Національний технічний університет України
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Тенденції розвитку сучасних комп'ютерних систем

МЕТОДИЧНІ ВКАЗІВКИ
до практичних робіт для аспірантів спеціальності
123 - Комп'ютерна інженерія

Рекомендовано Вченою радою факультету інформатики та обчислювальної техніки

Київ КПІ ім. Ігоря Сікорського

2022

Методичні вказівки до практичних занять з дисципліни «Тенденції розвитку СУЧАСНИХ комп'ютерних систем»

Ефективною формою організації навчання у вищій школі є семінарські і практичні заняття, з якими органічно поєднуються лекції. Ці заняття передбачають самостійне опрацювання студентами окремих тем і проблем у відповідності до змісту навчальної дисципліни і обговорення результатів цього вивчення, представлених у вигляді тез, повідомлень, доповідей, рефератів і т.д. Проведення семінарських занять дозволяє вирішувати наступні дидактичні цілі:

- оптимально поєднувати лекційні заняття з систематичної самостійної навчально-пізнавальною діяльністю студентів, їх теоретичну підготовку з практичною;

- розвивати вміння, навички самостійної творчої роботи, творчого мислення, вміння використовувати теоретичні знання для вирішення практичних завдань;

- формувати у студентів інтерес до науково-дослідницької роботи і залучення до наукових досліджень, які проводить кафедра;

- забезпечувати системне повторення, поглиблення і закріплення знань студентів з певної теми;

- формувати вміння і навички здійснення різних видів майбутньої професійної діяльності;

- здійснювати діагностику і контроль знань студентів з окремих розділів і тем програми, формувати вміння і навички виконання різних видів майбутньої професійної діяльності.

Цілі практичних занять:

Практична робота полягає у виконанні студентами під керівництвом викладача індивідуальних завдань за темами, передбачених робочою програмою. Крім того, одним з важливих компонентів навчання є розвиток творчої фантазії в пошуку нових ідей, які в даному випадку пов'язані з новими підходами до подальшого суттєвого підвищення продуктивності обчислювальних систем і припускають:

- вивчення основних напрямків розвитку існуючих підходів до побудови надвисокопродуктивних обчислювальних систем;

- вивчення нових технологій до побудови надвисокопродуктивних обчислювальних систем;

- вивчення нових методів організації обчислювальних процесів в надвисокопродуктивних обчислювальних системах;

- вивчення питань проектування надвисокопродуктивних обчислювальних систем;

В результаті практичного вивчення дисципліни «Тенденції розвитку сучасних комп'ютерніх систем» студенти повинні набути професійних навикіовладенія методами практичної роботи із застосуванням різноманітних інструментів, органиации і технологій.

ВСТУП

Можливості суттєвого підвищення продуктивності обчислювальних систем на основі розвитку частотних властивостей елементної бази досягла своєї межі. Тільки паралельна обробка інформації створює передумови для суттєвого підвищення продуктивності засобів обчислювальной техніки. Тому останні десятиріччя пов’язані з швидким розвитком паралельних комп’ютерних систем (ПКС). Зараз у світі існує велика кількість архітектурних рішень ПКС і потреба у їх використанні весь час зростає. Але всі ці архітектурні рішення базуються на просторовому (кількісному) розширенні систем, що вже зараз призводить до гігантоманії при їх проектуванні і, відповідно, до появи нових суттєвих проблем. Так, лідери сьогодення в плані продуктивності включають в себе десятки мільонів ядер. Споживана потужність для функціонування такої системи еквівалентна потужності, необхідної для функціонування невеликого міста з заводами і фабриками. Подальше підвищення продуктивності буде призводити до наступного підвищення споживаної потужності. І таких проблем можно нарахувати багато. В зв’язку з цим сьогодні однією з найактуальніших проблем розвитку ПКС є пошук нових методів підвищення продуктивності обчислювальних систем та організації обчислювальних процессів. У рамках дисципліни “Тенденнції розвитку сучасних комп’ютерних систем” розглядаються сучасні основні напрямки підвищення продуктивності ПКС, які пов’язані з новими технологіями побудови комп’ютерних систем, з розвитком існуючих технологій і нових методів

організаціїї паралельних обчислювальних процесів, впровадженням еволюційних обчислень. Таким чином, метою даної дисципліни є вивчення нових підходів до суттєвого підвищення ефективної продуктивності комп'ютерних систем, які дозволяють перейти від кількісного метода підвищення продуктивності до якісного на основі нових технологічних і організаціонних підходів.

Зміст навчальної дисципліни

Розділ 1. Нові принципи організації комп'ютерних систем і обчислювальних процесів масового розпаралелювання.

Тема 1.1. Загальні вимоги, що пред'являються до сучасних комп'ютерних систем.

Тема 1.2 Огляд нових принципів організації комп'ютерних систем масового розпаралелівання.

Тема 1.3. Огляд нових принципів організації обчислювальних процесів з можливостю реалізації масового розпаралелівання.

Тема 1.4. Віртуалізація і віртуальні машини.

Розділ 2. Розвиток методів обробки інформації на основі управління потоком даних.

Тема 2.1. Основні вимоги до систем, які управляються потоком даних.

Тема 2.2. Скалярна і векторна організації асоціативної памяті в системах, які управляються потоком даних.

Тема 2.3. Оптична асоціативна пам'ять.

Тема 2.4. Архітектура екзафлопсного суперкомп'ютера.

Тема 2.5. Біологічні мережі.

Розділ 3. Основи еволюційних обчислень.

Тема 3.1. Основи генетичних алгоритмів.

Тема 3.2. Теорія схем і моделі генетичних алгоритмів.

Тема 3.3. Модифікації і узагальнення генетичних алгоритмів.

Тема 3.4. Паралельні генетичні алгоритми.

Тема 3.5. Генетичне програмування.

Розділ 4. Фізика квантової інформації.

Тема 4.1 Кубіт.

Тема 4.2. Принцип суперпозіції.

Тема 4.3. Визначення та приклади.

Тема 4.4. Клонування станів.

Тема 4.5. Електромагнітна хвиля і фотон.

Розділ 5. Квантовий світ проти класичного.
Тема 5.1. Переплутані стани.
Тема 5.2. Телепортація квантових станів.
Тема 5.3. Квантова криптографія.
Тема 5.4. Квантове вимірювання.

Розділ 6. Принципи квантових обчислень.
Тема 6.1. Загальні принципи побудови.
Тема 6.2. Кодування, одно та дво-кубітні оперптори.
Тема 6.3. Структура квантового алгоритму.
Тема 6.4. Алгоритм Шора.
Тема 6.5. Алгоритм Гровера.
Тема 6.6. Швидке квантове перетворення Фур'є.
Тема 6.7. Декогеренція.

Реквізити навчальної дисципліни	
Рівень вищої освіти	***Перший (бакалаврський)***
Галузь знань	*12 Інформаційні технології*
Спеціальність	*123 Комп'ютерна інженерія*
Освітня програма	*Комп'ютерні системи та мережі*
Статус дисципліни	*Нормативна*
Форма навчання	*очна(денна)*
Рік підготовки, семестр	*4 курс, осінній семестр*
Обсяг дисципліни	*6 кредитів*
Семестровий контроль/ контрольні заходи	*Екзамен*
Розклад занять	*Лекцій - 54 годин Лабораторних - 18 годин*
Мова викладання	*Українська*
Інформація про керівника курсу / викладачів	Лектор: *науковий ступінь, вчене звання, Луцький Г.М., контактні дані* Лабораторні: науковий ступінь, вчене звання, Русанова О.В., контактні дані
Розміщення курсу	Посилання на дистанційний ресурс (Moodle, Google classroom, тощо)

Програма навчальної дисципліни

1. Опис навчальної дисципліни, її мета, предмет вивчання та результати навчання

Викладач обґрунтовує необхідність вивчення навчальної дисципліни, відповідаючи на питання «Чому майбутньому фахівцю варто вчити саме цю дисципліну?», визначає ***мету***, ***предмет*** *дисципліни та* ***програмні результати навчання*** *(компетентності, знання, уміння, навички, досвід, послідовність дій в стандартних виробничих ситуаціях тощо), які студент/аспірант набуде після вивчення дисципліни з розподілом на окремі освітні компоненти (якщо дисципліна вивчається декілька семестрів).*

2. Пререквізити та постреквізити дисципліни (місце в структурно-логічній схемі навчання за відповідною освітньою програмою)

Зазначається перелік дисциплін, або знань та умінь, володіння якими необхідні студенту (вимоги до рівня підготовки) для успішного засвоєння дисципліни (наприклад, «базовий рівень володіння англійською мовою не нижче А2»). Вказується перелік дисциплін які базуються на результатах навчання з даної дисципліни.

Необхідні дисципліни: “Програмування”, “Обєктно-орієнтоване програмування”, “Системне програмування”, “Структури даних та алгоритми”, “Інженерія програмного забезпечення”, “Алгоритми та методи обчислень”

Дисципліни, які базуються на результатах навчання з даної дисципліни: “Організація обчислювальних процесів”, “Комп’ютерні системи”

3. Зміст навчальної дисципліни

Загальні вимоги, что пред'являються до СУЧАСНИХ комп'ютерних систем.

Співвідношення вартість / продуктивність

При розробці сучасних засобів обчислювальної техніки у відповідності з вимогами ринку переслідуються різні першорядні цілі. Так одним з варіантів таких цілей може служити низька вартість, при якому питання продуктивності практично нівелюються. Іншим варіантом першорядних цілей є надвисока продуктивність, при якому практично нівелюються питання вартості. Між цими двома крайніми варіантами розробок можна виділити такий проміжний варіант, в рамках якого вирішуються оптимізаційні задачі, пов'язані з пошуком балансу між вартісними параметрами і продуктивністю. Оскільки інте-гральним показником рівня розвитку обчислювальної техніки є продуктивність, то в даному курсі основна увага буде приділена другому варіанту розробок, а точніше методам і засобам подальшого істотного підвищення продуктивності засобів обчислювальної техніки.

Надійність і відмовостійкість

Найважливішими характеристиками обчислювальних систем є **надійність** і **відмовостійкість**. Підвищення надійності засноване на принципі запобігання несправності шляхом зниження інтенсивності відмов і збоїв за рахунок застосування електронних схем і компонентів з високим і надвисоким ступенями інтеграції, зниження рівня перешкод, використання полегшених режимів роботи схем, забезпечення теплових режимів їх роботи, а також за рахунок вдосконалення методів збірки апаратури [ ]. Відмовостійкість - це така властивість системи, яка забезпечує їй можливість коректної роботи при виникненні несправностей. Відмовостійкість передбачає наявність надлишкового апаратного і програмного забезпечення. Напрями, пов'язані з попередженням несправностей і з відмовостійкістю, - основні в проблемі надійності. Концепції паралельності і відмовостійкості обчислювальних систем природним чином пов'язані між собою, оскільки в обох випадках потрібні

додаткові функціональни компоненти. Тому, власне, на паралельних обчислювальних системах досягається як найвища продуктивність, так і, в багатьох випадках, дуже висока надійність. Наявні ресурси надмірності в паралельних системах можуть гнучко використовуватись як для підвищення продуктивності, так і для підвищення надійності. Структура багатопроцесорних і багатомашинних систем пристосована до автоматичної реконфігурації і забезпечує можливість продовження роботи системи після виникнення несправностей.
Слід пам'ятати, що поняття надійності включає не лише апаратні засоби, а ле і програмне забезпечення. Головною метою підвищення надійності сист ем є цілісність даних, що збері-гаються в них.

Масштабованість. Масштабованість є можливість нарощування числа і п отужності процесорів, об'ємів оперативної і зовнішньої пам'яті і інших рес урсів обчислювальної системи. Масштабованість повинна забезпечуватися архітектурою і конструкцією комп'ютера, а також відповідними засобами програмного забезпечення. Додавання кожного нового процесора в дійсно масштабованій системі повинне давати прогнозоване збільшення продукти вності і пропускній спроможності при прийнятних витратах. Одним з осно вних завдань при побудові масштабованих систем є мінімізація вартості ро зширення комп'ютера і спрощення планування. У ідеалі додавання процес орів до системи повинне призводити до лінійного зростання її продуктивн ості. Проте це не завжди так. Втрати продуктивності можуть виникати, наприклад, при недостатній пропускній спроможності шин через зростання трафіку між процесорами і основною пам'яттю, а також між пам'яттю і пристроями введення / виводу. Насправді реальне збільшення продуктивності важко оцінити заздалегідь, оскільки воно в значній мірі залежить від динаміки поведінки прикладних задач. Можливість масштабування системи визначається не тільки архітектурою апаратних засобів, але залежить від закладених властивостей програмного

забезпечення. Масштабованість програмного забезпечення впливає на кожен системний рівень обробки інформації від простих механізмів передачі повідомлень до роботи з такими складними об'єктами як монітори транзакцій і все середовище прикладної системи. Зокрема, програмне забезпечення повинно мінімізувати трафік межпроцессорного обміну, який може перешкоджати лінійному зростанню продуктивності системи. Апаратні засоби (процесори, шини, комутаційни елементи і пристрої вводу / виводу) є тільки частиною масштабованої архітектури, на якій програмне забезпечення може забезпечити передбачене зростання продуктивності. Важливо розуміти, що простий перехід, наприклад, на більш потужний процесор може призвести до перевантаження інших компонентів системи. Це означає, що дійсно масштабована система повинна бути збалансована за всіма параметрами.

Сумісність і мобільність програмного забезпечення. Концепція програмної сумісності вперше в широких масштабах була застосована розробниками системи IBM / 360. Основне завдання при проектуванні всього ряду моделей цієї системи полягала в створенні такої архітектури, яка була б однаковою з точки зору користувача для всіх моделей системи незалежно від ціни і продуктивності кожної з них. Величезні переваги такого підходу, що дозволяє зберігати існуючий заділ програмного забезпечення під час переходу на нові (як правило, більш продуктивні) моделі були швидко оцінені як виробниками комп'ютерів, так і користувачами і починаючи з цього часу практично всі фірми-постачальники комп'ютерного устаткування взяли на озброєння ці принципи , поставляючи серії сумісних комп'ютерів.

Перехід від однорідних мереж програмно сумісних комп'ютерів до побудови неоднорідних мереж, що включають комп'ютери різних фірм-виробників, в корені змінив і точку зору на саму мережу: з порівняно простого засобу обміну інформацією вона перетворилася в засіб інтеграції

окремих ресурсів - потужну розподілену обчислювальну систему, кожен елемент якої (сервер або робоча станція) найкраще відповідає вимогам конкретної прикладної задачі. Цей перехід висунув ряд нових вимог. Перш за все така обчислювальне середовище повинне дозволяти гнучко змінювати кількість і склад апаратних засобів і програмного забезпечення відповідно до нових вимог вирішуваних завдань. По-друге, вона повинна забезпечувати можливість запуску одних і тих же програмних систем на різних апаратних платформах, тобто забезпечувати мобільність програмного забезпечення.

В умовах жорсткої конкуренції виробників апаратних платформ та програмного забезпечення сформувалася концепція відкритих систем, що представляє собою сукупність стандартів на різні компоненти обчислювальної середовища, призначених для забезпечення мобільності програмних засобів в рамках неоднорідною, розподіленої обчислювальної системи. Одним з варіантів моделей відкритої середовища є модель OSE (Open System Environment), запропонована комітетом IEEE POSIX. На основі цієї моделі національний інститут стандартів і технології США випустив документ "Application Portability Profile (APP). The US Government's Open System Environment Profile OSE / 1 Version 2.0", який визначає рекомендовані для федеральних установ США специфікації в області інформаційних технологій, що забезпечують мобільність системного і прикладного програмного забезпечення. Всі провідні виробники комп'ютерів і програмного забезпечення в США в даний час дотримуються вимог цього документа.

Двома основними проблемами побудови обчислювальних систем для критично важливих додатків, пов'язаних з обробкою транзакцій, управлінням базами даних і обслуговуванням телекомунікацій, є забезпечення високої продуктивності і тривалого функціонування систем. Найбільш ефективний спосіб досягнення заданого рівня продуктивності -

застосування паралельних масштабованих архітектур. Завдання забезпечення тривалого функціонування системи має три складових: надійність, готовність і зручність обслуговування. Всі ці три складових припускають, в першу чергу, боротьбу з несправностями системи, породжуваними відмовами і збоями в її роботі.

Підвищення рівня готовності припускає применшення в певних межах впливу відмов і збоїв на роботу системи за допомогою засобів контролю і корекції помилок, а також засобів автоматичного відновлення обчислювального процесу після прояву несправності, включаючи апаратурну і програмну надмірність, на основі якої реалізуються різні варіанти відмовостійких архітектур. Підвищення готовності є спосіб боротьби за зниження часу простою системи. Основні експлуатаційні характеристики системи істотно залежать від зручності її обслуговування, зокрема від ремонтопридатності, контролепригодности і т.д. В останні роки в літературі з обчислювальної техніки все частіше вживається термін "системи високої готовності" (High Availability Systems). Всі типи систем високої готовності мають спільну мету - мінімізацію часу простою.

