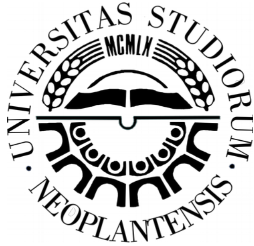


UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
NOVI SAD, SRBIJA

Bogdan Satarić

PARALELNO TRANSPONOVANJE PODATAKA U OKVIRU NUMERIČKOG ALGORITMA ZA REŠAVANJE GROS-PITAEVSKI JEDNAČINE

DOKTORSKA DISERTACIJA

Novi Sad, 2017.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	монографска публикација
Тип записа, **ТЗ**:	текстуални штампани материјал
Врста рада, **ВР**:	докторска дисертација
Аутор, **АУ**:	Богдан Сатарић
Ментор, **МН**:	Др Антун Балаж, научни саветник, Др Мирослав Хајдуковић, ред. проф.
Наслов рада, **НР**:	Паралелно транспоновање података у оквиру нумеричког алгоритма за решавање Грос-Питаевски једначине
Језик публикације, **ЈП**:	српски
Језик извода, **ЈИ**:	српски
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	Аутономна Покрајина Војводина
Година, **ГО**:	2017
Издавач, **ИЗ**:	ауторски репринт
Место и адреса, **МА**:	Факултет техничких наука, Трг Доситеја Обрадовића 6, Нови Сад
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	10 поглавља/151 стране/88 цитата/26 слика/51 листинг
Научна област, **НО**:	Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:	Примењене рачунарске науке
Предметна одредница/Кqучне речи, **ПО**:	Паралелно транспоновање података, Грос-Питаевски једначина, 3Д структуре података, Хибридни алгоритам
УДК	
Чува се, **ЧУ**:	Библиотека Факултета техничких наука
Важна напомена, **ВН**:	
Извод, **ИЗ**:	Ова докторска теза се бави проучавањем и развојем паралелних алгоритама за транспоновање дистрибуираних тродимензионалних структура података, као и имплементацијом ових алгоритама у оквиру Ц/ОпенМП/МПИ програмске парадигме. Развијена имплементација је примењена на решавање нелинеарне парцијалне диференцијеалне једначине Шредингеровог типа (Грос-Питаевски једначина) коришћењем Кренк-Николсон метода, а у оквиру тезе је представљен циклус развоја одговарајућег софтвера, као и резултати тестова валидности и мерења перформанси добијених на рачунарском кластеру.
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	

Чланови комисије, **КО**:

Председник:	Др Жарко Живанов, доцент	
Члан:	Др Срђан Шкрбић, ванредни професор	
Члан:	Др Душан Илић, доцент	Потпис ментора
Члан, ментор:	Др Антун Балаж, научни саветник	
Члан, ментор:	Др Мирослав Хајдуковић, редовни професор	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	monographic publication
Type of record, **TR**:	textual printed material
Contents code, **CC**:	PhD thesis
Author, **AU**:	Bogdan Satarić
Mentor, **MN**:	Dr Antun Balaž, research professor, Dr Miroslav Hajduković, full proffesor
Title, **TI**:	Parallel data transposition in numerical algorithm for solving the Gross-Pitaevski equation
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian / English
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Autonomous Province of Vojvodina
Publication year, **PY**:	2017
Publisher, **PB**:	author's reprint
Publication place, **PP**:	Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	10 chapters/151 pages/88 references/26 pictures/51 listings
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Applied computer science
Subject/Key words, **S/KW**:	Parallel data transposition, Gross-Pitaevskii equation, 3D data structures, Hybrid alghoritm
UC	
Holding data, **HD**:	Library of the Faculty of Technical Sciences
Note, **N**:	
Abstract, **AB**:	This thesis studies and develops parallel algorithms for transposing distributed three-dimensional data structures, and describes their technical implementation in C/OpenMP/MPI programing paradigm. The developed implementation is applied for solving of nonlinear partial differential equation of the Schroedinger type (Gross-Pitaevskii equation) using Crank-Nicolson method. The thesis presents the corresponding software development cycle, as well as results of validity tests and performance measurements obtained on a computer cluster.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	
Defended Board, **DB**: President:	Dr Žarko Živanov, Assistant Professor

Member:	Dr Srđan Škrbić, Associate Professor	
Member:	Dr Dušan Ilić, Assistant Professor	Menthor's signature
Member, Mentor	Dr Antun Balaž, Research Professor	
Member, Mentor:	Dr Miroslav Hajduković, Full professor	

Sadržaj

Zahvalnica....i
Sažetak....ii
1 Uvod....1
1.1 Boze-Ajnštajn kondenzacija....1
1.2 Nelinearna Gros-Pitaevski jednačina....6
1.3 Anizotropna Gros-Pitaevski jednačina u 3D....7
1.4 Gros-Pitaevski jednačina u 1D....8
1.5 Krenk-Nikolson metoda podeljenog koraka za Gros-Pitaevski jednačinu u 1D....9
1.5.1 Gros-Pitaevski jednačina u 1D....10
1.5.2 Propagacija u realnom vremenu....11
1.5.3 Propagacija u imaginarnom vremenu....15
1.5.4 Hemijski potencijal i energija sistema....16
1.5.5 Srednji kvadratni radijus....16
1.6 Krenk-Nikolson metoda podeljenog koraka za Gros-Pitaevski jednačinu u 3D....17
1.6.1 Hemijski potencijal i energija sistema u 3D....17
1.6.2 Srednji kvadratni radijus u 3D....18
1.7 Tema istraživanja....18
1.7.1 Predmet i problem....18
1.7.2 Ciljevi teze....19
2 Problem rezolucije fizičkog sistema....20
2.1 Analiza C/OpenMP programa za implementaciju Krenk-Nikolson algoritma....20
2.1.1 Fajl imagtime3d-th.h....21
2.1.2 Fajl cfg.h....24
2.1.3 Fajl cfg.c....25
2.1.4 Fajl diffint.h....29
2.1.5 Fajl diffint.c....29
2.1.6 Fajl mem.h....31
2.1.7 Fajl mem.c....33
2.1.8 Fajl imagtime3d-th.c....39
2.1.8.1 Funkcija readpar....50
2.1.8.2 Funkcija init....52
2.1.8.3 Funkcija gencoef....53
2.1.8.4 Funkcija calcnorm....54
2.1.8.5 Funkcija calcmuen....56
2.1.8.6 Funkcija calcrms....59
2.1.8.7 Funkcija calcnu....59
2.1.8.8 Funkcija calclux....60
2.1.8.9 Funkcija calcluy....62
2.1.8.10 Funkcija calcluz....63
2.2 C/OpenMP programi i problem rezolucije fizičkog sistema....64
2.3 Predlog hibridnog C/OpenMP/MPI rešenja....66
3 Dekompozicija i transponovanje distribuiranih 2D matrica....70
3.1 Slučaj kada su P i Q međusobno prosti brojevi (NZD = 1)....75
3.2 Slučaj kada P i Q nisu međusobno prosti brojevi (NZD > 1)....77
4 Razvoj algoritma za transponovanje distribuiranih 3D matrica....81
4.1 Problemi manuelne dekompozicije i transponovanja u 3D....81
4.1.1 Projekat MT_v_5.0....81
4.1.1.1 Fajl matrix_t_mpi_v5.c....82

4.1.1.2 Fajl transpose_prime.h....84
4.1.1.3 Fajl transpose_not_prime.h....86
4.1.2 Projekat MT_v_6.0....89
4.1.3 Projekat MT_v_7.0....89
4.1.4 Projekat MT_v_7.5....90
4.1.5 Projekat MT_v_8.0....91
4.2 Projekat GP-SCL-HYB....95
5 Tehnička implementacija hibridnog C/OpenMP/MPI rešenja....99
5.1 Fajl imagtime3d-hyb.h....99
5.2 Fajl mem.h....103
5.3 Fajl mem.c....104
5.4 Fajl misc.h....104
5.5 Fajl misc.c....105
5.6 Fajl imagtime3d-hyb.c....106
5.6.1 Funkcija readpar....118
5.6.2 Funkcija init....118
5.6.3 Funkcija gencoef....120
5.6.4 Funkcija calcnorm....121
5.6.5 Funkcija calcmuen....122
5.6.6 Funkcija calcrms....126
5.6.7 Funkcija calcnu....127
5.6.8 Funkcija calclux_trans....128
5.6.9 Funkcija calcluy....129
5.6.10 Funkcija calcluz....131
5.6.11 Funkcija calcpot....132
5.7 Realizacija testiranja algoritma....133
5.7.1 Pomoćne funkcije za testiranje vrednosti promenljivih....133
6 Merenje efikasnosti i diskusija....137
6.1 Skripte za automatizovanje testiranja....137
6.2 Performanse hibridnog rešenja....139
7 Zaključak....142
7.1 Doprinos rada i mogućnosti primene....142
7.2 Pravci daljeg istraživanja....143
8 Bibliografija....144
9 Indeks slika....148
10 Indeks listinga....150

Zahvalnica

Zahvaljujem se mentorima naučnom savetniku dr Antunu Balažu i redovnom profesoru dr Miroslavu Hajdukoviću na nesebičnoj pomoći i vođstvu tokom izrade doktorske teze. Zahvaljujem se i porodici koja me je podržavala tokom napornog rada.

Istraživanje je deo projekata Ministarstva prosvete, nauke i tehnološkog razvoja Republike Srbije ON171009, ON174027 i ON171017. Numeričke simulacije su pokretane na PARADOX-IV superračunaru u Laboratoriji za primenu računara u nauci, u okviru Nacionalnog centra izuzetnih vrednosti za izučavanje kompleksnih sistema Instituta za fiziku u Beogradu.

Novi Sad, 2017.
Bogdan Satarić

Sažetak

Ova doktorska teza se bavi proučavanjem i razvojem paralelnih algoritama za transponovanje distribuiranih trodimenzionalnih struktura podataka, kao i implementacijom ovih algoritama u okviru C/OpenMP/MPI programske paradigme. Razvijena implementacija je primenjena na rešavanje nelinearne parcijalne diferencijalne jednačine Šredingerovog tipa (Gros-Pitaevski jednačina) korišćenjem Krenk-Nikolson metoda, a u okviru teze je predstavljen ciklus razvoja odgovarajućeg softvera, kao i rezultate testova validnosti i merenja performansi dobijene na računarskom klasteru.

U prvom delu ove doktorske disertacije prikazan je algoritam za rešavanje Gros-Pitaevski jednačine uz pomoć poluimplicitne Krenk-Nikolson metode podeljenog koraka za vremensku propagaciju parcijalnih diferencijalnih jednačina, koja se koristi za numeričke simulacije vremenske evolucije Boze-Ajnštajn kondenzovanih sistema (BAK) na računarima sa deljenom memorijom. Pokazano je da implementacija opisanog algoritma u paradigmi deljene memorije, koja je ranije implementirana koristeći C/OpenMP okruženje, nije dovoljna ukoliko se želi posmatranje ovih fizičkih sistema sa velikim nivoom detalja, odnosno u velikoj rezoluciji, što je često neophodno. Ovo ograničenje proizilazi iz velike količine radne memorije neophodne za posmatranje sistema u velikoj rezoluciji. U računarskim sistemima sa deljenom memorijom radna memorija je ograničena na dostupnu memoriju na jednom računaru i rezolucije fizičkih sistema koje se na taj način mogu proučavati su veoma ograničene, čak i sa najnovijim serverima dostupnim na tržištu.

U nastavku teze opisan je paralelni algoritam za transponovanje dvodimenzionalnih matrica, pogodan za implementaciju u MPI okruženju. Nakon toga je predložen hibridni OpenMP/MPI algoritam koji omogućava proširenje prethodno opisanog algoritma na trodimenzionalne strukture podataka, kao i njegovu integraciju sa Krenk-Nikolson algoritmom. Ovim se implicitno omogućava memorijsko rasterećenje pojedinačanih računara unutar klastera, kao i dovoljno brza i precizna vremenska propagacija fizičkog sistema sa velikim nivoom detalja. Razvijeno je više verzija algoritama distribucije i transponovanja dvodimenzionalnih i trodimenzionlanih matrica, pri čemu je konačan algoritam transponovanja iskorišćen pri implementacji hibridnog C/OpenMP/MPI algoritma za rešavanje Gros-Pitaevski jednačine.

Na kraju teze prikazani su rezultati testiranja skalabilnosti i efikasnosti hibridnog C/OpenMP/MPI rešenja na računarskom klasteru, na osnovu kojih se vidi da se razvijena implementacija dobro skalira i na efikasan način koristi dostupne računarske resurse. Rezultati dobijeni ovim istraživanjem omogućavaju pouzdano i efikasno proučavanje i numeričko simuliranje vremenske evolucije Boze-Ajnštajn kondenzovanih sistema sa velikim nivoom detalja. Time će se zainteresovanim istraživačima omogućiti proširenje dosadašnjih istraživanja i detaljno razmatranje

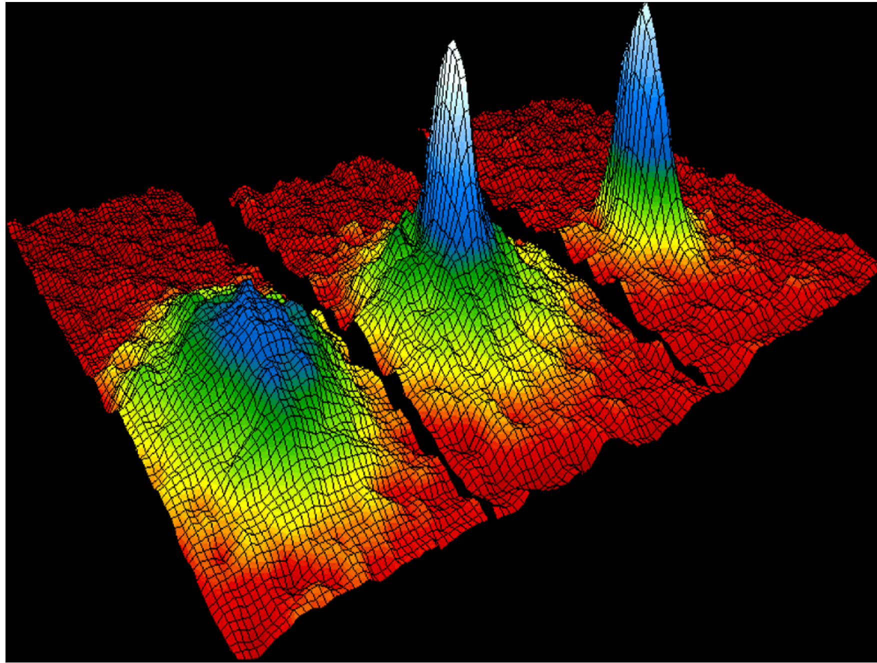
raznih fenomena u sistemima ultrahladnih atoma. Pored toga, razvijeni algoritmi i tehnička implementacija se mogu jednostavno primeniti i na proučavanje drugih sistema, kao što su optički (optička vlakna, nelinearni optički elementi) i kvantno-optički sistemi, a moguće su primene i u ekonomiji i finansijama, odnosno u oblastima koje koriste nelinearne diferencijalne jednačine slične Gros-Pitaevski jednačini.

1 Uvod

1.1 Boze-Ajnštajn kondenzacija

Eksperimentalna istraživanja fenomena Boze-Ajnštajn kondenzacije (BAK) [1-3], koji je teorijski predviđen još 1924. godine, pre skoro jednog veka, tokom zasnivanja kvantne mehanike i kvantne statističke fizike, započeta su 1938. godine (London) na primeru superfluidnosti helijuma ^{4}He. Međutim, superfluidno stanje helijuma ispod kritične temperature $T_c = 2.17\ \mathrm{K}$ nije dobar primer Boze-Ajnštajn kondenzacije jer su interakcije atoma helijuma u tečnoj fazi suviše jake, pa je svega oko 8% atoma u superprovodnom stanju čak i na ultraniskim temperaturama, dok je za kvantitativno proučavanje kondenzacije neophodno da je kondenzovana frakcija značajno veća i da predstavlja glavni proučavani efekat.

Prva eksperimentalna verifikacija BAK ostvarena je 1995. godine u seriji eksperimenata sa parom alkalnog metala rubidijuma (bozonski izotop ^{87}Rb) [4], a kasnije i sa bozonskim izotopima atoma natrijuma (^{23}Na) [5] i litijuma (^{7}Li) [6]. U tim eksperimentima atomi su najpre uhvaćeni u zamku formiranu pomoću šest snopova laserske svetlosti (po dva snopa usmerena u suprotnim smerovima u sve tri dimenzije) i hlađeni tako što se atomi sudaraju sa fotonima čeono, pri čemu se brzine atoma smanjuju (lasersko hlađenje). Potom je oblak ovako ohlađenih atoma obuhvaćen magnetnom zamkom koja ima paraboličnu trodimenzionalnu topologiju. Jačina magnetnog polja se nakon toga smanjuje tako da najbrži (najtopliji) atomi odleću iz zamke, a u njoj ostaju sporiji, što snižava temperaturu. Ovaj metod evaporativnog hlađenja je sličan efektu hlađenja čaja koji isparava iz šolje. Eksperimentalni dokaz o postojanju kondenzacije se dobija na osnovu fotografisanja CCD kamerom atomskog oblaka nakon što se zamka isključi. Atomski oblak tada počinje da se razleće, a fotografisanje se izvodi nakon odgovarajućeg vremena („time of flight“, vreme preleta) koje omogućava da oblak postane dovoljno veliki da CCD kamera može da ga registruje sa dovoljnom rezolucijom. Na ovako dobijenim fotografijama se vidi raspodela gustine atomskog oblaka i u eksperimentima je dobijen vrlo uzak oštar pik u distribuciji brzina atoma u jezgru oblaka kada je temperatura gasa pala ispod kritične vrednosti, što se vidi na slici 1.1.



Slika 1.1: Eksperimentalni rezultati grupe Volfganga Keterlea iz 1995. godine [5] kojima je pokazana pojava Boze-Ajnštajn kondenzacije ispod kritične temperature. (By NIST/JILA/CU-Boulder - NIST Image, Public Domain, https://commons.wikimedia.org/w/index.php?curid=403804)

Na slici 1.1 se sa leve strane vidi raspodela atoma za temperature iznad kritične $(T > T_c)$, dok se centralni deo odnosi na temperaturu blisku, ali nešto nižu od kritične, $T_c = 2\,\mu\text{K}$, gde još postoji dosta nekondenzovanih atoma, pa je zbog toga raspodela i dalje široka, iako je značajno uža od čiste termalne (Maksvelove) raspodele sa leve strane. Raspodela sa desne strane pokazuje raspodelu za skoro potpuno kondenzovani sistem, za koji je raspodela veoma bliska delta funkciji, odnosno praktično svi atomi imaju brzinu blisku nultoj.

BAK se sastoji od makroskopskog broja identičnih bozona (čestica sa celobrojnim ukupnim spinom) koji se nalaze u osnovnom energetskom stanju sistema. To je fazni prelaz koji ne zavisi od specifičnih interakcija među konstitutivnim atomima, nego od temperature na kojoj se sistem nalazi. BAK je posledica identičnosti atoma koji čine sistem, kao i njihovih izraženih talasnih svojstava na niskim temperaturama, u kontekstu de Broljeve teorije talasno-čestičnog dualizma. S obzirom da alkalni metali obično imaju više izotopa, njihov bozonski karakter zavisi od ukupnog broja protona, neutrona i elektrona koji čine jedan atom: ukoliko je taj broj paran, onda se atom ponaša kao kompozitni bozon, a ukoliko je ukupan broj gradivnih čestica neparan, dobijamo kompozitni fermion. Pošto je u svakom neutralnom atomu broj protona i elektrona jednak, njihov zbir je uvek paran, pa sve zavisi od broja neutrona, odnosno bozonski izotopi atoma su oni koji sadrže paran broj neutrona. Na primer, izotop litijuma $^{7}_{3}\text{Li}$ se sastoji od 3 protona i 3 neutrona (atomski broj Z=3), a iz masenog broja A=7, koji je jednak zbiru protona i neutrona u jezgru, zaključujemo da u atomu imamo 4 neutrona, pa je ovaj izotop bozonski. Sa druge strane, $^{6}_{3}\text{Li}$ je fermionski izotop, jer u jezgru ima 3 neutrona.

Alkalni metali su odabrani u prvim eksperimentima, a i danas se veoma široko koriste za proučavanje ultrahladnih atomskih sistema, jer su veoma pogodni za primenu metoda laserskog hlađenja. Razlog za to je povoljna struktura njihovog elektronskog omotača, odnosno to što njihovi brojni optički prelazi mogu biti pobuđeni čak i pomoću raspoloživih komercijalnih poluprovodničkih lasera. Kombinujući lasersko i evaportivno hlađenje alkalnih atoma, istraživači su konačno uspeli da dosegnu temperature i gustine gasa neophodne za pojavu BAK. U ovakvim sistemima temperatura se određuje merenjem raspodele brzina atoma nakon isključivanja zamke i odgovarajućeg vremena preleta.

Intuitivna predstava BAK sledi iz elementarnog razmatranja gasa identičnih neinteragujućih Boze čestica mase m pri temperaturi T . De Broljeva talasna dužina jednog atoma je data izrazom

$$\lambda_{dB}=\frac{h}{p} , \tag{1.1}$$

odakle je impuls atoma p dat sa

$$p=\frac{h}{\lambda_{dB}} . \tag{1.2}$$

Odgovarajuća kinetička energija atoma iznosi

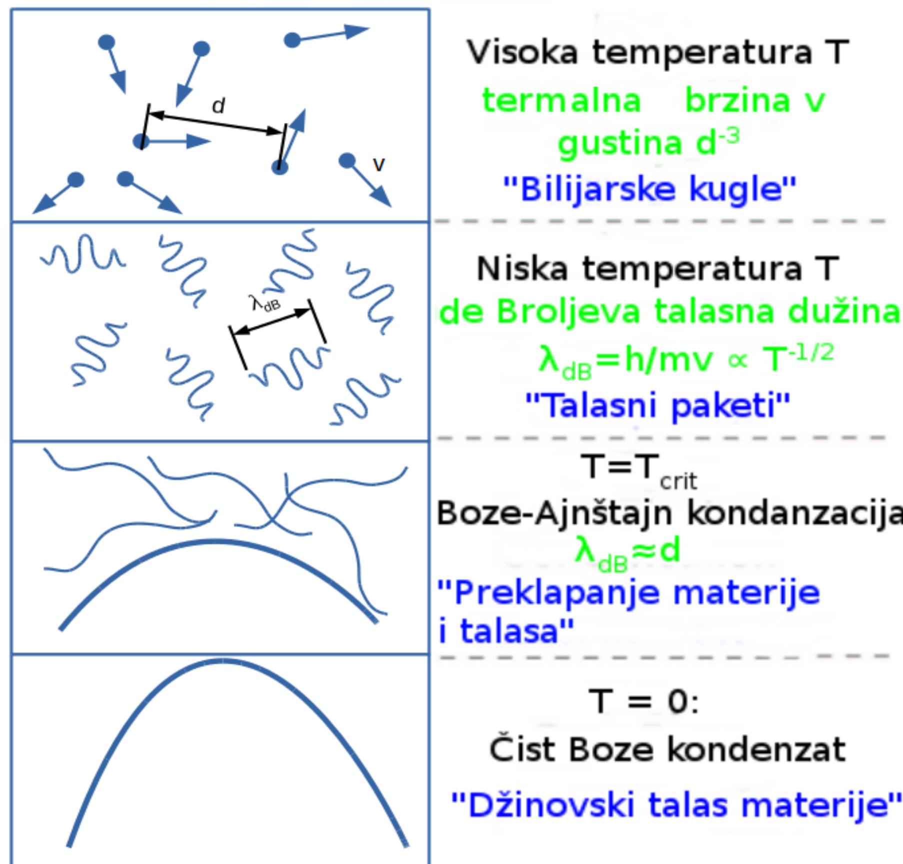
$$E_k=\frac{p^2}{2m}=\frac{h^2}{2m\lambda_{dB}^2} , \tag{1.3}$$

pa je veza između de Broljeve talasne dužine i temperature data sa

$$\lambda_{dB}=\frac{h}{\sqrt{3mk_BT}} , \tag{1.4}$$

gde su h i k_B Plankova, odnosno Bolcmanova konstanta.

Na osnovu ovoga vidimo da termička de Broljeva talasna dužina atoma raste sa snižavanjem temperature. U smislu talasno-čestičnog dualizma možemo da kažemo da je talas koji odgovara pojedinačnom atomu reda veličine λ_{dB} , pa snižavanjem temperature dolazi do preklapanja talasa različitih atoma. Na taj način atomi počinju da utiču jedni na druge i počinju da se ponašaju koherentno, odnosno mogu da se opišu jednom zajedničkom talasnom funkcijom, koju zovemo makroskopska talasna funkcija BAK. To je ilustrovano na slici 1.2, koja je inspirisana predavanjem Volfganga Keterlea o BAK prilikom dodele Nobelove nagrade 2001. godine.



Slika 1.2: Shematski prikaz Boze-Ajnštajn kondenzacije prilikom smanjenja temperature sistema, inspirisan Nobelovom lekcijom Volfganga Keterlea 2001. godine.

Na osnovu relacije (1.4) moguća je jednostavna procena kritične temperature BAK faznog prelaza. Ako je u zapremini V zamke smešteno N atoma, koncentracija gasa je

$$n=\frac{N}{V}\,. \tag{1.5}$$

Uslov za kondenzaciju je da zapremina po atomu bude reda veličine λ_{dB}^3, pošto tada počinje preklapanje individualnih talasnih paketa atoma, odnosno formira se makroskopska talasna funkcija celog sistema. Dakle, uslov za kondenzaciju se može izraziti kao:

$$\frac{V}{N}=\frac{1}{n}\sim\lambda_{dB}^3\,. \tag{1.6}$$

Korišćenjem izraza (1.4) iz uslova (1.6) dobijamo da je kritična temperatura T_c data relacijom:

$$T_c\approx\frac{1}{3}\frac{h^2}{m\,k_B}n^{2/3} \tag{1.7}$$

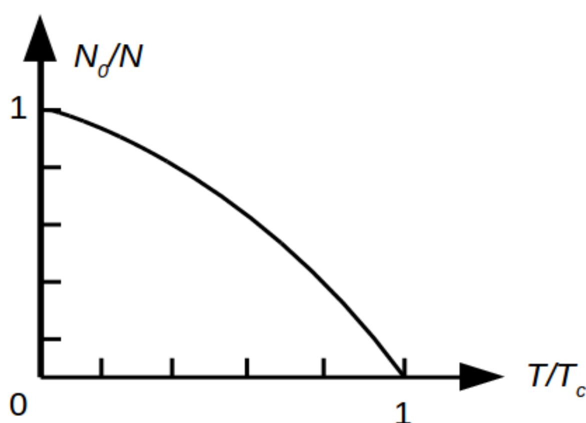
Uprkos ovom veoma pojednostavljenom postupku izvođenja, ova formula je iznenađujuće bliska izrazu dobijenom rigoroznom primenom kvantne mehanike, sa razlikom koja se pojavljuje samo u numeričkom faktoru:

$$T_c = 0.0839 \frac{h^2}{m k_B} n^{2/3} . \quad (1.8)$$

Tipične koncentracije za paru rubidijuma su $n \sim 10^{20} m^{-3}$, pa se na osnovu jednačine (1.8) dobija kritična temperatura od 400 nK do 600 nK. Kondenzovana frakcija atoma N_0 / N na temperaturi $T < T_c$, gde je N_0 broj atoma u kondenzatu a N je ukupan broj atoma, data je izrazom

$$\frac{N_0}{N} = 1 - \left(\frac{T}{T_c} \right)^3 , \quad (1.9)$$

i prikazana na slici 1.3.



Slika 1.3: Zavisnost kondenzovane frakcije atoma od temperature. BAK fazni prelaz se događa na temperaturi T_c , a ispod ove temperature je osnovno energetsko stanje sistema makroskopski naseljeno.

Pošto u bozonskim gasovima u kojima je ostvarena BAK postoji jasna hijerarhija skala dužina i energija, a interakcije su slabe, moguć je njihov teorijski opis polazeći od prvih principa. Uslov da su interakcije slabe je zadovoljen kada je prosečno rastojanje između atoma, dato sa $n^{-1/3}$, značajno veće od efektivnog dometa interatomskih sila. Taj domet je za razređene atomske pare u kojima se realizuje BAK okarakterisan dužinom atomskog rasejanja a_s u simetričnom (s) kanalu, koja tipično iznosi od 1 nm do 5 nm za alkalne atome. Dakle, uslov da su interakcije dovoljno slabe se može napisati kao

$$n a_s^3 \ll 1 . \quad (1.10)$$

Ovo podrazumeva da su binarni (dvoatomski) sudari mnogo češći nego trostruki sudari i može se primeniti teorija slabo interagujućeg Boze gasa. Energije međuatomskih interakcija su saglasno teoriji srednjeg polja („mean-field theory“) date izrazom:

$$U_{inter} = \frac{4 \pi \hbar^2 a_s n}{m} = g n , \quad (1.11)$$

gde se jačina interakcije g često zove i koeficijent nelinearnosti ili kraće nelinearnost. Kada se ovde zamene tipične vrednosti parametara za rubidijum, za U_{inter} se dobija vrednost reda veličine nanoelektronvolta.

Da bi kondenzat bio stabilan, neophodno je da su međuatomske interakcije odbojne, odnosno da je vrednost a pozitivna. U suprotnom je moguće postići, pri dovoljno niskim koncentracijama atoma, metastabilno kondenzovano stanje, a prilikom povećanja broja atoma dolazi do kolapsa kondenzata. Na primer, za ^{7}Li koji ima negativnu dužinu rasejanja, kritičan broj atoma pri kojem dolazi do kolapsa u tipičnim eksperimentalnim uslovima je veoma nizak, oko 2000. Ako to uporedimo sa brojem kondenzovanih atoma u sistemima sa odbojnom interakcijom, koji iznosi od nekoliko stotina hiljada do nekoliko miliona atoma, vidimo da je istraživanje BAK sa privlačnom interakcijom veoma ograničeno.

Razmatranje idealnog gasa bozona je samo nulta aproksimacija za opis BAK, a kompletna teorija mora da uzme u obzir i interakcije među atomima. Pošto su interakcije slabe, od posebnog interesa je teorija srednjeg polja, koja je razvijena posebno za ovakve situacije. Kao što smo već ranije objasnili, u slučaju slabih interakcija BAK-u se može pripisati makroskopska talasna funkcija, koja treba da zadovolji odgovarajuću Šredingerovu jednačinu, koja će imati i efektivne nelinearne članove zbog interakcije. Ova jednačina, koja će biti predstavljena u sledećoj sekciji, se obično zove Gros-Pitaevski jednačina [1] i daje adekvatan opis u izučavanju većine svojstava BAK u realističnim eksperimentima. Numeričko rešavanje ove jednačine pomoću paralelnih algoritama je centralni objekat ove teze.

1.2 Nelinearna Gros-Pitaevski jednačina

Vremenski zavisna Gros-Pitaevski (GP) jednačina je parcijalna diferencijalna jednačina sa prostornim i vremenskim izvodima nepoznate funkcije koja uključuje izvode prvog reda po vremenu i drugog reda po prostornim koordinatama, a sadrži i linearni član koji odgovara potencijalnoj zamci, kao i nelinearni član koji je posledica interakcija u sistemu. GP jednačina ima strukturu nelinearne Šredingerove jednačine, obično u harmonijskoj zamci, odnosno sa harmonijskim potencijalom. Jedan od standardnih načina za rešavanje vremenski zavisne GP jednačine jeste njena diskretizacija u prostoru i vremenu, nakon čega se vrši propagacija u vremenu ovako diskretizovane jednačine. Znajući rešenje jednačine u određenom vremenskom trenutku, odnosno polazeći od zadatih početnih uslova, navedena procedura nalazi novo rešenje nakon kratkog vremenskog koraka, rešavanjem diskretizovane jednačine, što se iterira kako bi se izračunala dinamika sistema za zadato vreme. Najčešće korišćena diskretizaciona šema za GP jednačinu je polu-implicitna ("semi-implicit") Krenk-Nikolson diskretizaciona šema [7-9], koja će biti korišćena u okviru našeg numeričkog algoritma.

Na temperaturi bliskoj apsolutnoj nuli, talasna funkcija BAK $\psi \equiv \psi(\boldsymbol{r};\tau)$ u tački $\boldsymbol{r}$ i u trenutku τ se u teoriji srednjeg polja opisuje GP jednačinom

$$i\hbar\frac{\partial\psi(\mathbf{r};\tau)}{\partial\tau}=\left[-\frac{\hbar^2\nabla^2}{2m}+V(\mathbf{r})+gN|\psi(\mathbf{r};\tau)|^2\right]\psi(\mathbf{r};\tau)\,, \tag{1.12}$$

gde je i imaginarna jedinica, $\hbar=h/(2\pi)$ je redukovana Plankova konstanta, a $V(\mathbf{r})$ je potencijal zamke. Kao i ranije, N je ukupan broj atoma u kondenzatu (ovde pretpostavljamo da su svi atomi kondenzovani), a nelinearnost $g=4\pi\hbar^2 a_s/m$ odgovara efektivnoj jačini međuatomskih interakcija, gde je a dužina rasejanja u simetričnom, s-kanalu. Normalizacioni uslov talasne funkcije je $\int dr\,|\psi(r;\tau)|^2=1$, što odgovara uslovu da je ukupan broj atoma jednak N. Tipične vrednosti navedenih fizičkih parametara za eksperimente sa ^{87}Rb iznose $m=1.44\cdot 10^{-25}\,\mathrm{kg}$, $a_s=5.1\,\mathrm{nm}$, dok je broj atoma u opsegu od nekoliko stotina do nekoliko miliona.

1.3 Anizotropna Gros-Pitaevski jednačina u 3D

Potencijal anizotropne trodimenzionalne zamke je izražen na sledeći način

$$V(\mathbf{r})=\frac{1}{2}m\omega^2(\nu^2\tilde{x}^2+\kappa^2\tilde{y}^2+\lambda^2\tilde{z}^2)\,, \tag{1.13}$$

gde su $\omega_x\equiv\omega\nu,\omega_y\equiv\omega\kappa$ i $\omega_z\equiv\omega\lambda$ ugaone frekvencije potencijalne zamke u x, y i z pravcu, respektivno, a $\mathbf{r}\equiv(\tilde{x},\tilde{y},\tilde{z})$ je vektor položaja. Koristeći za dužinsku skalu dužinu harmonijskog oscilatora $l=\sqrt{\hbar/(m\omega)}$, a za vremensku skalu inverznu frekvenciju $1/\omega$, možemo da uvedemo nove, bezdimenzione koordinate $x=\sqrt{2}\tilde{x}/l$, $y=\sqrt{2}\tilde{y}/l$, $z=\sqrt{2}\tilde{z}/l$, $t=\omega\tau$, dok talasna funkcija postaje $\phi(x,y,z;t)=\sqrt{l^3/(2\sqrt{2})}\,\Psi(\mathbf{r};\tau)$. Nakon ovoga, jednačina (1.12) postaje

$$\begin{aligned} &i\frac{\partial}{\partial t}\phi(x,y,z;t)=\\ &\left[-\frac{\partial^2}{\partial x^2}-\frac{\partial^2}{\partial y^2}-\frac{\partial^2}{\partial z^2}+\frac{1}{4}(\nu^2x^2+\kappa^2y^2+\lambda^2z^2)+\mathcal{N}|\phi(x,y,z)|^2\right]\phi(x,y,z;t)\,, \end{aligned} \tag{1.14}$$

gde je nova, bezdimenziona konstanta nelinearnosti $\mathcal{N}=8\sqrt{2}\pi a_s N/l$, a uslov normalizacije se funkcionalno ne menja i glasi:

$$\int_{-\infty}^{\infty}dx\int_{-\infty}^{\infty}dy\int_{-\infty}^{\infty}dz|\phi(x,y,z;t)|^2=1\ . \tag{1.15}$$

Slično ovome, koristeći reskaliranje $x=\tilde{x}/l$, $y=\tilde{y}/l$, $z=\tilde{z}/l$, $t=\tau\omega$, za talasnu funkciju dobijamo $\phi(x,y,z;t)=\sqrt{l^3}\,\Psi(\mathbf{r},\tau)$, a GP jednačina postaje:

$$i\frac{\partial}{\partial t}\phi(x,y,z;t)= \left[-\frac{1}{2}\frac{\partial^2}{\partial x^2}-\frac{1}{2}\frac{\partial^2}{\partial y^2}-\frac{1}{2}\frac{\partial^2}{\partial z^2}+\frac{1}{2}(\nu^2x^2+\kappa^2y^2+\lambda^2z^2)+\mathcal{N}|\phi(x,y,z;t)|^2\right]\phi(x,y,z;t)\,, \tag{1.16}$$

gde je $\mathcal{N}=4\pi a_s N/l$ i normalizacija (1.15) i dalje važi. Dodatno skalirajući vreme $t\rightarrow 2t$ gornju jednačinu možemo da dovedemo u nešto simetričniji oblik:

$$i\frac{\partial}{\partial t}\phi(x,y,z;t)= \left[-\frac{\partial^2}{\partial x^2}-\frac{\partial^2}{\partial y^2}-\frac{\partial^2}{\partial z^2}+\nu^2x^2+\kappa^2y^2+\lambda^2z^2+\mathcal{N}|\phi(x,y,z)|^2\right]\phi(x,y,z;t)\,, \tag{1.17}$$

gde je $\mathcal{N}=8\pi a_s N/l$.

U svim navedenim slučajevima granični uslovi za rešenje GP jednačine ostaju isti [7]:

$$\lim_{x\rightarrow\pm\infty}\phi(x,z,y;t)=0,\ \lim_{y\rightarrow\pm\infty}\phi(x,z,y;t)=0,\ \lim_{z\rightarrow\pm\infty}\phi(x,z,y;t)=0\,. \tag{1.18}$$

1.4 Gros-Pitaevski jednačina u 1D

Eksperimentalno se često susreće slučaj izdužene zamke, koju obično zovemo zamka u obliku cigarete („cigar-shaped trap“) i kod koje je longitudinalna frekvencija (ω_z) znatno manja od poprečnih $(\omega_z \ll \omega_x,\omega_y)$. Pod ovim uslovima, GP jednačina (1.14) može da se svede na efektivnu kvazijednodimenzionalnu formu. Ovo se postiže pretpostavkom da sistem nalazi u osnovnom stanju kvantnog harmonijskog oscilatora u poprečnim pravcima, s obzirom da su odgovarajuće frekvencije velike, pa se u tim pravcima interakcije mogu zanemariti. U ovom slučaju, talasna funkcija jednačine (1.14) može biti napisana kao:

$$\phi(x,y,z;t)=\tilde{\phi}(z;t)\ \Phi_0(x;\nu)\ \Phi_0(y;\kappa)\ e^{-i(\nu+\kappa)t/2}\,, \tag{1.19}$$

gde je $\Phi_0(r;\omega)=[\omega/(2\pi)]^{1/4}e^{-\omega r^2/4}$ talasna funkcija osnovnog stanja jednodimenzionalnog harmonijskog oscilatora sa (bezdimenzionom) frekvencijom ω, a odgovarajuću (bezdimenzionu) koordinatu smo označili sa *r*. Pretpostavljajući da rešenje jednačine (1.14) ima gornji oblik i množeći je sa $\Phi_0(x;\nu)\ \Phi_0(y;\kappa)$, nakon integracije po koordinatama *x* i *y*, uz brisanje oznake tilda iznad simbola talasne funkcije, dobijamo efektivnu jednačinu i stavljajući da je $\nu=1$ dobijamo:

$$i\frac{\partial}{\partial t}\phi(z;t)=\left[-\frac{\partial^2}{\partial z^2}+\frac{z^2}{4}+\mathcal{N}|\phi(z;t)|^2\right]\phi(z;t)\,, \tag{1.20}$$

gde je $\mathcal{N}=2a_s N\sqrt{2\nu\kappa}/l\,,$ pri čemu smo uzeli vrednost $\lambda=1$ bez smanjenja opštosti. Uslov normalizacije sada glasi:

$$\int_{-\infty}^{\infty} dz|\phi(z;t)|^2=1\ . \tag{1.21}$$

Sličnom procedurom se može dobiti jednodimenzionalna efektivna GP jednačina koja predstavlja analogon jednačine (1.16), koja ima oblik

$$i\frac{\partial}{\partial t}\phi(z;t)=\left[-\frac{1}{2}\frac{\partial^2}{\partial z^2}+\frac{z^2}{2}+\mathcal{N}|\phi(z;t)|^2\right]\phi(z;t)\,, \tag{1.22}$$

pri čemu je sada $\mathcal{N}=2a_s N\sqrt{\nu\kappa}/l\,,$ uz isti normalizacioni uslov (1.21). Skalirajući vreme u gornjoj jednačini prema $t\rightarrow 2t\,,$ dobijamo analogon jednačine (1.20) u obliku

$$i\frac{\partial}{\partial t}\phi(z;t)=\left[-\frac{\partial^2}{\partial z^2}+z^2+\mathcal{N}|\phi(z;t)|^2\right]\phi(z;t)\,, \tag{1.23}$$

gde je $\mathcal{N}=4a_s N\sqrt{\nu\kappa}/l$ uz normalizaciju (1.21). U ovom slučaju prilikom numeričkog rešavanja svih oblika jednodimenzionalne GP jednačine važi granične uslove $\lim_{z\rightarrow\pm\infty}\phi(z;t)=0$.

1.5 Krenk-Nikolson metoda podeljenog koraka za Gros-Pitaevski jednačinu u 1D

U najprostijem slučaju u 1D, kada imamo jednu prostornu i jednu vremensku promenljivu, Krenk-Nikolson algoritam [7-9] za pronalaženje rešenja se izvršava iterativno, u seriji od po dva polukoraka u svakoj iteraciji. U okviru jedne iteracije se u prvom polukoraku, koristeći talasnu funkciju iz prethodne iteracije kao polaznu tačku, nalazi međurešenje nakon malog vremenskog intervala Δ uzimajući u obzir samo deo Hamiltonijana koji sadrži multiplikativne članove, odnosno harmonijski potencijal i nelinearni član. U drugom polukoraku se rešava jednačina koja uključuje samo članove sa prostornim izvodima, i tako se dobija konačno rešenje nakon vremena Δ u datoj iteraciji, koje je polazni korak za sledeći. Ovo je detaljnije objašnjeno u sekciji 1.5.2. U slučaju dve ili tri prostorne promenljive, prostorni izvodi se obrađuju u dva ili tri međukoraka, posebno za

svaku prostornu dimenziju, što je predstavljeno u sekciji 1.6. S obzirom da se vremenska evolucija sistema odvija u nekoliko međukoraka (dva ili više, u zavisnosti od broja dimenzija), metoda se stoga zove metoda podeljenog koraka („split-step“) i koristi se za propagaciju u vremenu nelinearnih diferencijalnih jednačina GP tipa. Metoda je podjednako primenljiva za nalaženje stacionarnih, osnovnih i pobuđenih stanja, kao i za proučavanje dinamike sistema počev od zadatog početnog stanja. Važno pozitivno svojstvo poluimplicitne Krenk-Nikolson šeme [7-9] koju ćemo koristiti ovde, jeste da je bezuslovno stabilna i da čuva normalizaciju rešenja kod propagacije u realnom vremenu. Poluimplicitnost šeme označava da se diskretizacija po vremenu u međukoracima koji se odnose na prostorne izvode izražava kao linearna kombinacija odgovarajućih diskretizovanih izraza za izvode u početnom i krajnjem trenutku date iteracije, što uvodi stabilnost u ovaj algoritam.

Dok se vremenska dinamika sistema proučava direktnom primenom ovog metoda na GP jednačinu iz prethodnih sekcija, za nalaženje stacionarnih stanja se koristi modifikovana verzija ovog algoritma, kada se umesto propagacije u realnom vremenu prelazi na propagaciju u imaginarnom vremenu. U tom slučaju znamo da su odgovarajuće talasne funkcije realne, a GP jednačine koje rešavamo se menjaju tako da sa leve strane član koji sadrži izvod po vremenu postaje

$$i\frac{\partial}{\partial t}\phi(z;t) \quad \rightarrow \quad -\frac{\partial}{\partial t}\phi(z;t)\,. \tag{1.24}$$

Ovde odmah vidimo da GP jednačina u imaginarnom vremenu sadrži samo realne članove, pa je zbog toga i njeno rešenje dato pomoću realne funkcije u odgovarajućem broju prostornih dimenzija. Međutim, uvođenje imaginarnog vremena rezultuje eksponencijalnim opadanjem norme talasne funkcije, odnosno propagacija u imaginarnom vremenu ne čuva njenu normu, za razliku od propagacije u realnom vremenu [10]. Zbog toga u svakoj iteraciji moramo da renormiramo talasnu funkciju tako zadovoljava normalizacioni uslov (1.21). Sa druge strane, ovakva propagacija vodi ka osnovnom stanju sistema, što se može pokazati matematički [10]. Dakle, iako je u pitanju samo formalna, matematička zamena realnog vremena imaginarnim vremenom, ona nam omogućava da nađemo osnovno stanje sistema polazeći od proizvoljnog početnog stanja (pod uslovom da nije ortogonalno na osnovno stanje).

Metoda podeljenog koraka sa propagacijom u imaginarnom vremenu nam omogućava da izračunamo osnovno stanje na relativno jednostavan i računarski ne previše zahtevan način, pogotovo u terminima procesorskog vremena. Kada proračunavamo propagaciju sistema u realnom vremenu, tada moramo da koristimo kompleksne promenljive, pa su odgovarajuće simulacije računarski zahtevnije, kako u smislu procesorskog vremena, tako i u smislu memorijskih zahteva.

1.5.1 Gros-Pitaevski jednačina u 1D

Objasnićemo kako se primenjuje Krenk-Nikolson metoda [7-9] na GP jednačinu na primeru jednodimenzionalnog slučaja. Nelinearna GP jednačina (1.20) u ovom slučaju može da se predstavi u sledećoj formi:

$$i\frac{\partial}{\partial t}\phi(z;t)=\left[-\frac{\partial^2}{\partial z^2}+\frac{z^2}{4}+\mathcal{N}|\phi(z;t)|^2\right]\phi(z;t)\equiv H\phi(z;t)\,, \qquad (1.25)$$

gde Hamiltonijan *H* sadrži član sa prostornim izvodom, linearni i nelinearni član. Data jednačina se rešava uz pomoć gore opisane vremenske iterativne šeme, koristeći poluimplicitni Krenk-Nikolson metod podeljenog koraka [7-9]. Zadato početno stanje se propagira u vremenu kroz male vremenske korake, dok se ne dostigne zadato vreme propagacije *T,* i na taj način se dobija tražena talasna funkcija $\phi(z;t)$. Nakon diskretizacije GP jednačine u prostoru i vremenu koristeći šemu konačnih razlika dobija se sistem algebarskih jednačina koje mogu biti rešene uz pomoć vremenskog iteriranja, koristeći početno stanje, koje mora da bude konzistentno sa gore definisanim graničnim uslovima. Sada ćemo prvo opisati algoritam za propagaciju u realnom vremenu, a onda i za propagaciju u imaginarnom vremenu.

1.5.2 Propagacija u realnom vremenu

Vremensko iteriranje se vrši deljenjem Hamiltonijana (1.25) u dva dela $H=H_1+H_2$, gde je

$$H_1=\left[\frac{z^2}{4}+\mathcal{N}|\phi(z;t)|^2\right] \qquad (1.26)$$

deo koji ne sadrži izvode, a

$$H_2=-\frac{\partial^2}{\partial z^2} \qquad (1.27)$$

je operator drugog izvoda po prostornoj koordinati *z*. Suštinski, podelili smo jednačinu (1.25) na sledeće dve jednačine:

$$i\frac{\partial}{\partial t}\phi(z;t)=\left[\frac{z^2}{4}+\mathcal{N}|\phi(z;t)|^2\right]\phi(z;t)\equiv H_1\phi(z;t)\,, \qquad (1.28)$$

$$i\frac{\partial}{\partial t}\phi(z;t)=-\frac{\partial^2}{\partial z^2}\phi(z;t)\equiv H_2\phi(z;t)\,. \qquad (1.29)$$

Prvo se rešava jednačina (1.28) sa zadatim početnim uslovom $\phi(z;t_0)$ u trenutku $t=t_0$ kako bi se dobilo prvo međurešenje u trenutku $t=t_0+\Delta$, gde je Δ mali vremenski korak. Nakon toga se ovo međurešenje koristi kao početni uslov za rešavanje jednačine (1.29), nakon čega dobijamo

konačno rešenje na kraju date iteracije, odnosno u trenutku $t = t_0 + \Delta$, što označavamo sa $\phi(z; t_0 + \Delta)$. Ovaj postupak se ponavlja n puta kako bi se dobilo konačno rešenje u vremenu $t_{final} = t_0 + n\Delta = t_0 + T$. Naravno, na ovaj način smo izračunali kako izgleda talasna funkcija $\phi(z; t)$ u svakom trenutku između t_0 i t_{final}.

Vremenska promenljiva je diskretizovana kao $t_n = t_0 + n\Delta$, gde je Δ vremenski korak. Propagacija rešenja $\phi^n \equiv \phi(z; t_n)$ iz trenutka t_n u trenutak t_{n+1} se odvija kroz dva polukoraka, pri čemu ćemo dobijeno rešenje nakon prvog polukoraka, odnosno propagacije pomoću Hamiltonijana H_1, označiti sa $\phi^{n+1/2}$. Kako nema izvoda u Hamiltonijanu H_1, ova propagacija se može opisati sledećom jednačinom,

$$\phi^{n+1/2} = \mathrm{O}_{nd}(H_1)\phi^n \equiv e^{-i\Delta H_1}\phi^n, \quad (1.30)$$

gde $\mathrm{O}_{nd}(H_1)$ predstavlja operaciju vremenske evolucije pomoću H_1, a sufiks „nd” označava da je u pitanje deo Hamiltonijana bez izvoda („non-derivative“). Nakon toga numerički se izvodi vremenska propagacija koja odgovara operatoru H_2, uz pomoć poluimplicitne Krenk-Nikolson šeme [7-9], u kojoj se diskretizacija odgovarajuće jednačine (1.29) izražava na sledeći način:

$$i\frac{\phi^{n+1} - \phi^{n+1/2}}{\Delta} = \frac{1}{2}H_2(\phi^{n+1} + \phi^{n+1/2}) \quad (1.31)$$

Formalno rešenje jednačine (1.31) je

$$\phi^{n+1} = \mathrm{O}_{CN}(H_2)\phi^{n+1/2} \equiv \frac{1 - i\Delta H_2/2}{1 + i\Delta H_2/2}\phi^{n+1/2}, \quad (1.32)$$

Pa zajedno sa jednačinom (1.30) dobijamo kompletnu propagaciju tokom jedne iteracije kao

$$\phi^{n+1} = O_{CN}(H_2)O_{nd}(H_1)\phi^n, \quad (1.33)$$

gde $O_{CN}(H_2)$ predstavlja operaciju vremenske evolucije pomoću H_2 u skladu sa jednačinom (1.32), a sufiks „CN” govori da je u pitanju Krenk-Nikolson algoritam. Operator O_{CN} se koristi za propagaciju međurešenja $\phi^{n+1/2}$ za vremenski korak Δ radi dobijanja rešenja ϕ^{n+1} u sledećem vremenskom koraku $t_{n+1} = (n+1)\Delta$.

