



VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

FUNDAMENTINIŲ MOKSLŲ FAKULTETAS

MATEMATINIO MODELIAVIMO KATEDRA

Gedas Petrauskas

LYGIAGREČIOJO PROGRAMAVIMO TECHNOLOGIJŲ TYRIMAS
PARALLEL PROGRAMMING TECHNOLOGY RESEARCH

Baigiamasis magistro darbas

Inžinerinės informatikos studijų programa, valstybinis kodas 62409P111

Taikomosios informatikos specializacija

Informatikos studijų kryptis

Vilnius, 2008

Vilniaus Gedimino Technikos universitetas
Fundamentinių mokslų fakultetas
Matematinio modeliavimo katedra

ISBN ISSN
Egz. sk.
Data

Inžinerinės informatikos studijų programos baigiamasis magistro darbas

Pavadinimas: Lygiagrečiojo programavimo technologijų tyrimas

Autorius: Gedas Petrauskas

Vadovas: dr. Vadimas Starikovičius

Kalba



lietuvių



anglų

Anotacija

Baigiamajame magistro darbe nagrinėjamos OpenMP, UPC, MPI ir BSP lygiagrečiojo programavimo technologijos sprendžiant dvimatį šilumos laidumo uždavinį lygiagrečiuoju Jakobio iteraciniu metodu. Kiekvienai technologijai sudaromi lygiagretieji algoritmai, aptariamas jų realizacijos sudėtingumas programuotojo požiūriu ir efektyvumas skirtingose kompiuterių architektūrose.

Dabą sudaro 6 dalys: įvadas, technologijų apžvalga, šilumos laidumo uždavinys, programų realizacija, rezultatų palyginimas, išvados, literatūra.

Darbo apimtis –38 p. teksto be priedų, 9 paveikslėliai, 3 lentelės, 10 bibliografinių šaltinių.

Atskirai pridedami darbo priedai.

Reikšminiai žodžiai: lygiagrečiojo programavimo technologijos, OpenMP, UPC, MPI, BSP, šilumos laidumo uždavinys.

Vilnius Gediminas Technical University
Fundamental Science faculty
Mathematical Modeling department

ISBN ISSN
No. of copies
Date

Final master thesis of the engineering information science studies

Title: Parallel programming technology research

Author: Gedas Petrauskas

Leader: dr. Vadimas Starikovičius

Language
☒ Lithuanian
☐ English

Summary

In this thesis, we consider OpenMP, UPC, MPI and BSP parallel programming technologies - solving two dimensional heat equation, using parallel Jacobi iterative method. Parallel algorithms are constructed and implemented for each technology. Their effectiveness in different computer architectures is discussed as well as the complexity of different implementations from programmer's point of view.

Thesis consists of 6 parts: introduction, technology overview, heat conduction problem, program implementations, comparison of the results, conclusions and references.

Thesis consist of: 38 p. text without appendixes, 9 pictures, 3 tables, 10 bibliographical entries.

Appendixes included.

Keywords: parallel programming technologies, OpenMP, UPC, MPI, BSP, heat conduction problem.

Turinys

IVADAS.....	8
1. TECHNOLOGIJŲ APŽVALGA.....	9
1.1. BENDROSIOS ATMINTIES KOMPIUTERIŲ ARCHITEKTŪRA.....	9
1.1.1. OpenMP.....	9
1.1.2. UPC	11
1.2. PASKIRSTYTOSIOS ATMINTIES KOMPIUTERIŲ ARCHITEKTŪRA.....	14
1.2.1. MPI.....	14
1.2.2. BSP	16
2. ŠILUMOS LAIDUMO UŽDAVINYS (BENCHMARK‘AS).....	20
2.1. MATEMATINIS MODELIS	20
2.2. NUOSEKLUSIS IR LYGIAGRETUSIS ALGORITMAS.....	21
3. PROGRAMŲ REALIZACIJA.....	23
3.1. NUOSEKLI PROGRAMA	23
3.2. BENDROSIOS ATMINTIES KOMPIUTERIŲ ARCHITEKTŪRA.....	24
3.2.1. OpenMP.....	24
3.2.2. UPC	26
3.3. PASKIRSTYTOSIOS ATMINTIES KOMPIUTERIŲ ARCHITEKTŪRA.....	28
3.3.1. MPI.....	28
3.3.2. BSP	31
4. REZULTATŲ PALYGINIMAS	34
4.1. NAUDOJAMA TECHNINĖ ĮRANGA.....	34
4.2. BENDROSIOS ATMINTIES KOMPIUTERIŲ ARCHITEKTŪRA.....	34
4.3. PASKIRSTYTOSIOS ATMINTIES KOMPIUTERIŲ ARCHITEKTŪRA.....	37
4.4. GALUTINIS PALYGINIMAS	40
IŠVADOS	42
LITERATŪRA	44
1 PRIEDAS. NUOSEKLIOSIOS PROGRAMOS KODAS	45
2 PRIEDAS. OPENMP LYGIAGREČIOSIOS PROGRAMOS KODAS (DVI LYGIAGREČIOSIOS SRITYS).....	47

3 PRIEDAS. OPENMP LYGIAGREČIOSIOS PROGRAMOS KODAS (VIENA LYGIAGREČIOJI SRITIS).....	49
4 PRIEDAS. UPC LYGIAGREČIOSIOS PROGRAMOS KODAS.....	51
5 PRIEDAS. MPI LYGIAGREČIOSIOS PROGRAMOS KODAS (BLOKUOTAS DUOMENŲ PERSIUNTIMAS).....	54
6 PRIEDAS. MPI LYGIAGREČIOSIOS PROGRAMOS KODAS (NEBLOKUOTAS DUOMENŲ PERSIUNTIMAS).....	57
7 PRIEDAS. BSP LYGIAGREČIOSIOS PROGRAMOS KODAS (BSMP MODELIS)...	61
8 PRIEDAS. BSP LYGIAGREČIOSIOS PROGRAMOS KODAS (DRMA MODELIS)..	65

Paveikslėlių ir lentelių sąrašas

1 pav.	OpenMP vykdymas	10
2 pav.	UPC gijų ir atminties sąsajos.....	12
3 pav.	Diskretusis tinklas ir schemos šablonas	20
4 pav.	Tinklo taškų padalijimas.....	22
5 pav.	OpenMP laikų palyginimas, išskiriant atmintį dinamiškai ir statišškai	35
6 pav.	OpenMP ir UPC programų vykdymo laikų palyginimas	36
7 pav.	MPI ir BSP programų vykdymo laikų palyginimas	38
8 pav.	MPI ir BSP programų vykdymo laikai, iki 10 mazgų, kiekviename po 4 procesus.....	39
9 pav.	MPI, BSP, OpenMP ir UPC programų vykdymo laikų palyginimas	40
1 lentelė.	OpenMP programų vykdymo laikai, išskiriant atmintį dinamiškai ir statišškai	34
2 lentelė.	OpenMP ir BSP programų vykdymo laikai.....	35
3 lentelė.	MPI ir BSP programų vykdymo laikai	37

Santrumpos

MPI - (angl. Message Passing Interface)

MPMD - (angl. Multiple Program – Multiple Data)

SPMD - (angl. Single Program – Multiple Data)

SMP - (angl. Symmetric Multi-Processor)

BSP - (angl. Bulk Synchronous Parallelism)

DRMA - (ang. Direct Remote Memory Access)

BSMP - (angl. Bulk Synchronous Message Passing)

UPC - (angl. Unified Parallel C)

Ivadas

Šiandien jau nieko nebestebina tai, kad mokslininkai ir inžinieriai daugiau laiko praleidžia prie asmeninių kompiuterių nei atlikdami natūralius eksperimentus laboratorijose. Dabar sprendami matematinio modeliavimo uždavinius, realų objektą pakeičiame jo atvaizdu – matematiniu modeliu, o vėliau ir į virtualų objektą, realizuodami jį matematinio modelio kompiuterio programoje. Tinkamai sudarę matematinį modelį turime realaus objekto savybes, kurios leidžia atlikti eksperimentus su virtualiuoju objektu. Atlikdami eksperimentus su virtualiuoju, o ne su realiuoju objektu, galime gana lengvai ir greitai atlikti jo savybių ir elgesio tyrimą įvairiose įmanomose situacijose. Tokiu būdu galime atlikti eksperimentus su tokiais objektais, su kuriais realiai ne visada galime eksperimentuoti.

Šiuos eksperimentus pastaruoju metu atlikti vis paprasčiau, nes tobulėjant kompiuteriniai technikai, galima atlikti skaičiavimus su sudėtingesniais ir daugiau kompiuterio išteklių reikalaujančiais objektais. Tačiau vienas kompiuteris, kad ir koks galingas jis bebūtų, ne visada pajėgia išspręsti jam pateiktą uždavinį per norimą laiką. Todėl tenka naudotis superkompiuteriais, lygiagrečiaisiais kompiuteriais, bei internetu, kuris sujungia milijonus kompiuterių į globalų tinklą.

Darbo tikslas. Pagrindinis šio darbo tikslas išnagrinėti standartais tapusias ir naujas mažiau žinomas, tačiau kūrėjų pateiktas kaip pranašesnes, lygiagrečiojo programavimo technologijas. Pasirinkus konkretų uždavinį, jį išspręsti su pasirinktomis lygiagrečiojo programavimo technologijomis ir gavus rezultatus palyginti technologijų pranašumus.

Darbo aktualumas. Ištobulėjusi kompiuterinė technika, leidžia vartotojui pasirinkti įvairių architektūrų kompiuterius. Spartesni ir tobulesni kompiuteriai sąlygoja didesnę darbo efektyvumą, tačiau atliekant įvairiausius skaičiavimus, nepakanka turėti tiesiog galingą kompiuterį. Šiuolaikiniai superkompiuteriai savaime nesugeba vieno ar kito uždavinio sprendimo vykdyti lygiagrečiai. Tenka įdiegti ir panaudoti vieną ar kitą kompiuterio architektūrą atitinkančią lygiagrečiojo programavimo technologiją. Šiame darbe atlikę lygiagrečiųjų programavimo technologijų analizę, bandysime įvertinti, kuri iš jų yra pranašesnė efektyvumo klausimu, kuri paprastesnė ir lengviau suprantama naudotojo požiūriu.

Darbo struktūra. Darbas sudarytas iš keturių pagrindinių dalių. Pirmojoje apžvelgiami pasirinktų lygiagrečiojo programavimo technologijų teoriniai aspektai. Antrojoje pateikiamas pasirinktasis uždavinys, kuriuo bus remiamasi atliekant technologijų efektyvumo analizę. Trečiojoje dalyje pateiktas ir aptariamas lygiagrečiųjų programų realizavimas. Ketvirtojoje, paskutiniojoje dalyje, analizuojami gauti rezultatai ir lyginamos lygiagrečiojo programavimo technologijos.

1. Technologijų apžvalga

Šiame darbe lygiagrečiojo programavimo technologijos nagrinėjamos remiantis lygiagrečiųjų kompiuterių atminties modeliais. Išskiriamos bendrosios ir paskirstytosios atminties kompiuterių architektūros. Esminis skirtumas tarp šių technologijų yra tas, kad bendrosios atminties kompiuteriuose programą vykdančios procesai naudoja tą pačią atmintį ir nereikia rūpintis duomenų persiuntimu tarp procesų. Paskirstytosios atminties kompiuteriuose programos dalis, kurioje vykdomi duomenų mainai, yra labai svarbi ir jai turi būti skiriamas didelis dėmesys.

Apžvelgiant lygiagrečiojo programavimo technologijas, pirma aptariama bendrosios atminties kompiuterių architektūrai skirtos technologijos, atkreipiant didesnę dėmesį į pačius lygiagrečiojo programavimo principus, ir analizuojami konkretūs metodai. Vėliau perėjus prie paskirstytosios atminties kompiuterių architektūros, apžvelgiami duomenų mainams tarp procesų naudojami metodai.

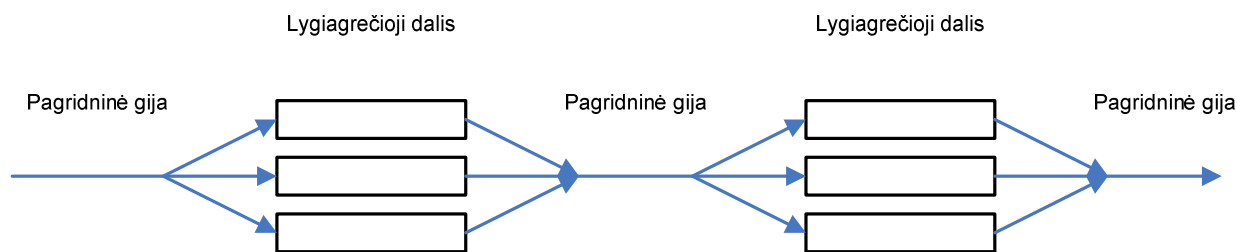
1.1. Bendrosios atminties kompiuterių architektūra

1.1.1. OpenMP

OpenMP yra lygiagrečiojo programavimo standartas, skirtas lygiagrečioms algoritmams realizuoti bendrosios atminties kompiuteriuose. Šia priemone realizuotas programos galima vykdyti ir paskirstytos atminties kompiuteriuose, jei juose yra įdiegta tam skirta programinė įranga. Ši priemonė suteikia galimybę kurti lygiagrečias programas C, C++ ir Fortrano kalbomis. OpenMP sudaro kompiliatoriaus direktyvos, specialių funkcijų biblioteka ir aplinkos kintamųjų rinkinys [5]. Kadangi OpenMP yra skirtas kompiuteriams, naudojantiems bendrąją atmintį, tai programuotojui nereikia rūpintis duomenų paskirstymu tarp lygiagrečiąją programą vykdančių gijų.

Pagrindinis tikslas realizuojant lygiagrečiąsias programas yra išskirti tas algoritmo dalis, kuriose vykdoma daugiausiai skaičiavimų ir programoje jas vykdyti lygiagrečiai. Programoje tą kodo dalį, kurioje vykdomi skaičiavimai, turime vykdyti lygiagrečiai.

Visa OpenMP programa yra skirstoma į nuosekliąsias ir lygiagrečiąsias dalis. Programą pradeda vykdyti viena pagrindinė gija (angl. *thread*), dar vadinama gija šeimininku (angl. *master thread*). Nuosekliosios dalies vykdymas yra nutraukiamas, kai panaudojama direktyva *#pragma omp parallel* lygiagrečiai sričiai (angl. *parallel region*) sukurti. Sukuriama vienu metu veikiančių gijų grupė, tada visi procesai vykdo tą pačią programos dalį. Kai visi procesai baigia vykdyti lygiagrečiąją dalį, jie sustabdomi ir toliau vėl vykdoma nuoseklioji programos dalis, kurią vykdo tik pagrindinė gija(1 pav.).



1 pav. OpenMP vykdymas

Visos lygiagrečiosios gijos turi bendrus (angl. *shared*) ir vietinius (angl. *private*) kintamuosius. Lygiagrečiosios srities kūrimo metu yra nurodoma kurie kintamieji bus bendri, o kurie vietiniai `#pragma omp parallel shared(...) private(...)`. Kiekvienam bendrajam kintamajam sukuriamas tik vienas jo egzempliorius, kurį gali modifikuoti visi grupės procesai. Vietinio kintamojo atskiras egzempliorius sukuriamas kiekvienoje gijoje. Tokiu atveju vietinio kintamojo pasikeitimas vienoje gijoje neturi jokios įtakos kitų gijų vietiniams kintamiesiems. Modifikuojant bendrą kintamąjį gali iškilti problema, nes kitos programą vykdančios gijos net nežina, kad toliau naudoja pakeistą kintamąjį.

Visos OpenMP direktyvos, kuriomis papildomas programos kodas, informatyvios tik šios lygiagrečiosios programavimo technologijos transliatoriams, o standartiniai C ar Fortran kalbų transliatoriai jas vertina kaip komentarus. Todėl OpenMP programa be jokių pakeitimų gali būti vykdoma ir kaip nuosekli. Tačiau kai kurios C programavimo kalbos kontrolės struktūros negali būti panaudotos lygiagrečiosiose srityse, nes jų nepalaiko kai kurie kompiliatoriai, pavyzdžiui, AIX C kompiliatorius *xlc* nepalaiko *break* lygiagrečioje srityje.

Pagrindinė OpenMP idėja yra paimiti ilgiausiai vykdomą skaičiavimo ciklą ir jį vykdyti lygiagrečiai. Prieš ciklą įdedame `#pragma omp for` direktyvą ir transliatorius sukuria lygiagrečiąją sritį, kurioje išorinis ciklas padalijamas visoms programą vykdančioms gijoms. Galima panaudoti ir kitokios formos `#pragma omp parallel for` direktyvą, kuri pati skurtų lygiagrečiąją sritį, sudarytą iš vieno lygiagretaus ciklo.

Kaip jau buvo minėta, vykdant lygiagrečiąją programos sritį ir joje naudojant bendrus kintamuosius, skirtingiems procesams juos modifikuojant gali kilti nesklandumų. Siekiant išvengti klaidų, OpenMP yra realizuota veiksmų atlikimo kontrolė. Šiame darbe apžvelgiamos tik pagrindinės direktyvos, su kuriomis vykdoma kontrolė ir kurios realizuotos lygiagrečiosiose programose.

Direktyva `#pragma omp master` nurodo, kad apibrėžtą kodo dalį vykdo tik pagrindinė gija. Kitos gijos, vykdančios programą, šią dalį praleidžia ir pereina prie kitos užduoties vykdymo.

Direktyva *#pragma omp critical* nurodo, kad šią kodo dalį vienu metu gali vykdyti tik viena gija. Jei atsiranda kita gija, kuri taip pat nori vykdyti tą pačią programos dalį, tai ji yra sustabdoma ir laukia, kol pirmoji gija baigs vykdyti *critical* direktyva apibrėžtą programos dalį.

Dar viena direktyva, kuri gali būti gana dažnai naudojama, tai *#pragma omp barrier*. Ši direktyva sinchronizuoja visų gijų darbą. Kai gija pasiekia šią direktyvą, ji yra sustabdoma ir laukia, kol visos kitos programą vykdančios gijos pasieks barjerą.

Naudojant šias ir keletą kitų direktyvų, galima nesunkiai išvengti klaidų, kurios kyla lygiagrečiai vykdant programą. Šios direktyvos, nors ir padeda išvengti klaidų, tačiau pernelyg jomis apkrovus kodą programos vykdymo efektyvumas gali nukentėti.

