УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Лука Петковић

Симулација двоструког клатна и анализа перформанси паралелних имплементација

ДИПЛОМСКИ 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 - ментор)

Студијски програм:	Софтверско инжењерство и информационе технологије		
Студент:	Лука Петковић	Број индекса:	SV 16/2021
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:

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

НАСЛОВ ЗАВРШНОГ РАДА:

Симулација двоструког клатна и анализа перформанси паралелних имплементација

ТЕКСТ ЗАДАТКА:

Анализирати нумеричке методе за решавање физичког система двоструког клатна. Дизајнирати и имплементирати систем за симулацију двоструког клатна у више програмских језика са акцентом на паралелно извршавање. Обухватити програмске језике Пајтон, Го и Раст. Дискутовати процес развоја и резултате симулације на овим програмских језицима.

При изради користити препоручену праксу из области софтверског инжењерства. Детаљно документовати решење.

Руководилац студијског програма:	Ментор рада:

Примерак за: □ - Студента; □ - Ментора

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

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски - бечелор рад
Аутор, **АУ**:		Лука Петковић
Ментор, **МН**:		Др Игор Дејановић, редовни професор
Наслов рада, **НР**:		Симулација двоструког клатна и анализа перформанси паралелних имплементација
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**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		7/45/6/9/21/0/2
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Примењене рачунарске науке и информатика
Предметна одредница/Кључне речи, **ПО**:		Двоструко клатно, паралелна обрада, Python, Rust, Go
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		Рад представља симулацију двоструког клатна и анализу перформанси паралелних имплементација. Коришћена је метода Рунге–Кута четвртог реда за нумеричко решавање једначина. Имплементације у Python, Rust и Go језицима упоређене су по времену извршавања, убрзању и скалабилности, уз анализу предности сваког приступа.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		06.11.2025
Чланови комисије, **КО**:	Председник:	Др Марко Марковић, ванредни професор
	Члан:	Др Синиша Николић, доцент
	Члан:	Потпис ментора
	Члан, ментор:	Др Игор Дејановић, редовни професор

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

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	
Author, **AU**:	Luka Petković
Mentor, **MN**:	Igor Dejanović, Phd., full professor
Title, **TI**:	Double pendulum simulation with performance comparison of parallel implementations
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/appendixes)	7/45/6/9/21/0/2
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Applied computer science and informatics
Subject/Key words, **S/KW**:	double pendulum, parallel processing, Python, Rust, Go
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad
Note, **N**:	
Abstract, **AB**:	This paper presents a simulation of the double pendulum and a performance comparison of parallel implementations. The fourth-order Runge–Kutta method was used for numerical integration. Implementations in Python, Rust, and Go were evaluated by execution time, speedup, and scalability, highlighting each language's advantages.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	06.11.2025

Defended Board, **DB**:		
	President:	Marko Marković, Phd., assoc. professor
	Member:	Siniša Nikolić, Phd., asist. professor
	Member:	
	Member, Mentor:	Igor Dejanović, Phd., full professor

Menthor'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 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

_____

Садржај

Глава 1

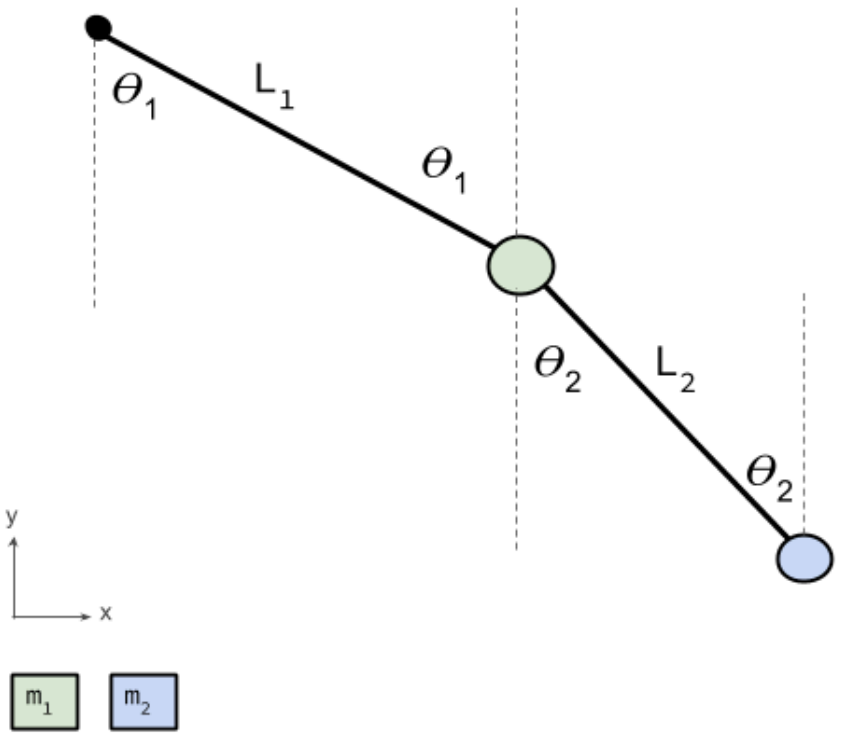
Увод

1.1 Шта је двоструко клатно?

Двоструко клатно (слика 1) је физички систем који се састоји од два међусобно повезана клатна. Први сегмент је причвршћен за фиксну тачку, док је други окачен на крај првог сегмента. Кретање система описује комбинацију кретања оба клатна, при чему се сваки сегмент помера под утицајем гравитације и сила које су последица њихове међусобне везе.

Овакав систем има четири степена слободе – два угла и две угаоне брзине – што доводи до сложене динамике. Када се енергија система повећа, двоструко клатно почиње да показује хаотично понашање, односно већу осетљивост на почетне услове. Иако се ради о класичном механичком систему, његово понашање није предвидиво на дуже стазе, па се често користи као илустрација **хаотичних система** у физици и нумеричкој анализи.

Стање система у сваком тренутку може се описати четворком вредности: (θ_1, ω_1, θ_2, ω_2) где су θ_1(тета 1) и θ_2(тета 2) углови сегмената у односу на вертикалу, а ω_1(омега 1) и ω_2(омега 2) њихове угаоне брзине. Ове променљиве су међусобно повезане системом нелинеарних диференцијалних једначина које немају аналитичко решење, па се решавање врши **нумеричким методама**.



Слика 1: Основни дијаграм двоструког клатна

1.2 Циљ и структура рада

Основни циљ овог рада јесте развој и анализа система за симулацију двоструког клатна у више програмских језика, са посебним освртом на поређење перформанси паралелних имплементација. Истражују се

разлике у брзини извршавања, ефикасности и скалабилности између имплементација у језицима *Python*, *Rust* и *Go (Golang)*, као и утицај различитих модела паралелизма на резултате симулације.

Рад има за циљ да:
- демонстрира примену нумеричких метода на хаотичном систему,
- анализира ефекте паралелне обраде код различитих програмских језика,
- упореди брзину и ефикасност имплементација,
- истакне практичне предности сваког језика у области научних симулација.

Структура рада организована је на следећи начин. У другом поглављу описане су теоријске основе система двоструког клатна, укључујући физички модел, једначине кретања и примењену нумеричку методу. Треће поглавље приказује дизајн решења и концепт ансамбл-паралелизације. Четврто поглавље садржи опис имплементација у *Python*, *Rust* и *Go* језицима. У петом поглављу дато је упоредно поређење резултата и дискусија, са посебном пажњом на јако и слабо скалирање. Након тога, у шестом поглављу, дати су неки занимљиви визуелни прикази двоструког клатна на основу израчунатих вредности. На крају, у седмом поглављу налази се закључак са прегледом резултата и предлогом за даљи рад.

Глава 2

Теоријске основе

2.1 Хаотични системи

Хаотични системи представљају класу динамичких система чије се понашање карактерише изузетном осетљивошћу на почетне услове [1]. Иако се њихова еволуција управља детерминистичким једначинама, и најмања промена у почетним параметрима доводи до драстичних промена резултата. Овај феномен је познат као **ефекат лептира** (*butterfly effect*) и чини да такви системи делују непредвидиво на дуже стазе.

Двоструко клатно је један од најпознатијих примера хаотичног система у класичној механици. Упркос томе што се заснива на једноставним законима физике, његово кретање може бити веома сложено, са путањама које се ретко понављају. Овакав систем је од посебног значаја у нумеричкој анализи, јер омогућава тестирање стабилности, тачности и перформанси различитих метода интеграције.

2.2 Физички модел двоструког клатна

Двоструко клатно се састоји од два сегмента дужина l_1 и l_2, са масама m_1 и m_2. Први сегмент је окачен о фиксну тачку, док је други сегмент окачен на крај првог. Положај система описује се угловима θ_1 и θ_2 у односу на вертикалу [2]. Систем ћемо посматрати у две димензије због једноставности, а сем тога, занемарићемо све стране силе попут силе отпора ваздуха или силе трења.

Енергија система се састоји од кинетичке (T) и потенцијалне (V) енергије:

$$E = T + V$$

Кинетичка енергија:

$$T = \tfrac{1}{2} m_1 (l_1 \omega_1)^2 + \tfrac{1}{2} m_2 \left[(l_1 \omega_1)^2 + (l_2 \omega_2)^2 + 2 l_1 l_2 \omega_1 \omega_2 \cos(\theta_1 - \theta_2) \right]$$

Први члан се односи на кинетичку енергију горњег клатна. Други члан се односи на кинетичку енергију доњег клатна које укључује:

- сопствено кретање
- кретање због ротације горњег сегмента
- међусобни унакрсни члан који настаје јер се вектори брзина не поклапају у правцу

Потенцијална енергија:

$$V = -m_1 g l_1 \cos(\theta_1) - m_2 g [l_1 \cos(\theta_1) + l_2 \cos(\theta_2)]$$

Потенцијална енергија се рачуна у односу на најнижи положај. Први члан се односи на висину масе m_1, а други на висину масе m_2 која зависи од оба угла. Знак минус стоји испред јер се енергија система смањује кад тела иду надоле. [2]

Ове једначине немају аналитичко решење, па се у пракси примењују нумеричке методе.

2.3 Нумеричке методе интеграције

Када кажемо да систем „нема аналитичко решење“, то значи да не постоји затворени математички израз који би омогућио да се вредности променљивих добију директним рачунањем. Другим речима, функције које описују кретање тела не могу се изразити у облику једноставних формула (попут синуса, косинуса или експоненцијала), већ се решење мора приближно израчунати.

Због тога се у пракси користе **нумеричке методе интеграције**, које омогућавају приближно решавање система диференцијалних једначина у малим временским корацима. Оне омогућавају да се за познато

почетно стање система — почетне углове (θ_1, θ_2) и угаоне брзине (ω_1, ω_2) — израчунају нове вредности у наредним корацима времена ($t + \Delta t$).

