



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У
НОВОМ САДУ



Ђорђе Ковачевић

**Једно решење аутоматског паралелизатора
асемблерског кода са паралелним петљама**

МАСТЕР РАД

Нови Сад, 2012



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 документација
Тип записа, ТЗ:	Текстуални штампани материјал
Врста рада, ВР:	Дипломски – мастер рад
Аутор, АУ:	Ђорђе Ковачевић
Ментор, МН:	Проф. Др. Мирослав Поповић
Наслов рада, НР:	Једно решење аутоматског паралелизатора асемблерског кода са паралелним петљама
Језик публикације, ЈП:	Српски / латиница
Језик извода, ЈИ:	Српски
Земља публиковања, ЗП:	Република Србија
Уже географско подручје, УГП:	Војводина
Година, ГО:	2012
Издавач, ИЗ:	Ауторски репринт
Место и адреса, МА:	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, ФО: (поглавља/страна/ цитата/табела/слика/графика/прилога)	7/30/0/4/9/2/0
Научна област, НО:	Електротехника и рачунарство
Научна дисциплина, НД:	Рачунарска техника
Предметна одредница/Кључне речи, ПО:	MIPS, програмски преводиоци, асемблер, анализатор синтаксе, SSA форма, граф тока података, METIS, линеарна додела регистара
УДК	
Чува се, ЧУ:	У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, ВН:	
Извод, ИЗ:	У раду је развијено једно ново решење програмског алата за паралелизацију секвенцијалног програмског кода и програмског кода са паралелним петљама. Циљ рада је развити паралелизатор који на улазу чита секвенцијални асемблерски код, а на излазу испоручује паралелизован код за МИПС процесор са више језгара. Идеја је следећа: анализатор синтаксе преводи улазну асемблерску датотеку у програмске објекте погодне за даљу обраду. Потом се изводи једнострука додела ресурса. На основу графа тока података, METIS алгоритам раздваја инструкције на различита језгра. После METIS алгоритма се додељују регистри алгоритмом линеарне доделе да би на крају програма добили расподељен асемблерски код на сваком језгру.
Датум прихватања теме, ДП:	
Датум одбране, ДО:	
Чланови комисије, КО:	Председник: Проф. Др. Илија Башичевић
	Члан: Проф. Др. Небојша Пјевалица
	Члан, ментор: Проф. Др. Мирослав Поповић
	Потпис ментора



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 :	Master Thesis
Author, AU :	Đorđe Kovačević
Mentor, MN :	PhD Miroslav Popović
Title, TI :	One solution automated parallelizator of assembly code with parallel loops
Language of text, LT :	Serbian
Language of abstract, LA :	Serbian
Country of publication, CP :	Republic of Serbia
Locality of publication, LP :	Vojvodina
Publication year, PY :	2012
Publisher, PB :	Author's reprint
Publication place, PP :	Novi Sad, Dositeja Obradovica sq. 6
Physical description, PD : (chapters/pages/ref./tables/pictures/graphs/appendixes)	7/30/0/4/9/2/0
Scientific field, SF :	Electrical Engineering
Scientific discipline, SD :	Computer Engineering, Engineering of Computer Based Systems
Subject/Key words, S/KW :	MIPS, compiler, assembler, syntax analyzer, SSA form, data flow graph, METIS, linear scan register allocation
UC	
Holding data, HD :	The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, N :	
Abstract, AB :	This paper describes one new software solution tool for parallelization of sequential program code and program code with parallel loops. The main goal is to make paralelizator which at the input of the system reads sequential assembler code and at the output provides parallelized code for MIPS processor with multiple cores. The idea is in the following: the parser translates assembler input file to program objects suitable for further processing. After that the static single assignment is done. Based on the data flow graph, the parallelization algorithm separates instructions on different cores. After the parallelization algorithm, registers are allocated with algorithm of linear allocation, and the result at the end of the program is distributed assembler code on each of the cores.
Accepted by the Scientific Board on, ASB :	
Defended on, DE :	
Defended Board, DB :	President: PhD Ilija Bašičević
	Member: PhD Nebojša Pjevalica
	Member, Mentor: PhD Miroslav Popović
	Menthor's sign



УНИВЕРЗИТЕТ У НОВОМ САДУ • ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

21000 НОВИ САД, Трг Доситеја Обрадовића 6

ЗАДАТАК ЗА МАСТЕР РАД

Број:

Датум:

(Податке уноси предметни наставник - ментор)

СТУДИЈСКИ ПРОГРАМ:	РАЧУНАРСТВО И АУТОМАТИКА
РУКОВОДИЛАЦ СТУДИЈСКОГ	Проф. Др. Иван Луковић

Студент:	Ђорђе Ковачевић	Број	Е2 43/2011
Област:	Системска програмска подршка		
Ментор:	Проф. Др. Мирослав Поповић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА
ИЗДАЈЕ СЕ ЗАДАТАК ЗА МАСТЕР РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ МАСТЕР РАДА:

Једно решење аутоматског паралелизатора асемблерског кода са паралелним петљама

ТЕКСТ ЗАДАТКА:

У раду је потребно реализовати аутоматски паралелизатор асемблерског кода за процесор МИПС са више језгара. Овај паралелизатор треба да буде способан да паралелизује секвенцијални програмски код и програмски код са паралелним петљама. Тестирање решења треба обавити на датим референтним примерима програма за обично множење матрица и Страсенов метод множења матрица.

Руководилац студијског програма:	Ментор рада:

SADRŽAJ

1. Uvod.....	1
2. Teorijske osnove rada	3
2.1 Višefazni algoritam za podelu grafova.....	4
2.1.1 Faza ukрупnjavanja grafa (klasterizacija)	4
2.1.2 Faza inicijalne podele grafa	5
2.1.3 Faza finije preraspodele	6
2.2 METIS	7
2.2.1 METIS samostalni (eng. <i>standalone</i>) programi.....	7
2.2.2 METIS API.....	8
2.3 Algoritam paralelizacije	10
3. Analiza problema	12
4. Realizacija.....	15
4.1 Prednji deo paralelizatora.....	15
4.1.1 Analizator sintakse asemblerskog koda.....	15
4.1.2 SSA međukod	16
4.1.3 Makro instrukcije.....	17
4.2 Srednji deo paralelizatora.....	18
4.2.1 Graf toka podataka.....	18
4.2.2 Analiza pristupa memoriji	18
4.2.3 METIS biblioteka	19
4.2.4 Algoritam paralelizacije.....	20
4.3 Zadnji deo paralelizatora.....	21

4.3.1	Dodela registara	21
4.3.2	Emiter koda.....	22
4.4	Paralelizacija paralelnih petlji	23
5.	Ispitivanje.....	25
5.1	Ispitivanje paralelizacije sekvencijalnog koda.....	25
5.2	Ispitivanje paralelizacije paralelnih petlji	27
6.	Zaključak	29
7.	Literatura.....	31

SPISAK SLIKA

Slika 2.1 Faza ukрупnjavanja grafa	5
Slika 2.2 Višefazna podela grafa na dve particije	6
Slika 2.3 Formati za različite tipove grafova	8
Slika 2.4 Primer jednostavnog grafa i dobijenih nizova <i>xadj</i> i <i>adjncy</i> iz grafa	9
Slika 3.1 UML dijagram aktivnosti paralelizatora	12
Slika 3.2 UML dijagram klasa paralelizatora	14
Slika 4.1 Struktura podataka analizatora sintakse asemblerskog koda	16
Slika 4.2 Pregled klasa za evidenciju podataka o pokazivačima na memoriju	19
Slika 4.3 Sečenje grafa na dve particije	20
Slika 5.1 Rezultati ubrzanja koda na primeru množenja matrica u zavisnosti od broja jezgara	26
Slika 5.2 Rezultati ubrzanja koda na primeru Strassenovog metoda množenja matrica u zavisnosti od broja jezgara	27
Slika 5.3 Rezultati ubrzanja koda na primeru <i>for</i> petlje u zavisnosti od broja jezgara	28

SPISAK TABELA

Tabela 4.1 Primer prebacivanja u SSA	17
Tabela 4.2 Tok dodeljivanja registara, od izvornog programa do registara hipotetičkog procesora.....	22
Tabela 5.1 Rezultati eksperimentalnog merenja ubrzanja koda na primeru množenja matrica	25
Tabela 5.2 Rezultati eksperimentalnog merenja ubrzanja koda na primeru Strassenovog metoda množenja matrica.....	26
Tabela 5.3 Rezultati eksperimentalnog merenja ubrzanja koda na primeru <i>for</i> petlje.....	27

SKRAĆENICE

OpenMP	- <i>Open Multiprocessing</i> , Otvoreni standard za višeprosorski rad
OpenCL	- <i>Open Computing Language</i> , Otvoreni standard za paralelno programiranje heterogenih računarskih sistema
RISC	- <i>Reduced instruction set computing</i> , Procesor sa smanjenim skupom naredbi
CISC	- <i>Complex instruction set computing</i> , Procesor sa složenim skupom naredbi
SSA	- <i>Static Single Assignment</i> , Jednostruka dodela resursa
VLSI	- <i>Very Large Scale Integration</i> , Integrisana kola velike gustine
KL	- <i>Kernighan-Lin algorithm</i>
API	- <i>Application Programming Interface</i> , Aplikativna programska sprega
MIPS	- <i>Microprocessor without Interlocked Pipeline Stages</i> , Procesor bez zaključavanja protočnih faza
UML	- <i>Unified modelling language</i> , Objedinjeni jezik za modelovanje
GCC	- <i>GNU Compiler Collection</i> , GNU kompajler

1. Uvod

Višeprocesorski sistemi, procesori sa masivnim paralelizmom i grafičke procesne jedinice sa paralelnom strukturom omogućavaju paralelno izvršavanje programa. Pojava ovakvih sistema pružila je nove mogućnosti za povećanje brzine izvršavanja programa, a samim tim pojavljuje se potreba za automatskom paralelizacijom programskog koda. Paralelizacije postojećih sekvencijalnih programa i generisanje novih, paralelnih, postale su značajne oblasti istraživanja. Cilj ovog rada je razviti paralelizator asemblerskog koda i videti kako paralelizacija doprinosi brzini izvršavanja samog programa. Paralelizator treba da bude sposoban da paralelizuje sekvencijalni asemblerski kod i asemblerski kod sa paralelnim petljama.

Sekvencijalni programski kod može se paralelizovati ručno, poluautomatski i automatski. Ručna paralelizacija je komplikovan i iscrpljujući posao za programera, koji zahteva mnogo promena u kodu. Često je to vremenski zahtevan i složen iterativni proces, podložan greškama. Programer je odgovoran i za prepoznavanje i za implementaciju paralelizma u nekom programu. Postiže se dobro ubrzanje koda. Poluautomatska paralelizacija je manje komplikovana, zahteva manje promena u kodu i postiže se manje ubrzanje koda u odnosu na ručnu paralelizaciju. Pogodna je za paralelizaciju sekvencijalnog koda. Postoji više različitih pristupa u paralelizaciji sekvencijalnog koda, ali najistaknutiji su Intel Cilk Plus, OpenMP, OpenCL i drugi, koji predstavljaju programske biblioteke namenjene nadogradnji programskih jezika sa ciljem da programerima omoguće razvoj aplikacija koje se izvršavaju istovremeno na više procesora. Kod automatske paralelizacije, programski prevodilac sam analizira izvorni kod i pronalazi delove pogodne za paralelizaciju. Petlje su najpogodnije za paralelizaciju. Automatska paralelizacija ne uvodi nikakve promene u originalni kod.

