Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

РОЗРОБКА СИСТЕМИ ВІЗУАЛІЗАЦІЇ КОНКУРЕНТНИХ І ПАРАЛЕЛЬНИХ ПРОЦЕСІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Текстова частина

магістерської роботи

за спеціальністю “Інженерія програмного забезпечення” 121

Керівник магістерської роботи
к.н., доцент М. В. Почебут

_____
(підпис)

“____” _____ 2021 р.

Виконав студент В. В. Наквасюк
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики, к.ф.-м.н.
_____С. С. Гороховський
(підпис)
„____”_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія програмного забезпечення Наквасюку Василю Васильовичу
Розробити Систему візуалізації конкурентних і паралельних процесів програмного забезпечення

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Огляд предметної області конкурентних та паралельних підходів до розробки програмного забезпечення
2 Розробка архітектури для збору та візуалізації метаданих, які отримані під час виконання програмних застосунків
3 Реалізація інструменту у якості web-додатку для візуалізації конкурентних та паралельних процесів
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2020 р.

Керівник
М.В. Почебут, кандидат наук, доцент _____
(підпис)

Завдання отримав
В.В. Наквасюк _____
(підпис)

Тема: Розробка системи візуалізації конкурентних і паралельних процесів програмного забезпечення

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проєкту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	01.11.2020	
2.	Огляд технічної літератури за темою роботи	15.11.2020	
3.	Виконання аналізу сучасних підходів до багатозадачності	29.11.2020	
4.	Розробка архітектури системи для візуалізації конкурентних і паралельних програм	27.12.2019	
5.	Реалізація програмного web-додатку на базі розробленої архітектури	17.01.2020	
6.	Підтримка можливих інтеграцій програмного застосунку з декількома джерелами метаданих	14.03.2020	
7.	Написання пояснювальної записки	24.04.2021	
8.	Створення слайдів для доповіді та написання доповіді	27.04.2021	
9.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	30.04.2021	
10.	Вдосконалення роботи за результатами попереднього захисту	05.05.2021	
11.	Остаточне оформлення пояснювальної записки та слайдів	10.05.2021	
12.	Захист магістерської роботи (проєкту)	18.06.2021	

Студент _____

Керівник _____

“____”_____

ЗМІСТ

Анотація **6**

ВСТУП **7**

РОЗДІЛ 1. Огляд предметної області конкурентних та паралельних підходів до розробки програмного забезпечення **9**

1.1. Моделі комунікації в програмному забезпеченні 9

1.2. Конкурентні та паралельні обчислення 10

1.3. Види багатозадачності 15

РОЗДІЛ 2. Розробка архітектури для збору та візуалізації метаданих, які отримані під час виконання програмних застосунків **17**

2.1. Візуальне представлення коду 17

2.2. Вибір об’єкту дослідження 21

2.3. Аналіз існуючих підходів до візуалізації 23

2.4. Збір метаданих по виконанню програмного забезпечення 26

2.4. Збереження метаданих для подальшого використання 30

2.5. Загальна архітектура “Concurrency Debug Tool” (CDT) 32

2.5.1 Аналізатор зліпку 33

2.5.2 Серверна частина 33

2.5.3 Фронтенд web-додатку 33

2.5.4 CLI (Common Language Infrastructure) 34

РОЗДІЛ 3. Використання програмного застосунку “Concurrency Debug Tool” (CDT) для збору та візуалізації метаданих **37**

3.1. Приклади застосування візуалізації для класичних шаблонів конкурентного програмування 37

3.1.1 Ticker 37

3.1.2 Fan-in 40

3.1.3 Fan-out 41

3.2. Використання CDT під час еволюції програмного коду проєкта 44

Висновки по роботі та рекомендації для подальших досліджень **49**

Список літератури **51**

Додаток А. Програмний код компонента системи dump analyzer **54**

Додаток Б. Програмний код фронтенд web-компоненти системи **55**

Додаток В. Програмний код серверної web-компоненти системи **56**

Анотація

В рамках даної роботи проведено огляд конкурентних і паралельних підходів до розробки програмних застосунків, розроблено архітектуру по візуалізації процесів і подій, що відбуваються під час їх виконання, а також реалізовано програмний комплекс на базі розробленої архітектури у вигляді web-додатку.

Ключові слова: візуалізація, конкурентне програмування, паралельне програмування.

ВСТУП

Актуальність. За останні 20 років IT-технології зробили величезний крок вперед і зі складністю задач, які вони вирішують, зросла і складність програмного забезпечення. Насамперед це програмні продукти, які використовують конкурентне програмування. З іншого боку за законом Мура кількість ядер на процесорах теж збільшилась в рази і програми стали використовувати цю особливість, намагаючись утилізувати всі ядра процесора. Таким чином програмне забезпечення почало розроблятись за допомогою парадигм паралельного програмування.

Як результат, програми стали швидші, але й набагато комплексніші і складніші для написання і розуміння. Тому інструменти для візуалізації процесів, які відбуваються під час виконання таких програм, є дуже актуальні і необхідні під час розробки та налагодження програмного забезпечення.

Мета дослідження. Розробити систему для візуалізації конкурентних і паралельних процесів програмного забезпечення, яка допоможе більш детально розуміти послідовність виконання програми і події, які відбуваються в сучасних програмних застосунках.

Завдання дослідження. Провести аналіз предметної області конкурентних та паралельних підходів до розробки програмного забезпечення. Проаналізувати існуючі підходи до розробки інструментів візуалізації виконання програмних застосунків. Вивчити та дослідити програмні продукти, що можуть бути використані як компоненти в архітектурі інструменту для візуального представлення процесів в програмному застосунку. Розробити систему візуалізації та перевірити можливості розширення API і архітектури на декілька різних мов програмування.

Об´єкт дослідження. Процеси, що відбуваються під час виконання програмних застосунків та метадані, що описують ці конкурентні та паралельні процеси.

Предмет дослідження. Компоненти системи по візуалізації, їх можливості до розширення та інтеграції, API, розробка програмного web-застосунку та його інтеграція з API.

Джерела дослідження. Електронні версії друкованої літератури, документація програмних продуктів і бібліотек, вихідні коди програм та бібліотек, електронні ресурси, відеозаписи доповідей з конференцій по розробці програмного забезпечення.

Наукова новизна одержаних результатів дослідження полягає в аналізі і створенні візуального представлення процесів, які відбуваються у складних програмних застосунках використовуючи 2D і 3D моделі подій і потоків даних між співпрограмами.

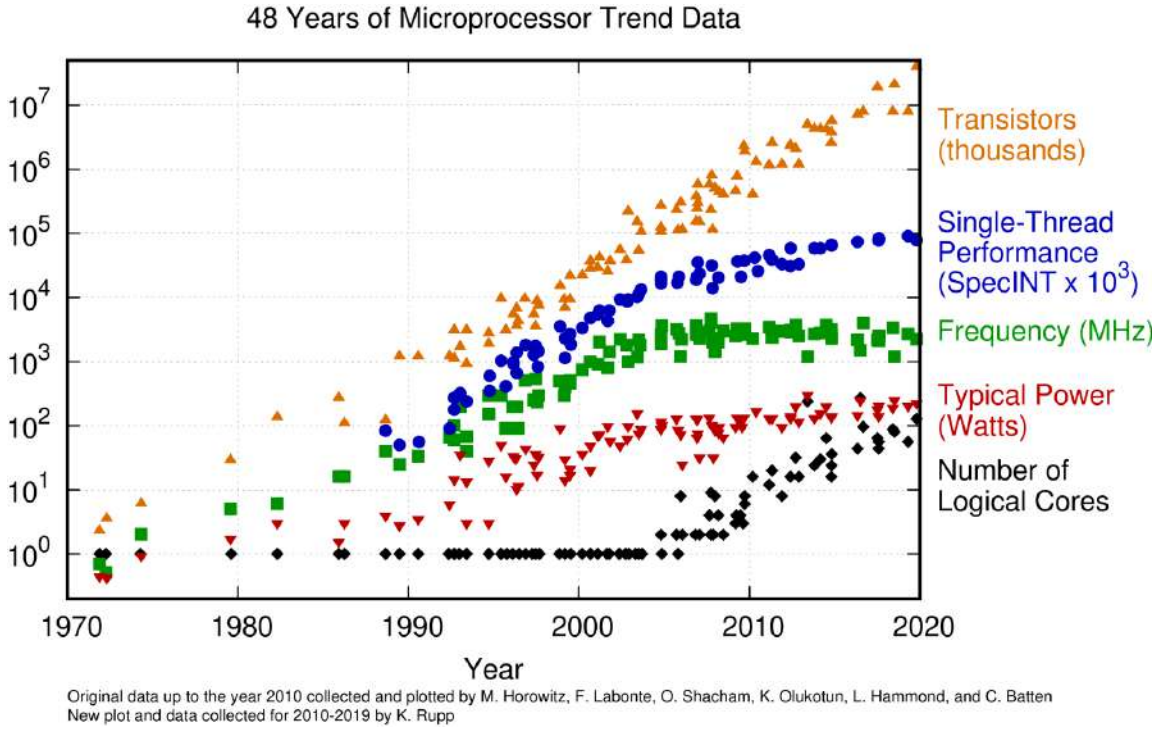
Практичне значення одержаних результатів. Використовуючи такий інструмент для візуалізації процесів конкурентних і паралельних програмах, зменшується час на налаштування і розробку, а також збільшується розуміння і уявлення про те, як саме працює програма в кожен момент часу. Розробник може легко проаналізувати, як співпрацюють співпрограми, як його програма змінювалась з часом і це значно покращить швидкість розробки і якість коду його програмного забезпечення.

РОЗДІЛ 1. Огляд предметної області конкурентних та паралельних підходів до розробки програмного забезпечення

1.1. Моделі комунікації в програмному забезпеченні

Як і передбачав емпіричний закон Мура, кожні 24 місяці кількість транзисторів на кристалі мікросхеми буде подвоюватися, при цьому частотні характеристики уповільнили своє зростання. З кожним роком кількість ядер у сучасних центральних процесорах збільшується і розробники програмного забезпечення намагаються їх ефективно утилізувати.

На наступному графіку [1] можемо побачити, як змінюється з часом і яка тенденція по кількість транзисторів, частотним характеристикам процесорів та іншим показникам:



Далі розглянемо дві моделі комунікації і взаємодії між співпрограмами, які часто використовуються для написання комплексних програмних продуктів:

1. Взаємодія через спільну память
2. Взаємодія через передачу повідомлень

Спільна пам'ять (англ. shared memory) – це коли ми використовуємо деяку загальну пам'ять і якась кількість ниток чи процесів мають до неї доступ і отримують з неї якісь дані. При цьому, щоб робота була коректною, доступ до

пам'яті треба синхронізувати. Ця синхронізація зазвичай будується на основі різного виду засобах синхронізації (локи, м’ютекси, тощо). Такий підхід часто називається неявною комунікацією.

На противагу першому підходу з використанням спільної пам'яті – існує підхід, який базується на передачі повідомлень (англ. message passing) і у своїй основі він використовує явну взаємодію. Інструментами для взаємодії між співпрограмами (актори, горутини) є більш високорівневі об'єкти, аніж локи, такі як “повідомлення”, “канали”, “поштова скринька процесу”, тощо. На такому принципі ґрунтується акторна модель і модель CSP (Communicating sequential processes). Як писав Роб Пайк “Не потрібно комунікувати через спільну пам’ять, а потрібно використовувати спільну пам’ять через комунікацію”.

В таких мовах програмування, які в своїй основі використовують підхід з передачею повідомлень, можна проаналізувати процеси і події, які відбуваються під час виконання програми. Так, наприклад, в мові програмування Golang існують такі більш високорівневі примітиви, як канали, повідомлення і співпрограми, що в Golang називаються горутинами (англ. goroutine). Такий підхід дозволяє значно зменшити всякого роду небажані стани гонитви (англ. race condition) та взаємні блокування (англ. deadlock). В мовах програмування Erlang і Elixir використовується акторна модель, яка схожа на CSP, і в ній також існують легкі процеси (співпрограми), які називаються акторами. Між собою актори комунікують повідомленнями напряму, без додаткових сутностей, як наприклад канали.

1.2. Конкурентні та паралельні обчислення

Розглянемо проблему, вирішити яку покликані конкурентність і паралелізм. Сучасні комп'ютери повинні вміти працювати з декількома завданнями одночасно, як наприклад, коли ми дивимося відео на YouTube, і без проблем можемо залишити під ним коментар чи відкрити якусь іншу web-сторінку у новій вкладці. При цьому відео без проблем продовжує відтворюватися.

Або ж ми можемо писати код в своєму улюбленому редакторі, в той час як у нас грає музика і викачуються файли в браузері.

Всі ці приклади об'єднує те, що в даних випадках відбувається кілька операцій одночасно, і при цьому вони не заважають один одному.

Однак архітектура сучасних комп'ютерів дозволяє нам виконувати безліч таких завдань одночасно, ну або майже одночасно, що ми і розглянемо далі. І досягається це якраз за рахунок конкурентності та паралелізму.

