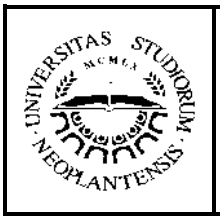


УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ЗАВРШНИ (BACHELOR) РАД

Кандидат: **Александар Станковић**
Број индекса: **СВ 25/2022**

Тема рада: **Имплементација и испитивање перформанси Кацове централности над великим графовима**

Ментор рада: **проф. др Мирослав Поповић**

Нови Сад, Јул, 2026

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**:		Завршни (Bachelor) рад
Аутор, **АУ**:		Александар Станковић
Ментор, **МН**:		проф. др Мирослав Поповић
Наслов рада, **НР**:		Имплементација и испитивање перформанси Кацове централности над великим графовима
Језик публикације, **ЈП**:		Српски / ћирилица
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2026
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)		7/40/25/12/16/0/0
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Софтверско инжењерство и информационе технологије
Предметна одредница/Кључне речи, **ПО**:		графови; Кацова централност; CSR формат; множење ретке матрице вектором; паралелно програмирање; дељена меморија
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		У раду је реализовано и испитано шест имплементација алгоритма за Кацову централност над великим ретким графовима. Све користе CSR формат, исти интерфејс командне линије и исти формат излазних датотека. Разликују се по моделу паралелизације: вишепроцесна обрада у језику Python, паралелни итератори у језику Rust, библиотечка и директивна паралелизација у језику C++, као и извршавање на графичкој процесорској јединици. Извршена су испитивања коректности, конвергенције, јаког и слабог скалирања на јавним и синтетичким графовима. Резултати показују нумеричку еквивалентност свих имплементација и указују да је за тумачење перформанси неопходно раздвојити припрему графа од итерационог језгра, јер је главно ограничење меморијски саобраћај.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	проф. др Игор Дејановић
	Члан:	доц. др Миодраг Ђукић
	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**:		Bachelor Thesis	
Author, **AU**:		Aleksandar Stanković	
Mentor, **MN**:		Miroslav Popović, PhD	
Title, **TI**:		Implementation and Performance Evaluation of Katz Centrality on Large Graphs	
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**:		2026	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		7/40/25/12/16/0/0	
Scientific field, **SF**:		Electrical Engineering and Computing	
Scientific discipline, **SD**:		Software Engineering and Information Technologies	
Subject/Key words, **S/KW**:		graphs; Katz centrality; CSR; SpMV; parallel programming; CUDA; OpenCL; OpenMP; TBB; Rayon	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		This thesis implements and evaluates six versions of Katz centrality for large sparse graphs. All versions use the same CSR graph representation, command-line interface and output format, while the parallelization model differs: Python multiprocessing, Rust Rayon, C++ oneTBB, C++ OpenMP, NVIDIA CUDA and Khronos OpenCL. Correctness, convergence, strong scaling and weak scaling experiments were performed on public SNAP graphs and synthetic graphs. The results show numerical equivalence across all variants, strong relative scaling of the Python multiprocessing version, limited scaling of efficient compiled CPU variants due to memory bandwidth, and a GPU crossover only for larger workloads.	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	Igor Dejanović, PhD	
	Member:	Miodrag Đukić, PhD	Menthor's sign
	Member, Mentor:	Miroslav Popović, PhD	

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА ИЗРАДУ ЗАВРШНОГ (BACHELOR) РАДА	Датум:

(Податке уноси предметни наставник - ментор)

Врста студија:	а) Основне академске студије б) Основне струковне студије
Студијски програм:	Софтверско инжењерство и информационе технологије
Руководилац студијског програма:	проф. др Игор Дејановић

Студент:	Александар Станковић	Број индекса:	СВ 25/2022
Област:	Електротехника и рачунарство		
Ментор:	проф. др Мирослав Поповић		

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

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

НАСЛОВ ЗАВРШНОГ (BACHELOR) РАДА:

Имплементација и испитивање перформанси Кацове централности над великим графовима

ТЕКСТ ЗАДАТКА:

У оквиру овог дипломског рада потребно је урадити следеће:

1. Упознати се са теоријским основама Кацове централности, репрезентацијом графова у CSR формату и множењем ретке матрице вектором (SpMV).
2. Пројектовати заједничку архитектуру и јединствени интерфејс за више имплементација алгоритма.
3. Имплементирати алгоритам Кацове централности у више модела паралелизације (Python, Rust, C++/TBB, OpenMP, CUDA и OpenCL).
4. Дефинисати методологију мерења перформанси и проверу коректности резултата.
5. Спровести експерименте јаког и слабог скалирања над великим реалним графовима.
6. Анализирати меморијски саобраћај, упоредити имплементације и продискутовати резултате.
7. Извести закључке и обезбедити упутство за репродукцију експеримената.

Руководилац студијског програма:	Ментор рада:
проф. др Игор Дејановић	проф. др Мирослав Поповић

Примерак за: ☐ - Студента; ☐ - Ментора

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

ИЗЈАВА О НЕПОСТОЈАЊУ СУКОБА ИНТЕРЕСА

Изјављујем да нисам у сукобу интереса у односу ментор – кандидат и да нисам члан породице (супружник или ванбрачни партнер, родитељ или усвојитељ, дете или усвојеник), повезано лице (крвни сродник ментора/кандидата у правој линији, односно у побочној линији закључно са другим степеном сродства, као ни физичко лице које се према другим основама и околностима може оправдано сматрати интересно повезаним са ментором или кандидатом), односно да нисам зависан/на од ментора/кандидата, да не постоје околности које би могле да утичу на моју непристрасност, нити да стичем било какве користи или погодности за себе или друго лице било позитивним или негативним исходом, као и да немам приватни интерес који утиче, може да утиче или изгледа као да утиче на однос ментор-кандидат.

У Новом Саду, дана _____

Ментор

_____

Кандидат

_____

Захвалност

Најпре желим да се искрено захвалим свом ментору, проф. др Мирославу Поповићу, на издвојеном времену, корисним саветима и смерницама, стрпљењу и подршци током израде овог рада.

Посебну захвалност дугујем и професору Јангу Чену са Универзитета Фудан, као и свом добром пријатељу Хаорану Дуу, такође са Универзитета Фудан. Они су ме први увели у област централности у графовима и подстакли моје интересовање за ову тему, што је имало важан утицај на избор и развој овог рада.

Захваљујем се и својој породици и пријатељима на разумевању, стрпљењу и подршци током студија и током израде завршног рада.

Садржај

Списак слика

Списак табела

Скраћенице

CSR	Сажети редни формат за представљање ретких матрица (енг. Compressed Sparse Row)
SpMV	Множење ретке матрице вектором (енг. Sparse Matrix–Vector Multiplication)
CPU	Централна процесорска јединица (енг. Central Processing Unit)
GPU	Графичка процесорска јединица (енг. Graphics Processing Unit)
L1	Норма збира апсолутних вредности компоненти вектора
SNAP	Јавна збирка графовских скупова података Универзитета Станфорд (енг. Stanford Network Analysis Project)
CUDA	NVIDIA платформа за програмирање графичких процесорских јединица (енг. Compute Unified Device Architecture)
OpenCL	Отворени стандард за хетерогено паралелно програмирање (енг. Open Computing Language)
OpenMP	Скуп директива за паралелно програмирање на дељеној меморији (енг. Open Multi-Processing)
TBB	C++ градивни блокови за вишенитне програме (енг. Threading Building Blocks)

1. Увод

Графови су природан модел за представљање великог броја система, као што су комуникационе и путне мреже, друштвени односи, мреже производа и препорука, као и зависности између софтверских компоненти [5]. Код великих графова нису све везе подједнако значајне; потребно је издвојити чворове који имају већи структурни значај у мрежи. За то служе мере централности које сваком чвору додељују нумеричку вредност на основу његовог положаја у графу [2].

Једна од познатијих мера из ове групе је Кацова централност, уведена као индекс статуса у социометријској анализи [1]. За разлику од степена чвора, који посматра само непосредне суседе, Кацова централност узима у обзир и индиректне путање. Краће путање имају већи допринос, док се допринос дужих путања умањује фактором пригушења α. Таква дефиниција омогућава да чвор буде оцењен као значајан не само зато што има велики број непосредних веза, већ и зато што је повезан са чворовима који су и сами добро повезани.

У рачунарској имплементацији израчунавање Кацове централности своди се на итеративно множење матрице суседства и вектора скорова. Како су реални графови најчешће ретки, матрица суседства се не чува као густа матрица, већ у формату погодном за ретке матрице [8, 9]. У овом раду користи се CSR формат, а доминантна операција у свакој итерацији је множење ретке матрице вектором, односно SpMV. Због тога је овај проблем добар пример за испитивање паралелизације, јер се рачунање по редовима матрице може делити између процеса, нити или GPU радних јединица, али укупне перформансе у великој мери зависе од меморијског приступа, расподеле степена чворова и трошкова синхронизације [10].

Циљ рада је реализација више упоредивих имплементација истог алгоритма, уз исте улазне параметре, исте излазне формате, проверу нумеричке еквивалентности резултата и мерење понашања на различитим класама графова. Посебно се посматрају јако скалирање, слабо скалирање, конвергенција и однос између измерених убрзања и теоријских модела Амдаловог и Густафсоновог закона [18, 19].

Посебан изазов код великих графова представља то што перформансе алгоритама не зависе само од математичког модела, већ и од распореда података у меморији, редоследа чворова и карактеристика хардвера.

Кацова централност је погодна за овакву анализу јер је довољно једноставна за реализацију у више модела паралелизације, али истовремено задржава типичне потешкоће алгоритама над великим графовима. У свакој итерацији пролази се кроз све гране, читају се

вредности суседа и уписује се нови скор, због чега перформансе у великој мери зависе од приступа меморији.

На први поглед, Кацова централност делује као погодан кандидат за паралелизацију, пошто се израчунавање своди на независна ажурирања по редовима матрице и лако се дели између процеса, нити или GPU радних јединица. Управо зато је она поучан пример шире класе проблема обраде савремених великих графова. Свака итерација сведена је на једно множење и једно сабирање по суседу, али сваки такав допринос захтева читање вредности из меморије. Код реалних графова од неколико милиона чворова вектор скорова може значајно да превазиђе капацитет скривене (кеш) меморије процесора. Стварно уско грло тада није број аритметичких јединица, већ меморијски пропусни опсег. У раду је та хипотеза проверена кроз шест имплементација на конкретним графовима из стварног света.

Допринос рада може се сажети у следећем:

- реализовано је шест имплементација алгоритма за Кацову централност: Python, Rust, TBB, OpenMP, CUDA и OpenCL;
- све имплементације користе исти CSR модел графа, исте параметре командне линије и исти формат излаза;
- примењене су три оптимизације ради смањења меморијског саобраћаја: 32-битни индекси у CSR структури, спајање SpMV и ажурирања вектора у једном пролазу и замена GPU бафера уместо копирања између итерација;
- уведен је излаз `profile.csv`, који раздваја време припреме графа од времена итерационог језгра;
- проверена је коректност поређењем финалних вектора скорова за свих тринаест секвенцијалних и паралелних варијанти имплементација;
- спроведено је укупно 1980 мерења јаког скалирања и 1950 мерења слабог скалирања;
- генерисане су табеле и графикони за време извршавања, убрзање, ефикасност, конвергенцију и процену серијског дела програма.

У другом поглављу приказане су теоријске основе: Кацова централност, CSR формат, SpMV операција и закони скалирања. Треће поглавље описује концепт решења и заједничку архитектуру пројекта. Четврто поглавље приказује програмско решење по моделима паралелизације. Пето поглавље представља експерименталну поставку, скупове података и резултате. Шесто поглавље доноси закључке и могуће правце даљег рада.

2. Теоријске основе

2.1. Графови и матрица суседства

Граф се дефинише као уређени пар $G = (V, E)$, где је V скуп чворова, а E скуп грана. У усмереном графу грана (u, v) означава везу од чвора u ка чвору v. У неусмереном графу иста грана представља симетричну везу. У овом раду улазни графови се читају из листе грана, а затим се по потреби третирају као неусмерени додавањем обрнуте гране.

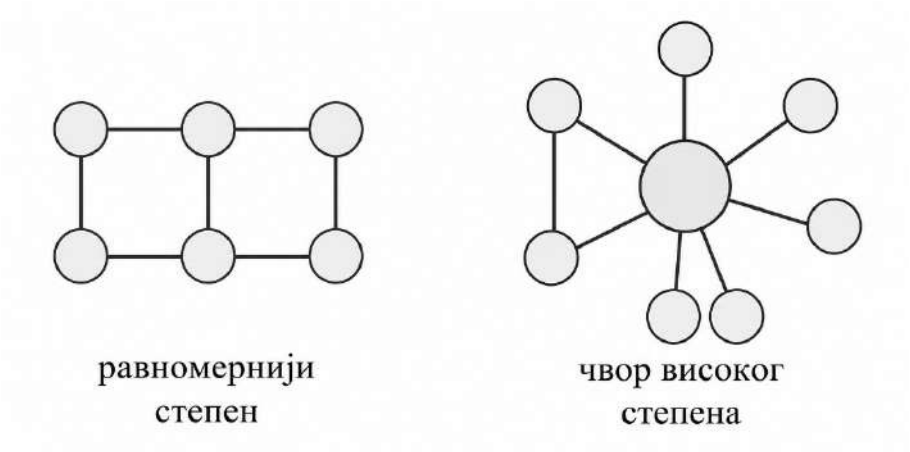
Матрица суседства A графа са n чворова је матрица димензија $n \times n$, где је елемент A_{ij} једнак 1 ако постоји грана од чвора i ка чвору j, а 0 иначе. За велике реалне графове матрица суседства је ретка, односно број постојећих грана је много мањи од n^2. Због тога би чување пуне матрице било меморијски неефикасно. На пример, граф са неколико милиона чворова практично није могуће обрађивати као густу матрицу у радној меморији стандардног рачунара, док се листа постојећих грана и CSR репрезентација могу чувати знатно компактније.