U ovom radu sekvencijalni asemblerski kod se paralelizuje automatski. Paralelizator je ispitan na pojedinim programima, pri čemu je posmatrana promena ubrzanja koda sa povećanjem broja jezgara. Postignuto je značajno ubrzanje za pojedine programe.

Informacione tehnologije otišle su korak dalje, odvajajući se od fizičke podrške u više slojeve apstrakcije računarskih sistema. Tu se prvenstveno misli na jezike i tehnologije sa automatski kontrolisanom dodelom memorije (Microsoft C# sa .Net , Java , itd.), koje programera odvajaju od brige za fizičku arhitekturu i najveću pažnju poklanjaju samom algoritmu. Nedostatak ovakvih jezika je mnogo sporije izvršavanje programa. Razvojem pametnih telefona, elektronskih beležnica (eng. *tablet*), pametnih televizora i njihovom sve većom popularnošću, ponovo se u upotrebu vraćaju niži jezici (Assembler i C). Sve većom upotrebom C/C++ programskih jezika, čemu doprinosi i novi C++11 standard, javljaju se i potrebe za asemblerskim programskim jezikom za obrade kojima je potrebna brzina u realnom vremenu (eng. *real time*).

RISC (eng. *reduces instruction set computing*) procesori, koji se najviše koriste u telefonima i sličnim ugrađenim (eng. *embedded*) uređajima počinju da prate CISC (eng. *complex instruction set computing*) procesore. Podstaknut sve većom popularnošću RISC procesora ovaj rad, oslanjajući se na postojeće algoritme, realizuje analizador izvornog koda koji automatski prevodi zadati kod koji radi na jednom jezgri u kod koji radi na procesoru sa zadatim brojem jezgara.

Tehnike korišćene u radu su preuzete iz projektovanja programskog prevodioca. Tehnike su prilagođene konkretnom problemu i to su: jednostruka dodela resursa [1] (eng. *static single assignment* u daljem tekstu SSA međukod) koja predstavlja međureprezentaciju podataka prilikom prevođenja programa, graf toka podataka i dodela registara algoritmom linearnog pretraživanja kao faza posle algoritma paralelizacije [2].

U odeljku 2 su date teorijske osnove neophodne pri rešavanju ovog rada. U odeljku 3 opisana je arhitektura paralelizatora i dat je dijagram aktivnosti i dijagram klasa. Paralelizator asemblerskog koda podeljen je na tri dela: prednji, srednji i zadnji deo, koji su opisani u odeljku 4. Prednji deo paralelizatora predstavlja analizador sintakse asemblerskog koda (eng. *parser*) sa transformacijom u SSA međukod i makro instrukcije. Srednji deo paralelizatora sadrži graf toka podataka, analizu pristupa memoriji (eng. *alias analysis*), METIS biblioteku i algoritam paralelizacije. Zadnji deo paralelizatora se sastoji od faze dodele registara nad obrađenim asemblerskim kodom i emitira koda. U ovom odeljku je objašnjena i paralelizacija paralelnih petlji. Odeljak 5 se odnosi na ispitivanje programa, dok je u poslednjem odeljku dat zaključak.

2. Teorijske osnove rada

U praksi se često susreće problem podele neke grupe objekata na više manjih delova, pri čemu je poželjno da se objekti koji su međusobno jako povezani nađu u istom modulu, nasuprot objektima koji nisu povezani i koji mogu da se nađu u različitim modulima. Time se dolazi do matematičkog problema particionisanja skupa nekakvih elemenata, tj. grupe objekata čija se povezanost može predstaviti koristeći strukture kao što su grafovi. Objekti koje je potrebno podeliti u module su u grafovima predstavljeni čvorovima, kojima može biti dodeljena neka vrednost, tj. težina čvora. Postojanje veza između objekata predstavljeno je granom u grafu.

Algoritmi za podelu grafova imaju primenu u brojnim sistemima gde je u cilju optimizacije rada potrebno izvršiti podelu na podsisteme [9]. Primeri ovakve primene su:

- VLSI (eng. *Very-Large-Scale Integration*) tehnologije – neophodno je izvršiti optimalnu podelu tranzistorskih komponenti u grupe tako da povezanost između komponenti u okviru različitih grupa bude što manja
- Veliki modeli podataka – zahtevaju skupe računare za obradu podataka, pa se distribucijom podataka i obrade na više jeftinih računara značajno smanjuje cena fizičke arhitekture
- Paralelni proračuni nad grupama podataka

U svakom od pomenutih sistema postoje međusobno spregnuti elementi koje je potrebno razvrstati u grupe tako da u svakoj grupi budu elementi koji su povezani čvrstim vezama. Problem podele je trivijalan kada postoje disjunktni podskupovi elemenata, ali praktični problemi su složeniji, jer se često nameće ograničenje na veličinu grupe i tada se kod svake moguće raspodele elemenata pojavljuju veze između grupa. Time se postupak podele postavlja kao optimizacioni problem gde je potrebno naći odgovarajuću podelu tako da veze među grupama budu minimalne i ujedno da su sve grupe približno jednake veličine. Postupkom minimizacije veza između grupa dobijaju se grupe čvrsto povezanih elemenata.

Problemi podele grafova se mogu odnositi na različite tipove grafova, pa se u skladu sa tim vrši izbor algoritma za podelu. U okviru ovog rada se razmatraju težinski grafovi, gde se svakom čvoru i grani u grafu dodeljuje određena težina. Algoritam koji je korišćen za podelu grafova je višefazni algoritam. Konkretno za ovaj rad, svrha algoritma je da razdvoji instrukcije asemblerskog koda na različita jezgra procesora.

2.1 Višefazni algoritam za podelu grafova

Višefazne algoritme za podelu grafova su razvili istraživači *Barnard* i *Simon* 1993. godine [4]. Narednih nekoliko godina istraživači su se bavili istraživanjima višefaznih algoritama, a najznačajnije rezultate je postigao *Karypis* (razvio je najpoznatiji i jedan od najčešće korišćenih – METIS algoritam [3]). Karakteristika ovih algoritama je velika brzina i mogućnost primene na grafove velikih dimenzija.

U ovom radu je korišćen METIS programski paket za podelu velikih neusmerenih grafova. Algoritmi realizovani u okviru METIS-a su zasnovani na višefaznim algoritmima za podelu grafova. Višefazna podela grafa se sastoji iz tri faze (slika 2.2):

1. ukрупnjavanje grafa (eng. *graph coarsening*)
2. inicijalne podele (eng. *initial partitioning*)
3. finije preraspodele (eng. *uncoarsening*)

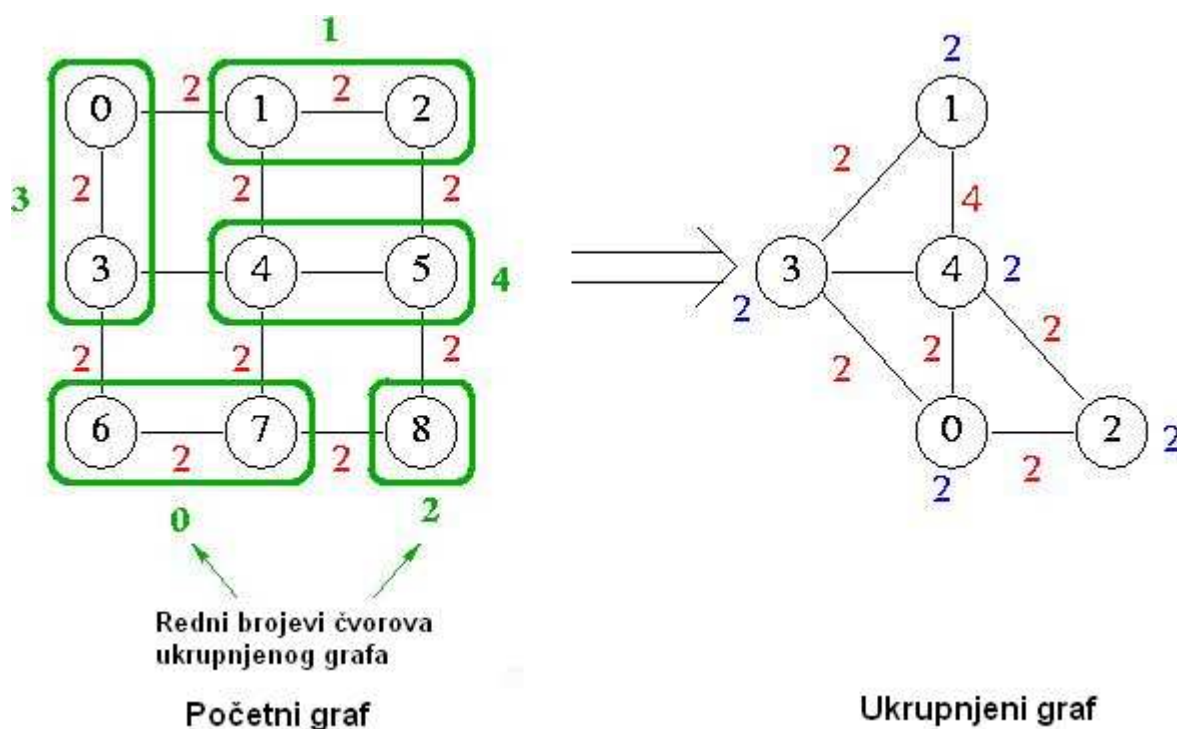
Ideja algoritma je sledeća: početni graf G se aproksimira u više nivoa, čime se dobijaju sve manji i manji grafovi, a samim tim smanjuje se prostor pretraživanja i povećava efikasnost algoritma podele. Inicijalna podela podrazumeva podelu grafa dobijenog ukрупnjavanjem grafa na najvišem nivou. Tokom finije preraspodele, dobijena particija sa najvišeg nivoa se projektuje na originalni graf opet prolazeći kroz različite nivoe aproksimacije. U nastavku su detaljnije objašnjene sve tri faze.

2.1.1 Faza ukрупnjavanja grafa (klasterizacija)

Početni graf $G = (V, E)$ koji se sastoji iz skupa čvorova $V = \{v_1, v_2, \dots, v_c\}$ i grana $E = \{e_1, e_2, \dots, e_g\}$, se transformiše u manje grafove $G_1, G_2, \dots, G_m$ za koje važi da je $|V| > |V_1| > |V_2| > \dots > |V_m|$ [4], gde $|V_i|$ predstavlja ukupan broj čvorova grafa G_i .

Transformacija grafa G_i se vrši tako što se pojedini čvorovi grafa spajaju u jedan čvor v_x . Taj čvor v_x predstavlja čvor grafa G_{i+1} (nižeg hijerarhijskog nivoa), a grane koje su sjedinjeni čvorovi gradili sa ostalim (sa njima nesjedinjenim u v_x) čvorovima se tretiraju kao grane čvora v_x sa ostalim čvorovima. Postoji više različitih algoritama za ukрупnjavanje čvorova, kao što su: slučajno sjedinjavanje, sjedinjavanje po najtežim granama, sjedinjavanje po najlakšim granama, itd. Na jednom nivou se primenjuje ukрупnjavanje 2 ili k čvorova na sve neuparene čvorove na

datom nivou (koje je moguće upariti). Broj nivoa zavisi od veličine početnog grafa. Na slici 2.1 predstavljena je faza ukрупnjavanja grafa.