Prednost prethodno opisane metode podeljenog koraka sa malim vremenskim korakom Δ se uočava kroz tri osobine [8, 9]. Prvo, sve iteracije čuvaju normalizaciju talasne funkcije. Drugo, greška nastala zbog deljenja Hamiltonijana je proporcionalna sa Δ^2 i može biti zanemarena u odnosu na linearnu grešku diskretizacije izvoda po vremenu. Konačno, s obzirom da se veliki deo Hamiltonijana (linearni i nelinearni član) tretira egzaktno, bez mešanja sa delikatnom Krenk-Nikolson propagacijom, metod može da se primeni na proizvoljno velik nelinearni član, a da i tako daje pouzdane i tačne rezultate, odnosno nije ograničen na slučajeve sa malom nelinearnošću.

Sada ćemo dati detaljan opis poluimplicitnog Krenk-Nikolson algoritma. GP jednačina (1.31) se diskretizuje po prostornoj koordinati na N_z tačaka na sledeći način:

$$i\frac{\left(\phi_i^{n+1}-\phi_i^{n+1/2}\right)}{\Delta}=-\frac{1}{2h^2}\left[\left(\phi_{i+1}^{n+1}-2\phi_i^{n+1}+\phi_{i-1}^{n+1}\right)+\left(\phi_{i+1}^{n+1/2}-2\phi_i^{n+1/2}+\phi_{i-1}^{n+1/2}\right)\right], \tag{1.34}$$

gde je $z_i=-N_z h/2+ih,\ i=0,1,2,\ldots,N_z$, ϕ_j^m je diskretizovana talasna funkcija u datom trenutku m (m=n, n+1/2, n+1) u prostornoj tački z_j (j=i-1, i, i+1), pri čemu je h prostorni korak diskretizacije. Ovakav izbor diskretizovanih vrednosti za z, čak i u slučaju parnih vrednosti N_z, ima prednost zbog jednakog broja prostornih tačaka sa obe strane koordinatnog početka (z = 0). Jednačina (1.34) predstavlja eksplicitnu formu diskretizovane jednačine (1.31). Šema je konstruisana aproksimacijom $\partial/\partial t$ uz pomoć formule sa dve tačke, koje predstavljaju sadašnji trenutak $(m=n+1/2)$ i naredni trenutak $(m=n+1)$. Prostorni izvod $\partial^2/\partial x^2$ aproksimira se formulom sa tri tačke, uz linearnu kombinaciju vrednosti u sadašnjem i narednom trenutku. Ova procedura rezultuje u nizu tridijagonalnih jednačina po nepoznatim veličinama ϕ_{i+1}^{n+1}, ϕ_i^{n+1} i ϕ_{i-1}^{n+1} u trenutku t_{n+1}, koje se rešavaju koristeći odgovarajuće granične uslove.

Krenk-Nikolson šema (1.34) poseduje određena svojstva, koja treba svakako naglasiti [8, 9]. Greška u šemi je drugog reda po prostornoj koordinati, tako da je za malo Δ i h greška zanemarljiva. Šema je takođe bezuslovno stabilna [9]. Granični uslov u beskonačnosti se čuva za male vrednosti Δ/h^2 [9].

Tridijagonalne jednačine koje proizilaze iz jednačine (1.34) mogu se eksplicitno zapisati kao [8]

$$A_i^-\phi_{i-1}^{n+1}+A_i^0\phi_i^{n+1}+A_i^+\phi_{i+1}^{n+1}=b_i, \tag{1.35}$$

gde je koeficijent b_i dat izrazom

$$b_i=\frac{i\Delta}{2h^2}\left(\phi_{i+1}^{n+1/2}-2\phi_i^{n+1/2}+\phi_{i-1}^{n+1/2}\right)+\phi_i^{n+1/2}, \tag{1.36}$$

dok je $A_i^-=A_i^+=-i\Delta/(2h^2)$, $A_i^0=1+i\Delta/h^2$. Svi koeficijenti b_i odnose se na prvi polukorak, koji je već sproveden, pa su zato poznati. Jedine nepoznate u jednačini (1.35) su veličine $\phi_{i\pm1}^{n+1}$ i ϕ_i^{n+1} u krajnjem trenutku aktuelne iteracije t_{n+1}. Kako bismo rešili jednačinu (1.35), koristićemo rekurzivne relacije, koje se dobijaju ako izrazimo nepoznate veličine u različitim prostornim tačkama pomoću linearne relacije

$$\phi_{i+1}^{n+1}=\alpha_i\phi_i^{n+1}+\beta_i, \tag{1.37}$$

gde su α_i i β_i koeficijenti koje treba da odredimo. Zamenjujući jednačinu (1.37) u jednačinu (1.35) dobija se

$$A_i^- \phi_{i-1}^{n+1} + A_i^0 \phi_i^{n+1} + A_i^+ (\alpha_i \phi_i^{n+1} + \beta_i) = b_i \ , \tag{1.38}$$

što dovodi do rešenja u obliku

$$\phi_i^{n+1} = \gamma_i (A_i^- \phi_{i-1}^{n+1} + A_i^+ \beta_i - b_i) \ , \tag{1.39}$$

gde su koeficijenti γ_i dati sa

$$\gamma_i = -1/(A_i^0 + A_i^+ \alpha_i) \ . \tag{1.40}$$

Iz jednačina (1.37) i (1.39) dobijaju se sledeće rekurzivne relacije za koeficijente α_i i β_i :

$$\alpha_{i-1} = \gamma_i A_i^- \ , \ \beta_{i-1} = \gamma_i (A_i^+ \beta_i - b_i) \ . \tag{1.41}$$

Koristeći rekurzivne relacije (1.39), (1.40) i (1.41), prvo prolazimo unatrag kroz rešetku („backward-sweep“) kako bismo odredili koeficijente α_i i β_i za tačku *i* polazeći od $i = N_z - 2$ ka *i*=0. Odabrane inicijalne vrednosti su takde da važi $\alpha_{N_{z-1}} = 0$, $\beta_{N_{z-1}} = \phi_{N_z}^{n+1} \equiv 0$. Nakon određivanja koeficijenata α_i , β_i i γ_i, može se iskoristiti rekurzivna relacija (1.37) pomoću koje u još jednom prolazu kroz rešetku, ovaj put u prolazu unapred („forward-sweep“), počev od $i = 0$ do $i = N_z - 1$, određujemo rešenje u konačnom trenutku iteracije za čitavu rešetku. Pri tome podrazumeva se da je vrednost talasne funkcije na granicama rešetke jednaka nuli, na osnovu graničnih uslova. Na ovaj način se završava jedna iteracija, koristeći dva rekurzivna prolaza kroz čitavu rešetku, pri čemu svaki skup uključuje N_z operacija. Ovo je posledica korišćenja poluimplicitne šeme. Kada bismo koristili eksplicitnu šemu, u kojoj bi se sa desne strane jednačine (1.34) nalazio samo diskretizovani izraz za drugi izvod po koordinati *z* u finalnom trenutku iteracije t_{n+1} , onda bi računarski situacija bila mnogo jednostavnija i imali bismo eksplicitne izraze u svakoj tački rešetke, koji ne bi zahtevali dva rekurzivna prolaza. Međutim, takav metod ne daje bezuslovno stabilan algoritam, pa smo se zbog toga odlučili za Krenk-Nikolson metod sa poluimplicitnom diskretizacijom koji rešava taj važan problem.

U numeričkoj implementaciji Krenk-Nikolson propagacije u realnom vremenu početno stanje u vremenu $t = t_0$ se bira u skladu sa zadatim uslovima, odnosno eksperimentalnom situacijom koja se modelira i koja zahteva proučavanje dinamike sistema pomoću propagacije u realnom vremenu.

1.5.3 Propagacija u imaginarnom vremenu

U slučaju propagacije u imaginarnom vremenu, koja se koristi za nalaženje osnovnog stanja sistema, GP jednačina je u potpunosti realna, što povlači da je i talasna funkcija realna, kao i sve veličine koje se koriste u algoritmu. U ovom slučaju vreme t se zamenjuje imaginarnom veličinom $t=-i\bar{t}$, a GP jednačina (1.25) postaje:

$$-\frac{\partial}{\partial\bar{t}}\phi(z;\bar{t})=\left[-\frac{\partial^2}{\partial z^2}+\frac{z^2}{4}+\mathcal{N}|\phi(z;\bar{t})|^2\right]\phi(z;\bar{t})\equiv H\phi(z;\bar{t}), \tag{1.42}$$

pri čemu podrazumevamo da je novo vreme $\bar{t}$ realna veličina i nadalje ćemo ga označavati sa *t*.

Iz jednačine (1.42) vidimo da se sopstveno stanje $\phi_{(i)}$ koje odgovara sopstvenoj energiji E_i Hamiltonijana *H*, odnosno zadovoljava relaciju $H\phi_{(i)}=E_i\phi_{(i)}$, u propagaciji u imaginarnom vremenu zadovoljava jednačinu $\partial\phi_{(i)}(z;t)/\partial t=-E_i\phi_{(i)}(z;t)$. Ona se može egzaktno rešiti i dobijamo da važi $\phi_{(i)}(z;t)=e^{-E_i(t-t_0)}\phi_{(i)}(z;t_0)$. Stoga, ako se krene od proizvoljnog početnog uslova $\phi(z;t=t_0)$, koje se uvek može izraziti kao linearna kombinacija svih sopstvenih funkcija Hamiltonijana *H*, tokom propagacije u imaginarnom vremenu će se doprinosi pojedinih sopstvenih funkcija eksponencijalno smanjivati, a zbog renormiranja talasne funkcije u svakoj iteraciji preostaće samo doprinos koji odgovara osnovnom stanju. Dakle, nakon dovoljno dugog vremena propagacije ostaje samo osnovno stanje, što se može pratiti po konvergenciji ostalih relevantnih fizičkih veličina, kao što su energija i hemijski potencijal, koje će biti uvedene u sledećoj sekciji.

Iteriranje u imaginarnom vremenu se izvodi razdvajanjem Hamiltonija *H* na dva dela, kao i kod propagacije u realnom vremenu, $H=H_1+H_2$, gde su izrazi za H_1 i H_2 dati sa jednačinama (1.26) i (1.27). Čitava analiza u poglavlju 1.5.2 ostaje validna, a odgovarajući izrazi za propagaciju u imaginarnom vremenu se dobijaju ako se zameni $i\Delta$ sa Δ. Pored toga, kao što smo već naglasili, na kraju svake iteracije potrebno je uvesti dodatnu opearciju renormalizacije talasne funkcije, u skladu sa jednačinom (1.21). To se svodi na računanje norme talasne funkcije i njenog množenja odgovarajućim faktorom tako da je na kraju norma jednaka jedinici.

Poredeći propagaciju u realnom i imaginarnom vremenu, možemo da zaključimo da je propagacija u imaginarnom vremenu robusnija. Početno stanje je praktično proizvoljno, a konvergencija ka osnovnom stanju je eksponencijalno brza. Takođe, puna nelinearnost može da se dodaje u malom broju vremenskih koraka ili čak odmah na početku, u jednom koraku, a ne u velikom broju koraka što je uglavnom neophodnu pri primeni drugih metoda. Ako ovome dodamo i činjenicu da su memorijski zahtevi za simulaciju propagacije u imaginarnom vremenu manji (jer su sve veličine realne), kao i da je potrebno manje procesorskog vremena (računske operacije sa kompleksnim brojevima su obično dva puta sporije od operacija sa realnim brojevima), vidimo da je algoritam za računanje osnovnog stanja veoma efikasan i da se može koristiti za proučavanje mnogo šire klase problema, koji ne zahtevaju detaljno modeliranje dinamike sistema.

1.5.4 Hemijski potencijal i energija sistema

Stacionarna stanja talasne funkcija u jednodimenzionalnom slučaju u propagaciji u realnom vremenu imaju trivijalnu vremensku zavisnost, $\phi(z;t)\equiv\hat{\phi}(z)e^{-i\mu t}$, gde je μ veličina koja se obično zove hemijski potencijal. Ako ovu faktorizaciju zamenimo u jednačinu (1.20), dobijamo da stacionarna stanja zadovoljavaju uslov

$$\left[-\frac{\partial^2}{\partial z^2}+\frac{z^2}{4}+\mathcal{N}\hat{\phi}^2(z)\right]\hat{\phi}(z)=\mu\hat{\phi}(z)\,. \tag{1.43}$$

Pošto je talasna funkcija normalizovana na jediničnu vrednost $\int_{-\infty}^{\infty}\hat{\phi}^2(z)\,dz=1$, hemijski potencijal se može izračunati iz gornjeg izraza množenjem jednačine sa $\hat{\phi}(z)$ i integracijom po čitavom prostoru, odnosno

$$\mu=\int_{-\infty}^{\infty}\left[\left(\frac{\partial\hat{\phi}(z)}{\partial z}\right)^2+\hat{\phi}^2(z)\left(\frac{z^2}{4}+\mathcal{N}\hat{\phi}^2(z)\right)\right]dz\,. \tag{1.44}$$

Pored ovoga, od fizičkog interesa je i energija sistema, koja je data sličnim izrazom, u kojem je nelinearni član pomnožen faktorom 1/2,

$$E=\int_{-\infty}^{\infty}\left[\left(\frac{\partial\hat{\phi}(z)}{\partial z}\right)^2+\hat{\phi}^2(z)\left(\frac{z^2}{4}+\frac{\mathcal{N}}{2}\hat{\phi}^2(z)\right)\right]dz\,. \tag{1.45}$$

1.5.5 Srednji kvadratni radijus

Jedna od važnih veličina za proučavanje Boze-Ajnštajn kondenzata je i srednji kvadratni radijus („root mean square“ ili „rms“), koji odgovara tipičnoj veličini sistema, a u 1D se računa kao kvadratni koren očekivane vrednosti kvadrata odgovarajuće koordinate. Ova očekivana vrednost je data izrazom

$$\langle z^2\rangle=\int_{-\infty}^{\infty}z^2\hat{\phi}^2(z)\,dz\,, \tag{1.46}$$

a srednji kvadratni radijus je $z_{\rm rms}=\sqrt{\langle z^2\rangle}$.

1.6 Krenk-Nikolson metoda podeljenog koraka za Gros-Pitaevski jednačinu u 3D

Jednačina (1.14) u ovom slučaju može da se predstavi sledećim Hamiltonijanom:

$$H=-\frac{\partial^2}{\partial x^2}-\frac{\partial^2}{\partial y^2}-\frac{\partial^2}{\partial z^2}+\frac{1}{4}\left(\nu^2 x^2+\kappa^2 y^2+\lambda^2 z^2\right)+\mathcal{N}\left|\phi(x,y,z;t)\right|^2 \tag{1.47}$$

gde je $\mathcal{N}=8\sqrt{2}\pi a_s N/l$. U 3D Hamiltonijan se rastavlja na 4 dela, $H=H_1+H_2+H_3+H_4$, gde je deo bez izvoda sličan kao i ranije,

$$H_1=\frac{1}{4}\left(\nu^2 x^2+\kappa^2 y^2+\lambda^2 z^2\right)+\mathcal{N}\left|\phi(x,y,z;t)\right|^2, \tag{1.48}$$

dok su tri komponente Laplasovog operatora preostali delovi Hamiltonijana,

$$H_2=-\frac{\partial^2}{\partial x^2},\ H_3=-\frac{\partial^2}{\partial y^2},\ H_4=-\frac{\partial^2}{\partial z^2}. \tag{1.49}$$

Za propagaciju u realnom i imaginarnom vremenu se koristi slična procedura, koja je objašnjena u prethodnim poglavljima, samo što se ovde ima više međukoraka. Vremenska propagacija po H_1 se izvodi kao u jednačini (1.30), a vremenske propagacije po H_2, H_3 i H_4 su date jednačinama analognim (1.32) i (1.33).

1.6.1 Hemijski potencijal i energija sistema u 3D

Slično kao i u jednodimenzionalnom slučaju, tako i ovde za stacionarna stanja talasne funkcija možemo da definišemo važne veličine kao što su hemijski potencijal i energija. Za stacionarna stanja propagacija u realnom vremenu je data sa $\phi(x,y,z;t)\equiv\hat{\phi}(x,y,z)e^{-i\mu t}$, gde je μ veličina koja se zove hemijski potencijal. Ako ovu faktorizaciju zamenimo u GP jednačinu u 3D (1.14), dobijamo da stacionarna stanja zadovoljavaju uslov

$$\left[-\frac{\partial^2}{\partial x^2}-\frac{\partial^2}{\partial y^2}-\frac{\partial^2}{\partial z^2}+\frac{1}{4}\left(x^2+y^2+z^2\right)+\mathcal{N}\hat{\phi}^2(x,y,z)\right]\hat{\phi}(x,y,z)=\mu\hat{\phi}(x,y,z). \tag{1.50}$$

Sledeći istu proceduru kao i ranije, za hemijski potencijal dobijamo integralni izraz

$$\mu = \int_{-\infty}^{\infty} \left[\left(\frac{\partial \hat{\phi}(x,y,z)}{\partial x} \right)^2 + \left(\frac{\partial \hat{\phi}(x,y,z)}{\partial y} \right)^2 + \left(\frac{\partial \hat{\phi}(x,y,z)}{\partial z} \right)^2 + \hat{\phi}^2(x,y,z) \left(\frac{x^2+y^2+z^2}{4} + \mathcal{N} \hat{\phi}^2(x,y,z) \right) \right] d\mathbf{r} , \tag{1.51}$$

dok je energija sistema data izrazom u kome je nelinearni član pomnožen faktorom 1/2,

$$E = \int_{-\infty}^{\infty} \left[\left(\frac{\partial \hat{\phi}(x,y,z)}{\partial x} \right)^2 + \left(\frac{\partial \hat{\phi}(x,y,z)}{\partial y} \right)^2 + \left(\frac{\partial \hat{\phi}(x,y,z)}{\partial z} \right)^2 + \hat{\phi}^2(x,y,z) \left(\frac{x^2+y^2+z^2}{4} + \frac{\mathcal{N}}{2} \hat{\phi}^2(x,y,z) \right) \right] d\mathbf{r} . \tag{1.52}$$

1.6.2 Srednji kvadratni radijus u 3D

Kao i u 1D slučaju, veličina Boze-Ajnštajn kondenzata se u 3D slučaju ocenjuje pomoću srednjeg kvadratnog radijusa („root mean square“ ili „rms“), koji se računa kao kvadratni koren očekivane vrednosti kvadrata radijusa. Ova očekivana vrednost je data izrazom

$$\langle r^2 \rangle = \int_{-\infty}^{\infty} r^2 \hat{\phi}^2(x,y,z) d\mathbf{r} = \int_{-\infty}^{\infty} (x^2+y^2+z^2) \ \hat{\phi}^2(x,y,z) d\mathbf{r} , \tag{1.53}$$

dok je srednji kvadratni radijus je $r_{\mathrm{rms}} = \sqrt{\langle r^2 \rangle}$.

1.7 Tema istraživanja

1.7.1 Predmet i problem

Prethodno razvijeni programi napisani na jezicima Fortran [10] i C [11], napravljeni za rešavanje GP jednačine uz pomoć Krenk-Nikolson metode, široko su korišćeni u naučnim zajednicama koje proučavaju ultrahladne atome, nelinearnu optiku, ali i u raznim drugim poljima [20-70].

Pomenuti programi, međutim, poseduju inherentna ograničenja. Program opisan u referenci [11], paralelizovan uz pomoć C/OpenMP platforme za rad sa deljenom memorijom („shared-memory“), ograničava rezoluciju posmatranog fizičkog sistema količinom radne memorije računara

na kojem se izvršava simulacija. Program opisan u referenci [10], realizovan u Fortranu, predstavlja inicijalnu sekvencijalnu verziju paralelizovanog C/OpenMP programa [11] i njegovo ograničenje, osim rezolucije fizičkog sistema, jeste i brzina izvršavanja, jer je ograničena sekvencijalnom obradom podataka, odnosno ne postoji nikakva vrsta paralelizacije.

1.7.2 Ciljevi teze

Prvo cilj ove teze je razvoj i implementacija algoritama za dekompoziciju i transponovanje dvodimenzionalnih (2D) i trodimenzionalnih (3D) matrica u skladu sa memorijskim rasporedom višedimenzionalnih matrica u programskom jeziku C. Nakon toga, ove algoritme je potrebno testirati i verifikovati njihovu korektnost u 2D slučaju i primeniti na računaru sa deljenom memorijom. Nakon verifikacije algoritma i implementacije 2D transponovanja, sledeći cilj je njegovo uopštavanje na 3D slučaj sa primenom na računarskom klasteru sa distribuiranom memorijom. Ovako uopšteni algoritam je takođe neophodno testirati i verifikovati njegovu korektnost u 3D slučaju i primeniti na računarskom klasteru sa distribuiranom memorijom.

Naredni cilj teze je primena razvijenog algoritma dekompozicije i transponovanja 3D matrica u okviru postojećeg C/OpenMP programa [11] na talasnu funkciju sistema, uz prilagođavanje postojećeg programa distribuiranoj softverskoj paradigmi. Nakon ovoga, novi paralelni program je potrebno detaljno testirati i utvrditi da se sve fizičke veličine i numerički rezultati novog rešenja podudaraju sa onima dobijenim uz pomoć prethodnog rešenja [11].

Pored toga, cilj teze je i razvoj grupe testova za merenje efikasnosti, ubrzanja i skaliranja novog rešenja, uz uporednu analizu dobijenih rezultata sa rezultatima postojećeg rešenja [11], naglašavajući prednosti i mane, kao i polja primene jednog i drugog rešenja.

2 Problem rezolucije fizičkog sistema

U ovom poglavlju ćemo prvo analizirati prethodno razvijene programe [11] za rešavanje GP jednačine u 3D pomoću Krenk-Nikolson algoritma. Nakon toga ćemo identifikovati ograničenja ovog pristupa i motivisati razvoj novog, hibridnog rešenja, koje je opisano u ovoj doktorskoj tezi.

2.1 Analiza C/OpenMP programa za implementaciju Krenk-Nikolson algoritma

Prethodno razvijeni C/OpenMP program [11] za implementaciju Krenk–Nikolson algoritma realizovan je uz pomoć skupa funkcija koje obavljaju pojedinačne delove numeričkog algoritma, opisanog u poglavlju 1.6. Ključne funkcije u algoritmu su:

- `calcnorm` – funkcija za računanje norme talasne funkcije sistema,
- `caclmuen` – funkcija za računanje hemijskog potencijala i energije,
- `calcrms` – funkcija za računanje srednjeg kvadratnog radijusa talasne funkcije,
- `calcnu` – funkcija za računanje vremenske propagacije u odnosu na H_1 deo Hamiltonijana (članovi bez prostornih izvoda),
- `calclux` – funkcija za računanje vremenske propagacije u odnosu na H_2 deo Hamiltonijana (x-komponenta Laplasijana),
- `calcluy` – funkcija za računanje vremenske propagacije u odnosu na H_3 deo Hamiltonijana (y-komponenta Laplasijana),
- `calcluz` – funkcija za računanje vremenske propagacije u odnosu na H_4 deo Hamiltonijana (z-komponenta Laplasijana).

U narednim poglavljima napravićemo pregled strukture postojećeg C/OpenMP programa za propagaciju u imaginarnom vremenu. Program za propagaciju u realnom vremenu ima gotovo istovetnu strukturu kao i program za propagaciju u imaginarnom vremenu i stoga neće biti posebno analiziran. Jedina bitna razlika je da su pojedine promenljive koje se odnose na talasnu funkciju kompleksnog tipa kod propagacije u realnom vremenu, dok su kod propagacije u imaginarnom vremenu odgovarajuće promenljive realnog tipa.

Program za propagaciju u imaginarnom vremenu realizovan je u okviru sledećih fajlova: `imagtime3d-th.c`, `imagtime3d-th.h`, `cfg.c`, `cfg.h`, `diffint.c`, `diffint.h`, `mem.c`, `mem.h`.

2.1.1 Fajl `imagtime3d-th.h`

Fajl `imagtime3d-th.h` predstavlja zaglavlje fajla glavnog programa `imagtime3d-th.c`. Zaglavlje `imagtime3d-th.h` sadrži:

- uključivanje svih biblioteka neophodnih za rad glavnog programa,
- definiciju konstanti i deklaraciju promenljivih koje koristi glavni program,
- deklaraciju funkcija koje definiše glavni program,
- eksterne funkcije koje potiču iz drugih fajlova a bitne su za rad glavnog programa.

U listingu 2.1 dat je izvorni kôd zaglavlja `imagtime3d-th.h`.

Listing 2.1: Izvorni kôd zaglavlja `imagtime3d-th.h.`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <omp.h>

#define MAX(a, b, c) (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c)
#define MAX_FILENAME_SIZE 256

char *output, *initout, *Npasout, *Nrunout;
long outstpx, outstpy, outstpz;

int opt;
long Nx, Ny, Nz;
long Nx2, Ny2, Nz2;
long Npas, Nrun;
double dx, dy, dz;
double dx2, dy2, dz2;
double dt;
double G0, G;
double kappa, lambda;
double par;

double *x, *y, *z;
double *x2, *y2, *z2;
double ***pot;

double Ax0, Ay0, Az0, Ax0r, Ay0r, Az0r, Ax, Ay, Az;
```

```
double *calphax, *calphay, *calphaz;
double *cgammax, *cgammay, *cgammaz;

void readpar(void);
void init(double ***);
void gencoef(void);
void calcnorm(double *, double ***, double **, double **, double **);
void calcmuen(double *, double *, double ***, double ***, double ***, double
***, double **, double **, double **, double **, double **, double **);
void calcrms(double *, double ***, double **, double **, double **);
void calcnu(double ***);
void calclux(double ***, double **);
void calcluy(double ***, double **);
void calcluz(double ***, double **);

extern double simpint(double, double *, long);
extern void diff(double, double *, double *, long);

extern double *alloc_double_vector(long);
extern double **alloc_double_matrix(long, long);
extern double ***alloc_double_tensor(long, long, long);
extern void free_double_vector(double *);
extern void free_double_matrix(double **);
extern void free_double_tensor(double ***);
extern int cfg_init(char *);
extern char *cfg_read(char *);
```

Na početku fajla `imagtime3d-th.h` uključuju se zaglavlja neophodna za rad glavnog programa. Pored standardnih zaglavlja `<stdio.h>`, `<stdlib.h>` i `<string.h>` uključuje se i zaglavlje za rad sa matematičkim funkcijama `<math.h>` kao i zaglavlje neophodno za rad sa OpenMP direktivama `<omp.h>`. Potom se definiše preprocesorski makro `MAX` za određivanje maksimuma od tri broja, kao i konstanta `MAX_FILENAME_SIZE`, koja predstavlja maksimalan broj karaktera, dozvoljen u imenu fajlova, koji će biti korišćeni u programu.

Sledi spisak i kratak opis promenljvih deklarisanih u zaglavlju:

- `output` - naziv izlaznog fajla, koji sadrži konačne vrednosti svih fizičkih veličina,
- `initout` - naziv izlaznog fajla sa inicijalnom talasnom funkcijom,
- `Npasout` - naziv izlaznog fajla sa talasnom funkcijom, dobijenom nakon `Npas` iteracija sa fiksnom nelinearnošću,
- `Nrunout` - naziv izlaznog fajla sa talasnom funkcijom, dobijenom nakon konačnih `Nrun` iteracija,
- `outspx, outspy, outspz` - diskretizacioni koraci u *x*, *y* i *z* pravcima, korišćeni za čuvanje talasne funkcije,
- `opt` - opcija kojom se bira koje će reskaliranje biti korišćeno za Gros-Pitaevski jednačinu,
- `Nx, Ny, Nz` - broj diskretizacionih tačaka u *x*, *y* i *z* pravcima respektivno,

- `Nx2, Ny2, Nz2` - tačka na polovini opsega u *x*, *y* i *z* pravcima respektivno,
- `Npas, Nrun` - broj uzastopnih i broj konačnih iteracija sa fiksnom nelinearnošću G0,
- `dx, dy, dz` - prostorni diskretizacioni koraci u *x*, *y* i *z* pravcima respektivno,
- `dx2, dy2, dz2` - kvadrati prethodno opisanih prostornih diskretizacionih koraka,
- `dt` - vremenski diskretizacioni korak,
- `G0, G` - konačna nelinearnost i skalirana konačna nelinearnost,
- `kappa`, `lambda` - koeficijenti anizotropnosti zamke,
- `par` - parametar reskaliranja Gros-Pitaevski jednačine,
- `x, y, z` - nizovi koji sadrže vrednosti prostorne mreže u *x*, *y* i *z* pravcima respektivno,
- `x2, y2, z2` - nizovi koji sadrže kvadrate vrednosti iz nizova prostorne mreže u *x*, *y* i *z* pravcima respektivno,
- `pot` - trodimenzionalni niz sa vrednostima potencijala,
- `Ax0, Ay0, Az0, Ax0r, Ay0r, Az0r, Ax, Ay, Az` - koeficijenti Krenk-Nikolson šeme (objašnjeni u poglavlju 1.5.2),
- `calphax, calphay, calphaz` - koeficijenti Krenk-Nikolson šeme (objašnjeni u poglavlju 1.5.2),
- `cgammax, cgammay, cgammaz` - koeficijenti Krenk-Nikolson šeme (objašnjeni u poglavlju 1.5.2).

Nakon spiska promenljvih sledi spisak funkcija deklarisanih u zaglavlju `imagtime3d-th.h`. Ove funkcije će detaljno biti opisane u okviru fajla u kom su definisane:

- `readpar` - funkcija za čitanje ulaznih parametara iz konfiguracionog fajla,
- `init` - funkcija za inicijalizaciju prostorne mreže, potencijala i inicijalne talasne funkcije,
- `gencoef` - funkcija za generisane Krenk-Nikolson koeficijenata,
- `calcnorm` - funkcija za izračunavanje norme talasne funkcije i za normalizaciju,
- `calcmuen` - funkcija za izračunavanje hemijskog potencijala i energije,
- `calcrms` - funkcija za izračunavanje srednje kvadratne vrednosti poluprečnika,
- `calcnu` – funkcija za računanje vremenske propagacije u odnosu na H_1 deo Hamiltonijana (članovi bez prostornih izvoda),
- `calclux` – funkcija za računanje vremenske propagacije u odnosu na H_2 deo Hamiltonijana (*x*-komponenta Laplasijana),

- `calcluy` – funkcija za računanje vremenske propagacije u odnosu na H_3 deo Hamiltonijana (y-komponenta Laplasijana),
- `calcluz` – funkcija za računanje vremenske propagacije u odnosu na H_4 deo Hamiltonijana (z-komponenta Laplasijana),
- `simpint` - eksterna funkcija za jednodimenzionalnu prostornu integraciju po Simpsonovom pravilu,
- `diff` - eksterna funkcija za izračuvanje prostornih izvoda uz pomoć Ričardsonove ekstrapolacije,
- `alloc_double_vector` - eksterna funkcija za alociranje vektora vrednosti u dvostrukoj preciznosti,
- `alloc_double_matrix` - eksterna funkcija za alociranje matrice vrednosti u dvostrukoj preciznosti,
- `alloc_double_tensor` - eksterna funkcija za alociranje tenzora (niza matrica) vrednosti u dvostrukoj preciznosti,
- `free_double_vector` - eksterna funkcija za oslobađanje memorije zauzete za vektor vrednosti u dvostrukoj preciznosti,
- `free_double_matrix` - eksterna funkcija za oslobađanje memorije zauzete za matricu vrednosti u dvostrukoj preciznosti,
- `free_double_tensor` - eksterna funkcija za oslobađanje memorije zauzete za tenzor vrednosti u dvostrukoj preciznosti,
- `cfg_init` - eksterna funkcija za parsiranje konfiguracionog fajla,
- `cfg_read` - eksterna funkcija za čitanje konfiguracionog fajla po sistemu ključ – vrednost.

2.1.2 Fajl `cfg.h`

Fajl `cfg.h` predstavlja zaglavlje fajla `cfg.c`. Zaglavlje `cfg.h` sadrži deklaracije funkcija koje se koriste pri čitanju konfiguracionih fajlova iz glavnog programa. U listingu 2.2 dat je izvorni kôd zaglavlja `cfg.h`.

Listing 2.2: Izvorni kôd zaglavlja `cfg.h`.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>

int cfg_size;
char cfg_key[256][256], cfg_val[256][256];

int cfg_init(char *);
char *cfg_read(char *);
```

Na početku zaglavlja cfg.h uključuju se standardna zaglavlja za rad sa C programima `<stdio.h>`, `<stdlib.h>` i `<string.h>`. Od promenljivih, zaglavlje deklariše promenljivu `cfg_size`, koja će sadržati tačan broj učitanih konfiguracionih promenljivih. Konfiguracioni fajl, koji se učitava u glavnom programu, može imati promenljivu dužinu konfiguracije (jer postoje opcione konfiguracione promenljive) i stoga je neophodno znati tačan broj učitanih konfiguracionih promenljivih. Promenljive `<cfg_key>` i `<cfg_val>` predstavljaju niz ključeva i niz odgovarajućih vrednosti za te ključeve. Dati nizovi će u sebi, nakon učitavanja konfiguracionog fajla, sadržati nazive konfiguracionih promenljivih (ključ) i njihove odgovarajuće vrednosti (vrednost). Na kraju fajla `cfg.h` date su i deklaracije funkcija `cfg_init` (funkcija za parsiranje konfiguracionog fajla) i `cfg_read` (funkcija za čitanje konfiguracionog fajla po sistemu ključ-vrednost).

2.1.3 Fajl `cfg.c`

Fajl `cfg.c` sadrži definiciju funkcija `cfg_init` i `cfg_read` deklarisanih u fajlu `cfg.h`. U listingu 2.3 dat je izvorni kôd zaglavlja `cfg.c`.

Listing 2.3: Izvorni kôd fajla `cfg.c`.

```c
#include "cfg.h"

/**
 *    Parsiranje konfiguracionog fajla.
 *    cfg_file – naziv konfiguracionog fajla koji se zadaje u komandnoj liniji
 */
int cfg_init(char *cfg_file) {
   FILE *file;
   char buf[256];

   file = fopen(cfg_file, "r");
   if (! file) return 0;

   cfg_size = 0;
   while (fgets(buf, 256, file) != NULL) {
      if (sscanf(buf, "%s = %s", cfg_key[cfg_size], cfg_val[cfg_size]) == 2)
         cfg_size ++;
```

```
    }

    fclose(file);
    return cfg_size;
}

/**
 *    Citanje vrednosti iz konfiguracije u odnosu na kljuc key
 */
char *cfg_read(char *key) {
    int i;

    for(i = 0; i < cfg_size; i ++)
       if (! strcmp(key, cfg_key[i])) return cfg_val[i];

    return NULL;
}
```

Funkcija `cfg_init` predstavlja funkciju za parsiranje ulaznog konfiguracionog fajla. Ulazni konfiguracioni fajl se parsira tako što se, nakon otvaranja fajla, vrši čitanje liniju po liniju teksta u `while` petlji uz pomoć funkcije `fgets`. Svaka od pročitanih linija se smešta u privremeni bafer podataka `buf`. Potom se u okviru same petlje vrši smeštanje odgovarajućih konfiguracionih parova ključ vrednost u promenljive `cfg_key[cfg_size]` i `cfg_val[cfg_size]`. Smeštanje se vrši pozivom funkcije `sscanf`, koja za argument uzima prethodno pročitanu liniju konfiguracionog fajla, koja se nalazi u baferu `buf`. Promenljiva `cfg_size` označava broj pročitanih konfiguracionih promenljivih i povećava se za svaku novu pročitanu konfiguracionu promenljivu.

Primer konfiguracionog fajla dat je u listingu 2.4 (ključevi su označeni zelenom bojom, a vrednosti crvenom).

Listing 2.4: Izgled konfiguracionog fajla glavnog programa.

```
# Parameter input file for solving the time-independent Gross–Pitaevskii
# nonlinear partial differential equation in a trap using split-step
# Crank-Nicolson method.

# Type of rescaling of Gross–Pitaevskii equation.
# Possible values: 1, 2, and 3.
# Required: yes
# Type: int
  OPTION = 2

# Coefficient of the nonlinear term.
# Required: yes
# Type: double
  G0 = 44.907

# Number of discretization points in the x-direction.
# Required: yes
# Type: long
  NX = 240
```

```
# Number of discretization points in the y-direction.
# Required: yes
# Type: long
  NY = 200

# Number of discretization points in the z-direction.
# Required: yes
# Type: long
  NZ = 160

# Spatial discretization step in the x-direction.
# Required: yes
# Type: double
  DX = 0.05

# Spatial discretization step in the y-direction.
# Required: yes
# Type: double
  DY = 0.05

# Spatial discretization step in the z-direction.
# Required: yes
# Type: double
  DZ = 0.05

# Time discretization step.
# Required: yes
# Type: double
  DT = 0.0004

# Kappa coefficient of anisotropy of the trap (omega_y / omega_x).
# Required: yes
# Type: double
  AL = 1.4142135623731

# Lambda coefficient of anisotropy of the trap (omega_z / omega_x).
# Required: yes
# Type: double
  BL = 2.0

# Number of subsequent iterations with fixed nonlinearity G0.
# Required: yes
# Type: long
  NPAS = 5000

# Number of final iterations with fixed nonlinearity G0.
# Required: yes
# Type: long
  NRUN = 500

# Output file with the summary of final values of all physical quantities. If
# not defined, standard output will be used.
# Required: no
# Type: string
  OUTPUT = imagtime3d-output
```

```
# Output file with the initial wave function. Only 1D sections along
# coordinate axes of the wave function are saved, which is designated by a
# suffix x, y, or z. If not defined, the initial wave function will not be
# saved.
# Required: no
# Type: string
  INITOUT = imagtime3d-initout

# Output file with the wave function obtained after the subsequent NPAS
# iterations, with the fixed nonlinearity G0. Only 1D sections along
# coordinate axes of the wave function are saved, which is designated by a
# suffix x, y, or z. If not defined, the wave function will not be saved.
# Required: no
# Type: string
  NPASOUT = imagtime3d-npasout

# Output file with the final wave function obtained after the final NRUN
# iterations. Only 1D sections along coordinate axes of the wave function are
# saved, which is designated by a suffix x, y, or z. If not defined, the wave
# function will not be saved.
# Required: no
# Type: string
  NRUNOUT = imagtime3d-nrunout

# Discretization step in the x-direction used to save wave functions.
# It's required if any of wave function outputs (INITOUT, NPASOUT,
# NRUNOUT) is defined.
# Required: conditionally
# Type: long
  OUTSTPX = 1

# Discretization step in the y-direction used to save wave functions. Required
# if any of wave function output files (INITOUT, NPASOUT, NRUNOUT) is defined.
# Required: conditionally
# Type: long
  OUTSTPY = 1

# Discretization step in the z-direction used to save wave functions. Required
# if any of wave function output files (INITOUT, NPASOUT, NRUNOUT) is defined.
# Required: conditionally
# Type: long
  OUTSTPZ = 1
```

S obzirom da C jezik ne poseduje ugrađenu biblioteku za manipulaciju strukturama podataka sa parovima ključ-vrednost, kao npr. C++ jezik u okviru STL klase mapa, bilo je neophodno napisati funkciju `cfg_read`, koja će za prosleđeni ključ vratiti odgovarajuću vrednost. Ovo je realizovano tako što se prolazi kroz čitav niz ključeva `cfg_key` i utvrđuje da li se u njemu nalazi prosleđeni ključ `key`. Ukoliko se nalazi, vraća se vrednost na istoj poziciji u nizu kao i ključ, a u suprotnom se vraća vrednost `NULL`.

2.1.4 Fajl `diffint.h`

Fajl `diffint.h` predstavlja zaglavlje fajla `diffint.c`. Zaglavlje `diffint.h` sadrži deklaracije funkcija koje se koriste pri numeričkoj integraciji i izvođenju. U listingu 2.5 dat je izvorni kôd zaglavlja `diffint.h`.

Listing 2.5: Izvorni kôd fajla `diffint.h`.

```
#include <stdio.h>

double simpint(double, double *, long);
void diff(double, double *, double *, long);
```

Deklaracija funkcije `simpint` predstavlja deklaraciju funkcije koja vrši integraciju po Simpsonovom pravilu. Deklaracija funkcije `diff` predstavlja deklaraciju funkcije koja vrši prostorno izvođenje uz pomoć Ričardsonove ekstrapolacije. Obe funkcije će biti detaljnije opisane u narednom poglavlju.

2.1.5 Fajl `diffint.c`

Fajl `diffint.c` sadrži definiciju funkcija `simpint` i `diff` deklarisanih u fajlu `diffint.h`. U listingu 2.6 dat je izvorni kôd zaglavlja `diffint.c`.

Listing 2.6: Izvorni kôd fajla `diffint.c`.

```
#include "diffint.h"

/**
 *    Prostorna 1D integracija Simpsonovim pravilom.
 *    h — prostorni korak
 *    f — niz sa vrednostima funkcije
 *    N — broj integracionih tacaka
 */
double simpint(double h, double *f, long N) {
   long cnti;
   double sum, sumi, sumj, sumk;

   sumi = 0.; sumj = 0.; sumk = 0.;

   for(cnti = 1; cnti < N - 1; cnti += 2) {
```

```
      sumi += f[cnti];
      sumj += f[cnti - 1];
      sumk += f[cnti + 1];
   }

   sum = sumj + 4. * sumi + sumk;
   if(N % 2 == 0) sum += (5. * f[N - 1] + 8. * f[N - 2] - f[N - 3]) / 4.;

   return sum * h / 3.;
}

/**
 *    Formula Ricardsonove ekstrapolacije za racunanje prostornih izvoda.
 *    h  - prostorni korak
 *    f  - niz sa vrednostima funkcije
 *    df – niz sa prvim izvodima funkcije
 *    N  - broj tacaka prostorne mreze
 */
void diff(double h, double *f, double *df, long N) {
   long cnti;

   df[0] = 0.;
   df[1] = (f[2] - f[0]) / (2. * h);

   for(cnti = 2; cnti < N - 2; cnti ++) {
      df[cnti] = (f[cnti - 2] - 8. * f[cnti - 1] + 8. *
                  f[cnti + 1] - f[cnti + 2]) / (12. * h);
   }

   df[N - 2] = (f[N - 1] - f[N - 3]) / (2. * h);
   df[N - 1] = 0.;

   return;
}
```

Funkcija `simpint` računa integral funkcije čije vrednosti su prosleđene u okviru niza `f`. Vrednosti funkcije su zabeležene u `N` tačaka, dok je udaljenost između dve susedne tačke `h`. Računanje integrala u funkciji se obavlja uz pomoć kompozitnog Simpsonovog pravila [12]. Kompozitno Simpsonovo pravilo, za razliku od običnog, uzima u obzir činjenicu da funkcija za koju se računa integral nije glatka. Ovo može značiti da je funkcija ili veoma oscilatorna ili da joj nedostaju izvodi u određenim tačkama. U ovom slučaju se interval, za koji se računa funkcija, deli na veći broj pod-intervala, na kojima se potom primenjuje osnovno Simpsonovo pravilo,

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right]. \tag{2.1}$$

Pretpostavimo da se interval $[a, b]$ podeli na n podintervala, pri čemu je n paran broj. Onda je kompozitno Simpsonovo pravilo dato sledećim izrazom:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\ldots+4f(x_{n-1})+f(x_n)\right]$$
$$= \frac{h}{3}\sum_{j=1}^{n/2}\left[f(x_{2j-2})+4f(x_{2j-1})+f(x_{2j})\right] \tag{2.2}$$

U skladu sa prethodnom formulom u funkciji `simpint` je implementirano sumiranje vrednosti funkcije u susednim tačkama od 1 do n-1, pri čemu se leva, srednja i desna tačka sumiraju u različite promenljive `sumj`, `sumi` i `sumk`, respektivno. Nakon sumiranja, srednja vrednost `sumi` se množi sa 4 i dodaje na `sumj` i `sumk`, što odgovara sumi elemenata u zagradi u okviru jednačine (2.2) U slučaju da je broj tačaka paran, na sumu se dodaje i ostatak u vidu skalirane sume vrednosti funkcije f u poslednje tri tačke. Na kraju, čitava suma se množi sa h/3, u skladu sa izrazom (2.2).

Funkcija `diff` računa Ričardsonovu ekstrapolaciju vrednosti funkcije f. Na ovaj način se aproksimiraju prostorni izvodi funkcije f. Sama funkcija `diff` prima kao parametre prostorni korak `h`, nizove `f` i `df` koji predstavljaju nizove vrednosti funkcije f i njenih izvoda, kao i promenljivu N koja označava broj tačaka u kojima se vrši aproksimacija prostornog izvoda. Primenjena je formula za Ričardsonovu ekstrapolaciju četvrtog reda [13], koja glasi:

$$f'(x)=\frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h}. \tag{2.3}$$

Data formula se jasno uočava u numeričkom kôdu u okviru funkcije `diff`. Krajne tačke izvoda su aproksimirane nulama, dok su prva i druga tačka izvoda sa levog i desnog kraja intervala aproksimirane kao količnik razlike susednih tačaka i dvostruke vrednosti koraka `h`, jer za njih ne može da se primeni ekstrapolacija četvrtog reda.

2.1.6 Fajl `mem.h`

Fajl `mem.h` predstavlja zaglavlje fajla `mem.c`. Zaglavlje mem.h sadrži deklaracije funkcija koje se koriste pri zauzimanju i oslobađanju memorije za različite strukture podataka, koje koristi glavni program `imatime3d-th.c`. U listingu 2.7 dat je izvorni kôd zaglavlja `mem.h`.

Listing 2.7: Izvorni kôd fajla `mem.h`.

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>

double *alloc_double_vector(long);
double complex *alloc_complex_vector(long);
```

```
double **alloc_double_matrix(long, long);
double ***alloc_double_tensor(long, long, long);
double complex ***alloc_complex_tensor(long, long, long);

void free_double_vector(double *);
void free_complex_vector(double complex *);
void free_double_matrix(double **);
void free_double_tensor(double ***);
void free_complex_tensor(double complex ***);
```

Na početku zaglavlja `mem.h` uključuju se standardna zaglavlja za rad sa C programima `<stdio.h>`, `<stdlib.h>`, kao i zaglavlje za rad sa kompleksnim brojevima `<complex.h>`. Kompleksni brojevi biće korišćeni u programu za propagaciju u realnom vremenu, i po automatizmu se zaglavlja za njih uključuju i u program za propagaciju u imaginarnom vremenu. Potom slede deklaracije funkcija za alociranje različitih tipova memorijskih promenljvih:

- `alloc_double_vector` - funkcija za alociranje vektora vrednosti u dvostrukoj preciznosti,
- `alloc_complex_vector` - funkcija za alociranje vektora kompleksnih vrednosti u dvostrukoj preciznosti,
- `alloc_double_matrix` - funkcija za alociranje matrice vrednosti u dvostrukoj preciznosti,
- `alloc_double_tensor` - funkcija za alociranje tenzora trećeg reda (niza matrica) vrednosti u dvostrukoj preciznosti,
- `alloc_complex_tensor` - funkcija za alociranje tenzora trećeg reda (niza matrica) kompleksnih vrednosti u dvostrukoj preciznosti.

Kao što se može primetiti, među funkcijama za alociranje memorije nedostaje funkcija za alociranje matrice kompleksnih vrednosti. Ovo proizilazi iz činjenice da numerički algoritam u slučaju propagacije u realnom vremenu ni u jednom momentu neće zahtevati strukturu podataka tog tipa, već samo kompleksne tenzore trećeg reda ili vektore. Slede deklaracije funkcija za oslobađanje alociranih memorijskih struktura podataka:

- `free_double_vector` - funkcija za oslobađanje memorije zauzete za vektor vrednosti u dvostrukoj preciznosti,
- `free_complex_vector` - funkcija za oslobađanje memorije zauzete za vektor kompleksnih vrednosti u dvostrukoj preciznosti,
- `free_double_matrix` - funkcija za oslobađanje memorije zauzete za matricu vrednosti u dvostrukoj preciznosti,
- `free_double_tensor` - funkcija za oslobađanje memorije zauzete za tenzor vrednosti u dvostrukoj preciznosti,

- `free_complex_tensor` - funkcija za oslobađanje memorije zauzete za tenzor kompleksnih vrednosti u dvostrukoj preciznosti.

2.1.7 Fajl `mem.c`

Fajl `mem.c` sadrži definiciju funkcija deklarisanih u fajlu `mem.h`. U listingu 2.8 dat je izvorni kôd zaglavlja `mem.c`.

Listing 2.8: Izvorni kôd fajla `mem.c`.