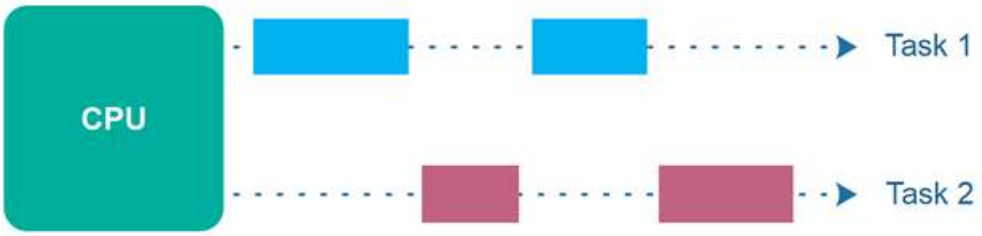
Конкурентні обчислення (англ. Concurrent computing) [2] — це форма обчислень, в якій кілька обчислень відбуваються в часових відрізках, що перетинаються, а не є послідовними (так, що кожне обчислення мусить закінчитись перед тим, як почнеться інше). Поняття рівночасних обчислень часто плутають з подібним, але дещо відмінним поняттям паралельних обчислень, хоча обидва описуються як «паралельні процеси що виконуються протягом спільних часових відрізків». У паралельних обчисленнях декілька обчислень існують і відбуваються одночасно в кожну мить фізичного часу (такт процесора), наприклад на різних процесорах багатопроцесорної машини, їх метою є зменшення часу виконання обчислень. У рівночасних обчисленнях декілька обчислень існують одночасно (як процеси або потоки виконання), але в кожну мить фізичного часу відбувається лише одне обчислення. Паралельні обчислення неможливі на одному одноядерному процесорі, оскільки в таких системах у кожен момент часу може відбуватися лише одне обчислення. Як наслідок, аспекти синхронізації паралельних обчислень визначаються особливостями апаратної платформи, а конкурентних — програмної. Поширеними, але не єдиними механізмами синхронізації конкурентних обчислень є сигнали, семафори та черги повідомлень. Метою застосування конкурентних обчислень є моделювання процесів реального світу, які відбуваються конкурентно, наприклад таких, як одночасний доступ кількох клієнтів до сервера. Побудова програмних систем, що складаються з багатьох комунікуючих частин, що працюють конкурентно, може бути корисною для зменшення складності таких систем незалежно від того, чи можуть

виконуватись їх частини паралельно. Типовою задачею конкурентних обчислень є реалізація механізмів планування процесів у багатозадачних системах.

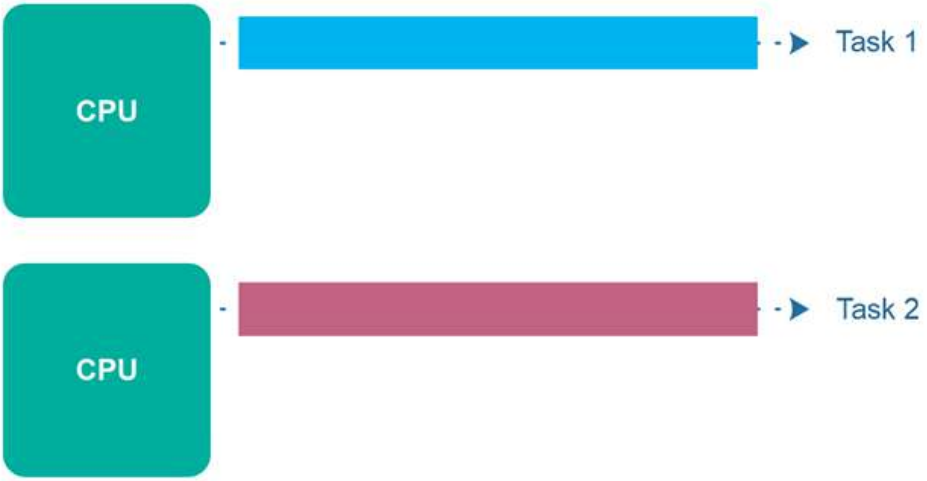
Отже, паралельні обчислення — це форма обчислень, в яких кілька дій проводяться одночасно. Паралелізм базується на тому, що великі задачі можна розділити на кілька менших, кожну з яких можна розв'язати незалежно від інших.

Як ми бачимо з визначення, конкурентна програма не обов'язково може бути паралельною, але паралельний підхід до написання програмного забезпечення потребує, щоб вона вміла працювати конкурентно.

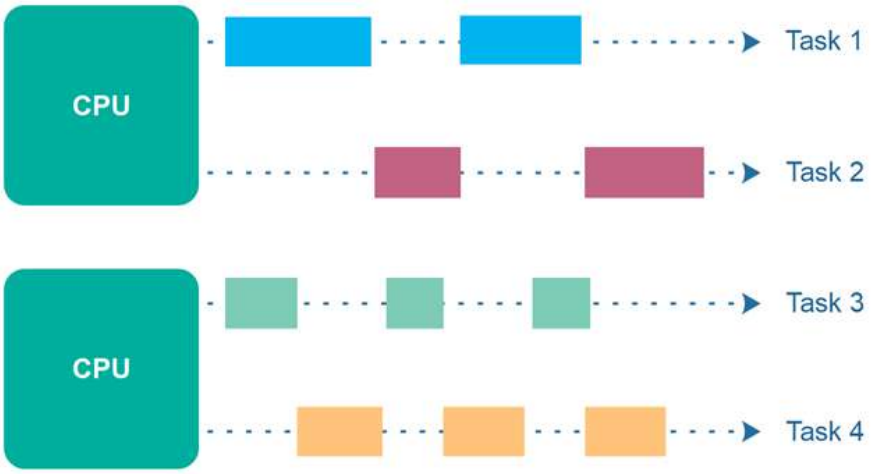
Розглянемо схематично, як виконуються задачі під час конкурентного взаємодії:



Як бачимо, якщо ми маємо 1 CPU ядро і 2 задачі (процеса), які потрібно виконати – то вони виконуються не одночасно, а послідовно. Аналогічно можемо розглянути схематичне виконання програми під час паралельного обчислення:



В цьому прикладі ми бачимо, що наші 2 задачі розпаралелились на 2 CPU ядра, і кожна з них виконуються налажено і одночасно з іншою. Хоча насправді в сучасному програмному забезпеченні все виконується конкурентно-паралельно, тому що зараз всі використовують багатоядерні процесори, операційні системи і програмні продукти, які працюють одночасно – і паралельно і конкурентно:



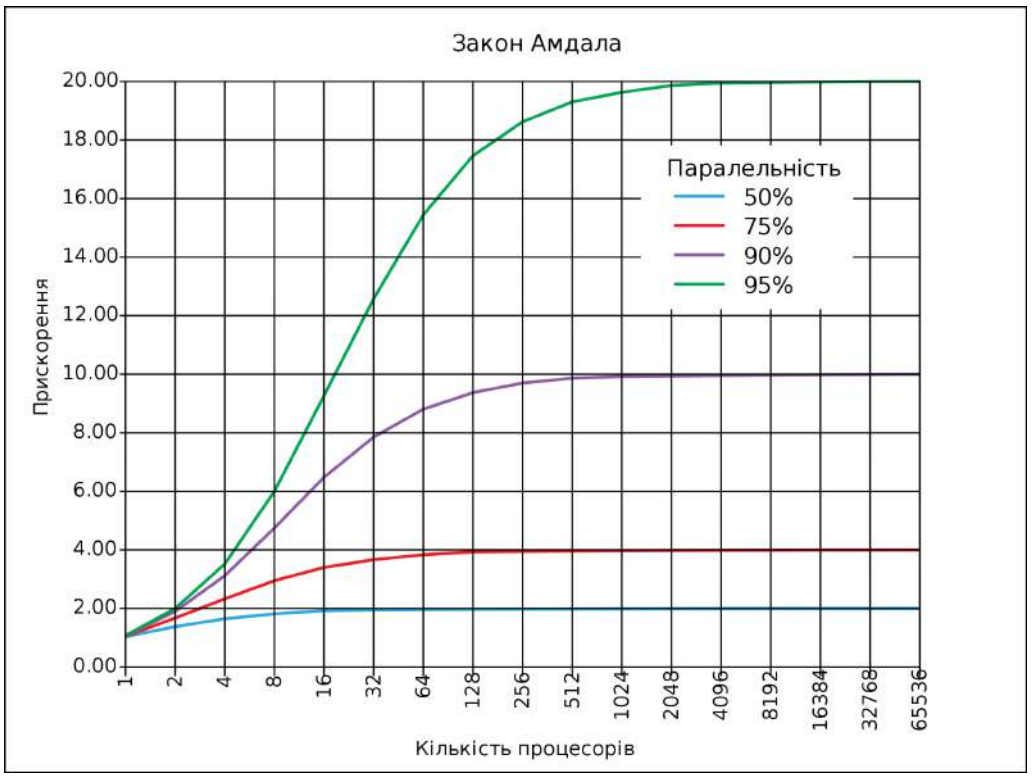
І хоча кількість ядер у центральних процесорах зростає і здається, що ефективно розпалалевши програмне забезпечення, можна прискорювати програму в рази – є досліди, які показують, що існує обмеження. Так званий Закон Амдала, який визначає потенційне прискорення алгоритму при збільшенні числа процесорів. Цей закон стверджує, що невелика частина програми, що не піддається розпаралелюванню, обмежить загальне прискорення від розпаралелювання. Будь-яка комплексна інженерна чи математична задача зазвичай буде складатись з декількох частин, що можуть виконуватись паралельно, та декількох частин задачі що виконуються тільки послідовно. Цей зв'язок задається рівнянням:

$$S = \frac{1}{p + \frac{1-p}{n}}$$

, де S — прискорення програми (як відношення до її початкового часу роботи); p — частина яку можна виконувати послідовно; n — кількість процесорів.

Якщо послідовна частина програмного забезпечення виконується 10 % всього часу роботи, неможливо прискорити виконання такого програмного забезпечення більше ніж у 10 разів незалежно від того, яку кількість процесорів використовує програма.

Схематично закон Амдала можна показати наступним чином:



Щоб зрозуміти різницю між конкурентністю і паралельністю, можна розглянути, як операційна система керує декількома пристроями одночасно (диск, екран, клавіатура тощо). Робота з цими пристроями по суті є незалежними, але в той самий час має бути відчуття одночасності подій. Однак ці події не обов'язково паралельні: якщо комп'ютер має лише одне CPU ядро, декілька речей не можуть працювати одночасно і вони працюють конкурентно. Модель взаємодії тут є конкурентною і вона структурована як система конкурентних процесів. Але насправді ця система подій може працювати і паралельно, тому що це залежить від обставин.

Конкурентність – це в першу чергу підхід проєктування програмного забезпечення, в той час як паралелізм – це спосіб його виконання.

1.3. Види багатозадачності

Існує 2 типи багатозадачності:

1. Кооперативна багатозадачність
2. Витісняюча багатозадачність

Кооперативна багатозадачність, також відома як багатозадачність без витіснення, це стиль комп'ютерної багатозадачності, в якому операційна система ніколи не ініціює перемикання контексту від запущеного процесу до іншого процесу. Замість того, процеси періодично добровільно поступаються контролем, або перебувають в режимі очікування, для того щоб кілька програмних застосунків мали можливість працювати одночасно. Цей тип багатозадачності називається «кооперативним», тому що всі програми повинні співпрацювати для роботи всієї схеми планування. У цій схемі процес планування операційної системи відомий як кооперативний планувальник, і його роль зводиться до запуску процесів і дозволу їм повернути управління назад добровільно.

Через те, що кооперативно багатозадачна система потребує того, щоб кожен процес регулярно віддавав час іншим процесам в системі, одна неефективно розроблена програма може використовувати весь процесорний час для свого виконання, або шляхом проведення великих розрахунків або через стан активного очікування. Обидва фактори можуть призвести до того, що система зависне. Проте, кооперативна багатозадачність дозволяє набагато простішу реалізацію програмних застосунків, оскільки їх виконання ніколи не може бути несподівано перерваним процесорним планувальником.

На противагу кооперативній багатозадачності існує витискальна багатозадачність — це вид багатозадачності, при якому операційна система може тимчасово перервати поточний процес без будь-якої допомоги з його боку. Витискальна багатозадачність перериває виконання програмного застосунку та передає управління іншим процесам поза контролем програми.

Таким чином в витискальній багатозадачності момент, в який переключити контекст, вирішує планувальник операційної системи чи будь-який інший в рамках виконання програми. А у кооперативній багатозадачності сам процес чи програма вирішує, коли йому переключитись на іншу співпрограму. Цікавий момент, що у корпоративній багатозадачності програма може монополювати весь ресурс процесора. При цьому в витісняючій багатозадачності таке неможливо, так як є контролюючий орган у вигляді планувальника і він розподілить, скільки часу буде виконуватись кожна співпрограма.

На практиці, наприклад, будь-яка сучасна операційна система використовує витісняючу багатозадачніть, щоб гарантувати, що кожен процес буде виконаний і ніякий з них не займе всі ресурси, спровокувавши таким чином підвисання всієї операційної системи. В мовах програмування використовуються і той, і інший підхід для керування конкурентними і паралельними процесами. Наприклад мова програмування Golang використовує кооперативний підхід для свого планувальника, а Erlang (Elixir) – некорпоративний (витіснячий).

РОЗДІЛ 2. Розробка архітектури для збору та візуалізації метаданих, які отримані під час виконання програмних застосунків

2.1. Візуальне представлення коду

Кажуть, що краще один раз побачити, ніж сто разів почути. Сьогодні, в епоху великих даних, коли компанії тонуть в інформації з самих різних локальних і хмарних джерел, ця приказка актуальна як ніколи. Візуальні засоби прискорюють і спрощують цей процес, а також дозволяють миттєво побачити найважливіше. Крім того, більшість людей сприймають візуальні образи краще, ніж текст: 90% інформації, що надходить в мозок, - це зображення, тому він обробляє їх в 60000 разів швидше, ніж текст.

