ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Методика підвищення ефективності багатопотокових Java-програм для високонавантажених систем»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Едуард ПЕТРЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Едуард ПЕТРЕНКО

Керівник: Сергій КОМПАН
к.ф.-м.н., доцент

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

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Петренко Едуарду Денисовичу

1. Тема кваліфікаційної роботи: «Методика підвищення ефективності багатопотокових Java-програм для високонавантажених систем»

керівник кваліфікаційної роботи Сергій КОМПАН, к.ф.-м.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, параметри методу багатопотокової обробки даних, вимоги до системи використання обробки високонавантаженої системи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження методів підвищення ефективності багатопотокових Java-програм.
2. Аналіз технологій використання GPU замість CPU в Java.

3. Розробка методу підвищення ефективності багатопотокових Java-програм для високонавантажених систем.

5. Перелік ілюстративного матеріалу: *презентація*

1. Мета, об’єкт та предмет дослідження
2. Аналіз методів для багатопоточної обробки.
3. Математична модель Hybrid Compute Method.
4. Алгоритм Hybrid Compute Method.
5. Діаграма класів.
6. Порівняння HCM з OpenGL та Forkjoinpool.
7. Порівняння результатів.
8. Висновки
9. Публікації та апробація роботи

6. Дата видачі завдання «16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-23.10.2024	
2	Вивчення матеріалів для аналізу багатопоточної обробки даних	24.10-27.10.2024	
3	Дослідження методів обробки даних на мові Java	28.10-31.10.2024	
4	Аналіз особливостей використання GPU в багатопотокових Java-програмах	01.11-03.11.2024	
5	Дослідження технологій впровадження відеокарт в Java-код.	04.11-09.11.2024	
6	Застосування технологій обробки даних з допомогою GPU та CPU	10.11-13.11.2024	
7	Оформлення роботи: вступ, висновки, реферат	14.11-19.11.2024	
8	Розробка демонстраційних матеріалів	20.11-24.11.2024	
9	Попередній захист роботи	29.11.2024	

Здобувач вищої освіти _____ Едуард ПЕТРЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Сергій КОМПАН
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 84 стор., 11 табл., 27 рис., 34 джерел.

Мета роботи – підвищення ефективності обробки інформації в багатопотокових Java-програмах для високонавантажених систем за рахунок використання GPU.

Об'єкт дослідження – процес обробки багатопотокових Java-програм для високонавантажених систем.

Предмет дослідження – методи багатопотокової обробки Java-програм для високонавантажених систем.

У роботі проведено аналіз багатопотокових підходів до обробки даних у Java-програмах для високонавантажених систем. На основі аналізу розроблено методику, яка поєднує використання CPU та GPU для підвищення продуктивності. Було запропоновано математичну модель та алгоритм роботи методу, а також проведено тестування ефективності запропонованого рішення. Отримані результати підтверджують доцільність використання методу в задачах із великими обсягами даних.

КЛЮЧОВІ СЛОВА: БАГАТОПОТОЧНІСТЬ, ВИСОКОНАВАНТАЖЕНІ СИСТЕМИ, JAVA, GPU, CPU, ОБРОБКА ДАНИХ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ,ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ.

ABSTRACT

Text part of the master's qualification work: 84 pages, 27 pictures, 11 table, 34 sources.

The purpose of the work - improving the efficiency of data processing in multithreaded Java applications for high-load systems through the use of GPU.

Object of research – the process of data processing in multithreaded Java applications for high-load systems.

Subject of research – methods of multithreaded data processing in Java applications for high-load systems.

Summary of the work: The study analyzes multithreaded approaches to data processing in Java applications for high-load systems. Based on the analysis, a methodology was developed that combines the use of CPU and GPU to improve performance. A mathematical model and an algorithm for the method were proposed, and the efficiency of the proposed solution was tested. The results confirm the feasibility of using the method for tasks involving large data volumes.

KEYWORDS: MULTITHREADING, HIGH-LOAD SYSTEMS, JAVA, GPU, CPU, DATA PROCESSING, PARALLEL COMPUTATIONS, PERFORMANCE OPTIMIZATION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

GPU - Graphics Processing Unit

CPU - Central Processing Unit

HCM - Hybrid Compute Method

OpenCL - Open Computing Language

OpenGL - Open Graphics Library

JVM - Java Virtual Machine

API - Application Programming Interface

CUDA - Compute Unified Device Architecture

RAM - Random Access Memory

ВСТУП

Розвиток сучасних інформаційних технологій супроводжується постійним зростанням обсягів даних і підвищенням вимог до продуктивності програмного забезпечення. Високонавантажені системи, такі як хмарні обчислення, серверні додатки та системи аналізу великих даних, стикаються з викликами, пов’язаними із забезпеченням ефективної обробки інформації в умовах зростаючих обсягів запитів.

Особливу увагу привертають методи багатопотокового програмування, які дозволяють ефективніше використовувати апаратні ресурси для розподілу задач. Водночас можливості сучасних графічних процесорів (GPU), здатних обробляти значну кількість паралельних потоків, залишаються недооціненими у контексті багатопотокових Java-програм. Інтеграція GPU у процес обробки даних Java-додатків дозволяє суттєво підвищити продуктивність системи, особливо в задачах із великими обсягами обчислень.

Мета роботи – підвищення ефективності обробки інформації в багатопотокових Java-програмах для високонавантажених систем за рахунок використання GPU.

Об’єкт дослідження – процес обробки багатопотокових Java-програм для високонавантажених систем.

Предмет дослідження – методи багатопотокової обробки Java-програм для високонавантажених систем.

Наукова новизна дослідження полягає у розробці нового методу HCM (Hybrid Compute Method), який поєднує можливості CPU та GPU для розподілу задач і автоматизації їх виконання у багатопотоковому середовищі. Запропонована методика базується на сучасних технологіях, таких як OpenCL, і адаптована для використання у високонавантажених системах.

Практична значущість отриманих результатів полягає у можливості застосування методу HCM у реальних програмних продуктах, що дозволить

забезпечити високу продуктивність і масштабованість у задачах аналізу великих даних, фінансових обчислень, моделювання та інших сферах.

Основні завдання роботи:

1. Провести аналіз існуючих підходів до багатопотокової обробки даних у Java-програмах.
2. Розробити математичну модель і алгоритм роботи методу HCM.
3. Провести тестування запропонованого методу на різних обсягах даних.
4. Здійснити порівняльний аналіз HCM із іншими методами (ForkJoinPool, OpenGL) та оцінити його ефективність.
5. Сформулювати рекомендації щодо застосування методу HCM у високонавантажених системах.

Таким чином, виконання даної роботи спрямоване на вирішення актуальної задачі підвищення продуктивності багатопотокових Java-програм, що забезпечить ефективне використання ресурсів сучасних обчислювальних систем.

1 АНАЛІЗ ПРЕДМЕТРОЇ ГАЛУЗІ

1.1 Мікросервіси та їх роль у розробці програм

Мікросервіси — це архітектурний підхід, у якому велика система поділяється на низку незалежних, але взаємопов'язаних сервісів. Кожен сервіс відповідає за виконання конкретної задачі, що дає змогу розвивати, масштабувати та підтримувати систему більш ефективно. Однією з основних переваг такої архітектури є можливість незалежного масштабування кожного мікросервісу, що дозволяє оптимізувати використання ресурсів і підвищує продуктивність системи. Крім того, мікросервіси дозволяють розробляти окремі компоненти з використанням різних технологій, що дає більшу гнучкість у виборі інструментів для кожного завдання.

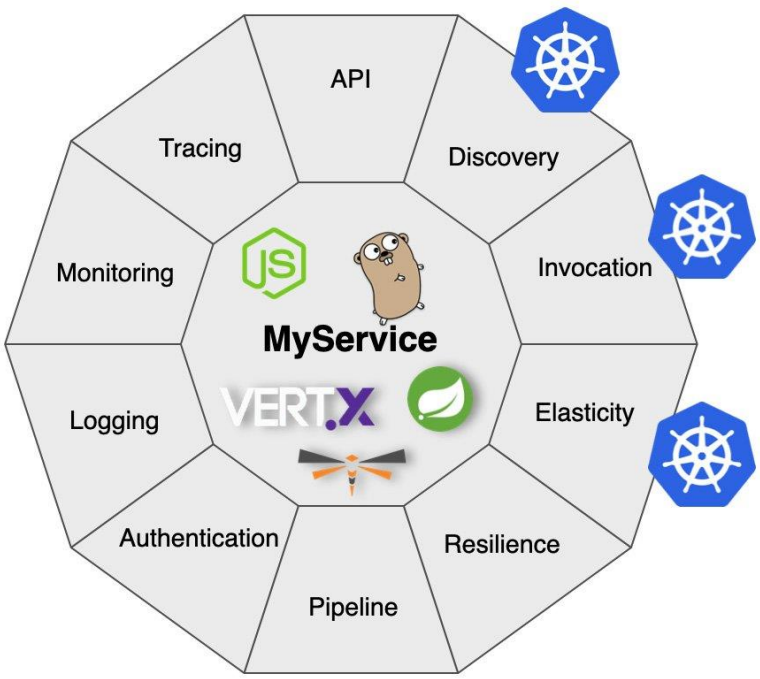


Рис. 1.1 Зображення мікросервісної архітектури

Серед основних переваг мікросервісної архітектури можна виділити:

- Масштабованість: кожен сервіс можна масштабувати окремо, що дозволяє ефективно управляти навантаженням у системі.
- Незалежність компонентів: мікросервіси можна оновлювати, тестувати і розгортати окремо, без необхідності переробляти інші частини системи.
- Вибір технологій: кожен сервіс може бути реалізований на найкращій технології для його задачі, що дозволяє зберігати ефективність кожного компонента на всіх етапах розробки.
- Простота у використанні CI/CD: мікросервіси знижують складність інтеграції і доставки, оскільки кожен сервіс є незалежним і має свою власну схему тестування і деплойменту.

Зазначені переваги роблять мікросервіси ідеальним вибором для великих, високонавантажених систем, де важливі масштабованість і гнучкість. Однак вони також мають свої недоліки:

- Складність управління: зростання кількості сервісів може ускладнити їх моніторинг, тестування та загальну координацію.
- Мережеві затримки: оскільки мікросервіси взаємодіють через мережу, можуть виникати проблеми із затримками, що негативно впливає на швидкість роботи системи.
- Управління станом і транзакціями: розподілені сервіси можуть ускладнити управління станом і обробку транзакцій.
- Проблеми безпеки: кожен новий мікросервіс створює потенційний вектор для атак, що вимагає додаткових зусиль для захисту системи.