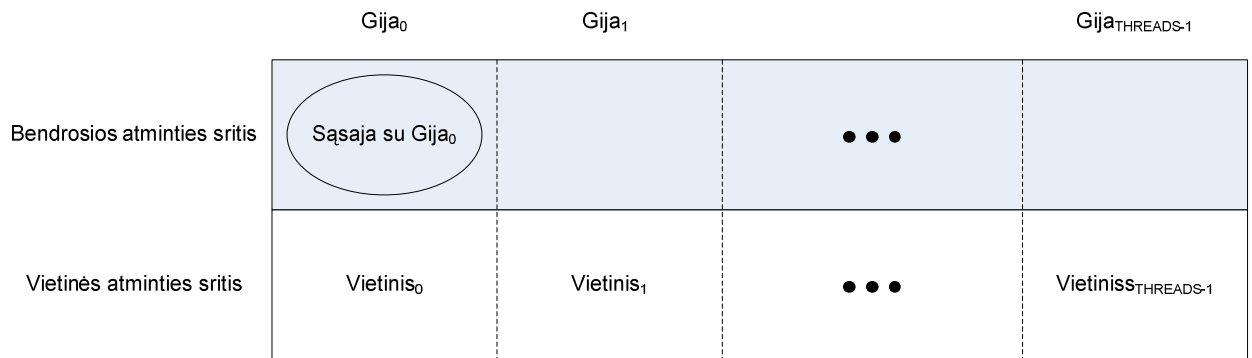
OpenMP privalumai:

- naudojami bendri duomenys;
- lengvai įgyvendinamas lygiagretumas bendrosios atminties kompiuterių architektūroje.

1.1.2. UPC

UPC (angl. Unified Parallel C), yra bendrosios atminties kompiuterių architektūrai skirta lygiagrečiojo programavimo technologija. Šia priemone realizuotas programos, kaip ir OpenMP, galima vykdyti PK telkinyje, jei yra įdiegta tam skirta programinė įranga. UPC išplečia ir papildo C kalbą pridėdama lygiagretumą. Visi procesai vykdo programą bendroje atminties srityje, kuri jiems logiškai padalijama. Kiekvienas procesas yra susietas su tam tikra bendros atminties sritimi. Procesai taip pat turi ir vietinę, tik jiems pasiekiamą atminties sritį, todėl UPC dar vadinama paskirstytos bendrosios (angl. distributed shared) atminties lygiagrečiojo programavimo kalba. UPC yra sukurtas naudojant įgytą patirtį iš paskirstytosios atminties C kompiliatoriaus pirmtakų, tokių kaip Split-C, AC ir PCP. UPC paprastumas, panaudojimas ir efektyvumas pritraukė naudotojus ir kūrėjus siekiančius didelio efektyvumo.

UPC naudojamas paskirstytosios bendrosios atminties adresacijos programinis modelis yra panašus į paskirstytosios atminties modelį su papildoma galimybe panaudoti duomenų saugojimo vietą. Šis modelis paskirsto visą bendrąją atmintį į segmentus kurie yra susieti su konkrečia gija (2 pav.). Čia sąryšis nurodo, kurioje gijos vietinės bendrosios atminties vietoje bus įdėtas bendras objektas.



2 pav. UPC gijų ir atminties sąsajos

Rodyklė rodanti į bendrą duomenų lauką, galės pasiekti visą bendrąją atmintį, o rodyklė rodanti į vietinę gijos atmintį, galės apsiekti tik vietinę atmintį arba jai priskirta bendrosios atminties dalį. Statinis ir dinaminis atminties išskyrimas yra palaikomas tiek bendrojoje atmintyje, tiek vietinėje atmintyje.

UPC turi kelis pagrindinius raktinius žodžius `THREADS` – apibrėžia programą vykdančių gijų skaičių ir `MYTHREAD` – apibrėžia vykdančios programą gijos numerį. Programą vykdančių gijų skaičių naudotojas gali nurodyti kompiliuodamas programą arba programos paleidimo metu.

UPC duomenų paskirstymą palengvina naudojamas programinis modelis. Norint tarp gijų paskirstyti N -matį masyvą, tereikia nurodyti, kad masyvas yra bendras ir nurodyti bloko dydį, kuris aprašo, kaip masyvas bus padalijamas tarp procesų, tada UPC paskirstys masyvo elementus visoms gijoms:

shared int array1[N] – nenurodant bloko dydžio, masyvo elementai padalijami cikliška visoms programą vykdančioms gijoms;

shared [N/THREADS] int array2[N] – tokiu būdu nurodžius bloko dydį, masyvo elementai padalijami vienodais blokais visoms gijoms;

shared [] int array3[N] – nurodant tuščią bloko dydį, masyvo elementai yra saugomi nulinėje gijoje.

Kadangi bendroji atmintis yra pasiekama visoms gijoms, svarbu atkreipti dėmesį į tai, kokia tvarka atmintis yra pasiekama. Siekiant valdyti atmintį pasiekiančias gijas, UPC pateikia keletą sinchronizavimo parametrų. Visų pirmą programuotojas gali apibrėžti griežtą arba laisvą atminties panaudojimą visai programai `#include <upc_strict.h>/#include <upc_relaxed.h>`, kodo daliai `#pragma upc strict/#pragma upc relaxed`, ar atskiriems kintamiesiems `strict shared [N/THREADS] array1[N]/relaxed shared [] array2[N]`. Antra, programuotojas gali naudoti užraktus `upc_lock/upc_unlock`, kurie neleidžia kelioms gijoms vienu metu pasiekti duomenų. Trečia, programuotojas gali naudoti barjerus `upc_barrier`, siekdamas užtikrinti, kad sinchronizavimas tarp visų gijų yra baigtas, prieš tai kai bus atlikti kiti veiksmai.

Aprašant lygiagrečius ciklus, vietoje įprasto *for* ciklo, naudojama *upc_forall*. Aprašant *upc_forall(i=0, i<N; i++; i)* ciklą, atsiranda ketvirtas parametras, kuris suprantamas kaip *i%THREADS*, tai yra nurodoma kuri gija vykdys *i*-tąjį ciklo etapą.

UPC taip pat apibrėžia tokias bendras operacijas (angl. *collective*), kurios leidžia atlikti visoms gijoms bendras operacijas: paskirstyti duomenis visoms gijoms, surinkti juos iš kitų gijų ir perduoti kuriai nors konkrečiai gijai. UPC taip pat leidžia dinaminės atminties išskyrimą. Tačiau ne taip kaip prieš tai aptartoje technologijoje, UPC reikalauja specialios funkcijos iškvietimo, kuri ir pasirūpina dinaminės atminties išskyrimu. Galimos kelios skirtingos dinaminės atminties išskyrimo funkcijos:

- o `shared void *upc_all_alloc (size_t nblocks, size_t nbytes),`
- o `shared void *upc_global_alloc (size_t nblocks, size_t nbytes),`
- o `shared void *upc_alloc(size_t nbytes),`
- o `shared void *upc_local_alloc(size_t nblocks, size_t nbytes),`
- o `void upc_free(shared void *ptr).`

Pirmosios dvi funkcijos skirtos bendrajai atminčiai išskirti, tik skirtumas tarp jų yra tas, kad *upc_global_alloc* nėra bendra funkcija, ją turi kviešti tik viena gija. Funkciją *upc_all_alloc* yra bendroji ir su tokiais pat parametrais kviečiama iš visų gijų, tokiu būdu kiekviena gija atlieka jai tekusios atminties išskyrimo operaciją. Funkcija *upc_alloc* išskiria atmintį bendrosios atminties srityje priklausančią funkciją kviečiančiai gijai. Funkcija *upc_local_alloc* išskiria atmintį vietinėje atminties srityje priklausančią ją kviečiančiai gijai. Paskutinioji funkciją *upc_free* skirta dinaminės atminties išlaisvinimui. Jei atmintis buvo išskirta naudojant *upc_all_alloc*, tada atminties išlaisvinimą gali atlikti bet kuri gija, o jei atmintis išskirta naudojant kurią kitą funkciją, tada ir atminties išlaisvinimą turi atlikti ta pati gija, kuri ją išskyrė.

Plačiau apie bendrąsias operacijas galima rasti UPC specifikacijoje [8].

UPC privalumai:

- paprastos komandos, leidžiančios pasiekti svetimos gijos atmintį;
- lengvai pritaikoma skirtingoms architektūroms;
- sumažinama gijų komunikavimo apkrova naudojant vietinius duomenis;
- palengvina duomenų pasidalijimą tarp skirtingų gijų.

UPC trūkumai:

- nevisiškai veikiantis optimizavimas kompiliuojant;
- nepakankama visų bendrųjų operacijų ir I/O biblioteka.

1.2. Paskirstytosios atminties kompiuterių architektūra

1.2.1. MPI

MPI (angl. Message Passing Interface) yra paskirstytosios atminties kompiuterių, lygiagrečiųjų algoritmų realizavimo standartas. MPI tai pranešimų perdavimo funkcijų rinkinys, kuris leidžia keisti duomenimis kompiuteriams, sujungtiems į PK telkinį, superkompiuterį ar įvairialypį tinklą. Šią funkcijų biblioteką galima naudoti programuojant C ir Fortrano 77 kalbą. Yra parengtas ir išleistas naujas MPI 2 standartas, kuris papildo MPI 1 standartą naujomis priemonėmis ir įgalina programuoti C++ ir Fortran 90 programavimo kalbomis.

MPI kaip standartą palaiko visi skirtingų architektūrų lygiagretieji kompiuteriai. Pakanka paprasčiausiai iš vieno tipo kompiuterio kodą perkelti į kitą ir programa puikiai veikia. Tai svarbu kuriant matematinių algoritmų bibliotekas.

Programuodamas paskirstytosios atminties kompiuteriuose, naudodamas MPI standartą programuotojas pats privalo pasirūpinti duomenų perdavimu tarp programą vykdančių procesų. Ne taip kaip bendrosios atminties kompiuteriuose čia svarbiausia atlikti tinkamą duomenų persiuntimą. Kadangi vienas procesas vienu metu gali tik siųsti arba priimti duomenis, tai tenka pasirūpinti, kad tarp daugybės procesų duomenų perdavimas būtų įvykdytas kuo greičiau. Siekiant išvengti nereikalingos apkrovos, yra patariama siųsti vieną ilgesnį pranešimą nei kelis trumpus ir grupuoti skirtingus pranešimus į vieną. Duomenų perdavimas daug priklauso nuo kompiuterių tinklų architektūros, į kurią yra sujungti kompiuteriai, taigi tenka rūpintis duomenų persiuntimo sinchronizavimui. Tarkim, žiediniame tinkle visi kompiuteriai yra sujungti į žiedą ir norint nusiųsti pranešimą kitoje žiedo pusėje esančiam kompiuteriui, pranešimą tenka persiųsti per visus iki jo esančius kompiuterius. Žvaigždiniame tinkle visus kompiuterius jungia vienas kompiuteris, per kurį ir yra siunčiami pranešimai, taigi jei pranešimą jau kažkas siunčia, kitas pranešimas negali būti siunčiamas. Labiausiai tinkamas būtų visiškai jungus tinklas, tačiau tokį tinklą fiziškai sunku realizuoti.

Rašant lygiagrečiąją programą, didžioji laiko dalis gali būti sugaišta ieškant tinkamo duomenų paskirstymo tarp procesų. Tai gali būti vienmatis blokinis ar ciklinis duomenų paskirstymas ir dvimatis taip pat blokinis ar ciklinis duomenų paskirstymas. Priklausomai nuo problemos tipo ir jos dydžio, programuotojas savo nuožiūra turi pasirinkti duomenų paskirstymo būdą, bet iš pirmo žvilgsnio pasirodęs tinkamas duomenų paskirstymo būdas ne visuomet yra tinkamiausias, taigi tenka išbandyti ne vieną variantą, tai yra gaišti laiką.

Vykdydami lygiagretų algoritmą galime naudoti MPMD (angl. multiple program – multiple data) tipo algoritmus, tokio tipo algoritmuose skirtingi procesai vykdo skirtingas programas, tačiau tokia galimybė yra retai naudojama. Kadangi su lygiagrečiaisiais algoritmais stengiamės išspręsti tą

patį uždavinį padalindami visus duomenis programą vykdančioms procesams. Toks lygiagrečiųjų programų sudarymo būdas vadinamas SPMD (angl. single program – multiple data).

Tokiam lygiagrečiųjų programų realizavimui naudojamas MPI standartas, sudarytas iš daugiau kaip 125 funkcijų, tačiau iš esmės naudojamos tik 6 pagrindinės, kurių visiškai pakanka parašyti visavertę lygiagrečią programą:

- Komunikatorių paruošimas:
 - o `MPI_Init(int *argc, char *argv)` – apibrėžia MPI darbo pradžią;
 - o `MPI_Comm_size(MPI_Comm comm, int *size)` – grąžina vykdančių procesų skaičių;
 - o `MPI_Comm_rank(MPI_Comm comm, int *rank)` – grąžina vykdančio proceso numerį.
- Duomenų persiuntimas tarp procesų:
 - o `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)` – pranešimo siuntimas;
 - o `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)` – pranešimo priėmimas.
- MPI bibliotekos darbo pabaiga
 - o `MPI_Finalize(void)` – visi procesai įvykdo šią funkciją ir sustabdo MPI bibliotekų darbą..

Funkcija `MPI_Init` yra kviečiama programos vykdymo pradžioje. Ji apibrėžia MPI aplinką ir todėl gali būti kviečiama tik vieną kartą. Su `MPI_Comm_size` funkcija lygiagretųjį algoritmą skaidome į procesus, kurie gali būti vykdomi vienu metu. Funkcijoje nurodomas komunikatorius `MPI_COMM_WORLD`, kurį visuomet apibrėžia funkcija `MPI_Init`. Su `MPI_Comm_rank` funkcija sužinome programą vykdančio proceso numerį.

Duomenų siuntimo funkcija `MPI_Send` siunčia duomenų paketą. Nurodome, ką norime siųsti, kokia duomenų apimtis ir duomenų tipas. Turime nurodyti, kam siunčiame pranešimą, tai yra proceso numerį, kuriam yra skirtas siunčiamas pranešimas. Prieš paskutinis parametras yra kaip etiketė, identifikuojanči pranešimą, iš jos gali būti atskirti to paties proceso siųsti pranešimai. Duomenų priėmimo funkcija `MPI_Recv` panaši į pranešimo siuntimo funkciją, skiriasi tik tuo, kad vietoje proceso, kuriam siuntėme pranešimą, turime nurodyti, iš ko laukiame pranešimo. Atsiranda dar vienas parametras, tai `MPI_Status`, kuris nurodo pranešimo siuntimo statusą.

Galiausiai `MPI_Finalize` funkcija, kurią turi įvykdyti visi lygiagrečią programą vykdančios procesai. Po šios funkcijos įvykdymo yra sustabdomas MPI bibliotekos veikimas ir toliau nebegalima naudoti MPI funkcijų.

Naudojant prieš tai aprašytas duomenų persiuntimo funkcijas `MPI_Send` ir `MPI_Recv`, vienas procesorius duomenis turi siųsti, o kitas juos priimti. Kol duomenų persiuntimas nebaigtas jokie kiti veiksmai negali būti vykdomi. Tai vadinama blokuotais pranešimų siuntimais. Sprendžiant tokius uždavinius kuriuose galima su vienais duomenimis atlikti veiksmus, o kitus siųsti kitiems procesams tuo pačiu metu, yra naudojami neblokuoti pranešimų siuntimai `MPI_Isend` ir `MPI_Irecv`. Atlikdami tokius pranešimų siuntimus, galima deklaruoti, kad norime siųsti pranešimą ir nebaigę duomenų persiuntimo tęsti darbą toliau. Tokiu būdu efektyviau išnaudojamas uždavinio sprendimui skirtas laikas.

Aptarėme paprastus pranešimų siuntimus tarp procesorių poros. Kita grupė pranešimų, kuriais galime pranešimą siųsti vienu metu visiems procesams, arba iš kitų procesų su viena funkcija surinkti pranešimus, yra vadinami kolektyviniais pranešimais. Plačiau apie kolektyvines komunikacijas galima rasti MPI standarte [3].

MPI privalumai:

- vieningas standartas paskirstytosios atminties kompiuterių architektūroje;
- portatyvumas. Programos, naudojančios MPI bibliotekas, veikia įvairaus tipo kompiuteriuose be jokio kodo modifikavimo;
- aiški lygiagretumo kontrolė;
- nėra bendrų duomenų, nereikia rūpintis, ar vienas procesas, modifikuodamas duomenis, nulems kitą procesą.

MPI trūkumai:

- reikalauja daug laiko tinkamam kodui parašyti;
- didėja kodo apimtis;
- didelis persiunčiamų duomenų skaičius labai lemia programos vykdymo laiką.

1.2.2. BSP

BSP (Bulk Synchronous Parallelism) yra programinis modelis, skirtas įvairaus tipo paskirstytosios atminties kompiuterių architektūrai. Tai reiškia, kad beveik kiekvienas lygiagretus kompiuteris gali simuliuoti BSP abstraktų mechanizmą. BSP programinis modelis yra skirtas programuoti C ir Fortran kalbomis.

BSP veikimas yra suskaidytas į vadinamuosius superetapus (angl. *superstep*), kurie yra laikomi pagrindinėmis lygiagrečiomis instrukcijomis BSP abstrakčiame mechanizme [6]. Superetas yra visuotinė lygiagrečiojo kompiuterio operacija, ir yra išskaidyta į tris dalis:

- o Vietiniai skaičiavimai, kuriuos atlieka kiekvienas procesas su savo turimais duomenimis.
- o Neblokuotos komunikacijos tarp procesų.
- o Barjerinis sinchronizavimas, apimantis visus procesus. Po tokio sinchronizavimo yra garantuojama, kad visos komunikacijos yra baigtos, perkelti duomenys yra matomi visiems procesams, kuriems jie buvo siųsti.

BSP yra SPMD tipo programinis modelis. Standartinė implementacija yra BSPLib, tačiau ją sudaro daugybė funkcijų. Šiame darbe naudojama BSPonMPI implementacija, kurią sudaro pagrindinės BSP funkcijos. BSPonMPI biblioteka ypatinga tuo, kad ją gali vykdyti bet kuris lygiagretusis kompiuteris, kuriame veikia MPI. Tokiu būdu pranešimų siuntimui naudojamos MPI funkcijos.