ПЕРСПЕКТИВИ ВИКОРИСТАННЯ ПОТОКІВ МОДЕЛІ ОБЧИСЛЕНЬ В УМОВАХ ІЄРАРХІЧНИХ КОМУНІКАЦІЙНИХ СЕРЕДОВИЩ

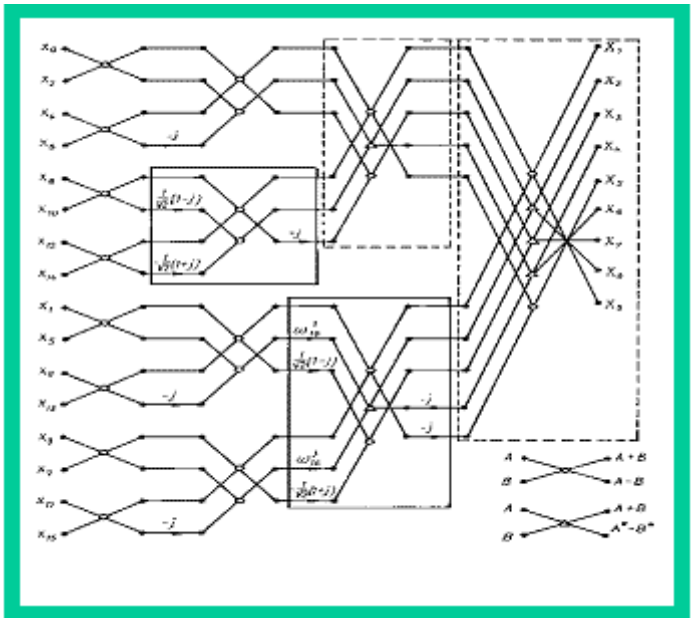
Зі збільшенням потужності комп’ютерні системи стають все менш симетричними. У їх структурі спостерігається ієрархія, яка впливає на комунікаційні можливості. Залежно від відстані між вузлами (яке слід розуміти як міру обсягу найменшого компонента системи, до якого належать обидва вузла) часи передачі між ними (затримки) можуть відрізнятися в десятки разів. Також зі збільшенням відстані, як правило, падає пропускна здатність комунікаційної мережі в перерахунку на один вузол. Зазвичай програми для суперкомп'ютерів пишуться в розрахунку на дворівневу модель обчислювача, де є вузли з швидким доступом до локальної пам'яті і помітно більшим часом передачі даних до інших вузлів (або доступу до пам'яті іншого вузла), причому цей час вважається однаковим для всіх пар вузлів. І коли в реальності ці часи сильно розрізняються, такі програми працюють погано - так, як якщо б часи і швидкості всіх передач зрівнялися з найгіршими. Виникає нетривіальне завдання адаптувати програми до таких неоднородностей, що, однак, може негативно позначитися на переносимості програм. І вже зовсім погано, коли для оптимальної роботи на різних рівнях доводиться застосовувати різні алгоритми.

Одна з причин виникнення складнощів полягає в тому, що при розпаралелювання ми спочатку прагнемо явно спланувати, що з чим в нашій задачі буде виконуватися паралельно, і що за чим буде слідувати, зокрема, використовуємо глобальні бар'єри і ефективні колективні операції, - і в результаті підвищуємо продуктивність за рахунок того, що робимо це все краще. І добре ще, якщо є тільки один-два рівня неоднорідностей (Наприклад, CPU-GPU). Але далі на цьому шляху складність стрімко

зростає. Хотілося б, щоб адаптація до різних неоднородностяей відбувалася автоматично. З вищесказаного випливає, що для цього доцільно відмовитися від ідеї планувати паралелізм заздалегідь і вагому частину роботи по распараллеливанию виконувати динамічно, без зайвих синхронізацій. Однією з відомих моделей обчислень із зазначеним властивістю є модель, заснована на управлінні потоком даних. Розглянемо версію потокової моделі обчислень, для якої в ІППМ РАН розробляється схема і модель апаратної реалізації і покажемо, як в ній відбувається автоматична адаптація до неоднородностей комунікаційного середовища. Також будуть розглянуті і інші проблеми, пов'язані з неоднорідністю і ієрархічністюобчислювальної системи, і як вони вирішуються в даній моделі обчислень.

Dataflow-архитектури.

- Висок продуктивність



Більшість сучасних обчислювальних машин, будь то суперкомп'ютер Fujitsu K, звичайна персоналки або навіть калькулятор, об'єднує загальний принцип роботи, а саме модель обчислень, засновану на потоці управління

(Controlflow). Однак, ця модель не є єдино можливою. У деякому роді її протилежністю є модель обчислень, керована потоком даних, або просто Dataflow.

Архітектуру потоку управління часто називають фон-неймановской (в честь Джона фон Неймана). Це не зовсім правильно, тому що архітектура фон Неймана - лише підмножина архітектур потоку управління. Існують не-Неймановская архітектури потоку управління, наприклад гарвардська, яку зараз можна зустріти хіба що в мікроконтролерах.

Проблеми controlflow

Справа в тому, що архітектура control flow має ряд недоліків, повністю позбутися від яких неможливо, так як вони виникають з самої організації обчислювального процесу, можна тільки зменшити негативний ефект з допомогою різних технічних рішень. Перелічимо основні проблеми:

• Перед виконанням інструкції її операнди необхідно завантажити з пам'яті в регістри процесора, а після виконання - вивантажити результат назад в пам'ять. Шина процесор-пам'ять стає вузьким місцем: процесор простоює частину часу, чекаючи завантаження даних. Проблема зі змінним успіхом вирішується за допомогою попереджувальних вибірки і декількох рівнів кеш-пам'яті.

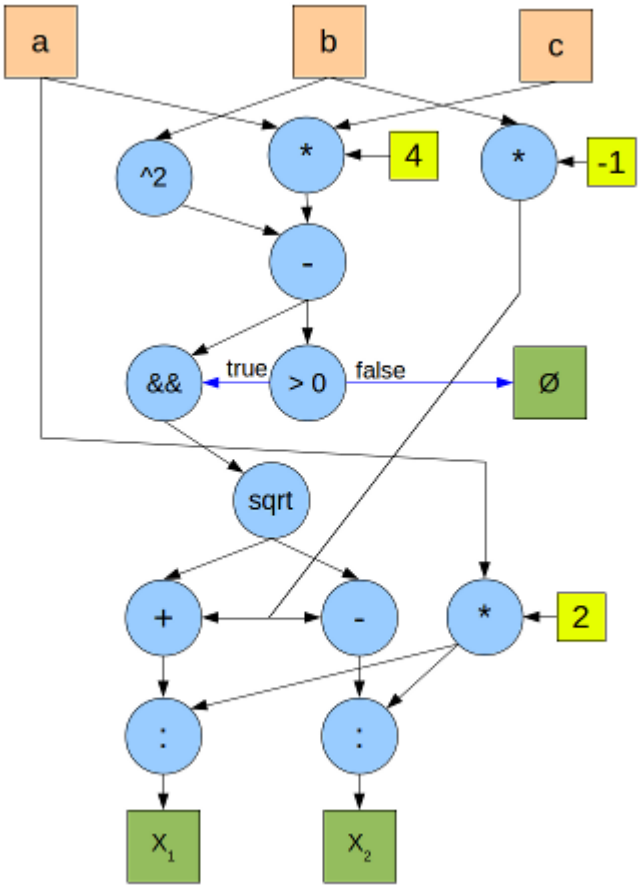
• Побудова багатопроцесорних систем пов'язане з рядом труднощів. Існують дві основні концепції таких систем: із загальною і з розподіленою пам'яттю. У першому випадку складно фізично забезпечити спільний доступ багатьох процесорів до одного ОЗУ. У другому випадку виникають проблеми когерентності даних і синхронізації. З ростом числа процесорів в системі все більше ресурсів витрачається на забезпечення синхронізації і все менше - на власне обчислення [03].

• Ніхто не гарантує, що на момент виконання будь-якої інструкції її операнди будуть знаходитися в пам'яті за вказаними адресами. Інструкція, яка повинна записати ці дані, може виявитися, ще не виконалася. У багатопотокових застосуваннях істотна частка ресурсів і нервів програміста витрачається на забезпечення синхронізації потоків.

Dataflow

Немає усталеного перекладу для терміна Dataflow architecture. Можна зустріти варіанти «потоковая архітектура», «архітектура потоку даних», «архітектура з керуванням потоком даних» і подібн

В архітектурі з керуванням потоком даних (Dataflow) [01] відсутнє поняття «послідовність інструкцій», немає Instruction Pointer'a, відсутнє навіть адресування пам'яті в звичному нам розумінні. Програма в потокової системі - це не набір команд, а обчислювальний граф. Кожен вузол графа представляє собою оператор або набір операторів, а гілки відображають залежності вузлів за даними. Черговий вузол починає виконуватись як тільки стають доступними всі його вхідні дані. У цьому полягає один з основних принципів dataflow: виконання інструкцій по готовності даних.
Ось для прикладу граф обчислення коренів квадратного рівняння. Сині кола - оператори, помаранчеві квадрати - вхідні дані, зелені - вихідні, жовті - константи. Чорні стрілки позначають передачу чисельних даних, сині -

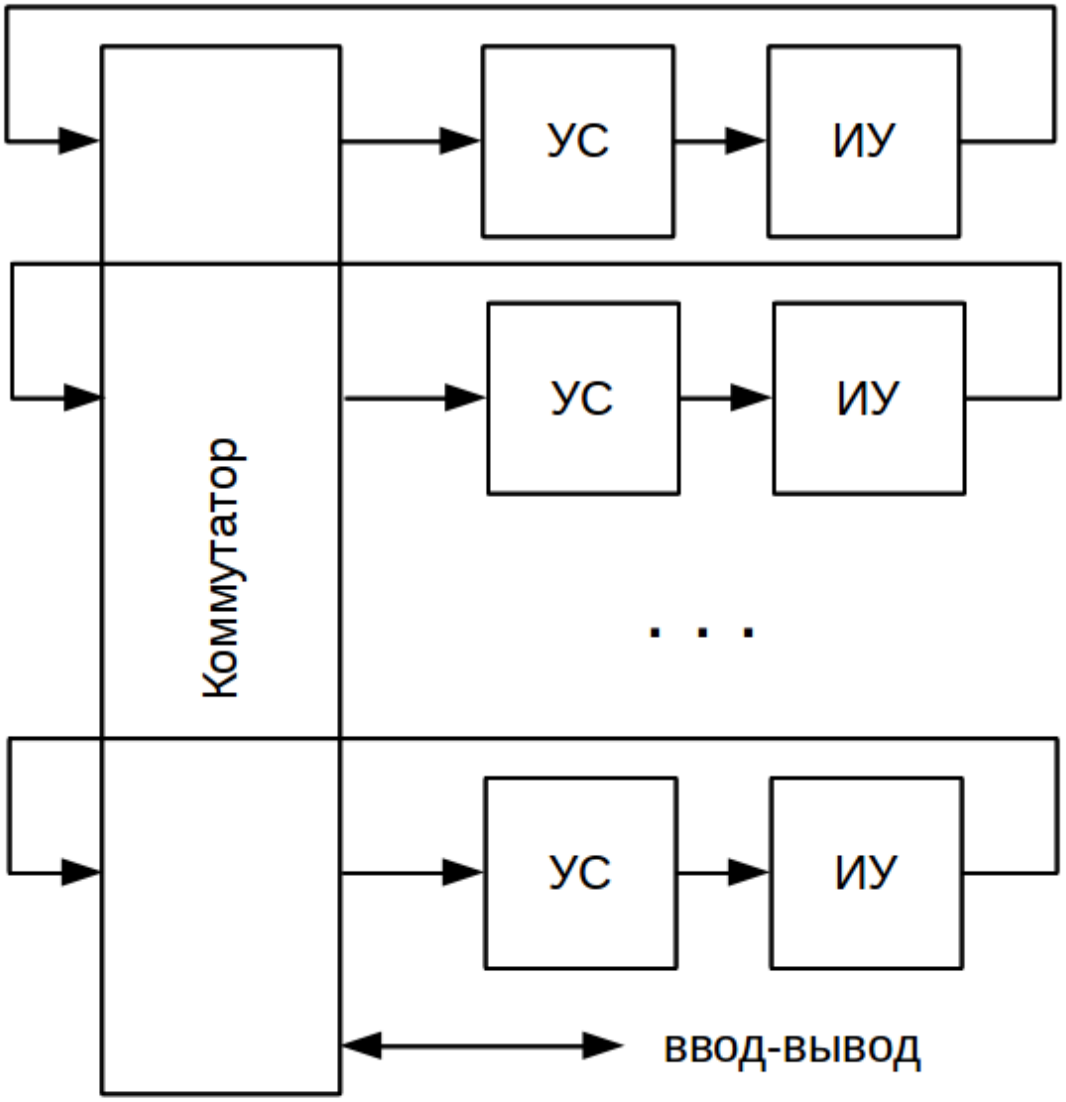


булевих.

У потокових машинах дані передаються і зберігаються у вигляді т.зв. токенов (token). Токен - це структура, яка містить власне передане значення і мітку - покажчик вузла призначення. Найпростіша потоковая обчислювальна система складається з двох пристроїв: виконавчого (execution unit) і пристрої зіставлення (matching unit) [11].

Виконавчий пристрій служить для виконання інструкцій і формування токенов з результатами операцій. Як правило, воно включає в себе пам'ять команд, доступну тільки для читання. Готовність вхідних даних вузла визначається за наявністю набору токенов з однаковими мітками. Для пошуку таких наборів і служить пристрій зіставлення. Зазвичай воно реалізується на базі асоціативної пам'яті. Використовується або «справжня», апаратна асоціативна пам'ять (CAM - content-addressable

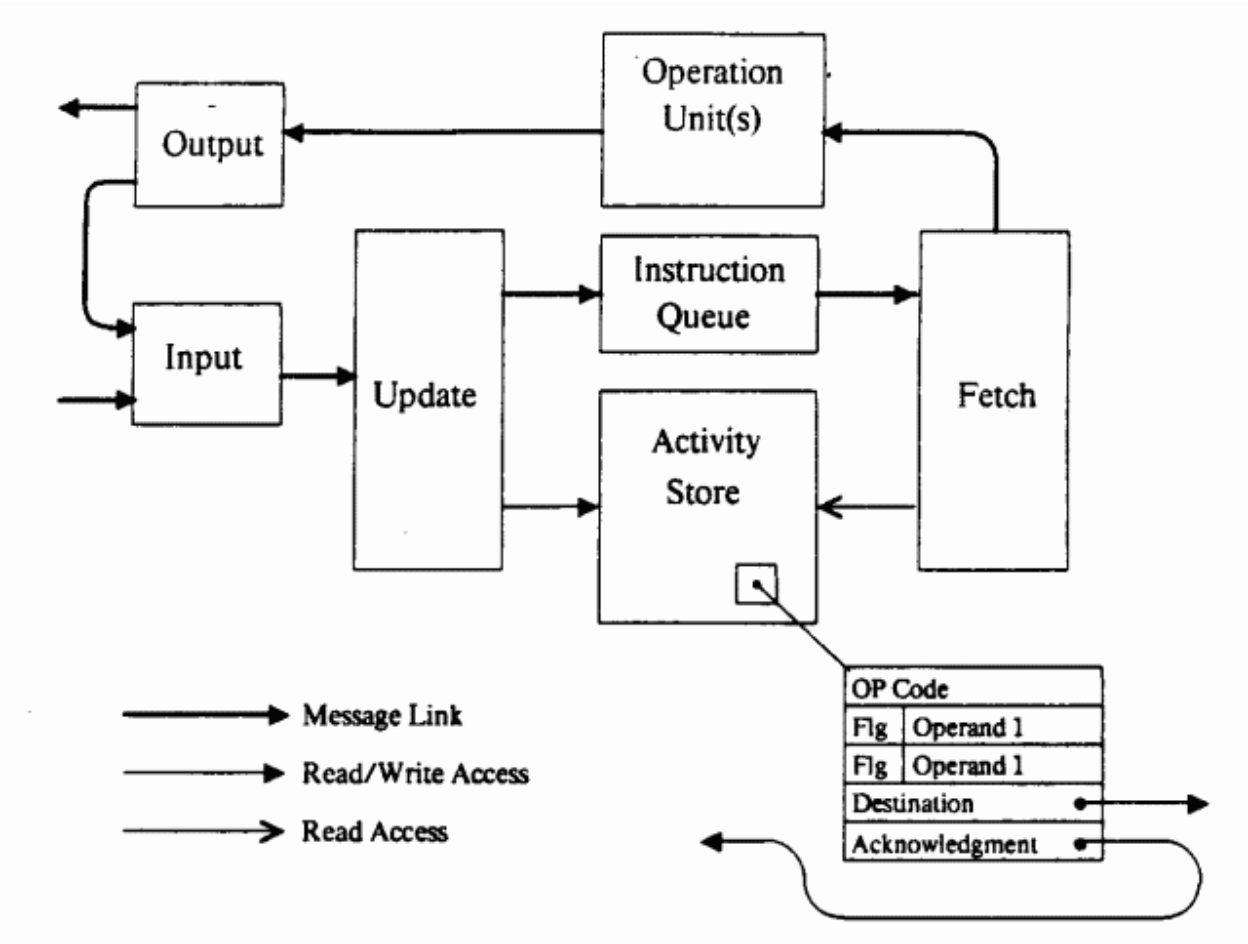
memory), або структури, що працюють аналогічно, наприклад, хеш-таблиці.Одним из основных достоинств dataflow-архитектуры является ее масштабируемость: не составляет труда собрать систему, содержащую множество устройств сопоставления и исполнительных устройств. Устройства объедиянются простейшим коммутатором, причем для адресации токенов служат их метки. Весь диапазон номеров узлов просто распределяется равномерно между устройствами. Никаких дополнительных мер для синхронизации вычислительного процесса, в отличие от многопроцессорной controlflow-архитектуры, не требуется.