Једноставне методе, попут Euler-ове, често производе велике грешке јер користе само једну процену нагиба функције у датом кораку. То доводи до акумулације грешке и нестабилности током дужих симулација, посебно код хаотичних система попут двоструког клатна.

Да би се добила већа тачност без већег повећања сложености, у овом раду примењује се **метода Рунге–Кута четвртог реда (RK4)** [2]. Ова метода представља компромис између једноставности имплементације и нумеричке стабилности. Основна идеја је да се нова вредност израчунава на основу комбинације више процена нагиба унутар једног корака, што повећава тачност у односу на једноставније методе.

Ако је општа форма једначине:

$\frac{dy}{dt} = f(t, y)$

онда се алгоритам RK4 дефинише на следећи начин:

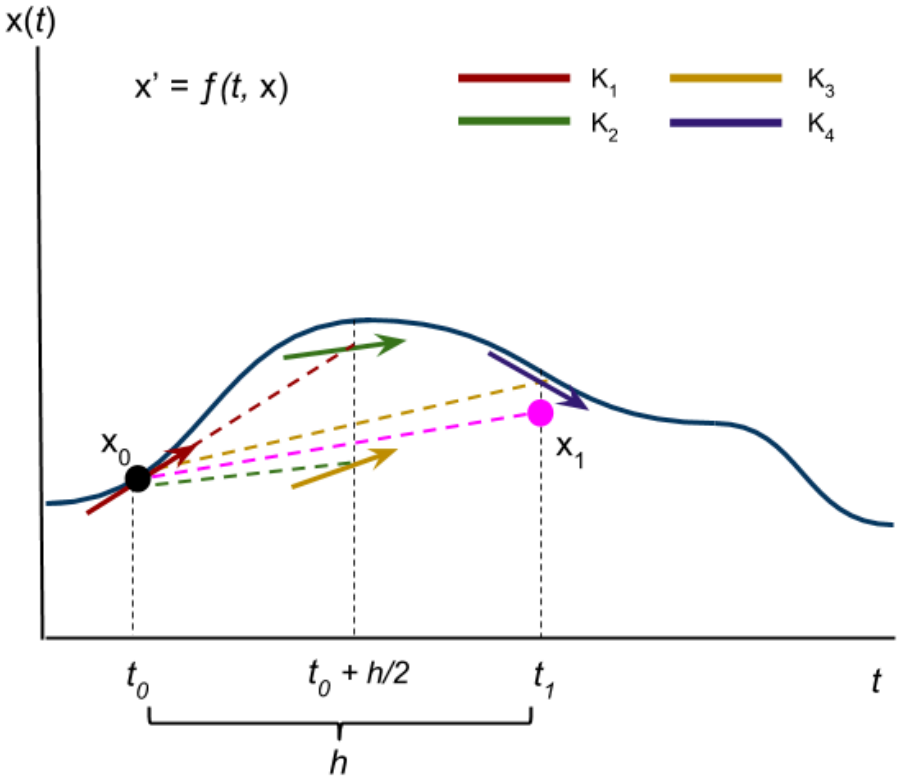
$k_1 = f(t, y)$

$k_2 = f\left(t + \frac{1}{2}h, y + \frac{1}{2}hk_1\right)$

$k_3 = f\left(t + \frac{1}{2}h, y + \frac{1}{2}hk_2\right)$

$k_4 = f(t + h, y + hk_3)$

$y_{\{n+1\}} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$



Слика 2: Приказ Runge–Kutta (RK4) метода

На слици 2 је приказано како се у сваком кораку интеграције израчунавају четири различите процене нагиба (k_1, k_2, k_3, k_4) које заједно дају бољу апроксимацију правог решења функције $x(t)$. Тачке t_0, $t_0 + \frac{h}{2}$ и t_1 представљају положаје у времену у којима се израчунавају међурезултати. Коначна вредност x_1 добија се као комбинација ових нагиба са различитим тежинама.

Метода Рунге–Кута се показала као стабилна и ефикасна чак и код система са израженом нелинеарношћу и хаотичним понашањем, што је чини изузетно погодном за ову симулацију.

2.4 Паралелизација и модели скалирања

Иако је *Runge–Kutta* метода изузетно погодна за тачно нумеричко решавање система диференцијалних једначина, она по својој природи није лако паралелизујућа. Разлог је што сваки временски корак зависи од резултата претходног — да би се израчунало стање у тренутку $t_{\{n+1\}}$, потребно је да буде познато стање у тренутку t_n. Ово уводи секвенцијалну зависност у израчунавање, која ограничава могућност директног распоређивања задатка на више процесора.

Да би се ипак искористиле предности савремених вишејезгарних процесора, примењује се тзв. **ансамбл паралелизација**. Уместо да се једна симулација раздваја на мање делове, истовремено се покреће више независних симулација са различитим почетним условима (различите комбинације углова и угаоних брзина). На овај начин сваки процес или нит извршава потпуно независан ток израчунавања, а резултати се на крају обједињују ради анализе понашања система.

Оваква стратегија има два кључна циља:
- боље искоришћење рачунарских ресурса (сви процесорски језгри су активни);
- омогућавање статистичке анализе резултата на већем броју различитих почетних услова.

Да би се ефикасност паралелизације објективно проценила, користе се два класична модела скалирања: **јако скалирање** и **слабо скалирање**.

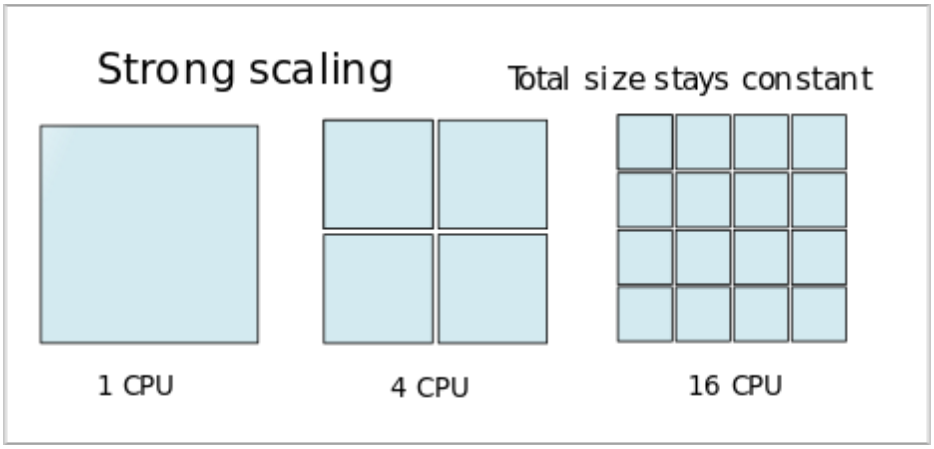
2.4.1 Јако скалирање

Јако скалирање (слика 3) подразумева извршавање истог задатка (фиксиране величине проблема) на различитом броју процесора [3]. Циљ је да се измери убрзање које се постиже повећањем броја процесора:

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

где је T_1 време извршавања на једном процесору, а T_p време извршавања на p процесора.

У идеалном случају, убрзање би требало да буде линеарно ($S_p = p$), али у пракси долази до одступања услед додатних трошкова синхронизације, преноса података и управљања нитима.



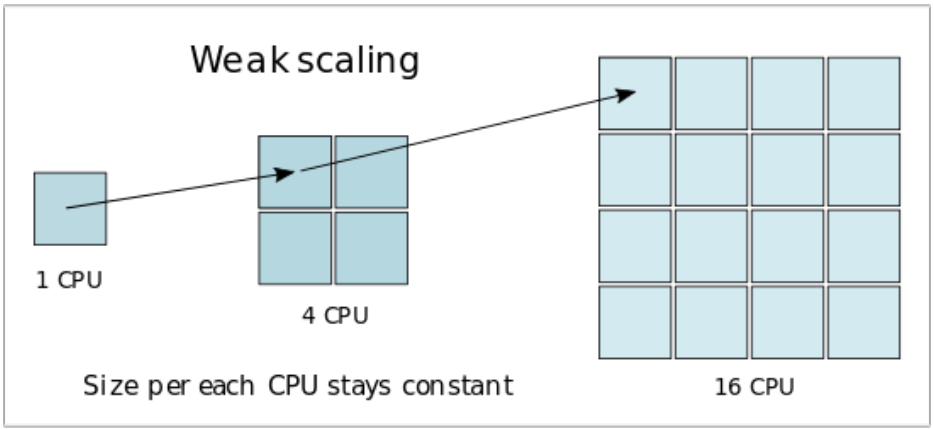
Слика 3: Јако скалирање

2.4.2 Слабо скалирање

Слабо скалирање (слика 4) подразумева пропорционално повећање величине проблема са бројем процесора [3]. У овом моделу, сваки процесор има приближно исти количински део посла као у случају једног процесора. Циљ је да време извршавања остане приближно константно:

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

где E_p представља ефикасност система за p процесора. Добра имплементација треба да показује малу промену у времену извршавања када се величина проблема повећава у складу са бројем процесора.



Слика 4: Слабо скалирање

Ови модели биће примењени у експериментима који следе, како би се проценила скалабилност имплементација у језицима *Python*, *Rust* и *Go*.

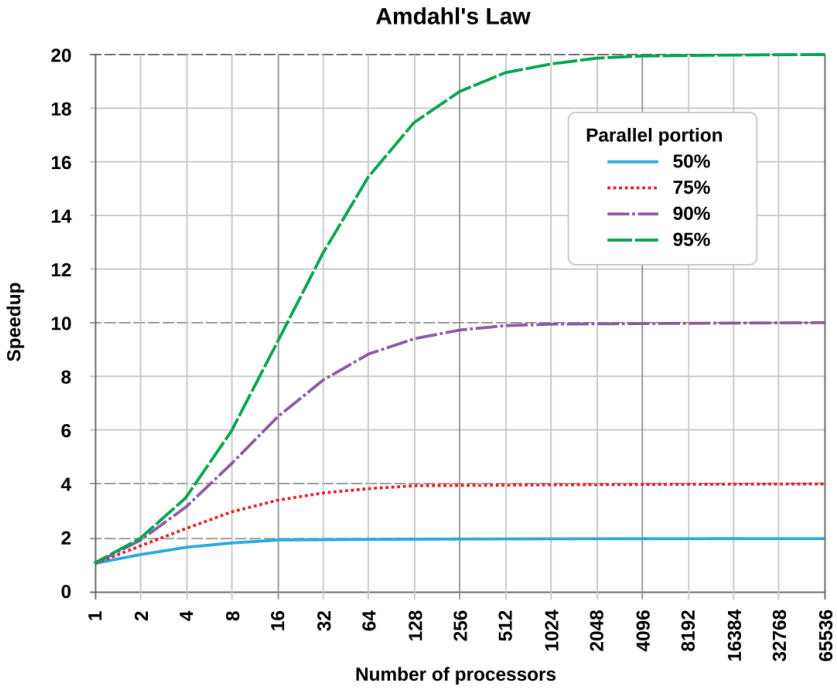
2.4.3 Амдалов и Густафсонов закон

Ефикасност паралелних програма се не може повећавати неограничено. Чак и када се број процесора повећава, део програма који се не може паралелизовати поставља природно ограничење убрзања. Ово ограничење описује **Амдалов закон** [3]:

$$S_p = \frac{1}{(1-f)+\frac{f}{p}}$$

где је:

- S_p – убрзање које се постиже коришћењем p процесора,
- f – део програма који се може паралелизовати,
- $(1-f)$ – серијски део програма који мора бити извршен секвенцијално.