```
#include "mem.h"

/**
 *    Alokacija memorije za vektor vrednosti u dvostrukoj preciznosti.
 */
double *alloc_double_vector(long Nx) {
   double *vector;

   if((vector = (double *) malloc((size_t) (Nx * sizeof(double)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the vector.\n");
      exit(EXIT_FAILURE);
   }

   return vector;
}

/**
 *    Alokacija memorije za vektor kompleksnih vrednosti
 *    u dvostrukoj preciznosti.
 */
double complex *alloc_complex_vector(long Nx) {
   double complex *vector;

   if((vector = (double complex *) malloc((size_t)
      (Nx * sizeof(double complex)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the vector.\n");
      exit(EXIT_FAILURE);
   }

   return vector;
}

/**
 *    Alokacija memorije za matricu vrednosti u dvostrukoj preciznosti.
 */
double **alloc_double_matrix(long Nx, long Ny) {
   long cnti;
   double **matrix;

   if((matrix = (double **) malloc((size_t)
```

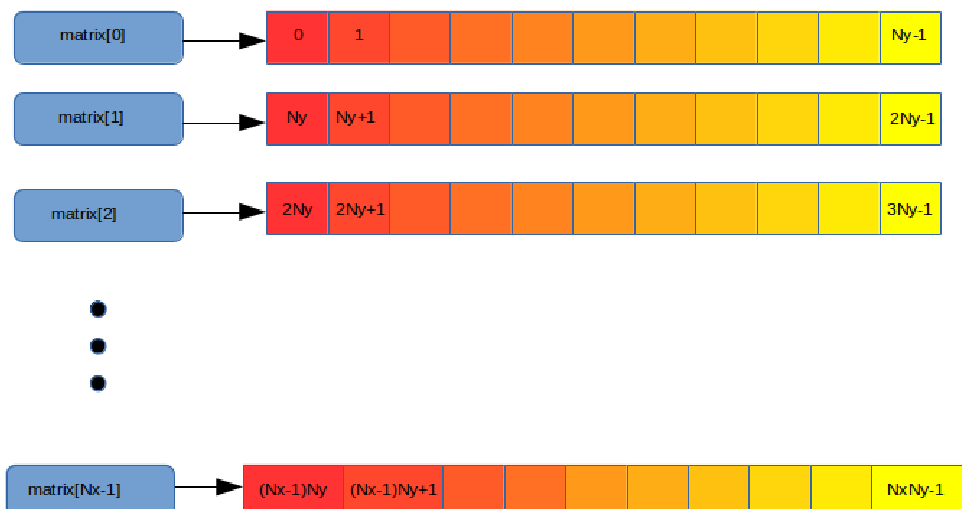
```
      (Nx * sizeof(double *)))) == NULL)    {
      fprintf(stderr, "Failed to allocate memory for the matrix.\n");
      exit(EXIT_FAILURE);
   }
   if((matrix[0] = (double *) malloc((size_t)
      (Nx * Ny * sizeof(double)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the matrix.\n");
      exit(EXIT_FAILURE);
   }
   for(cnti = 1; cnti < Nx; cnti ++)
      matrix[cnti] = matrix[cnti - 1] + Ny;

   return matrix;
}

/**
 *    Alokacija memorije za matricu kompleksnih vrednosti
 *    u dvostrukoj preciznosti.
 */
double complex **alloc_complex_matrix(long Nx, long Ny) {
   long cnti;
   double complex **matrix;

   if((matrix = (double complex **) malloc((size_t)
      (Nx * sizeof(double complex *)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the matrix.\n");
      exit(EXIT_FAILURE);
    }
   if((matrix[0] = (double complex *) malloc((size_t)
      (Nx * Ny * sizeof(double complex)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the matrix.\n");
      exit(EXIT_FAILURE);
   }
   for(cnti = 1; cnti < Nx; cnti ++)
      matrix[cnti] = matrix[cnti - 1] + Ny;

   return matrix;
}

/**
 *    Alokacija memorije za tenzor vrednosti u dvostrukoj preciznosti.
 */
double ***alloc_double_tensor(long Nx, long Ny, long Nz) {
   long cnti, cntj;
   double ***tensor;

   if((tensor = (double ***) malloc((size_t)
      (Nx * sizeof(double **)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n");
      exit(EXIT_FAILURE);
   }
   if((tensor[0] = (double **) malloc((size_t)
      (Nx * Ny * sizeof(double *)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n");
      exit(EXIT_FAILURE);
   }
   if((tensor[0][0] = (double *) malloc((size_t)
```

```
      (Nx * Ny * Nz * sizeof(double)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n");
      exit(EXIT_FAILURE);
   }
   for(cntj = 1; cntj < Ny; cntj ++)
      tensor[0][cntj] = tensor[0][cntj-1] + Nz;
   for(cnti = 1; cnti < Nx; cnti ++) {
      tensor[cnti] = tensor[cnti - 1] + Ny;
      tensor[cnti][0] = tensor[cnti - 1][0] + Ny * Nz;
      for(cntj = 1; cntj < Ny; cntj ++)
         tensor[cnti][cntj] = tensor[cnti][cntj - 1] + Nz;
   }

   return tensor;
}

/**
 *    Alokacija memorije za tenzor kompleksnih vrednosti
 *    u dvostrukoj preciznosti.
 */
double complex ***alloc_complex_tensor(long Nx, long Ny, long Nz) {
   long cnti, cntj;
   double complex ***tensor;

   if((tensor = (double complex ***) malloc((size_t)
      (Nx * sizeof(double complex **)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n");
      exit(EXIT_FAILURE);
   }
   if((tensor[0] = (double complex **) malloc((size_t)
      (Nx * Ny * sizeof(double complex *)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n");
      exit(EXIT_FAILURE);
   }
   if((tensor[0][0] = (double complex *) malloc((size_t)
      (Nx * Ny * Nz * sizeof(double complex)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n");
      exit(EXIT_FAILURE);
   }
   for(cntj = 1; cntj < Ny; cntj ++)
      tensor[0][cntj] = tensor[0][cntj-1] + Nz;
   for(cnti = 1; cnti < Nx; cnti ++) {
      tensor[cnti] = tensor[cnti - 1] + Ny;
      tensor[cnti][0] = tensor[cnti - 1][0] + Ny * Nz;
      for(cntj = 1; cntj < Ny; cntj ++)
         tensor[cnti][cntj] = tensor[cnti][cntj - 1] + Nz;
   }

   return tensor;
}

/**
 *    Oslobadjanje memorije zauzete za vektor vrednosti u
 *    dvostrukoj preciznosti.
 */
void free_double_vector(double *vector) {
   free((char *) vector);
```

```
}

/**
 *    Oslobadjanje memorije zauzete za vektor kompleksnih vrednosti u
 *    dvostrukoj preciznosti.
 */
void free_complex_vector(double complex *vector) {
   free((char *) vector);
}

/**
 *    Oslobadjanje memorije zauzete za matricu vrednosti u
 *    dvostrukoj preciznosti.
 */
void free_double_matrix(double **matrix) {
   free((char *) matrix[0]);
   free((char *) matrix);
}

/**
 *    Oslobadjanje memorije zauzete za matricu kompleksnih vrednosti u
 *    dvostrukoj preciznosti.
 */
void free_complex_matrix(double complex **matrix) {
   free((char *) matrix[0]);
   free((char *) matrix);
}

/**
 *    Oslobadjanje memorije zauzete za tenzor vrednosti u
 *    dvostrukoj preciznosti.
 */
void free_double_tensor(double ***tensor) {
   free((char *) tensor[0][0]);
   free((char *) tensor[0]);
   free((char *) tensor);
}

/**
 *    Oslobadjanje memorije zauzete za tenzor kompleksnih vrednosti u
 *    dvostrukoj preciznosti.
 */
void free_complex_tensor(double complex ***tensor) {
   free((char *) tensor[0][0]);
   free((char *) tensor[0]);
   free((char *) tensor);
}
```

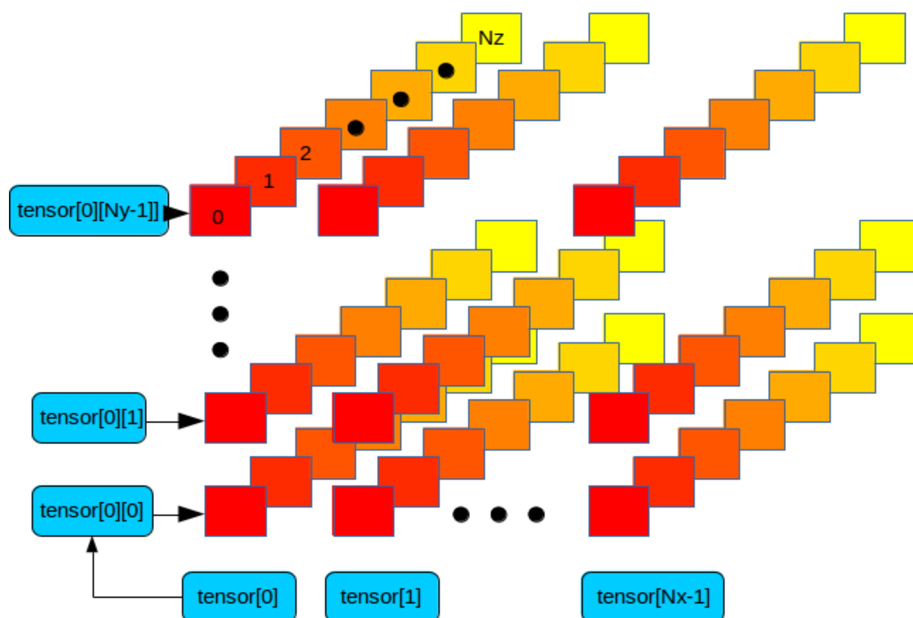
Koristeći listing 2.8 opisaćemo funkcije za alociranje i oslobađanje memorije za strukture podataka u dvostrukoj preciznosti. Isti mehanizmi se koriste za strukture podataka u kompleksnoj dvostrukoj preciznosti i stoga neće biti posebno opisivane. Sledi opis pojedinačnih funkcija fajla mem.c:

- `alloc_double_vector` predstavlja funkciju za alociranje memorije za vektor (jednodimenzionalni niz) vrednosti u dvostrukoj preciznosti. Alociranje se vrši pozivom standardne C funkcije za alociranje memorije `malloc`, pri čemu se vodi računa o grešci u slučaju da nema dovoljno memorije u sistemu. Funkcija vraća pokazivač na alocirani deo memorije ili vrednost `NULL` uz odgovarajuću poruku o grešci u slučaju da alokacija nije uspela.
- `alloc_double_matrix` predstavlja funkciju za alociranje memorije za matricu vrednosti u dvostrukoj preciznosti. Veličina alocirane matrice je `Nx * Ny` vrednosti u dvostrukoj preciznosti. Alociranje se vrši kreiranjem niza pokazivača `matrix` dužine `Nx`. Pojedinačni pokazivači u tom nizu pokazivača pokazuju na vrednosti u dvostrukoj preciznosti, čime se efektivno gradi struktura podataka dvodimenzionalne matrice. Nakon kreiranja niza pokazivača (na pokazivače) `matrix`, vrši se eksplicitno zauzimanje memorije za `Nx * Ny` elemenata tipa `double`, na koje će pokazivati nulti pokazivač u nizu pokazivača `matrix`. Potom se sukscesivno kroz petlju svakom sledećem pokazivaču iz niza pokazivača `matrix` dodeljuje memorija udaljena za `Ny` elemenata tipa `double` od početka memorije na koju pokazuje prethodni pokazivač u nizu pokazivača `matrix`. Na ovaj način se efektivno gradi dvodimenzionalna matrica koja poštuje memorijski raspored uobičajen za jezik C, tzv. („row-major order“), odnosno raspored u kome su susedne memorijske lokacije one koje se nalaze u vrstama dvodimenzionalne matrice (vertikalna dimenzija, ako zamislimo koordinatni sistem vezan za matricu). U slučaju da bilo koji od koraka pri alociranju memorije ne uspe, vraća se vrednost `NULL` uz odgovarajuću poruku o grešci. Ilustracija prethodne strukture podataka i pokazivača data je na slici 2.1, pri čemu su pokazivači obeleženi plavom bojom, a niz elemenata tipa `double` nijansama crvene i žute boje.



Slika 2.1: Alociranje matrice uz pomoć funkcije `allocate_double_matrix`.

- `alloc_double_tensor` predstavlja funkciju za alociranje memorije za tenzor trećeg reda (niz matrica) vrednosti u dvostrukoj preciznosti. Ovakvi tenzori su neophodni kao struktura podataka, jer se vrednosti talasne funkcije beleže u trodimenzionalnom koordinatnom sistemu. Za alociranje tenzora koristi se promenljiva `tensor`. Ona predstavlja trostruki pokazivač na tip `double`. Drugim rečima, tenzor će se formirati uz pomoć pokazivača trećeg reda (pokazivača na pokazivače na pokazivače) na osnovni tip `double`. Inicijalno će se alocirati memorija za `Nx` pokazivača na pokazivače na tip `double`. Potom će se alocirati memorija za `Nx * Ny` pokazivača na tip `double` i nultom elementu iz niza `tensor` će se dodeliti alocirana memorija. Na kraju će se alocirati memorija za `Nx * Ny * Nz` elemenata osnovnog tipa `double` i nultom elementu iz niza `tensor[0]` će se dodeliti alocirana memorija. Na ovaj način efektivno se kreiraju svi neophodni pokazivači i alocira se memorija za svih `Nx * Ny * Nz` elemenata tenzora. Nakon ovoga sledi manuelno podešavanje pokazivača u nizu pokazivača `tensor`. Na početku je podešen samo pokazivač `tensor[0][0]`, koji pokazuje na `Nz` uzastopnih elemenata duž *z* koordinate, a koji odgovaraju vrednostima preostalih koordinata X=0 i Y=0. Pri tome, koordinate označene velikim slovima odgovaraju celobrojnim vrednostima položaja po odgovarajućoj dimenziji u datoj strukturi podataka na računaru. Tako se vrednost X kreće od 0 do `Nx`-1, a X=0 odgovara (bezdimenzionalnoj) vrednosti koordinate *x* = `- Nx * h / 2`. Kako bi se podesili preostali pokazivači po Y osi (npr. pokazivač `tensor[0][1]`), na vrednost prethodnog pokazivača (efektivno na adresu na koju pokazuje) dodaje se vrednost `Nz`, tj. broj elemenata na koji pokazuje prethodni pokazivač. Na ovaj način svi pokazivači za X=0 (od `tensor[0][0]` do `tensor[0][Ny-1]`) će biti formirani i pokazivaće na odgovarajuće `Nz` segmente u memoriji.



Slika 2.2: Alociranje tenzora uz pomoć funkcije `allocate_double_tensor`.

Kada je formiran jedan segment pokazivača na ovakav način, istu proceduru treba primeniti i na preostalih `Nx-2` pokazivača po X osi, od `tensor[1]` do `tensor[Nx-1]`. Pri tome treba imati u vidu da je svaki od pokazivača u nizu od `tensor[0]` do `tensor[Nx-1]` udaljen za `Ny` memorijskih lokacija (`Ny` pokazivača na osnovni tip double) od prethodnog. Radi lakše vizuelizacije prethodnog algoritma data je slika 2.2.

- `free_double_vector` predstavlja funkciju za oslobađanje memorije alocirane za vektor vrednosti u dvostrukoj preciznosti. Operacija oslobađanja se obavlja standardnom C funkcijom `free`.
- `free_double_matrix` predstavlja funkciju za oslobađanje memorije alocirane za matricu vrednosti u dvostrukoj preciznosti. S obzirom da je matrica alocirana u dva koraka, istu dvostepenu proceduru primenjujemo i za oslobađanje memorije, ali u obratnom redosledu. Prvo se oslobađa memorija na koju pokazuje pokazivač `matrix[0]` (pogledati sliku 2.1), tj. memorija zauzeta za sve `double` vrednosti u okviru matrice. Nakon toga može da se oslobodi i memorija alocirana za pokazivače u nizu pokazivača `matrix`. Bitno je osloboditi memoriju na koju pokazuju pokazivači u ovom redosledu, jer u suprotnom može doći do curenja memorije i gubitka podataka.
- `free_double_tensor` predstavlja funkciju za oslobađanje memorije alocirane za tenzor trećeg reda vrednosti u dvostrukoj preciznosti. S obzirom da je tenzor alociran u tri koraka, slično oslobađamo i memoriju, ali u obratnom redosledu. Prvo se oslobađa memorija na koju pokazuje pokazivač `tensor[0][0]` (pogledati sliku 2.2), tj. memorija zauzeta za sve `double` vrednosti u okviru tenzora. Nakon toga se oslobodi i memorija alocirana za pokazivače u nizu pokazivača `tensor[0]`. Tek nakon toga može da se oslobodi memorija koja je alocirana u nizu pokazivača `tensor`. Bitno je osloboditi memoriju na koju pokazuju pokazivači u ovom redosledu, jer u suprotnom može doći do curenja memorije i gubitka podataka.

2.1.8 Fajl `imagtime3d-th.c`

Fajl `imagtime3d-th.c` sadrži izvorni kôd glavnog programa `imagtime3d-th`. U okviru izvornog kôda glavnog programa definisano je sledeće:

- promenljive deklarisane u zaglavlju `imagtime3-th.h`,
- privremene promenljive, koje koristi glavni program,
- funkcija za čitanje ulaznih parametara glavnog programa – `readpar`,
- kôd za upis parametara i kontrolnih promenljvih glavnog programa u izlazni fajl,
- funkcija za inicijalizaciju prostorne mreže, potencijala i inicijalne talasne funkcije - `init`,

- funkcija za generisanje koeficijenata Krenk-Nikolson šeme - `gencoef`,
- funkcija za izračunavanje normalizacije talasne funkcije – `calcnorm`,
- funkcija za izračunavanje hemijskog potencijala i energije – `calcmuen`,
- funkcija za izračunavanje srednje kvadratne vrednosti poluprečnika – `calcrms`,
- glavna programska petlja za izračunavanje propagacije talasne funkcije u imaginarnom vremenu,
- funkcija za vremensku propagaciju u odnosu na H_1 deo Hamiltonijana (članovi bez prostornih izvoda) – `calcnu`,
- funkcija za vremensku propagaciju u odnosu na H_2 deo Hamiltonijana (x-komponenta Laplasijana) - `calclux`,
- funkcija za vremensku propagaciju u odnosu na H_3 deo Hamiltonijana (y-komponenta Laplasijana) – `calcluy`,
- funkcija za vremensku propagaciju u odnosu na H_4 deo Hamiltonijana (z-komponenta Laplasijana) – `calcluz`.

Radi lakšeg raspoznavanja svaki od bitnih segmenata kôda je obeležen posebnim komentarom i biće analiziran zasebno. U listingu 2.9 dat je izvorni kôd glavnog programa `imagtime3d-th.c` bez definicija funkcija.

Listing 2.9: Izvorni kôd fajla `imagtime3d-th.c`.

```
#include "imagtime3d-th.h"

int main(int argc, char **argv) {

 /***********VARIABLES*************/
   FILE *out;
   FILE *file;
   int nthreads;
   char filename[MAX_FILENAME_SIZE];
   long cnti;
   double norm, rms, mu, en;
   double ***psi;
   double **cbeta;
   double ***dpsix, ***dpsiy, ***dpsiz;
   double **tmpxi, **tmpyi, **tmpzi, **tmpxj, **tmpyj, **tmpzj;

  /******************CHECK_INPUT_PARAMS***********************/
   if((argc != 3) || (strcmp(*(argv + 1), "-p") != 0)) {
      fprintf(stderr, "Usage: %s -p <parameterfile> \n", *argv);
      exit(EXIT_FAILURE);
   }

   if(! cfg_init(argv[2])) {
```

```
    fprintf(stderr, "Wrong input parameter file.\n");
    exit(EXIT_FAILURE);
 }

 readpar();


/*********CHECK_NUMBER_OF_THREADS*********/
#pragma omp parallel
    #pragma omp master
       nthreads = omp_get_num_threads();

 /*************ALLOCATE_MEMORY**************/
 x = alloc_double_vector(Nx);
 y = alloc_double_vector(Ny);
 z = alloc_double_vector(Nz);

 x2 = alloc_double_vector(Nx);
 y2 = alloc_double_vector(Ny);
 z2 = alloc_double_vector(Nz);

 pot = alloc_double_tensor(Nx, Ny, Nz);
 psi = alloc_double_tensor(Nx, Ny, Nz);

 dpsix = alloc_double_tensor(Nx, Ny, Nz);
 dpsiy = alloc_double_tensor(Nx, Ny, Nz);
 dpsiz = alloc_double_tensor(Nx, Ny, Nz);

 calphax = alloc_double_vector(Nx - 1);
 calphay = alloc_double_vector(Ny - 1);
 calphaz = alloc_double_vector(Nz - 1);
 cbeta =  alloc_double_matrix(nthreads, MAX(Nx, Ny, Nz) - 1);
 cgammax = alloc_double_vector(Nx - 1);
 cgammay = alloc_double_vector(Ny - 1);
 cgammaz = alloc_double_vector(Nz - 1);

 tmpxi = alloc_double_matrix(nthreads, Nx);
 tmpyi = alloc_double_matrix(nthreads, Ny);
 tmpzi = alloc_double_matrix(nthreads, Nz);
 tmpxj = alloc_double_matrix(nthreads, Nx);
 tmpyj = alloc_double_matrix(nthreads, Ny);
 tmpzj = alloc_double_matrix(nthreads, Nz);

 if(output != NULL) out = fopen(output, "w");
 else out = stdout;

 /******************FORM_OUTPUT_FILE_HEADER***********************/
 fprintf(out, "OPTION = %d\n", opt);
 fprintf(out, "NX = %12li   NY = %12li   NZ = %12li\n", Nx, Ny, Nz);
 fprintf(out, "NPAS = %10li   NRUN = %10li\n", Npas, Nrun);
 fprintf(out, "DX = %8le   DY = %8le   DZ = %8le\n", dx, dy, dz);
 fprintf(out, "DT = %8le\n", dt);
 fprintf(out, "AL = %8le   BL = %8le\n", kappa, lambda);
 fprintf(out, "G0 = %8le\n\n", G0);
 fprintf(out, "        %12s   %12s   %12s   %12s   %12s\n", "norm",
              "mu", "en", "rms", "psi(0,0,0)");
 fflush(out);
```

```
G = 0.;

/********************CALL_INITIAL_FUNCTIONS******************/
init(psi);
gencoef();
calcnorm(&norm, psi, tmpxi, tmpyi, tmpzi);
calcmuen(&mu, &en, psi, dpsix, dpsiy, dpsiz, tmpxi, tmpyi,
         tmpzi, tmpxj, tmpyj, tmpzj);
calcrms(&rms, psi, tmpxi, tmpyi, tmpzi);

/*************************PRINT_INITIAL_VALUES**************************/
fprintf(out, "INIT    %8le   %8le   %8le   %8le   %8le\n",
        norm, mu / par, en / par, rms, psi[Nx2][Ny2][Nz2]);
fflush(out);

/*********************PRINT_INIT_PSI_PROJECTIONS***********************/
if(initout != NULL) {
   sprintf(filename, "%s.x", initout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Nx; cnti += outstpx)
      fprintf(file, "%8le %8le\n", x[cnti], psi[cnti][Ny2][Nz2]);
   fclose(file);

   sprintf(filename, "%s.y", initout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Ny; cnti += outstpy)
      fprintf(file, "%8le %8le\n", y[cnti], psi[Nx2][cnti][Nz2]);
   fclose(file);

   sprintf(filename, "%s.z", initout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Nz; cnti += outstpz)
      fprintf(file, "%8le %8le\n", z[cnti], psi[Nx2][Ny2][cnti]);
   fclose(file);
}

G = par * G0;

/****************PERFORM_NPAS_ITERATIONS*********************/
for(cnti = 0; cnti < Npas; cnti ++) {
   calcnu(psi);
   calclux(psi, cbeta);
   calcluy(psi, cbeta);
   calcluz(psi, cbeta);
   calcnorm(&norm, psi, tmpxi, tmpyi, tmpzi);
}
calcmuen(&mu, &en, psi, dpsix, dpsiy, dpsiz,
         tmpxi, tmpyi, tmpzi, tmpxj, tmpyj, tmpzj);
calcrms(&rms, psi, tmpxi, tmpyi, tmpzi);

/*************************PRINT_NPAS_VALUES****************************/
fprintf(out, "NPAS    %8le   %8le   %8le   %8le   %8le\n",
        norm, mu / par, en / par, rms, psi[Nx2][Ny2][Nz2]);
fflush(out);

/*********************PRINT_NPAS_PSI_PROJECTIONS**********************/
```

```
if(Npasout != NULL) {
   sprintf(filename, "%s.x", Npasout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Nx; cnti += outstpx)
      fprintf(file, "%8le %8le\n", x[cnti], psi[cnti][Ny2][Nz2]);
   fclose(file);

   sprintf(filename, "%s.y", Npasout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Ny; cnti += outstpy)
      fprintf(file, "%8le %8le\n", y[cnti], psi[Nx2][cnti][Nz2]);
   fclose(file);

   sprintf(filename, "%s.z", Npasout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Nz; cnti += outstpz)
      fprintf(file, "%8le %8le\n", z[cnti], psi[Nx2][Ny2][cnti]);
   fclose(file);
}

/***********************PERFORM_NRUN_ITERATIONS************************/
for(cnti = 0; cnti < Nrun; cnti ++) {
   calcnu(psi);
   calclux(psi, cbeta);
   calcluy(psi, cbeta);
   calcluz(psi, cbeta);
   calcnorm(&norm, psi, tmpxi, tmpyi, tmpzi);
}
calcmuen(&mu, &en, psi, dpsix, dpsiy, dpsiz,
         tmpxi, tmpyi, tmpzi, tmpxj, tmpyj, tmpzj);
calcrms(&rms, psi, tmpxi, tmpyi, tmpzi);

/**************************PRINT_NRUN_VALUES***************************/
fprintf(out, "NRUN    %8le   %8le   %8le   %8le   %8le\n",
        norm, mu / par, en / par, rms, psi[Nx2][Ny2][Nz2]);
fflush(out);

/*********************PRINT_NRUN_PSI_PROJECTIONS***********************/
if(Nrunout != NULL) {
   sprintf(filename, "%s.x", Nrunout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Nx; cnti += outstpx)
      fprintf(file, "%8le %8le\n", x[cnti], psi[cnti][Ny2][Nz2]);
   fclose(file);

   sprintf(filename, "%s.y", Nrunout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Ny; cnti += outstpy)
      fprintf(file, "%8le %8le\n", y[cnti], psi[Nx2][cnti][Nz2]);
   fclose(file);

   sprintf(filename, "%s.z", Nrunout);
   file = fopen(filename, "w");
   for(cnti = 0; cnti < Nz; cnti += outstpz)
      fprintf(file, "%8le %8le\n", z[cnti], psi[Nx2][Ny2][cnti]);
   fclose(file);
}
```

```
  if(output != NULL) fclose(out);

  /****************FREE_MEMORY***************/
  free_double_vector(x);
  free_double_vector(y);
  free_double_vector(z);

  free_double_vector(x2);
  free_double_vector(y2);
  free_double_vector(z2);

  free_double_tensor(pot);
  free_double_tensor(psi);

  free_double_tensor(dpsix);
  free_double_tensor(dpsiy);
  free_double_tensor(dpsiz);

  free_double_vector(calphax);
  free_double_vector(calphay);
  free_double_vector(calphaz);
  free_double_matrix(cbeta);
  free_double_vector(cgammax);
  free_double_vector(cgammay);
  free_double_vector(cgammaz);

  free_double_matrix(tmpxi);
  free_double_matrix(tmpyi);
  free_double_matrix(tmpzi);
  free_double_matrix(tmpxj);
  free_double_matrix(tmpyj);
  free_double_matrix(tmpzj);

  return(EXIT_SUCCESS);
}
```

Na početku fajla `imagtime3d-th.c` uključuje se zaglavlje glavnog programa, `imagtime3d-th.h`. Ovo je dovoljno, jer zaglavlje u sebi uključuje sve preostale biblioteke neophodne za rad glavnog programa, kao što je opisano u sekciji 2.1.1. Sledi analiza pojedinih segmenata izvornog kôda fajla `imagtime3d-th.c`:

- `VARIABLES` segment služi za deklaraciju lokalnih promenljvih glavnog programa. Glavni program deklariše sledeće promenljive:
  - `out` - predstavlja pokazivač na glavni izlazni fajl, ukoliko je naziv izlaznog fajla definisan u konfiguracionom fajlu. Ukoliko naziv izlaznog fajla nije definisan, pokazivač `out` će pokazivati na standardni izlaz (`stdout`). Glavne fizičke veličine poput `Nx`, `Dx`, `Dt`, koeficijenata κ i λ, `G0`, norme, hemijskog potencijala, energije, srednjeg kvadratnog radijusa i vrednosti talasne funkcije u (fizičkom) koordinatnom

početku će se upisivati u ovaj fajl (ili na terminal) i to u sledećim trenucima: nakon incijalizacije svih vrednosti, nakon `NPAS` iteracija i nakon konačnih `NRUN` iteracija.

- `file` - predstavlja pokazivač na pomoćne izlazne fajlove. Tokom rada programa ovaj pokazivač će menjati vrednost, jer će pokazivati na nekoliko pomoćnih izlaznih fajlova. Dati fajlovi će u sebi sadržati vrednosti projekcija talasne funkcije po X, Y i Z osama i to nakon inicijalizacije, `NPAS` i `NRUN` konačnih iteracija. Pomoćni izlazni fajlovi su značajni, jer u sebi sadrže vrednosti projekcija talasne funkcije u velikom broju tačaka, što omogućava detaljno praćenje ponašanja BAK tokom vremenske propagacije.
- `nthreads` - predstavlja broj OpenMP niti u programu. Prosleđuje se kao ulazni parametar pri pokretanju programa.
- `filename` - string koji predstavlja osnovu naziva izlaznog fajla za projekcije talasne funkcije. Pored toga, svaka od projekcija će imati odgovarajući sufiks. Na primer, ako je vrednost ovog stringa `imagtime3d`, projekcija na X osu nakon incijalizacije će imati naziv `imagtime3d-initout.x` a projekcija na Y osu nakon `NPAS` iteracija će imati naziv `imagtime3d-npasout.y`.
- `cnti` - brojač različitih petlji u glavnom programu.
- `norm`, `rms`, `mu`, `en` - vrednosti norme, srednjeg kvadratnog radijusa, hemijskog potencijala i energije, respektivno.
- `psi` – tenzor trećeg reda (trodimenzionalna matrica), čiji su pojedinačni elementi realni brojevi u dvostrukoj preciznosti. Predstavlja najznačajniju promenljivu u glavnom programu, jer sadrži vrednosti talasne funkcije BAK u tačkama određenim prostornom mrežom, definisanom ulaznim paramterima.
- `cbeta` - matrica koja sadrži koeficijente Krenk-Nikolson šeme definisane u sekciji 1.5.2.
- `dpsix`, `dpsiy` i `dpsiz` – tenzori trećeg reda za smeštanje raznih privremenih vrednosti.
- `tmpxi`, `tmpyi`, `tmpzi`, `tmpxj`, `tmpyj`, `tmpzj` - matrice za smeštanje privremenih vrednosti.

◆ `CHECK_INPUT_PARAMS` segment započinje proveru ulaznih parametara glavnog programa, tj. da li je naveden naziv ulaznog konfiguracionog fajla koji u sebi sadrži sve bitne vrednosti za dalji rad programa. Naziv konfiguracionog fajla mora biti prosleđen na komandnoj liniji prilikom izvršavanja programa, inače dolazi do prekida rada uz poruku o grešci. Potom sledi poziv funkcije `cfg_init` koja je podrobnije opisana u sekciji 2.1.3, a kojoj se prosleđuje naziv konfiguracionog fajla. Cilj ovog poziva jeste parsiranje konfiguracionog fajla i kreiranje parova ključ-vrednost u kojima će biti smeštena

konfiguraciona podešavanja glavnog programa. Konfiguracioni fajl mora da sadrži definicije svih obaveznih parametara, a može da sadrži i definicije dodatnih parametara, koji nisu obavezni. Nakon učitavanja konfiguracionog fajla, sledi poziv funkcije `readpar`. Cilj poziva ove funkcije jeste direktno učitavanje konfiguracionih parametara glavnog programa iz parova ključ-vrednost u promenljive glavnog programa kao što su npr. `G0` ili `Nx`. Opis ove funkcije biće dat u sekciji 2.1.8.1.

- `CHECK_NUMBER_OF_THREADS` segment inicijalizuje promenljivu `nthreads`, koja predstavlja broj OpenMP niti u programu. Inicijalizacija se obavlja u glavnoj niti paralelnog regiona („master thread").
- `ALLOCATE_MEMORY` segment započinje alokaciju memorije za sledeće promenljive (u zagradi je broj sekcije u kojoj je promenljiva opisana):
    - `x`, `y`, `z` (2.1.1),
    - `x2`, `y2`, `z2` (2.1.1),
    - `pot` (2.1.1), `psi` (2.1.8),
    - `dpsix`, `dpsiy`, `dpsiz` (2.1.8),
    - `calphax`, `calphay`, `calphaz`, `cbeta`, `cgammax`, `cgammay`, `cgammaz` (2.1.1),
    - `tmpxi`, `tmpyi`, `tmpzi`, `tmpxj`, `tmpyj`, `tmpzj` (2.1.8).

    Svaka od prethodnih promenljvih spada u jednu od tri klase: vektor, matrica ili tenzor trećeg reda, a sadrži realne brojeve u dvostrukoj preciznosti. Za njihovu alokaciju koriste se funkcije `alloc_double_vector`, `alloc_double_matrix` i `alloc_double_tensor` opisane u sekciji 2.1.7. Skoro sve promenljive se alociraju u maksimalnoj dužini za odgovarajući broj dimenzija, računajući maksimalnu dužinu `Nx` za dimenziju X, `Ny` za dimenziju Y i `Nz` za dimenziju Z. U tom slučaju `x` predstavlja vektor dužine `Nx`, dok `psi` predstavlja tenzor dužine `Nx * Ny * Nz`. Ipak, nekoliko promenljvih nema ovu pravilnost. Prva takva promenljiva je `cbeta`, matrica veličine `nthreads * (m - 1)`, gde je `m` najveći od brojeva `Nx`, `Ny` i `Nz`. Iz ovoga se da zaključiti da svaka nit mora da ima privatni niz za koeficijente Krenk-Nikolson šeme. Razlog ovakve podele objasnićemo prilikom opisa funkcija `calcux` (sekcija 2.1.8.8), `calcuy` (sekcija 2.1.8.9) i `calcluz` (sekcija 2.1.8.10). Takođe ovo važi i za privremene nizove `tmpxi`, `tmpyi`, `tmpzi`, `tmpxj`, `tmpyj` i `tmpzj` - svaka nit ima svoje privremene privatne nizove.
- `FORM_OUTPUT_FILE_HEADER` segment vrši formiranje zaglavlja izlaznog fajla (ili zaglavlja izlaza koji se ispisuje na konzolu). Izgled izlaznog fajla je dat u listingu 2.10 (zaglavlje je obeleženo zelenom bojom).

Listing 2.10: Izgled izlaznog fajla programa za propagaciju u imaginarnom vremenu.

```
OPTION = 2
NX =          240   NY =          200   NZ =          160
NPAS =       5000   NRUN =        500
DX = 5.000000e-02   DY = 5.000000e-02   DZ = 5.000000e-02
DT = 4.000000e-04
AL = 1.414214e+00   BL = 2.000000e+00
G0 = 4.490700e+01

                 norm            mu            en           rms     psi(0,0,0)
INIT     1.000000e+00  2.207102e+00  2.207102e+00  1.050501e+00  5.495710e-01
NPAS     9.965305e-01  4.344625e+00  3.486230e+00  1.458340e+00  2.887706e-01
NRUN     9.965305e-01  4.344620e+00  3.486230e+00  1.458348e+00  2.887700e-01
```

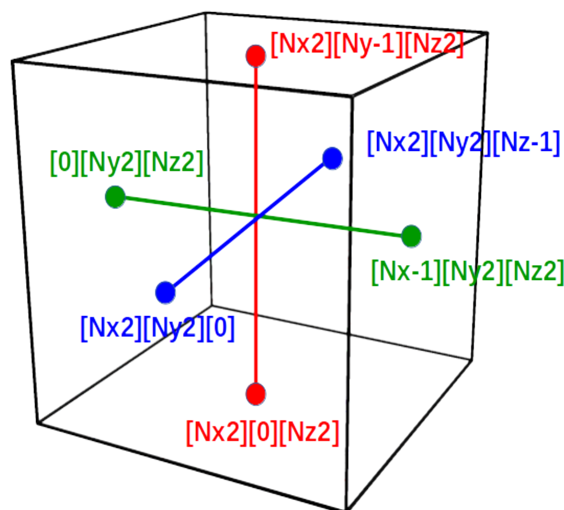
U okviru zaglavlja beleže se sledeće informacije:

- `OPTION` - vrednost opcije za reskaliranje GP jednačine (može biti 1, 2 ili 3),
- `Nx`, `Ny` i `Nz` - broj diskretizacionih tačaka po X, Y i Z osi, respektivno,
- `NPAS`, `NRUN` - brojevi iteracija prilikom propagacije u vremenu,
- `DX`, `DY`, `DZ` - prostorni diskretizacioni koraci u X, Y i Z pravcu, respektivno,
- `DT` - vremenski diskretizacioni korak,
- `AL`, `BL` - κ i λ koeficijenti anizotropnosti zamke
- `G0` - koeficijent nelinearnosti $\mathcal{N}$,
- `norm`, `mu`, `en`, `rms`, `psi(0, 0, 0)` - predstavljaju zaglavlje kolona odgovarajućih fizičkih veličina (norme talasne funkcije, hemijskog potencijala, energije, srednjeg kvadratnog radijusa talasne funkcije i talasne funkcije u koordinatnom početku). U date kolone će tokom izvršavanja programa biti upisivane odgovarajuće fizičke veličine u trenucima nakon: incijalizacije, `NPAS` i `NRUN` iteracija.

◆ `CALL_INITIAL_FUNCTIONS` segment vrši poziv funkcija za inicijalizaciju. U ovom segmentu izvršiće se incijalizacija promenljvih (preko funkcije `init`), generisanje koeficijenata Krenk-Nikolson šeme (`gencoef`), kao i inicijalno izračunavanje norme (`calcnorm`), hemijskog potencijala i energije (`calcmuen`), kao i srednjeg kvadratnog radijusa talasne funkcije (`calcrms`).

◆ `PRINT_INITIAL_VALUES` segment vrši upis prethodno izračunatih inicijalnih vrednosti u izlazni fajl. Primer upisanih inicijalnih vrednosti je dat u listingu 2.10 (obeležen svetlo žutom bojom). Naglašavamo se da se sve vrednosti upisuju u uobičajenom formatu sa mantisom i eksponentom, radi veće preciznosti.

- `PRINT_INIT_PSI_PROJECTIONS` segment vrši upis preseka talasne funkcije `psi` nakon inicijalizacije, i to u tri zasebna fajla. Vizuelna reprezentacija preseka je data na slici 2.3, koja pokazuje tačke prostorne mreže na kojima se posmatraju vrednosti talasne funkcije za dati presek. Tri zasebna fajla su neophodna jer su u pitanju preseci po sve tri ose koordinatnog sistema. Za dati presek se posmatraju vrednosti talasne funkcije duž odgovarajuće ose, dok su preostale dve koordinate fiksirane tako da u fizičkom koordinatnom sistemu imaju vrednost nula, što odgovara polovini diskretizacionog opsega za odgovarajuću osu u strukturi `psi` na računaru. Za to smo uveli odgovarajuće promenljive, `Nx2 = Nx / 2`, `Ny2 = Ny / 2` i `Nz2 = Nz / 2`. Na ovaj način se može pratiti vremenska evolucija talasne funkcije bez potrebe da se upisuju vrednosti talasne funkcije u svim tačkama, jer to može da predstavlja upisivanje velike količine podataka na hard disk, što značajno usporava izvršavanje programa. Na primer, u slučaju matrice veličine $240 \times 200 \times 160$ tačaka biće zabeležena sledeća količina podataka:
  - 240 vrednosti talasne funkcije u dvostrukoj preciznosti u fajlu `imagtime3d-initout.x`,
  - 200 vrednosti talasne funkcije u dvostrukoj preciznosti u fajlu `imagtime3d-initout.y`,
  - 160 vrednosti talasne funkcije u dvostrukoj preciznosti u fajlu `imagtime3d-initout.z`.



Slika 2.3: Preseci talasne funkcije se posmatraju duž odgovarajuće ose X, Y i Z. Na primer, presek duž X ose sadrži vrednosti talasne funkcije `psi[i][Ny2][Nz2]`, *gde brojač* `i` *uzima* `Nx` *vrednosti, od* 0 *do* `Nx - 1`.

- `PERFORM_NPAS_ITERATIONS` predstavlja segment koda sa jednom od dve glavne programske petlje. U okviru ovog segmenta izvršiće se `NPAS` iteracija (tipično, oko 5000 iteracija za sistem veličine $240 \times 200 \times 160$ tačaka), tokom kojih će se vršiti vremenska propagacija sistema. Propagacija se sprovodi pozivom sledećih funkcija:

- funkcija za vremensku propagaciju u odnosu na H_1 deo Hamiltonijana (članovi bez prostornih izvoda) – `calcnu`,
- funkcija za vremensku propagaciju u odnosu na H_2 deo Hamiltonijana (x-komponenta Laplasijana) - `calclux`,
- funkcija za vremensku propagaciju u odnosu na H_3 deo Hamiltonijana (y-komponenta Laplasijana) – `calcluy`,
- funkcija za vremensku propagaciju u odnosu na H_4 deo Hamiltonijana (z-komponenta Laplasijana) – `calcluz`,
- funkcija za računanje norme talasne funkcije sistema – `calcnorm`.

Nakon propagacije sistema neophodno je izračunati hemijski potencijal i energiju sistema što se vrši pozivom funkcije `calcmuen`. Takođe računa se i srednji kvadratni radijus talasne funkcije uz pomoć funkcije `calcrms`. Sve prethodno navedene funkcije će biti detaljnije opisane u nastavku poglavlja.

- Naredni segmenti koda su ekvivalentni prethodno opisanim segmentima koda, i stoga neće biti posebno opisivani. Jedina razlika je u tome što se koristi `NPAS` ili `NRUN` iteracija od prethodnog poziva istovetnog segmenta koda. Ovakvo odvijanje programa ima za cilj konvergenciju sistema ka stacionarnom stanju tokom `NPAS` iteracija i verifikaciju da je konvergencija dostignuta tako što ćemo proveriti da se nakon dodatnih `NRUN` iteracija vrednosti prethodno izračunatih fizičkih veličina ne menjaju. Sa leve strane je dat naziv novog segmenta koda, a sa desne strane naziv ekvivalentnog segmenta koda koji je već opisan:
  - `PRINT_NPAS_VALUES` (vrednosti ispisane narandžastom bojom u listingu 2.10) – `PRINT_INITIAL_VALUES`,
  - `PRINT_NPAS_PSI_PROJECTIONS - PRINT_INIT_PSI_PROJECTIONS`,
  - `PERFORM_NRUN_ITERATIONS - PERFORM_NPAS_ITERATIONS`,
  - `PRINT_NRUN_VALUES` (vrednosti ispisane crvenom bojom u listingu 2.10) – `PRINT_INITIAL_VALUES`,
  - `PRINT_NRUN_PSI_PROJECTIONS – PRINT_INIT_PSI_PROJECTIONS`.
- `FREE_MEMORY` segment započinje oslobađanje memorije za sve promenljive zauzete u segmentu koda `ALLOCATE_MEMORY`. Memorija se oslobađa funkcijama opisanim u poglavlju 2.1.7.

2.1.8.1 Funkcija `readpar`

Cilj poziva ove funkcije jeste direktno učitavanje konfiguracionih parametara glavnog programa iz parova ključ-vrednost u promenljive glavnog programa kao što su npr. `G0` ili `Nx`. U listingu 2.11 dat je izvorni kôd funkcije `readpar`.

Listing 2.11: Izvorni kôd funkcije `readpar`.

```
void readpar(void) {
   char *cfg_tmp;

   if((cfg_tmp = cfg_read("OPTION")) == NULL) {
      fprintf(stderr, "OPTION is not defined in the configuration file\n");
      exit(EXIT_FAILURE);
   }
   opt = atol(cfg_tmp);

   if((cfg_tmp = cfg_read("G0")) == NULL) {
      fprintf(stderr, "G0 is not defined in the configuration file.\n");
      exit(EXIT_FAILURE);
   }
   G0 = atof(cfg_tmp);

   if((cfg_tmp = cfg_read("NX")) == NULL) {
      fprintf(stderr, "NX is not defined in the configuration file.\n");
      exit(EXIT_FAILURE);
   }
   Nx = atol(cfg_tmp);

   if((cfg_tmp = cfg_read("NY")) == NULL) {
      fprintf(stderr, "NY is not defined in the configuration file.\n");
      exit(EXIT_FAILURE);
   }
   Ny = atol(cfg_tmp);

   if((cfg_tmp = cfg_read("NZ")) == NULL) {
      fprintf(stderr, "Nz is not defined in the configuration file.\n");
      exit(EXIT_FAILURE);
   }
   Nz = atol(cfg_tmp);

   if((cfg_tmp = cfg_read("DX")) == NULL) {
      fprintf(stderr, "DX is not defined in the configuration file.\n");
      exit(EXIT_FAILURE);
   }
   dx = atof(cfg_tmp);

   if((cfg_tmp = cfg_read("DY")) == NULL) {
      fprintf(stderr, "DY is not defined in the configuration file.\n");
      exit(EXIT_FAILURE);
   }
```

```
    dy = atof(cfg_tmp);

    if((cfg_tmp = cfg_read("DZ")) == NULL) {
        fprintf(stderr, "DZ is not defined in the configuration file.\n");
        exit(EXIT_FAILURE);
    }
    dz = atof(cfg_tmp);

    if((cfg_tmp = cfg_read("DT")) == NULL) {
        fprintf(stderr, "DT is not defined in the configuration file.\n");
        exit(EXIT_FAILURE);
    }
    dt = atof(cfg_tmp);

    if((cfg_tmp = cfg_read("AL")) == NULL) {
        fprintf(stderr, "AL is not defined in the configuration file.\n");
        exit(EXIT_FAILURE);
    }
    kappa = atof(cfg_tmp);

    if((cfg_tmp = cfg_read("BL")) == NULL) {
        fprintf(stderr, "BL is not defined in the configuration file.\n");
        exit(EXIT_FAILURE);
    }
    lambda = atof(cfg_tmp);

    if((cfg_tmp = cfg_read("NPAS")) == NULL) {
        fprintf(stderr, "NPAS is not defined in the configuration file.\n");
        exit(EXIT_FAILURE);
    }
    Npas = atol(cfg_tmp);

    if((cfg_tmp = cfg_read("NRUN")) == NULL) {
        fprintf(stderr, "NRUN is not defined in the configuration file.\n");
        exit(EXIT_FAILURE);
    }
    Nrun = atol(cfg_tmp);

    output = cfg_read("OUTPUT");
    initout = cfg_read("INITOUT");
    Npasout = cfg_read("NPASOUT");
    Nrunout = cfg_read("NRUNOUT");

    if((initout != NULL) || (Npasout != NULL) || (Nrunout != NULL)) {
        if((cfg_tmp = cfg_read("OUTSTPX")) == NULL) {
            fprintf(stderr, "OUTSTPX is not defined in
                             the configuration file.\n");
            exit(EXIT_FAILURE);
        }
        outstpx = atol(cfg_tmp);

        if((cfg_tmp = cfg_read("OUTSTPY")) == NULL) {
            fprintf(stderr, "OUTSTPY is not defined in
                             the configuration file.\n");
            exit(EXIT_FAILURE);
        }
        outstpy = atol(cfg_tmp);
```

```
        if((cfg_tmp = cfg_read("OUTSTPZ")) == NULL) {
            fprintf(stderr, "OUTSTPZ is not defined in
                             the configuration file.\n");
            exit(EXIT_FAILURE);
        }
        outstpz = atol(cfg_tmp);
    }

    return;
}
```

Pre poziva funkcije `readpar` mora biti pozvana funkcija `cfg_init`, opisana u sekciji 2.1.3. Poziv date funkcije stvara niz parova ključ-vrednost (mapu) iz koje se mogu čitati pojedinačni parametri konfiguracionog fajla datog u listingu 2.4. Čitanje parametara se vrši pozivima funkcije `cfg_read`, kojima se prosleđuju parametri (ključevi) koji se čitaju (`OPTION`, `G0`, `NX`, `NY`, `NZ`, itd). Sve pročitane brojčane vrednosti pre korišćenja moraju biti konvertovane iz znakovnog u celobrojni ili realni broj pozivima funkcija `atol` i `atof`. Ukoliko neki od predefinisanih obaveznih parametara nije u konfiguracionom fajlu, izbacuje se poruka o grešci i završava se rad programa. Ovo proizilazi iz činjenice da uspešno izvršavanje numeričkog algoritma zahteva većinu konfiguracionih parametra definisanih u konfiguracionom fajlu.

2.1.8.2 Funkcija `init`

U listingu 2.12 nalazi se izvorni kôd funkcije `init`.

Listing 2.12: Izvorni kôd funkcije `init`.

```
void init(double ***psi) {
    long cnti, cntj, cntk;
    double kappa2, lambda2;
    double pi, cpsi1, cpsi2;
    double tmp;

    if (opt == 2) par = 2.;
    else par = 1.;

    kappa2 = kappa * kappa;
    lambda2 = lambda * lambda;

    Nx2 = Nx / 2; Ny2 = Ny / 2; Nz2 = Nz / 2;
    dx2 = dx * dx; dy2 = dy * dy; dz2 = dz * dz;

    pi = 4. * atan(1.);
    cpsi1 = sqrt(pi * sqrt(pi / (kappa * lambda)));
    cpsi2 = cpsi1 * sqrt(2. * sqrt(2.));

    for(cnti = 0; cnti < Nx; cnti ++) {
```

```
      x[cnti] = (cnti - Nx2) * dx;
      x2[cnti] = x[cnti] * x[cnti];
      for(cntj = 0; cntj < Ny; cntj ++) {
         y[cntj] = (cntj - Ny2) * dy;
         y2[cntj] = y[cntj] * y[cntj];
         for(cntk = 0; cntk < Nz; cntk ++) {
            z[cntk] = (cntk - Nz2) * dz;
            z2[cntk] = z[cntk] * z[cntk];
            if(opt == 3) {
               pot[cnti][cntj][cntk] = 0.25 * (x2[cnti] + kappa2 * y2[cntj]
                                            + lambda2 * z2[cntk]);
            tmp = exp(- 0.25 * (x2[cnti] + kappa * y2[cntj]
                                 + lambda * z2[cntk]));
                psi[cnti][cntj][cntk] = tmp / cpsi2;
            } else {
               pot[cnti][cntj][cntk] = x2[cnti] + kappa2 * y2[cntj]
                                                + lambda2 * z2[cntk];
           tmp = exp(- 0.5 * (x2[cnti] + kappa * y2[cntj]
                                        + lambda * z2[cntk]));
                psi[cnti][cntj][cntk] = tmp / cpsi1;
            }
         }
      }
   }

   return;
}
```

Funkcija `init` služi za inicijalizaciju prostorne mreže tačaka (nizovi `x`, `y`, `z`, `x2`, `y2` i `z2`), potencijala (`pot`), kao i inicijalne vrednosti talasne funkcije `psi`. Osim toga, postoji i opcija skaliranja vrednosti potencijala i talasne funkcije u zavisnosti od konfiguracionog parametra `opt`, što odgovara izboru jednog od reskaliranja GP jednačine iz sekcije 1.3.

2.1.8.3 Funkcija `gencoef`

U listingu 2.13 nalazi se izvorni kôd funkcije `gencoef`.

Listing 2.13: Izvorni kôd funkcije `gencoef`.

```
void gencoef(void) {
   long cnti;

   Ax0 = 1. + dt / dx2;
   Ay0 = 1. + dt / dy2;
   Az0 = 1. + dt / dz2;

   Ax0r = 1. - dt / dx2;
   Ay0r = 1. - dt / dy2;
```

```
   Az0r = 1. - dt / dz2;

   Ax = - 0.5 * dt / dx2;
   Ay = - 0.5 * dt / dy2;
   Az = - 0.5 * dt / dz2;

   calphax[Nx - 2] = 0.;
   cgammax[Nx - 2] = - 1. / Ax0;
   for (cnti = Nx - 2; cnti > 0; cnti --) {
      calphax[cnti - 1] = Ax * cgammax[cnti];
      cgammax[cnti - 1] = - 1. / (Ax0 + Ax * calphax[cnti - 1]);
   }

   calphay[Ny - 2] = 0.;
   cgammay[Ny - 2] = - 1. / Ay0;
   for (cnti = Ny - 2; cnti > 0; cnti --) {
      calphay[cnti - 1] = Ay * cgammay[cnti];
      cgammay[cnti - 1] = - 1. / (Ay0 + Ay * calphay[cnti - 1]);
   }

   calphaz[Nz - 2] = 0.;
   cgammaz[Nz - 2] = - 1. / Az0;
   for (cnti = Nz - 2; cnti > 0; cnti --) {
      calphaz[cnti - 1] = Az * cgammaz[cnti];
      cgammaz[cnti - 1] = - 1. / (Az0 + Az * calphaz[cnti - 1]);
   }

   return;
}
```

Funkcija `gencoef` služi za generisanje koeficijenata Krenk-Nikolson šeme predstavljenih u poglavlju 1.5.2. To se prevashodno odnosi na nizove α i γ, koji se koriste u rekurzivnim relacijama (1.41). Dati koeficijenti se generišu za sve tri ose koordinatnog sistema.

2.1.8.4 Funkcija `calcnorm`

U listingu 2.14 nalazi se izvorni kôd funkcije `calcnorm`.

Listing 2.14: Izvorni kôd funkcije `calcnorm`.