Статична dataflow-архітектура

Описана вище схема називається статичною (static dataflow). У ній кожен обчислювальний вузол представлений в єдиному екземплярі, число вузлів заздалегідь відомо, також заздалегідь відомо число токенов, що циркулюють в системі. Як приклад реалізації статичної архітектури можна привести MIT Static Dataflow Machine [12] - потоковий комп'ютер, створений в Массачусетському технологічному інституті в 1974 році. Машина складалася з безлічі обробних елементів (Processing Element),

пов'язаних комунікаційною мережею. Схема одного елемента показана на малюнку:



Статична dataflow-архітектура

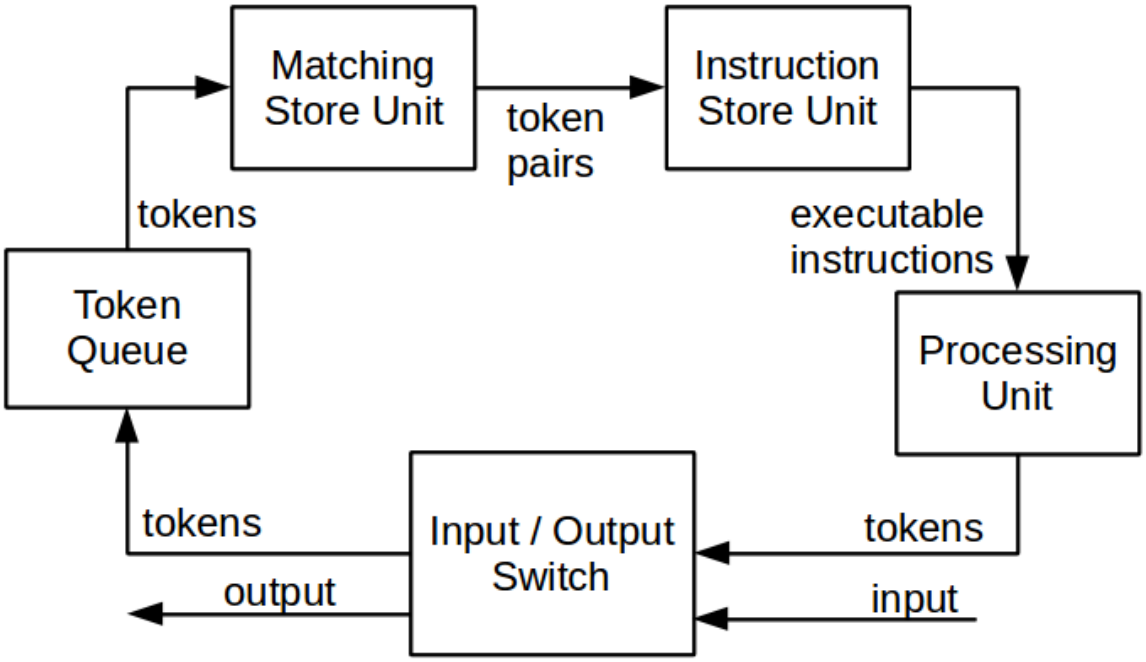
Описана вище схема називається статичною (static dataflow). У ній кожен обчислювальний вузол представлений в єдиному екземплярі, число вузлів заздалегідь відомо, також заздалегідь відомо число токенов, що циркулюють в системі. Як приклад реалізації статичної архітектури можна привести MIT Static Dataflow Machine [12] - потоковий комп'ютер, створений в Массачусетському технологічному інституті в 1974 році. Машина складалася з безлічі обробних елементів (Processing Element), пов'язаних комунікаційною мережею. Схема одного елемента показана на малюнку:

Динамічна dataflow-архітектура. У динамічної потокової архітектурі (dynamic dataflow) кожен вузол може мати безліч екземплярів. Для того, щоб розрізняти токени, адресовані врізні екземпляри одного вузла, в структуру токена вводиться додаткове поле - контекст. Зіставлення токенов тепер ведеться не тільки по мітках, а й за значеннями контексту. У порівнянні зі статичної архітектурою з'являється цілий ряд нових можливостей.

• Рекурсія. Вузол може направляти дані в свою копію, яка буде відрізнятися контекстом (але при цьому мати ту ж мітку).

• Підтримка процедур. Процедурою в рамках даної моделі обчислень буде послідовність вузлів, пов'язаних між собою і що має входи і виходи. Можна одночасно викликати кілька примірників однієї й тієї ж процедури, які будуть відрізнятися контекстом.

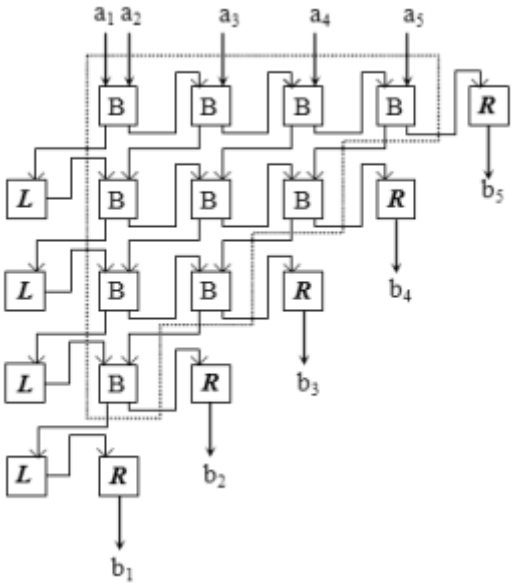
• Розпаралелювання циклів. Якщо між ітераціями циклу немає залежності за даними, можна обробляти відразу все ітерації одночасно. Номер ітерації, як ви вже напевно здогадалися, буде міститися в поле контексту. Одной из первых реализаций динамической потоковой архитектуры была система Manchester Dataflow Machine (1980 год) [13]. Машина содержала аппаратные средства для организации рекурсий, вызова процедур, раскрытия циклов, копирования и объединения ветвей вычислительного графа. Также в отдельный модуль была вынесена память команд (instruction store unit). На рисунке показана схема одного элемента машины:



Динамічна dataflow-архітектура

Динамічна dataflow-архітектура, в порівнянні зі статичною, демонструє більш високу продуктивність, за рахунок кращого паралелізму обчислень. Крім того, вона дає більше можливостей для програміста. З іншого боку, динамічна система складніше по апаратної реалізації, особливо це стосується пристроїв зіставлення і блоків формування контексту токенов.

Dataflow-архітектури. Висока продуктивність



Реалізація циклів в динамічних потокових системах

Розглянемо більш докладно роботу контексту на прикладі організації циклів. Контекст - це поле в структурі токена, що однозначно визначає екземпляр вузла dataflow-графа. У випадку з циклами контекстом буде номер ітерації.

Приклад 1. Числа Фібоначчі.

Обчислення чисел Фібоначчі є класичним прикладом циклу з залежністю ітерацій за даними. N-ну кількість дорівнює сумі (N-1) -го і (N-2) -го:

```
int fib [MAX_I];
fib [0] = 1;
fib [1] = 1;
for (i = 2; i <MAX_I; i ++)
{
fib [i] = fib [i-1] + fib [i-2];
}
```

Побудуємо потоковий граф обчислень:

Легко бачити, що граф складається з (MAX_I-2) однакових вузлів, що відрізняються тільки значеннями контексту (цифри під зображенням вузла). Логіка роботи вузла (в псевдокоді) буде виглядати так:

вузол fib (входи: токен A, токен B)
змінна i = поле_контекста (A);
змінна result = поле_данних (A) + поле_данних (B);
надіслати_токен (дане: result, мітка: host, контекст: i);
якщо (i <MAX_I-1)
надіслати _токен (дане: result, мітка: fib, контекст: i + 1);
надіслати _токен (дане: result, мітка: fib, контекст: i + 2);
кінець якщо
кінець вузол fib
Для старту програми потрібно надіслати три токена:

надіслати _токен (дане: 1, мітка: fib, контекст: 2);
надіслати _токен (дане: 1, мітка: fib, контекст: 2);
надіслати токен (дане: 1, мітка: fib, контекст: 3);

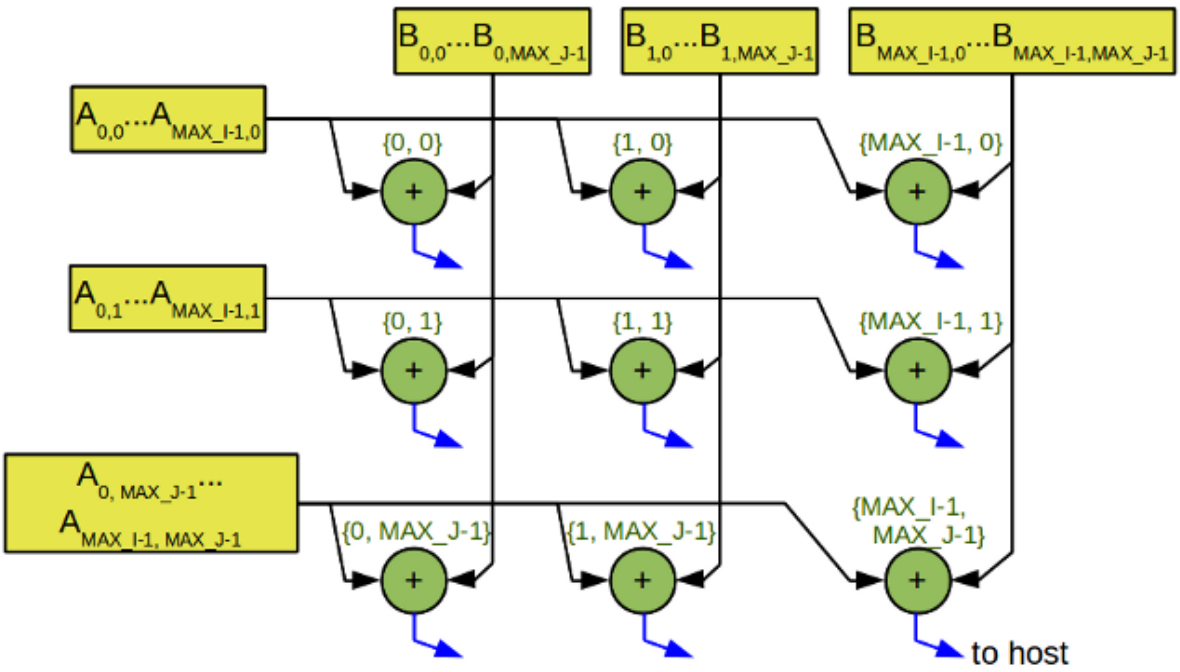
В результаті виконання потокової програми на хост буде відправлений набір токенів, кожен з яких в поле контексту буде нести номер числа Фібоначчі, а в поле даних - саме число. Зверніть увагу, ще один токен (відзначений на графі пунктирною лінією) надсилається «в нікуди», точніше, ніколи не запускає виконання вузла через відсутність парного йому токена. Позбутися його можна або додавши ще одну перевірку умови (i <MAX_I-2) в коді вузла, або організувавши програмний або апаратний «збирач сміття».

Приклад 2. Складання матриць

Розглянемо тепер приклад, де немає залежності між ітераціями за даними: додавання матриць C [i, j] = A [i, j] + B [i, j].

```
int A [MAX_I] [MAX_J];
int B [MAX_I] [MAX_J];
int C [MAX_I] [MAX_J];
GetSomeData (A, B);
for (i = 0; i <MAX_I; i ++)
for (j = 0; j <MAX_J; j ++)
{
C [i, j] = A [i, j] + B [i, j];
}
```

Всі (MAX_I * MAX_J) ітерацій можуть бути виконані одночасно. Ось так виглядає граф складання матриць:



Контекст в даному випадку є структурою з двох координат {i; j}. С урахуванням цього вузол графа буде виглядати так:

вузол add (входи: токен A, токен B)
змінна {i, j} = поле_контекста (A);
змінна result = поле_данних (A) + поле_данних (B);
надіслати_токен (дане: result, мітка: host, контекст: {i, j});
кінець вузол add

Зверніть увагу, ніяких перевірок кордонів тут немає! Число ітерацій автоматично вибирається виходячи з розмірності вхідних даних. Вхідні дані, до речі, повинні надходити в такому вигляді:

Контекст в даному випадку є структурою з двох координат {i; j}. С урахуванням цього вузол графа буде виглядати так:

вузол add (входи: токен A, токен B)
змінна {i, j} = поле_контекста (A);
змінна result = поле_данних (A) + поле_данних (B);
надіслати_токен (дане: result, мітка: host, контекст: {i, j});
кінець вузол add

Зверніть увагу, ніяких перевірок кордонів тут немає! Число ітерацій автоматично вибирається виходячи з розмірності вхідних даних. Вхідні дані, до речі, повинні надходити в такому вигляді:

Контекст в даному випадку є структурою з двох координат {i; j}. С урахуванням цього вузол графа буде виглядати так:

вузол add (входи: токен A, токен B)
змінна {i, j} = поле_контекста (A);
змінна result = поле_данних (A) + поле_данних (B);
надіслати_токен (дане: result, мітка: host, контекст: {i, j});
кінець вузол add

Зверніть увагу, ніяких перевірок кордонів тут немає! Число ітерацій автоматично вибирається виходячи з розмірності вхідних даних. Вхідні дані, до речі, повинні надходити в такому вигляді:

надіслати _токен (дане: A [0, 0], мітка: add, контекст: {0, 0});
надіслати _токен (дане: B [0, 0], мітка: add, контекст: {0, 0});
надіслати ть_токен (дане: B [0, 1], мітка: add, контекст: {0, 1});
...
надіслати _токен (дане: A [MAX_I-1, MAX_J-1], мітка: add, контекст: {MAX_I-1, MAX_J-1});
надіслати _токен (дане: B [MAX_I-1, MAX_J-1], мітка: add, контекст: {MAX_I-1, MAX_J-1});

Dataflow-системи дуже ефективні для обробки розріджених даних, так як фактично пересилаються і обробляються тільки значущі елементи._токен (дане: A [0, 0], мітка: add, контекст: {0, 0});
Надіслати _токен (дане: B [0, 0], мітка: add, контекст: {0, 0});
Надіслати _токен (дане: A [0, 1], мітка: add, контекст: {0, 1});
Надіслати _токен (дане: B [0, 1], мітка: add, контекст: {0, 1});
...
Надіслати _токен (дане: A [MAX_I-1, MAX_J-1], мітка: add, контекст: {MAX_I-1, MAX_J-1});

Надіслати _токен (дане: B [MAX_I-1, MAX_J-1], мітка: add, контекст: {MAX_I-1, MAX_J-1});

Dataflow-системи дуже ефективні для обробки розріджених даних, так як фактично пересилаються і обробляються тільки значущі елементи._токен (дане: A [0, 0], мітка: add, контекст: {0, 0});
Надіслати _токен (дане: B [0, 0], мітка: add, контекст: {0, 0});
Надіслати _токен (дане: A [0, 1], мітка: add, контекст: {0, 1});
Надіслати _токен (дане: B [0, 1], мітка: add, контекст: {0, 1});
...
Надіслати _токен (дане: A [MAX_I-1, MAX_J-1], мітка: add, контекст: {MAX_I-1, MAX_J-1});
Надіслати _токен (дане: B [MAX_I-1, MAX_J-1], мітка: add, контекст: {MAX_I-1, MAX_J-1});

Dataflow-системи дуже ефективні для обробки розріджених даних, так як фактично пересилаються і обробляються тільки значущі елементи.

Гібридні dataflow-архітектури.

«Чисті» Потокові архітектури, подібні опису MIT Static Dataflow Machine и Manchester Dataflow Machine, на жаль, мають багато Слабких Місць:

• Dataflow-машини давали величезні можливості для масового розпаралелювання. парале Зворотнього стороною цієї Можливості Було те, що на послідовніх ділянках Обчислювального графу симтема показувала різке Падіння продуктівності.

• Завантаження виконавчих прістроїв були далекі від максимально можливих. Велика частина машинного часу вітрачалася на пошук відповідності операндів, вибірку інструкцій, а виконавчий Пристрій весь цей час простоював, віконуючі тільки по одній інструкції.

• Важко Було конструювання прістрої зіставлення. Асоціативна пам'ять складніше, дорожча, повільніше, займає більше місця и споживає більше ЕНЕРГІЇ, в порівнянні зі звичайною оперативною пам'яттю такого ж обсягу.Сам принцип управління потоком даних не дозволяв організувати ефективний конвеєр. Майже всі пристрої працювали асинхронно, були потрібні буфери і черги в лініях зв'язку.
• У порівнянні з класичною мультипроцесорною архітектурою, в dataflow-машинах значно вище було навантаження на комутаційну мережу. Адже фактично, кожна операція вимагала пересилання двох токенов.

У спробах вирішити перераховані проблеми стали з'являтися гібридні архітектури, що поєднують в собі елементи як архітектур потоку даних, так і потоку управління.

Threaded dataflow

Цьому терміну немає взагалі ніякого адекватного перекладу на російську. Суть даного підходу полягає в тому, щоб послідовні ділянки обчислювального графа, які неможливо распараллелить, замінити тред (threads) - наборами послідовно виконуваних інструкцій. Відразу зникають «зайві» проміжні маркери, і зростає завантаження виконавчих пристроїв. Принципи threaded dataflow були втілені «в залізі» в процесорі Epsilon [21] в 1989 році.

Грубозерниста архітектура потоку даних

Подальшим розвитком threaded dataflow стали так назиавані грубозернисті потокові архітектури (large-grain dataflow). Коли стало ясно, що паралелізм «чистого» dataflow в багатьох випадках надлишковий, виникло рішення будувати потоковий граф не з окремих операторів, а з блоків. Таким чином, в крупнозернистій архітектурі кожен вузол являє собою не одну інструкцію,

а класичну послідовну програму. Взаємодія між вузлами і раніше здійснюється за принципом потоку даних. Одним з переваг такого підходу стала можливість використовувати в якості виконавчих пристроїв звичайні фон-Неймановская процесори. Слід зазначити, що незважаючи на назву «грубозерниста», блоки, на які розбивається завдання, все одно набагато дрібніше, ніж, скажімо, в кластерних системах. Типовий розмір блоку становить 10-100 інструкцій і 16-16-1К байт даних.