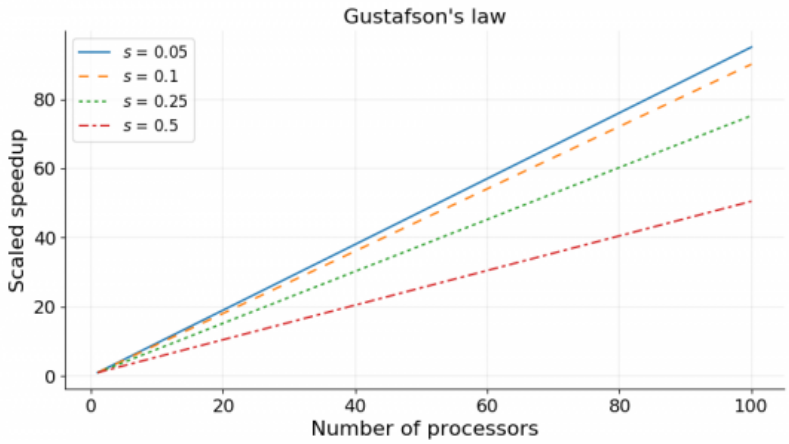
Слика 5: Амдалов закон

Из Амдаловог закона са слике 5 произилази да чак и ако је 95% програма паралелно, теоријски максимум убрзања није већи од 20, без обзира на број процесора. Зато је у пракси важно минимизовати серијске делове програма и комуникационе трошкове између процеса.

Међутим, Амдалов модел се користи у случају фиксне величине проблема. У стварности, повећање броја процесора често омогућава решавање већих проблема у истом временском оквиру. Ову идеју описује **Густафсонов закон** [3]:

$$S_p = p - \alpha(p - 1)$$

где је:
- S_p – постигнуто убрзање,
- p – број процесора,
- α – серијски удео извршавања (однос времена серијског дела и укупног времена)



Слика 6: Густафсонов закон

Густафсонов закон (слика 6) показује да се у пракси убрзање може приближити линеарном, ако се величина проблема повећава заједно са бројем процесора, што је карактеристично за **слабо скалирање**. Ово објашњење боље одражава реалне услове рада савремених система, када се повећање броја процесора користи за обраду већих количина података, а не само за брже извршавање истог посла.

Комбинацијом Амдаловог и Густафсоновог модела могуће је добити реалнију процену ефикасности паралелних система и боље разумети понашање симулација при повећању броја процесора.

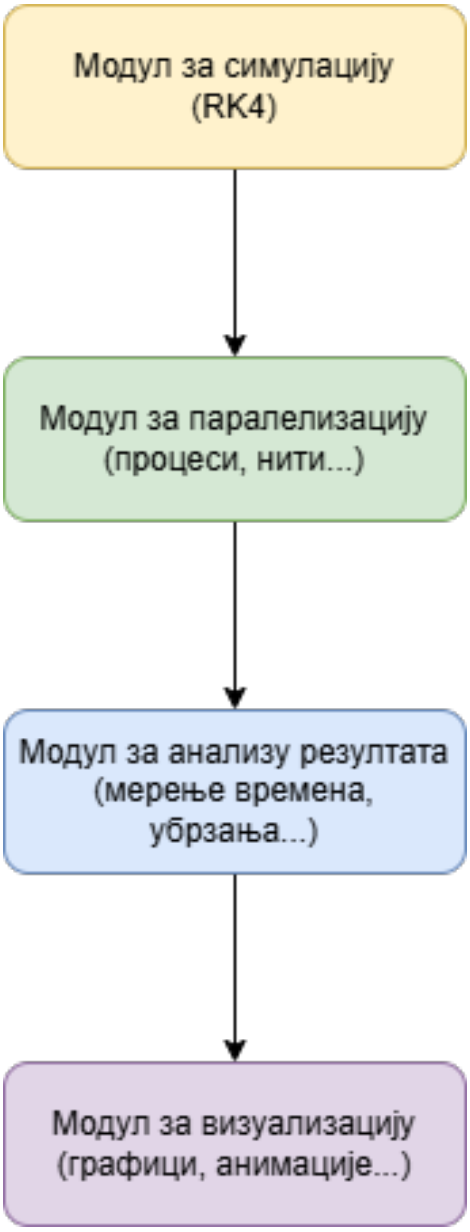
Глава 3

Дизајн система

Циљ овог поглавља је да прикаже укупну архитектуру развијеног система за симулацију двоструког клатна и анализу перформанси паралелних имплементација. Систем је осмишљен тако да омогући једноставно дефинисање почетних услова, покретање симулација у више програмских језика и прикупљање података о времену извршавања ради касније анализе.

3.1 Архитектура решења

Систем на слици 7 је дизајниран као скуп независних компоненти које се могу извршавати и тестирати појединачно. Основни циљ архитектуре је постизање јасне раздвојености између нумеричког језгра симулације, паралелизације и визуализације резултата.



Слика 7: Дијаграм архитектуре

Компоненте система су:

- **Модул за симулацију** — извршава нумеричку интеграцију система диференцијалних једначина користећи Runge–Kutta методу четвртог реда.
- **Модул за паралелизацију** — организује извршавање више независних симулација у различитим процесима или нитима у зависности од изабраног језика.
- **Модул за анализу резултата** — прикупља податке о времену извршавања, броју радних јединица, и израчунава убрзање и ефикасност.
- **Модул за визуализацију** — генерише графиконе и анимације трајекторија ради квалитативне анализе хаотичног понашања.

Сви модули деле заједнички улазни формат параметара, који садржи почетне услове и конфигурацију симулације. Излазни подаци се чувају у CSV формату ради лакше обраде и поређења између различитих имплементација.

3.2 Организација података

Подаци о симулацијама организовани су по моделу једноставне табеле (као нпр. табела 1), где сваки ред представља један експеримент са својим улазним параметрима и резултатима.

Табела 1: Пример структуре података о симулацијама.

ID	θ_{10}	θ_{20}	ω_{10}	ω_{20}	Време извршавања (ms)
1	π/2	π/2 + 0.01	0	0	123.4
2	π/2	π/2 + 0.02	0	0	118.6

Овакав формат омогућава једноставно поређење резултата, као и примену статистичке анализе и израду графикона попут зависности убрзања од броја процесора.

3.3 Паралелни модел извршавања

Паралелизација у систему се заснива на **ансамбл моделу**, где свака нит или процес извршава симулацију са различитим почетним условима. Овај приступ избегава проблем секвенцијалне зависности временских корака, јер су симулације међусобно независне.

У Python имплементацији користи се библиотека `multiprocessing`, у Rust-у `std::thread` и `Rayon`, док Go користи **goroutines** и **channels** за конкурентно извршавање. У свим случајевима, комуникација између нити је минимална, што омогућава скоро идеално скалирање са бројем процесора.

Овако дефинисан систем омогућава флексибилно тестирање различитих језика, метода и параметара, уз очување упоредивости резултата.

Глава 4

Имплементација

У овом делу приказане су 3 независне имплементације система за симулацију двоструког клатна: у програмским језицима *Python*, *Rust*, и *Go*. Свака имплементација прати исти нумерички модел и *RK4* методу, али се разликује у начину организације паралелног извршавања, управљања меморијом и комуникације између процеса.

4.1 Имплементација у програмском језику Python

4.1.1 Увод и очекивања

Python је изабран као полазна тачка за развој система због своје једноставности и изражајности. Иако је *Python* интерпретиран језик и самим тим спорији од компајлираних (попут *Rust* и *Go*), његова синтакса и подршка за паралелно програмирање (преко `multiprocessing` модула) омогућују брз развој и лако експериментисање са различитим моделима извршавања.

Главни циљ ове имплементације био је да се добије референтно, једноставно и прегледно решење које ће послужити као основа за касније поређења.

4.1.2 Секвенцијална верзија

Python имплементација се састоји из три главна дела:

1. **Runge–Kutta интегратор** — врши један корак нумеричке интеграције;
2. **Функција за интеграцију кроз време** — управља низом корака и бележи резултате;
3. **Физички модел система** — израчунава изведене величине и енергију система.

```
def rk4_step(y, dt, f, params):
  k1 = f(y, params)
  k2 = f(y + 0.5 * dt * k1, params)
  k3 = f(y + 0.5 * dt * k2, params)
  k4 = f(y + dt * k3, params)
  return y + (dt / 6.0) * (k1 + 2*k2 + 2*k3 + k4)
```

Листинг 1: Функција Runge–Kutta корака у Python-у

```
def integrate(y0, dt, steps, f, params, record_energy_fn=None):
  y = np.array(y0, dtype=float).copy()
  traj = np.zeros((steps + 1, y.size))
  times = np.zeros(steps + 1)
  energies = None
  if record_energy_fn:
      energies = np.zeros(steps + 1)

  traj[0] = y
  if energies is not None:
      energies[0] = record_energy_fn(y, params)

  for i in range(1, steps + 1):
      y = rk4_step(y, dt, f, params)
      traj[i] = y
      times[i] = i * dt
      if energies is not None:
          energies[i] = record_energy_fn(y, params)

  return times, traj, energies
```

Листинг 2: Функција за интеграцију кроз време

```
def integrate(y0, dt, steps, f, params, record_energy_fn=None):
    y = np.array(y0, dtype=float).copy()
    traj = np.zeros((steps + 1, y.size))
    times = np.zeros(steps + 1)
    energies = None
    if record_energy_fn:
        energies = np.zeros(steps + 1)

    traj[0] = y
    if energies is not None:
        energies[0] = record_energy_fn(y, params)

    for i in range(1, steps + 1):
        y = rk4_step(y, dt, f, params)
        traj[i] = y
        times[i] = i * dt
        if energies is not None:
            energies[i] = record_energy_fn(y, params)

    return times, traj, energies

def energy(state, params):
    theta1, omega1, theta2, omega2 = state
    m1 = params.get("m1", 1.0)
    m2 = params.get("m2", 1.0)
    l1 = params.get("l1", 1.0)
    l2 = params.get("l2", 1.0)
    g  = params.get("g", 9.81)

    x1 = l1 * np.sin(theta1)
    y1 = -l1 * np.cos(theta1)
    x2 = x1 + l2 * np.sin(theta2)
    y2 = y1 - l2 * np.cos(theta2)

    v1x = l1 * omega1 * np.cos(theta1)
    v1y = l1 * omega1 * np.sin(theta1)
    v2x = v1x + l2 * omega2 * np.cos(theta2)
    v2y = v1y + l2 * omega2 * np.sin(theta2)

    T = 0.5 * m1 * (v1x**2 + v1y**2) + 0.5 * m2 * (v2x**2 + v2y**2)
    V = m1 * g * y1 + m2 * g * y2

    return T + V
```

Листинг 3: Рачунање изведених величина и енергије система

4.1.3 Паралелна имплементација

Да би се постигло брже извршавање симулација над већим бројем почетних услова, *Python* користи `multiprocessing`, који омогућава паралелно извршавање процеса на више језгара. Свака симулација добија различит скуп почетних углова и угаоних брзина, а резултати се прикупљају по завршетку свих процеса.

