



УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ
НАУКА У НОВОМ САДУ



Владимир Чорненки

**Паралелизација PoW и PoS
блокчејн консензус алгоритма
у оквиру децентрализованог
P2P система**

ЗАВРШНИ РАД

Основне академске студије

Нови Сад, 2025



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

Број:

ЗАДАТАК ЗА ЗАВРШНИ РАД

Датум:

(Податке уноси предметни наставник - менџор)

Студијски програм:	Софтверско инжењерство и информационе технологије		
Студент:	Владимир Чорненки	Број индекса:	SV53/2021
Степен и врста студија:	Основне академске студије		
Област:	Електротехничко и рачунарско инжењерство		
Ментор:	Игор Дејановић		
НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:			
<ul style="list-style-type: none">- проблем – тема рада;- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;			

НАСЛОВ ЗАВРШНОГ РАДА:

Паралелизација PoW и PoS блокчејн консензус алгоритма у оквиру децентрализованог P2P система

ТЕКСТ ЗАДАТКА:

Истражити област блокчејн консензус алгоритама. Пројективати и имплементирати блокчејн систем са фокусом на паралелизацију Proof-Of-Work и Proof-Of-Stake консензус алгоритама. Испитати скалабилност паралелних имплементација на програмским језицима Rust и Python.

У изради користити препоручену праксу из области софтверског инжењерства.

Детаљно документовати решење.

Руководилац студијског програма:	Ментор рада:

Примерак за: - Студента; - Ментора



КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, РБР:		
Идентификациони број, ИБР:		
Тип документације, ТД:	Монографска документација	
Тип записа, ТЗ:	Текстуални штампани материјал	
Врста рада, ВР:	Дипломски - бачелор рад	
Аутор, АУ:	Владимир Чорненки	
Ментор, МН:	Др Игор Дејановић, редовни професор	
Наслов рада, НР:	Паралелизација PoW и PoS блокчејн консензус алгоритма у оквиру децентрализованог P2P система	
Језик публикације, ЈП:	српски/ћирилица	
Језик извода, ЈИ:	српски/енглески	
Земља публиковања, ЗП:	Република Србија	
Уже географско подручје, УГП:	Војводина	
Година, ГО:	2025	
Издавач, ИЗ:	Ауторски репринт	
Место и адреса, МА:	Нови Сад, трг Доситеја Обрадовића 6	
Физички опис рада, ФО: (поплављено/страница/чиштата/табела/слика/графика/прилога)	6/57/6/6/15/02	
Научна област, НО:	Електротехничко и рачунарско инжењерство	
Научна дисциплина, НД:	Примењене рачунарске науке и информатика	
Предметна одредница/Кључне речи, ПО:	Блокчејн, Консензус Алгоритми, PoW, PoS, Дистрибуирани Системи, P2P Мрежа, Rust, Python	
УДК		
Чува се, ЧУ:	У библиотеки Факултета техничких наука, Нови Сад	
Важна напомена, ВН:		
Извод, ИЗ:	Овај рад описује пројектовање и имплементацију комплетног блокчејн система са фокусом на паралелизацију консензус алгоритама и дистрибуирану P2P комуникацију. Пројекат обухвата паралелне имплементације Proof-of-Work (PoW) и Proof-of-Stake (PoS) алгоритама у програмским језицима Rust и Python, детаљну анализу перформанси кроз експерименте јаког и слабог скалирања, и функционалну децентрализовану мрежу са "gossip" протоколом и консензусом заснованим на правилу најдужег ланца. Резултати демонстрирају значајне предности паралелизације и истичу фундаменталне разлике у ефикасности између PoW-и PoS-приступа.	
Датум прихваташа теме, ДП:		
Датум одbrane, ДО:	01.01.2025	
Чланови комисије, КО:	Председник:	Др Петар Петровић, ванредни професор
	Члан:	Др Марко Марковић, доцент
	Члан:	Потпис ментора
	Члан, ментор:	Др Игор Дејановић, редовни професор



KEY WORDS DOCUMENTATION

Accession number, ANO:		
Identification number, INO:		
Document type, DT:	Monographic publication	
Type of record, TR:	Textual printed material	
Contents code, CC:		
Author, AU:	Vladimir Cornenki	
Mentor, MN:	Igor Dejanović, Phd., full professor	
Title, TI:	Parallelization of PoW and PoS blockchain consensus algorithm within a decentralized P2P system	
Language of text, LT:	Serbian	
Language of abstract, LA:	Serbian/English	
Country of publication, CP:	Republic of Serbia	
Locality of publication, LP:	Vojvodina	
Publication year, PY:	2025	
Publisher, PB:	Author's reprint	
Publication place, PP:	Novi Sad, Dositeja Obradovica sq. 6	
Physical description, PD: (chapters/pages/ref./tables/pictures/graphs/appendices)	6/57/6/6/15/0/2	
Scientific field, SF:	Electrical and Computer Engineering	
Scientific discipline, SD:	Applied computer science and informatics	
Subject/Key words, S/KW:	Blockchain, Consensus Algorithms, PoW, PoS, Distributed Systems, P2P Network, Rust, Python	
UC		
Holding data, HD:	The Library of Faculty of Technical Sciences, Novi Sad	
Note, N:		
Abstract, AB:	This thesis describes the design and implementation of a complete blockchain system focused on the parallelization of consensus algorithms and distributed P2P communication. The project encompasses parallel implementations of Proof-of-Work (PoW) and Proof-of-Stake (PoS) algorithms in Rust and Python, a detailed performance analysis through strong and weak scaling experiments, and a functional decentralized network with a gossip protocol and a longest-chain consensus rule. The results demonstrate significant benefits of parallelization and highlight the fundamental efficiency differences between the PoW and PoS approaches.	
Accepted by the Scientific Board on, ASB:		
Defended on, DE:	01.01.2025	
Defended Board, DB:	President: Petar Petrović, Phd., assoc. professor	
Member:	Marko Marković, Phd., asist. professor	
Member:		
Member, Mentor:	Igor Dejanović, Phd., full professor	Mentor's sign



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 партнери, родитељ или усвојитељ, дете или усвојеник), повезано лице (кровни 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

Садржај

1 Увод	1
1.1 Мотивацija	1
1.2 Циљ рада	2
1.3 Структура рада	2
2 Теоријске основе	5
2.1 Основе блокчејн технологије	5
2.2 Консензус алгоритми	6
2.3 Теорија Паралелног Рачунарства	9
3 Имплементација консензус алгоритама	13
3.1 Заједничке структуре података	13
3.2 <i>Proof-of-Work</i> : Секвенцијална имплементација	14
3.3 <i>Proof-of-Work</i> : Паралелна имплементација	17
3.4 Имплементација <i>Proof-of-Stake</i> симулације	20
4 Експерименти и анализа резултата	23
4.1 Спецификација система и методологија	23
4.2 Експеримент јаког скалирања	24
4.3 Експеримент слабог скалирања	28
4.4 Поређење <i>Python</i> и <i>Rust</i>	31
4.5 Анализа перформанси <i>Proof-of-Stake</i> симулације	32
5 Дизајн и имплементација P2P мреже	35
5.1 Архитектура мреже	35
5.2 Мрежни протокол	37
5.3 Мрежни процеси	38
5.4 Постизање консензуса: правило најдужег ланца	39
5.5 <i>Gossip</i> протокол и пропагација порука	40
6 Закључак	43
6.1 Резултати и доприноси рада	43
6.2 Ограничења и правци даљег развоја	44
6.3 Завршна реч	44
Списак слика	45
Списак листинга	47
Списак табела	49
Списак коришћених скраћеница	51
Списак коришћених појмова	53
Биографија	55

Литература	57
------------------	----

Глава 1

Увод

1.1 Мотивација

Блокчејн (енг. *Blockchain*) технологија је у последњој деценији прешла пут од идеје до важног система који мења начин на који разумемо дигитално поверење, власништво и размену вредности. Њена суштинска карактеристика, способност одржавања децентрализоване, непроменљиве и транспарентне евиденције трансакција без потребе за централним ауторитетом, отворила је врата за револуционарне примене у финансијама, логистици, здравству и многим другим областима.

Срце сваког блокчејн система чини консензус алгоритам. То је механизам који омогућава дистрибуиранију мрежи независних учесника да се усагласи око јединственог стања система. Први и најпознатији алгоритам, *Proof-of-Work* (PoW), који је прославио Биткоин [1], заснива се на решавању рачунарски захтевних криптографских задатака. Иако је доказао своју робусност и сигурност, PoW има и значајне недостатке. Највећи проблем је значајна потрошња електричне енергије [2], која на глобалном нивоу достиже разmere потрошње читавих држава. Поред тога, инхерентна ограничења у брзини, попут десетоминутног времена за креирање блока код Биткоина, чине га непрактичним за апликације које захтевају високу пропусност трансакција.

Као одговор на ове изазове, развијен је *Proof-of-Stake* (PoS) као енергетски ефикаснија алтернатива. Уместо да се ослања на рачунарску снагу, PoS сигурност темељи на економском улогу (енг. *stake*) учесника у мрежи. Прелазак мреже Етереум (енг. *Ethereum*) са PoW на PoS механизам 2022. године резултирао је смањењем потрошње енергије за преко 99.9% [3], што јасно илуструје предности овог приступа. PoS такође омогућава знатно брже креирање блокова и већу пропусност, што га чини атрактивнијим за модерне децентрализоване апликације.

Међутим, избор самог алгоритма представља тек први корак. Перформансе блокчејн система у пракси пресудно зависе од квалитета његове имплементације и способности да ефикасно искористи доступне хардверске ресурсе. У ери вишејезгарних процесора, паралелизација рачунарски интензивних операција постаје кључна за постизање оптималних перформанси. Поставља се питање како се ови алгоритми понашају када се имплементирају у паралелном окружењу и колико ефикасно могу да скалирају са повећањем броја процесорских језгара? Поред тога, избор програмског језика, као што су интерпретирани *Python*, познат по једноставности, и компајлирани *Rust*, познат

1.1 Мотивација

по перформансама и сигурности при раду са више нити, има драматичан утицај на коначне резултате.

1.2 Циљ рада

Циљ овог рада је развој, анализа и поређење два доминантна блокчејн консензус алгоритма, *Proof-of-Work* и *Proof-of-Stake*, са посебним нагласком на њихову паралелизацију и евалуацију перформанси. У оквиру рада реализује се потпуна имплементација оба механизма, при чему је *Proof-of-Work* развијен у секвенцијалној и паралелној верзији у *Python*-у и *Rust*-у, док је *Proof-of-Stake* представљен кроз симулацију у *Rust*-у са паралелном обрадом и валидацијом трансакција.

Посебна пажња посвећена је експерименталној анализи скалирања, где се кроз јако и слабо скалирање испитује утицај повећања процесорских ресурса на убрзанаје, време извршавања и пропусност система. Добијени резултати се затим упоређују са теоријским очекивањима дефинисаним Амдаловим и Густафсоновим законом [4].

У циљу симулације реалног система, у раду је имплементирана и децентрализована *peer-to-peer* (P2P) мрежа која омогућава чврсвима да се међусобно повезују, разменјују блокове, синхронизују ланац и учествују у процесима рударења и постизања консензуса кроз *gossip* комуникациони модел.

Кроз овако конципиран приступ, рад пружа свеобухватну студију која обједињује практичну имплементацију, теоријску анализу и експерименталну евалуацију, са циљем да се јасно прикажу предности и ограничења ових консензус система у погледу перформанси, ефикасности и скалабилности.

1.3 Структура рада

Рад је организован у више поглавља како би се читалац на систематичан начин увео у проблематику и спровео кроз све фазе пројекта.

У другом поглављу представљени су фундаментални концепти блокчејн технологије, детаљно су објашњени принципи рада *Proof-of-Work* и *Proof-of-Stake* алгоритама, и дате су теоријске основе паралелног рачунарства, укључујући Амдалов и Густафсонов закон.

Треће поглавље посвећено је техничким детаљима имплементације. Описане су основне структуре података као што су блок и блокчејн, а затим је приказана реализација секвенцијалних и паралелних верзија PoW алгоритма у *Python*-у и *Rust*-у, као и симулација PoS система.

Четврто поглавље детаљно описује архитектуру децентрализоване мреже. Обрађени су мрежни протокол, механизми за откривање и повезивање чврсова, *gossip* протокол

за пропагацију података, као и имплементација “најдужи ланац” правила за постизање консензуса.

Пето поглавље представља централни део анализе. Описана је методологија тестирања, након чега су приказани и продискутовани резултати експеримената јаког и слабог скалирања. Врши се поређење перформанси *Python* и *Rust* имплементација, као и упоредна анализа карактеристика PoW и PoS система.

На крају, у шестом поглављу, сумирани су кључни резултати и доприноси рада, анализиране су предности и мане развијеног решења и дати су предлози за могућа будућа истраживања и унапређења.