Метод візуалізації – це систематичне, засноване на правилах, зовнішнє, постійне та графічне зображення, яке зображує інформацію таким чином, що сприяє набуттю розумінь, виробленню детального розуміння або передачі досвіду. Перш за все, потрібно знати, що ж таке візуалізація даних і які її методи використовуються, в тому числі і в повсякденному житті.

Візуалізація даних - це наочне представлення масивів різної інформації. Існує кілька типів візуалізації [4]:

- Звичайне візуальне уявлення кількісної інформації в схематичній формі. До цієї групи можна віднести всім відомі кругові та лінійні діаграми, гістограми і спектрограми, таблиці і різні точкові графіки.
- Дані при візуалізації можуть бути перетворені в форму, яка посилює сприйняття і аналіз цієї інформації. Наприклад, карта і полярний графік, тимчасова лінія і графік з паралельними осями, діаграма Ейлера.
- Концептуальна візуалізація дозволяє розробляти складні концепції, ідеї і плани за допомогою концептуальних карт, діаграм Ганта, графів з мінімальним шляхом та інших подібних видів діаграм.
- Стратегічна візуалізація переводить в візуальну форму різні дані про аспекти роботи організацій. Це всілякі діаграми продуктивності, діаграми життєвого циклу і графіки структур організацій.

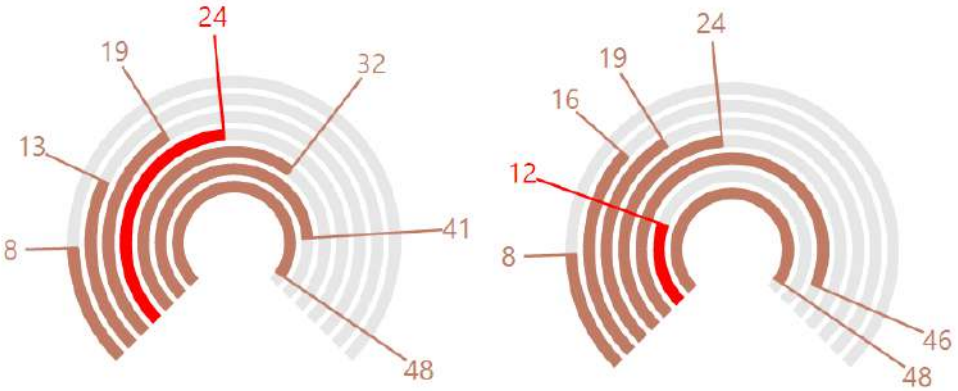
- Графічно організувати структурну інформацію за допомогою пірамід, дерев і карт даних допоможе метафорична візуалізація, яскравим прикладом якої є карта метро.
- Комбінована візуалізація дозволяє об'єднати кілька складних графіків в одну схему, як в карті з прогнозом погоди.

Навіщо використовувати візуалізацію даних? Візуальна інформація краще сприймається і дозволяє швидко і ефективно донести до глядача власні думки та ідеї. Фізіологічно, сприйняття візуальної інформації є основною для людини. Є численні дослідження[6], які підтверджують, що:

- 90% інформації людина сприймає через зір
- 70% сенсорних рецепторів знаходяться в очах
- близько половини нейронів головного мозку людини задіяні в обробці візуальної інформації
- на 19% менше при роботі з візуальними даними використовується когнітивна функція мозку, що відповідає за обробку і аналіз інформації
- на 17% вище продуктивність людини, що працює з візуальною інформацією
- на 4,5% краще згадуються деталі візуальної інформації
- в 60000 разів швидше сприймається візуальна інформація в порівнянні з текстовою
- 10% людина запам'ятовує з почутого, 20% - з прочитаного, і 80% - з побаченого і зробленого
- на 323% краще людина виконує інструкцію, якщо вона містить ілюстрації

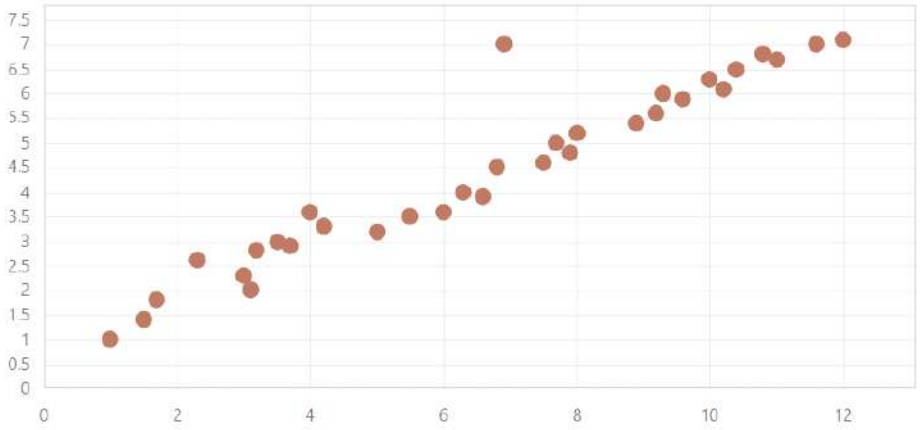
Очевидно, що людина схильна обробляти саме візуальну інформацію. Крім прекрасної обробки нашим мозком, візуалізація даних має кілька переваг:

- Акцентування уваги на різних аспектах даних



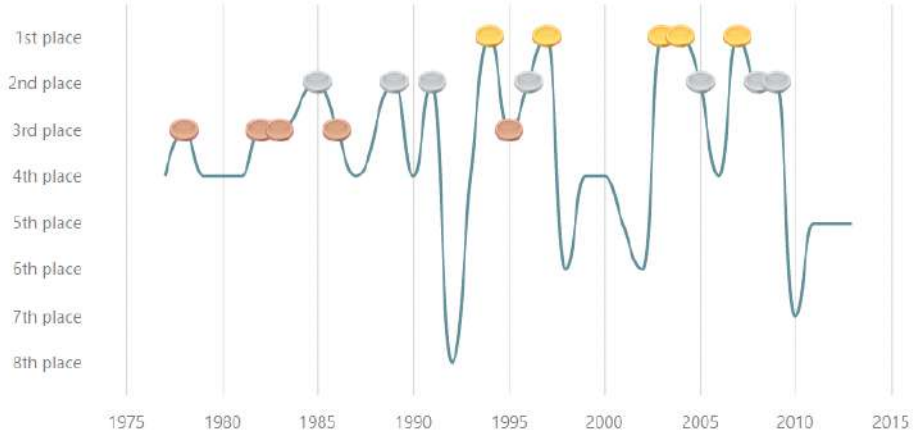
За допомогою графіків можна легко звернути увагу читача на червоні показники

- Аналіз великого набору даних зі складною структурою
- Зменшення інформаційного перевантаження людини і утримання його уваги
- Однозначність і ясність виведених даних
- Виділення взаємозв'язків і відносин, що містяться в інформації



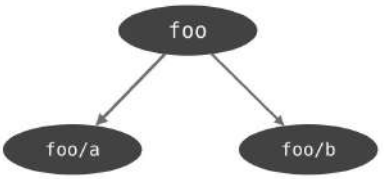
На графіку легко можна помітити важливі дані

- Естетична привабливість



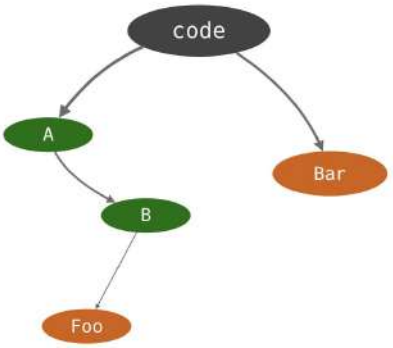
Якщо розглянути важливість правильного підходу до візуалізації сутностей в програмному забезпеченні, то можемо взяти наступний приклад. Нехай нам потрібно зробити візуалізацію структури програми. Спочатку можна відобразити високорівневі сутності, такі як пакети програмного забезпечення:

```
// foo.go
package foo

--
// a/a.go
package a

--
// b/b.go
package b
```



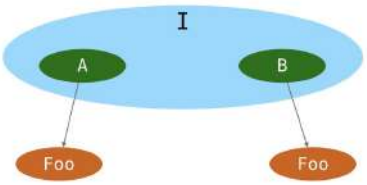
Потім можна розглянути детальніше кожен пакет і побудувати представлення структур, класів, типів, функцій в кожному з пакетів:

```
package code

type A struct {
    b *B
}

type B struct {}

func (B) Foo() {}

func Bar() {}
```



Далі по аналогії можна візуалізувати, наприклад, інтерфейси і візуально підкреслити це на графіку:

```
type I interface {
    Foo()
}

type A struct {}
func (A) Foo() {}

type B struct {}
func (B) Foo() {}
```



Візуалізація – потужний інструмент донесення думок і ідей до кінцевого користувача, помічник для сприйняття і аналізу даних. Але як і всі інструменти, її потрібно застосовувати в свій час і в своєму місці. В іншому випадку інформація може сприйматися повільно, а то і некоректно. Отже візуалізація є дуже важливим і потрібним інструментом для розуміння і маніпуляції з об'єктами нашої системи, для ширшого погляду на сутності і процеси, що відбуваються в ній.

2.2. Вибір об’єкту дослідження

В минулому розділі ми розглянули різного роду підходи до розробки конкурентних і паралельних програмних застосунків. В цій роботі ми сфокусуємося на реалізації через передачу повідомлень між “легкими” співпроцедурами. І для аналізу метаданих, які будемо збирати під час виконання програми, виберемо мову програмування Golang з її реалізацією конкурентності на горутинах.

В Golang використовується парадигма Communicating Sequential Processes (CSP) і вона дуже схожа до акторної моделі. Ідея CSP також базується на передачі повідомлень без спільного використання пам'яті. Однак у CSP та акторів є 2 ключові відмінності:

- Процеси в CSP є анонімними, тоді як актори мають ідентичність. Отже, CSP використовує явні канали для передачі повідомлень, тоді як з акторами ви надсилаєте повідомлення безпосередньо.

- За допомогою CSP відправник не може передати повідомлення, поки одержувач не буде готовий його прийняти. Актори можуть надсилати повідомлення асинхронно.

Основними примітивами для синхронізації в Golang є горутини, канали, та повідомлення, які по каналам надсилаються між горутинами.

Горутини – це функція, яка може працювати конкурентно з іншими функціями. Для створення горутини використовується ключове слово *go*, за яким слідує виклик функції:

```
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```

Ця програма складається з двох горутин. Функція main, сама по собі, є горутиной. Друга горутина створюється, коли ми викликаємо *go f(0)*. Зазвичай, при виконанні функції, програма виконає всі конструкції всередині функції, яка викликається, а тільки потім перейде до наступного, після виклику, рядка. З горутиной програма негайно перейде до наступного рядка, не чекаючи доки функція, яку ми викликали, завершиться. Ось чому тут присутній виклик *Scanln*, так як без нього програма завершиться ще перед тим, як їй вдасться вивести числа, які ми друкуємо на екран у горутині *f*.

Канали забезпечують можливість спілкування декількох горутин між собою, щоб синхронізувати їх виконання. Ось приклад програми з використанням каналів:

```
package main

import (
    "fmt"
)