```
void  calcnorm(double  *norm,  double  ***psi,  double  **tmpx,  double  **tmpy,
double **tmpz) {
   int threadid;
   long cnti, cntj, cntk;

   #pragma omp parallel private(threadid, cnti, cntj, cntk)
   {
      threadid = omp_get_thread_num();
```

```
      #pragma omp for
      for(cnti = 0; cnti < Nx; cnti ++) {
         for(cntj = 0; cntj < Ny; cntj ++) {
            for(cntk = 0; cntk < Nz; cntk ++) {
               tmpz[threadid][cntk] = psi[cnti][cntj][cntk] *
                                       psi[cnti][cntj][cntk];
            }
            tmpy[threadid][cntj] = simpint(dz, tmpz[threadid], Nz);
         }
         tmpx[0][cnti] = simpint(dy, tmpy[threadid], Ny);
      }
      #pragma omp barrier

      #pragma omp single
      *norm = sqrt(simpint(dx, tmpx[0], Nx));

      #pragma omp for
      for(cnti = 0; cnti < Nx; cnti ++) {
         for(cntj = 0; cntj < Ny; cntj ++) {
            for(cntk = 0; cntk < Nz; cntk ++) {
               psi[cnti][cntj][cntk] /= *norm;
            }
         }
      }
   }

   return;
}
```

Funkcija `calcnorm` izračunava normu talasne funkcije i normalizuje talasnu funkciju `psi` koristeći izračunatu vrednost, tako da je nakon toga norma jednaka jedinici, u skladu sa normalizacionim uslovom talasne funkcije (1.15). S obzirom da normalizacija sadrži tri koncentrične petlje, izvršavanje spoljašnje petlje (u ovom slučaju po X osi) se može paralelizovati. Na ovaj način čitav X opseg će biti podeljen na `Nx / nthreads` segmenata, koji će se izvršavati nezavisno na različitim jezgrima računarskog procesoral, što dovodi do ubrzanja izračunavanja normalizacije. Čitava normalizacija se izvršava u paralelnom regionu koji omogućava formiranje paralelnih petlji, ali i privatizaciju lokalnih promenljivih za svaku nit pojedinačno. U konkretnom slučaju privatizovane su promenljive `threadid`, `cnti`, `cntj` i `cntk`. Prva predstavlja identifikator niti, dok su naredne tri promenljive brojači petlji koje se nazivaju isto, ali su nezavisne od niti do niti. Treba obratiti pažnju da promenljiva `cnti` uzima različit opseg od niti do niti. Npr. ona za nit 0 uzima opseg od `0` do `Nx / nthreads - 1`, za nit 1 uzima opseg od `Nx / nthreads` do `2 * Nx / nthreads – 1,` itd.

Normalizacija talasne funkcije podrazumeva kvadriranje vrednosti talasne funkcije u skladu sa izrazom (1.15) i smeštanje tih vrednosti u elemente privremenog niza `tmpz[threadid][cntk]`. U ovom postupku svaka nit poseduje svoj niz `tmpz`, što nitima omogućava odvojeno pristupanje tim privatnim nizovima, bez mogućnosti da dođe do štetnog preplitanja niti („race condition“). Nakon završetka rada unutrašnje petlje, ceo Z opseg (niz `tmpz`)

se prosleđuje funkciji simpint (sekcija 2.1.5) kako bi se izračunao integral kvadrata talasne funkcije po Z dimenziji za fiksnu vrednost koordinata X i Y. Nakon izračunavanja integrala po Z osi data vrednost se upisuje u element niza tmpy[threadid][cntj]. Dati postupak se ponavlja za čitav opseg Y, pa tako na kraju petlje u sredini niz tmpy poseduje vrednosti integrala kvadrata talasne funkcije po Z osi za svaki element Y ose, pri čemu je vrednost po X osi fiksirana.

Slično ovom postupku počevši od unutrašnje ka srednjoj petlji, izračunavanje nastavljamo od srednje ka spoljnoj petlji. Kada se kompletira izračunavanje niza tmpy, vrši se njegovo integraljenje funkcijom simpint po Y osi. Na ovaj način element niza tmpx[0][cnti] će sadržati integral kvadrata talasne funkcije po Y i Z osi, čime se nastavlja implementacija formule (1.15). Preostalo je još da se kompletira niz tmpx[0], što će se desiti kada sve niti završe izračunavanje u okviru svojih podopsega. Da bi se sinhronizovalo kompletiranje niza tmpx od strane različitih niti pre nego što se krene u finalno integraljenje po X osi, neophodno je postaviti OpenMP barijeru, na kojoj će se zaustaviti sve niti kada završe svoj deo posla. Tek kada sve niti završe svoj posao moguć je nastavak rada programa. OpenMP barijera se postavlja pretprocesorskom pragmom #pragma omp barrier.

Kada su sve niti završile svoj deo posla, preostalo je integraljenje po X osi. S obzirom da je broj elemenata niza tmpx dovoljno mali (240 u osnovnoj konfiguraciji), ovaj algoritam izvršavamo sekvencijalno, tako da integraljenje po X opsegu vrši samo jedna nit. Ovo se postiže OpenMP pretprocesorskom direktivom #pragma omp single, kada će jedna od niti iz paralelnog regiona izvršiti kôd koji se nalazi ispod date pretprocesorske direktive.

Integraljenjem po X opsegu izračunava se ukupna norma talasne funkcije psi. Preostaje da se do kraja funkcije u trostrukoj paralelnoj petlji izvrši deljenje postojećih vrednosti talasne funkcije psi sa vrednošću dobijene norme norm. Na ovaj način efektivno se postiže normalizacija talasne funkcije u skladu sa uslovom (1.15).

2.1.8.5 Funkcija calcmuen

U listingu 2.15 nalazi se izvorni kôd funkcije calcmuen.

Listing 2.15: Izvorni kôd funkcije calcmuen.

```
void calcmuen(double *mu, double *en, double ***psi, double ***dpsix, double
***dpsiy, double ***dpsiz, double **tmpxi, double **tmpyi, double **tmpzi,
double **tmpxj, double **tmpyj, double **tmpzj) {
   int threadid;
   long cnti, cntj, cntk;
   double psi2, psi2lin, dpsi2;

   #pragma omp parallel private(threadid, cnti, cntj, cntk,
```

```
                              psi2, psi2lin, dpsi2)
{
   threadid = omp_get_thread_num();

   #pragma omp for
   for(cntj = 0; cntj < Ny; cntj ++) {
      for(cntk = 0; cntk < Nz; cntk ++) {
         for(cnti = 0; cnti < Nx; cnti ++) {
            tmpxi[threadid][cnti] = psi[cnti][cntj][cntk];
         }
         diff(dx, tmpxi[threadid], tmpxj[threadid], Nx);
         for(cnti = 0; cnti < Nx; cnti ++) {
            dpsix[cnti][cntj][cntk] = tmpxj[threadid][cnti];
         }
      }
   }

   #pragma omp for
   for(cnti = 0; cnti < Nx; cnti ++) {
      for(cntk = 0; cntk < Nz; cntk ++) {
         for(cntj = 0; cntj < Ny; cntj ++) {
            tmpyi[threadid][cntj] = psi[cnti][cntj][cntk];
         }
         diff(dy, tmpyi[threadid], tmpyj[threadid], Ny);
         for(cntj = 0; cntj < Ny; cntj ++) {
            dpsiy[cnti][cntj][cntk] = tmpyj[threadid][cntj];
         }
      }
   }

   #pragma omp for
   for(cnti = 0; cnti < Nx; cnti ++) {
      for(cntj = 0; cntj < Ny; cntj ++) {
         for(cntk = 0; cntk < Nz; cntk ++) {
            tmpzi[threadid][cntk] = psi[cnti][cntj][cntk];
         }
         diff(dz, tmpzi[threadid], tmpzj[threadid], Nz);
         for(cntk = 0; cntk < Nz; cntk ++) {
            dpsiz[cnti][cntj][cntk] = tmpzj[threadid][cntk];
         }
      }
   }
   #pragma omp barrier

   #pragma omp for
   for(cnti = 0; cnti < Nx; cnti ++) {
      for(cntj = 0; cntj < Ny; cntj ++) {
         for(cntk = 0; cntk < Nz; cntk ++) {
            psi2 = psi[cnti][cntj][cntk] * psi[cnti][cntj][cntk];
            psi2lin = psi2 * G;
            dpsi2 = dpsix[cnti][cntj][cntk] * dpsix[cnti][cntj][cntk] +
                    dpsiy[cnti][cntj][cntk] * dpsiy[cnti][cntj][cntk] +
                    dpsiz[cnti][cntj][cntk] * dpsiz[cnti][cntj][cntk];
            tmpzi[threadid][cntk] = (pot[cnti][cntj][cntk] + psi2lin)
                                     * psi2 + dpsi2;
            tmpzj[threadid][cntk] = (pot[cnti][cntj][cntk] + 0.5 * psi2lin)
                                     * psi2 + dpsi2;
```

```
            }
            tmpyi[threadid][cntj] = simpint(dz, tmpzi[threadid], Nz);
            tmpyj[threadid][cntj] = simpint(dz, tmpzj[threadid], Nz);
         }
         tmpxi[0][cnti] = simpint(dy, tmpyi[threadid], Ny);
         tmpxj[0][cnti] = simpint(dy, tmpyj[threadid], Ny);
      }
   }

   *mu = simpint(dx, tmpxi[0], Nx);
   *en = simpint(dx, tmpxj[0], Nx);

   return;
}
```

Funkcija `calcmuen` izračunava hemijski potencijal i energiju sistema u skladu sa formulama opisanim u sekciji 1.6.1. Kao i u prethodnoj funkciji `calcnorm` (sekcija 2.1.8.4), i ovde se vrši paralelizacija izvršavanja petlji uz pomoć OpenMP niti. Privatne promenljive svake od niti su `threadid`, `cnti`, `cntj`, `cntk`, `psi2`, `psi2lin` i `dpsi2`. Prve četiri promenljive imaju istu svrhu kao i u funkciji `calcnorm`, dok `psi2`, `psi2lin` i `dpsi2` predstavljaju lokalne promenljive niti, koje se koriste za izračunavanje podintegralnih vrednosti unutar formula za hemijski potencijal i energiju.

Za razliku od funkcije `calcnorm`, u funkciji `calcmuen` prvo se računaju neophodni parcijalni izvodi po prostornim promenljivama talasne funkcije `psi` i to po sve tri ose u tri zasebne OpenMP paralelizovane petlje. Izvodi se izračunavaju pozivom funkcije za izračunavanje Ričardsonove ekstrapolacije `diff` (sekcija 2.1.5). Parcijalni izvod u tri dimenzije se izračunava ekvivalentno izračunavanju prostornog izvoda u jednoj dimenziji, na osnovu formule (2.3). Treba obratiti pažnju da svaka nit poseduje svoje nizove `tmpxi`, `tmpyi` i `tmpzi`, što im omogućava da paralelno izračunavaju izvode i upisuju ih u odgovarajuće delove nizova `dpsix`, `dpsiy` i `dpsiz`, pri čemu zadnje slovo u nazivu prethodna tri niza predstavlja osu po kojoj je formiran parcijalni izvod.

Nakon paralelnih petlji za izračunavanje parcijalnih izvoda, sledi OpenMP barijera do koje moraju doći sve niti koje su izvršavale prethodne petlje pre nego što izračunavanje može da se nastavi. Potom sledi drugi deo funkcije, odnosno izračunavanje hemijskog potencijala i energije. U ovom delu vrši se integraljenje celokupnog izraza, u skladu sa formulama (1.44) i (1.45). Integraljenje se vrši paralelno, pri čemu različite niti obrađuju delove X opsega, slično kao u funkciji `calcnorm`. Podintegralne vrednosti se računaju u unutrašnjoj petlji po Z opsegu, u skladu sa ovim formulama.

Nakon izračunavanja integrala po Z i Y osi, hemijski potencijal i energija se računaju kao integrali nizova `tmpxi[0]` i `tmpxj[0]` po X opsegu.

2.1.8.6 Funkcija `calcrms`

U listingu 2.16 nalazi se izvorni kôd funkcije `calcrms`.

Listing 2.16: Izvorni kôd funkcije `calcrms`.

```
void calcrms(double *rms, double ***psi, double **tmpx, double **tmpy, double
**tmpz) {
   int threadid;
   long cnti, cntj, cntk;
   double psi2;

   #pragma omp parallel private(threadid, cnti, cntj, cntk, psi2)
   {
      threadid = omp_get_thread_num();

      #pragma omp for
      for(cnti = 0; cnti < Nx; cnti ++) {
         for(cntj = 0; cntj < Ny; cntj ++) {
            for(cntk = 0; cntk < Nz; cntk ++) {
               psi2 = psi[cnti][cntj][cntk] * psi[cnti][cntj][cntk];
               tmpz[threadid][cntk] = (x2[cnti] + y2[cntj] + z2[cntk]) * psi2;
            }
            tmpy[threadid][cntj] = simpint(dz, tmpz[threadid], Nz);
         }
         tmpx[0][cnti] = simpint(dy, tmpy[threadid], Ny);
      }
   }

   *rms = sqrt(simpint(dx, tmpx[0], Nx));

   return;
}
```

Kôd funkcije za izračunavanje srednjeg kvadratnog radijusa talasne funkcije veoma je sličan prvom delu kôda funkcije `calcnorm` (sekcija 2.1.8.4). Razlika između ove dve funkcije je u tome što se elementi niza `tmpz` u slučaju funkcije `calcrms` ne računaju kao prosti kvadrati vrednosti talasne funkcije, već se kvadrati vrednosti talasne funkcije množe zbirom kvadrata prostornih koordinata po sve tri ose (nizovi `x2`, `y2` i `z2`). Ostatak funkcije svodi se na integraljenje redom po Z, Y i X osi, analogno funkciji `calcnorm`, dok konačni rezultat predstavlja kvadratni koren vrednosti integrala po X osi, kao što je opisano u sekciji 1.6.2.

2.1.8.7 Funkcija `calcnu`

U listingu 2.17 nalazi se izvorni kôd funkcije `calcnu`.

Listing 2.17: Izvorni kôd funkcije `calcnu`.

```
void calcnu(double ***psi) {
   long cnti, cntj, cntk;
   double psi2, psi2lin, tmp;

   #pragma omp parallel for private(cnti, cntj, cntk, psi2, psi2lin, tmp)
   for(cnti = 0; cnti < Nx; cnti ++) {
      for(cntj = 0; cntj < Ny; cntj ++) {
         for(cntk = 0; cntk < Nz; cntk ++) {
            psi2 = psi[cnti][cntj][cntk] * psi[cnti][cntj][cntk];
            psi2lin = psi2 * G;
            tmp = dt * (pot[cnti][cntj][cntk] + psi2lin);
            psi[cnti][cntj][cntk] *= exp(- tmp);
         }
      }
   }

   return;
}
```

Funkcija `calcnu` predstavlja funkciju za računanje vremenske propagacije u odnosu na deo Hamiltonijana bez prostornih izvoda H_1, u skladu sa formulom (1.30). Funkcija `calcnu` predstavlja primer problema koji je idealan za paralelizaciju („embarrassingly parallel problem"). Ovo podrazumeva da su računske operacije unutar ugnježdenih petlji nezavisne i da nema potrebe za čekanjem da se određeni podatak izračuna, odnosno da ne postoji međuzavisnost unutar različitih iteracija petlji. U konkretnom slučaju to znači da je spoljna petlja po X osi paralelizovana uz pomoć OpenMP niti, a da se računi unutar druge dve petlje odvijaju na odgovarajućim segmentima X opsega potpuno nezavisno jedan od drugog. Takođe nije neophodna ni barijerna sinhronizacija unutar same funkcije. Merenja pokazuju najveće ubrzanje paralelizovanih funkcija ovakvog tipa u odnosu na sekvencijalnu verziju funkcija implementiranu u Fortran programu [10].

2.1.8.8 Funkcija `calclux`

U listingu 2.18 nalazi se izvorni kôd funkcije `calclux`.

Listing 2.18: Izvorni kôd funkcije `calclux`.

```
void calclux(double ***psi, double **cbeta) {
   int threadid;
   long cnti, cntj, cntk;
   double c;

   #pragma omp parallel private(threadid, cnti, cntj, cntk, c)
```

```
   {
      threadid = omp_get_thread_num();

      #pragma omp for
      for(cntj = 0; cntj < Ny; cntj ++) {
         for(cntk = 0; cntk < Nz; cntk ++) {
            cbeta[threadid][Nx - 2] = psi[Nx - 1][cntj][cntk];
            for (cnti = Nx - 2; cnti > 0; cnti --) {
               c = - Ax * psi[cnti + 1][cntj][cntk] +
                     Ax0r * psi[cnti][cntj][cntk] -
                     Ax * psi[cnti - 1][cntj][cntk];
               cbeta[threadid][cnti - 1] =
                     cgammax[cnti] * (Ax * cbeta[threadid][cnti] - c);
            }
            psi[0][cntj][cntk] = 0.;
            for (cnti = 0; cnti < Nx - 2; cnti ++) {
               psi[cnti + 1][cntj][cntk] =
                     calphax[cnti] * psi[cnti][cntj][cntk] +
                     cbeta[threadid][cnti];
            }
            psi[Nx - 1][cntj][cntk] = 0.;
         }
      }
   }

   return;
}
```

Funkcija `calclux` predstavlja funkciju za računanje vremenske propagacije u odnosu na H_2 deo Hamiltonijana (x-komponenta Laplasijana) pomoću Krenk-Nikolson algoritma opisanog u sekciji 1.5.2 na primeru propagacije u realnom vremenu u 1D slučaju. Funkcije `calclux`, `calcluy` i `calcluz` predstavljaju osnovu Krenk-Nikolson algoritma kako u imaginarnom, tako i u realnom vremenu. Vremenska propagacija u odnosu na H_2 deo Hamiltonijana se obavlja uz pomoć trostruke petlje, slično prethodno opisanim funkcijama (posledica trodimenzionalnosti prostora). Spoljna petlja (u ovom slučaju po Y osi) je ponovo paralelizovana uz pomoć OpenMP niti, dok se unutrašnje petlje izvršavaju sekvencijalno, unutar odgovarajuće niti. Svaka nit poseduje svoje privatne promenljive `threadid`, `cnti`, `cntj` i `cntk`, kao i promenljivu `c` i niz koeficijenata `cbeta`, formula (1.41).

U okviru funkcije `calclux` bitno je uočiti da se unutrašnja petlja obavlja po čitavom X opsegu. Ova činjenica će biti od naročitog značaja u nastavku teze, kada bude opisivana distribucija podataka u okviru algoritma paralelnog transponovanja podataka. Izvršavanje unutrašnjih petlji u funkciji `calclux` je inherentno sekvencijalno, zbog postojanja međuzavisnosti iteracija u okviru unutrašnjih petlji. Ovo uzrokuje da direktna paralelizacija unutrašnjih petlji nije izvodljiva za implementaciju. Detaljnija analiza unutrašnjih petlji sugeriše da u prvoj petlji postoji međuzavisnost između prethodne i buduće vrednosti koeficijenata (elemenata niza) `cbeta`. Npr. element `cbeta[threadid][cnti-1]` direktno zavisi od prethodno izračunatog elementa `cbeta[threadid][cnti]`, pri čemu petlja iterira unazad.

Osim ove međuzavisnosti, postoji i međuzavisnost rezultata između dve unutrašnje petlje. Naime, čitav niz `cbeta` mora biti izračunat pre nego što se izračunaju nove vrednosti talasne funkcije `psi` (u drugoj petlji). Takođe, postoji i međuzavisnost između novih vrednosti talasne funkcije `psi`, pa tako svaka nova vrednost `psi[cnti + 1][cntj][cntk]` zahteva prethodno izračunatu vrednost `psi[cnti][cntj][cntk]`.

Višestruke međuzavisnosti koje postoje u funkciji `calclux` i onemogućavaju njenu potpunu paralelizaciju kao kod funkcije `calcnu` su direktna posledica višestrukih rekurzija prisutnih u algoritmu za propagaciju u vremenu.

2.1.8.9 Funkcija `calcluy`

U listingu 2.19 nalazi se izvorni kôd funkcije `calcluy`.

Listing 2.19: Izvorni kôd funkcije `calcluy`.

```
void calcluy(double ***psi, double **cbeta) {
   int threadid;
   long cnti, cntj, cntk;
   double c;

   #pragma omp parallel private(threadid, cnti, cntj, cntk, c)
   {
      threadid = omp_get_thread_num();

      #pragma omp for
      for(cnti = 0; cnti < Nx; cnti ++) {
         for(cntk = 0; cntk < Nz; cntk ++) {
            cbeta[threadid][Ny - 2] = psi[cnti][Ny - 1][cntk];
            for (cntj = Ny - 2; cntj > 0; cntj --) {
               c = - Ay * psi[cnti][cntj + 1][cntk] +
                     Ay0r * psi[cnti][cntj][cntk] -
                     Ay * psi[cnti][cntj - 1][cntk];
               cbeta[threadid][cntj - 1] =
                     cgammay[cntj] * (Ay * cbeta[threadid][cntj] - c);
            }
            psi[cnti][0][cntk] = 0.;
            for (cntj = 0; cntj < Ny - 2; cntj ++) {
                psi[cnti][cntj + 1][cntk] =
                     calphay[cntj] * psi[cnti][cntj][cntk] +
                     cbeta[threadid][cntj];
            }
            psi[cnti][Ny - 1][cntk] = 0.;
         }
      }
   }

   return;
```

```
}
```

Funkcija `calcluy` predstavlja funkciju za računanje vremenske propagacije u odnosu na H_3 deo Hamiltonijana (y-komponenta Laplasijana). Struktura kôda funkcije `calcluy` je slična funkciji `calclux`. Prva razlika je u tome što se spoljna petlja paralelizuje po X opsegu, dok su unutrašnje (sekvencijalne petlje) po Y opsegu. Kao i kod funkcije `calclux`, raspodela opsega po petljama je bitna za implementaciju distribucije podataka u okviru algoritma paralelnog transponovanja. Druga razlika u odnosu na funkciju `calclux` je to što se koriste drugačiji koeficijenti (`Ay` i `Ay0r` umesto `Ax` i `Ax0r`), kao i drugačiji koeficijenti Krenk-Nikolson šeme (`cgammay` i `calphay` umesto `cgammax` i `calphax`). Međuzavisnosti između različitih promenljivih opisane u prethodnoj sekciji za funkciju `calclux` postoje i u funkciji `calcluy`.

2.1.8.10 Funkcija `calcluz`

U listingu 2.20 nalazi se izvorni kôd funkcije `calcluz`.

Listing 2.20: Izvorni kôd funkcije `calcluz`.

```
void calcluz(double ***psi, double **cbeta) {
   int threadid;
   long cnti, cntj, cntk;
   double c;

   #pragma omp parallel private(threadid, cnti, cntj, cntk, c)
   {
      threadid = omp_get_thread_num();

      #pragma omp for
      for(cnti = 0; cnti < Nx; cnti ++) {
         for(cntj = 0; cntj < Ny; cntj ++) {
            cbeta[threadid][Nz - 2] = psi[cnti][cntj][Nz - 1];
            for (cntk = Nz - 2; cntk > 0; cntk --) {
               c = - Az * psi[cnti][cntj][cntk + 1] +
                     Az0r * psi[cnti][cntj][cntk] -
                     Az * psi[cnti][cntj][cntk - 1];
               cbeta[threadid][cntk - 1] =
                     cgammaz[cntk] * (Az * cbeta[threadid][cntk] - c);
            }
            psi[cnti][cntj][0] = 0.;
            for (cntk = 0; cntk < Nz - 2; cntk ++) {
               psi[cnti][cntj][cntk + 1] =
                    calphaz[cntk] * psi[cnti][cntj][cntk] +
                    cbeta[threadid][cntk];
            }
            psi[cnti][cntj][Nz - 1] = 0.;
```

```
            }
        }
    }

    return;
}
```

Funkcija `calcluz` predstavlja funkciju za računanje vremenske propagacije u odnosu na H_4 deo Hamiltonijana (z-komponenta Laplasijana). Struktura kôda funkcije `calcluz` je slična funkcijama `calclux` (odnosno, `calcluy`). Prva razlika je u tome što se spoljna petlja paralelizuje po X opsegu, dok su unutrašnje (sekvencijalne petlje) po Z opsegu. Kao i kod funkcija `calclux` i `calcluy` raspodela opsega po petljama je bitna za implementaciju distribucije podataka u okviru algoritma paralelnog transponovanja. Druga razlika u odnosu na funkcije `calclux` i `calcluy` jeste ta što se koriste drugačiji skalarni koeficijenti (`Az` i `Az0r` umesto `Ax` i `Ax0r`, odnosno `Ay` i `Ay0r`), kao i drugačiji koeficijenti Krenk-Nikolson šeme (`cgammaz` i `calphaz` umesto `cgammax` i `calphax`, odnosno `cgammay` i `calphay`). Međuzavisnosti između različitih promenljivih prethodno opisane kod funkcija `calclux` i `calcluy` postoje i u funkciji `calcluz`.

2.2 C/OpenMP programi i problem rezolucije fizičkog sistema

Prethodno razvijeni C/OpenMP programi [11] omogućavaju ubrzanje Krenk–Nikolson algoritma uz pomoć paralelizacije petlji unutar funkcija. Sa korisničke tačke to je veoma značajno, ali je ograničeno brojem niti, odnosno brojem procesorskih jezgara koji su dostupni na računaru na kojem se simulacija izvršava. Drugo važno ogrančienje koje uvodi taj pristup baziran na paradigmi deljene memorije („shared memory“) se odnosi na veličinu posmatranog fizičkog sistema, odnosno na rezoluciju sa kojom se on može proučavati. Ovo je posledica ograničene količine raspoložive radne memorije računara na kome se izvršava algoritam, jer se C/OpenMP programi izvršavaju isključivo na jednom računaru i dostupna im je samo memorija tog računara. Drugim rečima, oni ne mogu da iskoriste prednosti računarskih klastera i distribuirane memorije.

Za ilustraciju količine radne memorije neophodne za rad OpenMP programa posmatraćemo osnovne strukture podataka u programu. Vrednosti talasne funkcije sistema beleže se u određenom broju diskretnih tačaka u trodimenzionalnom koordinatnom sistemu. Osnovna konfiguracija za program sa propagacijom u imaginarnom vremenu poseduje 240, 200 i 160 tačaka po X, Y i Z osi, respektivno. U svakoj od ovih tačaka beleži se vrednost talasne funkcije, reprezentovana brojem u dvostrukoj preciznosti (`double`). Naravno, ovo su samo tipične vrednosti, a korisnik sam mora da odabere odgovarajuće vrednosti za svaku konkretnu primenu. Nekada se može dogoditi da fizički

sistem ima nekoliko dužinskih skala, u zavisnosti od detalja potencijala, i u tom slučaju diskretizacija je diktirana najmanjom skalom. Dakle, može da se dogodi da je potrebna i značajno veća rezolucija od tipične i jedna od motivacija za ovu tezu je upravo rešavanje takve vrste problema.

Osnovna konfiguracija za program sa propagacijom u realnom vremenu poseduje 200, 160 i 120 tačaka po X, Y i Z osi, respektivno, ali se često događa da je potrebna i značajno veća rezolucija. Za razliku od propagacije u imaginarnom vremenu, ovde se vrednosti talasne funkcije beleže uz pomoć kompleksnih brojeva u dvostrukoj preciznosti (`complex double`). Samim tim zauzeće memorije je veće i broj tačaka po osama mora biti manji ako se želi približno isto zauzeće radne memorije kao u slučaju propagacije u imaginarnom vremenu. Ovako čuvana talasna funkcija predstavlja trodimenzionalnu strukturu podataka (tenzor trećeg reda ili trodimenzionalna matrica), odnosno niz dvodimenzionalnih matrica.

Prethodno opisane konfiguracije trodimenzionalne matrice talasnih funkcija i ostalih pridruženih struktura podataka dovode do sledećeg zauzeća radne memorije: za osnovnu konfiguraciju programa za propagaciju u imaginarnom vremenu $(240\times200\times160)$ neophodno je oko 307 MB radne memorije, a za propagaciju u realnom vremenu za osnovnu konfiguraciju $(200\times160\times120)$ neophodno je 410 MB radne memorije, usled dvostrukog zauzeća kompleksnog tipa podataka u odnosu na tip podataka u dvostrukoj preciznosti. U ovo zauzeće radne memorije uključuju se trodimenzionalne matrice talasne funkcije, ali i četiri dodatne trodimenzionalne matrice opisane u prethodnim sekcija, među kojima su matrice za skladištenje informacija o potencijalu i matrice za smeštanje privremenih podataka.

Ukoliko bi zainteresovani istraživač želeo da posmatra vremensku evoluciju sistema koristeći finiju rezolucije od osnovne konfiguracije, na primer ukoliko bi želeo da posmatra sistem u rezoluciji $480\times400\times320$ tačaka, potrošnja memorije bi se povećala $n_x n_y n_z$ puta, pri čemu je $n_i(i=x,y,z)$ množilac za koji je uvećana odgovarajuća dimenzija osnovne konfiguracije. Dakle, u slučaju analize sistema veličine $480\times400\times320$ tačaka zauzeće radne memorije bi poraslo na 2^3 * 307 MB, tj. na oko 2.4 GB. Na današnjim kućnim računarima ili radnim stanicama koje poseduju 8 do 16 GB radne memorije ovoliko zauzeće radne memorije je u granicama tolerancije.

Međutim, ukoliko zainteresovani istraživač želi dodatno da poveća rezoluciju fizičkog sistema na $960\times800\times640$ tačaka, zauzeće radne memorije bi se povećalo $4^3 = 64$ puta u odnosu na osnovnu konfiguraciju, što u ovom slučaju predstavlja blizu 20 GB radne memorije. Ovu količinu radne memorije u današnje vreme poseduje mali broj kućnih računara, dok se toliko memorije može naći u savremenim radnim stanicama koje poseduju istraživački instituti ili univerziteti.

Svako dalje povećavanje rezolucije posmatranog sistema dovodi do odgovarajućeg značajnog rasta količine radne memorije potrebne za pokretanje programa, pa je tako za fizički sistem od $1920\times1600\times1280$ tačaka neophodno čak $2^8 = 512$ puta više radne memorije u odnosu na inicijalnu konfiguraciju, tj. neophodno je oko 157 GB radne memorije. Ovu količinu radne memorije ne poseduje ni jedan lako dostupan računar na svetu. Na primer, jedan od najjačih superračunara na svetu Tian-2 („Tianhe-2“) poseduje individualne računarske čvorove („compute

nodes“) koji imaju po 64 GB radne memorije. Drugim rečima, na Tian-2 superračunaru za smeštanje ove količine podataka potrebna bi bila 3 računarska čvora, a za program za propagaciju u realnom vremenu 4 čvora. Pošto čak i na današnjim superračunarima tipični čvorovi imaju od 32 GB do 128 GB, za gore navedenu rezoluciju ne bi bilo moguće koristiti ranije razvijene C/OpenMP programe.

Iz svega navedenog zaključujemo da C/OpenMP programi koji se izvršavaju na samo jednom računaru imaju značajna ograničenja ukoliko želimo da posmatramo vremensku evoluciju Boze-Ajnštajn kondenzata sa finom rezolucijom sistema. Jedno ograničenje je vezano za brzinu obrade podataka, koja zavisi od broja dostupnih računarskih jezgara, a drugo, mnogo značajnije, je vezano za količinu radne memorije i maksimalnu rezoluciju koja se može proučavati.

2.3 Predlog hibridnog C/OpenMP/MPI rešenja

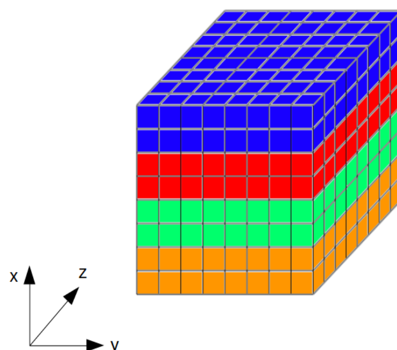
Kao što smo objasnili u prethodnoj sekciji, ranije razvijeni C/OpenMP programi [11] su pogodno rešenje za proučavanje dinamike Boze-Ajnštajn kondenzata u nižim rezolucijama. Za finije rezolucije ti programi ne pružaju dovoljno fleksibilnosti, odnosno ograničeni su količinom raspoložive radne memorije na računaru na kome se program izvršava.

Da bi se omogućilo proučavanje Boze-Ajnštajn kondenzata u višim rezolucijama neophodno je osmisliti proširenje (ili potpunu izmenu) postojećih C/OpenMP programa. Neophodno je takođe, da se novi programi dobro skaliraju u odnosu na postojeće C/OpenMP programe, odnosno da efikasno koriste distribuirane računarske resurse, pri čemu se pod skaliranjem podrazumeva odnos vremena izvršavanja dva programa, za istu ili finiju rezoluciju posmatranog sistema. Jedan od ciljeva je da se korišćenjem distribuiranih resursa (računarskih klastera) posmatrani problem fiksne rezolucije brže simulira, odnosno da se programi izvršavaju brže koristeći više procesora. Drugi cilj je da se omogući da se povećava rezolucija proučavanog sistema uz istovremeno povećanje računarskih resursa koji se koriste, pri čemu se očekuje da brzina izvršavanja ostane ista, odnosno da se ne smanji na značajniji način. Ovi ciljevi su povezani sa onim što se obično naziva jako i slabo skaliranje („strong and weak scaling“).

Među istraživačima koji se bave paralelnim programiranjem poznato je da je u slučaju obrade velikog skupa podataka neophodan prelazak na distribuiranu programsku paradigmu. Osnovni elementi i funkcije koje podržavaju distribuiranu programsku paradigmu implementirani su u okviru ustanovljenog standarda, biblioteke za razmenu poruka – „Message Passing Interface“ (MPI) [14]. Suština MPI biblioteke jeste distribuiranje i razmena podataka između skupa računara organizovanih kao računarski klaster. Pod klasterom podrazumevamo bilo koju konfiguraciju računara koja poseduje više od jednog računara povezanih brzom lokalnom mrežom (gigabitni eternet ili Infiniband mreža), pri čemu su računari podešeni tako da mogu međusobno da komuniciraju na način koji omogućava paralelno izvršavanje programa. Obično računarski čvorovi jednog klastera imaju deljen fajl sistem koji olakšava izvršavanje paralelnih simulacija, ali to nije

obavezno. Uz pomoć MPI biblioteke moguće je podeliti podatke između računara u klasteru i omogućiti svakom od računara da vrši deo obrade na svom podskupu podataka, kao i da čvorovi razmenjuju podatke međusobno, kroz direktnu („point to point“) ili kolektivnu komunikaciju. Upravo ova mogućnost komunikacije predstavlja način na koji se realizuje paralelno izvršavanje programa.

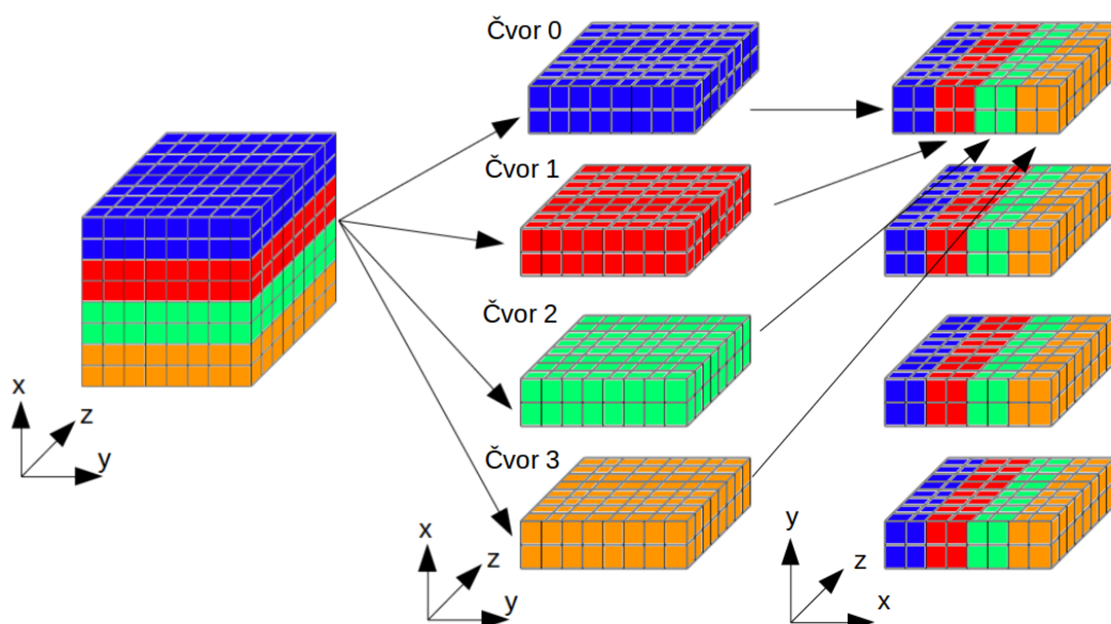
Kako C/OpenMP program efikasno paralelizuje numerički algoritam na jednom računaru, naša osnovna ideja je da izvršimo nadogradnju postojećeg rešenja u hibridno C/OpenMP/MPI rešenje, pri čemu će distribuciju i razmenu podataka između čvorova vršiti MPI sloj, dok će numeričke proračune na pojedinačnim čvorovima izvršavati OpenMP niti. Zahtev koji sledi iz korišćenja MPI distribucije podataka jeste da broj diskretizacionih tačaka talasne funkcije po X osi (`Nx`) bude deljiv brojem računarskih čvorova. Npr. za `Nx`=960 tačaka po X osi, dozvoljena konfiguracija je svaki broj čvorova koji deli broj 960 bez ostatka (npr. 2, 3, 4, 5, 6, 8, 10, ...). Ilustracija predložene distribucije podataka po X osi data je na slici 2.4. Pri tome, da bismo na najefikasniji način koristili sve resurse, primenjujemo ravnomernu raspodelu podataka po čvorovima.



Slika 2.4: Ravnomerna distribucija podataka po X osi na računarskom klasteru. Različite boje predstavlju skupove podataka koje su dodeljene različitim čvorovima (u ovom slučaju 4 čvora).

Ako primenimo prethodnu distribuciju podataka, svakom čvoru klastera će biti dodeljeno `Nx / gsize` tačaka po X osi (pri čemu je `gsize` broj čvorova), a `Ny` i `Nz` tačaka po preostale dve ose. Zamislimo računarski klaster sa 8 čvorova i sistem veličine $1920\times1600\times1280$ tačaka koji se distribuira po čvorovima tog klastera. Na jednom računarskom čvoru u tom slučaju nalaziće se $(1920/8)\times1600\times1280$ tačaka sistema, tj. $240\times1600\times1280$ tačaka. Uzimajući u obzir da ovakav sistem ukupno zauzima 157 GB radne memorije, distribucija podataka sa X ose će smanjiti potrošnju memorije na oko 20 GB memorije po čvoru klastera. Računajući da svaki čvor u klasteru poseduje 32 GB radne memorije, ovakva potrošnja memorije je odgovarajuća, čime se rešava problem zauzeća memorije u slučaju sistema visoke rezolucije. Neophodno je ipak, u slučaju veoma velikih sistema, imati pristup dovoljnom velikom klasteru, odnosno dovoljno velikom broju kompjuterskih čvorova, koja omogućava distribuciju podataka i rad programa. Ovo može predstavljati netrivijalan problem za korisnika kada se radi o hardverskim resursima, ali rešenje razvijeno u ovoj tezi daje principijelnu mogućnost da se za simulaciju koristi proizvoljno veliki računarski klaster.

Pored distribuiranja podataka po različitim čvorovima, neophodna je i razmena podataka između pojedinačnih čvorova. Ova razmena uslovljena je činjenicom da svaki od čvorova poseduje podatke samo iz dela domena duž X ose, dok numeričke funkcije `calclux, calcmuen` i druge (poglavlje 2.1) zahtevaju obradu svih podataka duž X ose, koji nisu prisutni ni na jednom pojedinačnom računarskom čvoru. Kod nekih od funkcija (`calcrms, calcnorm`) moguće je organizovati njihovo izvršavanje tako da svaki čvor obradi lokalne podatke koji se nakon toga prikupe na jednom čvoru i izračuna se finalni rezultat. Međutim, neke od funkcija zahtevaju da se celokupan domen podataka po X osi nađe na pojedinačnim čvorovima, odnosno za njihovo uspešno izvršavanje potrebno je transponovati 3D matrice talasne funkcije u X-Y ravni ili obratno (raspodela podataka duž Z ose se ne menja). Nakon ovog transponovanja umesto da čvorovi sadrže podatke iz dela domena duž X ose i celokupni Y domen, oni će sadržati podatke iz dela domena duž Y ose i celokupan X opseg. Ilustracija ovakve vrste transponovanja je data na slici 2.5.



Slika 2.5: 3D transponovanje distribuiranih podataka između čvorova klastera opisano u tekstu.

Implementacija prethodno ilustrovanog transponovanja je netrivijalna i zahteva detaljnu analizu koja će biti jedan od predmet ove doktorske teze.

Nakon MPI distribucije i transponovanja, obrada podataka na čvorovima nastavlja se uz pomoć OpenMP niti, sledeći prethodno razvijeni algoritam. Ovakva hibridna implementacija omogućava maksimalno ubrzanje algoritma za velike skupove podataka. Ubrzanje hibridnog programa najviše je ograničeno komunikacionim vremenom, neophodnim za razmenu podataka između čvorova klastera tokom svake vremenske iteracije, koje zavisi od: hardverske konfiguracije klastera (mreže), implementacije struktura podataka koje se razmenjuju između čvorova, kao i od načina komunikacija između čvorova (blokirajuća, neblokirajuća, kolektivna, itd.). U narednom poglavlju ćemo detaljno analizirati i predstaviti rešenje problema dekompozicije i transponovanja

2D matrica, a nakon toga ćemo preći na razmatranje transponovanja 3D matrica distribuiranih na gore opisani način.

3 Dekompozicija i transponovanje distribuiranih 2D matrica

Pre prelaska na razmatranje problema transponovanja distribuiranih 3D matrica, u ovom poglavlju ćemo obraditi analogan problem za 2D distribuirane matrice, na osnovu analize literature. Naša osnovna hipoteza je da se algoritmi dekompozicije i transponovanja u 2D mogu uopštiti i iskoristiti za 3D dekompoziciju i transponovanje neophodno za realizaciju hibridnog rešenja (sekcija 2.3). Referentni rad koji je predstavljao polaznu tačku istraživanja jeste rad Džajonga Čoija i Džeka Dongare iz 1995. godine [15]. S obzirom da je Džek Dongara jedan od pionira oblasti paralelnog programiranja i paralelnih numeričkih algoritama, kao i autor ili koautor brojnih numeričkih biblioteka (EISPACK, LINPACK, BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, PAPI) [16], ovaj referentni rad je uzet kao relevantna osnova za istraživanje.

U okviru referentnog rada [15] prikazana su dva pristupa i dekompozicije 2D distribuiranih matrica. Oba slučaja pretpostavljaju da je klaster računara u logičkom smislu organizovan (enumerisan) kao dvodimenzionalna matrica računara, pri čemu je broj računara u vrstama označen sa P, a u kolonama sa Q, što je i prikazano na slici 3.1 za $P = 2$, $Q = 3$.

Slika 3.1: Shematski prikaz matrice računara (klaster) za $P=2$, $Q=3$.

Treba napomenuti da nije neophodno da se računari fizički nalaze na datim pozicijama u okviru klastera. Data organizacija se može postići logički, uz upotrebu MPI topologija. MPI topologije omogućavaju formiranje logičkih rasporeda čvorova u okviru MPI klastera. Na primer, u konkretnom slučaju prikazanom na gornjoj slici, pozivom MPI funkcije `MPI_Cart_create` za stvaranje Dekartove (2D) topologije možemo logički da formiramo prethodno opisanu konfiguraciju, što je prikazano u listingu 3.1. Naravno, ako mrežna topologija odgovara logičkoj

organizaciji klastera, to će svakako dovesti do poboljšanja performansi komunikacije. Međutim, svi programi koje smo razvili ne zavise od ovih hardverskih detalja i uspešno će se izvršavati bez obzira na hardversku konfiguraciju klastera. Na korisniku ostaje da izabere hardeversku konfiguraciju koja je optimalna, odnosno najbliža optimalnoj u okviru raspoloživih resursa.

Listing 3.1: Primer stvaranja dvodimenzionalne MPI topologije.

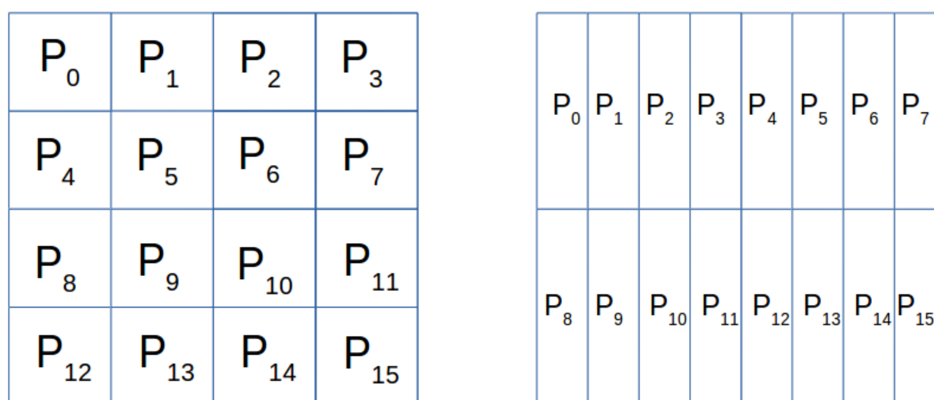
```
MPI_Comm comm; //novi MPI komunikator (stvoren od nove topologije)
int dims = 2; //broj osa
const int dims[] = {2, 3}; //broj cvorova po osama
const int periods[] = {1, 1}; //periodicnost po osama
int reorder = 0; //ne menjati rangiranje procesa

MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &comm_cart);
```

Algoritmi koje ćemo predstaviti imaju široku upotrebu zbog svoje generalnosti, pošto parametri P i Q mogu biti proizvoljni prirodni brojevi.

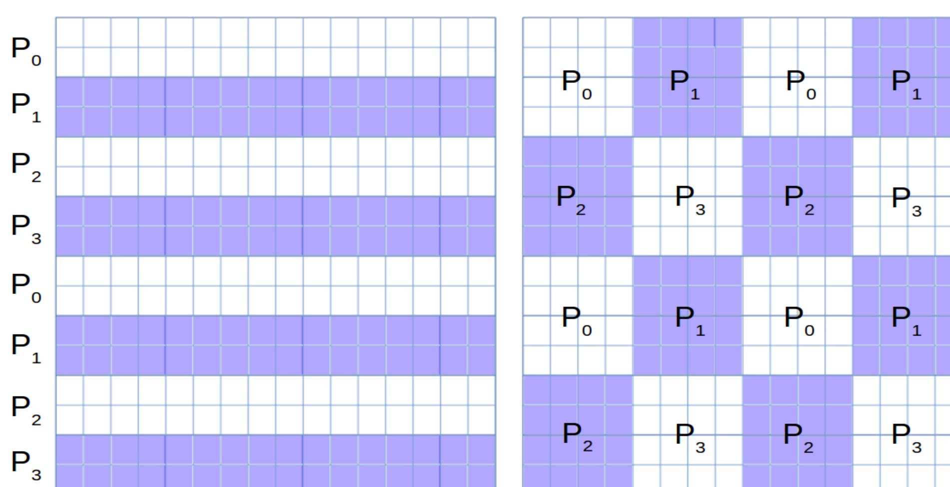
U okviru referentnog rada [15] razlikuju se dva pristupa transponovanju i distribuciji 2D matrica. Prvi pristup odgovara slučaju kada su P i Q uzajamno prosti brojevi, npr. $P = 2$, $Q = 3$. U tom slučaju najveći zajednički delitelj (NZD) broja čvorova po vrstama i kolonama je broj jedan (i eventualno broj čvorova po vrstama i kolonama, ako je $P = Q$). Drugi pristup transponovanju i distribuciji 2D matrica odgovara slučaju kada P i Q nisu međusobno prosti brojevi, tj. kada je njihov NZD veći od broja jedan (npr. ukoliko je $P = 12$, $Q = 8$, NZD = 4), pri čemu isključujemo slučaj $P = Q =$ NZD. Oba algoritma koriste neblokirajuću MPI komunikaciju, što implicira da je moguće da jedan čvor šalje i prima podatke ka i od više čvorova paralelno, bez čekanja da se završi komunikacija sa prethodnim čvorom.

Pre opisa algoritama transponovanja neophodno je opisati i nekoliko standardnih algoritama za distribuciju podataka organizovanih u 2D matrice. Dva standardna algoritma za distribuciju podataka 2D matrica su blokovska distribucija podataka i ciklična blokovska distribucija podataka:

- **Blokovska distribucija podataka** [17] podrazumeva distribuciju podataka tako da svaki MPI proces (koji se izvršava na jednom čvoru) sadrži podskup podataka iz originalne matrice A. Dati podskup podataka je određen u ravnomernom rasporedu, pri čemu svaki čvor uzima neprekidan podskup podataka matrice A. Primer dve blokovske distribucije podataka dat je na slici 3.2, pri čemu jedna distribucija uzima isti broj elemenata po X i Y osi (kvadratni oblik), a druga distribucija uzima više elemenata po Y osi u odnosu na broj elemenata po X osi (pravougaoni oblik).
- **Blokovska ciklična distribucija podataka** [17] takođe podrazumeva distribuciju podataka tako da svaki MPI proces (čvor) sadrži podskup podataka iz originalne matrice A. Dati podskup podataka je određen u pravilnom cikličnom rasporedu, pri čemu svaki čvor uzima ciklično podskup podataka matrice A. Primer dve ciklične blokovske distribucije podatka dat je na slici 3.3, pri čemu jedna distribucija ciklično uzima isti broj elemenata po X i Y osi (kvadratni oblik), a druga distribucija ciklično uzima više elemenata po X osi u odnosu na broj elemenata po Y osi (pravougaoni oblik).



Slika 3.2: Blokovska distribucija podataka [14].



Slika 3.3: Blokovska ciklična distribucija podataka [14].

Upotreba jedne ili druge distribucije dvodimenzionalnih matrica zavisi prevashodno od tipa problema koji se rešava. Na primer, ukoliko matematički ili fizički problem koji se rešava ima podjednaku kompleksnost računanja na svim delovima matrice, tada se može koristiti čista blokovska distribucija podataka. U ovom slučaju očekivano je da svi MPI procesi (čvorovi) obrađuju sličnu količinu podataka, tj. imaju slično opterećenje („load“). Sa druge strane, ukoliko određeni delovi distribuirane matrice podležu većem broju operacija od ostalih delova matrice (tzv. nebalansirano opterećenje), tada je bolje rešenje za distribuciju podataka blokovska ciklična raspodela podataka. Jedan takav primer jeste Gausova eliminacija za rešavanje sistema linearnih jednačina [12]. Blokovska ciklična distribucija podataka u tom slučaju omogućava relativno blisko opterećenje svih čvorova, s obzirom da čvorovi sadrže ciklične blokove matrice i favorizuju različite delove matrice podjednako. Takođe, na ovaj način se obezbeđuje i dobar memorijski pristup različitim delovima matrice, jer svaki ciklični blok sadrži kontinualne lokacije po X ili Y osama (redovima ili kolonama), što je bitno, znajući da je memorijski raspored podataka u programskom jeziku C tzv. „row-major order“, a kod Fortrana je u pitanju „column-major order“.

U okviru algoritama za transponovanje iz referentnog rada [15] pretpostavlja se da svaki čvor sadrži više blokova matrice. Na slici 3.4 dat je primer ciklične distribucije matrice veličine 12×12 blokova, koji je distribuiran na procesorsku mrežu veličine 2×3 ($P = 2$, $Q = 3$).

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2	0	1	2	0	1	2
11	3	4	5	3	4	5	3	4	5	3	4	5

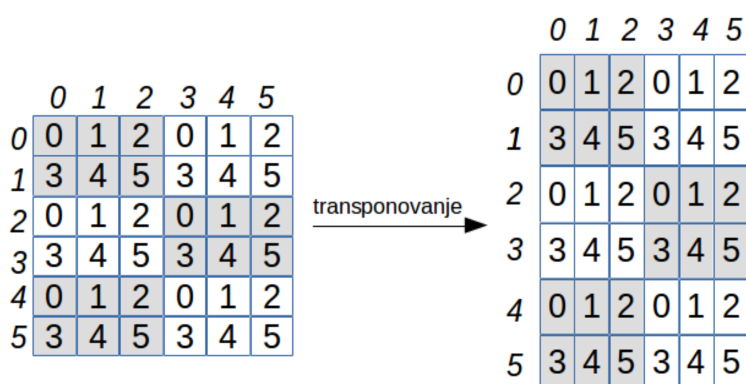
Slika 3.4: Ciklična blokovska distribucija matrice za P=2, Q=3.

Brojevi unutar blokova predstavljaju broj MPI procesa (čvora) na kom se nalazi blok matrice, a svi blokovi koji su označeni istim brojem se nalaze na istom čvoru, pri čemu se broj čvora nalazi u opsegu od 0 do $P\,Q$ - 1. Obeležavajući najmanji zajednički sadržalac P i Q sa NZS, kvadrat blokova veličine $\mathrm{NZS} \times \mathrm{NZS}$ ćemo zvati NZS blok. U prethodnom slučaju NZS = 2×3 = 6, pa je time NZS blok veličine 6×6 originalnih blokova matrice. Iz tog razloga, matrica sa slike 3.4 može biti posmatrana kao niz NZS blokova veličine 2×2, kao što je prikazano na slici 3.5.

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2	0	1	2	0	1	2
11	3	4	5	3	4	5	3	4	5	3	4	5

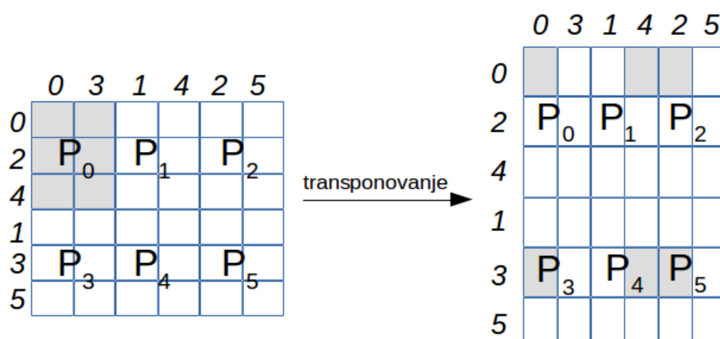
Slika 3.5: NZS blokovska distribucija matrice za P=2, Q=3.

Ukratko rečeno, algoritmi paralelnog transponovanja se mogu opisati na sledeći način. Matrica A je distribuirana na $P \times Q$ mrežu čvorova klastera i sastoji se od $Mb \times Nb$ blokova, a svaki od blokova sadrži $r \times c$ osnovnih elemenata matrice, pri čemu r i c mogu biti proizvoljni prirodni brojevi. Slika 3.6 pokazuje primer transponovanja matrice za $P = 2$, $Q = 3$. Ukoliko se matrica A transponuje, transponovana matrica A^T je takođe distribuirana na $P \times Q$ mrežu čvorova. Transponovana matrica A^T ima $Nb \times Mb$ blokova i svaki blok ima $c \times r$ elemenata, što znači zamenu dimenzija blokova matrice u odnosu na originalnu (netransponovanu) matricu, ali i zamenu dimenzija osnovnih elemenata unutar blokova. Elementi unutar svakog bloka ostaju u istom bloku, ali mogu biti na drugom čvoru, a elementi svakog bloka su interno transponovani.



Slika 3.6: Transponovanje matrice A.

Slika 3.7 pokazuje primer transponovanja sa stanovišta čvorova klastera. Ako su P i Q međusobno prosti brojevi, blokovi sa čvora P_0 se distribuiraju po svim drugim čvorovima.



Slika 3.7: Transponovanje matrice A (sa stanovišta čvorova klastera).

Konkretizacija algoritama paralelnog transponovanja zavisi od vrednosti NZD brojeva P i Q.

3.1 Slučaj kada su *P* i *Q* međusobno prosti brojevi (NZD = 1)

Ukoliko su brojevi čvorova po vrstama (*P*) i kolonama (*Q*) uzajamno prosti, tj. ukoliko je njihov najveći zajednički delilac NZD = 1, tada se blokovi koji se nalaze na čvoru P_0 distribuiraju po svim drugim čvorovima, kao na slici 3.7. U ovom slučaju obavlja se kompletna dvodimenzionalna razmena blokova između svih čvorova u klasteru. Pojedinačne komunikacije između pošiljaoca i primaoca odgovarajućeg bloka vrše se direktnom neblokirajućom komunikacijom.

Referentni rad [15] predstavlja i pseudokôd koji ilustruje ponašanje jednog čvora u razmeni blokova. Čvor je predstavljen oznakom P_i (engl. „process“). U radovima iz polja distribuiranog programiranja obično se spominju MPI procesi, a podrazumeva se da se to odnosi na kôd koji izvršava jedan čvor klastera. Moguće je simulirati i veći broj MPI procesa na jednom čvoru tako da mapiranje čvorova u MPI procese ne mora uvek biti jedan na jedan. Listing 3.2 pokazuje pseudokôd algoritma transponovanja. Ovde smo procese označili sa P(p, q), gde sada indeksi p i q označavaju položaj čvora u mreži (slika 3.1).

Listing 3.2: Pseudokôd algoritma paralelnog transponovanja 2D matrice sa stanovišta čvora (MPI procesa) kada su P i Q prosti brojevi (NZD = 1).

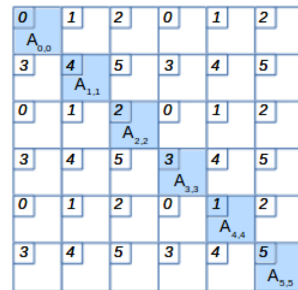
```
FOR (J = 0; J < Q; J++) {
   FOR (I = 0, I < P; I++) {
      [ Kopiraj sve blokove A zahtevane od P(p + I, q – J) u T1 (u zapakovanoj i transponovanoj formi) ]
      [ Pošalji Tl ka P(p + I, q - J) ]
      [ Primi T2 od P(p - I, q + J) ]
      [ Kopiraj T2 u C ]
   }
}
```

Čvor P(p, q) na početku algoritma transponuje blokove koji će ostati na ovom čvoru. Ovo odgovara prvoj iteraciji unutrašnje i spoljašnje petlje, tj. za vrednosti brojača I = J = 0. Nakon ovoga, čvor rukuje blokovima koje treba da razmeni sa čvorovima u istoj koloni (P(p – I, q), $0 \leqslant I < P$). Čvor šalje blokove svom gornjem susedu, čvoru P(p - 1, q), a prima blokove od svog donjeg suseda, čvora P(p + 1, q). Nakon ovoga, čvor šalje blokove sledećem čvoru iznad svog gornjeg suseda, čvoru P(p - 2, q), i prima blokove od čvora ispod svog donjeg suseda, P(p + 2, q). Ova procedura se ponavlja onoliko puta koliko ima čvorova u koloni.

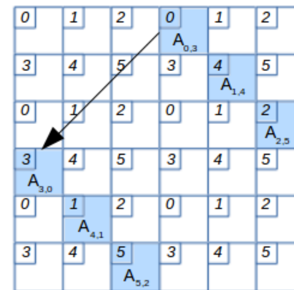
Nakon što završi razmenu sa čvorovima u istoj koloni, čvor šalje blokove čvorovima levo od sebe (P(p + I, q – 1), $0 \leqslant I < P$), i prima blokove od čvorova desno od sebe, P(p - I, q + 1). Ova

procedura se ponavlja onoliko puta koliko ima čvorova u vrsti. Sve operacije će se završiti u $P \times Q$ koraka.

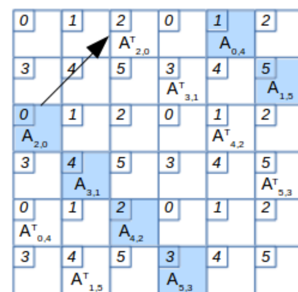
Sledi opis algoritma posmatranog sa stanovišta blokova matrice koju transponujemo. U svakom NZS bloku (u datom primeru veličine 6×6 osnovnih blokova), prethodno opisani algoritam vrši transponovanje šest NZS dijagonalnih blokova matrice u svakom koraku. Prvi korak u algoritmu sa listinga 3.2 ne zahteva eksplicitnu komunikaciju između čvorova. Čvorovi u prvom koraku razmenjuju blokove interno, što podrazumeva da blokovi sa glavne dijagonale $A(i, i)$ bivaju interno transponovani, ali ostaju na istom čvoru, slika 3.8(a).



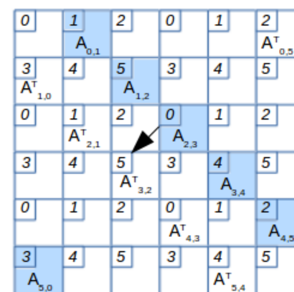
(a) nulta dijagonala, A(i,j) gde je MOD(j-i, NZS)=0



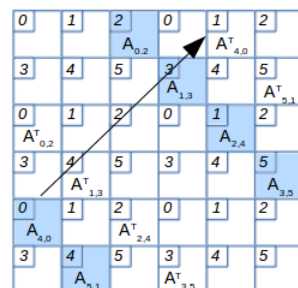
(b) treća dijagonala, A(i,j) gde je MOD(j-i, NZS)=3



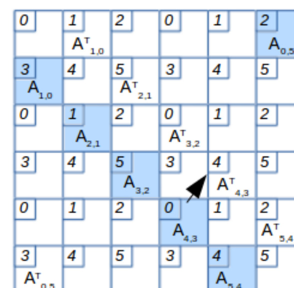
(c) četvrta dijagonala, A(i,j) gde je MOD(j-i, NZS)=4



(d) prva dijagonala, A(i,j) gde je MOD(j-i, NZS)=1



(e) druga dijagonala, A(i,j) gde je MOD(j-i, NZS)=2



(f) druga dijagonala, A(i,j) gde je MOD(j-i, NZS)=5

Slika 3.8: Transponovanje matrice sa stanovišta blokova koji se transponuju za P = 2, Q = 3 i $M_b = N_b = 6$.

Nakon ovoga između čvorova transponuju se svi blokovi koji se nalaze na trećoj dijagonali, slika 3.8(b). Na slici 3.8(c) čvor P_0, odnosno P(0, 0) u mreži, šalje blok ka čvoru P_2 (P(0, 2)), a prima blok od čvora P_1 (P(0, 1)), pri čemu su čvorovi P_0, P_1 i P_2 u istoj vrsti. Nakon ovoga, na slici 3.8(d) je prikazano kako čvor P_0 šalje blok ka čvoru P_5 (P (1, 2)), a prima blok od čvora P_4 (P (1, 1)), itd. Čvorovi treba da odrede koji dijagonalni blok matrice *A* treba da transponuju u kom koraku, kako bi

poslali odgovarajući blok odgovarajućem čvoru. Pseudokôd algoritma za račun odgovarajućeg bloka dat je u referentnom radu [15], kao i u listingu 3.3.

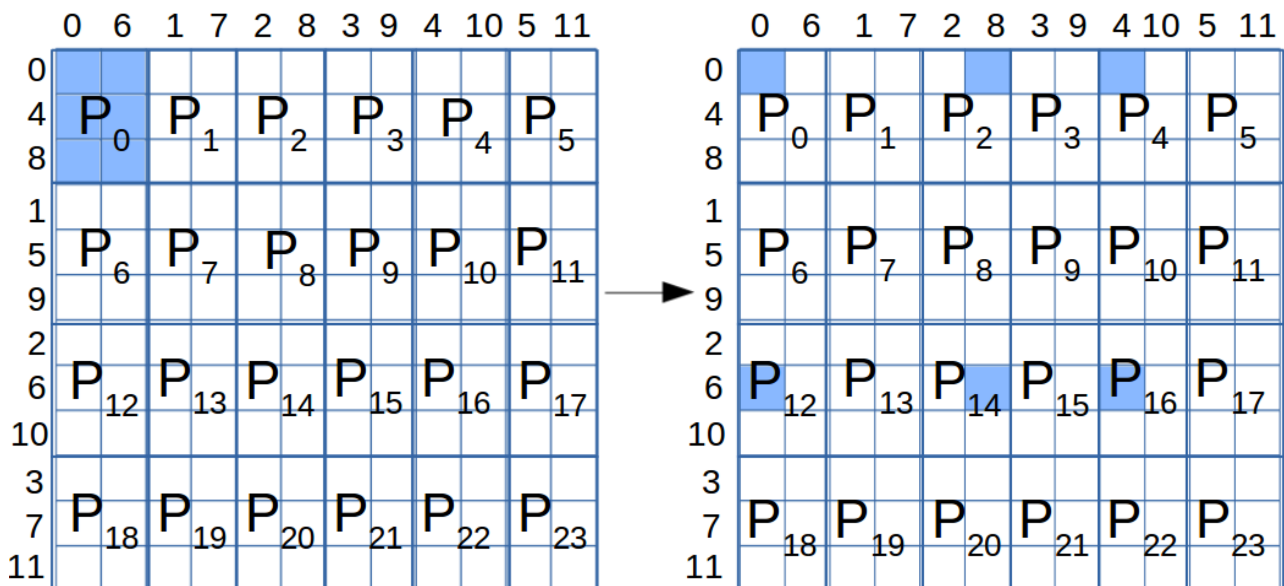
*Listing 3.3: Pseudokôd algoritma paralelnog transponovanja 2D matrice sa stanovišta blokova matrice koji se transponuju kada su P i Q međusobno prosti brojevi (NZD = 1). Jedan dijagonalni blok se transponuje u jednom koraku. Oznaka (start : granica : intv) predstavlja vrednosti x gde je x = start + intv * y, y = 0, 1, …, pri čemu x ne može da pređe zadatu granicu.*

```
FOR (J = 0; J < Q; J++) {
   K = J  //K predstavlja broj dijagonale koja se transponuje
   WHILE (MOD (K, P) != 0) {
      K = MOD(K + Q, NZS)
   }
   FOR (I = 0, I < P; I++) {
      [ Kopiraj sve blokove (K: Nb : NZS) dijagonalne blokove sa čvora
        P(p, q) u  T1 ]
      [ Pošalji T1 sa P(p, q) ka P(p + I, q - J) ]
      [ Kopiraj primljen T1 u C ]
      K = MOD(K + Q, NZS)
   }
}
```

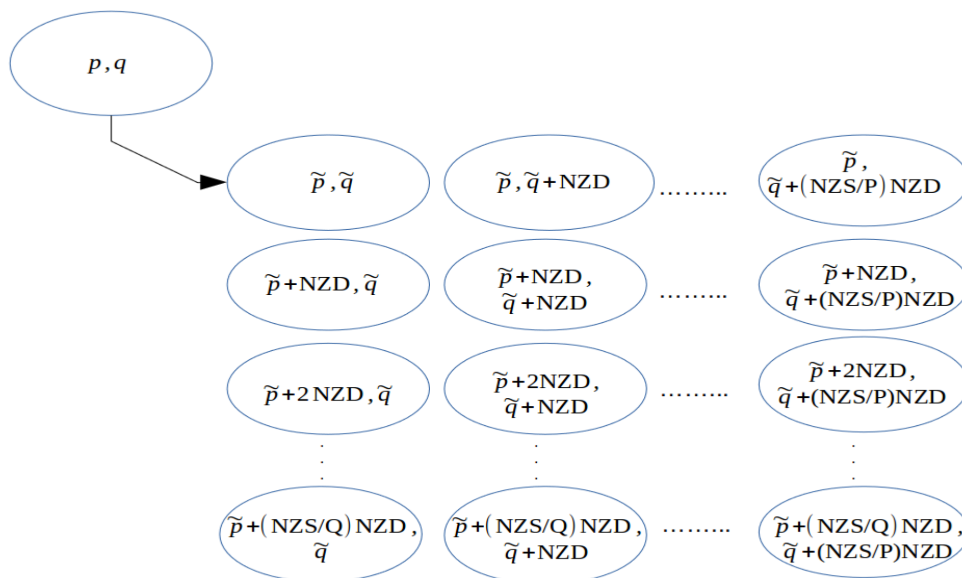
3.2 Slučaj kada *P* i *Q* nisu međusobno prosti brojevi (NZD > 1)

U prethodnoj sekciji opisali smo slučaj kada su *P* i *Q* međusobno prosti brojevi, što zahteva komunikaciju svakog čvora sa svakim čvorom. Referentni rad [15] detaljno opisuje i algoritam u slučaju kada *P* i *Q* nisu međusobno prosti brojevi (NZD > 1), tj. algoritam u slučaju složenije organizacije računarskog klastera.

Slika 3.9 prikazuje primer transponovanja matrice veličine 12×12 blokova, koja je distribuirana na procesorsku mrežu veličine 4×6 ($P = 4$, $Q = 6$). Svaki čvor (proces) sadrži sopstvenu mrežu blokova veličine 3×2, odnosno $NZS/P\times NZS/Q$. U ovom primeru neophodno je šest komunikacionih koraka za transponovanje matrice. Kao što je prikazano na slici 3.10, čvor P(p, q) započinje komunikaciju sa čvorom $P(\tilde{p},\tilde{q})$, gde se $\tilde{p}$ i $\tilde{q}$ izračunavaju iz p i q. Nakon što se odredi broj prvog čvora sa kojim se komunicira, čvor P(p, q) komunicira sa ostalim čvorovima koji su udaljeni u vertikalnom i horizontalnom smeru po NZD čvorova od prvog čvora, odnosno $P(\tilde{p},\tilde{q})$. Granice dve petlje algoritma sa listinga 3.2 su promenjene sa *Q* i *P* na NZS/*P* i NZS/*Q*. Pseudokôd algoritma je prikazan u listingu 3.4.



Slika 3.9: Transponovanje matrice u slučaju kada je P = 4 i Q = 6.



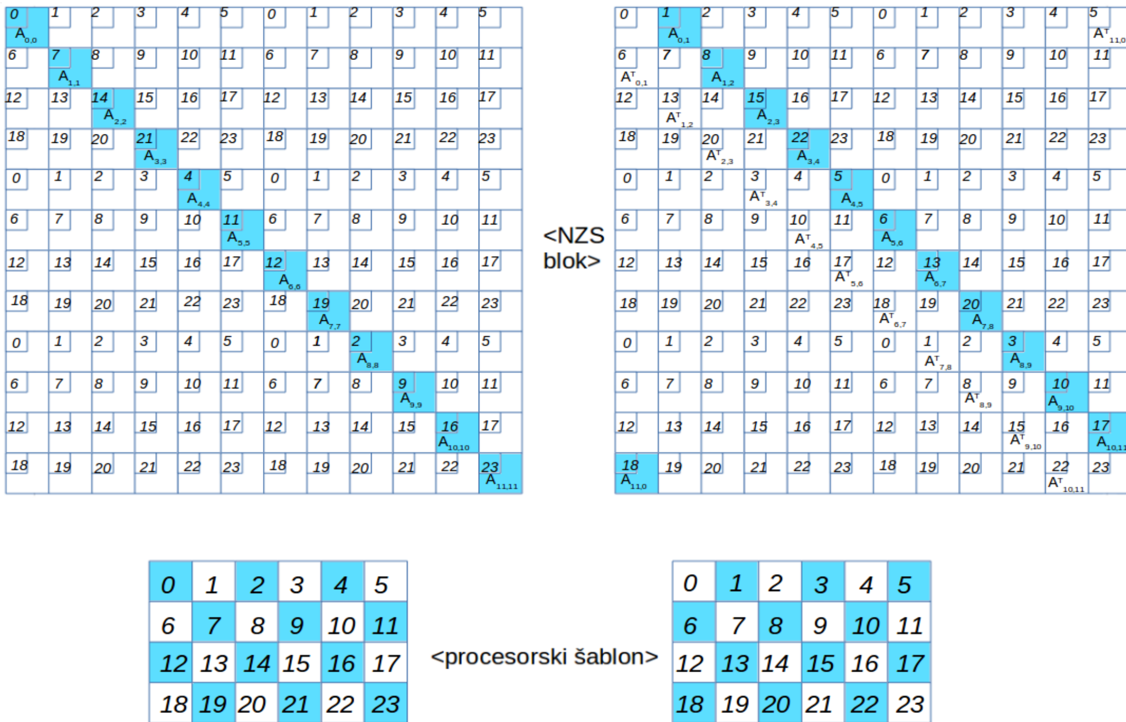
Slika 3.10: Mapa komunikacije čvorova u slučaju kada je NZD > 1.

Listing 3.4: Pseudokôd algoritma transponovanja sa stanovišta čvorova u slučaju kada je NZD > 1. Operacije NZD grupa procesora se preklapaju.

```
PARALLEL FOR (K = 0; K < NZD; K++) {
   g = MOD(q - p, NZD)
   p_tilda = MOD(p + g, P); q_tilda = MOD(q – g, Q)
   FOR (J = 0, J < NZS/P; J++) {
       FOR (I = 0, I < NZS/Q; I++) {
          [ Kopiraj u T1 (u kondenzovanoj i transponovanoj formi)
            sve blokove matrice A koje zahteva čvor
            P(p_tilda + I x NZD, q_tilda – J x NZD) ]
           [ Pošalji T1 ka P(p_tilda + I x NZD, q_tilda – J x NZD) ]
```

```
            [ Primi T2 od P(p_tilda – I x NZD, q_tilda + J x NZD) ]
            [ Kopiraj T2 u C ]
        }
    }
}
```

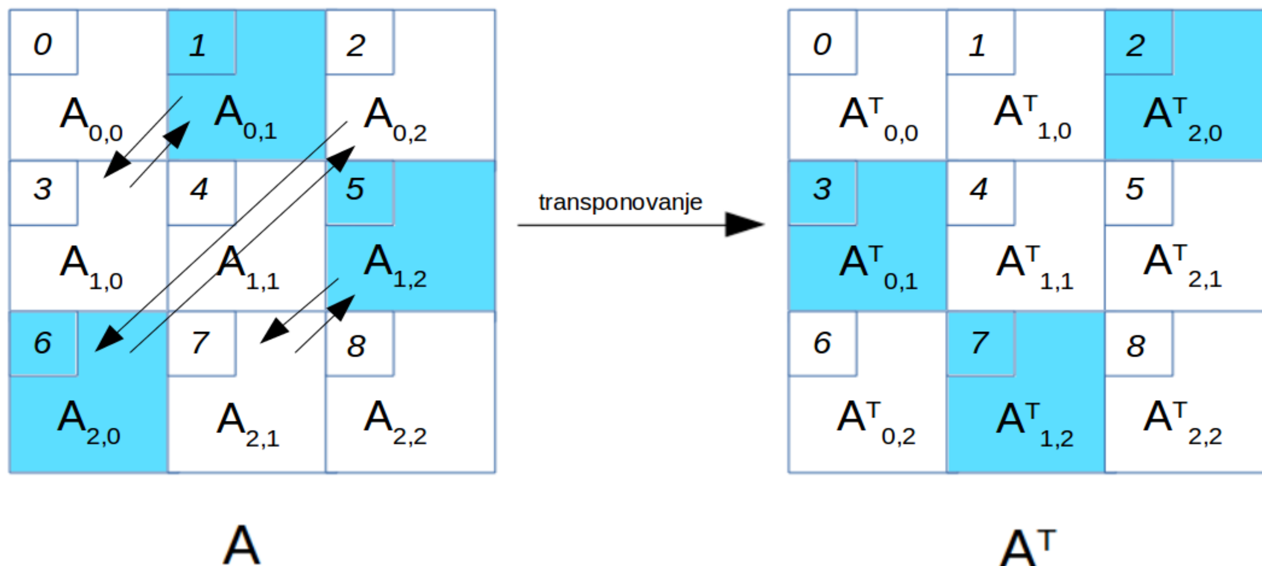
Slika 3.11, inspirisana referentnim radom [15], pokazuje transponovanje sa stanovišta matrice kada se transponuju nulta i prva dijagonala matrice *A* (odnosno delova matrice *A*(*i*, *j*) koji zadovoljavaju uslov MOD(*j*-*i*, *NZS*) = 0 i 1, respektivno) u slučaju kada je NZD > 1. Čvorovi koji šalju blokove su tamnije boje u odnosu na ostale čvorove. Kao što se može videti sa slike 3.11, *P Q* / NZD čvorova učestvuje u komunikaciji u svakom koraku. Sa druge strane, čvorovi se dele u NZD grupa čvorova, pri čemu čvorovi pripadaju istoj grupi *g* ukoliko imaju istu vrednost *g* = MOD (*q* – *p*, NZD). Čvorovi u grupi *g* šalju i primaju blokove od drugih čvorova u grupi *g*′ = MOD(NZD – *g*, NZD). Operacije svake od grupa mogu se vršiti uporedo (razlog za ovo su paralelne petlje u listingu 3.4).



Slika 3.11: Transponovanje nulte i prve dijagonale matrice za P = 4, Q = 6 i $M_b = N_b = 12$.

Posmatrajući problem sa stanovišta matrice, za transponovanje *k*-te dijagonale bloka matrice *A* (odnosno dela matrice *A*(*i*, *j*) koji zadovoljava uslov MOD(*j* – *i*, NZS) = *k*), grupa čvorova g_k = MOD(*k*, NZD) šalje dijagonalne blokove drugoj grupi g'_k = MOD(NZD - g_k, NZD). S obzirom da se operacije različitih grupa čvorova preklapaju, čvorovi su u stanju da transponuju NZD dijagonala paralelno, tako da se matrica može transponovati u NZS/NZD koraka. U graničnom slučaju, kada je

P = Q = NZD, što je prikazano na slici 3.12 za P = Q = NZD = 3, čvorovi transponuju sve tri dijagonale u jednom koraku. Čvor P(p, q) razmenjuje podatke sa čvorom P(q, p). Pseudokôd algoritma sa stanovišta matrice dat je u listingu 3.5.



Slika 3.12: Transponovanje matrice kada je P = Q = NZD = 3. Čvorovi transponuju sve tri NZD dijagonale u jednom koraku.

Listing 3.5: Pseudokôd algoritma paralelnog transponovanja 2D matrice sa stanovišta blokova matrice koji se transponuju kada P i Q nisu međusobno prosti brojevi (NZD > 1). NZD dijagonala matrice se transponuju simultano.

```
PARALLEL FOR (K = 0; K < NZD; K++) {
   g = MOD(q - p, NZD)
   p_tilda = MOD(p + g, P); q_tilda = MOD(q – g, Q)
   FOR (J = 0, J < NZS/P; J++) {
      K = J /*Odrediti K-tu dijagonalu za transponovanje*/
      WHILE (MOD(K-g, P) != 0) {
         K = MOD(K+Q, NZS)
      }
      FOR (I = 0, I < NZS/Q; I++) {
         [Kopiraj svaki (K:Nb:NZS) dijagonalni blok iz P(p,q) u T1]
         [Pošalji T1 sa P(p,q) ka P(p_tilda + I * GCD, q_tilda – J x GCD)]
         [Kopiraj primljeni T1 u C]
         K = MOD(K+Q, NZS)
      }
   }
}
```

4 Razvoj algoritma za transponovanje distribuiranih 3D matrica

4.1 Problemi manuelne dekompozicije i transponovanja u 3D

Tehnička implementacija teze zahtevala je kreiranje efikasnog algoritma paralelnog transponovanja 3D matrica i primenu datog algoritma u postojećem numeričkom rešenju Krenk-Nikolson propagacije BAK. Da bismo ostvarili zadati cilj bilo je neophodno razviti nekoliko verzija različitih algoritama paralelnog transponovanja 2D i 3D matrica uz upotrebu MPI biblioteke. Sve verzije algoritama prate inkrementalni model životnog ciklusa razvoja softvera, što podrazumeva dokumentaciju, testiranje i repozitorijumsko verziranje preko git-a. Prvi predstavljeni algoritam u verziji 5.0 implementira paralelno transponovanje uz pomoć MPI biblioteke. Algoritmi transponovanja u verzijama 1.0 – 4.0 predstavljaju radne verzije transponovanja i ne uključuju rad sa MPI bibliotekom, te stoga neće biti posebno analizirani.

U ovom poglavlju ćemo napraviti pregled razvoja algoritama transponovanja počevši od verzije 5.0 pa sve do konačnog rešenja zadatog problema.

4.1.1 Projekat MT_v_5.0

Prvi algoritam koji implementira paralelno transponovanje 2D distribuiranih matrica uz pomoć MPI biblioteke je algoritam implementiran u verziji koda 5.0 (projekat MT_v_5.0). Dati algoritam predstavlja programsku implementaciju teorijskih koncepata algoritama datih u referentnom radu [15], a predstavljenih detaljnije u sekcijama 3.1 i 3.2. Projekat MT_v_5.0 se sastoji od sledećih fajlova:

- `matrix_t_mpi_v5.c` – fajl glavnog programa,
- `variables.c` i `variables.h` – fajlovi koji sadrže deklaracije globalnih promenljivih,
- `functions.h` – fajl koji sadrži definicije pomoćnih funkcija,

- `matrix_generator.c` – fajl koji generiše testnu 2D matricu u skladu sa prosleđenim parametrima,
- `matrix_param_reader.h` – fajl koji sadrži funkciju za čitanje konfiguracionog fajla,
- `matrix_params.txt` – fajl koji sadrži različite testne konfiguracije (NZD = 1 i NZD > 1),
- `transpose_prime.c` – fajl koji sadrži funkciju za paralelno transponovanje matrice u slučaju kada je NZD između broja čvorova u vrstama i kolonama 1,
- `transpose_not_prime.c` – fajl koji sadrži funkciju za paralelno transponovanje matrice u slučaju kada je NZD između broja čvorova u vrstama i kolonama > 1,
- `generate_matrix.sh` – pomoćna skripta za generisanje matrice iz komandne linije,
- `run_transpose.sh` – pomoćna skripta za pokretanje paralelnog transponovanja iz komandne linije.
- `run_all.sh` – pomoćna skripta za čitanje konfiguracije, generisanje matrice i pokretanje transponovanja iz komandne linije.

4.1.1.1 Fajl `matrix_t_mpi_v5.c`

Fajl `matrix_t_mpi_v5.c` sadrži u sebi glavni program za paralelno transponovanje 2D matrica. Osnovna funkcionalnost ovog fajla jeste inicijalizacija MPI okruženja, kreiranje Dekartove 2D topologije u skladu sa ilustracijom datom na slici 3.1, kao i određivanje koordinata, ranga i grupe čvora na kojem se kôd izvršava. Važno je naglasiti da MPI model podrazumeva SPMD (jedan program – više podataka) paradigmu izvršavanja [17], tj. isti programski kôd izvršavaju svi čvorovi u klasteru (svaki na datim lokalnim podacima), a razlika u tokovima izvršavanja se postiže uz pomoć grananja, u zavisnosti od ranga ili grupe kojoj pripada određeni čvor. Inicijalizacija MPI okruženja data je u listingu 4.1.

Listing 4.1: Inicijalizacija MPI okruženja i najvažnijih promenljivih u glavnom programu.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &gsize);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);      //preuzmi rank trenutnog cvora

dim[0]=p; dim[1]=q;             //x i y dimenzije sablona
period[0]=1; period[1]=1;       //periodicna mreza
reorder=0;

//napravi 2D Dekartovu topologiju
MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);
MPI_Cart_coords(comm, rank, 2, coord); //dobavi koordinate cvora
```

```
    if (gcd > 1) {
        group = mod_func(coord[1]-coord[0], gcd);
        p_trans = mod_func(coord[0] + group, p);
        q_trans = mod_func(coord[1] - group, q);
```

Osim inicijalizacije promenljivih, zadatak glavnog programa je i učitavanje matrice iz ulaznog fajla. Bitno je napomenuti da u ovoj verziji svi čvorovi imaju kopiju celokupne matrice, ali učitavaju samo one podatke koji im pripadaju, u skladu sa cikličnim blokovskim rasporedom podataka, sličnom onom na slikama 3.4 (NZD = 1) i 3.9 (NZD > 1). Listing 4.2 prikazuje učitavanje blokova matrice.

Listing 4.2: Čitanje blokova matrica na jednom čvoru u cikličnom rasporedu.

```
    matrix_file = fopen("matrix.bin","rb");        //binarno citanje
    if (matrix_file==NULL)
    {
        printf("Matrix file doesn't exist!\n");
        return 1;
    }

    //trazenje offset-a inicijalnog bloka
    first_block_offset = (rank/q)*r*n + (rank%q)*c;
    fseek(matrix_file, first_block_offset*sizeof(double), SEEK_SET);


    for (block_num=0; block_num < number_of_blocks; block_num++) {
        //trazenje offseta trenutnog bloka u fajlu matrice
        current_block_offset = (block_num/block_columns)*p*r*n +
                               (block_num%block_columns)*q*c;
        rewind(matrix_file);
        fseek(matrix_file, first_block_offset*sizeof(double), SEEK_SET);

        fseek(matrix_file, current_block_offset*sizeof(double), SEEK_CUR);

        for (slice_x = 0; slice_x < r; slice_x++) {
            for (slice_y = 0; slice_y < c; slice_y++) {
                fread(&submatrix[submatrix_offset],
                    sizeof(double), 1, matrix_file);
                submatrix_offset++;
            }
            //pomeri pokazivac na novi odsecak istog bloka
            fseek(matrix_file, (n-c)*sizeof(double), SEEK_CUR);
        }
    }
```

Pored cikličnog rasporeda blokova, podaci u okviru blokova (elementi tipa `double`) zauzimaju prostor u memoriji redom u okviru jedne vrste podataka (tzv. „row-major order“), pa je stoga neophodno učitavanje `r` vrsta veličine `c` elemenata tipa `double` i to za svaki blok smešten na pojedinačni čvor. Da bi se ovo postiglo, matrica upisana u fajl se čita u preskocima („stride“) odgovarajuće veličine (`n-c` u kôdu). Ovakvo čitanje je validno pošto zadata matrica može da se posmatra kao jednodimenzionalni niz dužine `m * n` elemenata tipa `double`.

Nakon učitavanja podataka svaki čvor poseduje svoje lokalne blokove, slično ilustraciji datoj na slici 3.7. Preostali deo programa podrazumeva lokalno transponovanje osnovnih elemenata matrice u okviru svakog od blokova, što je prikazano u listingu 4.3.

Listing 4.3: Lokalno transponovanje blokova u okviru jednog čvora.

```
      for (block_num=0; block_num < number_of_blocks; block_num++) {
            for (slice_x = 0; slice_x < c; slice_x++) {
                  for (slice_y = 0; slice_y < r; slice_y++) {
                        submatrix_t[(block_num * r * c) +
                                    (slice_x * r + slice_y)] =
                        submatrix[(block_num * r * c) +
                                  (slice_y * c + slice_x)];
                  }
            }
      }
```

Kao što se može videti iz listinga 4.3, svaki čvor poseduje dva jednodimenzionalna niza za smeštanje podataka, pri čemu jedan niz služi za originalan sadržaj blokova, a drugi za transponovani sadržaj. Drugim rečima, nije realizovano transponovanje u mestu („in-place“), koje bi uštedelo potrošnju memorije. Razlog za ovo postoji u daljoj implementaciji algoritma, koja zahteva dva niza za pravilnu razmenu MPI poruka, a takođe postoje dokazi da transponovanje matrice u mestu dovodi do pada performansi i slabijeg iskorišćenja lokalizovane memorije zbog lošijeg keširanja podataka [18]. Nakon ovoga sledi poziv funkcija za paralelno transponovanje blokova koje su realizovane u posebnim fajlovima.

4.1.1.2 Fajl `transpose_prime.h`

Fajl `transpose_prime.h` sadrži definiciju funkcije `transpose_prime` koja implementira transponovanje u slučaju kada je NZD broja čvorova po vrstama i kolonama jednak jedan. Implementacija u verziji 5.0 ne računa korektno transponovanje između NZS grupe blokova van glavne dijagonale (beli blokovi na slici 3.5). Ovaj problem je ispravljen u verziji koda 7.0 i neće biti posebno razmatran u ovom poglavlju. Transponovanje dijagonalnih NZS grupa blokova (sivi blokovi na slici 3.5) je implementirano korektno. U listingu 4.4 prikazan je relevantan deo funkcije `transpose_prime`.

Listing 4.4: Relevantan deo funkcije `transpose_prime`.

```
for (lcm_r = 0; lcm_r < lcm_rows; lcm_r++) {
      for (lcm_c = 0; lcm_c < lcm_columns; lcm_c++) {
            for (lcm_local_i = 0; lcm_local_i < blocks_per_lcm_row;
                 lcm_local_i++) {
                  for (lcm_local_j = 0; lcm_local_j < blocks_per_lcm_column;
                 lcm_local_j++) {
                        local_i = lcm_r * blocks_per_lcm_row + lcm_local_i;
                        local_j = lcm_c * blocks_per_lcm_column + lcm_local_j;
```

```
                        local_to_global_ij(coord[0], coord[1],
                                           stride_p, stride_q,
                                           local_i, local_j,
                                           &global_i, &global_j);

                        find_processor_of_block(global_j, global_i,
                                                stride_p,stride_q,
                                                &find_proc_ij[0],
                                                &find_proc_ij[1]);

                        MPI_Cart_rank(comm, find_proc_ij, &send_to_rank);

                        sdispl[send_to_rank] = local_i * blocks_per_lcm_column
                                             * lcm_columns * r * c +
                                               local_j * r * c;
                        scounts[send_to_rank] = r * c;

                        }
                  }
                  MPI_Alltoallv(submatrix_t, scounts, sdispl, MPI_DOUBLE,
                                submatrix, scounts, sdispl, MPI_DOUBLE, comm);
            }
      }
```

Prva dvostruka petlja u okviru funkcije `transpose_prime` ima za cilj prolazak kroz sve NZS grupe blokova (sivi i beli blokovi na slici 3.5). U slučaju kada je $P = 2$, $Q = 3$, m = 48, n = 60, r = 4 i c = 5, svaki čvor će posedovati 2 * 2 * NZS = 24 bloka originalne matrice (jer je NZS = 6), odnosno četiri NZS grupe blokova od kojih svaka u sebi sadrži šest blokova originalne matrice. Za svaku NZS grupu blokova se ponavlja ista procedura, odnosno u dve unutrašnje petlje se prolazi kroz sve blokove u grupi (njih šest). Za svaki blok se određuju njegove lokalne koordinate na čvoru, a nakon toga i njegove globalne koordinate uz pomoć funkcije `local_to_global_ij`. Data funkcija računa globalnu koordinatu iz lokalne uz pomoć Dekartovih koordinata čvora koji je poziva i rastojanja („stride“) susednih blokova smeštenih na istom čvoru u cikličnom rasporedu. Nakon što dobijemo globalne koordinate bloka (npr. lokalni blok (0, 1) sa čvora 0 na slici 3.5 ima globalne koordinate (0, 3)), određuju se koordinate čvora kome taj blok treba poslati u okviru transponovanja. Ovo se vrši prosleđivanjem zamenjenih globalnih koordinata bloka (zbog transponovanja) funkciji `find_processor_of_block`. Koordinate odgovarajućeg čvora omogućuju funkciji `MPI_Cart_rank` da pronađe rang čvora kojem pripada blok koji se šalje. Prethodna procedura je neophodna kako bi se popunio niz udaljenosti („offset“) svakog bloka od početka niza podataka `submatrix_t`. Dati niz udaljenosti nosi naziv `sdispl`. Takođe bitno je znati i broj osnovnih elemenata (tipa double) svakog bloka koji se šalje. Dati niz broja osnovnih elemenata nosi naziv `scounts`, a vrednost svakog elementa niza je `r * c`.

Prethodne nizove `sdispl` i `scounts` smo uveli kako bi nakon završetka dve unutrašnje petlje mogli da pozovemo funkciju `MPI_Alltoallv`. Svrha poziva ove funkcije jeste razmena podataka između svih čvorova u topologiji, odnosno u konkretnom slučaju razmena šest blokova između šest čvorova. Odatle potiče i intuitivan naziv funkcije („all to all“ ili svako šalje i prima od

svakoga). Ovo rešenje je u skladu sa algoritmom opisanim u poglavlju 3.1 koji podrazumeva potpunu razmenu blokova. Bitno je naglasiti da se blokovi šalju i primaju na iste memorijske lokacije (npr. na mesto poslatog bloka (0, 3) biće smešten sadržaj primljenog bloka (3, 0)). Razlika funkcije `MPI_Alltoallv` u odnosu na funkciju `MPI_Alltoall` je u tome što prva funkcija dozvoljava proizvoljna odstojanja blokova koji se šalju i primaju u odnosu na početak niza podataka. Odstojanja se u tom slučaju nalaze u nizu `sdispl`. Poziv funkcije `MPI_Alltoallv` je neophodan s obzirom da niz podataka koji sadrži sve blokove (`submatrix_t`) nema linearan raspored blokova po NZS grupama, već je potrebno preskočiti blokove pojedinih NZS grupa u procesu slanja i prijema.

4.1.1.3 Fajl `transpose_not_prime.h`

Fajl `transpose_not_prime.h` sadrži definiciju funkcije `transpose_not_prime` koja implementira transponovanje u slučaju kada je NZD broja čvorova po vrstama i kolonama veći od jedan.

Listing 4.5: Relevantan deo funkcije `transpose_not_prime`.

```
for (proc_j = 0; proc_j < block_rows; proc_j++) {
            for (proc_i = 0; proc_i < block_columns; proc_i++) {
                  //find out the rank of the process above and below
                  //nadji rank cvora iznad i ispod
                  send_proc_ij[0] = p_trans + proc_i * gcd;
                  send_proc_ij[1] = q_trans - proc_j * gcd;

                  receive_proc_ij[0] = p_trans - proc_i * gcd;
                  receive_proc_ij[1] = q_trans + proc_j * gcd;

                 //modularno izracunaj i,j preko p,q
                  send_proc_ij[0] = mod_func(send_proc_ij[0], p);
                  send_proc_ij[1] = mod_func(send_proc_ij[1], q);

                  receive_proc_ij[0] = mod_func(receive_proc_ij[0], p);
                  receive_proc_ij[1] = mod_func(receive_proc_ij[1], q);

                  MPI_Cart_rank(comm, send_proc_ij, &send_to_rank);
                  MPI_Cart_rank(comm, receive_proc_ij, &receive_from_rank);

                  found_sb = 0;
                  found_rb = 0;

                  for (local_i = 0; local_i < block_rows; local_i++) {
                      for (local_j = 0; local_j < block_columns; local_j++) {
                              if (send_block_used[local_i * block_columns +
                                  local_j] == 1)
                                    continue;
                              local_to_global_ij(coord[0], coord[1],
                                                 stride_p, stride_q,
                                                 local_i, local_j,
```

```
                                    &global_i, &global_j);
                find_processor_of_block(global_j, global_i,
                                        stride_p, stride_q,
                                        &find_proc_ij[0],
                                        &find_proc_ij[1]);

                if ((find_proc_ij[0] == send_proc_ij[0] &&
                     find_proc_ij[1] == send_proc_ij[1])) {
                    if (send_block_used[local_i *
                        block_columns + local_j] == 0) {
                      send_block_number = local_i *
                                          block_columns +
                                          local_j;
                      send_block_used[send_block_number] =
                                          1;
                      found_sb = 1;
                    }
                }
                if (found_sb) break;
            }
            if (found_sb) break;
        }

        for (local_j = 0; local_j < block_columns; local_j++) {
            for (local_i = 0; local_i < block_rows; local_i++) {
                if (r_block_used[local_i * block_columns +
                    local_j] == 1) {
                      continue;
                }
                local_to_global_ij(coord[0], coord[1],
                                   stride_p, stride_q,
                                   local_i, local_j,
                                   &global_i, &global_j);
                find_processor_of_block(global_j, global_i,
                                        stride_p, stride_q,
                                        &find_proc_ij[0],
                                        &find_proc_ij[1]);

                if ((find_proc_ij[0] == receive_proc_ij[0] &&
                     find_proc_ij[1] == receive_proc_ij[1])) {

                    if (r_block_used[local_i *
                        block_columns + local_j] == 0) {
                      receive_block_number = local_i *
                      block_columns + local_j;

                      r_block_used[receive_block_number] =
                                 1;
                      found_rb = 1;
                    }
                }
                if (found_rb) break;
            }
            if (found_rb) break;
        }

        MPI_Irecv(&submatrix[receive_block_number * r * c],
```

```
                                r*c, MPI_DOUBLE, receive_from_rank, 0, comm,
                                &receive_requests[proc_j * block_columns +
                                proc_i]);

                MPI_Isend(&submatrix_t[send_block_number * r * c],
                          r*c, MPI_DOUBLE, send_to_rank, 0, comm,
                          &send_requests[proc_j * block_columns + proc_i]);

                MPI_Wait(&send_requests[proc_j * block_columns + proc_i],
                         &send_statuses[proc_j * block_columns + proc_i]);

        }
    }

    //sacekaj da se zavrse prijemi
    for (proc_j = 0; proc_j < block_rows; proc_j++) {
        for (proc_i = 0; proc_i < block_columns; proc_i++) {
            MPI_Wait(&receive_requests[proc_j * block_columns + proc_i],
             &receive_statuses[proc_j * block_columns + proc_i]);
        }
    }
```

Algoritam transponovanja u slučaju funkcije `transpose_not_prime` prati cikličnu raspodelu blokova ilustrovanu na slici 3.9. Svaki od čvorova poseduje svoj podskup blokova iz originalne matrice, pri čemu se ne obraća pažnja na NZS grupe blokova, za razliku od algoritma kada je NZD = 1. Podrazumevamo da se razmena u ovom slučaju završi u NZS/NZD koraka, što u slučaju kada je P = 4 i Q = 6 znači razmenu blokova u šest koraka. Značajna razlika u odnosu na algoritam kada je NZD = 1 je u tome što čvorovi ne razmenjuju podatke sa svim čvorovima, već samo sa pojedinim čvorovima koji su udaljeni NZD čvorova od trenutnog čvora po obe ose (slika 3.10). Time je i onemogućena upotreba funkcije `MPI_Alltoallv` i stoga se umesto kompletne vrši selektivna razmena uz pomoć funkcija za asinhrono (neblokirajuće) slanje i prijem, `MPI_Isend` i `MPI_Irecv`.

Dve petlje kojima počinje funkcija predstavljaju petlje prolaska kroz sve blokove matrice smeštene na jednom čvoru. Potom se za svaki blok određuje čvor kome se šalju i od koga se primaju podaci, i to pomoću modularne aritmetike opisane u okviru algoritma datog u sekciji 3.2. Nakon toga sledi par dvostrukih unutrašnjih petlji. Zadatak para dvostrukih petlji je pronalaženje odgovarajućeg bloka za slanje odgovarajućem čvoru. Data procedura je netrivijalna jer blokovi u okviru niza `submatrix_t` nisu složeni linearno u redosledu slanja, a takođe ni blokovi u okviru niza `submatrix` nisu složeni linearno u redosledu prijema. U retkim slučajevima, kod nekih čvorova može da se dogodi da se redosled bloka u memoriji (nizu `submatrix_t` ili `submatrix`) poklapa sa redosledom slanja i prijema, ali to nije pravilo. Drugim rečima, za svaki blok na svakom čvoru potrebno je pronaći koji je u redosledu slanja i prijema blokova datog čvora. Ovo je veoma bitno jer je algoritam u referentnom radu [15] koncipiran tako da slanja i prijemi sa pojedinih čvorova na pojedine čvorove funkcionišu potpuno sinhronizovano, u simetričnim koracima. Ovo znači da ukoliko bi se blok van redosleda poslao nekom čvoru, matrica bi bila transponovana pogrešno i podaci posledičnog numeričkog računa ne bi bili korektni.

Postavlja se i pitanje pojave mrtve petlje („deadlock“), gde čvor koji prima blok beskonačno čeka poruku čvora koji šalje, a čvor koji šalje ne može da pošalje odgovarajući blok, jer i sam čeka prijem. Takva mogućnost je sprečena implementacijom algoritma iz sekcije 3.2, koji garantuje simetriju operacija slanja i prijema. Takođe, mrtva petlja se eliminiše i asinhronim slanjem (`ISend`) i prijemom (`IRecv`) sa i na dva odvojena memorijska niza (`submatrix_t` i `submatrix`), kao i na odvojene memorijske lokacije u okviru datih nizova, označene brojačima blokova (`send_block_number` i `receive_block_number`). Ovo u praksi znači da nema komunikacionih i memorijskih nepodudarnosti. Jedino je bitno da se odgovarajućem čvoru pošalje odgovarajući blok, što algoritam i obezbeđuje.

4.1.2 Projekat MT_v_6.0

Projekat MT_v_6.0 zadržao je istu funkcionalnost kao projekat verzije 5.0. Zbog toga verzija kôda 6.0 neće biti posebno opisana. Sledi lista funkcionalnosti koje su u međuvremenu implementirane u kôdu:

- Dodati su delovi „bash“ skripta i funkcija za testiranje, kojima se vrši automatsko biranje slučajnog čvora i bloka na njemu kao i provera da li je taj blok validno transponovan.
- Promena određenih promenljivih: uvedeni su nizovi umesto pojedinačnih vrednosti.

4.1.3 Projekat MT_v_7.0

Projekat MT_v_7.0 sadrži istu funkcionalnost kao i projekti 5.0 i 6.0. Zbog toga verzija kôda 7.0 neće biti posebno opisana. Sledi lista funkcionalnosti koje su u međuvremenu implementirane u kôdu:

- Korigovano je transponovanje nedijagonalnih NZS blokova iz algoritma 5.0 (beli blokovi na slici 3.5).
- Ceo program je realizovan preko funkcija.
- Postoje posebne funkcije za inicijalizaciju i deinicijalizaciju promenljivih.
- Matrica se više ne čita sa svih čvorova već samo sa nultog čvora („master node“) iz posebne funkcije `read_and_scatter`. Funkcija šalje odgovarajuće blokove matrice sa nultog čvora svakom od čvorova uz pomoć MPI funkcije `MPI_Scatter`.

- Nakon transponovanja delovi matrice se šalju sa svih čvorova nultom čvoru iz posebne funkcije `gather_and_write.` Funkcija šalje odgovarajuće blokove transponovane matrice sa svih čvorova nultom čvoru uz pomoć MPI funkcije `MPI_Gather`.
- Kreirana je posebna funkcija za čitanje i upis matrice u fajl, kao i funkcija koja poredi dve matrice iz dva različita fajla. Na taj način lako se može utvrditi validnost transponovanja poredeći matrice dobijene sekvencijalnim i paralelnim algoritmom.

4.1.4 Projekat MT_v_7.5

Projekat MT_v_7.5 sadrži istu funkcionalnost kao i projekti 5.0, 6.0 i 7.0. Zbog toga verzija kôda 7.5 neće biti posebno opisana. Sledi lista funkcionalnosti koje su u međuvremenu implementirane u kôdu:

- U okviru funkcije `compare_matrices` uvedeni su pozivi Intel MKL funkcija [19]. Dati pozivi služe za proveru validnosti MPI transponovanja u skladu sa sledećim primerom. Ukoliko uzmemo da je originalna matrica $A=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$, onda važe sledeće relacije:

1. $B=A\cdot A^T \Rightarrow \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\cdot\begin{bmatrix}1 & 3\\ 2 & 4\end{bmatrix}=\begin{bmatrix}5 & 11\\ 11 & 25\end{bmatrix}$,
2. $C=B^T=\begin{bmatrix}5 & 11\\ 11 & 25\end{bmatrix}$,
3. $D=B-C^T=\begin{bmatrix}5 & 11\\ 11 & 25\end{bmatrix}-\begin{bmatrix}5 & 11\\ 11 & 25\end{bmatrix}=\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix}$,
4. suma svih elemenata matrice D je jednaka nuli.

Kôd provere inspirisane gornjim primerom je dat u listingu 4.6.

Listing 4.6: Kôd za proveru MPI transponovanja uz pomoć Intel MKL funkcija.

