journal List* duomenų bazę, dar 6 straipsniai kituose periodiniuose recenzuojamuose leidiniuose. Vienas straipsnis yra leidinyje, įtrauktame į *ISI Proceedings* duomenų bazę.

Buvo perskaityta 11 pranešimų konferencijose:

1. R. Čiegis, A. Jakušev. Parallel tool for solution of multiphase flow problems. Sixth international Conference on Parallel Processing and Applied Mathematics, Poznań, Poland, 2005 m. rugsėjo 11–14 d.
2. A. Jakusev, V. Starikovicius. Application of Parallel Arrays for Parallelisation of Data Parallel Algorithms. International Networking for Young

Scientists, Optimal Process Design, Vilnius, Lithuania, 2006 m. vasario 15–16 d.

3. A. Jakušev, V. Starikovičius. Application of ParSol library for parallelization of flow in porous media problem solver. 12th international conference Mathematical Modelling and Analysis, Trakai, Lithuania, 2007 m. gegužės 30 – birželio 2 d.
4. R. Čiegis, A. Jakušev. Improving efficiency of parallel computations using parallel arrays. 11th international conference Mathematical Modelling and Analysis, Jurmala, Latvia, 2006 m. birželio 1–4 d.
5. A. Jakušev. Application of parallel arrays for semiautomatic parallelization of flow in porous media problem solver. 10th international conference Mathematical Modelling and Analysis, Trakai, Lithuania, 2005 m. birželio 1–5 d.
6. M. Baravykaitė, R. Čiegis, A. Jakušev, G. Šilko. Parallelization tools and industrial applications. 9th international conference Mathematical Modelling and Analysis, Jurmala, Latvia, 2004 m. gegužės 27–29 d.
7. A. Jakušev, R. Čiegis. Poringų terpių uždavinių optimizacija ir lygiagretinimas. Lietuvos matematikų draugijos XLVII konferencija, Kaunas, Lietuva, 2006 m. birželio 20–21 d.
8. R. Čiegis, A. Jakušev. Lygiagretieji algoritmai vaizdų filtravime. Lietuvos matematikų draugijos XLVI konferencija, Vilnius, Lietuva, 2005 m. birželio 15–16 d.
9. A. Jakušev. Daugiafazio tekėjimo uždavinių sprendimo įrankis ir jo taikymas daugiamačiams uždaviniams. Lietuvos matematikų draugijos XLV konferencija, Kaunas, Lietuva, 2004 m. birželio 17–18 d.
10. A. Jakušev. Išraiškų šablonų naudojimas C++ masyvų efektyvumo didinimui. 9-oji Lietuvos jaunųjų mokslininkų konferencija „Mokslas – Lietuvos ateitis", Matematinio modeliavimo sekcija, Vilnius, Lietuva, 2006 m. balandžio 20 d.
11. A. Jakušev. Daugiafazio tekėjimo uždavinių sprendimo įrankis ir jo testavimas. 7-oji Lietuvos jaunųjų mokslininkų konferencija „Lietuva be mokslo – Lietuva be ateities", Vilnius, Lietuva, 2004 m. balandžio 7 d.

Disertacijos struktūra ir apimtis

Disertaciją sudaro įvadas, 3 skyriai, rezultatų apibendrinimas ir išvados, literatūros sąrašas ir priedai elektroninėje laikmenoje. Darbo apimtis yra 147 puslapiai, kuriuose pateikta: 20 paveikslų ir 10 lentelių.

1 skyriuje pateikiama užduoties problematika. Supažindinama su uždaviniais, kurių sprendimui buvo kuriama technologija, analizuojama, kaip tokie uždaviniai buvo sprendžiami panašaus funkcionalumo bibliotekose, formuluojami reikalavimai naujos technologijos kūrimui. Toliau aptariami egzistuojantys lygiagretinimo įrankiai, jų stipriosios ir silpnosios pusės.

2 skyriuje detaliai aprašoma sukurta nauja lygiagretinimo technologija ir jos realizacija – matematinių objektų biblioteka. Aprašomos naujos technologijos pagrindinės idėjos ir detalės. Aprašoma technologijos realizacija, jos struktūra, vartotojo sąsaja, panaudojimo principai. Toliau pateikiami realizacijos greitaveikos tyrimai ir palyginimas su kitais sprendimais. Galiausiai aptariamos technologijos vystymo galimybės ir pateikiamos išvados apie technologijos savybes.

3 skyrius skirtas naujos technologijos taikymams. Vienas iš jų yra tekėjimo procesų poringose terpėse modeliavimo įrankis. Aprašoma pradinė įrankio versija, kurios lygiagretinimui buvo skirta technologija, ir galiausiai galutinė įrankio versija. Kiti technologijos taikymai lygiagretinant taikomuosius uždavinius – vaizdų filtravimas ir variaciniai iteraciniai tiesinių lygčių sprendimo metodai. Šie uždaviniai naudojami tiriant bibliotekos lygiagretinimo realizacijos efektyvumą.

1

Tiesinės algebros algoritmų lygiagretinimas

1.1. Įvadas

Pastaruoju metu lygiagrečiųjų algoritmų kūrimas tampa vis aktualesnis. Anksčiau lygiagrečiųjų algoritmų taikymas buvo ribotas, nes pagreitinti skaičiavimus tokiu būdu buvo galima tik daugiaprocesoriniuose superkompiuteriuose, kurie tuo metu buvo retenybė. Dauguma kompiuterių buvo nuosekliosios *Neumano* tipo mašinos. Nors dauguma operacinių sistemų juose palaikydavo kelių procesų vykdymą vienu metu (angl. *concurrency*), tai buvo pasiekiama procesoriaus laiko kvantavimu tarp procesų. Tokiu atveju algoritmo lygiagretinimas galėjo tik jį sulėtinti. Išimtis buvo daugiagijinis (angl. *multithreaded*) programavimas, kai į skirtingas gijas buvo atskiriamos arba daug skaičiavimų reikalaujančios, arba ilgai laukiančios (pavyzdžiui, sąveika su vartotoju arba išoriniais įrenginiais) algoritmo dalys.

Šiuo metu situacija keičiasi iš esmės. Vis didėjant skaičiavimų apimtims reikia vis galingesnių procesorių, tačiau procesorių pajėgumų didėjimo spartą jau pradeda stabdyti tokie fizikiniai veiksniai, kaip, pavyzdžiui, atomo dydis. Todėl vis sparčiau populiarėja daugiaprocesoriniai kompiuteriai. Maža to, santykinis asmeninių kompiuterių pigumas ir jų paplitimas lemia kompiuterinių klasterių gausą. Tokioje situacijoje algoritmų lygiagretinimas tampa vienu iš svarbiausių aspektų didinant algoritmo efektyvumą.

Tiesinės algebros algoritmai labai svarbūs taikomuosiuose skaičiavimuose, nes įvairių sudėtingų uždavinių sprendimas skaitiniais metodais dažnai susiveda į

tiesinių lygčių sistemų sprendimą ir kitas tiesinės algebros operacijas. Vis didėjant sprendžiamų uždavinių apimtims reikia ieškoti būdų, kaip išspręsti vis didesnius tiesinės algebros uždavinius, ir algoritmų lygiagretinimas yra vienas iš sprendimo variantų.

Šiame skyriuje yra pateikti darbe pristatomos skaičiavimo matematikos algoritmų lygiagretinimo technologijos principai, reikšmė ir vieta tarp kitų pasaulyje egzistuojančių šios srities sprendimų. Skyrius prasideda nuo užduočių, sprendžiant kurias atsirado ir buvo realizuota ši technologija, aptarimo. Toliau eina egzistuojančių lygiagretinimo sprendimų apžvalga, jų tinkamumo analizė mūsų uždaviniams.

1.2. Diferencialinių lygčių ir jų sistemų sprendimo įrankių lygiagretinimas

Diferencialinių lygčių ir jų sistemų sprendimas yra daugelio modeliavimo ir kitų tipų uždavinių pagrindas. Kaip gerai žinoma, dažniausiai tokie uždaviniai neturi bendro analitinio sprendinio. Todėl skaitinis diferencialinių lygčių ir jų sistemų sprendimas yra plačiai paplitęs. Jo esmę sudaro tolydžiojo sprendinio diskrečiosios aproksimacijos tam tikruose uždavinio apibrėžimo srities taškuose radimas. Ieškant apytikslio sprendinio didžiąją skaičiavimų dalį užima didelės apimties *tiesinių lygčių* sistemų, kurios gaunamos, atlikus diferencialinių lygčių diskretizaciją ir netiesiškumų linearizaciją, sprendimas.

Priklausomai nuo konkrečios užduoties diferencialinių lygčių ir jų sistemų sprendimo procedūra gali turėti savų specifinių ypatumų, naudoti skirtingas atskirų etapų (diskretizacija, linearizacija, tiesinių lygčių sistemų sprendimas) realizacijas. Todėl pasaulyje egzistuoja *nemažai* įvairių tipų diferencialinių lygčių ir jų sistemų sprendimo įrankių, ir kuriami nauji. Vienas iš tokių įrankių, detaliai aprašytas 3.2 poskyryje, yra skirtas skysčių tekėjimui poringose terpėse modeliuoti. Procesai poringose terpėse modeliuojami, sprendžiant netiesinių, stipriai susijusių *hiperbolinių* ir *parabolinių* diferencialinių lygčių sistemas.

Pastaruoju metu iš modeliavimo uždavinių reikalaujama didelio tikslumo ir sprendimo spartos. Šių tikslų įgyvendinimui savo ruožtu reikia daugiau operatyviosios atminties ir galingesnių procesorių, negu gali suteikti įprastiniai kompiuteriai. Tokiu atveju yra prasmingas lygiagrečiųjų kompiuterių naudojimas. Todėl egzistuojančių ir *naujų* diferencialinių lygčių ir jų sistemų sprendimo įrankių lygiagretinimas yra ypač aktualus.

Šiame darbe buvo siekiama sukurti technologiją, kuri galėtų būti taikoma parabolinių ir hiperbolinių diferencialinių lygčių ir jų sistemų sprendimo įrankių ly-

giagretinimui. Tam tikslui pirmiausia buvo išanalizuotos paketų, skirtų panašiems uždaviniams spręsti, lygiagretinimo strategijos. Šiame poskyryje pateikiama šitų paketų analizė ir iš jos išplaukiančios išvados.

1.2.1. Egzistuojantys įrankiai ir jų lygiagretinimo metodai

Šiame poskyryje aprašomi paketai, skirti arba tiesiogiai procesams poringose terpėse modeliuoti, arba bendri diferencialinių lygčių dalinėmis išvestinėmis sprendimo paketai.

Čia pateikti tik šių paketų aprašymai, specialų dėmesį skiriant šių paketų lygiagretinimo ypatumams. 1.2.2 poskyryje šitie ypatumai bus apibendrinami, siekiant nustatyti įrankio, skirto skysčių tekėjimui poringose terpėse modeliuoti, lygiagretinimo tikslus ir optimalius metodus.

A. Diffpack

Diffpack paketas skirtas fizikiniams procesams modeliuoti, sprendžiant diferencialinių lygčių sistemas [18]. Jis pasižymi tuo, kad jame bandoma suderinti procedūrinį programavimą su objektiškai orientuotu. Procedūrinis programavimas naudojamas skaičiavimo algoritmų realizacijai, o OOP leidžia patogiai organizuoti biblioteką ir pateikti vartotojui aiškią sąsają. Tai yra komercinis produktas.

Vartotojo požiūriu, *Diffpack* paketas yra rinkinys C++ bibliotekų su savomis klasėmis, procedūromis ir pagalbinėmis programomis. *Diffpack* architektūra atkartoja matematines koncepcijas, taikomas sprendžiant diferencialinių lygčių sistemas. Biblioteka turi savo tinklų generavimo, rezultatų vizualizavimo priemones, gali keistis duomenimis su kitais paketais.

Lygiagretinimui *Diffpack* paketas pateikia specialų modulį. Naudojant šį modulį, lygiagretinimas vykdomas tiesinės algebros lygyje, paskirstant duomenis tarp procesų. Skaldant nestruktūrinius tinklus, gali būti panaudotas *METIS*[1] paketas arba vartotojas gali taikyti savo sprendimą. Naudojant standartinius duomenų paskirstymo įrankius, programos, naudojančios *Diffpack* paketą, lygiagretinimas reikalauja kelių papildomų eilučių. Duomenims apsikeisti naudojamas MPI standartas.

B. OpenFOAM

OpenFOAM[2] paketas yra skirtas įvairių procesų modeliavimui sprendžiant diferencialinių lygčių sistemas [57]. Jis naudojamas modeliuojant tokius procesus, kaip skysčių ir kietųjų kūnų dinamika, elektromagnetiniai bei ekonominiai reiški-

[1] Internetinė prieiga `http://glaros.dtc.umn.edu/gkhome/views/metis`, [2007-06-20].
[2] Nuo angl. *Open Field Operation and Manipulation.*

niai. Šis paketas turi atviro kodo licenciją ir yra laisvai platinamas. Jo tobulinimu užsiima komercinė kompanija *OpenCFD*, gaunanti pajamų iš su *OpenFOAM* paketų susijusių paslaugų – konsultavimo ir paketo taikymo specifiniams uždaviniams.

Biblioteka parašyta C++ kalba. Ji susideda iš įvairios paskirties klasių – sprendikliai, duomenų pradinio (angl. *preprocessing*) ir galutinio (angl. *post-processing*) apdorojimo klasės, fizikiniai modeliai ir kiti. Dėl savo atvirumo ir lanksčios struktūros ji gali būti pildoma naujomis klasėmis, jos funkcionalumas nuolat plinta. Daug parengtų klasių platinama kartu su biblioteka, kas leidžia ją iškart taikyti plačiam uždavinių spektrui.

OpenFOAM paketas leidžia vartotojui dirbti su nestruktūriniais tinklais. Jame yra nuosavas tinklelio (angl. *mesh*) generatorius. Diferencialinės lygtys sprendžiamos baigtinių tūrių diskretizavimo metodais. Pakete plačiai naudojamos tokios C++ savybės, kaip OOP, operatorių perkrovimas (angl. *overloading*), šabloninis metaprogramavimas ir kitos.

Paketas palaikomas *UNIX/Linux* platformose, nes būtent jose dažniausiai kuriamos ir vykdomos skaitinio modeliavimo programos. Kitose platformose, tokiose kaip *MS-Windows* arba *MacOS*, vartotojai patys turi taikyti paketą.

Programos, naudojančios *OpenFOAM* biblioteką, lygiagretinimas atliekamas, taikant duomenų skaidymą (angl. *domain decomposition*). Lygiagretinimas atliekamas automatiškai, naudojant įrankius, pateikiamus kartu su biblioteka. Duomenų apsikeitimui tarp procesų naudojamas MPI standartas, todėl sistemoje turi būti įdiegta veikianti MPI realizacija. Lygiagretinimas atliekamas, pakeičiant programos kode tam tikras klases jų lygiagrečiomis versijomis.

Norėdamas išlygiagretinti nuoseklųjį kodą, vartotojas privalo parengti specialų failą, kuriame nurodoma, kiek turi būti procesorių ir kaip tarp jų padalijama sprendžiama užduotis. Po to paleidžiama speciali bibliotekos pagalbinė programa, kuri generuoja lygiagrečiąją programos versiją, taip pat paruošia duomenis kiekvienam procesui. Po lygiagrečios programos paleidimo kiekvienas procesas atskirai išsaugoja savo rezultatų dalį, kurios vėliau yra sujungiamos standartinėmis bibliotekos priemonėmis.

C. UG

UG[3] yra paketas, skirtas skaitiniam diferencialinių lygčių sistemų sprendimui 2D ir 3D erdvėje naudojant nestruktūrinius tinklus [86]. Jis buvo sėkmingai pritaikytas uždaviniams poringose terpėse modeliuoti [49]. Šis paketas parašytas C kalba, su galima C++ sąsaja. Siekiama, kad paketas būtų lengvai pernešamas tarp platformų. Paketas tobulinamas nuo 1990 metų. Jo kodo apimtis šiuo metu sudaro apie 450 000 eilučių. Akademiniams tikslams šis paketas naudojamas nemokamai.

[3] UG yra nuo angl. *Unstructured Grids* – nestruktūriniai tinklai.

UG pasižymi sluoksnine struktūra. Pats žemiausias sluoksnis atlieka duomenų paskirstymą. Jis leidžia gauti nuoseklųjį arba lygiagretųjį kodą. Kitas, „aukštesnis" sluoksnis yra UG paketo *branduolys*, nepriklausantis nuo konkretaus uždavinio. Jame yra įvairūs lygčių sprendimo metodai, duomenų vizualizacijos būdai, diskretizacijos tinklo sudarymo valdymas ir netgi sava skriptinė kalba, kurią naudojant programos vykdymo metu galima valdyti užduoties sprendimo procesą. Toliau eina „užduoties" sluoksnis, kuris, naudojant branduolį, leidžia spręsti įvairių klasių uždavinius. Pavyzdžiui, yra užduoties klasės, skirtos skysčių tekėjimui, daugiafaziam tekėjimui poringose terpėse modeliuoti, ir kitos. Galiausiai paskutinis – *taikomasis* sluoksnis, kuriame gali būti nurodomos konkrečios užduoties sąlygos.

UG pakete lygiagretinimas atliekamas pačiame žemiausiame lygyje. Šiame lygyje manipuliuojama dinaminėmis duomenų struktūromis. Šiose struktūrose saugomi objektai ir informacija apie kaimyninius objektus (per rodykles) – toks saugojimo būdas tinka nestruktūrinių tinklų saugojimui. Lygiagretinimas atliekamas duomenų skaidymo būdu. Struktūra pati *dinamiškai* padalija elementus tarp procesų taip, kad duomenų apsikeitimas tarp kaimynų būtų minimalus.

Paketo lygiagretinimo lygis realizuotas C kalba, su C++ sąsaja. Vartotojui jis pateikia sąsają, nepriklausančią nuo naudojamų lygiagretinimo bibliotekų. Lygiagretinimui gali būti naudojamas platus lygiagretinimo standartų spektras, iš jų galima paminėti MPI, PVM, PARIX. Dėl savo savybių šis UG paketo lygis gali būti naudojamas kaip atskira biblioteka, kurią taikant gali būti lygiagretinami įvairūs uždaviniai. Šios bibliotekos pavadinimas yra DDD (angl. *Dynamic Distributed Data* – dinamiškai paskirstyta informacija).

D. TOUGH2

TOUGH2[4] yra programinis paketas, specifiškai skirtas procesams poringose terpėse modeliuoti. Jis leidžia modeliuoti daugiafazį ir daugiakomponentį tekėjimą 1D, 2D ir 3D uždaviniuose [84]. Šis paketas yra mokamas, jis kuriamas Lawrence Berkeley nacionalinėje laboratorijoje. Šiuo metu naujausia paketo versija yra TOUGH2 2.0.

TOUGH2 paketas leidžia modeliuoti procesus su įvairiomis fazėmis, tokiomis kaip skysčiai ir dujos. Taip pat gali būti modeliuojami šilumos pernešimo reiškiniai [3], medžiagų įterpimas/išsiurbimas. Erdvės diskretizacija atliekama taikant IFD metodą, laiko diskretizacija yra pilnai neišreikštinė. Vartotojas turi platų lygčių sistemų sprendimo algoritmų pasirinkimą. Į paketą įeina keli moduliai skirtingiems specifiniams uždaviniams spręsti.

TOUGH2 pakete daugiausia dėmesio yra skiriama skaitiniam modeliavimui, o pradinių duomenų paruošimas ir rezultatų interpretacija nėra trivialus uždavinys.

[4]TOUGH yra nuo angl. *Transport Of Unsaturated Groundwater and Heat.*

Pasaulyje egzistuoja sprendimai, skirti sąsajai su TOUGH paketu supaprastinti, vienas žinomiausių yra *PetraSim*.

TOUGH2 paketas yra parašytas FORTRAN-77 kalba. Tačiau šiuo metu yra vystomas TOUGH+ paketas, kuriame TOUGH2 savybės yra realizuojamos taikant OOP technologijas, taip pat pridedamos ir naujos galimybės. TOUGH+ paketo kalba yra Fortran-95, įeitinių duomenų failų formatai suderinami su originaliu TOUGH2.

Originalus TOUGH2 paketas, kurį tobulina Lawrence Berkeley nacionalinė laboratorija, neturi lygiagretinimo palaikymo. Tačiau egzistuoja atskirai vystoma TOUGH2 paketo lygiagrečioji versija, TOUGH2-MP[5]. Šis paketas, taip pat kaip ir TOUGH2, yra komercinis.

TOUGH2-MP pakete efektyvus lygiagretinimas pasiekiamas per

- efektyvų duomenų skaidymą (naudojant METIS paketą);
- jakobianų matricų generavimo lygiagretinimą;
- tiesinių lygčių sistemų sprendimo lygiagretinimą;
- efektyvų apsikeitimą duomenimis tarp procesų, panaudojant MPI.

E. Clawpack

Clawpack[6] yra programinis paketas, skirtas hiperbolinių diferencialinių lygčių sistemų 1D, 2D ir 3D erdvėje skaitiniam sprendimui, taikant bangų sklidimo (angl. *wave propagation*) metodą [15, 24]. Jis tobulinamas Vašingtono universitete. Paketas platinamas nemokamai.

Clawpack paketas parašytas FORTRAN-77 kalba. Tai yra paprogramių biblioteka su jau sukurtais realių uždavinių sprendimo pavyzdžiais. Diferencialinės lygtys, įeinančios į sistemą, gali būti

$$k(\mathbf{x})q_t + \sum_{i=1}^{m} f(q, \mathbf{x}, t)_{x_i} = \psi(q, \mathbf{x}, t) \tag{1.1}$$

arba

$$q_t + \sum_{i=1}^{m} A_i(\mathbf{x}, t)q_{x_i} = 0 \tag{1.2}$$

pavidalo, čia $q = q(\mathbf{x}, t) \in \mathrm{Re}^m$ yra nežinoma funkcija. Gali būti naudojami tiek struktūriniai, tiek nestruktūriniai tinklai. Rezultatai gali būti išvedami skirtingo pavidalo.

[5] Internetinė prieiga `http://www.tough2.com/`, [2007-03-22].

[6] Clawpack yra nuo angl. *Conservation LAW PACKage – tvermės dėsnių paketas.*

Kaip ir kiti paketai, Clawpack turi lygiagrečią versiją, kur duomenų apsikeitimui naudojama MPI realizacija. Lygiagretinimas atliekamas, skaldant duomenis tarp procesų. Kiekvienas procesas turi savo duomenų sritį, taip pat *šešėlinius* taškus, atitinkančius kaimyninių procesų elementus, reikalingus skaičiavimams.

Lygiagretinimas atliekamas automatiškai, pakeičiant `Makefile`'ą ir nurodant procesorių topologiją ir duomenų padalijimo būdą. Reikia atkreipti dėmesį, kad ne visi Clawpack bibliotekos moduliai gali būti išlygiagretinti.

1.2.2. Egzistuojančių įrankių lygiagretinimo metodų analizė

Išanalizavus egzistuojančių diferencialinių lygčių sprendimo įrankių lygiagretinimo ypatumus, buvo pastebėti keli daugeliui paketų būdingi lygiagretinimo bruožai.

Koncepciniai panašumai:

- Lygiagretinimas dažniausiai atliekamas tiesinės algebros lygyje, lygiagretinant duomenų struktūras bei tiesinių lygčių sistemų sprendimo metodus.
- Lygiagretinimas atliekamas duomenų skaidymo metodu. Nestruktūrinių tinklų skaidymo atveju dažnai naudojamas paketas METIS.

Lygiagretinimo vartotojo sąsajos panašumai:

- Dauguma modeliavimo paketų pateikia vartotojui *pusiau automatinio* lygiagretinimo galimybes. Vartotojui nereikia daug žinoti apie lygiagretų programavimą.
- Lygiagretinant vartotojas visada turi nurodyti, kaip duomenys turi būti skaldomi tarp procesų.

Taip pat pastebėti ir lygiagretinimo realizacijos panašumai:

- Dažniausiai paketai remiasi žemo lygio lygiagretinimo standartais, o ne aukštesnio lygio lygiagretinimo bibliotekomis. Išimtis yra UG paketas, tačiau net ir tuo atveju DDD lygiagretinimo biblioteka atsirado pirmiausia UG paketo poreikiams patenkinti.
- Duomenų apsikeitimui naudojamos įvairios bibliotekos ir standartai, tačiau populiariausias yra MPI, kaip lanksčiausias, užtikrinantis efektyvumą ir programų pernešamumą (angl. *portability*).

Kadangi tiesinės algebros uždavinių sprendimas sudaro didžiąją (iki 80–90 %) tokio tipo algoritmų skaičiavimų laiko dalį [51], lygiagretinimas šiame lygyje yra teisėtas pasirinkimas. Taip pat reikia pabrėžti, kad, išlygiagretinus tiesinės algebros lygį, išlygiagretinamas ir diferencialinių lygčių diskretizacijos lygis. Nors ši dalis dažnai sudaro tik 10–25 % viso uždavinio sprendimo laiko, kartu su tiesinės

algebros lygiu tai garantuoja lygiagrečiojo algoritmo išplečiamumą (angl. *scalability*).

Tiesinės algebros uždaviniams būdingas duomenų lygiagretumas. Pastebima tendencija, kad netrivialūs duomenų padalijimo uždaviniai (nestruktūrinių tinklų atveju) sprendžiami trečios šalies specializuotais paketais.

Dirbant su visais paketais, vartotojas gali pasinaudoti pusiau automatinio lygiagretinimo galimybėmis. Pusiau automatinis lygiagretinimas, kuris atliekamas nežymios kodo dalies pakeitimu, leidžia vartotojui nesunkiai palaikyti tiek *nuosekliąją*, tiek lygiagrečiąją programos versiją. Lygiagretinimo svarba yra akivaizdi, tačiau nuosekliosios versijos atsisakymas turi savo trūkumų. Pirmiausia, nuoseklioji programos versija praverčia, kai nėra priėjimo prie lygiagrečiųjų skaičiavimo resursų. Tokiais atvejais būtinybė diegti papildomą lygiagretinimo biblioteką tik sunkina vartotojo darbą ir mažina programos efektyvumą. Taip pat nuoseklioji įrankio versija gali būti reikalinga kūrimo ir derinimo metu, kai *taisymas* → *kompiliavimas* → *paleidimas* ciklas kartojamas daugelį kartų. Reikia pastebėti, kad pusiau automatiniu lygiagretinimu pasižymi lygiagrečiųjų duomenų lygiagretinimo modelis, kai taikomas duomenų skaldymas.

Kalbant apie lygiagretinimo realizaciją, tenka pastebėti, kad išanalizuotuose paketuose naudojami lygiagretinimo algoritmai paremti žemo lygio lygiagretinimo standartais ir sujungti su paketų kodu, todėl juos panaudoti kitiems paketams lygiagretinti yra praktiškai *neįmanoma*. Siekiant efektyviai pakartotinai panaudoti lygiagretinimo kodą, jis turi būti pateiktas *aukšto lygio* lygiagretinimo bibliotekos pavidalu. MPI panaudojimas atspindi faktą, kad įrankių lygiagretinimas orientuotas į *efektyvų* panaudojimą kompiuterių *klasteriuose*. Tai natūralu, sprendžiant uždavinius, kurie reikalauja resursų, daugelį kartų viršijančių vieno kompiuterio galimybes.

Atsižvelgiant į tai, diferencialinių lygčių ir jų sistemų sprendimo įrankių lygiagretinimo technologijai buvo keliami tokie *reikalavimai*:

- Tai turi būti aukšto lygio biblioteka, jos taikymas neturi reikšti, kad diferencialinių lygčių sprendimo įrankio kūrėjas rašys daug lygiagretinimo ar kitokio specifinio kodo, būdingo tik jo įrankiui (kaip tai, pavyzdžiui, yra vien MPI panaudojimo atveju).
- Lygiagretinimas turi būti atliekamas tiesinės algebros ir diferencialinių lygčių diskretizacijos lygiuose.
- Taikant technologiją turi būti lengvai kuriamos programos, lygiagretinamos pusiau automatiškai.
- Lygiagrečioji versija turi efektyviai veikti kompiuterių klasteriuose.

1.3. Lygiagretinimo technologijų apžvalga

Šiame poskyryje pateikiama pasaulyje egzistuojančių lygiagretinimo standartų ir įrankių apžvalga ir analizė.

1.3.1. Algoritmų lygiagretinimas

Šiame skyrelyje aprašomos pagrindinės algoritmų lygiagretinimo sąvokos ir principai. Remiantis jomis, toliau šiame darbe analizuojami įvairūs algoritmų lygiagretinimo sprendimai.

A. Lygiagrečiųjų kompiuterių klasifikacija

Lygiagretieji kompiuteriai gali būti klasifikuojami įvairiai, tačiau labiausiai paplitusi yra Flino klasifikacija, naudojama nuo 1996 metų [31]. Pagal ją kompiuteriai klasifikuojami pagal tai, kaip jie lygiagrečiai gali dirbti su programa ir duomenimis:

SISD – tai yra įprasti nuoseklieji Neumano kompiuteriai. Vienu metu jie vykdo vieną instrukciją, kuri apdoroja vieną duomenų vienetą;

SIMD – tai kompiuteriai, kurie vienu metu vykdo vieną komandą, tačiau ji vienu metu gali apdoroti kelis duomenų vienetus. Prie tokio tipo kompiuterių priskiriami vektoriniai kompiuteriai [37];

MIMD – tai kompiuteriai, kurie vienu metu gali vykdyti kelias operacijas, su skirtingais duomenimis. Tokie yra daugiaprocesoriniai kompiuteriai, tokiu kompiuteriu galima laikyti SISD tipo kompiuterių klasterį;

MISD – kompiuteriai, kurie vienu metu gali vykdyti daug instrukcijų su *ta pačia* informacija. Šis tipas pateiktas dėl pilnumo, praktiškai tokių lygiagrečiųjų kompiuterių nėra.

Taigi, kalbant apie lygiagrečiuosius kompiuterius, daugiausia bus nagrinėjami SIMD ir MIMD tipo kompiuteriai.

B. Atminties paskirstymas

Lygiagretinimo priemonės ir metodai labai priklauso nuo to, kokio tipo atmintis yra lygiagrečiajame kompiuteryje, todėl čia pateiksime kompiuterių klasifikaciją. Pagal atminties tipą lygiagretieji kompiuteriai skirstomi į:

- Bendrosios (angl. *shared*) atminties – tai kompiuteriai, kur visi procesoriai turi priėjimą prie visos atminties [37]. Tačiau ir šiuo atveju galimi skirtingi variantai:

  - UMA tipo kompiuteriuose priėjimas prie bet kurios atminties vietos yra vienodas visiems procesoriams;
  - NUMA tipo kompiuteriuose procesoriai turi tam tikras atminties sritis, prie kurių priėjimas yra greitesnis, ir atvirkščiai.

  Dažniausiai tai yra MIMD tipo kompiuteriai. Prie tokių kompiuterių priskiriami ir šiuo metu populiarėjantys SMP tipo kompiuteriai, kurie turi du ir daugiau procesorių. Šio tipo atminties kompiuteriams gerai tinka daugiagijinis programavimas.

- Paskirstytosios (angl. *distributed*) atminties – tai kompiuteriai, kur kiekvienas procesas turi savo atmintį, nepasiekiamą kitiems. Tai irgi yra MIMD tipo kompiuteriai. Iš pavyzdžių galima paminėti populiarius šiuo metu kompiuterių klasterius, kur kiekvienas mazgas yra paprastas SISD kompiuteris. Šio tipo atminties kompiuteriams geriausiai tinka programavimas, paremtas pranešimų siuntimais.

Taip pat reikia pabrėžti, kad realiuose kompiuteriuose pasitaiko šitų tipų kombinacijų. Pavyzdžiui, jei kompiuterių klasteris susideda iš SMP kompiuterių, tai visa sistema turės abiejų atminties tipų požymių.

C. Lygiagrečiojo programavimo modeliai

Rašant lygiagrečiąsias programas ir lygiagretinimo įrankius, naudojami tam tikri lygiagrečiojo programavimo modeliai. Nors kai kurie modeliai gali tiesiogiai atspindėti kompiuterio, kuriame vykdomi skaičiavimai, struktūrą, praktiškai galima įvairius modelius realizuoti įvairaus tipo kompiuteriuose. Todėl modelis yra parenkamas priklausomai nuo užduoties, esamų skaičiavimo resursų ir prieinamų esamų lygiagretinimo įrankių. Negalima nurodyti vieno modelio, geriausiai tinkančio visiems gyvenimo atvejams. Šiame skyrelyje aprašomi populiariausi modeliai.

Bendros atminties modelis. Šiuo atveju egzistuoja bendra, visiems procesams prieinama atminties erdvė, į kurią visi procesai gali asinchroniškai rašyti ir skaityti. Naudojant šį modelį, nereikia galvoti apie duomenų siuntimus, tačiau tenka spręsti priėjimo prie resursų sinchronizavimo problemą, kad išvengtume duomenų gadinimo (angl. *race conditions*). Šis modelis yra natūraliai įgyvendinamas populiariuose šiuo metu SMP kompiuteriuose. Tačiau jis sudėtingai realizuojamas kitų tipų kompiuteriuose.

Gijų modelis. Paleidžiama viena programa, tačiau priklausomai nuo situacijos ji gali dinamiškai generuoti ir valdyti gijas – lygiagrečiai vykdomas instrukcijų sekas. Natūralu, kad visos gijos turi priėjimą prie tos pačios proceso atminties, todėl, naudojant šį modelį, tenka spręsti daug problemų,

aptartų ir bendros atminties modeliui. Tačiau šiame modelyje gijos turi ne tik bendrą atmintį, bet ir kitus resursus, pavyzdžiui, įvedimo/išvedimo įrenginius.

Pranešimų siuntimo modelis. Šiuo atveju procesai neturi jokios bendrosios atminties, ir duomenimis apsikeičia siunčiant ir priimant pranešimus. Dažniausiai duomenų apsikeitimui reikia mažiausiai dviejų procesų suderinto darbo. Plačiai paplitęs MPI standartas pagrįstas būtent šiuo modeliu. Toks modelis yra universalus, jis gali būti naudojamas ir bendros atminties kompiuteriuose.

Lygiagrečiųjų duomenų modelis. Jis pasižymi tuo, kad programos vykdymo metu atliekama daug vienodų veiksmų su tam tikros duomenų struktūros (pavyzdžiui, masyvo) elementais bet kokia elementų tvarka. Programuojant turi būti naudojamos specialios lygiagrečiųjų duomenų konstrukcijos (tai gali būti kalbos konstrukcijos arba klasės ir funkcijos, kurias gali pateikti speciali biblioteka), kurios paskui gali būti *nesudėtingai* išlygiagretintos.

D. Lygiagretinimo efektyvumo įvertinimas

Kadangi šiame darbe pateikiami algoritmų lygiagretinimo efektyvumo įvertinimai, šiame skyrelyje aprašyti pagrindiniai lygiagretinimo efektyvumo įvertinimo principai.

Svarbiausias dydis, apibūdinantis lygiagrečiojo algoritmo efektyvumą, yra *spartinimo koeficientas*. Spartinimo koeficientu vadinamas santykis

$$S_p = \frac{T_0}{T_p}, \tag{1.3}$$

kuris įvertina pagreitėjimą, pasiekiamą sprendžiant uždavinį su p procesoriais [37]. Idealiu atveju mūsų algoritmas gali pagreitėti tiek kartų, kiek procesų mes naudosime, t. y. $S_p = p$. Realiai spartinimo koeficientas būna mažesnis. Jei mes gauname $S_p > p$ (supertiesinis pagreitėjimas), tai dažniausiai reiškia, kad parinkto algoritmo nuoseklioji versija yra netobula ir gali būti labiau optimizuota. Kitas tokio fenomeno paaiškinimas gali būti kompiuterių aparatinėje struktūroje, kai, pavyzdžiui, dirbant su mažesniais duomenų kiekiais, efektyviau išnaudojama kompiuterio spartinančioji atmintis (angl. *cache*).

Kitas svarbus lygiagrečiųjų algoritmų rodiklis, rodantis, kokia procesorių pajėgumo dalis naudojama sprendžiant uždavinį duotuoju algoritmu, yra *algoritmo*

lygiagretinimo efektyvumo koeficientas [37]

$$E_p = \frac{S_p}{p}\,. \tag{1.4}$$

Sprendžiant tą patį uždavinį su vis didesniu procesų skaičiumi, lygiagretinimo efektyvumas mažėja. Tai yra Amdalo dėsnio pasekmė. Amdalo dėsnis užrašomas tokia formule:

$$S_p \leq \frac{1}{s + \frac{r}{p}}\,, \quad s + r = 1\,. \tag{1.5}$$

Čia s yra algoritmo nuoseklioji dalis, kuri negali būti išlygiagretinta, o r yra lygiagrečioji dalis. Tačiau, kintant uždavinio dydžiui arba kitiems uždavinio parametrams, gali keistis s ir r santykis. Dažnai, didėjant uždaviniui, išlygiagretinama uždavinio dalis auga greičiau už s, tai leidžia efektyviau lygiagretinti didesnį uždavinį vis didesniam procesų skaičiui. Jeigu uždavinio dydį matuosime sugaištu skaičiavimo laiku, tai uždavinio dydis $W = T_0$ [37]. Tada, užfiksavę E_p, iš (1.3) ir (1.4) lygčių galime gauti *izoefektyvumo funkciją*

$$W = f(p)\,, \tag{1.6}$$

kuri parodo, kaip greitai turi didėti uždavinys, kad, sprendžiant jį su vis didesniu procesų skaičiumi, resursų panaudojimo efektyvumas išliktų toks pat. Kuo šios funkcijos augimo greitis yra mažesnis, tuo lygiagretusis algoritmas yra lengviau išplečiamas.

1.3.2. Lygiagretinimo standartai ir specifinės programavimo kalbos

Šiame poskyryje aprašomi įvairūs lygiagretinimo standartai, programavimo kalbos ir *plačiai* paplitusios bendros paskirties lygiagretinimo bibliotekos.

A. PVM

PVM yra vienas seniausių lygiagrečiojo programavimo standartų, tačiau jis dar išlieka populiarus. Šis standartas realizuotas daugelyje terpių, egzistuoja didelis kiekis paketų, naudojančių PVM. Atsiranda naujos šios bibliotekos versijos (paskutinė šiuo metu yra 3.4.5) ir naujų programavimo kalbų palaikymai [63].

Šis standartas buvo kuriamas kaip mokslinis projektas ir daugiausia dėmesio buvo skiriama lankstumui ir pernešamumui. Naudojant specialią programinę įrangą PVM sujungia į klasterį tam tikrą kompiuterių skaičių. Programuotojas per specialią sąsają gali matyti visus prieinamus resursus ir dinamiškai juos panaudo-

ti uždavinio sprendimui. Apsikeitimas duomenimis tarp procesų vyksta *siunčiant pranešimus* [64]. Šis standartas pasižymi tokiomis pagrindinėmis savybėmis:

Dinamiškumas – vartotojas gali dinamiškai pareikalauti tiek skaičiavimo resursų, kiek reikia uždaviniui, ir šiuos resursus valdyti;

Apsauga nuo avarijų – jei skaičiuojant (tai gali trukti savaites) vienas iš kompiuterių išeina iš rikiuotės, PVM turi galimybę apie tai pranešti vartotojo programai, kad ji imtųsi atitinkamų veiksmų;

Paskirstytosios atminties modelis – apsikeitimas duomenimis tarp procesų vyksta pranešimų siuntimo būdu.

B. MPI

Skirtingai nuo PVM, kuriant MPI (nuo angl. *Message Passing Interface* – duomenų perdavimo sąsaja) standartą dalyvavo ne tik mokslininkai, bet ir superkompiuterių gamintojai, kurių pagrindinis tikslas buvo *efektyvus* kompiuterių resursų panaudojimas [62]. Kuriant MPI standartą, buvo atsižvelgta į daugelio kitų jau tuo metu egzistuojančių standartų panaudojimo patirtį.

MPI naudojančia programa vykdoma naudojant specialų paleidėją[7]. Paleidėjas paleidžia vykdomąjį failą kelis kartus skirtinguose kompiuteriuose. Kiekviena programa taikant specialias MPI funkcijas gali sužinoti, kiek iš viso procesų paleista, koks yra proceso numeris, taip pat ir keistis duomenimis su kitais procesais. MPI yra standartas, kuriame realizuotas išreikštinis duomenų siuntimo modelis [37, 79].

MPI turi tokias savybes:

Fiksuotas procesų skaičius – procesų skaičius apibrėžiamas paleidimo metu (tačiau programa gali naudoti ne visus procesus);

Komunikatoriai ir topologijos – nurodant komunikatorių galima išskirti procesų grupę, kurioje siunčiami pranešimai nebus supainioti su kitais pranešimais, siunčiamais tarp tų pačių procesų. Tai ypač svarbu, kuriant *naujas bibliotekas* MPI pagrindu. Taip pat, kuriant naujas grupes, galima nurodyti norimą procesų topologiją ir tikėtis, kad MPI realizacija suskirstys procesus į grupę optimalia tvarka;

Platus apsikeitimo duomenimis funkcionalumas – nors visas apsikeitimas duomenimis gali būti realizuotas taikant 6 pagrindines MPI funkcijas [37], MPI 1.1 standartas apibrėžia daugiau kaip 120 funkcijų. Jas

[7] MPI standartas neapibrėžia paleidėjo iškvietimo, tačiau dažniausiai tai yra komanda `mpirun` su specialiais argumentais

taikant galima patogiai programuoti įvairius apsikeitimus duomenimis ir tikėtis, kad jie bus efektyviai realizuoti.

MPI standarto 1.1 versija atsirado 1995 metais ir buvo netrukus realizuota daugelyje sistemų. 1997 metais buvo išleista versija 2.0. Tačiau antroji versija yra žymiai sudėtingesnė už pirmąją, joje realizuota daugiau funkcijų, todėl jos *pilna* realizacija nėra plačiai paplitusi. Egzistuoja labai daug MPI standarto realizacijų, čia galima paminėti laisvai platinamas LAM/MPI [33] ir MPICH.

Kalbant apie MPI ir C++ suderinamumą, MPI 1.1 standarte buvo apibrėžtos tik C ir FORTRAN kalbų sąsajos, tačiau netrukus atsirado ir C++ sąsajos (pavyzdžiui, OOMPI). MPI 2.0 standarte jau atsirado ir C++ sąsajos apibrėžimas.

Galiausiai galima paminėti, kad, nors PVM ir MPI standartai istoriškai buvo kuriami skirtingų žmonių grupių, siekiant skirtingų tikslų, tarp jų yra daug bendro, ir tobulėdami standartai vieni iš kitų perima privalumus. Taip, pavyzdžiui, PVM standarte atsirado MPI komunikatorių analogai, o MPI standarto antrojoje versijoje – galimybė stebėti, ar visi kompiuteriai, dalyvaujantys skaičiavimuose, funkcionuoja. Egzistuoja pasaulinio lygio konferencija, skirta abiejų standartų PVM ir MPI pasiekimams aptarti.

C. OpenMP

Skirtingai nuo anksčiau aprašytų standartų, *OpenMP* (nuo angl. *Open Multi Processing*) standartas yra skirtas kompiuteriams su bendrąja atmintimi[8], tai yra standartas, realizuojantis *daugiagijinio* programavimo modelį [37, 80]. Kadangi šį standartą palaiko daugelis gamintojų, jis tampa vienu iš *de-facto* lygiagrečiojo programavimo standartų. Vis labiau plintant *multicore* tipo kompiuteriams, *OpenMP* gali tapti viena svarbiausių algoritmų lygiagretinimo priemonių.

Šis standartas apibrėžtas dviejuose lygiuose: vieną vertus – kaip kompiliatoriaus direktyvos, kitą vertus – kaip funkcijų biblioteka. Kompiliatoriaus direktyvas gali atpažinti tik kompiliatorius su *OpenMP* palaikymu, kiti kompiliatoriai jas tiesiog ignoruos. Taigi tas pats programos kodas gali būti panaudotas gaunant ir nuosekliąją, ir lygiagrečiąją programos versiją. Tačiau dažniausiai išlygiagretinimas taikant vien tik direktyvas tinka tuo atveju, kai mes turime algoritmą su išreikštu duomenų arba užduočių lygiagretumu. Sudėtingesnių algoritmų realizacijai gali būti panaudota *OpenMP* funkcijų biblioteka.

[8] Intel kompanija siūlo *Cluster OpenMP* – komercinį priedą prie savo kompiliatoriaus, leidžiantį sukompiliuoti *OpenMP* programą taip, kad ji veiktų kompiuterių klasteryje. Tačiau tai yra tik *atskiros* standarto realizacijos praplėtimas. Be to, praktiškai dažnai vien kompiliatoriaus neužtenka, reikia, nors ir nežymaus, programos kodo pakeitimo.

OpenMP programos vykdymo esmė yra ta, kad programa vykdoma nuosekliai, tačiau tam tikrose vykdymo stadijose sukuriamos kelios papildomos gijos[9], kurios atlieka tam tikras užduotis, o toliau tęsiamas nuoseklusis programos vykdymas iki kitos algoritmo vietos, kuri gali būti išlygiagretinta.

D. HPF

HPF yra standartas, skirtas pusiau automatiniam *lygiagrečiųjų duomenų* algoritmų, realizuotų FORTRAN kalba, lygiagretinimui [27, 37]. Šis standartas realizuoja lygiagrečiųjų duomenų lygiagrečiojo programavimo modelį.

Jis remiasi Fortran 90 kalbos konstrukcijomis, kurios leidžia daugelį lygiagrečiųjų duomenų operacijų (pavyzdžiui, masyvų elementų sudėtį) realizuoti specialiomis kalbos konstrukcijomis, o ne išreikštiniu būdu. Čia pirmiausia reikėtų paminėti įvairias operacijas su Fortran masyvais, taip pat tokius operatorius, kaip `FORALL`.

Lygiagretinimas atliekamas taikant specialias HPF direktyvas, kurias įprastas Fortran kompiliatorius supranta kaip komentarus. Taikant šias direktyvas nurodoma:

Procesų topologija – kaip procesai sąveikauja tarpusavyje;

Kas lygiagretinama – kokie masyvai paskirstomi tarp procesų;

Kaip lygiagretinama – kaip paskirstomi duomenys tarp procesorių.

Paskui programa turi būti perkompiliuota specialiu kompiliatoriumi, kuris supranta HPF direktyvas.

Deja, HPF standartas realiai nepaplito. Tam yra kelios priežastys. Pirmiausia, taikant šį standartą apsikeitimas duomenimis atliekamas ne pačiu efektyviausiu būdu, programuotojas negali tam turėti įtakos, negali tuo pačiu metu vykdyti ir skaičiavimų ir apsikeitimo duomenimis. Taip pat HPF lygiagretinimo pagrindas yra Fortran kalbos masyvų funkcionalumas. Nors, palyginti su tokiomis programavimo kalbomis, kaip C/C++, matematinės operacijos su masyvais Fortrane yra labiau išvystytos, jų taikymas, o atitinkamai ir lygiagretinimo galimybės, yra ribotos. Galiausiai HPF standartas reikalauja specialaus kompiliatoriaus sukūrimo, kas yra realizuojama sunkiau, palyginti su atveju, kai lygiagretinimo funkcionalumą realizuoja biblioteka.