func hello(done chan bool) {
    fmt.Println("Hello world goroutine")
    done <- true
}
func main() {
    done := make(chan bool)
    go hello(done)
    <-done
    fmt.Println("main function")
}
```

Ця програма створює канал з булевим типом даних *bool*. Після цього запускається горутина, яка пише на екран повідомлення і відсилає в канал значення *true*, яке в свою чергу очікує головна горутина *main*. Далі основна горутина отримує повідомлення з каналу, пише на екран повідомлення і програма завершується.

2.3. Аналіз існуючих підходів до візуалізації

Давайте розглянемо, які інструменти вже існують на ринку, що можуть візуалізувати багатозадачність програмного забезпечення.

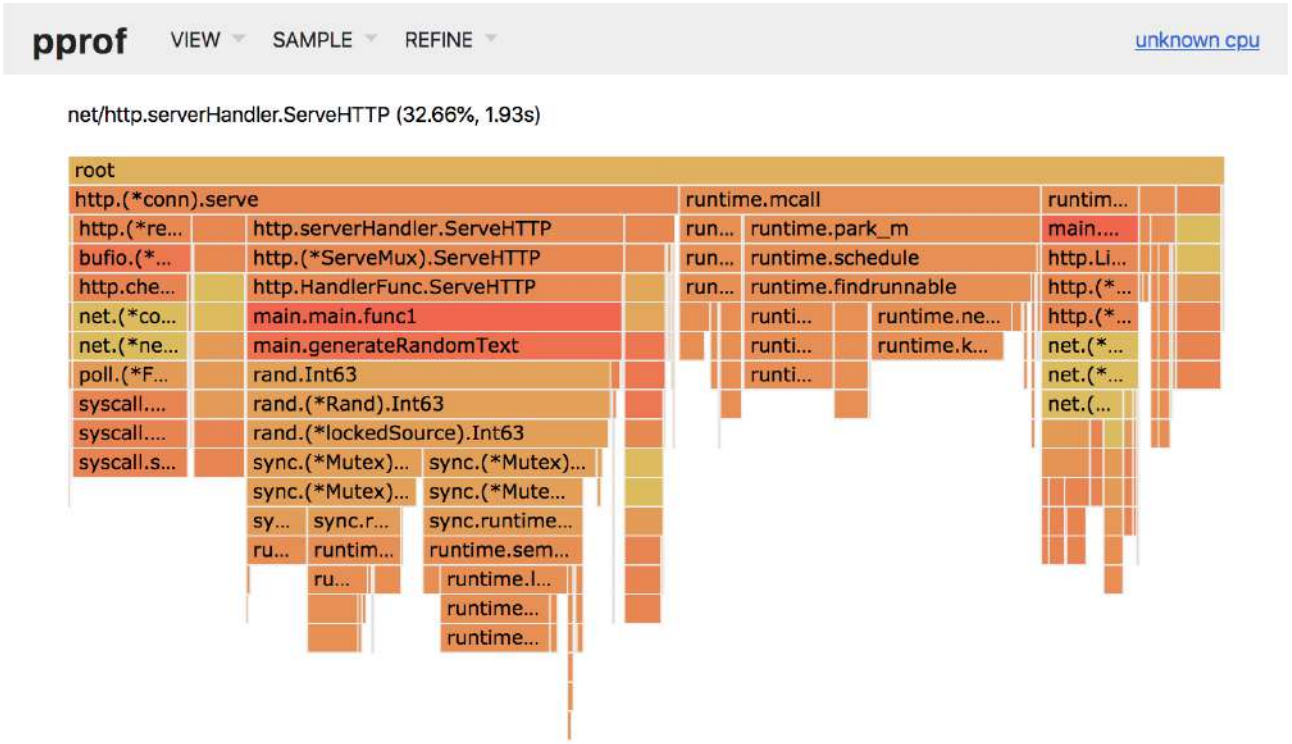
Для такого роду задач у Visual Studio існує опціональний модуль, який називається Visualizer. Ви можете використовувати цей інструмент, щоб побачити, як працює багатопотокова програма. Представлення у Visualizer надають графічні, табличні та текстові дані, які показують часові взаємозв'язки між потоками у вашій програмі та в системі в цілому. Ви можете використовувати цей візуалізатор конкурентності для пошуку вузьких місць

продуктивності, недостатньої утилізації центрального процесора, міграції між’ядерних потоків, затримок синхронізації, активності DirectX, точок взаємозв'язку операцій вводу-виводу та іншої інформації.

Для мови програмування Golang є вбудовані інструменти для профілювання і трейсингу – *go pprof* та *go trace* відповідно. Профайлинг може показати наступні характеристики нашого програмного застосунку:

- Профілювання процесора
- Профілювання пам’яті
- Профілювання блокувань
- Профілювання м’ютексів (англ. mutex)

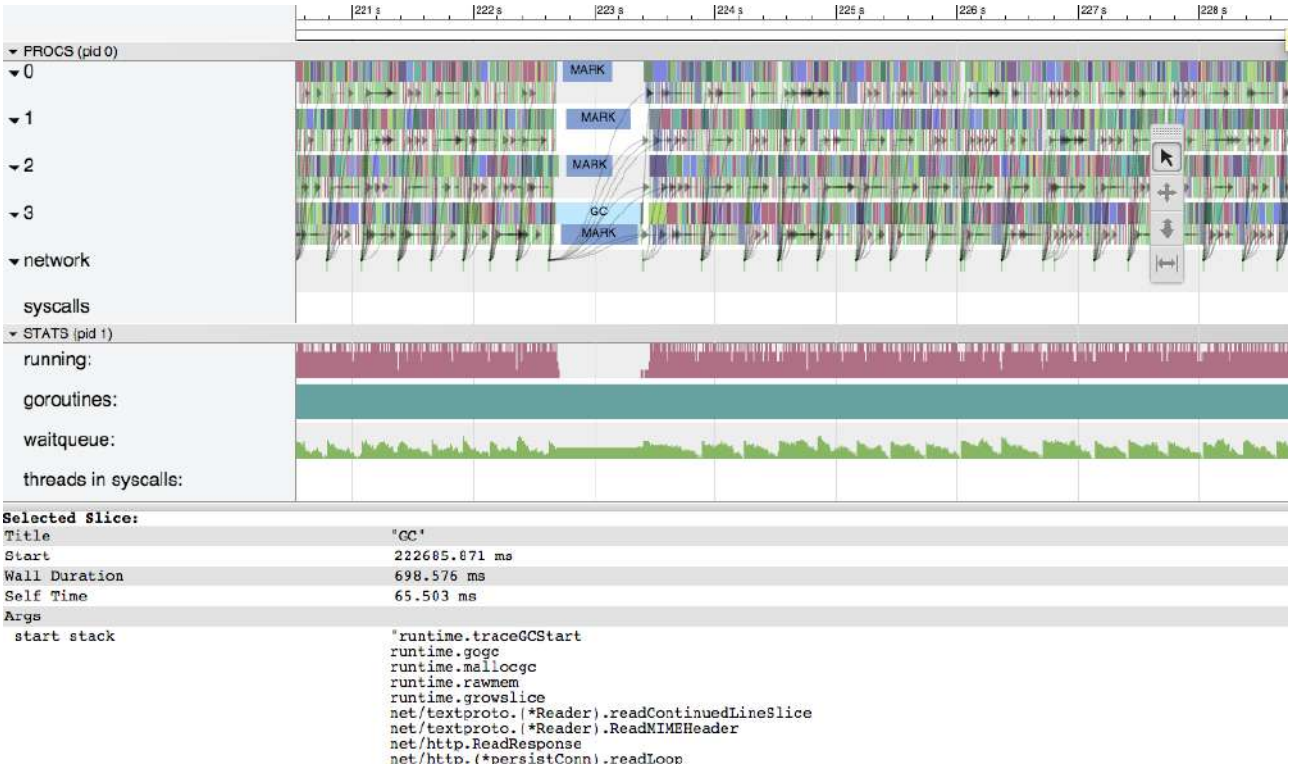
Такий вид інформації візуалізується, наприклад, через “Flame Graphs” графіки:



Якщо говорити про трейсинг, то у програм на Golang є можливість записувати ряд цікавих подій у файл. Ця інформація компілюється у всіх програмах завжди і включається на вимогу. Наприклад, якщо трейсинг відключено, програма має мінімальні накладні витрати. Трейсинг містить події, пов'язані з плануванням роботи програми:

- коли програма починає виконуватися на процесорі
- коли програма керує примітивами синхронізацій
- коли програма створює або розблоковує іншу програму
- інформацію, що пов'язана з мережею події
- коли програма керує IO (ввід-вивід)
- події, пов’язані з системними викликами
- події, пов'язані зі збирачем сміття (англ. garbage collection)
- події користувачів

Після отримання трейсинг-інформації її можна візуалізувати або обробити для отримання різних типів корисних звітів. Нижче ви можете побачити можливу візуалізацію:



Як ми бачимо, всі вищезазначені інструменти більшою мірою фокусуються на багатозадачність через призму процесів, потоків виконання, виділення пам'яті, тощо. Це корисна інформація, але працювати і аналізувати її буває досить складно, а отже і складно зрозуміти, що ж відбувається під час виконання програмного забезпечення. Також, як видно з цих графіків, інформація представлена у плоскому 2D вигляді, що не завжди зручно для розуміння. В цій роботі я б хотів сфокусуватися на виконанні більш високорівневих примітивів – а саме на “легких” процесах, таких як горутина та актори. Комунікація між ними відбувається явно, тому аналізувати, візуалізувати і зрозуміти такий вид інформації набагато простіше.

2.4. Збір метаданих по виконанню програмного забезпечення

Тепер, коли ми визначилися з мовою програмування і основними примітивами синхронізації, якими вирішують завдання конкурентності в ній, можемо перейти до аналізу програми і збору метаданих під час її виконання. Для цього будемо користуватися наступними інструментами:

- git
- gotrace
- go execution tracer

- go pprof

Для початку більш детально розглянемо вбудований інструмент для трейсингу, який називається *go trace*. Коли ви включаєте трейсинг, то після запуску програми на Golang в дуже компактному вигляді в файл записується майже все, що робить програма, і те, що відбувається з горутинами, пам'яттю чи нитками: очікування на каналі, старт після очікування на каналі, інформація про м'ютекси, системні виклики і т.д. Повний список з коментарями можна подивитися у вихідних кодах Golang на початку файлу *"runtime/trace.go"*. Візьмемо наступний приклад:

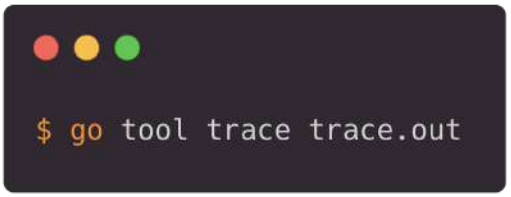
```
package main

import (
	"os"
	"runtime/trace"
)

func main() {
	trace.Start(os.Stderr)
	defer trace.Stop()
	// create new channel of type int
	ch := make(chan int)

	// start new anonymous goroutine
	go func() {
		// send 42 to channel
		ch <- 42
	}()
	// read from channel
	<-ch
}
```

Інструкція *trace.Start(os.Stderr)* говорить про те, щоб почати збір даних по трейсингу і перенаправляти все в стандартний потік виводу помилок *stderr* (потік номер 2).

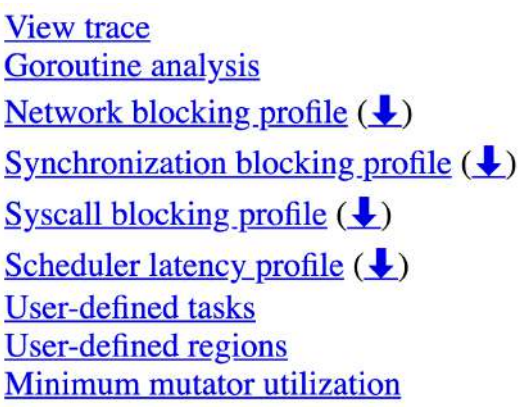
Інструкція *defer trace.Stop()* вказує, що після виходу з основної функції *main*, потрібно коректно завершити трейсинг і вивести метадані у стандартний потік *stderr*.

Після цього ми можемо запустити нашу програму і перенаправити потік виводу помилок в окремий файл *“trace.out”*:

Далі нам потрібно відкрити цей файл за допомогою стандартної утіліти *go trace* і можна побачити різного рода візуалізацію, а також технічну інформацію, яку ця утіліта витягує з бінарного файла *“trace.out”*.



Після цього ми бачимо, що на порту 55641 був піднятий web-сервер, який надає web-інтерфейс до наших результуючих даних:



Як ми вже описували і зазначали вище, інформація, що надається цим встроєним інструментом досить низькорівнева, а наша ціль зробити візуалізацію більш доступно для користувача. Для цього скористаємося утілітою gotrace, яка розширює можливості нативного *go trace* і додає більше інформації про горутини.

Нажаль, *go trace* і *go pprof* можуть видавати інформацію про використання пам'яті, CPU і про кількість горутин, але не в реальному часі. Тобто інформація сумарна, кінцева. Нам би хотілося бачити динаміку, тому що саме як змінюються дані під час виконання програми і є самим цікавим і корисним для розуміння внутрішніх процесів. Для цього напишемо самі простий пакет *monitor.go:*

```
package monitor

import (
	"log"
	"runtime"
	"time"
)

func System() {
	mem := &runtime.MemStats{}

	for {
		cpu := runtime.NumCPU()
		log.Println("<CDT> CPU:", cpu)

		rot := runtime.NumGoroutine()
		log.Println("<CDT> Goroutine:", rot)

		runtime.ReadMemStats(mem)
		log.Println("<CDT> Memory:", mem.Alloc)

		time.Sleep(100 * time.Millisecond)
		log.Println("-------")
	}
}
```

Після цього парсимо AST-дерево вихідного коду, який ми будемо запускати, і додаємо 1 додаткову строку:

```
go monitor.System()
```

Таким чином ми будемо додатково збирати інформацію про використання ресурсів у динаміці і також виводити це у вигляді графіків в наш програмний комплекс.

2.4. Збереження метаданих для подальшого використання

Після того, як ми зібрали сирі метадані в бінарному вигляді, нам потрібно зібрати цю інформацію, проаналізувати і вибрати тільки те, що цікавить саме нас. Після цього всі ці метадані ми будемо зберігати в базі даних у вигляді певних структур і типів.

Введемо таке поняття, як подія. Типів подій є декілька:

- "create_goroutine"
- "stop_goroutine"
- "send_to_channel"
- "block_goroutine"
- "unblock_goroutine"
- "sleep_goroutine"

А структура події має наступний вигляд:

time (integer) - timestamp часу, коли зафіксована подія
command (string) - ідентифікатор команди/події
name (string) - ідентифікатор горутини, в якій зафіксована подія (опційно)
parent (string) - ідентифікатор батьківської горутини, яка породила поточну
from (string) - ідентифікатор горутини, з якої надсилається щось в канал
to (string) - ідентифікатор горутини, яка приймає щось з каналу
channel (string) - ідентифікатор каналу
value (string) - значення, що надсилається в канал
duration (integer) - час між подіями в рамках однієї горутини
depth (integer) - рівень вкладеності горутин (0 для головної горутини main)

Додатково також у базу даних будемо зберігати номер git-коміта, номер ревізії (якщо в рамках одного git-коміта буде декілька збірок даних), вихідний код, а також інша інформація, яку можна гармонічно додати до нашої візуалізації.

Щоб вибрати, яку саме базу даних будемо використовувати, слід зауважити, що наш інструмент для візуалізації має легко розгортатися в середовищі користувача. Тому нам потрібно обрати просту і перевірену базу даних, до якої написано багато драйверів під різні мови програмування, і для якої не потрібно робити додаткових кроків по її встановленню і розгортці. Звичайно, ми могли б скористатися самою простою базою даних – це нашою файловою системою і просто зберігати все прості текстові чи бінарні файли, але ж нам також і потрібен зручний доступ до інформації. Отже, розуміючи всі наші обмеження і знаходячи компроміси, давайте розглянемо вбудовані (англ. embedded) бази даних. На сьогоднішній день на ринку існують наступні популярні варіанти:

- SQLite
- Apache Derby
- LevelDB

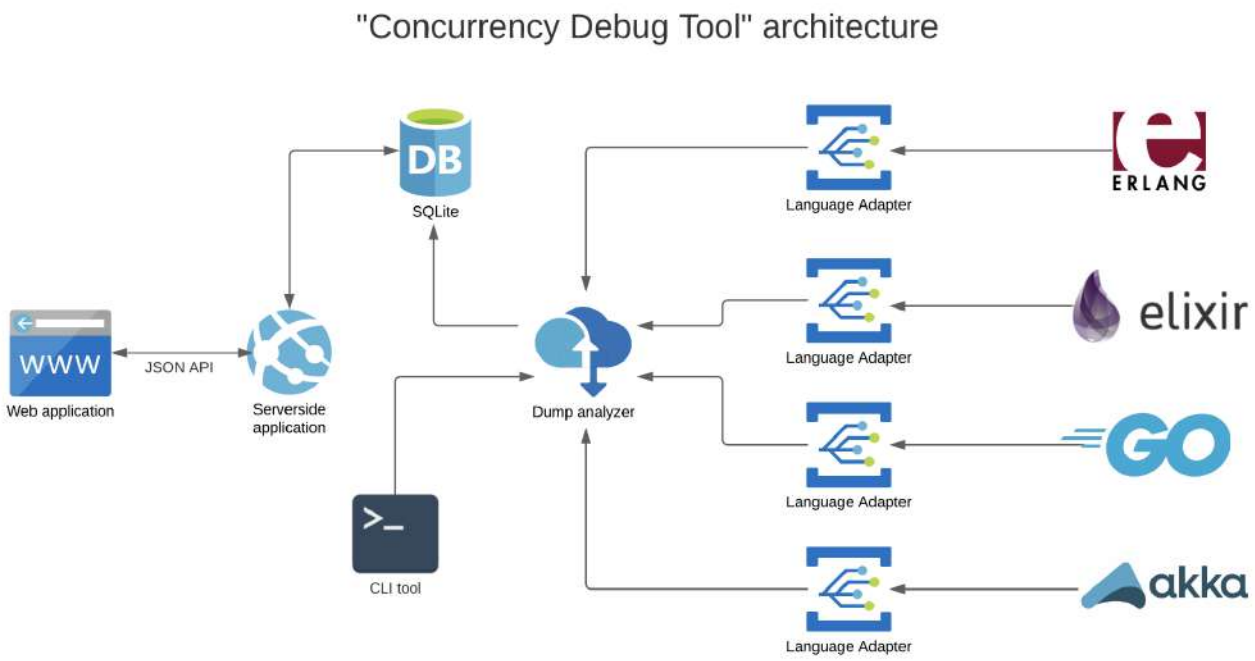
Зробимо коротке порівняння цих баз даних і оберемо ту, яка підходить нам краще за все.

Назва	**SQLite**	**Apache Derby**	**LevelDB**
Тип	реляційна	реляційна	ключ-значення
Транзакції	ACID	ACID	–
DB-Engines Ranking	126.69	4.62	2.36
Зовнішній ключ	+	+	–
Залежності	–	Java VM	–

Насправді, навантаження на нашу базу даних буде незначне, тому в принципі будь-який варіант із наведених вище нам підійде. Але, якщо спробувати все ж обрати щось конкретне, то нам потрібна база даних з мінімальними залежностями, і при цьому відповідати нашим чітким структурованим даним. Тому, для нашої задачі я обрав – SQLite. Вона є найпопулярнішою серед баз даних, розглянутих вище, і також має гарну підтримку різних мов програмування. В SQLite, як протокол обміну використовуються виклики функцій (API) бібліотеки SQLite. Такий підхід зменшує накладні витрати, час відгуку і спрощує програму. SQLite зберігає всю базу даних (включаючи визначення, таблиці, індекси і дані) в єдиному стандартному файлі на тому комп'ютері, на якому виконується застосунок. Простота реалізації досягається за рахунок того, що перед початком виконання транзакції весь файл, що зберігає базу даних, блокується; ACID-функції досягаються зокрема за рахунок створення файлу-журналу.

2.5. Загальна архітектура “Concurrency Debug Tool” (CDT)

Тепер давайте розглянемо архітектуру нашого додатку, з яких компонент воно складається і які інтерфейси використовує.



Отже, основі компоненти нашої системи:

- Аналізатор зліпку
- Серверна частина
- Фронтенд web-додатку
- CLI (Common Language Infrastructure)

Коротко опишемо призначення кожної із частин, а також як ці компоненти взаємодіють між собою.

2.5.1 Аналізатор зліпку

Аналізатор зліпку (англ. dump analyzer) запускає нашу програму, використовуючи потрібний нам адаптер для конкретної мови програмування, потім аналізує зліпки пам'яті та інші дані, і вихідний підготовлений результат зберігає в базу даних.

2.5.2 Серверна частина

Серверна частина нашої системи написана на мові програмування Python і являє собою простий web-сервер, який реалізує 3 кінцеві точки (англ. endpoints) API. Давайте введемо таке поняття, як збірка (англ. build). Збірка – це представлення програми в певний момент часу у вигляді метаданих, які зберігаються в базі даних в таблицю *“build”*. Наш web-сервер має наступні API-методи у вигляді JSON об'єктів:

1. *GET /api/projects* – віддає список проєктів, які існують в базі даних.
2. *GET /api/history?project={project_id}* – віддає список збірок, які існують в базі даних для поточного проєкту.
3. *GET /api/builds?hash={build_hash}* – віддає набір метаданих по збіркам, включаючи інформацію з розподіленої системи керування версіями git, такі як – номер останнього git-коміту, автор останнього git-коміту, дата останнього git-коміту, тощо.

2.5.3 Фронтенд web-додатку

Для створення 3D-анімації фронтенд-частини web-додатку було обрано WebGL. WebGL — це стандарт на базі OpenGL ES 2.0, що дозволяє розробникам веб-контенту інтегрувати в веб-оглядачі повноцінну 3D-графіку, не використовуючи сторонні плагіни. Ця технологія дає можливість

упроваджувати апаратно-прискорену 3D графіку у веб-сторінки на будь-якій платформі, що підтримує OpenGL або OpenGL ES. Зручною Javascript-бібліотекою для роботи з WebGL є *three.js* [22]. Вона надає зручний API для рендеру простих 3D-примітивів, таких як лінія, полігон, тощо.

Основною ідеєю для реалізації візуальної частини проекту є те, що ми будемо зображати роботу нашої Golang-програми в 3D-просторі, де одна із осей буде відповідати за час (нехай це буде вісь координат Z). Наші легковісні співпрограми (горутини, актори) будуть зображуватися, яка лінії, які зростають по вісі Z. Тобто з плином часу ми будемо бачити, як породжуються горутини, хто їх породжує, а також подію, коли горутина закінчує своє існування і знищується. Далі іншим кольором ми можемо візуально зобразити, як горутини обмінюються між собою повідомленнями. Це буде вектор, який буде виходити з точки, де горутина щось послала в канал, а кінець вектора – це точка, де з канала якась інша горутина забрала повідомлення. Так, технічно в моделі CSP в нас повідомлення літають не поміж горутинами, а через такий примітив, як канал. Тобто чисто в теорії, якісь дані можуть бути відправленні в канал і їх ніяка інша горутина не забере, бо в CSP немає такого терміну, як адресат повідомлення. Але для наглядності та щоб візуально було зрозуміліше, що відбувається під час виконання даного програмного застосунку, будемо робити спрощення і такі 2 події, як “відправлення даних в канал” і “приймання даних з каналу”, будемо відображати на нашому графіку як одну подію – “відправлення повідомлення”.

Всі елементи керування нашого web-додатку будуть знаходитися на одній сторінці, тому нашу фронтенду-частину будемо реалізовувати у вигляді SPA (Single-page application). Весь JS-код відповідатиме за поточний стан, всі проміжні дані будуть зберігатися в web-браузері у LocalStorage, а взаємодія з сервером буде побудована на JSON API.

2.5.4 CLI (Common Language Infrastructure)

Далі нам потрібно створити утіліту, яка допоможе кінцевому користувачу зручно застосовувати “Concurrency Debug Tool” в реальній роботі. Для цього

йому потрібно мати зручний інструмент, де всього лиш за декілька простих дій він зможе з легкістю створити і зберегти збірку, та зробити інші, потрібні йому, операції. Отже для цього створемо консольну програму CLI (Common Language Infrastructure) і назвемо її *cdt* (похідна назва від “Concurrency Debug Tool”). Для її розробки візьмемо мову програмування Python. Тепер давайте опишемо можливості нашої CLI-утіліти *cdt* і розглянемо покроково, як можна нею користуватися.

Отже, припустимо, що ми працюємо над розробкою свого програмного застосунку і нам потрібно зафіксувати поточний стан у наш інструмент для візуалізації, щоб потім більш детально розібрати, які процеси і події відбуваються в ньому. Для цього скористаємося наступною простою командою:

Вона фоново зробить всю роботу за нас, а саме – збере всю необхідну дебаг- та трейсинг- інформацію, розпарсить її та збереже потрібні нам дані у базу даних. Також ми може передати додаткову інформацію, наприклад назву збірки.

Якщо нам потрібно подивитися список наших збірок візуалізації для поточного проєкту (ідентифікатор поточного проєкту береться в git репозиторія і *cdt* на нього), то існує наступна команда:

Коли раптом ми випадково закомітили, а отже і зберегли в базу даних, збірку, яка нас не цікавить, то є можливість видали її по номеру білда. Номер білда генерується автоматично за UUID версії 4 (universally unique identifier), наприклад “febeb840-f4a8-4219-b958-a360b940df9d”. Щоб видалити збірку, потрібно скористатися командою:

РОЗДІЛ 3. Використання програмного застосунку “Concurrency Debug Tool” (CDT) для збору та візуалізації метаданих

3.1. Приклади застосування візуалізації для класичних шаблонів конкурентного програмування

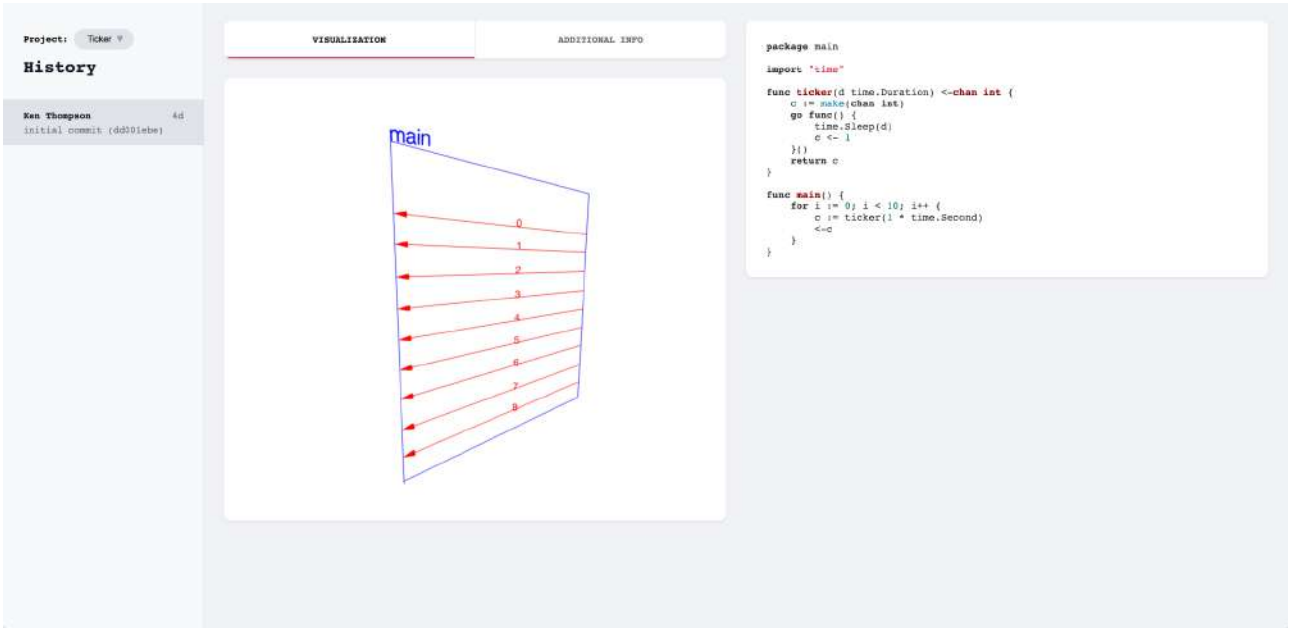
Давайте розглянемо декілька класичних шаблонів, які використовуються як приклади різноманітних підходів до проектування конкурентних програм.

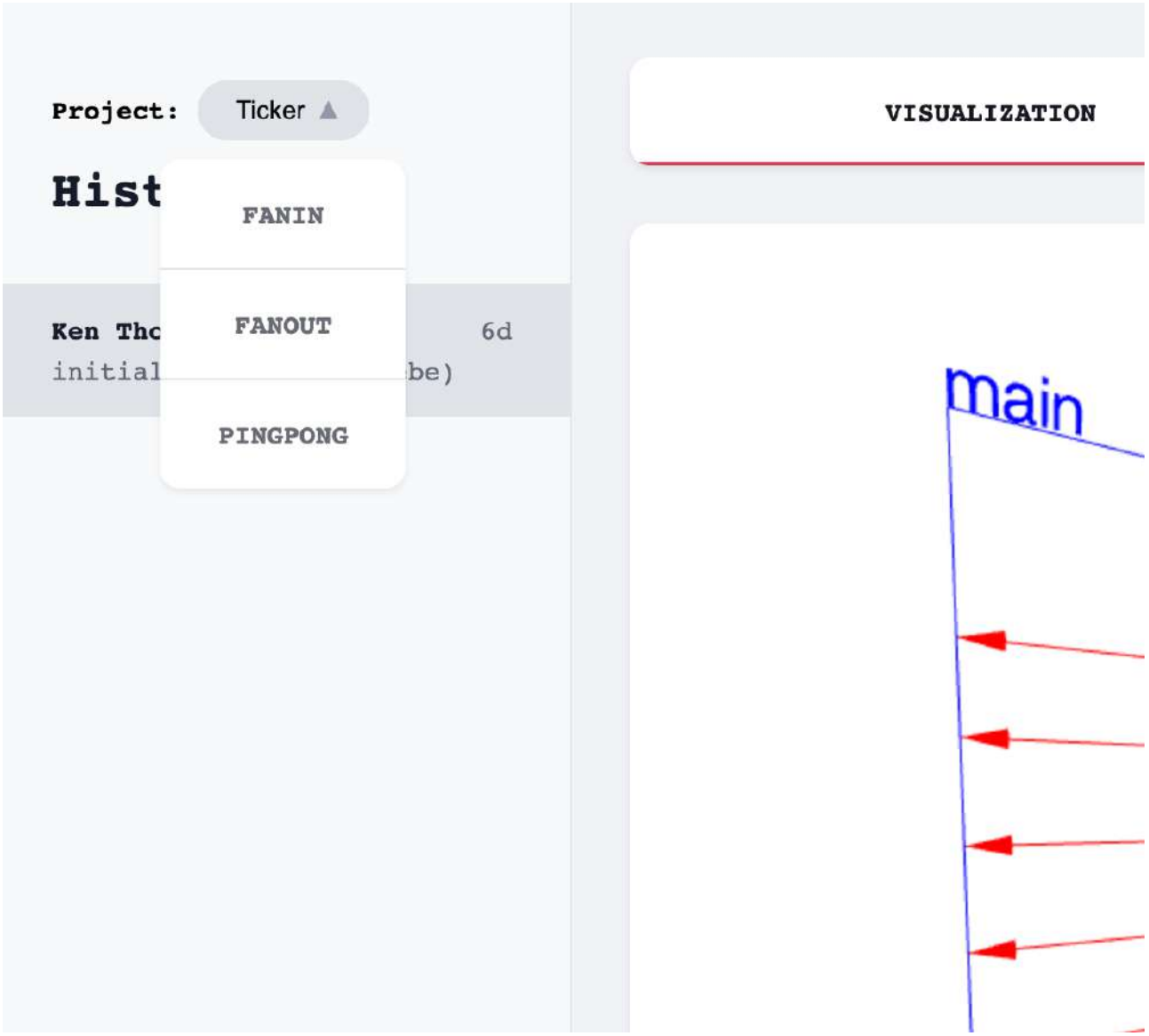
3.1.1 Ticker

Для початку, щоб зрозуміти саму концепцію і основні елементи системи, розглянемо простий шаблон – Ticker. Наша програма буде складатися з однієї головної горутини *main* і 10 горутин-тікерів, які ми породили в нашій основній горутині. Задача горутин-тікерів через 1 секунду надіслати в канал одне повідомлення, тобто відбувається такий собі генератор імпульсів кожну 1 секунду. Давай поглянемо на код:

```
package main

import "time"

func ticker(d time.Duration) <-chan int {
	c := make(chan int)
	go func() {
		time.Sleep(d)
		c <- 1
	}()
	return c
}

func main() {
	for i := 0; i < 10; i++ {
		c := ticker(1 * time.Second)
		<-c
	}
}
```

Тепер давайте скористаємося командою *cdt commit* і після цього запустимо наш веб-додаток, щоб поглянути, що візуально відбувається у програмі, які процеси і які активність подій знаходиться у системі.





Давайте на цьому першому прикладі більш детально розберемо наш інтерфейс. Зліва ви можете бачити поточний проект, який можна переключити в випадаючому меню. Далі йде історія збірок, які генерував користувач. В даному випадку у нас в історії всього декілька елементів, але в реальних проектах їх может бути в рази більше. По центру нашого інтерфейсу ми бачимо зону, де будується візуальне відображення виконання програмного забезпечення. Сам 3D-графік за допомогою комп'ютерної миші можна крутити ліворуч, праворуч, вверх та вниз, робити наближення та віддалення. За допомогою стрілок на клавіатурі можна рухати саме полотно. Якщо натиснути клавішу *“p”*, то тоді увімкнеться автоматична ротація графіка по колу, так званий режим ознайомлення. Справа від 3D-графіка знаходиться окрема область, де показується вихідний код програмного забезпечення, на основі якого була

сформована збірка. Вихідний код система додатково зберігається в нашій базі даних, тому що збірка може створюватись незалежно від кількості git-комітів, а отже треба десь тримати історію змін програмного вихідного коду. Зверху присутні 2 таби, якими зручно переключатись між візуалізацією і додаткової debug-інформацією, такою як кількість навантажених CPU ядер, кількість горутин, тощо.

3.1.2 Fan-in

Перейдемо до наступного прикладу. Один з найвідоміших шаблонів в конкурентному програмуванні – це так званий шаблон fan-in. Він є протилежністю паттерну fan-out, який ми розглянемо далі. Якщо коротко, то fan-in – це функція, що читає з декількох джерел і мультиплексує все в один канал. Наприклад, джерелом повідомлень можуть бути клієнти, а місце призначення – сервер.

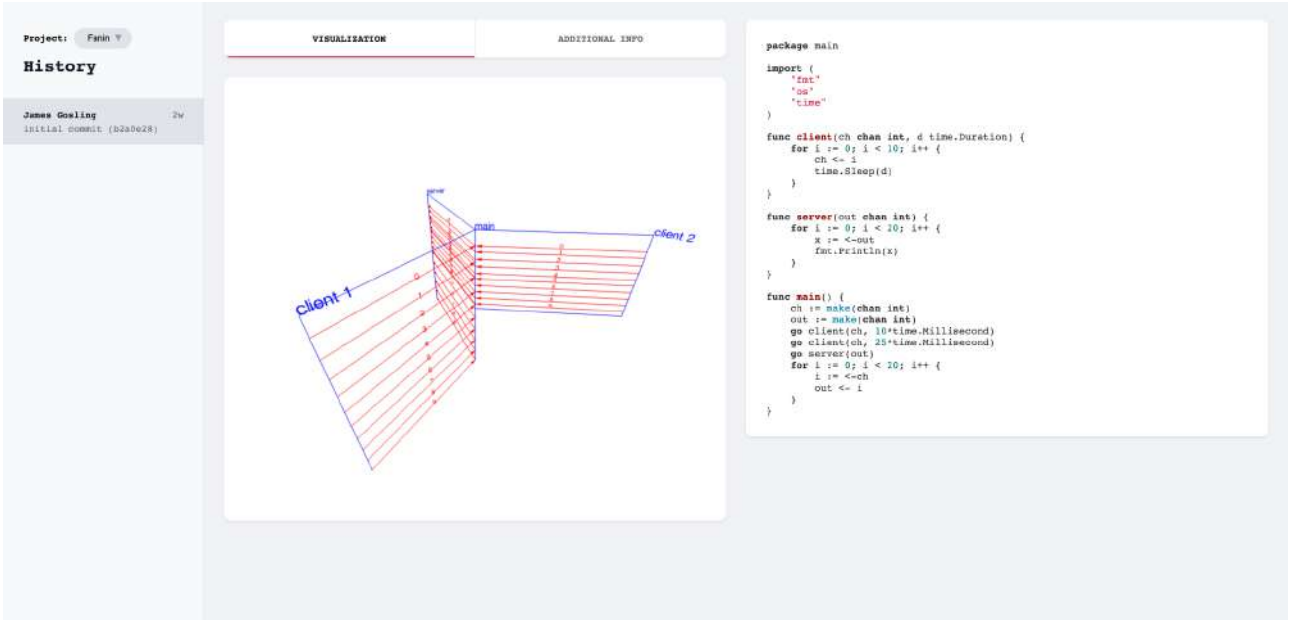
Наш базовий код буде виглядати приблизно так:

```
package main

import (
	"fmt"
	"os"
	"time"
)

func client(ch chan int, d time.Duration) {
	for i := 0; i < 10; i++ {
		ch <- i
		time.Sleep(d)
	}
}

func server(out chan int) {
	for i := 0; i < 20; i++ {
		x := <-out
		fmt.Println(x)
	}
}

func main() {
```

```
        ch := make(chan int)
        out := make(chan int)
        go client(ch, 10*time.Millisecond)
        go client(ch, 25*time.Millisecond)
        go server(out)
        for i := 0; i < 20; i++ {
                i := <-ch
                out <- i
        }
}
```

Розберемо детальніше, що відбувається в цій програмі. У нас є 2 клієнта, які являються генераторами інформації і продукують дані в канал. Третя горутина – сервер, по суті являється приймачем який отримує всі дані від клієнта через головну горутину *main*. Тобто, по суті, функція *main* – є для клієнтів мультиплексором, який збирає дані з усіх клієнтів і відправляє їх до сервера.

Тепер запустимо команду *cdt commit* і запустимо наш веб-додаток.



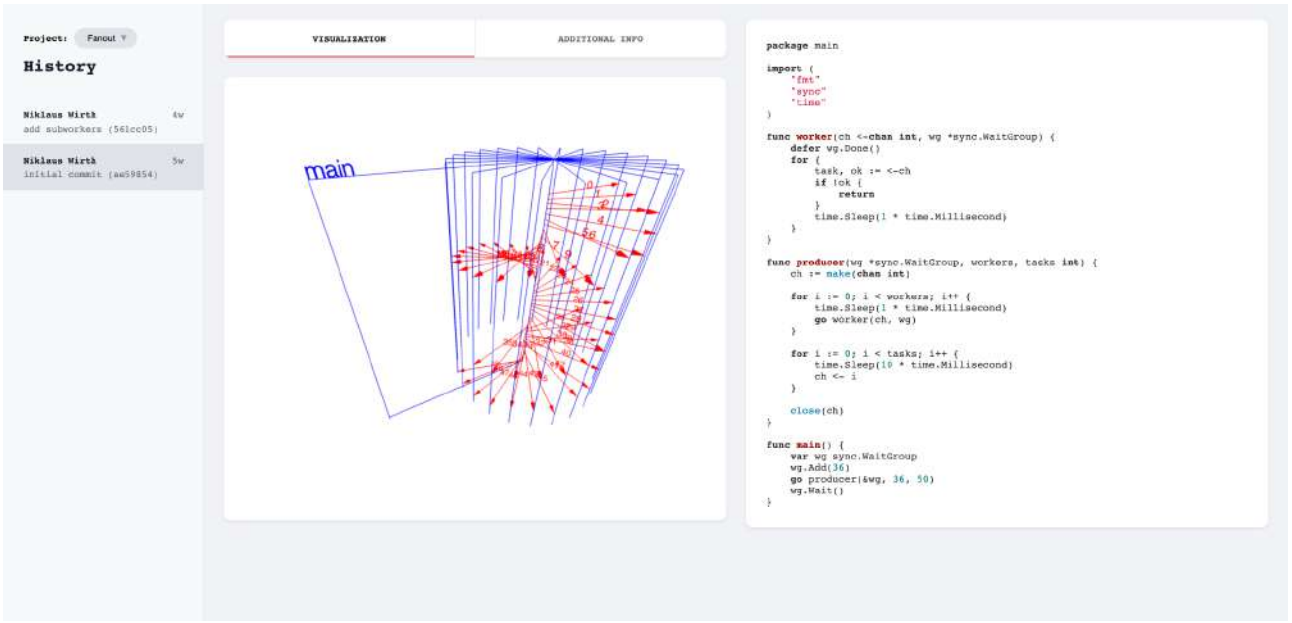
3.1.3 Fan-out

В протилежність до шалону fan-in розглянемо ще один з частих підходів у конкурентному програмуванні під назвою – fan-out. В ньому декілька горутин читають з одного й того ж каналу, забираючи на обробку якісь дані і ефективно розподіляючи роботу між ядрами CPU. Часто такий підхід називають системою розподілених задач, або робітників (англ. workers), де кожен робітник бере на

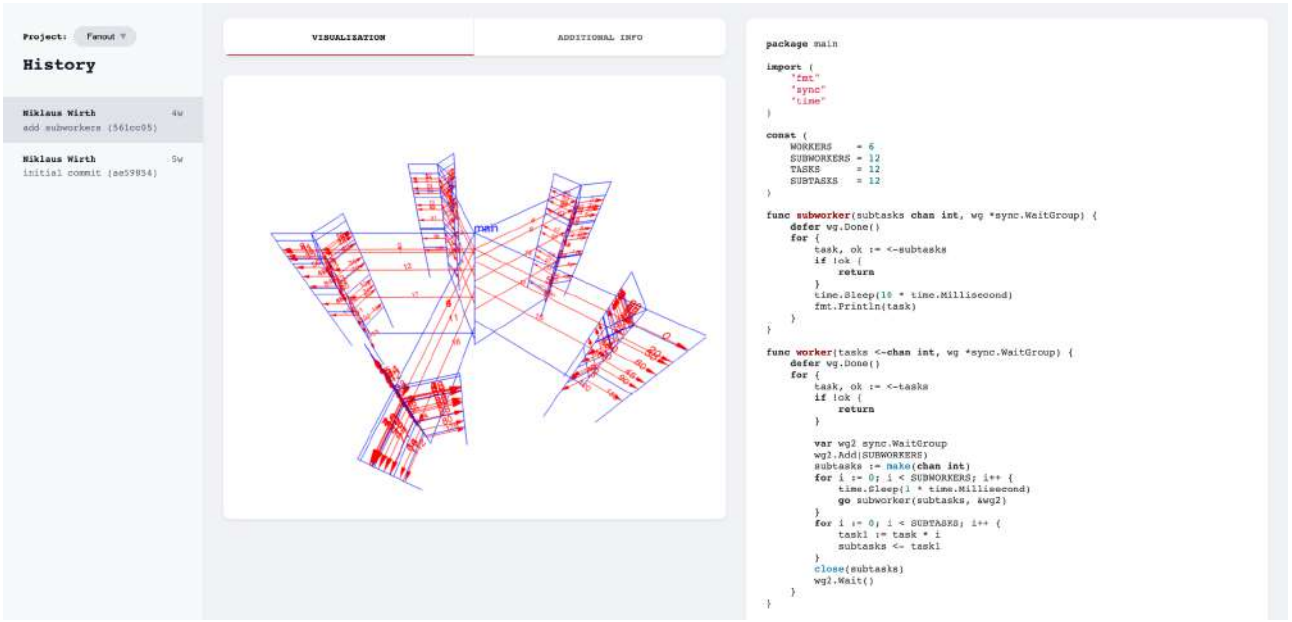
себе певний шматок задач, робить його, і знову забирає наступний пакет задач загальної черги. У Golang реалізовувати цей патерн дуже просто – потрібно запустити декілька горутин, передавши їм канал через параметр, і надсилати в канал дані, а мультиплексування і розподіл буде відбуватися автоматично завдяки планувальнику Golang.