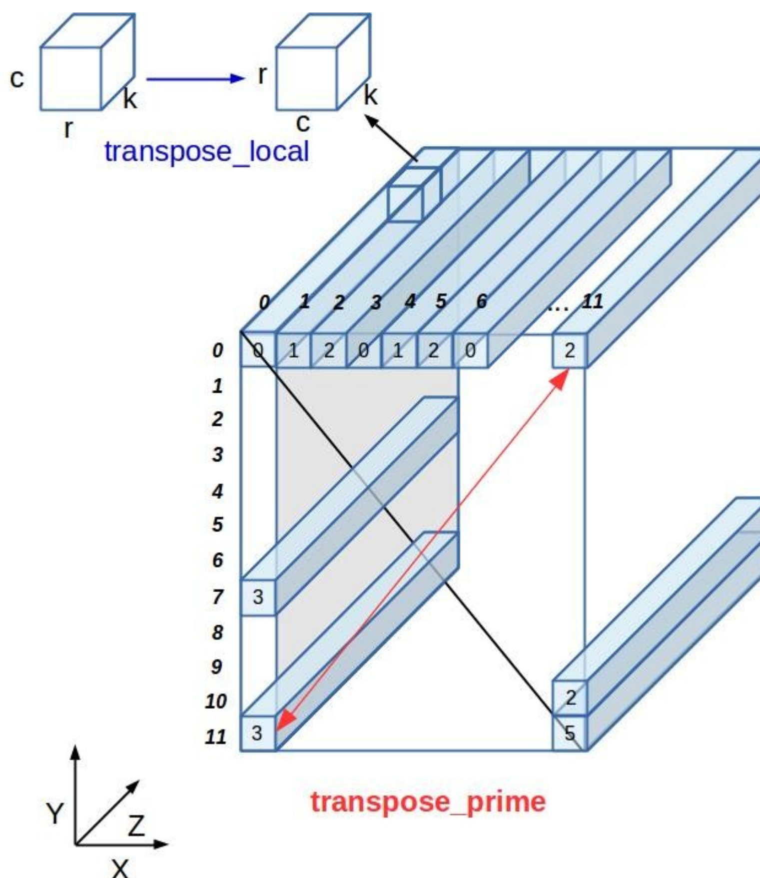
```
//MKL testiranje validnosti transponovanja
double * product = (double *) mkl_malloc(m*m*sizeof(double), 64);
double * product_trans = (double *) mkl_malloc(m*m*sizeof(double), 64);
double * p_ptrans_sum = (double *) mkl_malloc(m*m*sizeof(double), 64);
//product = op(matrix) * op(trans_matrix) (b = aaT)
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, m, n, 1, matrix, n,
            trans_matrix, m, 0, product, m);
//transponuj product u product_trans = productT (c = bT)
mkl_domatcopy('R', 'T', m, m, 1, product, m, product_trans, m);
//oduzmi product_trans transponovano od product (d = b - cT)
MKL_Domatadd('R', 'N', 'T', m, m, 1, product, m, -1, product_trans, m,
              p_ptrans_sum, m);
//saberi sve elemente d
double sum = cblas_scasum(m * m, p_ptrans_sum, 1);
```

Prethodni kôd verifikuje da je matrica koja se nalazi u baferu `trans_matrix` validno transponovana matrica iz originalnog bafera `matrix`, što još jedanput potvrđuje korektnost MPI transponovanja.

4.1.5 Projekat MT_v_8.0

Projekat MT_v_8.0 sadrži proširenje funkcionalnosti u odnosu na projekte 5.0 – 7.5. Proširenje se ogleda u tome što se u ovom projektu po prvi put implementira 3D transponovanje, bazirano na 2D transponovanju, primenjenom u prethodnim projektima.

Kao što se može videti sa slike 4.1, algoritam transponovanja MT_v_8.0 se oslanja direktno na primer raspodele blokova i transponovanja u slučaju kada je NZD = 1 (slika 3.5), tj. raspodela blokova ostaje ciklična. Koncept 3D transponovanja realizovan u algoritmu MT_v_8.0 se može zamisliti kao 2D transponovanje (NZD =1) algoritma MT_v_7.5 ponovljeno `Nz` puta, pri čemu je `Nz` broj diskretizacionih tačaka po Z osi.



Slika 4.1: Ideja 3D transponovanja u algoritmu MT_v_8.0.

Ono što je neophodno u realizaciji ovakvog algoritma je ravnomerna raspodela podataka iz originalne 3D matrice po različitim čvorovima klastera. Za ovo je zadužen nulti čvor, koji mora da prođe („sweep“) kroz celokupnu originalnu 3D matricu zapisanu u fajl. Pri tome, nulti čvor čita i pakuje u bafer sve slojeve pojedinačnih blokova svakog od čvorova kojima šalje podatke, kao što je prikazano na slici 4.1. Sve ovo se radi kao priprema za poziv funkcije `MPI_Scatter`. Ovakvu funkcionalnost smo mogli da implementiramo i na samim čvorovima, pod uslovom da svi poseduju lokalnu kopiju 3D matrice, za šta znamo da nije realistično za rešenje koje razvijamo, zbog velike željene rezolucije mreže u Krenk-Nikolson algoritmu. Osim raspodele podataka po čvorovima, ključne funkcionalnosti ovog algoritma su lokalno (funkcija `transpose_local`) i globalno (`transpose_prime`) transponovanje blokova, što je ilustrovano na slici 4.1. Sledi bliži opis obe funkcije.

Telo funkcije `transpose_local` je dato u listingu 4.7.

Listing 4.7: Telo funkcije `transpose_local`.

```
int dimension[3][3] = {{c, r, k}, {c, k, r}, {r, k, c}};        //XY, XZ, YZ

for (blocks_z_cnt=0; blocks_z_cnt<blocks_z_axis; blocks_z_cnt++) {
      for (blocks_x_cnt=0; blocks_x_cnt<blocks_x_axis; blocks_x_cnt++)    {
         for (blocks_y_cnt=0; blocks_y_cnt<blocks_y_axis; blocks_y_cnt++) {
                  for (sl_ijk[2]=0; sl_ijk[2]<dim[td][2]; sl_ijk[2]++){
                        for (sl_ijk[0]=0; sl_ijk[0]<dim[td][0]; sl_ijk[0]++) {
                              for (sl_ijk[1]=0; sl_ijk[1] < dim[td][1];
                                   sl_ijk[1]++) {
                                    submatix_trans[(blocks_z_cnt * in_layer
                                                  * r * c * k) +
                                   (blocks_x_cnt * blocks_y_axis * r * c * k) +
                                   (blocks_y_cnt * r * c * k) +
                                   (sl_ijk[2] * dim[td][1] * dim[td][0]) +
                                   (sl_ijk[0] * dim[td][1] + sl_ijk[1])] =
                                    submatrix_orig[(blocks_z_cnt * layer * r *
                                                    c * k) +(blocks_x_cnt *
                                                    blocks_y_axis * r * c * k)
                                                 + (blocks_y_cnt * r * c * k)
                                                 + (sl_ijk[2] * dim[td][1] *
                                                    dim[td][0]) + (sl_ijk[1] *
                                                    dim[td][0] + sl_ijk[0])];
                              }
                        }
                  }
            }
      }
}
```

Kôd u listingu 4.7 vrši lokalno transponovanje u okviru jednog 3D bloka veličine `r * c * k` osnovnih elemenata matrice. Kao što se može primetiti, potreban je veliki broj petlji (šest ugnježdenih petlji) da bi se obavilo lokalno transponovanje blokova u okviru jednog čvora. Takođe, potrebno je pažljivo rukovanje različitim brojačima i udaljenostima („offset“) kako bi se odabrali pravi elementi za transponovanje pojedinačnih 2D slojeva u okviru 3D bloka. Na početku je

funkcija zamišljena tako da podržava transponovanje slojeva po sve 3 dimenzije (X-Y, X-Z i Y-Z). Međutim, u krajnjoj realizaciji smo od toga odustali jer smo zaključili da implementacija transponovanja u X-Z i Y-Z ravnima nije neophodna za prirodu numeričkog problema koji rešavamo. Takvo transponovanje je i komplikovanije za implementaciju zbog rasporeda podataka u memoriji po X osi („row-major order“), što bi zahtevalo dodatne skokove kroz memoriju kako bi se pristupilo podacima za transponovanje po Z osi.

Pored lokalnog, realizovano je i globalno transponovanje uz pomoć funkcije `transpose_prime`, date u listingu 4.8.

Listing 4.8: Telo izmenjene funkcije `transpose_prime`.

```
    for (layer_number = 0; layer_number < blocks_z_axis; layer_number++) {
            for (lcm_r = 0; lcm_r < lcm_rows; lcm_r++) {
                    for (lcm_c = 0; lcm_c < lcm_columns; lcm_c++) {
                            for (lcm_loc_i = 0; lcm_loc_i < lcm_row; lcm_loc_i++) {
                                    for (lcm_loc_j = 0; lcm_loc_j < lcm_column;
                                         lcm_loc_j++) {
                                      loc_block_ij[0] = lcm_r * lcm_row +
                                                        lcm_loc_i;
                                      loc_block_ij[1] = lcm_c * lcm_column +
                                                        lcm_loc_j;
                                      //swap quadrants for receiving blocks
                                      r_loc_ij[0] = lcm_c * lcm_row + lcm_loc_i;
                                      r_loc_ij[1] = lcm_r * lcm_column +
                                                    lcm_loc_j;

                                      local_to_global_ij(coord, stride_PQ,
                                              loc_block_ij, global_block_ij);
                                      swap[0] = global_block_ij[1];
                                      swap[1] = global_block_ij[0];
                                      find_processor_of_block(swap, stride_PQ,
                                                        find_proc_ij);
                                      MPI_Cart_rank(comm, find_proc_ij,
                                                    &send_to_rank);

                                      sdispl[send_to_rank] = layer_number *
                                                 blocks_per_lcm_column
                                               * blocks_per_lcm_row * r * c * k
                                               + local_block_ij[0] *
                                                 blocks_per_lcm_column *
                                                 lcm_columns * r * c * k
                                               + local_block_ij[1] * r * c * k;
                                      rdispl[send_to_rank] = layer_number *
                                                 blocks_per_lcm_column
                                               * blocks_per_lcm_row * r * c * k
                                               + r_local_ij[0] *
                                                 blocks_per_lcm_column
                                               * lcm_columns * r * c * k
                                               + r_local_ij[1] * r * c * k;
                                       //1 3D block
                                      scounts[send_to_rank] = r * c * k;
                                    }
                            }
                            MPI_Alltoallv(submatrix_t, scounts, sdispl, MPI_DOUBLE,
```

```
                              submatrix, scounts, rdispl, MPI_DOUBLE, comm);
                  }
            }
    }
```

U listingu 4.8 se može uočiti korišćenje velikog broja promenljivih neophodnih za tačno lociranje pojedinačnih blokova za transponovanje.

Sledi opis pozitivnih i negativnih osobina projekta MT_v_8.0.

Pozitivne osobine:

- Omogućeno je 3D transponovanje uz pomoć MPI biblioteke po prvi put.
- Rešen je problem transponovanja nedijagonalnih NZS blokova, koji je bio prisutan u algoritmu iz poglavlja 3.1 (beli blokovi na slici 3.5) .

Negativne osobine:

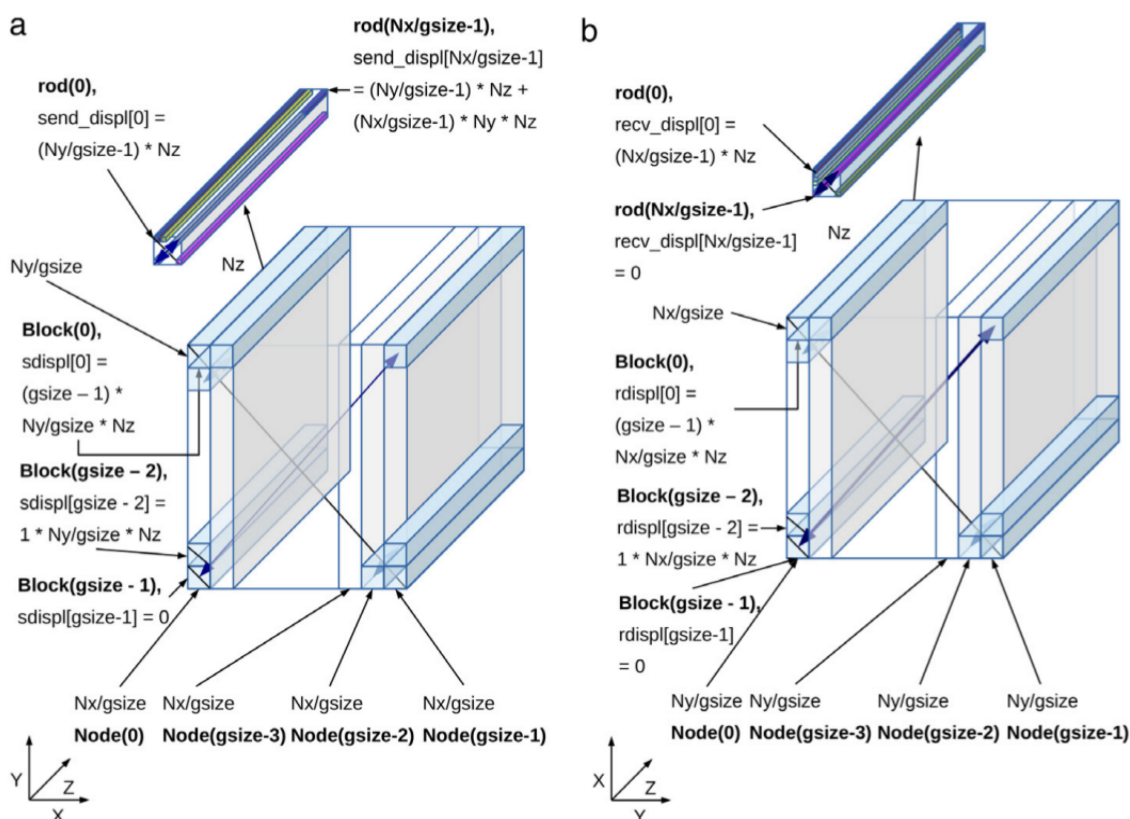
- Potreban je veliki broj petlji, promenljvih i aritmetičkih operacija da bi se transponovao jedan 3D blok, što stvara povećanu mogućnost pojave grešaka.
- Prisutan je veliki broj poziva funkcije `MPI_Alltoallv,` čime se povećava komunikaciono opterećenje klastera (tzv. „communication overhead“) i potencijalno usporava čitav numerički algoritam koji bi morao da ima zadovoljavajuće ubrzanje i skaliranje.
- Algoritam MT_v_8.0 je realizovan podrazumevajući da su podaci koji se transponuju smešteni redom u memoriji po X osi, dok su u stvarnom numeričkom algoritmu podaci smešteni redom po Z osi.
- Algoritam je vezan striktno za blok-cikličnu raspodelu podataka. Ovakva raspodela podataka ne odgovara numeričkom algoritmu koji zahteva kompletnu prisutnost jednog od opsega po X ili Y osi na pojedinačnim čvorovima u svakoj iteraciji algoritma.
- Nije implementiran algoritam u slučaju kada je NZD > 1, jer to nije neophodno za Krenk-Nikolson algoritam u kom bi trebalo da se primeni. U slučaju kada je NZD > 1, podaci duž bilo koje ose su distribuirani na više čvorova, odnosno pojedinačni čvorovi ne sadrže kompletan opseg ni po jednoj osi. Ovo bi značajno smanjilo efikasnost Krenk-Nikolson algoritma, pa zato nije implementirano.

Na osnovu prethodnih verzija algoritma za paralelno transponovanje distribuiranih 3D matrica smo razvili optimalno rešenje koje će biti predstavljeno u sledećoj sekciji.

4.2 Projekat GP-SCL-HYB

U prethodnoj sekciji predstavljena je evolucija algoritma za transponovanje kroz nekoliko verzija koje se odnose na distribuirane 2D i 3D matrice. U projektu MT_v_8.0 je implementirano 3D transponovanje, ali ono poseduje nedostatke i zahteva dodatnu optimizaciju i prilagođavanje postojećem Krenk-Nikolson algoritmu za propagaciju u realnom i imaginarnom vremenu. Optimizacija 3D transponovanja postignuta je kreiranjem korisničkih MPI tipova podataka („user datatypes“), kao i asinhronom razmenom poruka preko metoda `MPI_Isend` i `MPI_Irecv` [71].

Pre opisa kreiranih MPI tipova podataka i konačnog algoritma 3D transponovanja, bitno je naglasiti da je za razliku od distribucija primenjenih u projektima MT_v_5.0 – MT_v_8.0 konačno rešenje zahtevalo distribuciju podataka talasne funkcije koja je blokovska, a ne blok-ciklična. Ovo u praksi znači da svaki čvor na sebi sadrži jedan sloj („slab“) podataka duž X ose, dok sadrži celokupne podatke po Y i Z osi (slika 4.2) [71]. Ovo je značajno drugačije rešenje od blok-ciklične distribucije, gde čvorovi sadrže samo delove opsega po Y osi.



Slika 4.2: Struktura podataka talasne funkcije BAK u primenjenom algoritmu. Podaci su distribuirani tako da čvorovi poseduju deo podataka duž X ose i kompletne podatke po Y i Z osi. Slike prikazuju i parametre za kreiranje MPI indexed tipa u slučaju pošiljaoca (a) i primaoca (b). [71]

Ovakvo rešenje omogućava potpuno paralelno izvršavanje centralnih funkcija numeričkog algoritma, `calcluy` (sekcija 2.1.8.9), `calcluz` (sekcija 2.1.8.10) i `calclnu` (sekcija 2.1.8.7) na različitim MPI čvorovima klastera, bez ikakvog komunikacionog kašnjenja, s obzirom da čvorovi sadrže celokupne podatke po Y i Z osi u pojedinačnim X podopsezima. Međutim, kod izvšavanja funkcija `caclux` (sekcija 2.1.8.8), `calcrms` (sekcija 2.1.8.6) i `calclmuen` (sekcija 2.1.8.5) nastaje problem, s obzirom da je za njihovo izvršavanje neophodan celokupni X opseg podataka. Odatle potiče potreba za malom dodatnom komunikacijom tokom izvršavanja funkcije `calcrms` (podaci se mogu obraditi na lokalnom nivou svakog čvora, a onda se parcijalni rezultati mogu skupiti na jednom od čvorova i obraditi u finalni rezultat), dok je u slučaju funkcija `calclux` i `calcmuen` neophodno i transponovanje podataka u X-Y ravni. Transponovanje obezbeđuje čvorovima sve neophodne podatke po X osi kako bi se izvršile funkcije `calclux` i `calcmuen`. Pošto je transponovanje neophodno u svakoj iteraciji numeričkog algoritma, to podrazumeva neizbežno povećanje komunikacionog kašnjenja simulacije [71].

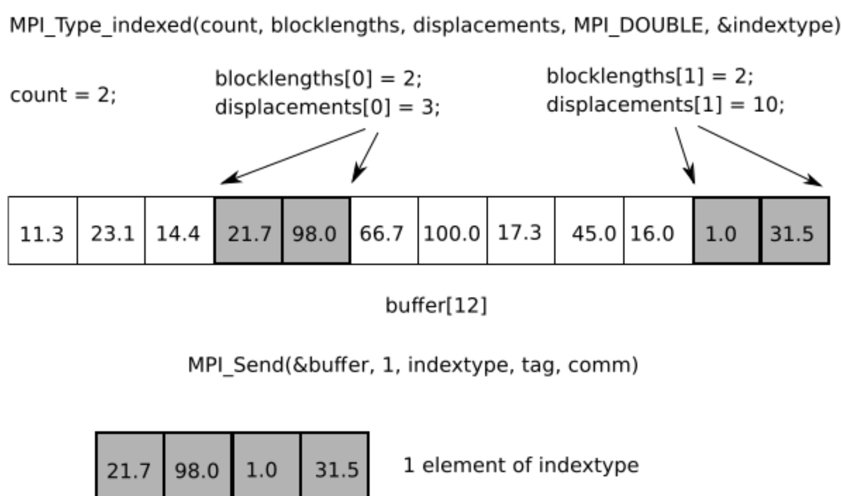
Implementacija transponovanja u projektu GP-SCL-HYB [71] podrazumeva da je NZD između broja čvorova u vrstama i kolonama mreže čvorova jednak jedan. Konkretno, broj čvorova po vrstama mreže mora biti $P = 1$, a broj čvorova u kolonama mreže mora da bude delilac broja tačaka talasne funkcije duž X ose (`Nx`), slika 4.2. Ovo ograničenje mreže čvorova proističe pre svega iz činjenice da podaci o talasnoj funkciji po dve ose moraju da budu kompletni u svakom trenutku na jednom čvoru (X-Z ili Y-Z), što ne dozvoljava deljenje tih opsega na više čvorova. Ovakvo deljenje bi bilo neizbežno u slučaju da je NZD > 1, a nekada i u slučaju kada je NZD = 1 (npr. $P = 2$, $Q = 3$). Takođe, ovo ograničenje potiče i iz činjenice da generisanje indeksiranih MPI tipova podataka podrazumeva određena pravila koja ne dozvoljavaju potpuno proizvoljan memorijski raspored podataka, što čini implementaciju u slučaju NZD > 1 veoma komplikovanom [71]. Kao što se može videti sa slike 4.2, u ovom slučaju svaki od čvorova poseduje sloj („slab“) X opsega i celokupne Y i Z opsege za dati sloj X ose. U okviru jednog sloja nalazi se više blokova, označenih podebljanim tekstom **Block** na slici 4.2. Svaki čvor poseduje `gsize` ovakvih blokova, što praktično znači da će svaki od blokova biti poslat jednom od čvorova u klasteru (slično pozivu `MPI_Alltoall`).

Da bi se obavila blokovska distribucija podataka (kao i transponovanje preko indeksnih MPI tipova) neophodno je promeniti strukturu podataka talasne funkcije [71]. Naime, podaci talasne funkcije `psi` na jednom čvoru se moraju čuvati kontinualno u jednodimezionalnom nizu. Pri ovakvoj raspodeli podataka *Z* dimenzija ima razmak („stride“) 1, *Y* dimenzija ima razmak `Nz`, dok *X* dimenzija ima razmak `Ny * Nz`. Pod razmakom se podrazumeva razmak između dva susedna elementa duž jedne ose, meren u broju elemenata osnovnog tipa podataka iz kog je sastavljen jednodimenzionalni niz (ovde tip `double`). Ovo je drugačije od prethodne implementacije [11], koja je podrazumevala da je talasna funkcija smeštena u eksplicitni trodimenzionalni niz.

Osim transponovanja blokova između čvorova, podaci u okviru jednog bloka se takođe moraju transponovati (redistribuirati) [71]. Ilustracija transponovanja je data na slici 4.2, gde na levoj strani jedan blok podataka ima veličinu `(Nx/gsize) * (Ny/gsize) * Nz` osnovnih elemenata tipa `double`. Ovakav blok podataka reprezentuje jedan 3D blok, razmenjen između dva

čvora klastera (preko jedne neblokirajuće `MPI_Isend` i jedne `MPI_Ireceive` operacije). Takođe, ovakav blok u sebi zadrži `(Nx/gsize) * (Ny/gsize)` duži („rods“) podataka po Z osi, od kojih svaka sadrži `Nz` brojeva tipa `double`. Ove duži moraju da budu transponovane lokalno unutar bloka koji se transponuje globalno sa drugim blokom. Ovo zapravo znači da se u algoritmu moraju odigrati dva nivoa transponovanja [71]. Na nivou jednog bloka, duži moraju da se transponuju međusobno (ilustrovano u gornjem levom uglu slike 4.2(a) za indeksirani tip podataka za slanje, kao i gornjem levom uglu slike 4.2(b) za indeksirani tip podataka za prijem). Drugi nivo transponovanja je transponovanje 3D blokova između različitih čvorova, što je predstavljeno plavim strelicama koje povezuju blokove na slici 4.2.

Slika 4.3 sadrži ilustraciju kreiranja korisničkog MPI indeksiranog tipa podataka, koji služe za transponovanje duži podataka unutar jednog 3D bloka [71].



Slika 4.3: Kreacija korisničkog MPI tipa podataka `indextype` *uz pomoć funkcije* `MPI_Type_indexed` [71].

Na slici 4.3 promenljiva `count` predstavlja broj blokova, niz `blocklengths` sadrži dužine svakog od blokova, dok niz `displacements` sadrži odstojanja svakog od blokova od početka odgovarajuće strukture podataka za koju se pravi indeksni tip. Na primer, ukoliko se šalje niz brojeva u dvostrukoj preciznosti (označen kao `buffer` na slici) uz pomoć operacije `MPI_Send`, a pri tome se tip podataka za slanje postavlja za `indextype`, struktura podataka koja se šalje se interpretira kao blok-distribuirana struktura podataka [17] specificirana u trenutku kreiranja indeksnog tipa podataka [71]. U primeru na slici, prilikom slanja iz bafera, od 12 elemenata će biti poslati samo dva bloka podataka (`count` = 2), pri čemu je udaljenost prvog bloka od početka bafera tri, dok je dužina tog bloka dva. Udaljenost drugog bloka od početka bafera je 10, a dužina takođe dva.

Kreiranjem indeknog tipa MPI podataka se efektivno mogu birati odgovarajuće duži unutar jednog bloka za slanje ili prijem u procesu transponovanja i zameniti u memorijskoj strukturi jednog bloka („swap“). Važno je naglasiti da ovakva operacija zahteva dva memorijska bafera kako ne bi došlo do prepisivanja lokacija prilikom transponovanja duži [71]. S obzirom da duži prilikom

slanja i prijema imaju drugačija odstojanja od početka bafera jednog 3D bloka (promenljive `send_displ` i `recv_displ` na slici 4.2), neophodno je kreirati dva odvojena korisnička indeksna tipa podataka, jedan za slanje i drugi za prijem [71]. Takođe je bitno primetiti da se za svaki 3D blok koji se šalje ili prima mora znati odstojanje od početka bafera svih blokova (bafera celokupne talasne funkcije na datom čvoru). Ova odstojanja su označena promenljivama `sdispl` i `rdispl` na slici 4.2 i imaju različite vrednosti kod pošiljaoca i primaoca.

5 Tehnička implementacija hibridnog C/OpenMP/MPI rešenja

U ovom poglavlju ćemo predstaviti implementaciju hibridnih C/OpenMP/MPI programa [71] za rešavanje GP jednačine u 3D koristeći Krenk-Nikolson algoritam. Oni su bazirani na prethodnim C/OpenMP programima [11], kao i na algoritmu za transponovanje distribuiranih 3D matrica koji je predstavljen u prethodnom poglavlju, u okviru sekcije 4.2.

Funkcionalnost hibridnog C/OpenMP/MPI rešenja datog u projektu GP-SCL-HYB [71] biće predstavljena poređenjem kôda hibridnog rešenja sa kôdom prethodnog C/OpenMP rešenja [11], pri pri čemu su izmene naznačene žutom pozadinom u listinzima. Ovakav vid poređenja naglašava količinu izmena prethodnog kôda neophodnu bi se dobilo hibridno rešenje, koje pruža zadovoljavajuće performanse u vidu dobrog ubrzanja, skalabilnosti i optimalnog zauzeća računarskih resursa. Naravno, bilo je neophodno verifikovati istovetnost svih izlaznih fizičkih veličina sa rezultatima originalnih programa, kako bi se proverila korektnost novog algoritma.

Hibridni programi za propagaciju u imaginarnom vremenu [71] realizovani su u okviru sledećih fajlova: `imagtime3d-hyb.c`, `imagtime3d-hyb.h`, `cfg.c`, `cfg.h`, `diffint.c`, `diffint.h`, `mem.c`, `mem.h`, `misc.c`, `misc.h`. Ovde će biti opisani samo fajlovi u kojima ima izmena u odnosu na prethodno rešenje [11].

5.1 Fajl `imagtime3d-hyb.h`

U listingu 5.1 dat je izvorni kôd izmenjenog zaglavlja `imagtime3d-hyb.h` [71] (žutom pozadinom su označene izmene u odnosu na fajl `imagtime3d-th.h` iz reference [11]).

Listing 5.1: Izmene fajla `imagtime3d-th.h` *kod hibridnog rešenja [71].*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#include <mpi.h>
```

```
#define MAX(a, b, c) (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c)
#define MAX_FILENAME_SIZE 256
#define MAX_FOLDERNAME_SIZE 128

char *output, *initout, *Npasout, *Nrunout, *folder;
long outstpx, outstpy, outstpz;

int opt;
long Nx, Ny, Nz;
long Nx_slice, Ny_slice;
long Nx2, Ny2, Nz2;
long Npas, Nrun;
double dx, dy, dz;
double dx2, dy2, dz2;
double dt;
double G0, G;
double kappa, lambda;
double par;

double *x, *y, *z;
double *x2, *y2, *z2;
int multiplier;

double Ax0, Ay0, Az0, Ax0r, Ay0r, Az0r, Ax, Ay, Az;
double *calphax, *calphay, *calphaz;
double *cgammax, *cgammay, *cgammaz;

//MPI promenljive
int gsize, rank;
MPI_Comm comm;
MPI_Status *send_statuses, *recv_statuses;
MPI_Request *send_requests, *recv_requests;

//Promenljive vezane za blokovsku raspodelu podataka
int * send_displ;
int * send_blocklen;
int * recv_displ;
int * recv_blocklen;
MPI_Datatype block_send_type;
MPI_Datatype block_recv_type;
int * sdispl;
int * rdispl;

void readpar(void);
int init(double *);
void gencoef(void);
void calcnorm(double *, double *, double **, double **, double **, double *);
void calcmuen(double *, double *, double *, double *, double ***, double ***,
double **, double **, double **, double **, double **, double **, double *,
double *, double *);
void calcrms(double *, double *, double **, double **, double **, double *);
void calcnu(double *);
void calclux_trans(double *, double **);
void calcluy(double *, double **);
void calcluz(double *, double **);
double calcpot(int, int, int);
```

```
extern double simpint(double, double *, long);
extern void diff(double, double *, double *, long);

extern double *alloc_double_vector(long);
extern int *alloc_int_vector(int);
extern double **alloc_double_matrix(long, long);
extern double ***alloc_double_tensor(long, long, long);
extern void free_double_vector(double *);
extern void free_int_vector(int *);
extern void free_double_matrix(double **);
extern void free_double_tensor(double ***);

extern int cfg_init(char *);
extern char *cfg_read(char *);

extern int checkDir(char *);
```

Sledi spisak i kratak opis dodatih/izmenjenih promenljivih:

- `<time.h>` – zaglavlje za merenje vremena,
- `<mpi.h>` – zaglavlje sa deklaracijama MPI funkcija,
- `MAX_FOLDERNAME_SIZE` – maksimalna dužina naziva foldera za čuvanje fajlova,
- `NX_slice, NY_slice` — dimenzije sloja u X-Y ravni, koje zavise od broja čvorova `gsize`,
- `multiplier` – koeficijent skaliranja prilikom izračunavanja potencijala, uveden da bi se smanjilo zauzeće memorije i umesto niza sa vrednostima potencijala omogućilo korišćenje funkcije za njegovo dinamičko izračunavanje `calcpot`,
- `gsize`, `rank` – veličina MPI grupe (broj čvorova u klasteru) i rang trenutnog čvora,
- `comm` — naziv MPI komunikatora,
- `send_statuses`, `recv_statuses` – nizovi MPI statusa za proveru validnosti slanja i prijema poruka,
- `send_requests`, `recv_requests` – nizovi MPI zahteva za slanje i prijem poruka,
- `send_displ` – rastojanja svake od duži od početka jednog 3D bloka prilikom slanja, slika 4.2(a),
- `send_blocklen` – dužina duži za slanje (ista dužina za sve duži),
- `recv_displ` – rastojanja svake od duži od početka jednog 3D bloka prilikom prijema, slika 4.2(b),
- `recv_blocklen` – dužina duži za prijem (ista dužina za sve duži),
- `block_send_type` – MPI tip podataka za slanje (računa se iz prethodnih promenljvih),

- block_recv_type – MPI tip podataka za prijem (računa se iz prethodnih promenljvih),
- sdispl – odstojanja 3D blokova za slanje od početka bafera svih podataka (bafera talasne funkcije psi, slika 4.2(a)),
- rdispl – odstojanja 3D blokova za prijem od početka bafera svih podataka (bafera talasne funkcije psi, slika 4.2(b))

Nakon spiska dodatih/izmenjenih promenljivih sledi spisak dodatih/izmenjenih funkcija deklarisanih u zaglavlju imagtime3d-hyb.h [71]. Dodate/izmenjene funkcije će biti detaljno opisane u okviru fajla u kom su definisane:

- init – u funkciji je promenjen tip ulaznog parametra (talasna funkcija psi). psi je sada jednodimenzionalni niz, a ne trodimenzionalni niz (ograničenje uslovljeno kreiranjem indeksnog tipa MPI podataka). U ostatku teze gde god nastaje ovakva promena parametara podrazumeva se da je ona uslovljena kreiranjem indeksnog tipa MPI podataka.
- calcnorm – u funkciji je promenjen tip ulaznog parametra (talasna funkcija psi). psi je sada jednodimenzionalni, a ne trodimenzionalni niz. Takođe, dodat je jednodimenzionalni niz double vrednosti kao parametar u deklaraciju funkcije. Ovaj niz predstavlja MPI bafer koji će poslužiti u pozivu funkcije MPI_Gather.
- calcmuen – u funkciji je promenjen i dodat veći broj parametara, s obzirom da je reč o jednoj do najkompleksnijih funkcija numeričkog algoritma. Promenjeni su i dodati sledeći parametri:
  - talasna funkcija psi je jednodimenzionalan umesto trodimenzionalan niz,
  - izvod talasne funkcije psi po X osi - dpsix je jednodimenzionalan umesto trodimenzionalan niz,
  - dodat je jednodimenzionalni niz double vrednosti mpi_buf koji će poslužiti u pozivu funkcije MPI_Gather,
  - dodat je jednodimenzionalni niz double vrednosti mpi_buf_1 koji će poslužiti u pozivu funkcije MPI_Gather,
  - dodat je jednodimenzionalni niz double vrednosti psi_tmp koji će poslužiti za smeštanje transponovane talasne funkcije psi,
- calcrms – u funkciji je promenjen tip ulaznog parametra (talasna funkcija psi). psi je sada jednodimenzionalni, a ne trodimenzionalni niz. Takođe dodat je jednodimenzionalni niz double vrednosti kao parametar u deklaraciju funkcije. Ovaj niz predstavlja MPI bafer koji će poslužiti u pozivu funkcije MPI_Gather.
- calcnu – u funkciji je promenjen tip ulaznog parametra (talasna funkcija psi). psi je sada jednodimenzionalni, a ne trodimenzionalni niz.

- `calclux_trans` – transponovana verzija funkcije `calclux` (odatle i sufiks). U funkciji je promenjen tip ulaznog parametra (talasna funkcija `psi`). `psi` je sada jednodimenzionalni, a ne trodimenzionalni niz.
- `calcluy` i `calcluz` – u funkcijama je promenjen tip ulaznog parametra (talasna funkcija `psi`). `psi` je sada jednodimenzionalni, a ne trodimenzionalni niz.
- `alloc_int_vector` i `free_int_vector` – dodate eksterne funkcije za zauzimanje i oslobađanje memorije za jednodimenzionalni niz celobrojnih vrednosti.
- `checkDir` – dodata eksterna funkcija za proveru postojanja direktorijuma za upis fajlova i proveru prava pristupa.

5.2 Fajl `mem.h`

U listingu 5.2 dat je izvorni kôd izmenjenog zaglavlja `mem.h` [71] (žutom pozadinom su označene izmene i ukratko opisane ispod listinga).

Listing 5.2: Izmene fajla `mem.h` *kod hibridnog rešenja [71].*

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>

double *alloc_double_vector(long);
int *alloc_int_vector(long);
double complex *alloc_complex_vector(long);
double **alloc_double_matrix(long, long);
double ***alloc_double_tensor(long, long, long);
double complex ***alloc_complex_tensor(long, long, long);

void free_double_vector(double *);
void free_int_vector(int *);
void free_complex_vector(double complex *);
void free_double_matrix(double **);
void free_double_tensor(double ***);
void free_complex_tensor(double complex ***);
```

Sledi spisak i kratak opis dodatih/izmenjenih zaglavlja funkcija:

- `alloc_int_vector` i `free_int_vector` – dodate funkcije za zauzimanje i oslobađanje memorije za jednodimenzionalni niz celobrojnih vrednosti.

5.3 Fajl mem.c

U listingu 5.3 dat je izvorni kôd izmenjenog fajla mem.c [71]. Biće prikazane samo funkcije koje su dodate, pošto je ostatak funkcija neizmenjen. Funkcije neće biti dodatno objašnjavane pošto imaju istu strukturu kao i ostatak funkcija u fajlu mem.c i služe za zauzimanju i oslobađanje niza celobrojnih vrednosti.

Listing 5.3: Izmene fajla mem.c *kod hibridnog rešenja [71].*

```
int *alloc_int_vector(long Nx) {
   int *vector;

   if((vector = (int *) malloc((size_t) (Nx * sizeof(int)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the int vector.\n");
      exit(EXIT_FAILURE);
   }

   return vector;
}

void free_int_vector(int *vector) {
   free((char *) vector);
}
```

5.4 Fajl misc.h

U listingu 5.4 dat je izvorni kôd dodatog zaglavlja misc.h [71]. U pitanju je novi fajl (uz fajl misc.c) u kôdu u odnosu na prethodni projekat [11].

Listing 5.4: Dodato zaglavlje misc.h kod hibridnog rešenja [71].

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sys/types.h"
#include "sys/stat.h"

int checkDir(char *);
```

U zaglavlju misc.h uključena su zaglavlja types.h i stat.h koja omogućuju funkciji checkDir da koristi sistemske konstante i strukture kako bi proverila prava pristupa određenoj putanji ili direktorijumu. Ovo ima primenu kod realne upotrebe programa na klasteru gde treba

upisivati ili čitati određene fajlove i gde treba proveriti da li je upis dozvoljen na čvorovima, kako ne bi došlo do nedozvoljenog upisa/pristupa.

5.5 Fajl `misc.c`

U listingu 5.5 dat je izvorni kôd dodatog fajla `misc.c` [71]. U pitanju je novi fajl (uz fajl `misc.h`) u kôdu u odnosu na prethodno C/OpenMP rešenje [11].

Listing 5.5: Dodatni fajl `misc.c` *kod hibridnog rešenja [71].*

```
#include "misc.h"

/**
 *   Proveri da li direktorijum postoji, da li je specificiran
 *   ili da li se moze pisati u njega – 1 DA, 0 NE.
 */
int checkDir(char * folder_name) {

    struct stat info;
    int ret_val;

    if (folder_name != NULL) {
            if (stat(folder_name, &info) == -1) {
                printf("Output folder %s doesn't exist. ", folder_name);
            } else if (info.st_mode & S_IFDIR) {
                  if (info.st_mode & S_IWUSR ||
                       info.st_mode & S_IWGRP ||
                       info.st_mode & S_IWOTH) {
                     ret_val = 1;
                }
                else {
                  ret_val = 0;
                  printf("Cannot write into output folder: %s.
                          Check folder permissions. ", folder_name);
                }
            } else {
                printf("Cannot access output folder: %s\n. ", folder_name);
                ret_val = 0;
            }
      } else {
            printf("No output folder specified in input file. ");
            ret_val = 0;
      }
    return ret_val;
}
```

Kao što je i opisano u poglavlju 5.4, fajl `misc.c` sadrži kôd funkcije za proveru ispravnosti direktorijuma u koji se upisuje – `checkDir`. Ova funkcija proverava nekoliko stvari: da li direktorijum postoji, da li se u dati direktorijum može upisivati, da li mu se može pristupiti, kao i da

li je direktorijum specificiran kao ulazni parametar funkcije. Sve ovo služi za proveru validnosti upisa, kako ne bi došlo do nedozvoljenog upisa u direktorijume koji su na klasteru zaštićeni, jer se klaster obično konfiguriše u višekorisničkom okruženju.

5.6 Fajl `imagtime3d-hyb.c`

U listingu 5.6 dat je izvorni kôd izmenjenog fajla `imagtime3d-hyb.c` [71] bez definicija funkcija (žutom pozadinom su označene izmene u odnosu na fajl `imagtime3d-th.c` iz prethodnog rešenja [11]). Ovaj fajl i njegove pridružene funkcije predstavljaju srž izmena u odnosu na C/OpenMP rešenje [11]. Fajl sadrži veći broj izmena koje su kompleksne za implementaciju i testiranje. Radi lakšeg raspoznavanja, svaki od bitnih segmenata kôda je obeležen posebnim komentarom i biće analiziran zasebno nakon listinga 5.6.

Listing 5.6: Izmenjeni/dodati delovi kôda u fajlu `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na odgovarajući kôd iz fajla* `imagtime3d-th.c` *prethodnog rešenja [11].*

```
#include "imagtime3d-hyb.h"

int main(int argc, char **argv) {
  /***********VARIABLES*************/
      FILE *out;
      FILE *file;
      int nthreads;
      char filename[MAX_FILENAME_SIZE];
      char file_path[MAX_FOLDERNAME_SIZE + MAX_FILENAME_SIZE];
      long cnti;
      double norm, rms, mu, en;

      double *psi;
      double *psi_tmp;
      double **cbeta;
      double *dpsix, ***dpsiy, ***dpsiz;
      double **tmpxi, **tmpyi, **tmpzi, **tmpxj, **tmpyj, **tmpzj;
      double * mpi_buf, *mpi_buf_1;
      int i, j;

      int middle_element_offset;
      double psi_middle;

      time_t clock_beg, clock_end;
      double wall_time;

   /**************** INITIALIZE_MPI_ENVIRONMENT ****************/

      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &gsize);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_dup(MPI_COMM_WORLD, &comm);
```

```
    if(rank == 0) clock_beg = time(NULL);

/*******************CHECK_INPUT_PARAMS************************/
    if((argc != 3) || (strcmp(*(argv + 1), "-p") != 0)) {
          if (rank == 0)
                fprintf(stderr, "Usage: %s -p <parameterfile> \n", *argv);
          MPI_Finalize();
          exit(EXIT_FAILURE);
    }

    if(!cfg_init(argv[2])) {
          if (rank == 0)
                fprintf(stderr, "Wrong input parameter file.\n");
          MPI_Finalize();
          exit(EXIT_FAILURE);
    }

    readpar();

    if (Nx % gsize != 0) {
          if (rank == 0)
                fprintf(stderr, "Nx is not divisible with 2 * # of MPI
                         processes.\n");
          MPI_Finalize();
          return(EXIT_SUCCESS);
    }

/*********CHECK_NUMBER_OF_THREADS*********/

    #pragma omp parallel
          #pragma omp master
                nthreads = omp_get_num_threads();

/*************ALLOCATE_MEMORY**************/

    Nx_slice = Nx / gsize;
    Ny_slice = Ny / gsize;

    x = alloc_double_vector(Nx_slice);
    y = alloc_double_vector(Ny);
    z = alloc_double_vector(Nz);

    x2 = alloc_double_vector(Nx_slice);
    y2 = alloc_double_vector(Ny);
    z2 = alloc_double_vector(Nz);

    psi = alloc_double_vector(Nx_slice * Ny * Nz);
    psi_tmp = alloc_double_vector(Nx_slice * Ny * Nz);

    dpsix = alloc_double_vector(Nx_slice * Ny * Nz);
    dpsiy = alloc_double_tensor(Nx_slice, Ny, Nz);
    dpsiz = alloc_double_tensor(Nx_slice, Ny, Nz);

    calphax = alloc_double_vector(Nx - 1);      //ceo X domen
    calphay = alloc_double_vector(Ny - 1);
    calphaz = alloc_double_vector(Nz - 1);
    cbeta =  alloc_double_matrix(nthreads, MAX(Nx, Ny, Nz) - 1);  //ceo X
```

```
    cgammax = alloc_double_vector(Nx - 1);         //ceo X domen
    cgammay = alloc_double_vector(Ny - 1);
    cgammaz = alloc_double_vector(Nz - 1);

    tmpxi = alloc_double_matrix(nthreads, Nx);      //ceo X domen
    tmpyi = alloc_double_matrix(nthreads, Ny);
    tmpzi = alloc_double_matrix(nthreads, Nz);
    tmpxj = alloc_double_matrix(nthreads, Nx);      //ceo X domen
    tmpyj = alloc_double_matrix(nthreads, Ny);
    tmpzj = alloc_double_matrix(nthreads, Nz);

    send_displ = alloc_int_vector(Nx_slice * Ny_slice);
    send_blocklen = alloc_int_vector(Nx_slice * Ny_slice);
    recv_displ = alloc_int_vector(Nx_slice * Ny_slice);
    recv_blocklen = alloc_int_vector(Nx_slice * Ny_slice);

    sdispl = alloc_int_vector(gsize);
    rdispl = alloc_int_vector(gsize);

    send_statuses = (MPI_Status*) malloc((size_t) gsize *
                     sizeof(MPI_Status));
    recv_statuses = (MPI_Status*) malloc((size_t) gsize *
                     sizeof(MPI_Status));
    send_requests = (MPI_Request*) malloc((size_t) gsize *
                     sizeof(MPI_Request));
    recv_requests = (MPI_Request*) malloc((size_t) gsize *
                     sizeof(MPI_Request));

    if (rank == 0) {
          mpi_buf = alloc_double_vector(Nx);         //ceo X domen
          mpi_buf_1 = alloc_double_vector(Nx);       //ceo X domen

          if(output != NULL && folder != NULL) {
                sprintf(file_path, "%s/%s", folder, output);
                out = fopen(file_path, "w");
          }
          else out = stdout;

      /******************FORM_OUTPUT_FILE_HEADER***********************/

          fprintf(out, "OPTION = %d\n", opt);
          fprintf(out, "NX = %12li   NY = %12li   NZ = %12li\n", Nx, Ny, Nz);
          fprintf(out, "NPAS = %10li   NRUN = %10li\n", Npas, Nrun);
          fprintf(out, "DX = %8le   DY = %8le   DZ = %8le\n", dx, dy, dz);
          fprintf(out, "DT = %8le\n", dt);
          fprintf(out, "AL = %8le   BL = %8le\n", kappa, lambda);
          fprintf(out, "G0 = %8le\n", G0);
          fprintf(out, "# of MPI processes = %d\n", gsize);
          fprintf(out, "# of OMP threads = %d\n\n", nthreads);
          fprintf(out, "        %12s   %12s   %12s   %12s   %12s\n",
                        "norm", "mu", "en", "rms", "psi(0,0,0)");
          fflush(out);
    }

    G = 0.;

  /********************CALL_INITIAL_FUNCTIONS******************/
```

```
    int init_success = init(psi);

    if (init_success == 2) {
        gencoef();

        calcnorm(&norm, psi, tmpxi, tmpyi, tmpzi, mpi_buf);
        calcmuen(&mu, &en, psi, dpsix, dpsiy, dpsiz, tmpxi, tmpyi, tmpzi,
                 tmpxj, tmpyj, tmpzj, mpi_buf, mpi_buf_1, psi_tmp);
        calcrms(&rms, psi, tmpxi, tmpyi, tmpzi, mpi_buf);

        if (rank == gsize/2) {              //posalji sa srednjeg cvora
            if (rank == 0)
                middle_element_offset = Nx2 * Ny * Nz + Ny2 *
                                        Nz + Nz2;
            else
                middle_element_offset = Ny2 * Nz + Nz2;

            MPI_Send(&psi[middle_element_offset], 1, MPI_DOUBLE, 0, 0,
                     comm);
        }

        if (rank == 0) {                    //primi na nultom cvoru
            MPI_Recv(&psi_middle, 1,  MPI_DOUBLE, gsize/2, 0,
                     comm, &recv_statuses[0]);

  /**************************PRINT_INITIAL_VALUES**************************/

            fprintf(out, "INIT    %8le    %8le    %8le    %8le    %8le\n",
                    norm, mu / par, en / par, rms, psi_middle);
            fflush(out);
        }


/*******************PRINT_INIT_PSI_PROJECTIONS**********************/

      if(initout != NULL && folder != NULL) {
            sprintf(filename, "%s/%s-proc-%03d.x", folder,
                    initout, rank);
            file = fopen(filename, "w");
            for(cnti = 0; cnti < Nx_slice; cnti += outstpx)
                fprintf(file, "%8le %8le\n", x[cnti],
                        psi[cnti * Ny * Nz + Ny2 * Nz + Nz2]);
            fclose(file);

            sprintf(filename, "%s/%s-proc-%03d.y", folder,
                     initout, rank);
            file = fopen(filename, "w");
            for(cnti = 0; cnti < Ny; cnti += outstpy)
                fprintf(file, "%8le %8le\n", y[cnti],
                        psi[Nx2 * Ny * Nz + cnti * Nz + Nz2]);
            fclose(file);

            sprintf(filename, "%s/%s-proc-%03d.z", folder,
                     initout, rank);
            file = fopen(filename, "w");
            for(cnti = 0; cnti < Nz; cnti += outstpz)
```

```
                        fprintf(file, "%8le %8le\n", z[cnti],
                                psi[Nx2 * Ny * Nz + Ny2 * Nz + cnti]);
                  fclose(file);
            }

            G = par * G0;

   /****************PERFORM_NPAS_ITERATIONS*********************/
      for(cnti = 0; cnti < Npas; cnti ++) {
                  calcnu(psi);
                  calcluy(psi, cbeta);
                  calcluz(psi, cbeta);
                  //X->Y petlja transponovanja
                  for (i = 0; i < gsize; i++) {
                        MPI_Irecv(psi_tmp + rdispl[i], 1, block_recv_type,
                                  i, 0, comm, &recv_requests[i]);
                        MPI_Isend(psi + sdispl[i], 1, block_send_type,
                                  i, 0, comm, &send_requests[i]);
                  }
                  for (i = 0; i < gsize; i++) {
                        MPI_Wait(&recv_requests[i], &recv_statuses[i]);
                        MPI_Wait(&send_requests[i], &send_statuses[i]);
                  }
                  calclux_trans(psi_tmp, cbeta);

                  //Y->X petlja transponovanja
                  for (i = 0; i < gsize; i++) {
                        MPI_Irecv(psi + sdispl[i], 1, block_send_type, i, 0,
                                  comm, &recv_requests[i]);
                        MPI_Isend(psi_tmp + rdispl[i], 1, block_recv_type, i,
                                  0, comm, &send_requests[i]);
                  }
                  for (i = 0; i < gsize; i++) {
                        MPI_Wait(&recv_requests[i], &recv_statuses[i]);
                        MPI_Wait(&send_requests[i], &send_statuses[i]);
                  }
                  calcnorm(&norm, psi, tmpxi, tmpyi, tmpzi, mpi_buf);
            }
            calcmuen(&mu, &en, psi, dpsix, dpsiy, dpsiz, tmpxi, tmpyi, tmpzi,
tmpxj, tmpyj, tmpzj, mpi_buf, mpi_buf_1, psi_tmp);
            calcrms(&rms, psi, tmpxi, tmpyi, tmpzi, mpi_buf);

   /*************************PRINT_NPAS_VALUES****************************/
          if (rank == gsize/2) {
                  MPI_Send(&psi[middle_element_offset], 1, MPI_DOUBLE,
                           0, 0, comm);
          }

            if (rank == 0) {
                  MPI_Recv(&psi_middle, 1, MPI_DOUBLE, gsize/2, 0, comm,
                           &recv_statuses[0]);

                  fprintf(out, "NPAS    %8le   %8le   %8le   %8le   %8le\n",
norm, mu / par, en / par, rms, psi_middle);
                  fflush(out);
            }
```

```
/*********************PRINT_NPAS_PSI_PROJECTIONS***********************/

        if(Npasout != NULL && folder != NULL) {
              sprintf(filename, "%s/%s-proc-%03d.x", folder, Npasout,
                      rank);
              file = fopen(filename, "w");
              for(cnti = 0; cnti < Nx_slice; cnti += outstpx)
                    fprintf(file, "%8le %8le\n", x[cnti],
                            psi[cnti * Ny * Nz + Ny2 * Nz + Nz2]);
              fclose(file);

              sprintf(filename, "%s/%s-proc-%03d.y", folder, Npasout,
                      rank);
              file = fopen(filename, "w");
              for(cnti = 0; cnti < Ny; cnti += outstpy)
                    fprintf(file, "%8le %8le\n", y[cnti],
                            psi[Nx2 * Ny * Nz + cnti * Nz + Nz2]);
              fclose(file);

              sprintf(filename, "%s/%s-proc-%03d.z", folder, Npasout,
                      rank);
              file = fopen(filename, "w");
              for(cnti = 0; cnti < Nz; cnti += outstpz)
                    fprintf(file, "%8le %8le\n", z[cnti],
                            psi[Nx2 * Ny * Nz + Ny2 * Nz + cnti]);
              fclose(file);
        }