Для вирішення цих проблем часто використовуються такі інструменти, як Kubernetes для автоматизації оркестрації контейнерів і Istio для забезпечення надійної взаємодії між сервісами. Istio дозволяє вирішити проблему з балансуванням навантаження, маршрутизацією запитів, моніторингом, безпекою та відновленням після збоїв, що особливо важливо для великих систем [1].

Одним з найбільш поширених підходів для створення мікросервісів на Java є використання фреймворків Spring Boot і Spring Cloud. Ці технології дозволяють швидко створювати масштабовані й безпечні мікросервіси, що можуть ефективно працювати в розподілених середовищах. Spring Boot надає інструменти для автоматизації конфігурації та запуску сервісів, а Spring Cloud забезпечує зручне керування компонентами системи, такими як балансування навантаження, виявлення сервісів, захист і керування станом. Завдяки таким інструментам, мікросервіси можна легко інтегрувати і використовувати в сучасних високонавантажених системах [2].

Завдяки мікросервісній архітектурі можна досягти високої гнучкості, масштабованості та надійності навіть у складних проєктах, де важлива ефективна організація роботи команд і швидкість впровадження нових функцій.

1.2 Багатопоточність в Java

Багатопоточність є важливим інструментом для створення високопродуктивних і масштабованих програм у Java. Вона дозволяє ефективно використовувати можливості багатоядерних процесорів, дозволяючи одночасно виконувати кілька задач. Це зменшує час очікування користувачів та підвищує загальну продуктивність системи. Однак розробка багатопотокових програм вимагає знання специфічних аспектів, таких як синхронізація, керування потоками та вирішення проблем, пов'язаних із конкурентним доступом до ресурсів.

1.2.1 Основи багатопоточності в Java

Java забезпечує базові можливості для багатопоточності через клас Thread та інтерфейс Runnable, які дозволяють створювати та керувати потоками. Клас Thread надає методи для ініціалізації та управління потоком, в той час як Runnable використовується для визначення задачі, яку потік повинен виконати. Традиційно для створення багатопоточних програм у Java застосовують ці класи, однак існують і більш сучасні підходи, які дозволяють полегшити роботу з потоками.

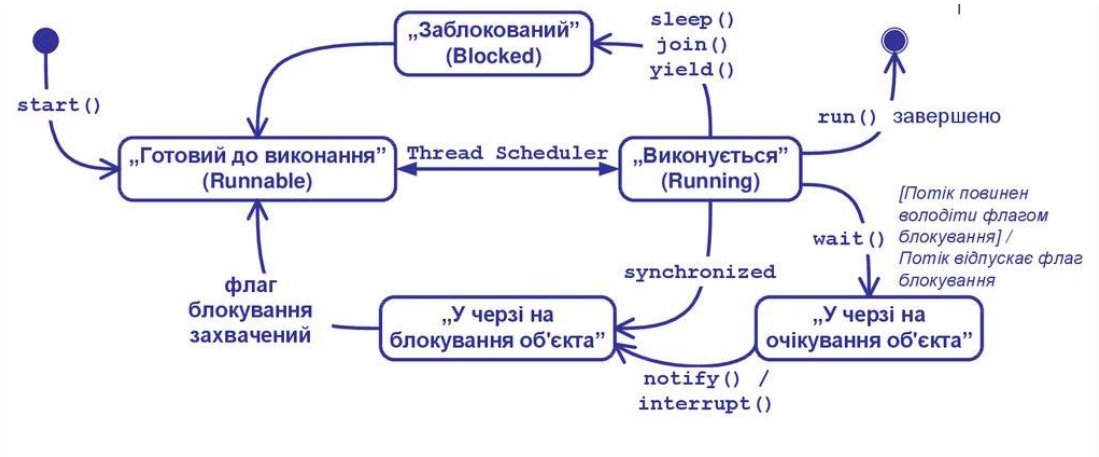


Рис 1.2 Діаграма станів потоку з врахуванням wait-notify

1.2.2 Віртуальні потоки в Java

Одним із найбільших досягнень у розвитку багатопоточності в Java стало введення віртуальних потоків в Java 19. Цей новий механізм дозволяє створювати значно більше потоків без витрат на їх управління. Віртуальні потоки забезпечують значну економію ресурсів порівняно з традиційними потоками Java. Це особливо корисно для високонавантажених серверів, які обробляють тисячі одночасних запитів.

Віртуальні потоки управляються за допомогою віртуальної машини Java, що дозволяє зменшити накладні витрати та підвищити ефективність обробки паралельних задач. Завдяки цим потокам можна значно збільшити кількість одночасно працюючих потоків без шкоди для продуктивності, що робить їх ідеальним рішенням для розподілених та високонавантажених систем. Це дозволяє ефективно масштабувати програму, не турбуючись про витоки пам'яті або великі накладні витрати на управління потоками [3].

1.2.3 ExecutorService та ForkJoinPool

Для зручнішого та ефективнішого керування потоками в Java існують ExecutorService та ForkJoinPool. Перший надає набір методів для керування пулом потоків і обробки задач асинхронно. ExecutorService дозволяє оптимізувати

використання системних ресурсів, автоматично створюючи та знищуючи потоки в залежності від навантаження.

ForkJoinPool особливо корисний для задач, що можуть бути паралелізовані, таких як обробка великих масивів або деревоподібних структур. Цей пул потоків дозволяє ефективно розподіляти завдання між потоками, зменшуючи загальний час виконання складних задач, що потребують багатоядерної обробки. Використання цих інструментів значно спрощує процес написання багатопотокових програм, оскільки зменшує потребу в ручному керуванні потоками та синхронізації [4].

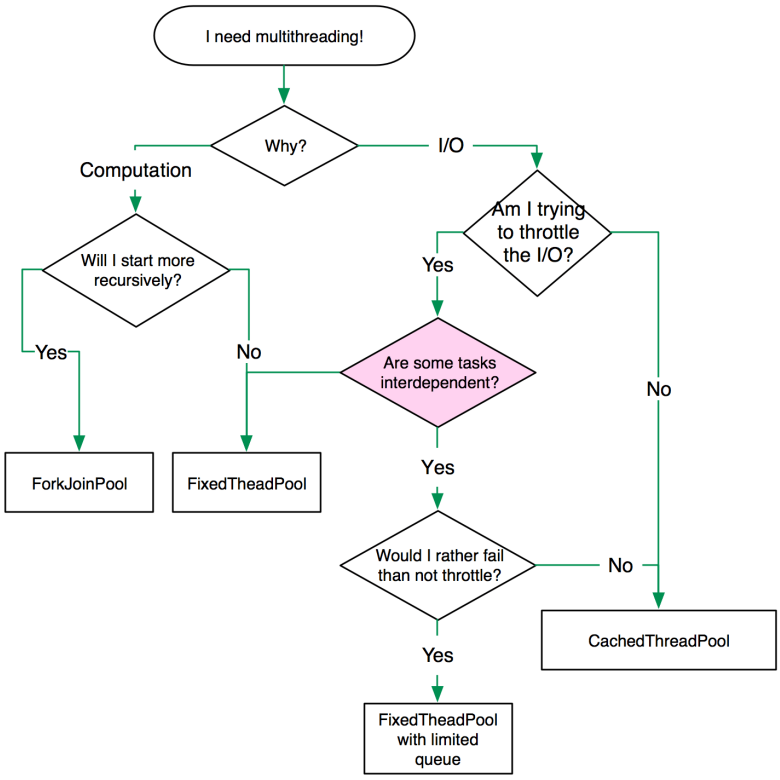


Рис 1.3 Блок-схема вибору класу для багатопоточності

1.2.4 Синхронізація та безпека потоків

Однією з найбільших проблем при роботі з багатопоточними програмами є синхронізація доступу до спільних ресурсів. Коли кілька потоків намагаються одночасно змінювати одні й ті ж дані, це може призвести до різноманітних проблем, таких як конкурентний доступ до ресурсів (race conditions) та взаємне

блокування потоків (deadlock). Для вирішення цих проблем Java надає механізми синхронізації, які дозволяють координувати взаємодію потоків.

Основними інструментами синхронізації є:

- synchronized — оператор, що дозволяє блокувати доступ до певних частин коду, гарантуючи, що тільки один потік зможе виконувати їх в будь-який момент часу.
- ReentrantLock — більш гнучкий механізм синхронізації, який дозволяє вирішувати складніші задачі з контролю доступу до ресурсів.
- ReadWriteLock — дозволяє зменшити блокування, дозволяючи кільком потокам одночасно читати дані, але гарантуючи ексклюзивний доступ при записі.

Також важливо враховувати використання immutable об'єктів, що дозволяють уникнути необхідності синхронізації, оскільки такі об'єкти не змінюються після створення і тому безпечні для доступу з кількох потоків одночасно [5].

1.2.5 Переваги та виклики багатопоточності

Переваги багатопоточності в Java очевидні: це підвищення продуктивності, зниження часу відгуку, ефективне використання багатоядерних процесорів. Проте цей підхід має і свої виклики. До них відносяться складність в управлінні потоками, необхідність синхронізації для забезпечення безпеки та проблеми, пов'язані з витоками пам'яті, які можуть виникати через неправильне управління потоками.

Ретельне проектування багатопотокових систем і належна синхронізація — ключові аспекти для створення надійних та ефективних програм. Багатопоточність може значно підвищити продуктивність програми, однак вимагає високої точності та уважності при розробці.

1.3 Важливість багатопоточності для високонавантажених систем

У сучасних високонавантажених системах, які обробляють тисячі одночасних запитів або великі обсяги даних, багатопоточність відіграє вирішальну

роль. Вона дозволяє одночасно виконувати декілька задач, підвищуючи ефективність використання ресурсів і зменшуючи час відгуку системи. Завдяки багатопоточності високонавантажені системи можуть досягати значно вищої продуктивності, обробляючи велику кількість запитів або операцій паралельно, що є критичним для забезпечення швидкої реакції на потреби користувачів і безперебійної роботи в режимі реального часу.[6]

1.3.1 Основні переваги багатопоточності для високонавантажених систем

- Ефективне використання багатоядерних процесорів: Багатоядерні процесори стали стандартом для серверів і хмарних обчислювальних платформ. Багатопоточність дозволяє програмам повністю використовувати доступні ядра, виконуючи завдання паралельно, що значно підвищує загальну продуктивність системи.
- Швидкий відгук і зменшення затримок: Багатопоточність дозволяє обробляти різні запити одночасно, що зменшує затримки в обробці запитів. Це особливо важливо для високонавантажених систем, де навіть незначні затримки можуть негативно впливати на користувацький досвід.
- Масштабованість і адаптивність: Багатопоточність полегшує масштабування системи відповідно до навантаження. Додавання нових потоків для обробки збільшеного обсягу запитів дозволяє системі адаптуватися до змін у навантаженні без втрат у продуктивності.