```
func worker(ch <-chan int, wg *sync.WaitGroup) {
	defer wg.Done()
	for {
		task, ok := <-ch
		if !ok {
			return
		}
		time.Sleep(1 * time.Millisecond)
	}
}

func producer(wg *sync.WaitGroup, workers, tasks int) {
	ch := make(chan int)

	for i := 0; i < workers; i++ {
		time.Sleep(1 * time.Millisecond)
		go worker(ch, wg)
	}

	for i := 0; i < tasks; i++ {
		time.Sleep(10 * time.Millisecond)
		ch <- i
	}

	close(ch)
}

func main() {
	var wg sync.WaitGroup
	wg.Add(36)
	go producer(&wg, 36, 50)
	wg.Wait()
}
```

Як бачимо, в даному прикладі ми створимо 36 робітників і будемо продукувати 50 завдань (повідомлень) в загальний канал *ch*, а далі наші робітники конкурентно будуть брати ці повідомлення і обробляти їх. Такий підхід дуже часто використовується для обробки розподілених фонових задач. Візуально це буде виглядати наступним чином:



Якщо ж ми спробуємо розшири нашу програму і зробити так, щоб наші робіники самі створювали свої власних робітників, і передавали їм завдання. В реальних проектах таке буде навряд чи потрібно, але ми це зробимо, щоб візуально продемонструвати як зміниться наше зображення.



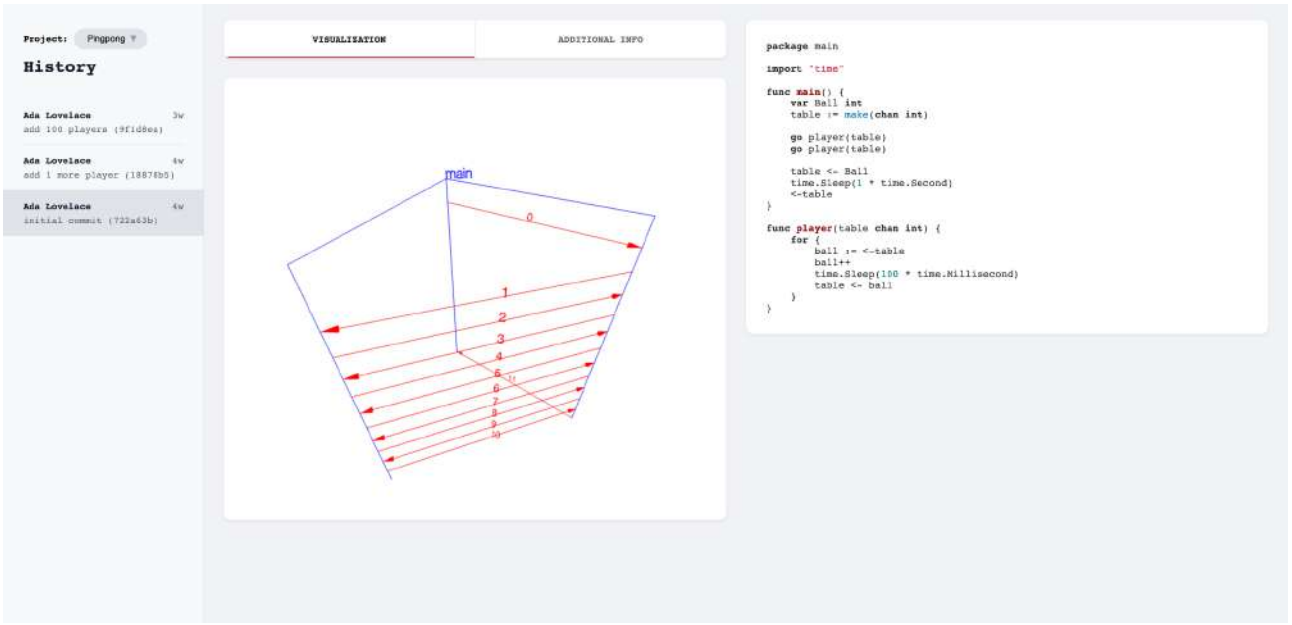
3.2. Використання CDT під час еволюції програмного коду проєкта

Тепер давайте більш детально розглянемо, як наша візуалізація буде змінюватися під час того, як розробник працює над проєктом і ітеративно змінює деяку частини вихідного коду. Для цього візьмемо приклад програми Ping-Pong [26].

```
package main

import "time"

func main() {
     var Ball int
     table := make(chan int)

     go player(table)
     go player(table)

     table <- Ball
     time.Sleep(1 * time.Second)
     <-table
}

func player(table chan int) {
     for {
          ball := <-table
          ball++
          time.Sleep(100 * time.Millisecond)
          table <- ball
     }
}
```

У цій програмі реалізована наступа механіка реальної гри настільного тенісу. В нас є стіл (канал *table)*, є м'яч *Ball* (змінна типу *integer*), який зберігає в собі кількість ударів по ньому, і є гравці (горутини), які “забирають м'яч зі столу” (читають з каналу *table*), “б'ють по ньому” (збільшують значення змінної *Ball*) і “кидають м'яч назад на стіл” (пишуть у канал *table*). Приклад здається досить простим, але на його основі можна зручно розглянути, як змінюється візуалізація і те, що відбувається всередині програми, змінивши всього декілька

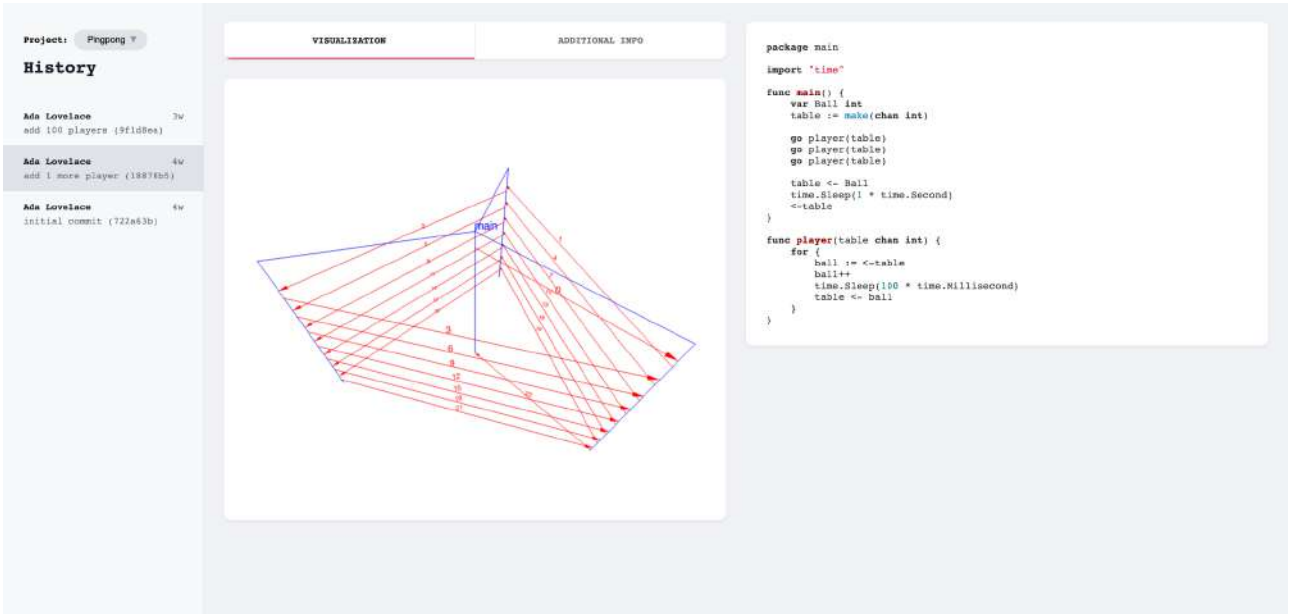
строк коду. Отже, давайте візуалізуємо процес за допомогою утіліти *cdt* і подивимось на те, що відбувається під виконання нашого вихідного коду:



Після цього давайте трішки змінимо наш вихідний код і додамо ще одного гравця до гри. Тепер всі вони втрьох буду грати на одному столі з одним м’ячем. Відповідно наш код зміниться наступним чином:

```
func main() {
      ...

      go player(table)
      go player(table)
      go player(table)