Овим приступом свака симулација ради независно, без потребе за синхронизацијом, јер сваки процес има сопствени интерпретер и меморијски простор, чиме се спречава глобално закључавање интерпретера (GIL).

4.1.4 Експерименти јаког и слабог скалирања

Да би се испитала скалабилност *Python* имплементације, извршена су два скупа тестова.

- **Јако скалирање:** број временских корака остаје фиксиран, а број процеса се мења (1, 2, 4, 8, 16). Циљ је да се измери убрзање дефинисано као $S_p = \frac{T_1}{T_p}$, где је T_p време извршавања са p процесора.
- **Слабо скалирање:** сваки процес има константан број корака, док укупан број корака расте пропорционално броју процесора. Циљ је да се провери да ли време извршавања остаје приближно константно када се посао равномерно дели на више јединица.

Резултати оба експеримента биће приказани у наредном поглављу, где ће се поредити *Python*, *Rust* и *Go* имплементације.

4.2 Имплементација у програмском језику Rust

4.2.1 Увод и мотивација

Rust је познат по својој способности да комбинује високе перформансе (сличне C/C++ језицима) са снажним системом типова и гаранцијама безбедности меморије. У контексту симулације двоструког клатна, Rust омогућава оптимално управљање меморијом, паралелно извршавање без *data race*-ова, и ефикасну обраду великих скупова података.

Rust имплементација служи као пример како се исти нумерички модел може реализовати у језику нижег нивоа уз максималну ефикасност.

4.2.2 Секвенцијална верзија

Секвенцијална имплементација у *Rust*-у користи исту *Runge–Kutta* методу четвртог реда (RK4) као и Python, али је прилагођена строгом систему типова и статичком управљању меморијом.

4.2.3 Паралелна верзија

Паралелна имплементација заснива се на употреби `rayon` библиотеке. Она омогућава декларативну обраду података у више нити без потребе за ручном синхронизацијом. Свака симулација представља засебан задатак са различитим почетним условима, што омогућава готово идеално скалирање са бројем језгара.

Имплементација користи структуру `Params` за складиштење физичких параметара система (масе, дужине и гравитације), док се стање система чува у низу дужине 4 (θ_1, ω_1, θ_2, ω_2). Свака нит извршава интеграцију Runge–Kutta методом за свој скуп почетних услова.

```
    use rayon::prelude::*;
    use std::time::Instant;

    #[derive(Clone, Copy)]
    struct Params {
        m1: f64, m2: f64,
        l1: f64, l2: f64,
        g: f64,
    }

    ...

    fn main() {
      let mut runs = 8usize;
      let mut steps = 600_000;
      for arg in std::env::args().skip(1) {
          if let Some(v) = arg.strip_prefix("--runs=") { runs = v.parse().unwrap(); }
          if let Some(v) = arg.strip_prefix("--steps="){ steps = v.parse().unwrap(); }
      }

      let dt = 0.001;
      let params = Params{ m1:1.0, m2:1.0, l1:1.0, l2:1.0, g:9.81 };
      let base = [std::f64::consts::FRAC_PI_2, 0.0, std::f64::consts::FRAC_PI_2+0.01, 0.0];

      println!("Pokrećem {runs} paralelnih simulacija (steps={steps})...");
      let start_all = Instant::now();

      (0..runs).into_par_iter().for_each(|i| {
          let mut y0 = base;
          y0[2] += (i as f64 - runs as f64/2.0) * 1e-3;
          let _ = {
              let mut y = y0;
              for _ in 0..steps { y = rk4_step(&y, dt, &params); }
          };
      });

      let total = start_all.elapsed().as_secs_f64();
      println!("Paralelno (runs={runs}, steps={steps}) za {:.4}s", total);
}
```

Листинг 4: Главни део паралелне имплементације помоћу *Rayon* библиотеке

У *main* методи можемо приметити како `into_par_iter()` аутоматски дели посао на доступна језгра, док *Rayon runtime* брине о балансирању оптерећења и поновној употреби радних нити. Мерењем времена извршавања показано је да Rust имплементација постиже најбоље резултате у односу на остала решења, што ће бити детаљније анализирано у наредном поглављу.

4.3 Имплементација у програмском језику Go

4.3.1 Увод и мотивација

Go је изабран као трећи приступ због своје једноставности, брзине компилације и уграђене подршке за конкурентно извршавање. Он комбинује синтаксну једноставност са снажним моделом конкурентности заснованим на *goroutines* [4] и *channels*. Овај модел омогућава развој програма који користе више језгара без експлицитног управљања нитима, што га чини идеалним за паралелне симулације.

У контексту овог рада, *Go* имплементација двоструког клатна представља **баланс** између брзине извршавања (сличне Rust-у) и једноставности развоја (сличне Python-у). Циљ је био демонстрирати како конкурентно извршавање може бити постигнуто уз минималан број линија кода.

4.3.2 Структура програма

Програм је организован у неколико целина:

1. **Модел система** — дефинише структуре `Params`, `State` и `Sample`, које представљају параметре система, тренутно стање и узорак података током симулације.
2. **Нумеричка интеграција** — реализована методом *Runge–Kutta* четвртог реда кроз функцију `rk4Step`.
3. **Енергетска функција** — израчунава кинетичку и потенцијалну енергију система за сваки корак.
4. **Паралелна симулација** — организована преко **worker pool** модела користећи *goroutines* и *channels*.
5. **Упис резултата** — свака симулација резултате уписује у засебан CSV фајл ради касније анализе.

```
func rk4Step(p Params, s State, dt float64) State {
    k1 := deriv(p, s)
    s2 := State{Th1: s.Th1 + 0.5*dt*k1.Th1, Om1: s.Om1 + 0.5*dt*k1.Om1,
                Th2: s.Th2 + 0.5*dt*k1.Th2, Om2: s.Om2 + 0.5*dt*k1.Om2}
    k2 := deriv(p, s2)
    s3 := State{Th1: s.Th1 + 0.5*dt*k2.Th1, Om1: s.Om1 + 0.5*dt*k2.Om1,
                Th2: s.Th2 + 0.5*dt*k2.Th2, Om2: s.Om2 + 0.5*dt*k2.Om2}
    k3 := deriv(p, s3)
    s4 := State{Th1: s.Th1 + dt*k3.Th1, Om1: s.Om1 + dt*k3.Om1,
                Th2: s.Th2 + dt*k3.Th2, Om2: s.Om2 + dt*k3.Om2}
    k4 := deriv(p, s4)
    return State{
        Th1: s.Th1 + dt*(k1.Th1+2*k2.Th1+2*k3.Th1+k4.Th1)/6.0,
        Om1: s.Om1 + dt*(k1.Om1+2*k2.Om1+2*k3.Om1+k4.Om1)/6.0,
        Th2: s.Th2 + dt*(k1.Th2+2*k2.Th2+2*k3.Th2+k4.Th2)/6.0,
        Om2: s.Om2 + dt*(k1.Om2+2*k2.Om2+2*k3.Om2+k4.Om2)/6.0,
    }
}
```

Листинг 5: Нумеричка интеграција методом Runge–Kutta четвртог реда у Go-у

4.3.3 Паралелна имплементација

Паралелизација се заснива на **worker pool** моделу где сваки **радник** извршава независну симулацију са различитим почетним условима. Послови се шаљу преко *jobs* канала, а резултати се прикупљају преко *results* канала. На овај начин се постиже висок степен искоришћења процесора уз минималан *overhead*.

```
func worker(p Params, dt float64, steps int, recordEvery int, outDir string,
            jobs <-chan job, results chan<- result) {
    for jb := range jobs {
        samples := simulate(p, jb.s0, dt, steps, recordEvery)
        path := fmt.Sprintf("%s/run_%04d.csv", outDir, jb.id)
        err := writeCSV(path, samples)
        results <- result{id: jb.id, err: err, path: path}
    }
}
```

Листинг 6: *Worker* функција

У главном делу програма, број радних горутина (*workers*) може бити експлицитно дефинисан или се аутоматски поставити на вредност `runtime.GOMAXPROCS(0)`, која представља број доступних језгара. Послови се креирају тако што се свакој симулацији задају благо различити почетни углови, чиме се демонстрира хаотична осетљивост система.

4.3.4 Карактеристике и предности

- Модел конкурентности у *Go*-у је изузетно једноставан — стварање хиљада *goroutine*-а има занемарљив трошак у меморији.
- Комуникација преко *channels* обезбеђује безбедну синхронизацију без експлицитних закључавања.
- Уз једноставну синтаксу, *Go* нуди стабилне перформансе које се у овом систему приближавају Rust-у.

Глава 5

Анализа резултата

У овом поглављу приказани су резултати експеримената извршених над имплементацијама двоструког клатна у *Python*, *Rust* и *Go* програмским језицима. Циљ анализе је да се упореде перформансе, скалабилност и ефикасност све три верзије програма при различитом броју процеса, као и да се утврди у којој мери резултати одговарају теоријским моделима паралелног израчунавања.