/**********************PERFORM_NRUN_ITERATIONS**********************/

        for(cnti = 0; cnti < Nrun; cnti ++) {
              calcnu(psi);
              calcluy(psi, cbeta);
              calcluz(psi, cbeta);
              //X->Y petlja transponovanja
              for (i = 0; i < gsize; i++) {
                    MPI_Irecv(psi_tmp + rdispl[i], 1, block_recv_type, i,
                              0, comm, &recv_requests[i]);
                    MPI_Isend(psi + sdispl[i], 1, block_send_type, i, 0,
                              comm, &send_requests[i]);
              }
              for (i = 0; i < gsize; i++) {
                    MPI_Wait(&recv_requests[i], &recv_statuses[i]);
                    MPI_Wait(&send_requests[i], &send_statuses[i]);
              }
              calclux_trans(psi_tmp, cbeta);

              //Y->X petlja transponovanja
              for (i = 0; i < gsize; i++) {
                    MPI_Irecv(psi + sdispl[i], 1, block_send_type, i, 0,
                              comm, &recv_requests[i]);
                    MPI_Isend(psi_tmp + rdispl[i], 1, block_recv_type, i,
                              0, comm, &send_requests[i]);
              }
              for (i = 0; i < gsize; i++) {
                    MPI_Wait(&recv_requests[i], &recv_statuses[i]);
                    MPI_Wait(&send_requests[i], &send_statuses[i]);
```

```
            }
            calcnorm(&norm, psi, tmpxi, tmpyi, tmpzi, mpi_buf);
        }
        calcmuen(&mu, &en, psi, dpsix, dpsiy, dpsiz, tmpxi, tmpyi, tmpzi,
                 tmpxj, tmpyj, tmpzj, mpi_buf, mpi_buf_1, psi_tmp);
        calcrms(&rms, psi, tmpxi, tmpyi, tmpzi, mpi_buf);

/***************************PRINT_NRUN_VALUES*****************************/

        if (rank == gsize/2) {
            MPI_Send(&psi[middle_element_offset], 1, MPI_DOUBLE,
                     0, 0, comm);
        }

        if (rank == 0) {
            MPI_Recv(&psi_middle, 1, MPI_DOUBLE, gsize/2, 0, comm,
                     &recv_statuses[0]);

            fprintf(out, "NRUN    %8le    %8le    %8le    %8le    %8le\n",
                    norm, mu / par, en / par, rms, psi_middle);
            fflush(out);
        }

/*********************PRINT_NRUN_PSI_PROJECTIONS***********************/
        if(Nrunout != NULL && folder != NULL) {
            sprintf(filename, "%s/%s-proc-%03d.x", folder, Nrunout,
                    rank);
            file = fopen(filename, "w");
            for(cnti = 0; cnti < Nx_slice; cnti += outstpx)
                fprintf(file, "%8le %8le\n", x[cnti],
                        psi[cnti * Ny * Nz + Ny2 * Nz + Nz2]);
            fclose(file);

            sprintf(filename, "%s/%s-proc-%03d.y", folder, Nrunout,
                    rank);
            file = fopen(filename, "w");
            for(cnti = 0; cnti < Ny; cnti += outstpy)
                fprintf(file, "%8le %8le\n", y[cnti],
                        psi[Nx2 * Ny * Nz + cnti * Nz + Nz2]);
            fclose(file);

            sprintf(filename, "%s/%s-proc-%03d.z", folder, Nrunout,
                    rank);
            file = fopen(filename, "w");
            for(cnti = 0; cnti < Nz; cnti += outstpz)
                fprintf(file, "%8le %8le\n", z[cnti],
                        psi[Nx2 * Ny * Nz + Ny2 * Nz + cnti]);
            fclose(file);
        }
  /*********************CHECK_IF_NX_IS_CORRECT*************************/
  } else if (init_success == 1 && rank == 0) {
        fprintf(stderr, "Nx is not divisible with 2 * # of MPI
                processes.\n");
  }

 /*************FREE_MEMORY_AND_MEASURE_EXECUTION_TIME*****************/
 if (rank == 0) {
```

```
		clock_end = time(NULL);
		wall_time = difftime(clock_end, clock_beg);
		fprintf(out, "\n Clock Time: %.f seconds\n", wall_time);
		if(output != NULL) fclose(out);
	}

	if (init_success == 2) {
		MPI_Type_free(&block_send_type);
		MPI_Type_free(&block_recv_type);
	}

	if (rank == 0) {
		free_double_vector(mpi_buf);
		free_double_vector(mpi_buf_1);
	}

	free_double_vector(x);
	free_double_vector(y);
	free_double_vector(z);

	free_double_vector(x2);
	free_double_vector(y2);
	free_double_vector(z2);

	free_double_vector(psi);
	free_double_vector(psi_tmp);

	free_double_vector(dpsix);
	free_double_tensor(dpsiy);
	free_double_tensor(dpsiz);

	free_double_vector(calphax);
	free_double_vector(calphay);
	free_double_vector(calphaz);
	free_double_matrix(cbeta);
	free_double_vector(cgammax);
	free_double_vector(cgammay);
	free_double_vector(cgammaz);

	free_double_matrix(tmpxi);
	free_double_matrix(tmpyi);
	free_double_matrix(tmpzi);
	free_double_matrix(tmpxj);
	free_double_matrix(tmpyj);
	free_double_matrix(tmpzj);

	free_int_vector(send_displ);
	free_int_vector(send_blocklen);
	free_int_vector(recv_displ);
	free_int_vector(recv_blocklen);

	free_int_vector(sdispl);
	free_int_vector(rdispl);

	free(send_statuses);
	free(recv_statuses);
	free(send_requests);
```

```
    free(recv_requests);

    MPI_Finalize();

    return(EXIT_SUCCESS);
```

Na početku fajla `imagtime3d-hyb.c` uključuje se zaglavlje glavnog programa `imagtime3d-hyb.h`. Ovo je dovoljno, jer zaglavlje u sebi uključuje sve preostale biblioteke neophodne za rad glavnog programa (sekcija 5.1). Sledi analiza pojedinih segmenata izvornog kôda fajla `imagtime3d-hyb.c` [71], tj. izmena koje su nastale u odnosu na izvorni kôd C/OpenMP rešenja [11]:

- `VARIABLES` segment služi za deklaraciju lokalnih promenljvih glavnog programa. Od promenljvih ćemo opisati samo novouvedene:
    - `file_path` – putanja do izlaznog fajla u koji se upisuju izlazi glavnog programa,
    - `psi` – bafer talasne funkcije, izmenjen u jednodimenzionalni niz tako da podržava formiranje MPI indeksnih tipova podataka,
    - `psi_tmp` – privremeni bafer za smeštanje transponovanih vrednosti talasne funkcije `psi`, neophodan jer nije izvodljivo transponovanje u mestu („in-place“) prilikom razmene MPI poruka,
    - `dpsix` – bafer izvoda talasne funkcije po X osi preveden je u jednodimenzionalni niz, jer će se i on koristiti prilikom transponovanja i mora podržavati formiranje MPI indeksnih tipova podataka,
    - `mpi_buf`, `mpi_buf1` – privremeni MPI baferi za operacije `MPI_Gather` (skupljanje rezultata na nulti čvor),
    - `i`, `j` – brojači petlji,
    - `middle_element_offset` — Udaljenost srednjeg elementa (po sve 3 ose) unutar bafera talasne funkcije `psi`, neophodna da bi se odabrao tačan element srednjeg čvora (po sve 3 ose) i poslao nultom čvoru koji vrši upis srednje vrednosti talasne funkcije u skladu sa presekom tri ose sa slike 2.3,
    - `psi_middle` — vrednost srednjeg elementa (po sve 3 ose) unutar bafera talasne funkcije `psi`, neophodna da bi se na nultom čvoru primila vrednost talasne funkcije od srednjeg čvora po X osi; nakon prijema vrednost se upisuje u skladu sa presekom 3 ose sa slike 2.3,
    - `clock_beg`, `clock_end` — trenuci početka i kraja rada programa zabeleženi samo na nultom čvoru (jer nulti čvor ima funkciju čvora koji vrši ispis svih izlaza iz programa),

- `wall_time` – vreme u sekundama potrebno za izvršavanje celokupnog programa.

◆ `INITIALIZE_MPI_ENVIRONMENT` segment služi za inicijalizaciju MPI okruženja na čvorovima klastera na kojima se program pokreće uz pomoć komande `mpirun`. Preuzimaju se vrednosti veličine MPI grupe (`gsize`), broj trenutnog čvora (`rank`) i MPI komunikator (`comm`) koji je kopija `MPI_COMM_WORLD` komunikatora, pošto se koriste svi čvorovi iz pokrenutog MPI okruženja, tj. nema specifičnih topologija koje bi zahtevale razdvojene komunikatore. Na kraju se na nultom čvoru beleži početni trenutak rada programa (`clock_beg`).

◆ `CHECK_INPUT_PARAMS` segment ima sličnu funkcionlnost kao i istoimeni segment iz sekcije 2.1.8. Glavna razlika je u tome što u slučaju da fajl sa ulaznim parametrima programa nije validno prosleđen, nulti čvor ispisuje poruku o grešci na standardni izlaz za greške (`stderr`). Provera se vrši na svim čvorovima, ali samo nulti čvor vrši ispis o grešci, kako ne bi došlo do višestrukog ispisa sa svih čvorova koji su pokrenuli program. Ovakav mehanizam ispisa (gde samo nulti čvor ispisuje) je uobičajan u MPI programima i SPMD paradigmi. Osim provere ulaznog fajla i ispisa o grešci, ovaj segment takođe proverava broj diskretizacionih tačaka po X osi (`Nx`). Ukoliko dati broj nije deljiv sa veličinom grupe čvorova (`gsize`) bez ostatka, program prekida rad i poziva se metoda `MPI_Finalize` za gašenje MPI okruženja.

◆ `CHECK_NUMBER_OF_THREADS` segment ima istu funkciju kao istoimeni segment iz sekcije 2.1.8. Razlika je u tome da broj niti `nthreads` sada predstavlja broj niti na svakom od čvorova klastera, čime se potvrđuje prava hibridna priroda ovog programa („shared and distributed memory").

◆ `ALLOCATE_MEMORY` segment ima sličnu funkciju kao istoimeni segment iz sekcije 2.1.8, uz značajne razlike prouzrokovane distribuiranjem memorije na čvorove klastera. Stoga pojedini baferi nemaju istu dužinu, a takođe uvedene su i neke nove promenljive i novi baferi:

- `Nx_slice` – broj diskretizacionih tačaka po X osi na jednom čvoru u okviru bloka koji se transponuje između čvorova,
- `Ny_slice` - broj diskretizacionih tačaka po Y osi na jednom čvoru u okviru bloka koji se transponuje između čvorova,
- novi baferi koji umesto `Nx` tačaka po X osi imaju `Nx_slice` tačaka: `x`, `x2`, `psi`, `dpsix`, `dpsiy`, `dpsiz`; ovo je direktna posledica činjenice da je talasna funkcija sistema izdeljena na `gsize` segmenata i da svaki čvor poseduje segmente sa `Nx_slice` tačaka po X osi (pogledati sliku 4.2),
- `psi_tmp` - predstavlja novi bafer koji služi za prihvat transponovanih elemenata iz bafera talasne funkcije `psi` i stoga ima istu veličinu,

- send_displ - predstavlja niz celobrojnih vrednosti veličine Nx_slice * Ny_slice; ovo je niz koji sadrži udaljenosti („offset“) duži od nulte lokacije 3D blokova koji se koriste u transponovanju (pogledati sekciju 4.2),
- send_blocklen - predstavlja niz celobrojnih vrednosti veličine Nx_slice * Ny_slice; ovo je niz koji sadrži odgovarajuće dužine duži za svaku udaljnost iz niza send_displ; s obzirom da su u pitanju duži iste dužine, sve vrednosti u ovom nizu biće jednake Nz (pogledati sekciju 4.2),
- recv_displ i recv_blocklen - nizovi iste namene kao i nizovi send_displ i send_blocklen, ali za slučaj prijemne strane,
- sdispl i rdispl - predstavlju nizove celobrojnih vrednosti veličine gsize; u pitanju su nizovi koji sadrže udaljenosti („offset“) pojedinačnih 3D blokova od početka čitavog bafera talasne funkcije psi (sekcija 4.2) i to u slučaju slanja i prijema blokova, respektivno,
- send_statuses, recv_statuses, send_requests, recv_requests - predstavljaju nizove statusa i zahteva za slanje i prijem MPI poruka; u pitanju su nizovi neophodni za gsize operacija asinhronog slanja i prijema 3D blokova prilikom operacije transponovanja; imaju formalnu vrednost, pošto svaka operacija slanja i prijema zahteva poseban status, kao i zahtev i proveru da li su slanje i prijem bili uspešni,
- mpi_buf i mpi_buf_1 - baferi dužine Nx koji služe za pozive operacije MPI_Gather na nultom čvoru klastera; ove bafere alocira samo nulti čvor.

◆ FORM_OUTPUT_FILE_HEADER segment ima sličnu funkciju kao istoimeni segment iz sekcije 2.1.8. Razlika je u tome što se u standardno zaglavlje izlaznog fajla dodaje i informacija o broju MPI čvorova (gsize), kao i broj OpenMP niti po čvoru (nthreads).

◆ CALL_INITIAL_FUNCTIONS segment ima sličnu funkciju kao istoimeni segment iz sekcije 2.1.8. Razlika je u tome što funkcija za inicijalizaciju init vraća povratnu vrednost jedan ukoliko Nx_slice nije paran broj, odnosno dva ukoliko Nx_slice jeste paran broj. Uslov daljeg rada programa je da je Nx_slice paran broj. Pored ovoga, promena u segmentu se ogleda u slanju srednjeg elementa po sve tri ose (projekciji talasne funkcije psi u skladu sa presekom tri duži na slici 2.3) sa srednjeg čvora klastera (gsize/2) na nulti čvor klastera. Cilj ovog slanja je da omogući nultom čvoru klastera da upiše vrednost projekcije talasne funkcije psi u izlazni fajl.

◆ PRINT_INITIAL_VALUES segment ima istu funkciju kao istoimeni segment iz sekcije 2.1.8.

◆ PRINT_INIT_PSI_PROJECTIONS segment ima sličnu funkciju kao istoimeni segment iz sekcije 2.1.8. Razlike su sledeće:

- Prilikom upisa projekcija talasne funkcije psi proverava se i validnost direktorijuma za upis.
- Svaki od fajlova koji sadrži projekcije talasne funkcije po sve tri ose dobija specifičan naziv koji opisuje nakon koliko iteracija algoritma je nastao fajl, kao i sa kojeg čvora na klasteru je formiran. Na ovaj način lako se mogu pronaći odgovarajuće projekcije talasne funkcije nastale na pojedinim čvorovima. Npr. projekcija po X osi nastala na nultom čvoru nakon nrun iteracija algoritma ima naziv izlaznog fajla imagtime3d-nrunout-proc-000.x.
- Granica projekcije po X osi je promenjena na Nx_slice s obzirom da svaki čvor sadrži samo toliko tačaka po X osi.
- Talasna funkcija psi se indeksira kao jednodimenzionalan niz umesto eksplicitnog trodimenzionalnog niza.

◆ PERFORM_NPAS_ITERATIONS segment ima sličnu funkciju kao istoimeni segment iz sekcije 2.1.8. Razlike su sledeće:

- Funkcija calclux_trans (calclux u C/OpenMP programima) zahteva transponovanje podataka u X-Y smeru pre početka rada funkcije, kao i još jedno transponovanje u istom smeru nakon završetka rada funkcije. Transponovanje se vrši u petlji koja se izvršava gsize puta, gde se asinhrono uz pomoć funkcija MPI_Isend šalju blokovi bafera psi drugim čvorovima, dok se istovremeno uz pomoć operacija MPI_Irecv od istih čvorova primaju odgovarajući interno transponovani blokovi u bafer psi_tmp. Slanje i prijem blokova se vrše u skladu sa ilustracijom datom na slici 4.2. Sva slanja i prijemi moraju biti kompletirani pre poziva sledeće numeričke funkcije calcnorm, zbog čega se ulazi u blokirajuće čekanje MPI_Wait na završetak svih slanja i prijema i to na svakom od čvorova, pre izvršenja sledećeg koraka.
- Umesto poziva ranije funkcije calclux poziva se nova funkcija calclux_trans, koja će biti opisana u poglavlju 5.6.8.
- Funkcije calcnorm, calcmuen i calcrms dobijaju kao dodatne argumente bafere mpi_buf, mpi_buf_1 i psi_tmp, neophodne za poziv funkcije MPI_Gather, kao i za interno transponovanje podataka.

◆ Naredni segmenti koda su ekvivalentni prethodno opisanim segmentima kôda i stoga neće biti posebno opisivani. Jedina razlika je u tome što je različit broj korišćenih iteracija (NPAS ili NRUN) od prethodnog poziva istovetnog segmenta kôda. Ovakvo odvijanje programa ima za cilj konvergenciju sistema ka stacionarnom stanju u slučaju propagacije u imaginarnom vremenu, odnosno praćenje dinamike sistema u nekoliko faza u slučaju propagacije u realnom vremenu. Sa leve strane je dat naziv novog segmenta kôda, a sa desne strane naziv ekvivalentnog segmenta kôda koji je već opisan:

- PRINT_NPAS_VALUES – PRINT_INITIAL_VALUES,

- PRINT_NPAS_PSI_PROJECTIONS - PRINT_INIT_PSI_PROJECTIONS,
- PERFORM_NRUN_ITERATIONS - PERFORM_NPAS_ITERATIONS,
- PRINT_NRUN_VALUES – PRINT_INITIAL_VALUES,
- PRINT_NRUN_PSI_PROJECTIONS – PRINT_INIT_PSI_PROJECTIONS.

◆ CHECK_IF_NX_IS_CORRECT segment kôda proverava da li je incijalizacija programa uspešna. Ukoliko to nije slučaj, ispisuje se poruka o grešci na standardni izlaz za greške.

◆ FREE_MEMORY_AND_MEASURE_EXECUTION_TIME segment ima sličnu funkciju kao istoimeni segment iz sekcije 2.1.8. Razlike su sledeće:

- Meri se vreme proteklo za rad čitavog programa, kao razlika krajnjeg i početnog trenutka.
- Oslobađaju se MPI tipovi podataka i to samo ukoliko je inicijalizacija prošla korektno.
- Oslobađaju se svi baferi koji su dodati za hibridnu verziju kôda.
- Poziva se funkcija MPI_Finalize koja završava korišćenje MPI okruženja.

5.6.1 Funkcija readpar

Izmene u funkciji readpar neće biti posebno opisivane zbog njihove trivijalnosti. Izmene se svode na ispis izlaza u slučaju greške sa nultog čvora i izlazak iz MPI okruženja funkcijom MPI_Finalize.

5.6.2 Funkcija init

U listingu 5.7 dat je izvorni kôd izmenjene funkcije init iz fajla imagtime3d-hyb.c (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju init iz fajla imagtime3d-th.c prethodnog C/OpenMP rešenja [11].

Listing 5.7: Izmenjeni/dodati delovi kôda u funkciji init iz fajla imatgtime3d-hyb.c *(projekat GP-SCL-HYB) [71] u odnosu na kôd funkcije init iz fajla* imagtime3d-th.c *[11].*

```
int init(double *psi) {
      long cnti, cntj, cntk;
      long cnti_offset;
      double kappa2, lambda2;
      double pi, cpsi1, cpsi2;
      double tmp;
```

```
int i, j;

for (i = Ny/gsize - 1; i >= 0 ; i--) {
      for (j = 0; j < Nx_slice; j++) {
            send_displ[(Ny_slice - 1 - i) * Nx_slice + j] =
                        i * Nz + j * Ny * Nz;
            send_blocklen[(Ny_slice - 1 - i) * Nx_slice + j] = Nz;

            recv_displ[(Ny_slice - 1 - i) * Nx_slice + j] =
                  (Nx_slice - 1 - j) * Nz +
                  (Ny_slice - 1 - i) * Nx_slice * gsize * Nz;
            recv_blocklen[(Ny_slice - 1 - i) * Nx_slice + j] = Nz;
      }
}
MPI_Type_indexed(Nx_slice * Ny_slice, send_blocklen,
                  send_displ, MPI_DOUBLE, &block_send_type);
MPI_Type_indexed(Nx_slice * Ny_slice, recv_blocklen,
                  recv_displ, MPI_DOUBLE, &block_recv_type);

if (Nx_slice % 2 != 0) {  //Nx_slice mora biti deljivo sa 2
      return 1;
}

MPI_Type_commit(&block_send_type);
MPI_Type_commit(&block_recv_type);

for (i = gsize - 1; i >= 0; i--) {
      sdispl[gsize - 1 - i] = i * Ny_slice * Nz;
      rdispl[gsize - 1 - i] = i * Nx_slice * Nz;
}

if (opt == 2) par = 2.;
else par = 1.;

if(opt == 3) multiplier = 0.25;
else multiplier = 1.;

cnti_offset = rank * Nx_slice;

kappa2 = kappa * kappa;
lambda2 = lambda * lambda;

Nx2 = Nx_slice / 2; Ny2 = Ny / 2; Nz2 = Nz / 2;
dx2 = dx * dx; dy2 = dy * dy; dz2 = dz * dz;

pi = 4. * atan(1.);
cpsi1 = sqrt(pi * sqrt(pi / (kappa * lambda)));
cpsi2 = cpsi1 * sqrt(2. * sqrt(2.));

for(cnti = 0; cnti < Nx_slice; cnti ++) {
      x[cnti] = (cnti_offset + cnti - Nx2 * gsize) * dx;
      x2[cnti] = x[cnti] * x[cnti];
      for(cntj = 0; cntj < Ny; cntj ++) {
            y[cntj] = (cntj - Ny2) * dy;
            y2[cntj] = y[cntj] * y[cntj];
            for(cntk = 0; cntk < Nz; cntk ++) {
                  z[cntk] = (cntk - Nz2) * dz;
```

```
                        z2[cntk] = z[cntk] * z[cntk];
                        if(opt == 3) {
                              tmp = exp(- 0.25 * (x2[cnti] + kappa * y2[cntj]
                                  + lambda * z2[cntk]));
                              psi[cnti * Ny * Nz + cntj * Nz + cntk] =
                                  tmp / cpsi2;
                        } else {
                              tmp = exp(- 0.5 * (x2[cnti] + kappa *
                                        y2[cntj] + lambda * z2[cntk]));
                              psi[cnti * Ny * Nz + cntj * Nz + cntk] =
                                  tmp / cpsi1;
                        }
                  }
            }
      }

      return 2;
}
```

Kao što se vidi u listingu 5.7, izmene u funkciji `init` su značajne:

- Ulazni bafer talasne funkcije `psi` sada je jednodimenzionalni niz, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Značajan deo inicijalizacije sada zauzima kreiranje nizova `send_displ`, `send_block_length`, `recv_displ`, `recv_block_length`, `sdislp`, `rdispl`, kao i MPI indeksnih tipova podataka `block_send_type` i `block_recv_type`. Ovi nizovi i MPI tipovi podataka su objašnjeni detaljnije u sekciji 4.2, kao i na slikama 4.2 i 4.3.
- Promenjeni su određeni koeficijenti skaliranja i brojači i granice u skladu sa podelom podataka talasne funkcije na slojeve dužine `Nx_slice` po X osi.
- Ukinuta je incijalizacija niza potencijala `pot`, čime se štedi memorijsko zauzeće, što je veoma značajno zbog inače velikog zauzeća memorije u programu. Umesto toga potencijal se izračunava dinamički u posebnoj funkciji `calcpot` (sekcija 5.6.11).
- Funkcija sada vraća vrednost jedan ili dva, u zavisnosti od uspešnosti inicijalizacije.

5.6.3 Funkcija `gencoef`

Funkcija `gencoef` nije imala nikakvih izmena u odnosu na originalnu.

5.6.4 Funkcija `calcnorm`

U listingu 5.8 dat je izvorni kôd izmenjene funkcije `calcnorm` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju `calcnorm` iz fajla `imagtime3d-th.c` [11].

Listing 5.8: Izmenjeni/dodati delovi kôda u funkciji `calcnorm` *iz fajla* `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na kôd funkcije* `calcnorm` *iz fajla* `imagtime3d-th.c` *[11].*

```
void calcnorm(double *norm, double *psi, double **tmpx, double **tmpy,
              double **tmpz, double *mpi_buf) {
      int threadid;
      long cnti, cntj, cntk;

      #pragma omp parallel private(threadid, cnti, cntj, cntk)
      {
            threadid = omp_get_thread_num();

            #pragma omp for
            for(cnti = 0; cnti < Nx_slice; cnti ++) {
                  for(cntj = 0; cntj < Ny; cntj ++) {
                        for(cntk = 0; cntk < Nz; cntk ++) {
                              tmpz[threadid][cntk] =
                                   psi[cnti * Ny * Nz + cntj * Nz + cntk] *
                                   psi[cnti * Ny * Nz + cntj * Nz + cntk];
                        }
                        tmpy[threadid][cntj] = simpint(dz, tmpz[threadid], Nz);
                  }
                  tmpx[0][cnti] = simpint(dy, tmpy[threadid], Ny);
            }
            #pragma omp barrier
      }

      MPI_Gather(tmpx[0], Nx_slice, MPI_DOUBLE, mpi_buf, Nx_slice,
                 MPI_DOUBLE, 0, comm);

      if (rank == 0) {
            *norm = sqrt(simpint(dx, mpi_buf, Nx));     //ceo X domen
      }

      MPI_Bcast(norm, 1, MPI_DOUBLE, 0, comm);

      #pragma omp parallel private(cnti, cntj, cntk)
      {
            #pragma omp for
            for(cnti = 0; cnti < Nx_slice; cnti ++) {
                  for(cntj = 0; cntj < Ny; cntj ++) {
                        for(cntk = 0; cntk < Nz; cntk ++) {
                              psi[cnti * Ny * Nz + cntj * Nz + cntk] /= *norm;
                        }
                  }
            }
      }
```

```
    return;
}
```

Izmene u funkciji `calcnorm` su sledeće:

- Ulazni bafer talasne funkcije `psi` sada je jednodimenzionalni niz, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Poziva se funkcija `MPI_Gather` kako bi se upisali svi parcijalni rezultati `tmpx[0]` u globalni `mpi_buf` bafer. Ovo je neophodno kako bi mogla da se izvrši kompletna integracija po X osi, i na taj način izračuna norma talasne funkcije `psi` na nultom čvoru.
- Poziva se funkcija `MPI_Bcast` koja šalje izračunatu normu sa nultog čvora svim čvorovima u klasteru, što je neophodno da bi svi čvorovi mogli da izvrše normalizaciju svojih delova talasne funkcije `psi`.

5.6.5 Funkcija `calcmuen`

U listingu 5.9 dat je izvorni kôd izmenjene funkcije `calcmuen` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju `calcmuen` iz fajla `imagtime3d-th.c` [11].

Listing 5.9: Izmenjeni/dodati delovi kôda u funkciji `calcmuen` *iz fajla* `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na kôd funkcije* `calcmuen` *iz fajla* `imagtime3d-th.c` *[11].*

```
void calcmuen(double *mu, double *en, double *psi, double *dpsix,
              double ***dpsiy, double ***dpsiz, double **tmpxi,
              double **tmpyi, double **tmpzi, double **tmpxj,
              double **tmpyj, double **tmpzj, double *mpi_buf,
              double *mpi_buf_1, double *psi_tmp) {
    int threadid, i;
    long cnti, cntj, cntk;
    double psi2, psi2lin, dpsi2;

    //X->Y petlja transponovanja (prebaci regularni psi
     //u transponovani psi_tmp
    for (i = 0; i < gsize; i++) {
        MPI_Irecv(psi_tmp + rdispl[i], 1, block_recv_type,
                  i, 0, comm, &recv_requests[i]);
        MPI_Isend(psi + sdispl[i], 1, block_send_type,
                  i, 0, comm, &send_requests[i]);
    }
    for (i = 0; i < gsize; i++) {
        MPI_Wait(&recv_requests[i], &recv_statuses[i]);
        MPI_Wait(&send_requests[i], &send_statuses[i]);
```

```
}

#pragma omp parallel private(threadid, cnti, cntj, cntk,
                              psi2, psi2lin, dpsi2)
{
      threadid = omp_get_thread_num();

      #pragma omp for
      for(cntj = 0; cntj < Ny_slice; cntj ++) {
            for(cntk = 0; cntk < Nz; cntk ++) {
                  for(cnti = 0; cnti < Nx; cnti ++) {   //ceo X domen
                        tmpxi[threadid][cnti] =
                              psi_tmp[(Nx - 1 - cnti) *
                              Nz + cntj * Nx * Nz + cntk];
                  }
                  diff(dx, tmpxi[threadid], tmpxj[threadid], Nx);
                  for(cnti = 0; cnti < Nx; cnti ++) {
                        psi_tmp[(Nx - 1 — cnti) * Nz + cntj *
                           Nx * Nz + cntk] =
                          tmpxj[threadid][cnti]; //dpsix (psi_tmp)
                  }
            }
      }
}

//Y->X petlja transponovanja (pomeri transponovani dpsix(psi_tmp) u
 //regularni dpsix)
for (i = 0; i < gsize; i++) {
      MPI_Irecv(dpsix + sdispl[i], 1, block_send_type,
                i, 0, comm, &recv_requests[i]);
      MPI_Isend(psi_tmp + rdispl[i], 1, block_recv_type,
                i, 0, comm, &send_requests[i]);
}
for (i = 0; i < gsize; i++) {
      MPI_Wait(&recv_requests[i], &recv_statuses[i]);
      MPI_Wait(&send_requests[i], &send_statuses[i]);
}

#pragma omp parallel private(threadid, cnti, cntj, cntk,
                              psi2, psi2lin, dpsi2)
{
      threadid = omp_get_thread_num();

      #pragma omp for
      for(cnti = 0; cnti < Nx_slice; cnti ++) {
            for(cntk = 0; cntk < Nz; cntk ++) {
                  for(cntj = 0; cntj < Ny; cntj ++) {
                        tmpyi[threadid][cntj] =
                            psi[cnti * Ny * Nz + cntj * Nz + cntk];
                  }
                  diff(dy, tmpyi[threadid], tmpyj[threadid], Ny);
                  for(cntj = 0; cntj < Ny; cntj ++) {
                        dpsiy[cnti][cntj][cntk] = tmpyj[threadid][cntj];
                  }
            }
      }
```

```
        #pragma omp for
        for(cnti = 0; cnti < Nx_slice; cnti ++) {
              for(cntj = 0; cntj < Ny; cntj ++) {
                    for(cntk = 0; cntk < Nz; cntk ++) {
                          tmpzi[threadid][cntk] =
                             psi[cnti * Ny * Nz + cntj * Nz + cntk];
                    }
                    diff(dz, tmpzi[threadid], tmpzj[threadid], Nz);
                    for(cntk = 0; cntk < Nz; cntk ++) {
                          dpsiz[cnti][cntj][cntk] = tmpzj[threadid][cntk];
                    }
              }
        }
        #pragma omp barrier

        #pragma omp for
        for(cnti = 0; cnti < Nx_slice; cnti ++) {
              for(cntj = 0; cntj < Ny; cntj ++) {
                    for(cntk = 0; cntk < Nz; cntk ++) {
                          psi2 = psi[cnti * Ny * Nz + cntj * Nz + cntk]
                               * psi[cnti * Ny * Nz + cntj * Nz + cntk];
                          psi2lin = psi2 * G;
                          dpsi2 = dpsix[cnti * Ny * Nz + cntj * Nz + cntk]
                                * dpsix[cnti * Ny * Nz + cntj * Nz + cntk]
                                + dpsiy[cnti][cntj][cntk]
                                * dpsiy[cnti][cntj][cntk]
                                + dpsiz[cnti][cntj][cntk]
                                * dpsiz[cnti][cntj][cntk];
                          tmpzi[threadid][cntk] =
                                (calcpot(cnti, cntj, cntk) + psi2lin)
                                * psi2 + dpsi2;
                          tmpzj[threadid][cntk] =
                               (calcpot(cnti, cntj, cntk) + 0.5 * psi2lin)
                                * psi2 + dpsi2;
                    }
                    tmpyi[threadid][cntj] =
                               simpint(dz, tmpzi[threadid], Nz);
                    tmpyj[threadid][cntj] =
                               simpint(dz, tmpzj[threadid], Nz);
              }
              tmpxi[0][cnti] = simpint(dy, tmpyi[threadid], Ny);
              tmpxj[0][cnti] = simpint(dy, tmpyj[threadid], Ny);
        }
  }

  MPI_Gather(tmpxi[0], Nx_slice, MPI_DOUBLE, mpi_buf,
             Nx_slice, MPI_DOUBLE, 0, comm);
  MPI_Gather(tmpxj[0], Nx_slice, MPI_DOUBLE, mpi_buf_1,
             Nx_slice, MPI_DOUBLE, 0, comm);

  if (rank == 0) {
        *mu = simpint(dx, mpi_buf, Nx);
        *en = simpint(dx, mpi_buf_1, Nx);
  }

  return;
}
```

Kao što se vidi u listingu 5.9, izmene u funkciji `calcmuen` su značajne:

- Ulazni bafer talasne funkcije `psi` kao i bafer izvoda talasne funkcije `dpsix` sada su jednodimenzionalni nizovi, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Dodati su ulazni parametri `mpi_buf`, `mpi_buf_1` i `psi_tmp`. Prva dva predstavljaju pomoćne bafere za pozive funkcije `MPI_Gather`, dok treći predstavlja postojeći (alocirani) bafer koji se koristi za skladištenje transponovane talasne funkcije na jednom čvoru i u drugim delovima programa, pa sa te strane ne povećava memorijsko zauzeće u programu. Treći bafer je neophodan, jer će se i u okviru funkcije `calcmuen` vršiti transponovanje podataka. Dodat je i pomoćni brojač `i` koji služi kao brojač petlje unutar asinhronog slanja i prijema transponovanih podataka.
- Unutar funkcije, pre prve grupe petlji koja paralelno računa izvode talasne funkcije `psi` po X osi (poziv funkcije `diff`), neophodno je kao i pre poziva funkcije `calclux_trans` (sekcija 5.6) prvo transponovati podatke, tako da su podaci po X osi kompletno dostupni na svakom čvoru. Ovo dovodi do toga da podaci po Y osi postaju distribuirani (zbog toga spoljna petlja unutar prve grupe petlji ide od 0 do `Ny_slice - 1`). Transponovanje se vrši na isti način kao i pre poziva funkcije `calclux_trans`.
- Nakon izračunavanja izvoda po X osi prihvatni bafer za izvode se menja iz `dpsix` u `psi_tmp` (što je i zabeleženo komentarom u kodu). Ovo se radi zato što je neophodno još jedno transponovanje u funkciji `calcmuen`, kako bi podaci duž Y ose bili kompletni na svakom čvoru. To transponovanje ne sme da poremeti originalne vrednosti talasne funkcije `psi` (koja nigde u funkciji `calcmuen` ne menja svoju vrednost) i bilo bi pogrešno koristiti bafer talasne funkcije `psi` za transponovanje u suprotnom smeru, jer bi se time izmenila vrednost talasne funkcije u nastavku algoritma. Iz tog razloga za dodatno transponovanje kombinuju se baferi `psi_tmp` i `dpsix` (a ne `psi_tmp` i `psi`, kao u slučaju funkcije `calclux_trans`).
- Nakon računanja izvoda po X domenu, vrši se transponovanje u suprotnom smeru, tj. kompletira se Y domen, kako bi se moglo izvršiti i računanje izvoda po Y domenu.
- Računanje izvoda po Y i Z osi se vrši imajući u vidu da su podaci duž X ose distribuirani, tj. da brojač petlje po X osi uzima vrednosti od 0 do `Nx_slice – 1.`
- Nakon računanja izvoda po svim osama, vrši se paralelno računanje kvadrata talasne funkcije `psi`, sume kvadrata izvoda talasne funkcije po svim osama `dpsi2`, kao i računanje potencijala u tačkama prostorne mreže, radi integraljenja i računanja hemijskog potencijala i energije BAK. Računanje potencijala se ovoga puta vrši preko poziva funkcije `calcpot`, koja zamenjuje korišćenje vrednosti smeštenih u niz potencijala, čime se vrši ušteda memorije na račun poziva dodatnih računskih instrukcija.

- Na kraju se vrši računanje vrednosti hemijskog potencijala (`mu`) i energije (`en`) sistema. Da bi se ovo postiglo neophodno je bafere, koji sadrže parcijalne integrale potencijala po sve tri ose (`tmpxi[0]` i `tmpxj[0]`), sakupiti na nulti čvor klastera u jedan bafer (`mpi_buf` i `mpi_buf_1` respektivno) uz pomoć funkcije `MPI_Gather`. Tek tada nulti čvor klastera može da izračuna konačan rezultat za tražene fizičke veličine. S obzirom da se date veličine štampaju u izlazni fajl sa nultog čvora klastera, nije neophodno njihovo naknadno slanje svim čvorovima.

5.6.6 Funkcija `calcrms`

U listingu 5.10 dat je izvorni kôd izmenjene funkcije `calcrms` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju `calcrms` iz fajla `imagtime3d-th.c` [11].

Listing 5.10: Izmenjeni/dodati delovi kôda u funkciji `calcrms` *iz fajla* `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na kôd funkcije* `calcrms` *iz fajla* `imagtime3d-th.c` *[11].*

```
void calcrms(double *rms, double *psi, double **tmpx, double **tmpy, double
**tmpz, double *mpi_buf) {
      int threadid;
      long cnti, cntj, cntk;
      double psi2;

      #pragma omp parallel private(threadid, cnti, cntj, cntk, psi2)
      {
            threadid = omp_get_thread_num();

            #pragma omp for
            for(cnti = 0; cnti < Nx_slice; cnti ++) {
                  for(cntj = 0; cntj < Ny; cntj ++) {
                        for(cntk = 0; cntk < Nz; cntk ++) {
                              psi2 = psi[cnti * Ny * Nz + cntj * Nz + cntk]
                                   * psi[cnti * Ny * Nz + cntj * Nz + cntk];
                              tmpz[threadid][cntk] = (x2[cnti] + y2[cntj]
                                                      + z2[cntk]) * psi2;
                        }
                        tmpy[threadid][cntj] = simpint(dz, tmpz[threadid], Nz);
                  }
                  tmpx[0][cnti] = simpint(dy, tmpy[threadid], Ny);
            }
      }

      MPI_Gather(tmpx[0], Nx_slice, MPI_DOUBLE, mpi_buf,
                 Nx_slice, MPI_DOUBLE, 0, comm);

      if (rank == 0) {
            *rms = sqrt(simpint(dx, mpi_buf, Nx));     //ceo X domen
      }
```

```
    return;
}
```

Izmene u funkciji `calcrms` su sledeće:

- Ulazni bafer talasne funkcije `psi` sada je jednodimenzionalni niz, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Poziva se funkcija `MPI_Gather` kako bi se upisali svi parcijalni rezultati `tmpx[0]` u globalni `mpi_buf` bafer. Ovo je neophodno kako bi mogla da se izvrši integracija po X osi, a nakon toga da se izračuna srednji kvadratni radijus talasne funkcije `psi` na nultom čvoru.

5.6.7 Funkcija `calcnu`

U listingu 5.11 dat je izvorni kôd izmenjene funkcije `calcnu` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju `calcnu` iz fajla `imagtime3d-th.c` [11].

Listing 5.11: Izmenjeni/dodati delovi kôda u funkciji `calcnu` *iz fajla* `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na kôd funkcije* `calcnu` *iz fajla* `imagtime3d-th.c` *[11].*

```
void calcnu(double *psi) {
    long cnti, cntj, cntk;
    double psi2, psi2lin, tmp;

    #pragma omp parallel for private(cnti, cntj, cntk, psi2, psi2lin, tmp)
    for(cnti = 0; cnti < Nx_slice; cnti ++) {
        for(cntj = 0; cntj < Ny; cntj ++) {
            for(cntk = 0; cntk < Nz; cntk ++) {
                psi2 = psi[cnti * Ny * Nz + cntj * Nz + cntk]
                    * psi[cnti * Ny * Nz + cntj * Nz + cntk];
                psi2lin = psi2 * G;
                tmp = dt * (calcpot(cnti, cntj, cntk) + psi2lin);
                psi[cnti * Ny * Nz + cntj * Nz + cntk] *= exp(-tmp);
            }
        }
    }

    return;
}
```

Izmene u funkciji `calcnu` su sledeće:

- Ulazni bafer talasne funkcije `psi` sada je jednodimenzionalni niz, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Gornja granica opsega na kome se izračunava vremenska propagacija u odnosu na H_1 deo Hamiltonijana je sada `Nx_slice - 1`, pošto je X osa distribuirana.
- Računanje potencijala se vrši pozivanjem funkcije `calcpot`, koja zamenjuje korišćenje niza vrednosti potencijala, čime se vrši ušteda memorije na račun poziva dodatnih računskih instrukcija.

5.6.8 Funkcija `calclux_trans`

U listingu 5.12 dat je izvorni kôd izmenjene funkcije `calclux (calclux_trans)` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju `calclux` iz fajla `imagtime3d-th.c` [11].

Listing 5.12: Izmenjeni/dodati delovi kôda u funkciji `calclux` *(*`calclux_trans`*) iz fajla* `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na kôd funkcije* `calclux` *iz fajla* `imagtime3d-th.c` *[11].*

```
void calclux_trans(double *psi_tmp, double **cbeta) {
      int threadid;
      long cnti, cntj, cntk;
      double c;

      #pragma omp parallel private(threadid, cnti, cntj, cntk, c)
      {
            threadid = omp_get_thread_num();

            #pragma omp for
            for(cntj = 0; cntj < Ny_slice; cntj ++) {
                  for(cntk = 0; cntk < Nz; cntk ++) {
                        cbeta[threadid][Nx - 2] = psi_tmp[0 * Nz + cntj
                                                * Nx * Nz + cntk];
                        for (cnti = Nx - 2; cnti > 0; cnti --) {
                              c = - Ax * psi_tmp[(Nx - 1 - cnti - 1)
                                        * Nz + cntj * Nx * Nz + cntk] +
                                    Ax0r * psi_tmp[(Nx - 1 - cnti) * Nz
                                        + cntj * Nx * Nz + cntk] -
                                    Ax * psi_tmp[(Nx - 1 - cnti + 1) *
                                         Nz + cntj * Nx * Nz  + cntk];
                              cbeta[threadid][cnti - 1] =  cgammax[cnti] *
                                         (Ax * cbeta[threadid][cnti] - c);
                        }
                        psi_tmp[(Nx - 1) * Nz + cntj * Nx * Nz + cntk] = 0.;
                        for (cnti = 0; cnti < Nx - 2; cnti ++) {
                              psi_tmp[(Nx - 1 - cnti - 1) * Nz +
                                   cntj * Nx * Nz + cntk] =
```

```
                                                calphax[cnti] *
                                                psi_tmp[(Nx - 1 - cnti) * Nz +
                                                cntj * Nx * Nz + cntk] +
                                                cbeta[threadid][cnti];
                                }
                                psi_tmp[0 * Nz + cntj * Nx * Nz + cntk] = 0.;
                        }
                }
        }

        return;
}
```

Izmene u funkciji `calclux_trans` su sledeće:

- Ulazni bafer talasne funkcije `psi` preimenovan je u `psi_tmp` (pošto je u pitanju bafer koji sadrži transponovane vrednosti talasne funkcije). `psi_tmp` je jednodimenzionalni niz, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Promenjeno je indeksiranje bafera `psi_tmp` (`psi` iz originalnog koda) na više mesta. Indeksiranje je sada, osim što je jednodimenzionalno, organizovano u drugom smeru. Naime, element originalnog `psi` bafera indeksiran sa `[Nx – 1][cntj][cntk]` se sada indeksira u `psi_tmp` baferu obratno, gledajući duž X ose i ima indeks `[0 * Nz + cntj * Nx * Nz + cntk]`. Ovo je posledica zamene X i Y osa prilikom transponovanja (slika 4.2), što znači da je krajnja tačka (`Nx – 1`) bafera `psi` po X osi nakon transponovanja zapravo nulta tačka bafera `psi_tmp` po transponovanoj X osi. Drugim rečima, tačka iz gornjeg desnog ugla bafera `psi` je donja leva tačka transponovanog bafera `psi_tmp`, računajući da baferi `psi` i `psi_tmp` počinju u donjem levom uglu, kao što je prikazano na slici 4.2. Svi ostali indeksi računaju se imajući u vidu zamenu osa, što znači da će se prolazak po X osi vršiti od dole na gore, dok se u originalnom programu prolazak vršio sa desne na levu stranu.

5.6.9 Funkcija `calcluy`

U listingu 5.13 dat je izvorni kôd izmenjene funkcije `calcluy` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju `calcluy` iz fajla `imagtime3d-th.c` [11].

Listing 5.13: Izmenjeni/dodati delovi kôda u funkciji `calcluy` *iz fajla* `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na kôd funkcije* `calcluy` *iz fajla* `imagtime3d-th.c` *[11].*

```
void calcluy(double *psi, double **cbeta) {
```

```
	int threadid;
	long cnti, cntj, cntk;
	double c;

	#pragma omp parallel private(threadid, cnti, cntj, cntk, c)
	{
		threadid = omp_get_thread_num();

		#pragma omp for
		for(cnti = 0; cnti < Nx_slice; cnti ++) {
			for(cntk = 0; cntk < Nz; cntk ++) {
				cbeta[threadid][Ny - 2] = psi[cnti * Ny * Nz + (Ny – 1)
				                              * Nz + cntk];
				for (cntj = Ny - 2; cntj > 0; cntj --) {
					c = - Ay * psi[cnti * Ny * Nz + (cntj + 1) *
					           Nz + cntk] + Ay0r *
					         psi[cnti * Ny * Nz + cntj * Nz + cntk]
					    - Ay * psi[cnti * Ny * Nz + (cntj - 1) *
					           Nz + cntk];
					cbeta[threadid][cntj - 1] =  cgammay[cntj] *
					    (Ay * cbeta[threadid][cntj] - c);
				}
				psi[cnti * Ny * Nz + 0 * Nz + cntk] = 0.;
				for (cntj = 0; cntj < Ny - 2; cntj ++) {
					psi[cnti * Ny * Nz + (cntj + 1) * Nz + cntk] =
					      calphay[cntj] *
					      psi[cnti * Ny * Nz + cntj * Nz + cntk] +
					      cbeta[threadid][cntj];
				}
				psi[cnti * Ny * Nz + (Ny - 1) * Nz + cntk] = 0.;
			}
		}
	}

	return;
}
```

Izmene u funkciji `calcluy` su sledeće:

- Ulazni bafer talasne funkcije `psi` sada je jednodimenzionalni niz, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Gornja granica opsega na kome se izračunava vremenska propagacija u odnosu na H_3 deo Hamiltonijana je sada `Nx_slice - 1`, pošto je X osa distribuirana.

5.6.10 Funkcija `calcluz`

U listingu 5.14 dat je izvorni kôd izmenjene funkcije `calcluz` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71]. Žutom pozadinom su označene izmene u odnosu na funkciju `calcluz` iz fajla `imagtime3d-th.c` [11].

Listing 5.14: Izmenjeni/dodati delovi kôda u funkciji `calcluz` *iz fajla* `imatgtime3d-hyb.c` *(GP-SCL-HYB) [71] u odnosu na kôd funkcije* `calcluz` *iz fajla* `imagtime3d-th.c` *[11].*

```
void calcluz(double *psi, double **cbeta) {
      int threadid;
      long cnti, cntj, cntk;
      double c;

      #pragma omp parallel private(threadid, cnti, cntj, cntk, c)
      {
            threadid = omp_get_thread_num();

            #pragma omp for
            for(cnti = 0; cnti < Nx_slice; cnti ++) {
                  for(cntj = 0; cntj < Ny; cntj ++) {
                        cbeta[threadid][Nz - 2] = psi[cnti * Ny * Nz + cntj
                                                     * Nz + Nz - 1];
                        for (cntk = Nz - 2; cntk > 0; cntk --) {
                              c = - Az * psi[cnti * Ny * Nz + cntj * Nz +
                                          cntk + 1] + Az0r *
                                        psi[cnti * Ny * Nz + cntj * Nz + cntk]
                                  - Az * psi[cnti * Ny * Nz + cntj * Nz +
                                          cntk - 1];
                              cbeta[threadid][cntk - 1] =  cgammaz[cntk] *
                                  (Az * cbeta[threadid][cntk] - c);
                        }
                        psi[cnti * Ny * Nz + cntj * Nz + 0] = 0.;
                        for (cntk = 0; cntk < Nz - 2; cntk ++) {
                              psi[cnti * Ny * Nz + cntj * Nz + cntk + 1] =
                                 calphaz[cntk] *
                                 psi[cnti * Ny * Nz + cntj * Nz + cntk] +

              cbeta[threadid][cntk];
                        }
                        psi[cnti * Ny * Nz + cntj * Nz + Nz - 1] = 0.;
                  }
            }
      }

      return;
}
```

Izmene u funkciji `calcluz` su sledeće:

- Ulazni bafer talasne funkcije `psi` sada je jednodimenzionalni niz, što znači da će indeksiranje datog niza biti u skladu sa jednodimenzionalnim indeksiranjem niza u kome su podaci poređani duž Z ose.
- Gornja granica opsega na kome se izračunava vremenska propagacija u odnosu na H_4 deo Hamiltonijana je sada `Nx_slice - 1`, pošto je X osa distribuirana.

5.6.11 Funkcija `calcpot`

U listingu 5.15 dat je izvorni kôd nove funkcije `calcpot` iz fajla `imagtime3d-hyb.c` (projekat GP-SCL-HYB) [71].

Listing 5.15: Kôd nove funkcije `calcpot` iz fajla `imatgtime3d-hyb.c` (projekat GP-SCL-HYB) [71].