1.3.2 Виклики реалізації багатопоточності у високонавантажених системах

- Синхронізація доступу до ресурсів: У багатопотокових системах кілька потоків можуть одночасно звертатися до спільних ресурсів (даних або об’єктів). Це може призвести до некоректної роботи програми, коли потоки "перебивають" одне одного, змінюючи дані у непередбачуваному порядку. Щоб уникнути цього, необхідно використовувати механізми синхронізації, як-от synchronized, ReentrantLock, або атомарні операції. Однак, надмірне

блокування ресурсів може знизити продуктивність, призводячи до проблем, таких як "вузькі місця" (bottlenecks).

- Взаємне блокування (deadlock): Високонавантажені системи можуть бути схильними до взаємного блокування, коли два або більше потоків чекають один на одного для завершення операцій з ресурсами. Це призводить до зупинки роботи системи. Запобігання deadlock вимагає ретельного проектування, зокрема впорядкування доступу до ресурсів і використання тайм-аутів.
- Перевантаження системи потоками: Занадто велика кількість потоків може перевантажити процесор і пам'ять, особливо якщо кожен потік створюється і завершується часто. Тому високонавантажені системи потребують оптимізованих підходів до управління потоками, таких як використання пулу потоків (ExecutorService), де потоки повторно використовуються для обробки різних задач.
- Ускладнене тестування та налагодження: Багатопоточність ускладнює тестування, оскільки важко передбачити, як потоки взаємодіятимуть в умовах високого навантаження. Тести мають враховувати різні сценарії взаємодії між потоками, а налагодження помилок потребує специфічних інструментів та підходів для діагностики і відтворення станів програми.
- Витрати на підтримку пам'яті та управління ресурсами: Кожен потік використовує певну кількість пам’яті та ресурсів. У високонавантажених системах це може призвести до вичерпання ресурсів, особливо при великій кількості одночасних потоків. Залишкове навантаження на систему може знижувати продуктивність, а витоки пам'яті — спричиняти нестабільність.[7]

1.3.3 Типові проблеми у багатопоточності для високонавантажених систем

- Гонки даних (race conditions): Це ситуація, коли результат виконання програми залежить від неконтрольованого порядку доступу потоків до

ресурсів. Це часто призводить до помилок, коли декілька потоків одночасно змінюють один і той самий ресурс.

- Ізоляція транзакцій: У системах з високим рівнем транзакційної обробки одночасне виконання потоків може створювати проблеми з ізоляцією та консистентністю даних, особливо в розподілених середовищах.
- Конкурентні блокування (livelock): Це стан, коли потоки постійно змінюють свої стани у відповідь на дії один одного, але не просуваються в роботі. Хоча це не призводить до повного блокування, система залишається неефективною.

1.3.4 Висновок

Багатопоточність є невід'ємною частиною високонавантажених систем, але її використання супроводжується низкою викликів, пов'язаних з управлінням ресурсами та синхронізацією. Ефективна багатопотоковість вимагає ретельного підходу до проектування, тестування та налагодження для забезпечення стабільності, продуктивності та надійності. Управління потоками, правильне використання механізмів синхронізації та врахування можливих проблем із deadlock та race conditions є необхідними елементами для успішної реалізації багатопотокових рішень у високонавантажених системах.

1.4 Моделі багатопоточності в мікросервісах

У сучасних мікросервісних архітектурах ефективне управління багатопоточністю є критично важливим для досягнення високої продуктивності та масштабованості. Залежно від специфіки системи, різні моделі багатопоточності можуть пропонувати різні підходи до обробки запитів, що впливає на ефективність обробки задач. Кожна модель має свої переваги та недоліки, і вибір оптимальної моделі залежить від вимог до системи.

1.4.1 Модель з одним потоком на запит

Ця модель передбачає, що кожен вхідний запит обробляється окремим потоком. Вона є досить простою у реалізації та підходить для невеликих або середніх систем, де навантаження є стабільним і обмеженим. Однак, ця модель може стати неефективною для високонавантажених систем, де кількість одночасних запитів значно зростає. Тому її застосування обмежене умовами, коли кількість одночасних запитів не перевищує можливості обробки окремими потоками.

1.4.2 Модель з пулом потоків

Модель з пулом потоків є більш ефективною для середніх і великих систем. У цій моделі створюється обмежений набір потоків, що використовуються для обробки запитів. Коли потік завершить свою задачу, він повертається до пулу та готовий до використання для нового запиту. Це зменшує накладні витрати на створення нових потоків і дозволяє ефективно використовувати ресурси. Однак важливо налаштувати розмір пулу, оскільки надмірно великий пул може призвести до витрат на управління потоками, а надмірно малий — до простоїв через недостатню кількість доступних потоків.

1.4.3 Асинхронна обробка

Асинхронні моделі обробки запитів є особливо корисними для систем з високим навантаженням. Вони дозволяють зменшити накладні витрати на управління потоками та значно підвищити масштабованість. За допомогою неблокуючих операцій (наприклад, неблокуючі ввід/вивід або асинхронні API), система може обробляти тисячі запитів одночасно, без необхідності створення окремих потоків для кожного з них. Ця модель часто застосовується у високонавантажених мікросервісних системах, де потрібно забезпечити обробку великої кількості запитів з мінімальними витратами на ресурси [3].

1.4.4 Вибір моделі багатопоточності

Вибір відповідної моделі багатопоточності залежить від типу навантаження і характеристик системи. У статті "Microservice Threading Models and their Tradeoffs" зазначено, що для систем з помірним навантаженням можна використовувати пул потоків, що дозволяє досягти оптимального балансу між ефективністю і використанням ресурсів. Для систем з високим навантаженням, як показано в дослідженнях Engstrand (2023), рекомендується використовувати асинхронну обробку, що дозволяє мінімізувати блокування та знизити накладні витрати на управління потоками [8].

1.4.5 Переваги та недоліки моделей

- Модель з одним потоком на запит проста у реалізації, але не підходить для високонавантажених систем, оскільки створення великої кількості потоків може призвести до значних витрат на ресурси.
- Модель з пулом потоків забезпечує кращу ефективність і підходить для систем середнього рівня навантаження, проте потребує налаштування кількості потоків.
- Асинхронна обробка дозволяє досягти найкращої масштабованості та продуктивності для систем з високим навантаженням, але реалізація може бути складнішою і вимагати використання спеціальних фреймворків і бібліотек для неблокуючого вводу/виводу.

Загалом, вибір моделі багатопоточності залежить від конкретних вимог до системи і рівня навантаження. Важливо провести профілювання і тестування, щоб вибрати найбільш підходящий підхід для конкретного випадку.

Для порівняльної характеристики моделей потоків зобразимо таблицю:

Таблиця 1.1

Порівняння моделей потоків

Модель потоків	Проблеми ефективності	Проблеми складності коду
Одиночний потік, одиничний процес	Служба не буде повністю використовувати ядра сервера. Очікуйте, що пропускна здатність не збільшиться при додатковому навантаженні, а використання процесора не перевищить 10%.	Найпростіший і найзрозуміліший підхід.
Одиночний потік, багатопроцесний, новий процес для кожного запиту	Перевантаження на створення процесів і постійне створення та знищення з'єднань з базою даних може підвищити затримку.	З'єднання з базою даних мають бути завантажені ліниво. Рекомендується використовувати кеш Opcode.
Одиночний потік, багатопроцесний, запити повторно використовують робочі процеси	Бази даних мають більше відкритих з'єднань, тому що їх неможливо ділити між межами процесів. Надмірна кількість відкритих з'єднань може погіршити продуктивність бази даних.	Потрібен додатковий код для управління життєвим циклом робочих процесів. Код має бути здатним відновлюватися після неактуальних з'єднань. Статичні змінні повинні скидатись з кожним запитом.

Продовження таблиці 1.1

Порівняння моделей потоків

Мультитредінг, одиночний довготривалий процес, виділений потік для кожного запиту	Перехресне пулінг з'єднань є дуже ефективним як для сервера, так і для бази даних, але для кожного запиту виділяється додатковий потік, що обмежує кількість одночасно оброблюваних запитів.	Оскільки потоки повинні ділити стан бази даних, розробники повинні вміти знаходити та виправляти проблеми з конкуренцією, такі як мертві блокування, голод, блокування потоків та умови гонки.
Мультитредінг, довготривалий одиночний процес, без виділеного потоку для кожного запиту	Перехресне пулінг з'єднань є дуже ефективним як для сервера, так і для бази даних. Очікується висока пропускна здатність для асинхронних операцій.	Операції з вводу/виводу повинні виконуватися в окремому пулі потоків. Якщо результати повинні бути повернуті виклику, обробник запиту повинен отримати їх із пулу потоків після завершення.

