ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «**ОПТИМІЗАЦІЯ РОБОТИ АЛГОРИТМУ ПОБУДОВИ МАРШРУТІВ ЗА ДОПОМОГОЮ OPEN MP**»

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми _____
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ _____
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав: здобувач вищої освіти гр. КСДМ-62
Петрик А.В.
(Ім'я, ПРІЗВИЩЕ)

Керівник: доктор філософії, доцент Лемешко А.В.
науковий ступінь, вчене звання
(Ім'я, ПРІЗВИЩЕ)

Рецензент: _____
науковий ступінь, вчене звання
(Ім'я, ПРІЗВИЩЕ)

Київ 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерної інженерії
Ступінь вищої освіти «Магістр»
Спеціальність
Освітньо-професійна програма 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерної інженерії
Н.О. Лащевська Ім'я, ПРІЗВИЩЕ
"____" _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Петрику Андрію Васильовичу
(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи Оптимізація роботи алгоритму побудови маршрутів за допомогою Open MP
керівник кваліфікаційної роботи доктор філософії, доцент кафедри КІ Лемешко А.В.
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «____» _____ 2023 р. № ____
2. Строк подання кваліфікаційної роботи «____» _____ 2023 р.
3. Вихідні дані до кваліфікаційної роботи:
3.1 Технічна документація стосовно розробки алгоритму побудови маршрутів;
3.2 Інтернет-ресурси стосовно програмно-конфігурованих мереж;
3.3 Науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити:
4.1 Аналіз проблем побудови маршрутів у програмно-конфігурованих мережах;
4.2 Проектування та реалізація;
4.3 Тестування та апробація.

5. Перелік графічного матеріалу: *презентація*

6. Дата видачі завдання «____» _____ 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури		
2	Аналіз проблем побудови маршрутів у програмно-конфігурованих мережах		
3	Проектування та реалізація		
4	Тестування та апробація		
5	Вступ, висновки, реферат		
6	Розробка обов'язкових демонстраційних матеріалів		
7	Попередній захист роботи		

Здобувач вищої освіти _____ Петрик А.В.
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Лемешко А.В.
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 70 стор., 2 табл., 9 рис., 24 джерел.

Мета роботи - підвищення ефективності та продуктивності функціонування програмно-конфігурованих мереж.

Об'єкт дослідження – алгоритм побудови маршрутів в програмно-конфігурованих мережах.

Предмет дослідження – програмно-конфігуровані мережі.

Короткий зміст роботи:

В дипломній роботі наведено аналіз програмно-конфігурованих мереж, їх слабкі місця та недоліки побудови маршрутів усередині складних сегментів мережі. Також буде представлено обґрунтування вибору середовища розробки, емулятора програмно-конфігурованої мережі для налагодження та мережної операційної системи. В роботі було зроблено проектування та реалізація алгоритму та наведено дані про результати проведених випробувань щодо продуктивності розробленого алгоритму та порівняння з його аналогами.

КОМП'ЮТЕРНІ МЕРЕЖІ, ПРОГРАМНО-КОНФІГУРОВАНІ МЕРЕЖІ, ПРОТОКОЛ OPENFLOW, OPEN MP, ОПЕРАЦІЙНІ СИСТЕМИ, АЛГОРИТМИ ПОБУДОВИ МАРШРУТІВ, АЛГОРИТМ ОПТИМІЗАЦІЇ ТРАФІКА, ЕМУЛЯЦІЯ СЕГМЕНТІВ МЕРЕЖІ.

ABSTRACT

Text part of the master's qualification work:
70 pages, 9 pictures, 2 table, 24 sources.

The purpose of the work is increasing the efficiency and productivity of software-configured networks.

Object of research is an algorithm for building routes in software-configured networks.

Subject of research is software-configured networks.

Summary of the work:

The thesis provides an analysis of software-configured networks, their weak points and the disadvantages of constructing routes within complex network segments. The rationale for choosing a development environment, a software-configured network emulator for debugging, and a network operating system will also be presented. The paper designed and implemented the algorithm and provided data on the results of tests on the performance of the developed algorithm and comparison with its counterparts.

KEYWORDS:

COMPUTER NETWORKS, SOFTWARE-CONFIGURED NETWORKS, OPENFLOW PROTOCOL, OPEN MP, OPERATING SYSTEMS, ROUTE CONSTRUCTION ALGORITHMS, TRAFFIC OPTIMIZATION ALGORITHM, EMULATION OF NETWORK SEGMENTS

ЗМІСТ

Стор.

ВСТУП

У XXI столітті постійне зростання підключених пристроїв до мережі «Інтернет» спричинило експоненційне зростання загальносвітового трафіку. Цей різкий стрибок стимулював розвиток нових швидших та інноваційних комп'ютерних мереж під назвою «програмно-конфігуровані мережі». Нове покоління комунікаційних мереж мало важливу перевагу у вигляді збільшеної пропускної спроможності кожного передавального пристрою, проте головним мінусом даних мереж став час створення нового з'єднання, воно збільшилося більш ніж у два рази. [1]

Проблеми викликані складністю побудови маршрутів на великих сегментах підконтрольних мереж, у яких кількість підключених передавальних пристроїв більше 300 штук.

Теорія побудови найкоротших шляхів на графах має великий вплив на сучасний світ. Завдяки своєму широкому спектру наданих можливостей, вона останнім часом інтенсивно розвивається: разом з поліпшенням вже розроблених методів винаходять принципово нові.

Пошук шляхів є важливим завданням, яке використовують у різних областях та сферах. Правильно розрахований маршрут, може заощадити кошти на одиничну або групову доставку, великий обсяг даних буде швидше доставлений користувачеві, штучний інтелект у комп'ютерних іграх покаже цікавішу поведінку і маршрут для автомобіля буде побудований ефективно з точки зору витраченого часу на шлях.

Починаючи з 1969 року, комп'ютерні мережі почали розвиватися. Спочатку в першій мережі під назву ARPANET було з'єднано 4 суперкомп'ютери того часу. Завдяки цій мережі було випробувано маршрутизацію пакетів за допомогою протоколу IP, який використовується і досі. Починаючи з 1990 року з'явилося всесвітнє павутиння, яке з'єднало між собою найвіддаленіші куточки світу. [2]

Починаючи з 1990 року складність комп'ютерних мереж зростає в геометричній прогресії, стаючи все більш об'ємними та заплутаними. У сучасних

комп'ютерних мережах пошук шляху здійснюється на два кроки вперед, тому що повний пошук шляху може зайняти величезну кількість часу, однак такий «швидкий» пошук не дозволяє завжди побудувати найоптимальніший і найшвидший маршрут [17]. Через збільшену складність сучасних мереж їм на зміну розробляють програмно-конфігуровані мережі суть, яких полягає в тому, щоб винести логіку побудови маршрутів з передаючих пристроїв і залишити пристроям тільки саму передачу даних. Для того, щоб це реалізувати, необхідно використовувати швидкий і ефективний алгоритм побудови маршруту, який враховуватиме навантаження на кожен окремий пристрій, щоб досягти максимальної продуктивності мережі.

Актуальність розробки ефективного алгоритму оптимізації трафіку для програмно-конфігурованих мереж обумовлена небажанням частини гравців IT ринку переходити на нове покоління мереж у зв'язку з великими затримками при створенні з'єднань. Тому зараз дуже важливо розробити швидкий та ефективний алгоритм для побудови маршруту, за допомогою якого затримки при створенні нового з'єднання суттєво знизяться.

Використання Open MP для оптимізації роботи алгоритмів побудови маршрутів є перспективним, оскільки цей стандарт дозволяє ефективно використовувати ресурси багатоядерних систем. В результаті, можна досягти значного прискорення обчислень та покращення продуктивності алгоритмів.

Виходячи з вищесказаного, метою даної магістерської роботи є дослідження алгоритмів побудови маршрутів, розробка та оптимізація їх швидкості виконання для програмно-конфігурованих мереж з метою підвищення ефективності та продуктивності.

1 АНАЛІЗ ПРОБЛЕМ ПОБУДОВИ МАРШРУТІВ У ПРОГРАМНО-КОНФІГУРОВАНИХ МЕРЕЖАХ

1.1 Актуальність проблеми побудови маршрутів у програмно-конфігурованих мереж

Проблема затримки перед створенням з'єднання для програмно-конфігурованих мереж є найголовнішим фактором, який зупиняє їхнє поширення по всьому світу. Пов'язано це з тим, що сервер, який називається програмно-конфігурований контролер, здійснює прокладку маршруту в складних сегментах мережі, в яких кількість вузлових пристроїв на даний момент може доходити до 1000 штук. Тривалість даних прокладок набагато вища ніж у класичних мережах, в них середній час прокладки маршруту в складному сегменті мережі займатиме близько 14-20 мікросекунд, а в програмно-конфігурованих 200 і більше мікросекунд. Для вирішення цієї проблеми потрібна розробка нового алгоритму. [3]

1.2 Загальні відомості про програмно-конфігуровані мережі

Програмно-конфігуровані мережі (ПКМ) - це нове покоління комп'ютерних мереж, в яких ключовою відмінністю є винесення логіки маршрутизації за межі пристрою на окремий виділений сервер. Поява цього покоління пов'язана з експоненційним зростанням трафіку та падінням ефективності поточних комп'ютерних мереж.

У ПКМ рівні управління мережею та передачі даних поділяються за допомогою функцій управління винесення (комутаторами, маршрутизаторами тощо) у додатки, що працюють на виділеному сервері, званому контролер. Перші концепції таких мереж були сформульовані фахівцями університетів Стенфорда та Берклі ще у 2006 році, а проведені ними дослідження отримали схвалення не лише у наукових колах, а й у сфері бізнесу. Ці ідеї тепло зустріли такі виробники

мережного обладнання як Cisco, HP та інші. У березні 2011 року було засновано консорціум Open Networking Foundation (ONF). Засновниками цієї спільноти є низка великих та впливових компаній у сфері IT. Цими компаніями були: Verizon, Yahoo, Microsoft, Google, Deutsche Telekom та Facebook. Склад ONF швидко поповнився й іншими не менш відомими компаніями, такими як Marvell, Citrix, IBM, NEC, Brocade, Oracle, HP, Dell, Ericsson та інші. Найперша практична реалізація ПКМ була запропонована компанією Nicira, яка нещодавно стала частиною VMware.

Інтерес IT-компаній до ПКМ викликаний тим, що така технологія дозволяє підвищити ефективність мережного обладнання на 25-30%, знизити на 30% витрати на експлуатацію мереж, дозволяє перетворити управління мережами з творчого проектування на інженерію, підвищити захищеність мережі та дозволити користувачам самостійно за допомогою програм створювати нові сервіси та оперативно завантажувати, та використовувати їх на мережному обладнанні.

У більшості напрацювань і досліджень ключові моменти в ПКМ пов'язані з програмою, що розвивається в США Global Environment for Network Innovations (GENI) дослідження майбутнього Інтернету, що включає в себе близько 40 провідних університетів США. Діяльність об'єднаного центру Стенфорда і Берклі, який проводить різні вивчення, дослідження, експерименти та напрацювання в галузі Internet2; а також із Сьомою рамковою програмою досліджень Європейського Союзу Ofelia та проектом FEDERICA. [38]

Основні концепції ПКМ:

- винесення процесу управління даних з передавального пристрою на виділений сервер, а процеси передачі даних залишити на передавальних пристроях;

- уніфікований та незалежний інтерфейс між рівнем управління та рівнем передачі даних;

- логічно централізоване управління мережею, що здійснюється за допомогою контролера з встановленою мережевою операційною системою та реалізованими поверх мережевими додатками;

- віртуалізація фізичних ресурсів мережі.

Головна проблема поточних комп'ютерних мереж полягає в тому, що приблизно 20-25% роботи передавальний пристрій витрачає на прокладку маршруту сегментом мережі. Концепція програмно-конфігурованих мереж полягає у тому, щоб прибрати логіку побудови маршрутів із пристрою передачі і залишити йому лише передачу даних. Новий тип мереж пропонує розміщувати логіку передачі на спеціальних серверах, званих програмно-конфігурованими контролерами, у яких відбувається обчислення маршруту у межах сегмента мережі. Дані мережі в першу чергу націлені на центри обробки даних, адже саме вони відповідають за великий медіаконтент, що надається на різних ресурсах. [4]

За рахунок централізованості керування сегментом мережі дана мережа пропонує низку серйозних переваг:

-зниження вартості устаткування, рахунок спрощення устаткування;

- більш детальне керування мережею;

- дуже низька ймовірність несанкціонованого доступу до ресурсів мережі;

- можливість розробки або доопрацювання існуючих інструментів для програмно-конфігурованих мереж для управління ресурсами та потоками даних;

У рамках розвитку концепції цієї технології було розроблено спеціальний протокол передачі OpenFlow. Цей протокол базується на концепції управління обробки потоків даних. Принцип роботи протоколу простий, якщо комутатор отримує дані певного потоку, він дивиться у спеціальні таблиці потоків, іменований як flow tables, у яких зазначені дії що необхідно зробити у разі отримання цього потоку, якщо він є новим, тоді комутатор запитує інформацію в контролера, згодом контролер виробляє створення нового маршруту і заносить їх у таблиці потоків пристроїв.

Головною проблемою даного підходу є величезна кількість інформації про просування пакетів, яка називається forwarding state explosion. Через це

навантаження на контролер є величезною і величезні сегменти мережі, що включають більше 1000 передавальних пристроїв, є непосильною ношею з точки зору затримок і стабільності передавальної інфраструктури. [5]

1.2.1 Архітектура програмно-конфігурованих мереж

В архітектурі ПКМ (software-defined network) є три рівні управління мережею (Рис. 1.1.):

1. Інфраструктурний рівень, що представляє собою набір мережевих пристроїв таких як комутаторів і каналів передачі даних;

2. Рівень управління, що включає мережну операційну систему, яка забезпечує додаткам мережеві сервіси і програмний інтерфейс для управління мережевими пристроями та мережею;

3. Рівень мережевих додатків для гнучкого та ефективного управління мережею.

Перший пункт дозволяє швидко змінювати та модифікувати інфраструктуру мережі без повної перебудови всього сегмента мережі.

Другий пункт вирішує проблему недостатньо ефективного використання ресурсів передаючих пристроїв передачі даних.

Третій пункт дає широкі можливості для керування доступом до мережі, управлінню передачею даних, модифікації генерації маршрутів та інших можливостей. [5]

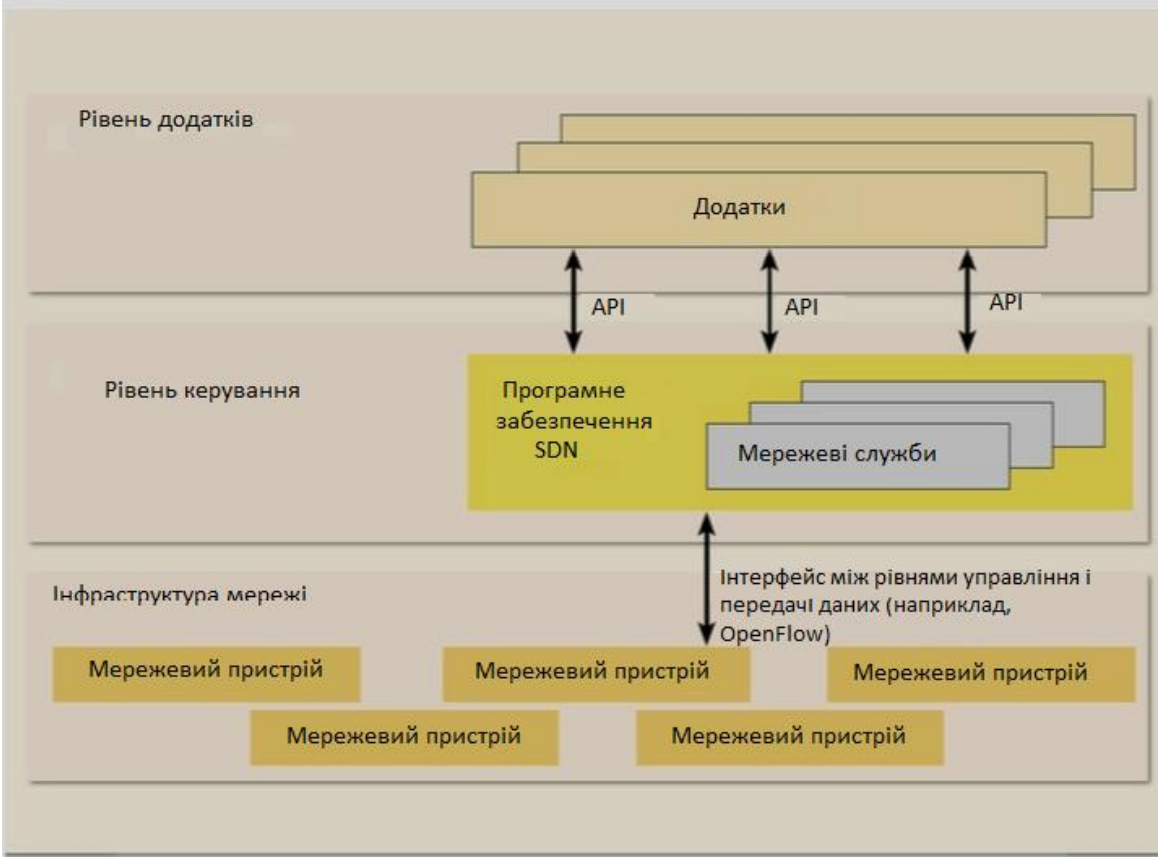

Рисунок 1.1 - Архітектура програмно-конфігурованих мереж

Найбільш перспективним і швидко розвиваючим стандартом для ПКМ є протокол OpenFlow. Це відкритий протокол взаємодії, розроблений спеціально для ПКМ, в якому описуються вимоги до комутатора, що підтримує протокол OpenFlow для віддаленого управління. [3]