Biblioteką sudaro du skirtingi komunikatorių tipai:

- o Direct Remote Memory Access (DRMA) tai toks tipas, kai vienas procesas gali nukopijuoti duomenis iš kito proceso atminties arba nukopijuoti duomenis į kito proceso atmintį, be aktyvios kito proceso veiklos;
- o Bulk Synchronous Message Passing (BSMP) tai toks tipas, kai vienas procesas išsiunčia pranešimą, kuris patalpinamas į eilę kitame procese. Pranešimo siuntimas gali būti baigtas tik kito superetapo metu.

BSPonMPI biblioteką sudaro šios pagrindinės operacijos:

- Inicializacijos:
 - o `bsp_init(void (*startproc)(void), int argc, char **argv)` – funkcija, kuri turėtų būti įvykdoma pati pirmoje BSP programoje. *Startproc* parametras nurodo procedūrą, kurią turės vykdyti visi procesai;
 - o `bsp_begin(int maxprocs)` – parengia procesus, kurie vykdys lygiagrečią programą. *Maxproc* kintamasis nurodo maksimalų procesų skaičių;
 - o `bsp_end()` – iškvietus šia procedūrą sustabdomas BSP komunikacijų veikimas;
- Teiravimosi:
 - o `bsp_nprocs()` – grąžina programą vykdančiųjų procesų skaičių;
 - o `bsp_pid()` – grąžina besikreipiančio proceso numerį, kuris yra $[0, \text{bsp_nprocs}()-1]$ ribose;
 - o `bsp_time()` – funkcija grąžina laiko reikšmę sekundėmis, nuo to laiko kai buvo iškviesta *bsp_begin* funkcija;
- Sinchronizacijos:
 - o `bsp_sync()` – nurodo, kad funkciją iškvietęs procesas yra pasiruošęs ir laukia bendrojo sinchronizavimo;

- DRMA:
 - o `bsp_pushregister(const void *addr, int size)` – funkcija yra skirta pranešti, kad tam tikra atminties dalis yra paruošta komunikacijoms;
 - o `bsp_popregister(const void *addr)` – funkcija skirta pranešti, kad atmintis nebegali būti naudojama komunikacijoms vykdyti;
 - o `bsp_get(int pid, const void *src, int offset, void *dst, int nbytes)` – funkcija skirta paimti duomenis iš nutolusio proceso atminties. Duomenys neįrašomi iki superetapo pabaigos;
 - o `bsp_put(int pid, const void *src, void *dst, int offset, int nbytes)` – funkcija skirta patalpinti duomenis į nutolusio proceso atmintį. Duomenys įrašomi į nutolusio proceso atmintį iki superetapo pabaigos;
- BSMP:
 - o `bsp_set_tag_size(int *tag_bytes)` – funkcija nustato pranešimo etiketės dydį;
 - o `bsp_send(int pid, const void *tag, const void *payload, int payload_bytes)` – funkcija naudojama duomenų persiuntimui. Duomenys yra nusiunčiami į nutolusio proceso atmintį iki kito superetapo pradžios.
 - o `bsp_get_tag(int *status, void *tag)` – funkcija naudojama patikrinti siunčiamiems pranešimams. Parametrui *status* suteikiama siunčiamo pranešimo ilgio reikšmė, o *tag* pranešimo etiketės reikšmė;
 - o `bsp_move(void *payload, int reception_nbytes)` – funkcija priima siunčiamą pranešimą;
- Didelio efektyvumo:
 - o `bsp_hpput(int pid, const void *src, void *dst, int offset, int nbytes)` – didelio efektyvumo *bsp_put* funkcija. Duomenys gali būti nusiųsti bet kuriuo metu, tarp pranešimo išsiuntimo ir superetapo pabaigos;
 - o `bsp_hpget(int pid, const void *src, int offset, void *dst, int nbytes)` – didelio efektyvumo *bsp_get* funkcija. Duomenys yra paimami bet kuriuo metu, nuo funkcijos iškvietimo iki superetapo pabaigos;
 - o `bsp_hpmove(void **tag_ptr_buf, void **payload_ptr_buf)` – didelio efektyvumo siunčiamų duomenų priėmimo funkcija;
- Pertraukimo:
 - o `bsp_abort(char* format, ...)` – funkcija skirta nutraukti BSP komunikacijas ir atspausdinti klaidos pranešimą *printf* stiliumi.

Dažniausiai naudojamas būdas vykdyti komunikacijas tarp procesų yra DRMA operacijos. Tai leidžia duomenų kopijas patalpinti arba paimti iš kito proceso, neįtakojant jo darbo. Plačiau apie BSP standartą ir įgyvendinimus galima rasti BSP Worldwide tinklapyje [7].

BSP privalumai:

- duomenų persiuntimo kontrolė;
- programa suskaidoma į superetapus.

BSP trūkumai:

- vykdoma bendroji sinchronizacija.

2. Šilumos laidumo uždavinys (benchmark'as)

2.1. Matematinis modelis

Šilumos laidumo uždavinys pasirinktas, nes šis uždavinys yra plačiai žinomas ir norint jį išspręsti skaitiškai reikia apdoroti didelį duomenų kiekį, todėl jis yra tinkamas kompiuterių skaičiavimo efektyvumui tirti. Sprendžiamame uždavinyje modeliuojame kvadratinę plokštelę, kurios visi kraštai yra šildomi, o temperatūra vidiniuose taškuose yra apskaičiuojama [1].

Norint skaitiškai išspręsti stacionarų dvimatį šilumos laidumo uždavinį [2]:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), (x, y) \in (0,1) \times (0,1) \quad (1)$$

$$u(x, y) = \mu(x, y), (x, y) \in \gamma \quad (2)$$

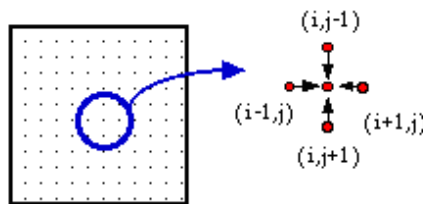
čia γ žymi srities kontūrą (x, y) , $u(x, y)$ yra temperatūra taške (x, y) , o $f(x, y)$ apibrėžia šilumos šaltinį. Naudojame baigtinių skirtumų aproksimaciją, todėl apibrėžiame diskretųjį tinklą, kuris tinka sprendžiamam uždaviniui:

$$w_h = \{(x_i, y_j) : x_i = ih, y_j = jh, 0 \leq i, j \leq N\}, \quad (3)$$

čia $h = \frac{1}{N}$ yra tinklo žingsnis.

Turime diskretųjį sprendinį $U_{ij} = U(x_i, y_j)$, kurį skaičiuosime tinklo taškuose, todėl bendras nežinomųjų skaičius $(N+1)^2$.

Kiekvieno vidinio taško reikšmei gauti, naudosime keturių gretimų taškų šabloną (3 pav.) ir aproksimuojame išvestines baigtiniais skirtumais.



3 pav. Diskretusis tinklas ir schemos šablonas

Gauname tiesinių lygčių sistemą:

$$U_{i,j} = \frac{1}{4} (U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - h^2 f_{i,j}), 1 \leq i, j \leq N-1, \quad (4)$$

Uždavinys sprendžiamas tik vidiniuose tinklo taškuose, todėl užsiduodamos kraštinės sąlygos srities kontūre:

$$U_{ij} = \mu(x_i, y_j), (x_i, y_j) \in \gamma, \quad (5)$$

čia (x_i, y_j) temperatūros reikšmė srities kontūre.

Sprendžiant šilumos laidumo uždavinį, Jakobio iteraciniu metodu, reikia nurodyti pradinės tinklo taškų temperatūros reikšmes (šiam uždavinyje pradinės temperatūros reikšmės lygios 0):

$$U_{i,j} = 0, \quad 1 \leq i, j \leq N-1. \quad (6)$$

2.2. Nuoseklusis ir lygiagretusis algoritmas

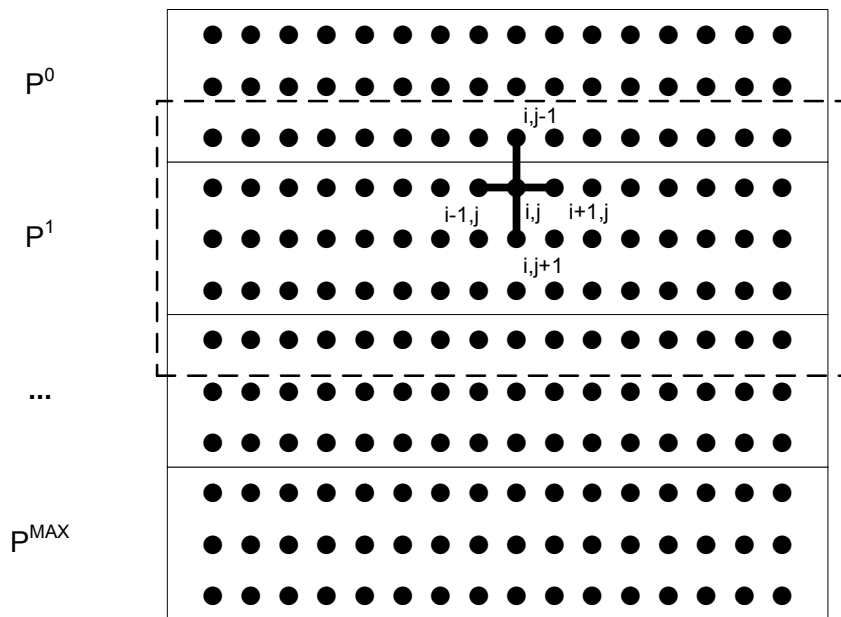
Tiesinių lygčių sistemą sudaro $(N-1)^2$ lygčių. Šiai sistemai spręsti naudojamas Jakobio iteracinis metodas:

```
while(|US-US-1| < ε)
  for(i = 1; i < N; i++)
    for(j = 1; j < N; j++)
      US(i,j)=(US-1(i-1,j)+ US-1(i+1,j)+ US-1(i,j-1)+ US-1(i,j+1)-h*h*F(i,j))/4.;
```

Šis šilumos laidumo uždavinys sprendžiamas iki tol, kol dvejų gretimų artinių skirtumo norma netenkins sąlygos $|U^S - U^{S-1}| < \varepsilon$. Tokiu būdu atliekamas tam tikras iteracijų skaičius, iki kol tenkinama anksčiau pateikta sąlyga. Priskyrus ε pakankamai mažą reikšmę iteracijos atliekamos iki tol, kol gaunama stabili temperatūra.

Siekiant realizuoti lygiagretųjį Jakobio iteracinį metodo algoritmą, pirmiausia reikia atlikti duomenų paskirstymą tarp lygiagrečiųjų procesų. Kadangi iteraciniai metodai priklauso duomenų lygiagretumo uždavinių tipui, tai reiškia, kad su visais duomenimis atliekami vienodi skaičiavimai.

Norint kuo efektyviau realizuoti lygiagretųjį algoritmą, reikia pasirūpinti, kad tinklo taškai tarp procesų būtų paskirstyti kuo tolygiau. Paprasčiausias ir gana efektyvus duomenų paskirstymo būdas yra paskirstyti tinklo taškus blokais visiems procesams (4 pav.):



4 pav. Tinklo taškų padalijimas

Tokiu būdu, visi tinklo taškai, padalijami procesams po lygiai. Reikia pastebėti, kad skaičiuojant konkretaus taško reikšmę, reikia žinoti ir keturių gretimų taškų reikšmes. Atliekant skaičiavimus pagal pateiktą schemą, susiduriame su problema, kad kai kurie skaičiavimams reikalingi duomenys priklauso gretimiesiems procesams, todėl patogų vietinę atminties sritį praplėsti fiktyviais kraštiniais taškais. Duomenų paskirstymas procesams, bei duomenų mainai tarp programą vykdančių procesų, skiriasi esant skirtingoms lygiagrečiojo programavimo technologijoms. Kitame skyriuje aptariamos skirtingos technologijos su kuriomis realizuotas Jakobio iteracinis metodas.

3. Programų realizacija

Programų realizacija, kaip ir lygiagrečiųjų programavimo technologijų apžvalga, išskaidyta į dvi grupes atsižvelgiant į kompiuterių architektūrą. Programuojant bendrosios atminties kompiuterių architektūrai skirtomis technologijomis, nereikia rūpintis duomenų persiuntimu tarp programą vykdančių procesų. Didesnė dėmesio dalis skiriama direktyvų, skirtų duomenų padalijimui tarp procesų, aptarimui. Aptariant paskirstytosios atminties kompiuterių architektūrai skirtų lygiagrečiojo programavimo technologijų programų realizaciją, didžiausias dėmesys skiriamas duomenų persiuntimo tarp procesų realizacijos ypatumams.

3.1. Nuosekli programa

Lygiagrečioji programa pradedama rašyti nuo nuoseklios programos realizavimo. Turint gerai veikiančią nuoseklią programą, paprasčiausiu atveju, ją reikia papildyti keliais kintamaisiais ir keliomis direktyvomis, kad gautumėme lygiagretų variantą, bendrosios atminties kompiuteriuose. Sudėtingiau yra realizuoti lygiagrečiąją programą, kai reikia ne tik įterpti papildomas direktyvas, bet dar ir modifikuoti esamą kodą, kaip tai bus padaryta paskirstytosios atminties kompiuterių architektūrai skirtose programose.

Pirmiausia išanalizuojamas nuosekliosios programos kodas, o vėliau aptariant lygiagrečias programas nebereikės gilintis į bendrą programos dalį, tinkančią visoms lygiagrečiojo programavimo technologijoms. Nuoseklios programos kodas pridedamas priede (1 priedas).

Programos pradžioje aprašoma, kaip programoje bus suprantami ir kaip bus kreipiamasi į duomenų masyvus, kuriuose saugomi visi duomenys. Panaudojant C kalbos `#define U(i,j) f[(j)*(N+1)+i]` direktyvą, aprašoma, kaip turės būti suprantami vėliau aprašomi duomenų masyvai. Apibrėžiami trys masyvai `*u`, `*us`, `*f`. Masyve `u` saugomas adresas į einamosios iteracijos temperatūrų masyvą. Masyve `us` saugomas adresas į prieš tai buvusios iteracijos masyvą, o masyve `f` apibrėžiamas šilumos šaltinis. Nors masyvai deklaruojami kaip vienmačiai, tačiau anksčiau jie buvo apibrėžti kaip dvimačiai, tad vėliau į juos galima kreiptis `U(i,j)`, kur `i` stulpelio indeksas, o `j` – eilutės (indeksai pagal matematinį modelį).

Programos *main* dalies pradžioje aprašomi reikiami kintamieji: *it* – iteracijų skaitiklis, *itmax* – maksimalus iteracijų skaičius, *paklaida* – maksimali paklaidos norma tarp gretimų iteracijų, *skirt* – paklaidos norma tarp visų gretimų iteracijos taškų, *h* – tinklo žingsnis lygus $1/N$, *e* – pakankama paklaida, *timeval tim* – laiko struktūra, naudojama nustatyti programos vykdymo laiką, *t1* ir *t2* – laiko reikšmės, prieš pradedant vykdyti skaičiavimus ir iškart po skaičiavimų pabaigos. Šie kintamieji yra pagrindiniai ir jie taip pat bus visose lygiagrečiose programose. Aprašius kintamuosius, dinaminiu būdu išskiriama atmintis masyvams: `u = (double *)`

`malloc(sizeof(double)*(N+1)*(N+1))`). Po to naudojama papildoma procedūra *Init()*, kuri skirta pradinių reikšmių priskyrimui ir kraštinių sąlygų apibrėžimui. Ši procedūra neturi įtakos skaičiavimams, todėl programos fragmente nepateikiama. Galiausiai prieš pradedant iteracijas yra laiko fiksavimas prieš iteracijų vykdymo pradžią.

Programoje iteracijos vykdomos *while*((*paklaida* > *e*) && (*it* < *itmax*)) ciklo pagalba. Ciklas vykdomas iki tol, kol tenkinamos abi sąlygos: paklaida nėra mažesnė už pageidaujamą; iteracijų skaitiklis nepasiekia maksimalaus iteracijų skaičiaus. Kiekvienos iteracijos metu vykdomi du dvigubi *for* ciklai. Pirmame dvigubame *for* cikle atliekami pagrindiniai skaičiavimai pagal keturių gretimų taškų šabloną. Antrame vertinama paklaida ir perkopijuojami duomenys iš masyvo *U* į masyvą *US*. *While* ciklo pabaigoje padidiname iteracijos skaitiklį. Jei paklaida ir iteracijos skaitiklis tenkina *while* ciklo sąlygas, tada pradedama vykdyti naują iteraciją. Jei viena iš sąlygų netenkinama, tada ciklą vykdymas yra baigiamas.

Po iteracijų vykdymo pabaigos, vėl fiksuojamas laikas. Apskaičiavę skirtumą tarp turimų laikų, skaičiavimų vykdymo pradžios ir pabaigos, randamas skaičiavimų vykdymo laikas.

Jakobio iteracinio metodo algoritmas yra nesudėtingas, tačiau jis puikiai tinka, norint įvertinti, kaip skirtingos technologijos susidoroja su duomenų lygiagrečiojo uždaviniu.

3.2. Bendrosios atminties kompiuterių architektūra

3.2.1. OpenMP

OpenMP lygiagrečiojo programavimo technologija yra skirta bendrosios atminties kompiuterių architektūrai. Pagrindinis lygiagrečiųjų uždavinių sprendimo principas yra tas, kad duomenys padalijami visiems programą vykdančioms procesams. Sprendžiant šilumos laidumo uždavinį, su visais duomenimis atliekami tie patys veiksmai, todėl tą patį nuoseklų uždavinio sprendimo algoritmą, reikia vykdyti visuose procesuose. Paprasčiausias būdas gauti lygiagrečią programą, tai paimti ciklą, kuriame atliekami skaičiavimai ir jį vykdyti lygiagrečiai.

OpenMP biblioteką sudaro nesudėtingos funkcijos ir direktyvos, kuriomis papildę nuoseklų kodą, gauname lygiagrečią programą. Įkėlus keletą direktyvų ir kode niekuo nepakeitus, gaunama lygiagrečioji programa.