1.5 Паралельні та розподілені обчислення в мікросервісах

У сучасних мікросервісних архітектурах ефективне управління паралельними та розподіленими обчисленнями є критично важливим для досягнення високої продуктивності та масштабованості систем. Розподілені обчислення дозволяють розподіляти обробку даних та задач між кількома сервісами або навіть між кількома машинами, що забезпечує ефективне використання ресурсів та зменшує час відгуку системи. [9]

Паралельні обчислення передбачають одночасне виконання декількох задач, що можуть бути виконані незалежно одна від одної. У контексті мікросервісів це означає, що різні сервіси можуть обробляти різні частини запиту або даних

паралельно, що значно підвищує ефективність системи. Наприклад, при обробці складного запиту, що вимагає збору даних з кількох джерел, кожен мікросервіс може паралельно виконувати свою частину роботи, а потім об'єднувати результати для формування відповіді користувачу. [10]

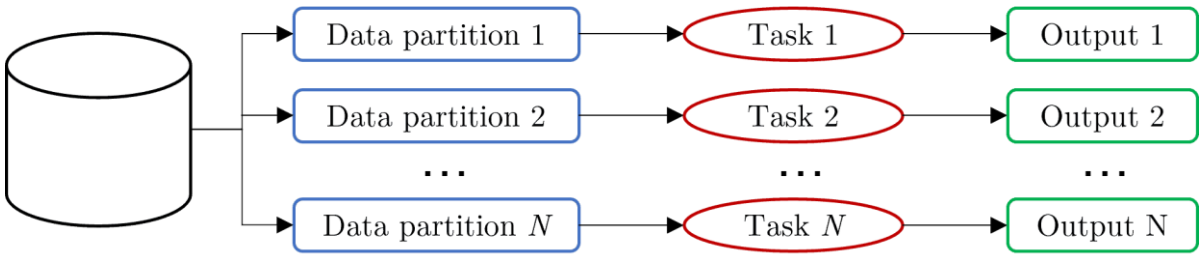


Рис 1.4 Візуалізація паралельного обчислення

Розподілені обчислення передбачають розподіл задач між кількома незалежними системами або сервісами, які можуть знаходитися на різних фізичних або віртуальних машинах. У мікросервісній архітектурі кожен сервіс є автономною одиницею, що може бути розгорнута на окремому сервері або контейнері. Це дозволяє масштабувати систему горизонтально, додаючи нові інстанси сервісів для обробки збільшеного навантаження. Наприклад, при високому навантаженні на певний сервіс можна додати додаткові його інстанси, що будуть обробляти запити паралельно, забезпечуючи балансування навантаження та високу доступність системи.

Для реалізації паралельних та розподілених обчислень у мікросервісах використовуються різноманітні технології та інструменти. Одним із таких інструментів є Apache Kafka, який забезпечує високу пропускну здатність та низьку затримку при передачі повідомлень між сервісами, що дозволяє ефективно реалізувати асинхронну обробку даних та подій. Іншим прикладом є Apache Spark, який надає потужні можливості для обробки великих обсягів даних у розподіленому середовищі, що може бути корисним для аналітичних задач у мікросервісних системах.

Використання паралельних та розподілених обчислень у мікросервісах має низку переваг, серед яких підвищення продуктивності, масштабованості та надійності системи. Однак, це також приносить певні виклики, такі як складність управління станом, забезпечення консистентності даних та обробка помилок у розподіленому середовищі. Наприклад, при розподілі задач між кількома сервісами необхідно враховувати питання транзакційності та забезпечення атомарності операцій, що може вимагати використання спеціальних патернів, таких як SAGA.

Ефективне використання паралельних та розподілених обчислень є ключовим фактором для досягнення високої продуктивності та масштабованості мікросервісних систем. Вибір відповідних технологій та інструментів, а також розуміння переваг та викликів цих підходів, є критично важливим для успішної реалізації мікросервісної архітектури.

Для порівняльної характеристики моделей потоків зобразимо таблицю:

Таблиця 1.2

Порівняння паралельного та розподіленого обчислення

Аспект	Паралельні обчислення	Розподілені обчислення
Архітектура	Зазвичай передбачає використання кількох процесорів або ядер в одній машині.	Передбачає використання кількох машин або вузлів, з'єднаних через мережу.
Комунікація	Комунікація між процесами зазвичай відбувається через спільну пам'ять або міжпроцесорні механізми комунікації в межах однієї машини.	Комунікація між вузлами відбувається через мережу, часто використовуючи передачу повідомлень або механізми віддалених процедурних викликів (RPC).

Продовження таблиці 1.2

Порівняння паралельного та розподіленого обчислення

Координація	Один контролюючий процес або потік зазвичай керує координацією між паралельними завданнями.	Координація між розподіленими вузлами вимагає більш складних механізмів через розподілену природу системи.
Масштабованість	Обмежена масштабованість через кінцеву кількість процесорів або ядер в одній машині.	Має вищий потенціал масштабованості завдяки додаванню нових вузлів до розподіленої системи.
Помехостійкість	Зазвичай не має вбудованих механізмів помехостійкості, окрім резервування в апаратному забезпеченні або програмному забезпеченні.	Часто включає механізми помехостійкості, такі як резервування, реплікація та виявлення помилок для обробки відмов вузлів або проблем з мережею.
Розгортання	Підходить для завдань, які можуть виграти від паралельного виконання в межах однієї машини, таких як багатоядерні процесори.	Підходить для завдань, які вимагають співпраці між кількома машинами або вузлами, таких як обробка великих даних або розподілені системи.
Приклади	Наукові симуляції, обробка зображень на багатоядерних ЦП, обчислення на ГПУ.	Хмарні обчислення, розподілені бази даних, пірингові мережі, обчислення в грідах.

1.6 Управління станом у розподілених системах

У розподілених системах управління станом є критично важливим аспектом, оскільки забезпечує консистентність, доступність та надійність даних, що зберігаються та обробляються різними компонентами системи. Управління станом охоплює стратегії зберігання, синхронізації та відновлення даних, що дозволяють системам ефективно функціонувати навіть у випадку відмови окремих компонентів.

Консистентність даних: консистентність означає, що всі копії даних у системі перебувають у узгодженому стані. У розподілених системах досягнення консистентності може бути складним через мережеві затримки та можливі відмови компонентів. Для забезпечення консистентності використовуються різні моделі, такі як CAP-теорема, яка стверджує, що система не може одночасно гарантувати консистентність, доступність та стійкість до розділення мережі. Вибір між цими властивостями залежить від конкретних вимог до системи.

Синхронізація стану в розподілених системах передбачає узгодження даних між різними компонентами. Для цього застосовуються механізми, такі як реплікація та шардінг. Реплікація передбачає створення копій даних на різних вузлах, що підвищує доступність та надійність. Шардінг розподіляє дані на різні сегменти (шарди), що дозволяє масштабувати систему горизонтально та зменшувати навантаження на окремі компоненти.[11]

Відновлення стану є процесом відновлення даних після відмови компонентів системи. Це включає використання журналів транзакцій, знімків стану та механізмів реплікації для забезпечення цілісності та доступності даних. Наприклад, у базах даних, таких як Cassandra, використовується підхід Eventual Consistency, який дозволяє системі відновлюватися до консистентного стану після відмови.

Основними викликами управління станом у розподілених системах є:

- Розділення мережі: Коли мережа розділяється, деякі компоненти можуть стати недоступними, що ускладнює синхронізацію стану.

- Конфлікти оновлень: Коли кілька компонентів одночасно оновлюють одні й ті ж дані, можуть виникати конфлікти, які потребують спеціальних механізмів для їх вирішення.
- Відновлення після відмови: Потрібно забезпечити механізми, які дозволяють системі відновлюватися до консистентного стану після відмови компонентів.

Управління станом у розподілених системах є складним, але критично важливим аспектом для забезпечення надійності та ефективності системи. Використання відповідних моделей консистентності, механізмів синхронізації та стратегій відновлення дозволяє створювати стійкі та масштабовані розподілені системи.[12]

1.7 Інструменти та технології для профілювання і моніторингу багатопотокових програм

Ефективне профілювання та моніторинг багатопотокових програм є критично важливими для забезпечення їхньої продуктивності, надійності та масштабованості. Використання спеціалізованих інструментів дозволяє розробникам виявляти вузькі місця, оптимізувати використання ресурсів та забезпечувати стабільну роботу системи.

1.7.1 Профілювання багатопотокових програм

Профілювання — це процес аналізу виконання програми з метою виявлення її "гарячих точок" та оптимізації продуктивності. Для багатопотокових програм особливо важливо враховувати взаємодію між потоками та синхронізацію доступу до спільних ресурсів.[13]

- Java Flight Recorder (JFR): Вбудований інструмент у Java, який дозволяє збирати детальну інформацію про виконання програми, включаючи дані про потоки, пам'ять та інші ресурси. JFR інтегрований з Java Virtual Machine (JVM) та забезпечує мінімальний вплив на продуктивність під час збору даних.

- VisualVM: Графічний інструмент для моніторингу та профілювання Java-додатків. Він надає інформацію про використання CPU, пам'яті, активні потоки та інші метрики, що допомагає виявляти проблеми з продуктивністю та управлінням потоками.
- YourKit Java Profiler: Комерційний інструмент, який пропонує глибокий аналіз продуктивності, включаючи профілювання багатопотокових додатків, виявлення витоків пам'яті та аналіз використання CPU.

1.7.2 Моніторинг багатопотокових програм

Моніторинг передбачає постійне спостереження за станом програми в реальному часі, що дозволяє оперативно виявляти та реагувати на проблеми. [14]

- Prometheus: Система моніторингу та збору метрик, яка підтримує збір даних про виконання програм, включаючи інформацію про потоки, використання ресурсів та інші показники. Prometheus інтегрується з різними системами та надає потужні можливості для аналізу та візуалізації даних.
- Grafana: Платформа для візуалізації метрик, яка часто використовується разом з Prometheus для створення дашбордів, що відображають стан багатопотокових програм у реальному часі.
- Datadog: Хмарна платформа для моніторингу, яка забезпечує збір та аналіз метрик, логів та трасувань, що дозволяє відстежувати продуктивність багатопотокових додатків та інфраструктури.

Профілювання та моніторинг багатопотокових програм мають свої особливості та виклики:

- Синхронізація даних: Потрібно враховувати взаємодію між потоками та забезпечувати коректне відображення даних, що можуть бути змінені одночасно кількома потоками.
- Вплив на продуктивність: Інструменти профілювання та моніторингу можуть впливати на продуктивність програми, тому важливо мінімізувати цей вплив під час збору даних.

- Об'єм даних: Багатопотокові програми генерують великий об'єм даних, що потребує ефективних механізмів збору, зберігання та аналізу інформації.

Використання спеціалізованих інструментів для профілювання та моніторингу багатопотокових програм є необхідним для забезпечення їхньої ефективності та надійності. Розуміння особливостей цих інструментів та їх правильне застосування дозволяє розробникам оперативно виявляти та усувати проблеми, забезпечуючи стабільну роботу системи.