Векторна dataflow-архітектура

У векторних потокових системах токен містить не одне значення, а відразу безліч. Відповідно, і операції виконуються не над парами операндів, а над парами векторів. Як приклад такої системи можна привести машину SIGMA-1 (1988) [22]. Іноді векторний режим підтримується тільки частиною виконавчих пристроїв системи. Часто також застосовуються гібридні архітектури, що поєднують відразу кілька підходів, наприклад грубозерниста архітектура з можливістю виконання векторних операцій.

Реконфігуровані системи

Розвиток технологій ПЛІС зробило можливим принципово новий підхід до архітектури dataflow. Що, якщо зібрати машину, орієнтовану на вирішення однієї конкретної задачі? Якщо реалізувати безпосередньо на схемотехническом рівні потрібний обчислювальний граф, можна досягти приголомшливих результатів. Замість пристроїв складних і повільних зіставлення можна використовувати безумовне перенаправлення даних від одного функціонального модуля до іншого. Самі виконавчі пристрої теж можна «заточити» під потрібне завдання: вибрати тип арифметики, розрядність, потрібний набір підтримуваних операцій.

Зрозуміло, подібна машина буде дуже вузькоспеціалізованою, але ж гідність ПЛІС як раз в можливості неодноразового перепрограмування.

Таким чином, під кожну окрему задачу збирається потрібна архітектура. Деякі системи дозволяють навіть здійснювати перенастроювання прямо в процесі роботи. Реконфігуровані системи на базі FPGA-мікросхем в даний час випускаються серійно в самих різних форматах - від блоку «прискорювача» для ПК до системи продуктивністю порядку декількох Тфлопс [31].

З недоліків реконфігурованих архітектур можна виділити наступні:

• Принципова однозадачность. Для запуску нового завдання потрібно зупинка системи і перепрограмування ПЛІС, що входять до її складу.

• Складність програмування. Програмування кожного завдання включає в себе синтез всієї обчислювальної архітектури під це завдання.

• Надлишкова апаратна складність. Зворотною стороною гнучкості ПЛІС є наявність на кристалі великого відсотка елементів, які безпосередньо не беруть участі в обчисленнях, а служать тільки для реконфігурації. Тим не менше, ці елементи споживають енергію і виділяють тепло під час роботи, що погіршує енергетичні показники ефективності системи (Гфлопс / Вт).

Опис моделі обчислень

Дана версія моделі обчислень з керуванням потоком даних не ставить за мету підтримати традиційну модель програмування. Більш того, пропонується забути про циклах і масивах. Замість них є величезна (загальним обсягом близько 264 і навіть більше) простір віртуальних вузлів (ВУ), що позначаються ім'ям з набором індексів (полів контексту), наприклад X1 {2,5,768}. Ім'я та контекст разом складають адреса ВУ. Програма в даній моделі обчислень - це кінцевий набір іменованих описів вузлів. В описі вузла є заголовок, що містить ім'я, формат контексту (число і типи полів), число входів і їх типи, а також програма вузла. Реально використовуються ВУ є вузлами обчислювального графа. Зв'язки між ВУ укладені в програмах вузлів, які обчислюють, в залежності від значень

входів і полів контексту, деякі нові значення і посилають їх на входи інших вузлів у вигляді токенов. При цьому набір випускаються токенов, включаючи як дані, так і адреси ВУполучателей, формується програмою ВУ-відправника. Це головна особливість нашої моделі обчислень, тому ми називаємо її моделлю потоку даних з динамічно обчислюваним контекстом - ПД ДВК. У ній програма є функцією, що обчислює безліч випускаються токенов (з усіма їхніми атрибутами) по вузлу-відправнику і наявної у нього інформації, отриманої у вигляді вхідних токенов. Робота в моделі ПД ДВК відбувається наступним чином. У робочому пулі (РП) є деяка кількість токенов, спрямованих на входи ВУ. Вміст токена, як і оператор посилки токена, має вигляд:

v → N.a {i1, i2, ...};

де v - посилається значення (в програмі - вираз), N - ім'я (типу) вузла, a - ім'я входу, i1, ... - значення полів контексту (вираження). Деякий невизначений час токен «рухається до мети», потім досягає її і поміщається в позицію входу певного ВУ. Коли у деякого ВУ є в наявності токени на всіх входах, ВУ спрацьовує і формується пакет - завдання на виконання програми вузла, в якому задані значення всіх полів контексту і всіх входів ВУ. Програма вузла схожа на звичайну фон-неймановскую підпрограму. У ній можуть знаходитися оператори посилки нових токенов. Виконані в процесі виконання програми вузла оператори посилки створюють нові маркери, які поміщаються в РП. Токени, використані при спрацьовуванні ВУ, зазвичай відразу видаляються з РП. Деякі вузли в програмі відзначені як вихідні: вони не мають програми та направляються на них токени передаються в якості результату на хост-процесор. Вихідні дані подаються з хост-процесора у вигляді початкових токенов на вхідні вузли. Такий базовий механізм. Можливі варіації, коли спеціальні маркери мають спеціальне поведінку. Наприклад, поле контексту при відправці може бути задано зірочкою - тоді токен йде на все ВУ як би зі всілякими значеннями в

цьому полі. Може бути задана кратність токена (як #n перед стрілкою) - тоді він втечуть з РП тільки після n використань. При кожному використанні з кратності віднімається 1. Спрацювання разом з вирахуванням або видаленням - атомарна операція. Символ ## означає нескінченну кратність. Є спеціальні маркери стирання і деякі інші.

Розглянемо приклад разностной задачі моделювання поширення тепла в двовимірної області (Рис.1). Тут b, c, d - константи, k - номер кроку по часу, i і j - координати точки в просторі. Входи x1 і x2 основного вузла F відповідають сусіднім точкам зліва і справа, y1 і y2 - знизу і зверху, u - поточній точці. Одновходовой вузол Copy розсилає обчислене значення за п'ятьма напрямками. Тут немає циклів: область завдання визначається сукупністю токенов з початковими значеннями виду

V0ij → Copy {0, i, j}

а з кожної точки кордону {i, j} повинен бути спрямований токен з її значенням на відповідний вхід вузла F кожної суміжній з нею внутрішньої точки області:G_{ij} ## → F.x1{*,i+1,j}

```
node F(x1,x2,u,y1,y2:real) {k,i,j};
b*(x1+x2)+c*(y1+y2)+d*u → Copy{k+1,i,j};
node Copy(u: real){k,i,j}; u → F.u {k,i,j},
                F.x1 {k,i+1,j},
          F.x2 {k,i-1,j},
F.y1 {k,i,j+1},
                  F.y2 {k,i,j-1};
```

Також на кожну точку {i, j} кордону поза області слід послати токен стирання

erase ## F {*, i, j},

який буде стирати зайві маркери, що посилаються «зовні». Для завершення на рівні k1 слід заздалегідь послати токен «опосередкованість»:

@E ## → Copy {k1, *, *},

який перенаправляє всі токени з вузлів Copy {k1, i, j} (для всіх i, j) на вузли E {k1, i, j}.

Реалізація моделі. Може здатися, що дана модель обчислень передбачає наявність вузького горла - єдиного сховища токенов, де зустрічаються токени з загальним ім'ям і контекстом. Але це не обов'язково так. З метою ефективності реалізації весь адресний простір ВУ ділиться на приблизно рівні фрагменти, які розміщуються на різних процесорах (ядрах). Номер процесора обчислюється як функція від адреси ВУ. Тим самим при створенні токена відразу визначається, в яке ядро він повинен бути переданий. Функція розподілу задається автором програми як доповнення до основного коду, причому воно ніяк не впливає на логіку роботи, а впливає тільки на продуктивність. Вибір функції розподілу має такі цілі:

а) забезпечити рівномірність навантаження на ядра,

б) мінімізувати потоки токенов між ядрами,

в) знизити чутливість програми до затримок в мережі.

Сама модель обчислень вносить в реалізацію певні накладні витрати, до яких можна віднести наступні:

о організація асоціативного доступу; о заходи щодо запобігання переповнення;организация динамической памяти для многовходовых виртуальных узлов; о накопление токенов в виртуальном узле;

о виконання перевірок умови готовності при кожній парафії токена; о формування пакету з набору токенов; о активація пакета; о формування

токена; о обчислення функції розподілу; о пересилання токенов, в тому числі з використанням broadcast і multicast.

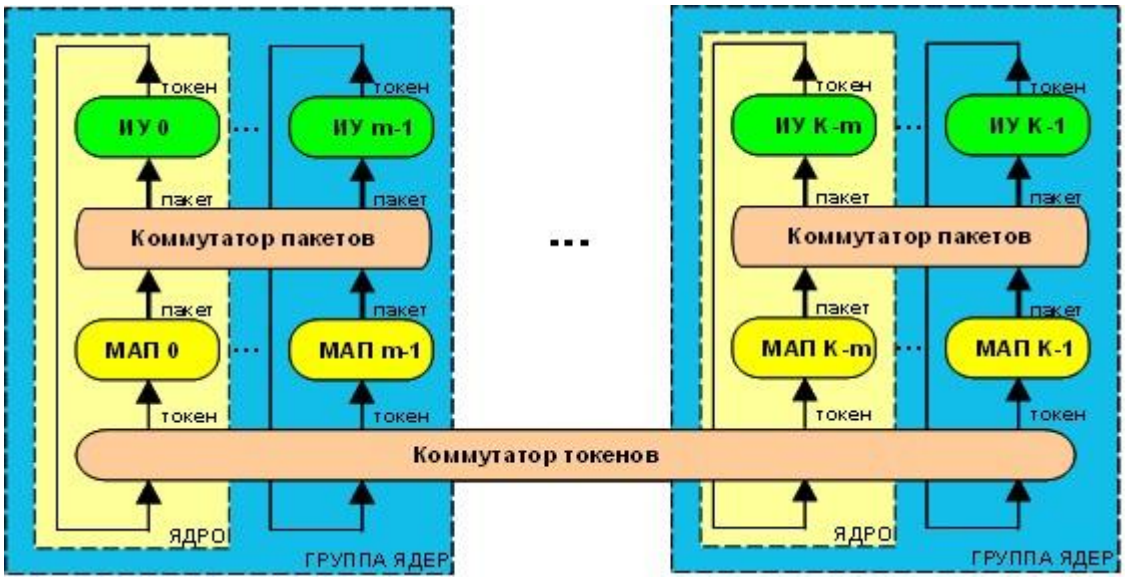
Якщо ці функції обчислювати програмно, використовуючи звичайні процесори, які застосовуються в суперкомп'ютерах, то на тлі мелкозернистости програми (яка визначається розміром програм вузлів) ці накладні витрати будуть дуже великі. Але, оскільки це конкретні функції, і їх перелік обмежений, то є сенс підтримати їх апаратно. Єдина з обчислення функцій розподілу, які задаються користувачем-програмістом. Але сам клас цих функцій дуже специфічний і може бути реалізований на ПЛІС. перерахованих функцій, що вимагає програмного управління - це

Проект апаратної реалізації моделі ПД ДВК у вигляді многоядерной однокристальної системи був запропонований в [6], а потім розвинений і узагальнений до розподіленої масштабованої системи в [7]. На Рис.2 показана структурна схема системи. Між ядрами в системі передаються одиниці інформації - токени. Токен є структурою, яка містить дане (операнд), ключ, маску ключа, кратність і набір службових полів. Як між ядрами в групі, так і між групами ядер діє один і той же протокол взаємодії, який здійснює доставку токенов. Завдяки цьому система може необмежено масштабироваться.

Кожне ядро обчислювальної системи складається з:

• модуля асоціативної пам'яті (МАП), де відбувається зіставлення токенов за певними правилами, їх накопичення і формування пакетів;

• виконавчого пристрою (ВП), де відбувається обробка пакетів шляхом виконання програми відповідного вузла, в процесі якого можуть породжуватися нові маркери;



• блоку хешування, де на апаратному рівні обчислюється функція розподілу, яка визначає номер цільового ядра токена.

Рис. 2. Структурна схема системи (МАП - модуль асоціативної пам'яті, ІУ - виконавчий пристрій, m - число ядер в групі, К - загальна кількість ядер в системі).

Кожне ядро підтримує свою локальну частину РП: здійснює зіставлення токенов по ключам, формує пакети і виконує програми вузлів. Для цього і потрібна асоціативна пам'ять (АП), необхідний обсяг якої визначається фактично максимальним числом одночасно фізично існуючих віртуальних вузлів в цій частині РП. Віртуальний вузол стає фізично існуючим, коли на нього прийде хоча б один токен, але ще не всі необхідні для його виконання токени. Тоді в АП з'являється ключ з адресою цього вузла, і в динамічної пам'яті відводиться місце для накопичення токенов (даних) для цього вузла. Коли прийде останній токен з числа необхідних, буде сформований пакет (можливо, в цьому ж місці), який буде поставлений в чергу на виконання в ІУ.

Реальна АП одного ядра не обов'язково виробляє фізично порівняння вхідного ключа з усіма наявними ключами. Вона може бути влаштована як

класичний кеш, де частково здійснюється адресний доступ (на основі тієї ж функції розподілу, але більш детальної), частково асоціативний. Необхідна ступінь асоціативності визначається ступенем неоднорідності використовуваного програмою безлічі адрес щодо використовуваної функції розподілу. Відмінність дисципліни роботи АП від кеша лише в реакції на відсутність адреси: в разі кеша пошук перенаправляється на зовнішній пам'ять або зовнішній кеш, а в разі АП вважається, що адреса відсутня і він створюється новим для нового токена.

Нові маркери формуються в ІУ спеціальними командами. Ці операції також повинні бути підтримані на апаратному рівні, оскільки пов'язані з дуже специфічними діями, які було б накладно виконувати програмно. Роль цих команд подібна до ролі операцій запису-читання по глобальному адресою в системах типу PGAS. Відмінність в тому, що тут використовуються тільки односторонні посилки типу записи (немає двосторонніх посилок типу читання), а також токен несе код-команду, яка визначає дію, яке необхідно зробити в цільовому ядрі (із зазначенням вузла, який, можливо, буде активований). На виході з блоку хеширования варто блок порівняння номера цільового ядра з номером свого ядра, який при збігу перенаправляє токен відразу в свій МАП, минаючи глобальний комутатор токенов.

В цілому обчислювальна система, що реалізує модель ПД ДВК, володіє наступними особливостями, корисними для високопродуктивних обчислень:

• є асоціативна пам'ять з розвиненою системою команд, на основі якої реалізується глобальне ВАП (ключів токенов, або віртуальних вузлів);

• система добре масштабується, що дозволяє реалізувати можливість створення многоядерного кристала, а в подальшому і високопродуктивних систем на базі цих кристалів;

• реалізується апаратне виявлення неявного паралелізму завдання в ході її рішення;

• є апаратно-програмні засоби управління паралелізмом завдання;

• має місце асинхронність роботи окремих блоків системи;

• потоковая організація обчислювального процесу дозволяє нівелювати затримки в комунікаційної мережі.

Основні труднощі на шляху здійснення даного проекту полягає в тому, що для його реалізації потрібна розробка спеціального «заліза» у вигляді багатоядерних кристалів нового типу, а також спеціальної комунікаційної середовища зі специфічними інтерфейсами і т.п. Інакше, через зазначених вище операцій, може бути втрачено 1-2 порядки продуктивності, що зробить весь результат безглуздим. Проте, ми вважаємо, що реалізація даної моделі обчислень на існуючій елементній базі теж має сенс як прототип, що демонструє можливості нової архітектури на різних завданнях. А для окремих завдань, де великий питомих вага обчислень в вузлах, можливий і реальний виграш.

В інформатиці існує великий набір проблем, які є NP-складними. Це означає, що навіть найпотужніші обчислювальні системи потребуватимуть для обчислення загальноприйнятими алгоритмами дуже багато часу.

Для вирішення схожих непересічних складних проблем природа завжди була великим джерелом натхнення для всього людства. Прикладом такого запозичення є генетичні алгоритми, засновані на концепції природного відбору та генетики, що виявляються ефективним інструментом для забезпечення рішень, близьких до оптимальних, за короткий проміжок часу.

ГЕНЕТИЧНІ АЛГОРИТМИ

Генетичні алгоритми - це процедури пошуку, засновані на механізмах природного відбору і успадкування [1]. У них використовується еволюційний принцип виживання найбільш пристосованих особин. Вони відрізняються від традиційних методів декількома базовими елементами. Зокрема, генетичні алгоритми здатні:

• обробляти не значення параметрів самого завдання, а їх закодовану форму;

• здійснюють пошук рішення, виходячи не з єдиної точки, а з їх деякої популяції;

• використовують тільки цільову функцію, а не її похідні або іншу додаткову інформацію;

• застосовують імовірнісні, а не детерміновані правила вибору.

Перераховані чотири властивості, які можна сформулювати також як кодування параметрів, операції на популяціях, використання мінімуму інформації про завдання і рандомізація операцій призводять в результаті до стійкості генетичних алгоритмів і до їх переваги над іншими широко вживаними технологіями [2].

При описі генетичних алгоритмів використовуються визначення, запозичені з генетики. Наприклад, йдеться про популяцію особин, а в якості базових понять застосовуються ген, хромосома, генотип, фенотип, аллель. Також використовуються відповідні цим термінам визначення технічного словника, зокрема, ланцюг, двійкова послідовність, структура.

Популяція - це кінцева множина особин.

Особини, що входять до популяцію, в генетичних алгоритмах представляються хромосомами з закодованими в них множинами

параметрів задачі, тобто рішень, які інакше називаються точками в просторі пошуку.