2.2. Структура реалних графова

Ретке матрице суседства великих графова не разликују се од густих матрица само по броју ненултих елемената. Њихова структура одражава својства домена из којег граф потиче [5]. У путним мрежама степен чворова је мали и приближно равномеран, јер раскрсница има ограничен број суседних раскрсница. Друштвене мреже, веб-графови и графови који описују заједничку куповину производа, насупрот томе, често имају истакнуте чворове високог степена. Иако се користи исти математички модел, трошак обраде по реду матрице није исти.

Ова разлика директно утиче на паралелно извршавање. Ако сваки ред матрице одговара једном чвору, онда ред са малим бројем суседа захтева мало читања и сабирања, док ред који одговара чвору високог степена захтева обраду много већег броја суседа. Када се редови поделе на више радних јединица, неједнака расподела степена може довести до тога да неке јединице заврше раније, а друге успоравају завршетак целе итерације. Зато се у тумачењу резултата не посматра само број чворова и грана, већ и расподела степена и локалност приступа претходном вектору скорова.

Слика 2.1 приказује две поједностављене структуре. Граф са леве стране има равномернију расподелу степена чворова и личи на локалну мрежу, док десни граф има један наглашен чвор који повезује велики број периферних чворова. У другом случају један ред CSR матрице може бити много дужи од осталих, што утиче и на уравнотеженост оптерећења и на начин на који се подаци читају из меморије.

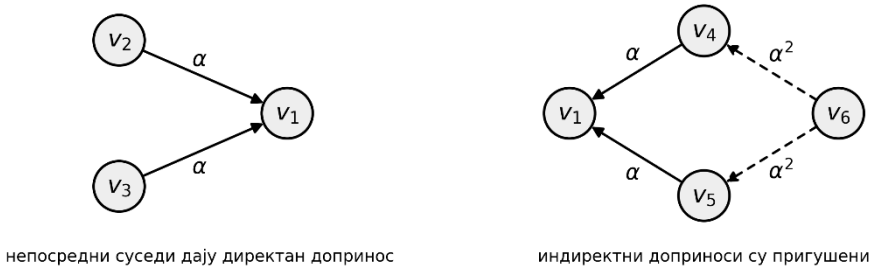


Слика 2.1 - *Различите структуре ретких графова и последице по дужину редова*

2.3. Мере централности у графовима

Мере централности служе да се чворовима додели нумеричка вредност која описује њихов значај у структури графа [2]. Различите мере одговарају на различита питања. Степен чвора мери број непосредних веза и зато је локална мера. Посредничка централност посматра колико се често чвор налази на најкраћим путањама између других чворова; брзи алгоритми за њено рачунање на великим графовима дати су у раду Брандеса [4]. Централност блискости посматра просечну удаљеност чвора од остатка графа. Сопствено-векторска централност и Кацова централност припадају породици мера које вредност чвора везују за вредности његових суседа [3].

Кацова централност проширује идеју да је чвор важан ако је повезан са важним чворовима. За разлику од мере степена, она не посматра само први ниво суседа. Путање дужине два, три и више такође доприносе скору, али њихов утицај опада фактором пригушења. Слика 2.2 приказује основну идеју на примеру чвора v_1, који има непосредне везе, али и посредне везе преко чворова који га повезују са остатком графа.



Слика 2.2 - *Интуиција Кацове централности: директни и индиректни доприноси*

2.4. Кацова централност

Кацова централност додељује чвору већи скор ако до њега из других чворова води више шетњи, при чему допринос опада са дужином шетње. Слична идеја итеративног рангирања чворова по структури веза јавља се и у познатом алгоритму PageRank [6, 7]. У

матричном запису, вредност сваког чвора зависи од вредности његових суседа и од основног доприноса који добија сваки чвор. Зато се Кацова централност може записати као једначина фиксне тачке:

$$x = \alpha A x + \beta \mathbf{1},$$

где је x вектор скорова, A матрица суседства, α фактор пригушења, β основни допринос сваком чвору, а $\mathbf{1}$ вектор јединица. Итеративна реализација коришћена у овом раду гласи:

$$x^{(t+1)} = \alpha A x^{(t)} + \beta \mathbf{1}.$$

Параметар α контролише јачину преноса скорова преко грана графа. Већа вредност α повећава утицај суседа и индиректних путања, док мања вредност јаче пригушује доприносе који долазе преко дужих путања. За конвергенцију класичне формулације потребно је да α буде мање од реципрочне вредности спектралног радијуса матрице A (спектрални радијус $\rho(A)$ је највећа апсолутна вредност карактеристичних вредности матрице A), односно $\alpha < 1/\rho(A)$. У експериментима је коришћено $\alpha = 0.005$, што је намерно мала вредност која изразито пригушује пренос утицаја и смањује ризик од нестабилности итеративног поступка. Параметар β даје сваком чвору позитивну основну вредност и спречава да изоловани или слабо повезани чворови добију нулту вредност.

Конвергенција се прати L1 резидуалом:

$$r^{(t)} = \|x^{(t+1)} - x^{(t)}\|_1 = \Sigma_{i=1}^{n} |x_i^{(t+1)} - x_i^{(t)}|.$$

Рачунање се зауставља када је $r^{(t)} < \varepsilon$ или када се достигне задати максимални број итерација. У мерењима скалирања коришћен је фиксан број од 50 итерација и $\varepsilon = 0$, чиме се мери трошак исте количине посла. У мерењу конвергенције коришћено је $\varepsilon = 10^{-9}$ и максимално 200 итерација.

2.5. Представљање у CSR формату и SpMV

CSR формат чува ретку матрицу по редовима и припада стандардним репрезентацијама ретких матрица у нумеричкој линеарној алгебри [8, 9]. У имплементацијама коришћеним у овом раду чувају се два низа:

- `row_ptr`, дужине n + 1, где `row_ptr[i]` и `row_ptr[i+1]` одређују опсег суседа за ред `i`;
- `col_idx`, дужине једнаке броју записаних грана, где се налазе индекси суседних чворова.

Пошто су вредности грана у овом раду имплицитно једнаке 1, није потребан посебан низ вредности матрице. Сви коришћени графови имају мање од 2^{31} чворова и уписаних грана, па се низови row_ptr и col_idx могу чувати помоћу 32-битних индекса. У односу на

64-битне индексе, ова одлука смањује меморијски саобраћај при обиласку CSR структуре. SpMV за ред *i* рачуна

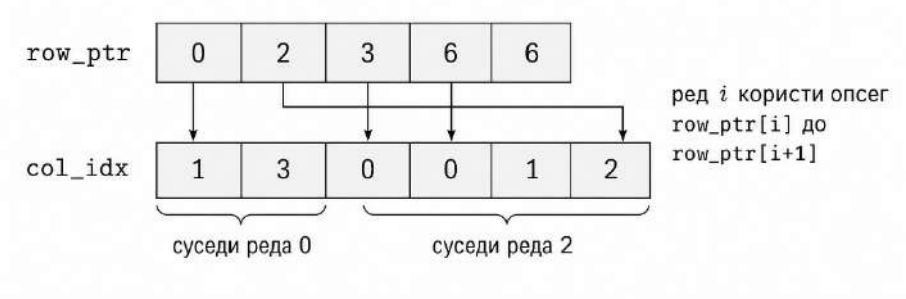
$$y_i = \Sigma_j \in N(i)\ x_j,$$

где је N(i) скуп суседа чвора *i*. Временска сложеност једне итерације је $O(n + m)$, где је m број уписаних грана у CSR структури.

```
for it in 1..iters:
   y = A * x            # SpMV nad CSR grafom
   y = alpha * y + beta   # Katz update
   residual = sum(abs(y - x))
   x = y
   if residual < tol:
      break
```

SpMV је погодан за паралелизацију зато што се различити редови матрице могу обрађивати независно. Ипак, перформансе нису одређене само бројем операција. Код великих ретких графова доминира читање меморије. Низ col_idx обилази се секвенцијално по редовима, док се вектор *x* чита по индексима суседа, често без просторне локалности. Зато се паралелно убрзање може смањити када више нити или GPU радних јединица почне да дели исти меморијски пропусни опсег [10, 11].

Пример распореда низова row_ptr и col_idx у CSR формату приказан је на слици 2.3.



Слика 2.3 - *Пример чувања суседа у CSR формату*

2.6. Меморијски саобраћај и сложеност итерације

Једна итерација Кацове централности има линеарну сложеност у односу на број чворова и грана, односно $O(n + m)$. Ова ознака, међутим, не говори довољно о стварном времену извршавања. У ретким графовима број аритметичких операција по учитаном податку је мали. За сваку грану потребно је прочитати индекс суседа, прочитати одговарајући елемент вектора *x* и додати га у суму. У поређењу са количином података која се чита, аритметички рад је скроман. Због тога је корисно посматрати не само број операција, већ и количину података која пролази кроз меморијски систем [12, 11].

У поједностављеном моделу, свака итерација чита низ `row_ptr`, чита низ `col_idx`, чита елементе претходног вектора скорова и уписује нови вектор. Ако се користе 32-битни индекси, индекси заузимају мање простора него код 64-битног представљања, па се смањује

притисак на кеш меморију и меморијски пропусни опсег. Ова уштеда је нарочито важна код великих графова, где низ индекса грана може бити највећи део радног скупа података.

Други важан фактор за време извршавања јесте локалност приступа меморији. Низ `col_idx` се чита редом, што је повољно за кеш и предучитавање. Насупрот томе, вектор *x* се чита по индексима суседа. Ако су суседи у меморији распоређени близу једни других, приступ је повољнији; ако су распоређени насумично, свака нит може читати удаљене делове меморије. Зато редослед чворова у графу и начин изградње CSR структуре могу утицати на време извршавања, иако се математички алгоритам није променио.

Због тога је у раду додато профилисање итерационог језгра. Укупно време процеса показује колико корисник чека једно покретање програма, док профилно време показује колико траје само понављани део алгоритма. Та разлика је важна за тумачење резултата јер ако припрема графа доминира, онда оптимизација SpMV језгра не може значајно смањити укупно време кратких покретања. Ако се исти CSR граф користи кроз велики број итерација или у више покретања, оптимизација итерационог језгра постаје знатно значајнија.

2.7. Јако и слабо скалирање

Јако скалирање мери понашање за фиксну величину проблема док се повећава број процесора *p,* односно број радних нити или процеса у конкурентној имплементацији. Убрзање се рачуна по формули,

$$S(p) = \frac{T_{seq}(1)}{T_{par}(p)}.$$

где је $T_{seq}(1)$ време секвенцијалне имплементације, а $T_{par}(p)$ време паралелне имплементације са *p* нити или процеса. Ефикасност се рачуна као,

$$E(p) = \frac{S(p)}{p}.$$

Идеално понашање при јаком скалирању имало би $S(p) = p$, али се у пракси јављају серијски део програма, синхронизација, додатни трошкови расподеле посла и засићење меморијског система.

Слабо скалирање мери понашање када се величина проблема повећава заједно са бројем ресурса. У овом раду синтетички графови имају N_p чворова и приближно константан просечан степен. За анализу су коришћена два начина приказа резултата. Први приказује апсолутно време извршавања како проблем расте. Други пореди секвенцијалну и паралелну имплементацију на истој величини графа:

$$S_G(p) = \frac{T_{seq}(N_p)}{T_{par}(N_p)}.$$

Овај однос је коришћен при процени параметара Густафсоновог закона.

2.8. Амдалов и Густафсонов закон

Амдалов закон описује горњу границу убрзања за фиксан проблем. Ако је *s* серијски део програма, очекивано убрзање за *p* ресурса је

$$S_A(p) = \frac{1}{\left(s + \frac{1 - s}{p}\right)}.$$

Чак и мали серијски део може постати доминантан када се број ресурса повећава. У раду се *s* процењује из измереног убрзања при јаком скалирању методом најмањих квадрата.

Густафсонов закон полази од претпоставке да се величина проблема може повећавати са бројем ресурса. Облик који се користи у овом раду је

$$S_G(p) = p - s(p - 1) = (1 - s)p + s.$$

Процењена вредност *s* овде не означава само део програма који је заиста секвенцијалан, већ и све ефекте који спречавају скалирање, попут меморијског пропусног опсега, трошка покретања процеса или кернела, синхронизације и небалансираног оптерећења.