1.8 Безпека в багатопоточних і розподілених системах

У сучасних інформаційних системах, що базуються на багатопоточності та розподілених архітектурах, забезпечення безпеки є критично важливим аспектом. Багатопоточність дозволяє одночасно виконувати кілька задач, що підвищує ефективність, але також створює нові виклики щодо захисту даних та ресурсів. Розподілені системи, у свою чергу, передбачають взаємодію численних компонентів, що можуть бути географічно віддаленими, що ускладнює контроль доступу та моніторинг безпеки.[11]

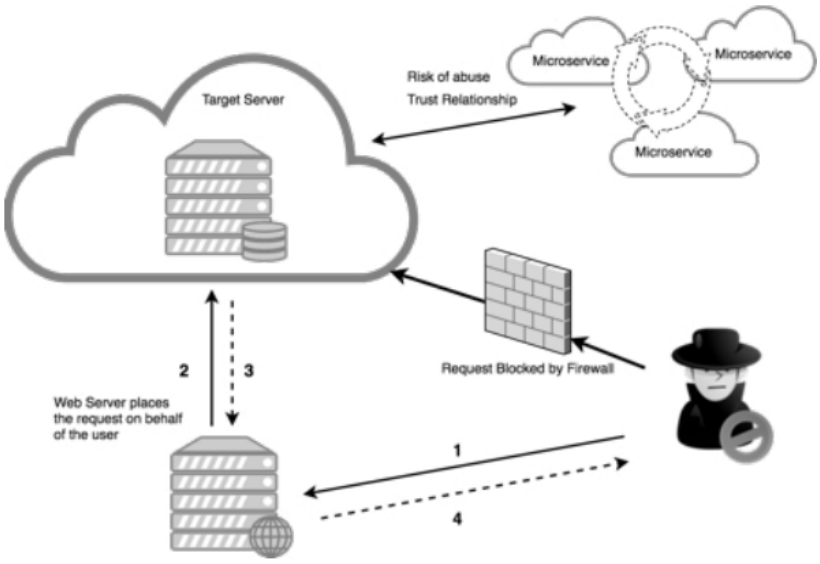


Рис 1.5 Типові зловживання SSRF

У багатопоточних програмах основними проблемами безпеки є:[15]

- Конкурентний доступ до ресурсів: Коли кілька потоків одночасно намагаються отримати доступ до спільних ресурсів, можуть виникати умови гонки (race conditions), що призводять до непередбачуваних результатів. Для запобігання цьому використовуються механізми синхронізації, такі як блокування (locks) та семафори.
- Взаємне блокування (deadlock): Ситуація, коли два або більше потоків чекають один на одного, утримуючи ресурси, що призводить до зупинки програми. Для запобігання deadlock необхідно ретельно проектувати порядок отримання ресурсів та використовувати тайм-аути.
- Витоки пам'яті: Невчасне звільнення ресурсів може призвести до витоків пам'яті, що негативно впливає на продуктивність та стабільність системи. Використання інструментів профілювання та моніторингу допомагає виявляти та усувати такі проблеми.

Розподілені системи стикаються з додатковими викликами безпеки:

- Аутентифікація та авторизація: Необхідно забезпечити, щоб лише авторизовані користувачі та сервіси мали доступ до ресурсів системи. Використання протоколів, таких як OAuth 2.0 та OpenID Connect, дозволяє ефективно керувати доступом.
- Конфіденційність та цілісність даних: Дані, що передаються між компонентами системи, повинні бути захищені від несанкціонованого доступу та змін. Для цього застосовуються методи шифрування, такі як TLS/SSL для захищеного каналу зв'язку.
- Моніторинг та аудит: Постійний моніторинг активності в системі дозволяє вчасно виявляти підозрілі дії та потенційні загрози. Використання систем моніторингу, таких як Prometheus та Grafana, допомагає відстежувати метрики безпеки та оперативно реагувати на інциденти.

Для підвищення безпеки багатопоточних та розподілених систем рекомендується:[15]

- Ретельне проектування: Забезпечення правильного проектування системи з урахуванням потенційних загроз та вразливостей.

- Використання сучасних протоколів безпеки: Застосування актуальних стандартів та протоколів для захисту даних та доступу.
- Регулярне тестування та аудит: Проведення регулярних тестів безпеки та аудитів коду для виявлення та усунення вразливостей.
- Навчання персоналу: Постійне підвищення кваліфікації розробників та адміністраторів щодо сучасних загроз та методів захисту.

Забезпечення безпеки в багатопоточних та розподілених системах є складним, але необхідним процесом. Використання сучасних інструментів, дотримання кращих практик та постійне навчання персоналу дозволяють створювати надійні та безпечні системи, здатні ефективно функціонувати в умовах сучасних кіберзагроз.

1.9 Порівняння багатопоточності та мікросервісів з іншими підходами до паралельного програмування

Паралельне програмування — це основа для забезпечення ефективної роботи сучасних програм, особливо коли вони повинні обробляти великі обсяги даних або виконувати складні операції в обмежений час. Існує кілька підходів до паралельного програмування, серед яких багатопоточність, мікросервіси, паралельне програмування на рівні процесів, та асинхронне програмування. Кожен з них має свої переваги та недоліки, що залежать від конкретних вимог системи.

1.9.1 Багатопоточність

Багатопоточність — це техніка, за якою кілька потоків виконуються одночасно в межах одного процесу. Це дозволяє програмам ефективно використовувати багатоядерні процесори, що значно підвищує продуктивність.[16]

Переваги:

- Ефективне використання ресурсів: Паралельне виконання декількох задач в межах одного процесу дозволяє зменшити накладні витрати на запуск нових процесів.

- Простота реалізації: Для простих задач багатопоточність є достатньо зручною та легкою у використанні.

Недоліки:

- Складність синхронізації: Для уникнення таких проблем, як гонки даних, необхідно ретельно синхронізувати доступ до спільних ресурсів, що додає складності.
- Ризик взаємного блокування (deadlock): Коли потоки чекають один на одного, виникає deadlock, який може призвести до зависання програми.

1.9.2 Мікросервіси

Мікросервісна архітектура передбачає розподіл програми на багато незалежних компонентів (сервісів), кожен з яких виконує конкретну задачу. Це дозволяє системам бути більш гнучкими, легко масштабованими і незалежними.[17]

Переваги:

- Масштабованість: Кожен мікросервіс може бути масштабований окремо в залежності від навантаження.
- Незалежність компонентів: Кожен мікросервіс може бути оновлений чи замінений без впливу на інші частини системи.

Недоліки:

- Складність управління: Мікросервіси додають значну складність в управлінні, оскільки кожен сервіс має свій стан, і необхідно організовувати їх взаємодію.
- Затримки через комунікацію: Розподілена природа мікросервісів може спричиняти затримки в комунікації між сервісами через мережу.

1.9.3 Паралельне програмування на рівні процесів

У цьому підході паралелізм досягається за допомогою кількох окремих процесів, які виконуються паралельно і можуть взаємодіяти між собою. Це дозволяє досягти високої ізоляції між компонентами системи.[18]

Переваги:

- Ізоляція процесів: Оскільки кожен процес працює незалежно, помилки в одному процесі не впливають на інші.
- Незалежність від апаратних ресурсів: Окремі процеси можуть бути запущені на різних машинах чи віртуальних середовищах, що дозволяє розподіляти навантаження.

Недоліки:

- Комунікація між процесами: Між процесами необхідна комунікація через більш повільні механізми, такі як сокети чи повідомлення, що може значно знизити продуктивність.
- Витрати на керування процесами: Створення і управління кількома процесами на різних машинах може бути складним і затратним.

1.9.4 Асинхронне програмування

Асинхронне програмування передбачає, що програма не чекає завершення задачі перед тим, як перейти до наступної. Це особливо ефективно при роботі з операціями вводу/виводу, де час очікування можна використовувати для виконання інших задач.

Переваги:

- Не блокуюче виконання: Програма може виконувати інші задачі під час очікування завершення операцій, що підвищує її продуктивність.
- Мінімальні витрати на потоки: Оскільки асинхронне програмування не потребує створення нових потоків, це знижує накладні витрати.

Недоліки:

- Складність управління потоками: Асинхронне програмування може бути складним для розуміння та підтримки, особливо коли потрібно керувати великою кількістю паралельних операцій.
- Виклики в обробці помилок: Оскільки задачі виконуються паралельно, виникають труднощі з управлінням помилками і станом програми.

1.9.5 Порівняння підходів

- Продуктивність: Багатопоточність і паралельне програмування на рівні процесів можуть бути більш ефективними для важких обчислювальних задач. Мікросервіси та асинхронне програмування можуть бути більш ефективними для задач, що вимагають обробки великої кількості одночасних запитів, наприклад, у веб-додатках.
- Масштабованість: Мікросервіси забезпечують найбільшу масштабованість, оскільки дозволяють масштабувати окремі компоненти системи незалежно. Паралельне програмування та багатопоточність обмежені апаратними ресурсами.
- Складність: Паралельне програмування та багатопоточність вимагають ретельного управління потоками та синхронізацією. Мікросервіси додають складність в управлінні та комунікації між компонентами, а асинхронне програмування може бути складним для розробників через необхідність управління асинхронними задачами.

1.9.6 Висновок

Вибір між багатопоточністю, мікросервісами, паралельним програмуванням та асинхронним програмуванням залежить від специфіки задачі та вимог до продуктивності, масштабованості та складності системи. У разі високих вимог до паралелізму обчислень найкращим варіантом може бути багатопоточність або паралельне програмування. Для великих, розподілених систем з вимогами до масштабованості, мікросервіси є оптимальним підходом. Асинхронне програмування добре підходить для задач, пов'язаних з обробкою запитів вводу/виводу.

2 АНАЛІЗ ТЕХНОЛОГІЙ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ОБРОБКИ ДАННИХ JAVA

2.1 ExecutorService та керування потоками

У багатопотокових Java-програмах ефективне управління потоками є ключовим для забезпечення продуктивності, особливо при обробці великих масивів даних. **ExecutorService** є потужним інструментом, який дозволяє легко організовувати пул потоків, повторно використовувати їх і зменшувати витрати на створення нових потоків.