Хромосоми - це впорядковані послідовності генів.

Ген - це атомарний елемент генотипу, зокрема, хромосоми.

Генотип або структура - це набір хромосом даної особини.

Отже, особинами популяції можуть бути генотипи або одиничні хромосоми (в досить поширеному випадку, коли генотип складається з однієї хромосоми).

Фенотип - це набір значень, що відповідають даному генотипу, тобто декодована структура або безліч параметрів задачі (рішення, точка простору пошуку).

Дуже важливим поняттям в генетичних алгоритмах вважається функція пристосованості, інакше звана функцією оцінки. Вона являє собою міру пристосованості даної особини в популяції. Ця функція відіграє найважливішу роль, оскільки дозволяє оцінити ступінь пристосованості конкретних особин в популяції і вибрати з них найбільш пристосовані (які мають найбільші значення функції пристосованості) відповідно до еволюційного принципу виживання «найсильніших» (які найкраще пристосувалися). Функція пристосованості також отримала свою назву безпосередньо з генетики. Вона впливає на функціонування генетичних алгоритмів і повинна мати точне і коректне визначення.

Наприклад, у завданні оптимізації функція пристосованості, як правило, оптимізується (точніше кажучи, максимізується) і є цільовою функцією. У задачах мінімізації цільова функція перетворюється, і проблема зводиться до максимізації. На кожній ітерації генетичного алгоритму пристосованість кожної особини даної популяції оцінюється за

допомогою функції пристосованості, і на цій основі створюється наступна популяція особин, що залишають безліч потенційних рішень проблеми, наприклад, завдання оптимізації.

РІЗНОВИДИ ГЕНЕТИЧНИХ АЛГОРИТМІВ

Слід зазначити, що в даний час генетичні алгоритми - це цілий клас алгоритмів, спрямований на вирішення різноманітних завдань. Прикладами різних ГА можуть бути наступні алгоритми.

Канонічний генетичний алгоритм

Дана модель алгоритму є класичною. Вона була запропонована Джоном Холландом в його роботі [3]. Відповідно до неї, популяція складається з N хромосом з фіксованою розрядністю генів. За допомогою пропорційного відбору формується проміжний масив, з якого випадковим чином вибираються два батька. Далі проводиться одноточковий кросинговер, і створені два нащадка мутують (одноточковий мутація) із заданою вірогідністю. Мутовані нащадки займають місця своїх батьків. Процес триває до тих пір, поки не буде досягнуто критерію закінчення алгоритму.

Генітор

У моделі генітор (Genitor) використовується специфічний спосіб відбору [4]. Спочатку, як і у класичному варіанту, популяція инициалізується, і її особини оцінюються. Потім випадковим чином обираються дві особини, схрещуються, причому створюється тільки один нащадок, який оцінюється і займає місце менш пристосованої особини в популяції (а не одного з батьків). Після цього знову випадковим чином вибираються дві особини, і їх нащадок займає місце батьківської особини з найнижчою пристосованістю.

Таким чином, на кожному кроці в популяції оновлюється лише одна особина. Процес триває до тих пір, поки придатності хромосом не стануть однаковими. В даний алгоритм можна додати мутацію нащадка після його створення. Критерій закінчення процесу, як і вид кросинговеру і мутації, можна вибирати різними способами.

Метод переривчастої рівноваги

Даний метод заснований на палеонтологічній теорії переривчастої рівноваги, яка описує швидку еволюцію за рахунок вулканічних і інших змін земної кори [5]. Для застосування даного методу в технічних завданнях пропонується після кожної генерації проміжного покоління випадковим чином перемішувати особини в популяції, а потім застосовувати основний генетичний алгоритм.

У даній моделі для відбору батьківських пар використовується принцип панміксії. Утворені в результаті кросинговеру нащадки і найбільш придатні батьки випадковим чином змішуються. Із загальної маси в нове покоління потраплять лише ті особини, придатність яких вище середньої. Тим самим досягається управління розміром популяції в залежності від наявності кращих особин.

Така модифікація методу переривчастого рівноваги може дозволити скоротити неперспективні популяції і розширити популяції, в яких знаходяться кращі індивідуальності. Як пише В.В Курейчик: «метод переривчастої рівноваги - це потужний стресовий метод зміни навколишнього середовища, який використовується для ефективного виходу з локальних ям» [6].

Гібридний алгоритм

Ідея гібридних алгоритмів (Hybrid algorithms) полягає в поєднанні генетичного алгоритму з деяким іншим класичним методом пошуку,

специфічним для даного завдання. У кожному поколінні згенеровані нащадки оптимізуються обраним методом і потім заносяться в нову популяцію [7]. Тим самим виходить, що кожна особина в популяції досягає локального оптимуму, поблизу якого вона знаходиться. Далі виробляються звичайні для генетичного алгоритму дії: відбір батьківських пар, кросинговер і мутації. На практиці гібридні алгоритми виявляються дуже вдалими. Це пов'язано з тим, що ймовірність потрапляння однієї з особин в область глобального максимуму зазвичай є досить значною. Після оптимізації така особина буде рішенням завдання.

Відомо, що генетичний алгоритм здатний швидко знайти у всій області пошуку непогані рішення, але він може відчувати труднощі в отриманні з них найкращих. Навпаки, звичайний оптимізаційний підхід може швидко досягти локального максимуму, але не може знайти глобальний. Поєднання двох алгоритмів дозволяє використовувати переваги обох.

CHC-алгоритми

Назва CHC (Eshelman, 1991) розшифровується як Cross-population selection, Heterogeneous recombination and Cataclysmic mutation [8]. Даний алгоритм досить швидко сходиться з-за того, що в ньому немає мутацій, наступних за оператором кросинговеру, використовуються популяції невеликого розміру, і відбір особин в наступне покоління ведеться і між батьківськими особинами, і між їхніми нащадками. В даному методі для кросинговеру вибирається випадкова пара, але не допускається, щоб між батьками було маленька хеммінгова відстань або мало відстань між крайніми бітами, що розрізняються.

Для схрещування використовується різновид однорідного кросовера HUX (Half Uniform Crossover), при якому нащадку переходить рівно

половина бітів кожного з батьків. Для нового покоління вибираються N кращих різних особин серед батьків і дітей. При цьому дублювання рядків не допускається.

У моделі CHC розмір популяції відносно малий - близько 50 особин. Це виправдовує використання однорідного кросинговеру і дозволяє алгоритму зійтися до вирішення. Для отримання більш-менш однакових рядків CHC застосовує cataclysmic mutation: всі рядки, крім найбільш пристосованої, піддаються сильній мутації (змінюється близько третини бітів). Таким чином, алгоритм перезапускается і далі продовжує роботу, застосовуючи тільки кроссинговер.

Генетичний алгоритм з нефіксованим розміром популяції

У генетичному алгоритмі з нефіксованим розміром популяції кожній особині приписується максимальний вік, тобто число поколінь, після чого особина гине [9]. Впровадження в алгоритм нового параметра - віку - дозволяє виключити оператор відбору в нову популяцію. Вік кожної особини індивідуальний і залежить від її пристосованості. У кожному поколінні на етапі відтворення створюється додаткова популяція з нащадків.

Для відтворення особини обираються з основної популяції з однаковою ймовірністю незалежно від їх пристосованості. Після застосування мутації і кросинговеру нащадкам приписується вік за значенням їх пристосованості. Вік є константою протягом всієї еволюції особини (від народження до загибелі). Потім з основної популяції видаляються ті особини, термін життя яких закінчився, і додаються нащадки з проміжної популяції.

ЗАСТОСУВАННЯ ГЕНЕТИЧНИХ АЛГОРИТМІВ ДЛЯ РІШЕННЯ ДЕЯКИХ ЗАВДАНЬ

Застосування ГА в оптимізації запитів СУБД

З ростом числа таблиць в запиті кількість можливих перестановок росте як n!, Отже, пропорційно зростає і час оцінки для кожної з них. Це робить проблематичним оптимізацію запитів на основі великого числа таблиць. У пошуках вирішення цієї проблеми в 1991 році Kristin Bennett, Michael Ferris, Yannis Ioannidis запропонували використовувати генетичний алгоритм для оптимізації запитів, який дає субоптимальное рішення за лінійний час. При використанні генетичного алгоритму досліджується тільки частина простору перестановок. Таблиці, які беруть участь в запиті, кодуються в хромосоми. Над ними виконуються мутації і схрещування. На кожній ітерації виконується відновлення хромосом для отримання осмисленої перестановки таблиць і відбір хромосом, які дають мінімальні оцінки вартості. В результаті відбору залишаються тільки ті хромосоми, які дають менше, в порівнянні з попередньою итерацией, значення функції вартості. Таким чином відбувається дослідження і знаходження локальних мінімумів функції вартості. Передбачається, що глобальний мінімум не дає істотних переваг, у порівнянні з найкращим локальним мінімумом. Алгоритм повторюється кілька ітерацій, після чого вибирається найбільш ефективне рішення.

Значне зростання обсягу даних, необхідних для того, щоб приймати правильні рішення, робить поточні оптимізатори запитів неадекватними в деяких ситуаціях. Компанії, що надають банківські послуги є хорошим прикладом типового клієнта, який потребує дуже великої місткості для зберігання дуже великих і складних конструкцій баз даних. У таких користувачів дійсно потрібно використовувати дуже великі запити на з'єднання (join queries), щоб підтримати їх в своїх бізнес-рішеннях. Різні

підходи були запропоновані для виправлення цієї ситуації. Евристичні алгоритми: простір пошуку скорочується з використанням оцінок. Вони, як правило, дуже швидкі, але рідко знаходять оптимальне рішення. Випадкові алгоритми: випадкова прогулянка по простору пошуку виконується для того, щоб знайти майже оптимальне рішення. Різні політики призводять до різних алгоритмах, а саме итеративного удосконалення, імітації відпалу, гібридних алгоритмів і т. д. Генетичні алгоритми: натхненні в природному відборі, генетичні алгоритми намагаються знайти оптимальне серед популяції. ця популяція страждає від постійних перетворень, здійснюється з використанням трьох основних видів діяльності: відбір, поєднання і мутації.

Оптимізація запитів може бути зведена до задачі пошуку, де СУБД повинна знайти оптимальний план виконання запиту (QEP) в великому просторі пошуку. Кожен QEP можна розглядати в якості можливого рішення (або програми) для завдання знаходження гарного шляху доступу для отримання даних, необхідних у запиті. Таким чином, в генетичному оптимізаторі запитів, кожен член популяції є QEP. оптимізація запитів за участю великої кількості об'єднань з використанням генетичних алгоритмів був введений Беннет Ель-Аль і випробувано пізніше Steinbrunn і ін. Поки, що це дуже конкурентний підхід [13,14].

Приклад застосування генетичного алгоритму на запиті з використанням 5 join.

Тут ми розглянемо наведений вище запит з N відносин, де (N = 5):

1. Принцип роботи генетичного алгоритму є створення популяції (рішення простору).

2. Після створення населення встановити No. Покоління і No. Нащадка, який створює основу для населення.

3. Потім, взявши цикл для всіх відносин в запиті створюємо простір рішень т. е. популяцію шляхом випадкового вибору співвідношення, використовуючи rand (). Тут для його відносини повинні бути з'єднані випадковим чином вибирається інше співвідношення враховуючи Left Deep дерево [15].

4. Після того, що відповідно до принципу ГА вибір батьків з популяції і підрахунок їх хромосом.

5. Обчислити фітнес значення кожної хромосоми і вибрати найбільш придатну хромосому

6. Для розрахунку в фітнес-функцію слід розглядати як P (x) = x2.

7. Після того, як фітнес значення розраховують для хромосом ці хромосоми схрещуються один з другом.

8. Після завершення кросинговер відбувається мутація.

Розглянемо загальний випадок векторної багатокритеріальної задачі. Необхідно знайти:

min f (x) = [f1 (x), f2 (x), ..., fk (x)] (1)

тут x = [x1, x2, ..., xi, ..., xn]^T - вектор рішень, i = 1,2, ..., n, n - кількість змінних; x∈ X, де X ⊂ R^n - безліч допустимих рішень; f j (x) - j -й критерій оцінки, j = 1,2, ..., k. Вектор f (x) називаємо критеріальним вектором, а f (X) = Y ⊂ R^k - безліччю допустимих оцінок, де R^k - критеріальне простір. Оскільки багатокритеріальна оптимізація (БО) полягає в пошуку оптимального рішення, задовольняє одночасно більш ніж однієї цільової функції, то для знаходження компромісного рішення в багатокритеріальних моделях в теорії оптимізації введене поняття рішення оптимального по Парето[12], яке відоме також як непокращуване рішення або рішення недомінірованне. Формальне визначення Парето-оптимального рішення

задачі БО дамо наступним чином. Вектор x ∈ X називають Парето-оптимальним рішенням задачі (1) тоді і тільки тоді, коли не існує іншого вектора рішень x * ∈ X такого, що f j (x *) ≤ f j (x) для j = 1,2, ..., k, причому хоча б для одного j це обмеження виконується строго.

Для вирішення такого виду завдань БО розроблені різні методи і підходи, які використовують традиційні техніки оптимізації та пошуку рішень. Одним з таких добре відомих методів є метод, який об'єднує оптимізуються критерії в одну цільову функцію з використанням зваженої суми цих критеріїв, взятих з певними вагами. Інший підхід відомий під назвою «методу функції відстані» (Method of distance functions). Так як генетичні алгоритми добре зарекомендували себе в Як методик пошуку у великих областях практично при повній відсутності інформації про властивості цільової функцій і обмежень, в різних дослідженнях було розроблено кілька методів і підходів використання генетичних алгоритмів для вирішення БО. Вперше ідею використання генетичного алгоритму для вирішення задач БО запропонував в своїх роботах Розенберг. Однак в практичної імплементації в біохімічних експериментах він свої ідеї не реалізував. Практичний метод був розроблений 17 роками пізніше Шаффером і представлений в програмі VEGA (Vector Evaluated Genetic Algorithm). Шаффер [11] модифікував стандартний генетичний алгоритм GENESIS Грефенстіта [10], розроблений для однокритерійним оптимізації таким чином, щоб можна було його застосувати для рішення БО. У цьому алгоритмі селекція виконується згідно з так званим турнірному методу. Причому «найкраща» особина в кожній під популяції обирається на основі своєї функції пристосованості. схематично алгоритм селекції з використанням «турнірного методу» при оптимізації двох функцій представлений на рис. 1., де F1 і F2 означають дві різні функції пристосованості відповідної оптимізується цільовий функції. «Найкраща»

особина з кожної під популяції змішується з іншими особинами і всі інші генетичні операції проводяться аналогічно алгоритму при оптимізації однієї функції.

ОГЛЯД КВАНТОВОГО КОМП'ЮТЕРА. АРХІТЕКТУРА ФОН НЕЙМАНА. АЛГОРИТМИ ШОРА, ГОВЕРА, ДОЙЧА-ЙОЖИ. СФЕРИ ВИКОРИСТАННЯ.

Архітектура та будова квантового комп'ютера

Розвиток теорії квантової механіки та квантова архітектура фон Неймана.

Усе почалося з Альберта Ейнштейна, та його теорії відносності, яка в свою чергу дала потужний імпульс, та фундамент для квантової механіки та квантової теорії поля [1]. Перейдемо до декількох основних понять квантової теорії, які знадобляться для пояснення квантових комп'ютерів. Сам термін кванту [1] запропонував Макс Планк у 1900 році . Квант – це мінімальна, неділима частка енергії, значення якого є дискретним, а не неперервним, як вважалося до цього. Квантова суперпозиція [1] – запропонована Ервін Шредінгером на прикладі відомого кота, який або живий, або мертвий – вводить цьому кванту аналогічну поведінку, що він може перебувати у декількох станах одночасно, що не так легко уявити. Але цей стан стає детермінованим та відомим одразу після вимірювання. Це є квантовим дуалізмом. Наступним явищем, яке спочатку збило з пантелику Альберта Ейнштейна, та який не бажав у це вірити, тому що воно нищило його теорію відносності – це квантова заплутаність [1]. Це явище при якому, два кванта, що знаходяться на будь якій відстані один між одним можуть впливати на один одного. Декогеренція [1] – це процесс порушення точності та цільності квантової системи, шляхом взаємодії квантомеханічної системи з оточуючим всесвітом. Чим менше цього спілкування, тим система буде точнішою.

В основу архітектури квантового комп'ютеру закладено основні принципи з квантової механіки – суперпозиція, квантова заплутаність та декогеренція – а саме її усунення. Задля уникнення великих відмінностей за

основу була взята класична архітектура фон Неймана [2], але дещо змінена і розширена (рис.1).

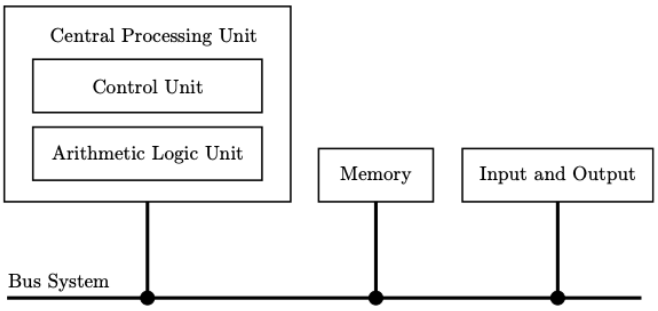


Рис. 1 Класична архітектура фон Неймана [2]

На рис. 2 показана квантова архітектура фон Неймана [2], відмінністю якої є наявність квантових арифметико-логічного пристрою та пам’яті.