E. UPC

UPC yra C99 standarto išplėtimas[10], realizuojantis mišrios (ir bendrosios, ir paskirstytosios) atminties modelį [87]. Panašiai kaip OpenMP, norint pasinaudo-

[9] Norint išvengti gijų sukūrimo kaštų, papildomos gijos gali būti kuriamos programos pradžioje, o paskui tik panaudojamos.

[10] Yra ir FORTRAN 95 išplėtimas, *Co-Array Fortran*, realizuojantis tokį pat modelį

ti visu įmanomu funkcionalumu, reikia naudoti ne tik specialius C kalbos išplėtimus, bet ir tam tikras specialias bibliotekas. Tačiau, kadangi UPC išplečia C kalbą, įvedant naujas kalbos konstrukcijas, UPC programos kompiliavimas taikant standartinį C kompiliatorių *niekada* nėra įmanomas.

UPC programos vykdymas panašus į MPI, kai iškart paleidžiamas reikiamas procesų skaičius. Kiekvienas procesas žino savo identifikatorių (`MYTHREAD`) ir bendrą procesų skaičių (`THREADS`). UPC įveda naują duomenų tipų atributą `shared`, kuris nurodo, ar šis duomenų tipas yra lokalus kiekvienam procesui, ar bendras visiems. Šis atributas gali būti taikomas ir rodyklėms. Prie bendrosios atminties realizuotas NUMA priėjimo modelis. Kiekvienas procesas pagal savo poziciją gali apskaičiuoti, kuri bendrosios atminties dalis priklauso jam – taip programa gali būti optimizuota.

Kadangi UPC yra C kalbos išplėtimas, jai reikia specialaus kompiliatoriaus. Labiausiai žinomas yra *Berkley UPC* kompiliatorius, kuris yra aktyviai tobulinamas. Šis standartas negali būti taikomas C++ programoms lygiagretinti.

F. Daugiagijinio programavimo tiesioginis palaikymas

Kompiuteriuose su bendrąja atmintimi daugiagijinis (angl. *multithreading*) programavimas yra vienas iš variantų pasinaudoti papildomų procesorių galimybėmis. Daugeliu atvejų, tai yra lanksčiausias programavimo būdas, leidžiantis realizuoti plačiausią algoritmų spektrą, tačiau ir pats sudėtingiausias. Taip yra todėl, kad daugiagijinės programos turi išreikštą lygiagretumą. Didžiausios daugiagijinio programavimo problemos yra, vieną vertus, gijų konkuravimas dėl resursų, kuriuo turi pasirūpinti pats programuotojas, o kitą vertus – daugiagijinių programų vykdymo neapibrėžtumas, kuris apsunkina klaidų paiešką ir tikrinimą [42].

Kai kuriose moderniose programavimo kalbose, tokiose kaip *Java* arba *C#*, daugiagijinis programavimas apibrėžtas kalbos lygyje. Tačiau egzistuoja nemažai programavimo kalbų, tarp jų ir C/C++, kurių standartas visiškai nieko nesako apie daugiagijinį programavimą[11]. Dėl to atsirado plati įvairovė bibliotekų, skirtų daugiagijinems programoms rašyti. Iš labiausiai žinomų reikia paminėti tokias:

- POSIX gijų standartas (*pthreads*) [61], kurio realizacijos egzistuoja praktiškai visose Unix/Linux sistemose, taip pat ir Windows terpėje;
- daugiagijinio programavimo komponentės, įeinančios į tokių plačiai žinomų ir paplitusių C++ bibliotekų, kaip Boost [14], POCO, ACE, wxWidgets;

FORTRAN'ui. Tačiau, kadangi šis darbas orientuotas į C++ kalbos taikymus, šis išplėtimas, kuris yra kitos FORTRAN kalbos specifikacijos dalis, čia nenagrinėjamas.

[11] Naujoje C++ standarto versijoje manoma, kad ši situacija pasikeis.

- daugiagijinio programavimo realizacija MS-Windows, kaip labiausiai paplitusioje terpėje.

Visos jos skirtingai realizuoja arba padeda realizuoti tokius daugiagijinio programavimo komponentus, kaip *mutex*'ai, semaforai, sąlyginiai kintamieji, *kanalai* ir kiti.

1.3.3. Egzistuojančių sprendimų, palengvinančių algoritmų lygiagretinimo procesą, apžvalga

Šiame poskyryje aprašomos aukštesnio lygio lygiagretinimo priemonės, kurios palengvina tam tikros klasės uždavinių lygiagretinimą.

A. FreePOOMA

FreePOOMA yra paketo POOMA[12] atviro kodo tęsinys [25]. Šis paketas naudoja tokias C++ kalbos technologijas, kaip OOP ir šablonus, siekiant suteikti vartotojui sąsają, panašią į FORTRAN masyvus. FreePOOMA teikia vartotojui tokias klases, kaip masyvus (pilnus), šablonus (angl. *stencils*), diapazonus ir kitas. Biblioteka *neturi* išretintų matricų. Tinkamai jas panaudojant, galima rašyti tokio tipo kodą

```
1 V(I,J)=0.25*(V(I+1,J)+V(I-1,J)+V(I,J+1)+V(I,J-1)-b(I,J));
```

Biblioteka leidžia vartotojui atlikti *pusiau automatinį* lygiagrečiųjų duomenų kodo lygiagretinimą. Lygiagretinimas atliekamas, sujungiant (angl. *linking*) vartotojo programą su skirtingomis FreePOOMA bibliotekos versijomis. Skirtingos bibliotekos versijos gali būti įdiegtos, nurodant skirtingus parametrus įdiegimo metu. Taip gali būti įdiegtos dvi bibliotekos versijos, viena nuoseklioji, kita daugiagijinė. Nurodant `Makefile` faile skirtingas FreePOOMA bibliotekos versijas, iš to paties vartotojo kodo galima gauti tiek nuosekliąją, tiek lygiagrečiąją programos versijas.

Norint, kad programa būtų lengvai išlygiagretinama, visos operacijos turi būti atliekamos taikant FreePOOMA metodus, panašiai, kaip pavaizduota ankščiau. Iki paskutinės versijos FreePOOMA buvo kuriama daugiagijinio programavimo su globalia atmintimi modeliui. Globalios operacijos buvo bibliotekos sudalijamos į dalis, kurios būdavo automatiškai išdalijamos atskiroms gijoms. Paskutinėje, 2.4.1 versijoje atsirado ir paskirstytosios atminties modelio palaikymas, naudojant MPI standartą.

[12] POOMA – nuo angl. *Parallel Object-Oriented Methods and Applications*.

B. Global Arrays

Biblioteka *Globalūs masyvai* (angl. *Global Arrays*) realizuoja *bendrosios atminties* modelį kompiuteriuose su paskirstytąja atmintimi [26]. Duomenų perdavimui ji gali naudoti kitas bibliotekas, kurios palaiko tokias funkcijas, kaip pranešimo transliavimas (angl. *broadcasting*), sinchronizacija (angl. *barrier*), galimybę nutraukti lygiagrečiai vykdomą darbą, jei atsiranda klaidų. Globalūs masyvai gali naudoti MPI duomenims perduoti, ir tokiu atveju programuotojas gali savo programoje naudotis tiek globaliais masyvais, tiek MPI funkcionalumu.

Globalūs masyvai turi sąsajas C ir Fortran kalbai. C sąsaja gali būti panaudota ir C++ programoje, tačiau nebus pasinaudota visomis C++ galimybėmis. Biblioteka pateikia vartotojui kelių skirtingų duomenų tipų ir dimensijų *masyvus*. Nors fiziškai kiekvienas procesas saugoja tik dalį visų masyvo elementų, naudojama globali elementų adresacija, tai palengvina programavimą. Tačiau bibliotekoje vis tiek naudojamas išreikštinio lygiagretumo principas. Norint dirbti su tam tikrais masyvų elementais, juos išreikštiniu būdu reikia nukopijuoti į privačią atmintį, panaudoti, ir, jei yra pakeitimu, įrašyti atgal.

Globalūs masyvai nėra matematiniai objektai. Tiesinės algebros operacijos turi būti realizuojamos išreikštiniu būdu. Globalūs masyvai būna tik užpildyti (angl. *dense*), išretintos duomenų struktūros nepalaikomos.

C. TBB

TBB (angl. *Threading Building Blocks*) yra C++ biblioteka, realizuojanti palyginamą su OpenMP funkcionalumą standartinėmis C++ kalbos priemonėmis [83]. Vartodama tokias STL sąvokas, kaip *talpyklos* (angl. *containers*), *iteratoriai* ir kitas, joje realizuotas funkcionalumas, leidžiantis programuotojui greitai ir patogiai kurti lygiagrečiųjų duomenų algoritmus SMP tipo kompiuteriams.

Ši biblioteka atsirado kaip komercinis Intel kompanijos[13] produktas. Tačiau 2007 m. liepos 24 d. bibliotekos kodas buvo išleistas pagal GPL licenciją, tai daro šią biblioteką prieinamą visiems. Nepaisant to, kad iš pradžių biblioteka buvo taikoma Intel procesoriams, šiuo metu ji palaiko ir kitus procesorių tipus (pavyzdžiui, AMD).

Ši biblioteka yra geras C++ kalbos naudojimo pavyzdys kuriant lygiagretinimo įrankius. Ateityje ji gali būti plačiai naudojama, kuriant lygiagrečias programas terpėse, kuriose *naudojamas* kompiliatorius nepalaiko OpenMP standarto. Kaip ir OpenMP, ji yra ypač aktuali *multicore* tipo kompiuteriuose. Vystant teikiamos lygiagretinimo technologijos realizaciją, ParSol biblioteką, gali būti panaudotos ir TBB galimybės.

[13] Intel, beje, buvo viena iš pirmųjų kompanijų, realizavusių OpenMP standartą savo kompiliatoriuje.

D. PETSc

PETSc[14] paketas susideda iš duomenų struktūrų ir paprogramių, skirtų skaitinio modeliavimo, sprendžiant diferencialinių lygčių sistemas, lygiagrečiųjų algoritmų programavimui [60]. Apsikeitimui duomenimis naudojamas MPI standartas.

Į PETSc paketą įeina nemažai lygiagrečiųjų tiesinių ir netiesinių lygčių sprendiklių, lygiagrečiųjų matematinių duomenų struktūrų, taip pat operacijų su jomis. PETSc naudoja įvairių lygiagrečiojo programavimo modelių elementus, tokius kaip, pavyzdžiui, globalios atminties modelį, kai bet koks procesas gali operuoti su bet kokiais lygiagretaus masyvo elementais. Nuosekliosios programos, parašytos taikant PETSc, *nėra* automatiškai lygiagretinamos.

PETSc paketas realizuotas C kalba, jo C sąsaja gali būti panaudota ir rašant C++ programas. Taip pat egzistuoja sąsajos FORTRAN kalbai ir tokioms paskutiniu metu populiarėjančioms skriptinėms kalboms, kaip *Python*. Nors paketas parašytas C kalba, jame taikoma daugelis technologijų, tokios kaip duomenų inkapsuliacija ir polimorfizmas, kurios yra lengviau realizuojamos naudojant C++ kalbą. Patys kūrėjai pripažįsta, kad C++ kalba geriau tiktų PETSc galimybių realizacijai, tačiau C kalba buvo pasirinkta dėl jos brandumo ir paplitimo.

1.3.4. Lygiagrečiojo programavimo standartų ir bibliotekų analizė

Čia buvo išvardyti lygiagretinimo standartai ir kiti sprendimai, realiai naudojami tiesinės algebros uždaviniams lygiagretinti. Čia analizuosime, ar jie atitinka 1.2.2 poskyryje iškeltus reikalavimus.

Pirmiausia aptarsime standartus ir įrankius, skirtus *daugiagijiniam* programavimui palaikyti, tokius kaip OpenMP, TBB ir kitus. Šis lygiagretinimo modelis realizuojamas kompiuteriuose su bendrąja atmintimi. Kadangi siekiama lygiagretinti diferencialinių lygčių sprendimo įrankius kompiuterių klasteriuose (t. y. sistemose su paskirstytąja atmintimi), šitie sprendimai *neatitinka* mūsų reikalavimų. Tačiau, plintant SMP tipo kompiuteriams, ateityje šios bibliotekos gali būti papildomai naudojamos siūlomos technologijos realizacijoje, norint geriau panaudoti kiekvieną klasterio kompiuterį. Ypač reikia atkreipti dėmesį į OpenMP standartą, vis didesnę reikšmę įgyjantį tarp lygiagretinimo priemonių.

Sistemas su paskirstytąja atmintimi efektyviausiai panaudoja sprendimai, paremti *duomenų persiuntimo* lygiagrečiojo programavimo modeliu, tokie kaip MPI ir PVM. Deja, tai yra žemo lygio modelis, reikalaujantis iš programuotojo daug pastangų. Maža to, duomenų persiuntimas yra išreikštinio lygiagrečiojo programavimo modelis, todėl gali būti *esminių skirtumų* nuoseklios ir lygiagrečiosios

[14] PETSc – nuo angl. *Portable, Extensible Toolkit for Scientific Computation.*

programos kode. Todėl galima teigti, kad sprendimai, realizuojantys duomenų persiuntimo modelį, *netenkina* iškeltų reikalavimų. Tačiau tokie sprendimai gali būti aukštesnio lygio technologijų pagrindas. Iš diferencialinių lygčių sprendimo įrankių lygiagretinimo analizės matėme, kad tokio tipo uždaviniuose plačiai naudojamas MPI standartas. PVM geriau tinka kito tipo, pavyzdžiui, šakų ir rėžių algoritmų [35] lygiagretinimui.

Taip pat tiesinės algebros algoritmų lygiagretinimui plačiai naudojamas *globalios atminties* lygiagretinimo modelis, kuriam priklauso tokie lygiagretinimo sprendimai, kaip UPC, Global Arrays ir PETSc. Šio modelio privalumas yra tai, kad apsikeitimas informacija tarp procesų vyksta aiškia ir programuotojui suprantama forma. Tačiau šis metodas turi ir savų trūkumų:

Lygiagretinimo efektyvumo praradimas. Iš tikrųjų bendrosios atminties modelis geriausiai realizuojamas SMP tipo kompiuteriuose. Kompiuteriuose su paskirstytąja atmintimi vis tiek turi būti atliekamas duomenų persiuntimas. Darant prielaidą, kad bet kas ir bet kada gali naudotis visais duomenimis iš bet kur, pirmiausiai siekiama, kad operacijos būtų vykdomos teisingai, nors ir neefektyviai.

Pusiau automatinio lygiagretinimo neužtikrintumas. Pirmiausia, taip yra todėl, kad globalios atminties naudojimas reikalauja, jog vartotojas užtikrintų, kad nebūtų resursų pasidalinimo konfliktų. Kita problema yra ta, kad, norint efektyviai išlygiagretinti algoritmą, vis tiek turi būti atsižvelgta į sistemos atminties heterogeniškumą. Tam specialiai turi būti nustatoma, kokia atminties sritis priklauso tam pačiam procesui, ir atitinkamai modifikuojamas lygiagretinimo algoritmas.

Šitie trūkumai leidžia teigti, kad globalios atminties lygiagretinimo modelis irgi *netenkina* mūsų suformuluotų reikalavimų.

Galiausiai turime *lygiagrečiųjų duomenų* lygiagretaus programavimo modelį, kurį realizuoja tokie standartai ir bibliotekos, kaip HPF ir FreePOOMA. Šis modelis leidžia atlikti pusiau automatinį algoritmų lygiagretinimą. Tačiau jo problema yra tame, kad šio tipo lygiagretinimo įrankiai tinka *siaurai* uždavinių klasei, kurią apsprendžia įrankyje realizuotos lygiagretinamos struktūros ir operacijos su jomis. Taip pat, kai duomenų apsikeitimas yra neišreikštinis ir vartotojas negali jo valdyti, *mažėja* lygiagretinimo efektyvumas. Šios ir kitos priežastys lėmė HPF nesėkmes. Galima teigti, kad išnagrinėtos šio lygiagretaus programavimo modelio realizacijos *netenkina* mūsų reikalavimų.

1.4. Pirmojo skyriaus rezultatai ir išvados

Šis disertacijos skyrius skirtas egzistuojančių lygiagretinimo sprendimų nagrinėjimui ir lyginamajai analizei. Pasiekti tokie *rezultatai*:

1. Išanalizuoti populiariausių hiperbolinių ir parabolinių diferencialinių lygčių sprendimo įrankių UG, Diffpack, TOUGH2, OpenFOAM ir kitų lygiagretinimo metodai. Nustatyta, kad visi jie turi bendrų savybių – lygiagretinami lygiagrečiųjų duomenų tiesinės algebros uždaviniai duomenų skaidymo metodu. Žiūrint iš vartotojo pusės, lygiagretinimas atliekamas pusiau automatiškai. Lygiagretinimo realizacijai dažniausiai naudojami plačiai paplitę *žemo lygio* standartai, ypač MPI.

2. Išanalizuoti egzistuojantys lygiagretinimo sprendimai, siekiant nustatyti, ar jie atitinka keliamus reikalavimus. Buvo analizuojami įvairūs lygiagretaus programavimo modeliai ir juos realizuojantys lygiagretinimo standartai bei programavimo kalbos, tokios kaip PVM, MPI, OpenMP, HPF ir kitos, taip pat aukštesnio lygio lygiagretinimo bibliotekos, tokios kaip Global Arrays, PETSc, FreePOOMA, TBB. Nustatyta, kad nė vienas iš jų neatitinka iškeltų reikalavimų.

Įvertinus šiuos rezultatus, gali būti padarytos tokios *išvados*:

1. Egzistuojančių populiariausių hiperbolinių ir parabolinių diferencialinių lygčių ir jų sistemų sprendimo įrankių lygiagretinimo metodai yra efektyvūs, tačiau jų realizacija atlikta žemame lygyje ir stipriai susijusi su konkrečiais įrankiais, todėl *negali* būti automatiškai naudojama kitiems įrankiams lygiagretinti.

2. Diferencialinių lygčių sprendimo įrankių lygiagretinimo technologija *turi užtikrinti*, kad ja paremtas lygiagretinimas pasižymėtų *savybėmis*, pastebėtomis nagrinėjant egzistuojančių diferencialinių lygčių sprendimo įrankių lygiagretinimo metodus. Tam ji turi būti aukšto lygio, skirta lygiagretinimui diferencialinių lygčių diskretizacijos ir tiesinės algebros lygiuose, efektyviai veikti kompiuterių klasteriuose, ir taikant ją turi būti lengvai kuriamos programos, lygiagretinamos pusiau automatiškai.

3. Kadangi nustatyta, kad iš nagrinėtų lygiagretaus programavimo modelių ir juos realizuojančių lygiagretinimo standartų ir aukštesnio lygio lygiagretinimo bibliotekų nė vienas *neatitinka* reikalavimų, keliamų diferencialinių lygčių sprendimo įrankių lygiagretinimui, *reikalinga* nauja tokio tipo uždavinių lygiagretinimo technologija.

2

Skaičiavimo matematikos algoritmų lygiagretinimo technologija

2.1. Įvadas

1 skyriuje buvo pateikta diferencialinių lygčių sprendimo algoritmų lygiagretinimo analizė, nustatyti tokio tipo uždavinių lygiagretinimo metodų reikalavimai, atlikta egzistuojančių lygiagretaus programavimo modelių ir sprendimų analizė. Buvo nustatyta, kad esamos lygiagretinimo technologijos neatitinka reikalavimų, keliamų diferencialinių lygčių sprendimo algoritmų lygiagretinimo technologijoms.

Šiame skyriuje pateikiama nauja lygiagretinimo technologija ir jos realizacija. Šioje technologijoje siekiama išvengti egzistuojančiuose sprendimuose pastebėtų trūkumų. Remiantis atlikta analize, pateikiami bendri naujos technologijos principai. Toliau eina reikalavimai bibliotekos realizacijai ir jų analizė. Šie reikalavimai lėmė bibliotekos realizacijos, kuri detaliai aprašyta 2.4 poskyryje, ypatumus. Po to pateikiami įvairūs bibliotekos veikimo efektyvumo palyginimai su alternatyviais tam tikrų uždavinių realizacijos būdais. Skyrių užbaigia bibliotekos vystymo ateities planai ir išvados. Šiame skyriuje pateikti rezultatai skelbti [92, 94–96, 98, 99, 101, 102] darbuose.

2.2. Naujos lygiagretinimo technologijos principai

2.2.1. Technologijos pagrindinės idėjos

Siekiant įveikti 1.3.4 poskyryje išvardintų lygiagretinimo modelių trūkumus, ir pilnai patenkinti reikalavimus, iškeltus 1.2.2 poskyryje, siūloma nauja lygiagretinimo technologija. Ši technologija paremta lygiagrečiųjų duomenų modeliu, tačiau įtraukia ir kai kurias globalios atminties modelio savybes.

Kadangi ši technologija yra pagrįsta lygiagrečiųjų duomenų modelių, joje turi būti realizuotos nuosekliosios tiesinės algebros objektų (masyvų, vektorių, matricų ir kitų) versijos, kurios lygiagretinimo metu pakeičiamos lygiagrečiosiomis versijomis. Turi būti realizuotos ir kai kurios globalios operacijos su tiesinės algebros objektais, pavyzdžiui, įvairių vektorių normų skaičiavimas. Taip pat lygiagretinimo metu turi būti nurodoma, kaip duomenys turi būti dalinami tarp procesų.

Panaši metodika naudojama ir HPF atveju. Čia lygiagretinami Fortran masyvai ir *globalios* operacijos su jais. Tačiau, kaip parodė praktika, net ir platus Fortran globalių operacijų su masyvais spektras yra nepakankamas sudėtingų uždavinių sprendimui. Tai buvo viena iš HPF nepopuliarumo priežasčių. Atsižvelgiant į tai, naujoje technologijoje, vietoj bandymo atkartoti ir net pranokti *visą* Fortran masyvų funkcionalumą, kaip tai yra daroma, pavyzdžiui, FreePOOMA bibliotekoje, siūloma įvesti kai kuriuos globalios atminties modelio elementus. Kaip žinome, globalios atminties modelis leidžia realizuoti sudėtingus algoritmus.

Kaip ir globalios atminties modelyje, siūloma naudoti bendrą masyvų elementų adresaciją. Tačiau, skirtingai nuo šio modelio, įvedami griežti reikalavimai, koks procesas gali pasiekti kokius kaimynams priklausančius elementus. Šitie reikalavimai išplaukia iš diferencialinių lygčių sprendimo ypatumų. Kaip yra gerai žinoma, skirtuminės schemos reikalauja apdorojamo elemento kaimyninių elementų pasiekimo. Tam pačiam uždaviniui reikalingų kaimyninių elementų padėtis dažniausiai nesikeičia, priešingu atveju galima tiesiog reikalauti galimybės pasiekti daugiau kaimyninių elementų, kad užtektų visais atvejais. Tokia kaimyninių elementų visuma vadinama šablonu (angl. *stencil*). Žinant šabloną, galima optimizuoti duomenų apsikeitimą, tačiau procesai turės priėjimą tik prie tų kaimyninių elementų, kuriuos reikalauja šablonas.

HPF kompiliatorius bando nustatyti šablonus *automatiškai*, analizuodamas programos kodą. Dažnai pasirenkami neoptimalūs šablonai, tai lemia *nereikalingų* duomenų siuntimą. Dėl šios (ir ne tik) priežasties HPF ne visada generuoja optimalų lygiagretųjį algoritmą. Naujoje technologijoje siūloma vartotojui pačiam nurodinėti savo šablonus. Tokiu būdu gali būti pasiektas minimalus persiunčiamų duomenų kiekis. Šablonų informacija gali būti reikalinga ne tik lygiagretinimui,

todėl paprašant vartotoją nurodyti šablonus jau nuosekliojoje programoje, sumažiname programos pakeitimų apimtį ją lygiagretinant.

Siekiant išvengti globalios atminties modelio nepatogumų, susijusių su lokalios bendros atminties dalies nustatymu, technologijos vartotojo sąsaja turi būti organizuota taip, kad vartotojui būtų patogu naudotis metodais, apsprendžiančiais vietinę masyvų dalį, net nuosekliojoje programos versijoje. Tokiu būdu vėl sumažinamas lygiagretinimo pakeitimų skaičius, priartėjant prie pusiau automatinio lygiagretinimo tikslo.

Kita vieta, kur HPF atveju gali būti prarandamas lygiagretinimo efektyvumas, yra duomenų apsikeitimo organizacija. Nėra žinoma, kaip tai daroma kiekvienoje konkrečioje HPF realizacijoje. Turėdami minimalų duomenų apsikeitimo (masyvų elementų sinchronizacijai) valdymą, vartotojui suteiktume galimybę efektyviai derinti skaičiavimus su duomenų apsikeitimu. Tam naujoje technologijoje numatyta vartotojui leisti pačiam inicijuoti duomenų apsikeitimą. Papildomų komandų įterpimas nežymiai prisidėtų prie pusiau automatinio lygiagretinimo kaštų.

Analizuojant HPF nesėkmės priežastis, taip pat reikėtų paminėti tai, kad lygiagretinimas pasiekiamas įvedant naujas *programavimo kalbos* konstrukcijas. Tai veda prie naujos programavimo kalbos kūrimo. Naujos programavimo kalbos kūrimas yra ne tik sudėtingesnis, reikalaujantis daugiau laiko ir pastangų, bet ir, jeigu tai nėra oficialus kalbos standartas, reikalaus ateity daugiau palaikymo, siekiant neatsilikti nuo motininės programavimo kalbos ir jos standartinių bibliotekų vystymosi. Pagal galimybes reikia siekti, kad nauja technologija būtų realizuota bibliotekos pavidalu, nekuriant naujos programavimo kalbos.

Galiausiai naujoje technologijoje turi būti realizuoti ne tik masyvai ir vektoriai, bet ir kitos lygiagretinamos matematinės konstrukcijos. Tai irgi yra vienas iš HPF trūkumų, nes Fortranas neturi standartinių duomenų struktūrų tokiems objektams, kaip, pavyzdžiui, išretintos matricos ir nestruktūriniai tinklai, vadinasi, HPF negali jų išlygiagretinti. Svarbiausia konstrukcija, naudojama tiesinių lygčių sistemų, atsirandančių sprendžiant diferencialines lygtis, yra *išretinta matrica*.

2.2.2. Technologijos detalizavimas

Šiame poskyryje aprašomos pasiūlytos technologijos detalės. Iš pradžių aprašomi pagrindiniai technologijos elementai. Dažnai naudojamas terminas *objektas* čia reiškia tam tikrą duomenų struktūrą ir su ją susijusių veiksmų visumą. Nors šis apibrėžimas iš esmės sutampa su objektais OOP prasme ir jų realizacija OOP priemonėmis yra natūrali ir patogi, tai nereiškia, kad OOP yra vienintelis šios technologijos realizacijos būdas. Kiti būdai galėtų būti, pavyzdžiui, nepermatomų rodyklių (angl. *opaque pointers*) taikymas, kaip tai daroma MPI ir PETSc sprendimuose.

Galiausiai aprašomi technologijos naudojimo principai, kuriant tiek nuoseklią, tiek lygiagrečiąją programos versijas. Jei nuoseklioji programos versija parašyta laikantis šių principų, tai jos lygiagretinimas yra pusiau automatinis.

A. Masyvai

Skaičiavimo eigoje dažniausiai operuojama dideliais duomenų masyvais. Pagrindiniai naujos technologijos objektai yra *masyvai*. Juos turime naudoti informacijos, kuri vėliau turės būti padalinta procesams, saugojimui. Kadangi programuojant naudojami skirtingi duomenų tipai, *skirtingų* duomenų tipų masyvų realizacija praplėstų technologijos panaudojimo galimybes.

A.1. Dimensijos. Kaip ir daugumoje programavimo kalbų, tokių kaip Fortran, siūlomi masyvai turi būti įvairių dimensijų, svarbiausios, kaip dažniausiai naudojamos, yra `1D`, `2D`, `3D`. Kaip ir Fortran masyvai, naujų masyvų elementai gali turėti adresus nebūtinai prasidedančius nuo 0 arba 1. Jei visi masyvo elementai saugojami vidiniame vienmačiame masyve, tai bet kokio masyvo elemento adresas $\mathbf{x}$ gali būti išreikštas formule

$$addr = \sum_{i=1}^{D} \left((x_i - st_i) \prod_{j=1}^{i-1} sz_j \right), \tag{2.1}$$

čia D yra dimensijų skaičius, st_i yra masyvo elementų i-osios dimensijos indeksų pradžia, sz_i – dydis, taip kad elementų i-osios dimensijos paskutinis indeksas yra $fin_i = st_i + sz_i - 1$. (2.1) lygtis gali būti optimizuota į

$$addr = a_0 + \sum_{i=1}^{D} x_i a_i \,, \tag{2.2}$$

čia koeficientai $a_0 = -\sum_{i=1}^{D} \left(st_i \prod_{j=1}^{i-1} sz_j \right)$ ir $a_i = \prod_{j=1}^{i-1} sz_j$ gali būti apskaičiuoti iš anksto. (2.2) lygtis leidžia apskaičiuoti masyvo elemento adresą optimaliu būdu.

A.2. Indeksų diapazonai. Tačiau skirtingai nuo, pavyzdžiui, Fortran masyvų, siūlomi masyvai pagal kiekvieną dimensiją gali turėti kelis skirtingus indeksų reikšmių diapazonus:

Globalus diapazonas – tai yra indeksų reikšmių diapazonas, kuriam priklauso visų masyvo elementų duotosios dimensijos adresai. Tai yra visos užduoties teorinis sprendimo diapazonas, paprastai jis yra nurodomas, ku-

riant masyvus. Praktiškai masyve gali būti paskiriami elementai visai kitame diapazone, todėl vien globalaus diapazono neužtenka.

Realus diapazonas – tai yra indeksų reikšmių diapazonas, kuriuo duotasis procesas gali *pasiekti* realiai egzistuojančius masyvų elementus. Visi realiai naudojami masyvo elementai *privalo* turėti adresus iš šio diapazono. Tačiau, siekiant pagreitinti programos veikimą šį sąlyga gali būti netikrinama. Tai yra vartotojo atsakomybė.

Lokalus diapazonas – tai yra diapazonas, kuriame su masyvo elementais galima *saugiai* atlikinėti visas pagal algoritmą numatytas operacijas. Tai yra *dažniausiai* skaičiavimuose naudojamas diapazonas. Kadangi veiksmus galima atlikti tik su egzistuojančiais elementais, lokalus diapazonas yra realaus diapazono dalis.

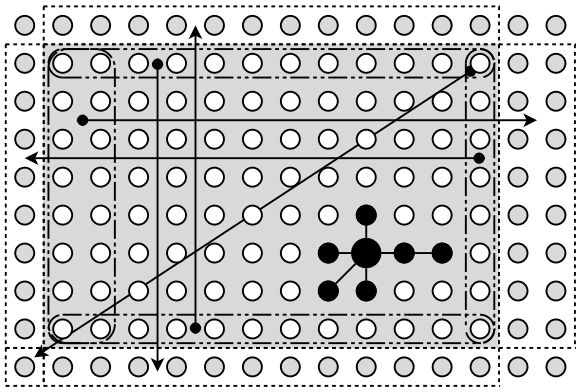
Šitie diapazonai reikalingi globalios atminties lygiagretaus programavimo modelio funkcionalumo realizacijai. Vartotojui naudojant lokalų diapazoną ciklų organizavimui, jo kodas vėliau galės būti pusiau automatiškai išlygiagretintas.

Nuosekliojoje versijoje masyvo lokalus ir globalus diapazonas sutampa, nes masyvas savyje laiko visus elementus, reikalingus uždaviniui. Jei masyvas nėra ciklinis, tai jokio duomenų apsikeitimo nereikia, ir realus diapazonas irgi sutampa su likusiais. Tačiau jei masyvas yra ciklinis, tai net ir nuosekliojoje versijoje gali būti reikalingas duomenų apsikeitimas, todėl realus diapazonas bus didesnis. Lygiagrečiojoje versijoje masyvai turi tik dalį visų elementų, todėl lokalus diapazonas būna mažesnis už globalų. Taip pat masyvai keičiasi duomenimis su kaimynais, todėl realus diapazonas būna didesnis už lokalų ir visi diapazonai dažniausiai būna skirtingi.

A.3. Ciklišku mas. Cikliniai (arba kitaip periodiniai) masyvai skirti periodiniams uždaviniams spręsti, kur masyvo elementams galioja sąlyga (cikliniams masyvams pirmosios koordinatės atžvilgiu)

$$a(i,j) = a\Big(x_{start} + \big((i - x_{start}) \mod x_{size}\big), j\Big)\,. \tag{2.3}$$

Iš (2.3) formulės aišku, kaip gali būti realizuotas ciklinių masyvų elemento adreso skaičiavimas, tačiau toks algoritmas pailgina elemento adreso paskaičiavimo laikus. Todėl siūloma nenaudoti elementų adresų perskaičiavimą pagal (2.3) formulę, o paskirti papildomus elementus už masyvo ribų pagal šablono konfigūraciją, kaip tai parodyta 2.1 paveiksle. Nuosekliųjų periodinių masyvų lokalus elementų adresų diapazonas sutampa su globalių (schemoje – centrinė masyvo dalis su tamsesniu fonu), tuo tarpu realus diapazonas yra platesnis.



2.1 pav. *Dvimačio periodinio masyvo realizacijos schema*

Tokiu metodu realizuotas masyvų ciklišKumas turi tam tikrus ypatumus:

- CiklišKumas realizuotas tik elementams, nutolusiems nuo kraštinių ne daugiau kaip maksimalus šablono taškų atstumas atitinkama kryptimi. Daugeliui lygiagrečiųjų uždavinių algoritmų to pakanka.
- Papildomi elementai (schemoje – pilkos spalvos) turi turėti atitinkamas reikšmes. Tai įvykdoma, kopijuojant atitinkamų lokalių elementų (schemoje pažymėti ovalais) reikšmes į tam tikrus išorinius elementus (schemoje pažymėti stačiakampiais). Tačiau kopijavimo procesas *nėra automatinis*, ir jį turi inicializuoti vartotojas. Tai atliekama taikant metodą `ExchangeData()`. Nors atrodo, kad tai gali reikalauti papildomo kodo rašymo iš vartotojo pusės, realiai vartotojui vis tiek teks rašyti šį kodą, norint užtikrinti duomenų apsikeitimą lygiagrečiojoje programos versijoje (apie tai bus parašyta vėliau).

Pagrindinis šio metodo privalumas yra tas, kad periodinių masyvų atveju elemento pasiekimo kaštai (o tai yra dažniausiai naudojama operacija) *nėra didesni*, lyginant su neperiodiniais masyvais. Tai pateisina šio metodo ankščiau išvardintus apribojimus ir papildomą atminties naudojimą (jeigu bendras D-dimensinio masyvo elementų skaičius yra N, tai ciklišKumas papildomai reikalauja $O\left(N^{\frac{D-1}{D}}\right)$ elementų).

B. Vektoriai

Realizuojant skaitinius (ir ne tik) algoritmus, matematiniai *vektoriai* dažniausiai saugomi duomenų struktūrose, vadinamose masyvais. Tačiau vektoriai matematikoje yra daugiau, negu sutvarkyta skaičių aibė, kurioje kiekvienas elementas turi savo unikalų indeksą. Vektoriai yra objektai, kuriems yra apibrėžtos įvairios

operacijos, tokios kaip sudėtis, daugyba iš skaliaro, normų skaičiavimas ir kitos. Vartotojas privalo naudotis šiomis matematinių operacijų realizacijomis, nes tik tuo atveju jos bus automatiškai išlygiagretintos.

Vektoriai išplečia masyvų funkcionalumą, tačiau tai uždeda papildomus reikalavimus vektorių duomenų tipams (pavyzdžiui, duomenų tipai turi palaikyti tokias operacijas, kaip sudėtis, atimtis, daugyba iš skaliaro, modulio apskaičiavimas ir kitas). Atskiriant masyvus nuo vektorių, praplečiamas technologijos funkcionalumas ir taikymo ribos, nes atskiriamas bendras lygiagretinimo funkcionalumas nuo matematinio technologijos funkcionalumo.

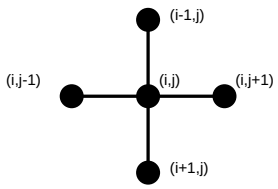
Kadangi vektoriai turi turėti tokias pat savybes, kaip ir masyvai, ir dar daugiau, tai, programuojant OOP metodais, natūralu juos daryti masyvų palikuoniais.

C. Šablonai

Šablonai reikalingi, norint užtikrinti teisingą ir efektyvų duomenų tarp procesų apsikeitimą. Tokia informacija yra gerai žinoma vartotojui, ji yra būtina ir realizuojant nuosekliuosius algoritmus.

Šablonas (angl. *stencil*) yra informacija, apibrėžianti, kokie kaimyniniai masyvo elementai reikalingi, siekiant atlikti įvairias operacijas (priklausomai nuo konkretaus algoritmo) su kiekvienu masyvo elementu. Kadangi biblioteka skirta lygiagrečiųjų duomenų algoritmų lygiagretinimui, daroma prielaida, kad su visais masyvo elementais atliekama tokia pati operacija, išimtį gali sudaryti visos srities kraštiniai elementai.

Pavyzdžiui, sprendžiant vaizdų filtravimo uždavinius, aprašytus 3.3 poskyryje, ir pasinaudojant (3.33) išreikštine schema, mes matome, kad tam, kad apskaičiuoti sekančio žingsnio elementą $U_{i,j}^{n+1}$, reikalingi elementai $U_{i,j}^{n}$, $U_{i+1,j}^{n}$, $U_{i-1,j}^{n}$, $U_{i,j+1}^{n}$ ir $U_{i,j-1}^{n}$. Šitie elementai grafiškai pavaizduoti 2.2 paveiksle. Šablonas, atitinkantis tokį algoritmą, vadinamas *kryžminiu*. Priklausomai nuo uždavinio, šablonai gali būti įvairiausių formų, ir turėti tiek dimensijų, kiek dimensijų turi naudojami masyvai arba vektoriai.



2.2 pav. *Kryžminio šablono schema*

Rekomenduojama šablonus parinkti taip, kad jie apibrėžtų tik *naudojamus* kai-

myninius elementus, bet *ne daugiau*. Jei bus parinktas per didelis šablonas, tai gali neigiamai paveikti programos efektyvumą, nes

- gali būti paskiriama daugiau negu reikia atminties;
- gali vykti nereikalingi duomenų apsikeitimai su kaimynais, arba apsikeičiama per dideliu duomenų kiekiu.

Per didelis pasirinktas šablonas neturės įtakos skaičiavimo rezultatams. Tuo tarpu, jei pasirinktas šablonas neįtraukia visų naudojamų kaimyninių elementų, gali įvykti kreipinys į neegzistuojantį masyvo elementą. Tokio kreipinio pasekmės gali būti įvairios, pradedant nuo avarinio programos užbaigimo su klaida ir baigiant klaidingo rezultato gavimu. Tokios pasekmės yra labai nepageidautinos, todėl vartotojas turi įsitikinti, kad šablonas *įtraukia visus* naudojamus kaimyninius elementus.

D. Matricos

Matematikoje, *matrica* (angl. *matrix*) yra stačiakampė skaičių lentelė, kuriai apibrėžtos specifinės operacijos, pavyzdžiui, jos elementai gali būti sudedami ir dauginami iš skaliaro. Matricos naudojamos tiesinių lygčių sistemų aprašymui, tiesinių transformacijų koeficientų laikymui. Matricos yra vienos svarbiausių objektų tiesinėje algebroje.

Matricų ir vektorių skirtumas yra tas, kad vektorių (ir masyvų) daugiamatiškumas yra tik patogi *elementų adresavimo* forma[1]. Tai, kad į matricą negalima žiūrėti kaip į vektorių su dvimate elementų adresacija, tampa aišku, nagrinėjant matricos ir vektoriaus sandaugą. Atlikę matricos ir vektoriaus sandaugos veiksmą $y = \mathbf{A}x$ gauname kitą vektorių, kurio kiekvienas elementas apskaičiuojamas pagal gerai žinomą formulę

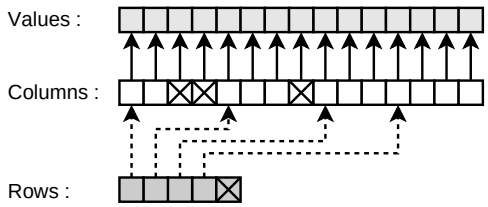
$$y_i = \sum_{j=1}^{n} (\mathbf{A}_{ij} x_j) \ . \tag{2.4}$$

Iš šios formulės matome, kad kiekviena matricos *eilutė* atitinka tam tikrą *rezultatinio* vektoriaus *Y* elementą, o kiekvienas *stulpelis* – tam tikrą vektoriaus *X*, su kuriuo atliekama *sandauga*, elementą. Taip pat galime pastebėti, kad pradinis ir rezultatinis vektoriai gali skirtis savo *struktūra* (elementų, dimensijų skaičiumi), tačiau visi galimi pradiniai vektoriai turi turėti vienodą struktūrą (tas pats galioja ir rezultatiniams vektoriams).

Siūlomoje technologijoje *matricos* yra objektai, kurie gali dalyvauti matricos ir vektoriaus sandaugos operacijoje, aprašytoje ankščiau. Taip pat turi būti realizuotos tokios dažnai su matricomis atliekamas operacijos, kaip ILU faktorizaciją.

[1] Iš tikrųjų šis teiginys teisingas tik žiūrint *matematiškai*. Masyvų dimensijų skaičius vaidina svarbų vaidmenį lygiagretinimo metu.

Matricų vidinė struktūra gali būti įvairi. Svarbu realizuoti *išretintas* (angl. *sparse*) ir *pilnas* (angl. *dense*) matricas. Pilna matrica paskiria visus elementus, reikalaudama $O\left(N^2\right)$ atminties, kur N yra X (arba Y) masyvų elementų skaičius. Išretinta matrica paskiria $\approx \alpha N$ elementų, čia N yra X masyvo elementų skaičius, o α – X masyvo šablono taškų skaičius. Kiti elementai laikomi nuliniais. Panašiai kaip PETSc pakete, siūloma išretintas matricas saugoti CSR formatu. Galimi nenuliniai elementai saugomi eilutėmis, kartu su atitinkamų stulpelių numerių masyvu, ir masyvu, kuriame saugomos nuorodos į kiekvienos eilutės pradžią, kaip parodyta 2.3 paveiksle. Jame mes matome matricos, kurios eilutėse yra $2, 3, 3, 4, 0$ nenulinių elementų, struktūrą.



2.3 pav. *Išretintų matricų saugojimas CSR formatu*

Išretintos matricos saugoja tik nenulinius elementus, taupant kompiuterio operatyvią atmintį. Tačiau išauga matricos elemento pasiekimo kaštai, taip pat naujo nenulinio elemento įterpimo kaštai[2]. Pilnos matricos yra neekonomiškos atminties atžvilgiu, tačiau turi maksimalų elemento pasiekimo greitį, ir ILU faktorizacijos rezultatas jose būna tikslesnis. Dažnai atminties taupymo faktorius yra svarbiausias.

Kaip jau buvo pastebėta, matricos, dauginamo ir rezultatinio vektoriaus struktūros yra griežtai susijusios. Galima teigti, kad, žinant dauginamojo ir rezultatinio vektoriaus struktūras, galima *vienareikšmiškai* apibrėžti matricos, dalyvaujančios tokioje sandaugoje, struktūrą. Tokiu atveju, norint sukurti matricą, reikia nurodyti dauginamąjį ir rezultatinį vektorius (kvadratinės matricos atveju užtenka vieno). Šis būdas yra žymiai patogesnis, kai vektoriaus charakteristikos apibrėžiamos daugiau nei vienu parametru (pavyzdžiui, šablonas, indeksų pradžia ir pabaiga pagal kiekvieną dimensiją). Matricos ir vektoriaus sandaugos metu matrica automatiškai suranda stulpelio, eilutės ir atitinkamų vektorių elementų atitikmenis (vienmačių vektorių atveju, kai pirmo elemento indeksas yra lygus 1, tai sutaptų su (2.4) formule).

[2]Gana tiksliai nustatyti, kokie matricos elementai turi būti nenuliniai, sprendžiant daugelį taikomųjų uždavinių, galima dar matricos kūrimo metu, ir iš anksto paskirti jiems vietą. Todėl elementų įterpimas būna reikalingas retai.

E. Technologijos panaudojimas nuosekliosios algoritmo versijos kūrimui

Apibendrinkime ankščiau aprašytų matematinių objektų savybes ir jų panaudojimo rekomendacijas į bendrą paveikslą. Taikant pasiūlytą technologiją nuosekliojo algoritmo realizacijai, reikia laikytis tokių žingsnių:

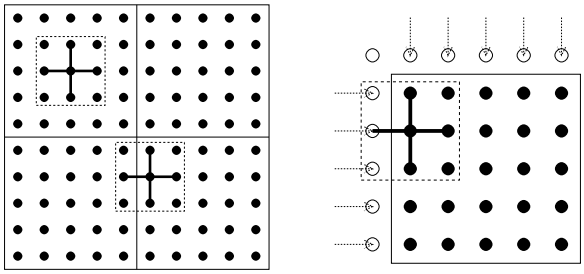
1. Atlikus algoritmo analizę, nustatyti algoritmo šabloną (arba šablonus);
2. Sukurti reikiamą šabloną (šablonus);
3. Saugojant duomenis, kurių apdorojimą vėliau norėsime išlygiagretinti, naudoti *tik* pasiūlytus masyvus arba vektorius. Masyvų sukūrimo metu nurodomas šablonas ir pilna masyvų konfigūracija;
4. Norint vėliau sėkmingai *išlygiagretinti* programą, su masyvais turi būti atliekamos tik lygiagrečiųjų duomenų algoritmams būdingos operacijos (operacijos, kurios naudoja tik kaimyninius kiekvieno masyvo elementus, ir kurioms neturi įtakos tvarka, kuria apdorojami masyvo elementai);
5. Atskirai reikėtų paminėti masyvų *dimensijų dydžius*. Kuriant masyvus, visada (kaip nuoseklios, taip ir lygiagrečios programos versijos atveju) turi būti nurodomi pilni dydžiai. Tačiau realiai paskirtų elementų skaičius ir adresavimo ribos gali priklausyti nuo įvairių faktorių: šablonų formos, nuosekliosios ar lygiagrečiosios programos versijos. Todėl, perrenkant visus masyvo elementus, reikia naudoti specialiai dimensijų dydžio nustatymui skirtus masyvų metodus;
6. Jeigu algoritmo realizacija reikalauja matricų-vektorių sandaugos, arba tiesinių lygčių sistemos sprendimo, matricoms realizuoti reikia naudoti pasiūlytas matricas. Jos gali būti naudojamos tik kartu su šios technologijos vektoriais arba masyvais;

F. Masyvų ir vektorių lygiagretinimas

Masyvų lygiagretinimo idėja parodyta 2.4 paveiksle. Masyvas dalinamas tarp procesų, priklausomai nuo topologijos. Priklausomai nuo pasirinkto šablono (stencil), kiekviena masyvo dalis turi buferinį sluoksnį (balti taškai), kur kiekvienas kaimynas įrašo atitinkamus savo elementus duomenų apsikeitimo metu. Tai panašu į ciklinio masyvo nuosekliąją realizaciją.