За допомогою сучасних маршрутизаторів повсюдно виконуються два головні завдання: передача даних - передача пакета з вхідного порту на певний вихідний порт і управління даними - обробка пакета та ухвалення рішення про те, куди його передавати далі, на основі поточної завантаженості маршрутизатора та інших факторів. [7] Це відображає рівень передачі даних, на якому об'єднані засоби передачі, різні лінії зв'язку, каналоутворювальне обладнання, маршрутизатори, комутатори та управління станами засобів передачі даних (Рис. 1.2.). Розвиток маршрутизаторів по сьогоднішній день рухався в напрямку об'єднання цих рівнів, проте зі ставкою на передачу, включаючи різні апаратні прискорення, вдосконалення ПЗ і впровадження нових функціональних

можливостей для збільшення швидкості обробки кожного пакета, в той же час рівень управління залишався досить примітивним і спирався на непрості розподілені алгоритми маршрутизації та складні інструкції з налаштування та конфігурування мережі. З огляду на складність програмного забезпечення маршрутизаторів, у яких розроблено рівень управління, компанії розробники пристроїв закривали вихідний код. [6]

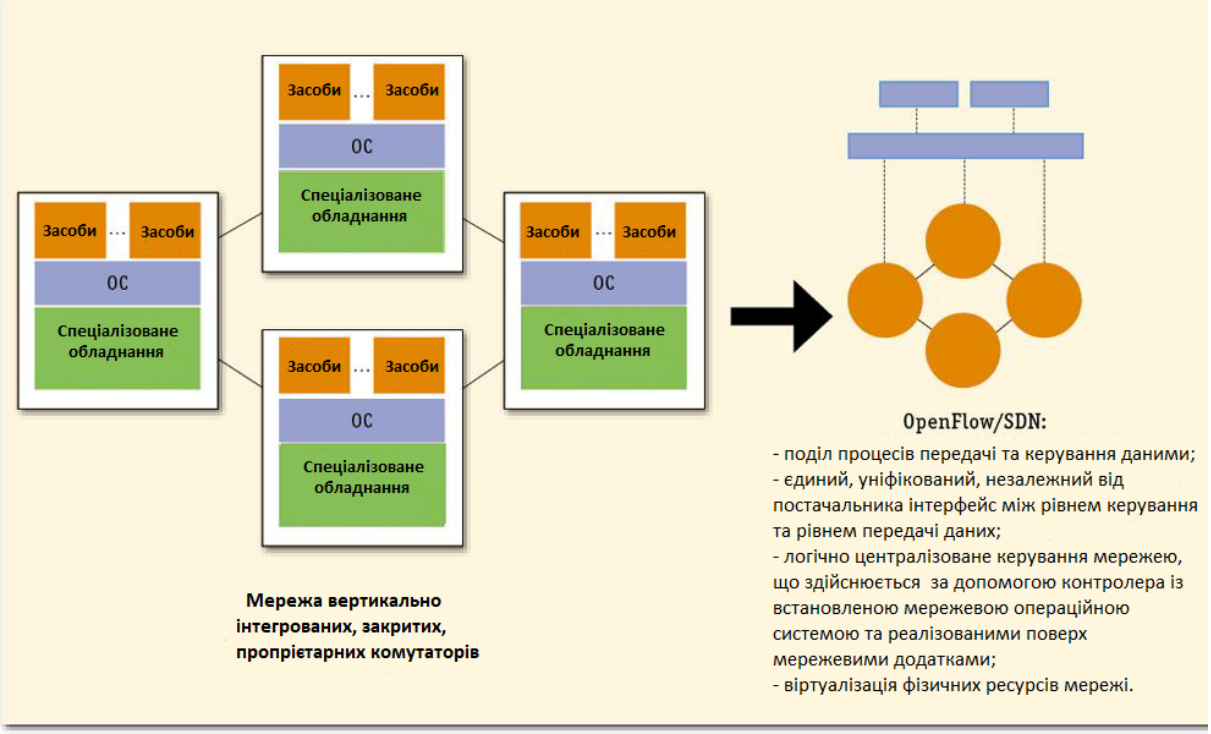

Рисунок 1.2 - Традиційні мережі та програмно-конфігуровані мережі

Згідно зі специфікацією 1.3 стандарту OpenFlow, взаємодія програмно-конфігурованого контролера з комутатором здійснюється засобами протоколу OpenFlow - в якому будь-який комутатор повинен містити як мінімум одну таблицю потоків і групову таблицю, а також підтримувати канал OpenFlow для зв'язку з віддаленим контролером, що є центральним сервером. Специфікація не описує внутрішній пристрій та архітектуру контролера, а також API для його внутрішніх програм. Будь-яка таблиця потоків усередині комутатора повинна містити набір записів з інформацією про потоки або правила маршрутизації. Будь-яка із записів складається з полів-ознак, лічильників та набору інструкцій. [7]

Механізм роботи комутатора OpenFlow дуже простий. У кожного нового вхідного пакета забирається заголовок, який є бітовим рядком певної довжини. Далі проводиться пошук правила в таблицях потоків для даної бітової маски. У разі знаходження збігу, над пакетом і його заголовком виконуються різні перетворення, що визначаються деяким набором інструкцій, визначених у знайденому правилі. Інструкції, асоційовані з кожним записом таблиці, описують дії, пов'язані з відправкою пакета, модифікацією його заголовка, обробкою в таблиці груп, обробкою в конвеєрі та відправкою пакета на певний порт комутатора. Інструкції конвеєра обробки дозволяють відправляти пакети в наступні таблиці для подальшої обробки та у вигляді метаданих передавати інформацію між таблицями. Інструкції також визначають правила модифікації лічильників, які можуть бути використані для збору різноманітної статистики. [7]

Якщо потрібне правило не знайдено, то пакет інкапсулюється і відправляється контролеру, той у свою чергу формує необхідне для обробки правило. Сформоване правило для певного типу пакетів завантажується до комутатора або групи підконтрольних комутаторів.

Також невідомий пакет може бути скинутий, при установці відповідного налаштування на комутаторі.

Запис про потік може повідомляти про відправлення пакета до певного фізичного, зарезервованого або віртуального порту. Зарезервовані віртуальні порти можуть визначати загальні дії пересилання такі як: відправка контролеру, широкомовне розсилання, або пересилання без використання протоколу OpenFlow. Віртуальні порти, визначені комутатором, можуть визначати групи об'єднання каналів, тунелі або інтерфейси зі зворотним зв'язком. [8]

Записи про потоки можуть також зберігати інформацію про групи, які мають додаткову обробку. Групи є набором дій для широкомовної розсилки. Також набори дій можуть представляти собою пересилання зі складною семантикою, наприклад, агрегування каналів або зміна маршруту. Механізм груп дозволяє ефективно змінювати однакові вихідні дії потоків. Таблиця груп зберігає у собі записи про групи, в свою чергу зберігаючи список контейнерів дій з

особливою семантикою, яка залежить від типу групи. Дії в одному або декількох контейнерах дій застосовуються до пакетів, що надсилаються до групи.

Розробники комутаторів можуть вільно реалізовувати будь-яку внутрішню начинку, проте процедура перегляду пакетів та семантика інструкцій мають бути спільними для всіх. Наприклад, у той час як потік може використовувати всі групи для пересилання в кілька портів, розробник комутатора може вибрати для реалізації цього функціоналу єдину бітову маску всередині великої апаратної таблиці маршрутизації. Ще одним прикладом є процедура перегляду таблиць: конвеєр фізично може бути реалізований з використанням деякої кількості апаратних таблиць. Встановлення, оновлення та видалення правил у таблицях потоків комутатора здійснюються лише контролером. Правила можуть встановлюватися реактивно (у відповідь на пакети, що прийшли) або проактивно (наперед, до приходу пакетів). [8]

Управління даними OpenFlow здійснюється на рівні їх потоків, а не на рівні окремих пакетів. Правило в комутаторі OpenFlow прописується за участю контролера лише для першого пакета, а вся решта пакетів потоку його використовують надалі.

Наявні на сьогоднішній день фізичні комутатори ПКМ відповідають поки специфікації OpenFlow 1.0 і містять лише одну таблицю потоків.

1.2.2 Протокол OpenFlow

Ідея розробки ПКМ полягає у розробці узагальненого та незалежного мережевого обладнання, що має єдиний інтерфейс взаємодії між передавальним середовищем мережі і контролером. Ця концепція є основною в протоколі OpenFlow. Він дозволяє користувачам самостійно контролювати та визначати, за яких умов, які вузли та з якою якістю можуть взаємодіяти в мережі між собою. Протокол підтримує три типи повідомлень: контролер-комутатор, асинхронні та симетричні. [9]

Повідомлення типу контролер-комутатор ініціюються контролером і використовуються для керування та відстеження стану комутатора. Повідомлення цього типу можуть використовуватися контролером для завантаження конфігурації комутатора, для вивантаження статистики, додавання, видалення та модифікації записів у таблицях потоків.

Асинхронні повідомлення ініціюються комутатором для повідомлення контролера про різні мережні події, наприклад, отримання пакетів, видалення запису з таблиці потоків через закінчення тайм-аута. Також вони можуть повідомляти про різні зміни стану комутатора та помилки в результаті роботи комутатора. [9]

Симетричні повідомлення можуть ініціюватися комутатором або контролером без запиту та використовуються при встановленні з'єднання. Також використовуються для вимірювання пропускної здатності з'єднання контролер-комутатор, затримок або перевірки стану з'єднання.

1.2.3 Мережеві операційні системи в ПКМ

Логічно-централізоване управління даними в мережі передбачає винесення вилучення функцій управління мережею з передаючих пристроїв на окремий сервер, званий програмно-конфігурованим контролером (ПКК), який знаходиться під контролем адміністратора сегмента мережі. ПКК може керувати одним або декількома OpenFlow комутаторами. Також містить у собі спеціальну мережну операційну систему, що надає мережеві сервіси та уніфіковані інтерфейси з низькорівневого управління мережею, сегментами мережі та моніторингу за станом мережевих елементів, а також додатків, що здійснюють високорівневе управління мережею та потоками даних. [10]

Мережева ОС є спеціальною обгорткою над низькорівневим протоколом. За допомогою вбудованого API надає програмам інструменти для доступу до керування мережею, а виконує моніторинг конфігурації засобів мережі. На відміну від традиційного тлумачення терміна ОС, під МОС розуміється

програмна система, що забезпечує моніторинг, доступ та управління ресурсами усієї мережі, а не її конкретного вузла.

Подібно до класичної ОС, МОС забезпечує програмний інтерфейс для додатків управління мережею та реалізує механізми управління таблицями комутаторів: додавання, видалення, модифікацію правил та збір різноманітної статистики. [10] Таким чином, фактично вирішення завдань управління мережею виконується за допомогою програмного забезпечення, розробленого на базі API мережної операційної системи. Цей інструментарій дозволяє створювати додатки лише на рівні високорівневих абстракцій, наприклад, ім'я хоста і ім'я користувача, а не низькорівневих параметрів конфігурації, наприклад, MAC- і IP-адрес. Це дозволяє виконувати керуючі команди, незважаючи на базову топологію мережі, проте необхідно, щоб МОС підтримувала відображення між високорівневими абстракціями та низькорівневими конфігураціями.

У кожному контролері зберігається як мінімум один додаток, для керування підключеними комутаторами. Також програмне забезпечення формує модель підконтрольної топології фізичної мережі, завдяки цьому централізується керування. Представлення топології мережі містить у собі топологію комутаторів, розташування хостів і користувачів та інших елементів, і різноманітних сервісів мережі. Представлення також містить у собі зв'язок між іменами та адресами, тому одним із найважливіших завдань, що вирішуються МОС, є щомиттєвий моніторинг мережі. Це дозволяє МОС створювати додатки у вигляді централізованого ПЗ, з використанням високорівневих імен, на базі різних алгоритмів, наприклад алгоритму Дейкстри пошуку найкоротшого шляху в графі, замість складних розподілених алгоритмів на кшталт алгоритму Беллмана-Форда, в термінах низькорівневих адрес, які використовуються в сучасних маршрутизаторах. [11]

На даний момент є велика кількість реалізацій мережевих ОС для програмно-визначених мереж. Одними з найвідоміших на сьогоднішній день є: RUNOS, BigSwitch, Beacon, POX, Maestro, FloodLight, NOX та Trema.

Для контролерів в ПКМ важливою вимогою є те, щоб усі додатки одного контролера в будь-який момент часу повинні бути синхронізовані і мати однакове представлення про топологію мережі. Проте перехід від розподіленого управління мережею до централізованого таїть у собі ряд недоліків. Наприклад, зниження надійності, відмовостійкості та масштабованості. [1]

Сьогодні активно розвиваються 3 підходи для побудови розподіленого контролера, що масштабується. Вони називаються Kandoo, HyperFlow та Onix. У рамках досліджень було виявлено, що найперспективнішим є альтернативний підхід, представлений на рис. 1.3. В силу того, що будь-який контролер може бути з'єднаний декількома комутаторами, а кожен комутатор може бути з'єднаний декількома контролерами, виходить, що є можливість об'єднати контролери в груповий контролер (ГК). Група контролерів, об'єднаних у ГК, повинні мати синхронізоване подання підконтрольної топології сегмента мережі. Як бачимо на рис. 1.3, C1-C3- контролери, S1-S4 - комутатори, а V1-V3 -фрагменти мережі, до яких забезпечує доступ комутатор S1, S2, S3 відповідно. Тоді ГКІ утворюють контролери С1 та С2, ГК2 - С2 та С3, а всі додатки у ГКІ повинні мати узгоджене уявлення про топологію VI та V2, усі додатки у ГК2- про топологію V2 та V3. У разі виходу з ладу, наприклад, контролера С1 його може замінити С2, взявши на себе управління V1. Уявлення про стан відповідної частини мережі контролери можуть узгоджувати через комутатор S4, або через S1, S2 і S3. [31]

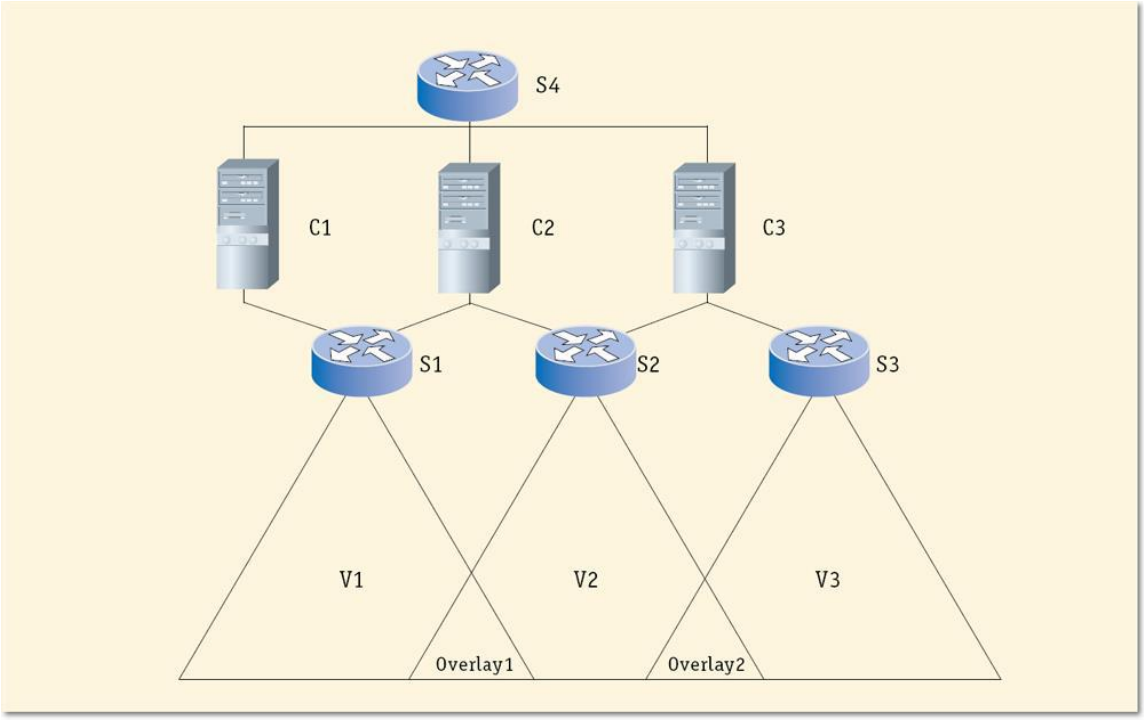

Рисунок 1.3 - Альтернативний підхід до побудови розподіленого масштабованого контролера

Такий принцип побудови розподіленого контролера вирішує проблему масштабованості та підвищує стійкість до відмови ПКМ.

1.3 Загальні відомості про побудову шляхів

Найкоротший шлях частіше всього розглядається за допомогою математичного об'єкта, званого графом.

Існують три найбільш ефективні алгоритми знаходження найкоротшого шляху:

1. Алгоритм Дейкстри (використовується для знаходження оптимального маршруту між двома вершинами);

2. Алгоритм Флойда (для знаходження оптимального маршруту між усіма парами вершин);

3. Алгоритм Йена (для знаходження k-оптимальних маршрутів між двома вершинами).

Основним завданням даної роботи є програмна реалізація алгоритму пошуку найкоротшого шляху між двома будь-якими вершинами графа. [12]

Програма повинна працювати так, щоб користувач вводив кількість вершин та довжини ребер графа, а після обробки цих даних на екран виводився найкоротший шлях між двома заданими вершинами та його довжина. Необхідно передбачити різні результати пошуку, щоб програма не видавала помилок та працювала правильно.

Ця програма може використовуватися в дискретній математиці для дослідження графів або як наочний посібник, що демонструє застосування алгоритму Флойда на практиці.