2.9. Рад, распон и теоријски паралелизам

Поред Амдаловог и Густафсоновог закона, за анализу паралелних алгоритама коришћен је и модел рада и распона [25]. Рад W, односно T_1, представља укупну количину посла коју алгоритам извршава на једном процесору. Распон D, односно T_∞, представља дужину најдужег ланца зависности, то јест најкраће могуће време извршавања када би био доступан неограничен број процесора. Однос W/D назива се теоријски паралелизам алгоритма и представља горњу границу корисне количине паралелног извршавања. У једној итерацији Кацове централности над CSR графом обрађује се сваки ред матрице и свака уписана грана. За ред i читају се границе row_ptr[i] и row_ptr[i+1], затим се пролази кроз суседе из низа col_idx и сабирају се одговарајуће вредности из претходног вектора скорова x. После тога се израчунава нова вредност $y_i = \alpha s + \beta$ и локални допринос резидуалу $|y_i - x_i|$. Због тога је рад једне итерације $W = \Theta(n + m)$, где је n број чворова, а m број уписаних грана у CSR структури. Редови CSR матрице могу се обрађивати независно, јер сваки ред уписује тачно једну позицију излазног вектора y, док се претходни вектор x само чита. Једина глобална зависност у оквиру итерације јавља се при сабирању резидуала. Ако је d_i степен чвора i, а $\Delta = \max_i d_i$ максимална дужина реда, онда се распон овакве паралелизације може приближно моделовати као $D = \Theta(\Delta + \log n)$. Члан Δ потиче од чињенице да се у овој имплементацији суседи једног реда сабирају секвенцијално. Члан $\log n$ представља висину графа рачунања који се користи за редукцију локалних доприноса резидуалу. Теоријски паралелизам једне итерације зато се може грубо изразити као $W / D = \Theta((n + m) / (\Delta + \log$

n)). Овај израз показује да алгоритам има висок теоријски паралелизам када граф има много редова и грана, а максимални степен није превелик у односу на величину графа. Међутим, висок теоријски паралелизам не гарантује линеарно убрзање у стварној имплементацији. Код ретких графова свака грана захтева читање индекса суседа и читање вредности из вектора x, често без добре просторне локалности. Зато се у пракси брзо јављају ограничења меморијског пропусног опсега, кеш промашаји, небалансирана расподела редова и трошкови редукције. У овом раду овај модел је посебно важан за тумачење TBB имплементације. TBB може да расподели независне опсеге редова на више радних нити и да аутоматски изврши редукцију резидуала. То одговара теоријском моделу у коме је већина рада паралелна. Ипак, пошто све нити читају исте велике структуре података и приступају вектору x по индексима суседа, стварно убрзање може бити знатно мање од теоријског паралелизма W/D. Због тога се у резултатима не посматра само број извршних нити, већ и ефективни меморијски проток и разлика између укупног времена процеса и времена итерационог језгра.

За поређење са експерименталним резултатима користи се и поједностављена процена времена извршавања на p процесора. Према моделу рада и распона, време извршавања може се грубо проценити као $T_p \approx W/p + D$, па је теоријско убрзање $S_p \approx W / T_p = W / (W/p + D)$. Ова процена представља оптимистичну горњу границу очекиваног убрзања, јер узима у обзир зависности у алгоритму, али не моделира меморијски пропусни опсег, кеш промашаје, трошкове распоређивања посла, синхронизацију и остале ефекте реалне имплементације.

2.10. Коришћени модели паралелизације

Python имплементација користи модул `multiprocessing` и дељену меморију. Овај приступ заобилази ограничење глобалног закључавања интерпретера тако што се рад дели на више процеса. Rust имплементација користи Rayon и паралелне итераторе [24]. C++/TBB имплементација користи oneTBB `parallel_reduce` над блоковима редова, tbb::global_control за контролу броја радних нити и, у посебној варијанти, affinity_partitioner ради очувања локалности расподеле посла између итерација [23]. OpenMP имплементација користи директиве `#pragma omp parallel for` и редукцију [20].

GPU имплементације користе CUDA [21] и OpenCL [22]; у обе GPU имплементације CSR и вектори преносе се на уређај, множење ретке матрице вектором спаја се са ажурирањем вектора, а копирање између итерација замењено је заменом бафера.

2.11. Паралелизација по редовима

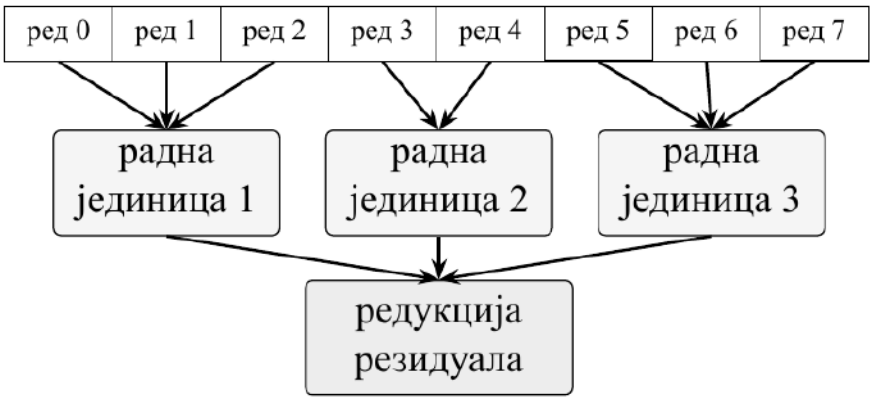
Основна јединица рада у имплементацијама је ред CSR матрице, односно чвор графа. Вредност y_i у новој итерацији зависи од реда i, од низа суседа тог реда и од претходног

вектора x. Због тога се више редова може обрађивати паралелно без међусобног уписа у исту позицију излазног вектора. На крају итерације потребно је само сабрати локалне доприносе резидуалу и заменити улоге вектора *x* и y.

Оваква подела је једноставна и преносива између коришћених модела паралелизације, али није увек савршено избалансирана. Ако су редови сличне дужине, статичка подела на блокове даје добре резултате јер свака радна јединица добија приближно исту количину посла. Ако граф има чворове високог степена, један блок може садржати много више грана од другог. Тада време итерације одређује најспорија радна јединица, чак и када су остале већ завршиле свој део.

У CPU имплементацијама овај проблем се делимично ублажава библиотечким распоређивачима и редукцијама. У GPU имплементацијама у овом раду коришћен је једноставан модел у којем једна нит обрађује један ред. Тај модел је једноставан и погодан за поређење, али код редова високог степена може оставити део паралелизма неискоришћеним. Зато је у закључку као даљи правац издвојена адаптивна расподела редова, где би један дугачак ред могао да се подели на више нити или групу нити.

Описана расподела редова CSR матрице на радне јединице шематски је приказана на слици 2.4.



Слика 2.4 - *Расподела редова CSR матрице на радне јединице*

3. Концепт решења

3.1. Пројектни захтеви и полазне одлуке

Решење је пројектовано тако да се различити модели паралелног извршавања могу поредити у што равноправнијим условима. Због тога није било довољно написати више независних програма који рачунају Кацову централност. Свака имплементација морала је да користи исту математичку формулацију, исти критеријум заустављања, исти улазни формат и исти формат представљања графа. На тај начин се разлике у резултатима могу приписати извршном моделу, меморијском распореду и трошковима паралелизације, а не разликама у самом алгоритму.

Полазне одлуке су биле следеће:

- граф се задаје као листа грана, јер је то природан формат јавних скупова података;
- чворови се пресликавају у компактан опсег индекса, како би се вектори могли чувати као једнодимензионални низови;
- граф се у свим имплементацијама после учитавања чува у CSR облику, јер је то компактан формат погодан за пролаз по редовима;
- свака итерација рачуна исти израз $x^{(t+1)} = \alpha A x^{(t)} + \beta \mathbf{1}$;
- свака имплементација записује исте врсте излаза, како би се коректност и време извршавања могли обрадити заједничким скриптама.

Овакво уједначавање донекле ограничава могућност да се сваки модел паралелизације оптимизује до краја. На пример, GPU имплементације би могле да користе сложеније расподеле рада по чворовима високог степена, а CPU имплементације би могле да користе посебне библиотеке за ретке матрице [10, 13]. Међутим, за циљ овог рада важнија је упоредивост, где се посматра ефекат промене модела паралелизације када алгоритамска основа остаје иста.

3.2. Заједничка архитектура пројекта

Основни захтев пројекта био је да се различити модели паралелизације пореде над истим рачунским проблемом. Зато су имплементације организоване тако да деле следеће особине:

- исти улазни формат: листа грана са опцијом учитавања графа као неусмереног;
- иста интерна репрезентација: CSR са низовима `row_ptr` и `col_idx`;
- исти параметри командне линије: `--input`, `--out`, `--alpha`, `--beta`, `--tol`, `--iters`, `--topk`, `--undirected`, `--parallel`, `--threads`;

- исти излази: `convergence.csv`, `topk.csv`, `x_final.csv`;
- исти профилни излаз: `profile.csv`, са временом итерације, процењеним бројем бајтова и ефективним меморијским протоком;
- иста дефиниција резидуала и исти критеријум заустављања.

Разлике између имплементација намерно су ограничене на модел паралелизације и начин расподеле рада. Тиме се смањује ризик да се разлике у резултатима припишу паралелизацији, иако потичу од различитог алгоритамског поступка.

Преглед имплементација, коришћених технологија и модела паралелизације дат је у табели 3.1.

Табела 3.1 - *Имплементације коришћене у експериментима*

Ознака	Технологија	Модел паралелизације
Python сек/мп	Python, NumPy	Секвенцијална петља и multiprocessing са дељеном меморијом
Rust сек/пар	Rust, Rayon	Паралелни итератори по редовима CSR матрице и паралелна редукција резидуала
C++ сек/тбб	C++17, oneTBB	parallel_reduce по блоковима редова, редукција резидуала и опциони affinity_partitioner
OpenMP сек/пар	C++17, OpenMP	Директиве parallel for и редукција резидуала
CUDA CPU/GPU	C++17, CUDA	CPU основа и GPU кернели са једном нити по реду CSR матрице
OpenCL CPU/GPU	C++17, OpenCL	CPU основа и OpenCL кернели еквивалентни CUDA решењу

3.3. Јединствени интерфејс имплементација

Да би аутоматизовано мерење било могуће, све имплементације имају исти спољашњи интерфејс. Под интерфејсом се овде подразумева скуп параметара командне линије, називи излазних датотека и значење вредности које се у њима налазе. Овај интерфејс је намерно једноставан, тако да свака извршна датотека добија путању до графа, нумеричке параметре алгоритма, број итерација или толеранцију и место где треба записати резултате.

Заједнички параметри командне линије и излазне датотеке приказани су у табели 3.2.

Табела 3.2 - *Заједнички улази и излази имплементација*

Елемент	Улога у пројекту
--input	путања до листе грана која се учитава у свакој имплементацији
--alpha, --beta	параметри Кацове централности који одређују пригушење и основни допринос
--tol, --iters	услов заустављања: толеранција резидуала или фиксан број итерација
--undirected	захтев да се свака улазна грана третира као неусмерена веза
convergence.csv	ток резидуала и времена по итерацијама, користан за анализу конвергенције
x_final.csv, x_final.npy	финални вектор скорова у текстуалном или бинарном облику
topk.csv	најбоље рангирани чворови када је тражено издвајање првих *K* резултата
profile.csv	профилни запис времена итерационог језгра и процењеног меморијског саобраћаја

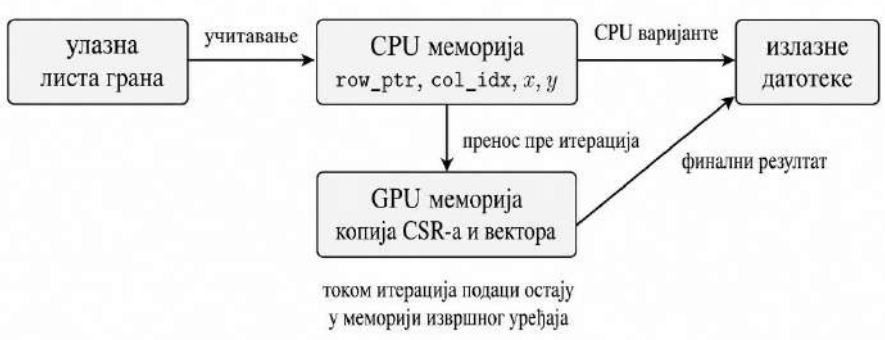
Овако дефинисан интерфејс поједностављује и проверу коректности. Скрипта за обраду резултата не мора да зна у ком је језику реализована имплементација; довољно је да прочита излазне датотеке и упореди финалне векторе. На исти начин се формирају и

графици. Подаци се најпре претварају у заједнички резиме, а затим се из њега генеришу табеле и слике које се укључују у рад.

3.4. Власништво над подацима и пренос података

Једна од важних концептуалних разлика између CPU и GPU имплементација јесте локација главних низова током итерација. У CPU имплементацијама CSR структура и вектори скорова остају у радној меморији процеса. Нити или процеси заједнички читају низ col_idx и претходни вектор x, док свака радна јединица уписује свој део новог вектора *y*. Код GPU имплементација подаци се после изградње CSR структуре преносе у меморију уређаја, а затим остају тамо кроз све итерације.

Ова разлика је приказана на слици 3.1. Код GPU приступа највећи почетни трошак представља пребацивање CSR структуре и почетних вектора у меморију уређаја. Ако би се вектор после сваке итерације враћао у меморију хоста, предност паралелног извршавања брзо би нестала. Зато се у CUDA и OpenCL имплементацијама између итерација мењају показивачи или меморијски бафери, док се на хост враћају само финални резултати или мањи помоћни подаци.