      ...
}
```

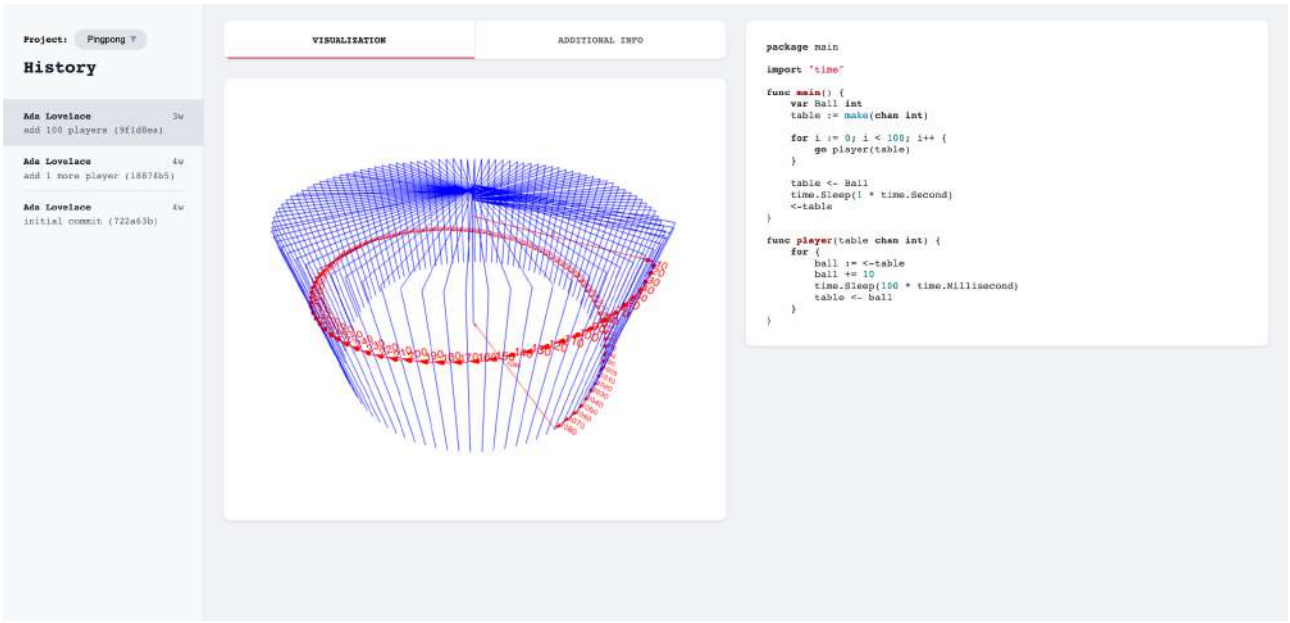
Зробимо збірку за допомогою *cdt commit*, наші дані збережуться в базу данних і після цього перезавантажимо наш веб-додаток. Тоді ми будемо бачити наступне:



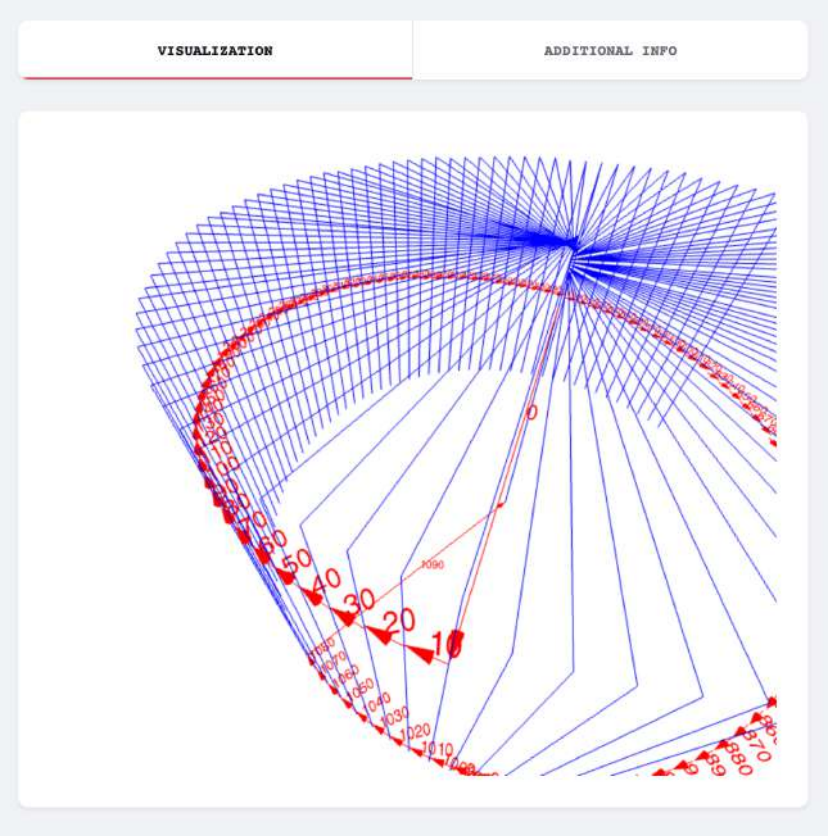
Як бачимо, тепер ми можемо працювати з історичними даними збірок і переключатися між старою версією і новою. Справа відображається вихідний код, який відповідає візуалізації по центру нашого інтерфейсу. Щож, 3 гравці – це було показово, але замало для демонстрації. Давайте створимо 100 гравців в настільний теніс, які будуть грати одним м’ячем, але при цьому “удар” кожного гравця по м’ячу буде збільшувати його значення на 10, а не на 1, як було раніше. Зробимо це для наглядності і демонстрації роботи web-додатку:

```
func main() {
      ...

      for i := 0; i < 100; i++ {
            go player(table)
      }

      ...
}

func player(table chan int) {
      ...

      ball += 10

      ...
}
```

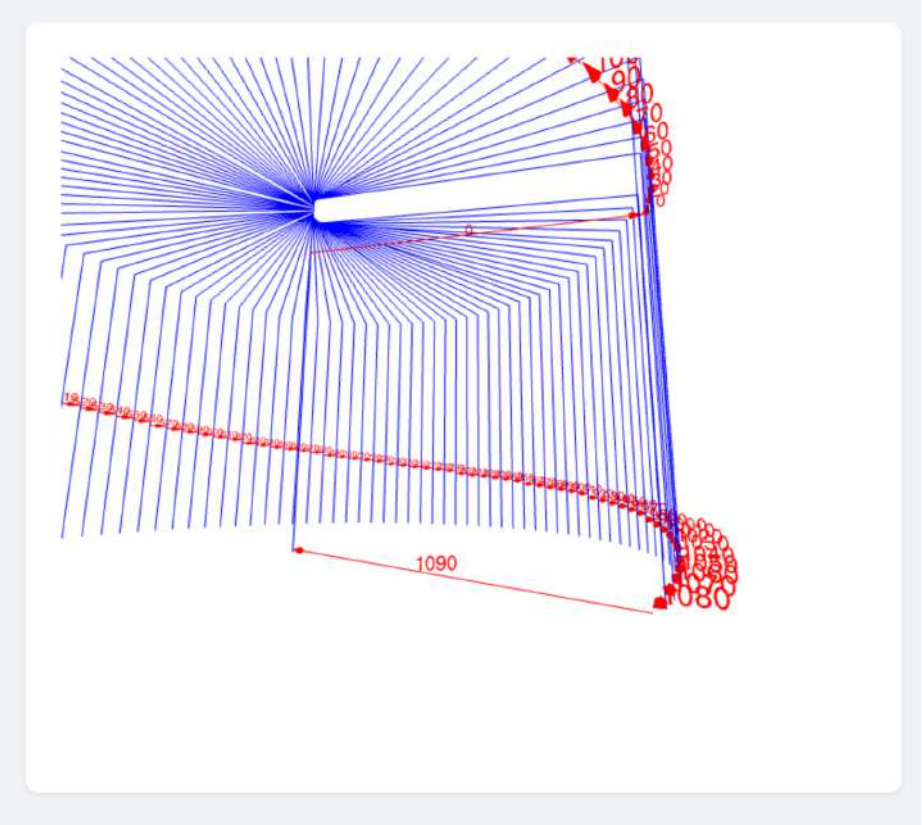
Запустимо знову команду *cdt commit* і тепер можемо побачити нову створену версію збірки.



Ми вже зазначали вище, но графік будується в середовищі WebGL і цей 3-D графік можна зручно збільшувати, зменшувати і рухати по різним осям координат. Наприклад, можна роздивитись деталі останньої візуалізації з іншого ракурсу і побачити, яке число у змінній *Ball* переходить від горутини до горутини і кожного разу збільшує своє значення на 10 одиниць:



Після того, як пройшла рівно 1 секунда, управління на себе тимчасово бере головна горутина *main*, отримує дані з каналу (число 1090) і завершує свою роботу:



Як бачимо, візуально буває набагато легше зрозуміти і відчути, які процеси відбувається у часом нетривіальних конкурентних програмах. Такий підхід можна використовувати при розробці складних систем, де потрібно провести налагодження коду чи швидко провести аналіз подій і процесів, що відбуваються у програмному забезпеченні.

Висновки по роботі та рекомендації для подальших досліджень

В першому розділі роботи було досліджено поняття конкурентності та паралельності при розробці програмного забезпечення. Також було розглянуто, що багатозадачність є дуже важливим аспектом в сьогоденні. Було розглянуто види багатозадачності і як вони використовуються в сучасних мовах програмування, та які види комунікації і синхронізації між співропрограмами існують на даний момент.

Другий розділ присвячено аналізу підходів до розробки архітектури системи по візуалізації конкурентних систем. Було обрано програмні компоненти, розроблено та представлено архітектуру інструмента по візуалізації “Concurrency Debug Tool” (CDT). Також було обрано ряд технологій, які краще за все підходять для розробки подібних систем.

В третьому розділі розглянуто практичну реалізацію програмного комплексу, показано приклади різних шаблонів конкурентного програмування. Було представлено можливості системи “Concurrency Debug Tool” (CDT), а також описано деталі та обмеження, які на даний час існують в ній. Головну увагу приділено простоті використання даного інструмента, доцільності використання такого роду візуалізацій та як він може допомогти під час розробки зі зміною та еволюцією програмного коду.

Отримана в результаті розробки архітектура може бути в подальшому розширена новими компонентами за необхідності. Зокрема, може бути додана підтримка нових мов програмування. Така можливість досягається за рахунок використання мовного адаптеру (Language adapter), який з одного боку забирає сирі дані, а з іншого зберігає ці дані в уніфікованому вигляді, який вже в подальшому “розуміє” наш візуалізатор.

В цілому, розроблений на базі архітектури програмний продукт підтверджує, що розглянута проблематика є актуальної, а візуалізація є ефективним підходом для розуміння процесів і подій, що відбуваються під час виконання програмних застосунків. Розроблений інструмент є прототипом для подальшого розширення та в подальшому може використовуватися, як для

різних мов програмування, так і для різних IDE та редакторів коду, як плагін по візуалізації та налагодженню програм. В даній роботі було розглянута візуалізації простих програм, і не досліджено аналіз та візуалізацію більш комплексних програмних продуктів з великою кодовую базою. На сьогоднішній момент “Concurrency Debug Tool” (CDT) являє собою інструмент для демонстрації процесів та подій, що відбуваються під час виконання програми і може використовуватися для прототипних простих програм, щоб більш детально зрозуміти певні незрозумілі моменти конкурентності, або проаналізувати побічні ефекти (англ. side effects), які можна спостерігати під час розробки.

Так як сфера інформаційних технологій є дуже динамічною, і ми можемо бачити, як постійно з'являються нові підходи та напрямки, тому можливості для розвитку та розширення функціоналу існують для будь-якого програмного рішення. Виходячи з цього, частиною майбутньою роботи може бути розширення та вдосконалення продукту, додавання нових можливостей, нового функціоналу, підтримка та інтеграції інших мов програмування.

Список літератури

1. 42 Years of Microprocessor Trend Data. [Електронний ресурс]. Режим доступу: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/
2. Wikipedia. [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Рівночасні_обчислення
3. Concurrency is not Parallelism by Rob Pike. [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=oV9rvDllKEg
4. A Periodic Table of Visualization Methods. [Електронний ресурс]. Режим доступу: https://www.visual-literacy.org/periodic_table/periodic_table.html
5. Towards A Periodic Table of Visualization Methods for Management
6. 13 Scientific Reasons Why Your Brain Craves Infographics. [Електронний ресурс]. Режим доступу: https://neomam.com/interactive/13reasons/
7. Visual Studio Visualizer. [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer
8. Офіційний сайт мови програмування Golang. [Електронний ресурс]. Режим доступу: https://golang.org/
9. Документація мови програмування Golang. [Електронний ресурс]. Режим доступу: https://golang.org/doc/
10. Основні концепції і приклади мови програмування Golang. [Електронний ресурс]. Режим доступу: https://tour.golang.org/
11. Mark Richards. Software Architecture Patterns. *O'Reilly Media, Inc.* 2015.
12. Система керування версіями Git. [Електронний ресурс]. Режим доступу: https://git-scm.com/
13. goTrace. [Електронний ресурс]. Режим доступу: https://github.com/staheri/goTrace
14. gotrace [Електронний ресурс]. Режим доступу: https://github.com/divan/gotrace/tree/master/trace
15. pprof. [Електронний ресурс]. Режим доступу: https://golang.org/pkg/runtime/pprof/

16. Go Execution Tracer. [Електронний ресурс]. Режим доступу: https://docs.google.com/document/u/1/d/1FP5apqzBgr7ahCCgFO-yoVhk4YZrNIDNf9RybngBc14/pub
17. Command trace. [Електронний ресурс]. Режим доступу: https://golang.org/cmd/trace/
18. High Performance Go Workshop. [Електронний ресурс]. Режим доступу: https://dave.cheney.net/high-performance-go-workshop/dotgo-paris.html
19. GopherCon 2017: Rhys Hiltner - An Introduction to "go tool trace". [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=V74JnrGTwKA
20. SQLite. [Електронний ресурс]. Режим доступу: https://sqlite.org/index.html
21. WebGL. [Електронний ресурс]. Режим доступу: https://www.khronos.org/webgl/
22. Javascript-бібліотека three.js. [Електронний ресурс]. Режим доступу: https://threejs.org/
23. Paul Butcher. Seven Concurrency Models in Seven Weeks: When Threads Unravel (The Pragmatic Programmers) 1st Edition. *O'Reilly Media, Inc.* 2014.
24. Katherine Cox-Buday. Concurrency in Go: Tools and Techniques for Developers 1st Edition. *O'Reilly Media, Inc.* 2017.
25. Google I/O 2012 - Go Concurrency Patterns. [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=f6kdp27TYZs
26. Google I/O 2013 - Advanced Go Concurrency Patterns. [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=QD Dww PbDtw
27. Mark Richards, Neal Ford. Fundamentals of Software Architecture. *O'Reilly Media, Inc.* 2020.
28. Alok Chaudhari, Ameya Kulkarni. Hands-On Concurrency with GO. *Packt Publishing.* 2019.
29. Concurrency vs Parallelism. [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=Y1pgpn2gOSg
30. Cooperative vs. Preemptive: a quest to maximize concurrency power. [Електронний ресурс]. Режим доступу:

https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe

31. Мова програмування Python. [Електронний ресурс]. Режим доступу: https://www.python.org/
32. Atul S. Khot. Concurrent Patterns and Best Practices: Build scalable apps with patterns in multithreading, synchronization, and functional programming. *Packt Publishing*. 2018.

Додаток А. Програмний код компонента системи dump analyzer

Додаток Б. Програмний код фронтенд web-компоненти системи

Додаток В. Програмний код серверної web-компоненти системи