Slika 2.1 Faza ukрупnjavanja grafa

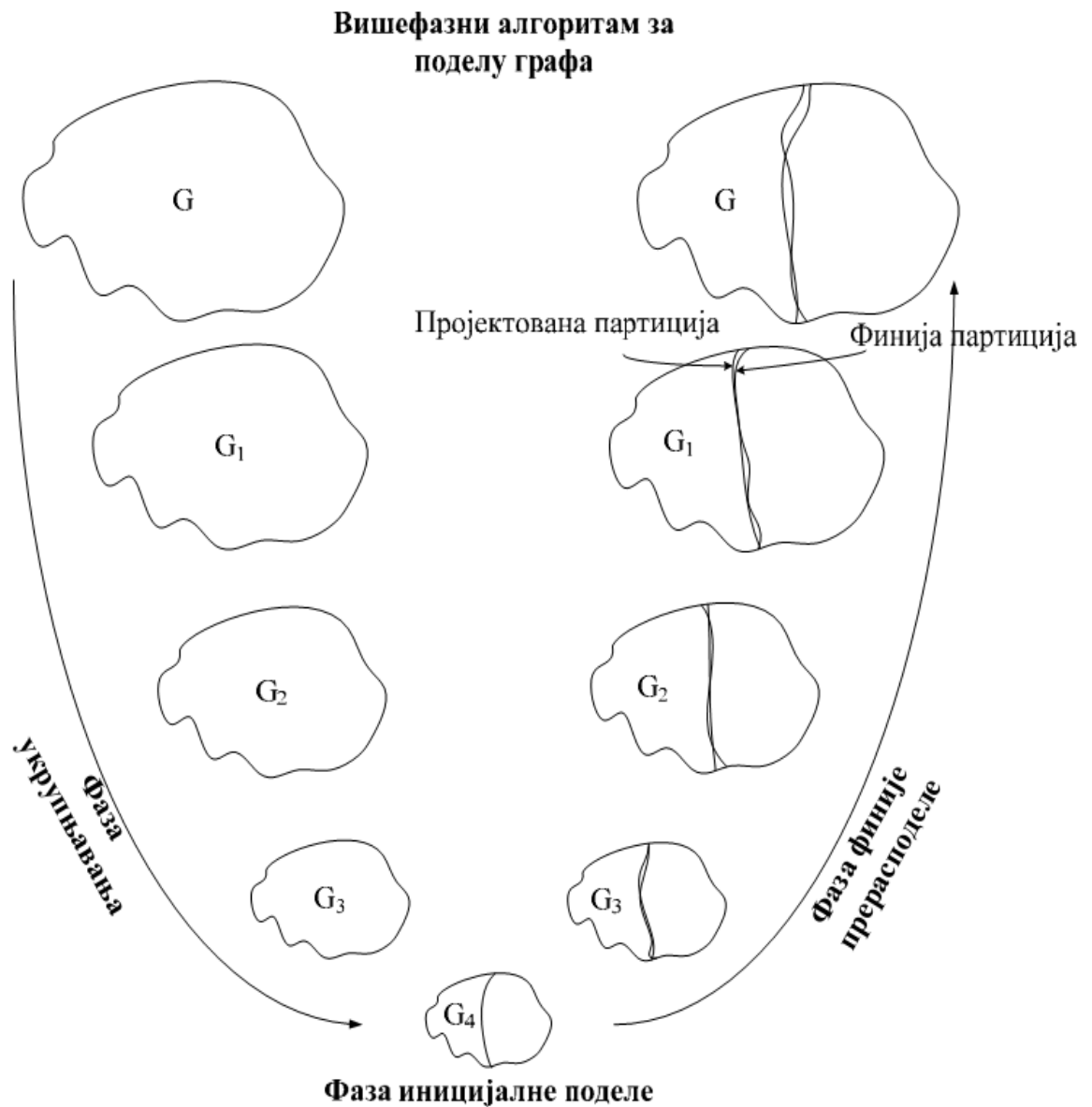
2.1.2 Faza inicijalne podele grafa

U ovoj fazi algoritma cilj je podeliti poslednji ukрупnjeni graf na p particija, ali tako da particija zadovoljava uslove uravnoteženja, tj. da veličina particije ne prelazi unapred zadatu dozvoljenu veličinu i da optimizuje funkciju cilja. Funkcija cilja predstavlja kvalitet particije grafa u smislu da su čvorovi unutar istih delova particije jako povezani, a čvorovi različitih delova slabo povezani, jer se posmatra situacija kada je skupo održavati vezu između čvorova različitih delova particije. Dobijena inicijalna particija se zatim u fazi finije preraspodele projektuje sa najmanjeg grafa na sve veće grafove, sve dok se ne stigne do početnog grafa.

Inicijalno particionisanje može biti dobijeno sa nekim od sledećih algoritama:

1. KL algoritam [5] – na slučajan način se izvrši podela grafa na particije, pa se zatim prebacuju čvorovi iz jedne u drugu particiju.
2. Algoritam narastanja čvorova (eng. *graph growing partitioning algorithm*) – proces formiranja particija polazi od slučajno izabranog čvora, pa se nakon toga skupljaju svi čvorovi oko njega, sve dok se ne dostigne željena veličina particije i tako za sve particije.

Na slici 2.2 je prikazan višefazni algoritam za podelu grafa na dve particije.



Slika 2.2 Višefazna podela grafa na dve particije

2.1.3 Faza finije preraspodele

Nakon izvršene inicijalne podele grafa G_m pristupa se poboljšavanju podele tako što se podela grafa G_m prevodi u podelu grafa iz višeg nivoa G_{m-1} . Nakon toga se pokušava poboljšati podela prebacivanjem čvorova iz skupa V_{m-1} (čvorova grafa sa višeg nivoa) i na taj način napraviti finiju preraspodelu grafa. Postupak se ponavlja (poboljšanjem podele na sledećem višem nivou grafa) sve dok ima poboljšanja podele ili dok se ne dođe do najvišeg nivoa. Za poboljšanje podele, tj. prebacivanje čvorova između particija koristi se KL algoritam.

Pseudokod koji se koristi u programskom paketu METIS za particionisanje grafa, dat je u daljem tekstu:

nivo = 0;

```

KlasterizovanGraf [nivo] = graf;
//klasterizacija
while (KlasterizovanGraf [nivo].BrojCvorova() > 200)
    SledeciGraf = KlastriZuj (KlasterizovanGraf[nivo]);
    Nivo = nivo + 1;
    KlasterizovanGraf [nivo] = SledeciGraf;
//inicijalno particionisanje
Particija [nivo] = slucajna inicijalna particija;
KL (KlasterizovanGraf [nivo], Particija [nivo]);
//finija preraspodela
while (nivo > 0)
    nivo = nivo - 1;
    Particija [nivo] = Projektuj (Particija [nivo+1], KlasterizovanGraf [nivo]);
    KL (KlasterizovanGraf [nivo], Particija [nivo]);
return Particija [0];

```

2.2 METIS

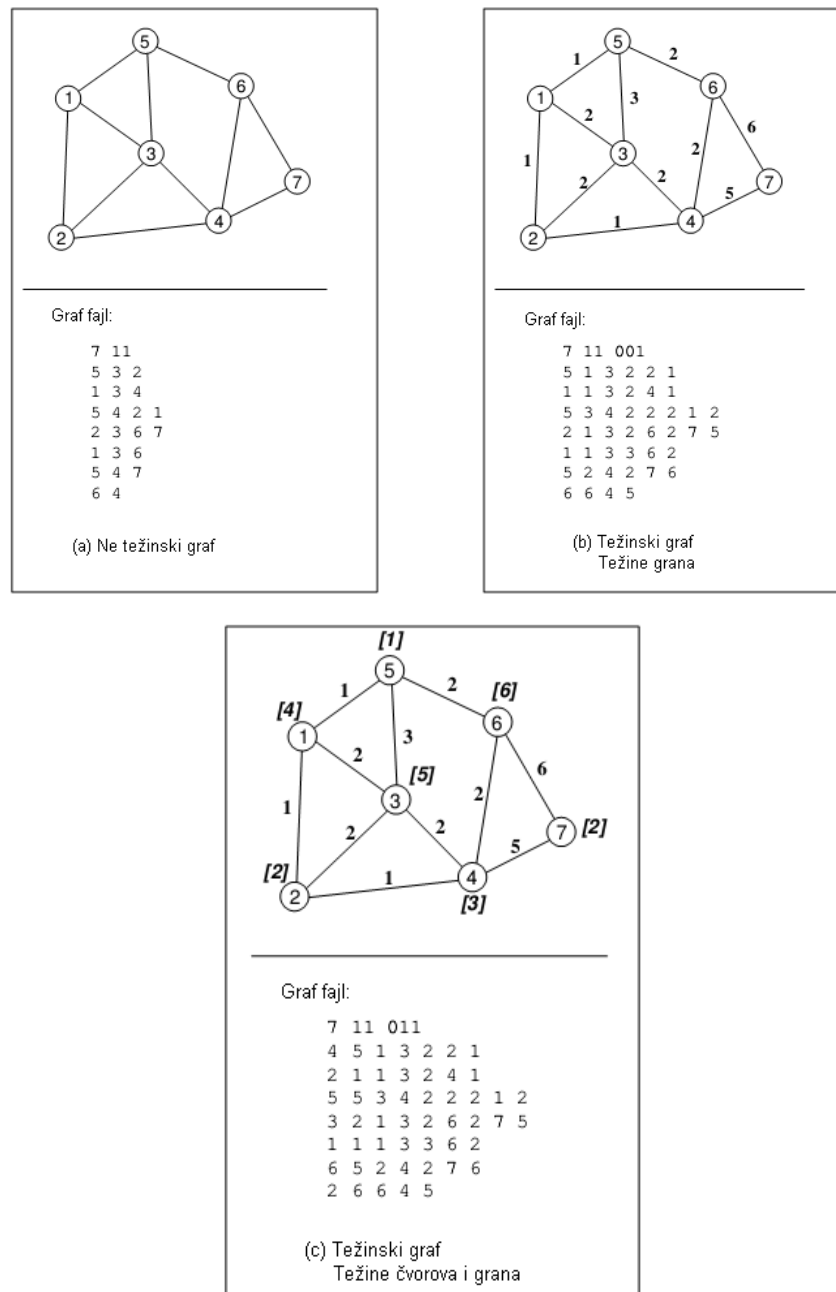
2.2.1 METIS samostalni (eng. *standalone*) programi

METIS sadrži razne programe koji se koriste za podelu grafova, podelu mreža, kao i programe za transformaciju mreža u grafove [6].

Razni programi u METIS-u zahtevaju kao ulaz datoteku u koju je smešten graf ili datoteku u koju je smeštena mreža. Graf $G = (V, E)$ sa n čvorova i m grana je smešten u tekstualnu datoteku koja sadrži $n+1$ liniju. Prva linija, predstavljena kao linija zaglavlja, sadrži podatke o veličini i tipu grafa, dok preostalih n linija sadrže informaciju o svakom čvoru grafa G .

U ovom radu linija zaglavlja sadrži tri parametra (n, m, fmt). Parametar n predstavlja broj čvorova, a parametar m broj grana. Parametar fmt sadrži informaciju o veličini čvorova, težini čvorova i težini grana.