5.1 Окружење и методологија тестирања

Сви експерименти су спроведени на лаптопу са карактеристикама датим у табели 2

Табела 2: Конфигурација лаптопа

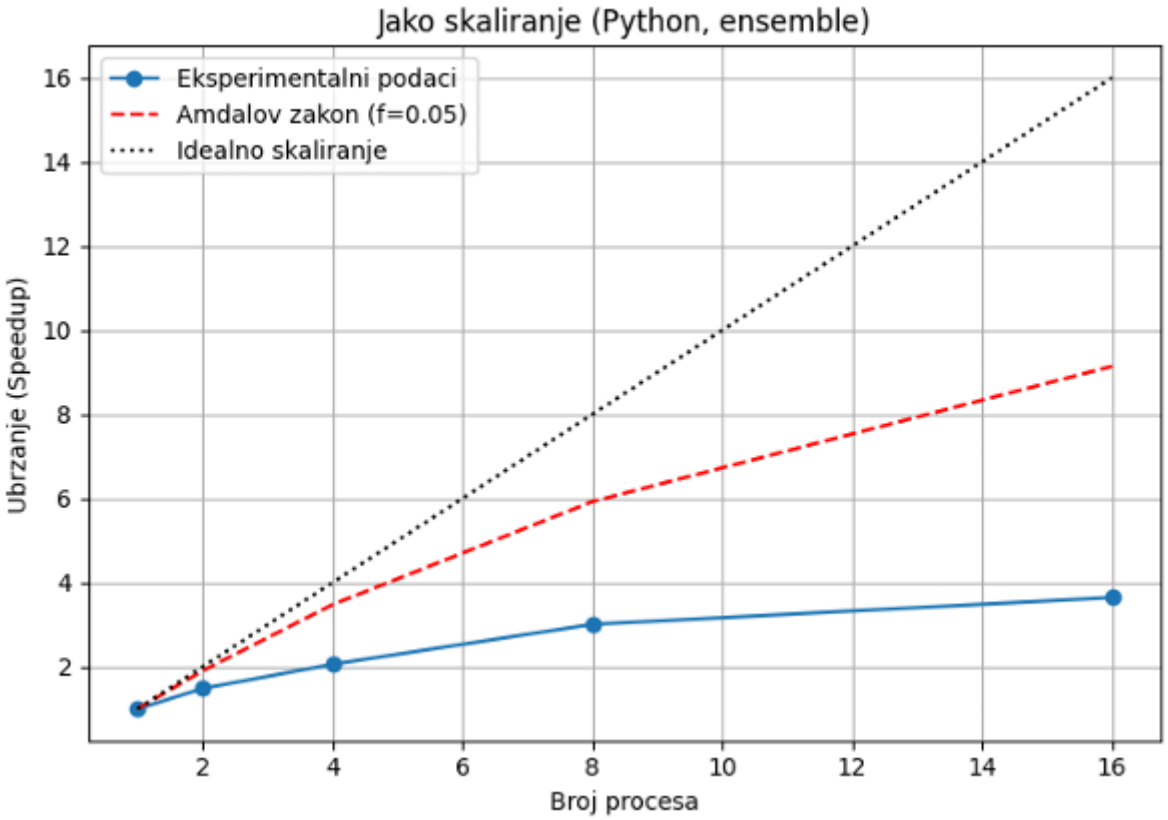
Процесор	*AMD Ryzen 7 5700U*
Радни такт	1.80 GHz
Cache меморија	L2 - 4MB, L3 - 8MB
RAM меморија	8GB
Оперативни систем	*Windows* 11

Симулације су извођене RK4 методом са временским кораком *dt=0.001* током 60 секунди, односно 60000 корака. Свака трајекторија представља један скуп почетних услова, а *ensemble* паралелизација омогућава извођење више различитих трајекторија истовремено. Сва мерења су понављана 30 пута, а у анализи се користи средња вредност времена извршавања као и стандардна девијација. Што се тиче стања сваког клатна, оно се чува у одговарајућем *csv* фајлу са 6 основних променљивих (колона): t - протекло време, θ_1, θ_2, ω_1, ω_2 и *energy* - укупна енергија система.

5.2 Анализа јаког и слабог скалирања у програмском језику *Python*

5.2.1 Јако скалирање

На слици 8 и табели 3 су приказани резултати јаког скалирања. На табели се јасно уочава да време извршавања опада са повећањем броја процеса, што потврђује да паралелизација успешно распоређује посао између процеса. Убрзање расте приближно линеарно за мали број процеса, али се касније јавља одступање од идеалне линије скалирања. Ово понашање је очекивано ако узмемо у обзир ограничења *Python* језика, и трошкове операција читања и писања података у csv фајлове.



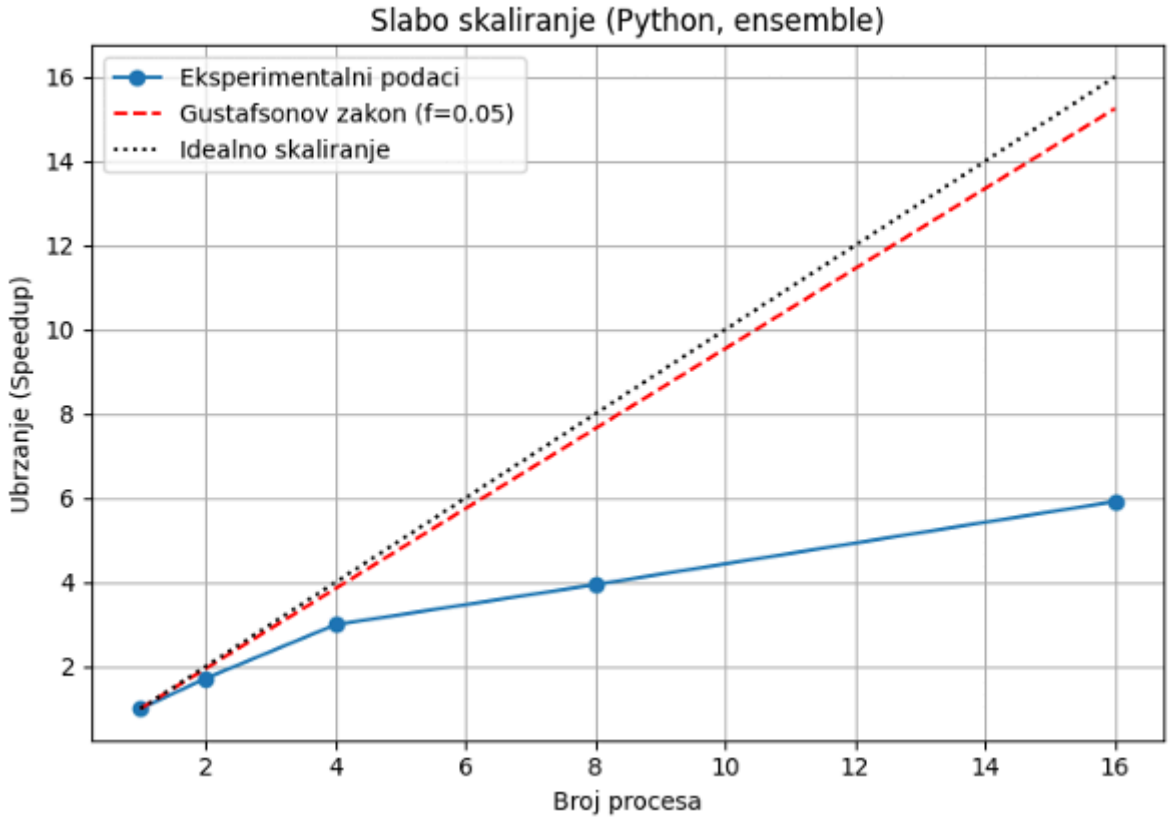
Слика 8: Јако скалирање (*Python*)

Табела 3: Резултати јаког скалирања (*Python*)

број процесора	време извршавања	стандардна девијација	убрзање	теоријско убрзање (Амдал)
1	9.57s	0.103s	1.00	1.00
2	6.45s	0.271s	1.48	1.90
4	4.64s	0.157s	2.06	3.48
8	3.18s	0.049s	3.01	5.92
16	2.62s	0.056s	3.65	9.14

5.2.2 Слабо скалирање

Резултати слабог скалирања приказани су на слици 9 и у табели 4. У овом експерименту укупан посао је повећаван тако да сваки процес извршава 60000 интеграционих корака. На тај начин се проверава колико добро се систем понаша при већем оптерећењу. У савршеном случају, укупно време извршавања би остало исто при било ком броју процеса. У нашем случају, време се благо повећава, што је последица *Python*-овог *multiprocessing* модела.



Слика 9: Слабо скалирање (*Python*)

Табела 4: Резултати слабог скалирања (*Python*)

број процесора	укупан посао	време извршавања	стандардна девијација	скалирано убрзање	теоријско убрзање (Густафсон)
1	60000	9.63s	0.07	1.00	1.00
2	120000	11.22s	0.78	1.71	1.95
4	240000	12.87s	1.62	2.99	3.85
8	480000	19.53s	0.36	3.94	7.65
16	960000	26.04s	0.38	5.92	15.25

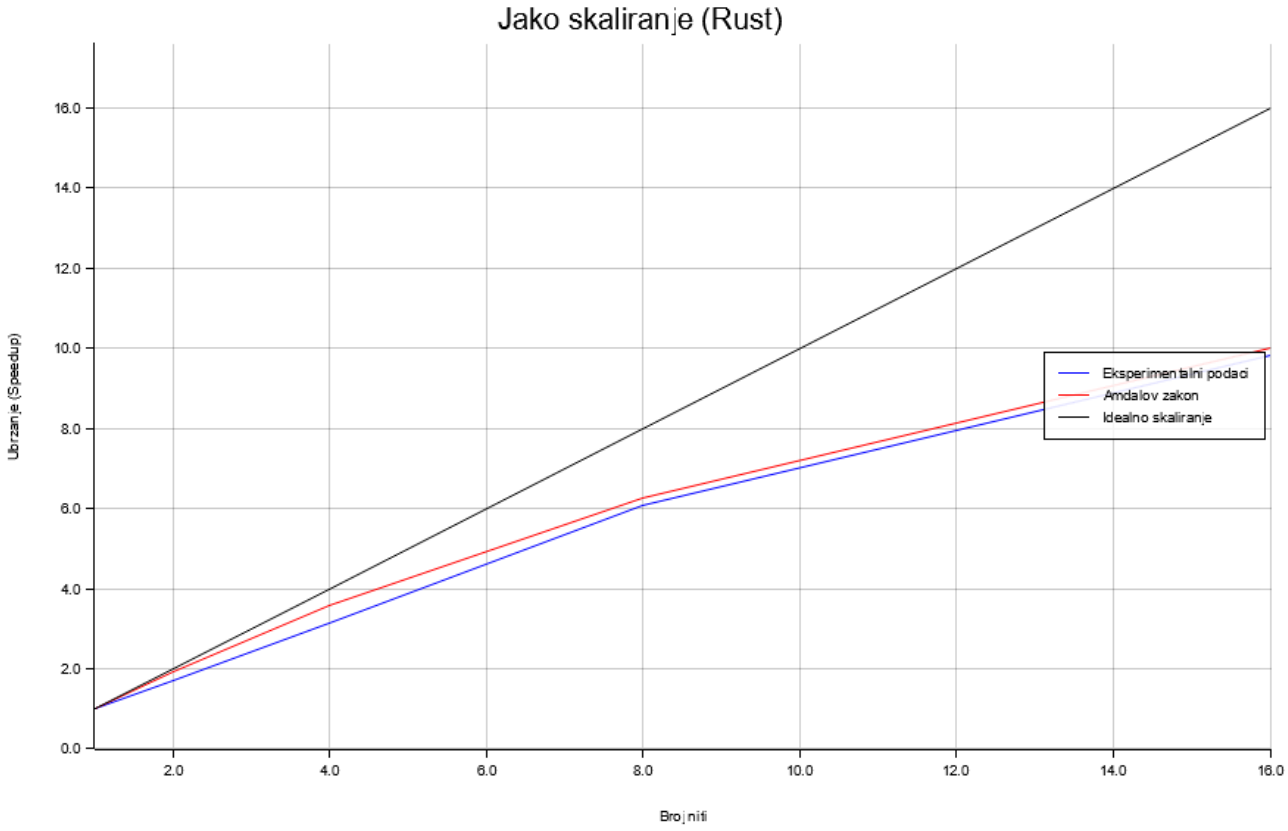
5.3 Анализа јаког и слабог скалирања у програмском језику *Rust*