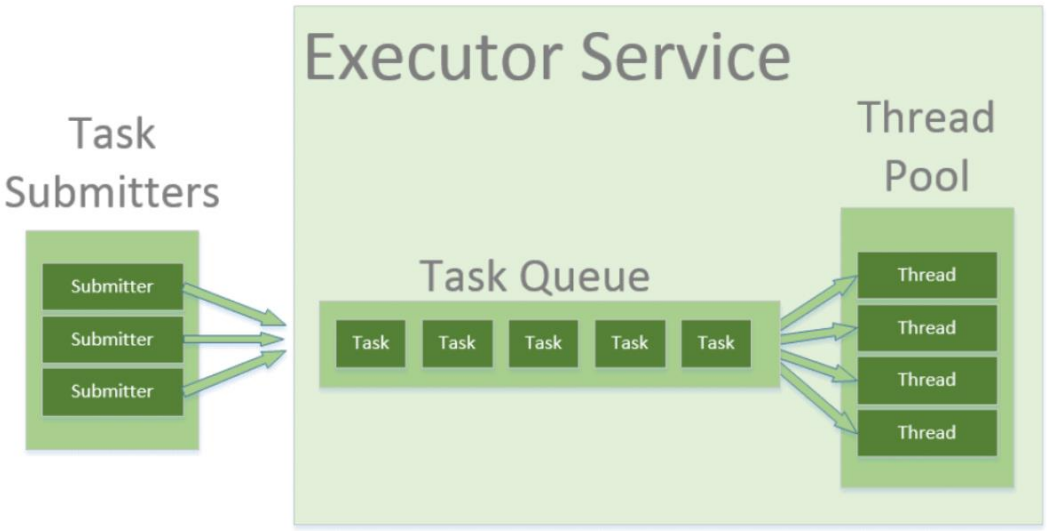


Рис 2.1 Візуалізація роботи Executor Service

2.1.1 Основи ExecutorService

ExecutorService входить до складу бібліотеки java.util.concurrent і надає розширений API для управління потоками:

- Асинхронне виконання задач: Ви можете запускати задачі в асинхронному режимі, отримуючи результат за допомогою об'єкта Future.

- Керування пулом потоків: Замість створення нового потоку для кожної задачі, ExecutorService дозволяє використовувати пул потоків для повторного виконання задач.
- Життєвий цикл потоків: Легке управління створенням, виконанням і завершенням потоків.

Основні методи:

- submit(): Додає задачу для виконання з можливістю відстеження результату.
- execute(): Виконує задачу без повернення результату.
- shutdown(): Завершує пул після виконання всіх задач.
- invokeAll() та invokeAny(): Виконують кілька задач одночасно.

Далі в пункті 2.1.2 та 2.1.3 буде наведено приклад коду, котрий буде використовуватись для подальших перевірок

2.1.2 Однопотокова перевірка

Цей код не використовує методи або технології за-для прискорення обробки даних, він нам потрібен для порівняння ефективності методів та технологій.

На рисунку 2.2 представлено код для перевірки однопотокового режиму, для цього я заповнюю ArrayList на 100 мільйонів випадкових чисел в діапазоні [1;10). Далі для перевірки часу виконання циклу, в якому буде вираховуватись сума чисел цього масиву, я використав System.currentTimeMillis(), в змінну startTime записується поточний час перед початком циклу, а в endTime, час одразу після виконання суми чисел. Далі в консоль виводиться сума чисел масиву ArrayList, та різниця endTime та startTime, котра вказує скільки часу програмі знадобилось на виконання ста мільйонів операцій.

SingleThreadTest.java

```java
import java.util.ArrayList;
import java.util.Random;

public class SingleThreadTest {
    public static void main(String[] args) {
        // Ініціалізація ArrayList з 100,000,000 чисел
        ArrayList<Double> numbers = new ArrayList<>( initialCapacity: 100_000_000);
        Random random = new Random();

        // Заповнення ArrayList випадковими числами від 1 до 10
        for (int i = 0; i < 100_000_000; i++) {
            numbers.add(1 + random.nextDouble() * 9); // Числа у діапазоні [1, 10)
        }

        long startTime = System.currentTimeMillis(); // Початок вимірювання часу

        double totalSum = 0.0;
        for (double number : numbers) {
            totalSum += number;
        }

        long endTime = System.currentTimeMillis(); // Кінець вимірювання часу
        System.out.println("Total Sum: " + totalSum);
        System.out.println("Time Taken (Single Thread): " + (endTime - startTime) + " ms");

    }
}
```

Рис 2.2 Код для порівняння, однопотоковий режим

Для більш точного результату зроблю 10 перевірок та занесу результати в таблицю

Таблиця 2.1

Результати десяти перевірок однопотокового режиму

№	Сума	Час, мс
1	5.499868330673376E8	289
2	5.500186600229688E8	290
3	5.5003024212039E8	293
4	5.499904192258786E8	300
5	5.500442483498882E8	298
6	5.50040550657824E8	291
7	5.500085900937008E8	300
8	5.499746245045645E8	289
9	5.500074582021265E8	284
10	5.499957698163519E8	298

Середній час обробки зайняв 293,2 ms. Опираючись на результати перевірки, можна побачити що час дуже схожий, та майже не відрізняється один від одного.

2.1.3 Багатопотокова перевірка з використанням ExecutorService

Далі додам в цей код бібліотеку java.util.concurrent.*, для використання ExecutorService. Також треба додати до методу static void main(String[] args) винятки, за допомогою ключового слова throws. InterruptedException – потрібен для потоків котрі залежать один від одного, так як в коді буде виконуватись сума, вона буде розбита на 4 частини(потоку), в кінці треба буде зробити суму результатів потоку, тому треба буде зупиняти потоки котрі вже завершили дію, поки виконуються інші, для цього і потрібно це виключення. ExecutionException - виключення цього типу, це як перевірена обгортка навколо довільного виключення, що генерується під час виконання завдання. Наприклад, метод get( ) об'єкта Future генерує виключення ExecutionException, якщо метод call( ) об'єкта Callable генерує виключення.

SingleThreadTest.java ExecutorServiceTest.java

```java
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.*;

public class ExecutorServiceTest {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        // Ініціалізація ArrayList з 100,000,000 чисел
        ArrayList<Double> numbers = new ArrayList<>( initialCapacity: 100_000_000);
        Random random = new Random();

        // Заповнення ArrayList випадковими числами від 1 до 10
        for (int i = 0; i < 100_000_000; i++) {
            numbers.add(1 + random.nextDouble() * 9); // Числа у діапазоні [1, 10)
        }

        ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4); // Пул із 4 потоків
        int chunkSize = numbers.size() / 4; // Розділення ArrayList на 4 частини
        Future<Double>[] futures = new Future[4];

        long startTime = System.currentTimeMillis(); // Початок вимірювання часу

        for (int i = 0; i < 4; i++) {
            final int start = i * chunkSize;
            final int end = (i == 3) ? numbers.size() : start + chunkSize;
            futures[i] = executor.submit(() -> {
                double sum = 0.0;
                for (int j = start; j < end; j++) {
                    sum += numbers.get(j);
                }
                return sum;
            });
        }

        double totalSum = 0.0;
        for (Future<Double> future : futures) {
            totalSum += future.get(); // Збирання результатів із кожного потоку
        }

        executor.shutdown(); // Завершення пулу потоків
```

Рис. 2.3 Код з використанням ExecutorService

Створюємо об’єкт класу ExecutorService та передаємо кількість потоків(4), за допомогою класу Future ми будемо розбивати суму на 4 різних частини та перевіряти чи закінчився підрахунок, перед тим як зібрати суму з усіх потоків в одну. Результати перевірок записую у таблицю:

Таблиця 2.2

Результати десяти перевірок ExecutorService

№	Сума	Час, мс
1	5.500247395656209E8	119
2	5.499661739393191E8	121
3	5.499735792401091E8	110
4	5.499950586816938E8	115
5	5.499690244922572E8	129
6	5.499965147339792E8	121
7	5.500044090182439E8	117
8	5.500400099605448E8	116
9	5.500046996627849E8	111
10	5.499871782375939E8	111

Середній час обробки становить 117 ms, порівнюючи з однопотоковою програмою, котра виконую ту ж саму функцію, розділивши підрахунок на чотири потоки вдалося прискорити дію на 2,506 рази. Звісно якщо брати «сухі» цифри, то програма була повинна прискоритись в 4 рази, але не треба забувати що додались додаткові розрахунки перед кожним потоком, тому четвертий потік трішки сповільнює інші.

2.1.4 Короткий аналіз використання ExecutorService

Обмеження ExecutorService:

1. Проблеми синхронізації: Для задач, які потребують спільного доступу до ресурсів, необхідно забезпечувати синхронізацію.

2. Параметри пулу: Неправильне налаштування розміру пулу потоків може призвести до перевантаження або неефективного використання ресурсів.
3. Витрати на управління: Використання ExecutorService накладає додаткові витрати на управління потоками.

ExecutorService є потужним інструментом для підвищення ефективності обробки великих обсягів даних у Java. Він забезпечує повторне використання потоків, гнучкість у виконанні задач та контроль життєвого циклу потоків, що робить його незамінним для створення високопродуктивних систем. Його використання спрощує управління потоками, дозволяючи зосередитися на логіці програми, а не на технічних деталях створення та синхронізації потоків.

2.2 Fork/Join Framework

Fork/Join Framework — це спеціалізований механізм для організації паралельних обчислень у Java. Він реалізує парадигму "розділяй та володарюй" (divide and conquer), розбиваючи великі задачі на менші підзадачі, які виконуються паралельно, а потім об'єднуються для отримання фінального результату. Цей фреймворк є частиною java.util.concurrent і був доданий у Java 7.

2.2.1 Основні компоненти Fork/Join Framework

ForkJoinTask:

- Абстрактний клас, що представляє задачу, яку можна розділити на підзадачі.
- Два основні підкласи:
  - RecursiveTask<V> — для задач із поверненням результату.
  - RecursiveAction — для задач без повернення результату.

ForkJoinPool:

- Спеціалізований пул потоків, оптимізований для виконання задач, створених ForkJoinTask.