Šiame darbe pateikiamos dvi skirtingos OpenMP programos realizacijos. Pirmoje programoje, pasitelkiant *#pragma omp parallel for* direktyvą, sukuriamos atskiros lygiagrečios sritys tose vietose, kuriuose yra vykdomi skaičiavimai. Antroje naudojama *#pragma omp parallel* direktyva, kuri sukuria lygiagrečią sritį visoje programoje, o vėliau vykdant skaičiavimų ciklus įterpiamos *#pragma omp for* direktyvos.

Anksčiau buvo pateikta nuosekli programa, tad šiame poskyryje bus aptariamos tik tos kodo dalys kuriomis papildoma programa.

Norint programoje naudoti papildomas OpenMP funkcijas, kuriomis galima kontroliuoti lygiagretumą, turime į programą įtraukti *omp.h* failą. Funkcija *omp_set_num_threads(int num_threads)*, kuri programoje nurodo, į kiek lygiagrečiųjų gijų skaidoma programa. Į šią funkciją privaloma kreiptis prieš išskiriant lygiagrečiąją sritį. Kita funkcija, kuria galima sužinoti programą vykdančiosios gijos numerį yra *int omp_get_thread_num(void)*. Funkcija gali būti iškviečiama bet kurioje lygiagrečios srities dalyje. Pateikiamose programose pastaroji funkcija reikalinga tik tam, kad sužinotume, kuri gija yra pagrindinė.

Realizuotose OpenMP programos yra keli nauji kintamieji: *num_threads* – saugomas gijų skaičius; *tid* – saugomas vykdančios gijos numeris intervale $(0, num_threads)$; *chunk_size* – duomenų bloko dalis tenkanti vienai gijai. Bloko dydis apskaičiuojamas visas eilutes padalinant iš programą vykdančių gijų skaičiaus $chunk_size = (N+1)/num_threads$. Jei dalyba be liekanos negalima, lygiomis dalimis paskirstyti nepavyksta, tada OpenMP programa vykdydama lygiagretųjį *for* ciklą, visas likusias eilutes atiduos nuliniai gijai.

Pateikiama pirmoje programoje (2 priedas) mažai skiriasi nuo nuoseklios programos. Šioje programoje prieš *for* ciklus įterpiama direktyva *#pragma omp parallel for private(i, j) schedule(static, chunk_size)*, kuri išskiria lygiagrečią sritį skaičiavimų metu. Programos vykdymo metu, kai sutinkama *parallel for* direktyva, OpenMP sukuria lygiagrečiąją sritį vykdančias gijas. Pasibaigus lygiagrečiai sričiai, gijų veikimas laikinai sustabdomas ir laukia kitos lygiagrečiosios srities. Lygiagrečiojoje srityje, kurioje vertinama paklaida, bendrojo kintamojo *paklaida* atnaujinimas atliekamas *#pragma omp critical* direktyvos srityje, tokiu būdu, vienu metu kintamąjį gali modifikuoti tik viena gija.

Vykdamas atskiras lygiagrečias sritis, sugaišamas laikas sustabdant ir vėl vykdamas lygiagrečiąsias sritis [10]. Papildomo apkrovimo galima išvengti sukuriant vieną lygiagrečią sritį visai programai (3 priedas). Lygiagrečioji programa pradedama vykdyti išskiriant lygiagrečiąją sritį *#pragma omp parallel shared(num_threads, chunk_size, paklaida) private(tid)*. Nurodoma, kad kintamieji *num_threads*, *chunk_size* ir *paklaida* yra bendrieji, o kintamasis *tid* yra vietinis ir skirtas kiekvienos gijos numeriui saugoti. Lygiagrečiosios srities pradžioje sužinomas vykdančios gijos numeris *tid = omp_get_thread_num()*. Paskutinis veiksmas, kurį atliekamas prieš pradedant vykdyti skaičiavimus, yra laiko fiksavimas. Lygiagrečiojoje programoje šis veiksmas atliekamas tik nuliniame procese.

Pradedamos vykdyti iteracijos *while* ciklo pagalba, kurios vykdomos iki tol kol dviejų gretimų artinių norma nėra mažesnė už anksčiau apibrėžtą paklaidą. Ciklo viduje vykdomi du dvigubi *for* ciklai. Prieš kiekvieną dvigubą ciklą įterpiama *#pragma omp for* direktyva. OpenMP

lygiagrečiai vykdo tik išorinį ciklą, taigi duomenys tarp procesorių yra padalijami eilutėmis. Prie kiekvienos direktyvos galime nurodyti, kurie kintamieji yra bendri, o kurie vietiniai. Prie *#pragma omp for* direktyvų nurodoma, kad cikluose naudojami indeksai *i* ir *j* yra vietiniai. Jei daugiau nenurodome nieko, tada duomenys tarp procesų yra padalinami cikliška. Norint duomenis paskirstyti blokais, reikia įterpti papildomą sakinį *schedule(kind, chunk_size)*, kur *kind* – nurodo, koku būdu eilutes norima padalinti: statiška ar dinamiška. *Chunk_size* – nurodo kokio dydžio blokais padalijami duomenys. Jeigu *chunk_size* reikšmė būtų nurodyta 1, gaunamas paprastas ciklinis duomenų padalijimas. Norint, kuo tobuliau padalinti duomenis tarp procesų, reikia visas eilutes padalinti į tiek blokų kiek yra programą vykdančių gijų:

$chunk_size = (N+)/num_threads$, kur *N* – eilučių skaičius, *num_threads* – gijų skaičius.

Iš pateikto lygiagrečiojo uždavinio sprendimo modelio žinoma, kad gijos turi pasiekti duomenis priklausančius gretimoms gijoms. Šiuo atveju OpenMP nereikia atlikti jokių papildomų veiksmų, nes direktyvos pačios pasirūpina duomenų paskirstymu tarp gijų, o naudojama bendra atmintis, nereikalauja jokių duomenų mainų tarp jų.

Antrajame dvigubame *for* cikle, kuriame įvertinama paklaida, *skirt* ir *pakl* kintamieji aprašomi kaip vietinis. Tokiu būdu kiekviena gija skirtumą tarp dviejų gretimų artinių taškų skaičiuoja individualiai. Kiekvienai gijai apskaičiavus vietinę paklaidą, tikrinama ar vietinė paklaida yra didesnė už bendrą paklaidą, jei taip yra bendras kintamasis *paklaida* yra atnaujinamas. Prieš bendros paklaidos atnaujinimą naudojama direktyvą *#pragma omp critical*, tam, kad vienu metu modifikuoti kintamąjį galėtų tik viena gija. Tai daroma norint išvengti klaidų. Pastaroji direktyva atlieka sinchronizavimą reikalinga norint, kad visi procesai vykdytų iteracijas vienu metu. Baigus vykdyti iteracijas, kaip ir prieš pradedant jas vykdyti, taip ir po jų vykdymo pabaigos, nulinė gija fiksuoja laiką.

Iš pateikiamų programų matoma, kad lygiagrečiosios programos nežymiai skiriasi nuo nuosekliosios. Tokias OpenMP programas, galima vykdyti nuosekliai, nes direktyvos informatyvios tik specializuotam OpenMP kompiliatoriui, o standartinis C kalbos kompiliatorius į jas reaguoja kaip į paprasčiausius komentarus.

3.2.2. UPC

UPC lygiagrečiojo programavimo technologija, kaip ir OpenMP, yra skirta bendrosios atminties kompiuterių architektūrai. Priešingai nei OpenMP, UPC programą iškart vykdo visos lygiagrečiosios gijos, todėl apibrėžiant duomenų tipus, reikia iš anksto nuspręsti, kurie duomenys bus bendri, o kurie vietiniai. Taip pat sudėtingesnis lygiagrečiųjų ciklų vykdymas, nes nepakanka vien tik įterpti direktyvą, o reikia pakeisti ciklo struktūrą [9]. Vietoje įprastinio *for* ciklo, naudojamas *upc_forall* ciklas, kuriame atsiranda ketvirtasis kintamasis, per kurį nurodoma, kuri gija

susieta su vykdančia gija. Nors ir sudėtingesnis lygiagretumo realizavimas, tačiau jis suteikia didesnę kontrolę programuotojui, bei leidžia pačiam nuspręsti, kaip konkrečiai duomenys turi būti paskirstomi procesams.

Realizuojant lygiagrečią UPC programą (4 priedas), reikia nepamiršti į programą įtraukti *upc.h* ir *upc_collective.h* failus, kuriuose aprašytos UPC funkcijos. Taip pat kaip ir nuoseklioje programoje, taip ir lygiagrečioje programoje, apibrėžiama, kaip bus suprantami duomenų masyvai programoje. Tačiau skiriasi pats duomenų išskirimas ir kintamųjų deklaravimas. Masyvas aprašomas kaip bendras visoms gijoms: *shared [BLOCK_SIZE] double *f, *u, *us*. Laužtiniuose skliaustuose nurodomas bloko dydis $BLOCK_SIZE = [(N+1)*100]$, tokiu būdu nurodoma, kad duomenys tarp gijų paskirstomi cikliška blokais po 100 eilučių. Siekiant uždavinį išlygiagretinti optimaliau, visą elementų kiekį reikėtų padalinti blokais po lygiai $[(N+1)*(N+1)/THREADS]$, tačiau dėl UPC apribojimų bloko dydžiui, tenka naudoti ciklinį blokų paskirstymą. Vėliau kintamiesiems *u*, *us*, *f* funkcija *upc_all_alloc* dinamiškai išskiriama atminties sritis, saugoti $(N+1)*(N+1)$ elementų. Aprašant kitus kintamuosius, pakinta tik kintamojo *shared double paklaida* aprašymas, jis tampa bendras. Atsiranda dar vienas kintamasis laikinai saugoti paklaidos reikšmei iteracijos metu *shared [1] pakl[THREADS]*. Išskiriamas bendras masyvas kuriame kiekvienas procesas saugos savo laikinąją paklaidos reikšmę. Laužtiniuose skliaustuose nurodomas vienetas, kuris reiškia, kad kiekvienai gijai yra skiriama saugoti po vieną, jam priklausančią masyvo reikšmę.

Vykdam programą ir paruošus pradines masyvų reikšmes, nulinis procesas fiksuoja programos skaičiavimų pradžios laiką. Taip pat kaip ir nuoseklioje programoje, pasitelkiant *while* ciklą, pradedamos vykdyti iteracijos. Kiekvienos iteracijos metu, atliekami skaičiavimai, paklaidos įvertinimai ir duomenų sukeitimo veiksmai.

Skaičiavimai atliekami pasitelkiant UPC lygiagretųjį *upc_forall* ciklą. Kaip jau buvo minėta anksčiau, *upc_forall* cikle atsiranda ketvirtasis kintamasis, kuris nusako kuriam procesui, kurią ciklo dalį vykdyti.

Kitu etapu vykdomas paklaidos įvertinimas. Prieš pradedant vykdyti paklaidos vertinimą, kiekviena gija apibrėžia, kad pradinė paklaidos reikšmė yra lygi nuliui ($pakl[MYTHREAD] = 0.$). Vėliau vykdomas lygiagretus *upc_forall* ciklas, taip pat, kaip ir atliekant skaičiavimus. Įvertinant paklaidą, kiekviena gija vertina tik jai tenkančių elementų paklaidą, o baigus vykdyti lygiagretų ciklą, pasitelkus procedūrą *upc_all_reduceD(&paklaida, pakl, UPC_MAX, THREADS, 1, NULL, UOC_IN_ALLSYNC | UPC_OUT_ALLSYNC)* iš visų gijų apskaičiuotų paklaidų išskiriama pati didžiausia ir išsaugoma bendrame kintamajame *paklaida*. Baigus vykdyti iteraciją, tikrinama, ar rasta paklaida nėra mažesnė už norimą. Jei sąlyga netenkinama, iteracijų vykdymas tęsiamas toliau.

Baigus vykdyti iteracijas, nulinis procesas vėl fiksuoja laiką. Po rezultatų išvedimo, dinaminės atminties išlaisvinimui naudojama funkcija *upc_free*.

Realizuojant UPC programą, galimi įvairūs duomenų paskirstymo variantai, bei skirtingos lygiagretumo kontrolės. Tačiau sprendžiant ganėtinai nesudėtingą uždavinį, kuriame skaičiavimams atlikti, pakanka vieno lygiagretaus ciklo, duomenų paskirstymas ir lygiagretumo kontrolė yra pakankama.

3.3. Paskirstytosios atminties kompiuterių architektūra

3.3.1. MPI

MPI lygiagrečiojo programavimo technologija skirta paskirstytosios atminties kompiuteriams. Ši technologija taip pat gali būti naudojama ir bendrosios atminties kompiuteriuose jei juose įdiegta tam pritaikyta MPI biblioteka, tačiau ne taip kaip su bendrosios atminties kompiuterių architektūrai skirtoms technologijomis, vis tiek reikia rūpintis duomenų mainais tarp lygiagrečiąją programą vykdančių procesų. Nepaisant to, kokios architektūros kompiuteriuose vykdome MPI lygiagrečiąją programą, programos kodo nereikia keisti.

Kaip ir bet kurioje kitoje lygiagrečiojo programavimo technologijoje, norint programoje (5 priedas) naudoti technologijos funkcijas, reikia į programą įtraukti MPI aprašomąjį failą *mpi.h*. Šiame faile aprašytos visos galimos funkcijos, su kuriomis yra vykdomas pranešimų persiuntimas tarp procesų. Aprašant duomenų masyvus, pasikeičia aprašymo forma. Čia vietoj *#define U(i,j) u[(j)*(N+1)+i]* programoje naudojama *#define U(i,j) u[(j-s+1)*(N+1)+i]*, kur *(j)* pakeičia *(j-s+1)*. Kintamųjų sąrašė atsiranda keletas papildomų kintamųjų, kurie reikalingi saugoti programą vykdančių procesų skaičių *p* ir programą vykdančio proceso numerį *myid*, kurio reikšmė yra intervale $(0..p-1)$. Kintamieji *s* ir *d* naudojami saugoti pirmos ir paskutinės procesui tenkančios eilutės reikšmei.

MPI lygiagrečiosios programos pradžioje iškviečiama funkcija *MPI_Init* kuri paruošia MPI darbo aplinką. Naudojant funkciją *MPI_Comm_size* sužinome lygiagrečią programą vykdančių procesų skaičių *p*, o naudojant funkciją *MPI_Comm_rank* sužinome programą vykdančio proceso numerį *myid*.

Toliau apskaičiuojama, kiek kiekvienam procesui tenka eilučių. Kadangi duomenys paskirstomi blokiniu būdu, tai visiems procesams eilutės padalijama po lygiai. Tačiau ne visada būna taip, kad visus duomenis pavyksta padalinti po lygiai. Anksčiau aptartose lygiagrečiojo programavimo technologijų programų realizacijose, skirtų bendrosios atminties kompiuteriams, pakako pasinaudojant direktyva nurodyti programai, kad skaičiavimai vykdomi lygiagrečiai ir duomenų lygiagretumu programa pasirūpindavo pati, arba nurodžius bloko dydį, visos perteklinės eilutės, būdavo automatiškai atiduotos nuliniam procesui. MPI tokios automatinės duomenų paskirstymo funkcijos neturi, todėl programuotojas turi tuo pasirūpinti pats. Šioje programos

realizacijoje, viso perteklinės eilutės yra paskirstomos pirmiesiems programą vykdančioms procesams.

Žinant, koks duomenų kiekis atitenka kiekvienam procesui, galima išskirti atmintį. Kadangi atmintį kiekvienas procesas turi išskirti pats, tai atminties išskyrimą atlieka visi procesai individualiai. Atminties išskyrimui MPI neturi savo funkcijos, o naudoja standartinę C kalbos funkciją *malloc()*: $u = (\text{double } *)\text{malloc}(\text{sizeof}(\text{double}) * (N+1) * (d-s+3))$. Anksčiau aptartose programose atminties būdavo išskiriama $(N-1) * (N+1)$, o šioje programoje kiekvienas procesas išskiria atminties tik tiek, kiek jam reikia, o ne visam masyvui. Išraiška $(N+1)$ keičiama į $(d-s+3)$, kur kintamasis d reiškia eilutę iki kurios tenka konkrečiam procesui, o kintamasis s reiškia eilutę, nuo kurios tenka konkrečiam procesui. Anksčiau išskiriant atmintį buvo pridėdama dar viena eilutė, skirta kraštinėms reikšmėms saugoti, o šioje programos realizacijoje, reikia saugoti papildomai tas kraštines eilutes, kurios priklauso gretimiesiems procesams, tad reikia saugoti papildomas dvi eilutes.

Išskyrę atmintį masyvams, galime iškviesti funkciją *Init()* ir inicializuoti duomenų reikšmes. Kiekvienas procesas inicializuoja tik jam skirtų duomenų pradines reikšmes. MPI programoje reikia modifikuoti pradinių reikšmių priskyrimo funkciją, nes reikia nurodyti, kad kiekvienas procesas atliktų darbą su jam priklausančiais duomenimis. Jei programa vykdoma tik su vienu procesu, tada kraštinių reikšmių priskyrimas atliekamas standartiškai, jei daugiau nei su vienu procesu, tada nulinis ir paskutinis procesai atlieka kraštinių reikšmių priskyrimą individualiai, o visi kiti tarpiniai procesai atlieka tuos pačius priskyrimo veiksmus.

Iteracijų vykdymas, kaip ir kitose programose, pradedamas *while* ciklu. Kiekvieno ciklo vykdymo pradžioje iškviečiama duomenų apsikeitimo tarp procesų procedūra *Exchange()*. Šioje procedūroje atliekami blokuoti persiuntimai *MPI_Send* ir *MPI_Recv*. Visi procesai, kurių numeris didesnis už nulį, persiunčia savo kraštinę eilutę gretimiesiems procesams, kurių numeris vienetu mažesnis $bottom = myid - 1$, o tie procesai, kurių numeris mažesnis už didžiausią programą vykdančio proceso numerį, persiunčia duomenis gretimiesiems procesams, kurių numeris vienetu didesnis $top = myid + 1$. Vykdamas blokuotą duomenų persiuntimą, procesų tolimesni veiksmai yra sustabdomi iki tol kol baigiamas atlikti duomenų apsikeitimas. Vėliau bus pateiktas programos variantas, kuriame duomenų apsikeitimas atliekamas pasitelkus neblokuotus persiuntimus *MPI_Isend* ir *MPI_Irecv*.