Na slici 2.3 predstavljeni su različiti formati za različite tipove grafova. Prva linija sa slike 2.3 (c) označava da postoji 7 čvorova, 11 grana i da graf sadrži podatke o težini čvorova i grana. U drugoj liniji sa slike 2.3 (c) navedeni brojevi predstavljaju sledeće: prvi broj predstavlja težinu čvora, dok ostali brojevi određuju sa kojim čvorovima je u vezi prvi broj i kolika je težina između tih čvorova. Vidi se da je čvor 1 u vezi sa čvorovima 5, 3 i 2, a težine veza između čvora 1 i čvorova 5, 3 i 2 su redom 1, 2 i 1.



Slika 2.3 Formati za različite tipove grafova

Izlaz iz METIS-a je datoteka u kojoj je graf podeljen na particije. Graf sa n čvorova sadrži n linija, gde svaka linija sadrži jedan broj. Taj broj predstavlja broj particije na kojoj je završio čvor. Brojevi particija se kreću od nule do ukupnog broja particija minus jedan.

METIS samostalni program za podelu grafa se naziva *gpmets*. Funkcionalnost koju obezbeđuje *gpmets* postiže se sa API (eng. *Application Programming Interface*) rutinama: METIS *PartGraphRecursive* i METIS *PartGraphKway*.

2.2.2 METIS API

Različitim rutinama realizovanim u METIS samostalnim programima može se nedosredno pristupiti iz C, C++ programa koristeći METIS biblioteke. METIS API opisuje različite

konvencije pozivanja i upotrebe, različite strukture podataka koje se koriste za ubacivanje informacija u i vađenje informacija iz rutina.

Svaki program koji koristi METIS API mora da uključi *metis.h* datoteku zaglavlje. Datoteka sadrži prototipove funkcija za različite API rutine i definicije različitih tipova podataka i konstanti korišćenih od strane rutina. Biblioteka *metis.h* definiše dva tipa podataka i njihove širine, kao i razne *enum* tipove podataka. Dva tipa podataka su: *idx_t* - celi tip podataka i *real_t* - realni tip podataka. *idx_t* je definisan kao 32 ili 64 označeni celi broj, dok je *real_t* definisan u jednostrukoj ili dvostrukoj preciznosti.

Većina API rutina uzima kao parameter niz nazvan *options*, koji dozvoljava programu da modifikuje različite aspekte algoritama u okviru METIS-a. Program mora definisati niz kao:

```
idx_t options [METIS_NOPTIONS];
```

Sve API rutine u METIS-u uzimaju kao ulaz listu povezanosti grafa i težine čvorova i grana, ako postoje. Lista povezanosti grafa od n čvorova i m grana se predstavlja preko dva niza *xadj* i *adjncy*. Niz *xadj* je veličine $n+1$, a niz *adjncy* je veličine $2m$.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

xadj	0	2	5	8	11	13	16	20	24	28	31	33	36	39	42	44																												
adjncy	1	5	0	2	6	1	3	7	2	4	8	3	9	0	6	10	1	5	7	11	2	6	8	12	3	7	9	13	4	8	14	5	11	6	10	12	7	11	13	8	12	14	9	13

Slika 2.4 Primer jednostavnog grafa i dobijenih nizova *xadj* i *adjncy* iz grafa

Sa slike 2.4 niz *adjncy* se dobija tako što se posmatra sa kim je u vezi čvor 0, sa kim je u vezi čvor 1, itd. Vidi se da je čvor 0 u vezi sa čvorovima 1 i 5, čvor 1 je u vezi sa čvorovima 0, 2, 6. Čvorovi 1, 5, 0, 2, 6 se smeštaju u niz *adjncy*. Niz *xadj* se formira tako što se broji sa koliko čvorova je svaki pojedinačni čvor u vezi. Čvor 0 je u vezi sa dva čvora, a čvor 1 je u vezi sa 3 čvora. Svaki tekući elemenat niza se dobija sabiranjem prethodnog elementa niza sa brojem čvorova sa kojim je sledeći čvor u vezi.

API za podelu grafova se koristi da podeli graf na k delova koristeći višefazni algoritam za podelu grafova na k particija. Prototip funkcije kojom se poziva algoritam je:

```
int METIS_PartGraphKway (idx_t *nvtxs, idx_t *ncon, idx_t *xadj, idx_t *adjncy, idx_t *vwgt, idx_t *vsize, idx_t *adjwgt, idx_t *nparts, real_t *tpwgts, real_t ubvec, idx_t *options, idx_t *objval, idx_t *part)
```

U sledećem tekstu navedena su značenja samo najvažnijih parametara:

nvtxs – broj čvorova u grafu

xadj, *adjncy* – prethodno opisani nizovi

vwgt – težina čvorova

adjwgt – težina grana

nparts – broj particija na koje se deli graf

2.3 Algoritam paralelizacije

Nakon što su instrukcije METIS algoritmom raspoređene po jezgrima, vrše se razne optimizacije u algoritmu paralelizacije.

U tekstu ispod dat je pseudokod algoritma paralelizacije.

```

for each instructionList IL from B do
  while (IL  $\neq \emptyset$ )
    W = doUrgencyRule (IL)
    while (W  $\neq \emptyset$ )
      for each instruction I from W do
        doSyncRule (I)
        doFillRule (W)
        add instruction I from W to core
      add I to core
    make all coreblocks equal lengths
    add all instruction from epilog to end of coreblocks

```

Algoritam paralelizacije se sastoji od nekoliko faza.

Faza *doUrgencyRule* – instrukcije iz bloka *B* se ne dodaju redom na jezgra (kao što su dodavane u listu instrukcija *IL* bloka *B*), nego se proverava da li postoje neke instrukcije u bloku za koje su definisani svi izvorišni registri. Ako postoje, te instrukcije se smeštaju u listu spremnih instrukcija *W*. Izuzetak su ulazni registri, koji su već definisani u nekom drugom bloku, pa se instrukcije koje sadrže te registre takođe smeštaju u *W*. Ova faza je ilustrovana sledećim asemblerskim kodom:

```

lw      $3,12($4)
lw      $2,12($5)
lw      $8,0($4)
addu    $7,$3,$8
addi    $11,$7,2
subu    $10,$8,$2

```

Ulazni registri su registri *\$4* i *\$5*, pa se instrukcije koje sadrže ta dva registra smeštaju u *W*. Instrukcije *addu*, *addi* i *subu* se ne smeštaju u *W*, jer za njih još nisu definisani izvorišni registri. Kada se instrukcije *lw* smeste u *W*, tek tada se definišu izvorišni registri *\$3*, *\$2*, *\$8* preostalih instrukcija. Zatim se prolazi kroz ostale faze algoritma, kako bi se ponovo došlo do faze

doUrgencyRule. Iz *W* se brišu instrukcije koje su se nalazile u njoj i dodaju se nove instrukcije *addu* i *subu*, kako bi na kraju u *W* ostala samo instrukcija *addi*.

Faza *doSyncRule* – ulaz u ovu fazu su instrukcije iz *W*. Uzima se jedna po jedna instrukcija iz *W*. Ako postoji neki registar u instrukciji koji se koristi u jednom jezgru, a nigde nije definisan na tom jezgru, potrebno je dodati sinhronizacione instrukcije na jezgra (*lw/sw*). Instrukcija *sw* se dodaje na jezgro gde je registar definisan, a *lw* na jezgro gde se registar koristi. Instrukcija *sw* se dodaje posle definisanja registra, a *lw* pre korišćenja registra. Prilikom izvršavanja programa na više jezgara, ne sme se u jednom te istom trenutku pojaviti dve sinhronizacione instrukcije. Ako se to desi, dodaje se *nop* instrukcija na odgovarajuće jezgro. Primer ove faze je:

Pre sinhronizacije		Posle sinhronizacije	
#jezgro 0	#jezgro 1	#jezgro 0	#jezgro 1
li \$3,4	addi \$4,\$3,1	li \$3,4	nop
		sw \$3,34(\$0)	nop
			lw \$3,34(\$0)
			addi \$4,\$3,1

Navedeni primer pokazuje da instrukcija *sw* mora biti dodata pre instrukcije *lw*, jer nije moguće u isto vreme čitati i pisati sa iste memorijske lokacije, kao što nije moguće čitati sa neke memorijske lokacije u koju ništa nije upisano.

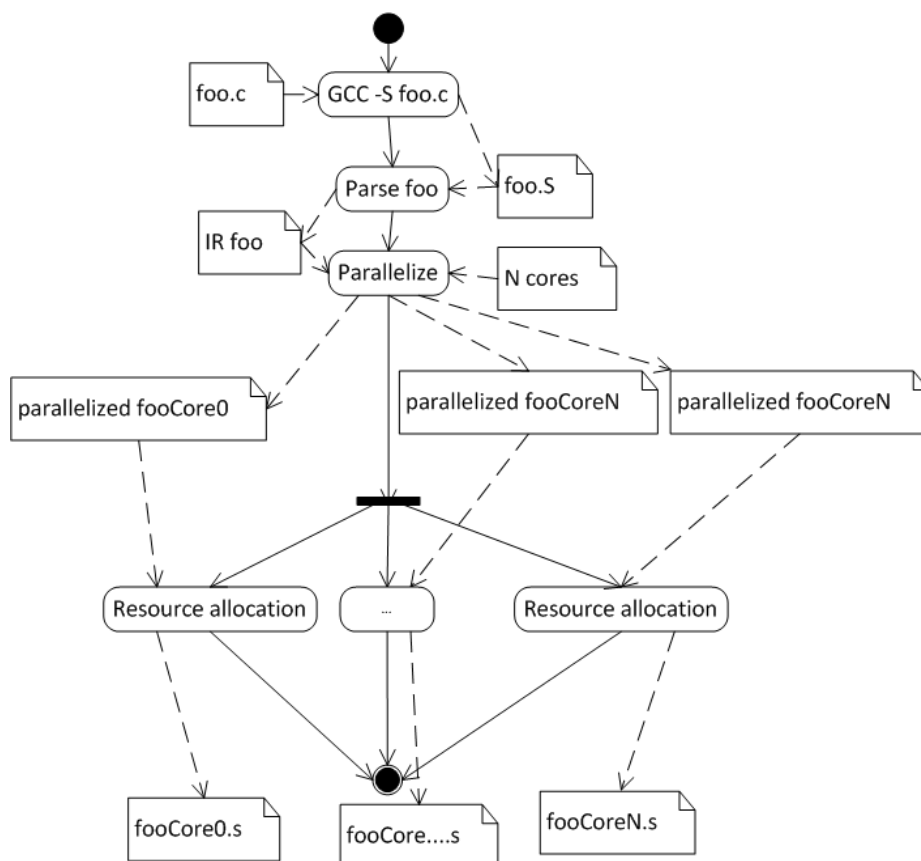
Faza *doFillRule* – ova faza zamenjuje *nop* instrukcije sa nekim drugim instrukcijama, tamo gde je moguće. Spremne instrukcije *W* ne moraju se redom, onako kako su dodavane u listu spremnih instrukcija, smeštati na jezgra. Tako se može smanjiti broj instrukcija, što pokazuje sledeći primer:

Bez <i>doFillRule</i> faze		Sa <i>doFillRule</i> fazom	
# jezgro 0	# jezgro 1	#jezgro 0	#jezgro 1
lw \$2,12(\$5)	lw \$7,4(\$4)	lw \$2,12(\$5)	lw \$7,4(\$4)
lw \$10,0(\$5)	nop	lw \$10,0(\$5)	addi \$9,\$7,2
addu \$11,\$2,\$10	nop	addu \$11,\$2,\$10	nop
sw \$10,34(\$0)	lw \$10,34(\$0)	sw \$10,34(\$0)	lw \$10,34(\$0)
	addi \$8,\$10,1		addi \$8,\$10,1
	addi \$9,\$7,2		

Iz primera se vidi da se broj instrukcija na prvom jezgru smanjio za jednu instrukciju. Instrukcija *nop* je zamenjena sa *addi \$9,\$7,2* instrukcijom.