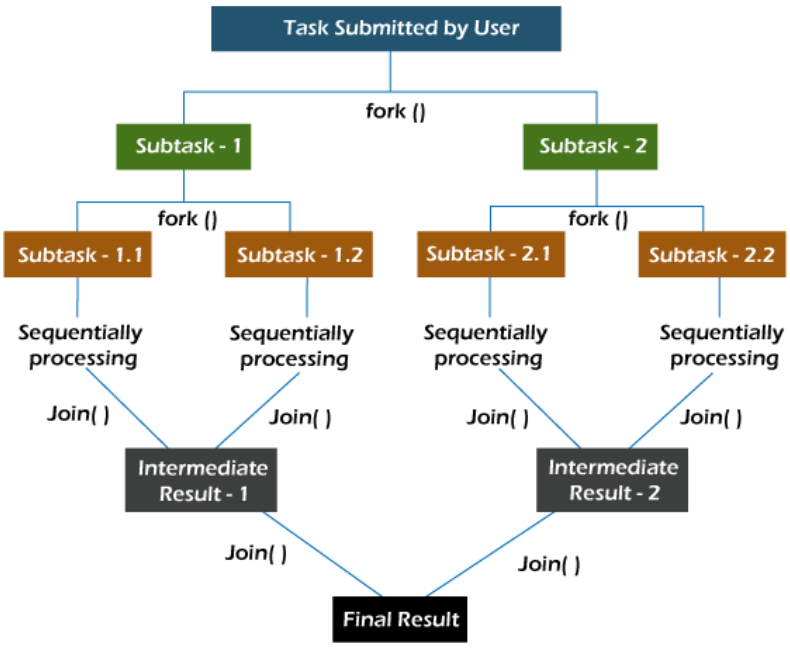


Рис 2.4 Ілюстрація роботи Fork/Join Framework

2.2.2 Переваги Fork/Join Framework

Ефективне управління потоками:

- Використовує пул потоків із алгоритмом "робочих крадіжок" (work-stealing), коли потоки, які завершили свої задачі, забирають підзадачі в інших потоків.

Розподіл обчислень:

- Великі задачі розбиваються на підзадачі, які можуть бути виконані паралельно.

Масштабованість:

- Підходить для програм із високою кількістю підзадач, які можуть виконуватись незалежно.

2.2.3 Багатопотокова перевірка з використанням Fork/Join Framework

Тепер модифікую ту ж саму програму на розрахунок суми 100 мільйонів випадкових чисел, для точності порівняння. Для цього знадобиться знову бібліотека java.util.concurrent.*; тільки в цей раз нам знадобиться звідти тільки два класи RecursiveTask та ForkJoinPool.

```
static class SumTask extends RecursiveTask<Double> {  6 usages
    private final ArrayList<Double> numbers;  4 usages
    private final int start;  5 usages
    private final int end;  5 usages

    public SumTask(ArrayList<Double> numbers, int start, int end) {  3 usages
        this.numbers = numbers;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Double compute() {
        if (end - start <= THRESHOLD) {
            // Якщо розмір задачі малий, обробляємо послідовно
            double sum = 0.0;
            for (int i = start; i < end; i++) {
                sum += numbers.get(i);
            }
            return sum;
        } else {
            // Розділяємо задачу на підзадачі
            int mid = (start + end) / 2;
            SumTask leftTask = new SumTask(numbers, start, mid);
            SumTask rightTask = new SumTask(numbers, mid, end);

            leftTask.fork(); // Виконуємо першу задачу асинхронно
            double rightResult = rightTask.compute(); // Виконуємо другу задачу
            double leftResult = leftTask.join(); // Чекаємо завершення першої задачі

            return leftResult + rightResult;
        }
    }
}
```

Рис 2.5 Клас SumTask

```
    public static void main(String[] args) {
        // Ініціалізація ArrayList із 100,000,000 чисел
        ArrayList<Double> numbers = new ArrayList<>( initialCapacity: 100_000_000);
        Random random = new Random();

        for (int i = 0; i < 100_000_000; i++) {
            numbers.add(1 + random.nextDouble() * 9);
        }

        ForkJoinPool pool = new ForkJoinPool(); // Створюємо пул потоків ForkJoin

        long startTime = System.currentTimeMillis();
        SumTask task = new SumTask(numbers,  start: 0, numbers.size());
        double totalSum = pool.invoke(task); // Запускаємо задачу

        long endTime = System.currentTimeMillis();
        System.out.println("Total Sum: " + totalSum);
        System.out.println("Time Taken (Fork/Join): " + (endTime - startTime) + " ms");
    }
}
```

Рис 2.6 Програма з використанням Fork/Join Framework

На рисунку 2.5 вказано клас SumTask, котрий успадковує абстрактний клас RecursiveTask, та перевизначає його метод compute(). Також на самому початку класу ForkJoinSum зазначається на яку кількість задач буде розбиватися програма.

На відміну від ExecutorService, у ForkJoin непотрібно задавати кількість потоків, цей фреймворк сам визначає скільки та які потоки використати.

Результати перевірок записую у таблицю:

Таблиця 2.3

Результати десяти перевірок ForkJoin

№	Сума	Час, мс
1	5.50022232498963E8	114
2	5.5002336861071E8	107
3	5.499980948997443E8	108
4	5.499894666653895E8	100
5	5.500074955963726E8	107
6	5.50019924205251E8	114
7	5.4997707690851E8	98
8	5.500432452703497E8	105
9	5.499984549781559E8	103
10	5.50016611031251E8	106

Середній час виконання циклу становить 106,2 ms, що в 2,76 разів швидше за однопотокову програму та на 10 відсотків ефективніше за ExecutorService.

При роботі з Fork/Join Framework не вказується кількість потоків які будуть використовуватись(хоча за потреби можна обмежити). Фреймворк сам обирає які потоки, та коли вони будуть працювати, єдине що треба вказати, це на яку кількість задач буде розбита програма. Через це можуть виникати іноді затримки в виконанні. З цих висновків можна виділити ще декілька плюсів:

Розподіл задач:

- Fork/Join Framework автоматично розбиває задачі на підзадачі, оптимізуючи використання ресурсів.
- У ExecutorService потрібно вручну розділяти задачі на частини.

Динамічне балансування:

- Завдяки work-stealing потокам, які завершили свої задачі, можуть виконувати залишкові підзадачі, зменшуючи час очікування.

Також декілька мінусів/обмежень:

- Складність розробки. Потребує ретельного планування, особливо коли підзадачі мають залежності.
- Розмір задачі. Якщо розмір підзадач не оптимізований (THRESHOLD занадто великий або маленький), продуктивність може знижуватись.

2.3 Stream API та паралельна обробка

Stream API — це потужний інструмент у Java, доданий у версії 8, для роботи з колекціями даних. Він дозволяє обробляти дані у функціональному стилі, забезпечуючи читабельність і ефективність коду. Stream API підтримує як послідовну, так і паралельну обробку даних, що робить його особливо корисним для багатопоточних задач.[21]

2.3.1 Основи Stream API

Stream API дозволяє працювати з даними як із потоком, використовуючи такі етапи:

Створення потоку:

- Потік створюється з колекції або іншого джерела даних (масиву, файлу тощо).

Проміжні операції:

- Операції, такі як map(), filter(), та sorted(), створюють новий потік, трансформуючи дані.

Термінальні операції:

- Операції, такі як forEach(), reduce(), та collect(), обчислюють кінцевий результат.

2.3.2 Переваги паралельного Stream API

Автоматичне управління потоками:

- При використанні parallelStream(), Java автоматично розподіляє обробку даних між доступними ядрами процесора.

Читабельність коду:

- Паралельна обробка інтегрується безпосередньо в Stream API, що робить код простішим у порівнянні з ручним управлінням потоками.

Продуктивність:

- Завдяки розподілу даних між потоками, паралельна обробка значно скорочує час виконання задачі.

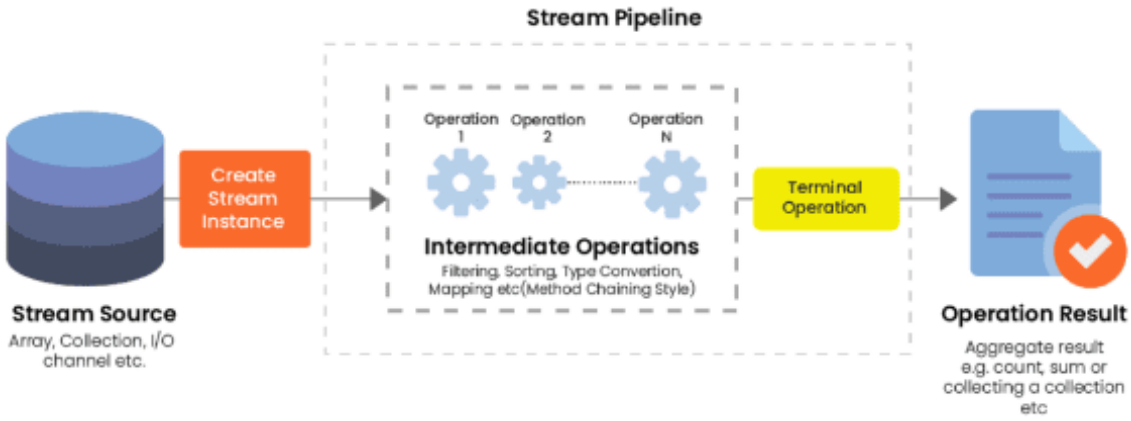


Рис 2.7 Зображення роботи потоків в Stream API

2.3.3 Багатопотокова перевірка з використанням Stream API

У порівнянні з іншими методами/технологіями при використанні Stream API код стає менше за обсягом. Наприклад в стандартній перевірці яку я роблю, а саме перевірка розрахунку суми 100 мільйонів випадкових чисел, ця дія, замість стандартних чотирьох строк, займає лише одну.

За таку простоту написання коду звісно що треба платити ефективністю. Після таблички з результатами приведу аргументи чому так і де краще використовувати цю бібліотеку.

SingleThreadTest.java ExecutorServiceTest.java ForkJoinSum.java ParallelStreamTest.java

```java
import java.util.ArrayList;
import java.util.Random;

public class ParallelStreamTest {
    public static void main(String[] args) {
        // Ініціалізація ArrayList з 100,000,000 чисел
        ArrayList<Double> numbers = new ArrayList<>( initialCapacity: 100_000_000);
        Random random = new Random();

        for (int i = 0; i < 100_000_000; i++) {
            numbers.add(1 + random.nextDouble() * 9); // Числа у діапазоні [1, 10)
        }

        long startTime = System.currentTimeMillis();

        double totalSum = numbers.parallelStream().reduce( identity: 0.0, Double::sum); // Паралельна обробка

        long endTime = System.currentTimeMillis();
        System.out.println("Total Sum: " + totalSum);
        System.out.println("Time Taken (Parallel Stream): " + (endTime - startTime) + " ms");
    }
}
```

Рис 2.8 Код з використанням Parallel Stream API

Таблиця 2.4

Результати десяти перевірок Parallel Stream API

№	Сума	Час, мс
1	5.500135179185774E8	394
2	5.500173689571774E8	371
3	5.499994386230533E8	368
4	5.500115904426843E8	364
5	5.499922330002346E8	378
6	5.50003324195848E8	390
7	5.500689202885091E8	357
8	5.499836076360576E8	382
9	5.499949957102544E8	393
10	5.500190532406356E8	383

Середній показник вийшов – 378 ms. Так, це повільніше навіть за звичайний код, але цьому є пояснення

2.3.4 Проблеми Parallel Stream API у порівнянні з іншими методами

Автоматизація vs. контроль:

- Parallel Stream API автоматично розподіляє дані між потоками за допомогою загального ForkJoinPool, що може створювати накладні витрати на управління.
- Fork/Join Framework і ExecutorService дозволяють розробникам вручну розділяти задачі та оптимізувати їх розподіл між потоками.