Atlikus duomenų mainus, galima vykdyti skaičiavimus. Vykdamas *for* ciklus nenaudojamos papildomos direktyvos ar funkcijos, o vietoj to išoriniame *for* cikle nurodoma, kad skaičiavimus kiekvienas procesas turi atlikti tik su tomis eilutėmis kurios jam atiteko, nuo eilutės s iki eilutės d . Įvertinant paklaidą, kiekvienas procesas skaičiavimus vykdo tik su savo eilutėmis ir kiekvienas atskirai apskaičiuoja paklaidą. Baigus vykdyti paklaidos įvertinimą, pasitelkus *MPI_Allreduce* funkciją, iš visų procesų apskaičiuotu paklaidų išrenkama maksimali ir išsaugoma kintamajame

paklaida, kurio reikšmė vėl perduodama visiems procesams. Ši funkcija taip pat atlieka darbų sinchronizavimą tarp programą vykdančių procesų. Kaip ir ankstesnėse programose, iteracijos yra vykdomos iki tol, kol tenkinamos abi *while* ciklo sąlygos.

Kadangi programoje naudojamas standartinis *malloc()* dinaminės atminties išskyrimas, tai ir atminties atlaisvinimo funkcija *free()* yra standartinė. Atminties atlaisvinimą atlieka kiekvienas procesas individualiai.

Paskutinė funkcija, kurią privalu iškviešti baigiant vykdyti MPI lygiagrečią programą yra *MPI_Finalize()*, kuri sustabdo MPI darbo aplinkos veikimą.

Kitoje programos realizacijoje naudojami neblokuoti duomenų persiuntimai *MPI_Isend* ir *MPI_Irecv*. Iš pradžių skelbiama, kad laukiama pranešimų, o vėliau paskelbiamas ir pats duomenų siuntimas:

```
if(myid < (p-1)) {
    MPI_Irecv(&US(1,d+1), nx, MPI_DOUBLE, top, 200, MPI_COMM_WORLD, &info[0]);
}
if(myid > 0) {
    MPI_Irecv(&US(1,s-1), nx, MPI_DOUBLE, bottom, 100, MPI_COMM_WORLD, &info[1]);
}
if(myid > 0) {
    MPI_Isend(&US(1,s), nx, MPI_DOUBLE, bottom, 200, MPI_COMM_WORLD, &info[2]);
}
if(myid < (p-1)) {
    MPI_Isend(&US(1,d), nx, MPI_DOUBLE, top, 100, MPI_COMM_WORLD, &info[3]);
}
```

Čia kintamasis *info* aprašomas kaip *MPI_Request info[4]*, nes vykdomos dvi siuntimo ir priėmimo operacijos. Neblokuotų siuntimų naudojimo tikslas – paskelbus duomenų siuntimą, nereikia laukti kol bus baigtas siuntimas, o iškart galima vykdyti kitus darbus. Kadangi nelaukiama persiuntimo pabaiga, galima pereiti prie skaičiavimų, kurie atliekami tik su tais elementais, kuriems nereikalingi kraštiniai elementai:

```
for(j = s+1; j < d; j++)
    for(i = 1; i < N; i++)
        U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;
```

Atlikę pagrindinius skaičiavimus, vykdome duomenų persiuntimo veiksmu užbaigimo. Tam naudojama funkcija *MPI_Waitany*, kuri laukia bet kurios iš pradėtų neblokuotų duomenų persiuntimo operacijos užbaigimo:

```
for(j = 0; j < 4; j++) {
    MPI_Waitany(4, info, &index, &status);
    if(index < 2)
        switch(index) {
            case 0:
                for(i = 1; i < N; i++)
```

```

        U(i,j) = (US(i-1,s)+US(i+1,s)+US(i,s-1)+US(i,s+1)-h*h*F(i,j))/4.;
        break;
    case 1:
        for(i = 1; i < N; i++)
            U(i,j) = (US(i-1,d)+US(i+1,d)+US(i,d-1)+US(i,d+1)-h*h*F(i,j))/4.;
            break;
    }
}

```

Kiekvienas procesas, išskyrus kraštinius, atlieka dvi duomenų siuntimo ir dvi duomenų priėmimo operacijas, tad yra laukiamas keturių operacijų užbaigimas. Tačiau, tik po dviejų iš jų atliekami skaičiavimai. Baigus priimti duomenis kiekvienas procesas atlieka paskutinius skaičiavimus su tais elementais, kuriems apskaičiuoti reikėjo apsikeisti duomenimis su gretimais procesais.

Paklaidos įvertinimas ir rezultatų surinkimas vykdomas lygiai taip pat kaip ir blokuotų duomenų persiuntimų programoje.

3.3.2. BSP

BSP lygiagrečiojo programavimo technologija, kaip ir MPI, yra skirta paskirstytos atminties kompiuterių architektūrai. Realizuotų programų struktūra taip pat ganėtinai panaši. Pagrindinis skirtumas tai komunikacijoms tarp procesų naudojamos funkcijos.

BSP galimi du skirtingi komunikacijų tipai, tai: DRMA ir BSMP. Pirmoje realizuotoje programoje (7 priedas), naudojamos BSMP komunikacijos, o antroje programoje (8 priedas) naudojamos DRMA komunikacijos.

Abejų programų, struktūra panaši į MPI programas. Kiekvienos iteracijos pradžioje vykdomas duomenų apsikeitimas, skaičiavimai, paklaidos vertinimas ir rezultatų surinkimas. Nors programos ir panašios, tačiau skiriasi pranešimų siuntimas.

Pirmoje programoje realizuotos BSMP komunikacijos. BSMP komunikacijos vykdo duomenų siuntimą. Apsikeitimo duomenimis procedūroje nustatomas pranešimo dydis *bsp_set_tagsize(&tag_size)*, kur *tag_size* yra lygus siunčiamų duomenų skaičiui. Visos duomenų siuntimo ir priėmimo operacijos vykdomos etapais. Po kiekvieno etapo, siekiant užtikrinti darbų pabaigą, vykdomas darbų sinchronizavimas. Duomenų siuntimas vykdomas panašiai kaip ir MPI blokuotų persiuntimų programoje. Pirmiausia duomenis siunčia tie procesai kurių *pid* yra mažesnis už procesų skaičių. Išsiuntus duomenis, visi procesai kurių *pid* didesnis už 0, kreipiasi į procedūrą *bsp_get_tag* ir gauna informaciją apie siunčiamus duomenis, o tada vykdomo duomenų paėmimą iš siuntimo eilės *bsp_move*. Toliau duomenų siuntimas vykdomas atvirkščiai ką tik įvykusiam, tie procesai kurie priėmė duomenis, dabar juos siunčia, o tie kurie siuntė juos priima.

Apsikeitę duomenimis, procesai vykdo vietinius skaičiavimus ir paklaidos vertinimą. Baigę vykdyti skaičiavimus, vis procesai kurių *pid* didesnis už 0, išsiunčia vietines paklaidas nuliniam

procesui. Išsiuntus paklaidas, nulinis procesas turi juos priimti, tik šį kartą, siunčiamas ne vienas duomenų masyvas, o atskiri duomenys. Kadangi iš anksto žinoma, kad duomenis išsiuntė visi išskyrus nulinį procesą, tai siunčiamų duomenų skaičius yra $nprocs-1$. Vykdamas duomenų priėmimo ciklą, gaunama informacija apie siunčiamus duomenis ir iš siuntimo eilės, paimama po vieną siųstą elementą. Iš kart atliekamas paklaidos vertinimas, lyginant ar paklaida nėra didesnė už turimą. Radę didžiausią paklaidą, turime ją išsiųsti visiems likusiems elementams. Pasinaudoję paprasčiausiu ciklu, iš nulinio proceso išsiunčiame paklaidos reikšmę kitiems procesams, kiti procesai, kiekvienas atskirai, iš savo siuntimų eilės pasiima paklaidos reikšmę.

Šioje programos realizacijoje, buvo naudojamos duomenų apsikeitimo funkcijos kurios labai panašios į MPI, o antroje programoje, kurioje realizuotos DRMA komunikacijos, vykdomas tiesioginis duomenų kopijavimas iš vieno proceso atminties į kito proceso atmintį.

```

bsp_push_reg(&US(1,s-1), nx*sizeof(double));
bsp_push_reg(&US(1,d+1), nx*sizeof(double));
bsp_sync();
if (pid < (nprocs-1)) {
    bsp_put(top, &US(1,d), &US(1,s-1), 0, nx*sizeof(double));
}
if (pid > 0) {
    bsp_put(bottom, &US(1,s), &US(1,d+1), 0, nx*sizeof(double));
}

for(j = s+1; j < d; j++)
    for(i = 1; i < N; i++)
        U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

bsp_sync();
bsp_pop_reg(&US(1,s-1));
bsp_pop_reg(&US(1,d+1));

```

Duomenų apsikeitimas vykdomas paruošiant statinę atminties sritį, kuri gali būti naudojama duomenims kopijuoti. Visi procesai, naudoja funkciją *bsp_push_reg*, kuri deklaruoja, kad paruošta atminties sritis kurioje bus saugomi gretimiems procesams priklausantys duomenys. BSP programose, visi veiksmai susiję su duomenų mainais, vykdomi atskirais etapais, todėl po kiekvieno etapo vykdomas sinchronizavimas. Toliau naudojama duomenų kopijavimo funkcija *bsp_put*. Procesai deklaruoja duomenų kopijavimą į kito proceso atmintį. Iš pradžių procesai kopijuoja duomenis į tų procesų atmintį kurių *pid* vienetu mažesnis, o vėliau kopijuos į tų procesų atmintį kurių *pid* vienetu didesnis. Duomenų kopijavimas į kito proceso atmintį vykdomas iki kito superetapo, todėl duomenų kopijavimo laiką, panašiai kaip ir MPI neblokuotų duomenų siuntimo programoje, galima išnaudoti skaičiavimams. Po šio veiksmo, kaip ir po atminties paruošimo yra vykdomas sinchronizavimas. Pabaigus vykdyti duomenų kopijavimą, naudojant funkciją *bsp_pop_reg* paskelbiame, kad atminties sritis, kuri buvo naudojama komunikacijoms, toliau

komunikacijoms nebenaudojama. Atliekant tokį duomenų kopijavimą, kiekvienas procesas nukopijuoja dalį savo atminties į gretimų procesų atmintį.

Baigus vykdyti duomenų mainus, vykdomi skaičiavimus ir paklaidos vertinimas. Visi procesai apskaičiuoja vietines paklaidas.

```
bsp_sync();
bsp_push_reg(pk,nprocs*sizeof(double));
bsp_sync();
bsp_put(0,&pakl,pk,pid*sizeof(double),sizeof(double));
bsp_sync();
bsp_pop_reg(pk);
bsp_sync();
if (pid == 0) {
    paklaida = 0.;
    for(i = 0; i < nprocs; i++) {
        if(pk[i] > paklaida) paklaida = pk[i];
    }
}
bsp_sync();
bsp_push_reg(&paklaida,sizeof(double));
bsp_sync();
if (pid == 0)
    for (i = 1; i < nprocs; i++)
        bsp_put(i,&paklaida,&paklaida,0,sizeof(double));
bsp_sync();
bsp_pop_reg(&paklaida);
bsp_sync();
```

Kaip ir apsieičiant duomenimis, paskelbiame apie komunikacijoms naudojama duomenų sritį. Paskelbiamas *pk[nprocs]* masyvas, į kurį visi procesai nukopijuos vietines paklaidas. Toliau nulinis procesas išrenka iš visų vietinių paklaidų, saugomų *pk* masyve, didžiausią paklaidą ir išsaugo ją kintamajame *paklaida*. Dabar, kai jau žinoma maksimali paklaida, jos reikšmė turi būti nukopijuota kitiems procesams. Vėl visi procesai deklaruoja apie komunikacijoms paruošta atminties sritį. Toliau nulinis procesas nukopijuoja kintamojo *paklaida* reikšmę į kitų procesų atmintį.

BSPonMPI bibliotekoje yra tik pagrindinės funkcijos, kurios naudojamas standartinėms komunikacijoms vykdyti, tačiau jų tarpe nėra alternatyvos funkcijai *MPI_Allreduce*, todėl rezultatų surinkimui naudojamos tokios pačios funkcijos kaip ir duomenų siuntimui. Tokių rezultatų surinkimą ir paskleidimą, vykdyti gana keblu, bet žinant komunikacijų veikimo principus, tai daryti nėra sudėtinga.

4. Rezultatų palyginimas

4.1. Naudojama techninė įranga

Šiame darbe skaičiavimai buvo atlikti naujajame VGTU PK telkinyje vilkas.vgtu.lt, kuriame veikia Linux operacinė sistema ir įdiegti Intel kompiliatoriai. Šį PK telkinį sudaro 15 asmeninių kompiuterių, kurių techniniai duomenys yra:

Procesorius: Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz / 4 branduoliai

RAM: 2 Gb

HDD: 320Gb

LAN: 1Gb.

4.2. Bendrosios atminties kompiuterių architektūra

Programuojant bendrosios atminties kompiuterių architektūrai skirtomis lygiagrečiojo programavimo technologijomis, programuotojui nereikia rūpintis duomenų mainams tarp lygiagrečiąją programą vykdančių procesų, tačiau kaip pamatysime iš gautų programų vykdymo rezultatų, ne visada tas paprastumas yra naudingas.

Toliau pateikiami OpenMP programų vykdymo rezultatai (1 lentelė), išskiriant atmintį dinamiškai ir statiniu būdu.

1 lentelė. OpenMP programų vykdymo laikai, išskiriant atmintį dinamiškai ir statiška

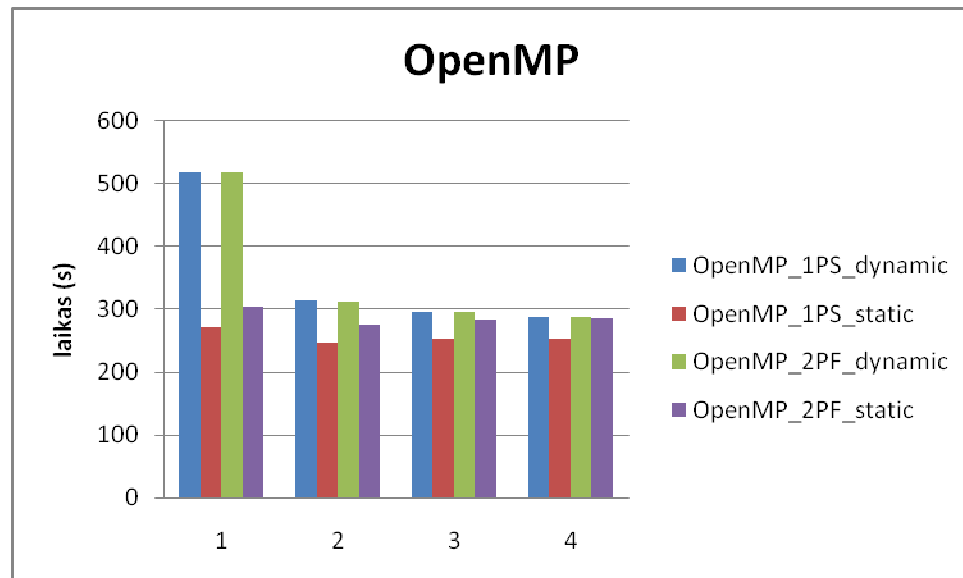
proces ai	OpenMP_1PS_dyna mic	OpenMP_1PS_st atic	OpenMP_2PF_dyna mic	OpenMP_2PF_st atic
1	518,920586	271,190946	518,138644	303,594015
2	313,331970	247,940218	312,817141	275,497807
3	296,502655	252,959095	296,438415	283,591742
4	287,030695	252,481873	287,093746	286,311610

Toliau pateikiamame laikų palyginimo grafike (5 pav.) matome, kaip skiriasi programų vykdymo laikai, esant skirtingam atminties išskyrimo būdui. Toks didelis laikų skirtumas atsiranda dėl to, kad kompiliuojant programą su statiška išskirta atmintimi, kompiliatorius gali automatiškai atlikti ciklą optimizavimą. Toks optimizavimas leidžia programą vykdyti geriau ir gauti geresnius vykdymo laikus.

Visose kitose realizuotose programose naudojama dinaminė atmintis, todėl lyginant rezultatus bus vadovaujamasi programos rezultatais, gautais iš dinamiškai būdu išskiriant atmintį.

Tarpusavyje lygindami rezultatus, gautus programa, kurioje išskiriama statinė atmintis, matome, kad programa, kurioje vykdoma viena lygiagreti sritis veikia geriau, už tą, kurioje išskiriamos atskiros lygiagrečios sritys vykdančios *for* ciklus. Tai yra todėl, kad išskiriant atskiras lygiagrečias sritis, programai yra daugiau duodama darbo ir vykdoma ilgiau [10]. Panašių rezultatų

reiktų tikėtis ir iš programų, kuriose vykdomas dinaminis atminties išskyrimas, tačiau taip nėra. Taip gali nutikti dėl to, kad ir taip nedidelis programų vykdymo skirtumas dingsta dėl padidėjusio programos vykdymo laiko, išskiriant dinaminę atmintį.