Граф - це система, яка інтуїтивно може бути розглянута як безліч вузлів і безліч ліній, що з'єднують їх, геометричний спосіб завдання графа представлений на рис. 1.4. Вузли називаються вершинами графа, лінії зі стрілками - дуги, без стрілок - ребрами. Граф, в якому - напрямок ліній не виділяється, тобто. всі лінії є ребрами, називається неорієнтованим; граф, у якому напрямок ліній принциповий, тобто лінії є дугами, що називаються орієнтованими.

Теорія графів може також розглядатися як розділ дискретної математики, а якщо точніше, то теорії множин, і формальне визначення графа є наступним: якщо задана кінцева множина X, що складається з n елементів (X= {1,2,..., n}), званих вершинами графа, і підмножина V декартового добутку, що називається безліччю дуг, орієнтованим графом G називається сукупність (X, V), неорієнтованим графом називається сукупність множини X і множини невпорядкованих пар елементів, кожен з яких належить множині X. Дугу між вершинами i і j позначатимемо (i, j). Число дуг графа позначатимемо m (V = (v1, v2 ...,. vm)). Приклад графа представлено на рис. 1.4.

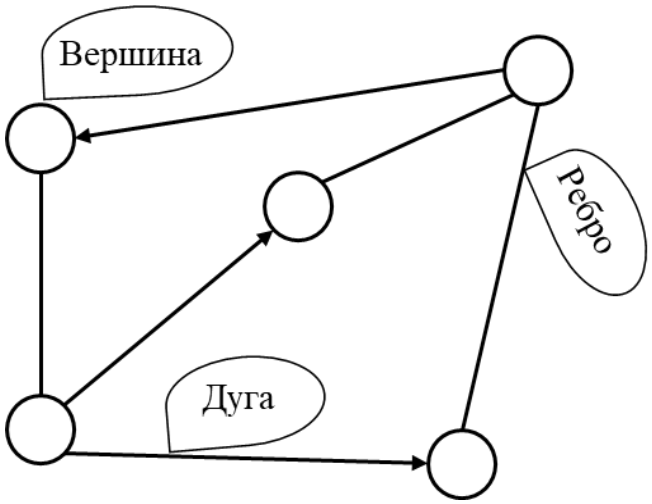

Рисунок 1.4 - Приклад графа

Мова графів є зручною для опису багатьох фізичних, технічних, економічних, біологічних, соціальних та інших систем.

У рамках теорії графів зазвичай вирішуються транспортні та технологічні завдання.

- «Транспортними» завданнями є величезний спектр логістичних завдань, у яких вершинами графа є пункти, а ребрами – дороги чи інші транспортні маршрути. Іншим прикладом є мережі постачання.

В них вершинами є пункти виробництва та споживання, а ребрами можливі маршрути переміщення. Відповідний клас завдань оптимізації потоків вантажів, розміщення пунктів виробництва та споживання тощо, іноді називається завданнями забезпечення чи завданнями про розміщення. Їхнім підкласом є завдання про вантажоперевезення;

- «Технологічні завдання», в яких вершини відображають виробничі елементи заводів, верстатів і т.д., а дугами є потоки сировини, матеріалів і продукції між ними. Суть цих завдань полягає у визначенні оптимального завантаження всіх виробничих елементів;

1.4 Вибір інструментальних засобів для реалізації алгоритму, емуляції складного сегменту ПКМ

1.4.1 Огляд мережевих операційних систем

З моменту розвитку програмно-конфігурованих мереж, з'явилося багато контролерів, які іноді називають мережевими операційними системами, проте, не всі ці контролери досить хороші. Одними з тих, що добре себе зарекомендували є контролери: RUNOS, HP Virtual Application Networks ПКС Controller і POX. Нижче буде проведено детальний огляд контролерів з метою виявити слабкі та сильні сторони кожного. [13]

1.4.1.1 Огляд RUNOS

Починаючи з 2013 року, Центр прикладних досліджень комп'ютерних мереж розпочав розробку контролера RUNOS. Це найшвидший на сьогоднішній момент контролер порівняно з існуючими аналогами. Даний контролер дуже ефективно використовує обчислювальні ресурси управління мережею. Пропонує величезний набір мережевих додатків, можливість фільтрації трафіку, роботу з мережевими протоколами та інше.

На сьогоднішній момент є однією з найшвидших і найефективніших мережевих ОС, за невеликої середньої швидкості генерації нового з'єднання 45 мкс і підтримкою 1 тисячі комутаторів одночасно. У RUNOS реалізовано балансування навантаження, узгоджене бачення всієї мережі, роботу з розподіленими мережними додатками, безпеку та протидію зовнішнім навантаженням.

Головними характеристиками є:

- обробка 30 мільйонів потоків за секунду;
- час встановлення нового з'єднання 45 мкс;
- підтримка 1000 комутаторів;

- можливість керування з графічного інтерфейсу.

Розробники запевняють, що це найшвидший ПКМ-контролер у світі. Його продуктивність досягається за рахунок використання можливостей багатоядерних та багатопроцесорних систем. У ньому задіяний набір мережевих додатків - як зі світу традиційних мереж, так і нових: L2/L3-маршрутизація з урахуванням якості обслуговування, багатопотокова маршрутизація, фільтрація трафіку, робота з мережевими протоколами (ARP, DNS, DHCP, BGP), трансляція адрес (NAT), балансування навантаження, віртуалізація мереж, анти-DDoS, верифікація мережі, інтеграція із системою управління ЦОД.

RUNOS насамперед орієнтований на корпоративний сегмент. Його споживчою аудиторією бачать мережевих адміністраторів та інженерів ЦОД, телеком-операторів, сервіс-провайдерів, а також учнів за напрямом «мережеві технології», дослідників у галузі комп'ютерних мереж та розробників перспективних мережевих технологій. [14]

1.4.1.2 Огляд контролера HP Virtual Application Networks SDN Controller

Цей контролер для програмно-конфігурованих мереж розробляється компанією HP. Є платним, проте має можливість роботи з декількома протоколами з програмно-конфігурованих мереж.

Компанія дотримується своєї політики відсутності прив'язки програмного забезпечення до апаратних засобів і тому їх контролер забезпечує можливість взаємодії з будь-яким мережним обладнанням, яке підтримує протоколи програмно-конфігурованих мереж.

Головними характеристиками є:

- максимальна кількість пристроїв, що підтримуються - 4 000;
- максимальна кількість підключених комп'ютерів - 30000;
- максимальна кількість потоків за секунду - 2.3 мільйона;
- час встановлення з'єднання - 50-60 мкс.

Як видно, цей контролер є потужним інструментом, однак, час на встановлення з'єднання та максимальна кількість потоків залишає бажати кращого.

1.4.1.3 Огляд NOX

NOX був розроблений інженерами Nicira Networks паралельно із протоколом OpenFlow. Спочатку був розроблений за допомогою двох мов: C++ та Python. У 2008 р. NOX був опублікований під ліцензією GPL і з того часу цей контролер є базовим для багатьох науково-дослідних груп, які тільки починають вивчати ПКМ. NOX орієнтований на дистрибутиви Linux (зокрема Ubuntu 11.10 та 12.04, але також можливе використання на Debian та RHEL 6). Містить сервіси для побудови топології мережі та L2-L3 комутації.

У процесі тестування з'ясувалося, що підтримка двох мов сильно позначається на продуктивності, тому частина, що відповідає за Python, витягли в окремий проект, який пізніше назвали POX.[14]

1.4.1.4 Огляд POX

Контролер POX є «молодшим братом» NOX. Якщо при розробці NOX основною метою була висока продуктивність, то POX насамперед спрямований на навчання і дослідження. За своєю суттю POX - це платформа для швидкої розробки та прототипування ПЗ управління мережею. Цей контролер написано на Python, його легко запустити на Window, Linux та Mac OS. Наприклад, дослідницька група у Стенфорді використовує POX на дослідження ключових проблем ПКМ. Є досить продуктивним контролером, розповсюджується безкоштовно та підтримує останню версію протоколу OpenFlow. POX перебуває у стадії активного розвитку: всі вдалі ідеї постійно переміщаються з лабораторних експериментів до офіційних релізів контролера POX (принаймні так стверджують його розробники зі Стенфорда).

Головними характеристиками є:

- максимальна кількість пристроїв - 3000;

- максимальна кількість потоків за секунду - 1 мільйон;

- час встановлення з'єднання - 90 мкс.

POX підтримує ті ж компоненти, графічний інтерфейс, засоби візуалізації, як і NOX. [15]

1.4.1.5 Огляд Beacon

Це досить швидкий, кросплатформовий модульний OpenFlow контролер на Java. Цей контролер розробляється вже понад два роки. Beacon використовується в багатьох науково-дослідних проектах та тестових впровадженнях/розгортаннях. Beacon застосовується в експериментальному ЦОДі Стенфорда, в якому він керує 100 віртуальними і 20 фізичними комутаторами.

Beacon написаний на Java та працює на багатьох платформах, починаючи від високопродуктивних багатоядерних Linux-серверів до смартфонів на Android. Розробник контролера – Девід Еріксон, учень Ніка МакКеона, Стенфорд.

1.4.1.6 Огляд Maestro

Maestro – це операційна система, розроблена в Rice University. Maestro надає інтерфейси для реалізації модульних додатків управління мережею для доступу та зміни стану мережі, а також координації їх взаємодії. Незважаючи на те, що цей проект спрямований на створення OpenFlow контролера, Maestro не обмежується тільки OpenFlow-мережами. Середовище програмування Maestro надає інтерфейси для додавання нових користувальницьких компонентів по управлінню мережею.

Крім того, Maestro намагається використовувати паралелізм у межах однієї машини для покращення пропускної спроможності системи.

Розробники заявляють основними властивостями Maestro портативність та масштабованість. Maestro розроблена на Java (і сама платформа, і її компоненти), є універсальною для різних ОС та архітектур. [16]

В результаті аналізу мережевих ОС було виявлено, що для реалізації даної магістерської дисертації найкраще підходить RUNOS завдяки швидкості роботи, суттєвого обсягу потоків, що одночасно підтримуються, і швидкості відгуку.

1.4.2 Огляд засобів для емуляції програмно-конфігурованої мережі

На даний момент існує велика кількість засобів для емуляції програмно-конфігурованих мереж. Найвідомішими та визнаними фахівцями є: ns-3 Network Simulator, OPNET, NetSim та Mininet. При виборі засобу для емуляції варто звернути увагу на функціональні можливості, підтримку необхідних протоколів переваги і недоліки. Також варто врахувати значущість недоліків емулятора для поставлених завдань.

1.4.2.1 Огляд ns-3 Network Simulator

ns-3 Network Simulator є проектом з відкритим кодом, який розповсюджується за ліцензією GNU GPL. Мета яку поставили перед собою розробники - це створити безкоштовний симулятор побудови комп'ютерних мереж, за допомогою якого можна проводити різні дослідження щодо роботи комп'ютерних мереж, без необхідності будувати реальні.

Перша версія ns з'явилася в далекому 97 року, тоді симулятор був досить слабким, незважаючи на досить швидке ядро, написане на C++, через сценарії, засновані на Tcl, сильно гальмувалася робота.

Трохи пізніше, DARPA, Xerox та ряд інших зацікавлених компаній та організацій почали допомагати у розробці такого програмного забезпечення, через кілька років з'явився ns-2, який став більш продуктивним і просунутим з погляду налаштувань.

Починаючи з 2006 року почалася розробка третьої версії програми, дана версія має істотну перевагу в порівнянні зі старими версіями, змінився скриптовий движок, тепер використовувані скрипти пишуться на мові Python, простий в освоєнні. Крім цього, забезпечена зворотна сумісність з другою версією.

Реліз третьої версії відбувся у червні 2008 року, після цього проект доопрацьовувався до дуже стабільної версії, у 2010 році вийшла досить стабільною, щоб повністю замінити другу версію, таким чином у 2010 році підтримка ns-2 припинилася. [17]

Найголовнішими плюсами даного емулятора є можливість створення топології, конфігурування вузлів і з'єднань, аналіз навантаження і візуалізація даних.

Найголовнішим мінусом є відсутність підтримки протоколів програмно-конфігурованих мереж.

1.4.2.2 Огляд OPNET

OPNET – це пропрієтарне програмне забезпечення, розроблене компанією OPNET Technologies. Перша версія програми була представлена публіці 2000 року.

Основна аудиторія даного продукту є комерційні компанії, яким необхідно розробити дата-центр, центр обробки даних або значну локальну мережу.

Їхній інструмент надає зручний інтерфейс при роботі з емульованим середовищем, за допомогою такої програми дуже просто та швидко будується мережа будь-якої складності.

Opnet містить бібліотеки, завдяки яким здійснюється формування телекомунікаційних мереж і полегшує вивчення моделі шляхом підключення різних типів вузлів, з використанням різних видів зв'язку і тощо [18]

Редактор вузлів представляє собою редактор, який використовується для створення моделей вузлів та вказівки їх внутрішньої структури. Ці моделі використовуються для створення вузлів у мережі в редакторі проекту.

Внутрішні вузли моделі мають модульну структуру, яка визначається як вузол підключення кількох модулів з пакетом потоків та кабелів. Це з'єднання дозволяє обмінюватися інформацією та пакетами між ними.

Кожен модуль має певну функцію у вузлі, таку як створення пакетів, склеювання, процес або передача і прийом.

Також має потужний редактор моделі з'єднань. Редактор дозволяє створювати нові типи об'єктів зв'язку. Кожен новий тип з'єднання може мати різні атрибути та представлення.

Типи підтримуваних редактором з'єднань: всі з'єднання, які ми можемо підтримувати, одне чи всі чотири, допускаються симулятором. Цими з'єднаннями є: крапка-крапка, дуплекс крапка-крапка, шина і болт. [20]

Об'єкти представляють у цьому редакторі процесори. Їхня поведінка визначається в процесі редактора. Є попередньо налаштовані моделі, такі як джерела даних, поглиначі і т.п. буд.

Найголовнішими плюсами даного емулятора є простота використання, підтримка всіх існуючих протоколів.

Найголовнішим мінусом є ціна цього товару.

1.4.2.3 Огляд NetSim

Програмне забезпечення розроблене компанією TETCOS. Розробку було розпочато у червні 2002 року. Це програмне забезпечення підтримує сучасні протоколи, забезпечує можливість емуляції безпроводових мереж. [21]

Проект створено мовою C, що робить симулятор дуже швидким. Цей симулятор є чудовим інструментом для розробки архітектури складних мереж.

В силу своєї ефективності та гнучкості, даний інструмент є складним у освоєнні та використовується лише професіоналами.

Цей продукт має велику вартість, що є його основним мінусом.

1.4.2.4 Огляд Mininet

Симулятор мережі Mininet є безкоштовним програмним забезпеченням розроблюваним співтовариством програмістів, які є прихильниками безкоштовного програмного забезпечення.

Mininet спроектований таким чином, щоб можна було легко та просто створювати віртуальні програмно-конфігуровані мережі. Крім цього було розроблено спосіб підключення віддаленого контролера, який дозволяє тестувати будь-які програмно-конфігуровані контролери.

Даний емулятор з'явився в першу чергу через швидко зростаючий інтерес до програмно-конфігурованих мереж. Має функції генерації даних, дозволяє тестувати отриману мережу на предмет продуктивності і відразу виявляти слабкі місця спроектованої. топології. [22]

Вся емуляція відбувається лише на рівні ядра. За допомогою зручного API дозволяє швидко будувати складні топології, які можуть зберігати не одну тисячі пристроїв.

У ході аналізів виявили, що Mininet є лідером серед емуляторів за рахунок наданих можливостей та безкоштовного поширення.

1.4.3 Огляд середовищ розробки

З моменту розвитку програмування для спрощення та прискорення розробки були розроблені різні середовища розробки, які надають широкий спектр можливостей та інструментів для прискорення проектування та написання коду.

За недовгу історію епоху програмування було розроблено багато середовищ розробки під різні мови програмування. Оскільки в рамках виконання проекту знадобиться мова програмування C++, було розглянуто найкращі середовища

розробки мови програмування C++. Найкращими на даний момент є Microsoft Visual Studio, Eclipse CDT, NetBeans, Code Lite.

1.4.3.1 Огляд Microsoft Visual Studio

Microsoft Visual Studio – це продукт компанії Microsoft, розроблений насамперед для розробки програмного забезпечення під операційну систему Windows. Завдяки модульності має підтримку кількох мов програмування такі як C++, C#, JavaScript тощо. Він включає прекрасний редактор вихідного коду з підтримкою технології IntelliSense.

IntelliSense спеціальна технологія автодоповнення тексту. Як і інші системи автодоповнення є зручним інструментом для перегляду опису функцій і списків аргументів. За допомогою неї виробляється прискорення розробки програмного забезпечення, за рахунок зменшення навантаження на програміста. Крім того, вона дозволяє зменшити кількість звернень до документації, завдяки виводу частини документації у вигляді спливаючих допоміжних вікон у редакторі коду. Під час роботи IntelliSense проводить сканування метаданих класів, змінних та інших конструкцій підключених бібліотек та зберігає ці дані у внутрішню базу даних. «Класична» реалізація IntelliSense здійснює пошук спеціальних маркерів у коді, наприклад, символ точки. Як тільки виявляється маркер, то програмісту відразу демонструються доступні методи класу, згідно з його рівнем доступу, а також параметри необхідні для цього методу.

Microsoft Visual Studio є дорогим та якісним середовищем розробки, проте для студентів та викладачів може надаватися безкоштовно за спеціальною програмою DreamSpark.

Мінуса у цього середовища розробки два: сильно орієнтована під розробку під Windows та відсутність крос платформенності.