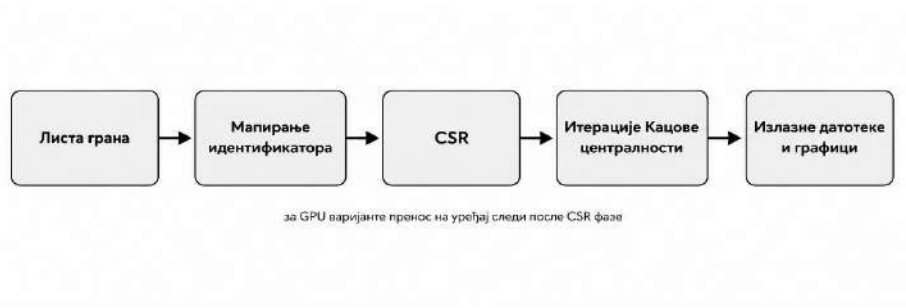


Слика 3.1 - *Власништво над подацима у CPU и GPU имплементацијама*

3.5. Ток обраде

Обрада се састоји од четири фазе. Прво се листа грана чита из датотеке и оригинални идентификатори чворова пресликавају у густ опсег [0, *n*). Затим се гране сортирају по изворишном чвору и формира CSR структура. Трећа фаза је извршавање итерација Кацове централности. У последњој фази записују се резултати и, ако је тражено, издваја се првих *K* чворова према финалним скоровима.

Овај заједнички ток обраде приказан је на слици 3.2.



Слика 3.2 - *Концептуални ток обраде у свим имплементацијама*

Код GPU имплементација додаје се посебна фаза преноса CSR структуре и вектора на уређај. Након тога подаци остају на уређају кроз све итерације. Тиме се избегава поновни пренос великих низова између хоста и уређаја. Изузетак су издвајање и упис најбољих K резултата, када је потребно вратити део вектора у меморију хоста.

3.6. Методологија мерења

Мерења су аутоматизована Python скриптама benchmark_all.py, benchmark_native.py, benchmark_gpu.py, collect_results.py и paper_plots.py. У свим мерењима коришћени су α = 0.005, β = 1.0, 50 итерација, ε = 0, K = 10, учитавање графа као неусмереног и 30 понављања. Вредности p су {1, 2, 4, 8, 16}. У експериментима јаког скалирања p означава број процеса или нити. GPU имплементације се у табелама јаког скалирања мере једном по графу, јер се не скалирају преко броја CPU нити. У експериментима слабог скалирања p означава и фактор величине синтетичког графа, па се GPU имплементације мере за све величине.

За C++/TBB додатно је покретана варијанта са укљученим affinity_partitioner-ом, за исте вредности p као и основна TBB варијанта. На тај начин се може раздвојити утицај самог TBB распоређивања од утицаја очувања сличне расподеле блокова редова између итерација.

Методологија мерења намерно раздваја два начина посматрања времена извршавања. Први обухвата укупно време које корисник чека од покретања програма до уписа резултата. Други обухвата само време итерационог језгра, након што је граф већ претворен у CSR формат. Ово раздвајање је важно зато што су у овом раду покретања кратка и свако покретање поново чита и припрема граф. У дуготрајној анализи, где би се исти CSR граф користио за више различитих параметара или више централности, релативни значај припреме био би мањи, а значај меморијског саобраћаја у итерацијама већи. Овај начин размишљања одговара анализи заснованој на уским грлима меморијског протока [11].

Разлика између укупног времена процеса и времена итерационог језгра приказана је на слици 3.3.

Слика 3.3 - *Разлика између укупног времена процеса и профилног времена језгра*

Резултати појединачних покретања чувају се у датотекама у табеларном облику и колонама `mode`, `graph`, `lang`, `variant`, `p`, `run_id`, `wall_ms`, `exit_code`, `out_dir`, `host` и `os`. Вредност `wall_ms` представља укупно време процеса, па укључује учитавање листе грана, пресликавање идентификатора, сортирање и изградњу CSR структуре. Да се трошак самог итерационог језгра не би помешао са трошком припреме графа, свако покретање додатно записује `profile.csv`. Након мерења врши се спајање резултата и рачунање средње вредности, стандардне девијације, медијане, минимума и максимума. За стабилнију процену просечних вредности примењује се правило интерквартилног опсега за уклањање одступајућих мерења.

3.7. Провера коректности

За проверу коректности користи се граф email-Eu-core. Свака од тринаест имплементација извршава се до толеранције $\varepsilon = 10^{-9}$, а финални вектор скорова пореди се са C++ секвенцијалном основом. Пошто различити редоследи сабирања могу довести до разлика на нивоу последњих битова у представи бројева у покретном зарезу, пореде се L_1 и $L\infty$ разлике, где је L_1 разлика збир апсолутних вредности разлика појединачних компоненти два вектора, а $L\infty$ разлика највећа апсолутна вредност разлике по компонентама. Имплементација се сматра исправном ако су добијене разлике у границама очекиване нумеричке прецизности.

4. Програмско решење

4.1. Организација изворног кода

Пројекат је организован према моделима паралелизације. Директоријум `python/` садржи секвенцијалну Python имплементацију, имплементацију са вишепроцесном обрадом, помоћне функције за читање графа и генератор синтетичких графова. Директоријум `rust/` садржи Rust имплементацију са модулима за CSR, улаз графа и Кацову централност. Директоријуми `cpp/`, `openmp/`, `cuda/` и `opencl/` садрже одговарајуће C++ имплементације и CMake конфигурације. Резултати мерења и готови графикони налазе се у директоријуму `results/`.

Поред поделе по моделима паралелизације, репозиторијум је организован и према улогама: имплементације, скрипте за мерење, скупови података и обрађени резултати. Ова подела је важна јер се завршни рад не ослања само на једну извршну датотеку, већ на читав поступак изградње извршних датотека, покретања експеримената, спајања резултата, провере коректности и генерисања графика.

Главне улоге директоријума и скрипти у пројекту сажете су у табели 4.1.

Табела 4.1 - *Улоге главних директоријума и скрипти у пројекту*

Путања	Улога
python/	Python имплементације, читачи графова и генератор синтетичких графова
rust/	Rust секвенцијална и паралелна имплементација са Rayon распоређивањем
cpp/	C++/TBB решење и основна C++ инфраструктура за CSR и мерење
openmp/	OpenMP имплементација заснована на паралелној петљи и редукцији
cuda/	CUDA извршна датотека са CPU основом и GPU кернелима
opencl/	OpenCL извршна датотека са избором платформе и уређаја
results/	сирови резултати, резимеи, профилни записи и графици за рад
benchmark_*.py	скрипте за покретање серија мерења
collect_results.py	спајање резултата, израчунавање резимеа и провера коректности

Оваква организација омогућава да се свака имплементација развија независно, али да се њени резултати на крају уклопе у исти експериментални ток, чиме се олакшава изолација грешака. Ако једна имплементација даје другачији финални вектор, проблем се тражи у њеном учитавању, CSR конструкцији или итерационом језгру, а не у накнадној обради резултата.

4.2. Учитавање графа и CSR конструкција

Све имплементације при учитавању прескачу коментаре у улазним SNAP датотекама и прихватају размаке или зарезе као сепараторе. Пошто јавни скупови података често користе оригиналне идентификаторе који не чине компактан опсег и не морају да почињу од нуле, најпре се прави пресликавање оригиналних идентификатора у компактан опсег. Ово пресликавање смањује меморијске потребе и омогућава директно индексирање низова.

За испис резултата чува се и обрнуто пресликавање, тако да се у `topk.csv` могу записати оригинални идентификатори.

При изградњи CSR структуре гране се сортирају по изворишном чвору, низ row_ptr се попуњава бројањем грана по чвору, а одредишни чворови се затим уписују у col_idx. Код учитавања графа као неусмереног, свака улазна грана *(u, v)* додаје се и као *(v, u)*. Таква одлука је важна за поређење јер исти улазни граф може бити тумачен као усмерен или неусмерен, а резултати централности би у та два случаја били различити. Слични поступци формирања ретких матрица по редовима описују се у литератури о ретким линеарним системима [8, 9].

Поступак изградње може се описати кроз пет корака. Прво се из улазне датотеке издвајају парови чворова и занемарују редови коментара. Друго, сваки оригинални идентификатор добија интерни индекс. Треће, по потреби се додаје обрнута грана. Четврто, гране се сортирају по изворишном индексу, а затим по одредишту, како би редови CSR матрице били компактни. Пето, једним пролазом кроз сортирану листу попуњавају се `row_ptr` и `col_idx`. Исти поступак се понавља у свим имплементацијама, уз разлике у конкретним типовима и библиотечким функцијама.

У завршној верзији кода CSR индекси се чувају као 32-битне целобројне вредности тамо где је то безбедно за величину улазних графова. Ово смањује радни скуп података у односу на 64-битне индексе. Пошто се у SpMV пролазу низ `col_idx` чита за сваку грану, уштеда од четири бајта по индексу директно смањује меморијски саобраћај. Та промена не мења математички резултат, али може утицати на време извршавања, јер је алгоритам у великој мери ограничен читањем података из меморије.

4.3. Итерационо језгро и излаз резултата

Итерационо језгро је заједничка логичка целина свих имплементација. За сваки ред i чита се опсег од row_ptr[i] до row_ptr[i+1], сабирају се вредности претходног вектора x за суседне чворове и затим се добија нова вредност $y_i = \alpha \cdot \Sigma_{j \in N(i)} x_j + \beta$. После тога се рачуна локални допринос резидуалу $|y_i - x_i|$. На крају итерације сабирају се доприноси резидуалу, проверава се услов заустављања, а вектори x и y мењају улоге.

Вектор x се не сме мењати док се рачунају вредности вектора y. Када би се резултати уписивали преко старог вектора, каснији редови би могли да читају већ ажуриране вредности, што би променило алгоритам. Зато све имплементације користе два вектора. Код CPU имплементација то су два низа у радној меморији, а код GPU имплементација два бафера на уређају. Замена улога на крају итерације је јефтинија од копирања садржаја једног вектора у други.

Излаз резултата је подељен на неколико датотека. `x_final.csv` је погодан за ручну проверу и поређење мањих графова, док је `x_final.npy` компактнији за имплементације које природно користе NumPy формат. У датотеку `topk.csv` уписују се најбоље рангирани чворови и враћа оригиналне идентификаторе кад је обрнуто пресликавање доступно. Датотека `convergence.csv` омогућава да се касније нацрта ток резидуала, без поновног покретања алгоритма.

4.4. Python имплементација

Python секвенцијална имплементација, python/katz_seq.py, користи нумеричке низове библиотеке NumPy за векторе x и y, али саму SpMV петљу извршава експлицитно по CSR редовима. Ова имплементација служи као једноставна основа и као референца за трошак извршавања помоћу интерпретера.

Вишепроцесна имплементација, python/katz_mp.py, дели редове CSR матрице на блокове. Сваки блок обрађује један процес. Вектори x и y смештају се у shared_memory, чиме се избегава копирање великих низова у сваком кораку. Радни процеси рачунају само SpMV за своје редове, док главни процес примењује трансформацију $y \leftarrow \alpha y + \beta$, рачуна резидуал и замењује улоге вектора. Ово решење смањује комуникацију, али уводи трошак покретања процеса, мапирања дељене меморије и синхронизације по итерацији.

Python имплементација је најпрегледнија и зато служи за проверу алгоритамске идеје. Његова слабост је што петља по гранама остаје у коду који се извршава преко интерпретера, па сваки приступ суседу носи много већи трошак него у осталим имплементацијама. Вишепроцесна обрада смањује релативно време зато што дели рад на више процеса, али не уклања у потпуности трошак Python окружења. Због тога се у резултатима види велико убрзање у односу на секвенцијалну Python основу, али не и најкраће апсолутно време извршавања.

Посебна пажња је посвећена дељеној меморији. Без ње би сваки процес морао да добије сопствену копију великих вектора, што би повећало меморијску потрошњу и време припреме. Уместо тога, главни процес формира бафере, а радни процеси их читају и тумаче као NumPy низове. На крају рада бафери се затварају и ослобађају, како би се избегло задржавање системских ресурса после завршетка програма.

4.5. Rust имплементација

Rust имплементација се налази у rust/src/katz.rs. Секвенцијална имплементација користи једноставне и компактне петље над CSR опсезима и векторима типа Vec<f64>. Паралелна имплементација користи Rayon скуп радних нити и par_iter_mut за обраду редова. Свака нит уписује у засебан елемент вектора y, па није потребно закључавање. Резидуал се рачуна паралелном редукцијом над паровима елемената x и y.

Rust је у експериментима дао најбржу секвенцијалну основу. То се може објаснити употребом једноставног меморијског распореда и одсуством слојева апстракције који би отежали оптимизацију петљи.

Једна од предности Rust имплементације је што језик на нивоу компајлирања спречава истовремени променљиви приступ истом делу меморије. Ова особина се добро уклапа у поделу рада у овом алгоритму, јер се претходни вектор *x* само чита, а сваки елемент новог вектора *y* има тачно једног власника у току итерације. Rayon тај образац претвара у паралелан пролаз без ручног управљања нитима. Тако код остаје релативно близак секвенцијалној верзији, али добија паралелно извршавање када је то исплативо.

Улазни и излазни слој у Rust-у прати исти интерфејс као остале имплементације. Због тога се и Rust резултати обрађују истим поступком као резултати осталих имплементација.