Masyvų padalinimas tarp procesų priklauso nuo pasirinktos topologijos. Pagal kiekvieną dimensiją, masyvas dalinamas į tiek vienodų dalių[3], kiek procesorių yra topologijos duotojoje dimensijoje. Pavyzdžiui, masyvas iš 200×300 taškų bus padalintas į 100×100 lokalių elementų dalis tarp procesų topologijoje 2×3.

[3] Apvalinant iki sveikųjų skaičių.



a) b)

***2.4* pav.** *Perėjimas nuo nuoseklaus prie lygiagretaus masyvo: a) nuoseklusis masyvas su šablonu, b) masyvo dalis, priklausanti vienam procesui*

Procesuose, kurie nėra įtraukti į topologiją (jei tokių yra), masyvų dydžiai bus 0, ir jokios operacijos nebus vykdomos.

Apsikeičiant duomenimis tarp procesų, svarbu yra ne tik teisingai nustatyti, kokią informaciją siųsti, bet ir informacijos siuntimo tvarka. Siūloma naujoje technologijoje realizuoti tokius duomenų apsikeitimo organizavimo tipus:

Visi kartu – visi procesai inicializuoja siuntimo ir gavimo pradžią visiems reikalingiems kaimynams vienu metu. Šis metodas, inicializuodamas visus siuntimus ir gavimus vienu metu, gali sukurti didesnį duomenų judėjimą, ko pasekoje gali sulėtėti duomenų apsikeitimas, ir programa gali pareikalauti daugiau resursų. Tačiau jis turi vieną privalumą – galima atskirti duomenų apsikeitimo pradžią ir pabaigą, ir tarp jų atlikti įvairius veiksmus.

Poromis – šiuo atveju, bet koks procesas vienu metu bendrauja tik su tam tikru vienu procesu. Tačiau abu procesai poroje vienu metu inicializuoja tiek siuntimą, tiek ir gavimą, ir laukia jų abiejų užbaigimo.

Poromis, tam tikra tvarka – šiuo atveju ne tik procesas vienu metu bendrauja tik su tam tikru kitu procesu, bet ir vienu metu vyksta tik viena operacija, siuntimas arba gavimas.

G. Pusiau automatinio lygiagretinimo principai

Šiame poskyryje aptariamos bendros nuosekliųjų programų lygiagretinimo taikant naują technologiją idėjos. Kaip jau buvo minėta, buvo siekiama, kad, žiūrint iš vartotojo pusės, lygiagretinimas būtų kuo paprastesnis ir reikalautų iš

vartotojo kuo mažiau specifinių lygiagretinimo žinių. Šiuo atžvilgiu puikus lygiagrečiųjų duomenų algoritmų lygiagretinimo pavyzdys yra HPF [27].

Norint išlygiagretinti algoritmą taikant HPF, reikia laikytis tam tikrų principų ir atlikti tokius žingsnius:

- Sukurti nuoseklią programos versiją, naudojant standartinį FORTRAN'ą. Nuosekli programos versija turi būti parašyta, laikantis tam tikrų taisyklių. Tam turi būti naudojami masyvo dalių išskyrimo operatoriai, taip pat specialūs operatoriai, kaip, pavyzdžiui, `FORALL`.
- Įterpti specialaus formato komentarus, kurie nurodo procesorių topologiją ir masyvų paskirstymą tarp procesorių.
- Perkompiliuoti programos kodą *su HPF kompiliatoriumi*.

Matome, kad, jeigu nuoseklioji programa atitinka tam tikrus kriterijus, jos lygiagretinimas yra paprastas ir pusiau automatinis. Algoritmų *lygiagretinimo* taikant naują technologiją žingsniai yra analogiški lygiagretinimui taikant HPF:

- Sukurti nuoseklią programos versiją, laikantis ankstesniuose poskyriuose aprašytų principų.
- Padaryti nedidelius programos pakeitimus, nurodančius procesorių topologiją ir masyvų paskirstymą tarp procesorių. Kadangi technologija turi būti realizuota bibliotekos, o ne kompiliatoriaus forma, negalima naudoti specialaus tipo komentarų, todėl pakeitimai turėtų būti kitokio pobūdžio, tačiau jų esmė yra ta pati.
- Galiausiai programa perkompiliuojama. Kompiliuojame tuo pačiu kompiliatoriumi, tačiau prijungiamos kitos bibliotekos.

2.3. Reikalavimai, keliami technologijos realizacijai

Realizuojant pasiūlytą technologiją, reikia siekti ne tik parodyti naujos koncepcijos veiksmingumą, bet ir sukurti produktą, kuriuo būtų galima naudotis, sprendžiant realius uždavinius. Todėl technologijos realizacija turi pasižymėti tokiomis savybėmis:

Patogumas. Tai reiškia, kad mažiau laiko sugaištama programavimui ir klaidų gaudymui, kodas yra aiškesnis, programa lengviau modifikuojama;

Efektyvumas. Bibliotekos naudojimas leidžia vartotojui kurti efektyvias programas, negaištant laiko standartinių veiksmų optimizavimui;

Pernešamumas. Realizacija turi būti be didelių pastangų perkeliama į įvairias platformas.

Šie tikslai lemia įrankių, naudojamų naujos technologijos realizacijai, pasirinkimą, taip pat kai kuriuos realizacijos ypatumus.

Kaip jau buvo pastebėta, technologiją geriau realizuoti bibliotekos pavidalu. Norint užtikrinti bibliotekos *pernešamumą*, turi būti naudojamos tik plačiai paplitusios programavimo priemonės ir standartai. Iš programavimo kalbų šį reikalavimą atitinka C ir C++ kalbos. MPI lygiagretinimo standartas irgi yra plačiai paplitęs įvairiose platformose. Taip pat, siekiant aptikti ankstyvojoje stadijoje bibliotekos klaidas, atsirandančias dėl aplinkai būdingų C/C++ ir MPI realizacijos ypatumų, kartu su biblioteka turi būti teikiami ir išsamūs bibliotekos funkcionalumo regresiniai tikrinimai.

MPI lygiagretinimo standartas pasirinktas taip pat todėl, kad siekiame, jog lygiagrečioji versija veiktų kompiuterių klasteriuose. Kaip žinoma, pranešimų siuntimo principu pagrįsti lygiagretinimo įrankiai leidžia pasiekti aukščiausio efektyvumo sistemose su paskirstyta atmintimi.

Patogumą sunku apibūdinti kiekybiškai. Vienas iš galimų būdų atsižvelgti į šį reikalavimą yra leisti vartotojui, naudojančiam šį įrankį, operuoti jam gerai pažįstamais matematiniais objektais (tokiais kaip vektoriai, matricos ir kiti), taip pat, kad bibliotekoje būtų realizuotos dažniausiai pasitaikančios vektorių ir matricų operacijos (pavyzdžiui, skaliarinės sandaugos, normų skaičiavimas). Tam tinka OOP stilius, kurį pateikia C++ kalba [75]. Taip pat C++ kalboje realizuotas šabloninis metaprogramavimas, kas leidžia su minimaliomis pastangomis realizuoti technologijos struktūras skirtingiems duomenų tipams. Todėl buvo nuspręsta technologiją realizuoti naudojant C++ kalbos. Beje, C++ pastaruoju metu vis dažniau naudojama skaitinių algoritmų realizacijai. Tai yra ir MFSolver įrankio, kurio lygiagretinimas yra vienas iš šios technologijos uždavinių, kalba.

C++ kalba pasižymi tuo, kad ji beveik pilnai suderinama su C, tai yra C programos gali būti kompiliuojamos C++ kompiliatoriumi praktiškai be pakeitimų arba su nežymiais pakeitimais. Iš vienos pusės, tai labai praplečia C++ kalbos galimybes. Tačiau, iš kitos pusės, toks „palikimas“ padidina kalbos sudėtingumą ir daro neigiamą įtaką programavimo stiliui. Tai patvirtina faktas, kad ne kartą buvo bandyta sukurti C++ kalbos dialektus, siekiant supaprastinti programavimą, pagerinti kodo aiškumą ir pan. Kaip pavyzdžius galima paminėti nesėkmingą EC++ projektą ir šiuo metu vystomą *D* programavimo kalbą.

Iš tikrųjų, dažnai tą patį tikslą C++ kalboje galima pasiekti skirtingais būdais, iš kurių tik keli yra numatyti kalbos kūrėjų, o kiti gali sulėtinti programą, tapti sunkiai surandamų klaidų šaltiniais ir apsunkinti programos palaikymą ateityje. Jeigu prie to pridėti faktą, kad C++ yra pakankamai sudėtinga kalba, kur viskas gali būti

žymiai sudėtingiau, negu atrodo iš pirmo žvilgsnio[4], tampa aišku, kodėl nemažai vartotojų arba pilnai nenaudoja C++ galimybių ir todėl nemato C++ privalumų lyginant su C, arba naudoja C++ kalbą ne optimaliu būdu, ko pasekoje rezultatas neatitinka kalbos galimybių. Tai ypač aktualu mokslinių skaičiavimų srityje, kur vartotojai dažnai nėra C++ kalbos profesionalai, o reikalavimai programos greitaveikai ir atminties sunaudojimui yra ypač aukšti.

Dėl neteisingo kalbos naudojimo dažnai rezultatas neatitinka kalbos galimybių. Tai ypač aktualu mokslinių skaičiavimų srityje, kur vartotojai dažnai nėra C++ kalbos profesionalai, o reikalavimai programos greitaveikai ir atminties naudojimui yra ypač aukšti. Viena iš galimybių įveikti šį sunkumą – specializuotų bibliotekų, skirtų skaitiniams algoritmams, kūrimas. Iš vienos pusės, naudojant C++ kalbos galimybes, šios bibliotekos galėtų pateikti vartotojui labai patogią ir intuityvią sąsają. Iš kitos pusės, jos galėtų efektyviai realizuoti pagrindines skaitines operacijas, leidžiant vartotojui susikoncentruoti ties savo algoritmo aukštesnio lygio realizacija. Kadangi kuriama biblioteka skirta skaičiavimo algoritmams realizuoti, natūralu yra siekti optimaliai realizuoti matematikos objektus tiek vartotojo sąsajos patogumo, tiek vykdymo efektyvumo prasme, pasinaudojant Blitz++ [13], Boost [14], MTL [78], ITL [76] bei kitų bibliotekų patirtimi.

2.4. Technologijos realizacija

Šiame poskyryje detaliai aprašyta naujos lygiagretinimo technologijos realizacija skaičiavimo matematikos objektų bibliotekoje, šios bibliotekos vartotojo sąsaja, vidinė struktūra ir panaudojimo detalės.

Technologija realizuota bibliotekoje *ParSol*, kuri yra *laisvai* prieinama internetu adresu

`http://techmat.vgtu.lt/~alexj/ParSol/` .

Biblioteka parašyta naudojant C++ kalba, lygiagretinimui naudojant MPI-1.1 standartą. Bibliotekos kodo dydis yra $\approx 20\,000$ eilučių, dar $\approx 10\,000$ eilučių užima bibliotekos regresiniai tikrinimai (naudojama *CppUnit* tikrinimų terpė). Naudojamos tik standartinės C++ kalbos ir MPI standarto priemonės, tai užtikrina bibliotekos pernešamumą į skirtingas platformas. Biblioteka buvo bandyta įvairiose ***nix** tipo operacinėse sistemose, naudojant įvairias `gcc` kompiliatoriaus versijas ir skirtingas MPI realizacijas (LAM/MPI, MPICH, MPI realizacija SP4 superkompiuteriuose). Taip pat biblioteka buvo bandoma MS-Windows terpėje, naudojant MPICH ir MS-VC++ kompiliatorių.

[4] Tai įvairių operatorių perkrovimai, pagal nutylėjimą veikiantys klasių konstruktoriai, automatiniai tipų konvertavimai ir kiti aspektai.

2.4.1. Vartotojo sąsajos lygmuo

Šiame poskyryje aptariami praktiniai bibliotekos naudojimo aspektai.

A. Bendri bibliotekos naudojimo aspektai

Norint panaudoti biblioteką, į C++ programą turi būti įjungti tam tikri bibliotekos antraštiniai failai (angl. *headers*). Nors biblioteką sudaro virš 30 įvairių modulių, susidedančių iš antraštinių failų ir kodinės dalies, realiai dauguma ParSol modulių automatiškai panaudoja vienas kitą, ir specialiai jas į vartotojo programą įtraukinėti (naudojant `#include` direktyvą) *nėra* būtina. Kuo mažiau įtraukiama antraštinių failų į programą, tuo mažiau teks daryti pakeitimų ją lygiagretinant.

ParSol antraštinių failų įdėjimo taisyklės yra tokios:

- Jeigu naudojami skirtingų dimensijų objektai, pavyzdžiui, 1D, 2D ir panašūs masyvai, reikia įjungti tik atitinkamos dimensijos antraštinį failą. Tokie failai turi vienodus pavadinimus, skiriasi tik pavadinimo pabaiga `_?D`, kur `?` yra atitinkama dimensija (1, 2, 3, ...). Jei programa naudoja skirtingų dimensijų objektus, *visi* reikiamų dimensijų antraštiniai failai turi būti įjungti. Toliau bus naudojamas failų pavadinimų žymėjimas su klaustuku, pavyzdžiui, `PS_CommonVector_?D.h`, kuris reiškia, kad reikia rinktis antraštinius failus pagal situaciją.
- Kadangi ParSol bibliotekoje lygiagretieji objektai yra nuosekliųjų objektų palikuonys, antraštiniai failai, reikalingi lygiagrečiajai versijai, jau įtraukia ir nuosekliuosius failus, todėl papildomai atitinkamo antraštinio failo įtraukinėti nebūtina. ParSol bibliotekoje lygiagrečiosios versijos antraštiniai failai turi pavadinimus, kurie skiriasi nuo nuosekliosios versijos tuo, kad žodis `Common` keičiamas į `Parallel`. Pavyzdžiui, jei nuosekliojoje programos versijoje buvo įtrauktas failas `PS_CommonVector_?D.h`, tai lygiagrečiojoje versijoje bus reikalingas failas `PS_ParallelVector_?D.h`.

ParSol biblioteka plačiai naudoja vardų sritis (angl. *namespace*). Pagrindinė ParSol bibliotekos vardų sritis šiuo metu vadinasi `ParSol`. Tačiau nėra garantijos, kad kitoje versijoje šis pavadinimas nepasikeis, tarkime, į `ParSol_v_1_3`. ParSol bibliotekoje yra apibrėžtas vardas `NS_PARSOL`, kuris yra *įdiegtos* ParSol bibliotekos pagrindinės vardų srities pavadinimas. Todėl, norint pasinaudoti bibliotekos galimybėmis, visus bibliotekos elementus reikia iškvietinėti arba su pradžia `NS_PARSOL::`, arba kiekviename kompiliuojamame modulyje turi būti eilutė

```
1 using namespace NS_PARSOL;
```

B. Šablonų (stencil) panaudojimas

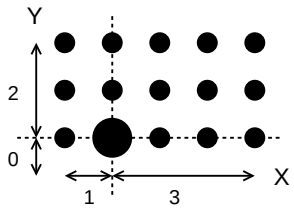
Kaip jau buvo minėta, šablonai vaidina svarbų vaidmenį ParSol bibliotekoje. Vartotojas turi mokėti parinkti tinkamą šabloną savo algoritmams. ParSol biblioteka pateikia vartotojui skirtingas šablonų klases, kurias galima naudoti priklausomai nuo poreikių. Šiame poskyryje aprašome šablonų klases.

Šablonai, kaip ir ParSol masyvai, gali turėti skirtingas dimensijas. Šablono dimensija *turi* sutapti su masyvo arba vektoriaus, kurio sukūrimui šis šablonas bus panaudotas, dimensija. Kadangi šablonai turi įtakos vidinei ParSol masyvų, vektorių ir matricų struktūrai, jie yra *būtini* šių objektų konstruktorių argumentai. Šablonai perduodami ir saugojami masyvuose per nuorodą (angl. *by reference*), todėl svarbu, kad jie būtų sukurti prieš inicializuojant masyvus, ir egzistuotų visą masyvų gyvenimo trukmę. Tačiau labai dažnai, žinant realizuojamą algoritmą, užtenka vieno šablono, kuris paskui tiesiogiai arba ne (per kopijavimo konstruktorių) gali būti naudojamas visų kitų ParSol objektų sukūrimui.

To pačio tipo šablono klasių pavadinimai skirtingoms dimensijoms yra panašūs, skiriasi tik pabaiga `_?D`, čia `?` yra dimensijų skaičius. Vartotojui *nereikia* nurodinėti jokių standartinių ParSol bibliotekos *šablonų antraštinių failų*, nes šitie failai jau būna įtraukti į standartinius ParSol masyvų antraštinius failus.

Toliau pateikiame šiuo metu ParSol bibliotekoje realizuotų šablonų klasių aprašymus.

B.1. Paprastas šablonas (dėžutė) gauna parametrus, per kiek taškų pagal kiekvieną dimensiją pirmyn ir atgal gali nutolti šablono taškai, ir įtraukia visus taškus, nutolusius ne daugiau negu šis atstumas. Tokio tipo šablonui reikia daugiausiai $2N$ parametrų, čia N yra dimensijų skaičius. 2.5 paveiksle parodytas dvimačio šablono pavyzdys. Pavaizduotas šablonas gali būti apibūdintas kaip šablonas, susidedantis iš *visų* taškų, nutolusių nuo centro ne daugiau, kaip 3 taškais į priekį **X** kryptimi, 1 tašku atgal **X** kryptimi, 2 taškais į priekį **Y** kryptimi ir nei vienu tašku (0) atgal **Y** kryptimi.



2.5 pav. *Paprasto dvimačio šablono schema*

Parametrus šablonui galima nurodyti dviem būdais: per konstruktorių sukūrimo metu ir po sukūrimo. Paprasto šablono klasės pavadinimas yra `SimpleStencil_?D`, kur vietoj klaustuko pateikiamas dimensijų skaičius. Visų dimensijų paprasti šablonai turi bendrus konstruktorius pagal nutylėjimą (angl. *default constructor*), ir konstruktorius su vienu parametru, pavyzdžiui:

- `SimpleStencil_?D St` – konstruktorius pagal nutylėjimą, sukuria šabloną su visais *nuliniais* parametrais. Naudingas, jei kiekvieną šablono parametrą norima užduoti vėliau.
- `SimpleStencil_?D St(x)` – konstruktorius su vienu parametru, sukuria simetrinį šabloną, kur visomis kryptimis maksimalus atstumas yra x (sveikas skaičius).

Taip pat šablonų klasės turi konstruktorius, reikalaujančius iškart $2N$ parametrų, pradžioje minimumus pagal **X**, **Y**, **Z** dimensiją (priklausomai nuo konkretaus dimensijų skaičiaus), paskui maksimumus tokia pat tvarka. Pavyzdžiui, šabloną, pavaizduotą 2.5 paveiksle, būtų galima apibrėžti taikant tokį konstruktorių:

Paprasto šablono bendras konstruktorius

```
1 SimpleStencil_2D St(1,0,3,2);
```

Toks kodas gali tapti sunkiai skaitomu, ypač didėjant dimensijų skaičiui. Todėl šablono parametrus galima nurodinėti ir *išreikštiniu* būdu. Tam egzistuoja tokie metodai:

- `Min?()` – nustatyti šablono tam tikros dimensijos (vietoj klaustuko gali būti **X**, **Y**, **Z**, priklausomai nuo to, kiek dimensijų turi šablonas) minimumą;
- `Max?()` – nustatyti šablono tam tikros dimensijos maksimumą.

2.5 paveikslo pavyzdys galėtų būti apibrėžtas taip:

Paprasto šablono parametrų keitimas

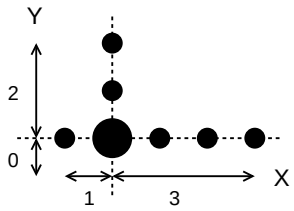
```
1 SimpleStencil_2D St; // 0 visomis kryptimis
2 St.MinY() = 0;
3 St.MinX() = 1;
4 St.MaxY() = 2;
5 St.MaxX() = 3;
```

Tuos pačius metodus galima naudoti ir šablono parametrų *nuskaitymui*, tačiau tik tuo atveju, jei nagrinėjamas šablonas nėra konstanta.

Taip pat matome, kad šiuo būdu šablono parametrai gali būti keičiami ir po šablono sukūrimo. Griežtai rekomenduojama šiuos parametrus keisti *iškart* po šablono sukūrimo, *prieš* pradedant naudoti juos masyvų apibrėžimui, ir *tik* tada.

Šablono parametrų keitimas gali lemti sunkiai atsekamas klaidas. Jeigu reikalingas šablonas su kitais parametrais, reikia kurti *naują* šabloną.

B.2. Kryžminis šablonas (angl. *cross stencil*) naudoja tuos pačius parametrus, kaip ir paprastas šablonas, tik šiuo atveju šablonui priklauso visi taškai, kurie yra ne tik nutolę nuo centro ne didesniu atstumu nei galima (toks yra paprasto šablono apibrėžimas), bet ir yra nutolę tik pagal kokią nors vieną iš dimensijų. 2.6 paveiksle parodytas kryžminis šablonas, sukurtas su tokiais pat parametrais, kaip ir paprastas šablonas, pavaizduotas 2.5 paveiksle.



2.6 pav. *Kryžminio dvimačio šablono schema*

Kadangi naudojami tie patys parametrai, kryžminio šablono sukūrimas yra *toks pat*, ir jam galioja *tos pačios* taisyklės, kaip ir paprastam šablonui, skiriasi tik klasės pavadinimai – vietoj `SimpleStencil_?D` reikia naudoti `CrossStencil_?D` šabloną. Pavyzdžiui, 2.2 paveiksle parodytas šablonas yra simetrinis, jis galėtų būti sukurtas taip:

Simetrinio kryžminio šablono konstruktorius

```
1 CrossStencil_2D St(1);
```

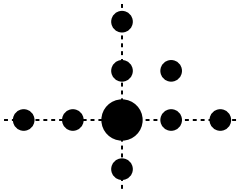
B.3. Universalus šablonas gali būti panaudotas *bet kokiam* šablonui apibrėžti. Tačiau jis yra lėtesnis, naudoja daugiau atminties ir reikalauja sudėtingesnio programavimo. Rekomenduojama jį naudoti tik tada, kai reikiamą šabloną negalima išreikšti kitais bibliotekoje egzistuojančiais šablonais.

Šis šablonas apibrėžiamas, tiesiogiai nurodant kiekvieną šablono tašką po šablono sukūrimo. Kaip ir kitiems šablonams, rekomenduojama visus šablono taškus nurodyti *iškart* po šablono sukūrimo. Universalaus šablono klasės pavadinimas priklauso nuo dimensijos ir yra `UniversalStencil_?D`. Kadangi iš anksto nežinoma, kiek šablone gali būti taškų, konstruktorius yra be parametrų.

Sukūrus šabloną, taškai pridedami taikant `AddPoint(PosX,PosY,PosZ)` metodą. Argumentai `Pos?` yra sveikieji skaičiai, rodantys taško nuotolį nuo šablono

centro pagal atitinkamą dimensiją. Jų skaičius priklauso nuo konkrečios klasės dimensijų skaičiaus. Toks pat taškas gali būti pridėtas keletą kartų, efektas yra toks pat, kaip pridėjus tašką tik vieną kartą. Iškart kelis taškus galima pridėti taikant metodą `AddCross(WL)`. Šio metodo iškvietimas iškart po universalaus šablono sukūrimo yra panašus į kryžminio simetrinio šablono su sparno ilgiu `WL`, sukūrimą. Skirtumas nuo kryžminio šablono yra tas, kad pridedami tik *kraštiniai* taškai.

Neteisingai pridėtą tašką galima panaikinti taikant `RemovePoint(PosX,PosY,PosZ)` metodą. Šio metodo argumentai yra tokie pat kaip ir metodo `AddPoint`. Tas pats taškas gali būti panaikintas kelis kartus – tai nėra klaida. Visus taškus iškart galima panaikinti taikant metodą `Clear()`.



2.7 pav. *Universalaus šablono pavyzdys*

Universalaus šablono pavyzdys parodytas 2.7 paveiksle. Toks šablonas gali būti sukurtas naudojant žemiau pateiktą kodą[5]:

Universalaus šablono kūrimas

```
1 UniversalStencil_2D St; // kuriama šablono klasė
2 St.AddPoint( 0, 0 ); // centrinis taškas
3 St.AddCross( 1 ); // pridedami keturi taškai
4 St.AddCross( 2 ); // dar keturi taškai toliau
5 St.RemovePoint( 0, -2 ); // vieno taško iš 4 nereikia
6 St.AddPoint( 1, 1 );
```

B.4. Bendras šablonų funkcionalumas. Dažniausiai vartotojui užtenka teisingai sukurti šabloną ir jį naudoti kaip parametrą masyvų ir vektorių iniciacijai. Šiame skyrelyje aptariamas šablonų funkcionalumas ir jo naudojimo galimybės.

Tikrinimo tikslais gali būti reikalinga išvesti informaciją apie naudojamą šabloną. Tai gali būti ypač aktualu tikrinant lygiagrečias programų versijas, kai derinimo programos (angl. *debugger*) naudojimas ne visada yra galimas. Tai gali būti padaryta taikant metodą `DumpConfig(std::ostream& os)`. Pavyzdžiui, *visos* informacijos apie šabloną išvedimas į terminalą galėtų būti atliktas taip:

[5] Šis variantas nėra vienintelis, galimi ir kiti.

Šablono informacijos išvedimas

```
1 St.DumpConfig( std::cout );
```

Vien šablono tipą galima sužinoti taikant metodą `char* StencilName()`.

Taip pat tarp šablonų yra apibrėžtos tokios palyginimo operacijos:

== šablonų lygybė. Du šablonai yra lygūs, jeigu jie aprėpia tokį patį *plotą*. Lygių šablonų pavyzdžiai yra 2.8 paveiksle pateikti šablonai a) ir b). Kaip galime matyti, tas pats užimamas plotas (pažymėtas pilkai) gali būti pasiektas skirtingais taškais;

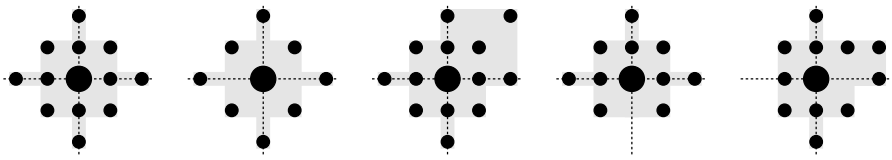
!= Šablonai yra nevienodi, jeigu jie nėra lygūs pagal ankstesnį apibrėžimą. 2.8 paveiksle šablonas a) nelygus nei šablonui c), nei d), nei e);

<= šablonas X yra mažesnis arba lygus šablonui Y, jei šablono X užimamas plotas pilnai telpa šablono Y plote. 2.8 paveiksle šablonas a) yra mažesnis arba lygus šablonui b) ir c);

< šablonas X yra mažesnis už šabloną Y, jei šablono X užimamas plotas pilnai telpa šablono Y plote, ir jų plotai nėra tokie pat. 2.8 paveiksle šablonas a) yra mažesnis už šabloną c). Taip pat kryžminis šablonas visada yra mažesnis už paprastą šabloną su tokiais pat parametrais;

>= šablonas X yra didesnis arba lygus šablonui Y, jei šablono X užimamas plotas pilnai aprėpia šablono Y plotą. 2.8 paveiksle šablonas a) yra didesnis arba lygus šablonui b) ir d);

> šablonas X yra didesnis už šabloną Y, jei šablono X užimamas plotas pilnai aprėpia šablono Y plotą, ir jų plotai skirtingi. 2.8 paveiksle šablonas a) yra didesnis už šabloną d).



2.8 pav. *Šablonų palyginimas*

Reikėtų pastebėti, kad, jeigu šablonas X nėra didesnis už šabloną Y, tai dar nereiškia, kad X yra mažesnis arba lygus Y. Ir panašiai, jei X nėra mažesnis už Y, tai dar nereiškia, kad $X \geq Y$. Toks pavyzdys yra šablonai a) ir e) 2.8 paveiksle. Apie juos galima pasakyti tik tiek, kad jie nėra lygūs.

C. Masyvų panaudojimas

ParSol bibliotekoje masyvų klasės turi pavadinimus `CmArray_?D<ElemType>`, kur `?` yra dimensijų skaičius, o `ElemType` yra šabloninis (angl. *template*) parametras, kuris nurodo masyvo elemento duomenų tipą. Galima apibrėžti ParSol masyvą *bet kokiam* vidiniam C/C++ arba programoje apibrėžtam duomenų tipui, kuriam egzistuoja priskyrimo (angl. *assignment*) ir palyginimo (`==` ir `!=`) operatoriai.

Jei programoje naudojami *tik* ParSol nuoseklieji masyvai, reikia įtraukti į programą antraštinį failą(-us) `PS_CommonArray_?D.h`, kur `?` irgi yra dimensijų skaičius, priklausomai nuo to, kokios dimensijos masyvai bus naudojami. Šis failas bus automatiškai įtrauktas ir bus galima naudoti nuosekliuosius masyvus, jei į programa įtraukiami atitinkamos dimensijos vektorių, arba lygiagrečiųjų masyvų ir vektorių antraštiniai failai.

C.1. Elementų adresacija

ParSol masyvų elementai adresuojami indeksų rinkiniu, kur kiekvienas indeksas nurodo elemento padėtį tam tikroje dimensijoje. ParSol masyvuose paskutinis indeksas (ir dimensija) turi pavadinimą `X`, priešpaskutinis – `Y`, toliau – `Z`. Palyginus su standartiniais C/C++ masyvais, elemento pasiekimui vietoj laužtinių skliaustų operatoriaus, pavyzdžiui, `a[i]` arba `a[i][j]`, naudojami paprasti skliaustai, pavyzdžiui, `a(i)` arba `a(i,j)`.

Visų diapazonų pirmus indeksus galima sužinoti taikant funkcijas, prasidedančias žodžiu `Start`. `StartGlobal(int Dim)` leidžia nustatyti globalaus diapazono pradžią pagal tam tikrą dimensiją `Dim`, `StartReal(int Dim)` – analogiška `StartGlobal`, tik realiam diapazonui, o `StartLocal(int Dim)` – lokaliam. Egzistuoja ir specializuoti šių funkcijų variantai, pavyzdžiui, `StartLocalX()` – konkrečiai **X** dimensijai (yra analogiškos funkcijos ir kitiems diapazonų tipams ir dimensijoms). Panašiai į diapazonų pradžias, egzistuoja visas spektras funkcijų diapazonų pabaigai ir ilgiui nustatyti. Jų pavadinimai sudaromi taip pat, kaip ir diapazonų pradžios funkcijų, tik vietoj `Start` naudojami `Finish` (diapazono paskutiniam indeksui nustatyti) ir `Size` (diapazono ilgis).

C.2. Masyvų sukūrimas

Norint sukurti ParSol masyvą, galima pasinaudoti įvairiais konstruktoriais, priklausomai nuo užduoties sudėtingumo. Visų masyvų konstruktorių (išskyrus kopijavimo konstruktoriaus) *pirmas argumentas* yra šablonas. Kiti argumentai lemia masyvo konfigūraciją. Pats *paprasčiausias* konstruktorius leidžia nurodyti tik masyvo dydžius pagal kiekvieną dimensiją:

Skirtingų dimensijų masyvų sukūrimas naudojant tik masyvų dydžius

```
1 CmArray_1D<double> Arr( St, Xsize );
2 CmArray_2D<double> Arr( St, Ysize, Xsize );
3 CmArray_3D<double> Arr( St, Zsize, Ysize, Xsize );
```

Taip sukuriamas *neperiodinis* masyvas, skirtas darbui su elementais, kurių adresai pagal kiekvieną dimensiją yra nuo `0` iki `{X|Y|Z}size-1` (globalus diapazonas). Nuosekliajam masyvui tai bus ir tiesioginė nuoroda, kokiais elementais algoritmas gali operuoti, todėl realus ir lokalus diapazonai sutaps su globaliu.

Norint sukurti masyvą, kurio elementų adresacija prasideda ne nuo `0`, reikia naudoti tokius konstruktorius:

Masyvų su pasirenkamu pradiniu indeksu kūrimas

```
1 CmArray_1D<int> Arr(St,Xstart,Xsize);
2 CmArray_2D<int> Arr(St,Ystart,Ysize,Xstart,Xsize);
3 CmArray_3D<int> Arr(St,Zstart,Zsize,Ystart,Ysize,Xstart,Xsize);
```

Tokiu būdu sukurto masyvo elementų adresai yra diapazone

nuo `{X|Y|Z}start` iki `{X|Y|Z}start + {X|Y|Z}size - 1`.

Visais kitais aspektais tokie masyvai niekuo nesiskiria nuo masyvų, kurių elementų adresacija prasideda nuo `0`.

C.3. Periodiniai masyvai

ParSol masyvuose periodiškumo sąlygą galima užduoti masyvo sukūrimo metu pagal kiekvieną dimensiją atskirai. Tam naudojami konstruktoriai, panašūs į ankščiau aprašytus. Pavyzdžiui, norint sukurti ciklinius masyvus, kurių elementų adresai prasideda nuo `0`, reikia naudoti konstruktorius:

Periodinių masyvų su nuliniu pradiniu indeksu kūrimas

```
1 CmArray_1D<double> Arr(St,Xsize,Xcyclic);
2 CmArray_2D<double> Arr(St,Ysize,Ycyclic,Xsize,Xcyclic);
```

o norint turėti laisvai pasirenkamą adresacijos pradžią, konstruktorius

Periodinių masyvų su nenuliniu pradiniu indeksu kūrimas

```
1 CmArray_1D<double> Arr(St,Xstart,Xsize,Xcyclic);
2 CmArray_2D<double> Arr(St,Ystart,Ysize,Ycyclic,Xstart,Xsize,Xcyclic);
```

Ciklišumą nurodantis argumentas yra `bool` tipo, jo reikšmė yra `true`, jei masyvas ciklinis, kitaip `false`.

C.4. Masyvo elementų saugojimas

ParSol bibliotekoje masyvų elementai saugomi tokia pat tvarka, kaip ir standartiniuose C/C++ masyvuose, t.y. naudojama eilučių pirmenybė. Ši informacija gali būti svarbi, optimaliai realizuojant ciklus per visus masyvo elementus. Gretimų atmintyje masyvo elementų indeksuose labiausiai keičiasi paskutinė (`X`) koordinatė, paskui priešpaskutinė (`Y`), paskui dar ankstesnė (`Z`) ir taip toliau.

Atliekant tam tikras operacijas su masyvo elementais, kartais reikalingas tiesioginis priėjimas prie ParSol masyvo elementų buferio. Tai daroma taikant funkciją `Buffer()`. Ši funkcija grąžina rodyklę (angl. *pointer*) į pirmą *visų* ParSol masyvo elementų vidinį vienmatį masyvą. Tai yra standartinis C/C++ masyvas, kurio elementai turi adresus nuo `0` iki `ElemCountReal()-1`. Pavyzdžiui, pakelti visus ParSol masyvo elementus kvadratu, nepriklausomai nuo masyvo dimensijų skaičiaus, būtų galima naudojant tokį kodą:

Visų ParSol masyvo elementų pakelimas kvadratu

```
1  CmArray_2D<double> Arr(...);
2  // programa ...
3  double *start = Arr.Buffer();
4  double *end = start + Arr.ElemCountReal();
5  for( double *e = start; e < end; ++e )
6    *e = (*e) * (*e);
```

Kadangi šis kodas nenaudoja jokių konstantų, jis yra automatiškai lygiagretinamas. Reikia atkreipti dėmesį, kad daugeliui tokio tipo operacijų ParSol jau turi patogias priemones, kuriomis rekomenduojama naudotis.

Jeigu norima atlikti operacijas tik su lokaliais masyvo elementais, tada nustatyti, ar elementas yra lokalus, ar ne, galima taikant funkciją `IsLocal(int i)`. Jos argumentas yra elemento padėtis vidiniame masyve, o rezultatas – loginė reikšmė, kuri yra `true`, jei elementas lokalus. Pavyzdžiui, norint pakelti kvadratu tik lokalius masyvo elementus, galima panaudoti tokį kodą:

Tik lokalių ParSol masyvo elementų pakelimas kvadratu

```
1  CmArray_2D<double> Arr(...);
2  // programa ...
3  double buf[] = Arr.Buffer();
4  for( int i = Arr.ElemCountReal()-1; i >= 0; --i )
5    if ( Arr.IsLocal(i) ) buf[i] = buf[i] * buf[i];
```

Taip pat tiesioginis priėjimas prie buferio gali būti panaudotas operacijose tarp masyvų. ParSol masyvai turi metodą `SameConfig(Arr)`, kuris patikrina, ar dviejų ParSol masyvų konfigūracijos sutampa. Jei taip, tai tokių masyvų buferiai yra vienodo ilgio, ir elementai juose yra išsidėstę tokia pat tvarka. Tada galima pato-

giai atlikinėti įvairias operacijas su atitinkamais masyvų elementais. Pavyzdžiui, dviejų masyvų atitinkamų elementų geometrinių vidurkių masyvas galėtų būti užpildytas naudojant tokį kodą:

Dviejų vektorių atitinkamų elementų geometrinių vidurkių skaičiavimas

```
1 CmArray_2D a(...), b(a), c(a);
2 // programos kodas ...
3 assert( a.SameConfig(b) );
4 assert( a.SameConfig(c) );
5 for( int i = a.ElemCountReal()-1; a >= 0; --a )
6   a.Buffer()[i] = sqrt( b.Buffer()[i] * c.Buffer()[i] );
```

C.5. Šablonų valdymas

Kuriant masyvus, reikia pasirinkti didžiausią įmanomą šabloną (angl. *stencil*), siekiant, kad būtų paskirti visi reikalingi papildomi elementai. Tačiau skaičiavimo metu tam tikruose etapuose algoritmas gali sutilpti ir į mažesnį šabloną. Mažesnis šablonas reiškia mažesnį duomenų apsikeitimą ir greitesnę programą.

ParSol masyvai palaiko laikiną šablono pakeitimą. Tai atliekama taikant funkciją `SetTemporaryStencil(St)`. Šios funkcijos argumentas yra naujas, laikinas šablonas, kuris privalo būti mažesnis arba lygus pradiniam masyvo šablonui. Norint grįžti prie pradinio šablono, reikia iškviesti metodą `ResetStencilToDefault()`. Sužinoti, koks šiuo metu naudojamas šablonas, galima taikant funkciją `CurrentStencil()`, o funkcija `TemporaryStencilNow()` nurodo, ar naudojamas šablonas yra laikinas. Funkcija `StencilIsCurrent(St)` nurodo, ar šiuo metu naudojamas šablonas `St`, ar koks kitas.

C.6. Pagrindinės operacijos su ParSol masyvais

ParSol masyvai palaiko visą eilę operacijų su jų elementais. Pirmiausiai, tai yra priskyrimo (angl. *assignment*) operatorius. ParSol masyvai palaiko tiek kito masyvo, tiek ir vieno elemento priskyrimą. Priskyrus masyvui elemento reikšmę, ši reikšmė suteikiama visiems masyvo elementams. Masyvo priskyrimas atlieka vieno masyvo elementų priskyrimą atitinkamiems kito masyvo elementams (prieš vykdant veiksmą patikrinama ar sutampa masyvų konfigūracijos). Visi priskyrimo operatoriai grąžina nuorodą į masyvą, kuriam priskiriama. Taigi, įvykdžius kodą

Visų masyvų elementų priskirimas

```
1 CmArray_2D<double> A(...), B(A), C(A);
2 A = B = C = 5.0;
```

visų trijų masyvų visi elementai bus lygūs `5.0`. Taip pat galima paminėti dvie-

jų masyvų palyginimo operatorių `==`, kuris grąžina `true` tuo atveju, jei masyvų konfigūracija vienoda, ir atitinkami elementai yra lygūs.

Iš pagalbinių masyvų funkcijų galima paminėti `DumpElems(os,inner)` (išveda visų masyvo elementų reikšmes) ir `DumpConfig(os)` (išveda masyvo konfigūraciją) metodus. Abiejuose metoduose argumentas `os` yra tipo `std::ostream` ir nurodo, kur turi būti išvedama informacija. Jeigu argumentas `inner` yra `true` (taip yra pagal nutylėjimą, jei šis argumentas nenurodytas), tai bus išvedami tik lokalūs elementai, kitaip – visi.

D. Vektorių panaudojimas

ParSol bibliotekoje vektoriai yra masyvų palikuonys. Tai reiškia, kad, jeigu įmanoma sukurti tam tikro duomenų tipo elementų vektorių, su vektoriais gali būti atliekami tokie pat veiksmai, kaip ir su masyvais, ir vektoriai gali būti naudojami visur, kur ir masyvai.

ParSol bibliotekoje vektorių klasių pavadinimai yra `CmVector_?D<ElemType>`, čia `?` yra dimensijų skaičius, o `ElemType` yra šabloninis (angl. *template*) parametras, kuris nurodo vektoriaus elementų duomenų tipą. Norint naudoti vektorius, reikia įtraukti į programą antraštinį failą(-us) `PS_CommonVector_?D.h`, kur `?` irgi yra dimensijų skaičius, priklausomai nuo to, kokios dimensijos vektoriai bus naudojami. Įtraukus šį failą, automatiškai bus įtrauktas ir ParSol *masyvų* antraštinis failas.

Be ParSol masyvų funkcionalumo, vektoriai pateikia vartotojui ir visą eilę realizuotų matematinių operacijų. Šios operacijos ne tik gali palengvinti programavimą, bet ir turi lygiagrečias versijas, todėl programa, parašyta naudojant šias operacijas, gali būti pusiau automatiškai išlygiagretinta. Toliau pateikiamas tokių operacijų sąrašas.

Sum() – grąžina vektoriaus elementų sumą: `a.Sum()`=$\sum_i a_i$;

InnerProduct(vec) – grąžina dviejų vektorių skaliarinės sandaugos rezultatą: `a.InnerProduct(b)`=$\sum_i a_i b_i$;

NormL1() – grąžina vektoriaus L_1 normą: `a.NormL1()`=$\sum_i |a_i|$;

NormL2() – grąžina vektoriaus L_2 normą: `a.NormL2()`=$\sqrt{\sum_i |a_i|^2}$;

NormL2sqr() – grąžina vektoriaus L_2 normos kvadratą: `a.NormL2sqr()`=$\sum_i |a_i|^2$;

NormLmax() – grąžiną vektoriaus L_∞ normą: `a.NormLmax()`=$\max_i |a_i|$;

Min() – grąžina vektoriaus mažiausią elementą: `a.Min()`=$\min_i a_i$;

Max() – grąžina vektoriaus didžiausią elementą: `a.Max()`=$\max_i a_i$;

Aukščiau išvardintos funkcijos savo skaičiavimams naudoja tik tuos vektorių elementus, su kuriais galima atlikinėti *visus* veiksmus. Nuosekliesiems vektoriams tai – elementai iš *lokalaus diapazono*, lygiagretiems vektoriams – *visų* tarp procesų paskirstytų dalių lokalūs elementai. Jeigu mes norime atlikti veiksmus tik su šiam procesui priklausančiais elementais, nesvarbu, ar tai yra nuoseklus, ar lygiagretusis kodas, reikia naudoti lokalius šių funkcijų analogus. Jie turi tokius pat pavadinimus, prie kurių prisideda dalis `_Local`. Pavyzdžiui, norint paskaičiuoti tik šiam procesui priklausančių lokalių elementų sumą, reikia naudoti funkciją `Sum_Local()`.

ParSol vektoriai gali dalyvauti išraiškose, susidedančiose iš matematinių veiksmų `+`, `-`, `*` ir `/`. Taip pat, be prilyginimo operatoriaus, realizuoti tokie veiksmai, kaip `+=`, `-=`, `*=` ir `/=`. Išraiškose gali būti maišomi tiek vektoriai, tiek ir skaliarai, todėl išraiška `C=(A+B)/2` yra leistina. Pagal poreikį, gali būti realizuotos ir kitos operacijos. Visos operacijos realizuotos taikant išraiškų šablonus (angl. *expression templates*) [92]. Tai leidžia sugeneruoti labai efektyvų kodą, pavyzdžiui, kodas

Globalios operacijos su ParSol masyvais, naudojant IŠ

```
1  CmArray_2D<double> a(...), b(a), c(a), d(a);
2  ...
3  a = b * c - d;
```

yra ekvivalentus kodui

Globalios operacijos su ParSol masyvais, naudojant standartinius C/C++ metodus

```
1  CmArray_2D<double> a(...), b(a), c(a), d(a);
2  ...
3  if ((!a.SameConfig(b)) || (!a.SameConfig(c)) || ...)
4    throw "Arrays do not match !!!";
5  for( int i = a.ElemCountReal()-1; i>=0; --i )
6    a.Buffer()[i] =
7      b.Buffer()[i] * c.Buffer()[i] - d.Buffer()[i];
```

Dėl ankščiau minėto papildomo vektorių funkcionalumo, duomenų tipai, tinkantys vektorių sukūrimui, papildomai privalo palaikyti šias operacijas: `+`, `-`, `*`, `/`, `std::abs()`, `<` ir `>`. Jei duomenų tipas tokių operatorių nepalaiko, jas galima realizuoti patiems.

E. Bendroji elementų adresacija

Kaip jau buvo minėta ankščiau, ParSol masyvų ir vektorių elementai pasiekiami panaudojant skliaustų operatorių `a(i)`, `a(i,j)`, `a(i,j,k)` ir taip toliau, priklausomai nuo masyvų dimensijos. Panašiai gali būti pasiekiami ir matricų ele-

mentai. Programuojant konkretų uždavinį, toks būdas yra paprastas, akivaizdus ir patogus, tačiau taip realizuotas algoritmas gali būti pritaikytas tik tam tikros dimensijos uždaviniui spręsti.

Kaip bus parodyta 2.4.3 poskyryje, pagrindinis ParSol klasių kodas yra bendras, nepriklausomai nuo dimensijų skaičiaus. Vartotojas irgi gali pasinaudoti šiuo bendrumu. ParSol masyvai, vektoriai ir matricos palaiko taip vadinamuosius *daugiadimensinius adresus*. Paprasčiausiu atveju daugiadimensinis adresas yra C/C++ masyvas, kuriame turi būti tiek elementų, koks yra dimensijų skaičius. Kaip daugiadimensinius adresus galima naudoti ir specialias klases, apie kurias detaliau bus parašyta 2.4.1 poskyryje. Naudojant daugiadimensinius adresus, galima pasiekti, kad parašytas kodas arba jo dalys atrodys identiškai nepriklausomai nuo dimensijų skaičiaus. Tačiau programavimas naudojant daugiadimensinius adresus yra sudėtingesnis, kodas tampa sunkiau įskaitomu.

Rekomenduojama naudoti įprastinę elementų adresaciją, kai realizuojamas konkretus algoritmas konkrečiam dimensijų skaičiui, ir svarbus yra algoritmo realizacijos greitis ir kodo aiškumas. Kai rašomas kodas, kuris vėliau gali būti panaudotas realizuoti įvairiems uždaviniams, kurių dimensija nėra iš anksto žinoma (pavyzdžiui, kai rašoma biblioteka), vartotojas gali pasirinkti sudėtingesnį, tačiau universalesnį programavimo stilių.

F. Matricų panaudojimas

Šiame poskyryje aprašomi veiksmai su ParSol matricomis.