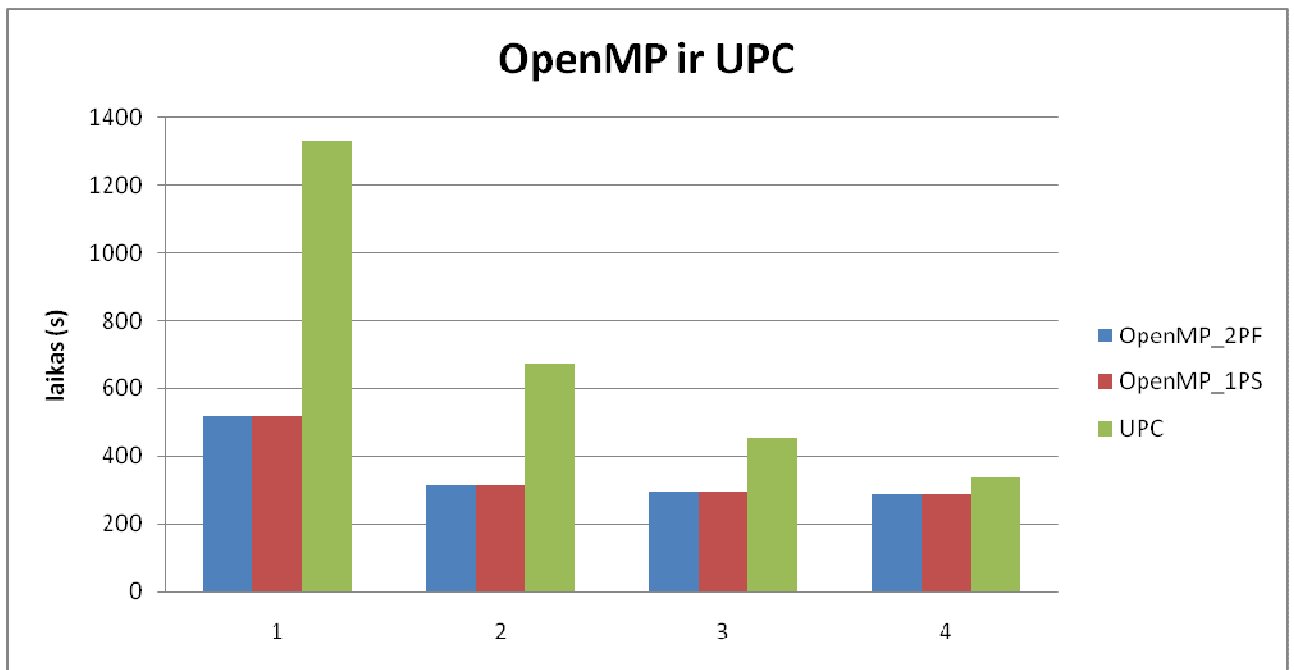
5 pav. OpenMP laikų palyginimas, išskiriant atmintį dinamiškai ir statiškai

Bendrosios atminties kompiuterių architektūrai realizuota programa su UPC lygiagrečiojo programavimo technologija. Kaip matome iš toliau pateikiamų OpenMP ir UPC programų vykdymo laikų palyginimo (2 lentelė), UPC programa atsilieka nuo OpenMP.

2 lentelė. OpenMP ir BSP programų vykdymo laikai

Procesai	OpenMP_2PF	OpenMP_1PS	UPC
1	518,138644	518,920586	1332,44678
2	312,817140	313,33197	672,984002
3	296,438415	296,502655	450,470677
4	287,093746	287,030695	336,555458

Toks didelis UPC programos vykdymo laiko skirtumas (2 lentelė) (6 pav.), vykdant programą su vienu procesu, gali atsirasti dėl kitokio nei kitur atminties išskyrimo ir UPC naudojamo *for* ciklo. Ši programa nėra visiškai išbaigta, nes su UPC teko susipažinti per labai trumpą laiko tarpą. Tačiau žvelgiant į lygiagrečiojo uždavinio vykdymą, didėjant procesų skaičiui, rezultatai tampa panašesni į tokius, kurių tikimasi iš lygiagrečiosios programos.



6 pav. OpenMP ir UPC programų vykdymo laikų palyginimas

Programuojant bendrosios atminties kompiuterių architektūrai skirtomis lygiagrečiojo programavimo technologijomis, nereikia rūpintis duomenų persiuntimu tarp procesų, tačiau programuojant su UPC, tenka rūpintis atminties padalijimu procesams ir vykdyti specifinį *for* ciklą, kuris gali sąlygoti ilgesnį programos vykdymo laiką.

4.3. Paskirstytosios atminties kompiuterių architektūra

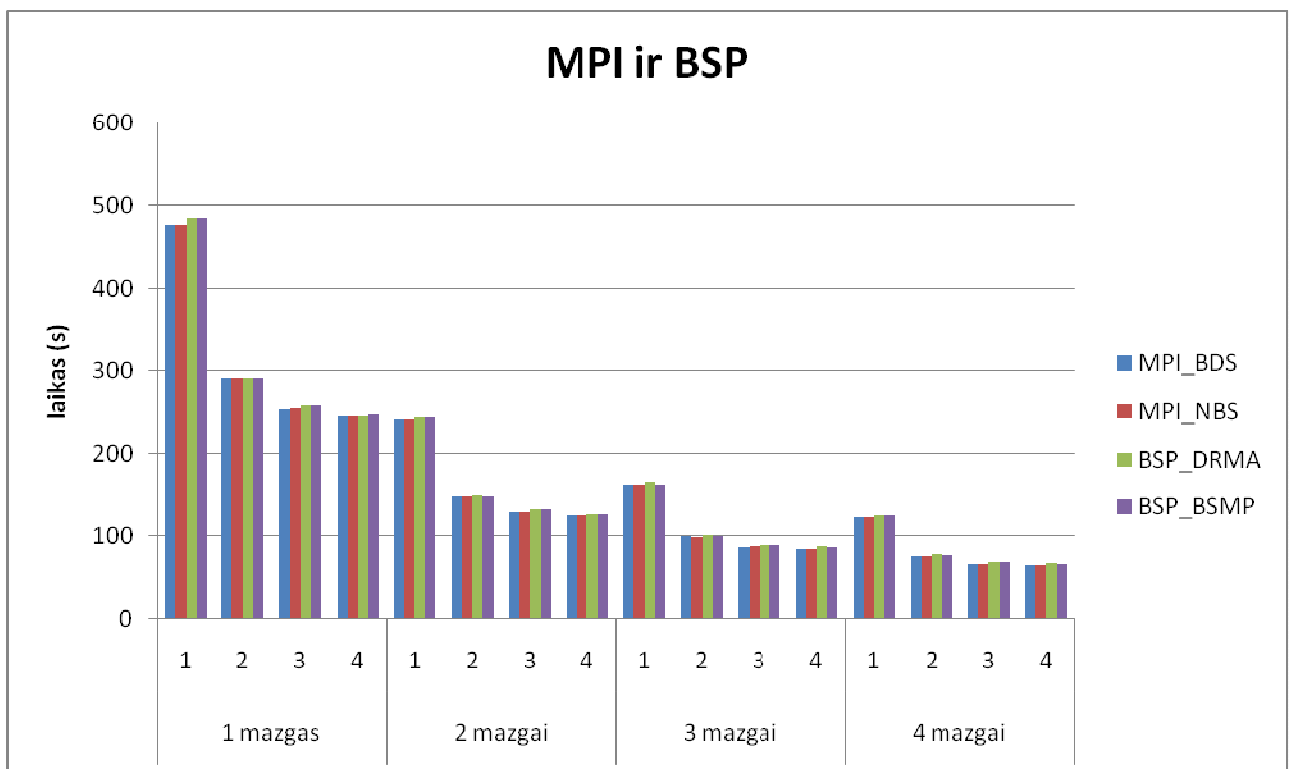
Paskirstytos atminties kompiuterių architektūrai skirtose programų realizacijose su MPI ir BSP programų vykdymo laikai yra labai panašūs (3 lentelė).

3 lentelė. MPI ir BSP programų vykdymo laikai

Mazgai	Procesai	MPI_BDS	MPI_NBS	BSP_DRMA	BSP_BSMP
1 mazgas	1	476,570553	477,285798	484,484571	485,011407
	2	290,285219	291,309881	290,668988	290,37666
	3	253,424527	254,761657	258,431791	258,701905
	4	246,425573	245,898459	246,914168	247,318802
2 mazgai	1	241,905038	241,779844	245,300554	244,292578
	2	147,210952	146,946891	148,570525	148,301767
	3	128,554206	128,671397	132,195364	131,779397
	4	124,982574	124,988159	126,865078	126,391824
3 mazgai	1	162,748771	162,006248	165,387683	163,74633
	2	99,644093	99,148323	101,709853	99,909421
	3	86,75338	86,945651	89,698705	89,680738
	4	84,827111	84,678518	87,067044	86,634206
4 mazgai	1	123,279382	122,835746	124,138637	124,607331
	2	75,656471	75,514034	77,87434	76,507986
	3	66,06198	65,713654	69,125566	68,592496
	4	64,7239	64,009575	67,912527	65,878889
5 mazgai	1	100,06379	98,186969	100,411115	99,921258
	2	62,069203	61,24447	62,391513	61,669341
	3	54,160055	53,195421	57,190771	56,228009
	4	53,111807	51,589246	55,854355	55,450847
6 mazgai	1	83,722229	82,601527	84,240422	83,918489
	2	53,121431	50,899964	65,451988	57,043914
	3	46,825779	45,223933	48,712998	48,79394
	4	46,190805	44,059037	48,838412	47,162634
7 mazgai	1	73,830288	72,080756	72,973665	72,592039
	2	47,613651	44,773369	57,202216	56,783915
	3	41,160286	38,948488	44,033095	42,59924
	4	40,399573	38,326367	45,33125	43,353029

Mazgai	Procesai	MPI_BDS	MPI_NBS	BSP_DRMA	BSP_BSMP
8 mazgai	1	66,148194	63,043806	64,577095	63,780267
	2	43,138987	42,822774	51,62374	41,198399
	3	37,45358	34,304246	39,825235	40,300372
	4	36,993512	32,945306	40,98321	38,03821
9 mazgai	1	59,666266	56,0471	57,971606	57,045088
	2	39,371532	37,014132	39,688008	38,764853
	3	34,453571	30,151233	35,743274	34,858511
	4	34,04948	28,308902	39,492518	36,525037
10 mazgų	1	55,281255	50,92207	52,902344	52,056605
	2	36,265542	31,644172	43,438128	38,03587
	3	31,734166	27,010048	33,602725	31,093302
	4	31,196934	25,575634	37,053987	33,334299

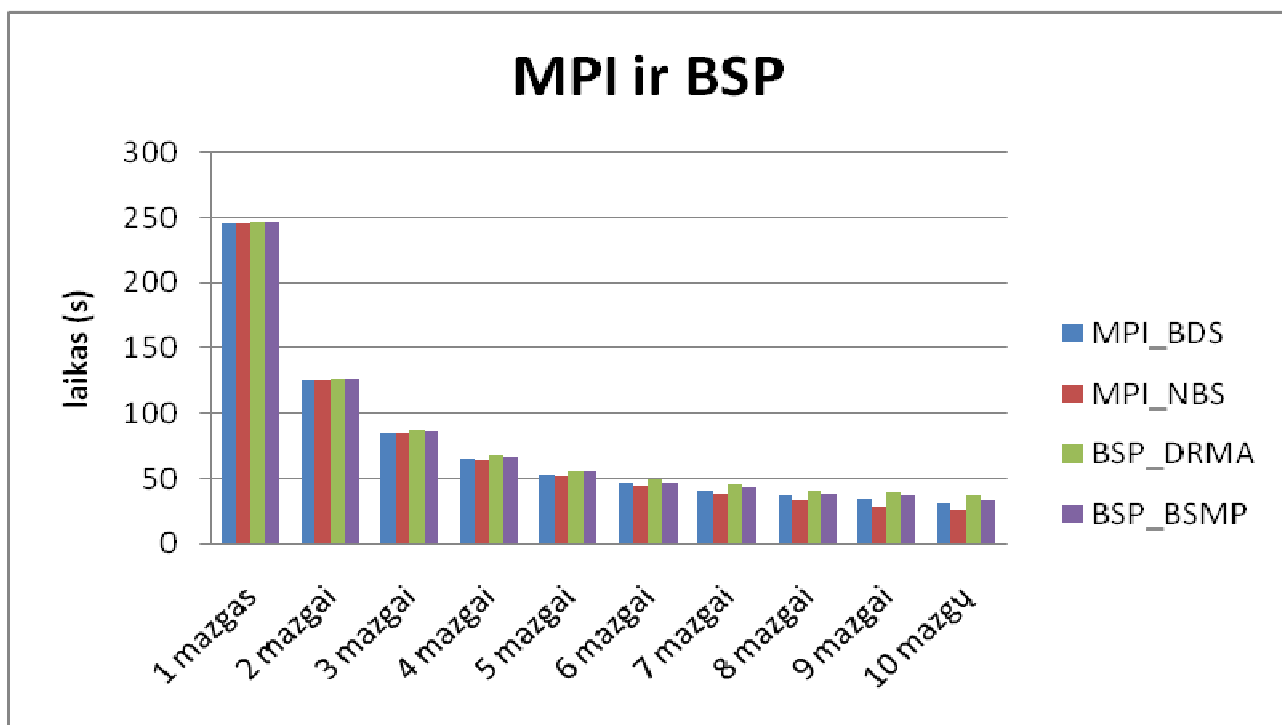
Toliau pateikiami skaičiavimo laikai (7 pav.), programą vykdant keturiuose mazguose. Matome tendencingą skaičiavimo laikų mažėjimą, didėjant programą vykdančių procesų skaičiui.



7 pav. MPI ir BSP programų vykdymo laikų palyginimas

Skaičiuojama buvo iš pradžių su vienu mazgu, skaičiavimus vykdant su 1, 2, 3 ir 4 procesais, po to su dviem mazgais, kiekviename skaičiavimus vykdant su 1, 2, 3 ir 4 procesais ir t.t. iki tol, kol pasiekiamas maksimalus procesų skaičius. Nors realiai PK telkinyje yra 15 kompiuterių,

tačiau skaičiavimus pavyko atlikti tik su 10 iš jų. Toliau pateikiami programų vykdymo laikai, kai kiekviename mazge, skaičiavimai vykdomi visais 4 procesais (8 pav.).



8 pav. MPI ir BSP programų vykdymo laikai, iki 10 mazgų, kiekviename po 4 procesus

Su MPI ir BSP buvo realizuotos po dvi lygiagrečiąsias programas. MPI_BDS tai MPI blokuotų duomenų persiuntimų programa, MPI_NBS – MPI neblokuotų duomenų persiuntimo programa. BSP_DRMA tai BSP naudojanti DRMA duomenų persiuntimo mechanizmą, o BSP_BSMP tai BSP naudojanti BSMP duomenų persiuntimo mechanizmą.

Kaip ir buvo galima tikėtis, MPI neblokuotų duomenų persiuntimo programa MPI_NBS vykdoma greičiau nei blokuotų duomenų persiuntimo MPI_BDS. Sprendžiamas uždavinys yra nesudėtingas ir didesnio neblokuoto duomenų persiuntimo programos efektyvumo pagauti nepavyko, nes persiunčiamas duomenų kiekis nėra didelis, o skaičiavimai atliekami gana sparčiai, todėl sunkiai galima paslėpti duomenų persiuntimą.

BSP programų vykdymo laikai labai panašūs į MPI, tačiau MPI visgi vykdoma geriau. BSP įgyvendinant BSPonMPI tam daro įtaką naudojamos MPI komunikacijos duomenims persiųsti. Be to, abiem technologijomis realizuotos programos labai panašios ir tai taip pat turi įtakos jų vykdymo laikui.

BSP_BSMP programos vykdymo laikas, didėjant vykdančių procesų skaičiui, yra geresnis negu BSP_DRMA. Šių programų realizacijose BSP_BSMP programoje vykdoma mažiau superetapų, tai yra buvo vykdoma mažiau darbų sinchronizavimo etapų, todėl ir gautas geresnis

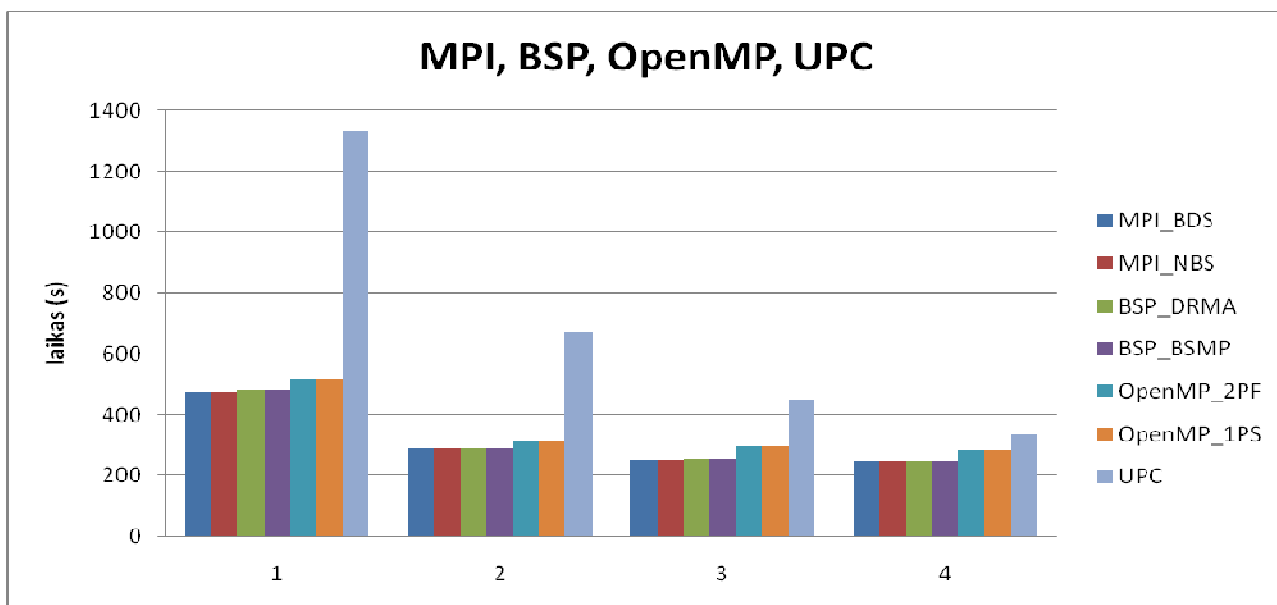
programos vykdymo laikas. Tačiau tai gali skirtis priklausomai nuo pasirinkto uždavinio ir nuo pasirinkto BSP įgyvendinimo būdo.

Žvelgdami apskritai į rezultatus matome, kad skaičiuodami vienu mazgu, su skirtingu procesų skaičiumi, rezultatus gauname ne tokius gerus kaip skaičiuodami skirtingais mazgas. Taip yra todėl, kad skaičiuojant vienu mazgu, procesai naudoja bendrą atmintį ir kreipiasi į ją per tą pačią magistralę (angl. Fron Side Buse). Tokiu būdu, bandydami pasiekti atmintį per tą pačią magistralę, jie ją užkemša ir nebesugeba sparčiai paimti duomenų. Siekdami geresnio rezultato už gautą rezultatą skaičiuojant vienu mazgu visais keturiais procesais, pakanka programą vykdyti dviem atskirais mazgais, kiekviename iš jų vykdant tik po vieną procesą. Iš kart pastebimas spartesnis duomenų paėmimas iš atminties. Kuo daugiau mazgų, tuo geresnis programos vykdymo laikas, vykdant kiekvienam mazge po keturis procesus, programos vykdymo laikas tampa dar geresnis, aišku nepaisant to, kad viename mazge procesai turi dalintis viena magistrale.