1.4.3.2 Огляд Eclipse CDT

Eclipse це вільне модульне інтегроване середовище розробки, написане мовою Java. Історія розробки починається з компанії IBM, як приймач IBM VisualAge, як корпоративний стандарт середовища розробки, різними мовами під різні платформи IBM. Згідно з даними IBM, початкова розробка коштувала 40 мільйонів доларів. Вихідний код був повністю відкритий і зроблений доступним після того, як Eclipse був переданий для подальшого розвитку незалежному від IBM спільноти.

Однією з найважливіших переваг цього середовища розробки є можливість розробки різних розширень, які можуть допомогти програмісту у розробці. Вже зараз є потужний інструмент для Java під назвою Java Development Tools, для C/C++ під назвою C/C++ Development Tools. Ці та інші розширення під різні мови було розроблено спільнотою спільно з розробниками IBM.

Завдяки розширенням Eclipse має хороші розширення для роботи з різними системами контролю версій, таких як CVS та GIT. Також має чудові засоби для зв'язку із системами керування помилок типу Youtrack або Jira.

Завдяки безкоштовному поширенню і високій якості програмного забезпечення, що не сильно поступається такому важкоатлету як Microsoft Visual Studio отримала популярність серед програмістів одинаків і величезної кількості організацій.

Завдяки тому, що сам продукт написаний на Java, середовище розробки є кросплатформенним, що дозволяє йому працювати на багатьох платформах, починаючи з Windows-подібних систем і закінчуючи широким спектром Unix-подібних систем та MacOS.

Архітектура складається з наступних компонентів:

- ядро платформи відповідає за завантаження Eclipse і запуск підключених модулів, наприклад, CDT або JDT;

- OSGi;

- SWT;

- JFace;

- робоче середовище Eclipse, що представляє собою різні панелі, редактори тощо.

OSGi (Open Services Gateway Initiative) – це бібліотека для динамічної модульної системи та сервісної платформи для Java-додатків. Вона розробляється консорціумом OSGi Alliance. Специфікації дають модель для побудови додатків з компонентів, пов'язаних разом за допомогою сервісів. Суть полягає у можливості динамічної переустановки різних компонентів та складових частин додатка без необхідності перезапуску.

Коло застосувань цього стандарту досить широке. Спочатку розробка велася для створення вбудованих систем, але зараз на базі OSGi будують багатофункціональні автономні настільні додатки та корпоративні рішення.

OSGi Alliance, раніше відома як Open Services Gateway initiative - організація відкритих стандартів (open Standards Development Organization-SDO). Протягом останніх кількох років вона розробляла засновану на Java сервісну платформу OSGi (також відома як Dynamic Module System for Java), яка могла керуватися віддалено. Основна частина цієї розробки фреймворк (каркас), що визначає модель життєвого циклу додатка та службового реєстру.

SWT (Standard Widget Toolkit) - спеціальна відкрита бібліотека для швидкої розробки графічних інтерфейсів користувача мовою Java.

Розробка велася фундацією Eclipse. Ліцензується під Eclipse Public License, однією з ліцензій відкритого програмного забезпечення.

SWT не є самостійною графічною бібліотекою, а лише являє собою крос-платформну оболонку для графічних бібліотек конкретних платформ, наприклад, під Linux SWT використовує бібліотеку GTK+, SWT написана на стандартній Java і отримує доступ до OS-специфічних бібліотек через Java Native Interface, який розглядається як сильний засіб, незважаючи на те, що це не є чистою Java.

SWT є гідною альтернативою для розробників замість AWT і Swing (Sun Microsystems). Він необхідний тим розробників, які хочуть отримати звичний зовнішній вигляд програми у цій операційній системі. Використання SWT робить Java-додаток більш ефективним і гнучким, але знижує незалежність від

операційної системи та обладнання, вимагає ручного звільнення ресурсів і певною мірою порушує Sun-концепцію платформи Java.

JFace це великий набір допоміжних Java-класів, що реалізує найзагальніші завдання побудови GUI. В рамках проекту Eclipse бібліотека JFace має наступний опис: «Елементи інтерфейсу користувача, реалізація яких може бути стомлюючою». JFace є додатковим програмним шаром над SWT, що реалізує патерн програмування Model-View-Controller. JFace надає такі можливості:

1. Надає «Viewer» класи, що відповідають за відображення та реалізують трудомісткі завдання щодо заповнення, сортування, фільтрації, а також оновлення віджетів;

2. Надає «Action» класи, які дозволяють розробнику специфічну поведінку для окремих елементів інтерфейсу користувача, таких як пункти меню, кнопки тощо;

3. Надає регістри, що містять шрифти та зображення;

4. Надає набір стандартних діалогових вікон та віджетів, а також надає фреймворк для створення складного графічного інтерфейсу взаємодії з користувачем.

Основна мета JFace полягає у наданні розробнику великої кількості інструментів для розробки інтерфейсу користувача, дозволяючи йому насамперед зосередитися на бізнес-логіці додатка [23]

Основним завданням групи розробників Eclipse було приховування реалізації компонентів графічного інтерфейсу, побудованих на основі бібліотеки SWT та по можливості максимальне використання бібліотеки JFace як більш високорівневої та найпростішої у використанні. Бібліотека JFace використовує SWT, але SWT є незалежною від JFace. Однак робоче середовище Eclipse побудовано з використанням обох бібліотек і в деяких місцях SWT використовується безпосередньо в обхід JFace.

1.4.3.3 Огляд NetBeans

NetBeans IDE - вільне інтегроване середовище розробки додатків (IDE) мовами програмування Java, Python, PHP, JavaScript, C. C++ та інших.

Проект NetBeans IDE підтримується та спонсорується компанією Oracle, проте розробка NetBeans ведеться незалежною спільнотою розробників-ентузіастів (NetBeans Community) та компанією NetBeans Org.

Останні версії NetBeans IDE підтримують рефакторинг, профіль, виділення синтаксичних конструкцій кольором, автодоповнення набираючих конструкцій на літу та безліч визначених шаблонів коду.

Для розробки програм у середовищі NetBeans та для успішної інсталяції та роботи самого середовища NetBeans має бути попередньо встановлений Sun JDK або J2EE SDK відповідної версії. Середовище розробки NetBeans за замовчуванням підтримувало розробку для платформ J2SE і J2EE. Починаючи з версії 6.0 NetBeans підтримує розробку для мобільних платформ 12ME, C++ (тільки g++) і PHP без встановлення додаткових компонентів.

У вересні 2016 року Oracle передала інтегроване середовище розробки NetBeans до рук фонду Apace.

NetBeans IDE підтримує плагіни, що дозволяє розробникам розширювати можливості середовища. Одним із найпопулярніших плагінів є потужний дизайнер звітів iReport (заснований на бібліотеці JasperReports).

На ідеях, технологіях і в значній частині на вихідному коді NetBeans IDE базуються пропоновані фірмою Sun комерційні інтегровані середовища розробки для Java - Sun Java Studio Creator, Sun Java Studio Enterprise та Oracle Solaris Studio (для ведення розробки на C, C++) . Порівняно недавно Sun стала пропонувати ці середовища розробки безкоштовно для розробників, що зареєструвалися в Sun Developer Network (SDN), сама ж реєстрація на сайті безкоштовна і не вимагає жодних попередніх умов, крім згоди з ліцензією CDDL.

NetBeans IDE доступна у вигляді готових дистрибутивів (прекомпільованих бінарних файлів) для платформ Microsoft Windows, Linux, FreeBSD, Mac OS X,

OpenSolaris і Solaris (як для SPARC, так і для x86 - Intel і AMD). Для решти платформ доступна можливість скомпілювати NetBeans самостійно з вихідних текстів.

1.4.3.4 Огляд Code Lite

CodeLite - вільне кросплатформове середовище розробки програмного забезпечення мови C/C++ з відкритим вихідним кодом.

Є простим і нехитрим середовищем розробки з вбудованою підтримкою інтеграції системи контролю версій Subcersion і Git. Має автодоповнення ctags+ clang, зручний рефакторинг коду.

CodeLite поширюється за ліцензією GNU General Public License v2 або пізнішою версією. Є вільним програмним забезпеченням. CodeLite в даний час, будучи розроблений та налагоджений, використовує себе як платформу розробки.

Серед представлених вище середовищ розробки, для розробки ефективного алгоритму оптимізації трафіку підходить Eclipse CDT завдяки своїй кросплатформенності та відсутності прив'язки до будь-якої операційної системи.

1.5 Аналітичний огляд технології проектування паралельних застосувань: Open MP

Одним з найбільш популярних засобів програмування комп'ютерів із загальною пам'яттю, що базуються на традиційних мовах програмування і використання спеціальних коментарів, в даний час є технологія OpenMP. Інтерфейс OpenMP задуманий як стандарт для програмування на масштабованих SMP-системах в моделі загальної пам'яті (shared memory model). У стандарт OpenMP входять специфікації набору директив компілятора, процедур і змінних середовища. Розробкою стандарту займається організація OpenMP ARB (ARchitecture Board), до якої увійшли представники найбільших компаній - розробників SMP-архітектур і програмного забезпечення.

До появи OpenMP не було відповідного стандарту для ефективного програмування на SMP-системах. Найбільш гнучким, мобільним і загальноприйнятим інтерфейсом паралельного програмування є MPI (інтерфейс передачі повідомлень). Проте модель передачі повідомлень недостатньо ефективна на SMP-системах та відносно складна в освоєнні.

Проект стандарту X3H5 провалився, оскільки був запропонований під час загального інтересу до MPP-систем, а також через те, що в ньому підтримується тільки паралелізм на рівні циклів. OpenMP розвиває багато ідей X3H5.

POSIX-інтерфейс для управління потоками (нитками) виконання (Pthreads) підтримується широко (практично на всіх UNIX-системах), однак з багатьох причин не підходить для практичного паралельного програмування: немає підтримки Fortrana, занадто низький рівень, немає підтримки паралелізму за даними, механізм ниток спочатку розроблявся не для цілей організації паралелізму.

OpenMP можна розглядати як високо рівневу надбудову над Pthreads (чи аналогічними бібліотеками). Багато постачальників SMP-архітектур (Sun, HP, SGI) у своїх компіляторах підтримують спец директиви для розпаралелювання циклів. Однак ці набори директив, як правило дуже обмежені, несумісні між собою; в результаті чого розробникам доводиться розпаралелювать застосування окремо для кожної платформи. OpenMP є багато в чому узагальненням і розширенням згаданих наборів директив. OpenMP надає розробнику наступні переваги:

1. За рахунок ідеї інкрементального розпаралелювання" OpenMP ідеально підходить для розробників, що бажають швидко розпаралелить свої обчислювальні програми з великими паралельними циклами. Розробник не створює нову паралельну програму, а просто послідовно додає в текст послідовної програми OpenMP-директиви.

2. При цьому OpenMP - досить гнучкий механізм, що надає розробнику більші можливості контролю над поведінкою паралельного застосування.

3. Передбачається, що OpenMP-програма на однопроцесорній платформі може бути використана в якості послідовної програми. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs).

Ше одним з достоїнств OpenMP його розробники вважають підтримку так званих "orphan" (відірваних) директив, тобто директиви синхронізації і розподілу роботи можуть не входити безпосередньо в лексичний контекст паралельної області.

На даний момент технологія OpenMP підтримується більшістю компіляторів мови C / C + +. Дещо гірше справа йде з інструментами тестування паралельних OpenMP програм. Інструменти аналізу, перевірки та оптимізації паралельних програм хоча й існують давно, до недавнього часу були мало задіяні при розробці прикладного програмного забезпечення. Тому вони часто є менш зручними, ніж інші інструментальні засоби розробки. Найбільш повно процес розробки паралельних OpenMP програм підтриманий у пакеті Intel Parallel Studio. Є інструмент попереднього аналізу коду, для виявлення ділянок коду, які потенційно можна ефективно розпаралелить. Є добре оптимізуючий компілятор з підтримкою OpenMP, інструмент динамічного аналізу для виявлення паралельних помилок. Додатково можна виділити інструмент VivaMP, що входить до складу PVS-Studio. Це статичний аналізатор коду, спеціалізований на виявленні помилок у OpenMP програмах на етапі їх написання.

Робота OpenMP-застосунку починається з єдиного потоку - основного. У застосунку можуть міститися паралельні регіони, входячи в які, основний потік створює групи потоків (що включають основний потік). Наприкінці паралельного регіону групи потоків зупиняються, а виконання основного потоку триває. У паралельний регіон можуть бути вкладені інші паралельні регіони, в яких кожен потік первісного регіону стає основним для своєї групи потоків. Вкладені регіони можуть у свою чергу включати регіони більш глибокого рівня вкладеності. Паралельну обробку в OpenMP ілюструє рис. 1.5.

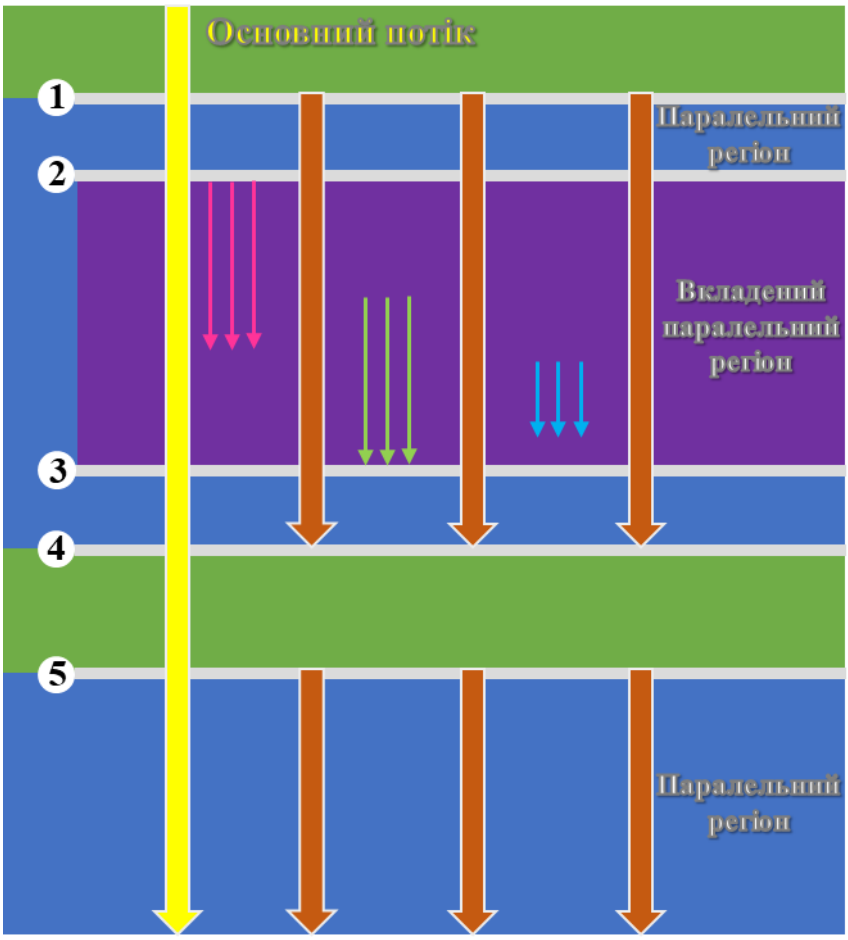

Рисунок 1.5 - Паралельна обробка в OpenMP

Сама ліва стрілка представляє основний потік, який виконується самостійно, поки не досягає першого паралельного регіону в точці 1. У цій точці основний потік створює групу потоків, і тепер всі вони одночасно виконуються в паралельному регіоні. У точці 2 три з цих чотирьох потоків, досягнувши вкладеного паралельного регіону, створюють нові групи потоків. Вихідний основний і потоки, що створили нові групи, стають власниками своїх груп (основними в цих групах). Потоки можуть створювати нові групи в різні моменти або взагалі не зустріти вкладений паралельний регіон. У точці 3 вкладений паралельний регіон завершується. Кожен потік вкладеного паралельного регіону синхронізує свій стан з іншими потоками в цьому регіоні, але синхронізація різних регіонів між собою не виконується. У точці 4 закінчується перший паралельний регіон, а в точці 5 починається новий. Локальні дані кожного потоку в проміжках між паралельними регіонами зберігаються.

2 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ

2.1 Проектування структури ефективного алгоритму оптимізації трафіку

Суть ефективного алгоритму оптимізації трафіку у тому, щоб знизити затримки створення з'єднання. Насамперед варто звернути увагу на те, що таке потоки в рамках програмно-конфігурованих мереж.

Потік даних - це певний маршрут, який прокладається всередині сегмента мережі між комутатором А і комутатором В, для того, щоб кількість пакетів досягла деякий вузол мережі.

В програмно-конфігурованому контролері використовується стандартний алгоритм Дейкстри для побудови маршруту на графі, що є дуже швидким рішенням, т.к. використовується універсальна версія алгоритму з допомогою складних структур даних. У разі його модифікації можна зробити поліпшення продуктивності у межах даної задачі.

Алгоритм Дейкстри не гарантує знаходження шляху на графі, для вирішення цієї проблеми необхідно додатково розробити механізм відновлення маршруту з цільової вершини, яка не потрапила до маршруту.

Таким чином ми отримуємо, що наш алгоритм буде розбитий на дві підзадачі, перша це розробка модифікація алгоритму Дейкстри на графах для прискорення створення нових маршрутів в рамках даної задачі, друга - це розробка швидкого алгоритму відновлення маршруту від кінцевої вершини до стартової в рамках побудованого алгоритмом Дейкстри шляху.

Алгоритм генерації маршрутів завжди повинен будувати оптимальний маршрут, щоб виключити ситуацію, необґрунтованого навантаження одного комутатора по відношенню до інших, якщо це можливо.