F.1. Matricų tipai ir iniciacija

Viena iš pagrindinių matricos operacijų – matricos ir vektoriaus sandaugos $Y = MX$ skaičiavimas. Žinant X ir Y vektorių struktūrą, galima vienareikšmiškai nustatyti ir matricos, tinkančios sandaugai, formą. Šiuo faktu pasinaudota ParSol bibliotekoje. Bendriausiu atveju, matrica gali būti sukurta tokiu būdu:

Bendras matricos sukūrimo kodas

```
1 CmArray_2D<double> Y( St, 30, 31 );
2 CmArray_2D<double> X( St, 30, 31 );
3 CmMatrix<double,2,2> M(Y,X);
```

Čia `CmMatrix` yra ParSol matricų klasės pavadinimas, pirmas šabloninis parametras nurodo matricos elemento duomenų tipą, antras ir trečias – atitinkamai `Y` ir `X` vektorių dimensijas. Galiausiai patys vektoriai pateikiami kaip konstruktoriaus argumentai.

Kai kuriems konkretiems atvejams ParSol bibliotekoje yra numatytos kitos klasės. Pavyzdžiui, kai `X` ir `Y` vektorių dimensijos sutampa, galima naudoti klasę `CmMatrixSD<double,DC>` (`SD` yra nuo angl. *Same Dimension* matricų klasę, `DC` –

X ir Y dimensijų skaičius). Jeigu naudojamos kvadratinės matricos, joms egzistuoja klasė `CmMatrixSquare<double,DC>`. Jos konstruktoriuje pakanka nurodyti tik vieną vektorių.

Kitas matricų sukūrimo aspektas yra jų *vidinės struktūros* parinkimas. ParSol matrica gali saugoti visus savo elementus ir būti pilna (angl. *dense*), arba gali saugoti tik nenulinius elementus ir būti išretinta (angl. *sparse*). Abiem atvejais vidinė matricos realizacija skiriasi. Matricos vidinė struktūra apibrėžiama papildomu šabloniniu parametru, kurio reikšmės yra arba `PM_Dense` (pilnai matricai), arba `PM_Sparse` (išretintai). Kadangi dažniausiai naudojamos išretintos matricos, šis parametras pagal nutylėjimą yra `PM_Sparse` ir gali būti nenurodomas, kaip anksčiau pateiktuose pavyzdžiuose. Pilna kvadratinė matrica, pavyzdžiui, gali būti sukurta taip:

Pilnos kvadratinės matricos sukūrimas

```
1 CmMatrixSquare<double,2,PM_Dense> M(X);
```

Taip pat, kalbant apie vidinę matricų struktūrą, reikia paminėti vektorių elementų pasiekimą. Atliekant matricų ir vektorių sandaugą, vektorių elementų pasiekimas yra viena dažniausiai naudojamų operacijų. ParSol matricose realizuoti skirtingi elementų pasiekimo mechanizmai. Panašiai kaip ir su elementų saugojimu, jie pasirenkami matricos tipo apibrėžimo metu per šabloninius parametrus, pavyzdžiui,

Vektorių elementų pasiekimo tipo nurodymas

```
1 CmMatrixSD<double,2,PM_Sparse,Internal::MAM_FullCache> M(Y,X);
```

Egzistuoja tokie elementų pasiekimo mechanizmai:

Internal::MAM_FullCalc – kiekvieną kartą, norint pasiekti vektoriaus elementą, adresas skaičiuojamas iš naujo. Tai reikalauja daugiausiai skaičiavimų, tačiau nereikalauja papildomos atminties;

Internal::MAM_FullCache – visi adresai apskaičiuojami iš anksto ir išsaugomi, o esant reikalui, paimami. Tai greičiausias metodas, tačiau jis reikalauja daugiausiai atminties, proporcingai masyvų elementų skaičiui;

Internal::MAM_CalcCache – tai tarpinis variantas, jis išsaugoja dalį tarpinių skaičiavimų ir šią informaciją panaudoja, apskaičiuojant adresus. Jis reikalauja mažiau atminties, negu `MAM_FullCache`, o skaičiuoja greičiau, negu `MAM_FullCalc`.

Norint naudotis ParSol matricomis, reikia, kad į programą būtų įtrauktas antraštinis failas `PS_CommonMatrix_?D`, čia `?` yra vektorių, naudojamų matricos sukūrimui, dimensijos skaičius. Priklausomai nuo situacijos, gali būti įtraukti keli

tokie failai. Matricų veiksmų lygiagretinimui naudojamas vektorių funkcionalumas, ir nesvarbu, ar yra nuoseklioji, ar lygiagrečioji programos versija, naudojami tie patys antraštiniai failai.

F.2. Veiksmai su matricos elementais

Sukūrus matricą, reikia mokėti užpildyti ir nuskaityti jos elementus. ParSol matricos tam pateikia tokius metodus:

ActiveElem – grąžina požymį ar matricos elementas yra *aktyvus*, t.y. ar jis gali būti pakeistas. Pilnoms matricoms visi elementai yra aktyvūs. Išretintoms matricoms neaktyvūs elementai yra 0, tačiau iškart po sukūrimo pagal šį požymį negalima tiksliai tai nustatyti. Tikrinti elemento aktyvumą reikia visada *prieš pat* jo pakeitimą.

GetElem – grąžina elemento reikšmę.

SetElem – suteikia elementui reikšmę.

AddElem – prideda prie elemento tam tikrą reikšmę.

ProdElem – padaugina matricos elementą iš tam tikros reikšmės.

ZeroEntries() – visus matricos elementus prilygina nuliui. PETSc pakete šios operacijos analogas yra `MatZeroEntries`.

Atskirai reikia paminėti matricos elementų adresaciją. Visos ankščiau minėtos funkcijos priima elementų adresus kaip du argumentus `X` ir `Y`, kurie yra tam tikri vidiniai elementų indeksai X ir Y kryptimi. Šitų indeksų apskaičiavimui naudojami specialūs matricų objektai `X` ir `Y`. Jie turi operatorių `()`, kuris paima tokias pat koordinates, kaip ir *atitinkamas vektorius*, ir paverčia jas vidiniais indeksais. Taigi, norint sužinoti matricos elemento reikšmę, pilnas užklausos užrašas turi atrodyti taip:

ParSol matricos elemento pasiekimo pavyzdys

```
1  M.GetElem( M.Y(yy,yx), M.X(xy, xx) );
```

Matricos, kurių `X` ir `Y` vektorių dimensijos sutampa, turi specialius elemento nustatymo metodų analogus, kur prie pavadinimo prisideda `?D`, čia `?` yra dimensijų skaičius. Pavyzdžiui, `CmMatrixSD<...,2>` tipo matricos elementą galima skaityti taikant metodą `GetElem2D(yy,yx,xy,xx)`.

F.3. Veiksmai su matricomis

Su matricomis, kaip su matematiniais objektais, galima atlikti tokius veiksmus:

1. **Aritmetinės operacijos su matricomis**. Panašiai kaip su vektoriais, matricos gali būti sudedamos, atimamos, taip pat dauginamos ir dalinamos iš konstantos. Iš šių operacijų gali būti sudaromos įvairiausio ilgio išraiškos. Išraiškos yra realizuotos taikant išraiškų šablonus (angl. *expression templates*). Taip pat realizuotos operacijos `+=` ir `-=` (su matricomis arba išraiškomis), ir operatoriai `*=` ir `/=`. Pateikiame kai kurių, dažniausiai naudojamų, išraiškų pavyzdžius:
   - `M *= 2;` – PETSc funkcijos `MatScale` analogas;
   - `M2 = M;` – PETSc funkcijos `MatCopy` analogas;
   - `M2 += 3.5*M;` – PETSc funkcijos `MatAXPY` analogas;
2. **ILU faktorizacija**. Tai atliekama taikant `CmMatrixSquareLocal` klasės metodą `PerformLU`. Taigi, norint, kad ParSol matricos atliktų faktorizaciją, reikia, kad matrica būtų šio tipo, arba nukopijuojame duomenis į tokią matricą. Tai galima padaryti, nurodžius kvadratinę matricą kaip `CmMatrixSquareLocal` tipo matricos konstruktoriaus parametrą. Šios operacijos rezultatas saugomas toje pačioje matricoje.
3. **Operacijos su vektoriais**. Pagrindinė yra matricos-vektoriaus sandaugos operacija `M.Mult(Y,X)`, kuri apskaičiuoja sandaugą $Y = MX$. Taip pat realizuota BLAS 2 [12] *saxpy* tipo operacija $Z = Y + MX$, ją atlieka metodas `MultAdd(Z,Y,X)` (PETSc analogas `MatMultAdd`).

F.4. Matricų diagonaliniai elementai

Kiekvienoje kvadratinėje matricoje labai svarbūs yra diagonaliniai elementai. Darbui su diagonaliniais elementais ParSol kvadratinės matricos turi objektą `Diag`. Šis objektas leidžia tiesiogiai dirbti su diagonaliniais elementais. Pavyzdžiui, norint padidinti visus diagonalinius elementus tam tikru dydžiu, tai galima padaryti taip: `M.Diag += 5`. Be `+=`, taip pat palaikomi `-=`, `*=` ir `/=` operatoriai.

Išskirti matricos diagonalinius elementus galima naudojant paprastą priskyrimo operatorių, o nukopijuoti diagonalinius elementus į vektorių galima metodu `CopyToArray`. Metodas `CopyToArrayInverted` kopijuoja į masyvą invertuotus diagonalinius elementus, pagal formulę $a_i = \frac{1}{M_{ii}}$, jis gali būti naudingas naudojant diagonalines modifikuojančias matricas.

Reikia pastebėti, kad diagonalinių elementų pasiekimas ParSol bibliotekoje yra santykinai lėtas procesas. Jeigu diagonalinių elementų nuskaitymas dažnai naudojamas vidiniuose cikluose, rekomenduojama juos nukopijuoti į atskirą vektorių.

2.4.2. Nuoseklusis bibliotekos panaudojimo pavyzdys

Šiame poskyryje demonstruojamas bibliotekos ParSol naudojimas realizuojant nuoseklųjį algoritmą. Šiuo pavyzdžiu siekiama praktiškai parodyti ParSol bibliotekos taikymo paprastumą ir efektyvumą. Kaip pavyzdys pasirinkta programa, kuri, paėmus 2D masyvą, užpildo ji pradinėmis reikšmėmis, o paskui per 100 iteracijų jį glodina. Rezultatas išvedamas į `stdout`. Originalios programos, parašytos besinaudojant standartinėmis C/C++ priemonėmis, tekstas pateikiamas žemiau:

2D masyvo glodinimas, naudojant standartines C/C++ priemones

```
#include <algorithm>
#include <iostream>

const int Xsize = 20;
const int Ysize = 20;
typedef double Arr[Ysize][Xsize];

void SetInit( Arr& val ) {
  for( int y = 0; y < Ysize; ++y )
    for( int x = 0; x < Xsize; ++x )
      val[y][x] = ((y>5)&&(y<15)&&(x>5)&&(x<15)) ? 1 : 0 ;
}

void Smooth( const Arr& From, Arr& To ) {
  for( int y = 0; y < Ysize; ++y )
    for( int x = 0; x < Xsize; ++x )
      if ((y==0)||(y==Ysize-1)||(x==0)||(x==Xsize-1))
        To[y][x] = From[y][x];
      else
        To[y][x] = From[y][x] + 0.001*
          (From[y+1][x] + From[y-1][x] + From[y][x+1] +
           From[y][x-1] - 4.0*From[y][x]);
}

void Print( const Arr& a ) {
  for( int y = 0; y < Ysize; ++y )
    for( int x = 0; x < Xsize; ++x )
      std::cout << '[' << y << ',' << x << "] = "
        << a[y][x] << std::endl;
}

int main( int argc, char* argv[] ) {
  Arr X1, X2;
  SetInit( X1 );
```

```
35
36   Arr* From = &X1;
37   Arr* To = &X2;
38   for( int i = 0; i < 100; ++i ) // iterations
39   {
40     Smooth( *From, *To );
41     std::swap( From, To );
42   }
43
44   Print( *To );
45   return 0;
46 }
```

Ši programa, perrašyta taikant ParSol biblioteką, atrodo taip:

2D masyvo glodinimas, naudojant ParSol biblioteką

```
1  #include <algorithm>
2  #include <iostream>
3  #include "PS_CommonArray_2D.h"
4
5  using namespace NS_PARSOL;
6  typedef CmArray_2D<double> Arr;
7
8  void SetInit( Arr& val ) {
9    for( int y = val.StartLocalY(); y <= val.FinishLocalY(); ++y )
10     for( int x = val.StartLocalX(); x <= val.FinishLocalX(); ++x )
11       val(y,x) = ((y>5)&&(y<15)&&(x>5)&&(x<15)) ? 1 : 0 ;
12 }
13
14 void Smooth( Arr& From, Arr& To ) {
15   for( int y = From.StartLocalY(); y <= From.FinishLocalY(); ++y )
16     for( int x = From.StartLocalX(); x <= From.FinishLocalX(); ++x )
17       if ((y==From.StartGlobalY())||(y==From.FinishGlobalY()) ||
18           (x==From.StartGlobalX())||(x==From.FinishGlobalX()))
19         To(y,x) = From(y,x);
20       else
21         To(y,x) = From(y,x) + 0.001*
22           (From(y+1,x) + From(y-1,x) + From(y,x+1) +
23            From(y,x-1) - 4.0*From(y,x));
24 }
25
26 int main( int argc, char* argv[] ) {
27   CrossStencil_2D St(1);
28   Arr X1(St,20,20), X2(X1);
29   SetInit( X1 );
30
```

```
31  Arr* From = &X1;
32  Arr* To = &X2;
33  for( int i = 0; i < 100; ++i ) // iterations
34  {
35    From->ExchangeData();
36    Smooth( *From, *To );
37    std::swap( From, To );
38  }
39
40  To->DumpElems( std::cout );
41  return 0;
42 }
```

Kaip matome, teko pakeisti tokias programos vietas:

- Pridėtas naujas antraštinis failas `PS_CommonArray_2D` ir `NS_PARSOL` pavadinimų erdvė (angl. *namespace*) – dvi eilutės programoje.
- Pasikeitė masyvų tipų apibrėžimas – kadangi jis buvo apibrėžtas tik vienoje vietoje, teko padaryti pakeitimus tik vienoje programos eilutėje.
- Teko *visur* pakeisti kreipinius į masyvo elementą: pvz., vietoj `X[y][x]` dabar reikia naudoti `X(y,x)`. Tai jau nemažai pakeitimų. Tačiau, naudojant tinkamas priemones, ir šis procesas yra labai greitas. Pavyzdžiui, *VIM* tekstiniame redaktoriuje [89] ši užduotis galėtų būti atlikta taikant *vieną* komandą:
  `:%s/\[\([^[]\{-}\)]\[\(.\{-}\)]/(\1,\2)/g`, Čia naudojame reguliarias išraiškas (angl. *regular expressions*).
- Masyvų dimensijos, anksčiau saugotos kaip globalios konstantos, dabar pakeistos atitinkamais ParSol masyvų klasių metodais visur, išskyrus pirmo masyvo iniciacijos.
- Pridėtas kryžminio šablono iniciacijos kodas (viena eilutė).
- Prisidėjo komanda keistis duomenimis (viena eilutė). Šiuo atveju, kai turime neciklinį masyvą, šios komandos galėjome ir nerašyti. Tačiau programos kokybės ši eilutė nemažina, o, nusprendus naudoti ciklinius masyvus arba darant lygiagrečiąją programos versiją, jos vis tiek reikėtų. Ją verta pridėti dabar, kad ateityje būtų galima koncentruotis tik ties algoritminiu programos aspektu.
- Galiausiai duomenų išvedimas buvo pakeistas vidiniu ParSol masyvų me-

todu. Jis, galbūt, išveda duomenis ne visai tokiame formate, kaip originali programa, tačiau yra automatiškai lygiagretinamas.

Visa tai apibendrinant, galima pasakyti, kad egzistuojančios C++ programos migracija į programą, naudojančią ParSol biblioteką, susideda iš dviejų dalių:

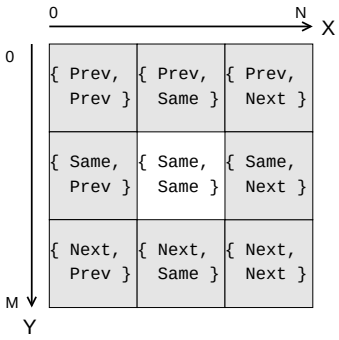
1. Globalių nustatymų pakeitimas. Jis atliekamas keliose visos programos vietose;
2. Masyvų vartotojo sąsajos elementų (elemento pasiekimas, diapazonų ribų nustatymas ir pan.) pakeitimas. Tokių pakeitimų gali būti daug, tačiau jie yra panašūs, ir gali būti atlikti labai greitai, jei naudojame atitinkamas priemones.

2.4.3. Vidinės logikos lygmuo

Šiame poskyryje aprašomi ParSol bibliotekos realizacijos ypatumai. Tai leidžia geriau suprasti bibliotekos sandarą ir realizavimo principus, padeda orientuotis bibliotekos vidiniame kode (apimtis apie 20 000 eilučių, realizuota virš 70 įvairios paskirties klasių), efektyviai naudoti ir plėsti ParSol įrankį.

A. Kaimynų adresacija

Lygiagrečiųjų algoritmų realizacijos efektyvumas esminiai priklauso nuo kaimyninių procesų vykdomų komunikacijų ir sąveikos, todėl labai svarbu suprasti, kaip ParSol bibliotekoje vykdoma kaimynų adresacija. Pradžioje panagrinėkime 1D atvejį, kai visi procesai išdėstyti tiesiškai. Tada kiekvienas procesas gali tiesiogiai kontaktuoti tik su dviem gretimais procesais: su prieš jį esančiu ir už jo patalpintu kaimynais. Kryptis, kuri nulemia, kuris kaimynas sekantis, o kuris prieš tai esantis, sutampa su tos dimensijos elementų indekso didėjimo tvarka.



2.9 pav. *Kaimynų adresacija ParSol bibliotekoje (2D)*

ParSol bibliotekoje yra specialiai apibrėžtas duomenų tipas

Duomenų tipo ***NeighbourPos*** *apibrėžimas*

```
enum NeighbourPos {
  Prev = -1, // ankstesnis kaimynas
  Same =  0, // ta pati vieta
  Next = +1  // sekantis kaimynas
};
```

kurio paskirtis - nusakyti kaimyno padėtį 1D atveju. Daugiamačiu atveju, kaimyno padėtis nusakoma `NeighbourPos` tipo elementų masyvu, kurio ilgis yra lygus dimensijų skaičiui. Dimensijos masyve yra pasiskirsčiusios taip pat, kaip ir bendrojoje elementų adresacijoje. 2D atveju, kaimynų adresacija schematiškai pavaizduota 2.9 paveiksle.

Taip pat ParSol bibliotekoje svarbi *priešpriešinio kaimyno* sąvoka. Vadinkime kaimynus priešpriešiniais, jeigu jų daugiamačiuose adresuose ten, kur yra vieno kaimyno `Prev`, kito kaimyno yra `Next`, `Next` → `Prev`, `Same` → `Same`. Pavyzdžiui, 2D atveju kaimyno `{Next,Prev}` priešpriešinis kaimynas yra `{Prev,Next}`, o kaimyno `{Next,Same}` – kaimynas `{Prev,Same}` (tam gali būti panaudota ParSol bibliotekos funkcija `Opposite`). Grafiškai, kaip galima matyti 2.9 paveiksle, priešpriešiniai kaimynai yra simetriški centrinio elemento atžvilgiu. Priešpriešiniai kaimynai yra svarbūs, nes, sąveikaujant su priešpriešiniais kaimynais, daugelis sąveikos parametrų išlieka tokie patys.

B. Šablonų (*stencil*) realizacija

Šablonai ParSol bibliotekoje atlieka keletą vaidmenų:

- Masyvams ir vektoriams šablonai pateikia informaciją, kiek ir kokius elementus reikia gauti iš kiekvieno kaimyno, taip pat kokius elementus pateikti kaimynams.
- Matricoms šablonai papildomai pateikia informaciją, naudojant kurią galima apytiksliai numatyti, kiek vietos turi būti rezervuota nenulinių koeficientų saugojimui (išretintų matricų atveju).

Tam, kad būtų aišku, kokiais elementais reikia keistis su kaimynais, šablonai *kiekvienam kaimynui* gali nurodyti maksimaliai nutolusio elemento atstumą *abiem kryptimis* pagal *kiekvieną* dimensiją. Galima naudoti šablono metodus

Labiausiai nutolusio šablono elemento suradimas kiekvienam kaimynui

```
int GetStencilMinPos( int Dim, NeighbourPos* Pos );
int GetStencilMaxPos( int Dim, NeighbourPos* Pos );
```

Argumentas `Pos` yra kaimyno daugiamatis adresas ir nurodo, kokio kaimyno reikšmes norime žinoti. Argumentas `Dim` nurodo, pagal kokią dimensiją norime žino-

ti informaciją. Atstumą *į priekį* (atitinka kitą kaimyną) galime sužinoti su `Max`-funkcija, atstumą *atgal* – su `Min`-funkcija. Funkcijos grąžina maksimaliai nutolusio elemento atstumą (visada teigiama reikšmė). Priešingų kaimynų atveju šios funkcijos grąžina tokias pat reikšmes. Masyvai ir vektoriai priklausomai nuo kaimynų padėties naudoja `Min`- arba `Max`-funkcijas.

Paprasti ir kryžminiai šablonai naudoja savo parametrus maksimaliai nutolusių elementų atstumams skaičiuoti. Universalus šablonas nustato šias reikšmes, perrinkdamas visus užregistruotus taškus. Reikšmės apskaičiuojamos taškų pridėjimo arba išmetimo metu, paskui naudojamos jau išsaugotos reikšmės – tai padidina šablono veikimo greitį.

Nustatyti, kiek taškų turi šablonas, galima taikant metodą `ActivePointCount`. Šį metodą skirtingo tipo šablonai realizuoja skirtingai. Pavyzdžiui, kryžminio šablono taškų skaičius apskaičiuojamas pagal formulę

$$P = \sum_{i=1}^{DC} \Big(Max(i) + Min(i) \Big) + 1 \,, \tag{2.5}$$

o paprasto šablono – pagal formulę

$$P = \prod_{i=1}^{DC} \Big(Max(i) + Min(i) + 1 \Big) \,. \tag{2.6}$$

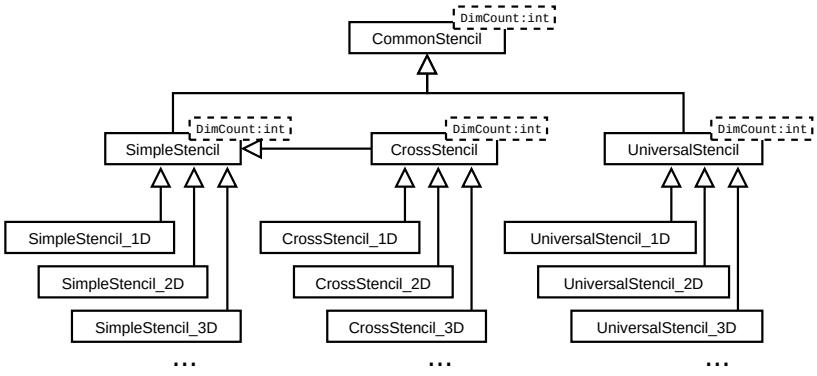
Čia P pažymėtas šablono taškų skaičius, DC – dimensijų skaičius, Min ir Max funkcijos grąžina minimalų ir maksimalų taškų nuotolį nuo centro pagal tam tikrą dimensiją (funkcijos `StencilMin(i)`, `StencilMax(i)` arba jų analogai konkrečioms dimensijoms `Min{X|Y|Z}()` ir `Max{X|Y|Z}`). Universalus šablonas tiesiog grąžina užregistruotų taškų skaičių.

Taip pat šablonas turi metodą, atsakantį į klausimą, ar taškas priklauso šablono uždengiamam plotui (apie šablonų plotus buvo kalbėta prie šablonų palyginimų). Pavyzdžiui, į šablono, pavaizduoto 2.8 c) paveiksle, plotą patenka 16 taškų, nors jame yra tik 14 taškų. Šis metodas yra `bool PointIsMine(int p[])`. Jo argumentas yra taško daugiamatė koordinatė, o grąžinama reikšmė – loginė, *true*, jei taškas priklauso šablono plotui, kitaip – *false*.

Šablonų klasių diagrama pavaizduota 2.10 paveiksle. Visi šablonai yra bibliotekos klasės `CommonStencil` palikuonys. Norint kurti naują šabloną, jis taip pat turi būti šios klasės palikuonis. Kalbant apie konkretaus tipo šabloną, jis realizuojamas naudojant kelias klases:

- Tėvo klasė, kurioje yra bendras šablono kodas, nepriklausomas nuo dimensijų skaičių.

- Taikomosios klasės-palikuonys, pritaikytos konkrečiam dimensijų skaičiui. Šis dimensijų skaičius atsispindi klasės pavadinime. Standartinėse ParSol bibliotekos šabloninėse klasėse pagrindinis funkcionalumas nepriklauso nuo dimensijų skaičių, todėl taikomųjų klasių užduotis – tiesiog pateikti vartotojui patogią sąsają.



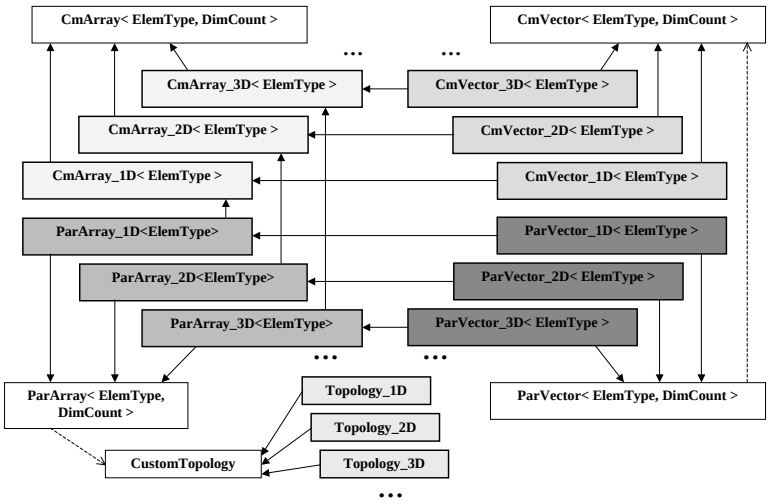
2.10 pav. *Šablonų klasių diagrama*

C. Masyvų realizacija

Masyvų klasių struktūra pavaizduota 2.11 paveiksle. Pagrindinis masyvų funkcionalumas realizuotas klasėje `CmArray`. Ši klasė atlieka tokius pagrindinius veiksmus:

- apskaičiuoja visų diapazonų koeficientus, paskiria ir atlaisvina atmintį masyvo elementams;
- valdo šablonus (stencil);
- apskaičiuoja, kokie elementai priklauso duotajai sričiai, atlieka duomenų apsikeitimą;
- optimizuoja elementų pasiekimo greitį. Tai atliekama, iš anksto apskaičiavus koeficientus, iš kurių paskui tiesiogiai dauginami ir sudedami elemento atitinkamų dimensijų indeksai.

Teoriškai skaičiavimams galima naudoti ir šią klasę, tačiau žymiai patogiau yra naudotis konkrečiomis klasėmis, pritaikytomis konkrečioms dimensijoms. Jos ne tik pateikia vartotojui patogią sąsaja, bet kai kuriais atvejais ir optimizuoja bendrą daugiadimensinį kodą konkrečiam dimensijų skaičiui.



2.11 pav. *ParSol masyvų ir vektorių klasių diagrama*

ParSol masyvuose vidinė elementų išdėstymo tvarka yra tokia pat, kaip ir C masyvuose – eilutėmis. Pradžioje keičiasi paskutinis indeksas (vadinamas X), paskui priešpaskutinis, ir taip toliau.

D. Vektorių realizacija

Vektorių klasių hierarchija kartu su masyvais parodyta 2.11 paveiksle. Pagrindinis ParSol vektorių funkcionalumas (šis kodas yra bendras visoms dimensijoms), yra realizuotas klasėje `CmVector`. Tačiau ši klasė negali būti naudojama atskirai. Joje yra tik *matematinis* funkcionalumas, ji nekartoja visų tų funkcijų, kurias atlieka klasė `CmArray`. Kita vertus, `CmVector` klasė nėra sukurta tam, kad būtų taikoma atskirai. Visi vektoriai, skirti vartotojų naudojimui, yra klasės, kurios paveldi abiejų klasių `CmArray_?D` ir `CmVector` savybes. Jose kodo kiekis yra minimalus, ir jis atlieka tokias funkcijas:

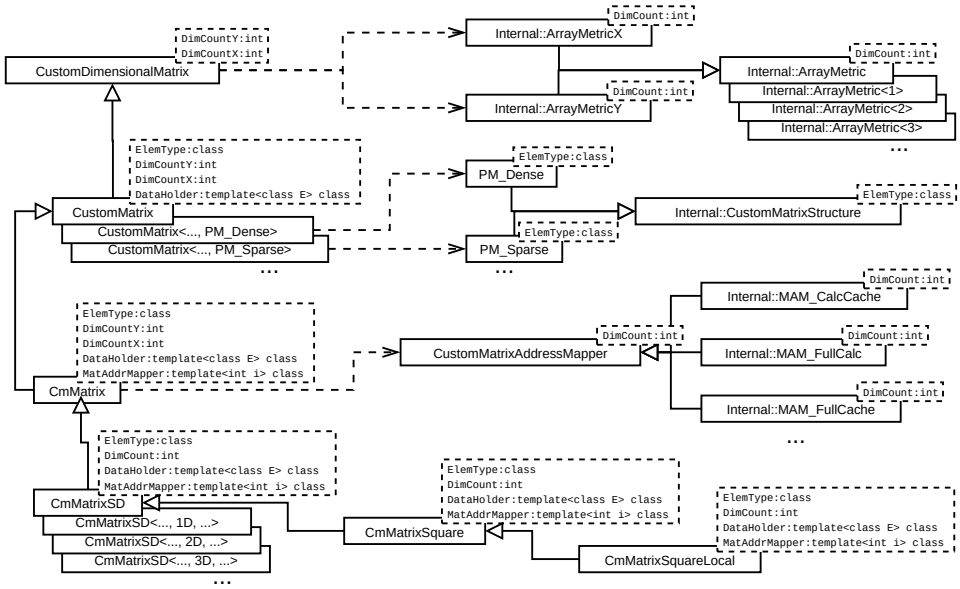
- nurodo reikiamus ryšius tarp masyvų ir vektorių funkcionalumo;
- pateikia vartotojui patogesnę sąsają;
- optimizuoja kodą konkrečiam dimensijų skaičiui.

Taip pat būtent ParSol vektorių lygyje yra realizuotas IŠ palaikymas. Tai yra natūralu, nes matematinėms operacijoms su vektoriais atlikti reikalaujama, kad šias operacijas būtų galima atlikti ir su kiekvienu elementu – vektoriaus reikalavimas. Iš pradžių IŠ technologija buvo sukurta paprastų, nešabloninių masyvų ir vektorių optimizavimui [22]. Tačiau be pakeitimų ši technologija netinka šabloninių

vektorių operacijų realizacijai [92]. Maža to, pradinis IŠ variantas buvo realizuotas, panaudojant iteratorius. Tai buvo padaryta, siekiant turėti apibendrintą kodą, kaip STL. ParSol bibliotekoje, kaip ir [23], buvo nuspręsta perdavinėti tiesiai nuorodas į ParSol vektorius. Nors toks sprendimas neleidžia panaudoti IŠ kodą kitur, išskyrus ParSol bibliotekoje, užtat tampa įmanomas papildomas optimizavimas ir klaidų patikrinimas.

E. Matricų realizacija

Matricų klasių diagrama pavaizduota 2.12 paveiksle. Tokia didelė klasių įvairovė pasirinkta tam, kad ParSol matricos būtų kuo lankstesnės, ir jų funkcionalumą būtų galima kuo lengviau išplėsti.



2.12 pav. *ParSol matricų klasių diagrama*

Visų matricų tėvo klasė yra `CustomDimensionalMatrix`, kurios pagrindinis funkcionalumas yra tas, kad ji moka išsaugoti informaciją apie X ir Y vektorius. Tam naudojamos pagalbinės klasės `ArrayMetricX` ir `ArrayMetricY`, kurios yra klasės `ArrayMetric` palikuonys. Būtent šioje klasėje yra pagrindinis ParSol masyvų parametrų saugojimo ir apdorojimo funkcionalumas. Tai yra šabloninė klasė, ir egzistuoja jos specialūs variantai skirtingiems dimensijų skaičiams, su specialiais metodais ir optimizuotu funkcionalumu.

Jos palikuonis yra klasė `CustomMatrix`, kurioje papildomai atsiranda matricos vidinės struktūros palaikymas. Specialiai struktūros palaikymui išskirtas pali-

kuonis tam, kad būtų galima kuo lanksčiau palaikyti įvairias matricų vidines struktūras. Ši klasė turi specializacijas skirtingoms duomenų struktūroms, kur realizuoti specifiniai, tik tom struktūroms būdingi metodai.

`CustomMatrix` klasės palikuonis yra klasė `CmMatrix`, pirmoji klasė, kuri gali būti naudinga vartotojui. Ši klasė, papildomai iškvietusi į pagalbą vieną iš vektorių elementų pasiekimo mechanizmo klasių, kurios jau buvo aprašytos ankščiau, turi visą funkcionalumą, reikalingą operacijų su matricomis atlikimui. Tačiau tai yra bendra klasė, panaši į `CmArray`. Realiai dažnai pasitaiko specifiniai matricų atvejai, kuriems sukurtos atskiros, patogesnės klasės.

Tiesioginis klasės `CmMatrix` palikuonis yra klasė `CmMatrixSD`, kuri skirta atvejams, kai X ir Y vektorių dimensijos sutampa. Ši klasė turi specializacijas, kurios pateikia vartotojui patogesnę sąsają tam tikrų dimensijų skaičiaus atvejais. Galiausiai šios klasės pagrindu sukurta klasė `CmMatrixSquare`, kuri skirta kvadratinėms matricoms, dažniausiai naudojamoms skaičiavimuose. Specifinis kvadratinės matricos atvejis yra klasė `CmMatrixSquareLocal`, naudojama LU faktorizacijai. Ji pasižymi tuo, kad iniciacijai naudoja tik lokalią masyvo X dalį.

2.4.4. Lygiagrečioji bibliotekos versija

Šiame poskyryje kalbama apie ParSol bibliotekos nuosekliųjų klasių lygiagretinimo realizacijos ypatumus.

A. Bendras MPI palaikymas

Kaip yra žinoma, prieš vykdant visas MPI komandas (išimtis yra komandą `MPI_Initialized`) MPI terpė turi būti inicializuota (naudojant `MPI_Init`). Taip pat, užbaigus darbą, turi būti iškviesta darbą su MPI terpe užbaigianti komanda `MPI_Finalize`. ParSol bibliotekoje šis funkcionalumas yra realizuotas klasėje `MPI_Environment`, panaudojant RAII projektavimo šabloną[6] [21]. MPI biblioteka yra inicializuojama šios klasės sukūrimo metu, ir uždaroma, kai ši klasė panaikinama. Taigi, jeigu `main` funkcijos pradžioje tiesiog sukuriama tokio tipo klasė, tai ji bus automatiškai panaikinta išėjimo iš programos metu, ir MPI biblioteka bus uždaroma, kai pasibaigs funkcija `main`, nesvarbu, ar išėjimas iš programos yra normalus ar avarinis.

Nors sėkmingam programos vykdymui užtenka vienos klasės sukūrimo, ParSol biblioteka leidžia kurti daug tokio tipo klasių. Tokiu atveju skaičiuojama, kiek klasių atidaryta. MPI biblioteka inicializuojama tik pirmos klasės sukūrimo metu, o užbaigiama tik tada, kai naikinama paskutinė klasė. Taigi bibliotekos ar tam tikros vartotojo programos komponentai gali užtikrinti, kad jų vykdymo

[6] Nuo angl. *Resource Acquisition is Initialization* – resursų gavimas yra iniciacija.

metu MPI terpė bus visada inicializuota, jei vykdymo metu saugos sukurtą *savo* `MPI_Environment` tipo objektą.

B. Topologijos

Nagrinėdami HPF lygiagretinimo procesą, matėme, kad jame svarbią vietą užima procesų topologijos nurodymas, tai yra nurodymas, kokių masyvų duomenys ir kaip turi būti paskirstyti tarp procesų. ParSol bibliotekoje, lygiagretinant programą, procesų topologija taip pat turi būti nurodoma. Tai daroma taikant ParSol topologijos klases. Šių klasių schema pavaizduota 2.11 paveiksle.

ParSol bibliotekoje šiuo metu naudojamos dekartinės topologijos. Kuriant biblioteką, buvo nuspręsta, kad topologijos turi būti kuriamos naudojant MPI bibliotekos priemones. MPI realizacijos gali naudoti papildomą informaciją apie platformą ir paskirstyti procesus optimaliu, komunikavimo kaštus mažinančiu būdu. Naudojant MPI, topologija turi būti sukurta (funkcija `MPI_Cart_create`), taikant specialius metodus ištraukta informacija apie kiekvieno proceso padėtį joje, galiausiai, programos vykdymo pabaigoje, turi būti atlaisvinti visi topologijos resursai (funkcija `MPI_Comm_free`).

ParSol bibliotekoje šį funkcionalumą realizuoja klasė `Topology` ir jos palikuonys. Labai svarbus šiose klasėse yra *konstruktorius*, kuriais galima nurodyti, į kiek dalių pagal kiekvieną dimensiją yra dalinamas vektorius ar masyvas (palikuonys turi specialius konstruktorius, patogiausius jų dimensijų skaičiui). Pagal tam tikrą dimensiją galima nurodyti ir *nulį*, tada sistema bandys automatiškai parinkti optimalų procesų skaičių. Taip pat galima nurodyti, ar topologija turi būti ciklinė, ar ne. Nėra privaloma cikliniams masyvams naudoti tik ciklines topologijas, ir atvirkščiai, tačiau rekomenduojama taip daryti, norint pasiekti maksimalų efektyvumą.

Visada, vykdant lygiagrečiąją programos versiją, reikia jai paskirti tiek procesų, kad topologija galėtų juos išnaudoti efektyviausiai. Pavyzdžiui, jei programoje nurodyta topologija 2×2, nėra prasmės vykdyti tokią programą penkiuose ir daugiau procesų, nes bus panaudoti tik keturi. Net kai topologijos automatiškai adaptuojasi prie procesų skaičiaus, ne bet koks procesų skaičius yra optimalus. Pavyzdžiui, dvimatė topologija iš 7 procesų galės panaudoti daugiausiai 6 (2×3). Patikrinti, ar procesas priklauso topologijai, galima taikant topologijos metodą `IAmUsed()`.

Programoje gali būti sukurta ne viena topologija. Iš visų topologijų, viena yra pagrindinė, ir į ją rodo rodyklė `DefaultTopology`. Kuriant naują topologiją, ji šitą nuorodą peradresuoja į save, tačiau vėliau jos reikšmę galima pakeisti į bet kokią kitą. Pagrindinės topologijos ypatumas yra tas, kad ji naudojama kaip topologija visiems naujai kuriamiems ParSol masyvams ir vektoriams. Kai užduočiai

reikia tik vienos topologijos, pakanka tiesiog sukurti jos klasę, ji automatiškai taps pagrindine, ir visi masyvai naudos ją.

Tačiau masyvai nėra vieninteliai topologijos naudotojai. Topologijos pateikia vartotojui visą reikiamą informaciją apie proceso padėtį joje, taip pat visą eilę globalių funkcijų, operuojančių su procesais, priklausančias šiai topologijai:

- `Barrier()` – sinchronizuoja procesų darbą, panaudodama `MPI_Barrier`;
- `SimpleTextOut(s)` – išveda tekstą `s` į `stdout`. Skirtingai nuo standartinio „cout <<“ išvedimo, išveda tik vieną karta, o ne tiek, kiek yra procesų;
- `GlobalAND(b)` – grąžina `true` tik tuo atveju, jei visi topologijoje dalyvaujantys procesai perdavė šiai funkcijai reikšmę `true`;
- `GlobalOR(b)` – grąžina `true` tuo atveju, jei bent vienas topologijoje dalyvaujantis procesas perdavė šiai funkcijai reikšmę `true`;
- `GlobalSUM(x)` – grąžina sumą reikšmių, perduotų šiai funkcijai topologijoje dalyvaujančių procesų;
- `GlobalMAX(x)` – grąžina didžiausią iš visų perduotų šiai funkcijai topologijoje dalyvaujančių procesų reikšmių;
- `GlobalMIN(x)` – grąžina mažiausią iš visų perduotų šiai funkcijai topologijoje dalyvaujančių procesų reikšmių;

C. Masyvų ir vektorių lygiagretinimas

Masyvia ir vektoriai dalinami tarp procesų priklausomai nuo to, kokia topologija yra pagal nutylėjimą masyvo kūrimo metu. Masyvų ir topologijų dimensijų skaičius gali ir nesutapti. ParSol bibliotekoje topologijos teoriškai turi be galo daug dimensijų, tik pagal visas aukštesnes dimensijas procesų skaičius yra 1. Taigi, vienmatė topologija, kurioje yra 2 procesai, padalins dvimatį masyvą taip pat, kaip dvimatė topologija 2×1. Naudoti aukštesnę, nei masyvų, dimensijų topologiją galima, tačiau nėra prasmės, nes visos dimensijos, aukštesnės nei masyvų, kur yra daugiau nei vienas procesas, sukels masyvų kopijavimą. Pavyzdžiui, jei mes bandysime dvimatį masyvą paskirstyti trimatėje topologijoje $2 \times 3 \times 4$, tai rezultate gausime keturius lygiagrečiai veikiančius ir tas pačias operacijas atliekančius algoritmus, operuojančius su masyvais, padalintais dvimatėje topologijoje 2×3.

Visas lygiagretinimo funkcionalumas yra sutelktas klasėje `ParArray`. Kaip ir klasės `CmVector` atveju, tai nėra savarankiškam naudojimui skirta klasė, tačiau ji turi būti paveldėta, kartu su `CmArray` klasės palikuonimi, kad jos palikuonis turėtų lygiagretinimo funkcionalumą. Šios klasės pagrindinė užduotis – vykdyti duomenų apsikeitimą tarp masyvo procesų dalių. Teoriškai, kiekviena masyvo dalis, priklausanti tam tikram procesui, gali bendrauti daugiausiai su 3^D kaimynais

(įskaitant ir patį save), kur D yra dimensijų skaičius. Klasė `ParArray` saugo visą masyvą specialių klasių, kiekvienos iš kurių paskirtis – bendravimas su tam tikru kaimynu. Taikant komandas `MPI_Type_indexed` ir `MPI_(Send/Recv)_init` iš anksto paruošiama informacija, iš kur turi būti kopijuojami duomenys, po to, kai ateina laikas keistis duomenimis, tai atliekama maksimaliai efektyviai.

Įgyvendinant technologijos principus, ParSol bibliotekoje realizuoti tokie duomenų apsikeitimo organizavimo tipai:

detAllAtOnce (visi kartu) – Siuntimo ir gavimo pradžią atliekama taikant MPI funkciją `MPI_Start`. Duomenų apsikeitimas pasibaigia, kai baigiamos vykdyti *visos* atitinkamos `MPI_Wait` komandos. Šiame metode galima atskirti duomenų apsikeitimo pradžią ir pabaigą, ir tarp jų atlikinėti įvairius veiksmus. ParSol bibliotekoje tokios duomenų siuntimo pradžios metodas yra `ExData_Init`, o užbaigimo – `ExData_Finish`. Bibliotekos metodas `ExchangeData` būtent juos ir naudoja vieną paskui kito, kai duomenimis apsikeičiama šiuo būdu. Atitinkamai šiuos metodus galima naudoti vietoj metodo `ExchangeData`, tada tarp jų gali būti įterptos įvairios operacijos. Tiesa, šiuo atveju jau nebus galima pasinaudoti kitais duomenų apsikeitimo mechanizmais.

detPairByPair (poromis);

detPairOrdered (poromis, tam tikra tvarka).

Pagal nutylėjimą, kuriant naują ParSol lygiagretų masyvą, pasirenkamas duomenų apsikeitimo būdas `detAllAtOnce`. Tačiau jį bet kada galima pakeisti taikant metodą `DataExType`, pavyzdžiui, `Arr.DataExType()=detPairByPair`.

Panašiai kaip ir nuosekliųjų vektorių atveju, lygiagrečiųjų vektorių funkcionalumas sutelktas `ParVector` klasėje. Klasė `ParVector` negali būti naudojama savarankiškai, ji skirta lygiagrečiųjų vektorių ir funkcionalių lygiagrečiųjų masyvų klasių palikuonių kūrimui, kaip tai pavaizduota 2.11 paveiksle. Realizuojant lygiagrečius vektorių metodų analogus, naudojami nuosekliųjų ParSol vektorių metodai, taip pat topologijų globalių operacijų (prasideda žodžiu `Global...`) funkcijos.

Galiausiai reikia paminėti naudojamus MPI duomenų tipus. MPI standartas apibrėžia primityvius MPI duomenų tipus, atitinkančius svarbiausius C/C++ duomenų tipus, tokius kaip `double`, `int` ir pan. Kai ParSol masyvai sudaryti iš elementų, kuriems egzistuoja primityvūs MPI tipai, kompiliatorius pasirenka juos. Priešingu atveju, sukuriamas naujas MPI duomenų tipas, kurio pagrindas yra `MPI_BYTE`, o ilgis lygus reikiamo duomenų tipo ilgiui. ParSol biblioteka laiko visą sąrašą tokių sukurtų duomenų tipų, ir duomenų tipams su vienodu ilgiu

nekuriami skirtingi MPI duomenų tipai. Primityvūs duomenų tipai nuo visų kitų atskiriami taikant šabloninę specializaciją.

D. Matricų lygiagretinimas

Skirtingai nuo masyvų ar vektorių, ParSol matricos šiuo metu neturi specifinio lygiagretinimo kodo. Taip yra todėl, kad ParSol matricų sukūrimui naudojami ParSol masyvai. Lygiagrečiojoje programos versijoje ParSol masyvai iškart turi tik šiam procesui reikiamą elementų dalį, ir tai automatiškai panaudojama matricų konfigūracijos sukūrimui. Šiuo metu ParSol matricos neapsikeičia elementais tarpusavyje.

2.4.5. Lygiagrečiųjų duomenų algoritmų lygiagretinimas taikant biblioteką

Šiame poskyryje praktiškai parodoma, kaip atliekamas programų lygiagretinimas taikant ParSol biblioteką. Lygiagretinimas susideda iš tokių veiksmų:

1. Pakeičiami kai kurie antraštiniai failai, iš nuosekliosios versijos į lygiagrečiąją;
2. Vietoj nuosekliųjų ParSol klasių naudojami jų lygiagretieji analogai;
3. Programos pradžioje pridedamas MPI terpės iniciacijos kodas;
4. Aprašoma norima procesų topologija;
5. Jei reikia, pridedamos duomenų apsikeitimo komandos reikiamose vietose;
6. Programa kompiliuojama su MPI bibliotekomis;
7. Programa leidžiama taikant MPI paleidėją.