3. Analiza problema

Svrha paralelizatora je da na ulazu čita sekvencijalni asemblerski kod, kod koji je samo za jedno jezgro, a na izlazu isporučuje paralelizovan kod za MIPS procesor sa više jezgara. Na slici 3.1 je prikazan dijagram aktivnosti paralelizatora, koji ilustruje sledeći redosled aktivnosti:



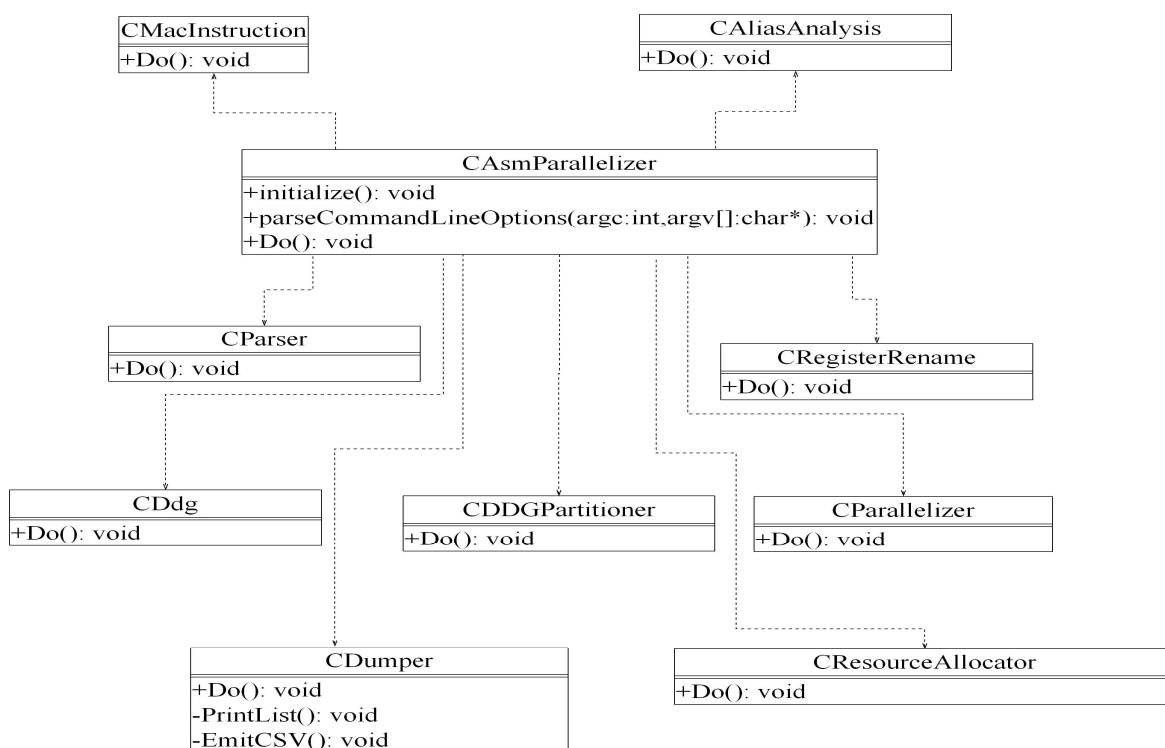
Slika 3.1 UML dijagram aktivnosti paralelizatora

- Asemblerski kod je dobijen propuštanjem C programskog koda kroz GCC prevodilac. Uključivanjem opcije -S u GCC prevodiocu na izlazu se umesto mašinskog koda dobija asemblerski kod.
- Analizator sintakse prevodi ulaznu asemblersku datoteku u programske objekte pogodne za dalju obradu. Objekti oponašaju uobičajeni međukod programskog prevodioca. Objekat u sebi sadrži pokazivače na registre sa kojima je u vezi. Registri su predstavljeni kao promenljive zbog prilagođenja SSA međukodu.
- Za razliku od originalnog izvornog koda gde se obično svaka promenljiva ponovo definiše više puta, u SSA međukodu svaka promenljiva se definiše tačno jednom, a može se koristiti više puta. Realizacijom SSA međukoda dobija se lista upotreba i definicija promenljivih, koja se koristi za dobijanje međukoda sa kojim je kasnije lakše rukovati.
- Svrha uvođenja makro instrukcije je da METIS algoritam makro instrukciju, odnosno skup instrukcija u okviru makro instrukcije, raspodeli na isto jezgro. Da nema makro instrukcije, METIS algoritam bi instrukcije koje pripadaju makro instrukciji, raspodelio na različita jezgra, pa bi se moralo uvoditi mnogo sinhronizacionih instrukcija. Uvođenjem sinhronizacionih instrukcija porastao bi broj instrukcija na jezgrima, pa bi umesto ubrzanja koda došlo do usporenja koda.
- Graf toka podataka se izvodi nad svim instrukcijama asemblerskog koda, gde svaka instrukcija ima listu instrukcija od kojih zavisi.
- Zadatak analize pristupa memoriji je da pronade sve pristupe memoriji zadatog programskog koda. Pod pristupima memoriji podrazumevamo čitanje i pisanje u memoriju (*load/store* operacije). Analiza pristupa memoriji je potrebna da bi se odredilo da li dva ili više pokazivača (u ovom radu misli se na registre) pokazuju na istu ili različitu memorijsku lokaciju.
- Na osnovu grafa toka podataka, METIS algoritam razdvaja instrukcije na različita jezgra. Nakon algoritma paralelizacije, može se desiti da se životni vek neke promenljive prekine, odnosno promenljiva nastavlja da živi na nekom drugom jezgru. U tom slučaju potrebno je dodati sinhronizacione instrukcije (*lw/sw* instrukcije) na jezgra.
- Klasični prevodioci koriste algoritam dodele registara pomoću matematičke teorije bojenja grafova. Najveća mana ove metode je da je potrebno mnogo procesorske moći da bi se izvršila. Zbog svoje brzine, koja je do 70 puta veća od bojenja grafova [2], koristi se linearna dodela registara. Kada se izvodi linearna dodela registara posle SSA međukoda, potrebno je odraditi analizu životnog veka. Ova analiza se postiže

samo jednim prolazom kroz listu instrukcija, pri čemu je dovoljna informacija o položaju promenljive u SSA međukodu.

- Na izlazu iz paralelizatora se dobiju datoteke spremne za izvršavanje. Broj datoteka zavisi od broja jezgara višejezgarnog (eng. *multicore*) procesora.

Na slici 2 su predstavljeni moduli (sa odgovarajućim klasama) paralelizatora: analizador asemblerskog koda (CParser), SSA međukod (CRegisterRename), makro instrukcije (CMacInstruction), graf toka podataka (CDdg), analiza pristupa memoriji (CAliasAnalysis), METIS biblioteka za podelu grafa toka podataka na željeni broj celina (CDDGPartitioner), algoritam paralelizacije (CParallelizer), dodela registara (CResourceAllocator) i emiter koda (CDumper).



Slika 3.2 UML dijagram klasa paralelizatora

Ceo program je realizovan tako što se u okviru *main* funkcije poziva metoda *Do()* klase *CAsmParallelizer*. U metodi *Do()* se pozivaju metode *Do()* preostalih klasa koje realizuju već navedene module paralelizatora.

4. Realizacija

Pri realizaciji su korišćeni različiti alati. Za razvoj paralelizatora je korišćen Microsoft Visual Studio 2010, programski jezik C++.

Paralelizator je podeljen na tri dela: prednji, srednji i zadnji deo paralelizatora.

4.1 Prednji deo paralelizatora

Prednji deo paralelizatora se sastoji od analizatora sintakse asemblerskog koda sa transformacijom u SSA međukod i makro instrukcije.

4.1.1 Analizator sintakse asemblerskog koda

Da bi se, za analizator sintakse, dobila razumljiva predstava asemblerskog koda potrebno je izvršiti prevođenje iz obične tekstualne datoteke u programske objekte. Za tu svrhu razvijen je analizator sintakse asemblerskog koda.

Analizator sintakse je realizovan podelom niza znakova na simbole (eng. *token*) i switch-case konstrukcije i podeljen je na dva nivoa:

1. Prvi deo izvlači liniju po liniju iz asemblerskog koda, zanemarujući linije koje nisu od interesa za dalju obradu.
2. Drugi deo tekstualnu liniju deli na simbole koji su od interesa za dalju obradu.

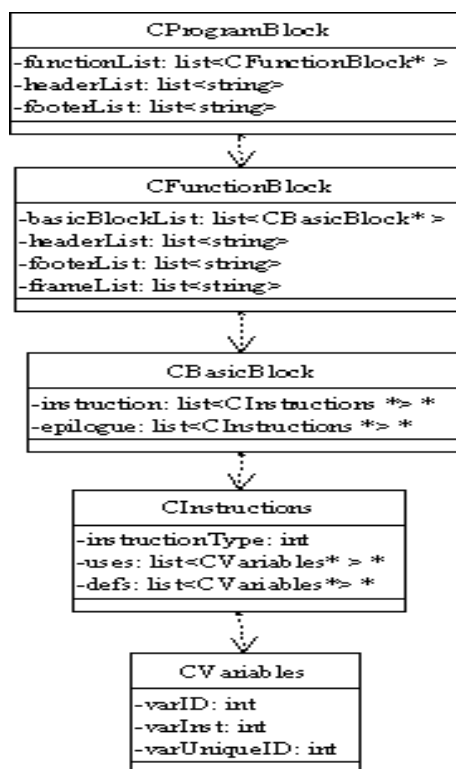
Na primer, analizator sintakse liniju "ADD \$t1,\$t2,\$t3", prebacuje u 4 simbola: "ADD" – tip instrukcije, "\$t1" – odredišni registar, "\$t2" – izvorišni registar i "\$t3" – izvorišni registar.

Ulaz u switch-case konstrukciju je tip instrukcije. Struktura prepoznaje tip instrukcije i podešava listu upotreba i definicija promenljivih.

Analizator sintakse izvlači liniju po liniju iz asemblerskog koda i tako deli ceo asemblerski kod na: program, funkcije i osnovne blokove (eng. *basic block*). Svi podaci su spregnuti pokazivačima. Polazni asemblerski program je predstavljen listom objekata *CFunctionBlock*, pri

čemu svaki element liste sadrži podatke o jednoj funkciji iz polaznog programa. Funkcija počinje direktivom *.align*, a završava se direktivom *.size*. Funkcija sadrži podatke o osnovnim blokovima odnosno listu objekata tipa *CBasicBlock*, a osnovni blok sadrži podatke o instrukcijama, tačnije listu objekata tipa *CInstruction*. Svaka instrukcija sadrži dve liste i tip instrukcije. Liste koje sadrži su lista upotreba (eng. *uses list*) i lista definicija (eng. *defs list*), koje se koriste u određivanju liste upotreba i definicija promenljivih.

Na slici 4.1 prikazana je struktura podataka analizatora sintakse asemblerskog koda.