4.6. C++/TBB имплементација

C++/TBB имплементација налази се у cpp/. За паралелни пролаз користи се tbb::parallel_reduce над блокираним опсегом редова CSR матрице. У телу петље за сваки ред се израчунава SpMV сума, нови Кацов скор и локални допринос резидуалу. На крају TBB спаја локалне вредности резидуала помоћу редукције. Број радних нити ограничава се помоћу tbb::global_control, тако да се исти извршни програм може покретати са различитим вредностима p у експериментима јаког скалирања.

Ова имплементација је добар пример библиотечког приступа паралелизацији на дељеној меморији. Програмер описује опсег редова и операцију која се над тим редовима извршава, док TBB распоређивач дели опсег на задатке и додељује их радним нитима. Са становишта модела рада и распона, рад једне итерације остаје $W = \Theta(n + m)$, јер се и даље обилазе сви редови и све гране. Ефективни распон паралелног пролаза мањи је од секвенцијалног рада зато што се независни редови обрађују паралелно, а серијски део у оквиру итерације углавном чине најдужи појединачни ред и редукција резидуала.

У основној TBB варијанти библиотека самостално распоређује блокове редова. У додатној варијанти користи се tbb::affinity_partitioner. Исти објекат partitioner-а чува се између итерација, тако да TBB може, када је то могуће, да сличне блокове редова поново додели истим радним нитима. Циљ ове варијанте није промена математичког алгоритма, већ побољшање локалности приступа подацима и смањење дела трошкова који настају услед кеш промашаја и поновног распоређивања рада.

Са становишта модела рада и распона, TBB имплементација паралелизује обраду по редовима CSR матрице. Један ред представља основну јединицу рада: суседи тог реда сабирају се секвенцијално, док се различити редови могу обрађивати паралелно. Због тога

рад једне итерације остаје $W = \Theta(n + m)$, док распон зависи од најдужег реда CSR матрице и редукције резидуала, односно приближно $D = \Theta(\Delta + \log n)$. Параметар Δ одговара највећој вредности разлике row_ptr[i+1] - row_ptr[i] након изградње CSR структуре. affinity_partitioner не мења овај теоријски рад и распон, али може да утиче на реално време извршавања очувањем локалности распореда блокова редова између итерација.

Ипак, ни parallel_reduce ни affinity_partitioner не уклањају основно ограничење алгоритма. У свакој итерацији велики део времена троши се на читање низа col_idx и вредности из вектора x. Пошто приступ вектору x зависи од структуре графа, читања често нису просторно локална. Због тога стварно убрзање може бити знатно мање од теоријског паралелизма W/D. Управо зато се у експериментима пореди основна TBB варијанта са варијантом која користи affinity_partitioner, а резултати се тумаче кроз убрзање, стабилност мерења и ефективни меморијски проток.

4.7. OpenMP имплементација

OpenMP имплементација налази се у `openmp/`. SpMV, ажурирање скора и резидуал рачунају се у истој паралелној петљи:

```
#pragma omp parallel for schedule(dynamic, 256) reduction(+:residual)
for (int64_t i = 0; i < csr.n; ++i) {
   double s = 0.0;
   for (int64_t j = row_ptr[i]; j < row_ptr[i + 1]; ++j) {
      s += x[col_idx[j]];
   }
   double next = alpha * s + beta;
   y[i] = next;
   residual += fabs(next - x[i]);
}
```

Динамичко распоређивање са блоком од 256 редова изабрано је да би се ублажила неравномерна расподела степена чворова код графова као што је `com-amazon.ungraph`. На путној мрежи, где су степени регуларнији али су путање кроз меморију велике, ограничење чешће долази од меморијског пропусног опсега него од расподеле редова.

Предност OpenMP имплементације је мала промена у односу на секвенцијални C++ код. Довољно је додати директиву петљи и назначити редукцију резидуала. То је корисно у пракси јер омогућава постепено увођење паралелизма у постојећи код. Са друге стране, једноставност директиве не значи да су добијене перформансе увек оптималне. Избор распореда, величина блока и трошак синхронизације могу значајно утицати на резултат, па је OpenMP имплементација у раду коришћена као пример експлицитног модела паралелизације вишег нивоа.

У овој имплементацији је посебно важно што су ажурирање y и рачунање резидуала спојени у исти пролаз. Да су раздвојени, програм би морао још једном да чита векторе x и y, што би повећало меморијски саобраћај без промене математичког резултата.

4.8. CUDA имплементација

CUDA имплементација налази се у cuda/. CPU секвенцијална основа је укључена у исту извршну датотеку, док се GPU режим бира опцијом --parallel. CSR низови и вектори преносе се на GPU пре итерација. Кернел користи једну CUDA нит по реду CSR матрице и непосредно рачуна $y_i = \alpha \cdot \Sigma_{j \in N(i)} x_j + \beta$. Овај основни образац често се користи као полазни модел за SpMV на графичким процесорским јединицама, иако сложенији формати могу боље искористити хардвер [10]. Резидуал се рачуна помоћу редукције, а на крају итерације мењају се показивачи на векторе x и y, уместо копирања целог вектора.

Предност овог приступа је једноставно и директно пресликавање редова на нити. Недостатак је што редови имају различите дужине, па радне нити не морају завршити посао у исто време. За веће графове трошак преноса и покретања кернела постаје мање значајан у односу на укупан рад, док је за мале графове GPU често спорији од CPU основе.

CUDA део програма има две јасне целине: хост код, који припрема податке и покреће кернеле, и уређајски код, који извршава пролаз по редовима. Код на хосту задужен је за алокацију меморије уређаја, копирање CSR низова, постављање величине мреже блокова и нити и читање финалног резултата. Код на уређају нема знање о улазној датотеци нити о оригиналним идентификаторима чворова; он добија већ припремљене компактне низове.

Ова подела чини CUDA имплементацију добром за анализу трошка преноса података. Ако се посматра укупно време процеса, учитавање графа и пренос на уређај могу сакрити време самог кернела. Ако се посматра датотека `profile.csv`, издваја се само трошак поновљених итерација. Зато су у раду приказана оба мерења.

4.9. OpenCL имплементација

OpenCL имплементација, у `opencl/`, прати исти алгоритамски распоред као CUDA имплементација. Изворни код кернела је уграђен у C++ заглавље `opencl/ocl_kernels.hpp`. Користе се спојени кернел за SpMV и ажурирање вектора, рачунање резидуала по радним групама и замена `cl_mem` бафера између итерација. OpenCL имплементација додатно омогућава избор платформе и уређаја параметрима `--platform` и `--device`.

OpenCL је користан за поређење јер представља преносиви модел хетерогеног извршавања. Ипак, преносивост долази уз већу сложеност кода на хосту, јер је потребно експлицитно изабрати платформу, уређај, контекст, ред команди, превести OpenCL програм и управљати меморијским објектима.

У поређењу са CUDA имплементацијом, OpenCL код има више иницијализационих корака, али је алгоритамски истоветан. Таква одлука је донета како се једна GPU имплементација не би фаворизовала другачијим алгоритмом, већ да се упореде два практична начина покретања истог обрасца на уређају. Зато су и ограничења иста. Једна радна ставка обрађује један ред, дуги редови могу довести до неуједначеног рада, а добитак зависи од односа трошка преноса, покретања кернела и стварне количине обраде.

4.10. Профилни запис у извршним датотекама

Имплементације бележе и профилни запис `profile.csv`. За свако покретање чува се просечно време итерационог језгра, број обрађених итерација, процењени број прочитаних и уписаних бајтова и ефективни меморијски проток. Овај запис је уведен зато

што вредност `wall_ms` није довољна да сама објасни понашање програма. Укупно време може бити велико због припреме графа, док само итерационо језгро може бити веома брзо.

Процена меморијског саобраћаја не представља мерење хардверског бројача, већ аналитичку процену на основу величина низова и броја приступа у једној итерацији. Она је ипак корисна јер омогућава поређење имплементација истог алгоритма. Ако две имплементације обрађују исти граф и исти број итерација, а једна постиже већу ефективну пропусност, онда боље користи расположиви меморијски систем или има мање додатних трошкова у језгру. У том смислу, овај запис има сличну улогу као практични модели перформанси који време извршавања објашњавају доступним меморијским протоком [12, 11].

4.11. Аутоматизација експеримената

Мерења су покретана и обрађивана Python скриптама наведеним у §3.6. Допунска скрипта benchmark_native.py покреће само компајлиране имплементације (C++/TBB, OpenMP, CUDA, OpenCL) и коришћена је за проширење експеримената након првобитних мерења. Аутоматизација обезбеђује поновљивост, јер свако мерење оставља траг у излазној датотеци, а све табеле и графици настају из истих резимираних података.

5. Испитивање и резултати

5.1. Експериментално окружење

Експерименти су извршени на Виндовс радној машини приказаној у табели 5.1. Подаци о хардверу су прикупљени на истој машини на којој се налази пројекат и на којој су генерисани резултати.

Табела 5.1 - *Хардверско и софтверско окружење*

Компонента	Вредност
CPU	AMD Ryzen 7 7700X, 8 језгара / 16 логичких нити, максимално 4.5 GHz
PAM	64 GB
GPU	NVIDIA GeForce GTX 1060 3 GB; интегрисана AMD Radeon графика присутна у систему
Оперативни систем	Microsoft Windows 11 Pro, верзија 10.0.26200
Python	3.13.7, NumPy 2.4.1
Rust	rustc 1.93.0, cargo 1.93.0
CMake	4.1.0-rc4
CUDA	CUDA алати 12.9, nvcc 12.9.86

5.2. Скупови података

За мерења су коришћена три јавна SNAP графа и пет синтетичких графова. SNAP скупови су преузети из Станфордове збирке великих мрежа [14]. Појединачне странице скупова дају статистике за `email-Eu-core`, `roadNet-CA` и `com-Amazon` [15, 16, 17]. Синтетички графови генерисани су локалном скриптом `python/gen_synth.py` са фиксним семеном, тако да се експерименти могу поновити.

Основне карактеристике и намена коришћених скупова података приказане су у табели 5.2.

Табела 5.2 - *Скупови података коришћени у раду*

Скуп	Чворови	Гране	Тип	Намена
email-Eu-core	1005	25571	комуникациона мрежа	конвергенција и коректност
roadNet-CA	1965206	5533214	путна мрежа	јако скалирање на ретком регуларнијем графу
com-amazon.ungraph	334863	925872	мрежа заједничке куповине	јако скалирање на нерегуларнијем графу
weak_p1–weak_p16	20000–320000	око 160000–2560000	случајни граф, степен 16	слабо скалирање

Граф `email-Eu-core` је довољно мали да омогући брзо поређење свих имплементација до конвергенције. Граф `roadNet-CA` је велики и веома редак граф, са структуром путне мреже. Граф `com-amazon.ungraph` је мањи по броју чворова, али има другачију расподелу степена и погодан је за посматрање понашања на нерегуларнијим мрежама.

5.3. Параметри експеримената

Параметри коришћени у експериментима сажети су у табели 5.3.

Табела 5.3 - *Параметри извршавања*

Експеримент	Параметри
Јако скалирање	$\alpha = 0.005$, $\beta = 1.0$, 50 итерација, $\varepsilon = 0$, $p \in \{1, 2, 4, 8, 16\}$, неусмерено учитавање, 30 понављања, $K = 10$
Слабо скалирање	исти нумерички параметри; синтетички графови величине N_p, просечан степен 16
Конвергенција	$\alpha = 0.005$, $\beta = 1.0$, $\varepsilon = 10^{-9}$, највише 200 итерација, $p = 1$, неусмерено учитавање

5.4. Коректност и конвергенција

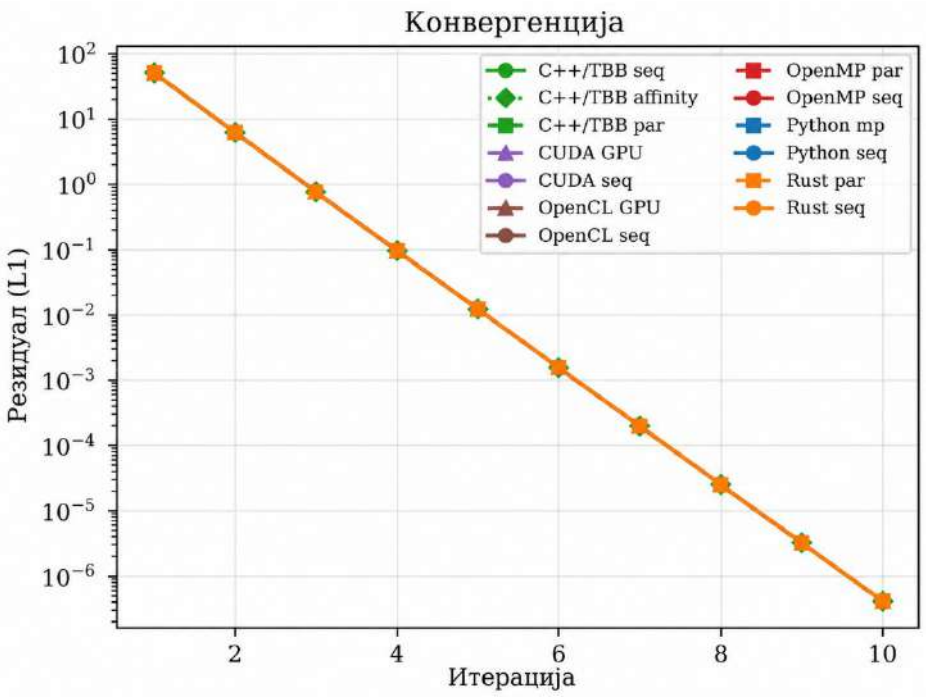
Табела 5.4 приказује највеће разлике у односу на C++ секвенцијалну основу. Све имплементације пролазе проверу. CPU имплементације у Rust-у, основној C++/TBB варијанти, C++/TBB affinity варијанти, OpenMP-у и CPU основама GPU програма дају идентичне резултате. GPU имплементације одступају за око једну јединицу последњег места, а Python имплементације за око две јединице последњег места, што је очекивано због различитог редоследа сабирања у покретном зарезу.