Kaip pavyzdį imsime jau anksčiau aptartą programą, kuri buvo realizuota panaudojant ParSol bibliotekos klases (62 p.). Atliekant 1 žingsnį, pakeičiamas antraštinis failas iš `PS_CommonArray_2D.h` į `PS_ParallelArray_2D.h`. 2 žingsnio metu pakeičiamas tipo `Arr` apibrėžimas. 3 ir 4 žingsniai realizuojami funkcijos `main` pradžioje. 5 žingsnio šiuo atveju realizuoti nebereikia. Gautasis kodas pateiktas žemiau (nepasikeitusios kodo dalys nerodomos trumpumo dėlei):

2D masyvo glodinimo programos lygiagrečioji versija

```
1 #include <algorithm>
2 #include <iostream>
3 #include "PS_ParallelArray_2D.h"
4
5 using namespace NS_PARSOL;
```

```
6 typedef ParArray_2D<double> Arr;
7
8 ... - tas pats kodas
9
10 int main( int argc, char* argv[] ) {
11   MPI_Environment mi( argc, argv );
12   Topology_2D Topology(2,2);
13
14   ... - tas pats kodas
15   return 0;
16 }
```

6 ir 7 žingsniai priklauso nuo terpės, kurioje vyksta programos kūrimas. Pavyzdžiui, Linux sistemoje, kur yra įdiegta LAMMPI MPI realizacija [33], 6 žingsnis atliekamas, pakeitus g++ į `mpiCC` kompiliavimo ir sujungimo metu. Lygiagrečioji programos versija paleidžiama, įvykdžius komandą

```
> mpirun -np 4 executable
```

Iš ankščiau pateikto proceso aprašymo galima padaryti išvadą, kad taisyklingai parašytos programos lygiagretinimas susideda iš kelių fiksuoto dydžio papildymų arba pakeitimų, dažniausiai žinomose vietose. Kitaip tariant, lygiagretinimo procesas yra *pusiau automatinis*.

2.5. Realizacijos efektyvumo analizė

Šiame poskyryje tiriamas technologijos realizacijos efektyvumas. Atliekami skaitiniai eksperimentai, lyginant tam tikrus bibliotekos ParSol realizacijos aspektus su žinomais standartiniais sprendimais.

2.5.1. Masyvo elemento pasiekimo greičio įvertinimas

Tiesioginis darbas su masyvų elementais, tai yra rašymas ir skaitymas, yra vienos dažniausiai naudojamų masyvų savybių, todėl greita šių operacijų realizacija yra būtinas faktorius, norint pasiekti didelį su masyvais dirbančio algoritmo greitį. Šiame eksperimente buvo bandomas ParSol masyvų elementų pasiekimo greitis, lyginant su standartiniais C/C++ masyvais.

Eksperimentas buvo atliekamas kompiuteryje su tokiais parametrais: CPU 2,4 GHz AMD 64 bitų, RAM 512 MB, Debian Linux operacinė sistema, kompiliatorius g++ v. 3.3.5. Buvo matuojamas sugaištas procesoriaus laikas *user ti-*

me. Buvo tiriami 1D, 2D ir 3D `double` tipo masyvai, kiekvienu atveju bendras masyvo elementų kiekis buvo 8 000 000 elementų (8 000 000, 2 000 × 4 000 ir 200 × 200 × 200 atitinkamai). Matavimas buvo atliekamas taikant tokį algoritmą (jo realizacija pateikiama prieduose):

1. Atsimenamas pradinis bandymo laikas T_{bp};
2. Apklausiami[7] visi masyvo elementai po vieną kartą;
3. Žingsnis 2 kartojamas N kartų;
4. Paimamas galutinis bandymo laikas T_{bg} ir paskaičiuojamas sunaudotas laikas – skirtumas $T_b = T_{bg} - T_{bp}$;
5. Atsimenamas pradinis palyginimo laikas T_{pp};
6. Vykdomas algoritmas, identiškas 2 žingsniui, tik be kreipimosi į masyvo elementus[8];
7. Žingsnis 6 kartojamas N kartų;
8. Paimamas galutinis palyginimo laikas T_{pg} ir apskaičiuojamas sunaudotas laikas – skirtumas $T_p = T_{pg} - T_{pp}$;
9. Rezultatas yra skirtumas $T_b - T_p$.

Aukščiau aprašytas algoritmas realizuotas dviem būdais – panaudojant ParSol ir statinius C/C++ masyvus (pakeičiant ParSol masyvus `Arr(i,j)` į C++ `Arr[i][j]`). Bandymai buvo atliekami, kompiliuojant programą su `-O3` optimizavimo raktu ir be jokio optimizavimo. Skaitymai ir rašymai visais atvejais buvo kartojami po 250 kartų. Bandymo su optimizavimu rezultatai pateikiami 2.1 lentelėje, be optimizavimo – 2.2 lentelėje.

2.1 lentelė. *ParSol masyvų elemento pasiekimo greitis, esant* `-O3` *optimizavimui*

	Skaitymas (10^{-9} s)		Rašymas (10^{-9} s)	
Dim	ParSol	C++	ParSol	C++
1D	6,75	6,32	6,29	6,24
2D	7,50	4,83	6,29	5,83
3D	9,22	5,23	6,26	5,88

[7] Priklausomai nuo bandymo tai gali būti skaitymas arba rašymas.

[8] Tam, kad optimizavimo eigoje kompiliatorius neišmestų „nebereikalingo" kodo, 2 ir 6 žingsniuose yra specialus papildomas kodas.

2.2 lentelė. *ParSol masyvų elemento pasiekimo greitis, nenaudojant optimizavimo*

	Skaitymas (10^{-9} s)		Rašymas (10^{-9} s)	
Dim	ParSol	C++	ParSol	C++
1D	16,6	10,0	8,49	6,16
2D	21,8	10,8	14,6	6,10
3D	24,9	12,5	16,6	6,69

Matome, kad `-O3` optimizavimo atveju bibliotekos masyvų skaitymas yra lėtesnis už standartinę C/C++ realizaciją tik 7–76 %, o rašymas – 1–8 %. Kai programa kompiliuojama be optimizavimo, masyvų skaitymas yra lėtesnis už standartinę C/C++ realizaciją tik 66–102 %, o rašymas – 37–148 %. Programos, naudojančios sukurtą technologiją, optimizavimas davė 145–191 % pagreitėjimą skaitymui ir 35–165 % rašymui. Palyginimui, programos, naudojančios standartinius C/C++ masyvus, optimizavimas davė tik 58–139 % pagreitėjimą skaitymui ir iki 14 % rašymui.

Būtina atkreipti dėmesį į elemento pasiekimo greičio matavimo sudėtingumą, kuris atsiranda dėl to, kad elemento pasiekimas yra labai greita operacija. Dėl to jos greičio matavimui gali daryti įtaką daugelis veiksmų, tokių kaip pagalbinio kodo (ciklų organizavimas, laikų skaičiavimas ir kiti) kiekis ir tai, kaip šis kodas paverčiamas į mašininį. Tai įneša didelę, sunkiai įvertinamą matavimų paklaidą. Todėl negalima teigti, kad pateikti eksperimentiniai rezultatai yra *tikslūs* operacijų greičių įverčiai.

Lyginant programų, kurių realizacija skiriasi tik panaudotais masyvais, vykdymo laikus, galima teigti, kad:

- Masyvų elementų pasiekimo greitis ParSol masyvuose yra tos pačios eilės kaip ir naudojant C/C++ masyvus.
- Naudojant kompiliatorių optimizavimą, ParSol masyvų (ir, atitinkamai, visos programos) veikimo greitis gali būti padidintas daugiau, nei panaudojant C/C++ masyvus.

2.5.2. Išraiškų šablonų veikimo greičio įvertinimas

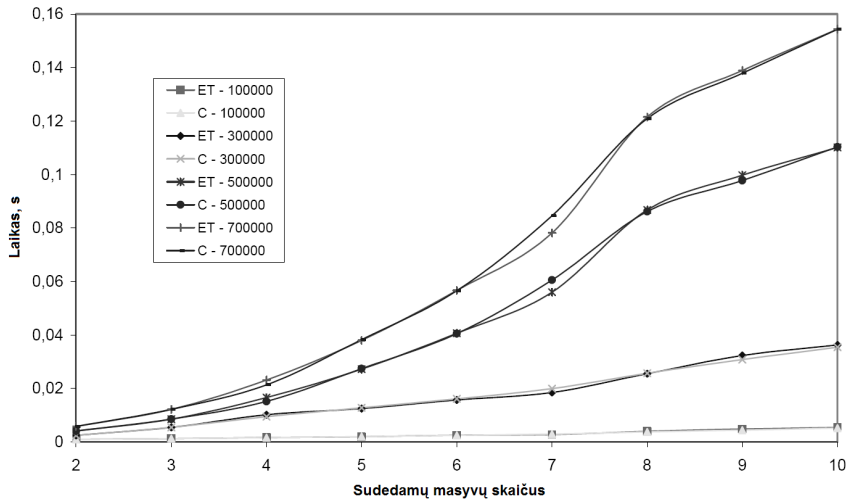
Buvo tikrinama vektorių sudėties operacija. Tie patys vektoriai buvo sudedami naudojant arba IŠ (`A=B+C+...;`) arba *C kalbos* `for` ciklą:

Masyvų elementų sudėtis standartinėmis C/C++ priemonėmis

```
1 for( int i = 0; i < VecSize; i++ )
2   A(i)=B(i)+C(i)+...;
```

Buvo naudojami dvimačiai `CmVector_2D` vektoriai, jų elementų tipas – `double`. Eksperimento metu buvo imami skirtingų dydžių vektoriai, taip pat keitėsi ir sudedamų vektorių skaičius. Eksperimentas buvo atliekamas tokiomis pat sąlygomis, kaip ir kitas eksperimentas, aprašytas 2.5.1 poskyryje, buvo naudojamas didžiausias `-O3` optimizavimas. Buvo matuojamas skaičiavimo laikas, panaudojant *user time* parametrą, grąžinamą `time` funkcijos. Kiekvienas bandymas buvo kartojamas 10 000 kartų, ir apskaičiuojamas vidurkis.

Bandymų rezultatai pavaizduoti 2.13 paveiksle. Pavaizduota skaičiavimo laiko priklausomybė nuo sudedamų masyvų kiekio keliems skirtingiems masyvų dydžiams. Masyvo dydis tiesiogiai įtakoja skaičiavimo laikus, tačiau skirtumas tarp sudėties metodų (C stiliaus ar panaudojant išraiškų šablonus) dažnai yra paklaidų ribose. Blogiausiu atveju buvo pastebėtas 9,5 % IŠ efektyvumo praradimas, vidutinis praradimas yra apie 2,6 %.



2.13 pav. *Palyginimas ParSol IŠ veikimo greičio su C stiliaus kodu*

Tai leidžia teigti, kad operacijų tarp vektorių realizacija taikant išraiškų šablonus yra arti optimaliai. Palyginimai tarp IŠ ir klasikinės operacijų realizacijos, kai naudojami papildomi masyvai tarpiniams rezultatams saugoti, buvo atlikti [22, 92] darbuose, ir parodo, kad IŠ realizacija yra $\approx 2{,}5 - 4$ kartus greitesnė, jau nekalbant apie kelis kartus mažesnį operatyvios atminties sunaudojimą.

2.6. Technologijos tolesnio vystymo galimybės

Pasiūlyta technologija ir jos realizacija, biblioteka ParSol, leidžia spręsti ir lygiagretinti platų uždavinių spektrą, tačiau tai nereiškia, kad ji nebus tobulinama ir vystoma. Jos tobulinimas gali vykti tokiomis kryptimis:

Klaidų taisymas. Nors biblioteka jau sėkmingai naudojama daugelio uždavinių realizacijai ir lygiagretinimui, negalima garantuoti, kad joje nebeliko klaidų. Didėjant bibliotekos naudotojų skaičiui, gali atsirasti situacijos, kompiliatoriai ir MPI realizacijos, kuriose biblioteka dar nebuvo išbandyta, ir gali pasireikšti iki šiol nepastebėtos klaidos. Tokios klaidos turi būti taisomos.

Optimizavimas. Bus siekiama, kad tas pats vartotojo programos kodas būtų vykdomas dar greičiau, naudotų dar mažiau resursų, būtų dar efektyviau lygiagretinamas.

Naujas funkcionalumas. Atsirandant naujiems uždaviniams, esant reikalui bus plečiamas ParSol bibliotekos funkcionalumas, siekiant praplėsti jos naudojimo ratą. Pridedant naują funkcionalumą, bus siekiama nenusižengti pagrindiniams bibliotekos kūrimo principams–jau esamo kodo efektyvumui ir naudojimo paprastumui.

Šiame poskyryje pateikiamos kai kurios sritys, kurios yra aktualiausios šiai dienai ir kuriomis vyks bibliotekos tobulinimas artimiausiu metu.

2.6.1. Nestruktūrinių tinklų palaikymas

Masyvai yra duomenų struktūra, patogi dekartinių tinklų (angl. *mesh*) realizacijai. Tačiau dekartiniai tinklai nevisada efektyviai aproksimuoja uždavinio apibrėžimo sritį. Todėl dažnai, sprendžiant realius uždavinius, naudojami nestruktūriniai tinklai. Nestruktūrinių tinklų palaikymas yra svarbus žingsnis, norint išplėsti uždavinių, kur gali būti taikoma biblioteka ParSol, klasę. Šiame skyrelyje aprašomi galimi šio funkcionalumo realizacijos aspektai.

A. Sąsajos su kitais paketais

Nestruktūriniuose tinkluose tinklo elementų tvarka ir sąryšiai (kaimyniniai elementai) turi būti nurodomi išreikštiniu būdu. Paprastai toks tinklas generuojamas atskirai, priklausomai nuo užduoties, panaudojant specialiai tam skirtas priemones, pavyzdžiui, ADMesh arba Bamg. ParSol yra lygiagretinimo biblioteka, kuri naudos jau sugeneruotus tinklus. Numatyta tinklų pakeitimo sąsaja, pakrau-

nanti tinklą į atmintį. Pridedant skirtingas šios sąsajos realizacijas, bus galima plėsti biblioteką, realizuojant skirtingų formatų tinklų palaikymą.

Skirtingai nuo dekartinio tinklo, nestruktūrinio tinklo padalinimas (angl. *partitioning*) yra sudėtingas procesas. Egzistuoja daug programinių paketų, skirtų šiai užduočiai spręsti. Tarp jų galima paminėti tokius, kaip METIS, Chaco[9] arba JOSTLE[10]. Planuojama ParSol bibliotekoje pasinaudoti jau egzistuojančiais įrankiais. Tai irgi planuojama daryti naudojant specialią sąsają, skirtingos realizacijos leis ParSol bibliotekoje naudoti skirtingus nestruktūrinių tinklų dalinimo įrankius.

B. Klasių struktūra

Kuriant ParSol versiją nestruktūriniams tinklams, bus laikomasi tos pačios ideologijos – vartojimo ir lygiagretinimo paprastumas. Tai bus pasiekta, realizuojant ParSol bibliotekoje dvi naujas klases – *Mesh* ir *UArray* (unstructured array). Šios klasės turės tiek nuosekliąją, tiek ir lygiagrečiąją versijas.

Mesh klasėje saugojama visa informacija apie tinklo topologiją. Programos paleidimo metu ši informacija gali būti pakraunama iš išorinio failo, naudojant atitinkamas tinklo pakeitimo sąsajas. Lygiagrečiojoje klasės versijoje bus saugojama tik ta bendro tinklo dalis, kuri priklauso šiam procesui, taip pat informacija apie elementų apsikeitimą su kaimynais. Savo daliai nustatyti lygiagrečioji Mesh versija naudos bendrą tinklelį kartu su duomenų skaidymo paketu per atitinkamą sąsają.

UArray klasė paskiria elementus (elementų duomenų tipas – šabloninis parametras, kaip ir ParSol masyvuose), tai pat pateikia priemones pasiekti kiekvieną elementą ir jo kaimynus. Kuriant UArray klasę, jai bus perduodamas būtinas Mesh tipo parametras, panašiai kaip dabar yra su ParSol masyvų ir šablonų atveju. Lygiagrečioji klasės versija taip pat realizuos duomenų apsikeitimą tarp kaimynų, panaudojant MPI standartą. Panašiai kaip ir ParSol masyvų atveju, bus realizuotos *UVector* tipo klasės, kurios, padarant prielaidą, kad vektorio elementų tipas palaiko tam tikrus matematinius veiksmus, pateiks vartotojui papildoma matematinį funkcionalumą (normas, išraiškų šablonus ir kitus).

C. Naudojimas

Algoritmų, naudojančių nestruktūrinius tinklus, išlygiagretinimas ParSol bibliotekoje bus atliekamas pagal tą pačią ideologiją, kaip tai daroma naudojant ParSol masyvus:

[9] Internetinė prieiga `http://www.cs.sandia.gov/~bahendr/chaco.html`, [2007-06-20].

[10] Internetinė prieiga `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`, [2007-06-20].

1. Vartotojas sukuria nuosekliąją programos versiją, naudodamas *Mesh* ir *UArray* arba *UVector* klases. Vykdymo metu Mesh klasė generuoja tinklo topologiją, kuri saugoma tik vieną kartą ir naudojama visuose masyvuose.
2. Vartotojas nuosekliojoje versijoje atlieka tokius pakeitimus:
   - pakeičia antraštinius failus;
   - pakeičia naudojamas Mesh ir UArray klases jų lygiagrečiaisiais analogais;
   - prideda iniciacijos/užbaigimo kodą;
   - prideda nurodymus apsikeisti duomenimis (jei dar nepridėta);
   - prideda topologijos klasę(-es).
3. Programa perkompiliuojama, naudojant MPI bibliotekas;
4. Lygiagrečioji versija paleidžiama kaip MPI programa.

2.6.2. Bibliotekos optimizavimas SMP tipo kompiuteriams

Plintant kompiuteriams su *multicore* tipo procesoriais, bibliotekos optimizavimas šiai architektūrai šiuo metu yra vienas iš prioritetinių uždavinių. Kaip jau buvo minėta, siekiant efektyviau išnaudoti SMP tipo kompiuterių resursus, bibliotekoje turi būti realizuotas daugiagijinio vykdymo palaikymas. Šiuo metu tam egzistuoja visa eilė daugiagijinio programavimo priemonių [43], iš jų kai kurios, pavyzdžiui, OpenMP, leidžia efektyviai lygiagretinti lygiagrečiųjų duomenų algoritmus.

Tačiau vien gijų panaudojimas neužtikrina optimalaus lygiagretinimo efektyvumo. Problemos esmė yra tame, kad, nors kiekviena gija gali turėti savo CPU, lieka daug kitų bendrų resursų. Ypač šiame kontekste reikėtų paminėti kompiuterio atmintį, tiek operatyviąją (RAM), tiek ir spartinančiąją (angl. *cache*). Esant šiuolaikinių CPU spartoms, duomenų apsikeitimas tarp procesoriaus ir operatyviosios atminties yra labai lėta operacija, lyginant su operacijomis, kurioms reikalingi tik vidiniai CPU resursai. SMP tipo kompiuteriams ši problema yra dar aktualesnė, nes, kadangi į RAM per magistralę vienu metu gali kreiptis tik vienas CPU[11], kreipimasis į RAM gali tapti ta siaura vieta, mažinančia lygiagretinimo efektyvumą.

Kreipimosi į RAM poveikis sumažėja, kai naudojama CPU spartinančioji atmintis. Spartinančioji atmintis dažniausiai susideda iš kelių lygių, vadinamų L1, L2, L3. Kuo aukštesnis lygis, tuo didesnė jo talpa, tačiau lėtesnis priėjimas. Šiuo-

[11] Ateityje ši situacija gali pasikeisti. Pavyzdžiui, kiekvienas CPU gali turėti savo atminties valdiklį. Tačiau tuo atveju atsiras kitos optimizavimo problemos, nes priėjimas prie skirtingų RAM vietų taps nevienodas (NUMA).

laikiniuose kompiuteriuose L1 atminties talpa yra keliasdešimt kilobaitų, tuo tarpu L3 lygio spartinančiosios atminties talpa sudaro kelis megabaitus. Tačiau šiuolaikiniuose CPU spartinančioji atmintis yra nematoma programuotojui, ir valdoma automatiškai procesoriaus lygyje. Žinant metodus, kaip CPU naudoja spartinančiąją atmintį, galima atitinkamai modifikuoti savo algoritmus, kad tuo galima būtų pasinaudoti [90]. Tokio tipo optimizavimas yra naudingas ir vykdant algoritmus kompiuteriuose su vienu branduoliu.

Atsiradus multicore tipo procesoriams, efektyvus spartinančiosios atminties panaudojimas tampa dar sudėtingesnis, bet ir atveria naujas optimizavimo galimybes. Taip yra todėl, kad šioje architektūroje keli branduoliai turi ne tik bendrą priėjimą prie RAM atminties, bet ir bendrą spartinančiosios atminties dalį. Pastebima tendencija, kad branduolių (angl. *core*) sudėtingumas ir kiekis didėja sparčiau, nei *cache* atminties talpa. Tai reiškia, kad, nors bendras spartinančiosios atminties kiekis ateities procesoriuose didės, jos dalis vienam branduoliui gali net mažėti.

Iš vienos pusės, ši architektūra tam tikrais atvejais leidžia efektyviau panaudoti spartinančiąją atmintį (pavyzdžiui, kai du branduoliai skaito tą pačią atminties sritį), tačiau atsiranda ir naujos, kitoms architektūroms nebūdingos problemos, pavyzdžiui, klaidingas spartinančiosios atminties pasidalinimas (angl. *false sharing*) [43, 200 p.].

Apibendrinant ankščiau pateiktus argumentus, bibliotekos ParSol optimizavimas *multicore* tipo sistemoms galėtų susidėti iš tokių žingsnių:

1. Daugiagijinio programavimo palaikymas tiek nuosekliojoje, tiek ir lygiagrečiojoje bibliotekos versijoje. Tinkamas kandidatas lygiagrečiųjų duomenų daugiagijinio programavimo realizacijai yra populiarus OpenMP standartas. Panašiai kaip ir HPF atveju, didelė lygiagretinimo nurodymų dalis gali būti surašyta tik OpenMP kompiliatoriui suprantamose direktyvose. Tai leidžia turėti tą patį bibliotekos kodą, kuris, priklausomai nuo kompiliatoriaus, sugeneruos viena- arba daugiagijinę programą.
2. Optimizavimas, susijęs su efektyvesniu spartinančiosios atminties naudojimu. Šiam etapui priklausytų galimi duomenų struktūrų pakeitimai, taip pat pakeitimai įvairių ciklų realizacijoje.
3. Optimizavimas, specifinis *multicore* tipo procesoriams. Čia galėtų būti, pavyzdžiui, specifiniai duomenų struktūrų pakeitimai, siekiant atskirti duomenų dalis, kurias apdoroja skirtingos gijos.

2.6.3. Statinis krūvio balansavimas

Matematinių objektų bibliotekos ParSol lygiagrečiojoje versijoje iki šiol nebuvo atliekamas *krūvio balansavimas*, o krūvis buvo dalinamas tarp procesų lygiomis dalimis. Toks sprendimas pasiteisina, kai uždaviniai sprendžiami homogeniniuose klasteriuose, ir tuo metu nevykdomas joks kitas uždavinys. Tačiau šis sprendimas riboja bibliotekos taikymo galimybes, nes, kaip yra žinoma, klasterio homogeniškumas yra lengvai pažeidžiamas, o šios savybės išlaikymas yra brangus [7].

Šiame poskyryje nagrinėjamos galimybės realizuoti ParSol bibliotekoje krūvių balansavimą. Pasaulyje egzistuoja įvairūs krūvio balansavimo metodai, kurie taikomi skirtingo tipo uždaviniams [5, 34, 74]. Krūvio balansavimo problemos taip pat buvo nagrinėjamos ir disertacijose [35, 36].

Siekiama, kad krūvių balansavimas ParSol bibliotekoje atitiktų bibliotekos ideologiją – galbūt tiktų siauresniam uždavinių ratui, tačiau veiktų greitai, patikimai ir reikalautų iš vartotojo minimalių pastangų. Tai daro įtaką reikalavimams, kurie buvo apibrėžti siekiamam tikslui:

- Nagrinėjamas *statinis* krūvių balansavimas, kai uždavinys padalinamas vieną kartą sprendimo pradžioje. Lygiagrečiųjų duomenų uždaviniams šis būdas tinka, kai procesų sparta išlieka pastovi sprendimo metu.

  Užtikrinti tai yra nesudėtinga – pavyzdžiui, kai kiekvienas procesas paleidžiamas atskirame kompiuteryje, reikia garantuoti, kad šitie kompiuteriai sprendimo metu neturėtų jokių kitų uždavinių. Tačiau mes gauname daug privalumų, tarp jų paprastesnis ir greitesnis kodas, mažiau duomenų apsikeitimų, kas ypač aktualu kompiuterių klasteriams, kuriuose ir taikoma ši biblioteka.

- Vartotojas turi pateikti bibliotekai proceso spartos įvertinimą. Kompiuteris turi daug parametrų, galinčių įtakoti uždavinio sprendimo spartą: CPU architektūra, dažnis, spartinančiosios atminties dydis, RAM dydis, pagrindinės magistralės greitis ir pralaidumas, ir kiti. Vienus uždavinius labiau riboja vieni parametrai, kitus – kiti. Dėl visų šitų priežasčių, būtina proceso spartą vertinti sprendžiamo uždavinio atžvilgiu.

  ParSol bibliotekai vartotojas pateikia informaciją, kurios pati biblioteka nustatyti negali. Vienas iš pavyzdžių – kai apibrėžiami šablonai (angl. *stencils*). Kadangi automatiškai nustatyti programos algoritmą (ir, atitinkamai, atlikti teisingus spartos matavimus) biblioteka negali, tai yra vartotojo atsakomybė. Kaip ir šablonų atveju, stengiamasi, kad vartotojas tam galėtų sugaišti kuo mažiau pastangų.

- Ignoruojami komunikavimo kaštai. Tai yra griežtas reikalavimas, ir jis

reiškia, kad toks balansavimas bus efektingas uždaviniams, kurių skaičiavimo laikas gerokai viršija duomenų apsikeitimų laiką. Mažėjant uždavinio skaičiavimo ir duomenų persiuntimo laikų santykiui, mažės ir balansavimo efektyvumas.

Tačiau, darydami tokias prielaidas, mes laimime paprastumą vartotojo atžvilgiu. Kaip matysis iš žemiau pateiktų skaičiavimų, operuojant vien procesų sparta, ji gali būti matuojama bet kokiais (vienodais) vienetais, nes rezultatas priklauso tik nuo *spartų santykių*. Priešingu atveju, vartotojui tektų *pačiam* skaičiuoti tiek spartos, tiek ir duomenų apsikeitimo greičio koeficientus, ir skaičiavimo metodika taptų žymiai sudėtingesnė.

Toliau aprašomos galimos šių reikalavimų realizacijos ParSol bibliotekoje. Čia ir toliau mes laikome, kad skaičiavimo apimtis yra proporcinga masyvo elementų (lygiagrečiu atveju – tenkančių vienam procesui) skaičiui. Taigi, krūvio balansavimas mūsų atveju susiveda prie optimalaus masyvo elementų paskirstymo tarp procesų.

A. Vienmatis atvejis

Mūsų užduotis – paskirstyti ParSol masyvus tarp procesorių taip, kad būtų kuo mažesnis skirtumas tarp laikų, kuriuos kiekvienas procesorius sunaudoja uždavinio sprendimui. Šį uždavinį pradžioje panagrinėsime paprasčiausiu, vienmačiu atveju, o paskui išnagrinėsime atvejį, kai uždavinio sprendimui naudojamos daugiamatės procesorių topologijos.

Tegu N yra dydis masyvo, kuris dalinamas tarp p procesų. Kiekvienas procesas gauna masyvo dalį N_i ($i = 1, 2, \ldots, p$), taip kad galioja lygtis

$$N = \sum_{i=1}^{p} N_i \,. \tag{2.7}$$

Tegu T_i yra laikas, kurį i-asis procesas sugaištų viso uždavinio sprendimui, tada apibrėžiame sprendimo laiką vienam duomenų vienetui τ_i:

$$\tau_i = \frac{T_i}{N_i} \,. \tag{2.8}$$

Lygiagrečiųjų duomenų algoritmų, kuriems taikoma ParSol biblioteka, viena iš savybių yra ta, kad tam pačiam procesui, jei nesikeičia kiti veiksniai (pavyzdžiui, kompiuterio, kuriame tas procesas paleistas, apkrovimas), τ_i turi praktiškai nepriklausyti nuo duomenų kiekio N_i. Tada mūsų užduotis gali būti suformuluota tokia lygtimi:

$$N_1\tau_1 = N_2\tau_2 = \cdots = N_i\tau_i = \cdots = N_p\tau_p \,. \tag{2.9}$$

Mes išspręsime užduotį, jei surasime visus N_i, kurių yra p. Lygtyje (2.9) iš tikrųjų yra $(p-1)$ nepriklausomų lygčių, ir kartu su (2.7) lygtimi tai sudaro pilną lygčių sistemą. Šios sistemos sprendinys yra toks:

$$N_i = N\frac{\frac{1}{\tau_i}}{\sum_{j=1}^{p}\frac{1}{\tau_j}}\,. \tag{2.10}$$

Nesunku pastebėti, kad, jei mes turime homogeninį kompiuterių klasterį, kur $\tau_i = const$, tai optimalus sprendinys yra $\frac{N}{p}$, taigi daliname masyvą lygiomis dalimis tarp procesų.

B. Vienmačio atvejo naudingumas

Pasinaudojus ankščiau gautomis formulėmis, išnagrinėkime statinio krūvio balansavimo naudingumą. Paimkime realų atvejį, kai turime homogeninį kompiuterių klasterį iš p kompiuterių, kur kiekvienas kompiuteris atlieka operacijas su vienu masyvo elementu per laiką τ. Tada, idealiai išlygiagretinus dydžio N uždavinį, jį išspręstume per laiką $N\tau/p$. Pagal Muro dėsnį kompiuterių pajėgumai dvigubėja apytiksliai kas 1,5 metų, taigi, po kurio laiko teks spręsti klasterio atnaujinimo klausimą. Tarkime, kad tuo metu kompiuteriai taps dvigubai greitesni, ir vienam masyvo elementui sugaiš laiką $\tau/2$.

Nenaudojant krūvio balansavimo, norint spręsti mūsų uždavinį dvigubai greičiau, reikės papildomai nupirkti p naujų kompiuterių. Spręsdami uždavinį vien su naujais dvigubai greitesniais kompiuteriais, akivaizdu, kad pasieksime dvigubo pagreitėjimo. Jei mes bandysime spręsti uždavinį, naudodami *visus* esamus kompiuterius, tai bendras pagreitėjimas palyginus su senu klasteriu vis tiek išliks tik du kartus, nes, nors duomenų kiekis kiekvienam kompiuteriui sumažės perpus, nuo N/p iki $N/(2p)$, bendras uždavinio sprendimo laikas, skaičiuojamas pagal lėčiausią kompiuterį, išliks $N\tau/(2p)$. Vadinasi, nupirkus naujus kompiuterius, senų klasterio kompiuterių panaudojimas nebetenka prasmės.

Situacija iš esmės keičiasi, kai yra statinis krūvio balansavimas. Šiuo atveju, norint pasiekti dvigubo pagreitėjimo, užteks nupirkti *dvigubai* mažiau naujų kompiuterių, arba $p/2$. Tuo galime įsitikinti iš (2.10) formulės. Nauji kompiuteriai gaus po N/p elementų, senieji – $N/(2p)$, o programos vykdymo laikas visuose kompiuteriuose taps vienodas ir lygus $N\tau/(2p)$. Jei nupirksime p naujų kompiuterių, tai, atlikus analogiškus skaičiavimus, matome, kad su tokiu klasteriu uždavinį išspręstume per laiką $N\tau/(3p)$.

Taigi, matome, kad krūvio balansavimas

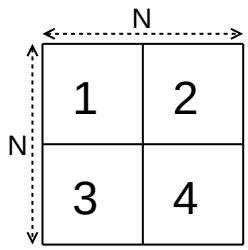
- leidžia prailginti senos technikos naudojimo terminus;
- leidžia sumažinti kompiuterių klasterių atnaujinimo kaštus;

- leidžia pagreitinti uždavinių sprendimą nehomogeniniuose klasteriuose.

C. Daugiamatis atvejis

Daugiamačiu atveju situacija yra sudėtingesnė. Iš vienos pusės, neįmanoma išvesti statinio krūvio balansavimo formulės, panašios į (2.10) formulę todėl, kad dabar nežinomųjų kiekis yra mažesnis, negu lygčių kiekis. Šiuo atveju mes turime optimizavimo uždavinį, kuriam reikia taikyti euristinius metodus, ieškančius optimalaus krūvio pasiskirstymo. Tokių metodų gali būti daug. Vienas iš metodų galėtų būti iteracinis (2.10) lygties taikymas pagal kiekvieną dimensiją atskirai tol, kol nenusistovės optimalus pasiskirstymas. Galiausiai jeigu iškart būtų atmesti neperspektyvūs variantai, galima būtų tiesiog perrinkti visus įmanomus pasiskirstymus, siekiant išrinkti geriausią. Tačiau ar geriausio daugiamačio krūvio pasiskirstymo ieškojimas turi prasmės?

Pavyzdžiui, panagrinėkime 2×2 kompiuterių topologiją. Tarp šių procesų padalintas $N \times N$ elementų masyvas, kaip parodyta 2.14 paveiksle.



2.14 pav. *Masyvo padalinimo 2×2 procesų topologijoje pavyzdys*

Tegu vienas kompiuteris yra *dvigubai* greitesnis už kitus, apdorojančius vieną elementą per laiką τ. Paprastas visų įmanomų padalinimų perrinkimas parodo, kad optimalus yra variantas, kai greičiausias procesas gauna $\frac{2}{3}N \times \frac{2}{3}N$ elementų. Tokiu atveju uždavinys bus išspręstas per laiką $2N^2\tau/9$, kas yra tik 11 % greičiau už laiką $N^2\tau/4$, per kurį šis uždavinys būtų išspręstas, jeigu būtų padalintas tarp procesų lygiomis dalimis. Palyginimui, vienmačiam atvejui vieno kompiuterio iš keturių greičio padvigubėjimas duotų sprendimo laiką $N^2\tau/5$, arba 20 % greičiau.

Toks krūvio balansavimo efektyvumo sumažėjimas susijęs su tuo, kad daugiamatėse topologijose krūviai, skiriami skirtingiems procesams, yra stipriai susiję. Mūsų atveju, tarkime, kad greitesnis yra 1 procesas. Norint išspręsti užduotį greičiau, reikia padidinti jo apkrovą, tarkim, x kryptimi, tačiau tai padidins apkrovą ir 3 procesui, todėl uždavinio sprendimo laikas pailgės.

Spręsti realius daugiamačius uždavinius nehomogeniniuose klasteriuose sudėtingiau. Tai reiškia, kad

- Išrinkti optimalų variantą gali būti labai sudėtinga (pavyzdžiui, perrenkant visus įmanomus $15\,000 \times 15\,000$ elementų masyvo padalinimus mūsų atveju 2,4 GHz 64 bitų procesorius užtruko apie 2 s), arba bus išrenkamas ne optimalus variantas.
- Net ir optimalus krūvio balansavimo variantas gali duoti *nedidelį* pagreitėjimą lyginant su vienodu krūvio padalinimu tarp procesų.

Iš šios analizės galima padaryti išvadą, kad statinio krūvio balansavimo realizacija daugiamačiu atveju neduotų laukiamo efektyvumo. Norint taikyti krūvio balansavimą daugiamačiams uždaviniams, rekomenduojama paskirstyti jį vienmatėje topologijoje pagal vieną iš dimensijų, ir taikyti vienmatį krūvio balansavimą.

D. Galima realizacija *ParSol* bibliotekoje

Vienas iš ParSol bibliotekos kūrimo tikslų – ji turi būti patogi ir paprasta naudoti. Tai liečia visus jos aspektus, tarp jų ir krūvio balansavimą. Minimalios vartotojo pastangos lygiagrečiųjų duomenų programos atveju yra įvertinti procesoriaus spartą, sprendžiant *jo* uždavinį. Kaip galime matyti iš (2.10) formulės, vartotojui užtektų nustatyti bendrą sugaištą laiką ir net nėra būtina skaičiuoti laiką vienam duomenų vienetui, nes formulės priklauso nuo *laikų santykių*.

Taigi bendras krūvio balansavimo algoritmas ParSol bibliotekos vartotojui galėtų būti toks:

1. Lygiagrečiojoje programoje kiekviename procese išspręsti fiksuoto dydžio uždavinį ir įvertinti sprendimo laiką. MPI biblioteka suteikia galimybes tiksliai išmatuoti laiką.
2. Šiuos laikus kiekvienas procesas pateikia savo ParSol *topologijos* klasei *prieš kuriant masyvus*.
3. Masyvo sukūrimo metu duomenis paskirstomi pagal ankščiau aprašytą metodą, atsižvelgiant į procesorių našumus.

2.7. Antrojo skyriaus rezultatai ir išvados

Šiame skyriuje pateikiama nauja skaičiavimo matematikos algoritmų lygiagretinimo technologija ir jos realizacija. Buvo pasiekti tokie *rezultatai*:

1. Remiantis 1 skyriuje atlikta analize, pasiūlyti diferencialinių lygčių sprendimo algoritmų lygiagretinimo naujos technologijos principai. Joje naudojami lygiagrečiųjų duomenų (lygiagrečiosios duomenų struktūros ir opera-

cijos su jais) ir globalios atminties (globali adresacija ir kai kurių kaimyninių duomenų skaidrus pasiekimas) lygiagretaus programavimo modelių elementai. Vartotojas neišreikštiniu būdu, apibrėždamas aproksimavimo šabloną, nurodo, kokiais duomenimis turi būti apsikeičiama, taip pat jis jau išreikštiniu būdu nurodo, kada šis pasikeitimas turi vykti.

2. Sukurta biblioteka, realizuojanti 2.2 poskyryje pasiūlytą naują lygiagretinimo technologiją. Biblioteka sukurta naudojant standartines C++ kalbos priemones, lygiagretinimui naudojamas MPI standartas. Naudojamos tokios C++ kalbos priemonės, kaip OOP, šabloninis metaprogramavimas, operatorių perkrovimas ir kitos. Bibliotekos apimtis yra apie 20 000 eilučių, dar apie 10 000 eilučių sudaro bibliotekos tikrinimai testais, realizuota daugiau kaip 70 įvairios paskirties klasių. Biblioteka susideda iš tokių skaičiavimo matematikos objektų, kaip masyvai, vektoriai, pilnos ir išretintos matricos. Taip pat realizuotos operacijos su jais – normų skaičiavimas, matricų ir vektorių sandauga, ILU faktorizacija, matricų bei vektorių išraiškos. Egzistuoja tiek nuosekliosios, tiek lygiagrečiosios objektų bei operacijų versijos.

3. Patikrintas bibliotekos efektyvumas. Buvo tiriamas vienos iš bibliotekos pagrindinių klasių – masyvų realizacijos efektyvumas lyginant su standartiniais analogais, C/C++ masyvais. Buvo lyginamas masyvo elementų pasiekimo efektyvumas. Taip pat buvo lyginamas globalių skaičiavimo operacijų tarp masyvų realizacijos bibliotekoje (naudojant išraiškų šablonus) ir standartinėmis C/C++ priemonėmis efektyvumas.

4. Numatytas tolesnis bibliotekos tobulinimas, kurio pagrindinės kryptys yra bibliotekos optimizavimas *multicore* tipo kompiuteriams, nestruktūrinių tinklų palaikymas bei krūvio balansavimo (pradžioje – statinio) galimybės. Jomis numatytas detalus vystymo planas, nuo naujų savybių vietos bendroje bibliotekos struktūroje iki tam tikrų realizacijos ypatumų.

Atsižvelgus į pasiektus rezultatus, galima *teigti*, kad

1. Nauja technologija *skirta* tiesinės algebros uždavinių, atsirandančių sprendžiant diferencialines lygtis ir jų sistemas skaitiniais metodais, realizacijai ir pusiau automatiniam lygiagretinimui.

2. Plačiai naudojami lygiagrečiųjų duomenų ir globalios atminties modeliai turi trūkumų, dėl kurių šie modeliai tiesiogiai neatitinka visų iškeltų reikalavimų. Pasiūlyti naujos technologijos principai leidžia *įveikti* šiuos *trūkumus*. Naujoji technologija leidžia spręsti platesnę uždavinių klasę, nei lygiagrečiųjų duomenų modelio realizacijos, *neprarandant* pusiau automatinio lygiagretinimo galimybės. Palyginti su globalios atminties modeliu,

nauja technologija nėra tokia bendra, užtat galimas persiunčiamų duomenų *optimizavimas*. Kadangi vartotojui nesudėtinga pačiam nurodyti, kada, kaip ir kokiais duomenimis turi būti apsikeičiama, galima pasiekti didesnio, palyginti su minėtais lygiagretinimo modeliais ir jų realizacijomis, lygiagretinimo *efektyvumo*.

3. Perėjimas nuo standartinių C/C++ priemonių prie naujos technologijos atliekamas remiantis aiškia, formalizuota schema. Analizuojamame uždavinyje pereiti prie nuosekliosios bibliotekos versijos prireikė tik 6 žingsnių. Dalis žingsnių yra pakeitimai tik vienoje programos vietoje, kitiems reikėjo globalių, tačiau tokio pat tipo pakeitimų (pvz., iš `a[i][j]` į `a(i,j)`), kuriuos galima lengvai automatizuoti. Lygiagretinimas taikant naują technologiją iš vartotojo reikalauja tik 5 analogiškų žingsnių. Programa išlaiko savo buvusią vartotojui pažįstamą struktūrą.

4. Technologijos realizacija gali būti *naudojama* įvairiose *terpėse*. Šiuo metu ji išbandyta MS-Windows, Linux ir AIX operacinėse sistemose, naudojant MSVC++ ir g++ kompiliatorius bei MPICH, LAM/MPI ir IBM MPI realizacijas.

5. Nustatyta, kad blogiausiu atveju bibliotekos masyvų elementų pasiekimo greitis nusileidžia standartiniams C/C++ masyvams 2,5 karto, geriausiu atveju skirtumas yra tik 1–8 %. Globalių masyvų operacijų realizacija taikant išraiškų šablonus, palyginti su standartine realizacija naudojant `for`-ciklus, lemia tik apie 2,6 % efektyvumo pararadimą. Taigi galime teigti, kad biblioteka leidžia *efektyviai* realizuoti tiesinės algebros algoritmus, neatsisakant C++ kalbos privalumų.

6. Rekomenduojama biblioteką naudoti su naujausiais šiuolaikiniais kompiliatoriais. Pirmiausia, tai užtikrins, kad bus palaikomos visos standartinės C++ galimybės, kurios naudojamos technologijai realizuoti. Taip pat technologijos realizacijai buvo taikomi šabloninio metaprogramavimo metodai, efektyviai išnaudojantys šiuolaikinių kompiliatorių optimizavimo galimybes. Kaip parodė eksperimentų rezultatai, optimizavimas kompiliatoriaus lygyje bibliotekos panaudojimo efektyvumą pagerina ≈2,5 karto, standartiniams C/C++ masyvams – ≈1,5 karto.

7. Svarbiausios šiuo metu technologijos vystymo kryptys, tokios kaip optimizavimas SMP tipo kompiuteriams, nestruktūrinių tinklų palaikymas ir krūvio balansavimas, gali būti realizuotos iš esmės *nekeičiant* sukurtos bibliotekos *struktūros*.

3

Lygiagretinimo technologijos taikymai

3.1. Įvadas

Šiame skyriuje aprašomi sukurtos lygiagretinimo technologijos taikymai. *Parenkant* uždavinius, buvo siekiama kelių tikslų:

- Tai turėjo būti šiuolaikiniai aktualūs uždaviniai, kurių lygiagretinimas turėtų praktinę naudą.
- Tai turėjo būti uždaviniai, priklausantys sukurtos technologijos lygiagretinamų uždavinių klasei – diferencialinių lygčių sprendimo įrankiai arba tiesinės algebros uždaviniai, iškylantys spendžiant diferencialines lygtis skaitiniais metodais.

Taikant naują technologiją uždavinių realizacijai ir lygiagretinimui, buvo siekiama patikrinti, ar realizuota technologija leidžia realizuoti ir išlygiagretinti iškilusias problemas, taip pat įvertinti lygiagretinimo *efektyvumą*. Siekiant įvertinti lygiagretinimo efektyvumą, buvo atliekama teorinė algoritmų *išplečiamumo analizė* konkrečiai skaičiavimo aplinkai (kompiuterių klasteris, SP4 superkompiuteris ir t. t.). Gavus teorinius „geriausius“ įverčius, buvo atliekami naujos technologijos paremto algoritmo lygiagretinimo efektyvumo matavimai ir lyginama su teorija.

Toliau pateikti trys naujos technologijos taikymai: daugiafazio skysčių tekėjimo poringose terpėse modeliavimas (susiję rezultatai skelbti [91, 93, 96, 99,

100] darbuose), vaizdų glodinimas taikant netiesinius difuzinius filtrus [94, 95, 102] ir variaciniai iteraciniai tiesinių lygčių sistemų sprendimo algoritmai [97].

3.2. Lygiagretusis įrankis, skirtas tekėjimams poringose terpėse modeliuoti

3.2.1. Įvadas

Pastaruoju metu daugiafaziam tekėjimui ir šilumos pernešimui poringose terpėse yra skiriama labai daug dėmesio. Taip yra todėl, kad yra daugybė uždavinių, kur yra sutinkami šitie reiškiniai. Galima nurodyti tokius pavyzdžius, kaip poringų medžiagų (pvz., medienos) džiovinimas arba prisotinimas [39, 69], žemės užteršimas ir jos valymas [44], naftos gavyba [82], radioaktyviųjų atliekų saugojimas [3], popieriaus gamyba [38], biologija (įvairių audinių veikla), filtrų konstravimas ir t.t.

Šiuo metu ypač populiarus skaitinis procesų modeliavimas naudojant kompiuterius, kaip vienas greičiausių ir ekonomiškai efektyviausių tyrimo priemonių. Tačiau, modeliuojant procesus poringose terpėse, tenka susidurti su sudėtingais pernešimo reiškiniais, dėl kurių daugiafazis tekėjimas ir šilumos pernešimas yra sunkiai analitiškai analizuojami. Paskutiniais metais daug pastangų dedama, norint sukurti fundamentinius šių reiškinių matematinius modelius.

Modeliavimas atliekamas, skaitiškai sprendžiant diferencialinių lygčių, aprašančių procesus poringose terpėse, sistemas. Nors dauguma modelių gaunama, atsižvelgiant į tuos pačius gamtos dėsnius, matematinė jų išraiška gali skirtis, kaip gali skirtis ir kai kurių koeficientų (sistemos ir fazių savybių) interpretacija ir formulavimas. Diferencialinės lygtys tokiose sistemose yra netiesinės ir stipriai susietos. Lygčių sudėtingumas sparčiai didėja, didėjant sprendžiamo uždavinio dimensijų skaičiui, dalyvaujančių fazių skaičiui, atsižvelgiant į fazines komponentes, norint apskaičiuoti daugiau sistemos parametrų.