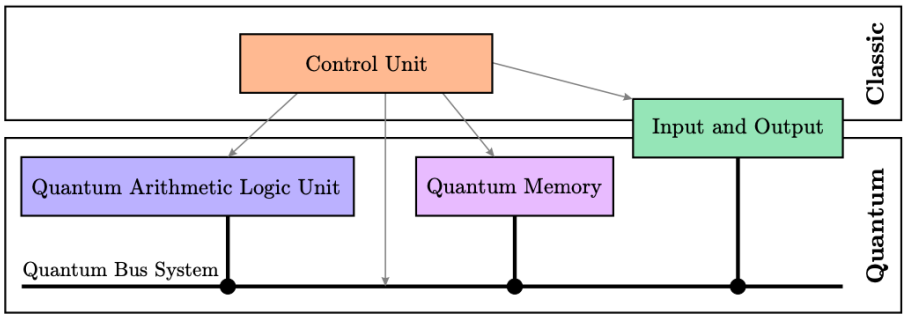


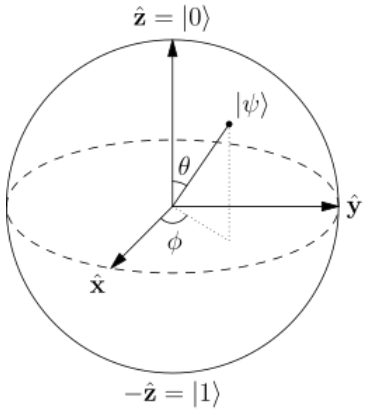
Рис. 2 Квантова архітектура фон Неймана [2]

Принцип дії квантової архітектури фон Неймана подібний до принципу класичної архітектури. Квантова машина Неймана виконує серію квантових операцій затвора, завантажуючи кубіти, якими слід маніпулювати в квантові регістри QALU [2]. Теоретично довжина квантового регістру може бути довільно довгою. Для того, щоб заплутати два кубіти з довільних позицій у пам'яті, два кубіти завантажуються з квантової пам'яті в QALU, де виконуються операції затвора. Після квантових операцій затвора кубіти можуть залишатися в QALU для

подальшої обробки або кубіти повертаються назад у квантову пам'ять. Для виявлення квантових станів необхідну квантову інформацію можна перемістити у вихідну область або область виявлення. Оскільки ця область виявлення працює незалежно від QALU, QIP[2] в QALU і виявлення у вихідній області можуть виконуватися одночасно.

Після виявлення квантового стану кубіти можуть бути переміщені у вхідну область для ініціалізації в один конкретний стан, перш ніж (тепер ініціалізована) квантова інформація може бути переміщена назад у квантову пам'ять. Якщо бажаний початковий стан є більш складним, наприклад, переплутаний стан, ініціалізовані кубіти можуть бути переміщені в QALU, який виконує квантові операції затвора, щоб сформувати бажаний початковий квантовий стан.

Кубіт



Фізичні принципи побудова квантових комп’ютерів

У принципі квантовозарядженого зв’язаного пристрою (QCCD) показаному на рис. 3, сегментована іонна пастка використовується для переміщення іонів у різні положення на пастці шляхом зміни осьово-обмежуючих напруг постійного струму. Це дозволяє використовувати одну

частину пастки як квантову пам'ять, а іншу частину, як зону обробки або QALU.

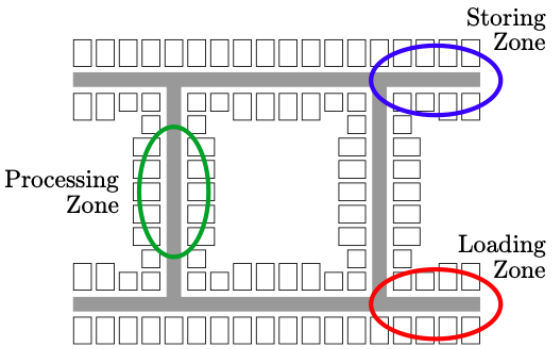


Рис. 3 Будова квантової архітектури з використання іонних газів [2]

Для кубітів основного стану основним джерелом декогерентності є коливання магнітного поля. Тому генерація постійного магнітного поля та магнітна екранізація є найбільш критичними завданнями для досягнення тривалого часу когерентності в захоплених іонних системах. Інші джерела декогеренції, такі як витоки резонансного світла, повинні бути зменшені таким чином, щоб ними можна було знехтувати в квантовій пам'яті.

Оскільки основний шум магнітного поля в лабораторному середовищі надходить від джерел змінного струму, один із способів захистити від магнітного поля змінного струму - використання шкірного ефекту у високопровідному матеріальному оточенні експерименту. Інший спосіб - інкапсулювати експеримент у металевий щит, який забезпечує захист від магнітних коливань змінного та постійного струму. Однак повільні магнітні поля такі дрейфують, оскільки зміни в магнітному полі Землі все ще проникають через магнітний щит, виготовлений з високопровідних матеріал або му-метал. Таким чином, ці прості схеми магнітного екранування не дозволять отримати бажаний час когерентності, наприклад, хоча б декілька годин або днів. На практиці не просто визначити

певну напруженість магнітного поля всередині надпровідника, як це потрібно для тактових переходів у надтонких кубітах під час фазового переходу в надпровідний.

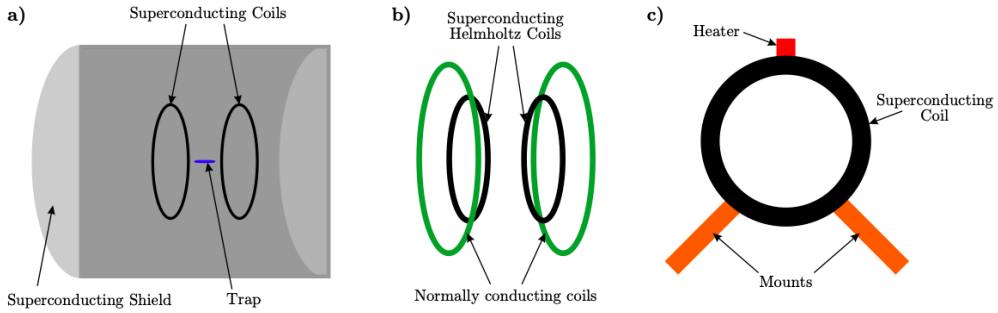


Рис. 4 Прийоми для когеренції квантової системи [2]

Магнітне поле зміщення в положенні іонів може генеруватися надпровідними котушками всередині магнітного екрану, як показано на рис. 4 a. Під час охолодження в середовищі з нульовим полем надпровідні котушки не містять постійного струму. За допомогою додаткових нормально провідних котушок можна генерувати магнітне поле всередині екрану, як показано на мал. 4 b. Коли надпровідні котушки нагріваються локально, як показано на мал. 4 c, згенероване магнітне поле може проникати через надпровідні котушки. Після того, як вони знову охолонуть до надпровідного режиму, магнітне поле, що створюється нормально провідними котушками, може бути вимкнено. Отриманий постійний струм у надпровідних котушках буде генерувати ультрастабільне магнітне поле всередині надпровідного екрану.

Огляд квантових алгоритмів

Розробка квантового обчислювального набору інструментів починається з операцій на найпростішій квантовій системі з усіх - одному кубіті. Нижче на мал.5 представлені можливі квантові гейти – вони і є операторами для обрахування. Однією з основних задач алгоритмів, які

будуються на базі гейтів[3] – це робити корекцію квантових помилок, які виникають внаслідок декогеренції системи. Основними відомими алгоритмами є наступні: Шора, Гровера та Дойча-Йожи.

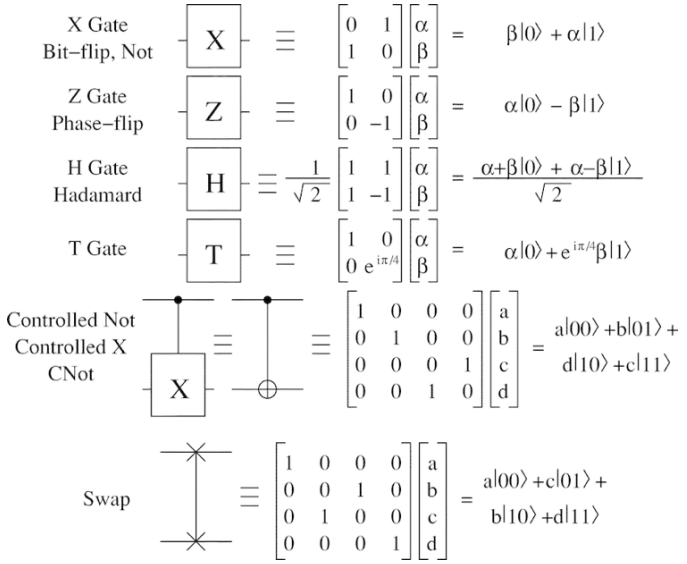


Рис. 5 Квантові Гейти [3]

Основа алгоритму Шора[4]: здатність інформаційних одиниць квантових комп'ютерів - кубітів - приймати кілька значень одночасно і перебувати в стані «квантової заплутаності». Тому він дозволяє проводити обчислення в умовах економії кубітів. Принцип роботи алгоритму Шора можна розділити на 2 частини: перша-класичне зведення розкладання на множники до знаходження періоду деякої функції, друга - квантове знаходження періоду цієї функції.

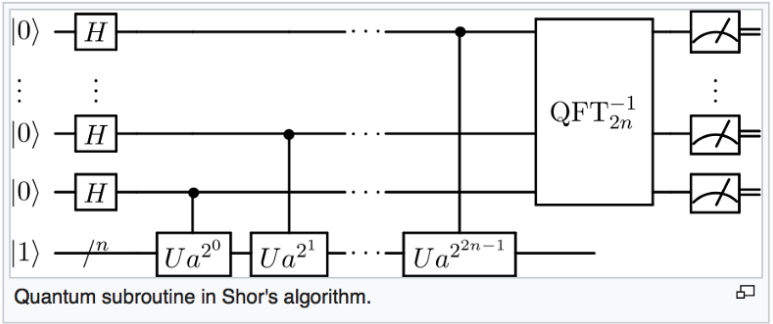


Рис. 6 Схема алгоритму Шора [4]

Алгоритм Гровера [5] може бути використаний для знаходження медіани і середнього арифметичного числового ряду. Крім того, він може застосовуватися для вирішення NP-повних задач шляхом вичерпного пошуку серед безлічі можливих рішень. Це може спричинити значний приріст швидкості в порівнянні з класичними алгоритмами.

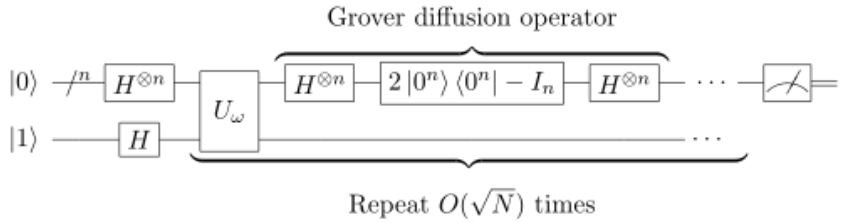


Рис. 7 Схема алгоритму Говера [5]

Задача Дойча – Йожі [6] полягає у визначенні, чи є функція кількох довічних змінних f(x1, x2, x3, ..., xn) постійною (приймає або значення 0, або 1 при будь-яких аргументах) або збалансованою (для половини області визначення приймає значення 0, для іншої половини 1). При цьому вважається апріорі відомим, що функція або є константою, або збалансована.

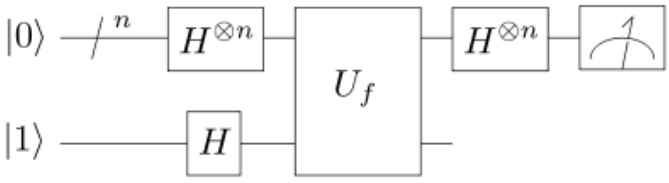


Рис. 8 Схема алгоритму Дойча-Йожи [6]

Прикладне використання

Перший приклад - це точне моделювання молекулярних взаємодій, пошук оптимальних конфігурацій для хімічних реакцій. Така «квантова хімія»[7] настільки складна, що за допомогою сучасних цифрових комп'ютерів можна проаналізувати лише найпростіші молекули. Але

повністю розроблені квантові комп'ютери зможуть без проблем розраховувати навіть такі складні процеси. Або таке точно моделювання [9] молекул дозволяє виробляти більш продуктивні, та якісні зарядні батареї. Чи навіть моделювання фінансових систем задля якісної побудови економіки держав.

Такі обчислювальні можливості навіть можуть нас привести в критичну ситуацію, коли усі існуючі банківські та державні ключи шифрування можуть бути зламані за декілька днів, а ось створити нові використовуючи квантові комп’ютери через їх нестабільність – буде неможливим.

Зараз напрямки квантових комп’ютерів активно розвивають такі компанії як: IBM[8], Microsoft Research, Intel та Google. Microsoft навіть вже створила свою мову для програмування Q# на основі квантових гейтів, IBM в свою чергу навіть створили на своїй сторінці пісочницю – де засобами візуального вибудовування блоків можна створювати схеми на зразок алгоритмів Говера та інших.

Виконання алгоритму Саймона

Проблема Саймона[10], коли маємо функцію невідомої поведінки – чи вертає вона кожний раз різний результат для одних і тих самих вхідних аргументів, або вертає однаковий.

$f(1)\rightarrow 1, f(2)\rightarrow 2, f(3)\rightarrow 3, f(4)\rightarrow 4$

або

$f(1)\rightarrow 1, f(2)\rightarrow 2, f(3)\rightarrow 1, f(4)\rightarrow 2$

Для рішення цієї проблеми, квантові алгоритми гарно підходять.

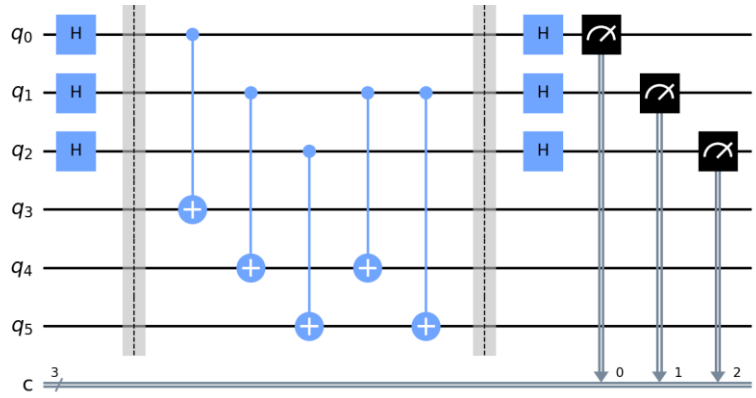


Рис. 9 Схема алгоритму рішення проблеми Саймона [10]

За допомогою опен соурс проекту Qiskit[11], можливо використати необхідний алгоритм, та зробивши симуляцію квантового комп'ютеру також її вирішити. Приклад побудованого алгоритму рис. 10. Результатом його роботи є рис. 11

Квантові комп'ютери надають нам можливість вирішувати усі існуючи проблеми в тисячі разів скоріше. Це однозначно вплине на усі сфери людини, та людству доведеться встигати за технологіями та становитися розумнішими не відстаючи від комп'ютерів та роботів.

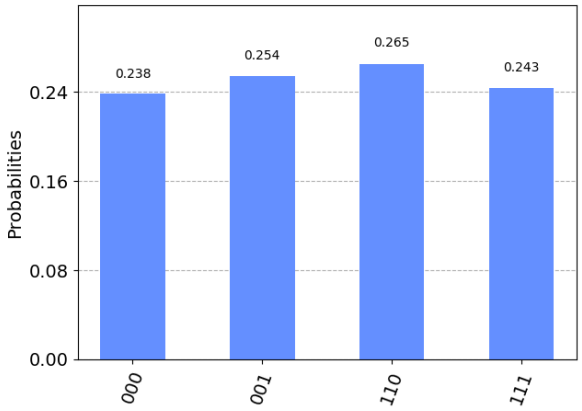


Рис. 9 Результат алгоритму Саймона[10]

Але до поки немає точної відповіді про появу цих комп’ютерів у загальній доступності. Про це говорити важко – через те, що є багато проблем, вирішення яких нам ще тільки доведеться шукати, наприклад та ж сама декогеренція[1].

СПИСОК ЛІТЕРАТУРИ

1. Bohm, David. Quantum theory. Courier Corporation, 2012.
2. Brandl, Matthias F. "A quantum von Neumann architecture for large-scale quantum computing." arXiv preprint arXiv:1702.02583 (2017).
3. DiVincenzo, David P. "Quantum gates and circuits." Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences 454.1969 (1998): 261-276.
4. Beauregard, Stephane. "Circuit for Shor's algorithm using 2n+ 3 qubits." arXiv preprint quant-ph/0205095 (2002).
5. Kwiat, P. G., et al. "Grover's search algorithm: an optical approach." Journal of Modern Optics 47.2-3 (2000): 257-266.
6. Collins, David, K. W. Kim, and W. C. Holton. "Deutsch-Jozsa algorithm as a test of quantum computation." Physical Review A 58.3 (1998): R1633.
7. Terhal, Barbara M. "Quantum supremacy, here we come." Nature Physics 14.6 (2018): 530-531.
8. 8] IBM internet source
9. Steane, Andrew. "Quantum computing." Reports on Progress in Physics 61.2 (1998): 117.
10. Shor P. W. Introduction to quantum algorithms //Proceedings of Symposia in Applied Mathematics. – 2002. – Т. 58. – С. 143-160.
11. https://qiskit.org
12. Бураков М. В. Генетический алгоритм: теория и практика / М. В.

Бураков. – Санкт-Петербург: ГУАП, 2008. – 168 с.