5.3.1 Јако скалирање

Слика 10 и табела 5 описују јако скалирање. Као и у *Python*-у, симулирали смо 16 независних трајекторија, при различитом броју нити. Видимо да се убрзање креће близу Амдалове криве, што значи да се додавањем нових нити постиже реална добит у брзини.

Rust је постигао одлично јако скалирање, што је видљиво са графика. Треба нагласити да је *Rust* **компајлирани језик** познат по високој ефикасности и минималним режијским трошковима. Узет је већи укупан посао у односу на *Python* како би додатно указали на мање време извршавања.

Rust је показао врхунске апсолутне перформансе, што га чини погодним за израду научних симулација високих перформанси.



Слика 10: Јако скалирање (*Rust*)

Табела 5: Резултати јаког скалирања (*Rust*)

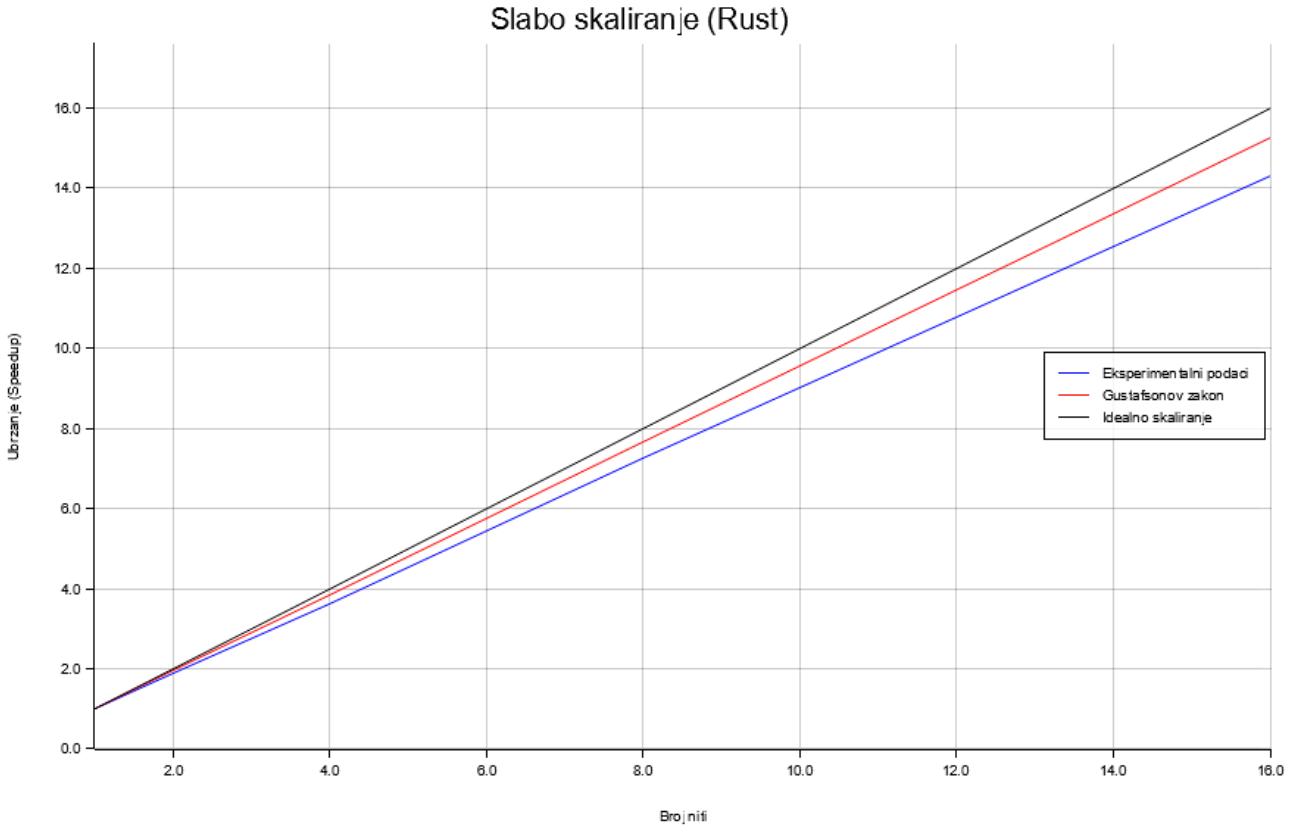
број процесора	време извршавања	стандардна девијација	убрзање	теоријско убрзање (Амдал)
1	1.76s	0.24s	1.00	1.00
2	1.04s	0.09s	1.70	1.92
4	0.56s	0.01s	3.13	3.57
8	0.29s	0.003s	6.07	6.25
16	0.18s	0.003s	9.82	10.0

5.3.2 Слабо скалирање

Резултате слабог скалирања можемо видети на слици 11 и табели 6. На графику се може уочити да се експериментална крива налази непосредно испод Густафсонове линије, што је у складу са очекивањима. Разлика између теоријског и стварног убрзања потиче углавном од режијских трошкова синхронизације и ограничења I/O (*input/output*) подсистема, који постају релевантни тек при великом броју нити.

За 16 нити, измерено скалирано убрзање износи 14.32, док теоријска вредност по Густафсоновом моделу износи 15.25, што представља преко 93% идеалног резултата. Ово показује да *Rust* имплементација приближава стварне перформансе теоријским границама.

Закључно, *Rust* се у овом експерименту показао као високо ефикасан језик за паралелну симулацију, са готово идеалним слабим скалирањем и минималним осцилацијама у времену извршавања.



Слика 11: Слабо скалирање (*Rust*)

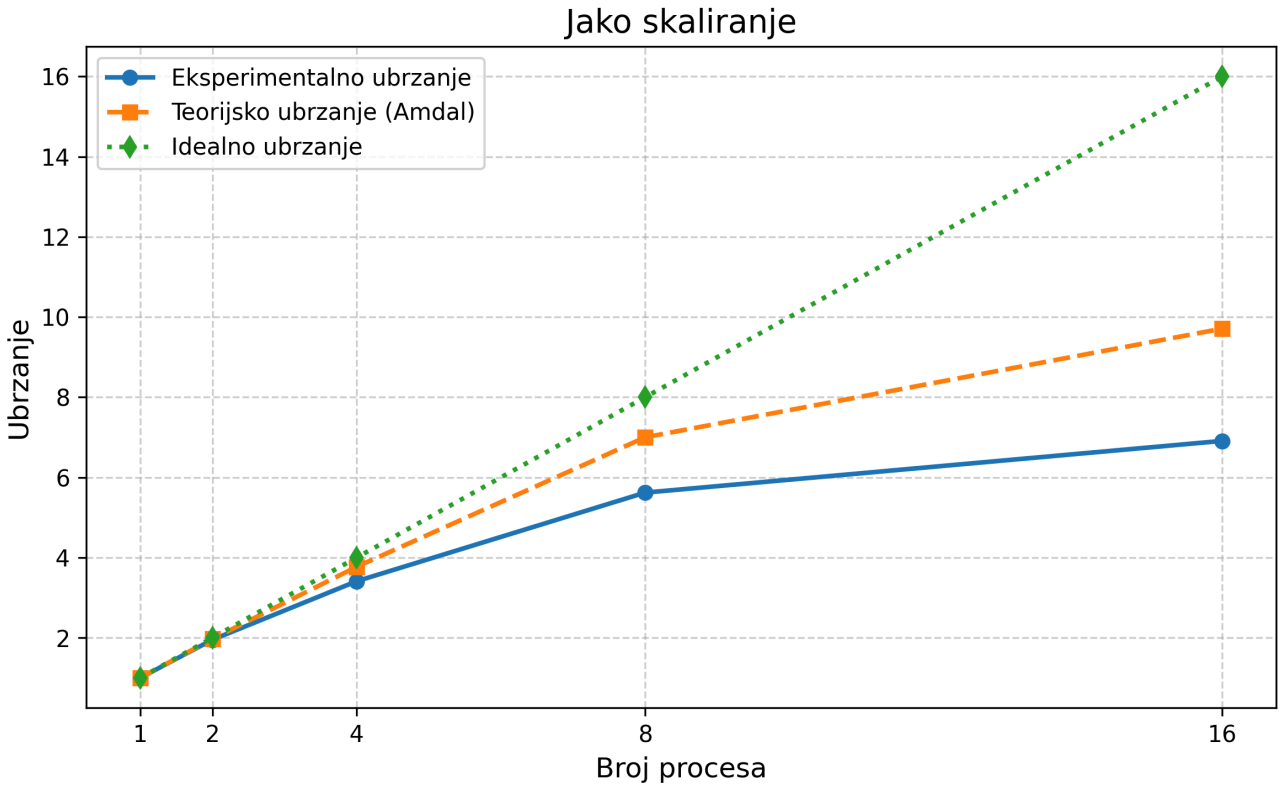
Табела 6: Резултати слабог скалирања (*Rust*)

број процесора	укупан посао	време извршавања	скалирано убрзање	теоријско убрзање (Густафсон)
1	60000	0.092s	1.00	1.00
2	120000	0.101s	1.82	1.95
4	240000	0.103s	3.56	3.85
8	480000	0.105s	7.02	7.65
16	960000	0.112s	14.32	15.25

5.4 Анализа јаког и слабог скалирања у програмском језику *Go*

5.4.1 Јако скалирање

Јако скалирање у *Go*-у и његове резултате (слика 12) можемо видети у наставку.