Skaitinio modelio sudėtingumas lemia tai, kad nėra vieno universalaus metodo, tinkančio visiems poringų terpių uždaviniams. Parenkant metodą, tenka atsižvelgti į metodo universalumą, skaičiavimo greitį ir į tai, kiek atminties išnaudojama, gaunamo rezultato tikslumą. Dažnai universalūs metodai nėra optimalūs, o metodai, kurie efektyviai modeliuoja tam tikros klasės poringą terpę, yra sunkiai pritaikomi kitiems uždaviniams. Dėl to pasaulyje egzistuoja nemažai įrankių, skirtų tokio tipo uždaviniams spręsti. Dažniausiai kiekviename įrankyje realizuojamas savo modelis, su jo privalumais ir trūkumais. Vienas iš šio darbo tikslų buvo tęsti jau pradėtą darbą ir išlygiagretinti jau sukurtą procesų poringose terpėse modeliavimo įrankį MFSolver [93, 100].

Šiomis dienomis sprendžiamų uždavinių apimtys reikalauja diskretizacijos tinklų, susidedančių iš milijonų elementų. Esant tokioms uždavinių apimtims, lygiagrečiųjų kompiuterių panaudojimas tampa privalomas, norint pasiekti optimalų sprendimo greitį ir spręsti atitinkamo dydžio uždavinius. Todėl kuriamo įrankio lygiagretinimas yra vienas svarbiausių uždavinių, siekiant užtikrinti, kad šiuo įrankiu galėtume modeliuoti realaus sudėtingumo procesus. Žinoma, įrankio vystymas neapsiriboja vien lygiagretinimu, taip pat išplečiamas modelis ir naudojami nauji tikslesni diskretizavimo metodai.

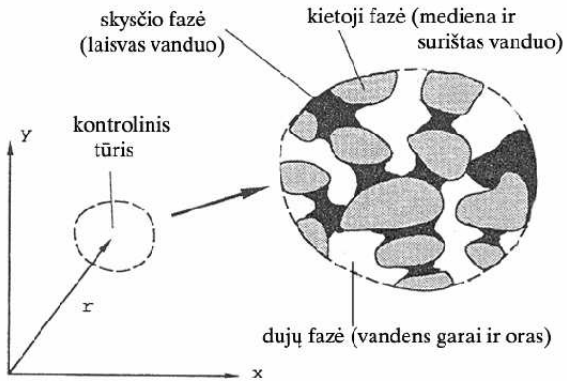
Toliau šiame poskyryje pateiktas nuoseklioji įrankio aprašymas, pradedant nuo matematinio modelio ir baigiant realizacijos ypatumais. Poskyrį užbaigia naujosios, lygiagrečiosios įrankio versijos kūrimo ir lygiagretinimo aprašymas.

3.2.2. Nuoseklioji įrankio versija

Čia aptariama pradinė nuoseklioji įrankio versija. Pradžioje pateikiamas modeliuojamų uždavinių matematinis modelis, skaitinio modelio sudarymo ypatumai. Toliau aprašoma įrankio struktūra, ir, galiausiai, šis poskyris užbaigiamas įrankio tikrinimo pavyzdžiais.

A. Fizikinis modelis

Poringa terpė susideda iš kietosios fazės (angl. *solid matrix*) ir likusių ertmių (angl. *void space arba pore space*), kurios gali būti užpildytos viena arba daugiau komponenčių (pavyzdžiui, vanduo, dujos, nafta). Tipiniai poringų terpių pavyzdžiai gali būti dirva, smėlis, mediena, betonas, duona arba žmogaus plaučiai [49]. Tipinė poringos terpės schema pavaizduota 3.1 paveiksle.



3.1 pav. *Poringos terpės schema*

Pagal apibrėžimą *fazė* yra chemiškai homogeninė nagrinėjamos sistemos dalis, kuri nuo kitų dalių atskirta tam tikra fizine riba. Vienfaziu atveju sistemos ertmės užpildytos vienu skysčiu, pavyzdžiui, vandeniu, arba keliais skysčiais, kurie tarpusavyje visiškai susimaišo, pavyzdžiui, paprastas ir sūrus vanduo. Daugiafaziu atveju ertmės būna užpildytos skysčiais, kurie tarpusavyje nesimaišo, tokiais kaip, pavyzdžiui, vanduo ir nafta. Tarp tokių skysčių galima tiksliai nusakyti ribą, kur pasibaigia vienas ir prasideda kitas. Kadangi visos dujos tarpusavyje pilnai susimaišo, dujinė fazė gali būti tik viena. 3.1 paveiksle parodyta dvifazė sistema.

Komponentėmis vadinamos fazės dalys, kurios gali būti chemiškai identifikuojamos. Komponenčių kiekis priklauso nuo nagrinėjamos fizikinės sistemos. Pavyzdžiui, paprasto ir sūraus vandens mišinys yra dvikomponentė fazė.

Norint kurti poringų terpių matematinius modelius, poringų terpių geometrija turi pasižymėti tokiomis savybėmis:

1. poringų terpių ertmės turi būti sujungtos tarpusavyje;
2. ertmių dydis d turi būti žymiai didesnis už fazių molekulių vidutiniojo laisvojo kelio ilgį l;
3. ertmės turi būti žymiai mažesnės nei sistemos išmatavimai L;
4. skysčių tekėjimas turi vykti lėtai.

1 savybė yra akivaizdi, nes priešingu atveju nebūtų įmanomas skysčių tekėjimas. 2 savybė leidžia mums nagrinėti skysčius ne molekuliniame lygyje, o taikyti jiems kontinumo sąvoką. Pavyzdžiui, vidutinės temperatūros[1] orui $l \approx 6 \cdot 10^{-8}$m. Galiausiai 3 savybė reiškia, kad ertmės turi būti pakankamai mažos, kad kapiliarinės jėgos vaidintų didelį vaidmenį (pavyzdžiui, miesto kanalizacijos vamzdynas nėra poringoji terpė), ir, visos sistemos atžvilgiu, būtų galima taikyti makroskopinius modelius [49, 93]. 2 ir 3 savybes galima užrašyti sąryšiais

$$l \ll d \ll L\,. \tag{3.1}$$

4 savybė reikalinga, kad būtų galima neatsižvelgti į judėjimo inertiškumą.

B. Matematinis sprendimo modelis

Įrankyje modeliuojamas dvifazis skysčių tekėjimas. Šiame skyrelyje aprašomas įrankyje panaudotas bendras daugiafazio tekėjimo modelis ir jo variacija – globalaus slėgio modelis [2], taip pat šių modelių taikymai dvifazėms sistemoms.

[1] Čia vidutinė temperatūra laikoma 20°C

B.1. Daugiafazis tekėjimas

Kuriant įrankį, buvo realizuotas vienas iš daugiafazio tekėjimo modelių (MFM) variantų. Šiame modelyje skirtingos fazės traktuojamos kaip atskiri skysčiai su individualinėmis termodinaminėmis ir mechaninėmis savybėmis [19]. Makroskopinės lygtys gaunamos, vidurkinant mikroskopines lygtis kontroliniame tūryje. Mikroskopinės lygtys yra masės, momento ir energijos tvermės lygtys visose fazėse ir sąryšiai tarp jų. Kontrolinis tūris pavaizduotas 3.1 paveiksle, jo išmatavimai turi tenkinti (3.1) sąlygas. Tada įmanoma pakeisti realią netolydžią medžiagą fiktyvia tolydžia. Visi makroskopiniai dydžiai gaunami vidurkinant mikroskopinius. Vidurkinimas (integravimas) atliekamas, darant įvairias prielaidas (homogeniškumo, periodinės medžiagos struktūros ir kitas) [69].

MFM modelio lygtys, aprašančios daugiafazinio, daugiakomponentinio pernešimo reiškinius nedeformuojamoje poringoje terpėje, atrodo taip:

1. Fazės k masės tvermės dėsnis:

$$\varepsilon \frac{\partial(\rho_k s_k)}{\partial t} + \nabla \cdot (\rho_k \mathbf{u}_k) = \bar{m}_k \,, \tag{3.2}$$

čia ε yra medžiagos poringumas (angl. *porosity*), ρ_k yra fazės tankis, s_k yra fazės prisisotinimas, kuris nurodo, kokią viso tūrio dalį užima ši fazė medžiagos ertmėse, taip kad

$$\sum_k s_k = 1 \,, \tag{3.3}$$

o $\mathbf{u}_k$ yra fazės Darcy greičio vektorius. Narys $\bar{m}_k$, nusako masės pernešimo greičius iš kitų fazių į duotąją. Jei sistemoje nėra išorinių masės šaltinių arba naikinimo taškų, tai

$$\sum_k \bar{m}_k = 0 \,. \tag{3.4}$$

2. Momento tvermės dėsnį, kai į inerciją galima nekreipti dėmesio, pakeičia Darcy dėsnis

$$\mathbf{u}_k = -\mathbf{K} \frac{k_{rk}}{\mu_k} \left(\nabla p_k - \rho_k \mathbf{g} \right) \,, \tag{3.5}$$

čia $\mathbf{K}$ yra poringos terpės absoliutaus pralaidumo (angl. *permeability*) tenzorius, k_{rk} yra fazės santykinis pralaidumas, μ_k yra fazės dinaminis klampumas, p_k yra fazės slėgis ir, galiausiai, $\mathbf{g}$ yra gravitacinė konstanta. Skirtumas tarp fazių k ir j slėgiu vadinamas *kapiliariniu slėgiu*

$$p_{c\,kj} = p_k - p_j \,. \tag{3.6}$$

Panašiai apibrėžiamas ir fazių komponenčių masės tvermės dėsnis, taip pat fazės energijos tvermės dėsnis. Kapiliariniam slėgiui, pralaidumams, klampumams ir kitiems parametrams išreikšti naudojamos funkcijos, modeliuojančios šių parametrų elgesį realiame gyvenime. Visos šitos lygtys sudaro pilną lygčių sistemą, iš kurios galima rasti nežinomus greičio vektorių $\mathbf{u}_k$, skaliarinį slėgį p_k ir prisisotinimą s_k, fazių komponenčių masių koncentracijas C_k^α ir bendrą temperatūrą T.

B.2. Globalus slėgis

Spręsdami diferencialines lygtis, turime tam tikrą laisvę pasirenkant kintamuosius. Toliau aprašomas globalaus slėgio modelis, kartais dar vadinamas daugiafazio mišinio modeliu [2, 48], kuris realizuotas pradinėje įrankio versijoje. Pagrindinė šio modelio idėja yra globalaus slėgio įvedimas. O pats daugiafazis tekėjimas yra aprašomas performuluotais per mišinio savybes ir charakteristikas (tankį, klampumą, slėgį, greitį) tvermės dėsniais. Šis modelis pasižymi tuo, kad jo lygtys yra mažiau susijusios, be to, į jas įeinantys dydžiai yra glodesni.

Modelyje visų fazių mišinys traktuojamas kaip tam tikras skystis, kurio parametrai išreiškiami per *fazių* parametrus. Pavyzdžiui, globalus tankis ir greitis apibrėžiami tokiu būdu:

$$\rho = \sum_k \rho_k s_k \,, \tag{3.7}$$

$$\rho\,\mathbf{u} = \sum_k \rho_k \mathbf{u}_k \,. \tag{3.8}$$

Analogiškai apibrėžiami komponenčių koncentracijos ir bendroji entalpija. Mišinio kinematinis klampumas apibrėžiamas taip:

$$\nu = \left(\sum_k \frac{k_{rk}}{\nu_k} \right)^{-1} , \tag{3.9}$$

čia $\nu_k = \mu_k/\rho_k$ yra k-sios fazės kinematinis klampumas. Kiekvienos fazės mobilumas mišinyje apibrėžiamas sąryšiais

$$\lambda_k = \frac{k_{rk}}{\nu_k}\nu \,, \sum_k \lambda_k = 1 \,. \tag{3.10}$$

Ypač įdomus globalaus slėgio apibrėžimas, kuris apibrėžiamas taip, kad galiotų lygtis:

$$\nabla p = \sum_j \lambda_j \nabla p_j = \nabla p_k + \sum_j \lambda_j \nabla p_{c\,jk} \,. \tag{3.11}$$

Toks slėgis visada egzistuoja homogeninėje terpėje, tačiau, modeliuojant heterogeninius uždavinius, atsiranda gana didelių sunkumų, ir tai yra šio modelio silpnoji pusė.

Taigi, globalaus slėgio modelis aprašomas tokiomis lygtimis:

1. Daugiafazio mišinio masės tvermės dėsnis. Susumavus visas (3.2) lygtis ir atsižvelgus į (3.4) sąlygą, gauname

$$\varepsilon \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \,, \tag{3.12}$$

   kas sutampa su atitinkama vienfazio tekėjimo lygtimi.

2. Daugiafazio mišinio momento tvermės dėsnis. Pirmiausiai, naudodami (3.10) apibrėžimą, pertvarkome (3.5) lygtis tokiu būdu:

$$\rho_k \mathbf{u}_k = -\mathbf{K} \frac{k_{rk}}{\nu_k} \left(\nabla p_k - \rho_k \mathbf{g} \right) = -\frac{\mathbf{K}}{\nu} \left(\lambda_k \nabla p_k - \lambda_k \rho_k \mathbf{g} \right) . \tag{3.13}$$

   Tada, sumuodami šias lygtis ir naudojant mišinio greičio (3.8) ir slėgio (3.11) apibrėžimus, gauname momento tvermės lygtį daugiafaziam mišiniui:

$$\rho \, \mathbf{u} = -\frac{\mathbf{K}}{\nu} \left(\nabla p - \left(\sum_k \lambda_k \rho_k \right) \mathbf{g} \right) = -\frac{\mathbf{K}}{\nu} \left(\nabla p - \gamma_\rho \rho \, \mathbf{g} \right) , \tag{3.14}$$

   čia γ_ρ yra vadinama tankio korekcijos faktoriumi ir apibrėžiama kaip

$$\gamma_\rho = \frac{\sum_k \lambda_k \rho_k}{\sum_k \rho_k s_k} \,. \tag{3.15}$$

   Difuzinio masės pernešimo vektorius fazei k apibrėžiamas taip:

$$\mathbf{j}_k = \rho_k \, \mathbf{u}_k - \lambda_k \rho \, \mathbf{u} \,, \quad \sum_k \mathbf{j}_k = 0 \,, \tag{3.16}$$

   arba, sukeitus narius vietomis,

$$\rho_k \, \mathbf{u}_k = \mathbf{j}_k + \lambda_k \rho \, \mathbf{u} \,. \tag{3.17}$$

   Įrašius į (3.16) lygtį (3.13) ir (3.14), gauname, kad difuzinio masės pernešimo fazėje k narys gali būti išreikštas tokiu būdu:

$$\mathbf{j}_k = \frac{\mathbf{K} \lambda_k}{\nu} \sum_i \lambda_i \left(\nabla p_{c\,ik} + (\rho_k - \rho_i) \, \mathbf{g} \right) . \tag{3.18}$$

Komponenčių masės tvermės dėsniai ir energijos tvermės dėsniai apibrėžiami analogiškai.

B.3. Dvifazis modelis

Kai turime tik dvi fazes, ankščiau išvardytos lygtys gali būti pertvarkytos. Darcy dėsnio (3.14) formuluotė dabar atrodo taip:

$$\rho\,\mathbf{u} = -\frac{\mathbf{K}}{\nu}\left[\nabla p - (\lambda_1\rho_1 + \lambda_2\rho_2)\,\mathbf{g}\right]. \tag{3.19}$$

Įrašius ją į masės tvermės dėsnį (3.12), gauname

$$\varepsilon\frac{\partial\rho}{\partial t} - \nabla\cdot\left(\frac{\mathbf{K}}{\nu}\left[\nabla p - (\lambda_1\rho_1 + \lambda_2\rho_2)\,\mathbf{g}\right]\right) = \bar{m}\,, \tag{3.20}$$

čia $\bar{m} = \bar{m}_1 + \bar{m}_2$ yra mišinio masės šaltinis, kuris, jei tokio šaltinio nėra, lygus 0.

Ši lygtis vadinasi slėgio lygtimi, nes ji naudojama mišinio slėgiui p surasti. Mišinio kinematinis klampumas ν (3.9) yra teigiamas, vadinasi, jei poringos terpės absoliutaus pralaidumo tenzorius $\mathbf{K}$ yra teigiamai apibrėžtas, toks bus ir santykis $\mathbf{K}/\nu$. Taigi, iš (3.20) išraiškos seka, kad slėgio lygtis yra *elipsinė*.

Toliau išvesime pernešimo lygtį pirmos fazės prisisotinimui s_1. Pirmiausia, panagrinėkime masės tvermės lygtį (3.2) pirmai fazei. Įstačius į ją (3.17) išraišką masės srautui $\rho_1\,\mathbf{u}_1$ ir (3.18) išraišką difuziniam masės srautui $\mathbf{j}_1$, gauname

$$\varepsilon\frac{\partial\left(\rho_1 s_1\right)}{\partial t} + \nabla\cdot\left(\rho\,\mathbf{u}\lambda_1\right) = -\nabla\cdot\left(\frac{\mathbf{K}\lambda_1\lambda_2}{\nu}\left[\nabla p_c + (\rho_1 - \rho_2)\,\mathbf{g}\right]\right) + \bar{m}_1\,. \tag{3.21}$$

Ši lygtis dažnai vadinama *prisisotinimo-tvermės* lygtimi. Priklausomai nuo dominuojančių narių, ji gali būti hiperbolinė arba parabolinė.

Darcy dėsnis (3.19) kartu su slėgio (3.20) ir prisisotinimo-tvermės (3.21) lygtimi sudaro pilną lygčių sistemą nežinomiesiems p, $\mathbf{u}$ ir s_1 rasti. Ši sistema gali būti panaudota modeliuojant dvifazį nesusimaišantį tekėjimą, kai galima nekreipti dėmesį į fazių sudėtį, molekulinę difuziją ir hidraulinę dispersiją. Esant reikalui, šis modelis gali būti išplėstas, didinant fazių skaičių (tokiu atveju gali atsirasti papildoma prisisotinimo-tvermės lygtis kitoms fazėms), arba įskaitant energijos pernešimus (tada atsiras papildoma lygtis temperatūrai arba entalpijai).

B.4. Skaitinis modelis

Sudarant skaitinį modelį, pirmiausia aptarsime (3.19), (3.20) ir (3.21) lygčių sistemos netiesiškumą ir ryšius. Reikia atkreipti dėmesį, kad slėgio lygties (3.20) koeficientai priklauso nuo prisisotinimo, bet ne nuo slėgio. Taigi, pagal [8], slėgio

lygtis yra silpnai netiesinė. Be to, mišinio koeficientas ν yra žymiai glodesnis nei atskirų fazių koeficientai. Taigi slėgio lygties (3.20) koeficientai gali būti aproksimuojami išreikštiniu būdu.

Iš kitos pusės, prisisotinimo-tvermės lygtis (3.21) yra stipriai netiesinė. Netiesiškumą sąlygoja fazių santykiniai pralaidumai ir kapiliarinis slėgis. Prisisotinimo-tvermės ir slėgio lygtys yra stipriai susijusios per mišinio greitį iš Darcy dėsnio (3.19), tačiau globalaus slėgio modelyje šis ryšys yra silpnesnis, palyginant su modeliais, kur naudojamas fizikinis slėgis [17].

Sprendimo procedūra

Atsižvelgiant į ankščiau minėtus samprotavimus, naudojama speciali sprendimo procedūra, atskirianti skirtingų lygčių sprendimo algoritmus. Ji yra panaši į procedūrą, panaudotą [8] darbe. Ši procedūra susideda iš tokių žingsnių:

1. Laiko momentu $t = t_{i-1}$ žinomi pagrindiniai kintamieji p ir s_1 (kai $t = 0$ šią informaciją gauname iš pradinių sąlygų);
2. Skaičiuojami slėgio lygties (3.20) koeficientai $\nu(s_1)$, $\lambda_1(s_1)$, $\lambda_2(s_1)$ ir $\rho(s_1)$;
3. Taikoma baigtinių tūrių metodu pagrįsta skirtuminė schema, kurios detalės pateiktos žemiau, iš (3.20) lygties randame slėgį p. Gautoji tiesinių lygčių sistema sprendžiama BiCGStab metodu, nes lygčių sistemos matrica yra nesimetrinė. Šiame etape atliekame didžiąją dalį skaičiavimų;
4. Iš Darcy dėsnio (3.19) randame mišinio greitį $\mathbf{u}$. Naudojama ta pati diskretizacijos schema, kaip ir 3 žingsnyje, koeficientai, apskaičiuoti 2 žingsnyje, ir mišinio slėgis, apskaičiuotas 3 žingsnyje;
5. Pikaro arba Niutono iteraciniu metodu iš prisisotinimo-tvermės lygties (3.21) randame prisisotinimą s_1. Naudojamas greitis $\mathbf{u}$, apskaičiuotas 4 žingsnyje. Tiesinių lygčių sistemos vėl sprendžiamos BiCGStab metodu;
6. Patikslinami 2 žingsnyje apskaičiuoti koeficientai ir atliekamos kelios iteracijos (žingsniai 2–6) šiame laiko žingsnyje tol, kol rezultatas konverguos ((3.20) lygties netiktis taps pakankamai maža);
7. Pereinama prie sekančio laiko žingsnio, ir kartojama procedūra, aprašyta ankščiau. Taip daroma tol, kol nebus pasiektas galutinis laiko momentas $t = T$.

Standartinėje daugiafazinio tekėjimo poringoje terpėje modeliavimo procedūroje IMPES [45], slėgio lygties koeficientai įvertinami išreikštiniu būdu, o pati lygtis sprendžiama neišreikštiniu būdu. Koncentracijos-prisisotinimo-tvermės lygtys sprendžiamos išreikštiniu būdu. Taigi, sudėtingo netiesiškumo atveju, norint

garantuoti algoritmo stabilumą ir konvergavimą tenka dažnai naudoti labai mažą laiko žingsnį. Tai ženkliai sumažina skaičiavimų efektyvumą ir padidina skaičiavimų apimtį.

Mūsų sprendimo procedūroje slėgio ir kitos lygtys pradžioje atskiriamos, linearizuojamos, ir tik paskui sprendžiamos neišreikštiniu būdu. Tokia procedūra kartais dar vadinama nuosekliuoju pusiau neišreikštiniu metodu. Naudojant šį metodą, nelieka laiko žingsnio apribojimų dėl stabilumo reikalavimų. Taip pat šis metodas gali būti lengvai išplėstas, jei lygčių sistemoje atsiras naujos lygtys (pavyzdžiui, įvedant daugiau fazių arba įskaitant temperatūras ir fazių komponentes).

Skirtuminės schemos

Be stabilumo ir konvergavimo taip pat siekiama, kad skaitiniai metodai, naudojami daugiafazio tekėjimo uždaviniuose, išlaikytų masės tvermės dėsnį, turėtų monotoninius sprendinius, be nefizikinių virpesių ir teigiamai apibrėžtas matricas.

Pradinėje įrankio versijoje užduoties sritis Ω dalinama į kontrolinius tūrius, naudojant stačiakampius netolygius paslinktus tinklus (angl. *non-uniform staggered grid*). Diferencialinės lygtys aproksimuojamos skirtuminėmis schemomis šiuose kontroliniuose tūriuose. Integruojant slėgio lygtį (3.20) kontroliniame tūryje V ir taikant konvergavimo teoremą, gauname lygtį

$$\int_V \varepsilon \frac{\partial \rho}{\partial t} d\mathbf{x} - \oint_{\partial V} \frac{\mathbf{K}}{\nu} \left[\nabla p - (\lambda_1 \rho_1 + \lambda_2 \rho_2)\, \mathbf{g}\right] \cdot \mathbf{n} ds = \int_V \bar{m} d\mathbf{x}\,.$$

Naudojant šį metodą, masės tvermės dėsnis galioja kiekviename tūryje. Aproksimuojant pilną srautą kontrolinio tūrio kraštuose, naudojami koeficientų harmoniniai arba aritmetiniai vidurkiai. Diskretizacijai pradinėje įrankio versijoje buvo naudojama vienpusių skirtumų aproksimacija (angl. *upwind*). Tokiu būdu gaunamos matricos 1D atveju yra tridiagonalinės, 2D ir 3D atveju matricos turi po 5 ir 7 diagonalių atitinkamai.

Analogiškai, iš prisisotinimo-tvermės lygties (3.21) gauname

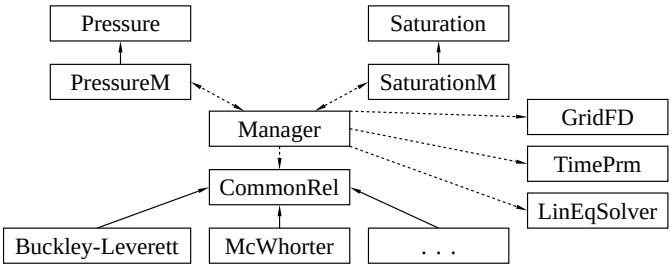
$$\int_V \varepsilon \frac{\partial \rho_1 s_1}{\partial t} d\mathbf{x} + \oint_{\partial V} \left\{ \lambda_1 \rho \mathbf{u} + \frac{\mathbf{K} \lambda_1 \lambda_2}{\nu} \left[\nabla p_c + (\rho_1 - \rho_2)\, \mathbf{g}\right] \right\} \cdot \mathbf{n} ds = \int_V \bar{m}_1 d\mathbf{x}\,.$$

Ši lygtis pasižymi stipriu netiesiškumu, todėl ją linearizuojant mūsų sprendimo procedūros 5 žingsnyje atliekamos keletas Pikaro arba Niutono iteracijų. Jei pradinė reikšmė pataikyta gerai, Niutono iteracijos konverguoja greičiau, tačiau Pikaro metodas mažiau priklauso nuo pradinės reikšmės tikslumo.

C. Įrankio struktūra

Kuriant įrankį, buvo panaudotos paskutiniuoju metu populiarėjančios moksliniuose skaičiavimuose OOP technologijos. Naudojant OOP technologijas, sumažinamas programavimo ir tikrinimų laikas, programos kodas tampa aiškesniu, lengviau palaikomu ir išplečiamu. Įrankio kūrimui buvo pasirinkta C++ kalba. Kaip ir C kalba, ji leidžia efektyviai realizuoti skaitinius algoritmus, programos kodas yra lengvai pernešamas į naujas platformas, gali būti panaudotos daugelis bibliotekų, turinčių C sąsają (jau nekalbant apie didelę klasę skaičiavimų ir lygiagretinimo bibliotekų, skirtų specialiai C++ kalbai). Tačiau, skirtingai nuo tokių programavimo kalbų, kaip C ir FORTRAN[2], ji puikiai skirta OO programavimui.

Įrankio klasių diagrama pavaizduota 3.2 paveiksle. Panaudojant strategiją, pasiūlytą [72], ir panaudotą *Diffpack* bibliotekoje, mūsų diferencialinių lygčių sistemos sprendiklis kuriamas, apjungiant nepriklausomus sprendiklius kiekvienai iš lygčių, įeinančių į sistemą. Taigi, pirmiausia kuriamas slėgio lygties (3.20) sprendiklis ir tikrinamas su paprastais koeficientais – konstantomis arba žinomomis funkcijomis, kurioms galime rasti analitinius sprendinius. Šio sprendiklio klasė 3.2 paveiksle pavaizduota kaip `Pressure`. Panašiai kuriama ir tikrinama prisisotinimo-tvermės lygties (3.21) sprendiklio klasė `Saturation`.



3.2 pav. *Įrankio klasių diagrama*

Ryšys tarp lygčių sistemoje yra per koeficientus. Mūsų įrankyje šitie koeficientai, į kuriuos įeina fazių šaltiniai, pradinės ir kraštinės sąlygos, realizuoti per virtualias funkcijas. Diferencialinių lygčių sprendiklių klasių palikuonys gali pateikti kitas šių koeficientų išraiškas, labiau tinkančias sprendžiamam uždaviniui. Tam tikram fizikiniam uždaviniui koeficientai grupuojami, ir pasiekiami per klasės `CommonRel` sąsają. Ši klasė turi palikuonių įvairiems specifiniams modeliuojamiems uždaviniams. Tokiu būdu koeficientai gali būti lengvai pakeisti, nekeičiant lygčių sprendiklių kodo.

[2]Fortran 2003 kalbos specifikacijoje yra OO programavimo galimybių palaikymas, tačiau C++ kalba yra *žymiai* populiaresnė šioje srityje

Klasė `Manager` kontroliuoja visą diferencialinių lygčių sprendimo procesą. Ši klasė turi rodykles į klases `PressureM` ir `SaturationM` – palikuonis konkretiems uždaviniams spręsti. Klasė `Manager` taip pat yra atsakinga už laiko ir erdvinę diskretizaciją. Ji paskiria atmintį ir sprendžia tiesinių lygčių sistemas (naudojamos specialios klasės), taip pat pateikia klasėms-sprendikliams visą reikiamą informaciją.

Toks įrankio dizainas pritaikytas anksčiau pateiktai diferencialinių lygčių sistemos sprendimo schemai. Įrankio architektūra tinka ir plečiant įrankį pagal anksčiau aprašytą scenarijų.

Šiuo metu įrankyje naudojami dekartiniai struktūriniai tinklai. Įrankis leidžia spręsti 1D, 2D ir 3D uždavinius. Gaunamos tiesinės lygčių sistemos sprendžiamos iteraciniais metodais. MFSolver realizuotų algoritmų analizė parodė, kad pagrindinė skaičiavimų dalis (iki 80 % viso skaičiavimo laiko) tenka tiesinės algebros algoritmams, o kode operuojama su tiesinės algebros objektais vektoriais ir išretintomis (angl. *sparse*) matricomis. Vektorių operacijos priklauso duomenų lygiagretumo operacijų klasei, bei dalis operacijų yra globalios (skaliarinės sandauga, normų skaičiavimas). Kita pagrindinė operacija – matricos ir vektoriaus sandaugos skaičiavimas.

D. Įrankio bandymo rezultatai

Šiame poskyryje pateikti ankščiau aprašyto įrankio bandymo rezultatai. Kadangi bandymų tikslas buvo patikrinti *sukurto* įrankio funkcionalumą, buvo parinkti uždaviniai, kurių sprendimai yra žinomi. Tokie yra vienmačiai Buckley-Leverett ir McWhorter uždaviniai.

D.1. Įrankio taikymas Buckley-Leverett uždaviniui spręsti

Buckley-Leverett uždavinys gaunamas iš bendro modelio, paimant vienmatį atvejį ir padarant prielaidą, kad nėra kapiliarinių jėgų ir gravitacijos. Pavyzdys – naftos išstūmimas vandeniu, kai taikomos jėgos yra žymiai didesnės už gravitaciją ir kapiliarinio slėgio gradientą. Tada uždavinys susiveda į tokios lygties sprendimą:

$$\varepsilon \frac{\partial s}{\partial t} + u \frac{\partial \lambda(s)}{\partial x} = 0\,. \tag{3.22}$$

Ši lygtis yra gerai išnagrinėta [50], ir turi analitinius sprendinius. Žinoma, kad ji gali turėti daug apibendrintų (angl. *weak*) sprendinių, iš kurių tik vienas yra fizikinis. Norint nustatyti fizikinį sprendinį, naudojamos papildomos, pavyzdžiui, entropijos sąlygos. Tai, kad yra visa aibė galimų sprendinių reiškia, kad skaitinis sprendinys gali sukonverguoti į bet kurį iš jų. Kadangi buvo ieškomas fizikinis

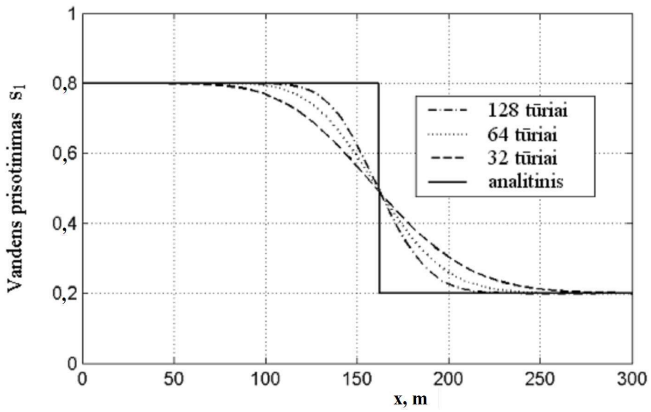
sprendinys, iš sprendiko buvo reikalaujama, kad jis surastų būtent fizikinį sprendinį.

Mūsų atveju tiriamas nestacionarus naftos išstūmimas vandeniu. Uždavinys yra sprendžiamas srityje $\Omega = (0,\ 300[m])$ ir laiko intervale $(0,\ 1500[d])$ su parametrais iš [45]. Buvo pasirinktas pakankamai mažas laiko žingsnis ir keičiamas tinklelio tankis. Atlikti du eksperimentai, kai santykinis pralaidumas k_{rl} buvo paimtas tiesinis, ir kai naudota pralaidumo išraiška, pasiūlyta Brukso ir Kori [28]:

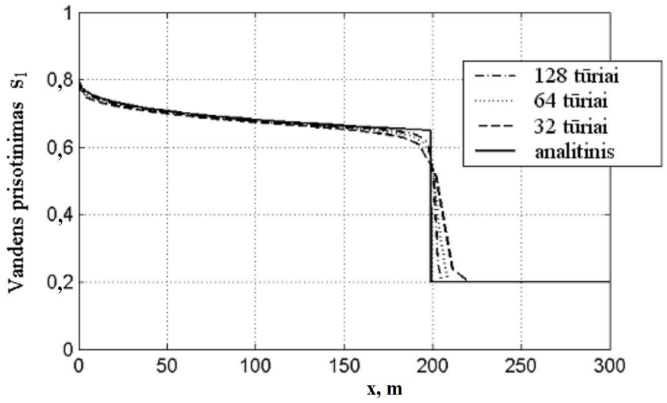
$$k_{rl} = S^{\frac{2+3\lambda}{\lambda}}, \quad k_{rg} = (1-S)^2\left(1 - S^{\frac{2+\lambda}{\lambda}}\right). \tag{3.23}$$

Pradinis vandens prisisotinimas visur yra 0,2. Vanduo pumpuojamas iš kairės pusės, kur vandens prisisotinimas yra 0,8, greičiu $1{,}5 \cdot 10^{-4}\left[\frac{kg}{m^2 s}\right]$.

Rezultatai pavaizduoti 3.3 ir 3.4 paveiksluose. Kaip matome, skaitinis sprendinys konverguoja į analitinį. Jis elgiasi monotoniškai ir neturi nefizinių virpesių, ko mes ir siekėme pasirinkdami skaitinėje schemoje konvekcinių narių aproksimacijas vienpusiais skirtumais. Nesunku pastebėti, kad skaitinis sprendinys teisingai aproksimuoja bendrą vandens masę ir fronto judėjimo greitį. Deja, pasirinktas skaitinis metodas pasireiškia ir skaitine difuzija, kuri suglodina vandens prisotinimo frontą. Be to, šis metodas yra tik pirmosios tikslumo eilės. Aukštesnios eilės tikslumo metodai pagreitina konvergavimo greitį ir pagerina fronto formos aproksimaciją.



3.3 pav. *Buckley-Leverett uždavinio analitinis ir skaitiniai sprendiniai (tiesinis atvejis)*



3.4 pav. *Buckley-Leverett uždavinio analitinis ir skaitiniai sprendiniai (Brooks-Corey atvejis)*

D.2. Įrankio taikymas McWhorter uždaviniui spręsti

McWhorter uždavinyje irgi nagrinėjamas nestacionarus naftos išstūmimas vandeniu vienmatėje horizontalioje sistemoje, tačiau šį kartą kapiliarinių jėgų poveikis nėra atmetamas. McWhorter ir Sunada gavo kvazi-analitinį šio uždavinio sprendinį, kurio radimas reikalauja integralinės lygties sprendimo [20].

Į naftą prisotintą terpę pro jos kairįjį kraštą yra įstumiamas vanduo. Dešinysis sistemos kraštas yra uždaras (nepralaidus). Šiuo atveju nafta išeina iš sistemos pro tą patį kairįjį kraštą, o bendras mišinio greitis yra lygus nuliui $u = 0$ (angl. *countercurrent displacement*).

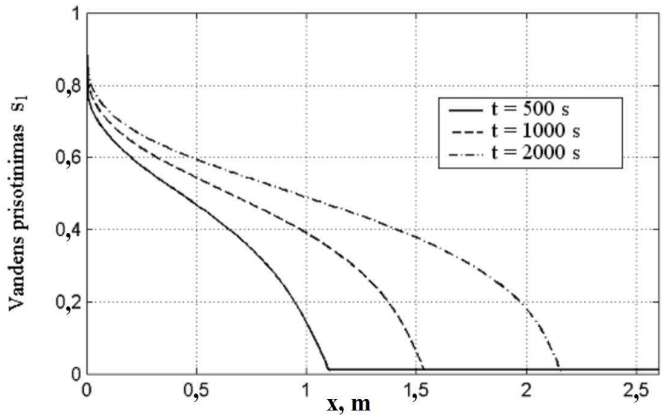
3.5 paveiksle yra parodytas McWhorter uždavinio skaitinis sprendinys skirtingais laiko momentais. Jis buvo gautas su diskretizavimo parametrais iš [45]: $\Delta x = 0{,}01[m]$, $\Delta t = 100[s]$. Analizuojant vandens prisotinimo profilius, galima pastebėti suglodinantį (difuzinį) kapiliarinių jėgų poveikį.

3.2.3. Lygiagrečioji, patobulinta įrankio versija

Šiame paragrafe kalbama apie daugiafazio skysčių tekėjimo poringose terpėse modeliavimo įrankio naujosios versijos, turinčios lygiagretinimo galimybes, kūrimą. Taip pat aprašomi naujosios versijos modelio patobulinimai.

A. Globalaus slėgio modelio išplėtimas

Pradinėje įrankio versijoje buvo realizuotas dvifazis tekėjimas, kur buvo neatsižvelgiama į šiluminius reiškinius bei fazių komponenčių susimaišymą. Siekiant



3.5 pav. *McWhorter uždavinio skaitinis sprendinys trims laiko momentais*

įskaityti šiuos reiškinius, prie lygčių (3.2) ir (3.5) į sistemą reikia pridėti tokias lygtis:

- Komponenčių masės tvermės dėsnį

$$\frac{\partial}{\partial t}\left(\varepsilon\rho_k s_k C_k^\alpha\right) + \nabla\cdot\left(\rho_k \mathbf{u}_k C_k^\alpha\right) = -\nabla\cdot\mathbf{j}_k^\alpha + \bar{J}_k^\alpha\,, \tag{3.24}$$

  čia C_k^α yra komponentės α masės koncentracija fazėje k, taip kad

$$\sum_\alpha C_k^\alpha = 1\,, \tag{3.25}$$

  $\mathbf{j}_k^\alpha$ yra komponenčių pernešimo tarp fazių dėl difuzijos vektorius, o $\bar{J}_k^\alpha$ nurodo komponenčių virsmus dėl cheminių reakcijų.

- Energijos tvermės dėsnį fazėje k

$$\frac{\partial}{\partial t}\left(\varepsilon\rho_k s_k h_k\right) + \nabla\cdot\left(\rho_k \mathbf{u}_k h_k\right) = \nabla\cdot\left(\varepsilon s_k k_k \nabla T\right) + \bar{q}_k\,, \tag{3.26}$$

  čia k_k yra fazės šiluminis laidumas, o

$$h_k = \int_0^T c_k dT + h_k^0 \tag{3.27}$$

  yra fazės entalpija.

Panaudojant efektyvųjį šiluminį laidumą, efektyviąją difuziją ir kitus apibendrintus rodiklius, gaunamos atitinkamos lygtys ir globalaus slėgio modeliui, kurios leidžia mums surasti bendrą mišinio temperatūrą ir komponenčių C^{α} koncentracijas.

Šios lygtys įtraukiamos į įrankį, kuriant naujas šių lygčių sprendimo klases *Temperature* ir *Concentration*, bei naują valdymo `Manager` tipo klasę, sugebančią valdyti temperatūros ir koncentracijų skaičiavimą.

B. Slėgio-Prisisotinimo modelis

Globalaus slėgio modelis yra *patogus* realizuojant skaitinius algoritmus, tačiau jame daromos matematinės prielaidos, kurios nevisada galioja realiuose uždaviniuose. Viena iš tokių prielaidų terpės homogeniškumas. Norint spręsti nehomogeninius uždavinius, kai, pavyzdžiui, įterpiamas kitos medžiagos „lęšis", tenka modifikuoti uždavinį, pakeičiant trūkius perėjimus tolydžiais, kas nevisada priimtina.

Norint išspręsti šią problemą, įrankyje realizuotas naujas, papildomas slėgio-prisisotinimo modelis. Tai yra to pačio daugiafazio tekėjimo modelio variantas, kai parenkami kiti kintamieji – vienas iš slėgių ir vienas iš prisisotinimų (dvifazio tekėjimo atveju). Kadangi šitie kintamieji jau egzistuoja modelyje, jų panaudojimas nereikalauja jokių papildomų sąlygų, kaip tai yra globalaus slėgio modelyje, todėl taikant šį modelį galima tiesiogiai spręsti heterogeninius uždavinius. Tačiau šis modelis skaitiškai sunkiau realizuojamas.

C. Naujų skirtuminių schemų realizacija

Kaip jau buvo minėta ankščiau, pradinėje įrankio versijoje buvo naudojama vienpusių skirtumų (angl. *upwind*) aproksimacija. Ši schema buvo pasirinkta dėl to, kad ji duoda sprendimus be virpesių (angl. *oscillations*). Tačiau ši schema yra tik pirmos eilės tikslumo. Ji įneša į sprendinį skaitinę difuziją, mažinančią rezultato tikslumą. Daugiamačiu atveju šios schemos naudojimas pasireiškia jautrumu tinklelio orientacijos krypčiai.

Naujoje įrankio versijoje realizuojama nauja, antros eilės centrinė vienpusių skirtumų aproksimacija [91]. Tokio tipo schemą pirmieji pasiūlė Tadmor ir Jiang [47]. Vėliau ji buvo dažnai modifikuojama (žr. [4, 46, 55] darbus ir nuorodas juose). Tokio tipo schemos yra paprastos ir atsparios (angl. *robust*), ir gali būti naudojamos kaip universalūs sprendikliai procesų poringose terpėse modeliavimo įrankiuose.

Naujos schemos palyginimai su įprastine vienpusių skirtumų aproksimacija, naudojama pradinėje įrankio versijoje, parodė, kad nauja schema pasižymi konvergavimo greičiu, artimu optimaliam. Taip pat nauja schema įneša žymiai mažesnę

skaitinę difuziją [91]. Iš vienos pusės, tai yra gerai, nes tiksliau aproksimuojamas bangos tipo sprendinio status frontas. Tačiau yra žinoma, kad difuzija padeda iš visų galimų sprendinių išrinkti fizikinį [45, 49, 50]. Mažėjant skaitinei difuzijai, didėja pavojus, kad bus gautas nefizikinis sprendinys. Savo darbe [91] parodėme, kad didesniojo tikslumo schemų sprendiniai gali konverguoti į nefizikinį sprendinį, kai srauto funkcionalo išvestinė yra nepastovaus ženklo. Šiuo atveju gali padėti difuzinis narys, atsiradęs dėl kapiliarinių jėgų.

Taip pat buvo atlikti naujos schemos lygiagretinimo eksperimentai, naudojant ParSol biblioteką. Buvo ištirta 2D centrinė vienpusių skirtumų aproksimacija, pasiūlyta [47] darbe. Sprendinio išvestinė laiko koordinatės atžvilgiu buvo diskretizuojama išreikštiniu Eulerio metodu. Buvo sprendžiama netiesinė hiperbolinė lygtis su periodinėmis kraštinėmis sąlygomis. Procesoriai buvo apjungti į dvimačią topologiją. Lygiagretaus algoritmo pagreitėjimas $S_p(J)$ ir efektyvumas $E_p(J)$, sprendžiant uždavinį VGTU klasteryje „Vilkas“, yra pateikti 3.1 lentelėje. Matome, kad pasiektas praktiškai *idealus* lygiagretinimo efektyvumas.

3.1 lentelė. *Išreikštinio algoritmo pagreitėjimas ir efektyvumas PC klasteryje „Vilkas“*

p	$T_p(640)$	$S_p(640)$	$E_p(640)$
1	111,0	1,00	1,000
2	55,4	2,00	1,002
4	27,5	4,04	1,009
8	13,9	7,99	0,998
16	7,06	15,7	0,982

D. Įrankio lygiagretinimas taikant ParSol biblioteką

MFSolver įrankio lygiagretinimas buvo atliekamas taikant biblioteką ParSol. Lygiagretinant, buvo atliekami tokie veiksmai:

- C masyvai, kuriuose buvo laikomi nežinomieji, pakeičiami į ParSol masyvus. Lygiagrečiai su tuo atliekamas kodo prastinimas, kai išreikštiniu būdu atliekami vektorių ir matricų veiksmai pakeičiami vidinėmis ParSol funkcijomis;
- Modifikuojami iteraciniai tiesinių lygčių sprendimo metodai. Panaudojamas ParSol matricų ILU skaičiavimo funkcionalumas;
- Keičiama daugelio MFSolverio klasių ir funkcijų sąsaja, kadangi dabar operuojama kitais duomenų tipais;

- Keičiamas MFSolver funkcionalumo, bendro skirtingų dimensijų skaičiui, realizavimo principas. Senoje įrankio versijoje algoritmai, bendri skirtingam dimensijų skaičiui, naudodavo tas pačias duomenų struktūras, o dimensijų skaičius buvo perduodamas kaip parametras. Priklausomai nuo šio parametro, skirtingos vykdymo strategijos buvo pasirenkamos *vykdymo* metu. Naudojant ParSol masyvus, tokia strategija neįmanoma, nes ParSol masyvai skirtingiems dimensijų skaičiams yra atskiros, tarpusavyje nesuderinamos duomenų struktūros. Todėl dabar dimensijų skaičius yra šabloninis parametras, ir jis įtakoja kodo generavimą *kompiliavimo* metu.

  Perėjimas nuo paprasto prie šabloninio parametro neįtakoja kodo kiekio pakeitimui, tačiau pailgėja kompiliavimo laikas ir vykdomasis failas tampa žymiai didesnis. Šiuolaikiniai atminties resursai yra labai dideli, todėl toks padidėjimas nėra svarbus. Iš kitos pusės, kadangi dalis sprendimų daroma kompiliavimo, o ne vykdymo metu, pagreitėja programos vykdymo laikas.

Didelė dalis pakeitimų, susijusių su perėjimų nuo nuosekliosios versijos prie lygiagrečiosios, yra išnešta į antraštinį failą `Parsol.h`. Jame, naudojant sąlyginio kompiliavimo direktyvą

```
1 #ifdef MPIversion
```

automatiškai parenkami atitinkami antraštiniai failai bei ParSol klasės, priklausomai nuo to, kaip kompiliuojama programa.