```
double calcpot(int cnti, int cntj, int cntk) {

      double kappa2 = kappa * kappa;
      double lambda2 = lambda * lambda;

      return  (multiplier  *  (x2[cnti]  +  kappa2  *  y2[cntj]  +  lambda2  *
z2[cntk]));
}
```

Funkcija `calcpot` je dodata kao pomoćna funkcija za izračunavanje potencijala u jednoj tački prostorne mreže sa koordinatama `cnti`, `cntj`, `cntk`. Ova funkcija zamenjuje statičko izračunavanje niza vrednosti potencijala `pot` koji se u originalnom kodu izračunava u funkciji `init` (sekcija 2.1.8.2). Na ovaj način se štedi memorijsko zauzeće od `Nx` * `Ny` * `Nz` vrednosti tipa `double` (veličina bafera talasne funkcije `psi`), što je značajno u slučajevima kada je prostorna mreža veoma gusta, kao što je u primeru sa mrežom veličine $1920 \times 1600 \times 1280$ tačaka. Tada se ušteda u zauzetoj memoriji meri desetinama gigabajta (oko 32 GB), što je veoma značajno, jer su računarski resursi svakog klastera ograničeni. U testnom okruženju, na klasteru PARADOX-IV u Laboratoriji za primenu računara u nauci Centra izuzetnih vrednosti za izučavanje kompleksnih sistema na Institutu za fiziku u Beogradu, svaki od čvorova poseduje 32 GB memorije, što znači da je samo ovom izmenom postignuta ušteda memorije dva čvora, pošto nije moguće zauzeti celokupnu radnu memoriju čvora za korisnički program.

Sa druge strane, uvođenjem funkcije `calcpot` povećava se količina računskih operacija u programu, ali ne značajno, jer se funkcija `calcpot` poziva na samo tri mesta u dve različite funkcije, `calcnu` i `cacmuen`.

5.7 Realizacija testiranja algoritma

Testiranje programa pisanih u distribuiranoj paradigmi pomoću MPI biblioteke predstavlja poseban izazov. Naročito zahtevno je testiranje programa koji se izvršavaju na udaljenom skupu računara (klasteru), a ne simulirano na lokalnom računaru. Testiranju algoritama težinu je dodala i sama priroda problema, tj. Krenk-Nikolson algoritam, koji opisuje kompleksni fizički sistem BAK i zahteva dodatno razumevanje netrivijalnih numeričkih procedura.

Svaki od prethodno opisanih algoritama nastalih tokom razvoja, sve do konačnog algoritma GP-SCL-HYB [71], posebno je testiran i verifikovan. U ovom poglavlju biće opisani načini testiranja algoritma GP-SCL-HYB. Zbog kompleksne prirode Krenk-Nikolson metoda, testiranje je vršeno na više nivoa programa:

- na nivou jedne linije koda,
- na nivou grupe linija koda (najčešće jedne ili više ugnježdenih petlji),
- na nivou jedne funkcije,
- na nivou celog programa,
- testiranje u okviru različitih niti na jednom čvoru,
- testiranje vrednosti na različitim čvorovima.

Testiranje je podrazumevalo direktno poređenje vrednosti promenljvih dobijenih pomoću algoritma GP-SCL-HYB [71] sa vrednostima promenljvih izračunatih pomoću prethodnih C/OpenMP programa [11], pri čemu je uslov korektnosti bio istovetnost odgovarajućih rezultata. Ovakvo poređenje je zahtevalo pažnju zbog podeljenog domena više promenljivih u algoritmu GP-SCL-HYB u odnosu na prethodni algoritam razvijen za okruženje sa deljenom memorijom.

Sledi opis pomoćnih funkcija za testiranje kao i ilustracija testiranja u okviru jedne funkcije.

5.7.1 Pomoćne funkcije za testiranje vrednosti promenljivih

U listingu 5.16 dat je izvorni kôd pomoćnih funkcija za testiranje programa `dvar_debug`, `lvar_debug`, `tensor_debug` i `tensor_rod_debug`.

Listing 5.16: Kôd funkcija `dvar_debug`, `lvar_debug`, `tensor_debug` *i* `tensor_rod_debug`.

```
/*Debag funkcije*/
void dvar_debug(double var) {
```

```
      if (rank == debug_rank) {
         fprintf(debug, "%8le\n", var);
      }
}

void lvar_debug(long var) {
      if (rank == debug_rank) {
         fprintf(debug, "%li\n", var);
      }
}

void tensor_debug(char * message, double * tensor, long x_ub,
                  long y_ub, long z_ub) {
   long cnti, cntj, cntk;
   if (rank == debug_rank) {
       fprintf(debug, "%s:\n", message);
       for(cnti = 0; cnti < x_ub; cnti ++) {
          for(cntj = 0; cntj < y_ub; cntj ++) {
             for(cntk = 0; cntk < z_ub; cntk ++) {
                fprintf(debug, "%8le\n", tensor[cnti * Ny * Nz +
                                      cntj * Nz + cntk]);
             }
          }
       }
    }
}

void tensor_rod_debug(char * message, double * tensor,
                     long x, long y, long z_ub,
                     long X_dimension) {  //0 = X, 1 = Y
    long cntk;
    long Y_dimension = (X_dimension == 0) ? Ny : Nx * gsize;
    if (rank == debug_rank) {
      // fprintf(debug, "%s:\n", message);
       for(cntk = 0; cntk < z_ub; cntk ++) {
          fprintf(debug, "%8le\n", tensor[x * Y_dimension *
                                  Nz + y * Nz + cntk]);
       }
    }
}
```

Sve četiri funkcije se koriste za testiranje ispravnosti pojedinih delova koda. Prve dve funkcije upisuju u fajl vrednost celobrojnih promenljvih i promenljivih u dvostrukoj preciznosti i to samo sa čvora koji je određen za testiranje, definisanog u globalnoj promenljivoj `debug_rank`. Bitno je naglasiti da je testiranje u distribuiranoj paradigmi najjednostavnije preko upisa vrednosti u fajl (pokazivač na fajl je promenljiva `debug`), zbog često velike količine podataka, koja bi se potencijalno ispisivala na ekran, što bi gotovo onemogućilo analizu ispisa. Dodatno, u realnom testnom okruženju na klasteru ispis na ekran je onemogućen i naknadno testiranje uz pomoć fajlova predstavlja jedinu opciju. Funkcije `tensor_debug i tensor_rod_debug` služe za testiranje vrednosti celokupnog ili parcijalnog tenzora trećeg reda (3D matrice), kao i celokupne ili parcijalne duži u okviru 3D matrice (pogledati sliku 4.2).

U listingu 5.17 prikazano je testiranje jedne od funkcija numeričkog algoritma, `calcluy`. Treba imati u vidu da je u listingu 5.17 prikazan samo mali deo ukupnog testiranja. Detaljno testiranje bilo je neophodno u svim delovima numeričkog algoritma na različitim čvorovima klastera.

Listing 5.17: Testiranje pojedinih delova kôda funkcije `calcluy` *(delovi kôda sa testnim funkcijama su zadebljani).*

```
void calcluy(double *psi, double **cbeta) {
   int threadid;
   long cnti, cntj, cntk;
   double c;

   tensor_debug("INSIDE CALCUY", psi, Nx, y_ub, z_ub);
   #pragma omp parallel private(threadid, cnti, cntj, cntk, c)
   {
      threadid = omp_get_thread_num();
      #pragma omp for
      for(cnti = 0; cnti < Nx; cnti ++) {
         for(cntk = 0; cntk < Nz; cntk ++) {
            cbeta[threadid][Ny - 2] = psi[cnti * Ny * Nz + (Ny - 1) *
                                          Nz + cntk];
            if (cnti == 1) {
               dvar_debug(psi[cnti * Ny * Nz + (Ny - 1) * Nz + cntk]);
               dvar_debug(psi[cnti * Ny * Nz + (0) * Nz + cntk]);
               dvar_debug(cbeta[threadid][Ny - 2]);
            }
            for (cntj = Ny - 2; cntj > 0; cntj --) {
               c = - Ay * psi[cnti * Ny * Nz + (cntj + 1) * Nz + cntk]
                   + Ay0r * psi[cnti * Ny * Nz + cntj * Nz + cntk] -
                    Ay * psi[cnti * Ny * Nz + (cntj - 1) * Nz + cntk];
              if (cnti == 1) {
                  dvar_debug(c);
               }
               cbeta[threadid][cntj - 1] =  cgammay[cntj] *
                                           (Ay * cbeta[threadid][cntj] - c);
               if (cnti == 1) {
                  dvar_debug(cbeta[threadid][cntj - 1]);
                  dvar_debug(cbeta[threadid][cntj]);
               }
            }
            psi[cnti * Ny * Nz + 0 * Nz + cntk] = 0.;
            for (cntj = 0; cntj < Ny - 2; cntj ++) {
              psi[cnti * Ny * Nz + (cntj + 1) * Nz + cntk] =
                  calphay[cntj] *
                  psi[cnti * Ny * Nz + cntj * Nz + cntk] +
                  cbeta[threadid][cntj];
              if (cnti < 12) {
                  dvar_debug(calphay[cntj]);
                  dvar_debug(cbeta[threadid][cntj]);
                  dvar_debug(calphay[cntj] *
                             psi[cnti * Ny * Nz + cntj * Nz + cntk] +
                             cbeta[threadid][cntj]);
                  dvar_debug(psi[cnti * Ny * Nz + cntj * Nz + cntk]);
               }
               if (cnti < 12 && cntk == 1 && cntj == 1) {
```

```
                    dvar_debug(psi[cnti * Ny * Nz + cntj * Nz + cntk - 1]);
                }
            }
            psi[cnti * Ny * Nz + (Ny - 1) * Nz + cntk] = 0.;
        }
    }
  }

  return;
}
```

U listingu 5.17 može se videti testiranje pojedinih delova kôda funkcije `calcuy`. Morali smo da vodimo računa o granicama testiranih jednodimenzionalnih i višedimenzionalnih nizova, tj. o različitim brojačima i njihovim opsezima, kao i o indeksiranju testiranih nizova.

6 Merenje efikasnosti i diskusija

Merenje efikasnosti, ubrzanja i skaliranja algoritma realizovanog u projektu GP-SCL-HYB [71] predstavlja bitan segment ove teze i potvrđuje validnost rešenja. Program koji bi sa aspekta efikasnosti postigao loše rezultate ne bi bio primenljiv u realnim uslovima, odnosno ne bi bio od interesa za krajnje korisnike, i stoga je bilo bitno izvršiti detaljno testiranje prethodno navedenih parametara uspešnosti, odnosno performansi programa koji su razvijeni u okviru ove teze. Merenje je izvršeno na klasteru PARADOX-IV u Laboratoriji za primenu računara u nauci Centra izuzetnih vrednosti za izučavanje kompleksnih sistema na Institutu za fiziku u Beogradu. Čvorovi klastera sadrže po dva „Sandy Bridge Xeon“ procesora sa 8 računarskih jezgara na radnoj frekvenciji 2.6 GHz, kao i 32 GB RAM memorije. Ukupno 106 čvorova klastera je povezano „Infiniband QDR“ („Quad Data Rate“) mrežom koja radi na brzini 40 Gbps.

6.1 Skripte za automatizovanje testiranja

S obzirom da je testiranje efikasnosti, ubrzanja i skaliranja u realnom okruženju zahtevalo pokretanje programa sa većim brojem različitih konfiguracija klastera, kao i sa većim brojem ulaznih parametara, korisno je bilo razviti automatizovane skripte za pokretanje različitih testnih scenarija. Skripte su se prevashodno sastojale od „bash“ fajlova za različite funkcionalnosti, čije bi ručno izvršavanje bilo dugotrajno za realizaciju i podložno greškama. Neke od realizovanih „bash“ skripti su:

- `clear_mpi-output.sh` – za brisanje svih prethodnih izlaznih fajlova koje je generisao program,
- `clear_std-out-err.sh` – za brisanje svih prethodnih izlaznih fajlova koji su nastali kao posledica određene greške u programu,
- `compile_imag3d-mpi` – za kompajliranje `imag3d` verzije programa,
- `compile_real3d-mpi` – za kompajliranje `real3d` verzije programa,
- `del_jobs.sh` – skripta za brisanje aktiviranih PBS poslova („jobs“) na klasteru,
- `run_imag3d_mpi.sh` – pokretanje `imag3d` verzije programa na klasteru sa odgovarajućim parametrima,
- `run_imag3d_mpi_local.sh` – pokretanje `imag3d` verzije programa na lokalnom računaru,

- `run_real3d_mpi.sh` – pokretanje `real3d` verzije programa na klasteru sa odgovarajućim parametrima,
- `run_real3d_mpi_local.sh` – pokretanje `real3d` verzije programa na lokalnom računaru,
- `start_N_node_jobs` – pokretanje PBS poslova na N čvorova klastera, gde N uzima vrednosti 1, 2, 4, 8, 16 i 32,
- `sub_jobs.sh` – skripta koja pokreće sve PBS poslove, za sve moguće vrednosti broja čvorova N.

Osim „bash“ skripti, važnu ulogu u atomatskim testovima imale su i gore pomenute PBS skripte tj. PBS poslovi. PARADOX-IV klaster na kojem je testiran program poseduje TORQUE („Terascale Open-source Resource and QUEue Manager“) sistem za raspoređivanje PBS poslova. U zavisnosti od zauzeća klastera, TORQUE raspoređivač omogućava izvršenje pojedinih poslova koji su poslati na klaster kada za to postoje dostupni resursi, u skladu sa definisanim prioritetima sistema i pravima pojedinih korisnika. U listingu 6.1 prikazan je jedan tipičan PBS posao.

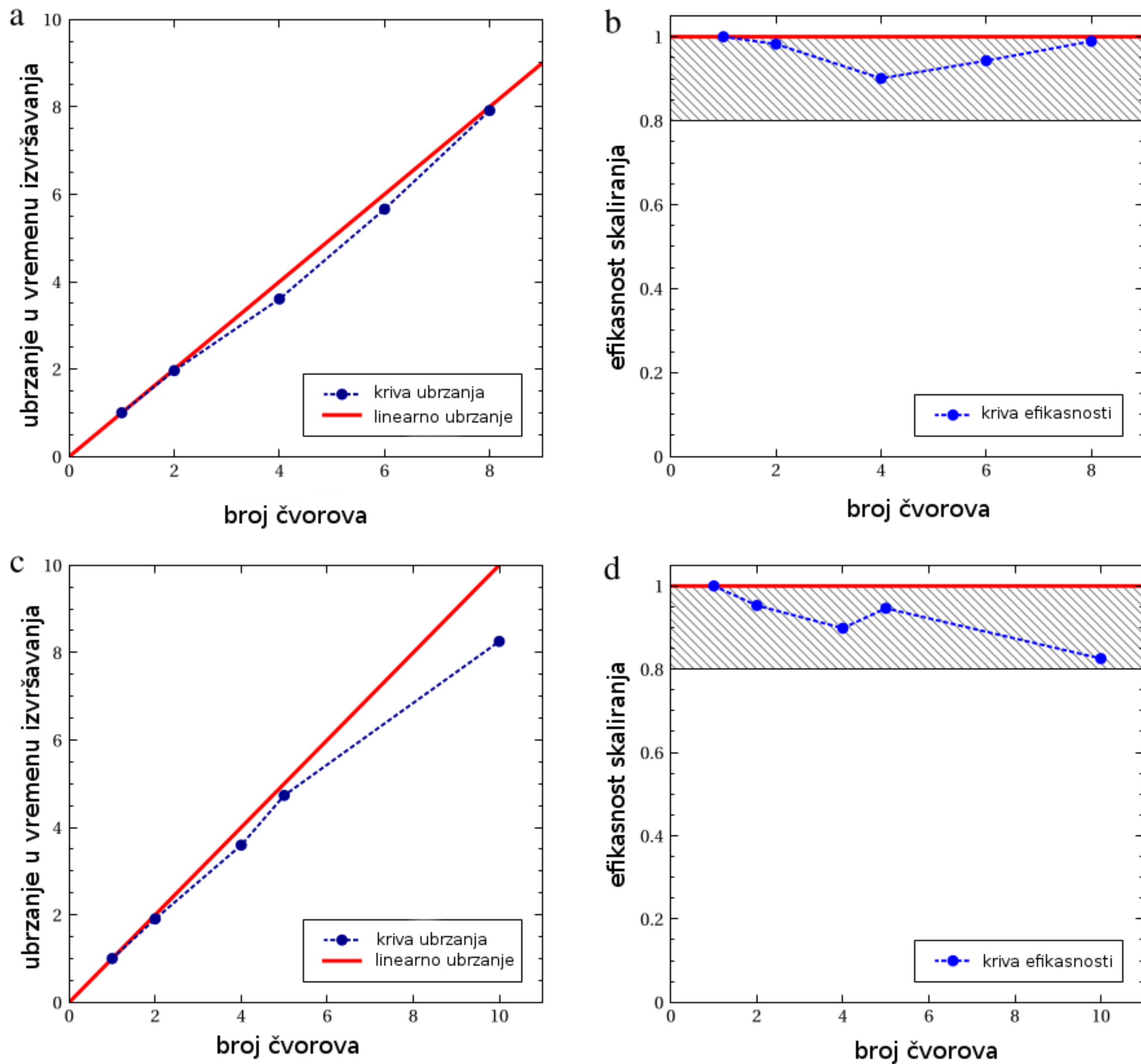
Listing 6.1: Izgled jednog fajla koji omogućava izvršavanje simulacije kroz PBS sistem.

```
#!/bin/bash
#PBS -q standard
#PBS -l nodes=16:ppn=16
#PBS -l walltime=10:00:00
#PBS -e std-out-err/imag3d/NP16/${PBS_ARRAYID}_NP16_THREADS16.err
#PBS -o std-out-err/imag3d/NP16/${PBS_ARRAYID}_NP16_THREADS16.out
#PBS -t 1-5
module load openmpi/1.6.5

export OMP_NUM_THREADS=16

cd $PBS_O_WORKDIR
chmod +x run_imag3d_mpi.sh

time mpirun -np 16 -npernode 1 ./run_imag3d_mpi.sh ${PBS_ARRAYID} #dodati 1
kao drugi parametar za regularno izvrsavanje (neskraceno)
```

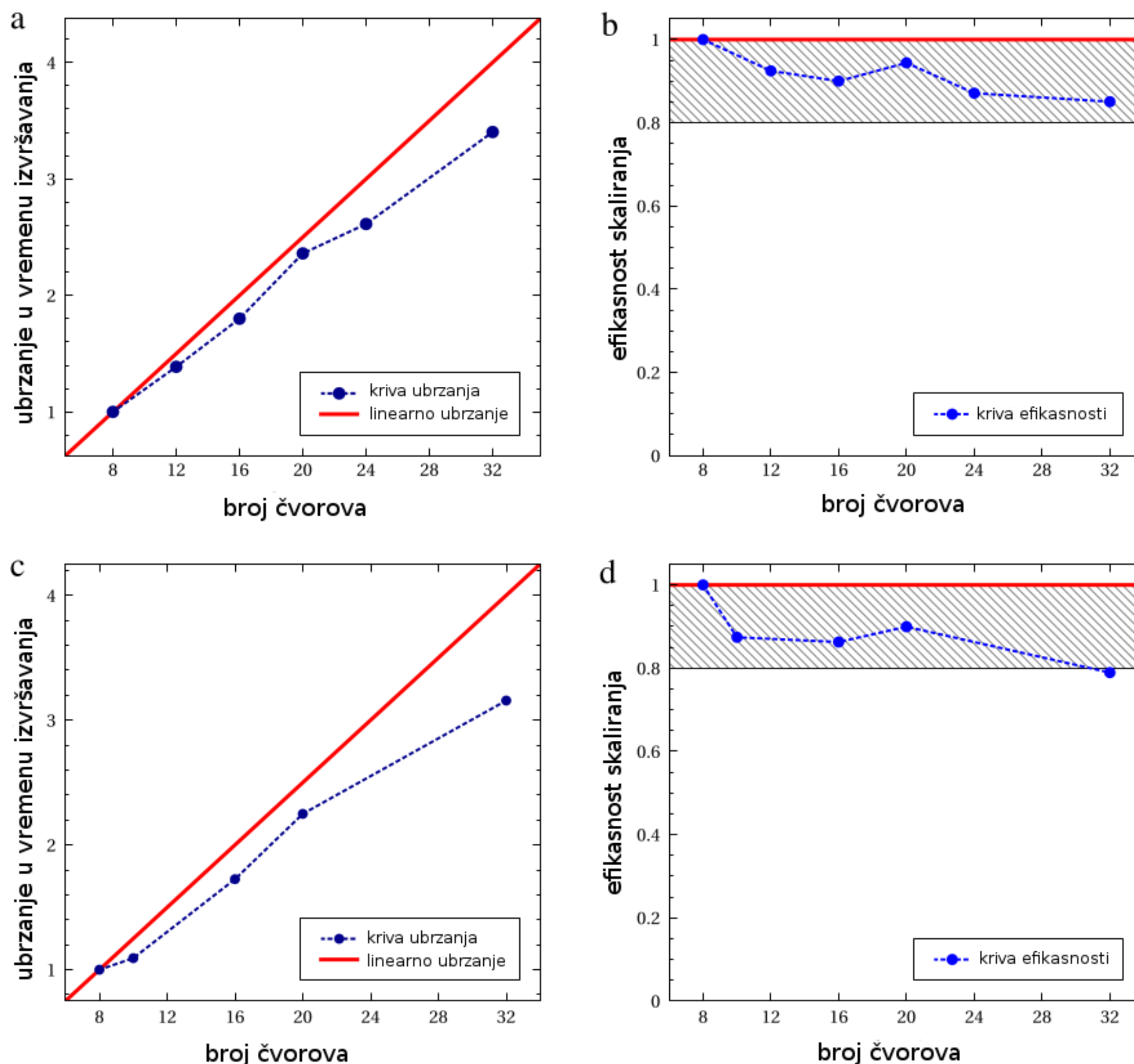
U prethodnom listingu prikazan je PBS posao koji pokreće `imag3d` program na 16 čvorova klastera (`nodes` parametar), pri čemu svaki čvor koristi 16 niti (`ppn` i `OMP_NUM_THREADS` parametri). Maksimalno dozvoljeno trajanje programa je 10 časova (`walltime` parametar), a izlazi i greške u programu će biti upisani u odgovarajućim direktorijumima, koji su dodatno označeni brojem PBS posla (`PBS_ARRAYID`). Pored ovoga, PBS posao pokreće izvršavanje pet ulaznih konfiguracija, tj. veličina 3D matrice, što se vidi kroz PBS argument „`-t 1-5`“ koji uzima iste vrednosti kao i parametar `PBS_ARRAYID`. Ovo znači da će izlazni fajlovi imati prepoznatljiv naziv u zavisnosti od ulaza. Npr. izlazni fajl za ulaznu veličinu 3D matrice $480\times400\times320$ (ulazna konfiguracija broj 2) će imati relativnu putanju `imag3d/NP16/2_NP16_THREADS16.out`.

6.2 Performanse hibridnog rešenja

Slike 6.1 i 6.2 prikazuju rezultate testiranja skalabilnosti hibridnih verzija programa [71] za male i velike rezolucije sistema (odnosno, male i velike distribuirane 3D matrice) kao funkciju broja korišćenih MPI čvorova.



Slika 6.1: Krive ubrzanja vremena izvršavanja i efikasnosti programa `imagtime3d-hyb` *i* `realtime3d-hyb` [71] *kao funkcije broja MPI čvorova za male rezolucije 3D matrice: (a) ubrzanje* `imagtime3d-hyb` *za 3D matricu veličine* $240\times200\times160$ *; (b) efikasnost* `imagtime3d-hyb` *za 3D matricu veličine* $240\times200\times160$ *; (c) ubrzanje* `realtime3d-hyb` *za 3D matricu veličine* $200\times160\times120$ *; (d) efikasnost za* `realtime3d-hyb` *za 3D matricu veličine* $200\times160\times120$ *. Zasenčeni regioni na grafovima (b) i (d) predstavljaju regione visoke efikasnosti, sa ubrzanjem barem 80% od idealnog.*

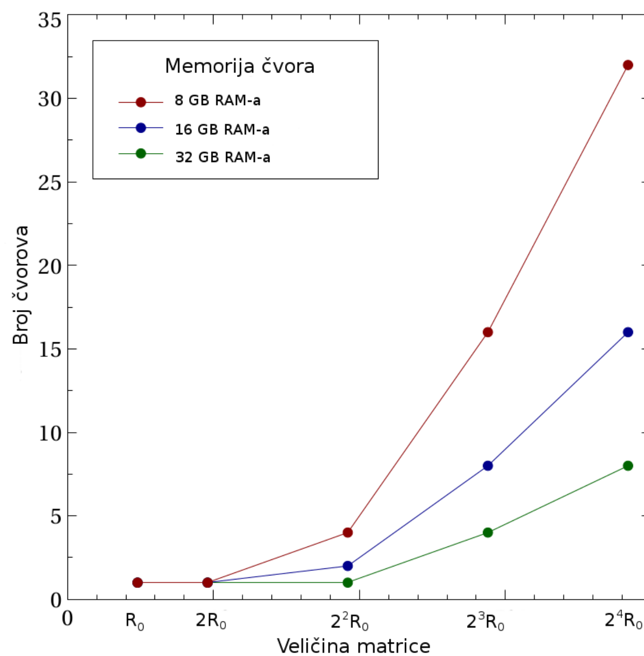


Slika 6.2: Krive ubrzanja vremena izvršavanja i efikasnosti programa `imagtime3d-hyb` *i* `realtime3d-hyb` [71] *kao funkcije broja MPI čvorova za velike rezolucije 3D matrice*: *(a) ubrzanje* `imagtime3d-hyb` *za 3D matricu veličine* 1920×1600×1280 *; (b) efikasnost* `imagtime3d-hyb` *za 3D matricu veličine* 1920×1600×1280 *; (c) ubrzanje* `realtime3d-hyb` *za 3D matricu veličine* 1920×1600×1280 *; (d) efikasnost* `realtime3d-hyb` *za 3D matricu veličine* 1920×1600×1280 *. Zasenčeni regioni na grafovima (b) i (d) predstavljaju regione visoke efikasnosti, sa ubrzanjem barem 80% od idealnog.*

Osnova za računanje ubrzanja vremena izvršavanja za male 3D matrice je prethodno, C/OpenMP rešenje [11], dok je za velike rezolucije, koje ne mogu da se izvrše na jednom čvoru klastera, pa njih nije ni moguće simulirati pomoću prethodnih C/OpenMP programa, osnova je vreme izvršavanja hibridnih programa [71] sa minimalnom konfiguracijom koji se pokreće na 8 čvorova. Slike takođe pokazuju rezultate za efikasnost, definisanu kao procenat izmerenog ubrzanja u odnosu na idealno moguće ubrzanje. Može se primetiti odlična skalabilnost (preko 80% od idealne) za upotrebu programa na klasteru veličine do 32 čvora. Bitno je naglasiti da skalabilnost

značajno zavisi od odnosa između vremena utrošenog na računske operacije i komunikacije u jednoj iteraciji algoritma. Za izbor optimalnog broja čvorova pri zadatim ostalim parametrima potrebna je posebna analiza za drugačije tipove procesora, kao i za druge tipove mrežne tehnologije.

Pored krivih ubrzanja i efikasnosti, na slici 6.3 prikazana je i kriva memorijskog opterećenja klastera u slučaju programa `realtime3d-hyb` (memorijski zahtevnija verzija programa) [71], tj. zavisnost broja čvorova koje je neophodno angažovati za odgovarajuću veličinu 3D matrice, pri čemu je ova zavisnost data parametarski, za različite veličine RAM memorije čvorova klasteru. Za program `imagtime3d-hyb` i 3D matricu veličine $240\times200\times160$ neophodno je 300 MB RAM memorije, a znamo da je zauzeće memorije proporcionalno sa Nx * Ny * Nz. Za program `realtime3d-hyb` i 3D matricu veličine $200\times160\times120$ neophodno je 410 MB RAM memorije, a zauzeće se skalira na isti način.



Slika 6.3: Memorijski zahtevi programa `realtime3d-hyb` *u zavisnosti od veličine 3D matrice, gde je* $R_0 = 200\times160\times120$.

7 Zaključak

Istraživanje predstavljeno u ovoj tezi obuhvata razvoj paralelnih algoritama za transponovanje distribuiranih trodimenzionalnih struktura podataka u okviru paradigme distribuirane memorije, odnosno računarskih klastera i MPI programskog okruženja. Na osnovu ovih algoritama i njihove tehničke implementacije, razvili smo hibridne C/OpenMP/MPI programe [71] za numeričku simulaciju propagacije u imaginarnom i realnom vremenu parcijalanih diferencijalnih jednačina tipa Gros-Pitaevski, koje se koriste za opisivanje Boze-Ajnštajn kondenzovanih sistema ultrahladnih atomskih gasova. Razmotrili smo različite izazove nastale tokom razvoja hibridnih programa i predstavili validan i efikasan pristup za njihovo rešavanje. Realizovali smo i skripte za automatsko testiranje hibridnih programa za veći broj ulaza, kao i za veći broj konfiguracija klastera. Predstavili smo tipične rezultate merenja ubrzanja vremena izvršavanja, skaliranja i efikasnosti, koji pokazuju da hibridni C/OpenMP/MPI programi postižu gotovo linearno ubrzanje u odnosu na prethodne C/OpenMP programe [11]. Dobijene performanse i validnost i korektnost dobijenih rešenja pomoću ovih programa realizovanih u projektu GP-SCL-HYB pokazuju da su ciljevi istraživanja, predočeni u poglavlju 1.7.2, u potpunosti ostvareni.

7.1 Doprinos rada i mogućnosti primene

Glavni doprinos ovog rada je razvoj paralelnog algoritma za transponovanje distribuiranih trodimenzionalnih struktura podataka, koji predstavlja uopštenje ranijeg algoritma [15]. Drugi važan i praktično primenljiv doprinos je tehnička implementacija razvijenog algoritma u C/OpenMP/MPI programskom okruženju, koje može da se koristi za rešavanje jednačina tipa Gros-Pitaevski pomoću poluimplicitne Krenk-Nikolson metode deljenog koraka, i to kako za nalaženje stacionarnih stanja pomoću propagacije u imaginarnom vremenu, tako i za proučavanje dinamičke vremenske evolucije propagacijom u realnom vremenu. Iz navedenog, može se zaključiti da hibridni C/OpenMP/MPI programi predstavljaju potencijalni alat zainteresovanim istraživačima koji se bave proučavanjem ultrahladnih atoma, nelinearne optike i drugih oblasti nauke [20-70]. Ovi programi omogućavaju ispitivanje odgovarajućih fizičkih sistema u velikim rezolucijama, što je od ključnog značaja za primene kod kojih u sistemima postoji nekoliko različitih dužinskih skala koje se bitno razlikuju. Nekoliko istraživačkih grupa je već započelo sa korišćenjem naših programa [72-87].

7.2 Pravci daljeg istraživanja

Mogući pravci daljeg istraživanja na ovoj temi obuhvataju:

1. Proširenje projekta GP-SCL-HYB sa ciljem da obuhvati još jedan nivo hibridizacije. Dodati nivo bi predstavljao ubrzanje lokalnih proračuna na grafičkim karticama (CUDA [88] ili OpenACC), umesto samo na CPU.
2. Dodatna optimizacija kôda kojom bi se postigao viši nivo upotrebe keš memorije, uz smanjenje upotrebe radne memorije.
3. Razvoj automatskog softvera koji bi pronalazio optimalnu konfiguraciju klastera (broj čvorova i niti) za određenu veličinu ulaza i ponuđene računske resurse klastera.

8 Bibliografija

[1] F. Dalfovo, S. Giorgini, L.P. Pitaevskii, S. Stringari, Rev. Mod. Phys. 71 (1999) 463.

[2] C.J. Pethick, H. Smith, Bose–Einstein Condensation in Dilute Gases, Cambridge University Press, Cambridge, 2002.

[3] L. Pitaevskii, S. Stringari, Bose–Einstein Condensation, Oxford University Press, Oxford, 2003.

[4] M. H. Anderson, J. R. Ensher, M. R. Matthews, C. E. Wieman, and E. A. Cornell, Science 269 (1995) 198.

[5] K. B. Davis, M. O. Mewes, M. R. Andrews, N. J. van Druten, D. S. Durfee, D. M. Kurn, and W. Ketterle, Phys. Rev. Lett. 75 (1995) 3969.

[6] C. C. Bradley, C. A. Sackett, J. J. Tollett, and R. G. Hulet, Phys. Rev. Lett. 75 (1995) 1687.

[7] S.E. Koonin, D.C. Meredith, Computational Physics: Fortran version, Addison-Wesley, Reading, 1990.

[8] W.F. Ames, Numerical Methods for Partial Differential Equations, 3rd ed., Academic Press, New York, 1992.

[9] R. Dautray, J.L. Lions, Mathematical Analysis and Numerical Methods for Science and Technology, vol. 6, Springer-Verlag, Berlin, 1993 (Ch. XX, Sec. 2).

[10] P. Muruganandam, S. K. Adhikari, Comput. Phys. Commun. 180 (2009) 1888

[11] D. Vudragović, I. Vidanović, A. Balaž, P. Muruganandam, S. K. Adhikari, Comput. Phys. Commun. 183 (2012) 2021.

[12] K.E. Atkinson, An Introduction to Numerical Analysis, John Wiley & Sons, 1989.

[13] G. Birkhoff, G-C. Rota, Ordinary Differential Equations, John Wiley & Sons, 1978.

[14] MPI: A Message-Passing Interface Standard Version 3.0, http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[15] J. Choi, J. J. Dongarra, D. W. Walker, Parallel matrix transpose algorithms on distributed memory concurrent computers, Parallel. Comput. 21 (1995) 1387.

[16] Netlib repository, http://www.netlib.org/liblist.html

[17] A.Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Second Edition, Addison-Wesley, 2003.

[18] P. F. Windley, Comput. J. 2 (1959) 47.

[19] Intel MKL library documentation, https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation

[20] R. K. Kumar and P. Muruganandam, J. Phys. B: At. Mol. Opt. Phys. 45 (2012) 215301; L. E. Young-S. and S. K. Adhikari, Phys. Rev. A 86 (2012) 063611.

[21] S. K. Adhikari, J. Phys. B: At. Mol. Opt. Phys. 45 (2012) 235303.

[22] I. Vidanović, N. J. van Druten, and M. Haque, New J. Phys. 15 (2013) 035008.

[23] S. Balasubramanian, R. Ramaswamy, and A. I. Nicolin, Rom. Rep. Phys. 65 (2013) 820.

[24] L. E. Young-S. and S. K. Adhikari, Phys. Rev. A 87 (2013) 013618.

[25] H. Al-Jibbouri, I. Vidanović, A. Balaž, and A. Pelster, J. Phys. B: At. Mol. Opt. Phys. 46 (2013) 065303.

[26] X. Antoine, W. Bao, and C. Besse, Comput. Phys. Commun. 184 (2013) 2621.

[27] B. Nikolić, A. Balaž, and A. Pelster, Phys. Rev. A 88 (2013) 013624.

[28] H. Al-Jibbouri and A. Pelster, Phys. Rev. A 88 (2013) 033621.

[29] S. K. Adhikari, Phys. Rev. A 88 (2013) 043603.

[30] J. B. Sudharsan, R. Radha, and P. Muruganandam, J. Phys. B: At. Mol. Opt. Phys. 46 (2013) 155302.

[31] R. R. Sakhel, A. R. Sakhel, and H. B. Ghassib, J. Low Temp. Phys. 173 (2013) 177.

[32] E. J. M. Madarassy and V. T. Toth, Comput. Phys. Commun. 184 (2013) 1339.

[33] R. K. Kumar, P. Muruganandam, and B. A. Malomed, J. Phys. B: At. Mol. Opt. Phys. 46 (2013) 175302.

[34] W. Bao, Q. Tang, and Z. Xu, J. Comput. Phys. 235 (2013) 423.

[35] A. I. Nicolin, Proc. Rom. Acad. Ser. A-Math. Phys. 14 (2013) 35.

[36] R. M. Caplan, Comput. Phys. Commun. 184 (2013) 1250.

[37] S. K. Adhikari, J. Phys. B: At. Mol. Opt. Phys. 46 (2013) 115301.

[38] Ž. Marojević. E. Göklü , and C. Lä Ammerzahl, Comput. Phys. Commun. 184 (2013) 1920.

[39] X. Antoine and R. Duboscq, Comput. Phys. Commun. 185 (2014) 2969.

[40] S. K. Adhikari and L. E. Young-S, J. Phys. B: At. Mol. Opt. Phys. 47 (2014) 015302.

[41] K. Manikandan, P. Muruganandam, M. Senthilvelan, and M. Lakshmanan, Phys. Rev. E 90 (2014) 062905.

[42] S. K. Adhikari, Phys. Rev. A 90 (2014) 055601.

[43] A. Balaž, R. Paun, A. I. Nicolin, S. Balasubramanian, and R. Ramaswamy, Phys. Rev. A 89 (2014) 023609.

[44] S. K. Adhikari, Phys. Rev. A 89 (2014) 013630.

[45] J. Luo, Commun. Nonlinear Sci. Numer. Simul. 19 (2014) 3591.

[46] S. K. Adhikari, Phys. Rev. A 89 (2014) 043609.

[47] K.-T. Xi, J. Li, and D.-N. Shi, Physica B 436 (2014) 149.

[48] M. C. Raportaru, J. Jovanovski, B. Jakimovski, D. Jakimovski, and A. Mishev, Rom. J. Phys. 59 (2014) 677.

[49] S. Gautam and S. K. Adhikari, Phys. Rev. A 90 (2014) 043619.

[50] A. I. Nicolin, A. Balaž, J. B. Sudharsan, and R. Radha, Rom. J. Phys. 59 (2014) 204.

[51] K. Sakkaravarthi, T. Kanna, M. Vijayajayanthi, and M. Lakshmanan, Phys. Rev. E 90 (2014) 052912.

[52] S. K. Adhikari, J. Phys. B: At. Mol. Opt. Phys. 47 (2014) 225304.

[53] R. K. Kumar and P. Muruganandam, Proceedings of the 22nd International Laser Physics Workshop, J. Phys. Conf. Ser. 497 (2014) 012036.

[54] A. I. Nicolin and I. Rata, Density waves in dipolar Bose-Einstein condensates by means of symbolic computations, High-Performance Computing Infrastructure for South East Europe's Research Communities: Results of the HP-SEE User Forum 2012, in Springer Series: Modeling and Optimization in Science and Technologies 2 (2014) 15.

[55] S. K. Adhikari, Phys. Rev. A 89 (2014) 043615.

[56] R. K. Kumar and P. Muruganandam, Eur. Phys. J. D 68 (2014) 289.

[57] J. B. Sudharsan, R. Radha, H. Fabrelli, A. Gammal, and B. A. Malomed, Phys. Rev. A 92 (2015) 053601.

[58] S. K. Adhikari, J. Phys. B: At. Mol. Opt. Phys. 48 (2015) 165303.

[59] F. I. Moxley III, T. Byrnes, B. Ma, Y. Yan, and W. Dai, J. Comput. Phys. 282 (2015) 303.

[60] S. K. Adhikari, Phys. Rev. E 92 (2015) 042926.

[61] R. R. Sakhel, A. R. Sakhel, and H. B. Ghassib, Physica B 478 (2015) 68.

[62] S. Gautam and S. K. Adhikari, Phys. Rev. A 92 (2015) 023616.

[63] D. Novoa, D. Tommasini, and J. A. Nóvoa-López, Phys. Rev. E 91 (2015) 012904.

[64] S. Gautam and S. K. Adhikari, Laser Phys. Lett. 12 (2015) 045501.

[65] K.-T. Xi, J. Li, and D.-N. Shi, Physica B 459 (2015) 6.

[66] R. K. Kumar, L. E. Young-S., D. Vudragović, A. Balaž, P. Muruganandam, and S. K. Adhikari, Comput. Phys. Commun. 195 (2015) 117.

[67] S. Gautam and S. K. Adhikari, Phys. Rev. A 91 (2015) 013624.

[68] A. I. Nicolin, M. C. Raportaru, and A. Balaž, Rom. Rep. Phys. 67 (2015) 143.

[69] S. Gautam and S. K. Adhikari, Phys. Rev. A 91 (2015) 063617.

[70] E. J. M. Madarassy and V. T. Toth, Phys. Rev. D 91 (2015) 044041.

[71] B. Satarić, V. Slavnić, A. Belić, A. Balaž, P. Muruganandam, and S. K. Adhikari, Comput. Phys. Commun. 200 (2016) 411.

[72] S. K. Adhikari, Laser Phys. Lett. 14 (2017) 025501.

[73] S. Gautam and S. K. Adhikari, Phys. Rev. A 95 (2017) 013608.

[74] S. Bhuvaneswari, K. Nithyanandan, P. Muruganandam, J. Phys. B: At. Mol. Opt. Phys. 49 (2016) 245301.

[75] H. Gargoubi, T. Guillet; S. Jaziri, Phys. Rev. E 94 (2016) 043310.

[76] S. K. Adhikari, Phys. Rev. E 94 (2016) 032217.

[77] I. Vasić, A. Balaž, Phys. Rev. A 94 (2016) 033627.

[78] R. R. Sakhel, R. A. Sakhel, Journal of Low Temperature Phys. 184 (2016) 1092.

[79] J.B. Sudharsan, R. Radha, M. C. Raportaru, J. Phys. B: At. Mol. Opt. Phys. 49 (2016) 165303.

[80] S. K. Adhikari, Laser Phys. Lett. 13 (2016) 085501.

[81] L. E. Young-S, D. Vudragović, P. Muruganandam, Comput. Phys. Commun. 204 (2016) 209.

[82] T. Khellil, A. Balaž, A. Pelster, New J. Phys. 18 (2016) 063003.

[83] J. Akram, B. Girodias, A. Pelster, J. Phys. B: At. Mol. Opt. Phys. 49 (2016) 075302.

[84] K. Manikandan, P. Muruganandam, M. Senthilvelan, Phys. Rev. E 93 (2016) 032212.

[85] J. Akram, A. Pelster, Phys. Rev. A 93 (2016) 033610.

[86] S. K. Adhikari, Laser Phys. Lett. 13 (2016) 035502.

[87] J. Akram, A. Pelster, Phys. Rev. A 93 (2016) 023606.

[88] V. Lončar, L. Young-S, S. Škrbić, P. Muruganandam, S. K. Adhikari and A. Balaž, Comput. Phys. Commun. 209 (2016) 190.

9 Indeks slika

Slika 1.1: Eksperimentalni rezultati grupe Volfganga Keterlea iz 1995. godine [5] kojima je pokazana pojava Boze-Ajnštajn kondenzacije ispod kritične temperature. (By NIST/JILA/CU-Boulder - NIST Image, Public Domain, https://commons.wikimedia.org/w/index.php?curid=403804)....2

Slika 1.2: Shematski prikaz Boze-Ajnštajn kondenzacije prilikom smanjenja temperature sistema, inspirisan Nobelovom lekcijom Volfganga Keterlea 2001. godine....4

Slika 1.3: Zavisnost kondenzovane frakcije atoma od temperature. BAK fazni prelaz se događa na temperaturia ispod ove temperature je osnovno energetsko stanje sistema makroskopski naseljeno..5

Slika 2.1: Alociranje matrice uz pomoć funkcije allocate_double_matrix....37

Slika 2.2: Alociranje tenzora uz pomoć funkcije allocate_double_tensor....38

Slika 2.3: Preseci talasne funkcije se posmatraju duž odgovarajuće ose X, Y i Z. Na primer, presek duž X ose sadrži vrednosti talasne funkcije psi[i][Ny2][Nz2], gde brojač i uzima Nx vrednosti, od 0 do Nx - 1....48

Slika 2.4: Ravnomerna distribucija podataka po X osi na računarskom klasteru. Različite boje predstavlju skupove podataka koje su dodeljene različitim čvorovima (u ovom slučaju 4 čvora)....67

Slika 2.5: 3D transponovanje distribuiranih podataka između čvorova klastera opisano u tekstu....68

Slika 3.1: Shematski prikaz matrice računara (klaster) za P=2, Q=3....70

Slika 3.2: Blokovska distribucija podataka [14]....72

Slika 3.3: Blokovska ciklična distribucija podataka [14]....72

Slika 3.4: Ciklična blokovska distribucija matrice za P=2, Q=3....73

Slika 3.5: NZS blokovska distribucija matrice za P=2, Q=3....73

Slika 3.6: Transponovanje matrice A....74

Slika 3.7: Transponovanje matrice A (sa stanovišta čvorova klastera)....74

Slika 3.8: Transponovanje matrice sa stanovišta blokova koji se transponuju za P = 2, Q = 3 i Mb = Nb = 6....76

Slika 3.9: Transponovanje matrice u slučaju kada je P = 4 i Q = 6....78

Slika 3.10: Mapa komunikacije čvorova u slučaju kada je NZD > 1....78

Slika 3.11: Transponovanje nulte i prve dijagonale matrice za P = 4, Q = 6 i Mb = Nb = 12....79

Slika 3.12: Transponovanje matrice kada je P = Q = NZD = 3. Čvorovi transponuju sve tri NZD dijagonale u jednom koraku....80

Slika 4.1: Ideja 3D transponovanja u algoritmu MT_v_8.0....91

Slika 4.2: Struktura podataka talasne funkcije BAK u primenjenom algoritmu. Podaci su distribuirani tako da čvorovi poseduju deo podataka duž X ose i kompletne podatke po Y i Z osi. Slike prikazuju i parametre za kreiranje MPI indexed tipa u slučaju pošiljaoca (a) i primaoca (b). [71]....95

Slika 4.3: Kreacija korisničkog MPI tipa podataka indextype uz pomoć funkcije MPI_Type_indexed [71]..97
Slika 6.1: Krive ubrzanja vremena izvršavanja i efikasnosti programa imagtime3d-hyb i realtime3d-hyb [71] kao funkcije broja MPI čvorova za male rezolucije 3D matrice: (a) ubrzanje imagtime3d-hyb za 3D matricu veličine; (b) efikasnost imagtime3d-hyb za 3D matricu veličine; (c) ubrzanje realtime3d-hyb za 3D matricu veličine; (d) efikasnost za realtime3d-hyb za 3D matricu veličine. Zasenčeni regioni na grafovima (b) i (d) predstavljaju regione visoke efikasnosti, sa ubrzanjem barem 80% od idealnog...139
Slika 6.2: Krive ubrzanja vremena izvršavanja i efikasnosti programa imagtime3d-hyb i realtime3d-hyb [71] kao funkcije broja MPI čvorova za velike rezolucije 3D matrice: (a) ubrzanje imagtime3d-hyb za 3D matricu veličine; (b) efikasnost imagtime3d-hyb za 3D matricu veličine; (c) ubrzanje realtime3d-hyb za 3D matricu veličine; (d) efikasnost realtime3d-hyb za 3D matricu veličine . Zasenčeni regioni na grafovima (b) i (d) predstavljaju regione visoke efikasnosti, sa ubrzanjem barem 80% od idealnog...140
Slika 6.3: Memorijski zahtevi programa realtime3d-hyb u zavisnosti od veličine 3D matrice, gde je R0 =...141

10 Indeks listinga

Listing 2.1: Izvorni kôd zaglavlja imagtime3d-th.h......21
Listing 2.2: Izvorni kôd zaglavlja cfg.h......24
Listing 2.3: Izvorni kôd fajla cfg.c......25
Listing 2.4: Izgled konfiguracionog fajla glavnog programa......26
Listing 2.5: Izvorni kôd fajla diffint.h......29
Listing 2.6: Izvorni kôd fajla diffint.c......29
Listing 2.7: Izvorni kôd fajla mem.h......31
Listing 2.8: Izvorni kôd fajla mem.c......33
Listing 2.9: Izvorni kôd fajla imagtime3d-th.c......40
Listing 2.10: Izgled izlaznog fajla programa za propagaciju u imaginarnom vremenu......47
Listing 2.11: Izvorni kôd funkcije readpar......50
Listing 2.12: Izvorni kôd funkcije init......52
Listing 2.13: Izvorni kôd funkcije gencoef......53
Listing 2.14: Izvorni kôd funkcije calcnorm......54
Listing 2.15: Izvorni kôd funkcije calcmuen......56
Listing 2.16: Izvorni kôd funkcije calcrms......59
Listing 2.17: Izvorni kôd funkcije calcnu......60
Listing 2.18: Izvorni kôd funkcije calclux......60
Listing 2.19: Izvorni kôd funkcije calcluy......62
Listing 2.20: Izvorni kôd funkcije calcluz......63
Listing 3.1: Primer stvaranja dvodimenzionalne MPI topologije......71
Listing 3.2: Pseudokôd algoritma paralelnog transponovanja 2D matrice sa stanovišta čvora (MPI procesa) kada su P i Q prosti brojevi (NZD = 1)......75
Listing 3.3: Pseudokôd algoritma paralelnog transponovanja 2D matrice sa stanovišta blokova matrice koji se transponuju kada su P i Q međusobno prosti brojevi (NZD = 1). Jedan dijagonalni blok se transponuje u jednom koraku. Oznaka (start : granica : intv) predstavlja vrednosti x gde je x = start + intv * y, y = 0, 1, …, pri čemu x ne može da pređe zadatu granicu......77
Listing 3.4: Pseudokôd algoritma transponovanja sa stanovišta čvorova u slučaju kada je NZD > 1. Operacije NZD grupa procesora se preklapaju......78
Listing 3.5: Pseudokôd algoritma paralelnog transponovanja 2D matrice sa stanovišta blokova matrice koji se transponuju kada P i Q nisu međusobno prosti brojevi (NZD > 1). NZD dijagonala matrice se transponuju simultano......80

Listing 4.1: Inicijalizacija MPI okruženja i najvažnijih promenljivih u glavnom programu.............82
Listing 4.2: Čitanje blokova matrica na jednom čvoru u cikličnom rasporedu.................................83
Listing 4.3: Lokalno transponovanje blokova u okviru jednog čvora..84
Listing 4.4: Relevantan deo funkcije transpose_prime..84
Listing 4.5: Relevantan deo funkcije transpose_not_prime...86
Listing 4.6: Kôd za proveru MPI transponovanja uz pomoć Intel MKL funkcija............................90
Listing 4.7: Telo funkcije transpose_local..92
Listing 4.8: Telo izmenjene funkcije transpose_prime...93
Listing 5.1: Izmene fajla imagtime3d-th.h kod hibridnog rešenja [71]..99
Listing 5.2: Izmene fajla mem.h kod hibridnog rešenja [71]..103
Listing 5.3: Izmene fajla mem.c kod hibridnog rešenja [71]..104
Listing 5.4: Dodato zaglavlje misc.h kod hibridnog rešenja [71]...104
Listing 5.5: Dodatni fajl misc.c kod hibridnog rešenja [71]...105
Listing 5.6: Izmenjeni/dodati delovi kôda u fajlu imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na odgovarajući kôd iz fajla imagtime3d-th.c prethodnog rešenja [11]..106
Listing 5.7: Izmenjeni/dodati delovi kôda u funkciji init iz fajla imatgtime3d-hyb.c (projekat GP-SCL-HYB) [71] u odnosu na kôd funkcije init iz fajla imagtime3d-th.c [11].................................118
Listing 5.8: Izmenjeni/dodati delovi kôda u funkciji calcnorm iz fajla imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na kôd funkcije calcnorm iz fajla imagtime3d-th.c [11]................................121
Listing 5.9: Izmenjeni/dodati delovi kôda u funkciji calcmuen iz fajla imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na kôd funkcije calcmuen iz fajla imagtime3d-th.c [11]...............................122
Listing 5.10: Izmenjeni/dodati delovi kôda u funkciji calcrms iz fajla imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na kôd funkcije calcrms iz fajla imagtime3d-th.c [11]..................................126
Listing 5.11: Izmenjeni/dodati delovi kôda u funkciji calcnu iz fajla imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na kôd funkcije calcnu iz fajla imagtime3d-th.c [11]....................................127
Listing 5.12: Izmenjeni/dodati delovi kôda u funkciji calclux (calclux_trans) iz fajla imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na kôd funkcije calclux iz fajla imagtime3d-th.c [11]..........128
Listing 5.13: Izmenjeni/dodati delovi kôda u funkciji calcluy iz fajla imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na kôd funkcije calcluy iz fajla imagtime3d-th.c [11]...................................129
Listing 5.14: Izmenjeni/dodati delovi kôda u funkciji calcluz iz fajla imatgtime3d-hyb.c (GP-SCL-HYB) [71] u odnosu na kôd funkcije calcluz iz fajla imagtime3d-th.c [11]...................................131
Listing 5.15: Kôd nove funkcije calcpot iz fajla imatgtime3d-hyb.c (projekat GP-SCL-HYB) [71]. 132
Listing 5.16: Kôd funkcija dvar_debug, lvar_debug, tensor_debug i tensor_rod_debug................133
Listing 5.17: Testiranje pojedinih delova kôda funkcije calcluy (delovi kôda sa testnim funkcijama su zadebljani)...135
Listing 6.1: Izgled jednog fajla koji omogućava izvršavanje simulacije kroz PBS sistem.............138