Слика 12: Јако скалирање (*Go*)

Табела 7: Резултати јаког скалирања (*Go*)

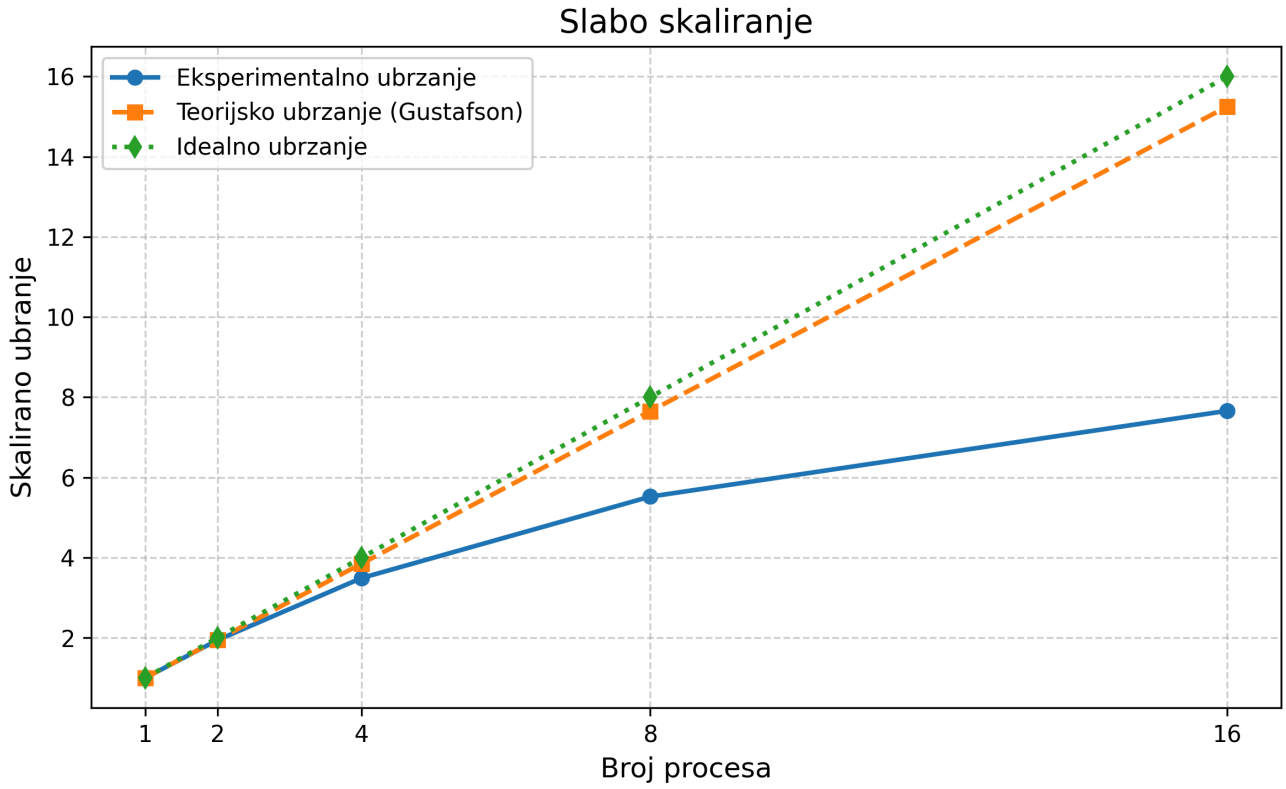
број процесора	време извршавања	стандардна девијација	убрзање	теоријско убрзање (Амдал)
1	4.8022s	0.051s	1.00	1.00
2	2.4633s	0.037s	1.95	1.97
4	1.4089s	0.028s	3.41	3.77
8	0.8542s	0.019s	5.62	7.00
16	0.6952s	0.022s	6.91	9.71

Резултати приказани у табели 7 показују да ова имплементација постиже боље јако скалирање у односу на *Python* и *Rust*. При фиксном укупном послу од 16 симулација двоструког клатна, време извршавања опада скоро линеарно са порастом броја процеса. Највећи пад времена примећује се при преласку са једног на два процеса, док се даљим повећањем броја процеса добија све мањи релативни добитак, што је у складу са Амдаловим законом.

За 16 процеса остварено је убрзање од 6.91, што представља приближно 71% теоријског убрзања (9.71) израчунатог за проценат паралелизованог кода $p = 97\%$. Времена извршавања су стабилна и показују минималну стандардну девијацију, што указује на стабилно управљање процесима унутар *Go runtime*-а.

5.4.2 Слабо скалирање

График слабог скалирања видимо на слици 13



Слика 13: Слабо скалирање (*Go*)

Табела 8: Резултати слабог скалирања (*Go*)

број процеса	укупан посао (број путања)	време извршавања	стандардна девијација	скалирано убрзање	теоријско убрзање (Густафсон)
1	2	0.6321s	0.013s	1.00	1.00
2	4	0.6511s	0.015s	1.94	1.95
4	8	0.7243s	0.018s	3.49	3.85
8	16	0.9153s	0.021s	5.52	7.65
16	32	1.3220s	0.029s	7.66	15.25

Резултати приказани у табели 8 показују да Go имплементација постиже **успешно слабо скалирање**, са постепеним растом времена извршавања како се број трајекторија повећава пропорционално броју процеса. Иако би у идеалном случају време извршавања остало константно, уочено је благо повећање — од 0.63 s на једном процесу до 1.32 s на шеснаест процеса. Ово повећање је очекивано, јер са већим бројем процеса расту и режијски трошкови синхронизације и расподеле посла.

Скалирано убрзање расте скоро линеарно до осам процеса, након чега долази до благог засићења. Најбољи однос добијен је за осам процеса, где је експериментално убрзање 5.52, што представља приближно 72% теоријског Густафсоновог убрзања.

Резултати потврђују да *Go* захваљујући горутинама и ефикасном *scheduler*-у веома добро распоређује додатни посао између процеса без значајног губитка перформанси.

Глава 6

Визуелизација

6.1 Мотивација и циљ

Резултати симулација добијени у овом раду иницијално се чувају у облику **CSV фајлова**, који садрже нумеричке вредности параметара система у сваком временском кораку интеграције. Сваки ред у овом фајлу представља једно стање система у времену, дефинисано следећим пољима: t - протекло време, θ_1, θ_2, ω_1, ω_2, *energy* - укупна енергија система

Пример садржаја оваквог фајла приказан је у табели 9:

Табела 9: Пример првих пар редова CSV табела који описују стање двоструког клатна.

t	θ_1	ω_1	θ_2	ω_2	energy
0.00	1.57	0.00	1.58	0.00	0.10
0.001	1.57	−0.01	1.58	0.00	0.10
0.002	1.57	−0.02	1.58	0.00	0.10
0.003	1.57	−0.03	1.58	0.00	0.10

Иако овакав нумерички приказ пружа све потребне информације о стању система, он не омогућава фин приказ динамике покрета. Вредности углова и брзина саме по себи не приказују визуелно сложену путању коју прави двоструко клатно, нити откривају хаотичну природу његовог кретања.

Због тога је **визуелизација** веома важна - омогућава да резултате симулације прикажемо графички, односно да посматрамо стварно кретање система током времена. На овај начин можемо лакше уочити карактеристичне обрасце као што су периодично, полупериодично или хаотично кретање, као и визуелно проверити физичку исправност решења.

6.2 Приказ трајекторије двоструког клатна

За визуелизацију резултата симулације развијен је програмски модул у језику **Rust**, који користи библиотеку `plotters` за генерисање графичких приказа [5]. Овај модул чита CSV фајлове са нумеричким вредностима добијеним из симулације и на основу углова θ_1 и θ_2 рачуна координате положаја сваке масе у систему.

Координате се рачунају помоћу углова на следећи начин:

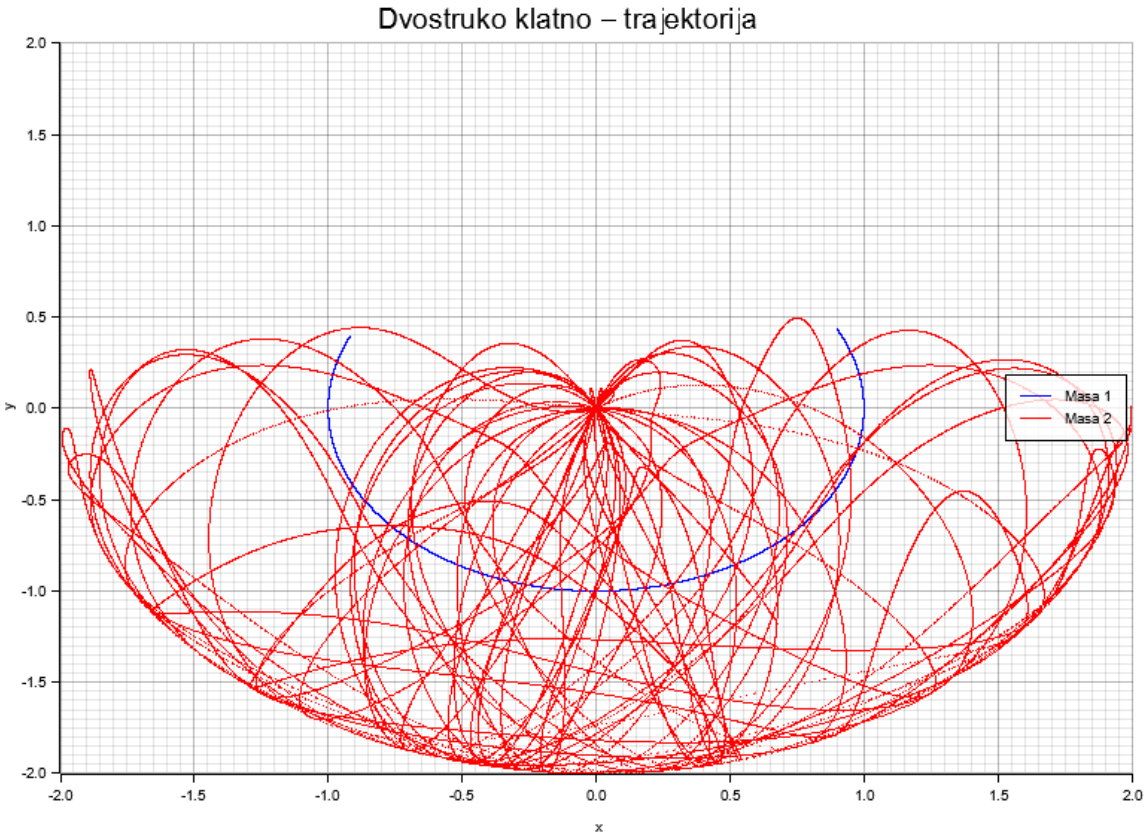
$$x_1 = l_1 \sin(\theta_1)$$

$$y_1 = -l_1 \cos(\theta_1)$$

$$x_2 = x_1 + l_2 \sin(\theta_2)$$

$$y_2 = y_1 - l_2 \cos(\theta_2)$$

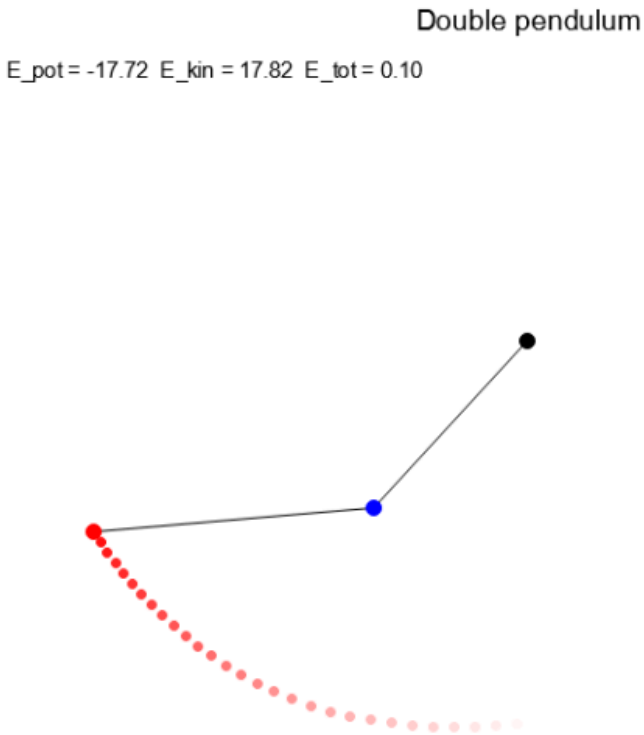
Програм затим црта путање обе масе током 60 секунди. На графику 14 су плава линија (путања прве масе) и црвена линија (путања друге масе), што омогућава јасан увид у динамику система.



Слика 14: Трајекторије горње и доње масе двоструког клатна.

6.3 Анимација кретања двоструког клатна

Поред статичког приказа путање, у оквиру овог рада реализована је и **динамичка визуелизација** - анимација кретања двоструког клатна у времену. Такође је коришћен *Rust* уз `plotters` библиотеку. Програм на исти начин као и пре рачуна положаје обе масе у простору, и у сваком временском кораку црта тренутни положај система. На крају је цела анимација сачувана у GIF формату.



Слика 15: Приказ једног кадра из анимације двоструког клатна.

На сваком кадру (слика 15) приказане су:

- позиције обе масе (плава и црвена)
- линије које представљају кракове клатна
- **тренутне вредности енергија** у горњем левом делу слике (потенцијална, кинетичка и укупна)

На анимацији се може видети да укупна енергија система остаје приближно **константна**, што потврђује нумеричку стабилност RK4 шеме. Један од кључних ефеката анимације је **ефекат трага** (*fade-out*). Наиме, на сваком кораку чува се одређени број претходних положаја доње масе, који се приказују постепено све блеђим нијансама црвене боје. Помоћу овог ефекта можемо интуитивно, чак и само уз помоћ слике одредити смер кретања клатна.

Глава 7

Закључак

7.1 Ретроспектива

У овом раду реализована је и анализирана симулација двоструког клатна у три програмска језика: *Python*, *Rust* и *Go*. Циљ је био да се пореде њихове перформансе, скалабилност и читљивост кода, као и да се испита ефикасност примене методе Рунге–Кута четвртог реда (RK4) у различитим извршним окружењима.

Резултати експеримената показали су следеће разлике између језика:

- *Python* је најједноставнији за имплементацију и најчитљивији по синтакси, али има изражена ограничења у перформансама због интерпретираног окружења и *Global Interpreter Lock (GIL)* механизма. Ипак, коришћењем `multiprocessing` библиотеке постиже се умерено јако и слабо скалирање, што га чини погодним за мање симулације и брз развој прототипа.
- *Rust* се показао као најбржи у погледу времена извршавања - много мање време него код Python-а, и у нашем случају је постигао ефективно јако скалирање. Стабилност резултата и стандардне девијације и строга контрола меморије потврђују да је *Rust* одличан избор за прецизне, детерминистичке научне симулације.
- *Go* је показао најбољи однос једноставности и перформанси. Његов модел конкурентности заснован на горутинама омогућио је скоро идеално јако и слабо скалирање до осам процеса, са стабилним временом извршавања и минималним *overhead*-ом. Код је прегледан, кратак и лако проширив, што чини овај језик веома практичним избором за паралелне симулације овог типа.

Поред анализе скалирања, посвећена је и пажња визуелизацији резултата. У *Rust*-у су реализовани модули за генерисање графичких приказа и анимација који приказују кретање клатна у реалном времену, са очувањем укупне енергије система и визуелним ефектима који илуструју хаотичну природу кретања.

Овим радом потврђено је да избор програмског језика зависи од приоритета — брзине развоја, стабилности или максималне перформансе — али да се у свим случајевима може постићи физички исправна и енергетски стабилна симулација сложених динамичких система.

7.2 Будући рад

Иако је симулација двоструког клатна успешно реализована и анализирана у оквиру овог рада, постоји више праваца у којима би се пројекат могао даље унапредити:

- **Већи скуп симулација** – извршавање стотина или хиљада независних симулација ради статистичке анализе хаотичког понашања и визуелизације фазног простора.
- **GPU акцелерација** – имплементација нумеричких интеграција на графичком процесору (*CUDA* или *OpenCL*) ради постизања вишеструког убрзања при већем броју паралелних трајекторија.
- **Веб интерфејс** – интеграција резултата симулација у веб апликацији (нпр. помоћу *WebAssembly* или *WebGL*), што би омогућило интерактивно покретање и визуелизацију клатна преко прегледача (нпр. [6]).
- **Интерактивна 3D визуелизација** – проширење *Rust* или *Go* модула помоћу *OpenGL/Vulkan* библиотека за приказ кретања у тродимензионалном простору са контролом углова и зумирањем у реалном времену.

Списак слика

Списак листинга

Списак табела

Списак коришћених скраћеница

Скраћеница	Опис
RK4	*Runge-Kutta* метода четвртог реда
CSV	Comma-Separated Values (текстуални формат за табеларне податке)
GIL	Global Interpreter Lock (механизам *Python*-а који омогућава извршавање само једне нити у исто време)
RAM	Random Access Memory (радна меморија рачунара)
GB/MB	Гигабајт/мегабајт (јединице меморије)
s	Секунд
GIF	Graphics Interchange Format (формат за анимиране растерске слике)
GPU	Graphics Processing Unit (графички процесор намењен за паралелна рачунања и обраду слика)
CUDA	Compute Unified Device Architecture (платформа за паралелно израчунавање на GPU-у)
2D/3D	Дводимензиони/тродимензиони простор

Списак коришћених појмова

Појам	Објашњење
θ_1	Угао првог сегмента двоструког клатна у односу на вертикалу
θ_2	Угао другог сегмента двоструког клатна у односу на вертикалу
ω_1	Угаона брзина првог сегмента клатна
ω_2	Угаона брзина другог сегмента клатна
Хаотични систем	Систем који показује осетљивост на почетне услове, што доводи до непредвидивог понашања током времена
Рунге–Кута метода	Нумеричка метода четвртог реда за решавање система диференцијалних једначина
Скалабилност	Мера способности алгоритма да ефикасно користи повећање рачунарских ресурса
Трајекторија	Путања тачке (масе) у равни током времена
Потенцијална енергија	Енергија услед положаја у гравитационом пољу
Кинетичка енергија	Енергија услед кретања маса
Јако скалирање	Промена времена извршавања при расту броја процеса за фиксан укупан посао
Слабо скалирање	Понашање система кад се посао увећава пропорционално броју процеса
Убрзање (speedup)	Однос T_1/T_p; колико је извршавање брже на p процеса/нити
Амдалов закон	Модел који процењује максимум убрзања на основу серијског процента кода
Густафсонов закон	Модел који описује раст ефикасности при увећању оптерећења са p
Стандардна девијација	Мера распршености мерених времена извршавања око средње вредности
Процес	Извршна јединица са сопственим адресним простором на оперативном систему
Нит (*thread*)	Лака извршна јединица унутар процеса која дели меморију
Goroutine	Лагана нит у *Go*-у коју планира *Go runtime scheduler*
Scheduler (Go)	Компонента *runtime*-а која распоређује горутине на нитима оперативног система
Ensemble паралелизација	Паралелно покретање више независних трајекторија/експеримената
I/O подсистем	Део система задужен за улаз/излаз; често уско грло при великом броју задатака
Plotters	*Rust* библиотека за израду дијаграма/слика (2D графички прикази)

Биографија

Зовем се Лука Петковић и рођен сам 14. јула 2002. годину у Новом Саду. Завршио сам Гимназију “Јован Јовановић Змај” 2021. године, смер обдарени ученици у математичкој гимназији, и добитник сам Вукове дипломе. Након тога, уписао сам Факултет Техничких наука, смер Софтверско инжењерство и информационе технологије, преко буџета. У току студирања, завршио сам тронедељну праксу у “Schneider Electric”-у, као и двомесечну праксу у компанији “Synechron”. Положио сам све предмете основних академских студија са просеком оцена 9.36. Тренирам стони тенис.

Литература

[1] Wikipedia contributors, „Chaos theory“. Приступљено: 28. Октобар 2025. [На Интернету]. Доступно на https://en.wikipedia.org/wiki/Chaos_theory

[2] J. A. Jr., „The Double Pendulum“. Приступљено: 28. Октобар 2025. [На Интернету]. Доступно на https://www.jhallard.com/blog/double_pendulum.html

[3] HPC Wiki, „Scaling in High Performance Computing“. Приступљено: 28. Октобар 2025. [На Интернету]. Доступно на https://hpc-wiki.info/hpc/Scaling

[4] GeeksforGeeks Contributors, „Goroutines and Concurrency in Golang“. Приступљено: 28. Октобар 2025. [На Интернету]. Доступно на https://www.geeksforgeeks.org/go-language/goroutines-concurrency-in-golang/

[5] Plotters Developers, „Plotters Rust Library Documentation“. Приступљено: 28. Октобар 2025. [На Интернету]. Доступно на https://docs.rs/plotters/latest/plotters/

[6] E. Neumann, „Double Pendulum Simulation“. Приступљено: 28. Октобар 2025. [На Интернету]. Доступно на https://www.myphysicslab.com/pendulum/double-pendulum-en.html