3.3. Vaizdų filtravimo uždaviniai

3.3.1. Įvadas

Šiame skirsnyje aptariamas ParSol matematikos objektų bibliotekos taikymas dvimačių vaizdų apdorojimui. Matematiniai modeliai, parinkti vaizdų filtravimui, naudoja metodus, taikomus sprendžiant diferencialines lygtis su dalinėmis išvestinėmis. Kai kurie dažnai taikomi vaizdų filtrai gaunami konvoliucijos su skirtingos variacijos Gauso funkcijos G_σ būdu. Gauso filtro taikymas yra tolygus tiesinio parabolinio uždavinio

$$\begin{cases} \dfrac{\partial u(X,t)}{\partial t} = \sum\limits_{i=1}^{2} \dfrac{\partial^2 u(X,t)}{\partial x_i^2}, \quad X = (x_1, x_2) \in Q, \\ u(X,0) = u_0(X) \end{cases} \tag{3.28}$$

sprendimui [66], čia u_0 yra pradinis vaizdas, o $t^{1/2}$ nurodo Gauso funkcijos variaciją.

Svarbų patobulinimą kraštų nustatymo (angl. *edge detection*) teorijoje savo novatoriškame darbe pasiūlė Perona ir Malikas [65]. Jie pasiūlė filtrą generuoti sprendžiant netiesinę difuzinę lygtį, kurios tipas artimas poringų terpių modeliams:

$$\frac{\partial u(X,t)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g(|\nabla u(X,t)|\,\frac{\partial u(X,t)}{\partial x_i}\Big)\,, \tag{3.29}$$

čia g yra glotni (angl. *smooth*) nedidėjanti funkcija, begalybėje artėjanti į nulį. Čia mes naudojame tokį gradiento operatoriaus žymėjimą:

$$\nabla u(X,t) = \Big(\frac{\partial u(X,t)}{\partial x_1},\ \frac{\partial u(X,t)}{\partial x_2}\Big).$$

Ši lygtis taiko stiprią difuziją vaizdo paviršiuje pasirinktinai tose srityse, kur u yra glotni, tačiau difuzija netaikoma ten, kur $\nabla u(X,t)$ reikšmė yra didelė. Taigi difuzija stipriai slopinama vaizdo kraštuose, kur didelis gradientas. Kitose vietose šis netiesinis filtras elgiasi panašiai į tiesinį. Funkcijos g pavyzdžiai pateikti [65]:

$$g(s) = \frac{1}{1+s^2}, \quad g(s) = e^{-s^2}\,.$$

Tačiau tiesioginis šios lygties taikymas yra ribotas dėl tokių priežasčių:

- Ji yra neveiksminga, kai vaizdas turi daug triukšmų (o taip yra daugelyje praktinių uždavinių).
- Esant tam tikroms g, pati lygtis gali būti nestabili. Pavyzdžiui, jei $sg(s)$ mažėja, esant tam tikroms s reikšmėms, tai gauname atvirkštinės krypties šilumos lygtį, kuri yra blogai sąlygota (angl. *ill posed*) [30].

Catte, Lions, Morelis ir Collis pasiūlė [65] modifikuotą netiesinį parabolinį uždavinį su pradine-kraštine sąlyga srityje $(X,t) \in Q \times (0,T]$:

$$\begin{cases} \dfrac{\partial u(X,t)}{\partial t} = \sum\limits_{i=1}^{2} \dfrac{\partial}{\partial x_i}\Big(g(|\nabla G_\sigma * u(X,t)|), \dfrac{\partial u(X,t)}{\partial x_i}\Big) + f(u_0 - u), \\ \partial_\nu u = 0, \quad (X,t) \in \partial Q \times (0,T], \\ u(X,0) = u_0(X)\,, \quad X \in Q\,. \end{cases} \tag{3.30}$$

Difuziniai filtrai išsamiai nagrinėjami Veikerto knygoje [10], taip pat gerą

vaizdų apdorojimo metodų, naudojančių diferencialines lygtis su dalinėmis išvestinėmis, apžvalgą yra pateikęs Mikula [29].

3.3.2. Netiesiniai difuziniai filtrai

Kai kuriais atvejais yra svarbu, kad vaizdo kelių mastelių (angl. *multiscale*) analizė lokaliai priklausytų ne tik nuo intensyvumo funkcijos u reikšmių, bet ir nuo padėties vaizde X. Pavyzdžiui, vartotojui gali būti reikalinga taikyti skirtingų greičių difuzinius procesus skirtingose vaizdo vietose arba priklausomai nuo intensyvumo reikšmės. Tokiais atvejais galima naudoti tokius netiesinius difuzinius uždavinius [71]:

$$\frac{\partial b(X,u)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g(|\nabla G_\sigma * \beta(X,u)|), \frac{\partial \beta(X,u)}{\partial x_i}\Big) + f(u_0 - u)\,, \quad (3.31)$$

arba

$$\frac{\partial b(X,u)}{\partial t} = \sum_{i=1}^{2} \frac{\partial}{\partial x_i}\Big(g(|\nabla G_\sigma * b(X,u)|), \frac{\partial \beta(X,u)}{\partial x_i}\Big) + f(u_0 - u)\,. \quad (3.32)$$

Taškuose, kur išvestinė β'_u yra maža (b'_u didelis), difuzijos procesas slopinamas, tuo tarpu srityse, kur β'_u yra didelis (b'_u mažas), šis procesas pagreitinamas. Įdomūs tokių netiesinės difuzijos uždavinių taikymo pavyzdžiai pateikti [29, 71].

3.3.3. Baigtinių skirtumų aproksimacija

Dažniausiai vaizdas apibrėžiamas stačiakampio formos struktūroje. Nagrinėkime diskretųjį erdvinį tinklą

$$\omega_h = \{\, (x_{1i}, x_{2j}) : \;\; x_{\alpha k} = kh,\; k = 0, 1, \ldots, N_\alpha \,\}.$$

Taip pat naudosime tolygųjį (angl. *uniform*) laiko tinklelį

$$\omega_\tau = \{\, t^n : \;\; t^n = n\tau,\; n = 0, 1, \ldots, M \,\}.$$

Diskretinėms funkcijoms ir baigtiniams skirtumams naudojami tokie žymėjimai:

$$U_{ij}^n = U(x_{1i}, x_{2j}, t^n), \quad (x_{1i}, x_{2j}, t^n) \in \omega_h \times \omega_\tau \,,$$

$$\partial_t U_{ij}^n = \frac{U_{ij}^n - U_{i,j}^{n-1}}{\tau}, \quad \partial_{x_1}^- U_{ij}^n = \frac{U_{ij}^n - U_{i-1,j}^n}{h}, \quad \partial_{x_1}^+ U_{ij}^n = \frac{U_{i+1,j}^n - U_{ij}^n}{h},$$

$$\partial_{x_2}^- U_{ij}^n = \frac{U_{ij}^n - U_{i,j-1}^n}{h}, \quad \partial_{x_2}^+ U_{ij}^n = \frac{U_{i,j+1}^n - U_{ij}^n}{h}.$$

A. Išreikštinė aproksimacija

Baigtinių tūrių metodu aproksimuojame erdvines išvestines, laiko išvestinę aproksimuojame tiesiogine Eulerio formule ir imdami difuzinius ir netiesinius (3.30) lygties narius iš paskutinio laiko žingsnio, gauname išreikštinę schemą

$$\partial_t U_{ij}^{n+1} = \sum_{\alpha=1}^{2} \partial_{x_\alpha}^+ \left(a_\alpha(U_{ij}^n)\, \partial_{x_\alpha}^- U_{ij}^n \right) + f\left(u_{0,ij} - U_{ij}^n\right). \tag{3.33}$$

Čia a_α apibrėžia netiesinio difuzijos koeficiento diskrečiąją aproksimaciją, pavyzdžiui:

$$a_{1,i+1/2,j} = g\Big(\Big((\partial_{x_1}^+ V_{ij}^n)^2 + \Big(\frac{\partial_{x_2}^+ V_{ij}^n + \partial_{x_2}^- V_{ij}^n + \partial_{x_2}^+ V_{i+1,j}^n + \partial_{x_2}^- V_{i+1,j}^n}{4} \Big)^2 \Big)^{1/2} \Big),$$

čia $V_{ij}^n = G_\sigma * U_{ij}^n$. Šios schemos konvergavimas ištirtas [30] darbe.

B. Pusiau neišreikštinė aproksimacija

Išreikštinė schema (3.33) yra stabili tik tuo atveju, kai $\tau \leqslant ch^2$. Norint gauti besąlygiškai stabilią (tiesinės stabilumo sąlygos atžvilgiu) schemą, galima nagrinėti tokią pusiau neišreikštinę aproksimaciją:

$$\partial_t U_{ij}^{n+1} = \sum_{\alpha=1}^{2} \partial_{x_\alpha}^+ \left(a_\alpha(U_{ij}^n)\, \partial_{x_\alpha}^- U_{ij}^{n+1} \right) + f\left(u_{0,ij} - U_{ij}^n\right). \tag{3.34}$$

Šios schemos konvergavimo analizė pateikta [68] darbe. Kai kurias neišreikštines ir pusiau neišreikštines baigtinių skirtumų schemas, skirtas netiesinių parabolinių uždavinių sprendimui, pasiūlė ir ištyrė Čiegis *ir kiti* [52, 54].

Kiekvienos iteracijos metu gaunama išretinta tiesinių lygčių sistema, kurią sprendžiame iteraciniu JG algoritmu. Gerai žinoma, kad iteraciniai metodai lygiagretinami žymiai paprasčiau, negu tiesioginiai metodai.

Užrašykime tiesinę lygčių sistemą vienai (3.34) iteracijai spręsti:

$$AV = F\,, \quad V = U^{n+1}\,, \tag{3.35}$$

tada JG algoritmas su modifikuojančia matrica (angl. *Preconditioned Conjugate Gradient*, arba PCG) gali būti užrašytas tokiu būdu [40]:

procedure `Nuoseklusis PCG algoritmas`
begin
(1) $V^0, \quad n = 0, \quad R^0 = AV^0 - F,$
(2) $BW^0 = R^0, \quad P^0 = W^0\,.$
(3) **while** $\left((W^n, R^n) > \varepsilon\,(W^0, R^0) \right)$
(4) $G^n = AP^n\,,$
(5) $\tau_{n+1} = \dfrac{(W^n, R^n)}{(G^n, P^n)}\,,$
(6) $V^{n+1} = V^n - \tau_{n+1} P^n\,,$
(7) $R^{n+1} = R^n - \tau_{n+1} G^n\,,$
(8) $BW^{n+1} = R^{n+1}\,.$
(9) $\beta_n = \dfrac{(W^{n+1}, R^{n+1})}{(W^n, R^n)}\,,$
(10) $P^{n+1} = W^{n+1} + \beta_n P^n\,,$
(11) $n := n + 1\,.$
(12) **end while**
end

Čia B yra modifikuojanti matrica. Bus naudojamos tik diagonalinės modifikuojančios matricos, nes tokios matricos gali būti labai efektyviai lygiagretinamos. Žinomos ir efektyvesnės modifikuojančios matricos, pavyzdžiui, nepilnos ILU faktorizacijos modifikuojanti matrica [40], tačiau jos yra nuosekliosios iš prigimties. Vykdoma daug tyrimų, susijusių su įvairių modifikuojančių matricų panaudojimu, norint pasiekti balansą tarp nepilnos faktorizacijos konvergavimo greičio ir lygiagretinimo efektyvumo (pvz., [1, 58]). Lygiagrečiojo PCG algoritmo išplečiamumo analizė pateikta [9, 59] darbuose.

3.3.4. Algoritmų lygiagretinimo efektyvumo įvertinimas

Aukščiau aprašyti algoritmai buvo realizuoti taikant ParSol biblioteką, gautos tiek nuoseklioji, tiek ir lygiagrečioji (po nedidelių pusiau automatinių pakeitimų) realizacijos, jomis buvo atliekami eksperimentai. Šiame skirsnyje pateikti skaičiavimo eksperimentų rezultatai. Eksperimentai buvo atlikti Vilniaus Gedimino

Technikos universiteto kompiuterių klasteryje „Vilkas“ ir IBM SP4 superkompiuteryje CINECA, Bolonijoje. Rezultatai skelbti [94, 95, 102] darbuose.

A. Išreikštinis netiesinis algoritmas (3.33)

Buvo filtruojamas dirbtinis $N \times N$ dydžio paveikslas. Pirmiausia, pažvelkime į išreikštinio netiesinio algoritmo (3.33) realizaciją. 3.2 lentelėje pateiktos eksperimentiškai gautos lygiagrečiojo algoritmo pagreitėjimo $S_p(N)$ ir efektyvumo $E_p(N)$ reikšmės. Kompiuterių klasteryje „Vilkas“ buvo sprendžiami įvairių dydžių uždaviniai. Čia p yra procesų skaičius,

$$S_p(N) = \frac{T_1(N)}{T_p(N)}, \quad E_p(N) = \frac{S_p(N)}{p},$$

o $T_p(N)$ yra CPU laikas, reikalingas tam, kad išspręstume uždavinį naudodami p procesus.

Buvo naudojama dvimatė procesų topologija $p_1 \times p_2$, p_1 ir p_2 reikšmės parenkamos kuo artimesnės viena kitai. Jeigu p yra pirminis skaičius, gaunama vienmatė topologija. Vaizdas buvo apdorojamas iki laiko momento $T(N)$,

$$T(160) = 0{,}1\,, \quad T(240) = 0{,}03\,, \quad T(320) = 0{,}01\,,$$

ir nuosekliuoju atveju buvo gauti tokie CPU laikai $T_1(N)$ (*sekundėmis*)

$$T_1(160) = 213{,}3\,, \quad T_1(240) = 332{,}8\,, \quad T_1(320) = 361{,}6\,.$$

3.2 lentelė. *Išreikštinio algoritmo (3.33) pagreitėjimas ir efektyvumas PC klasteryje „Vilkas“*

p	$S_p(160)$	$E_p(160)$	$S_p(240)$	$E_p(240)$	$S_p(320)$	$E_p(320)$
2	1,56	0,780	1,76	0,880	1,87	0,934
4	2,36	0,590	3,00	0,750	3,45	0,862
6	2,78	0,463	3,93	0,655	4,77	0,795
8	2,95	0,369	4,69	0,585	5,88	0,735
9	3,16	0,351	5,04	0,560	6,28	0,698
11	3,33	0,303	5,50	0,500	7,09	0,644
12	3,35	0,279	5,64	0,470	7,47	0,623
15	3,39	0,226	6,38	0,425	8,56	0,571

3.3 lentelėje pateiktos eksperimentiškai gautos lygiagrečiojo algoritmo pagrei-

tėjimo $S_p(N)$ ir efektyvumo $E_p(N)$ reikšmės, kai buvo sprendžiami tie patys uždaviniai SP4 superkompiuteryje. Nuosekliojo algoritmo skaičiavimo laikai $T_1(N)$ (*sekundėmis*)

$$T_1(80) = 57{,}24\,, \quad T_1(160) = 471{,}2\,, \quad T_1(320) = 770{,}4\,.$$

3.3 lentelė. *Išreikštinio algoritmo (3.33) pagreitėjimas ir efektyvumas SP4 superkompiuteryje*

p	$S_p(80)$	$E_p(80)$	$S_p(160)$	$E_p(160)$	$S_p(320)$	$E_p(320)$
2	1,975	0,988	1,984	0,992	2,004	1,002
3	2,794	0,931	2,950	0,985	2,970	0,990
4	3,741	0,935	3,928	0,982	3,986	0,996
6	5,168	0,861	5,463	0,910	5,916	0,986
8	6,766	0,846	7,293	0,911	7,831	0,979
9	6,784	0,754	7,604	0,845	8,467	0,941
12	8,701	0,725	10,19	0,849	11,216	0,934
16	10,84	0,677	12,75	0,797	15,041	0,940
24	14,18	0,591	18,24	0,760	21,961	0,915

B. Sudėtingumo analizė

Šiame skirsnyje bus įvertintas diskrečiosios schemos (3.33) lygiagrečiosios versijos sudėtingumas. Nuoseklaus algoritmo sudėtingumas apskaičiuojamas pagal formulę

$$W = \gamma M N^2\,,$$

čia M yra iteracijų skaičius, $N \times N$ yra nuotraukos dydis, o γ yra CPU laikas, per kurį atliekama viena algoritmo elementarioji operacija.

Lygiagretusis algoritmas pagrįstas duomenų skaidymo (angl. *domain decomposition*) metodu. Išskaidymą automatiškai atlieka lygiagrečiųjų masyvų objektai. Dvimatis paveikslas gali būti dalinamas tarp procesų naudojant 1D arba 2D dekartinį paskirstymą (angl. *Cartesian mapping*).

Tegu $p_1 \times p_2$ yra procesų topologija. Tada lygiagrečiojo algoritmo (3.33) skaičiavimo sudėtingumas priklauso nuo duomenų masyvo didžiausios dalies, tenkančios vienam procesui, dydžio. Tai yra

$$T_{p,comp} = \gamma M \left\lceil \frac{N}{p_1} \right\rceil \times \left\lceil \frac{N}{p_2} \right\rceil,$$

čia $\lceil x \rceil$ nurodo mažiausią sveiką skaičių, didesnį arba lygų už x. Lygiagretaus algoritmo efektyvumas priklauso nuo paskirstytų tarp procesų paveikslo dalių dydžių disbalanso. Apibrėžkime papildomus lygiagrečiojo algoritmo skaičiavimo kaštus

$$H_{comp}(p_1, p_2) = p\,T_{p,comp} - W\,.$$

Visada svarbu mažinti šį disbalansą, pasirenkant tinkamą procesų paskirstymą. Šį teiginį galima pademonstruoti dviem paprastais atvejais. Pirmiausia, tarkime, kad $N = 84$ ir $p = 16$. Trims skirtingiems procesų paskirstymams gauname tokius lygiagrečiojo algoritmo papildomus skaičiavimo kaštus

$$H_{comp}(1, 16) = 504M\gamma, \quad H_{comp}(2, 8) = 336M\gamma, \quad H_{comp}(4, 4) = 0\,.$$

Darome prielaidą, kad lygiagrečioji algoritmo versija nesukelia papildomų skaičiavimo kaštų. Tada lygiagrečiojo algoritmo efektyvumas gali būti užrašytas taip:

$$E_{p_1 \times p_2}(N) = \frac{W}{W + H_{comp}(p_1, p_2)} = \frac{1}{1 + H_{comp}(p_1, p_2)/W}\,.$$

Vadinasi, gauname tokias koeficientų reikšmes:

$$E_{1\times 16}(84) = 0933, \quad E_{2\times 8}(84) = 0{,}955, \quad E_{4\times 4}(84) = 1\,.$$

Kitas pavyzdys parodo, kad tam tikrais atvejais lygiagretusis algoritmas gali būti greitesnis, jei naudojame mažiau procesų. Tegu $N = 80$, tada palyginkime pagreitėjimą, kai turime 12 ir 13 procesų:

$$S_{3\times 4}(80) = 11{,}85, \quad S_{1\times 13}(80) = 11{,}43\,.$$

Papildomi kaštai atsiranda bet kokiame lygiagrečiajame algoritme, kai vienas procesas siunčia duomenis kitam. Tarkime, du procesai apsikeičia n elementais. Tai gali būti atlikta per laiką $\alpha + \beta n$ naudojant, pavyzdžiui, *lyginį-nelyginį* duomenų apsikeitimo algoritmą [31]. Čia α yra persiuntimo iniciacijos kaštai, o β yra laikas, reikalingas vieno duomenų vieneto persiuntimui.

C. Komunikacijos kaštų analizė

Šiame skirsnyje nagrinėsime, kaip duomenų perdavimo kaštai priklauso nuo procesų skaičiaus ir nuo pasirinktos jų paskirstymo topologijos. Apsiribosime tik *lyginio-nelyginio* algoritmo analize ir laikysime, kad procesus jungiantis tinklas turi dvimačio toro topologiją.

Tegu paveikslo dydis yra $N \times N$, o M yra iteracijų skaičius. Pirmiausia,

nagrinėsime 1D topologiją iš $p \times 1$ procesų. Tada duomenų apsikeitimo laikas yra

$$K_{p\times 1}(N) = (2\alpha + 2\beta N)M\,.$$

$\frac{p}{2} \times 2$ topologijos atveju šitie papildomi algoritmo kaštai tampa lygūs

$$K_{p/2\times 2}(N) = \Big(3\alpha + \frac{p+2}{p}\beta N\Big)M\,.$$

Jei procesai yra paskirstyti $\sqrt{p} \times \sqrt{p}$ būdu, tada duomenų apsikeitimo laikas yra

$$K_{\sqrt{p}\times\sqrt{p}}(N) = \Big(4\alpha + \frac{4N}{\sqrt{p}}\beta\Big)M\,.$$

Taigi, toro topologijoje komunikacijos kaštai mažėja, didėjant procesų skaičiui.

Toliau pateikti skaičiavimo eksperimento rezultatai PC klasteryje „Vilkas" ir IBM SP4 superkompiuteryje. Reikia pabrėžti, kad skaičiavimo greitis viename PC mazge yra dvigubai didesnis, nei SP4 mazgo greitis. Taip pat gerai žinoma, kad šio PC klasterio tinklui būdingas labai didelis užlaikymas (angl. *latency*), t.y. $\alpha \gg \beta$. Tai paaiškina, kodėl lygiagretaus algoritmo efektyvumas yra didesnis IBM SP4 superkompiuteryje ir kodėl pagreitėjimas S_p prisisotina greičiau PC klasteryje.

3.4 lentelėje pateikti algoritmų, vykdytų PC klasteryje, papildomi kaštai. Kaip buvo pažymėta ankščiau, jie sutampa su duomenų perdavimo kaštais. Kadangi iteracijų skaičius keičiasi priklausomai nuo N, rezultatai pakeisti proporcingai $N = 160$ atvejo atžvilgiu.

3.4 lentelė. *Išreikštinio algoritmo (3.33) duomenų apsikeitimo kaštai PC klasteryje*

p	$K_p(160)$	$K_p(240)$	$K_p(320)$
2	29,8	34,2	32,1
4	37,0	41,8	36,0
8	41,3	44,0	40,8
12	45,8	46,9	45,7

Iš rezultatų, pateiktų 3.4 lentelėje, galima matyti, kad komunikavimo kaštai tik nežymiai priklauso nuo N, vadinasi, dominuoja pranešimo parengimo kaštai α. Iš šio fakto išplaukia, kad, sprendžiant mažesnį 80×80 dydžio uždavinį, praktiškai negausime jokio pagreitėjimo. Nuoseklaus algoritmo skaičiavimo laikas yra $T_1(80) = 26{,}03$ (iteracijos vykdomos iki $T = 0{,}2$). Taikant šį teorinį modelį,

galima prognozuoti tokį lygiagretaus algoritmo sudėtingumą:

$$T_{2,pred}(80) = \frac{26{,}03}{2} + \frac{29{,}8}{2} = 27{,}9, \quad T_{4,pred}(80) = \frac{26{,}03}{4} + \frac{37}{2} = 25{,}0,$$
$$T_{8,pred}(80) = \frac{26{,}03}{8} + \frac{41{,}3}{2} = 23{,}9.$$

Skaičiavimo eksperimentai patvirtino šiuos teorinius įverčius:

$$T_2(80) = 27{,}35, \quad T_4(80) = 24{,}51, \quad T_8(80) = 24{,}32.$$

D. Išplečiamumo analizė

Bet kokio lygiagretaus algoritmo išplečiamumo analizė leidžia surasti greitį, kuriuo uždavinio dydis W turi augti procesų skaičiaus p atžvilgiu tam, kad išliktų pastovus algoritmo efektyvumas. Izoefektyvumo funkcija $W = g(p, E)$ apibrėžiama neišreikštine lygtimi [31]

$$W = \frac{E}{1-E} H(p, W)\,. \tag{3.36}$$

Paprastumo dėlei imsime $E = 0{,}5$.

Nagrinėkime procesų topologiją $p \times 1$. Bendrieji papildomi diskrečios schemos (3.33) lygiagretaus algoritmo kaštai nustatomi pagal duomenų perdavimo kaštus, vadinasi $H(p, W) = (2\alpha + 2\beta N)M$. Dėl išreikštinės schemos stabilumo reikalavimo turime $M = cN^2$, tada gauname lygtį

$$\gamma c N^4 = 2\alpha c N^2 p + 2c\beta N^3 p$$

arba

$$W = \frac{2\alpha\sqrt{c}}{\sqrt{\gamma}} pW^{1/2} + \frac{2\beta c^{1/4}}{\gamma^{3/4}} pW^{3/4}\,.$$

Šiuo atveju galime gauti izoefektyvumo funkciją kaip išraišką nuo p, tačiau patogiau analizuoti kiekvieno atskiro nario įtaką. Ta dalis, kuri priverčia uždavinio dydį augti sparčiausiai, bendrai apsprendžia asimptotinį izoefektyvumo funkcijos pavidalą. 1D procesų topologijai asimptotinę izoefektyvumo funkciją apsprendžia antras papildomų kaštų funkcijos narys

$$W = \frac{16c\beta^4}{\gamma^3} p^4 = \mathcal{O}(p^4)\,.$$

Jis reikalauja, kad N turi didėti tiesiškai p atžvilgiu tam, kad būtų palaikomas tam tikras fiksuotas efektyvumas E.

Dabar nagrinėkime $\sqrt{p} \times \sqrt{p}$ procesų topologiją. Šiuo atveju izoefektyvumo funkcija apibrėžiama lygtimi

$$W = \frac{4\alpha\sqrt{c}}{\sqrt{\gamma}}\, pW^{1/2} + \frac{2\beta c^{1/4}}{\gamma^{3/4}}\, \sqrt{p}\, W^{3/4}\,.$$

Dabar abi komponentės reikalauja, kad problemos dydis augtų panašiu greičiu, ir bendra asimptotinė izoefektyvumo funkcija atrodo taip:

$$W = \mathcal{O}(p^2)\,.$$

Ji reikalauja, kad matricos dydis $N \times N$ augtų tiesiškai p atžvilgiu, tada bus išlaikomas tam tikras vienodas efektyvumas E.

E. Pusiau neišreikštinis netiesinis algoritmas (3.34)

Šiame skirsnyje analizuojama lygiagrečioji pusiau-neišreikštinio netiesinio algoritmo (3.34) realizacija. Pagrindiniai skaičiavimo žingsniai yra tokie:

- Pradinis vaizdo glodinimas (angl. *presmoothing*). Atliekami keli tiesinės išreikštinės schemos žingsniai.
- Sprendžiama tiesinių lygčių sistema panaudojant iteracinį Jungtinių Gradientų metodą.

Viena JG metodo iteracija reikalauja tokių skaičiavimų:

- Apskaičiuoti matricos-vektoriaus sandaugą, kuri yra ekvivalenti išreikštinės skirtuminės schemos taikymui;
- Globali redukcijos operacija, kai skaičiuojama dviejų vektorių skaliarinė sandauga. Ši operacija jau įdėtą į *ParSol* įrankį.

Lygiagretaus JG metodo su modifikuojančia matrica išplečiamumo analizė atlikta [9, 59] darbuose. 3.5 lentelėje pateikti CPU laikai $T_p(N)$ (*sekundėmis*), reikalingi, norint išspręsti šį vaizdo apdorojimo uždavinį SP4 superkompiuteryje, esant skirtingiems paveikslo dydžiams. Superpagreitėjimo efektas pasiekiamas dėl SP4 architektūros, kuri leidžia efektyviau dirbti su mažesnio dydžio masyvais. Šis efektas gali būti išvengtas, naudojant kompiuterių architektūrai būdingą optimizavimą nuosekliojo kodo greitinimui. Tačiau tokio tipo optimizavimas yra sudėtingas (tam net kuriamos specialios bibliotekos, pavyzdžiui, ATLAS [11]), ir šiuo metu dar nėra realizuotos ParSol bibliotekoje. Jo būtinumas yra žinomas, ir tai yra viena iš prioritetinių bibliotekos vystymo krypčių, kaip aprašyta 2.6.2 poskyryje.

3.5 lentelė. *Neišreikštinio netiesinio algoritmo (3.34) CPU laikai* $T_p(N)$

p	$T_p(160)$	$T_p(320)$	$T_p(480)$
1	64,97	241,4	281,9
2	26,82	86,71	118,5
4	12,89	40,37	63,94
6	9,24	26,91	42,84
8	7,59	21,37	32,44
16	4,83	10,30	16,44

3.3.5. Kompiuterinės tomografijos vaizdų apdorojimas

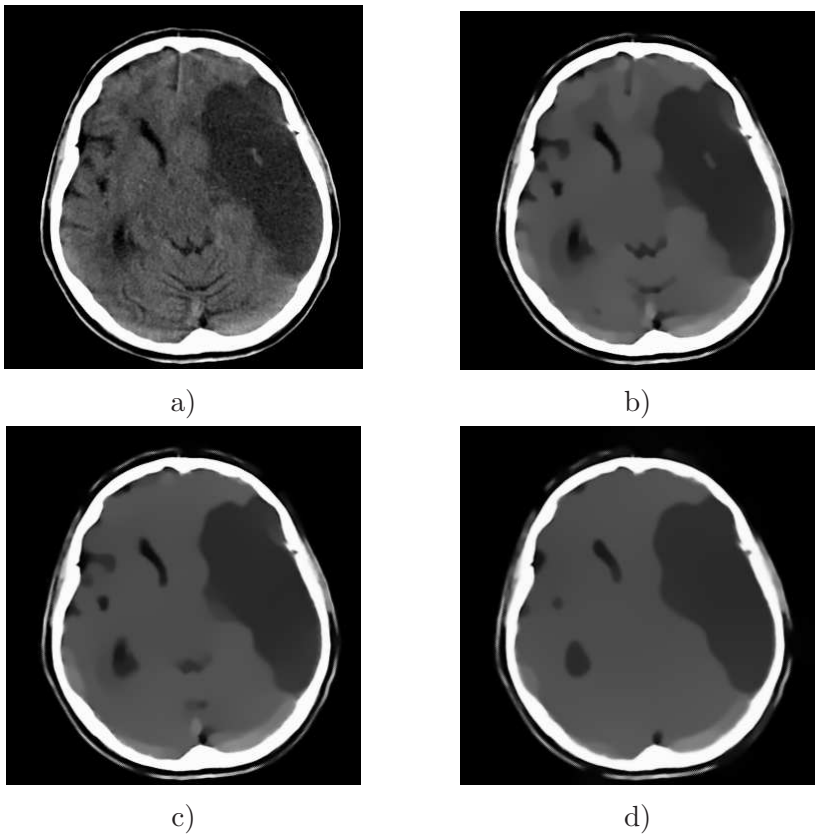
Netiesiniai difuziniai filtrai gali būti taikomi apdorojant KT vaizdus. Viena iš svarbių KT problemų yra patikimas insulto sričių nustatymas. Tiksliai išspręstas, šis uždavinys leidžia įvertinti insulto apimtį ir padeda medikams geriau pasirinkti gydymo metodiką. Šis uždavinys yra tiriamas [6, 67, 88] darbuose, skirtingos jo sprendimo metodikos nagrinėjamos [32, 41] disertacijose.

Smūgio sritys KT nuotraukose gali būti įvairaus dydžio ir formos, tačiau bet kokiu atveju pažeistos srities imlumas sorbcijai (angl. *sorption susceptibility*) yra 1,5–2 kartus didesnis. Tokios nuotraukos pavyzdys pateiktas 3.6*a* paveiksle, nuotraukos dydis 512×512 taškų.

Apdorojant KT vaizdus, yra svarbu nepakeisti smūgio srities kontūrų, nes jie naudojami apskaičiuojant smūgio srities tūrį. Ši informacija yra svarbi medikams. Vienas iš *netiesinių filtrų* privalumų yra tas, kad jie išsaugoja vaizdo kontūrus. 3.6 *b,c,d* paveiksluose parodomi KT nuotraukų filtravimo netiesiniais difuziniais filtrais rezultatai po 20, 40 ir 100 iteracijų.

Kaip galima matyti iš filtravimo rezultatų, vaizdo kontūrai išlaiko savo formą, vadinasi, smūgio srities lokalizacija įmanoma, taikant standartinę procedūrą (pvz., diferencialinius filtrus). Tokiai lokalizacijai net nereikia atlikinėti 100 iteracijų, nes geri rezultatai matosi ir su mažesniu kiekiu. Tačiau iš anksto nustatyti pakankamą iteracijų skaičių nėra lengvas uždavinys. Todėl svarbu pastebėti (žr. 3.6 *c,d* paveikslus), kad ir su dideliu iteracijų skaičiumi netiesiniai filtrai nekenkia vaizdo kontūrams. Tai palengvina jų panaudojimą kuriant automatinio smūgio sričių atpažinimo įrankius.

Tyrimo metu buvo bandoma parinkti tokius netiesinius difuzinius filtrus ir jų parametrus, kad jie tiktų KT vaizdų apdorojimui, ir atliktų šią funkciją kuo greičiau



3.6 pav. *Žmogaus smegenų insulto KT nuotrauka (smūgio sritis tamsesnė): a) pradinė nuotrauka, b) po 20 iteracijų, c) po 40 iteracijų, d) po 100 iteracijų*

ir efektyviau. Efektyvumas buvo vertinamas statistiškai, atliekant didelį eksperimentų skaičių.

3.4. Variacinių iteracinių tiesinių lygčių sistemų sprendimo metodų lygiagretinimas

Šiame skirsnyje aprašomas tiesinės algebros objektų bibliotekos ParSol taikymas lygiagretinant įvairius variacinius iteracinius tiesinių lygčių sistemų sprendimo metodus. Pradžioje pateikiami paprasti gerai žinomi metodai, vėliau jų pagrin-

du aprašomi kiti, sudėtingesni metodai. Pateikta kai kurių metodų konvergavimo greičio ir lygiagretinimo efektyvumo analizė. Galiausiai pateikti eksperimentiniai lygiagretinimo efektyvumo tyrimo rezultatai.

Baigtinių skirtumų arba baigtinių tūrių metodu spręsdami elipsinių arba parabolinių diferencialinių lygčių dalinėmis išvestinėmis sistemas, gauname tokio pavidalo tiesinių lygčių sistemas:

$$AX = F\,, \tag{3.37}$$

čia A yra teigiamai apibrėžta $A > 0$, simetrinė $N \times N$ dimensijos matrica, F yra žinomas vektorius, o X yra nežinomųjų vektorius. Tokių tiesinių lygčių sistemų matricos yra išretintos ir dažnai būna juostinės.

Efektyvus tokių sistemų sprendimas yra svarbi matematinio modeliavimo dalis. Iteraciniai metodai pasinaudoja tuo, kad lygčių sistemų matricos yra išretintos [70, p. 88], todėl jie naudojami daugumoje šiuolaikinių skaičiavimo programų ir bibliotekų. Taip pat atkreipkime dėmesį, kad iteraciniai algoritmai gali būti lengviau lygiagretinami, negu tiesioginiai tiesinių lygčių sistemų sprendimo algoritmai.

Tarp iteracinių algoritmų ypač populiarios yra įvairios Krylovo poerdvių metodų modifikacijos. Taikant šias metodikas, galima sukurti sprendimo metodus, pasižyminčius labai geromis savybėmis. Svarbu pastebėti, kad daugelis iteracinių Krylovo tipo algoritmų modifikacijų gali spręsti tiesinių lygčių sistemas su nesimetrinėmis matricomis [40]. Tai svarbu sprendžiant poringų terpių modeliavimo uždavinius.

3.4.1. Vienažingsniai variaciniai iteraciniai metodai

Siekiant sudaryti pilną vaizdą, trumpai aprašysime bendrą vienažingsnių variacinių iteracinių algoritmų schemą. Gerai žinoma, kad tiesinių lygčių sistemos (3.37) sprendimas yra ekvivalentus kvadratinio funkcionalo

$$Q(Y) = (A(Y - X), Y - X) \tag{3.38}$$

minimumo uždavinio sprendimui [40], [70, p. 114]. Čia (X, Y) yra dviejų vektorių skaliarinė sandauga. Nagrinėkime variacinį iteracinį metodą, kuris apibrėžiamas lygtimi

$$X^{n+1} = X^n - \tau_n P^n, \quad n \geq 0\,, \tag{3.39}$$

kai duotas tam tikras pradinis artinys X^0. P^n yra žinomi vektoriai, kurie nurodo kryptį funkcionalo $Q(X^{n+1})$ minimizavimui. Išsprendę kvadratinę lygtį pagal τ_n,

mes gauname optimalią šio parametro reikšmę

$$\tau_n = \frac{(AX^n - F, P^n)}{(AP^n, P^n)}. \tag{3.40}$$

Priklausomai nuo P^n vektorių parinkimo, egzistuoja daug tokio tipo algoritmų, tarp jų labiausiai taikomi *Didžiausio nuolydžio* (angl. *Steepest Descent*, arba SD) ir *Mažiausių Netikčių* (angl. *Minimal Residuals*, sutrumpintai MINRES arba MR) metodai [40].

Tegu A yra simetrinė ir teigiamai apibrėžta matrica, kurios tikrinės reikšmės yra

$$0 < \lambda_1 \leq \lambda_1 \leq \cdots \leq \lambda_N.$$

Tada $\kappa(A) = \lambda_N/\lambda_1$ yra matricos A sąlygotumo skaičius (angl. *condition number*). Sprendžiant antros eilės elipsines diferencialines lygtis dalinėmis išvestinėmis, šis skaičius yra $\kappa(A) = \mathcal{O}(h^{-2}) \gg 1$, čia h yra diskrečiojo tinklo žingsnis [81]. Apibrėžkime vektoriaus normą tokiu būdu:

$$\|Y\|_A = (AY,\, Y)^{1/2}.$$

Didžiausio nuolydžio metodo konvergavimo greitis yra tiesinis, ir jam galioja toks paklaidos įvertis [40]:

$$\|X^{n+1} - X\|_A \;\leq\; \rho \|X^n - X\|_A,$$

$$\rho = \frac{1-\eta}{1+\eta}, \quad \eta = \frac{1}{\kappa(A)}.$$

Taigi, įteracijų skaičius K, reikalingas, norint sumažinti pradinės aproksimacijos paklaidą $1/\varepsilon$ kartų, t.y. gauti aproksimaciją

$$\|X^K - X\|_A \;\leq\; \varepsilon \|X^0 - X\|_A,$$

apskaičiuojamas iš $K = \frac{1}{2}\kappa(A)\ln(1/\varepsilon)$ formulės.

Panašūs įverčiai galioja ir mažiausių netikčių metodui, tik šiuo atveju paklaida vertinama

$$\|A(X^{n+1} - X)\| = \|R^n\|$$

normoje.

Žymiai didesnis konvergavimo greitis gaunamas, kai taikomas *Jungtinių Gradientų* metodas (CG). Gaunamos iteracijos X^n yra geriausia tiksliojo lygčių siste-

mos (3.37) sprendinio X aproksimacija Krylovo erdvėje

$$K_n(A;F) = \text{ span } \{F, AF, \ldots, A^{n-1}F\},$$

kuri susideda iš vektorių $\{A^iF,\ i = 0, \ldots, n-1\}$ tiesinių kombinacijų. Paklaida įvertinama taip [40]:

$$\|X^n - X\|_A \ \leq\ 2\left(\frac{1-\sqrt{\eta}}{1+\sqrt{\eta}}\right)^n \|X^0 - X\|_A.$$

Iteracijų skaičius K_{CG}, reikalingas, norint sumažinti pradinę paklaidą $1/\varepsilon$ kartų, išreiškiamas formule $K = \frac{1}{2}\sqrt{\kappa(A)}\ln(2/\varepsilon)$.

Užrašykime iteracinį algoritmą kanoniniu pavidalu

$$X^n = C_n X^{n-1} + \tau_n F,$$

čia $C_n = I - \tau_n A$. Norint sukurti efektyvius iteracinius algoritmus ir ištirti jų konvergavimo greitį, naudojami du metodai. Pirmas metodas grindžiamas operatoriaus

$$\min_{\tau_1,\ldots,\tau_k} \max_{a_m \leq \lambda \leq a_M} |(1-\tau_1\lambda)\cdots(1-\tau_k\lambda)|$$

spektro *minimax* įverčiais, čia a_m, a_M yra matricos A spektriniai įverčiai. Išreikštinė parametrų $\tau_1, \ldots, \tau_k$ seka apskaičiuojama taikant Čebyševo polinomų šaknis [81]. Pagrindinis šio metodo privalumas yra tas, kad parametrai apskaičiuojami *a priori*, jeigu žinomi matricos A spektro įverčiai. Galima pastebėti, kad minimizavimas atliekamas atsižvelgiant į blogiausią spektro pasiskirstymo atvejį, kuris nebūtinai galioja matricai A.

Antros grupės iteraciniai metodai grindžiami variaciniu tiesinių lygčių sistemų formulavimu. Šitie algoritmai naudoja *godžius* metodus, t.y. jie mažina paklaidą tam tikra kryptimi. Gerai žinoma, kad variacinių algoritmų konvergavimo greitis sumažėja po kiekvienos iteracijos, ir globalios paklaidos vektorius konverguoja asimptotiškai prie tiesinės kombinacijos dviejų matricos A tikrinių vektorių, kurie atitinka didžiausią ir mažiausią tikrinę reikšmę. Variacinių iteracinių algoritmų konvergavimo greitis įvertinamas iš ankščiau minėtų formulių, panaudojant asimptotines sugeneruotos iteracijų sekos savybes.

3.4.2. Įvairios standartinių metodų modifikacijos

Šiame skirsnyje aprašomi metodai, gaunami modifikuojant ankščiau aprašytus metodus.

A. Dvipakopis gradientinio nuolydžio metodas

Dvipakopis gradientinio nuolydžio metodas (angl. *two step gradient descent*, arba TSGD) yra SD ir MR metodų kombinacija. Šis metodas buvo pasiūlytas [85] darbe, kur išvados dėl jo konvergavimo greičio buvo paremtos skaitiniais eksperimentais, tame tarpe ir sprendžiant 2D elipsinius uždavinius. [53] darbe buvo iš dalies paaiškintas šio metodo supertiesinis konvergavimas, ir detaliai jis buvo ištirtas [97].

Didžiausio nuolydžio metodas gaunamas, kai vektoriai $P^n = \nabla Q(X^n)$ sutampa su funkcionalo Q gradientu. Pažymėkime netiktį $R^n = AX^n - F$, tada SD metodas iš (3.38) ir (3.40) apibrėžiamas taip:

$$X^{n+1} = X^n - \tau_n R^n, \quad \tau_n = \frac{(R^n,\, R^n)}{(AR^n,\, R^n)}\,. \tag{3.41}$$

Mažiausių netikčių metodo atveju minimizuojamas kvadratinis netikčių funkcionalas $Q_2(Y) = (AY - F, AY - F)$. Šiuo atveju iteracinis žingsnis sudaromas tokiu būdu:

$$X^{n+1} = X^n - \tau_n R^n, \quad \tau_n = \frac{(AR^n,\, R^n)}{(AR^n,\, AR^n)}\,. \tag{3.42}$$

Nelyginiuose TSGD metodo žingsniuose naudojamas didžiausio nuolydžio, o lyginiuose žingsniuose taikomas mažiausių netikčių metodai. Tokiu būdu, TSGD algoritmas gali būti užrašytas taip:

TSGD metodas

TSGD (Matr A, Vec x, Vec f, ε)
begin
(1) Nustatome pradines reikšmes $i = 0, \quad r = Ax - f$;
(2) **while** ($\|r\| > \varepsilon$) **do**
(3) $v = Ar$;
(4) **if** ($i == 0$) **then**
(5) $\tau = (r, r)/(v, r)$;
(6) $i = 1$;
else
(7) $\tau = (v, r)/(v, v)$;
(8) $i = 0$;
end if
(9) $x := x - \tau\, r; \quad r := r - \tau\, v$;
end do
end TSGD

TSGD metodo teorinė konvergavimo greičio analizė gali būti padalinta į du žingsnius. *Pirmiausiai*, panagrinėkime atvejį, kai pradinio vektoriaus X^0 netikties vektorius $R^0 = AX^0 - F$ turi pavidalą

$$R^0 = c_1 V_1 + c_N V_N,$$

čia V_j yra matricos A tikriniai vektoriai, atitinkantys λ_j tikrines reikšmes. Nesudėtingi skaičiavimai parodo, kad po dviejų TSGD metodo iteracijų netikties vektorius X^2 atrodo taip:

$$R^2 = \frac{c_1^2 c_N^2 (\lambda_N - \lambda_1)^2 \lambda_N}{(c_1^2\lambda_1 + c_N^2\lambda_N)(c_1^2\lambda_1^2 + c_N^2\,\lambda_N^2)} \Big(c_1 V_1 + c_N \frac{\lambda_1}{\lambda_N} V_N\Big).$$

Mes matome, kad $R^2 \neq cR^0$, ir konvergavimo greitis iš esmės nėra tiesinis. Po kelių iteracijų netikties vektorius $R^j \approx d_j V_1$, t.y. tiesinė dviejų vektorių kombinacija susiprastina ir duoda vektorių, lygiagretų pirmam tikriniam vektoriui. Gerai žinoma, kad tokiu atveju ir SD, ir MR metodai konverguoja po vienos iteracijos [40].

Nemažindami analizės bendrumo, tarkime, kad pradinis netikties vektorius turi tokį pavidalą:

$$c_1 = 1, \quad c_N = 1, \quad \eta = \frac{\lambda_1}{\lambda_N} \ll 1.$$

Tada randame tokius konvergavimo greičius $\|R^j\| \leq \rho_j \|R^0\|$:

$$\rho_2 = \mathcal{O}(1), \quad \rho_4 = \mathcal{O}(\eta^{-1}), \quad \rho_6 = \mathcal{O}(1), \quad \rho_{6+2k} = \mathcal{O}(\eta^{2k}).$$

Pateikta analizė paaiškina supertiesinį TSGD metodo konvergavimo greitį esant specifiniam pradiniam vektoriui.

Antrame konvergavimo analizės žingsnyje atsižvelkime į specialias asimptotines SD ir MR metodų savybes. Pradedant nuo bendro pradinio vektoriaus

$$R^0 = c_1 V_1 + c_2 V_2 + \ldots + c_N V_N$$

iteracijos monotoniškai konverguoja prie artinio, kuris susideda tik iš dviejų tikrinių vektorių

$$R^j \approx d_1 V_1 + d_N V_N.$$

Tada supertiesinis konvergavimo pagreitėjimas gaunamas per kelias iteracijas, po ko mes gauname aproksimaciją, kur netikties vektoriaus spektriniame atvaizdavime yra svarbūs visi tikriniai vektoriai, ir visas ciklas kartojamas iš naujo.

Galiausiai galima pastebėti, kad šių dviejų savybių, t.y. supertiesinio TSGD metodo konvergavimo, kai pradiniai vektoriai priklauso tam tikram vektorių po-

erdviui, ir asimptotinio visų vektorių konvergavimo į šio poerdvio aplinką, kombinacija paaiškina greitą TSGD metodo konvergavimą.

Pateikta analizė skirta ne tam, kad tiksliai įvertintume konvergavimo greitį, bet tam, kad kokybiškai ištirtume iteracinių sekų elgesį. Ji paaiškina, kodėl galima tikėtis supertiesinio pagreitėjimo. Dar daugiau, ji padeda sukurti naujus efektyvius algoritmus, vienas algoritmas pateiktas žemiau.

B. Modifikuotas TSGD metodas

Pasinaudojant ankščiau gautais rezultatais, galima pasiūlyti TSGD metodo modifikaciją, kai $(m+n)$ iteracijų blokas atliekamas tokiu būdu:

1. Skaičiuojamos m iteracijų SD metodu,
2. Paskui n iteracijų atliekamos TSGD metodu.

Pavadinkime šį algoritmą MSD(m,n) (nuo angl. *Modified Steepest Descent*).