1.3 Сұрекшұра пага

Глава 2

Теоријске основе

2.1 Основе блокчејн технологије

Блокчејн (енг. *blockchain*) је дистрибуирана, децентрализована и криптографски обезбеђена дигитална књига трансакција. За разлику од традиционалних централизованих база података, где један ентитет има потпуну контролу над подацима, копија блокчејна је чувана и синхронизована међу свим учесницима у *peer-to-peer* (*P2P*) мрежи. Оваква архитектура елиминише потребу за посредником или централним ауторитетом, чиме се омогућава сигурна и транспарентна размена директно између учесника.

Основу технологије чине две кључне компоненте: блок и ланац.

Блок представља контејнер за податке. Сваки блок у ланцу садржи скуп трансакција које су се додали у одређеном временском периоду, као и неколико кључних метаподатака који га повезују са остатком ланца:

- **Хеш претходног блока (енг. *previous_hash*):** Криптографски отисак претходног блока у ланцу, који служи као веза и осигурува континуитет.
- **Временска ознака (енг. *timestamp*):** Запис о времену када је блок креiran.
- **Нонс (енг. *nonce*):** Случајан број који рудари покушавају да погоде у *Proof-of-Work* систему како би задовољили унапред дефинисану тежину.
- **Подаци (енг. *data*):** Садржи листу трансакција које су укључене у блок.
- **Хеш блока (енг. *hash*):** Јединствени криптографски отисак који се израчујава на основу свих осталих података у блоку. Овај хеш служи као јединствени идентификатор блока.

Ланац настаје повезивањем блокова у хронолошки низ. Сваки нови блок у себи садржи хеш претходног блока, чиме се ствара криптографска веза. Први блок у ланцу, познат као генесис (енг. *genesis*) блок, једини је који не садржи референцу на претходни блок. Ова ланчана структура је темељ сигурности и непроменљивости система.

Криптографско хеширање је математички процес који узима улазне податке произвољне величине и од њих креира излаз фиксне дужине, хеш. Овај процес је детерминистички (исти улаз увек производи исти излаз) и једносмеран, што значи да је рачунарски практично немогуће из излазног хеша реконструисати оригиналне улазне податке. У већини блокчејн система, укључујући и имплементацију у овом

раду, користи се алгоритам SHA-256. Хеширање има двоструку улогу: користи се за креирање јединственог идентификатора сваког блока и за повезивање блокова у ланац путем `previous_hash` поља.

Из овакве структуре произилазе кључне карактеристике блокчејна:

- **Децентрализација:** Подаци нису склadiштени на једном централном серверу, већ су дистрибуирани широм мреже. Ово систем чини отпорним на појединачне тачке отказа и цензуру јер не постоји један ауторитет који може самостално да мења или брише податке.
- **Непроменљивост (енг. *immutability*):** Једном када су подаци уписаны у блок и тај блок додат у ланац, практично их је немогуће изменити. Ако би нападач покушао да измене податке у неком од претходних блокова, хеш тог блока би се променио. Пошто наредни блок садржи стари хеш, веза би била прекинута. Да би сакрио своју измену, нападач би морао поново да израчуна хешеве свих наредних блокова у ланцу, што је рачунарски изузетно захтеван процес и практично неизводљив на великим мрежама. Ово својство гарантује интегритет података и гради поверење међу учесницима.

2.2 Консензус алгоритми

Како би децентрализована мрежа, састављена од великог броја независних и потенцијално злонамерних учесника, могла да функционише као јединствен и кохерентан систем, неопходно је да постоји механизам који им омогућава да се усагласе око заједничког стања. Овај механизам се назива консензус алгоритам. Његова основна улога је да осигура да сви чворови у мрежи имају идентичну копију блокчејна и да се сви слажу око тога који су блокови валидни и којим редоследом се додају у ланац.

У дистрибуираним системима, постизање консензуса је нетривијалан проблем, познат као Проблем византијских генерала (енг. *Byzantine Generals Problem*) [5]. Овај проблем описује сценарио у којем група генерала мора да се договори о заједничком плану напада, али комуницирају само путем гласника, а међу њима може бити издајника који шаљу лажне поруке. Циљ је пронаћи протокол који омогућава лојалним генералима да донесу исту одлуку, чак и у присуству издајника. У контексту блокчејна, чворови у мрежи су генерали, а злонамерни чворови који покушавају да компромитују систем су издајници.

Консензус алгоритам мора да обезбеди својство познато као толеранција на византијске грешке (енг. *Byzantine Fault Tolerance*, BFT), што је способност система да настави са исправним радом чак и ако се неки од његових чворова понашају непредвидиво или злонамерно.

Функција консензус алгоритма у блокчејн мрежи је вишеструка:

- Избор креатора блока:** Дефинише правила по којима се одређује који учесник у мрежи има право да креира и предложи следећи блок.
- Валидација и сигурност:** Осигурува да се у ланац додају само валидни блокови који поштују правила протокола, чиме се мрежа штити од напада попут двоструког трошења (енг. *double-spending*).
- Одржавање јединственог ланца:** Пружа механизам за решавање конфликтата у случају да два или више учесника истовремено предложе валидан блок, осигурујући да се мрежа на крају усагласи око једне верзије.

Постоји велики број различитих приступа за постизање консензуса, од којих сваки има своје компромисе у погледу сигурности, брзине, скалабилности и потрошње ресурса. У наредним потпоглављима биће детаљно описана два најзначајнија и најраспрострањенија алгоритма који су предмет анализе у овом раду, *Proof-of-Work* и *Proof-of-Stake*.

2.2.1 Proof-of-Work (PoW)

Proof-of-Work (PoW), или доказ о раду, је први и најпознатији консензус алгоритам имплементиран у Биткоину и многим другим блокчејн системима. Основни принцип овог механизма је да учесници у мрежи морају да обаве значајан рачунарски рад како би добили право да додају нови блок у ланац. Овај процес је намерно дизајниран да буде тежак и скуп, али да резултат тог рада буде лако проверљив од стране осталих учесника у мрежи.

Централни концепт PoW алгоритма је рударење (енг. *mining*). То је процес у којем се учесници (рудари) такмиче у решавању сложене криптографске загонетке. Циљ је пронаћи хеш за нови блок који задовољава одређени услов. Пошто је излаз хеш функције практично непредвидив, једини начин да се пронађе одговарајући хеш јесте методом грубе силе (енг. *brute-force*). Он подразумева поновно испробавање различитих улазних вредности.

Да би се ово омогућило, сваки блок садржи посебно поље названо нонс (енг. *nonce*, скраћено од “*number used only once*”). Рудари непрекидно мењају вредност нонса, израчунавају хеш целог блока за сваку нову вредност, и проверавају да ли добијени хеш задовољава задати услов. Први рудар који пронађе валидан нонс побеђује, добија право да свој блок дода у ланац и бива награђен одређеном количином криптовалуте.

Услов који хеш мора да задовољи дефинисан је параметром тежина (енг. *difficulty*). У пракси, тежина представља захтев да хеш блока мора бити мањи од одређене циљне вредности, што је еквивалентно томе да мора почињати одређеним бројем нула. На пример, ако је тежина подешена тако да захтева пет водећих нула, рудари ће тражити нонс који производи хеш облика “00000abcdef...”. Што је већи захтевани број нула, то је мања вероватноћа проналаска одговарајућег хеша у једном покушају, па је и процес рударења тежи и дуготрајнији.

Тежина није статична вредност. Мрежа је аутоматски периодично подешава како би се време креирања новог блока одржало на приближно константном нивоу (на пример око 10 минута за Биткоин), без обзира на промене у укупној рачунарској снази мреже. Ако се више рудара придружи мрежи и блокови се проналазе пребрзо, тежина се повећава. Ако рудари напусте мрежу, тежина се смањује.

Сигурност PoW система произилази директно из рачунарске тежине рударења. Да би нападач изменио трансакцију у неком од претходних блокова, морао би поново да изрудари тај блок и све наредне блокове који су на њега повезани, и то брже него што остатак мреже ствара нове блокове на исправном ланцу. Овакав напад, познат као напад 51%, захтевао би да нападач контролише више од половине укупне рачунарске снаге мреже, што је на великим мрежама изузетно скупо и тешко изводљиво.

Иако изузетно сигуран, највећи недостатак PoW алгоритма је његова енергетска неефикасност. Конкурентска природа рударења, где се милиони специјализованих уређаја широм света такмиче у насумичном погађању нонса, доводи до потрошње огромне количине електричне енергије, што представља значајан еколошки и економски изазов [2].

2.2.2 Proof-of-Stake (PoS)

Као директан одговор на изазове, пре свега на велику потрошњу енергије *Proof-of-Work* алгоритма, развијен је *Proof-of-Stake* (PoS). Овај консензус механизам мења основни принцип заштите мреже, уместо да се сигурност заснива на рачунарској снази, она се темељи на економском учешћу. У PoS систему, учесници познати као валидатори (енг. *validators*) не троше електричну енергију на решавање криптоографских задатака, већ морају да закључају одређену количину криптовалуте као свој улог (енг. *stake*), који служи као залог за њихово поштено понашање.

Процес креирања новог блока у PoS систему се назива ковање (енг. *forging*) или потврђивање (енг. *attesting*). Избор валидатора који ће креирати следећи блок није такмичарски процес као у PoW, већ се врши путем детерминистичког или псевдослучајног алгоритма. Најчешћи приступ је да вероватноћа да неки валидатор буде изабран буде директно пропорционална величини његовог улога, што је већи улог, већа је и шанса да баш тај валидатор добије право да предложи следећи блок и за то добије награду у виду трансакционих провизија.

Сигурност мреже је загарантована економским подстицајима. Пошто су валидатори финансијски уложили у систем, у њиховом је најбољем интересу да се понашају поштено и одржавају интегритет мреже, јер би сваки напад умањио вредност њиховог улога. Уколико се валидатор понаша злонамерно (покуша да потврди неважећу трансакцију или предложи више блокова за исту позицију у ланцу), систем га може казнити одузимањем дела или чак целог уложеног износа. Овај казнени механизам је познат

као слешинг (енг. *slashing*) и представља снажан демотивишући фактор за било какав облик напада. “Напад 51%” у PoS контексту би захтевао да нападач поседује већину укупне уложене криптовалуте, што би било изузетно скupo, а успех напада би вероватно довео до пада вредности те исте валуте, чинећи га економски бесмисленим.

Proof-of-Stake нуди неколико кључних предности у односу на *Proof-of-Work*:

- **Енергетска ефикасност:** Најзначајнија предност PoS система је драстично мања потрошња енергије. Пошто нема потребе за интензивним рачунарским операцијама, хардверски захтеви за учешће у мрежи су минимални. Преласком мреже Етереум на PoS, процењено је да је њена укупна потрошња енергије смањена за преко 99.9% [3].
- **Низки праг за учешће:** У PoW системима, за успешно рударење је потребан скуп и специјализован хардвер. У PoS системима, праг за улазак је поседовање одређене количине криптовалуте, што може учинити учешће у обезбеђивању мреже доступнијим ширем кругу корисника.
- **Већа брзина и пропусност:** Пошто не постоји потреба за решавањем тешког криптографског задатка, процес избора валидатора и креирања блока може бити знатно бржи. Ово омогућава креирање блокова у интервалима од неколико секунди, што резултира већом пропусношћу трансакција.

2.3 Теорија Паралелног Рачунарства

Развој рачунарског хардвера у последње две деценије обележен је променом парадигме. Уместо повећања брзине једног процесорског језгра, фокус је прешао на повећање броја језгара унутар једног процесора. Ова промена је условила да се за постизање бољих перформанси софтвер мора дизајнирати тако да може истовремено да извршава више задатака. Тај приступ је познат као паралелно рачунарство (енг. *parallel computing*), и представља темељ рачунарства високих перформанси (енг. *High-Performance Computing*, HPC).

Процес рударења у *Proof-of-Work* систему, који се своди на независно испробавање огромног броја нонс вредности, представља пример проблема који је инхерентно паралелизован (енг. *embarrassingly parallel*). То значи да се укупан посао може лако поделити на мање независне делове који се могу извршавати истовремено на различитим језгрима процесора без потребе за међусобном комуникацијом током извршавања. Ова карактеристика чини PoW идеалним кандидатом за примену техника паралелног рачунарства.

Да би се измерила и анализирала ефикасност паралелне имплементације, неопходно је увести теоријске концепте и метрике које омогућавају објективну процену перформанси. У наредним потпоглављима биће дефинисане основне метрике попут убрзања

и ефикасности, представљени модели јаког и слабог скалирања, и објашњени теоријски оквири које постављају Амдалов и Густафсонов закон.