Slika 4.1 Struktura podataka analizatora sintakse asemblerskog koda

4.1.2 SSA međukod

Modul SSA međukod je deo analizatora sintakse, ali kao važna tehnika zahteva da bude posebno predstavljen. U SSA međukodu svaka promenljiva se definiše tačno jednom, a može se koristiti više puta. Postojanje takvih promenljivih predstavlja uvod za lakše primenjivanje optimizacionih tehnika u toku prevođenja programa. Postoje radovi koji pokazuju da dodela registara algoritmom linearnog pretraživanja na SSA međukodu daje iste ili čak bolje rezultate u odnosu na klasičnu dodelu registara putem matematičke metode bojenja grafova [2].

Na primeru iz tabele 4.1, promenljiva X na osnovu SSA analize se preimenuje u X.2, jer se obavlja ponovno definisanje. U primeru se vidi da X.3 ne uzima vrednost X.1, nego X.2 pošto je X ponovo definisan.

1. $X = 2$	1. $X.1 = 2$
2. $Y = 3$	2. $Y.1 = 3$
3. $X = X + Y$	3. $X.2 = X.1 + Y.1$
4. $X = X + 5$	4. $X.3 = X.2 + 5$

Tabela 4.1 Primer prebacivanja u SSA

Registri koji su vitalni za pravilno funkcionisanje asemblerskog programa su izuzeti iz transformacije. Tu se misli na: nula registar (\$0), registar koji koristi assembler za prebacivanje iz pseudoinstrukcija u fizičke instrukcije (\$at), registre za prenos parametara iz funkcije (\$v0, \$v1), registre za prenos parametara u funkciju (\$a0,\$a1,\$a2,\$a3), registre rezervisane za jezgro (eng. *kernel-\$k0,\$k1*), globalni pokazivač (\$gp) i registar povratne adrese (\$ra). Menjanjem ovih registara ugrozilo bi se pravilno izvršavanje programa i mogli bi se dobiti nekonzistentni rezultati izvršavanja.

Sama realizacija koda urađena je na sledeći način:

Prolazi se kroz listu instrukcija osnovnog bloka. Nailaskom na svaku novu definiciju promenljive X u instrukciji, stvara se nova promenljiva tipa CVariables i uvećava njena instanca za jedan. Instanca predstavlja polje u klasi CVariables. Kada se stvori nova promenljiva X.1 (promenljiva X sa instancom jedan), sve instrukcije koje koriste promenljivu X, se sada postavljaju da pokazuju na promenljivu X.1.

4.1.3 Makro instrukcije

Već je napomenuto u odeljku 3, da je svrha uvođenja makro instrukcije da METIS algoritam makro instrukciju, odnosno skup instrukcija u okviru makro instrukcije, raspodeli na isto jezgro procesora.

Makro instrukcija sadrži skup instrukcija čija je prva instrukcija u skupu instrukcija, instrukcija koja definiše \$acc registar, a ne koristi \$acc registar (to su instrukcije *mult*, *div*, *mthi*, *mtlo*), a poslednja instrukcija u skupu instrukcija je instrukcija koja poslednja koristi \$acc registar, pre nego što se \$acc ponovo definiše. Registar \$acc se ponovo definiše kod *mult*, *div*, *mthi*, *mtlo* instrukcija. Makro instrukcija kao listu upotreba i listu definicija promenljivih ima sve liste upotreba i definicija promenljivih od svih instrukcija koje čine makro instrukciju. Cena koštanja makro instrukcije je ukupan broj instrukcija u okviru makro instrukcije. Sve makro instrukcije su smeštene u jedan skup instrukcija. Tip skupa je STL skup (*set*).

4.2 Srednji deo paralelizatora

Srednji deo paralelizatora sastoji se od sledećih modula: grafa toka podataka, analize pristupa memoriji, METIS biblioteke za podelu grafa toka podataka na željeni broj celina i algoritma paralelizacije.

4.2.1 Graf toka podataka

Apstrakcijom programa kao grafa veoma se dobro obuhvataju strukture zavisnosti koje postoje u programu. Graf toka podataka se izvodi nad svim instrukcijama asemblerskog koda. U ovom radu čvorovi grafa predstavljaju instrukcije, a grane grafa spajaju međusobno zavisne instrukcije. Grane grafa predstavljaju promenljive. Svaka instrukcija ima listu instrukcija od kojih zavisi. Da bi se odredila zavisnost između instrukcija, za svaku instrukciju se posmatraju njene promenljive. Dve instrukcije su zavisne ako se jedna te ista promenljiva pojavljuje i u jednoj i u drugoj instrukciji.

Realizacija koda grafa toka podataka je urađena tako što svaka promenljiva tipa *CVariables* ima kao polje vektor instrukcija. U vektor instrukcija promenljive *P* se upisuju sve instrukcije koje sadrže promenljivu *P*, tako da se za jednu instrukciju posmatraju njene promenljive i u skup *s* upisuje vektor instrukcija koji sadrži te promenljive. Na kraju grafa toka podataka svaka instrukcija ima listu instrukcija od kojih zavisi.

4.2.2 Analiza pristupa memoriji

Analiza pokazivača asemblerskog koda za MIPS procesor je urađena u dva prolaza.

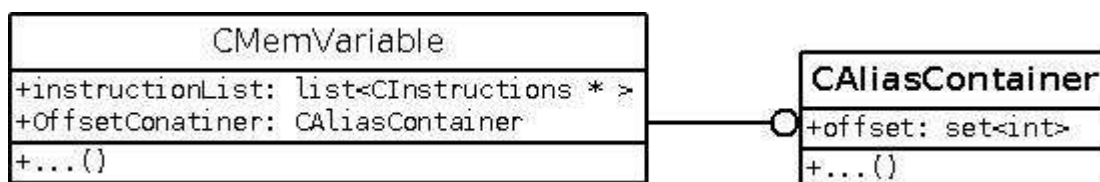
Prvi prolaz beleži pristupe memoriji. Kod MIPS procesora postoje samo osnovni pristupi memoriji. Svi pristupi memoriji su piši (eng. *write*) i čitaj (eng. *read*) operacije. Instrukcije koje pristupaju memoriji su:

- Za pisanje: lb, lh, lw, lbu, lhu, lwl, lwr, ulw
- Za čitanje: sb, sh, sw swl, swr, usw

Preko instrukcija za pristup memoriji se beleži pristupi memoriji. Za potrebe skupljanja podataka razvijena je specifičan programski objekat. Klasa koja opisuje objekat data je na slici 4.2. Objekat simulira promenljivu koja definiše tačku pristupa memoriji. U okviru promenljive je definisana potklasa koja vodi računa o odstojanju od pristupa memoriji. Naime format asemblerske instrukcije za čitanje/pisanje je *OPNAME \$R,N(\$M)*. Pri čemu *OPNAME* predstavlja tip instrukcije (*OPNAME* = lw, sw, sb, lb ...). *\$R* predstavlja registar procesora u koji se pišu ili iz koga se čitaju podaci. *\$M* je registar koji sadrži adresu u memoriji iz koje se čita/piše, a *N* je odstojanje (eng. *offset*) od adrese. Asemblerski programi imaju tendenciju da

koriste jedan registar za pristup podacima. Taj registar je najčešće dodeljen prvoj lokaciji gde su smešteni podaci, a sa promenom N pristupa se podacima u okolini.

Struktura (objekat) koja je razvijena za pronalaženje pristupa memoriji se oslanja na ovakvu asemblersku organizaciju. Objekat promenljive (*CMemVariable*) je apstrakcija pristupnog registra ($\$M$), a odstupanja (N) su objekat sadržan u strukturi. Naziv pod objekta je *CAliasContainer*. *CAliasContainer* sadrži C++ STL skup (eng. *set*) u kome se beleže odstupanja (*offseti*) od $\$M$.



Slika 4.2 Pregled klasa za evidenciju podataka o pokazivačima na memoriju

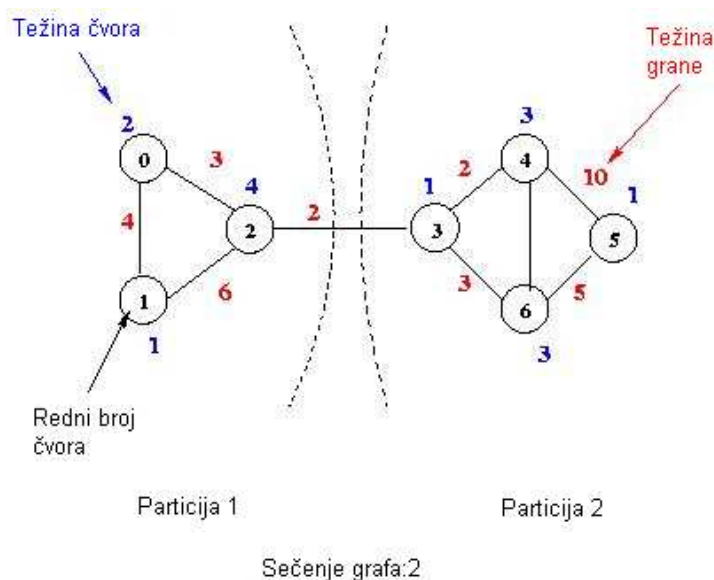
Drugi prolaz podrazumeva dodeljivanje *CMemVariable* objekata instrukciji koja pristupa memoriji. Ovime su susedne memorijske lokacije stavljene pod jednu promenljivu. Primer ispod ilustruje postupak.

1. lw \$3,0(\$2)
2. lw \$4,4(\$2)
3. add \$5,\$4,\$3

Registar \$3 je pokazivač na adresu u memoriji. Programska promenljiva koja je dodeljena instrukciji 1. i 2. je ista, ali su odstojanja različita iz čega možemo zaključiti da su u pitanju različite memorijske lokacije sa kojih, u ovom slučaju, čitamo. Analogno ovom primeru isti postupak se postiže i za lw/sw ili bilo koju drugu kombinaciju instrukcija.

4.2.3 METIS biblioteka

Na osnovu grafa toka podataka, METIS algoritam razdvaja instrukcije na različita jezgra. Algoritam deli graf toka podataka na željeni broj particija, obezbeđujući približno jednako opterećenje po jezgrima procesora. Algoritam pokušava da pronađe što manji broj veza između particija na koje seče graf, što je prikazano na slici 4.3.



Slika 4.3 Sečenje grafa na dve particije

Ovaj modul je realizovan na sledeći način:

Svakoj instrukciji se dodeljuje jedinstveni identifikacioni broj ID, koji počinje od nule, a završava se sa ukupnim brojem instrukcija. Stvara se matrica *dependencyArray* koja sadrži ID-ove instrukcija, gde svaka instrukcija ima listu instrukcija od kojih zavisi. U svakom redu matrice se nalaze te liste instrukcija, odnosno njeni ID-ovi. Matrica se funkcijom *TransponeToCSR* transformiše na *CSRXadj* niz i *CSRadj* niz. Ovi nizovi predstavljaju *xadj* i *adjncy* nizove objašnjene u odeljku 2.2.2, dok *dependencyArray* prestavlja listu povezanosti grafa takođe objašnjeno u odeljku 2.2.2. Nizovi *CSRXadj* i *CSRadj*, težine čvorova i težine veza instrukcija se kao parametri prosleđuju funkciji *metisAPI*, koja kao povratnu vrednost vraća niz *cores*. Svaki element niza *cores* predstavlja broj jezgra na koje je odgovarajuća instrukcija raspodeljena.