Табела 5.4 - *Коректност финалних вектора скорова на графу* `email-Eu-core`

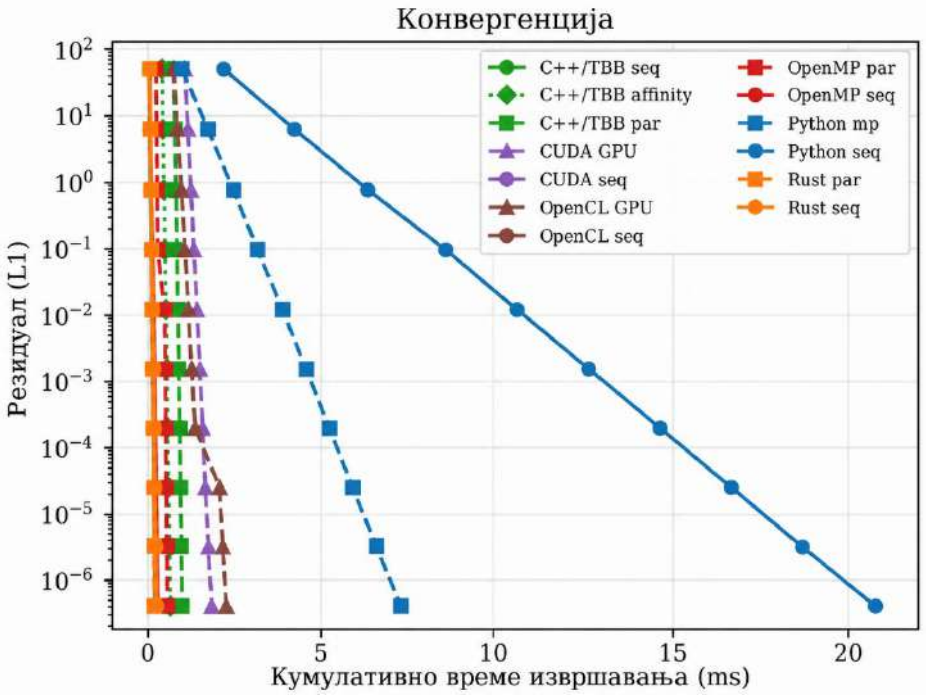
Група поређења	Максимална $L\infty$ разлика	Тумачење
CPU имплементације	0	идентични резултати
CUDA и OpenCL GPU	≈ 1 ULP	приближно једна јединица последњег места
Python сек/мп	≈ 2 ULP	приближно две јединице последњег места

Све имплементације конвергирају у 10 итерација до резидуала $r^{(t)} < 10^{-9}$. Најкраће кумулативно време итерационог дела на овом малом графу постижу Rust паралелна и OpenMP паралелна имплементација, око 0.015 мс према записаним convergence.csv датотекама. Python секвенцијална имплементација је знатно спорија, око 2.091 мс, док Python вишепроцесна обрада достиже 0.704 мс. GPU имплементације су коректне, али је граф сувише мали да би се надокнадио трошак покретања кернела и синхронизације.

Ток конвергенције резидуала по итерацијама и у односу на кумулативно време приказан је на сликама 5.1 и 5.2.



Слика 5.1 - *Конвергенција резидуала кроз итерације на графу* `email-Eu-core`



Слика 5.2 - *Конвергенција резидуала у односу на кумулативно време*

Финална листа од 10 најбољих чворова за референтну C++ секвенцијалну имплементацију почиње чворовима 160, 121, 107, 62 и 86. Иста листа се добија и у осталим имплементацијама, уз нумеричке разлике наведене у табели 5.4.

5.5. Јако скалирање на графу `roadNet-CA`

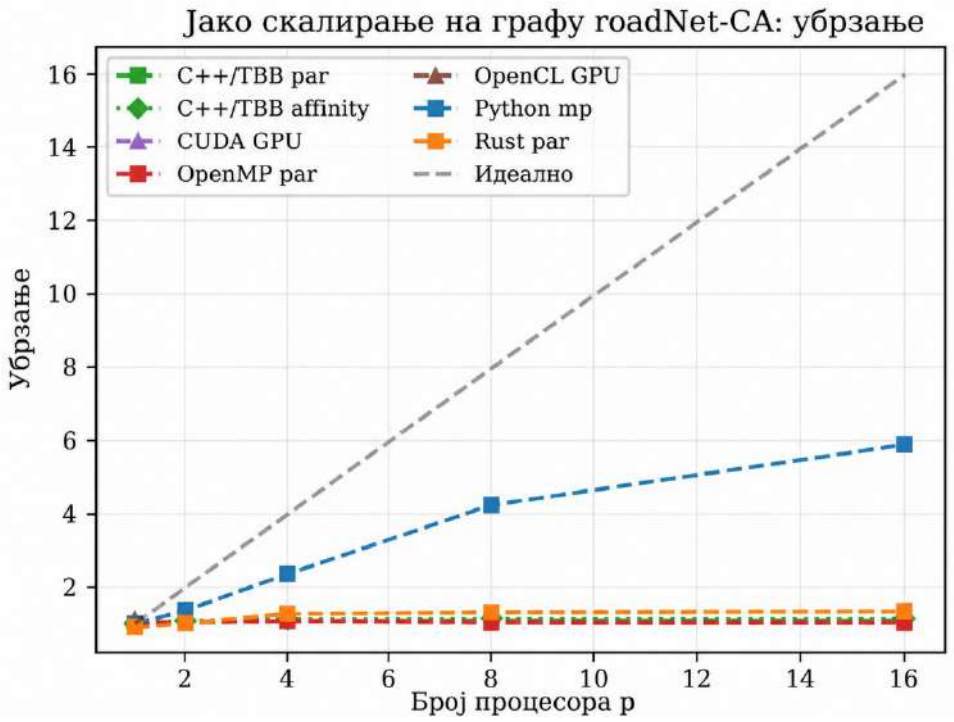
Граф `roadNet-CA` има скоро два милиона чворова и више од пет милиона грана. Због величине графа, апсолутна времена су довољно велика да трошкови покретања програма и самог мерења не доминирају укупним временом. Резултати су приказани у табели 5.5 и на сликама 5.3 и 5.4.

Табела 5.5 - *Најбољи резултати јаког скалирања на графу* `roadNet-CA`

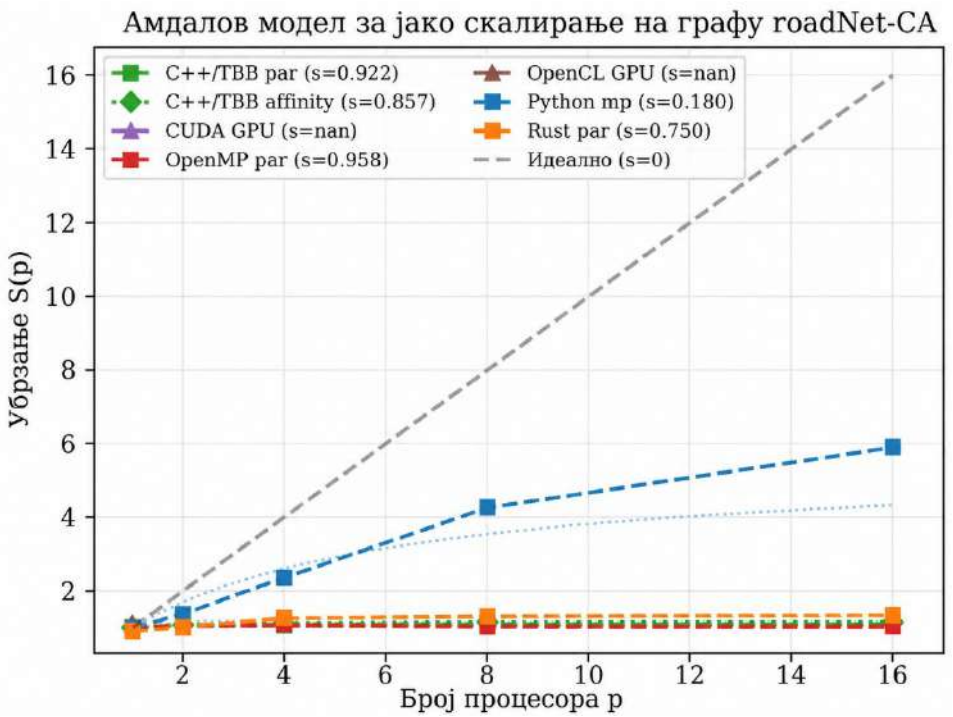
Имплементација	Сек [мс]	Најбоље пар. [мс]	Најбоље p	Убрзање
C++/TBB affinity	10664.2	9264.7	8	1.15
CUDA	7894.2	7317.2	1	1.08
OpenMP	10297.4	9605.3	4	1.07
OpenCL	9758.3	8962.3	1	1.09
Python	203786.3	34587.4	16	5.89
Rust	2458.2	1829.8	16	1.34

Python вишепроцесна обрада остварује највеће релативно убрзање, 5.89× при 16 процеса. То не значи да је Python апсолутно најбржи; напротив, и паралелни Python је знатно спорији од осталих имплементација. Велико релативно убрзање потиче од веома споре секвенцијалне Python основе и од чињенице да се рад успешно дели између процеса.

Rust има најбољу апсолутну CPU основу, са секвенцијалним временом од 2458.2 мс, док паралелна имплементација достиже 1829.8 мс. C++/TBB affinity варијанта побољшава резултат основне TBB варијанте и достиже убрзање 1.15× при p = 8, али и даље не достиже Rust имплементацију. OpenMP остварује убрзање 1.07×, а GPU имплементације су брже од својих CPU основа у истом извршном програму, али не довољно да престигну најбољу CPU имплементацију. На овом графу OpenCL има већу варијансу мерења од CUDA имплементације.



Слика 5.3 - *Убрзање при јаком скалирању на графу* `roadNet-CA`



Слика 5.4 - *Амдалов модел за јако скалирање на графу* `roadNet-CA`

Процењени серијски делови додатно објашњавају резултате. За Python вишепроцесну имплементацију на roadNet-CA добија се s ≈ 0.18, док Rust има s ≈ 0.75. Основна C++/TBB варијанта има s ≈ 0.92, а C++/TBB affinity варијанта s ≈ 0.86, што показује да affinity_partitioner смањује део трошкова који се у Амдаловом моделу појављују као непаралелизабилна компонента. OpenMP има s ≈ 0.96. Високе вредности не значе да је код заиста већински секвенцијалан, већ да се из угла модела убрзање понаша као да постоји велика непаралелизабилна компонента. У овом случају ту компоненту чине меморијски пропусни опсег, расподела посла, синхронизација и трошкови паралелног режима.

5.6. Јако скалирање на графу `com-amazon.ungraph`

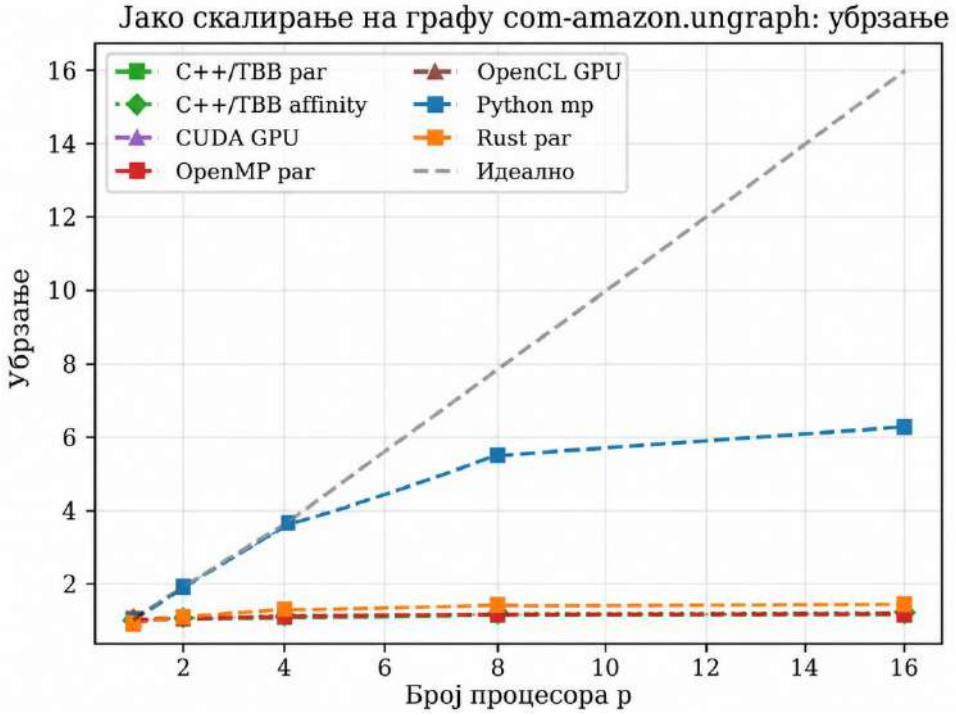
Граф `com-amazon.ungraph` је мањи од `roadNet-CA`, али има другачију структуру. Расподела степена је нерегуларнија и више личи на структуре које се јављају у социјалним мрежама или мрежама препорука. Резултати су приказани у табели 5.6 и на сликама 5.5 и 5.6.