2.3.1 Метрике за мерење перформанси

Да би се објективно процениле предности паралелне имплементације у односу на секвенцијалну, користе се стандардне метрике из области рачунарства високих перформанси. Две најважније метрике су убрзање и ефикасност.

- **Убрзање (енг. Speedup)** мери колико је пута паралелна верзија програма бржа од одговарајуће секвенцијалне верзије. Рачуна се као однос времена извршавања на једном процесорском језгру (T_1) и времена извршавања на p језгара (T_p):

$$S_p = \frac{T_1}{T_p}$$

У идеалном случају, убрзање је линеарно ($S_p = p$), што би значило да програм ради p пута брже на p језгара. У пракси, ово је ретко оствариво због додатног оптерећења (енг. overhead) које настаје услед креирања и синхронизације нити, комуникације и расподеле посла.

- **Ефикасност (енг. Efficiency)** представља нормализовану вредност убрзања и показује колико добро су процесорски ресурси искоришћени. Рачуна се као однос постигнутог убрзања (S_p) и броја искоришћених процесорских језгара (p):

$$E_p = \frac{S_p}{p}$$

Ефикасност се обично изражава у процентима. Идеална вредност је 1 (или 100%), што означава савршено линеарно убрзање. Вредности мање од 1 указују на то да је део рачунарских ресурса потрошен на додатно оптерећење уместо на користан рад.

2.3.2 Модели скалирања

Скалабилност је кључна карактеристика паралелног система која описује како се његове перформансе понашају са повећањем броја процесорских ресурса. Постоје два основна модела за анализу скалабилности:

- **Јако скалирање (енг. Strong Scaling)** се користи за процену способности система да исти проблем реши брже коришћењем већег броја процесора. Укупна величина проблема остаје фиксна док се број процесорских језгара (p) повећава. Циљ је да се време извршавања (T_p) смањи што је више могуће. Јако скалирање је посебно важно за апликације где је време одзива критично.
- **Слабо скалирање (енг. Weak Scaling)** процењује способност система да реши већи проблем у истом временском року коришћењем већег броја процесора. Величина

проблема расте пропорционално са повећањем броја језгара тако да свако језгро обрађује приближно исту количину посла. Циљ је да време извршавања (T_p) остане константно. Слабо скалирање је релевантно за научне симулације и обраду великих скупова података, где додатни ресурси омогућавају детаљнију или обимнију анализу.

2.3.3 Теоријски модели убрзања: Амдалов и Густафсонов закон

Ефикасност паралелних програма није неограничена. Чак и у идеалним условима, постојање дела кода који се мора извршавати секвенцијално представља ограничење. Ова ограничења су дефинисана кроз два теоријска модела.

- **Амдалов закон** дефинише теоријски максимум убрзања који се може постићи за проблем фиксне величине, што одговара моделу јаког скалирања [4]. Закон каже да је убрзање ограничено делом програма који се не може паралелизовати (серијским делом). Формула гласи:

$$S(p) = \frac{1}{s + \frac{1-s}{p}}$$

где је s пропорција укупног времена извршавања која припада серијском делу, $1 - s$ пропорција која припада паралелном делу, а p је број процесорских језгара. Из ове формуле следи да како p тежи бесконачности, максимално убрзање тежи вредности $\frac{1}{s}$.

- **Густафсонов закон** нуди другачију перспективу која је у складу са моделом слабог скалирања [4]. Овај закон претпоставља да се са већим бројем процесора решавају и већи проблеми. Он не мери колико се брже решава исти проблем, већ колико се већи проблем може решити у истом временском року. Убрзање које се постиже дефинисано је формулом:

$$S(p) = p - s(p - 1)$$

где је s пропорција укупног времена извршавања која припада серијском делу, а p је број процесорских језгара. Докле год се величина проблема може повећавати, убрзање се може приближити линеарном. Овај модел боље одражава стварну употребу великих паралелних система, где је циљ најчешће повећање обима или прецизности обраде, а не само смањење времена извршавања.

Глава 3

Имплементација консензус алгоритама

У овом поглављу детаљно ће бити приказана техничка реализација кључних компоненти блокчејн система. Фокус је на имплементацији *Proof-of-Work* и *Proof-of-Stake* консензус алгоритама. За потребе анализе резултата, *Proof-of-Work* алгоритам је имплементиран у програмским језицима *Python* и *Rust*, док је за енергетски ефикаснији *Proof-of-Stake* алгоритам коришћен *Rust* због његових напредних могућности за сигуран рад са више нити.

3.1 Заједничке структуре података

У основи сваког блокчејн система налазе се две фундаменталне структуре података: *Block*, која представља појединачну карику, и *Blockchain*, која представља цео ланац. У овом пројекту, ове структуре су дефинисане у програмском језику *Rust*, са циљем да буду меморијски ефикасне и робусне.

Структура *Block* је дизајнирана да садржи све неопходне елементе за функционисање ланца.

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Block {
    pub previous_hash: String,
    pub timestamp: i64,
    pub nonce: u64,
    pub data: String,
    pub hash: String,
}
```

Листинг 1: Структура *Block* у *Rust*-у.

Кључна поља су:

- *previous_hash: String* који чува хеш претходног блока, чиме се остварује веза у ланцу
- *timestamp: i64* вредност која представља UNIX временску ознаку креирања блока
- *nonce: u64* ово поље је кључно за PoW алгоритам, јер се његова вредност мења у свакој итерацији рударења
- *data: String* који у овом раду садржи информације о трансакцијама унутар блока
- *hash: String* који чува SHA-256 хеш блока, израчунат на основу свих осталих поља

3.1 Заједничке структуре њога јака

Структура `Block` имплементира неколико важних метода. Метода `calculate_hash()` конкатенира све остале податке у блоку, претвара их у низ бајтова и на њих примењује SHA-256 хеш функцију. Ова метода је централна за процес рударења и валидације. Метода `meets_difficulty(difficulty: usize)` проверава да ли израчунати хеш задовољава тренутну тежину мреже, односно да ли почиње са одговарајућим бројем нула.

Друга основна структура, `Blockchain`, обједињује блокове у ланац.

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Blockchain {
    pub chain: Vec<Block>,
    pub difficulty: usize,
}
```

Листинг 2: Структура `Blockchain` у *Rust*-у.

Ова структура садржи два поља:

- `chain`: Вектор који чува низ свих блокова, почевши од генесис блока.
- `difficulty`: вредност која дефинише тежину рударења за цео ланац.

Приликом креирања новог `Blockchain` објекта, аутоматски се иницијализује и додаје генесис блок, који служи као почетна тачка ланца. Структура такође садржи методе за додавање новог блока (`add_block`), као и за валидацију интегритета целог ланца (`is_valid`), која проверава да ли сваки блок исправно референцира хеш претходног и да ли задовољава задату тежину. Ове структуре представљају основу на којој се граде сви алгоритми описани у наредним поглављима.

3.2 Proof-of-Work: Секвенцијална имплементација

Секвенцијална имплементација *Proof-of-Work* алгоритма представља најједноставнији приступ рударењу и служи као темељ за разумевање рачунарске захтевности овог процеса. У овом моделу, једна процесорска нит (енг. *thread*) извршава целокупан посао, тестирајући ноанс вредности једну по једну, редом, док не пронађе решење.

Ова имплементација је од кључног значаја за експерименталну анализу, јер време извршавања на једном језгру (T_1) представља основну референтну вредност за израчунавање убрзања и ефикасности паралелних верзија. Логика је идентична у оба програмска језика, започиње се са ноанс вредношћу 0, у свакој итерацији се креира нови блок, рачуна његов SHA-256 хеш и проверава да ли задовољава задату тежину. Уколико не, ноанс се инкрементира за 1 и процес се понавља.

3.2.1 Имплементација у програмском језику Python

У програмском језику *Python*, секвенцијално рударење је реализовано унутар функције `mine_block`. Ова функција садржи “*brute-force*” петљу која итеративно проверава ноанс вредности.

```

def mine_block(
    previous_hash: str,
    transactions: List[Transaction],
    difficulty: int,
    start_nonce: int = 0,
    max_nonce: int = None,
    progress_callback=None,
) -> tuple[Block, int]:
    required_prefix = "0" * difficulty
    nonce = start_nonce
    nonces_tested = 0
    timestamp = time.time()

    while max_nonce is None or nonce < max_nonce:
        block = Block(previous_hash, transactions, timestamp, nonce)
        nonces_tested += 1

        if progress_callback and nonces_tested % 100000 == 0:
            progress_callback(nonce, block.hash)

        if block.hash.startswith(required_prefix):
            return block, nonces_tested

    nonce += 1

return None, nonces_tested

```

Листинг 3: Секвенцијална функција за рударење у *Python*-у.

Анализа функције `mine_block`:

- На почетку се дефинише `required_prefix` који представља циљни услов који хеш мора да задовољи
- `while` петља се извршава бесконачно док се не пронађе решење
- У свакој итерацији креира се нова инстанца класе `Block` са тренутном попсе вредношћу. Хеш се аутоматски рачуна приликом иницијализације објекта користећи `hashlib` библиотеку
- Проверава се да ли израчунати хеш почиње са захтеваним префиксом (`block.hash.startswith(required_prefix)`)
- Ако је услов испуњен, функција враћа пронађени блок и укупан број тестираних попсе вредности
- У супротном, попсе се инкрементира за 1 и процес се наставља

Овај приступ, иако функционалан, ограничен је перформансама самог *Python* интерпретера. Као интерпретирани језик, *Python* извршава код линију по линију, што је

значајно спорије од компајлираних језика за рачунарски интензивне задатке као што је хеширање.

3.2.2 Имплементација у програмском језику Rust

Имплементација у *Rust*-у прати исту “*brute-force*” логику, али са значајним предностима у перформансама јер је *Rust* компајлирани језик који генерише оптимизован машински код. Централна логика је у функцију `mine_block`.

```
fn mine_block(
    previous_hash: String,
    timestamp: i64,
    data: String,
    difficulty: usize,
) -> (Block, u64) {
    let mut nonce = 0u64;
    let mut attempts = 0u64;

    loop {
        let block = Block::new(previous_hash.clone(), timestamp, nonce,
data.clone());
        attempts += 1;

        if block.meets_difficulty(difficulty) {
            return (block, attempts);
        }

        nonce += 1;
    }
}
```

Листинг 4: Секвенцијална функција за рударење у *Rust*-у.

Функција користи бесконачну `loop` петљу. У свакој итерацији:

1. Иницијализује се нова инстанца `Block` структуре са тренутном `nonce` вредношћу
2. Бројач покушаја (`attempts`) се инкрементира
3. Позива се метода `meets_difficulty()` над блоком како би се проверило да ли његов хеш задовољава услов
4. Ако је услов испуњен, петља се прекида и функција враћа пронађени блок заједно са укупним бројем покушаја
5. Ако услов није испуњен, `nonce` се инкрементира за 1 и петља наставља са следећом итерацијом

Ова имплементација је директна, меморијски ефикасна и представља чисту демонстрацију рачунарског рада који је потребно уложити да би се пронашло решење у *Proof-of-Work* систему. Њене перформансе ће послужити као референтна тачка за

мерење ефикасности паралелне верзије у *Rust*-у. Ова имплементација је директна и меморијски ефикасна и демонстрира рачунарски напор потребан за проналажење решења у *Proof-of-Work* систему. Њено извођење чини основу за процену ефикасности паралелне имплементације.

3.3 Proof-of-Work: Паралелна имплементација

Прелазак са секвенцијалног на паралелни модел извршавања је кључан корак ка постизању високих перформанси у *Proof-of-Work* рударењу. С обзиром на то да је процес провере сваке нонс вредности потпуно независан од осталих, проблем је инхерентно паралелизабилан. Основни циљ је расподелити целокупан простор нонс вредности на више радника (енг. *workers*) како би се претрага одвијала истовремено на свим доступним процесорским језгрима.

Иако је циљ исти, приступ паралелизацији се суштински разликује у програмским језицима *Python* и *Rust*, пре свега због њихових фундаменталних разлика у управљању конкурентним извршавањем.

3.3.1 Имплементација у програмском језику Python

Због ограничења које намеће Глобални интерпретерски катанац (енг. Global Interpreter Lock, GIL), стандардне нити у *Python*-у не могу истовремено извршавати CPU-интензиван код. Стога је за праву паралелизацију неопходно користити модул `multiprocessing`, који заобилази GIL тако што креира засебне процесе уместо нити. Сваки процес добија сопствени *Python* интерпретер и меморијски простор. Иако је овај приступ ефикасан за паралелизацију, он уводи додатно оптерећење у виду сложеније комуникације и синхронизације између процеса.