13. Holland J. H. Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence/J. H. Holland. — The MIT Press, Cambridge, 1992. — ISBN 0262581116
14. Darrel Whitley: A Genetic Algorithm Tutorial; November 10, 1993; Technical Report CS-93-103 (Revised); Department of Computer Science, Colorado State University, Fort Collins, US
15. Cohoon J. P., Paris W.D. Genetic Placement. - IEEE Trans. on CAD, Vol.6, No 6, November, Publisher, 198 - 216 p.
16. Гладков Л. А., Курейчик В. В., Курейчик В. М. Генетические алгоритмы: Учебное пособие. — 2-е изд. — М: Физматлит, 2006. — 320 с. — ISBN 5-9221-0510-8.
17. K. Deb and S. Agrawal, Understanding Interactions Among Genetic Algorithm Parameters, 1998.
18. Eshelman L (1991) The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. In: Rawlins G (ed) FOGA – 1, Morgan Kaufmann, Los Altos, CA, pp 265–283.
19. GAVaPS – a genetic algorithm with varying population size. In Proc. of the First IEEE Conf. on Evolutionary Computation, Piscataway, NJ, 1994. IEEE Press 73-78.
20. Grefenstette J.J. GENESIS: A system for using genetic search procedures.Proceedings of the 1984 Conference on Intelligent Systems and Machines, 161–165 p.
21. Schaffer J.D. Multiple objective optimization with vector evaluated genetic algorithm // J.J. Grefenstete (Kd.): Genetic Algorithms and Their Applications. Proc. of the First Int. Conf. on Genetic Algorithms, Hillsdale, NJ: L. Eribaum. — 1985. — P. 93–100.
22. Подиновский В.В., Ногин В.Д. Парето-оптимальные решения

многокритериальных задач. — М.: Наука. — 1982. — 254 с.

23. Y. E. Ioannidis and Y. Kang. “Randomized algorithms for optimizing large join queries” In Proc. of the 1990 ACM-SIGMOD Conference on the Management of Data, pages 312–321, Atlantic City, NJ, May 1990.
24. A. Swami and A. Gupta. “Optimization of large join queries” In Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data, pages 8–17, Chicago, IL, June 1988.
25. Ioannidis, Yannis E., and Younkyung Cha Kang. "Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization." ACM SIGMOD Record 20.2 (1991).

Додаток 1. Приклад виконання наукової роботи.

Кузьмич Валентин

ГІБРИДНІ ТОПОЛОГІЇ НА ОСНОВІ ПЕРЕТВОРЕНЬ ДЕБРУЙНА

Актуальність теми дослідження. Покращення відмовостійкості топологій розподілених обчислювальних систем є одним з найважливіших завдань в проектуванні таких мереж. Одним з перспективних методів його вирішення є метод синтезу відмовостійких топологій, що дозволяє апаратно забезпечити високий рівень відмовостійкості систем.

Постановка проблеми. Топологічна структура це важлива складова розподілених обчислювальних систем. Структура топології є головним фактором відмовостійкості обчислювальної системи. Ця стаття пропонує огляд нових відмовостійких гібридних топологій на основі перетворень де Бруйна.

Аналіз останніх досліджень і публікацій. Наразі існує ряд робіт, присвячених топологіям, на базі перетворень де Бруйна. Був здійснений доклданий огляд гібридних топологій на основі де Бруйна та гіперкубу. Наразі графи на основі послідовностей де Бруйна мають широке застосування в генетиці та біоінформатиці.

Виділення не вирішених раніше частин загальної проблеми. Однією з широко розповсюджених модифікацій дерева є товсте дерево, показники якого дозволяють виконувати масштабування відмовостійких топологій. Проте раніше не були розглянуті варіанти гібридних топологій на основі декартового добутку графів на основі перетворень де Бруйна та стандартних топологій – зірка, меш, гіперкуб.

Мета статті. Метою дослідження є розгляд та аналіз можливостей використання декартового добутку графів на основі перетворень де Бруйна та стандартних топологій для синтезу відмовостійких топологій, створення прикладів цих топологій та їх аналіз

Виклад основного матеріалу.

1. Граф де Бруйна

В роботі використаний граф де Бруйна який створюється за допомогою зсувів вліво двійкових кодів вузлів. При формуванні нової послідовності, старша цифра витискається (зникає), молодша цифра після зсуву отримує значення 0 або 1. Отримані таким чином вузли мають зв’язок з вузлом, від номера якого зроблено зсув. В якості прикладу, візьмемо наступну таблицю формування послідовності Де Бруйна для трьохзначних чисел (ранг 3).

Таблиця 1

Таблиця формування дебруйнівських послідовностей

000		**001**		**010**		**011**	
001	000	010	011	100	101	110	111
100		**101**		**110**		**111**	
000	001	010	011	100	101	110	111

Таким чином утворюється послідовність Де Бруйна. Для масштабування графу, додається ще один розряд, що збільшує кількість вузлів в два рази. Потрібно зазначити, що існуючий граф можна повторно використати для наступних кроків масштабування. При цьому потрібно продублювати граф два рази, і додати нові зв’язки, тому що при збільшенні

кількості розрядів, дебруйнівські зсуви будуть генерувати нові значення, які розширяють попердній етап масштабування.

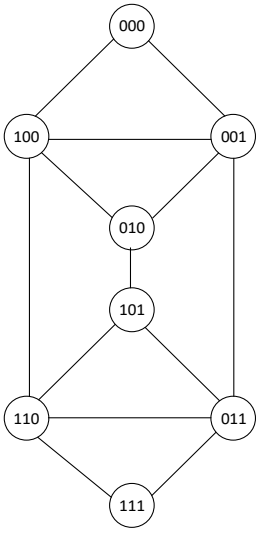


Рис. 1. Топологія де Бруйна

Таблиця 2

Характеристики топології де Бруйна рангу r

Параметр (r – ранг)	**К-сть вершин**	**Ступінь**	**Діаметр**	**Ціна**
Рівняння зросту	2^r	4	r	$4*2^r*r$

Для генерації гібридних топологій були обрані двійкові графи де Бруйна, за рангами від 2 до 12

2. Стандартні топології

В якості другого компоненту гібридної топології були використані наступні:

меш рангу 2 (4 вузли в кластері), гіперкуб рангу 3 і 4 (8 і 16 вузлів), та зірка ранку 7 (8 вузлів) – рисунки 2-5.

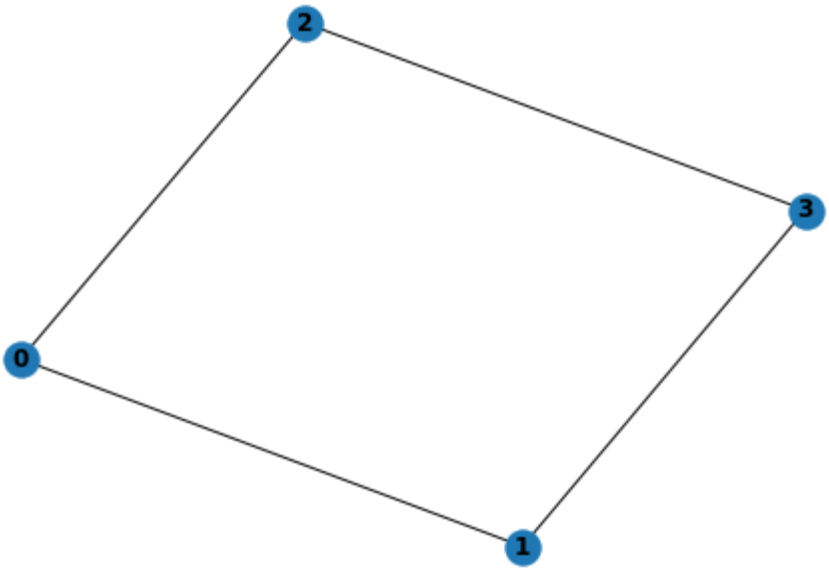


Рис. 2 – меш рангу 2

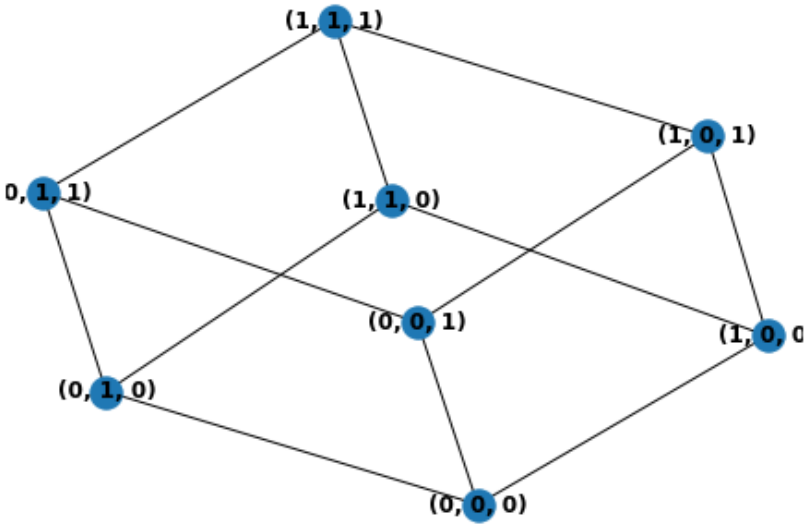


Рис.3 – гіперкуб рангу 3

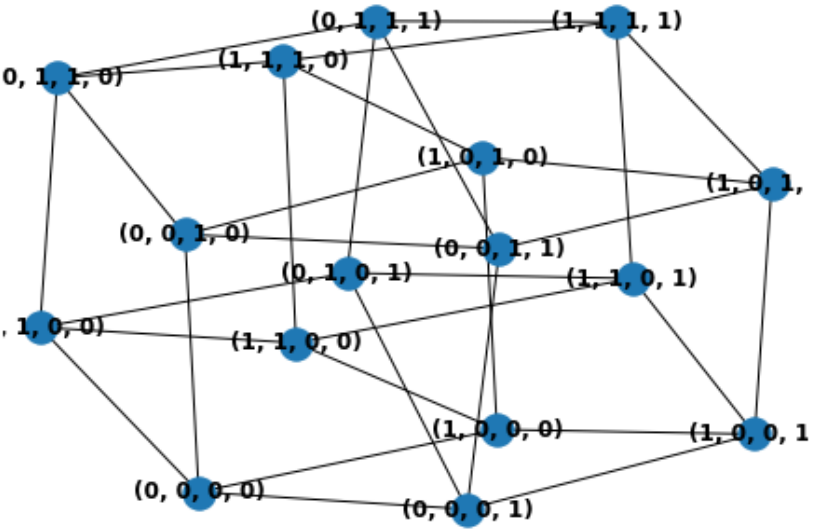


Рис. 4 – гіперкуб рангу 4

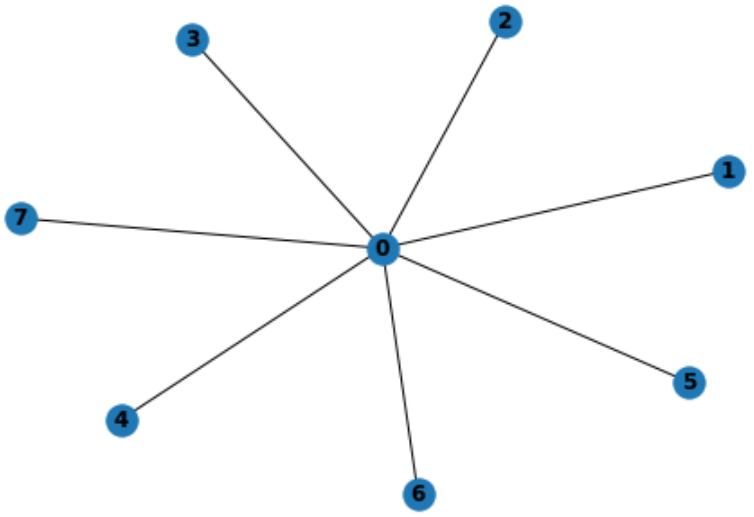


Рис. 5 – зірка рангу 7

3. Гібридна топологія

Остаточна гібридна топологія формується шляхом декартового добутку графу де Бруйна, із заданим рангом, на одну із заданих стандартних топологій.

Приклад декартового добутку меш рангу 2 та графу де Бруйна рангу 3 зображений на рис. 6

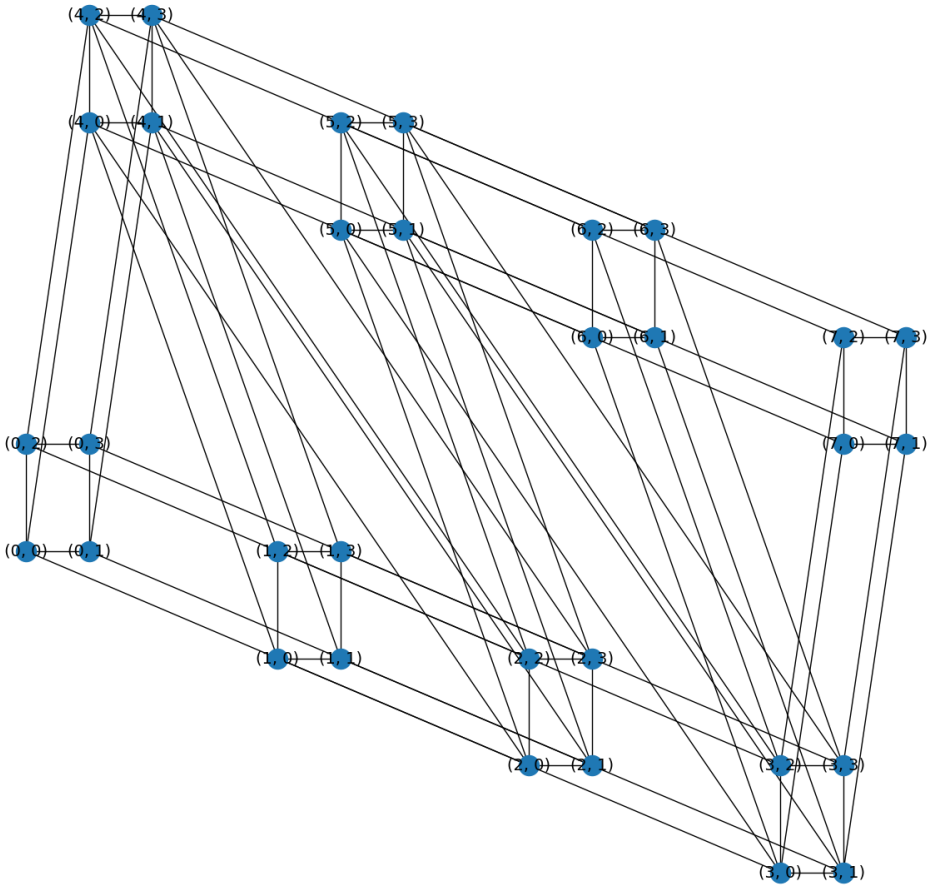


Рис. 6 – приклад гібридної топології

Оскільки кількість вузлів в поєднуваних топологіях кратна степені 2, то відповідно кількість вузлів у гібридних топологіях також буде кратна степені 2, що значно спрощує їх порівняння одне з одним.