2.1.1 Проектування та реалізація алгоритму побудови маршрутів на базі простих структур даних

RUNOS є контролером з відкритим вихідним кодом, а значить, можна вільно модифікувати вихідні файли. Насамперед варто відзначити, що спочатку використовувалася бібліотека boost для розрахунку маршруту за алгоритмом Дейкстри, але є ряд мінусів, таких як надмірно складна структура зберігання даних і універсальна реалізація алгоритму. Це викликано тим, що бібліотека boost є вкрай універсальною у своїх реалізаціях, у зв'язку з цим за її допомогою не завжди можна отримати максимальну продуктивність. [24]

Перше на що варто звернути увагу - це структура зберігання даних, вона є складною з не найшвидшою швидкістю доступу.

Класичною структурою даних для роботи з графами є двовимірний масив (рис. 2.1.). Головною відмінністю є сталий час доступу до будь-якої комірки даних, і він мізерно малий чого не скажеш про складні структури даних. Завдяки цій перевазі масиви дозволяють швидко модифікувати зв'язок між окремими вузлами мережі.

При розрахунку маршруту є ряд складнощів, і найголовніша полягає в тому, що для повного обходу всіх можливих ребер одного вузла, потрібно буде здійснити O(n) операцій, де n – це кількість комутаторів, а в гіршому випадку це буде $O(n^2)$.

В більшості мов програмування масиви простіше ініціалізувати та використовувати, ніж складні структури даних.

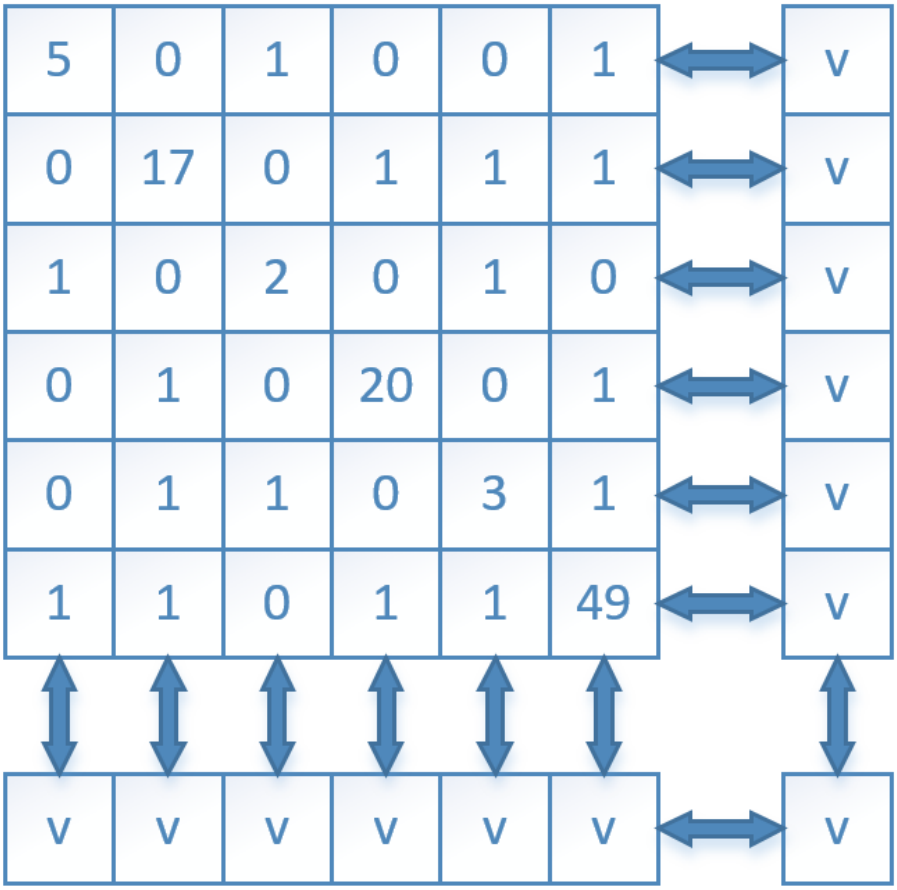

Рисунок 2.1 - Подання двовимірного масиву

Проблему обходу можна буде вирішити за допомогою розробки спеціальної, простої структури даних, яка дозволить швидко обходити всі ребра певної точки завдяки тому, що в ній зберігаються лише реальні зв'язки. Крім переваги в швидкості розрахунків, є ще одна важлива перевага, це займана пам'ять, в більшості випадків вона буде менше, ніж у масиву, винятком буде ситуація, коли у кожного комутатора буде більше 50 з'єднань з іншими пристроями. Важливим недоліком такої структури буде швидкість модифікації такої структури.

У лістингу 2.1 представлено реалізацію двовимірного масиву для зберігання даних про топологію мережі.

Лістинг 2.1 - Реалізація двовимірного масиву для зберігання даних про топологію мережі.

```
graphData = new *int32_t [count];
        for (int32_t i = 0; i < count; i++) {
                graphData[i] = new int32_t[count];
                for (int j=0; j < count; j++) {
                        graphData[i][j]=0;
                }
        }
```

Для простої взаємодії з масивом, необхідно розробити обгортку з методами, що дозволяють робити різні дії з масивом, такі як ініціалізація масиву, додавання або видалення зв'язку між вузлами, збільшення або зменшення кількості потоків, що проходять. Також усередині обгортки має бути реалізація пошуку маршруту у вигляді data_link_route. Цю логіку реалізують такі методи: addLink, removeLink, incrementFlows, decrimentFlows і computeRoutes.

Метод addLink виконує генерацію зв'язку між двома вузлами. На лістингу 2.2 представлено пошук початкового вузла в масиві даних.

Лістинг 2.2 - Пошук початкового вузла у масиві даних.

```
map::iterator it=dpidToGraphld.find(from.dpid);
if (it!= dpidToGraphld.end()) {
        fromId = it->second;
}
else {
        dpidToGraphId[from.dpid] = switchIdCounter++;
        fromId=dpidToGraphId[from.dpid];
}
```

На лістингу 2.3 представлено пошук кінцевого вузла у масиві даних.

Лістинг 2.3 - Пошук кінцевого вузла у масиві даних.

```
map::iterator it = dpidToGraphId.find(to.dpid);
if (it != dpidToGraphId.end()) {
        toId= it->second;
}
else {
        dpidToGraphId[to.dpid] = switchIdCounter++;
        toId=dpidToGraphId[to.dpid];
}
```

На лістингу 2.4 представлено створення зв'язку між двома комутаторами.

Лістинг 2.4 - Створення зв'язку між комутаторами.

```
graphData[fromId][toId] = 0;
graphData[toId][fromId] = 0;
```

Метод додавання виявився таким великим у першу чергу через те, що з ПКК ми отримуємо дані у вигляді структури switch_and_port, тому необхідно спочатку знайти відповідність між ідентифікатором і ідентифікатором в рамках протоколу OpenFlow, а потім зробити встановлення самого з'єднання.

Метод removeLink знищує зв'язок між двома вузлами. Перед зміною. На лістингу 2.5 представлено видалення зв'язку між комутаторами.

Лістинг 2.5 - Видалення зв'язку між комутаторами.

```
graphData[fromId][toId] = -1;
graphData[toId][fromId] = -1;
```

Метод знищення зв'язку виявився таким великим через те, що з ПКК отримуємо дані у вигляді структури switch_and_port, у якій зберігається ідентифікатор у рамках протоколу OpenFlow після знаходження ідентифікатора у масиві відбувається знищення зв'язку.

Метод incrementFlows виконує збільшення кількості потоків. На лістингу 2.6 представлено реалізацію методу.

Лістинг 2.6 - Реалізація методу incrementFlows.

```
int fromId, told;
map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
map::iterator toIt = dpidToGraphId.find(toSwitch.dpid);
if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
        fromId = fromIt->second;
        toId = toIt->second;
        graphData[fromId][toId]++;
        graphData[toId][fromId]++;
}
```

Метод decrimentFlows виконує зменшення кількості потоків, що проходять через цей комутатор. На лістингу 2.7 представлено реалізацію методу.

Лістинг 2.7 - Реалізація методу decrimentFlows.

```
int fromId, toId;
map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
map::iterator toIt=dpidToGraphId.find(toSwitch.dpid);
if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
        fromId = fromIt->second;
        toId = toIt->second;
        graphData[fromId][toId] --;
        graphData[toId][fromId]--;
}
```

Саме сам пошук шляху розбитий на кілька невеликих підзавдань. Перше підзавдання - це побудова самого маршруту за допомогою алгоритму Дейкстри, у разі знаходження мінімального маршруту до комутатора, що цікавить,

завершуємо пошук. Другий етап – відновлення шляху з кінцевої точки, до стартової.

Для покращення читаності коду та об'єднання повторюваних операцій було розроблено кілька допоміжних функцій, перша - це облік мінімальних відстаней до певних вузлів, що має назву setMinVals і findNextMinId для пошуку наступної точки, з якої необхідно проводити наступний пошук.

Метод setMinVals приймає мінімальні значення і робить сканування ваг ребер і визначає те, наскільки далеко знаходиться наступна точка. Реалізація методу представлена на лістингу 2.8.

Лістинг 2.8 - Реалізація методу setMinVals.

```
int32_t startValue = graphData[fromId][fromId];
for (int i = 0; i < count; i++) {
    int32_t flowsCount = graphData[fromId][i];
    if (flowsCount == -1) {
        continue;
    }
    int32_t currentValue = graphData[i][i];
    if (currentValue == -1 || currentValue > startValue + flowsCount)
{
        graphData[i][i] = startValue + flowsCount;
    }
```

Метод findNextMinId здійснює пошук наступного не відвіданого комутатора, у якого найменша відстань від стартової точки. Реалізація цього методу представлено у лістингу 2.9.

Лістинг 2.9 - Реалізація методу findNextMinId

```
int32_t minVal = -1;
int32_t minId = -1;
for (int32_t I = 0; i < count; i++) {
    if (!visited[i]) {
```

```
                currentVal = graphData[i][i];
                if (minVal == -1 || minVal > currentVal) {
                        minVal = currentVal;
                        minId = i;
                 }
        }
}
return minId;
```

Для відновлення маршруту використовується спеціальний алгоритм, реалізований у методі restoreRoute, якому повідомляється ідентифікатор кінцевого вузла і з нього алгоритм будує маршрут. Реалізація алгоритму представлено у лістингу 2.10.

Лістинг 2.10 - Реалізація алгоритму відновлення маршруту

```
data_link_route result;
int32_t prevId = endId;
do {
        result.insert(0, graphIdToSwitch[prevId]);
        prevId = findPrevId();
} while (prevId != -1);
return result;
```

У ході вивчення особливостей роботи алгоритму Дейкстри було виявлено важливу закономірність, що з будь-якої кінцевої точки можна знайти маршрут до початкової. Так як в ході алгоритму кожній точці проводиться призначення мінімальної відстані від початкової точки, яка є сумою ваг усіх з'єднань, що беруть участь у цьому, то можна побачити, що якщо брати кожне ребро графа і віднімати його вагу та перевіряти значення мінімальної відстані, то ця мінімальна відстань збігається тільки у тому випадку, якщо рухатися у бік початкової позиції.

Пошук попередньої точки реалізовано у функції findPrevId та представлено у лістингу 2.11.

Лістинг 2.11 - Реалізація findPrevId.

```
if (graphData[fromId][fromId]= 0) {
        return -1;
}
for (int32_t i = 0; i < count; i++) {
        if (graphData[fromId][i] < 0) {
                continue;
        }
        if (graphData[i][i] == graphData[fromId][fromId] -
graphData[fromId][i]) {
                return i;
        }
  }
  return -1;
```

Були зроблені такі оптимізації алгоритму Дейкстри, реалізованого у ПКК RUNOS:

1. Завершення алгоритму відбувається одразу після знаходження мінімального маршруту до кінцевої точки. При генерації випадкових з'єднань оптимізація дозволяє зберегти величезну кількість часу, у середньому 10 мс від початкового часу;

2. Перехід на просту структуру двовимірного масиву для зберігання даних. При операціях побудови маршрутів відбувається приріст продуктивності за рахунок гарантованої швидкості доступу до кожного ребра і вузла у вигляді O(1). За рахунок цього вдається виграти від 10 до 5 мс залежно від розміру сегмента;

3. Також було оптимізовано зберігання вже відвіданих вершин у вигляді масиву bool значень. Цей масив створюється лише один раз під час старту програми. Таке рішення було зроблено для прискорення роботи, адже операція

генерації та видалення масиву є найдорожчим задоволенням. Спочатку цей масив генерувався всередині роботи алгоритму, після аналізу продуктивності за допомогою профайлера було виявлено, що 2 мс витрачається на створення та видалення цього масиву і було ухвалено рішення створювати його лише один раз.

Оптимізований алгоритм Дейкстри розбитий на кілька етапів: ініціалізація даних, пошук мінімальних маршрутів зі стартової позиції, відновлення маршруту. Алгоритм реалізований у функції computeRoute. У лістингу 2.12 представлено ініціалізацію даних перед пошуком.

Лістинг 2.12 - Ініціалізація даних перед пошуком.

```
for (int i = 0; i < count; i++) {
        visited = false;
        graphData[i][i] = -1;
}
int fromId, toId;
fromId = dpidToGraphId[from.dpid];
toId = dpidToGraphId[to.dpid];
```

Після успішної ініціалізації відбувається виконання першого етапу циклу. При першому кроці здійснюється пошук мінімальних відстаней з початкової точки до всіх доступних. Реалізація цього етапу роботи алгоритму представлена на лістингу 2.13.

Лістинг 2.13 - Реалізація першого кроку алгоритму.

```
if (fromId-toId) {
        return route;
}
visited[fromId] = true;
graphData[fromId][fromId] = 0;
setMinVals(fromId);
int32_t nextMinId = findNextMinId(visited);
```

Далі основний цикл повторює той самий крок, доки не виявить побудований маршрут до кінцевої точки. Як тільки це відбувається, виробляється вихід з алгоритму. Цикл представлений у лістингу 2.14.

Лістинг 2.14 - Основний цикл алгоритму.

```
while (nextMinId != toId) {
        setMinVals(nextMinId);
        visited[nextMinId] = true;
        nextMinId = findNextMinId(nextMinId);
}
```

Після побудови маршруту виконується відновлення маршруту у форматі зрозумілим для ПКК Runos. Виклик методу представлений на лістингу 2.15.

Лістинг 2.15 - Виклик методу відновлення маршруту.

```
return restoreRoute(nextMinId);
```

Як можна побачити основні кроки збереглися, у першому кроці відбувається початковий пошук відстаней до інших вершин. Потім запускається цикл, у якому проводиться пошук мінімальної відстані до інших вершин. Як тільки буде знайдено відстань до кінцевої вершини буде проведено вихід із циклу з наступним відновленням маршруту.

2.1.2 Оптимізація роботи алгоритму побудови маршрутів на базі простих структур даних за допомогою OpenMP

На жаль, алгоритм Дейкстри є рекурентним алгоритмом і не дозволяє провести обчислення маршрутів паралельно, проте при детальному розгляді реалізації можна побачити, що даний алгоритм має одне місце, яке можна розпаралелити. Це місце знаходиться в методі setMinVals. У ньому відбувається простий нерекурентний перебір ребер та розрахунок нових значень мінімального маршруту. Реалізація паралельного методу представлена на лістингу 2.16.

Лістинг 2.16 - Паралельна версія setMinVals.

```
        int32_t startValue = graphData[fromId][fromId];
#pragma omp parallel for num_threads(4)
        for (int i = omp_get_thread_num(); i < count; i += 
omp_get_max_threads()) {
            int32_t flowsCount = graphData[fromId][i];
            if (flowsCount == -1) {
                continue;
            }
            int32_t currentValue = graphData[i][i];
            if (currentValue = -1 || currentValue > startValue + flowsCount)
 {
                graphData[i][i] = startValue + flowsCount;
            }
        }
```

Паралелізм залежно від сегмента може як прискорити, і уповільнити роботу. Для невеликих сегментів мережі, в яких менше 100 комутаторів, робота уповільнюється на 5 мс. При роботі з великими сегментами, від 500 до 1000 вузлів, спостерігається прискорення продуктивності роботи алгоритму на 3-5 мс.

OpenMP простий у використанні і включає лише два базових типів конструкцій: директиви pragma і функції виконуючого середовища OpenMP. Директиви pragma, як правило, вказують компілятору реалізувати паралельне виконання фрагментів коду. Всі ці директиви починаються з # pragma omp. Як і будь-які інші директиви pragma, вони ігноруються компілятором, що не підтримують конкретну технологію - в даному випадку OpenMP. Функції OpenMP служать в основному для зміни і отримання параметрів середовища. Крім того, OpenMP включає API-функції для підтримки деяких типів синхронізації. Щоб задіяти ці функції, бібліотеки OpenMP, період виконання (виконуючого середовища), в програму потрібно включити заголовки omp.h. Для реалізації

паралельного виконання блоків програми потрібно просто додати в код директиви pragma і, якщо потрібно, скористатися функціями бібліотеки OpenMP періоду виконання. Директиви pragma мають наступний формат:

#pragma omp <директива> [розділ [[,] розділ]...]

Найважливіша і поширена директива - parallel. Вона створює паралельний регіон для наступного за нею структурованого блоку, наприклад:

pragma omp parallel [розділ [[,] розділ]...] структурований блок

Ця директива повідомляє компілятору, що структурований блок коду повинен бути виконаний паралельно, в декількох потоках. Кожен потік буде виконувати один і той же потік команд, але не один і той же набір команд - все залежить від операторів, керуючих логікою програми, таких як if-else. В якості прикладу розглянемо класичну програму «Hello World»:

```
#pragma omp parallel
{
printf("Hello World\n");
}
```

У двопроцесорній системі, звичайно ж, розраховували б отримати наступне:

Hello World
Hello World
Тим не менш, результат міг би бути й таким:
HellHell oo WorWlodrld

Другий варіант можливий через те, що два виконуваних паралельно потоки можуть спробувати вивести рядок одночасно. Коли два або більше потоки

одночасно намагаються прочитати або змінити загальний ресурс (у нашому випадку їм є вікно консолі), виникає ймовірність гонок (race condition). Це не детерміновані помилки в коді програми, знайти які вкрай важко. За запобігання помилок відповідає програміст як правило, для цього використовують блокування, або зводять до мінімуму звернення до загальних ресурсів.