У овој имплементацији, креиран је скуп радничких процеса (енг. *Pool*) којима се додељује посао. Срж логике сваког радника налази се у функцији `worker_mine`

```
def worker_mine(
    worker_id, previous_hash, transactions_data, timestamp,
    difficulty, start_nonce, step, found_flag, result_queue
):
    # ... (inicijalizacija)
    nonce = start_nonce

    while True:
        block = Block(previous_hash, transactions, timestamp, nonce)
        nonces_tested += 1
        if block.hash.startswith(required_prefix):
            found_flag.value = 1
            result = { "found": True, "block_data": ..., "nonce": nonce, ... }
            result_queue.put(result)
            return result

        if nonces_tested % check_interval == 0:
            if found_flag.value == 1:
                result = {
                    "worker_id": worker_id,
                    "found": False,
                    "nonces_tested": nonces_tested,
                }
                result_queue.put(result)

        return result
    nonce += step
```

Листинг 5: Функција `worker_mine` за паралелно рударење у *Python*-у

Кључни аспекти ове имплементације су:

- **Подела посла:** Простор нонс вредности је подељен тако да сваки радник обрађује само део. Главни процес додељује сваком раднику јединствену почетну нонс вредност (`start_nonce = worker_id`), а сваки радник затим инкрементира свој нонс за укупан број радника (`step = num_workers`). Овим се гарантује да неће доћи до преклапања и дуплог рада.
- **Синхронизација и прекид рада:** Да би се избегао непотребан рад након што један од радника пронађе решење, користе се два механизма из `multiprocessing` модула:
 - ▶ `found_flag = manager.Value("i", 0)`: Дељена променљива која служи као заставица. Када радник пронађе решење, поставља вредност ове заставице на 1.
 - ▶ `result_queue = manager.Queue()`: Дељени ред у који радници смештају своје резултате.

Сваки радник периодично проверава вредност `found_flag`. Уколико примети да је вредност 1, прекида свој рад и завршава извршавање. Овај механизам обезбеђује “рано заустављање” (енг. *early termination*).

3.3.2 Имплементација у програмском језику Rust

Rust омогућава знатно ефикаснију паралелизацију захваљујући свом моделу власништва који гарантује сигурност при раду са меморијом без потребе за “garbage collector”-ом или GIL-ом. У овом пројекту коришћена је библиотека *Rayon*. Имплементира “*work-stealing*” распоређивач који омогућава веома ефикасну расподелу посла међу нитима уз минимално оптерећење.

Целокупна паралелна логика је смештена унутар функције `mine_block_parallel`, где се користи *Rayon* креатор скупа нити (енг. *ThreadPoolBuilder*)

```
rayon::ThreadPoolBuilder::new()
    .num_threads(num_workers)
    .build()
    .unwrap()
    .install(|| {
        (0..num_workers).into_par_iter().for_each(|thread_id| {
            // ... (иницијализација)

            while !found.load(Ordering::Relaxed) {
                let block = Block::new(previous_hash.clone(), timestamp,
nonce, data.clone());
                local_attempts += 1;
                if block.meets_difficulty(difficulty) {
                    if !found.swap(true, Ordering::SeqCst) {
                        let mining_result = MiningResult {
                            block,
                            attempts: local_attempts,
                            thread_id,
                        };
                        *result.lock() = Some(mining_result);
                    }
                    break;
                }

                nonce += num_workers as u64;
            }
            // чување података за анализу...
        });
    });
});
```

Листинг 6: Паралелна петља за рударење у Rust-у коришћењем библиотеке *Rayon*

Кључни аспекти ове имплементације су:

- **Подела посла:** Примењена је идентична стратегија подели нонс простора као у *Python* имплементацији. Свака нит, идентификована својим `thread_id`, добија јединствену почетну нонс вредност (`nonce = thread_id as u64`) и инкрементира је за укупан број радника (`nonce += num_workers as u64`). Ово омогућава директно и фер поређење ефикасности два приступа.
- **Синхронизација и прекид рада:** Уместо тежих механизама за комуникацију између процеса, *Rust* користи атомске операције и мутексе за синхронизацију међу нитима:
 - ▶ `found = Arc<AtomicBool::new(false)>`: Дељена атомска променљива која служи као заставица за заустављање. Атомске операције су изузетно брзе јер се извршавају као једна, непрекидна инструкција на нивоу хардвера. Свака нит у петљи проверава ову заставицу са `found.load(Ordering::Relaxed)`.
 - ▶ `if !found.swap(true, Ordering::SeqCst)`: Ова линија представља кључни део механизма. `swap` је атомска “*test-and-set*” операција која истовремено поставља вредност заставице на `true` и враћа њену претходну вредност. Само она нит која прва изврши ову операцију ће добити `false` као повратну вредност и ући у `if` блок како би сачувала резултат. Све остале нити које стигну касније ће видети да је вредност већ `true`, неће ући у блок и само ће прекинути свој рад.
 - ▶ `result: Arc<parking_lot::Mutex<...>>`: Резултат се чува у меморијској локацији заштићеној мутексом, како би се гарантовао сигуран упис од стране победничке нити. `parking_lot` је алтернативна имплементација мутекса која има мањи меморијски *overhead* и боље перформансе.

3.4 Имплементација *Proof-of-Stake* симулације

За разлику од рачунарски интензивног *Proof-of-Work* алгоритма, *Proof-of-Stake* се фокусира на економски подстицај и процес валидације. Имплементација у овом раду представља симулацију PoS консензуса у *Rust*-у, дизајнирану да демонстрира кључне механизме и измери перформансе система. Цео процес се може поделити на два главна корака, избор валидатора који предлаже блок, и паралелна валидација тог блока од стране свих учесника у мрежи.

3.4.1 Избор предлагача блока

Први корак у свакој рунди креирања блока је одређивање који валидатор има право да га предложи. Уместо такмичења, овај процес је заснован на алгоритму пондерисаног случајног избора (енг. *weighted random selection*), где је тежина сваког валидатора одређена величином његовог улога (*stake*). Ова логика је имплементирана у функцији `select_validator_weighted`.

```

pub fn select_validator_weighted(validators: &[Validator]) -> &Validator {
    let mut rng = rand::thread_rng();
    let total_stake: u64 = validators.iter().map(|v| v.stake).sum();
    let mut random_stake = rng.gen_range(0..total_stake);

    for validator in validators {
        if random_stake < validator.stake {
            return validator;
        }
        random_stake -= validator.stake;
    }

    &validators[0]
}

```

Листинг 7: Функција за пондерисани избор валидатора у Rust-у.

Функција прво израчунава укупан улог свих валидатора у мрежи. Затим генерише случајан број у опсегу од 0 до укупног улога. Након тога, пролази кроз листу валидатора и од сваког редом одузима његов улог од генерисаног броја. Први валидатор код којег је преостала вредност случајног броја мања од његовог улога бива изабран. Овај алгоритам осигурује да валидатори са већим улогом имају пропорционално већи шансу да буду изабрани, што је кључно за PoS систем.

3.4.2 Паралелна валидација блока

Након што је валидатор изабран да предложи блок, он га попуњава трансакцијама и шаље остатку мреже. У овој симулацији, сви валидатори истовремено започињу процес валидације предложеног блока. Овај корак је имплементиран као “трка”, први валидатор који успешно заврши валидацију свих трансакција у блоку јавља мрежи, након чега се блок сматра потврђеним.

Процес валидације је рачунарски захтеван. У методи `Transaction::validate()`, симулира се криптографски рад (као што је провера дигиталног потписа) извршавањем великог броја SHA-256 хеш итерација за сваку трансакцију. Ово омогућава да се измери утицај паралелизације на укупно време валидације.

Паралелна валидација је реализована коришћењем *Rayon* библиотеке, слично као у PoW имплементацији. Главна петља у функцији `run_pos_consensus` покреће паралелну итерацију над свим валидаторима, позивајући функцију `validate_block_parallel` за сваког од њих.

```

fn validate_block_parallel(
    transactions: &[Transaction],
    validator_id: u32,
    block_ready: Arc<AtomicBool>,
) -> (f64, u32, bool) {
    let start = Instant::now();

    if block_ready.load(Ordering::Relaxed) {
        return (start.elapsed().as_secs_f64() * 1000.0, validator_id,
false);
    }

    let all_valid = transactions.iter().all(|tx| {
        if block_ready.load(Ordering::Relaxed) {
            return false;
        }
        tx.validate()
    });

    let elapsed_ms = start.elapsed().as_secs_f64() * 1000.0;

    if all_valid && !block_ready.swap(true, Ordering::SeqCst) {
        return (elapsed_ms, validator_id, true);
    }

    (elapsed_ms, validator_id, false)
}

```

Листинг 8: Паралелна валидација блока у PoS симулацији.

Слично као у паралелној PoW имплементацији, за синхронизацију се користи атомска заставица `block_ready: Arc<AtomicBool>`.

- Сваки валидатор (нит) прво проверава да ли је блок већ спреман (`block_ready.load`). Ако јесте, одмах прекида рад.
- Унутар `transactions.iter().all()`, који проверава да ли су све трансакције валидне, такође се проверава заставица како би се процес прекинуо што је раније могуће.
- Први валидатор који успешно заврши валидацију свих трансакција користи атомску `swap` операцију (`!block_ready.swap(true, Ordering::SeqCst)`). Само тај валидатор ће проћи ову проверу и вратити резултат `success: true`. Сви остали ће видети да је заставица већ постављена и завршиће свој рад без проглашења успеха.

Овакав “racing” модел ефикасно користи све доступне процесорске ресурсе за убрзавање процеса валидације, што резултира знатно краћим временом креирања блока у поређењу са PoW системом. Уместо нонс, у `Block` структуру се уписује `id` валидатора који је предложио блок.

Глава 4

Експерименти и анализа резултата

Ово поглавље представља експерименталну валидацију и анализу перформанси имплементација *Proof-of-Work* алгоритма. Циљ је измерити утицај паралелизације на брзину извршавања и проценити скалабилност система у различитим условима. Кроз тестове јаког и слабог скалирања врши се анализа резултата имплементација у програмским језицима *Python* и *Rust*.

Резултати представљени у овом поглављу имају за циљ да покажу практичне предности паралелног приступа. Анализа ће истаћи кључне разлике у перформансама између самих програмских језика и њихових модела за конкурентно извршавање.

4.1 Спецификација система и методологија

Да би се обезбедила поновљивост и валидност добијених резултата, сви експерименти су спроведени у идентичном окружењу.

Конфигурација система

Сва мерења су извршена на рачунару са следећим хардверским и софтверским спецификацијама:

- **Процесор:** AMD Ryzen 5 5600X (6 језгара, 12 логичких процесора, 3.70GHz)
- **Кеш меморија:** L1-384KB, L2-3.0MB, L3-32.0MB
- **RAM меморија:** 16GB DDR4 на 3200MHz
- **Оперативни систем:** Windows 10
- **Python верзија:** 3.10.5
- **Rust (Cargo) верзија:** 1.87.0 (99624be96 2025-05-06)
- **Кључне библиотеке:** Rayon = “1.10”, Python multiprocessing

Методологија тестирања

Експериментална анализа је усмерена искључиво на *Proof-of-Work* алгоритам, с обзиром на то да је његов “*brute-force*” приступ рачунарски најзахтевнији и најпогоднији за мерење убрзања. Тестирање је подељено у два главна експеримента:

- **Експеримент јаког скалирања:** Циљ је био измерити како се време извршавања смањује са повећањем броја процесорских ресурса за проблем фиксне величине. Величина проблема је била фиксирана на тежину рударења $d=5$, док је број радника варирао у корацима: 1, 2, 4, 8 и 12.

- **Експеримент слабог скалирања:** Циљ је био тестирати способност система да обради већи проблем са повећањем броја ресурса, уз очекивање да време извршавања остане приближно константно. Идеалан начин за скалирање проблема у PoW систему био би пропорционално повећање тежине (*difficulty*). Међутим, због експоненцијалне природе овог параметра, где повећање тежине за 1 повећава обим после 16 пута, такав експеримент би захтевао значајно већи број процесорских језгара него што је било доступно. Примењен је алтернативни приступ где се величина проблема скалира линеарно повећањем броја блокова који се рударе. Величина проблема је расла пропорционално броју радника:
 - **Основни случај:** 1 радник рудари 5 блокова.
 - **Скалирани случајеви:** 2 радника рударе 10 блокова, 4 радника рударе 20 блокова, итд., одржавајући константан обим после по раднику (5 блокова).

Ради постизања статистичке поузданости, свака конфигурација у оба експеримента је покренута 30 пута. У анализи су коришћене средње вредности времена извршавања, на основу којих су израчунате метрике убрзања и ефикасности.

4.2 Експеримент јаког скалирања

У експерименту јаког скалирања, величина проблема је остала фиксна док се број доступних процесорских ресурса повећавао. Циљ овог теста био је да се измери како се време извршавања смањује и да се израчуна постигнуто убрзање (*Speedup*). За овај експеримент, тежина проблема је била фиксирана на $d=5$, док је број радника вариран кроз вредности 1, 2, 4, 8 и 12. Свака конфигурација је извршена 30 пута како би се обезбедила статистичка поузданост резултата.