MSD(m,n) algoritmas

MSD (Matr A, Vec x, Vec f, ε, m, n)
begin
(1) Nustatyti pradines reikšmes $i=1; \quad j=0;$
(2) $r = Ax - f;$
(3) **while** ($\|r\| > \varepsilon$) **do**
(4) $j := j+1; \quad v = Ar;$
(5) **if** ($j \le m \,\|\, i == 0$) **then**
(6) $\tau = (r,r)/(v,r);$
(7) $i = 1;$
else
(8) $\tau = (v,r)/(v,v);$
(9) $i = 0;$
end if
(10) $x := x - \tau\, r; \quad r := r - \tau\, v;$
(11) **if** ($j == (n+m)$) $j = 0;$
end do
end MSD

Detalesnė šio algoritmo konvergavimo teorinė analizė ir skaičiavimo rezultatai pateikti [97].

C. Didžiausio nuolydžio metodas su nepilna relaksacija

Nauja įdomi SD algoritmo modifikacija pasiūlyta [73] darbe. Naudojamas toks skaičiavimo algoritmas:

$$X^{n+1} = X^n - 0{,}9\,\tau_n R^n, \quad \tau_n = \frac{(R^n,\, R^n)}{(AR^n,\, R^n)}\,. \tag{3.43}$$

Pagrindinė idėja yra apriboti iteracinio žingsnio ilgį, kurį rekomenduoja klasikinis metodas. Konstanta $0{,}9$ buvo gauta eksperimento būdu. Šis algoritmas pavadintas *didžiausio nuolydžio metodu su nepilna relaksacija* (angl. *Sub-Relaxation Steepest Descent*, arba SRSD).

Šis algoritmas yra įdomus pavyzdys, demonstruojantis, kad godus variacinis Didžiausio nuolydžio metodas *nėra* optimalus. Konvergavimo greitį gali didinti ir *Zeidelio* metodas, jis naudoja priešingą, padidintos relaksacijos technologiją.

SRSD algoritmas

SRSD (Matr A, Vec x, Vec f, ε)
begin
(1) Nustatyti pradines reikšmes $i = 1$; $r = Ax - f$;
(2) **while** ($\|r\| > \varepsilon$) **do**
(3) $v = Ar$;
(4) $\tau = (r, r)/(v, r)$;
(5) $x := x - 0{,}9\tau\, r$;
(6) $r := r - 0{,}9\tau\, v$;
end do
end SRSD

Detalesnė šio algoritmo konvergavimo teorinė analizė ir skaičiavimo rezultatai pateikti [97].

3.4.3. Dvipakopio gradientinio nuolydžio metodo lygiagretinimo analizė

Algoritmų, aprašytų ankščiau, lygiagrečiosios versijos sudarytos naudojant lygiagrečiųjų duomenų dekompozicijos metodą [31]. Kaip pavyzdį galima išnagrinėti dvipakopį gradientinio nuolydžio algoritmą:

procedure `Nuoseklusis TSGD algoritmas`
begin
(1) $X^0, \quad n = 0, \quad R^0 = AX^0 - F$
(2) $RN = (R^n, R^n)$
(3) **while** $(RN > \varepsilon\,(R^0, R^0)\,)$ **do**
(4) $G^n = AR^n$,
(5) **if** (n is even) **then**
(6) $\tau_{n+1} = RN/(G^n, R^n)$,
else
(7) $\tau_{n+1} = (G^n, R^n)/(G^n, G^n)$,
end if
(8) $X^{n+1} = X^n - \tau_{n+1} R^n$,
(9) $R^{n+1} = R^n - \tau_{n+1} G^n$,
(10) $RN = (R^{n+1}, R^{n+1}), \quad n := n + 1$.
end while
end

Kaip modifikuojanti matrica (angl. *preconditioner*) naudojama diagonalinė matricos **A** dalis. Diagonalinės modifikuojančios matricos (MM) gali būti labai efektyviai lygiagretinamos. Gerai žinoma [40], kad, naudojant sudėtingesnias modifikuojančias matricas, tokias kaip ILU arba daugiatinklinius (angl. *multigrid*) algoritmus, galima pasiekti didesnio konvergavimo greičio, tačiau tokių MM lygiagretinimas yra neefektyvus.

Lygiagrečioji ILU versija gaunama, atliekant lokalios kiekvienam procesui matricos dalies ILU faktorizaciją. Gerai žinoma, kad tokia strategija mažina konvergavimo greitį, tačiau tokio MM lygiagretinimo efektyvumas yra toks pat, kaip ir diagonalinio. Tam, kad sumažinti neigiamą lygiagrečiosios ILU faktorizacijos versijos poveikį konvergavimo greičiui, naudojamos įvairios technologijos. Lygiagrečiųjų MM *jungtinių gradientų* metodui kūrimas ir analizė tiriami darbuose [1, 9, 16].

A. Bandomoji užduotis

Sprendžiame trimatį Puasono uždavinį

$$\begin{cases} -\sum\limits_{\alpha=1}^{3} \dfrac{\partial^2 u}{\partial x_\alpha^2} = f, \quad x \in Q := (0,1) \times (0,1) \times (0,1)\,, \\ u(x) = g, \quad x \in \partial Q\,. \end{cases} \tag{3.44}$$

Jį diskretizuojame *baigtinių tūrių* metodu ir gauname tiesinių lygčių sistemą

$$-a_{i+1,j,k}U_{i+1,j,k} - a_{i-1,j,k}U_{i-1,j,k} - a_{i,j+1,k}U_{i,j+1,k} - a_{i,j-1,k}U_{i,j-1,k} \tag{3.45}$$

$$-a_{i,j,k+1}U_{i,j,k+1} - a_{i,j,k-1}U_{i,j,k-1} + a_{ijk}U_{ijk} = F_{ijk}, \quad 1 \leq i,j,k \leq n.$$

Aproksimuojant, naudojamas 7 taškų šablonas (angl. *stencil*). Gautoji tiesinių lygčių sistema gali būti užrašyta kaip $AU = F$, čia A yra simetrinė teigiamai apibrėžta matrica. Daugiau detalių apie tokių diskretinių aproksimacijų savybes pateikia Samarskis [81], Golubas ir Van Lounas [40].

Reikėtų pastebėti, kad variacinių iteracinių algoritmų konvergavimas stipriai priklauso ne tik nuo standumo parametro $\kappa(A)$ (angl. *stiffness parameter*), bet ir nuo visų matricos A tikrinių reikšmių pasiskirstymo (pavyzdžiui, [56]). Šis bandomasis uždavinys yra tipinis 3D elipsinio uždavinio pavyzdys. Tokios problemos sprendžiamos daugelyje realių taikomųjų uždavinių, naudojant variacinius iteracinius algoritmus.

B. Algoritmo formulavimas

Tegu mes turime p procesų, kurie yra sujungti į trimatį tinklą, t.y. $p = p_1 \times p_2 \times p_3$. Diskretusis tinklas (duomenų masyvas) ω_h padalinamas į p trimačių potinklių, panaudojant blokinio paskirstymo schemą. Tada kiekvienam potinkliui ω_{np} priklauso

$$\frac{n}{p_1} \times \frac{n}{p_2} \times \frac{n}{p_3} = \frac{n^3}{p}$$

taškų, kurie priskiriami duotajam procesui. Visi procesai vykdo tą pačią programą, tačiau su skirtingais duomenimis. Kiekvienas procesas yra atsakingas už visus skaičiavimus lokalioje vektoriaus X dalyje.

Reikia papildomai atkreipti dėmesį į vektoriaus G^n taškų, esančių šalia skiriančių plokštumų (t.y. ribinių lokalios vektoriaus U dalies taškų), apskaičiavimą. Norint apskaičiuoti naujas šių taškų reikšmes, reikalingos gretimiems procesams priklausančių taškų reikšmės. Kadangi (3.45) lygtyje panaudotas kryžminis šablonas (3D atveju tai yra žvaigždė), kiekvienoje dimensijoje lokaliuose potinkliuose prisideda po du *šešėlinius* taškus. Kiekvienas procesas keičiasi duomenimis su savo kaimynais nurodytoje 3D topologijoje.

Panašiai, bendravimo tarp visų procesų reikalauja ir dviejų vektorių skaliarinės sandaugos skaičiavimas.

C. Lygiagrečiojo algoritmo sudėtingumas

Šiame skirsnyje pateikiama lygiagretaus TSGD algoritmo išplečiamumo analizė. Pagal vienodo efektyvumo (arba izoefektyvumo, angl. *isoefficiency*) apibrėžimą, reikia surasti uždavinio dydžio W augimo greitį, kintant procesų skaičiui p, kad lygiagretinimo efektyvumas išliktų pastovus.

Pažymėkime $S_p = W/T_p$, $E_p = S_p/p$, čia T_p yra lygiagretaus algoritmo sudėtingumas (angl. *complexity*), kai naudojami p procesai, S_p yra lygiagretaus algoritmo pagreitėjimas ir E_p yra efektyvumas. Taip pat $H(p,W) = pT_p - W$ yra bendrieji lygiagretaus algoritmo papildomi kaštai. Tada izoefektyvumo funkcija $W = g(p, E_p)$ gali būti apibrėžta neišreikštine lygtimi [31]:

$$W = \frac{E_p}{1 - E_p} H(p, W)\,. \tag{3.46}$$

Paprastumo dėlei imkime $E_p = 0{,}5$. Pirmiausia, įvertinkime lygiagretaus TSGD algoritmo sudėtingumą.

1. (8) ir (9) lygiagretaus TSGD algoritmo žingsniuose skaičiuojamos *saxpy* ($Y = \alpha X + Y$, nuo angl. ***s**calar* αX ***p**lus* Y) tipo operacijos. Kiekvienas procesas naudoja tik jam priklausančius taškus, ir duomenų apsikeitimas tarp procesų nereikalingas. Šios dalies sudėtingumas įvertinamas

$$T_{1p} = g_1 \frac{n^3}{p}.$$

2. (4) žingsnyje, matricos ir vektoriaus sandaugos metu, procesai apsikeičia savo šešėliniais vektoriais R^n elementais, kurie atitinka lokalius kraštinius tinklo taškus. Šio žingsnio sudėtingumas

$$T_{2p} = g_2 \frac{n^3}{p} + 6\Big(\alpha + \beta \frac{n^2}{p^{2/3}}\Big)\,,$$

   čia α yra pranešimo iniciacijos laikas, o β yra laikas, reikalingas vieno duomenų elemento siuntimui.

3. Lygiagretusis dviejų vektorių skaliarinės sandaugos skaičiavimas (2), (6), (7) ir (10) žingsniuose reikalauja globalaus bendravimo tarp visų procesų lokalių sandaugos dalių sumavimo metu. Šio žingsnio sudėtingumas aprašomas

$$T_{3p} = g_3 \frac{n^3}{p} + 2R(p)(\alpha_b + \beta_b),$$

   lygtimi, čia $R(p)$ priklauso nuo algoritmo, realizuojančio *ReduceAll* ope-

raciją ir nuo kompiuterio architektūros. Paprasčiausiu atveju mes turime $R(p) = p$.

Susumavus visus gautus įverčius, bendras lygiagretaus TSGD algoritmo efektyvumas yra

$$T_p(J) = g\frac{n^3}{p} + 5R(p)(\alpha_b + \beta_b) + 6\Big(\alpha + \beta\frac{n^2}{p^{2/3}}\Big).$$

Nuoseklaus algoritmo dydis yra $W = gn^3$.

Bendrieji lygiagretaus TSGD algoritmo papildomi kaštai yra

$$H(p, W) = 6\alpha p + 6\beta p^{1/3}n^2 + 5pR(p)(\alpha_b + \beta_b).$$

Panagrinėkime kiekvieno nario individualią įtaką. Narys, kuris priverčia uždavinio dydį augti sparčiausiai, asimptotiškai sąlygoja visą izoefektyvumo funkciją. Po nesudėtingų skaičiavimų galima gauti tokias tris izoefektyvumo funkcijas

$$W = \mathcal{O}(p), \quad W = \mathcal{O}(p), \quad W = \mathcal{O}\big(pR(p)\big)\,.$$

Taigi, bendra asimptotinė izoefektyvumo funkcija sąlygojama globalios redukcijos operacijos papildomų kaštų.

Galima įvertinti paprasčiausią algoritmą, kai visi procesai siunčia savo lokalius rezultatus centriniam procesui, kuris apskaičiuoja globalią skaliarinę sandaugą ir išsiunčia ją visiems procesams (angl. *broadcast*). Tokiu atveju $R(p) = p$ ir $W = \mathcal{O}(p^2)$. 3D dekartinės topologijos tinkle globalios *reduce* and *broadcast* operacijos realizuojamos su $R(p) = p^{1/3}$. Taigi uždavinio dydis W turi augti $\mathcal{O}(p^{4/3})$ greičiu, norint palaikyti pastovų efektyvumą. Hiperkubinei topologijai globalios redukcijos operacijos kaštai yra mažesni ir yra $R(p) = \log p$, tada izoefektyvumo funkcija yra panaši į tiesinę $W = \mathcal{O}(p \log p)$. Taip pat reikėtų pastebėti, kad, kai procesų skaičius yra nedidelis, globalios redukcijos operacijos kaštai gali būti ignoruojami ir tokiu atveju izoefektyvumo funkcija $W = \mathcal{O}(p)$ tiesiškai priklauso nuo p.

3.4.4. Skaitiniai lygiagretinimo eksperimentai

Šiame skirsnyje pateikiami skaičiavimų rezultatai. Bandymai buvo atliekami IBM SP4 superkompiuteryje CINECA, Bolonijoje. Rezultatai paskelbti [97] darbe.

Kaip galima matyti iš teorinės analizės, pateiktos ankstesniame skirsnyje, supertiesinis TSGD metodo konvergavimas pasireiškia dėl stipraus gaunamo iteracinio operatoriaus netiesinio efekto. Iš tiesų čia susiduriama su funkcijos, apra-

šančios iteracinės sekos netikčių paklaidas, bifurkacija. Glodi šių funkcijų seka keičiama staigiais šuoliais tam tikrose iteracijose. Ši savybė yra labai jautri mažoms iteracinių parametrų perturbacijoms. Taigi, galima tikėtis, kad lygiagrečiųjų algoritmų versijų iteracijų skaičius priklausys nuo procesų skaičiaus. Šią išvadą patvirtina skaitiniai eksperimentai, kai skaičiuojama užduotis su skirtuminiu tinkleliu $80 \times 80 \times 80$ (žr. 3.6 lentelę, kur pateikti lygiagrečiųjų variacinių algoritmų iteracijų skaičiai). Klasikiniai iteraciniai algoritmai ir MSD algoritmas yra stabilūs procesų skaičiaus atžvilgiu, tuo tarpu SRSD ir TSGD metodų konvergavimo greitis priklauso nuo p.

3.6 lentelė. *Lygiagrečiųjų variacinių algoritmų iteracijų skaičius*

Metodas	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$
CG	118	118	118	118	118
SD	3920	3920	3920	3920	3920
MSD(30, 10)	198	198	198	198	198
SRSD	252	268	279	234	218
TSGD	318	342	320	412	382

Toliau tiriamas lygiagrečiųjų CG ir MSD(30,10) algoritmų efektyvumas. Šitie algoritmai yra tipiniai dvižingsnių ir vienažingsnių iteracinių algoritmų pavyzdžiai. Rezultatai pateikti 3.7 ir 3.8 lentelėse. Čia pateikiamos eksperimentiškai nustatytos lygiagrečiojo algoritmo pagreitėjimo $S_p(n) = \dfrac{T_1(n)}{T_p(n)}$ ir efektyvumo $E_p(n) = S_p(n)/p$ koeficientų reikšmės, sprendžiant tam tikro dydžio diskrečius uždavinius.

3.7 lentelė. *MSD(30,10) algoritmo pagreitėjimo ir efektyvumo koeficientai*

p	$S_{p,80}$	$E_{p,80}$	$S_{p,120}$	$E_{p,120}$	$S_{p,160}$	$E_{p,160}$
2	1,920	0,960	1,962	0,981	1,971	0,986
4	3,688	0,922	3,776	0,941	3,808	0,952
8	7,104	0,878	7,232	0,904	7,922	0,911
16	13,15	0,822	13,62	0,851	14,58	0,865
32	24,99	0,781	25,70	0,803	26,01	0,813

3.8 lentelė. *Lygiagretaus CG algoritmo pagreitėjimo ir efektyvumo koeficientai*

p	$S_{p,80}$	$E_{p,80}$	$S_{p,120}$	$E_{p,120}$	$S_{p,160}$	$E_{p,160}$
2	1,891	0,945	1,926	0,963	1,962	0,981
4	3,612	0,903	3,688	0,922	3,760	0,940
8	6,768	0,846	6,968	0,871	7,160	0,895
16	12,82	0,801	13,17	0,823	13,49	0,843
32	24,03	0,751	24,96	0,780	25,38	0,793

Iš rezultatų, pateiktų 3.7 ir 3.8 lentelėse, matoma, kad abu lygiagretieji algoritmai yra tiesiškai išplečiami, ir rezultatai patvirtina teorinę išvadą, kad lygiagretaus MSD algoritmo izoefektyvumo funkcija yra beveik tiesinė.

3.5. Trečiojo skyriaus rezultatai ir išvados

Šiame skyriuje nagrinėjami naujos lygiagretinimo technologijos taikymai lygiagretinant sudėtingus skaitinius užduočių sprendimo algoritmus. Pasiekti tokie *rezultatai*:

1. Taikant pasiūlytą ir realizuotą technologiją, sukurta nauja poringųjų terpių modeliavimo įrankio MFSolver versija. Naujoji versija, be lygiagretinimo galimybės, išplečiama ir kitais aspektais:
   - Išplėstas globalaus slėgio matematinis modelis, jame įskaitomi šiluminio pernešimo reiškiniai bei fazių komponentės;
   - Pridėtas slėgio-prisisotinimo modelis, leidžiantis spręsti heterogenius uždavinius;
   - Realizuotas naujas diskretizavimo metodas – antrosios eilės centrinė vienpusių skirtumų aproksimacija.

   Lygiagretinant įrankyje realizuotą diferencialinių lygčių sprendimo algoritmą, pasiektas artimas vienetui lygiagretinimo efektyvumas, kai procesorių skaičius yra iki 8, ir pastebėtas algoritmo išplečiamumas, artimas teoriniam.
2. ParSol biblioteka pritaikyta vaizdų glodinimo naudojant netiesinius difuzinius filtrus algoritmo realizacijai ir lygiagretinimui. Lygiagretinimo efektyvumas buvo tirtas kompiuterių klasteryje „Vilkas“ ir SP4 superkompiu-

teryje CINECA, Bolonijoje. Nustatyta, kad klasteryje „Vilkas" duomenų apsikeitimo kaštai mažai priklauso nuo apsikeičiamų duomenų kiekio, taip pat lygiagretinimo efektyvumas ir išplečiamumas mažesni, palyginti su superkompiuteriu.

3. ParSol biblioteka sėkmingai pritaikyta tokiems variaciniams iteraciniams tiesinių lygčių sistemų sprendimo metodams, kaip CG, SD, TSGD ir kt. lygiagretinti. Nustatyta, kad lygiagrečiosios algoritmų realizacijos yra tiesiškai išplečiamos, kas sutampa su teoriniais įverčiais.

Tai leidžia daryti *išvadas*:

1. Technologija pritaikyta realių sudėtingų diferencialinių lygčių ir tiesinės algebros sprendimo uždavinių realizacijai ir lygiagretinimui, ji gali būti taikoma tokio tipo uždaviniams.

2. Uždavinių sprendimo programos *be pakeitimų* buvo kompiliuojamos ir naudojamos skirtingų architektūrų kompiuteriuose. Programose *nebuvo* naudojamas kodas, būdingas tik tam tikrai specifinei architektūrai. Galime teigti, kad technologijos realizacija leidžia kurti pernešamas programas.

3. Taikant technologiją diferencialinių lygčių sprendimo įrankiui lygiagretinti, buvo pasiektas lygiagretinimo efektyvumas nė mažesnis už 0,982. Galime teigti, kad sukurta technologija efektyviai realizuoja jai keliamus tikslus.

4. Tas pats algoritmas kompiuterių klasteryje „Vilkas" rodo mažesnį lygiagretinimo efektyvumą ir blogesnį išplečiamumą, nei CINECA superkompiuteryje. Taip yra dėl lėtesnių superkompiuterio procesorių ir efektyvesnės duomenų apsikeitimo realizacijos.

5. Ateityje, lygiagretinant kitus diferencialinių lygčių sprendimo įrankius, galima taikyti ne tik sukurtą technologiją, bet ir šiame darbe *išlygiagretintus* variacinius iteracinius tiesinių lygčių sistemų sprendimo metodus, nes tokie metodai sudaro daugelio diferencialinių lygčių sprendimo algoritmų pagrindą.

6. Taikant technologijos realizaciją galima *efektyviai* lygiagretinti diferencialinių lygčių sprendimo algoritmus. Taip teigiame todėl, kad uždavinių lygiagretinimo rezultatai artimi teorinei prognozei, vadinasi, kitų įrankių naudojimas negali *iš esmės* pagerinti efektyvumo.

Darbo rezultatų apibendrinimas ir išvados

Šiame darbe nagrinėjamos diferencialinių lygčių sprendimo įrankių lygiagretinimo technologijos. Darbe pasiekti tokie *rezultatai*:

1. Išanalizuoti populiariausių hiperbolinių ir parabolinių diferencialinių lygčių sprendimo įrankių lygiagretinimo metodai. Nustatyta, kad visi jie turi bendrų savybių – lygiagretinami lygiagrečiųjų duomenų tiesinės algebros ir diskretizacijos uždaviniai duomenų skaidymo metodu. Vartotojas lygiagretinimą atlieka pusiau automatiškai. Lygiagretinimo realizacijai dažniausiai naudojami plačiai paplitę *žemo lygio* standartai. Išanalizuoti egzistuojantys lygiagretinimo sprendimai, siekiant nustatyti, ar jie atitinka keliamus reikalavimus. Nustatyta, kad nė vienas iš jų neatitinka iškeltų *reikalavimų*: veikimas kompiuterių klasteriuose, pusiau automatinis lygiagretinimas, lygiagretinimas tiesinės algebros ir diferencialinių lygčių diskretizacijos lygyje.
2. Remiantis atlikta analize, pasiūlyti naujos diferencialinių lygčių sprendimo įrankių lygiagretinimo technologijos principai. Naujoje technologijoje naudojami lygiagrečiųjų duomenų (lygiagrečiosios duomenų struktūros ir operacijos su jais) ir globalios atminties (globali adresacija ir kai kurių kaimyninių duomenų skaidrus pasiekimas) lygiagretaus programavimo modelių elementai. Taip pat vartotojas išreikštiniu būdu nurodo, kada ir kokiais duomenimis turi būti apsikeičiama.

3. Lygiagretinimo technologija realizuota naudojant standartines C++ kalbos priemones, lygiagretinimui naudojamas MPI standartas. Naudojamos tokios C++ kalbos priemonės, kaip OOP, šabloninis metaprogramavimas, operatorių perkrovimas ir kitos. Realizacija susideda iš tokių skaičiavimo matematikos objektų, kaip masyvai, vektoriai, pilnos ir išretintos matricos. Taip pat realizuotos operacijos su jais – normų skaičiavimas, matricų ir vektorių sandauga, ILU faktorizacija, matricų bei vektorių išraiškos. Egzistuoja tiek nuosekliosios, tiek lygiagrečiosios objektų bei operacijų versijos. Numatytos pagrindinės naujos technologijos ir jos realizacijos vystymo kryptys.

4. Patikrintas technologijos realizacijos efektyvumas. Buvo tiriamas vienos iš realizacijos pagrindinių klasių – *masyvų* realizacijos efektyvumas, palyginti su standartiniais analogais, C/C++ masyvais. Buvo lyginamas masyvo elementų pasiekimo efektyvumas. Taip pat lyginamas globalių skaičiavimo *operacijų* tarp masyvų realizacijos efektyvumas. Nustatyta, kad šių operacijų bibliotekos realizacijos efektyvumas sulyginamas su realizacija standartinėmis C/C++ priemonėmis.

5. Technologija pritaikyta sudėtingiems uždaviniams lygiagretinti. Naudojant biblioteką, sukurta nauja poringųjų terpių modeliavimo įrankio versija, biblioteka pritaikyta vaizdų glodinimo naudojant netiesinius difuzinius filtrus algoritmo lygiagrečiajai versijai kurti, taip pat variaciniams iteraciniams tiesinių lygčių sistemų sprendimo metodams lygiagretinti.

Remiantis šiais rezultatais, gali būti daromos tokios *išvados*:

1. Populiariausių diferencialinių lygčių sprendimo įrankių lygiagretinimo metodai yra efektyvūs, tačiau jų realizacija atlikta žemame lygyje, todėl *negali* būti automatiškai naudojama kitiems įrankiams lygiagretinti. Egzistuojančios lygiagretinimo technologijos ir priemonės, kurios gali būti naudojamos diferencialinių lygčių sprendimo įrankiams lygiagretinti, *neleidžia* gauti norimų savybių.

2. Šiame darbe pasiūlytos naujos lygiagretinimo technologijos principai leidžia *pašalinti* plačiai naudojamų lygiagrečiųjų duomenų ir globalios atminties modelių *trūkumus*, tokius kaip ribotas funkcionalumas arba žemas lygiagretinimo efektyvumas, dėl kurių šie modeliai tiesiogiai neatitinka visų iškeltų reikalavimų: veikimas kompiuterių klasteriuose, pusiau automatinis lygiagretinimas, lygiagretinimas tiesinės algebros ir diferencialinių lygčių diskretizacijos lygyje.

3. Nauja technologija *skirta* diskretizacijos ir tiesinės algebros uždavinių, atsirandančių sprendžiant diferencialines lygtis ir jų sistemas skaitiniais me-

todais, realizacijai ir lygiagretinimui. Ji gali būti taikoma kuriant pusiau automatiškai lygiagretinamus diferencialinių lygčių sprendimo įrankius.

4. Perėjimas nuo standartinių C/C++ priemonių prie naujos technologijos atliekamas remiantis aiškia, formalizuota schema. Prie nuosekliosios versijos gali būti pereita per 6 žingsnius, lygiagretinimas iš vartotojo reikalauja tik 5 žingsnių. Išlaikoma pažįstama programos struktūra. Technologija išbandyta MS-Windows, Linux ir AIX operacinėse sistemose, naudojant MSVC++ ir g++ kompiliatorius bei MPICH, LAM/MPI ir IBM MPI realizacijas.

5. Biblioteka leidžia *efektyviai* realizuoti tiesinės algebros ir diferencialinių lygčių diskretizacijos algoritmus, neatsisakant C++ kalbos privalumų, nes bibliotekos panaudojimas sumažina tam tikrų operacijų efektyvumą daugiausia 2,5 karto, kai kurioms operacijoms efektyvumas sumažėja tik 1–8 %. Biblioteka efektyviai išnaudoja šiuolaikinių kompiliatorių optimizavimo galimybes – eksperimentiškai nustatyta kad efektyvumo padidėja $\approx$2,5 karto (standartinėmis C/C++ priemonėmis parašyta programa po optimizavimo pagreitėja tik $\approx$1,5 karto).

6. Pasiūlytos technologijos taikymų analizė parodė, kad ją taikant galima pasiekti užsibrėžtų tikslų. Technologijos galimybių *užteko* pasirinktų diferencialinių lygčių ir tiesinės algebros uždavinių realizacijai ir lygiagretinimui. Taikant technologiją realizuoti algoritmai *be pakeitimų* buvo kompiliuojami ir vykdomi skirtingose platformose. Lygiagretinimo efektyvumas ir išplečiamumas atitiko *teorines* „geriausio atvejo“ prognozes.

7. Pasiūlytos technologijos realizacija turi plačias taikymo galimybes, realizuojant diferencialinių lygčių sprendimo algoritmus. Taip galima teigti dėl bibliotekos sprendžiamų problemų, tokių kaip efektyvus C++ naudojimas ir algoritmų lygiagretinimas, svarbos.

Literatūros sąrašas

[1] MONGA-MADE, M.; van der VORST, H. A. A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings. *Future Generation Computer Systems*, 17, 2001, p. 925–932.

[2] CHAVENT, G. A new formulation of diphasic incompressible flow in porous media. *Lecture Notes in Mathematics*, 503, 1976.

[3] DOUGHTY, C.; PRUESS, K. A semianalytical solution for heat-pipe effects near high-level nuclear waste packages buried in partially saturated geological media. *Int. J. Heat Mass Transfer*, 31, 1988, p. 79–90.

[4] KURGANOV, A.; PETROVA, G.; POPOV, B. Adaptive Semi-Discrete Central-Upwind Schemes for Nonconvex Hyperbolic Conservation Laws. *SIAM Journal on Scientific Computing*, straipsnis pateiktas spausdinimui.

[5] HU, Y.; BLAKE, R. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, series 25, 1999, p. 417–444.

[6] GRIGAITIS, D.; BARTKUTĖ, V.; SAKALAUSKAS, L. An optimization of system for automatic recognition of ischemic stroke areas in computed tomography images. *Informatica*, 1, 2007. (pateikta spausdinimui)

[7] DONGARRA, J.; LASTOVETSKY, A. An Overview of Heterogeneous High Performance and Grid Computing. *Engineering the Grid: Status and Perspective*, Nova Science Publishers, 2005.

[8] CHEN, Z.; QIN, G.; EWING, R. E. Analysis of compositional model for fluid flow in porous media. *SIAM Journal of Applied Mathematics*, 60(3), 2000, p. 747–777.

[9] ČIEGIS, R. Analysis of parallel preconditioned conjugate gradient algorithms. *Informatica*, 16(3), 2005, p. 317–332.

[10] WEICKERT, J. *Anisotropic Diffusion in Computer Vision.* Stuttgart: Teubner, 1998.

[11] Automatically Tuned Linear Algebra Software (ATLAS). Internetinė prieiga http://math-atlas.sourceforge.net/, [2007.04.30].

[12] BLAS (Basic Linear Algebra Subprograms). Internetinė prieiga http://www.netlib.org/blas/, [2007.04.30].

[13] Blitz++ library. Internetinė prieiga http://www.oonumerics.org/blitz/, [2007.04.30].

[14] Boost C++ libraries. Internetinė prieiga http://www.boost.org/, [2007.04.30].

[15] CLAWPACK — Conservation Law Package. Internetinė prieiga http://www.amath.washington.edu/~claw/, [2007.04.30].

[16] MA, S. Comparisons of the parallel preconditioners for large nonsymmetric sparse linear systems on a parallel computer. *International Journal of High Speed Computing*, 12(1), 2004, p. 55–68.

[17] CHEN, Z.; EWING, R. E. Comparison of various formulations of three-phase flow in porous media. *Journal of Computational Physics*, 132, 1997, p. 362–373.

[18] LANGTANGEN, H. P. *Computational Partial Differential Equations — Numerical Methods and Diffpack Programming (Lecture Notes in Computational Science and Engineering).* Telos: Springer-Verlag, 1999. 682 p.

[19] BEAR, J. *Dynamics of Fluids in Porous Media.* New York: Elsevier, 1972. 764 p.

[20] McWHORTER, D.; SUNADA, D. Exact integral solutions for two-phase flow. *Water Resources Research*, 26(1), 1990, p. 399–413.

[21] SUTTER, H. Exceptional C++. *Addison-Wesley*, 1999.

[22] VELDHUIZEN, T. Expression Templates. *C++ Report*, 7(5), June 1995, p. 26–31.

[23] PFLAUM, C. Expression templates for partial differential equations. *Com-*

puting and Visualization in Science, Springer-Verlag, 4(1), November 2001, p. 1–8.

[24] LeVEQUE, R. *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press, 2002. 578 p.

[25] FreePOOMA. Internetinė prieiga http://www.nongnu.org/freepooma/, [2007.05.20].

[26] Global Arrays. Internetinė prieiga http://www.emsl.pnl.gov/docs/global/, [2007.05.20].

[27] High Performance Fortran. Internetinė prieiga http://hpff.rice.edu/, [2007.04.30].

[28] BROOKS, R. H.; COREY, A. T. Hydraulic properties of porous media. *Hydrol. Pap.*, 3, 1964.

[29] MIKULA, K. Image processing with partial differential equations. Electornic version available at http://citeseer.ist.psu.edu/607193.html, (2007.03.13).

[30] CATTE, F.; LIONS, P.L.; MOREL, J.M.; COLL, T. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM J. Numer. Anal*, 29(1), 1992, p. 182–193.

[31] KUMAR, V.; GRAMA, A.; GUPTA, A.; KARYPIS, G. *Introduction to parallel computing: design and analysis of algorithms.* Redwood City: Benjamin/Cummings, 1994. 597 p.

[32] GRIGAITIS, D. *Kompleksinių metodų taikymas medicinos atvaizdams apdoroti*: daktaro disertacija. Vilniaus Gedimino technikos universitetas. Vilnius: Technika, 2007.

[33] LAM/MPI Parallel Computing. Internetinė prieiga http://www.lam-mpi.org/, [2007.04.19].

[34] XU, C.; LAU, F. Load Balancing in Parallel Computers. Theory and Practice. *Kluwer Academic Publishers*, 1997.

[35] BARAVYKAITĖ, M. M. *Lygiagrečiųjų algoritmų šablonų tyrimas ir kūrimas*: daktaro disertacijos santrauka. Vilnius Gedimino technikos universitetas. Vilnius: Technika, 2006. 19 p.

[36] ŠILKO, G. *Lygiagrečiųjų algoritmų tyrimas ir taikymas lazerinėse technologijose*: daktaro disertacijos santrauka. Vilniaus Gedimino technikos universitetas. Vilnius: Technika, 2003.

[37] ČIEGIS, R. *Lygiagretieji algoritmai ir tinklinės technologijos*. Vilnius: Technika, 2005. 334 p.

[38] HILTUNEN, K. *Mathematical and numerical modeling of consolidation processes in paper machines*: PhD thesis. University of Jyväskylä, Departament of Mathematics. 1995. 112 p.

[39] ČIEGIS, R.; STARIKOVIČIUS, V. Mathematical modeling of wood drying process. *Mathematical Modelling and Analysis*, 7(2), 2002, p. 177–190.

[40] GOLUB, G. H.; Van LOAN, Ch. *Matrix Computations*. Baltimore: The Johns Hopkins University Press, 1996. 728 p.

[41] ŽITKEVIČIUS, E. *Medicininių vaizdų analizė ir tyrimas spektriniais metodais*: daktaro disertacija. Vilniaus Gedimino technikos universitetas. Vilnius: Technika, 2007.

[42] CARVER, R.H.; TAI, K. *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs.* Wiley-Interscience, 2005. 480 p.

[43] AKHTER, S.; ROBERTS, J. *Multi-Core Programming: Increasing Performance through Software Multithreading*. Intel Press, 2006. 336 p.

[44] ABRIOLA, L. M. *Multiphase Flow and Transport Models for Organic Chemicals: A Review and Assessment*. EA-5976, EPRI Palo Alto, CA, 1988. 107 p.

[45] HELMIG, R. *Multiphase Flow and Transport Processes in Subsurface: A Contribution to the Modeling of Hydrosystems*. Berlin: Springler Verlag, 1997. 367 p.

[46] KURGANOV, A.; TADMOR, E. New High Resolution Central Schemes for Nonlinear Conservation Laws and Convection-Diffusion Equations. *Journal of Computational Physics*, 160, 2000, p. 241–282.

[47] JIANG, G. S.; TADMOR, E. Nonoscillatory Central Schemes For Multidimensional Hyperbolic Conservation Laws. *SIAM Journal on Scientific Computing*, 19(6), 1998, p. 1892–1917.

[48] ČIEGIS, R.; ILIEV, O.; STARIKOVIČIUS, V.; STEINER, K. Numerical algorithms for solving problems of multiphase flows in porous media. *Mathematical Modelling and Analysis*, 11(2), 2006, p. 133–148.

[49] BASTIAN, P. *Numerical Computation of Multiphase Flows in Porous Media*: Hab. PhD thesis. Christian–Albrechts–Universität. Kiel, 1999. 222 p.

[50] LeVEQUE, R. *Numerical Methods for Conservation Laws*. Birkhauser, 1992. 214 p.

[51] ČIEGIS, R; ILIEV, O.; LAKDAWALA, Z. On parallel numerical algorithms for simulating industrial filtration problems. *Computational Methods in Applied Mathematics*, 2007, 7(2), p. 118–134. ISSN 1609-4840.

[52] ČIEGIS, Raim.; ČIEGIS, Rem.; MEILŪNAS, M. On the convergence in the L_2 norm of difference schemes for systems of parabolic partial differential equations. *Lietuvos matematikos rinkinys*, 33(3), 1993, p. 269–279.

[53] ČIEGIS, R. On the convergence rate of some iterative methods of variational type. *Mathematical modelling*, 9(4), 1997, p. 115–125.

[54] ČIEGIS, R.; MEILŪNAS, M. On the difference scheme for a nonlinear diffusion – reaction type problem. *Lietuvos matematikos rinkinys*, 33(1), 1993, p. 16–29.

[55] KURGANOV, A.; LIN, C. On the Reduction of Numerical Dissipation in Central-Upwind Schemes. *Communications in Computational Physics*, 2(1), 2007, p. 141–163.

[56] ČIEGIS, R.; PAKENIENĖ V. On the solution of one minimization problem. *Lietuvos matematikos rinkinys*, 42, spec. nr., 2002, p. 604–608.

[57] OpenFOAM: The Open Source CFD Toolbox. Internetinė prieiga http://www.opencfd.co.uk/openfoam/, [2007.04.19].

[58] DOI, S.; WASHIO, T. Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations. *Parallel Computing*, 25, 1999, p. 1995–2014.

[59] GUPTA, A.; KUMAR, V.; SAMEH, A. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(5), 1997, p. 455–469.

[60] PETSc — Portable, Extensible Toolkit for Scientific Computation. Internetinė prieiga http://www-unix.mcs.anl.gov/petsc/petsc-as/index.html, [2007.04.19].

[61] BUTENHOF, D. R. *Programming with POSIX Threads*. Addison-Wesley, 1997. 400 p.

[62] GEIST, G. A.; KOHLA, J. A.; PAPADOPOULOS, P. M. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2), 1996, p. 137–150.

[63] PVM: Parallel Virtual Machine. Internetinė prieiga

http://www.csm.ornl.gov/pvm/, [2007.04.25].

[64] GEIST, A.; DONGARRA, J. and others. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. 299 p.

[65] PERONA, P.; MALIK, J. Scale space and edge detection using anisotropic diffusion. Proceedings of IEEE Computer Society Workshop on Computer Vision, 1987.

[66] WITKIN, A. P. Scale-space filtering. Proceedings of Eight International Conference on Artificial Inteligence, 2, 1983, p. 1019–1022.

[67] UŠINSKAS, A.; NAVAKAUSKAS, D. Segmentation of Ischemic Stroke in CT Images of Human Brain. *Information technology and control*, 3(28), 2003, p. 47–53.

[68] MIKULA, K.; RAMOROSY, N. Semi-implicit finite volume scheme for solving nonlinear diffusion equations in image processing. *Numer. Math.*, 89, 2001, p. 561–590.

[69] WHITAKER, S. Simultaneous heat, mass, and momentum transfer in porous media: A theory of drying. Advances in Heat Transfer, 13, 1977, p. 119–203.

[70] ČIEGIS, R.; BŪDA, V. *Skaičiuojamoji matematika*. Vilnius: TEV, 1997. 240 p.

[71] KAČUR, J.; MIKULA, K. Slow and fast diffusion effects in image processing. Comput. Visual. Sci., 3, 2001, p. 185–195.

[72] LANGTANGEN, H. P.; MUNTHE, O. Solving systems of partial differential equations using object-oriented programming techniques with coupled heat and fluid flow as example. *ACM Transactions on mathematical software*, 27(1), 2001, p. 1–26.

[73] KACHAR, O.; BOLTNEV, A.; KALITKIN, N. Speeding up the convergence of simple gradient methods. Proceedings of the 10th International Conference MMA2005 & CMAM2, Trakai, Lithuania, 2005, p. 349–354.

[74] WILLEBEEK, M. H.; REEVES, A. P. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(9), 1993, p. 979–993.

[75] STROUSTRUP, B. *The C++ Programming Language, Third edition*. Addison-Wesley, 1997. 1040 p.

[76] The Iterative Template Library. Internetinė prieiga

http://www.osl.iu.edu/research/itl/, [2007.04.19].

[77] ASANOVIC, K.; BODIK, R. and others. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No. UCB/EECS-2006-183, December 18, 2006. Internetinė prieiga http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html, [2007.03.19].

[78] The Matrix Template Library. Internetinė prieiga http://osl.iu.edu/research/mtl/, [2007.04.19].

[79] The Message Passing Interface (MPI) standard. Internetinė prieiga http://www-unix.mcs.anl.gov/mpi/, [2007.04.19].

[80] The OpenMP standard. Internetinė prieiga http://www.openmp.org/, [2007.04.19].

[81] SAMARSKII, A. A. *The theory of difference schemes.* Marcel Dekker, Inc., 2001. 786 p.

[82] BOBERG, T. *Thermal Methods of Oil Recovery*. New York: Wiley, 1988. 411 p.

[83] Threading Building Blocks library. Internetinė prieiga http://osstbb.intel.com/, [2007.09.03].

[84] TOUGH2 — a general-purpose numerical simulation program for multiphase fluid and heat flow in porous and fractured media. Internetinė prieiga http://www-esd.lbl.gov/TOUGH2/, [2007.09.03].

[85] ERMAKOV, V. V.; KALITKIN, N. N. Two stage gradient descent method. *J. Comput. Math. and Math. Physics*, 20(4), 1980, p. 1040–1045.

[86] UG — A Flexible Software Toolbox for Solving Partial Differential Equations. Internetinė prieiga http://sit.iwr.uni-heidelberg.de/~ug/, [2007.01.03].

[87] Unified Parallel C. Internetinė prieiga http://upc.gwu.edu/, [2007.04.19].

[88] MIKELAITIS, V.; MARTAVIČIUS, R.; ŽITKEVIČIUS, E. Vilnelių transformacijos taikymas žmogaus galvos smegenų kompiuterinių tomogramų su insulto sritimi analizei. *Elektronika ir elektrotechnika*, 8(64), 2005, p. 73–78.

[89] VIM — the text editor. Internetinė prieiga http://www.vim.org/, [2007.04.19].

[90] DREPPER, U. What every programmer should know about memory. Internetinė prieiga http://lwn.net/Articles/250967/, [2007.09.30].

Autoriaus publikacijų disertacijos tema sąrašas

Periodiniuose mokslo leidiniuose

[91] STARIKOVIČIUS, V.; ČIEGIS, R.; JAKUŠEV, A. Analysis of upwind and high-resolution schemes for solving convection dominated problems in porous media. *Mathematical Modelling and Analysis*, 2006, 11(4), p. 451–474. ISSN 1392-6292.

[92] JAKUŠEV, A. Application of Template Metaprogramming Technologies to Improve the Efficiency of Parallel Arrays. *Mathematical Modelling and Analysis*, 2007, 12(1), p. 71–79. ISSN 1392-6292.

[93] JAKUŠEV, A.; STARIKOVIČIUS, V. Daugiafazio tekėjimo uždavinių sprendimo įrankis ir jo taikymas daugiamačiams uždaviniams. *Lietuvos matematikos rinkinys*, 2004, 44, spec. nr., p. 634–638. ISSN 0132-2818.

[94] ČIEGIS, R.; JAKUŠEV, A. Lygiagretieji algoritmai vaizdų filtravime. *Lietuvos matematikos rinkinys*, 2005, 45, spec. nr., p. 411–416. ISSN 0132-2818.

[95] ČIEGIS, R.; JAKUŠEV, A.; KRYLOVAS, A.; SUBOČ, O. Parallel algorithms for solution of nonlinear diffusion problems in image smoothing. *Mathematical Modelling and Analysis*, 2005, 10(2), p. 155–172. ISSN 1392-6292.

[96] ČIEGIS, R.; JAKUŠEV, A.; STARIKOVIČIUS, V. Parallel tool for solution of multiphase flow problems. In *Lecture Notes in Computer Science*, 6th International Conference, PPAM-2005, Poznań, Poland, September 11–14, 2005. Revised Selected Papers, 2006, Vol. 3911, p. 312–319. ISSN 0302-9743 (ISI Master Journal List).

[97] ČIEGIS, Raim.; ČIEGIS, Rem.; JAKUŠEV, A.; ŠALTENIENĖ, G. Parallel Variational Iterative Linear Solvers. *Mathematical Modelling and Analysis*, 2007, 12(1), p. 1–16. ISSN 1392-6292.

Kituose mokslo leidiniuose

[98] JAKUŠEV, A.; STARIKOVIČIUS, V. Application of Parallel Arrays for Parallelisation of Data Parallel Algorithms. *Computer Aided Methods in Optimal Design and Operations*, Series on Computers and Operations research, 2006, Vol. 7, p. 109–118. ISBN 981-256-909-X (ISI Proceedings).

[99] JAKUŠEV, A.; STARIKOVIČIUS, V.; ČIEGIS, R. Application of parallel arrays for semiautomatic parallelization of flow in porous media problem solver. In *Proceedings of the 10th International Conference MMA2005*

& CMAM2, Trakai, Lithuania, 2005. Vilnius: Technika, 2005, p. 171–177. ISBN 9986-05-924-0.

[100] JAKUŠEV, A.; STARIKOVIČIUS, V. Daugiafazio tekėjimo uždavinių sprendimo įrankis ir jo testavimas. Iš *Matematika (2004 m. balandžio 7–8 d.). Informatika (2004 m. balandžio 13–14 d). 7-osios Lietuvos jaunųjų mokslininkų konferencijos „Lietuva be mokslo – Lietuva be ateities" pranešimų rinkinys*. Vilnius: Technika, 2004, p. 58–65. ISBN 996-05-724-8.

[101] JAKUŠEV, A. Išraiškų šablonų naudojimas C++ masyvų efektyvumo didinimui. Iš *Matematika (2006 m. balandžio 2 d.). Informatika (2006 m. balandžio 12–13 d.). 9-osios Lietuvos jaunųjų mokslininkų konferencijos „Mokslas – Lietuvos ateitis" medžiaga*. Vilnius: Technika, 2006, p. 94–101. ISBN 9986-05-997-6.

[102] ČIEGIS, R.; JAKUŠEV, A.; SUBOČ, O. Nonlinear diffusion problems in image smoothing. In *Proceedings of the 10th International Conference MMA2005 & CMAM2*, Trakai, Lithuania, 2005. Vilnius: Technika, 2005, p. 381–388. ISBN 9986-05-924-0.

Aleksandr JAKUŠEV

DIFERENCIALINIŲ LYGČIŲ IR JŲ SISTEMŲ
SKAITINIO SPRENDIMO ALGORITMŲ
LYGIAGRETINIMO TECHNOLOGIJOS
KŪRIMAS, ANALIZĖ IR TAIKYMAI

Daktaro disertacija
Technologijos mokslai, informatikos inžinerija (07T)

2007 12 14. 9,75 sp. l. Tiražas 20 egz.
Vilnius Gedimino technikos universiteto
leidykla „Technika“, Saulėtekio al. 11, 10223 Vilnius
http://leidykla.vgtu.lt
Spausdino UAB „Baltijos kopija“
Kareivių g. 13B, 09109 Vilnius
http://www.kopija.lt