4. Результати

В якості метрик, за якими був здійснений порівняльний аналіз, були обрані топологічний трафік, вартість та SD-параметр (діаметер*ступінь). Графіки залеженості кожної з метрик від кількості вузлів в топології зображені на рис. 7-9

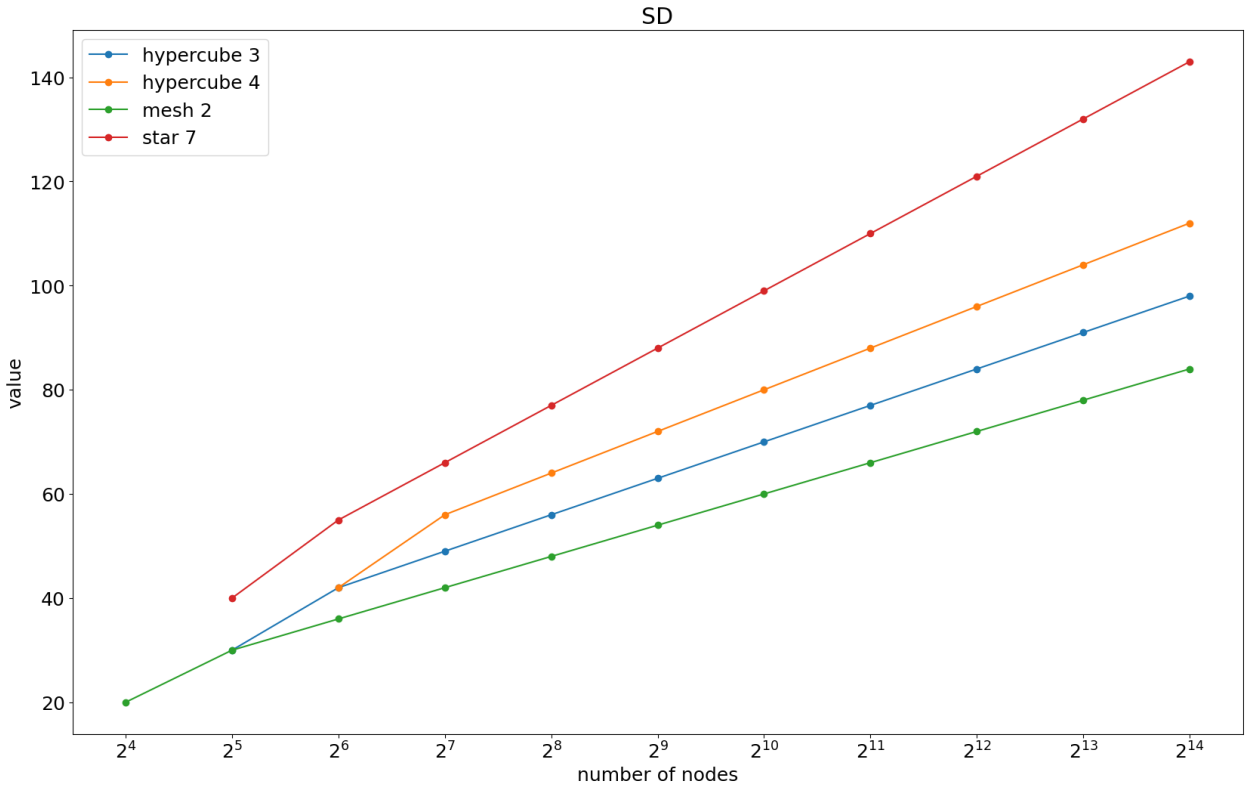


Рис. 7 – SD-параметер

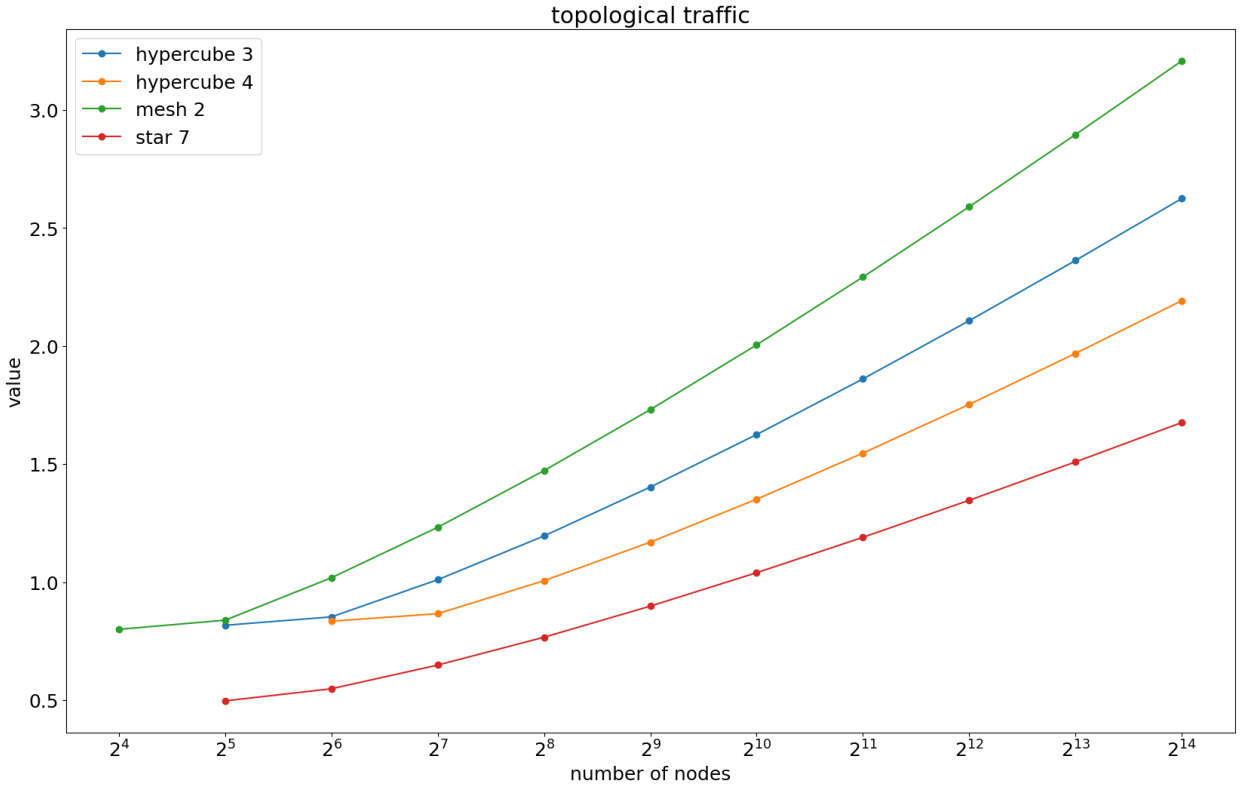


Рис. 8 – Топологічний трафік

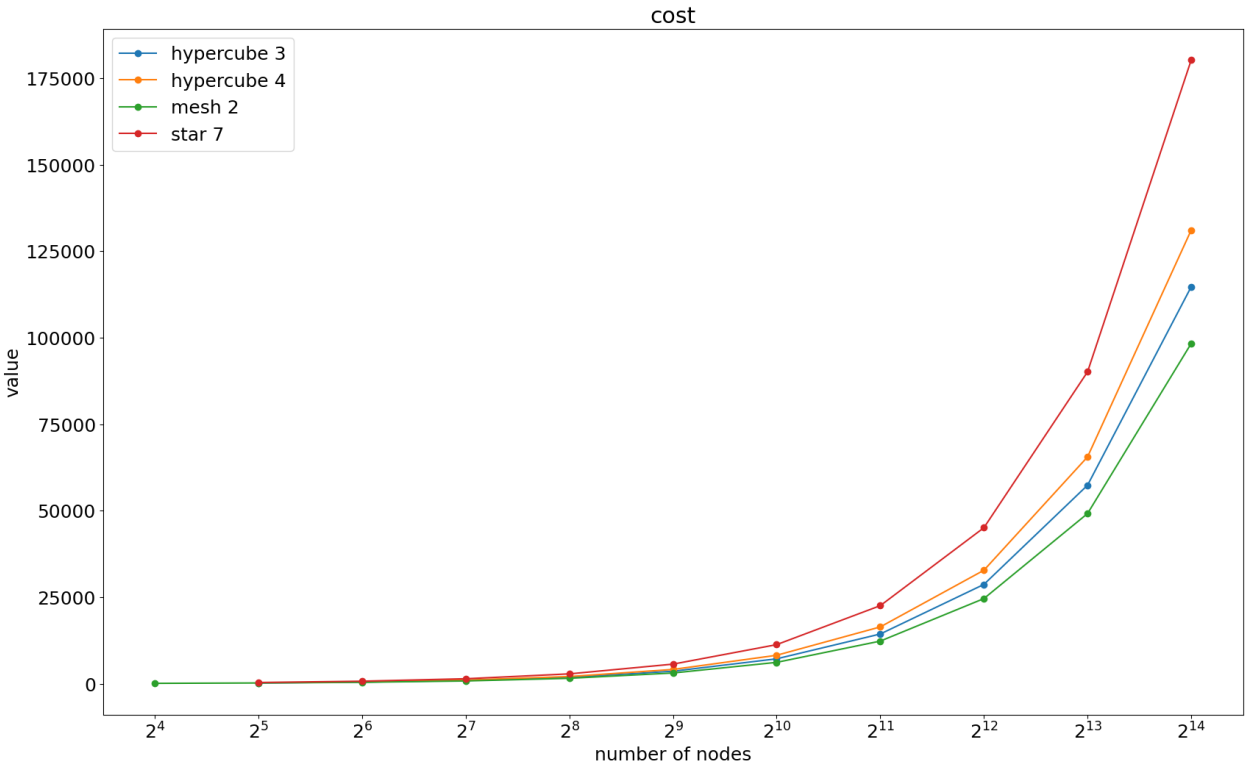


Рис. 9 – Варітсть

Таблиця 3

Характеристики досліджених топологій

nodes	degree	diameter	avg_diameter	topological_traffic	cost	SD	cluster	rank	cluster_size
16	5	4	2	0,8	80	20	mesh	2	2
32	6	5	2,451613	0,817204	192	30	hypercube	2	3
32	6	5	2,516129	0,83871	192	30	mesh	3	2
32	10	4	2,483871	0,496774	320	40	star	2	7
64	7	6	2,920635	0,834467	448	42	hypercube	2	4
64	7	6	2,984127	0,852608	448	42	hypercube	3	3
64	6	6	3,055556	1,018519	384	36	mesh	4	2
64	11	5	3,015873	0,548341	704	55	star	3	7
128	8	7	3,464567	0,866142	1024	56	hypercube	3	4
128	7	7	3,535433	1,010124	896	49	hypercube	4	3
128	6	7	3,69685	1,232283	768	42	mesh	5	2
128	11	6	3,566929	0,648533	1408	66	star	4	7
256	8	8	4,023529	1,005882	2048	64	hypercube	4	4
256	7	8	4,184314	1,195518	1792	56	hypercube	5	3
256	6	8	4,416667	1,472222	1536	48	mesh	6	2
256	11	7	4,215686	0,766488	2816	77	star	5	7
512	8	9	4,677104	1,169276	4096	72	hypercube	5	4
512	7	9	4,909002	1,402572	3584	63	hypercube	6	3
512	6	9	5,192025	1,730675	3072	54	mesh	7	2
512	11	8	4,940313	0,898239	5632	88	star	6	7
1024	8	10	5,404692	1,351173	8192	80	hypercube	6	4
1024	7	10	5,687439	1,624983	7168	70	hypercube	7	3

1024	6	10	6,014266	2,004755	6144	60	mesh	8	2
1024	11	9	5,718719	1,039767	11264	99	star	7	7
2048	8	11	6,184905	1,546226	16384	88	hypercube	7	4
2048	7	11	6,511572	1,860449	14336	77	hypercube	8	3
2048	6	11	6,876298	2,292099	12288	66	mesh	9	2
2048	11	10	6,542837	1,189607	22528	110	star	8	7
4096	8	12	7,010104	1,752526	32768	96	hypercube	8	4
4096	7	12	7,374741	2,107069	28672	84	hypercube	9	3
4096	6	12	7,768943	2,589648	24576	72	mesh	10	2
4096	11	11	7,405998	1,346545	45056	121	star	9	7
8192	8	13	7,873901	1,968475	65536	104	hypercube	9	4
8192	7	13	8,268055	2,362302	57344	91	hypercube	10	3
8192	6	13	8,685859	2,895286	49152	78	mesh	11	2
8192	11	12	8,299309	1,508965	90112	132	star	10	7
16384	8	14	8,767581	2,191895	131072	112	hypercube	10	4
16384	7	14	9,18536	2,624389	114688	98	hypercube	11	3
16384	6	14	9,621687	3,207229	98304	84	mesh	12	2
16384	11	13	9,216612	1,675748	180224	143	star	11	7

Результати. Характеристики всіх досліджених топологій показані в таблиці 4. Проаналізувавши отримані дані параметрів синтезованих топологій, можна зробити наступні висновки:

- SD параметр починає зростати лінійно у всіх видів топологій, починаючи з 128 вузлів, що дає змогу передбачити кількість зв'язків на кожному ранзі графу де Бруйна;
- Зі зростанням кількості вузлів, топологічний трафік збільшується, і починає перевищувати 1, що може спричинити затримки в передачі повідомлень між вузлами.
- З точки зору пропускної спроможності, найефективнішою топологією є **зірка рангу 7**, найбільша кількість вузлів, при якій топологічний трафік не більший ніж 1 – **512**; далі йдуть гіперкуб рангу 4, гіперкуб рангу 3 та меш рангу 2
- З точки зору вартості мережі, порядок протилежний – найоптимальнішою є топологія з кластером меш рангу 2, далі гіперкуб рангу 3, гіперкуб рангу 4 та зірка рангу 7

Висновки і пропозиції. Таким чином, найефективнішою топологією з усіх досліджених є декартовий добуток зірки та графу де Бруйна, при буль-якій кількості вузлів. Однак, вона ж є і найдорожчою.

В майбутніх дослідженнях пропонується дослідіти трійкове та інші кодування графу де Бруйна, та інші поєднання таких графів із стандартними топологіями, що дозволить досягти ефективної пропускної спроможності із меншою вартістб топології.

Код:

```
import multiprocessing as mp

import networkx as nx
import numpy as np
import pandas as pd


# from .clusters import (generate_hypercube, generate_mesh_cluster,
#                        generate_star, generate_ring_cluster)


def generate_mesh_topology(rank=3, mesh_size=3):
    topo = nx.Graph()
    mesh = generate_mesh_cluster(size=mesh_size)
    pos = {}
    rows = 2 ** (rank // 2)
    cols = 2 ** ((rank + 1) // 2)
    H = 5
    h = H / (mesh_size + 2)
    for num_clust in range(2**rank):
        cl_row = num_clust // (2**rank // rows)
        cl_col = num_clust % cols
        clust = mesh.copy()
        new_labels = {}
        for i, node in enumerate(clust):
            new_labels[node] = (num_clust, i)
            x = cl_col * H + (i % mesh_size + 1) * h + cl_row * h
            y = cl_row * H + (i // mesh_size + 1) * h + (cols - cl_col - 1) * h
            pos[(num_clust, i)] = (x, y)
        nx.relabel_nodes(clust, new_labels, copy=False)
        topo = nx.compose(topo, clust)
    mask = 2**rank - 1
    for num_clust in range(2**rank):
        n1 = (num_clust << 1) & mask
        n2 = n1 + 1
        clust_edges = []
        if n1 != num_clust:
            clust_edges.append((num_clust, n1))
        if n2 != num_clust:
            clust_edges.append((num_clust, n2))
        for c1, c2 in clust_edges:
            for j in range(len(mesh)):
                edge = ((c1, j), (c2, j))
                if edge not in topo.edges:
                    topo.add_edge(*edge)
    return topo, pos


def generate_hybrid_topo(rank=3, cluster_func=None, cluster_size=3):
    if cluster_func is None:
        cluster_func = generate_mesh_cluster
    cluster = cluster_func(size=cluster_size)
    topo = nx.Graph()
```

```
    for num_clust in range(2**rank):
        clust = cluster.copy()
        new_labels = {}
        for i, node in enumerate(clust):
            new_labels[node] = (num_clust, i)
        nx.relabel_nodes(clust, new_labels, copy=False)
        topo = nx.compose(topo, clust)
    mask = 2**rank - 1
    for num_clust in range(2**rank):
        n1 = (num_clust << 1) & mask
        n2 = n1 + 1
        clust_edges = []
        if n1 != num_clust:
            clust_edges.append((num_clust, n1))
        if n2 != num_clust:
            clust_edges.append((num_clust, n2))
        for c1, c2 in clust_edges:
            for j in range(len(cluster)):
                edge = ((c1, j), (c2, j))
                if edge not in topo.edges:
                    topo.add_edge(*edge)
    return topo


def topo_stats(topo):
    topo_stats = {}
    N = len(topo.nodes)
    topo_stats['nodes'] = N
    S = max(n[-1] for n in topo.degree)
    topo_stats['degree'] = S
    D = nx.algorithms.distance_measures.diameter(topo)
    topo_stats['diameter'] = D
    avg_D = nx.average_shortest_path_length(topo)
    topo_stats['avg_diameter'] = avg_D
    topo_stats['topological_traffic'] = 2 * avg_D / S
    topo_stats['cost'] = N * S
    topo_stats['SD'] = S * D
    return topo_stats


def generate_hybrid_topo_ranks(n, cluster_nodes):
    div = np.log2(n / cluster_nodes)
    return [int(np.ceil(div)), int(np.floor(div))]


def generate_target_topos(n: int, cluster_params: dict):
    total_stats = []
    args = []
    cluster_types = []
    for cluster_type, cl_params in cluster_params.items():
        func = cl_params['func']
        sizes = cl_params['sizes']
        for size in sizes:
            c = func(size=size)
```

```
            cluster_nodes = len(c.nodes)
            ranks = generate_hybrid_topo_ranks(n, cluster_nodes)
            for rank in ranks:
                cluster_types.append(cluster_type)
                args.append((rank, func, size))
    with mp.Pool(mp.cpu_count()) as p:
        topos = p.starmap(generate_hybrid_topo, args)
        print(f'Genenated {len(topos)}')
        total_stats = list(p.map(topo_stats, topos))
        print('Calculated stats')
        for i, ct in enumerate(cluster_types):
            total_stats[i]['cluster'] = ct
            total_stats[i]['rank'] = args[i][0]
            total_stats[i]['cluster_size'] = args[i][2]
    return pd.DataFrame(total_stats)


def graph_cartesian_product(upper_topo, cluster_topo):
    return nx.cartesian_product(upper_topo, cluster_topo)
```

```
from pathlib import Path

from . import generate_target_topos
from .clusters import generate_mesh_cluster, generate_hypercube, generate_star
from .clusters import generate_ring_cluster


if __name__ == "__main__":
    params = {'mesh': {'func': generate_mesh_cluster, 'sizes': [2, 3, 4]},
              'hypercube': {'func': generate_hypercube, 'sizes': [2, 3, 4]},
              'ring': {'func': generate_ring_cluster,
                       'sizes': [i for i in range(3, 11, 2)]},
              'star': {'func': generate_star,
                       'sizes': [i for i in range(3, 11, 2)]}}
    N = 8000
    cur_file = Path(__file__)
    stats_df = generate_target_topos(N, params)
    stats_df.to_csv(str(cur_file.parent.parent / f'{N}.json'))
```

```
import networkx as nx
import numpy as np


def generate_mesh_cluster(size=3):
    cluster = nx.grid_2d_graph(size, size)
    return cluster


def generate_ring_cluster(size=3):
    return nx.circulant_graph(size, [1])
```

```
def generate_hypercube(size=3):
    return nx.hypercube_graph(size)


def generate_star(size=3):
    return nx.star_graph(size)


def generate_de_bruijne(size=3):
    mask = 2**size-1
    arr = np.arange(2**size)
    dst1 = np.left_shift(arr, 1) & mask
    dst2 = (dst1+1) & mask
    edges = np.vstack((np.vstack((arr, dst1)).T, np.vstack((arr, dst2)).T))
    topo = nx.Graph()
    topo.add_edges_from(edges)
    return topo
```