Добијени резултати су упоређени са теоријским моделом Амдаловог закона. На основу експерименталних података, процењено је да серијски део програма, који обухвата операције попут припреме блока и синхронизације, чини приближно 2% укупног времена извршавања ($s = 0.02$).

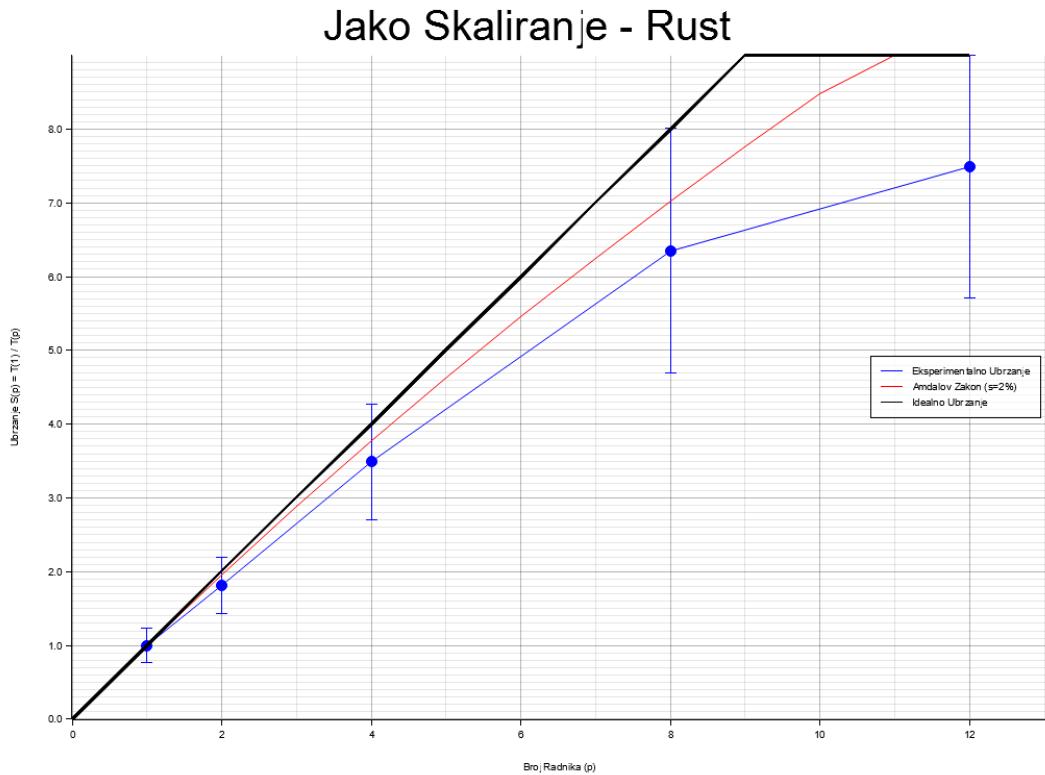
4.2.1 Резултати за *Rust* имплементацију

Rust имплементација је показала одличне карактеристике јаког скалирања. Резултати мерења су приказани у табели 1.

Табела 1: Резултати јаког скалирања за *Rust* имплементацију (тежина d=5).

Радници (p)	Средње време (s)	Стандардна девијација	Убрзање
1	28.08	6.47	1.00x
2	15.47	3.24	1.81x
4	8.04	1.81	3.49x
8	4.42	1.15	6.35x
12	3.75	0.89	7.49x

Анализа резултата показује да се експериментално добијено убрзање веома добро поклапа са идеалним линеарним убрзањем све до 4 радника. Ово указује на веома ниско додатно оптерећење (енг. *overhead*) паралелизације и ефикасно коришћење доступних процесорских језгара које омогућава *Rayon* библиотека. Након 4 радника, убрзање почиње да одступа од идеалног, али и даље близко прати теоријску криву дефинисану Амдаловим законом (слика 1). Пад ефикасности постаје израженији при преласку са 8 на 12 радника, што сугерише да се систем за дату величину проблема приближава тачки засићења, где трошкови синхронизације и комуникације између нити постају значајнији фактор.



Слика 1: Графички приказ јаког скалирања за *Rust* имплементацију

4.2.2 Резултати за *Python* имплементацију

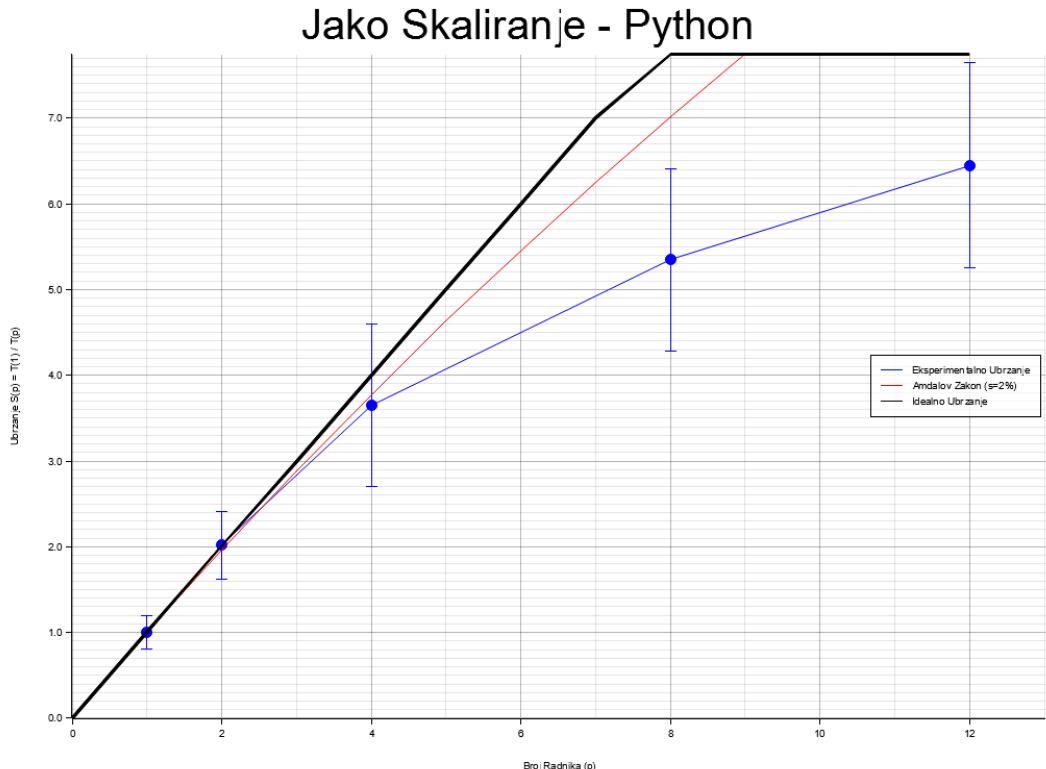
Python имплементација је такође постигла значајно убрзање, што потврђује да је проблем инхерентно паралелизован. Резултати мерења су приказани у табели 2.

Табела 2: Резултати јаког скалирања за *Python* имплементацију (тежина d=5).

Радници (p)	Средње време (s)	Стандардна девијација	Убрзање
1	533.13	102.10	1.00x
2	263.90	51.49	2.02x
4	146.00	38.06	3.65x
8	99.74	19.48	5.34x
12	82.67	15.34	6.45x

Почетно скалирање са 2 и 4 радника је готово савршено. Убрзање од 2.02x са два радника је вероватно последица статистичке варијације, где је у 30 мерења просечно време било нешто боље од идеалног. Међутим, након 4 радника, пад ефикасности је знатно драстичнији у поређењу са *Rust* имплементацијом (слика 2). Главни узрок

овоме је додатно оптерећење које уводи `multiprocessing` библиотека. За разлику од *Rust*-ових лаких нити, *Python* креира знатно теже процесе. Комуникација између ових процеса захтева серијализацију и десеријализацију података (*pickling*), што је рачунарски скупа операција која постаје уско грло при већем броју радника.



Слика 2: Графички приказ јаког скалирања за *Python* имплементацију

4.2.3 Упоредна анализа резултата

Директно поређење две имплементације открива неколико кључних разлика:

- Апсолутне перформансе:** Најзначајнија разлика је у основној брзини извршавања. Секвенцијална *Rust* имплементација (28.08s) је приближно 19 пута бржа од секвенцијалне *Python* имплементације (533.13s). Ова разлика произилази из природе компајлираног језика који генерише оптимизован машински код, наспрот интерпретираном језику, као и *Rust*-ове контроле над меморијом на ниском нивоу.
- Ефикасност скалирања:** Иако обе имплементације скалирају, *Rust* то ради знатно ефикасније при већем броју језгара. Модел паралелизације у *Rust*-у, који користи дељену меморију и лаке нити, показује се супериорним за CPU-интензивне задатке у односу на *Python*-ов модел са више процеса и скупом серијализацијом података.

- Утицај додатног оптерећења:** Резултати за *Python* показују да оптерећење које уводе платформа и коришћене библиотеке може постати доминантан фактор који ограничава скалабилност. Код *Rust* имплементације, ово оптерећење је знатно мање изражено.

4.3 Експеримент слабог скалирања

У експерименту слабог скалирања, испитује се способност система да реши већи проблем са повећањем броја процесорских ресурса, уз очекивање да време извршавања остане приближно константно. Због експоненцијалне природе *PoW* тежине, која би захтевала велики број језгара, примењен је алтернативни приступ где се величина проблема скалира повећањем укупног броја блокова који се рударе. Обим посла по раднику је одржаван константним, 5 блокова по раднику.

Резултати се анализирају кроз метрику скалираног убрзања, а упоређују се са теоријским моделом Густафсоновог закона.

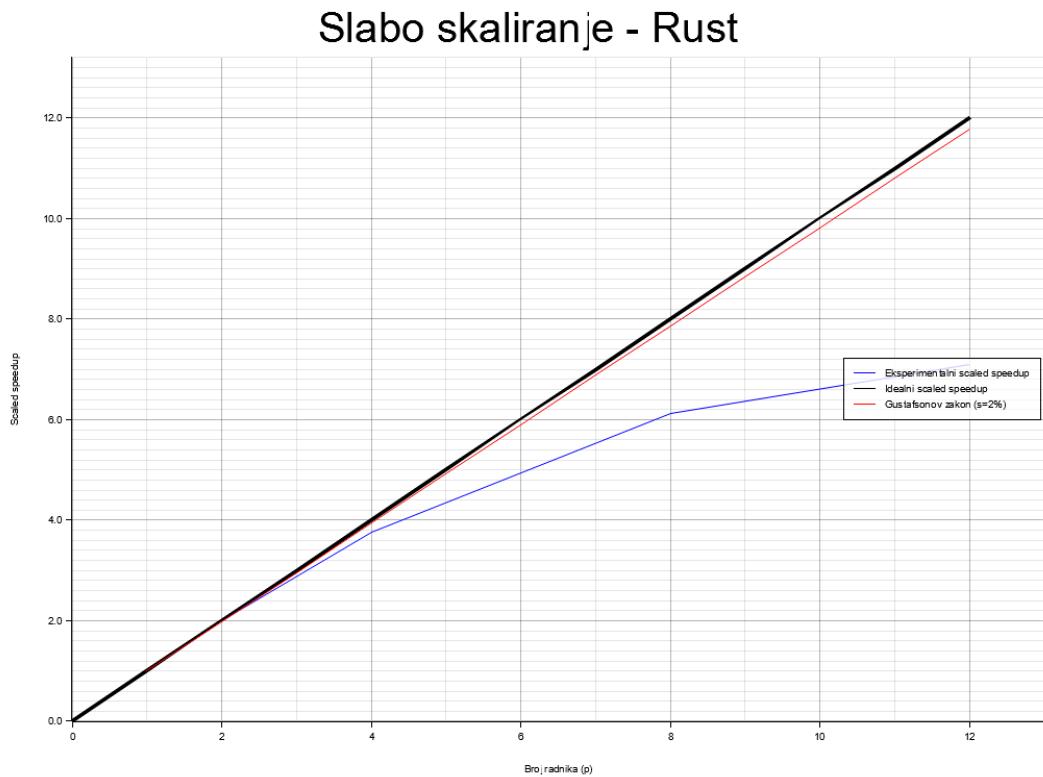
4.3.1 Резултати за *Rust* имплементацију

Rust имплементација је показала добре карактеристике слабог скалирања, иако не савршене.

Табела 3: Резултати слабог скалирања за *Rust* имплементацију

Радници (p)	Број блокова	Средње време (s)	Стандардна девијација (s)
2	10	0.4862	0.1215
4	20	0.5163	0.1033
8	40	0.6357	0.1057
12	60	0.8218	0.0849

Резултати показују да време извршавања остаје релативно стабилно при мањем броју радника. Са повећањем броја радника на 8 и 12, приметан је пораст времена извршавања. Овај пораст указује на то да систем не скалира са савршеном ефикасношћу.