Табела 5.6 - *Најбољи резултати јаког скалирања на графу* `com-amazon.ungraph`

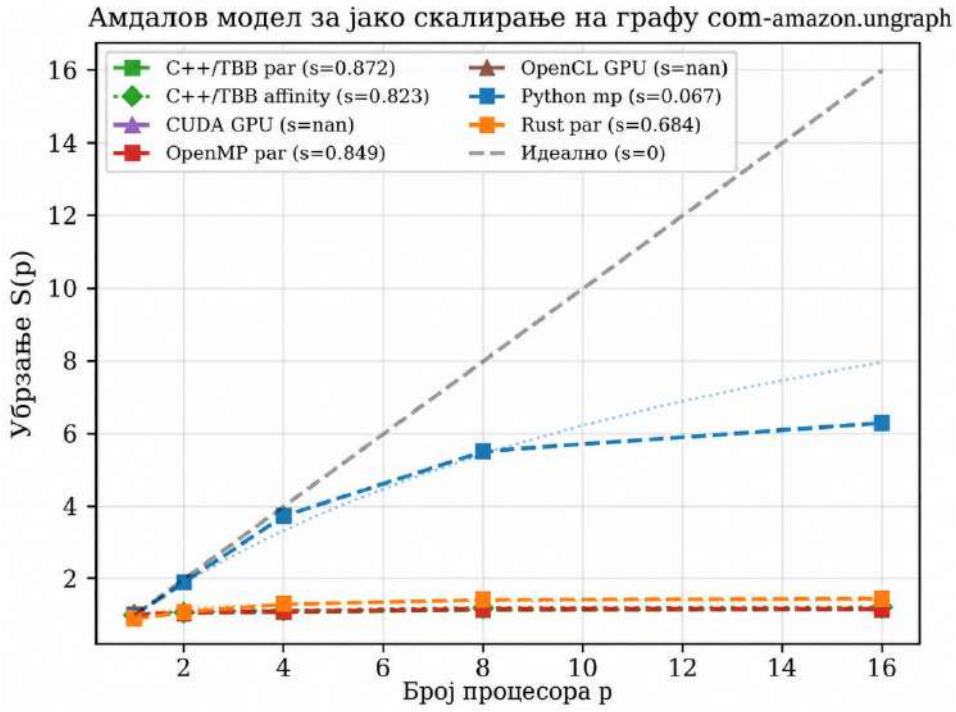
Имплементација	Сек [мс]	Најбоље пар. [мс]	Најбоље p	Убрзање
C++/TBB affinity	1913.1	1571.9	16	1.22
CUDA	1902.5	1886.0	1	1.01
OpenMP	1952.0	1655.3	8	1.18
OpenCL	1915.4	1926.2	1	0.99
Python	55318.2	8795.7	16	6.29
Rust	422.8	289.6	16	1.46

На овом графу све CPU паралелне имплементације скалирају боље него на roadNet-CA. Python вишепроцесна обрада достиже 6.29×, Rust 1.46×, C++/TBB affinity 1.22×, а OpenMP 1.18×. Основна TBB варијанта је слабија од affinity варијанте и достиже 1.15×, што

је детаљније приказано у табели 5.8. Разлика у односу на путну мрежу показује да структура графа утиче на паралелну ефикасност, а не само број чворова и грана.



Слика 5.5 - *Убрзање при јаком скалирању на графу* `com-amazon.ungraph`



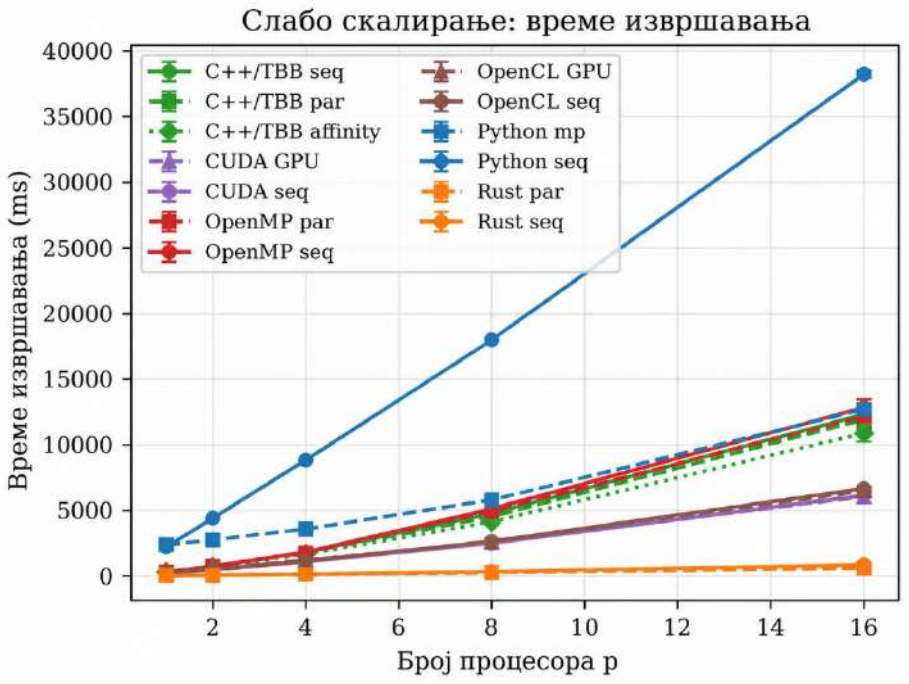
Слика 5.6 - *Амдалов модел за јако скалирање на графу* `com-amazon.ungraph`

Процењени Амдалови серијски делови такође су нижи него на графу roadNet-CA: за Python s ≈ 0.067, за Rust s ≈ 0.684, за основни TBB s ≈ 0.872, за TBB affinity s ≈ 0.823, а за OpenMP s ≈ 0.849. Увођење affinity_partitioner-а поново смањује процењени серијски део TBB варијанте, али не мења основни закључак да је овај SpMV образац ограничен приступом меморији.

Rust остаје апсолутно најбржи, са најбољим паралелним временом од 289.6 мс, док су C++/TBB affinity, OpenMP, CUDA и OpenCL око 1.6–1.9 с. Python је, упркос добром релативном убрзању, и даље апсолутно најспорији. Ова разлика наглашава да релативно убрзање увек треба посматрати заједно са апсолутним временом.

5.7. Слабо скалирање

Експерименти слабог скалирања користе синтетичке графове `weak_p1` до `weak_p16`. Број чворова расте од 20000 до 320000, а просечан степен остаје приближно 16. Слика 5.7 приказује време извршавања кроз величине проблема.



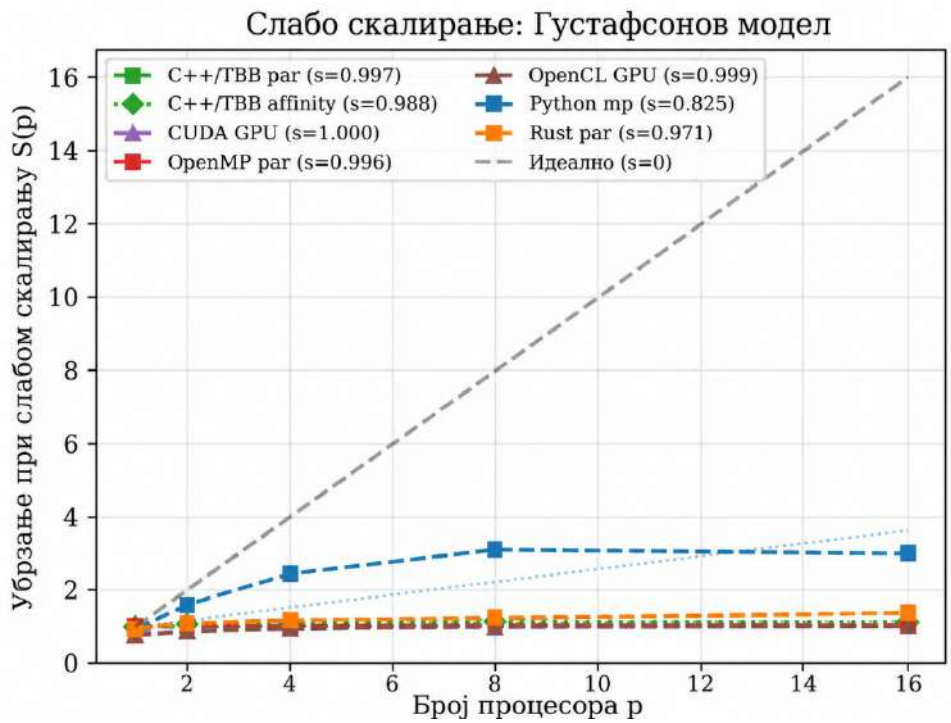
Слика 5.7 - *Слабо скалирање: време извршавања за растуће синтетичке графове*

Табела 5.7 - Однос T_seq(N_p)/T_par(N_p) при слабом скалирању

Имплементација	20 000	40 000	80 000	160 000	320 000
Python мп	0.93	1.60	2.46	3.10	3.01
Rust пар	0.94	1.09	1.18	1.25	1.38
C++/TBB пар	0.99	1.01	1.01	1.05	1.03
C++/TBB affinity	0.99	1.08	1.09	1.15	1.13
OpenMP пар	1.02	1.03	1.02	1.04	1.06
CUDA GPU	0.81	0.91	0.97	0.98	1.02
OpenCL GPU	0.76	0.88	0.95	1.02	1.02

Табела 5.7 показује однос секвенцијалне и паралелне имплементације за исту величину синтетичког графа. Python вишепроцесна обрада највише добија за веће графове, али се убрзање стабилизује око 3×. Rust постепено расте до 1.38×. Основна TBB варијанта остаје близу 1×, док TBB affinity варијанта показује умерено боље понашање и достиже 1.13× на највећем синтетичком графу. OpenMP, CUDA и OpenCL такође остају близу 1×. Ово указује да се код ефикаснијих имплементација највећи део времена троши на меморијски ограничен SpMV, где додатни извршни ресурси не доносе пропорционално смањење времена.

Поређење слабог скалирања са Густафсоновим моделом приказано је на слици 5.8.



Слика 5.8 - *Слабо скалирање: Густафсонов модел за однос $T_{seq}(N_p)/T_{par}(N_p)$*

Густафсонов модел даје s ≈ 0.825 за Python, s ≈ 0.971 за Rust, s ≈ 0.997 за основни C++/TBB, s ≈ 0.988 за C++/TBB affinity, s ≈ 0.996 за OpenMP, s ≈ 1.000 за CUDA и s ≈ 0.999 за OpenCL. Пошто су вредности s близу јединици за компајлиране CPU и GPU имплементације, измерено слабо скалирање не одговара идеалном линеарном убрзању. TBB affinity варијанта има нешто бољу процену од основне TBB варијанте, али и даље не уклања ограничење меморијског система.

5.8. Профилисање, оптимизације и TBB affinity partitioner

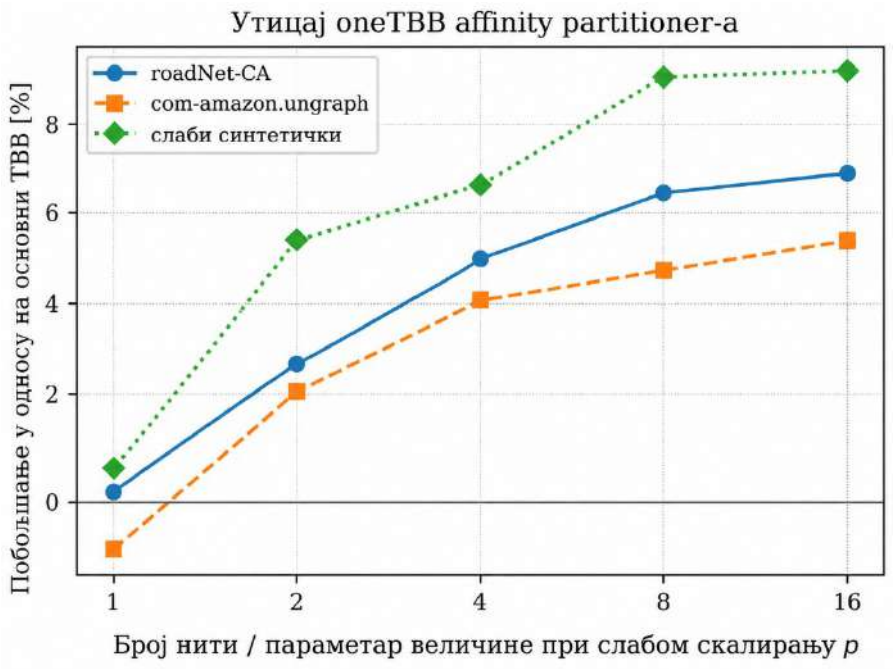
Након основних експеримената додато је профилисање итерационог језгра. Тако је раздвојено укупно време процеса од времена самог итерационог језгра. Такво раздвајање је важно јер скрипте за мерење покрећу извршну датотеку за свако понављање, па `wall_ms` укључује учитавање датотеке, пресликавање идентификатора чворова, сортирање грана и изградњу CSR структуре. Код већих синтетичких графова ова припрема може бити доминантна у односу на 50 итерација алгоритма.