Dar vienas svarbus aspektas, į kurį tenka atsižvelgti vykdant lygiagrečiąsias programas, yra tas, kad siekiant maksimalių rezultatų, reikia atsižvelgti į tai, ar mazge, kuriame vykdomi darbai, tuo metu nėra vykdomos kitos užduotys. Jei kelios programos vienu metu naudoja tą patį mazgą skaičiavimams, tai tokių programų skaičiavimo rezultatais negalima tikėti. Taip yra dėl to, kad viename mazge, nors ir skirtingos programos, jos vis vien naudoja tą pačią magistralę, todėl darbus patartina vykdyti rezervavus visą mazgą.

4.4. Galutinis palyginimas

Apibendrinant visų programų vykdymo rezultatus (9 pav.) matome, kad iš realizuotų programų geriausiai su uždaviniu susidoroja MPI, antroje vietoje BSP, trečioje OpenMP, o paskutiniojoje UPC.



9 pav. MPI, BSP, OpenMP ir UPC programų vykdymo laikų palyginimas

Lygiagrečiojo programavimo technologijos skirtos paskirstytos atminties kompiuterių architektūrai, reikalauja daug daugiau laiko iš programuotojo, nei realizuojant tas pačias programas bendrosios atminties kompiuterių architektūrai skirtomis lygiagrečiojo programavimo technologijomis. Vykdamas skaičiavimus PK telkinyje, patrauklesnės paskirstytos atminties kompiuterių architektūrai skirtos lygiagrečiojo programavimo technologijos MPI ir BSP. Norint tuos pačius skaičiavimus vykdyti bendrosios atminties kompiuterių architektūrai skirtomis priemonėmis OpenMP ar UPC, PK telkinyje reikia įdiegti papildomą programinę įrangą, kuri leistų darbus vykdyti atskiruose mazguose. Tačiau OpenMP ir UPC priemonės, kuriose lygiagrečiojo programavimo technologijos pačios atlieka duomenų mainus, atsiranda, papildomas kompiuterinio tinklo apkrovimas, kurio programuotojas negali kontroliuoti. MPI ir BSP programų realizacijose, atliekami vienkartiniai duomenų mainai, kiekvienos iteracijos metu, o OpenMP ir UPC, programose vyksta nekontroliuojamas duomenų nuskaitymas iš gretimo mazgo, todėl atsiranda papildomas apkrovimas.

Skaičiuojant vienu mazgu, skaičiavimus vykdamas su 1, 2, 3 ar 4 procesais, papildomas apkrovimas tenka ne kompiuteriniam tinklui, o magistralei per kuria visi procesai pasiekia bendrą atmintį.

Išvados

Šiame darbe aptartos ir realizuotos keturios skirtingos lygiagrečiojo programavimo technologijos MPI, BSP, OpenMP ir UPC. Programos vykdytos naujame vilkas.vgtu.lt PK telkinyje, kuris sudarytas iš Intel(R) Core(TM)2 Quad Q6600 keturių branduolių procesorius turinčių kompiuterių.

Atlikus skaičiavimus su skirtingomis technologijomis, nustatyta, kad efektyviausia yra MPI technologija. Atskiru atveju, išskiriant statinę atmintį, OpenMP lygiagrečiojo programavimo technologija yra pranašesnė, tačiau siekiant gauti lygiaverčius rezultatus realizuotoje programoje naudojama dinaminė atmintis ir dėl to programos skaičiavimo efektyvumas nukenčia.

MPI technologija nors ir sudėtinga, tačiau perpratus jos veikimą, galima pasiekti gerų rezultatų. Lyginant programų realizavimą laiko atžvilgiu, tai prie MPI tenka sugaišti bene daugiausiai laiko. Nesudėtingam uždaviniui, tokiam kaip šilumos laidumo uždavinys, laiko reikia daugiau nei programuojant kitomis technologijomis.

BPS programos realizacija buvo įgyvendinama BSPonMPI, todėl programų vykdymo laikai panašūs. Siekiant geriau išanalizuoti BSP lygiagrečiojo programavimo technologiją, reikėtų išbandyti kitus BSP įgyvendinimo būdus, kuriais būtų realizuota daugiau pagalbinių funkcijų. Programos realizavimo laiko klausimu, laiko reikia mažiau nei MPI programai, tačiau tai yra dėl to, kad BSP veikia panašiai kaip ir MPI, todėl perpratus vieną, galima nesunkiai perprasti ir kitą.

OpenMP realizuotos programos išskiriant statinę atmintį veikia geriau nei programos, išskiriančios dinaminę atmintį, tačiau toks rezultatas gali pasireikšti ir modifikavus kitomis technologijomis realizuotas programas. OpenMP realizacijai sugaišta mažiausiai laiko, nes pakanka į nuosekliają programą įterpti vos kelias direktyvas ir gauname lygiagrečiąją programą.

UPC technologija, nors ir skirta bendrosios atminties kompiuterių architektūrai, tačiau reikalauja didesnio dėmesio iš programuotojo nei OpenMP technologija. Iš gautų testų rezultatų, UPC technologija pasirodė prasčiausiai, tačiau tiriant technologiją toliau, būtų galima išstudijuoti UPC technologiją atidžiau.

Realizuojant programas paskirstytos atminties kompiuterių architektūrai skirtomis lygiagrečiojo programavimo technologijomis, tenka sugaišti žymiai daugiau laiko, tačiau perpratus programų veikimo būdą galima nesunkiai pasiekti norimų rezultatų.

Bendrosios atminties kompiuterių architektūrai skirtų programų realizacija yra paprastesnė, bent jau su OpenMP. Programuoti su UPC teko labai trumpai, todėl neviseškai išbaigta programa duoda nepatikimus rezultatus.

Renkantis lygiagrečiojo programavimo technologiją vienai ar kitai kompiuterių architektūrai, bei priklausomai nuo sprendžiamo uždavinio, reiktų atsižvelgti į tai, kiek laiko bus galima skirti programos realizacijai, bei iki kokio lygio siekiama išlygiagretinti uždavinį. Paprastesniems atvejams pilnai pakanka OpenMP, na o sudėtingesniems reiktų rinktis MPI arba BSP.

Literatūra

1. Čiegis, R. 2005. Lygiagretieji algoritmai ir tinklinės technologijos. Technika, Vilnius. 334 p.
2. Čiegis, R. 2003. Diferencialinių lygčių skaitiniai sprendimo metodai. Technika, Vilnius. 439 p.
3. MPI: A Message-Passing Interface Standard [žiūrėta 2008 sausio 10 d.]. Prieiga per internetą:
<http://www.mpi-forum.org>
4. Message Passing Interface (MPI) [žiūrėta 2008 sausio 10 d.]. Prieiga per internetą:
<https://computing.llnl.gov/tutorials/mpi/>
5. OpenMP Application Program Interface specifikacija, 2005. [žiūrėta 2008 sausio 11 d.]. Prieiga per internetą:
<http://www.openmp.org>
6. D.B. Skillicorn. Predictable Parallel Performance: The BSP Model. [žiūrėta 2008 balandžio 20 d.]. Prieiga per internetą:
<http://citeseer.ist.psu.edu/330915.html>
7. BSP Worldwide Home Page [žiūrėta 2008 kovo 15 d.]. Prieiga per internetą:
<http://www.bsp-worldwide.org/>
8. S.Chauvin, P.Saha, F.Cantonnet, S.Anareddy, T.El-Ghazawi, UPC Language Specification v1.2. The George Washington University. 2005 [žiūrėta 2008 kovo 15 d.]. Prieiga per internetą:
<http://www.gwu.edu/~upc/documentation.html>
9. High Performance Parallel Programming with Unified Parallel C at SC05 [žiūrėta 2008 kovo 15 d.]. Prieiga per internetą:
<http://www.gwu.edu/~upc/tutorials.html>
10. More Work-Sharing with OpenMP [žiūrėta 2008 vasario 20 d.]. Prieiga per internetą:
<http://softwarecommunity.intel.com/articles/eng/1197.htm>

1 priedas. Nuosekliosios programos kodas

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define N 2000
#define U(i,j) u[(j)*(N+1)+i]
#define US(i,j) us[(j)*(N+1)+i]
#define F(i,j) f[(j)*(N+1)+i]

double *u, *us, *f;

void Init()
{
    int i, j;
    for(j = 1; j < N; j++)
        for(i = 1; i < N; i++)
            US(i,j) = 0;

    for(i = 0; i <= N; i++) {
        US(i,0) = 1;
        US(i,N) = 1;
    }
    for(j = 1; j < N; j++) {
        US(0,j) = 1;
        US(N,j) = 1;
    }
}

int main(int argc, char *argv[]) {
    int i, j, m;
    int it = 0;
    int itmax = 200;
    double paklaida;
    double skirt;
    double h = 1/N;
    double e = 0.001;
    struct timeval tim;
    double t1, t2;

    u = (double *) malloc(sizeof(double)*(N+1)*(N+1));
    us = (double *) malloc(sizeof(double)*(N+1)*(N+1));
    f = (double *) malloc(sizeof(double)*(N+1)*(N+1));

    Init();
    paklaida = 2*e;

    gettimeofday(&tim, NULL);
    t1=tim.tv_sec+(tim.tv_usec/1000000.0);
```

```

printf("pries iteracijas\n");

while((paklaida > e) && (it < itmax)) {
    for(j = 1; j < N; j++)
        for(i = 1; i < N; i++)
            U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

    paklaida = 0.;
    for(j = 1; j < N; j++)
        for(i = 1; i < N; i++) {
            skirt = fabs(U(i,j)-US(i,j));
            if(skirt > paklaida)
                paklaida = skirt;
            US(i,j) = U(i,j);
        }
    it++;
}

printf("po iteraciju\n");

gettimeofday(&tim, NULL);
t2=tim.tv_sec+(tim.tv_usec/1000000.0);

printf("N - %d Iteration Id: %d nuosekli programa uztruko - %.6f\n", N, it, t2-t1);

free (u);
free (us);
free (f);

return(0);
}

```

2 priedas. OpenMP lygiagrečiosios programos kodas (dvi lygiagrečiosios sritys)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define N 9000
#define U(i,j) u[(j)*(N+1)+i]
#define US(i,j) us[(j)*(N+1)+i]
#define F(i,j) f[(j)*(N+1)+i]

double *u, *us, *f;

void Init()
{
    int i, j;
    for(j = 1; j < N; j++)
        for(i = 1; i < N; i++)
            US(i,j) = 0;
    for(i = 0; i <= N; i++) {
        US(i,0) = 1;
        US(i,N) = 1;
    }
    for(j = 1; j < N; j++) {
        US(0,j) = 1;
        US(N,j) = 1;
    }
}

int main(int argc, char *argv[]) {
    int i, j;
    int it = 0;
    int itmax = 1000;
    double paklaida;
    double skirt;
    double h = 1/N;
    double e = 0.001;
    struct timeval tim;
    double t1, t2;

    int num_threads;
    int tid;
    int chunk_size;

    u = (double *) malloc(sizeof(double)*(N+1)*(N+1));
    us = (double *) malloc(sizeof(double)*(N+1)*(N+1));
```

```

f = (double *) malloc(sizeof(double)*(N+1)*(N+1));

Init();

num_threads = 4;
omp_set_num_threads(num_threads);
tid = omp_get_thread_num();

paklaida = 2*e;
chunk_size = (N+1) / num_threads;

gettimeofday(&tim, NULL);
t1=tim.tv_sec+(tim.tv_usec/1000000.0);

while((paklaida > e) && (it < itmax)) {

    #pragma omp parallel for private(i, j) schedule(static, chunk_size)
    for(j = 1; j < N; j++)
        for(i = 1; i < N; i++)
            U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

    paklaida = 0.;
    #pragma omp parallel for shared(paklaida) private(i, j, skirt) schedule(static, chunk_size)
    for(j = 1; j < N; j++)
        for(i = 1; i < N; i++) {
            skirt = fabs(U(i,j)-US(i,j));
            if(skirt > paklaida) {
                #pragma omp critical
                paklaida = skirt;
            }
            US(i,j) = U(i,j);
        }
    it++;
}
gettimeofday(&tim, NULL);
t2=tim.tv_sec+(tim.tv_usec/1000000.0);

printf("N - %d 2 p for Number of threads: %d Iteration id: %d OpenMP uztruks - %.6f\n",N,
num_threads, it, t2-t1);

free (u);
free (us);
free (f);

return(0);
}

```

3 priedas. OpenMP lygiagrečiosios programos kodas (viena lygiagrečioji sritis)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define N 9000
#define U(i,j) u[(j)*(N+1)+i]
#define US(i,j) us[(j)*(N+1)+i]
#define F(i,j) f[(j)*(N+1)+i]

double *u, *us, *f;

void Init()
{
    int i, j;
    for(j = 1; j < N; j++)
        for(i = 1; i < N; i++)
            US(i,j) = 0;

    for(i = 0; i <= N; i++) {
        US(i,0) = 1;
        US(i,N) = 1;
    }
    for(j = 1; j < N; j++) {
        US(0,j) = 1;
        US(N,j) = 1;
    }
}

int main(int argc, char *argv[]) {
    int i, j;
    int it = 0;
    int itmax = 360;
    double paklaida;
    double skirt;
    double h = 1/N;
    double e = 0.001;
    struct timeval tim;
    double t1, t2;

    int num_threads;
    int tid;
    int chunk_size;

    u = (double *) malloc(sizeof(double)*(N+1)*(N+1));
    us = (double *) malloc(sizeof(double)*(N+1)*(N+1));
    f = (double *) malloc(sizeof(double)*(N+1)*(N+1));
```

```

Init();

paklaida = 2*e;

#pragma omp parallel shared(paklaida) private(tid, it, skirt)
{
    num_threads = omp_get_num_threads();
    chunk_size = (N+1) / num_threads;
    tid = omp_get_thread_num();
    if (tid == 0) {
        gettimeofday(&tim, NULL);
        t1=tim.tv_sec+(tim.tv_usec/1000000.0);
    }

    while((paklaida > e) && (it < itmax)) {

        #pragma omp for private(i, j) schedule(static, chunk_size)
        for(j = 1; j < N; j++)
            for(i = 1; i < N; i++)
                U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

        paklaida = 0.;
        #pragma omp for private(i, j) schedule(static, chunk_size)
        for(j = 1; j < N; j++)
            for(i = 1; i < N; i++) {
                skirt = fabs(U(i,j)-US(i,j));
                if(skirt > paklaida) {
                    #pragma omp critical
                    paklaida = skirt;
                }
                US(i,j) = U(i,j);
            }
        it++;
    }
    if (tid == 0) {
        gettimeofday(&tim, NULL);
        t2=tim.tv_sec+(tim.tv_usec/1000000.0);
        printf("N - %d 1 ps Number of threads: %d Iteration id: %d OpenMP uztruko - %.6f\n",N,
num_threads, it, t2-t1);
    }
}

free (u);
free (us);
free (f);

return(0);
}

```

4 priedas. UPC lygiagrečiosios programos kodas

```
#include <upc.h>
#include <upc_collective.h>
#include <upc_relaxed.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define N 9000
#define U(i,j) u[(j)*(N+1)+i]
#define US(i,j) us[(j)*(N+1)+i]
#define F(i,j) f[(j)*(N+1)+i]
#define BLOCK_SIZE (N+1)*100

shared [BLOCK_SIZE] double *f, *u, *us;
shared double paklaida;
shared int it;
shared [1] double pakl[THREADS];

void Init()
{
    int i, j;
    upc_forall(j = 1; j < N; j++; j)
        for(i = 1; i < N; i++)
            US(i,j) = 0;

    if (MYTHREAD == 0)
        for(i = 0; i <= N; i++)
            US(i,0) = 1;

    if (MYTHREAD == THREADS-1)
        for(i = 0; i <= N; i++)
            US(i,N) = 1;

    upc_forall(j = 1; j < N; j++; j) {
        US(0,j) = 1;
        US(N,j) = 1;
    }

    upc_forall(j = 0; j <= N; j++; j)
        for(i = 0; i <= N; i++)
            US(i,j) = U(i,j);
}

int main(int argc, char *argv[]) {
    int i, j;
    int itmax = 1000;
    double skirt;
    double h = 1/N;
```

```

double e = 0.001;
struct timeval tim;
double t1, t2;

printf("MYTHREAD - %d; THREADS - %d\n",MYTHREAD,THREADS);

f = (shared [BLOCK_SIZE] double *) upc_all_alloc(THREADS,
BLOCK_SIZE*sizeof(double));
u = (shared [BLOCK_SIZE] double *) upc_all_alloc(THREADS,
BLOCK_SIZE*sizeof(double));
us = (shared [BLOCK_SIZE] double *) upc_all_alloc(THREADS,
BLOCK_SIZE*sizeof(double));