Слика 3: Графички приказ слабог скалирања за *Rust* имплементацију

График за слабо скалирање *Rust* имплементације (слика 3) јасно приказује ово понашање. Експериментална линија скалираног убрзаша благо одступа од идеалне и теоријске линије, са падом ефикасности који постаје значајнији како се број нити повећава. Овај пораст времена извршавања може се повезати са додатном оптерећењу које настаје услед синхронизације и управљања већим бројем нити.

4.3.2 Резултати за *Python* имплементацију

Python имплементација показује знатно веће проблеме у слабом скалирању. Време извршавања не остаје константно већ расте са сваким повећањем броја радника и величине проблема.

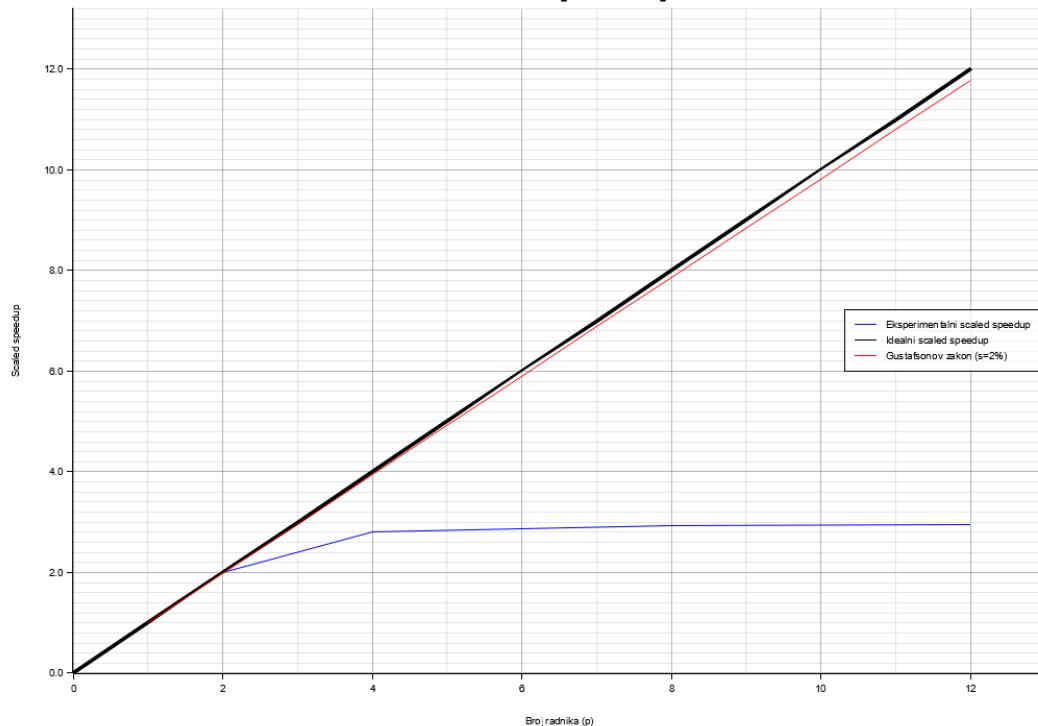
4.3 Експерименћ слабог скалирања

Табела 4: Резултати слабог скалирања за *Python* имплементацију

Радници (p)	Број блокова	Средње време (s)	Стандардна девијација (s)
2	10	9.3645	2.6114
4	20	13.3507	2.1815
8	40	25.5783	2.1543
12	60	38.1508	1.8488

Време потребно за рударење 60 блокова са 12 радника је преко четири пута дуже од времена потребног за рударење 10 блокова са 2 радника, што указује на веома лошу ефикасност слабог скалирања.

Slabo skaliranje - Python



Слика 4: Графички приказ слабог скалирања за *Python* имплементацију

На графику за *Python* (слика 4) се овај проблем јасно види. Експериментална линија скалираног убрзања стагнира већ након 2 радника, показујући огромно одступање од идеалног скалирања. Ово указује на то да систем не успева да ефикасно искористи додатне ресурсе за обраду већег проблема.

Главни узрок овако лошег скалирања је велико додатно оптерећење *Python multiprocessing* модула. Са повећањем броја радника и количине посла, ово оптерећење расте брже од користи која се добија паралелизацијом, што доводи до драстичног пада ефикасности. У директном поређењу, *Rust* показује значајно супериорније карактеристике слабог скалирања. Ово потврђује да је његов модел са лаким нитима и дељеном меморијом далеко погоднији за овакав проблема.

4.4 Поређење *Python* и *Rust*

Резултати експеримената јаког и слабог скалирања омогућавају анализу перформанси *Proof-of-Work* имплементација у програмским језицима *Python* и *Rust*. Обе имплементације успешно користе паралелизацију за убрзавање процеса рударења, али разлике у њиховим основним карактеристикама и моделима за конкурентно извршавање доводе до различитих резултата. Анализа се може поделити на три кључна аспекта: апсолутне перформансе, ефикасност скалирања и утицај додатног оптерећења.

4.4.1 Апсолутне перформансе

Највећа разлика између две имплементације је у основној брзини извршавања. Као што је показано у експерименту јаког скалирања, секвенцијална *Rust* имплементација је завршила задатак за просечно 28.08 секунди. У идентичним условима, секвенцијалној *Python* имплементацији је било потребно 533.13 секунди. То значи да је *Rust* верзија приближно 19 пута бржа.

Ова разлика је директна последица природе два језика. Главни део рачунарског рада у PoW алгоритму је константно израчунавање хешева. Као компајлирани језик, *Rust* преводи ове операције у високо оптимизован машински код који се извршава на самом процесору. *Python* као интерпретирани језик сваку операцију прослеђује кроз неколико слојева апстракције унутар свог интерпретера пре него што се она изврши. У задатку који се састоји од милиона понављања једне те исте криптографске операције, ово додатно оптерећење *Python* интерпретера долази до изражaja и даје значајно лошије перформансе.

4.4.2 Ефикасност скалирања

Док апсолутне перформансе показују колико је један језик бржи од другог у основи, ефикасност скалирања показује колико добро сваки од њих користи додатне процесорске ресурсе.

У експерименту јаког скалирања, обе имплементације су показале добро почетно скалирање. Међутим, док *Rust* одржава високу ефикасност и при већем броју радника, *Python* показује значајно бржи пад ефикасности.

Разлике постају још драстичније у експерименту слабог скалирања. *Rust* показује благи пораст времена извршавања, што указује на добро, али не савршено скалирање. *Python* не успева да одржи константно време извршавања.

4.4.3 Утицај модела паралелизације

Узрок овако различитог понашања у скалирању лежи у различitim моделима паралелизације које ова два језика користе.

- **Rust: Лаке нити и дељена меморија** Имплементација у *Rust*-у користи праве системске нити којима управља библиотека *Rayon*. Ове нити се извршавају унутар истог процеса и деле исти меморијски простор. Комуникација и синхронизација међу њима се одвијају преко изузетно ефикасних, хардверских убрзаних атомских операција. *Rayon* такође имплементира “*work-stealing*” алгоритам, где незапослене нити краду посао од оптерећених, што обезбеђује оптималну искоришћеност свих језгара. Додатно оптерећење овог модела је минимално, што омогућава одлично скалирање.
- **Python: Тешки процеси и серијализација** Због GIL-а, *Python* за CPU-интензивне задатке мора да користи *multiprocessing*, који покреће потпуно одвојене процесе. Сваки процес има свој меморијски простор, а комуникација између њих захтева да се подаци серијализују у главном процесу, пошаљу преко оперативног система, а затим десеријализују у радничком процесу. Овај процес је рачунарски веома скуп и постаје уско грло како се повећава број радника и количина података који се разменjuју. Управо је ово оптерећење главни разлог зашто ефикасност *Python* имплементације драстично опада, нарочито у експерименту слабог скалирања где се обрађује већа количина послана.

Док *Python* нуди једноставност и брзину развоја, *Rust* показује супериорност за развој високо перформанских, паралелних система. У контексту решавања проблема као што је *Proof-of-Work* рударење, предности које нуди компајлирани програмски језик напредним моделом паралелизације су неоспорне.

4.5 Анализа перформанси *Proof-of-Stake* симулације

За разлику од *Proof-of-Work*, где је перформанса система директно везана за рачунарску снагу и брзину хеширања, *Proof-of-Stake* систем је ограничен брзином валидације трансакција и мрежном комуникацијом. Због тога се на њега не могу применити класични експерименти скалирања на исти начин. Уместо мерења убрзаша “*brute-force*” претраге, кључне метрике постају *пројусносћ* (енг. *throughput*) и време креирања блока.

У оквиру овог рада, извршена су мерења перформанси имплементиране PoS симулације у *Rust*-у.

4.5.1 Резултати мерења

Извршена су два сета мерења, један са 8 и један са 12 симулираних валидатора.

Табела 5: Резултати перформанси за PoS симулацију

Број валидатора	Укупно блокова	Укупно трансакција	Пропусност (TPS)	Просечно време по блоку (ms)
8	20	100	1716.21	2.91
12	30	150	1444.20	3.46

Анализа резултата показује изузетно високе перформансе. У конфигурацији са 8 валидатора, систем је постигао пропусност од преко 1700 трансакција у секунди (TPS), док је просечно време потребно за валидацију и креирање једног блока било мање од 3 милисекунде. Са повећањем броја валидатора на 12, долази до благог пада пропусности и повећања времена по блоку, што је очекивано понашање због већег оптерећења система услед координације већег броја нити. Ипак, перформансе остају на веома високом нивоу.

Анализа статистике валидатора потврђује да механизам пондерисаног избора исправно функционише. Као што се види, иако случајност игра улогу, постоји јасна корелација где су валидатори са већим улогом (*stake*), попут валидатора 10 (улог 852), чешће бирани да предложе блок, док је валидатор са најмањим улогом (*ID* 4, улог 323) био изабран нула пута.

Табела 6: Пример статистике избора валидатора у симулацији са 12 учесника

ID Валидатора	Улог (stake)	Број избора
10	852	5
3	843	4
7	541	4
2	992	3
8	832	3
6	646	3
9	747	2
11	607	2
5	443	2
0	578	1
1	635	1
4	323	0

4.5.2 Анализа PoW наспрам PoS

Директно поређење ових резултата са перформансама PoW система открива фундаменталне разлике у архитектури и ефикасности ова два приступа:

- Брзина креирања блока:** У PoW систему, време креирања блока је намерно отежано и у експериментима се мерило у десетинама секунди. У PoS симулацији, исто време се мери у милисекундама. Ова разлика од неколико редова величине је највећа предност PoS система. Она произилази из чињенице да PoS елиминише потребу за рачунарски скрупим и спорим процесом рударења.
- Пропусност:** Због изузетно брзог креирања блокова, PoS систем може да обради значајно већи број трансакција у секунди. Док је PoW систем у овом раду имао ефективну пропусност мању од 1 TPS, PoS симулација је показала капацитет од преко 1400 TPS. Ово чини PoS далеко погоднијим за апликације које захтевају брзе и јефтине трансакције, као што су системи за плаћање или децентрализоване апликације високе фреквенције.
- Потрошња ресурса:** Иако није директно мерена, разлика у потрошњи ресурса је очигледна. PoW систем је током рударења константно користио 100% свих доступних CPU језгара. Са друге стране, PoS систем користи ресурсе само у кратким интервалима током процеса валидације, док је већину времена у стању мировања. Ово потврђује теоријску претпоставку да је PoS значајно енергетски ефикаснији.

У закључку, иако је имплементирани PoS систем симулација, он јасно демонстрира огромне предности у перформансама у односу на *Proof-of-Work*. Елиминисањем “*brute-force*” рударења, PoS омогућава креирање блокчејн система који су знатно бржи, ефикаснији и скалабилнији, што отвара врата за много шири спектар практичних примена.

Глава 5

Дизајн и имплементација P2P мреже

Након детаљног објашњења како консензус алгоритми раде самостално, у овом поглављу ће се објаснити како се они уклапају у реалну P2P мрежу, која има више чворова који комуницирају међусобно. Да би блокчејн систем био заиста децентрализован, неопходно је успоставити механизме који омогућавају независним чворовима (енг. *nodes*) да се међусобно проналазе, комуницирају, синхронизују податке и заједнички учествују у постизању консензуса.