Розглянемо приклад, який визначає середні значення двох сусідніх елементів масиву і записує результати в інший масив. У цьому прикладі використовується OpenMP-конструкція # pragma omp for, яка відноситься до директив поділу роботи (work-sharing directive). Такі директиви застосовуються не для паралельного виконання коду, а для логічного розподілу групи потоків, щоб реалізувати вказані конструкції керуючої логіки.

Директива # pragma omp for повідомляє, що при виконанні циклу for в паралельному регіоні ітерації циклу повинні бути розподілені між потоками групи:

```
#pragma omp parallel
{
#pragma omp for
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
}
```

Якщо б цей код виконувався на чотирьох процесорному комп'ютері, а у змінній size було б значення 100, то виконання ітерацій 1-25 могло б бути доручено першому процесору, 26-50 - другому, 51-75 - третьому, а 76-99 - четвертим. Це характерно для політики планування, так званої статичної політики. Слід зазначити, що наприкінці паралельного регіону виконується бар'єрна синхронізація (barrier synchronization). Інакше кажучи, досягнувши кінця регіону, всі потоки блокуються до тих пір, поки останній потік не завершить свою

роботу. Якщо з тільки що наведеного прикладу виключити директиву # pragma omp for, кожен потік виконає повний цикл for, проробивши багато зайвої роботи:

```
#pragma omp parallel
{
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
}
```

Так як цикли є найпоширенішими конструкціями, де виконання коду можна розпаралелить, OpenMP підтримує скорочений спосіб запису комбінації директив # pragma omp parallel і # pragma omp for:

```
#pragma omp parallel for
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
```

В цьому циклі немає залежностей, тобто одна ітерація циклу не залежить від результатів виконання інших ітерацій. А ось у двох наступних циклах є два види залежності:

```
for(int i = 1; i <= n; ++i) // цикл 1
a[i] = a[i-1] + b[i];
for(int i = 0; i < n; ++i) // цикл 2
x[i] = x[i+1] + b[i];
```

Розпаралелить цикл 1 проблематично тому, що для виконання ітерації і потрібно знати результат ітерації і-1, тобто ітерація і залежить від ітерації і-1. Розпаралелить цикл 2 теж проблематично, але з іншої причини. У цьому циклі

можемо вирахувати значення x [i] до x [i-1], однак, зробивши так, ми більше не зможете обчислити значення x [i-1].

Спостерігається залежність ітерації i-1 від ітерації i. При розпаралелюванні циклів потрібно переконатися в тому, що ітерації циклу не мають залежностей. Якщо цикл не містить залежностей, компілятор може виконувати цикл в будь-якому порядку, навіть паралельно. Дотримання цієї важливої вимоги компілятор не перевіряє. Якщо ми вкажемо компілятору розпаралелить цикл, у якому є залежності, компілятор підкориться, що призведе до помилки. Крім того, OpenMP накладає обмеження на цикли for, які можуть бути включені в блок # pragma omp for або # pragma omp parallel for block. Цикли for повинні відповідати формату:

for ([цілочисельний тип] i = інваріант циклу; i {<,>,=,<=,>=} інваріант циклу; i {+,-}= інваріант циклу)

Ці вимоги введені для того, щоб OpenMP міг при вході в цикл визначити число ітерацій.

Порівняємо тільки що наведений приклад, що включає директиву # pragma omp parallel for, з кодом, який довелося б написати для вирішення тієї ж задачі на основі Windows API. Як видно в приклад 1, для досягнення того ж результату потрібно набагато більше коду. Так, конструктор класу ThreadData визначає, якими мають бути значення start і stop при кожному виклику потоку. OpenMP обробляє всі ці деталі сам і надає програмісту додаткові засоби конфігурування паралельних регіонів і коду.

Приклад 1. Багатопоточність в Win32 class ThreadData { public:

/ / Конструктор ініціалізує поля start і stop ThreadData (int threadNum); int start; int stop; }; DWORD ThreadFn (void * passedInData) { ThreadData * threadData = (ThreadData *) passedInData; for (int i = threadData-> start; i <threadData-> stop; + + i) x [i] = (y [i-1] + y [i +1]) / 2; return 0; }

void ParallelFor () {

/ / Запуск груп потоків for (int i = 0; i <nTeams; + + i) ResumeThread (hTeams [i]); / / Для кожного потоку тут неявно викликається / / метод ThreadFn / /

Очікування завершення роботи WaitForMultipleObjects (nTeams, hTeams, TRUE, INFINITE); }

int main (int argc, char * argv []) { / / Створення груп потоків for (int i = 0; i <nTeams; + + i) { ThreadData * threadData = new ThreadData (i); hTeams [i] = CreateThread (NULL, 0, ThreadFn, threadData, CREATE_SUSPENDED, NULL); } ParallelFor (); / / імітація OpenMP-конструкції parallel for / / Очищення for (int i = 0; i <nTeams; + + i) CloseHandle (hTeams [i]); }

Розробляючи паралельні програми, потрібно розуміти, які дані є загальними (shared), а які приватними (private), від цього залежить не тільки продуктивність, але і коректна робота програми. У OpenMP це розходження очевидно, до того ж існує можливість налаштовувати вручну. Загальні змінні доступні всім потокам з групи, тому зміни таких змінних в одному потоці видимі іншим потокам у паралельному регіоні. Що стосується приватних змінних, то кожен потік з групи має в своєму розпорядженні їх окремі екземпляри, тому зміни таких змінних в одному потоці ніяк не позначаються на їх екземплярах, що належать іншим потокам. За замовчуванням всі змінні в паралельному регіоні - загальні, але з цього правила є три винятки. По-перше, приватними є індекси паралельних циклів for. Наприклад, це відноситься до змінної i в коді, показаному в прикладі 2. Змінна j за замовчуванням не є приватною, але явно зроблена такою через розділ firstprivate.

Приклад 2. Розділи директив OpenMP і вкладений цикл for float sum = 10.0f; MatrixClass myMatrix; int j = myMatrix.RowStart (); int i; # pragma omp parallel { # pragma omp for firstprivate (j) lastprivate (i) reduction (+: sum) for (i = 0; i <count; + + i) { int doubleI = 2 * i; for (; j <doubleI; + + j) { sum + = myMatrix.GetElement (i, j); } } }

По-друге, приватними є локальні змінні блоків паралельних регіонів. В прикладі 2 така змінна doubleI, тому що вона оголошена в паралельному регіоні. Будь-які нестатичні і які не є членами класу MatrixClass змінні, оголошені в методі myMatrix:: GetElement, будуть приватними. По-третє, приватними будуть будь-які змінні, зазначені в розділах private, firstprivate, lastprivate і reduction. У

прикладі 2 змінні i, j та sum зроблені приватними для кожного потоку з групи, тобто кожен потік буде розпоряджатися своєю копією кожної з цих змінних.

Кожен з названих розділів приймає список змінних, але семантика цих розділів розрізняється. Розділ private говорить про те, що для кожного потоку повинна бути створена приватна копія кожної змінної зі списку. Приватні копії будуть ініцилізуватись значенням за замовчуванням (із застосуванням конструктора за замовчуванням, якщо це доречно). Наприклад, змінні типу int мають за замовчуванням значення 0. У розділу firstprivate така ж семантика, але перед виконанням паралельного регіону він вказує копіювати значення приватної змінної в кожен потік, використовуючи конструктор копій, якщо це доречно. Семантика розділу lastprivate теж збігається з семантикою розділу private, але при виконанні останньої ітерації циклу або розділу конструкції розпаралелювання значення змінних, зазначених у розділі lastprivate, присвоюються змінним основного потоку. Якщо це доречно, для копіювання об'єктів застосовується оператор присвоювання копій (copy assignment operator). Схожа семантика і в розділу reduction, але він приймає змінну і оператор. Підтримувані цим розділом оператори перераховані в табл. 1, а у змінної повинен бути скалярний тип (наприклад, float, int або long, але не std:: vector, int [] і т. д.). Змінна розділу reduction ініціалізується в кожному потоці значенням, зазначеним у таблиці. В кінці блоку коду оператор розділу reduction застосовується до кожної приватної копії змінної, а також до початкового значення змінної.

Таблиця 2.1 - Змінна розділу reduction

Просте блокування OpenMP	Вкладене блокування OpenMP	Win32-функція
omp_lock_t	omp_nest_lock_t	CRITICAL_SECTION
omp_init_lock	omp_init_nest_lock	InitializeCriticalSection
omp_destroy_lock	omp_destroy_nest_lock	DeleteCriticalSection
omp_set_lock	omp_set_nest_lock	EnterCriticalSection
omp_unset_lock	omp_unset_nest_lock	LeaveCriticalSection
omp_test_lock	omp_test_nest_lock	TryEnterCriticalSection

В прикладі 2 змінна sum неявно ініціалізується в кожному потоці значенням 0.0f (в таблиці вказано канонічне значення 0, але в даному випадку воно приймає форму 0.0f, так як sum має тип float). Після виконання блоку # pragma omp for над усіма приватними значеннями і вихідним значенням sum (яке в нашому випадку дорівнює 10.0f) виконується операція +. Результат присвоюється вихідний загальній змінній sum.

Як правило, OpenMP використовується для розпаралелювання циклів, але OpenMP підтримує паралелізм і на рівні функцій. Цей механізм називається секціями OpenMP (OpenMP sections). Він досить простий і часто буває корисний. Розглянемо один з найбільш важливих алгоритмів у програмуванні - швидке сортування (quicksort). Як приклад реалізуємо рекурсивний метод швидкого сортування списку цілих чисел. Заради простоти не створюємо універсальну шаблонну версію методу, але суть справи від цього анітрохи не змінюється. Код методу, реалізованого з використанням секцій OpenMP, показаний в приклад 3 (код методу Partition опущений, щоб не захаращувати загальну картину).

Приклад 3. Швидке сортування з використанням паралельних секцій void QuickSort (int numList [], int nLower, int nUpper) { if (nLower <nUpper) { / / Розбиття інтервалу сортування int nSplit = Partition (numList, nLower, nUpper); # pragma omp parallel sections { # pragma omp section QuickSort (numList, nLower, nSplit - 1); # pragma omp section QuickSort (numList, nSplit + 1, nUpper); } } }

У даному прикладі перша директива # pragma створює паралельний регіон секцій. Кожна секція визначається директивою # pragma omp section. Кожній секції в паралельному регіоні ставиться у відповідність один потік з групи потоків, і всі секції виконуються одночасно. У кожній секції рекурсивно викликається метод QuickSort. Як і у випадку конструкції # pragma omp parallel for, потрібно переконатися в незалежності секцій один від одного, щоб вони могли виконуватися паралельно. Якщо в секціях змінюються загальні ресурси без синхронізації доступу до них, результат може виявитися непередбачуваним. В цьому прикладі використовується скорочення # pragma omp parallel sections, аналогічне конструкції # pragma omp parallel for. За аналогією з # pragma omp for

директиву # pragma omp sections можна використовувати в паралельному регіоні окремо. Паралельні секції викликаються рекурсивно. Рекурсивні виклики підтримуються і паралельними регіонами, і паралельними секціями. Якщо створення вкладених секцій дозволено, в міру рекурсивних викликів QuickSort будуть створюватися все нові й нові потоки. Можливо, це не те, що потрібно програмісту, тому що такий підхід може призвести до створення великого числа потоків. Щоб обмежити число потоків, в програмі можна заборонити вкладення. Тоді наш додаток буде рекурсивно викликати метод QuickSort, використовуючи тільки два потоки. При компіляції цього додатка без параметра / openmp буде згенерована коректна послідовна версія. Одна з переваг OpenMP в тому, що ця технологія сумісна з компіляторами, що не підтримують OpenMP.

При одночасному виконанні декількох потоків часто виникає необхідність їх синхронізації. OpenMP підтримує кілька типів синхронізації, котрі допомагають у багатьох ситуаціях. Один з типів - неявна бар'єрна синхронізація, яка виконується в кінці кожного паралельного регіону для всіх зіставлених з ним потоків. Механізм бар'єрної синхронізації такий, що, поки всі потоки не досягнуть кінця паралельного регіону, жоден потік не зможе перейти його кордон. Неявна бар'єрна синхронізація виконується також у кінці кожного блоку # pragma omp for, # pragma omp single і # pragma omp sections. Щоб відключити неявну бар'єрну синхронізацію в будь-якому з цих трьох блоків поділу роботи, вкажіть розділ nowait:

```
#pragma omp parallel
{
#pragma omp for nowait
for(int i = 1; i < size; ++i)
x[i] = (y[i-1] + y[i+1])/2;
}
```

Крім вже описаних директив OpenMP підтримує ряд корисних підпрограм. Вони діляться на три великих категорії: функції виконуючого середовища,

блокування / синхронізації і роботи з таймерами. Всі ці функції мають імена, що починаються з omp_, і визначені в заголовному файлі omp.h.

Підпрограми першої категорії дозволяють запитувати і ставити різні параметри операційного середовища OpenMP. Функції, імена яких починаються на omp_set_, можна викликати тільки поза паралельних регіонів. Всі інші функції можна використовувати як в середині паралельних регіонів, так і поза ними. Щоб дізнатися або задати число потоків у групі, використовують функції omp_get_num_threads і omp_set_num_threads. Перша повертає число потоків, що входять в поточну групу потоків. Якщо викликаний потік виконується не в паралельному регіоні, ця функція повертає першу групу. Метод omp_set_num_thread задає число потоків для виконання наступного паралельного регіону, який зустрінеться поточному виконуваному потоку. Крім того, число потоків, використовуваних для виконання паралельних регіонів, залежить від двох інших параметрів середовища OpenMP: підтримки динамічного створення потоків і вкладення регіонів. Підтримка динамічного створення потоків визначається значенням булевого значення, яке за замовчуванням дорівнює false.

Якщо при вході потоку в паралельний регіон ця властивість має значення false, виконуюче середовище OpenMP створює групу, число потоків в якій дорівнює значенню, що повертається функцією omp_get_max_threads. За замовчуванням omp_get_max_threads повертає число потоків, підтримуваних апаратно, або значення змінної OMP_NUM_THREADS. Якщо підтримка динамічного створення потоків включена, виконуюче середовище OpenMP створить групу, яка може містити змінне число потоків, що не перевищує значення, яке повертається функцією omp_get_max_threads.

Вкладення паралельних регіонів також визначається булевою властивістю, яке за замовчуванням встановлено в false. Вкладення паралельних регіонів відбувається, коли поток, вже виконує паралельний регіон, зустрічається інший паралельний регіон. Якщо вкладення дозволено, створюється нова група потоків, при цьому дотримуються правила, описані раніше. А якщо вкладення не дозволено, формується група, що містить один потік. Для установки і читання

властивостей, що визначають можливість динамічного створення потоків і вкладення паралельних регіонів, служать функції omp_set_dynamic, omp_get_dynamic, omp_set_nested і omp_get_nested. Крім того, кожен потік може запросити інформацію про своє середовище. Щоб дізнатися номер потоку в групі потоків, потрібно викликати omp_get_thread_num. Необхідно пам’ятати , що вона повертає не Windows-ідентифікатор потоку, а число в діапазоні від 0 до omp_get_num_threads - 1. Функція omp_in_parallel дозволяє потоку дізнатися, чи виконує він в даний час паралельний регіон, а omp_get_num_procs повертає число процесорів в комп'ютері. У прикладі 4 ми реалізували чотири окремі паралельних регіону і два вкладених.

Приклад 4. Використання підпрограм виконуючого середовища OpenMP # include <stdio.h> # include <omp.h> int main () { omp_set_dynamic (1); omp_set_num_threads (10); # pragma omp parallel / / паралельний регіон 1 { # pragma omp single printf ("Num threads in dynamic region is =% d \ n", omp_get_num_threads ()); } printf ("\ n"); omp_set_dynamic (0); omp_set_num_threads (10); # pragma omp parallel / / паралельний регіон 2 { # pragma omp single printf ("Num threads in non-dynamic region is =% d \ n", omp_get_num_threads ()); } printf ("\ n"); omp_set_dynamic (1); omp_set_num_threads (10); # pragma omp parallel / / паралельний регіон 3 { # pragma omp parallel { # pragma omp single printf ("Num threads in nesting disabled region is =% d \ n", omp_get_num_threads ()); } } printf ("\ n"); omp_set_nested (1); # pragma omp parallel / / паралельний регіон 4 { # pragma omp parallel { # pragma omp single printf ("Num threads in nested region is =% d \ n", omp_get_num_threads ()); } } }

Скомпілювавши цей код і виконавши його на звичайному двопроцесорної комп'ютері, отримуємо такий результат:

Num threads in dynamic region is = 2

Num threads in non-dynamic region is = 10

Num threads in nesting disabled region is = 1

Num threads in nesting disabled region is = 1

Num threads in nested region is = 2

Num threads in nested region is = 2

Для першого регіону ми включили динамічне створення потоків і встановили кількість потоків у 10. За результатами роботи програми видно, що при включеному динамічному створенні потоків виконуюче середовище OpenMP вирішила створити групу, що включає всього два потоки, так як у комп'ютера два процесори. Для другого паралельного регіону виконуюче середовище OpenMP створила групу з 10 потоків, тому що динамічне створення потоків для цього регіону було відключено. Результати виконання третього і четвертого паралельних регіонів ілюструють наслідок включення і відключення можливості вкладення регіонів.