4.2.4 Algoritam paralelizacije

Sam algoritam paralelizacije objašnjen je ukratko pseudokodom u odeljku 2.3. Sada će detaljnije biti objašnjena realizacija koda algoritma paralelizacije.

Metoda *Do()* klase *CParallelizer* realizuje ceo algoritam paralelizacije. U metodi *Do()* se prolazi kroz sve osnovne blokove programa i za svaki blok se poziva metoda *DoPartitioning* nad blokom. Metoda *DoPartitioning* radi sledeće:

Stvara onoliko blokova *coreBlock* koliko ima jezgara, gde se na kraju algoritma paralelizacije u svakom *coreBlock* nalaze instrukcije raspodeljene po jezgrima. Ukoliko je instrukcija tipa makro instrukcije, potrebno je makro instrukciju razložiti na instrukcije od kojih se makro instrukcija sastoji, tako da se umesto jedne makro instrukcije dobija više instrukcija. Zatim se prolazi kroz sve instrukcije bloka i za svaku instrukciju se poziva faza *doUrgencyRule*

data u odeljku 2.3. U toj fazi se proverava koje su instrukcije već spremne, odnosno za koje instrukcije su definisane sve promenljive iz liste korišćenih promenljiva. Instrukcija je spremna ako promenljiva ima instancu *varInst* jednaku nuli i ako je njeno polje *readyVar* jednako jedan. Polje *readyVar* neke promenljive dobija vrednost jedan nakon što se neka instrukcija ubaci u listu spremnih instrukcija *listReadyInstructions*, pa *readyVar* odredišne promenljive spremne instrukcije dobija vrednost jedan. Potom se prolazi kroz listu spremnih instrukcija i primenjuje faza *doSyncRule* data u odeljku 2.3. U toj fazi se uzima prva instrukcija *FI* iz liste spremnih instrukcija i za nju se proverava da li su njene korišćene promenljive definisane na istom jezgru kao i *FI* instrukcija. Ako nisu, potrebno je dodati sinhronizacione instrukcije. Instrukciju *sw* treba dodati na ono jezgro na kom je promenljiva definisana, a instrukciju *lw* treba dodati na jezgro gde se promenljiva koristi. Instrukcija *sw* se mora dodati na jezgro pre instrukcije *lw* u suprotnom dodaju se *nop* instrukcije na odgovarajuća jezgra. Ako se istovremeno pojave dve sinhronizacione instrukcije na jezgrima, takođe se dodaju *nop* instrukcije. Pre nego što se ubace *nop* instrukcije, poziva se faza *doFillRule* data u odeljku 2.3. U njoj se prolazi kroz listu spremnih instrukcija i proverava da li je moguće ubaciti neku instrukciju *SI* iz liste spremnih instrukcija pre instrukcije *FI* koja se izvršava. To ubacivanje instrukcije se radi tako što se proverava da li su neka instrukcija *SI* iz liste spremnih instrukcija i instrukcija *FI* raspoređene na isto jezgro. Ako jesu, proverava se za instrukciju iz liste spremnih *SI* da li su definisane sve njene korišćene promenljive na jezgru na kom se nalazi. Ako je i to zadovoljeno, dodaje se instrukcija koja nije ni *lw* ni *sw* instrukcija na jezgro. Ako je instrukcija *lw* ili *sw* poziva se metoda *Reschedule*. Metoda *Reschedule* obezbeđuje da se u istom trenutku na jezgrima ne mogu pojaviti instrukcije *lw* i *sw*, *sw* i *lw*, *sw* i *sw* koje koriste isti registar i odstojanja su im ista. Ako je to slučaj dodaju se *nop* instrukcije, pa tek se onda instrukcija raspoređuje na jezgro. U suprotnom instrukcija se odmah raspoređuje na jezgro, bez dodavanja *nop* instrukcija. Ukoliko za neku instrukciju skoka na nekom jezgru nisu definisane sve korišćene promenljive na tom jezgru, prolazi se kroz isti algoritam opisan za instrukcije koje nisu skokovi. Instrukcije skoka se dodaju poslednje na sve blokove *coreBlock*.

4.3 Zadnji deo paralelizatora

Zadnji deo paralelizatora sadrži dodelu registara nad obrađenim asemblerskim kodom i emiter koda.

4.3.1 Dodela registara

Pod dodelom registara podrazumeva se dodeljivanje fizičkih lokacija, registara i memorije, logičkim podacima, u ovom slučaju podacima predstavljenim u SSA međukodu.

Pre nego što se počne sa linearnom dodelom registara, potrebno je uraditi analizu životnog veka promenljivih. Iz tabele 4.2 se vidi da su promenljive Rs1 i Rs2 “žive” na 1, 2 i 3 liniji izvornog koda, dok je Rs3 živa na 3 i 4 liniji izvornog koda.

Model SSA međukoda u slučaju analizatora sintakse uzima u obzir jedan izuzetak. Taj izuzetak su registri koji ne smeju da se menjaju. Menjanje bilo kojeg od tih registara ima za posledicu nedefinisano izvršenje programa.

Registri koji ne smeju da se menjaju su: nula registar (\$0), registar koji koristi asembler za prebacivanje iz pseudoinstrukcija u fizičke instrukcije (\$at), registri rezervisani za jezgro (\$k0,\$k1), globalni pokazivač (\$gp), pokazivač na stek (\$sp), pokazivač na okvir (\$fp), registar povratne adrese (\$ra), registri za povratak rezultata iz funkcije (\$v0,\$v1) i registri za prenos parametara u funkciju (\$a0,\$a1,\$a2,\$a3) [7].

Ostali registri se nalaze u skupu registara za dodelu. Tip ovog skupa je STL skup (*set*). Skupom je postignuto da se očuva jedinstvenost registara. Kao takvi, slobodni su za korišćenje u dodeli.

Kada se zauzme registar, on se uzima iz skupa slobodnih registara i prebacuje se u skup zauzetih registara. Kada dođe do kraja životnog veka registra, registar se vraća u skup slobodnih registara. Tabela 4.2 prikazuje tok dodeljivanja registara.

	1	2	3	4
Izvorni kod	X=2	Y=3	X=X+Y	X= X +5
SSA	X.0 = 2	Y.0 = 3	X.1 = X.0 + Y.0	X.2 = X.1 + 5
Žive promenljive	Rs1	Rs1,Rs2	Rs1,Rs2, Rs3	Rs3,Rs4
Dodeljeni registar	R1	R1,R2	R1,R2,R3	R1,R3
Objašnjenje	Dodeljen registar R1	Dodeljen registar R2	Redefinisano X dodeljen novi registar	R1 se ponovo uvodi, ali kao Rs4 promenljiva

Tabela 4.2 Tok dodeljivanja registara, od izvornog programa do registara hipotetičkog procesora

4.3.2 Emiter koda

Prikaz asemblerskog koda razdvojenog po jezgrima omogućava emiter koda, odnosno klasa *CDumper*. U zavisnosti od parametara prosleđenih konstruktoru *CDumper*, izlaz iz emitera koda je datoteka u formatu *.txt* ili *.csv*. Emiter koda računa ubrzanje pojedinih blokova, funkcija i celog programa, koje se prikazuje u datoteci. U datoteci se ne prikazuje asemblerski kod vezan za zaglavlja programa i funkcija, nego samo tekst segment asemblerskog koda.

4.4 Paralelizacija paralelnih petlji

U paralelizatoru je omogućena i paralelizacija asemblerskog koda sa paralelnim petljama. U asemblerski kod se dodaje direktiva koja označava da petlju treba paralelizovati. Ta direktiva je *#begin_parallel_for i=0, n=4*. U direktivi se nalaze podaci o inicijalnoj vrednosti brojača petlje *i* i ukupnom broju prolaza kroz petlju *n*. Direktiva *#end_parallel_for* označava da se petlja završila i da počinje asemblerski kod koji nije programska petlja.

U metodi *Do()* klase *CParser* nailaskom na direktivu *#begin_parallel_for i=0, n=4* prilikom čitanja linije po linije iz asemblerskog koda, stvara se novi blok *block* i postavlja se polje *parallelFor* bloka *block* na jedan. U blok *block* se dodaju sve instrukcije koje se nalaze između direktiva *#begin_parallel_for i=0, n=4* i *#end_parallel_for*. Posle direktive *#end_parallel_for*, polje *parallelFor* se postavlja na nulu i stvara se novi blok. Blok čije je polje *parallelFor* jednako jedan ne prolazi kroz klase *CRegisterRename*, *CDdg* i *CDDGPartitioner*.

U klasi *CParallelizer* metoda *DoPartitioning* obrađuje blok *block* čije je polje *parallelFor* jednako jedan na sledeći način: iz direktive *#begin_parallel_for i=0, n=4* se preuzimaju podaci o inicijalnoj vrednosti brojača petlje i ukupnom broju prolaza kroz petlju i te dve vrednosti se upisuju redom u promenljive *init* i *limit*. Potom se prolazi kroz sve instrukcije bloka *block* i za svaku instrukciju se proverava da li njeno polje *instruction* u sebi sadrži *# for_init* ili *# for_limit*. Polje *instruction* je tipa *string* i sadrži jednu liniju asemblerskog koda, odnosno jednu instrukciju sa komentarom, ako ga ima (npr. *li \$2,4 # for_limit*). Ako polje *instruction* neke instrukcije sadrži *# for_init*, znači da instrukcija predstavlja inicijalnu vrednost brojača, a ako polje *instruction* neke instrukcije sadrži *# for_limit*, onda instrukcija predstavlja ukupan broj prolaza kroz petlju. U zavisnosti od broja jezgara, instrukcije koje sadrže *# for_init* i *# for_limit*, se menjaju. Ovo je objašnjeno na sledećem primeru:

```
# jezgro 0
li $2, 0 # for_init
li $3,8  # for_limit
```

Nakon algoritma paralelizacije, navedene dve instrukcije se menjaju u sledeće instrukcije (pretpostavlja se da postoje 2 jezgra):

```
#jezgro 0      #jezgro 1
li $2,0         li $2,4
li $3,3         li $3,7
```

Inicijalna vrednost brojača se dobija sledećom formulom:

```
init+=limit/broj_jezgara;
```

Na taj način se dobijaju brojevi 0 i 4 u instrukcijama i pri tom se instrukcije *li \$2,0* i *li \$2,4* smeštaju na nulto i prvo jezgro.

Vrednost za ukupan broj prolaza kroz petlju se dobija sledećim formulama:

$\text{limit1} = \text{limit}/\text{broj_jezgara}-1;$

$\text{limit1} += \text{limit}/\text{broj_jezgara};$

gde se prilikom prolaska kroz blok *block*, za prvu instrukciju bloka koristi prva navedena formula, a za sve ostale instrukcije bloka druga navedena formula. Formulama se dobijaju brojevi 3 i 7 u instrukcijama i pri tom se instrukcije *li \$3,3* i *li \$3,7* smeštaju na nulto i prvo jezgro.

Samo se instrukcije koje inicijalizuju brojač petlje i postavljaju ukupan broj prolaza kroz petlju menjaju u bloku *block*, dok ostale instrukcije iz bloka *block* ostaju iste i smeštaju se na zadati broj jezgara. Na taj način dobija se assemblyski kod koji paralelnu petlju, odnosno njene instrukcije, raspodeljuje na zadati broj jezgara.