5.1 Архитектура мреже

Да би се омогућила флексибилност и лака замена мрежних топологија, целокупна мрежна логика је апстрахована кроз NetworkLayer који дефинише основни уговор који свака мрежна имплементација мора да испуни.

```
pub trait NetworkLayer: Send + Sync {  
    fn broadcast(&self, message: &P2PMessage) -> Result<(), NetworkError>;  
    fn send_to(&self, node_id: &str, message: &P2PMessage) -> Result<(),  
    NetworkError>;  
    fn get_connected_peers(&self) -> Vec<String>;  
}
```

Листинг 9: NetworkLayer у Rust-у.

Овај приступ омогућава да се у раду истраже и имплементирају два фундаментално различита модела мрежне топологије: централизована звезда (*Star*) топологија и децентрализована мрежа (*Mesh*) топологија.

5.1.1 Звезда (енг. *Star*) топологија

Звезда топологија представља једноставнији, централизовани приступ. У овој архитектури, један чвор, назван *BootstrapNode*, има улогу централног сервера. Сви остали чворови у мрежи, *RegularNode*, успостављају директну и једину конекцију са овим централним чворма.

Комуникација се одвија на следећи начин:

1. *RegularNode* (клијент) шаље поруку ка *BootstrapNode* (серверу).
2. *BootstrapNode* прима поруку, обрађује је и прослеђује је свим осталим повезаним клијентима користећи своју *broadcast* методу.

Овај модел је имплементиран кроз две структуре:

- **StarNetworkServer:** Користи се од стране BootstrapNode чвора за ослушкивање долазећих конекција и управљање листом повезаних клијената.
- **StarNetworkClient:** Користи се од стране RegularNode чворова за успостављање и одржавање конекције са централним сервером.

Главна предност ове топологије је једноставност имплементације и управљања мрежом. Идеја је била да се покаже конкурентност у процесу рударења између више чворова и постизање консензуса. Али ова топологија има ману, она уводи централну тачку отказа (енг. *single point of failure*). Уколико BootstrapNode престане са радом, целокупна мрежа постаје нефункционална.

5.1.2 Мрежа (енг. mesh) топологија

Као потпуно децентрализована алтернатива, имплементирана је и мрежа топологија. У овом моделу не постоји централни сервер. Сваки чвор (*MeshNode*) директно се повезује са више других чворова у мрежи, формирајући децентрализовану мрежасту структуру.

Улога BootstrapNode чвора је у овом случају сведена на сервис за откривање (енг. *discovery service*):

1. Нови MeshNode се приликом покретања привремено повезује са BootstrapNode чворма.
2. BootstrapNode му шаље листу адреса осталих активних MeshNode чворова у мрежи.
3. Након добијања листе, нови чвор прекида везу са BootstrapNode-ом и започиње процес директног повезивања са другим чворовима са добијене листе.

Да би се избегла фрагментација мреже, где би се могла формирати изолована острва чворова, поставља се питање колико је конекција потребно успоставити да би мрежа са великим вероватноћом остала повезана. Одговор на ово питање даје теорија случајних графова Ердеш-Рењи модел (енг. *Erdos–Renyi model*) [6]. Према овом моделу, тачка у којој граф са n чворова са високом вероватноћом прелази из неповезаног у повезан, дешава се када је просечан број конекција по чвиру приближно једнак $\ln(n)$. За очекивану величину мреже од око 20 чворова, $\ln(20) \approx 2.99$. Под том претпоставком, у овој имплементацији је изабрано да сваки нови чвор успостави три одлазне конекције, што представља практичну примену овог теоријског принципа како би се осигурала повезаност мреже.

Комуникација се одвија директно од чвора до чвора. Када један чвр жели да пошаље поруку цеој мрежи, он је шаље само својим директно повезаним суседима. Ти суседи затим прослеђују поруку својим суседима, и тако даље, док се порука не прошири кроз целу мрежу. Овај механизам пропагације се назива *Gossip* протокол. Ова топологија је знатно отпорнија на отказе, али је комплекснија за имплементацију због потребе за управљањем већим бројем конекција и спречавањем бесконачног кружења порука.

5.2 Мрежни протокол

Да би чворови у P2P мрежи могли међусобно да комуницирају на структуриран начин, неопходно је дефинисати јасан комуникациони протокол. У овој имплементацији, протокол је реализован преко TCP/IP сокета, где се све поруке серијализују у JSON формат пре слања.

Протокол је дефинисан са P2PMessage енумерацијом (енг. *enum*) која дефинише све типове порука које чворови могу размењивати.

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum P2PMessage {
    Join { node_id: String, address: String, timestamp: u64 },
    PeerList { peers: Vec<PeerInfo> },
    Heartbeat { node_id: String, timestamp: u64 },
    Pong { node_id: String },
    Disconnect { node_id: String },
    Rejected { reason: String },

    RequestBlockchain { requester_id: String },
    BlockchainSync { chain: Vec<Block> },

    MiningStart { template: BlockTemplate },
    MiningStop,
    NewBlock { block: Block, miner_id: String },

    NewTransaction { transaction: String, from_node: String },
}
```

Листинг 10: P2PMessage енумерација која дефинише комуникациони протокол

Значење и сврха кључних типова порука су следећи:

- **Управљање конекцијама:**
 - ▶ **Join:** Прва порука коју нови чвор шаље BootstrapNode чвиру како би затражио улазак у мрежу
 - ▶ **PeerList:** Порука којом BootstrapNode одговара на Join у којој шаље новом чвиру листу адреса осталих активних чворова
 - ▶ **Heartbeat и Pong:** Механизам за одржавање конекције
 - ▶ **Disconnect:** Порука којом чвор обавештава мрежу да се искључује
 - ▶ **Rejected:** Порука којом чвор одбија долазећу конекцију, на пример због достигнутог максималног броја конекција
- **Синхронизација и консензус:**
 - ▶ **RequestBlockchain:** Порука којом чвор тражи од другог чвора да му пошаље своју комплетну копију блокчејна

- ▶ **BlockchainSync:** Одговор на `RequestBlockchain` који садржи комплетан низ блокова (`Vec<Block>`)

- **Координација рударења:**

- ▶ **MiningStart:** Порука којом се иницира рударење. Порука садржи `BlockTemplate` структуру која дефинише све потребне податке за креирање новог блока
- ▶ **MiningStop:** Сигнал који се шаље свим чвровима да прекину рударење
- ▶ **NewBlock:** Порука којом рудар који је пронашао блок емитује (енг. *broadcast*) нови блок остатку мреже

- **Трансакције:**

- ▶ **NewTransaction:** Порука којом се нова трансакција додаје како би била укључена у наредни блок

5.3 Мрежни процеси

Функционисање P2P мреже се заснива на неколико кључних процеса који омогућавају чвровима да се приједоре, одржавају усаглашеност и заједнички доприносе расту блокчејна. У овом раду, два најважнија процеса су синхронизација новог чвора са мрежом и циклус дистрибуираног рударења.

5.3.1 Прикључивање и синхронизација новог чвора

Када се нови чвор покрене, он не поседује никакве информације о тренутном стању мреже. Први корак је да се повеже са мрежом и синхронизује. Овај процес се одвија кроз следеће кораке:

1. **Проналажење улазне тачке:** Нови чвор се прво повезује са унапред познатом адресом `BootstrapNode` чвора, који служи као почетна тачка за улазак у мрежу.
2. **Захтев за информацијама:** Чвор шаље `Join` поруку како би се представио, а затим и `RequestBlockchain` поруку како би затражио актуелну верзију блокчејна.
3. **Пријем и обрада података:**
 - У звезда топологији, `BootstrapNode` директно одговара `BlockchainSync` поруком која садржи цео ланац.
 - У мрежа топологији, `BootstrapNode` одговара `PeerList` поруком. Нови чвор затим прекида везу са `bootstrap` чвром, повезује се са чвровима са листе и од њих тражи синхронизацију блокчејна.
4. **Валидација и усвајање ланца:** По пријему, нови чвор извршава валидацију примијеног ланца позивањем функције `Blockchain::validate_chain`. Ова функција проверава повезаност, исправност хешева и поштовање PoW правила за сваки блок. Уколико је ланац валидан, чвор га усваја као своју локалну копију.

5.3.2 Циклус дистрибуираног рударења

Процес креирања новог блока је дистрибуиран и конкурентан, одвија се кроз следеће кораке:

- Иницирање рударења:** Рударење започиње када један од чворова одлучи да је време за креирање новог блока (на пример када се у `tempool`-у накупи довољан број трансакција). Тај чвор креира *BlockTemplate* и емитује *MiningStart* поруку осталим чворовима у мрежи.
 - У звезда топологији, ову улогу увек има *BootstrapNode*.
 - У мрежа топологији, било који чвор може иницирати рударење.
- Паралелно рударење:** По пријему *MiningStart* поруке, сваки чвор покреће рударење. Сви чворови у мрежи се сада истовремено такмиче ко ће први пронаћи валидан блок.
- Проналазак решења и емитовање блока:** Чвор који први пронађе валидан нонс формира комплетан блок. Он прекида рударење и емитује *NewBlock* поруку остатку мреже.
- Пропагација и прекид рада:** Остали чворови примају *NewBlock* поруку. Након што валидирају блок додају га у свој ланац и прекидају процес рударења. Ово рано заустављање осигурује да се рачунарски ресурси не троше узалуд након што је решење већ пронађено.

Овај координисани процес осигурује да се рачунарски ресурси целе мреже ефикасно користе за рударење, уз механизам који спречава непотребан рад након што је решење већ пронађено.

5.4 Постизање консензуса: правило најдужег ланца

У дистрибуираној мрежи може се дрогодити да два или више рудара пронађу решење за исти блок у приближно исто време. Када се то деси, оба чвора ће емитовати своје валидне блокове у мрежу. Чворови који су у мрежи ближи првом рудару ће прво примити његов блок, док ће други део мреже прво примити блок другог рудара. Ова ситуација доводи до привременог рачвања ланца, познатог као форк (енг. *fork*), где у мрежи истовремено постоје две различите, али валидне верзије блокчејна.

Да би се овакви конфликти разрешили и мрежа усагласила око једне верзије ланца, блокчејн системи засновани на *Proof-of-Work* алгоритму углавном користе једноставно правило најдужег ланца (енг. *longest chain rule*). Ово правило каже да се верзија блокчејна која је најдужа сматра валидном.

Овај механизам је имплементиран кроз логику за обраду долазећих *NewBlock* и *BlockchainSync* порука унутар *handle_message_static* функција у *RegularNode* и *MeshNode* структурима. Процес се одвија на следећи начин:

1. **Пријем новог блока или ланца:** Када чвор прими нови блок путем *NewBlock* поруке, он прво извршава валидацију. Уколико прими цео ланац путем *BlockchainSync* поруке, валидира се цео примљени ланац.
2. **Валидација:** Процес валидације проверава неколико услова:
 - Да ли се хеш блока поклапа са његовим садржајем.
 - Да ли *previous_hash* у новом блоку одговара хешу последњег блока у тренутном локалном ланцу.
 - Да ли хеш блока задовољава задату тежину.
3. **Одлучивање:**
 - Ако је примљени блок валидан и директно се надовезује на врх тренутног ланца, чвор га додаје на крај.
 - Ако примљени блок није валидан или се не надовезује на тренутни ланац (што указује на форк), чвор игнорише тај блок и покреће процес синхронизације. Он шаље *RequestBlockchain* поруку другим чвровима како би добио њихове верзије ланца.
 - Када чвор прими цео ланац, он прво проверава његову дужину. Ако је примљени ланац дужи од његовог тренутног, чвор покреће процес реорганизације.
4. **Реорганизација ланца:** Процес реорганизације, имплементиран у функцији *Blockchain::reorganize*, једноставно замењује тренутни локални ланац са новим, дужим ланцем.

5.5 *Gossip* протокол и пропагација порука

У децентрализованој мрежи топологији неопходно је имати ефикасан механизам за ширење информација кроз целу мрежу. У овом раду је за ту сврху имплементиран *Gossip* протокол. Основна идеја је да када чвор прими нову поруку коју није видео раније, он је прослеђује свим својим повезаним суседима. Ти суседи затим понављају исти процес, што доводи до ширења поруке кроз мрежу.

Овај механизам је имплементиран у функцији *MeshNetwork::gossip_broadcast*. Овај приступ уводи два потенцијална проблема:

1. **Дуплирање порука:** Чвор може примити исту поруку више пута од различитих суседа.
2. **Бесконачне петље:** Порука може бесконачно кружити кроз мрежу ако не постоји механизам за заустављање.

Да би се ови проблеми решили, имплементиран је механизам за праћење виђених порука, чија се логика налази у структури *MessageTracker* и функцији *compute_message_id*.

5.5.1 Механизам за спречавање дупликата

У овом протоколу спречавање дуплираних порука реализовано је кроз следеће кораке:

- Генерисање идентификатора поруке:** За сваку поруку која се шаље кроз мрежу, генерише се јединствени идентификатор.