У третьому паралельному регіоні вкладення було відключено, тому для вкладеного паралельного регіону не було створено жодних нових потоків - і зовнішній, і вкладений паралельні регіони виконувалися двома потоками. У четвертому паралельному регіоні, де вкладення було включено, для вкладеного паралельного регіону була створена група з двох потоків (тобто в цілому цей регіон виконувався чотирма потоками). Процес подвоєння числа потоків для кожного вкладеного паралельного регіону може тривати, поки не вичерпається простір в стеку. На практиці можна створити кілька сотень потоків, хоча пов'язані з цим витрати легко переважать будь-які переваги.

OpenMP включає і функції, призначені для синхронізації коду. У OpenMP два типи блокувань: прості і вкладені (nestable); блокування обох типів можуть знаходитися в одному з трьох станів - неініціалізованому, заблокованому і розблокованому. Прості блокування (omp_lock_t) не можуть бути встановлені більше одного разу, навіть тим самим потоком. Вкладені блокування (omp_nest_lock_t) ідентичні простим з тим винятком, що, коли потік намагається встановити вже приналежну йому вкладене блокування, він не блокується. Крім того, OpenMP веде облік посилань на вкладені блокування і стежить за тим,

скільки разів вони були встановлені. OpenMP надає підпрограми, що виконують операції над цими блокуваннями. Кожна така функція має два варіанти: для простих і для вкладених блокувань. Можна виконати над блокуванням п'ять дій: ініціалізувати її, встановити (захопити), звільнити, перевірити і знищити. Всі ці операції дуже схожі на Win32-функції для роботи з критичними секціями, і це не випадковість: насправді технологія OpenMP реалізована як оболонка цих функцій. Відповідність між функціями OpenMP і Win32 ілюструє табл. 2.2.

Таблиця 2.2 – Відповідність між функціями OpenMP і Win32

Просте блокування OpenMP	Вкладене блокування OpenMP	Win32-функція
omp_lock_t	omp_nest_lock_t	CRITICAL_SECTION
omp_init_lock	omp_init_nest_lock	InitializeCriticalSection
omp_destroy_lock	omp_destroy_nest_lock	DeleteCriticalSection
omp_set_lock	omp_set_nest_lock	EnterCriticalSection
omp_unset_lock	omp_unset_nest_lock	LeaveCriticalSection
omp_test_lock	omp_test_nest_lock	TryEnterCriticalSection

Просте блокування OpenMP Вкладене блокування OpenMP Win32-функція
omp_lock_t omp_nest_lock_t CRITICAL_SECTION
omp_init_lock omp_init_nest_lock InitializeCriticalSection
omp_destroy_lock omp_destroy_nest_lock DeleteCriticalSection
omp_set_lock omp_set_nest_lock EnterCriticalSection
omp_unset_lock omp_unset_nest_lock LeaveCriticalSection
omp_test_lock omp_test_nest_lock TryEnterCriticalSection

Для синхронізації коду можна використовувати і підпрограми виконуючого середовища, і директиви синхронізації. Перевага директив в тому, що вони прекрасно структуровані. Це робить їх більш зрозумілими і полегшує пошук місць входу в синхронізовані регіони і виходу з них. Перевага підпрограм виконуючого середовища - гнучкість. Наприклад, можливість передати блокування в іншу функцію і встановити / звільнити її в цій функції. При використанні директив це неможливо. Як правило, якщо потрібна гнучкість, що

забезпечується лише підпрограмами виконуючого середовища, краще використовувати директиви синхронізації.

В прикладі 5 показано код двох паралельно виконуваних циклів, на початку яких виконуючому середовищі невідома кількість їх ітерацій. У першому прикладі виконується перебір елементів STL-контейнера std:: vector, а в другому - стандартного зв’язаного списку.

Приклад 5. Виконання заздалегідь невідомого числа ітерацій

pragma omp parallel { / / Паралельна обробка вектора STL std:: vector <int>:: iterator iter; for (iter = xVect.begin (); iter! = xVect.end (); + + iter) { # pragma omp single nowait { process1 (* iter); } } / / Паралельна обробка стандартного пов'язаного списку for (LList * listWalk = listHead; listWalk! = NULL; listWalk = listWalk-> next) { # pragma omp single nowait { process2 (listWalk); } } }

У прикладі з вектором STL кожен потік з групи потоків виконує цикл for і має власний примірник ітератора, але при кожній ітерації лише один потік входить в блок single (така семантика директиви single). Всі дії, що гарантують одноразове виконання блоку single при кожній ітерації, бере на себе виконуюче середовище OpenMP. Такий спосіб виконання циклу пов'язаний зі значними витратами, тому він корисний, тільки якщо у функції process1 виконується багато роботи. У прикладі зі зв'язаним списком реалізована та ж логіка. Варто відзначити, що в прикладі з вектором STL ми можемо до входу в цикл визначити число його ітерацій за значенням std:: vector.size, що дозволяє привести цикл до канонічної форми для OpenMP:

pragma omp parallel for for (int i = 0; i <xVect.size (); + + i) process (xVect [i]);

Це суттєво зменшує витрати в період виконання, і саме такий підхід найкраще застосовувати для обробки масивів, векторів і будь-яких інших

контейнерів, елементи яких можна перебрати в циклі for, відповідно канонічній формі для OpenMP.

За умовчанням в OpenMP для планування паралельного виконання циклів for застосовується алгоритм, так званий статичним плануванням (static scheduling). Це означає, що всі потоки з групи виконують однакове число ітерацій циклу. Якщо n - число ітерацій циклу, а T - число потоків у групі, кожний потік виконає n / T ітерацій (якщо n не ділиться на T без залишку, нічого страшного). Однак OpenMP підтримує й інші механізми планування, оптимальні в різних ситуаціях: динамічне планування (dynamic scheduling), планування в період виконання (runtime scheduling) і кероване планування (guided scheduling). Щоб задати один з цих механізмів планування, використовують розділ schedule в директиві # pragma omp for або # pragma omp parallel for. Формат цього розділу виглядає так:

schedule(алгоритм планирования[, число итераций])

Ось приклади цих директив: # pragma omp parallel for schedule (dynamic, 15) for (int i = 0; i <100; + + i) ... # pragma omp parallel # pragma omp for schedule (guided)

При динамічному плануванні кожен потік виконує вказане число ітерацій. Якщо це число не задано, за замовчуванням воно дорівнює 1. Після того як потік завершить виконання заданих ітерацій, він переходить до наступного набору ітерацій. Так триває, доки не будуть пройдені всі ітерації. Останній набір ітерацій може бути менше, ніж спочатку заданий.

Завершивши виконання призначених ітерацій, потік запитує виконання іншого набору ітерацій, число яких визначається по щойно наведеній формулі. Таким чином, число ітерацій, що призначаються кожному потоку, з часом зменшується. Останній набір ітерацій може бути менше, ніж значення, обчислене за формулою.

Необхідно зазначити, що OpenMP - не панацея від всіх бід. Ця технологія орієнтована в першу чергу на розробників високопродуктивних обчислювальних систем і найбільш ефективна, якщо код включає багато циклів і працює з розділеними масивами даних. Створення як звичайних потоків, так і паралельних регіонів OpenMP має свою тривалість. Щоб застосування OpenMP стало вигідним, виграш у швидкості, що забезпечується паралельним регіоном, повинен перевершувати витрати на створення групи потоків. У версії OpenMP, реалізованої в Visual C + +, група потоків створюється при вході в перший паралельний регіон. Після завершення регіону група потоків призупиняється, поки не знадобиться знову. За лаштунками OpenMP використовує пул потоків Windows. Рис. 2.2 ілюструє приріст швидкодії простої програми, який досягається завдяки OpenMP на двопроцесорному комп'ютері при різній кількості ітерацій. Максимальний приріст швидкодії становить приблизно 1,7 % від вихідного, що типово для двопроцесорних систем.

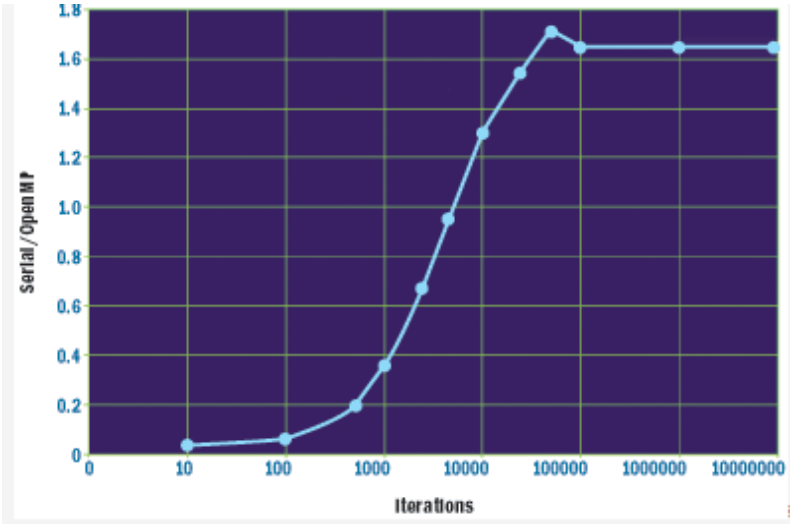

Рисунок 2.2 - Приріст швидкодії простої програми, який досягається завдяки OpenMP

На даному графіку вісь «y» представляє відношення часу послідовного виконання коду до часу паралельного виконання того ж коду. Паралельна версія наздоганяє за швидкодією послідовну приблизно при 5000 ітерацій, але це поганий сценарій. Більшість паралельних циклів будуть виконуватися швидше

послідовних навіть при значно меншій кількості ітерацій. Це залежить від обсягу роботи, виконуваної на кожній ітерації.

Як би там не було, цей графік показує, наскільки важливо оцінювати продуктивність ПЗ. Саме по собі застосування OpenMP не гарантує, що швидкодію вашого коду підвищиться.

На завершення відзначимо кілька моментів, серед яких варто особливо підкреслити два. По-перше, технологія спочатку спроектована таким чином, щоб користувач міг працювати з єдиним текстом для паралельної і послідовної програм. Справді, звичайний компілятор на послідовній машині директиви OpenMP просто “не помічає”. Єдиним джерелом проблем можуть стати змінні оточення і спеціальні функції, однак для них у специфікаціях стандарту передбачені спеціальні "заглушки", що гарантують коректну роботу OpenMP-програми в послідовному випадку, потрібно тільки перекомпілювати програму і підключити іншу бібліотеку. Іншою перевагою OpenMP є можливість поступового, "інкрементного" розпаралелювання програми. Взявши за основу послідовний код, користувач крок за кроком додає нові директиви, що описують нові паралельні секції. Немає необхідності відразу писати паралельну програму, її створення ведеться послідовно, що спрощує і процес програмування, і налагодження.

OpenMP - проста, але потужна технологія розпаралелювання програм. Вона дозволяє реалізувати паралельне виконання як циклів, так і функціональних блоків коду. Вона легко інтегрується в існуючі програми і включається / вимикається одним параметром компілятора. OpenMP дозволяє більш повно використовувати обчислювальну потужність багатоядерних процесорів.

Поради коли використовувати технологію OpenMP:

- Цільова платформа є багатопроцесорною або багатоядерною. Якщо програма повністю використовує ресурси одного ядра або процесора, то, зробивши її багатопотоковою за допомогою OpenMP, то це майже напевно підвищить швидкодію.

- Програма повина бути кросплатформенною. OpenMP - багатоплатформовий і широко підтримуваний API. А так як він реалізований на основі директив pragma, програму можна скомпілювати навіть за допомогою компілятора, який не підтримує стандарт OpenMP.

- Виконання циклів потрібно розпаралелить. Весь свій потенціал OpenMP демонструє при організації паралельного виконання циклів. Якщо в програмі є тривалі цикли без залежностей, OpenMP - ідеальне рішення.

- Перед випуском програми потрібно підвищити її швидкодію. Оскільки технологія OpenMP не вимагає переробки архітектури програми, вона чудово підходить для внесення в код невеликих змін, що дозволяють підвищити швидкодію.

2.1.3 Емуляція тестових сегментів мережі

У рамках першого розділу було розглянуто різні інструменти для емуляції сегментів комп'ютерних мереж, в ході аналізу було виявлено, що інструмент Mininet є чудовим варіантом для тестування алгоритму побудови маршрутів.

Mininet дозволяє конфігурувати топології за допомогою скрипту на мові Python. Це дозволяє генерувати сегменти з випадковими зв'язками будь-якої складності та розмірності.

Насамперед необхідно зробити імпорт бібліотек для коректної роботи засобу емуляції. На лістингу 2.17 продемонстровано імпорт необхідних бібліотек.

Лістинг 2.17 - Імпорт бібліотек.

```
from mininet.net import Mininet
from mininet.node import NOX, OVSSwitch, Controller, RemoteController
from mininet.topo import Topo
from mininet.log import setLogLevel
from mininet.cli import CLI
```

Клас з назвою Mininet, що знаходиться в просторі імен mininet.net, використовується для емуляції мережі з логічними комп'ютерами.

Бібліотека mininet.node зберігає опис класів, які відповідають за контролери та комутатори.

Бібліотека mininet.topo використовується для коректної роботи топології сегмента мережі. До завдань цієї бібліотеки входить: зберігання інформації про вузли, створення/видалення/модифікація підключень між вузлами.

Бібліотека mininet.log є допоміжною бібліотекою за допомогою якої здійснюється логування налагоджувальної інформації, виведення тих чи інших помилок. Це дуже корисна бібліотека на етапі налагодження та тестування, т.к. допомагає виявляти проблеми практично одразу.

Бібліотека mininet.cli необхідна для можливості конфігурування та виконання різних команд вже після запуску засобу та емулювання, за допомогою командного рядка.

Створення підключення до віддаленого ПКК RUNOS продемонстровано у лістингу 2.18.

Лістинг 2.18 - Створення підключення до ПКК RUNOS

```
runos = RemoteController('c0', ip='127.0.0.1')
```

Для створення власної топології необхідно успадкувати клас Topo, і після цього можна буде модифікувати його роботу. Реалізація топології користувача продемонстровано в лістингу 2.19.

Лістинг 2.19 - Створення топології користувача.

```
class Deikstra Topo (Topo):
    def__init__(self):
        Topo.__init__(self)
```

Після створення топології необхідно створити та зробити запуск мережі, для цього необхідно додати всі логічні комутатори в мережу, а після з'єднати їх у

випадковому порядку для коректної роботи мережі. У лістингу 2.20 продемонстровано генерацію логічних комутаторів.

Лістинг 2.20 - Генерація логічних комутаторів.

```
RemoteController = self.addSwitch('s0')
i = 0
while i < topoSize:
    NewSwitch = self.addSwitch('deikstraSwitch' + str(i))
    i = i + 1
    self.addLink( RemoteController, NewSwitch)
i = 0
while i < topoSize:
    Switch = self.getSwitch('deikstraSwitch' + str(i))
    j = 0
    while j < 20:
    ForConnect = self.getSwitch('deikstraSwitch' + str(random.randInt(0,
topoSize))
    self.addLink( Switch, ForConnect)
    j = j + 1
```

Після генерації топології необхідно зробити створення та запуск екземпляра Mininet. Для запуску мережі необхідно створити екземпляр топології DeikstraTopo. Наступним кроком буде передача необхідних параметрів для створення екземпляра мережі. Наприкінці необхідно буде зробити побудову і запуск мережі. Після запуску необхідно викликати консольну оболонку CLI, яка прийматиме різні команди від користувача, за допомогою яких під час виконання можна вносити різні зміни в роботу створеної мережі.

Створення та запуск представлені на лістингу 2.21.

Лістинг 2.21 - Створення та запуск екземпляра мережі Mininet.

```
topo = DeikstraTopo()
net = Mininet( topo=topo, switch=OVSSwitch, build=False )
```

```
net.build()
net.start()
CLI( net )
net.stop()
```

Включення Mininet здійснюється за допомогою спеціальної консольної команди, представленої на лістингу 2.22. У вхідних параметрах ми вказуємо, де знаходиться файл скрипта для запуску і яку топологію використовувати.

Лістинг 2.22. Команда для запуску Mininet.

```
sudo mn --custom ~/mininet/custom/mydiplom.com --topo mydiplom
```

3 ТЕСТУВАННЯ ТА АПРОБАЦІЯ

Після розробки алгоритму для програмно-конфігурованої мережі на базі мережної операційної системи RUNOS, була проведено емуляцію різних складних сегментів мережі, для визначення ступеня досягнення поставлених цілей та завдань.

Загалом було проведено 2 групи тестів. На основі тестів були отримані дані про продуктивність існуючого, паралельного реалізованого алгоритму та однопотокового алгоритму. Було складено діаграми продуктивності для більш зрозумілого відображення результату виконаної роботи. На цих діаграмах можна побачити відмінність за швидкістю виконання побудови маршруту на різних розмірах сегментів програмно-конфігурованих мереж.

Метою тестів першого етапу було виявити, наскільки ефективними є різні реалізації алгоритму Дейкстри, на базі різних структур даних та оптимізації. Отримані дані відображають сильні та слабкі сторони того чи іншого підходу. Схема проведення тестів була різною, перша група тестів була орієнтована на дослідження часу генерації з’єднань. Для дослідження часу розрахунку маршруту на тестовому сегменті кожні 100 мс створювався запит на передачу даних із одного випадкового комутатора до іншого. Кожен тест цієї групи проводився протягом 24 годин. Між тестами змінювався лише розмір сегмента та реалізація алгоритму. Результати тестування представлено на рис. 3.1.