Завршна верзија кода зато уводи три оптимизације меморијског саобраћаја. Прво, CSR индекси се чувају као 32-битне вредности, јер сви коришћени графови стају у тај опсег. Друго, CPU имплементације спајају SpMV, трансформацију $y \leftarrow \alpha y + \beta$ и рачунање резидуала у један пролаз по редовима. Треће, CUDA и OpenCL имплементације спајају SpMV и трансформацију у један кернел и на крају итерације мењају показиваче, односно бафере, уместо копирања вектора y у x.

Додатно је испитана и TBB варијанта са affinity_partitioner-ом. Циљ овог мерења био је да се провери да ли очување сличне расподеле блокова редова између итерација може да побољша локалност приступа и смањи део трошкова распоређивања. Поређење основне TBB варијанте и TBB affinity варијанте приказано је у табели 5.8 и на слици 5.9. Побољшање је израчунато као релативно смањење времена у односу на основну TBB варијанту.

Табела 5.8 - Утицај TBB affinity partitioner-а на време извршавања

Режим	Граф/скала	p	TBB [мс]	TBB affinity [мс]	Побољшање [%]
јако	roadNet-CA	1	10688.878	10670.300	0.17
јако	roadNet-CA	2	10176.981	9905.600	2.67
јако	roadNet-CA	4	10000.982	9488.400	5.13
јако	roadNet-CA	8	9898.110	9264.700	**6.40**
јако	roadNet-CA	16	9981.391	9312.500	6.70
јако	com-amazon.ungraph	1	1904.431	1918.600	-0.74
јако	com-amazon.ungraph	2	1830.324	1787.900	2.32
јако	com-amazon.ungraph	4	1763.252	1688.400	4.25
јако	com-amazon.ungraph	8	1671.217	1589.700	4.88
јако	com-amazon.ungraph	16	1663.263	1571.900	**5.49**
слабо	n = 20k	1	322.465	319.800	0.83
слабо	n = 40k	2	716.385	675.900	5.65
слабо	n = 80k	4	1801.728	1683.700	6.55
слабо	n = 160k	8	4535.096	4145.600	**8.59**
слабо	n = 320k	16	11918.528	10880.200	8.71



Слика 5.9 - Побољшање TBB affinity варијанте

На графу roadNet-CA побољшање је занемарљиво при p = 1, али расте до 6.40% при p = 8 и 6.70% при p = 16. На графу com-amazon.ungraph affinity варијанта је при p = 1 незнатно спорија, али од p = 2 до p = 16 даје стабилно побољшање, до 5.49%. Код слабог скалирања побољшање је најизраженије на већим синтетичким графовима и достиже 8.71% за n = 320000 и p = 16.

Ови резултати показују да affinity_partitioner не мења математички алгоритам и не уклања меморијско ограничење SpMV операције, али може да смањи део трошкова расподеле рада и побољша локалност приступа када се исти CSR граф обрађује кроз више итерација. Ефекат је највидљивији када постоји више радних нити и већи радни скуп

података. Ипак, и после ове оптимизације убрзање TBB варијанте остаје далеко од идеалног, што потврђује да је главно ограничење и даље меморијски саобраћај.

Да би се резултати TBB имплементације упоредили са моделом рада и распона, може се посматрати поједностављена процена $S_p = W/(W/p + D)$. За коришћене графове рад је $W = n + m$, док је распон приближно $D = \Delta + \lceil \log_2 n \rceil$, где је Δ највећи степен чвора у CSR представљању. Пошто је W код графова roadNet-CA и com-amazon.ungraph знатно већи од D, овај модел за $p \leq 16$ предвиђа убрзање блиско идеалном, односно приближно p. Међутим, експериментални резултати су знатно нижи. Поређење теоријске процене и измереног убрзања TBB affinity варијанте приказано је у табели 5.9.

Табела 5.9 - Поређење теоријског и измереног убрзања TBB affinity варијанте

Скуп	p	Теоријско убрзање	Измерено убрзање TBB affinity
roadNet-CA	2	≈ 2.00	≈ 1.08
roadNet-CA	4	≈ 4.00	≈ 1.12
roadNet-CA	8	≈ 8.00	≈ 1.15
roadNet-CA	16	≈ 16.00	≈ 1.15
com-amazon.ungraph	2	≈ 2.00	≈ 1.07
com-amazon.ungraph	4	≈ 4.00	≈ 1.14
com-amazon.ungraph	8	≈ 8.00	≈ 1.21
com-amazon.ungraph	16	≈ 16.00	≈ 1.22

Разлика између теоријског и измереног убрзања показује да ограничење TBB имплементације није недостатак независног рада у алгоритму. Напротив, модел рада и распона показује да постоји висок теоријски паралелизам. Стварно убрзање је знатно мање зато што више радних нити дели исти меморијски пропусни опсег, приступ вектору x зависи од индекса суседа и често нема добру просторну локалност, а додатни трошкови настају и због распоређивања блокова редова и редукције резидуала. Зато се убрзање TBB affinity варијанте поправља у односу на основну TBB варијанту, али остаје далеко од идеалног теоријског убрзања.

5.9. Поређење имплементација

Из резултата се може издвојити неколико главних закључака. Све имплементације су нумерички еквивалентне, што потврђује да се пореде модели паралелизације, а не различите имплементације алгоритма. Rust има најбржу секвенцијалну основу и најбоље укупно време на посматраним реалним графовима. Python вишепроцесна обрада даје највеће релативно убрзање, али због споре основе не достиже компајлиране имплементације у апсолутном времену. OpenMP и основна TBB варијанта показују ограничено убрзање, посебно на roadNet-CA. TBB affinity варијанта поправља TBB резултате, али не мења основни закључак да је овај алгоритам на коришћеним графовима ограничен меморијским приступом. GPU приступи постају оправданији са повећањем величине улазног графа, али на коришћеној GTX 1060 картици и за дати SpMV распоред не остварују јасну предност над најбољом CPU имплементацијом.

Важно је нагласити да се резултати не тумаче као универзална ранг-листа језика, библиотека или TBB начина распоређивања. Мерен је конкретан алгоритам, конкретна CSR репрезентација, конкретна машина и конкретни графови. Другачији распоред података, другачији GPU кернел, другачији TBB partitioner, боља редукција или другачији хардвер могли би променити однос перформанси. Ипак, резултати јасно показују образац типичан за ретке графовске алгоритме у којима је меморијски приступ често важнији од саме аритметике.

6. Закључак

У раду је реализовано и испитано рачунање Кацове централности над великим ретким графовима у шест модела паралелизације: Python вишепроцесна обрада, Rust/Rayon, C++/TBB, C++/OpenMP, NVIDIA CUDA и Khronos OpenCL. Све имплементације користе исти CSR формат, исте параметре командне линије и исти формат излаза, чиме је омогућено равноправно поређење.

Провера коректности показала је да свих тринаест секвенцијалних и паралелних варијанти даје исте скорове у границама нумеричке прецизности. На графу email-Eu-core све имплементације конвергирају у 10 итерација до резидуала $r^{(t)} < 10^{-9}$. Тиме је потврђено да су разлике у мерењима последица начина имплементације и модела паралелизације, а не грешке у алгоритму.

Резултати јаког скалирања показују да структура графа значајно утиче на паралелну ефикасност. На графу `roadNet-CA` Rust постиже најкраће укупно време извршавања, док Python вишепроцесна обрада постиже највеће релативно убрзање у односу на своју секвенцијалну основу. На графу `com-amazon.ungraph` ефекат паралелизације израженији је код свих CPU имплементација, што се види и по нижим проценама серијског дела у Амдаловом моделу. Ипак, CPU имплементације брзо достижу засићење убрзања, што указује на меморијски ограничен карактер SpMV операције.

Резултати слабог скалирања потврђују да повећање величине графа не обезбеђује линеарно скалирање за ефикасне имплементације на датом хардверу. Python вишепроцесна обрада и Rust остварују видљиво побољшање на већим синтетичким графовима, док основна TBB варијанта, OpenMP, CUDA и OpenCL остају близу односа 1× када се пореде са својим секвенцијалним основама на истој величини проблема. TBB affinity варијанта даје умерено боље резултате од основне TBB варијанте, али ни она не уклања ограничење меморијског система. GPU имплементације су коректне и конкурентне на већим улазима, али на коришћеној картици и са једноставним кернелом који једном реду додељује једну нит не достижу најбољу Rust CPU имплементацију.

Додатно профилисање је показало да укупно време процеса није исто што и време итерација Кацове централности. Пошто се граф у сваком покретању мерења поново чита, пресликава, сортира и претвара у CSR, припрема графа може доминирати укупним временом. Због тога је уведен `profile.csv`, а код је оптимизован 32-битним CSR индексима, спајањем пролаза кроз податке и заменом GPU бафера без копирања. Ова допуна додатно поткрепљује главни закључак рада да се перформансе ретких графовских

алгоритама морају анализирати кроз меморијски саобраћај и кроз јасну поделу на припрему података и итерационо језгро.

Као могући правци даљег рада издвајају се оптимизација припреме графа и чување бинарног CSR формата између покретања, као и испитивање алтернативних SpMV формата и библиотека за ретке матрице. Додатно, рад би се могао проширити адаптивном поделом редова, понављањем експеримената на новијем GPU хардверу и укључивањем пондерисаних и усмерених графова.

7. Репродукција експеримената

Експерименти су аутоматизовани и могу се покренути из корена репозиторијума. Комплетно упутство налази се у датотеци `runs.txt`. У тој датотеци наведени су редослед изградње извршних датотека, покретање скрипти за јако и слабо скалирање, као и поступак спајања резултата у табеле и графиконе.

Литература

1. L. Katz, “A New Status Index Derived from Sociometric Analysis,” *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953. DOI: https://doi.org/10.1007/BF02289026.

2. L. C. Freeman, “Centrality in Social Networks: Conceptual Clarification,” *Social Networks*, vol. 1, no. 3, pp. 215–239, 1978/1979. DOI: https://doi.org/10.1016/0378-8733(78)90021-7.

3. P. Bonacich, “Power and Centrality: A Family of Measures,” *American Journal of Sociology*, vol. 92, no. 5, pp. 1170–1182, 1987. DOI: https://doi.org/10.1086/228631.

4. U. Brandes, “A Faster Algorithm for Betweenness Centrality,” *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001. DOI: https://doi.org/10.1080/0022250X.2001.9990249.

5. A.-L. Barabasi, *Network Science*. Cambridge University Press, 2016. http://networksciencebook.com/.

6. L. Page, S. Brin, R. Motwani and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web,” Stanford InfoLab Technical Report, 1999. https://ilpubs.stanford.edu/422/.

7. A. N. Langville and C. D. Meyer, *Google’s PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

8. T. A. Davis, *Direct Methods for Sparse Linear Systems*. SIAM, 2006. DOI: https://doi.org/10.1137/1.9780898718881.

9. Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, 2003.

10. N. Bell and M. Garland, “Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009. DOI: https://doi.org/10.1145/1654059.1654078.

11. S. Williams, A. Waterman and D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. DOI: https://doi.org/10.1145/1498765.1498785.

12. J. D. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” IEEE Computer Society Technical Committee on Computer Architecture Newsletter, 1995. https://www.cs.virginia.edu/~mccalpin/papers/bandwidth/bandwidth.html.

13. J. Kepner et al., “Mathematical Foundations of the GraphBLAS,” 2016. arXiv: https://arxiv.org/abs/1606.05790.

14. J. Leskovec and A. Krevl, “SNAP Datasets: Stanford Large Network Dataset Collection,” 2014. https://snap.stanford.edu/data/.

15. Stanford Network Analysis Project, “email-Eu-core network.” https://snap.stanford.edu/data/email-Eu-core.html.

16. Stanford Network Analysis Project, “California road network.” https://snap.stanford.edu/data/roadNet-CA.html.

17. Stanford Network Analysis Project, “Amazon product co-purchasing network and ground-truth communities.” https://snap.stanford.edu/data/com-Amazon.html.

18. G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” *AFIPS Conference Proceedings*, vol. 30, pp. 483–485, 1967. https://cds.cern.ch/record/407343.

19. J. L. Gustafson, “Reevaluating Amdahl’s Law,” *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988. https://cacm.acm.org/research/reevaluating-amdahls-law/.

20. OpenMP Architecture Review Board, “OpenMP API Specifications.” https://www.openmp.org/specifications/.

21. NVIDIA Corporation, “CUDA C++ Programming Guide.” https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

22. Khronos OpenCL Working Group, “The OpenCL Specification.” https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

23. UXL Foundation, “oneTBB documentation: parallel_reduce and Partitioner Summary.” https://uxlfoundation.github.io/oneTBB/main/tbb_userguide/parallel_reduce.html; https://uxlfoundation.github.io/oneTBB/main/tbb_userguide/Partitioner_Summary.html.

24. Rayon developers, “Rayon crate documentation.” https://docs.rs/crate/rayon/latest.

25. M. Popović and V. Kovačević, *Paralelno programiranje*, 2nd ed. Novi Sad, Serbia: Fakultet tehničkih nauka, 2021.