5. Ispitivanje

5.1 Ispitivanje paralelizacije sekvencijalnog koda

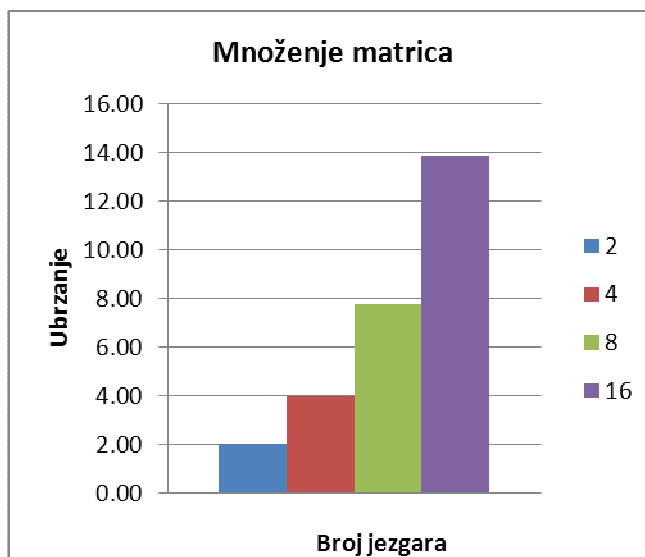
Ispitivanje je rađeno na primerima standardnog množenja matrica i Strassenovog metoda množenja matrica. U ovim primerima je rađena samo paralelizacija tela petlji, dok same petlje nisu paralelizovane. Na tim primerima je posmatrana promena ubrzanja koda sa povećanjem broja jezgara. Ubrzanje je definisano kao odnos: T_s / T_p , gde je T_s vreme izvršavanja sekvencijalnog programa, dok je T_p vreme izvršavanja paralelnog programa. Pretpostavka je da sve instrukcije imaju jednako vreme izvršenja. T_s je u stvari ukupan broj instrukcija sekvencijalnog asemblerskog koda, dok je T_p najveći broj raspoređenih instrukcija po jezgru.

Iz tabele 5.1 može se videti da ubrzanje koda na primeru množenja matrica raste, kako se povećava broj jezgara. Sa samo dva jezgra, dobija se dvostruko veće ubrzanje u odnosu na sekvencijalni kod. Sa 16 jezgara ubrzanje je skoro 14 puta.

Broj jezgara	Ubrzanje koda na primeru množenja matrica
2	1.99448
4	3.94536
8	7.76344
16	13.8846

Tabela 5.1 Rezultati eksperimentalnog merenja ubrzanja koda na primeru množenja matrica

Na slici 5.1 su grafički prikazani rezultati merenja ubrzanja koda na primeru množenja matrica u zavisnosti od broja jezgara.



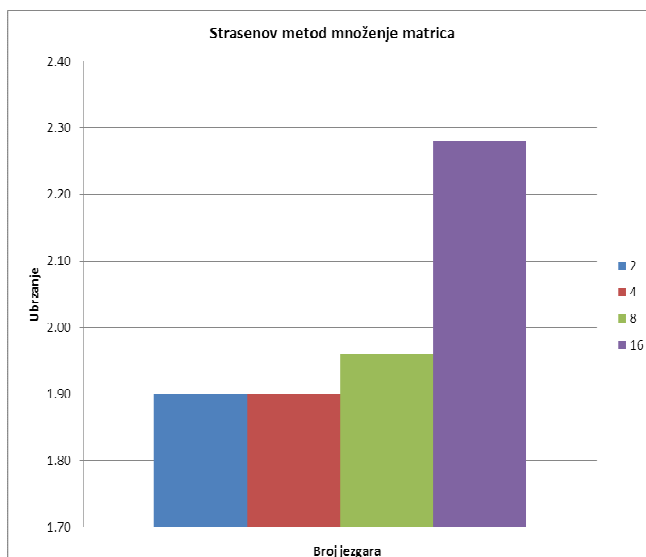
Slika 5.1 Rezultati ubrzanja koda na primeru množenja matrica u zavisnosti od broja jezgara

Iz tabele 5.2 može se videti da ubrzanje koda na primeru Strassenovog metoda množenja matrica raste, kako se povećava broj jezgara. Međutim, može se videti da se sa 2 i 4 jezgra postiže isto ubrzanje koda, dok za 8 i 16 jezgara ubrzanje počinje da raste. Sa povećanjem broja jezgara ubrzanje kod standardnog množenja matrica je mnogo veće nego kod Strassenovog metoda množenja matrica, što se može videti iz tabela 5.1 i 5.2.

Broj jezgara	Ubrzanje koda na primeru Strassenovog metoda množenja matrica
2	1.9
4	1.9
8	1.96
16	2.28

Tabela 5.2 Rezultati eksperimentalnog merenja ubrzanja koda na primeru Strassenovog metoda množenja matrica

Na slici 5.2 su grafički prikazani rezultati merenja ubrzanja koda na primeru Strassenovog metoda množenja matrica u zavisnosti od broja jezgara.



Slika 5.2 Rezultati ubrzanja koda na primeru Strasenovog metoda množenja matrica u zavisnosti od broja jezgara

5.2 Ispitivanje paralelizacije paralelnih petlji

Ispitivanje paralelizacije paralelnih petlji je rađeno na primeru *for* petlje koja računa skalarni zbir dva niza i rezultat upisuje u treći niz. Na primeru je posmatrana promena ubrzanja koda sa povećanjem broja jezgara. Ubrzanje je već definisano u odeljku 5.1. Formula na osnovu koje se računa ubrzanje je:

$$T_s / T_p = (P_s + E_s + (n * T)) / (P_p + E_p + (n / \text{broj_jezgara}) * T)$$

P_s - ukupan broj instrukcija sekvencijalnog programa pre tela *for* petlje

E_s - ukupan broj instrukcija sekvencijalnog programa posle tela *for* petlje

n - broj ponavljanja *for* petlje

T - ukupan broj instrukcija tela *for* petlje

P_p - ukupan broj instrukcija paralelnog programa pre tela *for* petlje

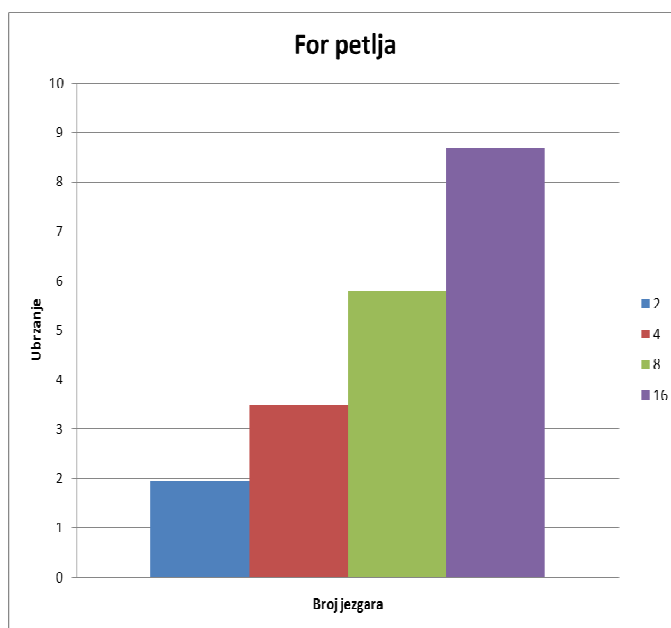
E_p - ukupan broj instrukcija paralelnog programa posle tela *for* petlje

Iz tabele 5.3 može se videti da ubrzanje koda na primeru *for* petlje raste, kako se povećava broj jezgara.

Broj jezgara	Ubrzanje koda na primeru <i>for</i> petlje
2	1.93
4	3.47
8	5.79
16	8.69

Tabela 5.3 Rezultati eksperimentalnog merenja ubrzanja koda na primeru *for* petlje

Na slici 5.3 su grafički prikazani rezultati merenja ubrzanja koda na primeru *for* petlje u zavisnosti od broja jezgara.



Slika 5.3 Rezultati ubrzanja koda na primeru *for* petlje u zavisnosti od broja jezgara

6. Zaključak

Paralelizator asemblerskog koda se pokazao kao tema koja ima veoma široku primenu. Imati program koji može ulazni sekvencijalni kod da pretvori u asemblerski kod prilagođen radu na više jezgara čini se kao ideal. Ovaj rad realizuje paralelizaciju sekvencijalnog koda i predstavlja prvi korak ka tom ambicioznom cilju.

Analizom rezultata ispitivanja može se zaključiti da su rezultati zadovoljavajući. Postignuto je veliko ubrzanje koda, sa povećanjem broja jezgara.

Kao deo analizatora sintakse realizovano je prebacivanje u SSA međukod, čime je dobijena jednostavnija obrada podataka, tj. jednostavnije određivanje zavisnosti između registara (promenljivih u SSA međukodu).

Jedno od mogućih rešenja za analizu preklapanja memorijskih referenci, da bi se raščlanile tačke pisanja i čitanja iz memorije kako bi se izbegao rizik trke do podataka, je korišćenje Strasengardovog (eng. *Bjarne Steensgaard*) ili Andersenovog (eng. *Andersen*) algoritma za dereferenciranje memorije. Ta dva rešenja prave liste pokazivača na memoriju (kao u ovom radu), a nakon toga preračunavaju moguća odstojanja. Vršanjem takvih preračunavanja moguće je zaključiti da su neki registri pokazivali na istu memorijsku lokaciju.

METIS algoritam se pokazao kao dobar algoritam u podeli grafa toka podataka na željeni broj particija. Omogućio je veliku brzinu i veoma kvalitetno particionisanje, što je i karakteristika algoritma.

Algoritam paralelizacije se pokazao kao dobar, jer se njegovim korišćenjem na pojedinim jezgrima smanjuje broj instrukcija (zamenjivanjem nepotrebnih instrukcija korisnim instrukcijama), čime se postiže ubrzanje koda.

Izbor linearne dodele registara u ovom radu je ispunio svoju svrhu. Relativno jednostavan algoritam koji ne troši puno resursa za dodelu. Po efikasnosti dodeljivanja ne zaostaje za klasičnim algoritmom dodele registara, što je pokazano u radu [8].

Paralelizator je sposoban da paralelizuje osim sekvencijalnog programskog koda i programski kod sa paralelnim petljama.

Jedan od pravaca daljeg razvoja je uvođenje analize zasnovane na grafu toka upravljanja i određivanje zavisnosti između osnovnih blokova.

7. Literatura

- [1] Alfred V. Aho, Monica S. Lam, Ravi Seth, Jeffrey D. Ullman, "*Compilers, Principles, Techniques & Tools*", second edition, Pearson Education, Inc, 2007
- [2] Christian Wimmer Michael Franz, "*Linear Scan Register Allocation on SSA Form*", Department of Computer Science University of California, Irvine
- [3] Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, Vol.20, No.1, pp. 359-392, 1998.
- [4] Barnard, S.T., Simon, H.D.: A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, in *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, SIAM, pp. 711-718., 1993.
- [5] Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, Vol. 49, No. 2, pp. 291-307, 1970.
- [6] George Karypis, METIS, University of Minnesota, Minneapolis, MN 55455, August 4, 2011
- [7] Dominic Sweetman, "*See MIPS run second edition*", Morgan Kaufman, 2007
- [8] "*Linear Scan Register Allocation*", Laboratory for Computer Science, MIT and VIVEK SARKAR, IBM Thomas J. Watson Research Center
- [9] Darko Čapko, "Optimalna podela velikih modela podataka u okviru nadzorno-upravljačkih elektroenergetskih sistema", doktorska disertacija