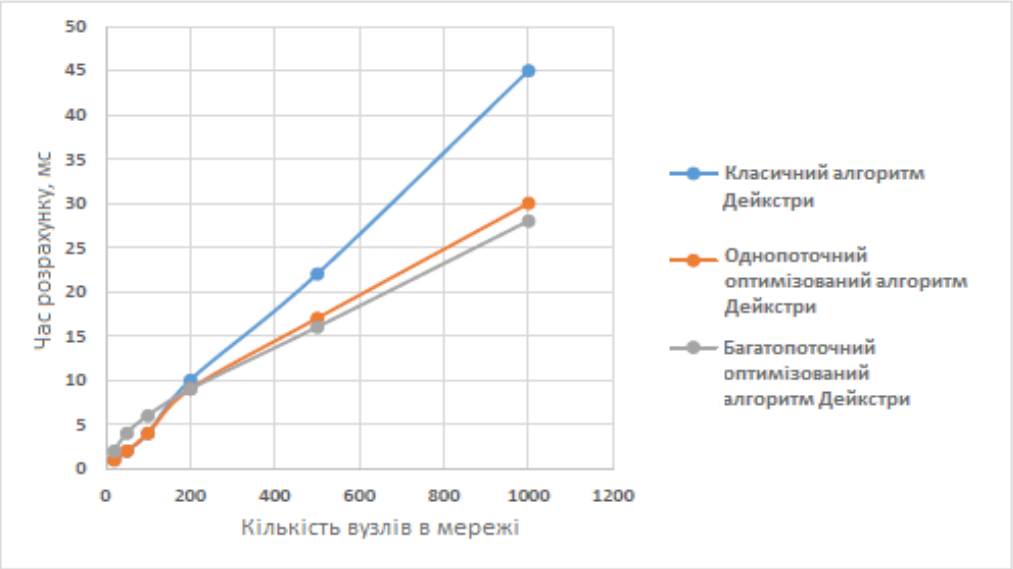

Рисунок 3.1 - Результат першого етапу тестування

Як видно в рамках результату тестування, при збільшенні кількості вузлів у мережі графік зростає не лінійно для класичної реалізації та однопоточної оптимізованої реалізації. Однак реалізація багатопоточного оптимізованого алгоритму Дейкстри за часом виконання трохи програє на невеликих сегментах мережі через збільшені витрати на створення потоків для роботи в багатопоточному режимі. Як можна бачити, на сегменті розмірів в 100 вузлів, оптимізований алгоритм Дейкстри працює трохи швидше ніж класична реалізація, використана в мережній ОС RUNOS. Починаючи з 200 вузлів, багатопотокова версія оптимізованої реалізації починає працювати швидше, т.к. кількість оброблюваних даних збільшується і разом із цим зростає ефективність роботи паралельної версії алгоритму.

Другий етап тестів був націлений на тестування продуктивності програмно-конфігурованої мережі за умов передачі дрібних файлів розміром 1 - 1024 байта. Результати другого етапу тестування представлено на рис.3.2.

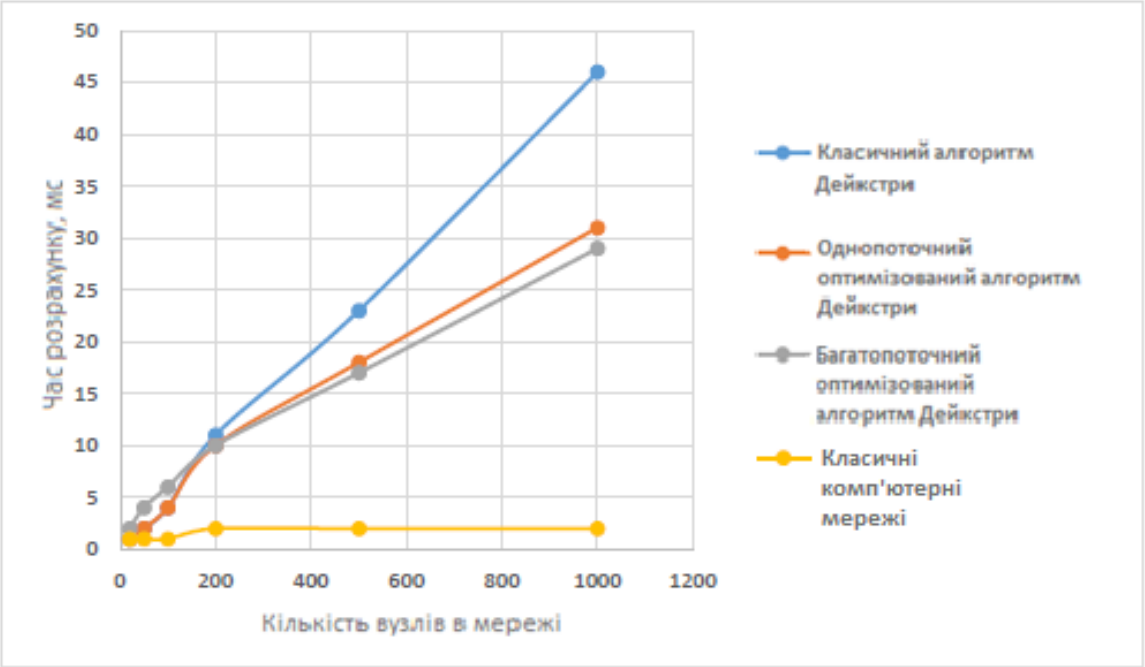

Рисунок 3.2 - Результати другого етапу тестування

Варто звернути увагу, що в рамках даного тесту порівнювалася ефективність роботи не тільки між різними реалізаціями алгоритмів, але ще й між класичними мережами. Як бачимо, на невеликих обсягах даних програмно-конфігуровані мережі не є дуже ефективним рішенням. Також варто зауважити, що будь-який створений шлях у ПКМ зберігається деякий час і якщо відбувається повторний запит на відправлення в те саме місце, то шлях не генерується повторно.

ВИСНОВКИ

Метою магістерської роботи є дослідження алгоритмів побудови маршрутів, розробка та оптимізація їх швидкості виконання для програмно-конфігурованих мереж з метою підвищення ефективності та продуктивності. Для досягнення зазначеної мети перед роботою було поставлено низку завдань.

У роботі було вивчено проблему, важливість та актуальність проблем побудови маршрутів у програмно-конфігурованих мережах. У ході вивчення було встановлено, що таке завдання є проблемним і його необхідно вирішувати.

Було проведено докладний аналіз різних середовищ для розробки програмного забезпечення, мережевих операційних систем для ПКМ, різних систем емуляції. В результаті детального аналізу було виявлено, що для використання в роботі підходить середовище розробки Eclipse CDT, мережна ОС RUNOS та емулятор Mininet.

Було проведено розробку оптимізованого алгоритму Дейкстри, за допомогою спрощених структур зберігання та усічення зайвих кроків алгоритму в рамках ПКМ. Після цього було здійснено налаштування емулятора Mininet та налаштування RUNOS для взаємодії з емульованим сегментом мережі, для налагодження та тестування коректності роботи.

Були проведені дослідження працездатності алгоритму на основі емульованих сегментів мережі різного розміру, для виявлення переваг і недоліків розробленого оптимізованого алгоритму.

ПЕРЕЛІК ПОСИЛАНЬ

1. Agouros K., Software Defined Networking / K. Agouros - Берлін: De 1. Gruyter, 2016. – 268 с.

2. Azodolmolky S., Software Defined Networking with OpenFlow - Second Edition / S. Azodolmolky – Бірмінгем: Packt Publishing, 2017. – 312 с.

3. Chou E., Mastering Python Networking: Your one stop solution to using Python for network automation, DevOps, and SDN / E. Chou - Бірмінгем: Packt Publishing, 2020. - 446 с.

4. Cisco, Cisco Network Simulator, Router Simulator & Switch Simulator [Електронний ресурс], режим доступу http://www.boson.com/netsim-cisco-network-simulator;

5. Doherty J., SDN і NFV Simplified: A Visual Guide до Understanding Software Defined Networks and Network Function Virtualization / J. Doherty - Бостон: Addison-Wesley Professional, 2021. - 320 с.

6. Duan Q., Virtualized Software-Defined Networks and Services / Q. Duan, Toy M. – Норвуд: Arttech House, 2019. – 334 с.

7. Duan Q., Network як Service для Next Generation Internet (Telecommunications) Q. Duan, Wang S. - Стівенідж: The Institution of Engineering and Technology, 2017. - 440 с.

8. Dumka A., Innovations in Software-Defined Networking and Network 8. Functions Virtualization (Advances in Systems Analysis, Software Engineering, and High Performance Computing) / A. Dumka - Херши: IGI Global, 2018. - 364 с.

9. Goransson P., Software Defined Networks, A Comprehensive Approach/P. Goransson - Нью-Йорк: Morgan Kaufmann, 2022 - 436 с.

10. Hamburger V., Building VMware Software-Defined Data Centers / V. Hamburger - Бірмінгем: Packt Publishing, 2021. - 432 с.

11. Katti M. Learn About Software-Defined Secure Networks (SDSN)/M. Katti - Саннивейл: Juniper Networks Books, 2016. - 95 с.

12. Khondoker R. SDN i NFV Security: Security Analysis of Software - Defined Networking and Network Function Virtualization (Lecture Notes in Networks and Systems) / R. Knodoker - Берлін: Springer, 2018. - 134 с.

13. Liyanage M., Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture (Wiley Series on Communications Networking & Distributed Systems) / M. Liyanage – Хобокен: Wiley, 2020. – 438 с.

14. Mininet Community, Mininet Walkthrough [Електронний ресурс], режим доступу - http://mininet.org/walkthrough/.

15. NS Community, WHAT IS NS-3 [Електронний ресурс], режим доступу https://www.nsnam.org/overview/what-is-ns-3/.

16. OPNET Community, OPNET [Електронний ресурс], режим доступу - https://sandilands.info/sgordon/teaching/resources/opnet.html.

17. Pujolle G., Software Networks. Virtualization, SDN, 5G, Security/G. Pujolle - Хобокен: John Wiley & Sons Limited, 2022. - 260 с.

18. POX Community, Using the POX SDN controller [електронний ресурс], режим доступу http://www.brianlinkletter.com/using-the-pox-sdn-controller/.

19. Qi H., Software Defined Networking Applications in Distributed Datacenters (SpringerBriefs в Electrical and Computer Engineering) / H. Qi - Берлін: Springer, 2019 - 68 с.

20. Robertazzi T.G., Introduction to Computer Networking / T.G. Reobertazzi - Берлін: Springer, 2017. - 154 с.

21. Shukla V., Introduction to Software Defined Networking - OpenFlow & VxLAN / V. Shukla Сиєтл: CreateSpace Independent Publishing Platform, 2018.-114 с.

22. Subramanian S., Software Defined Networking (SDN) with OpenStack/S. Subramanian – Бірмінгем: Packt Publishing, 2022. – 216 с.

23. Stallings W., Foundations of Modern Networking: SDN, NFV, QoE, IoT, Cloud / W. Stallings - Бостон: Addison-Wesley Professional, 2022. - 544 с.

24. Zhang Y. Network Function Virtualization: Concepts and Applicability in 5G Networks (Wiley - IEEE) / Y. Zhang - Нью-Йорк: Wiley-IEEE Press, 2017. 179 с.

ДОДАТОК 1

```
#pragma once
#include "ILinkDiscovery.hh"
#include "exception.hh"
#include "stdint.h"
#include "Topology.hh"
class GraphArray {

	public:
		GraphArray() {
			this->count = 1000;
			switchIdCounter = 0;
			initGraph Data();
		};
		~Graph()
		{
			destroyGraphData();
		}
		void addLink(switch_and_port from, switch_and_port to);
		void removeLink(switch_and_port from, switch_and_port to);
		void incrementFlows(switch_and_port fromSwitch,
switch_and_port toSwitch);
		void decrimentFlows(switch_and_port fromSwitch,
switch_and_port toSwitch);

		data_link_route computeRoute(uint64_t from_dpid, uint64_t
to_dpid);
```

```
private:
        uint32_t count;
        int32_t**graphData;
        bool *visited;
        int32_t switchIdCounter;
        std::map<uint64_t, uint32_t>dpidToGraphId;
        std::map<uint32_t, uint64_t> graphIdToDpid;
        std::map<uint32_t, switch_and_port> graphIdToSwitch;

        void initGraphData();
        void destroyGraphData();
        void setMinVals(int32_t fromId);
        void setMinVaslParallel(int32_t fromId);
        int32_t findNextMinId(bool *visited);
        data_link_route restoreRoute(int32_t fromId, int32_t endId);
        int32_t findPrevId(int32_t fromId);
};
```

ДОДАТОК 2

```
#include "GraphArray.hh"

void GraphArray::addLink(switch_and_port from, switch_and_port to)
{
        int fromId, toId;
        map::iterator it = dpidToGraphId.find(from.dpid);
        if (it != dpidToGraphId.end()) {
                fromId = it->second;
        }
        else {
                dpidToGraphId[from.dpid] = switchIdCounter++;
                fromId = dpidToGraphId[from.dpid];
        }
        map::iterator it = dpidToGraphId.find(to.dpid);
        if (it != dpidToGraphId.end()) {
                toId=it->second;
        }
        else {
                dpidToGraphId[to.dpid] = switchIdCounter++;
                toId = dpidToGraphId[to.dpid];
        }
        graphIdToSwitch[fromId] = from;
        graphIdToSwotch[toId] = to;
        graphData[fromId][told] = 1;
        graphData[toId][fromId] = 1;
}
void GraphArray::removeLink(switch_and_port from, switch_and_port to)
{
```

```
        int fromId, toId;
        map::iterator it = dpidToGraphId.find(from.dpid);
        if (it != dpidToGraphId.end()) {
            fromId = it->second;
        }
        else {
            dpidToGraphId[from.dpid] = switchIdCounter++;
            fromId = dpidToGraphId[from.dpid];
        }
        map::iterator it = dpidToGraphId.find(to.dpid);
        if (it != dpidToGraphId.end()) {
            toId = it->second;
        }
        else {
            dpidToGraphId[to.dpid] = switchIdCounter++;
            toId = dpidToGraphId[to.dpid];
        }
        graphData[fromId][toId] = -1;
        graphData[toId][fromId] = -1;
    }

    void GraphArray::incrementFlows(switch_and_port fromSwitch,
switch_and_port toSwitch)
    {
        int fromId, toId;
        map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
        map::iterator toIt = dpidToGraphId.find(toSwitch.dpid);
        if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
            fromId = fromIt->second;
            toId = toIt->second;
```

```
                graphData [fromId][toId]++;
                graphData [toId][fromId]++;
            }
        }

        void GraphArray::decrimentFlows(switch_and_port fromSwitch,
switch_and_port toSwitch)
        {
            int fromId, toId;
            map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
            map::iterator toIt = dpidToGraphId.find(toSwitch.dpid);
            if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
                fromId = fromIt->second;
                toId = toIt->second;
                graphData[fromId][toId]--;
                graphData[toId][fromId]--;
            }
        }
         data_link_route GraphArray::computeRoute(uint64_t from_dpid, uint64_t
to_dpid)
        {
            for (int i=0; i < count; i++) {
                visited = false;
                graphData[i][i] = -1;
            }
            int fromId, toId;
            fromId=dpidToGraphId[from.dpid];
            toId-dpid ToGraphId[to.dpid];
            if (fromId == toId) {
                return route;
```

```
        }
        visited[fromId] = true;
        graphData[fromId][fromId] = 0;
        setMinVals(fromId);
        int32_t nextMinId = findNextMinId(visited);
        while (nextMinId != toId) {
                setMinVals(nextMinId);
                visited[nextMinId] = true;
                nextMinId = findNextMinId(nextMinId);
        }
        return restoreRoute(nextMinId);
}


void GraphArray::initGraphData()
{
        graphData = new *int32_t[count];
        for (int32_t i = 0; i < count; i++) {
                graphData[i] = new int32_t[count];
                for (int j = 0; j < count; j ++) {
                        graphData[i][j] = -1;
                }
        }
        visited = new bool[count];
}
void GraphArray::destroyGraphData()
{
        for (int i=0; i < count; i++) {
                delete[] graphData[i];
         }
```

```
            delete [] graphData;
            delete[] visited;
        }

        void GraphArray::setMinVals(int32_t fromId)
        {
            int32_t startValue = graphData[fromId][fromId];
            for (int i = 0; i < count; i ++) {
                int32_t flowsCount = graphData[fromId][i];
                if (flowsCount == -1) {
                    continue;
                }
                 int32_t currentValue = graphData[i][i];
                 if (currentValue == -1 || currentValue > startValue + flowsCount)
{
                        graphData[i][i] = startValue + flowsCount;
                 }
            }
        }
        void GraphArray::setMinVaslParallel(int32_t fromId)
        {
            int32_t startValue = graphData[fromId][fromId];
        #pragma omp parallel for num_threads(4)
            for (int i =  omp_get_thread_num(); i < count; i +=
omp_get_max_threads()) {
                int32_t flowsCount = graphData[fromId][i];
                if (flowsCount == -1) {
                    continue;
                }
                int32_t current Value = graphData[i][i];
```

```
                if (current Value ==-1 || currentValue > startValue + flowsCount)
{
                        graphData[i][i] = startValue + flowsCount;
                }
        }
}

 int32_t GraphArray::findNextMinId(bool * visited)
{
        int32_t minVal = -1;
        int32_t minId = -1;
        for (int32_t i = 0; i < count; i++) {
                if (!visited[i]) {
                        currentVal = graphData[i][i];
                        if (minVal = -1 || minVal > currentVal) {
                                minVal = currentVal;
                                minId = i;
                        }
                }
        }
        return minId;
}
data_link_route GraphArray::restoreRoute(int32_t fromId, int32_t endId)
{
        data_link_route result;
        int32_t prevId=endId;
        do {
                result.insert(0, graphIdToSwitch[prevId]);
                prevId = findPrevId();
        } while (prevId! = -1);
```

```
        return result;
}

int32_t GraphArray::findPrevId(int32_t fromId)
{
        if (graphData[fromId][fromId] == 0) {
                return -1;
        }
        for (int32_t i = 0; i < count; i++) {
                 if (graphData[fromId][i] < 0) {
                        continue;
                 }
                  if (graphData[i][i == graphData[fromId][fromId]] –
graphData[fromId][i]) {
                                return i;
                        }
                }
                return -1;
       }
```

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