```
pub fn compute_message_id(message: &P2PMessage) -> String {
    let serialized = serde_json::to_string(message).unwrap_or_default();

    let mut hasher = Sha256::new();
    hasher.update(serialized.as_bytes());
    let hash = hasher.finalize();

    format!("{:x}", hash)[..16].to_string()
}
```

Листинг 11: Функција за генерисање ID поруке.

- Праћење виђених порука:** Сваки *MeshNode* поседује инстанцу *MessageTracker* структуре, која у себи садржи хеш мапу (*HashMap<String, Instant>*). Ова мапа чува идентификаторе свих порука које је чвор видео, заједно са временском ознаком када их је видео.
- Процес обраде и прослеђивања:** Када *MeshNode* прими поруку он прво израчунава њен идентификатор и проверава да ли се тај идентификатор већ налази у његовом *MessageTracker*-у.
 - Ако је идентификатор већ забележен, чвор игнорише поруку и прекида даљу обраду.
 - Ако идентификатор није забележен, чвор га уписује у *MessageTracker*, обрађује поруку и прослеђује је свим својим суседима.

Овај механизам ефикасно спречава да се иста порука обрађује више пута и зауставља њено бесконачно кружење.

Глава 6

Закључак

Кроз овај рад представљен је процес пројектовања, имплементације и анализе комплетног блокчејн система, са фокусом на паралелизацију *Proof-of-Work* и *Proof-of-Stake* консензус алгоритама. Рад је за циљ имао да се анализирају перформансе и скалабилност ових алгоритама, упореде имплементације у компајлираном (*Rust*) и интерпретираном (*Python*) програмском језику, и интегришу у децентрализовану *P2P* мрежу.

Анализом резултата експеримената јаког скалирања потврђено је да паралелизација доноси значајна убрзања, али и да избор програмског језика и библиотека има пре- судан утицај на ефикасност. *Rust* имплементација се показала супериорном, како у апсолутним перформансама, тако и у способности да ефикасније искористи већи број процесорских језгара. Имплементацијом *P2P* мреже са две различите топологије и *Gossip* протоколом демонстрирани су кључни механизми потребни за функционисање дистрибуираног система.

6.1 Резултати и доприноси рада

Главни допринос овог рада се у анализи перформанси која повезује теоријске моделе паралелног рачунарства са практичном имплементацијом блокчејн алгоритама.

Кључни резултати су:

1. Развијене су потпуно функционалне секвенцијалне и паралелне имплементације *Proof-of-Work* алгоритма у програмским језицима *Rust* и *Python*, као и симулација *Proof-of-Stake* система у *Rust*-у. Ове имплементације послужиле су као основа за детаљну анализу.
2. Анализирана је скалабилност *PoW* алгоритма. Експерименти јаког скалирања су показали значајно убрзање, са *Rust* имплементацијом која је постигла убрзање до 7.49x на 12 логичких процесора. Резултати су упоређени са теоријском кривом Амдаловог закона, чиме је потврђено да оптерећење модела паралелизације постаје кључни ограничавајући фактор при већем броју језгара.
3. Демонстрирана супериорност компајлираног језика за *CPU*-интензивне задатке. Секвенцијална *Rust* имплементација је била приближно 19 пута бржа од *Python* верзије, што је директна последица ефикаснијег извршавања криптографских хеш

функција. Такође, *Rust*-ов модел са лаким нитима и *Rayon* библиотеком показао је знатно бољу ефикасност скалирања у односу на *Python*-ов *multiprocessing* модел.

4. Имплементирана је P2P мрежа са подршком за звезда и мрежа топологије. Кроз дефинисани мрежни протокол и имплементацију механизама за синхронизацију, дистрибуирано рударење и правило најдужег ланца, успешно је симулирана децентрализована мрежа. Увођењем *Gossip* протокола са механизмом за праћење порука, демонстрирано је ефикасно ширење информација у мрежа топологији.

6.2 Ограничења и правци даљег развоја

Највеће ограничење експерименталне анализе је доступност хардверских ресурса. Тестирање на систему са већим бројем физичких језгара омогућило би детаљнију анализу скалирања. Такође, имплементирани *PoS* модел представља поједностављену симулацију и не обухвата сложене механизме као што је *slashing*.

Неки од потенцијалних праваца за даљи развој су:

- **Оптимизација и GPU акцелерација:** Истраживање могућности за даљу оптимизацију *PoW* рударења, пре свега пребацањем рачунски интензивног хеширања на графичку картицу (*GPU*) коришћењем технологија попут *CUDA*.
- **Унапређење P2P мреже:** Имплементација сложенијих механизама за откривање чворова, као и увођење сигурносних мера против мрежних напада.

6.3 Завршна реч

Овај рад је успешно демонстрирао како избор програмског језика и модела паралелизације има огроман утицај на перформансе рачунарски интензивних блокчејн алгоритама. Кроз детаљну анализу, потврђено је да док *Proof-of-Work* нуди робустан механизам за постизање консензуса, његова цена у виду потрошње ресурса је огромна. Промена консензус алгоритма, као што је прелазак на *Proof-of-Stake*, нуди далеко веће добитке у ефикасности него што се може постићи било каквом оптимизацијом или паралелизацијом.

Списак слика

Слика 1 Графички приказ јаког скалирања за <i>Rust</i> имплементацију	26
Слика 2 Графички приказ јаког скалирања за <i>Python</i> имплементацију	27
Слика 3 Графички приказ слабог скалирања за <i>Rust</i> имплементацију	29
Слика 4 Графички приказ слабог скалирања за <i>Python</i> имплементацију	30

Списак листинга

Листинг 1	Структура <code>Block</code> у <i>Rust</i> -у	13
Листинг 2	Структура <code>Blockchain</code> у <i>Rust</i> -у.	14
Листинг 3	Секвенцијална функција за рударење у <i>Python</i> -у.	15
Листинг 4	Секвенцијална функција за рударење у <i>Rust</i> -у.	16
Листинг 5	Функција <code>worker_mine</code> за паралелно рударење у <i>Python</i> -у	18
Листинг 6	Паралелна петља за рударење у <i>Rust</i> -у коришћењем библиотеке <i>Rayon</i>	19
Листинг 7	Функција за пондерисани избор валидатора у <i>Rust</i> -у	21
Листинг 8	Паралелна валидација блока у PoS симулацији.	22
Листинг 9	<code>NetworkLayer</code> у <i>Rust</i> -у.	35
Листинг 10	P2PMessage енумерација која дефинише комуникациони протокол	37
Листинг 11	Функција за генерисање ID поруке.	41

Списак табела

Табела 1 Резултати јаког скалирања за <i>Rust</i> имплементацију (тежина d=5)	25
Табела 2 Резултати јаког скалирања за <i>Python</i> имплементацију (тежина d=5)	26
Табела 3 Резултати слабог скалирања за <i>Rust</i> имплементацију	28
Табела 4 Резултати слабог скалирања за <i>Python</i> имплементацију	30
Табела 5 Резултати перформанси за PoS симулацију	33
Табела 6 Пример статистике избора валидатора у симулацији са 12 учесника	33

Списак коришћених скраћеница

Скраћеница	Опис
P2P	Peer-to-Peer (мрежа равноправних чворова)
PoS	Proof-of-Stake (доказ о улогу)
PoW	Proof-of-Work (доказ о раду)
BFT	Byzantine Fault Tolerance (толеранција на византијске грешке)
CPU	Central Processing Unit (централна процесорска јединица)
GIL	Global Interpreter Lock (глобални интерпретаторски закључак)
HPC	High-Performance Computing (рачунарство високих перформанси)
ID	Identifier (идентификатор)
JSON	JavaScript Object Notation (формат за размену података)
SHA-256	Secure Hash Algorithm 256-bit (256-битни алгоритам за сигурносно хеширање)
TCP/IP	Transmission Control Protocol / Internet Protocol (протокол за контролу преноса / интернет протокол)
TPS	Transactions Per Second (трансакције у секунди)

Списак коришћених појмова

Појам	Објашњење
Блокчејн	Дистрибуирана и криптоографски обезбеђена дигитална књига (база података) која се састоји од ланца блокова.
Генесис блок	Први блок у блокчејну, који представља почетну тачку ланца и једини је блок који нема референцу на претходни.
Децентрализација	Архитектонски принцип где контрола и подаци нису смештени на једном централном месту, већ су распоређени међу свим учесницима у мрежи.
Додатно оптерећење (Overhead)	Додатни рачунарски рад који систем мора да обави поред корисног рада, као што су креирање нити, синхронизација и серијализација података.
Gossip протокол	Механизам за ширење порука у децентрализованој мрежи где сваки чврт прослеђује примљену информацију својим суседима.
Јако скалирање (Strong Scaling)	Модел за мерење перформанси у којем величина проблема остаје фиксна, док се број процесорских ресурса повећава, са циљем смањења времена извршавања.
Консензус алгоритам	Механизам који омогућава учесницима у дистрибуираној мрежи да се усагласе око јединственог и исправног стања система.
Криптоографско хеширање	Једносмерни математички процес који претвара улазне податке произвољне дужине у излазни низ бајтова фиксне дужине (хеш).
Меморијско спремиште (Mempool)	Привремено складиште унутар чвора где се чувају непотврђене трансакције пре него што буду укључене у блок.
Непроменљивост (Immutability)	Својство блокчејна које гарантује да се подаци, једном уписаны у ланац, не могу накнадно мењати без нарушавања криптоографске везе.
Нонс (Nonce)	Случајан број (енгл. <i>Number used only once</i>) који рудари у PoW систему итеративно мењају како би пронашли хеш блока који задовољава задату тежину.
Правило најдужег ланца	Основни механизам за решавање форкова у PoW системима, који налаже да се верзија блокчејна са највише блокова сматра исправном.
Proof-of-Stake (PoS)	Консензус алгоритам где се сигурност мреже темељи на економском улогу (уложену криптовалуту) учесника, а не на рачунарској снази.

Proof-of-Work (PoW)	Консензус алгоритам који захтева од учесника (рудара) да реше рачунарски захтеван задатак како би добили право да додају нови блок у ланац.
Рударење (Mining)	Процес у PoW системима где се учесници такмиче у проналажењу валидног хонса за нови блок, користећи “brute-force” метод.
Слабо скалирање (Weak Scaling)	Модел за мерење перформанси у којем величина проблема расте пропорционално са бројем процесорских ресурса, са циљем одржавања константног времена извршавања.
Тежина (Difficulty)	Параметар у PoW систему који дефинише колико је тешко пронаћи валидан хеш, обично изражен као захтев за одређеним бројем водећих нула.
Убрзање (Speedup)	Метрика која показује колико је пута паралелна верзија програма бржа од секвенцијалне верзије.
Улог (Stake)	Количина криптовалуте коју валидатор у PoS систему закључава као залог како би учествовао у процесу валидације блокова.
Форк (Рачвање ланца)	Привремена појава у дистрибуираној мрежи где истовремено постоје две или више валидних, али различитих верзија блокчејна.
Чвор (Node)	Појединачни рачунар или учесник у P2P мрежи који покреће блокчејн софтвер, чува копију ланца и валидира трансакције.

Биографија

Владимир Чорненки рођен је 6. октобра 2002. године у Новом Саду, Србија. Завршио је средњу Електротехничку школу “Mihajlo Pupin” у Новом Саду, смер Електротехничар рачунара 2020. године. 2021. године уписао је основне академске студије на Факултету техничких наука у Новом Саду, смер Софтверско инжињерство и информационе технологије. Положио је све испите предвиђене планом и програмом са просечном оценом 9.32.

Литература

- [1] S. Nakamoto, „Bitcoin: A Peer-to-Peer Electronic Cash System“. Приступљено: 10. Новембар 2025. [На Интернету]. Доступно на <https://bitcoin.org/bitcoin.pdf>
- [2] Karl J. O’Dwyer, „Bitcoin Mining and its Energy Footprint“. Приступљено: 10. Новембар 2025. [На Интернету]. Доступно на <https://mural.maynoothuniversity.ie/id/eprint/6009/1/DM-Bitcoin.pdf>
- [3] Ethereum Foundation, „The Merge | ethereum.org“. Приступљено: 10. Новембар 2025. [На Интернету]. Доступно на <https://ethereum.org/sr/roadmap/merge/>
- [4] hpc wiki, „Scaling“. Приступљено: 10. Новембар 2025. [На Интернету]. Доступно на <https://hpc-wiki.info/hpc/Scaling>
- [5] Leslie Lamport and Robert Shostak and Marshall Pease, „The Byzantine Generals Problem“. Приступљено: 12. Новембар 2025. [На Интернету]. Доступно на <https://dl.acm.org/doi/pdf/10.1145/357172.357176>
- [6] P. Erdos и A. Renyi, „On the evolution of random graphs “. Приступљено: 17. Новембар 2025. [На Интернету]. Доступно на <https://pages.cs.wisc.edu/~cs809-1/ErdosRenyi.pdf>