Init();
it = 0;
paklaida = 2*e;

if (MYTHREAD == 0) {
    gettimeofday(&tim, NULL);
    t1=tim.tv_sec+(tim.tv_usec/1000000.0);
}

while((paklaida > e) && (it < itmax)) {
    upc_forall(j = 1; j < N; j++; j)
        for(i = 1; i < N; i++)
            U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;
    pakl[MYTHREAD] = 0.;
    upc_forall(j = 1; j < N; j++; j)
        for(i = 1; i < N; i++) {
            skirt = fabs(U(i,j)-US(i,j));
            if(skirt > pakl[MYTHREAD])
                pakl[MYTHREAD] = skirt;
            US(i,j) = U(i,j);
        }
    upc_all_reduceD(&paklaida, pakl, UPC_MAX, THREADS, 1, NULL, UPC_IN_ALLSYNC |
UPC_OUT_ALLSYNC);
    if (MYTHREAD == 0) it++;
}

if (MYTHREAD == 0) {
    gettimeofday(&tim, NULL);
    t2=tim.tv_sec+(tim.tv_usec/1000000.0);
    printf("N - %d Number of threads: %d Iteration id: %d  UPC uztruko - %.6f\n",N,
THREADS, it, t2-t1);
}

if (MYTHREAD == 0) {
    upc_free(f);
    upc_free(u);
    upc_free(us);
}

```

```
    return(0);  
}
```

5 priedas. MPI lygiagrečiosios programos kodas (blokuotas duomenų persiuntimas)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define N 9000
#define F(i,j) f[(j-s+1)*(N+1)+i]
#define U(i,j) u[(j-s+1)*(N+1)+i]
#define US(i,j) us[(j-s+1)*(N+1)+i]

int myid, p, s, d;
double *u, *us, *f;

FILE *fopen(), *fr;
char *result_file = "result.dat";

void Exchange()
{
    int top, bottom, nx = N - 1;
    MPI_Status status;
    top = myid + 1;
    bottom = myid - 1;
    if(myid < (p-1)) {
        MPI_Send(&US(1,d), nx, MPI_DOUBLE, top, 100, MPI_COMM_WORLD);
    }
    if(myid > 0) {
        MPI_Recv(&US(1,s-1), nx, MPI_DOUBLE, bottom, 100, MPI_COMM_WORLD, &status);
    }
    if(myid > 0) {
        MPI_Send(&US(1,s), nx, MPI_DOUBLE, bottom, 200, MPI_COMM_WORLD);
    }
    if(myid < (p-1)) {
        MPI_Recv(&US(1,d+1), nx, MPI_DOUBLE, top, 200, MPI_COMM_WORLD, &status);
    }
}

void Init()
{
    int i,j;
    for(j = s; j <= d; j++)
        for(i = 1; i < N; i++)
            US(i,j) = 0.;

    if(p == 1) {
        for(i = 0; i <= N; i++) {
            US(i,0) = 1;
        }
    }
}
```

```

        US(i,N) = 1;
    }
    for(j = s; j <= d; j++) {
        US(0,j) = 1;
        US(N,j) = 1;
    }
} else {
    if(myid == 0) {
        for(i = 0; i <= N; i++)
            US(i,0) = 1;
    } else if(myid == p-1) {
        for(i = 0; i <= N; i++)
            US(i,N) = 1;
    }
    for(j = s; j <= d; j++) {
        US(0,j) = 1;
        US(N,j) = 1;
    }
}
}

int main(int argc, char *argv[]) {
    int i, j, m;
    int it = 0;
    int itmax = 1000;
    double paklaida;
    double pakl;
    double skirt;
    double h = 1/N;
    double e = 0.001;
    struct timeval tim;
    double t1, t2;
    double t3, t4;

    if (( fr = fopen( result_file , "a")) == NULL) {
        printf ( " Error : can't open %s\n " , result_file );
        exit(1);
    }

    MPI_Status stat;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    m = (N-1)/p;
    j = (N-1)%p;
    d = 0;
    for (i = 0; i <= myid; i++) {
        if(i == myid) s = d+1;
        d += m;

```

```

    if(i < j) d++;
}

u = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
us = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
f = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));

Init();

paklaida = 2*e;

if ( myid == 0) {
    gettimeofday(&tim, NULL);
    t1=tim.tv_sec+(tim.tv_usec/1000000.0);
}

while((paklaida > e) && (it < itmax)) {
    Exchange();
    for(j = s; j <= d; j++)
        for(i = 1; i < N; i++)
            U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

    pakl = 0.;
    for(j = s; j <= d; j++)
        for(i = 1; i < N; i++) {
            skirt = fabs(U(i,j)-US(i,j));
            if(skirt > pakl)
                pakl = skirt;
            US(i,j) = U(i,j);
        }
    it++;
    MPI_Allreduce(&pakl, &paklaida, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
}

if ( myid == 0){
    gettimeofday(&tim, NULL);
    t2=tim.tv_sec+(tim.tv_usec/1000000.0);
    fprintf(fr,"N - %d BDS Number of threads: %d Iteration Id: %d MPI uztruko - %.6lf\n", N, p,
it, t2-t1);
}

free(u);
free(us);
free(f);

MPI_Finalize();
return(0);
}

```

6 priedas. MPI lygiagrečiosios programos kodas (neblokuotas duomenų persiuntimas)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define N 4000
#define F(i,j) f[(j-s+1)*(N+1)+i]
#define U(i,j) u[(j-s+1)*(N+1)+i]
#define US(i,j) us[(j-s+1)*(N+1)+i]

int myid, p, s, d;
double *u, *us, *f;

MPI_Request *request;

FILE *fopen(), *fr;
char *result_file = "result.dat";

void Exchange() {

    int top, bottom, nx;
    nx = N - 1;
    top = myid + 1;
    bottom = myid - 1;
    if(myid < (p-1)) {
        MPI_Irecv(&US(1,d+1), nx, MPI_DOUBLE, top, 200, MPI_COMM_WORLD,
&request[1]);
    }
    if(myid > 0) {
        MPI_Irecv(&US(1,s-1), nx, MPI_DOUBLE, bottom, 100, MPI_COMM_WORLD,
&request[0]);
    }
    if(myid > 0) {
        MPI_Isend(&US(1,s), nx, MPI_DOUBLE, bottom, 200, MPI_COMM_WORLD,
&request[3]);
    }
    if(myid < (p-1)) {
        MPI_Isend(&US(1,d), nx, MPI_DOUBLE, top, 100, MPI_COMM_WORLD, &request[2]);
    }
}

void Init()
{
    int i,j;
    for(j = s; j <= d; j++)
```

```

        for(i = 1; i < N; i++)
            US(i,j) = 0.;

if(p == 1) {
    for(i = 0; i <= N; i++) {
        US(i,0) = 1;
        US(i,N) = 1;
    }
    for(j = s; j <= d; j++) {
        US(0,j) = 1;
        US(N,j) = 1;
    }
} else {
    if(myid == 0) {
        for(i = 0; i <= N; i++)
            US(i,0) = 1;
    } else if(myid == p-1) {
        for(i = 0; i <= N; i++)
            US(i,N) = 1;
    }
    for(j = s; j <= d; j++) {
        US(0,j) = 1;
        US(N,j) = 1;
    }
}
}

int main(int argc, char *argv[]) {
    int i, j, m;
    int it = 0;
    int itmax = 1000;
    double paklaida;
    double pakl;
    double skirt;
    double h = 1/N;
    double e = 0.001;
    struct timeval tim;
    double t1, t2;

    MPI_Status status;
    int index;

    if (( fr = fopen( result_file ,"a")) == NULL) {
        printf ( " Error : can't open %s\n " , result_file );
        exit(1);
    }

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

```

```

m = (N-1)/p;
j = (N-1)%p;
d = 0;
for (i = 0; i <= myid; i++) {
    if(i == myid) s = d+1;
    d += m;
    if(i < j) d++;
}

u = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
us = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
f = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
request = (MPI_Request*) malloc(sizeof(MPI_Request)*(4));

Init();

paklaida = 2*e;

for (i = 0; i < 4; i++)
    request[i] = MPI_REQUEST_NULL ;

if ( myid == 0){
    gettimeofday(&tim, NULL);
    t1=tim.tv_sec+(tim.tv_usec/1000000.0);
}

while((paklaida > e) && (it < itmax)) {

    Exchange();

    if (myid == 0)
        for(i = 1; i < N; i++)
            U(i,s) = (US(i-1,s)+US(i+1,s)+US(i,s-1)+US(i,s+1)-h*h*F(i,s))/4.;

    if (myid == p-1)
        for(i = 1; i < N; i++)
            U(i,d) = (US(i-1,d)+US(i+1,d)+US(i,d-1)+US(i,d+1)-h*h*F(i,d))/4.;

    for(j = s+1; j < d; j++)
        for(i = 1; i < N; i++)
            U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

    for(j = 0; j < 4; j++) {
        MPI_Waitany(4, request, &index, &status);
        if(index < 2)
            switch(index) {
                case 0:
                    for(i = 1; i < N; i++)
                        U(i,s) = (US(i-1,s)+US(i+1,s)+US(i,s-1)+US(i,s+1)-h*h*F(i,s))/4.;
                    break;
                case 1:
                    for(i = 1; i < N; i++)

```

```

        U(i,d) = (US(i-1,d)+US(i+1,d)+US(i,d-1)+US(i,d+1)-h*h*F(i,d))/4.;
        break;
    }
}

pakl = 0.;
for(j = s; j <= d; j++)
    for(i = 1; i < N; i++) {
        skirt = fabs(U(i,j)-US(i,j));
        if(skirt > pakl)
            pakl = skirt;
        US(i,j) = U(i,j);
    }
it++;
MPI_Allreduce(&pakl, &paklaida, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

if (myid == 0) printf("It - %d; paklaida - %.6lf\n", it, paklaida);
}

if ( myid == 0){
    gettimeofday(&tim, NULL);
    t2=tim.tv_sec+(tim.tv_usec/1000000.0);
    fprintf(fr,"N - %d NBS Number of threads: %d Iteration Id: %d MPI uztruko - %.6lf\n", N, p,
it, t2-t1);
}

free(u);
free(us);
free(f);
free(request);

MPI_Finalize();
return(0);
}

```

7 priedas. BSP lygiagrečiosios programos kodas (BSMP modelis)

```
#include <bsp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define F(i,j) f[(j-s+1)*(N+1)+i]
#define U(i,j) u[(j-s+1)*(N+1)+i]
#define US(i,j) us[(j-s+1)*(N+1)+i]

double *u, *us, *f;
int pid;
int nprocs;
int d, s;
int N;

void Init()
{
    int i,j;
    for(j = s; j <= d; j++)
        for(i = 1; i < N; i++) {
            US(i,j) = 0.;
            F(i,j) = 0.;
        }
    if(nprocs == 1) {
        for(i = 0; i <= N; i++) {
            US(i,0) = 1;
            US(i,N) = 1;
        }
        for(j = s; j <= d; j++) {
            US(0,j) = 1;
            US(N,j) = 1;
        }
    }
    else{
        if(pid == 0) {
            for(i = 0; i <= N; i++)
                US(i,0) = 1;
        }
        else if(pid == nprocs-1){
            for(i = 0; i <= N; i++)
                US(i,N) = 1;
        }
        for(j = s; j <= d; j++) {
            US(0,j) = 1;
            US(N,j) = 1;
        }
    }
}
```

```

int main(int argc, char *argv[]) {
    int i, j, m;
    int it = 0;
    int itmax = 1000;
    double paklaida;
    double pakl;
    double pk;
    double skirt;
    double h;
    double e = 0.001;
    struct timeval tim;
    double t1, t2;

    int top, bottom, nx;
    int tag_size;
    int status;
    int tag1 = 1;
    int tag2 = 2;
    int tag_p;
    int pk_count;
    int pk_bytes;

    bsp_begin(bsp_nprocs());

    nprocs = bsp_nprocs();
    pid = bsp_pid();

    N = 9000;
    h = 1/N;
    nx = N - 1;
    top = pid + 1;
    bottom = pid - 1;

    m = (N-1)/nprocs;
    j = (N-1)%nprocs;
    d = 0;
    for (i = 0; i <= pid; i++) {
        if(i == pid) s = d+1;
        d += m;
        if(i < j) d++;
    }

    u = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
    us = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
    f = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));

    Init();
    paklaida = 2*e;

    if (pid == 0){
        gettimeofday(&tim, NULL);

```

```

    t1=tim.tv_sec+(tim.tv_usec/1000000.0);
}

while((paklaida > e) && (it < itmax)) {

    tag_size = sizeof(int);
    bsp_set_tagsize(&tag_size);

    if (pid < (nprocs-1)) {
        bsp_send(top, &tag1, &US(1,d), nx*sizeof(double));
    }
    if (pid > 0) {
        bsp_send(bottom, &tag2, &US(1,s), nx*sizeof(double));
    }

    for(j = s+1; j < d; j++)
        for(i = 1; i < N; i++)
            U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

    bsp_sync();
    bsp_qsize(&pk_count, &pk_bytes);
    for (j = 0; j < pk_count; j++) {
        bsp_get_tag(&status, &tag_p);
        switch(tag_p) {
            case 1:
                bsp_move(&US(1,s-1), status);
                break;
            case 2:
                bsp_move(&US(1,d+1), status);
                break;
        }
    }
    bsp_sync();

    for(i = 1; i < N; i++) {
        U(i,s) = (US(i-1,s)+US(i+1,s)+US(i,s-1)+US(i,s+1)-h*h*F(i,s))/4.;
        U(i,d) = (US(i-1,d)+US(i+1,d)+US(i,d-1)+US(i,d+1)-h*h*F(i,d))/4.;
    }

    pakl = 0.;
    for(j = s; j <= d; j++)
        for(i = 1; i < N; i++) {
            skirt = fabs(U(i,j)-US(i,j));
            if(skirt > pakl)
                pakl = skirt;
            US(i,j) = U(i,j);
        }
    it++;

    if (pid > 0) {
        tag_p = pid;
        bsp_send(0, &tag_p, &pakl, sizeof(double));
    }
}

```

```

    }
    bsp_sync();
    if (pid == 0) {
        for (i = 0; i < nprocs-1; i++){
            bsp_get_tag(&status, &tag_p);
            bsp_move(&pk, status);
            if (pk > pakl) pakl = pk;
        }
        paklaida = pakl;
        for (i = 1; i < nprocs; i++)
            bsp_send(i, &tag_p, &paklaida, sizeof(double));
    }
    bsp_sync();
    if (pid > 0) {
        bsp_get_tag(&status, &tag_p);
        bsp_move(&paklaida, status);
    }
    bsp_sync();
}

if (pid == 0){
    gettimeofday(&tim, NULL);
    t2=tim.tv_sec+(tim.tv_usec/1000000.0);
    printf("N - %d BSMP Number of threads: %d Iteration Id: %d PBS uztruko - %.6lf\n", N,
nprocs, it, t2-t1);
}
bsp_sync();

free (u);
free (us);
free (f);

bsp_end();
return(0);
}

```

8 priedas. BSP lygiagrečiosios programos kodas (DRMA modelis)

```
#include <bsp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

#define F(i,j) f[(j-s+1)*(N+1)+i]
#define U(i,j) u[(j-s+1)*(N+1)+i]
#define US(i,j) us[(j-s+1)*(N+1)+i]

double *u, *us, *f;
int pid;
int nprocs;
int d, s;
int N;

void Init()
{
    int i,j;
    for(j = s; j <= d; j++)
        for(i = 1; i < N; i++) {
            US(i,j) = 0.;
            F(i,j) = 0.;
        }

    if(nprocs == 1) {
        for(i = 0; i <= N; i++) {
            US(i,0) = 1;
            US(i,N) = 1;
        }
        for(j = s; j <= d; j++) {
            US(0,j) = 1;
            US(N,j) = 1;
        }
    }
    else{
        if(pid == 0) {
            for(i = 0; i <= N; i++)
                US(i,0) = 1;
        }
        else if(pid == nprocs-1){
            for(i = 0; i <= N; i++)
                US(i,N) = 1;
        }
        for(j = s; j <= d; j++) {
            US(0,j) = 1;
            US(N,j) = 1;
        }
    }
}
```

```

int main(int argc, char *argv[]) {
    int i, j, m;
    int it = 0;
    int itmax = 1000;
    int top, bottom, nx;
    double paklaida;
    double pakl;
    double *pk;
    double skirt;
    double h;
    double e = 0.001;
    struct timeval tim;
    double t1, t2;

    bsp_begin(bsp_nprocs());

    nprocs = bsp_nprocs();
    pid = bsp_pid();

    N = 9000;
    h = 1/N;
    top = pid + 1;
    bottom = pid - 1;
    nx = N - 1;

    m = (N-1)/nprocs;
    j = (N-1)%nprocs;
    d = 0;
    for (i = 0; i <= pid; i++) {
        if(i == pid) s = d+1;
        d += m;
        if(i < j) d++;
    }

    u = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
    us = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
    f = (double *) malloc(sizeof(double)*(N+1)*(d-s+3));
    pk = (double *) malloc(sizeof(double)*nprocs);

    Init();
    paklaida = 2*e;

    if (pid == 0){
        gettimeofday(&tim, NULL);
        t1=tim.tv_sec+(tim.tv_usec/1000000.0);
    }

    while((paklaida > e) && (it < itmax)) {

```

```

bsp_push_reg(&US(1,s-1), nx*sizeof(double));
bsp_push_reg(&US(1,d+1), nx*sizeof(double));
bsp_sync();
if (pid < (nprocs-1)) {
    bsp_put(top, &US(1,d), &US(1,s-1), 0, nx*sizeof(double));
}
if (pid > 0) {
    bsp_put(bottom, &US(1,s), &US(1,d+1), 0, nx*sizeof(double));
}

for(j = s+1; j < d; j++)
    for(i = 1; i < N; i++)
        U(i,j) = (US(i-1,j)+US(i+1,j)+US(i,j-1)+US(i,j+1)-h*h*F(i,j))/4.;

bsp_sync();
bsp_pop_reg(&US(1,s-1));
bsp_pop_reg(&US(1,d+1));

for(i = 1; i < N; i++) {
    U(i,s) = (US(i-1,s)+US(i+1,s)+US(i,s-1)+US(i,s+1)-h*h*F(i,s))/4.;
    U(i,d) = (US(i-1,d)+US(i+1,d)+US(i,d-1)+US(i,d+1)-h*h*F(i,d))/4.;
}

pakl = 0.;
for(j = s; j <= d; j++)
    for(i = 1; i < N; i++) {
        skirt = fabs(U(i,j)-US(i,j));
        if(skirt > pakl)
            pakl = skirt;
        US(i,j) = U(i,j);
    }
it++;
bsp_sync();
bsp_push_reg(pk,nprocs*sizeof(double));
bsp_sync();
bsp_put(0,&pakl,pk,pid*sizeof(double),sizeof(double));
bsp_sync();
bsp_pop_reg(pk);
bsp_sync();
if (pid == 0) {
    paklaida = 0.;
    for(i = 0; i < nprocs; i++) {
        if(pk[i] > paklaida) paklaida = pk[i];
    }
}
bsp_sync();
bsp_push_reg(&paklaida,sizeof(double));
bsp_sync();
if (pid == 0)
    for (i = 1; i < nprocs; i++)
        bsp_put(i,&paklaida,&paklaida,0,sizeof(double));

```

```

        bsp_sync();
        bsp_pop_reg(&paklaida);
        bsp_sync();
    }

    if (pid == 0){
        gettimeofday(&tim, NULL);
        t2=tim.tv_sec+(tim.tv_usec/1000000.0);
        printf("N - %d DRAM Number of threads: %d Iteration Id: %d BSP uztruko - %.6lf\n", N,
nprocs, it, t2-t1);
    }

    free (u);
    free (us);
    free (f);
    free (pk);

    bsp_end();
    return(0);
}

```