Накладні витрати на розбиття задач:

- Parallel Stream API генерує підзадачі на основі розбиття даних. Для великих обсягів це може створювати більше накладних витрат, ніж оптимізоване розбиття у Fork/Join Framework.

Використання загального пулу потоків:

- Parallel Stream API використовує загальний ForkJoinPool, який може бути перевантажений іншими задачами, що виконуються в програмі.

Ефективність залежить від обсягу обробки кожного елемента:

- Якщо кожен елемент вимагає мінімальної обробки (наприклад, проста сума), паралельна обробка може бути менш ефективною через накладні витрати на синхронізацію.

2.3.5 Коли використовувати Parallel Stream API

Прості задачі:

- Паралельна обробка може бути корисною для задач із великими наборами даних, які легко розділяються на незалежні підзадачі.

Низький рівень складності коду:

- Stream API значно зменшує кількість коду, необхідного для виконання паралельної обробки, у порівнянні з Fork/Join Framework або ExecutorService.

Обмежений контроль над потоками:

- Якщо програма не має складних вимог до управління потоками, Parallel Stream API може бути зручним.

2.3.6 Висновок щодо Parallel Stream API

Parallel Stream API може бути повільнішим, ніж Fork/Join Framework або ExecutorService, через автоматизацію процесу розподілу задач і відсутність можливості ручного контролю.

Він найкраще підходить для задач, де розподіл даних і операцій простий, а також для середовищ із високим рівнем багатозадачності, де ForkJoinPool не перевантажений іншими задачами.

Також треба зазначити що цей приклад не розкриває всіх можливостей цього фреймворку, він підходить для обробки даних, коли ми заздалегідь не знаємо кількість даних, або інші умови. В такому разі він буде швидше за інші, тому що там доведеться «вгадувати», а Stream API сам підбирає на скільки задач розбити програму та яку кількість потоків використати.

2.4 Java Virtual Threads (Project Loom)

Java Virtual Threads — це інноваційний підхід до багатопоточності, представлений у рамках Project Loom, який став доступним із Java 19. Віртуальні потоки (Virtual Threads) відрізняються від традиційних потоків тим, що вони легші, ефективніші та можуть створюватися у великих кількостях із мінімальними витратами пам’яті та ресурсів.[21]

2.4.1 Основи Virtual Threads

Що таке Virtual Threads:

- Це легковагові потоки, які працюють поверх класичних потоків JVM, але використовують значно менше ресурсів.
- Дозволяють одночасно виконувати тисячі (або навіть мільйони) потоків.

Як це працює:

- Віртуальні потоки відключені від системних потоків і управляються самим JVM.

- Коли потік блокується, JVM автоматично звільняє базовий системний потік, дозволяючи йому виконувати інші задачі.

Основні переваги:

- Значне скорочення витрат на створення потоків.
- Простота у використанні, оскільки API практично не змінюється.
- Ідеально підходить для високонавантажених систем, які обробляють тисячі одночасних запитів.

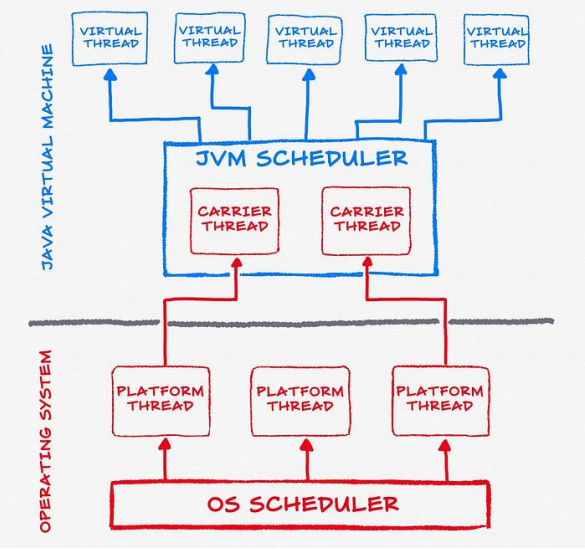


Рис 2.9 Візуалізація віртуальних потоків

2.4.2 Відмінності між Virtual Threads та ExecutorService

На перший погляд, використання Virtual Threads та ExecutorService виглядає схожим, адже обидва використовують API потоків (Runnable, Callable) і підтримують схожий підхід до управління задачами. Однак між ними є важливі відмінності[19]:

Таблиця 2.5

Відмінності між ExecutorService та Virtual Threads

Характеристика	ExecutorService	Virtual Threads
Тип потоків	Класичні потоки (OS Threads).	Віртуальні потоки (не прив'язані до OS Threads).
Кількість потоків	Обмежена ресурсами ОС.	Майже необмежена (мільйони потоків).
Витрати на створення	Високі, оскільки кожен потік прив'язаний до ядра ОС.	Низькі, оскільки потоки легковагові.
Блокування потоків	Блокування зупиняє системний потік, що призводить до простою.	Блокування лише зупиняє віртуальний потік, а системний потік звільняється.
Масштабованість	Обмежена через накладні витрати на створення та управління потоками.	Висока, підходить для обробки мільйонів запитів.
Залежність від версії JVM	Підтримується у всіх версіях Java.	Доступно лише починаючи з Java 19 (поки в експериментальному режимі).

2.4.3 Основні переваги Virtual Threads над класичним ExecutorService

- Масштабованість. У випадку високонавантажених систем (наприклад, серверів, які обробляють тисячі або мільйони одночасних з'єднань), класичний ExecutorService обмежений кількістю системних потоків. Virtual Threads дозволяють запускати десятки тисяч потоків без суттєвого збільшення використання пам’яті чи витрат на створення.
- Блокування. В класичних потоках блокування (наприклад, Thread.sleep() або доступ до I/O) призводить до простою системного потоку. У Virtual Threads блокування звільняє системний потік, дозволяючи йому виконувати інші задачі.

- Простота API. Virtual Threads інтегруються в стандартний API потоків Java, роблячи їх використання надзвичайно простим. Наприклад, метод Executors.newVirtualThreadPerTaskExecutor() дозволяє створити пул, який автоматично працює з віртуальними потоками.

2.4.4 Коли використовувати Virtual Threads

Ідеально підходять для:

- Обробки тисяч одночасних HTTP-запитів (наприклад, у веб-серверах).
- Систем із великою кількістю блокуючих операцій (наприклад, читання з баз даних або файлів).
- Сценаріїв, де потрібна масштабована багатопоточність із мінімальними витратами.

Не підходять для:

- Завдань із високим рівнем синхронізації між потоками.
- Сценаріїв, де використовується стара версія JVM (менш ніж Java 19).

ExecutorService все ще актуальний, оскільки:

- Він дає більше контролю над розподілом задач, ніж Virtual Threads у своєму поточному стані.
- У багатьох системах ExecutorService уже інтегрований і оптимізований для конкретних задач.
- Virtual Threads поки що знаходяться на експериментальній стадії, тому не всі проєкти готові їх прийняти.

Virtual Threads — це крок уперед у напрямку спрощення багатопоточності, але ExecutorService лишається важливим інструментом, особливо для задач, де контроль над потоками критично важливий.

2.4.5 Багатопотокова перевірка з використанням Virtual Threads (Project Loom)

Як вже зазначалось раніше, Virtual Threads дуже схожий за написанням коду на ExecutorService, навіть використовує об’єкт того ж класу, але основна

відмінність за якою можна одразу відрізнити їх, це те, що непотрібно зазначати кількість потоків заздалегідь.

```
15      ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor(); // Virtual Threads
16
17      int chunkSize = numbers.size() / 4; // Розділення масиву на 4 частини
18      Future<Double>[] futures = new Future[4];
19
20      long startTime = System.currentTimeMillis();
21
22      for (int i = 0; i < 4; i++) {
23          final int start = i * chunkSize;
24          final int end = (i == 3) ? numbers.size() : start + chunkSize;
25          futures[i] = executor.submit(() -> {
26              double sum = 0.0;
27              for (int j = start; j < end; j++) {
28                  sum += numbers.get(j);
29              }
30              return sum;
31          });
32      }
33
34      double totalSum = 0.0;
35      for (Future<Double> future : futures) {
36          totalSum += future.get();
37      }
```

Рис 2.10 Код з використанням Virtual Threads (Project Loom)

Таблиця 2.6

Результати десяти перевірок Virtual Threads (Project Loom)

№	Сума	Час, мс
1	5.49959800960512E8	110
2	5.499909300869877E8	111
3	5.500415297340193E8	108
4	5.500040325756155E8	112
5	5.499776946626239E8	115
6	5.49973099456774E8	117
7	5.500053324865122E8	119
8	5.500082605672246E8	108
9	5.500101143713695E8	113
10	5.500228216011755E8	118

Середній час виконання завдання – 113,1 ms. Що в 2,59 разів швидше за однопоточну програму, та всього на 3,5 відсотки ефективніше за ExecutorService.

Virtual Threads є революційною технологією для високонавантажених систем. Вони дозволяють досягти високої масштабованості без значних накладних витрат, але все ще перебувають на експериментальній стадії. ExecutorService залишається важливим інструментом, доповнюючи Virtual Threads для сценаріїв, які вимагають більше контролю.

2.5 Реактивне програмування (Reactive Programming)

Реактивне програмування — це стиль програмування, орієнтований на асинхронну, неблокуючу обробку даних, яка відбувається у вигляді потоку подій. Реактивні системи працюють ефективніше в багатопотоковому середовищі та забезпечують високу масштабованість навіть у системах із великим навантаженням.[22]

У Java найпоширеніші реалізації реактивного програмування базуються на бібліотеках, таких як Project Reactor і RxJava, які відповідають специфікації Reactive Streams.

2.5.1 Основи реактивного програмування

Асинхронність та неблокуючий характер:

- Обробка даних виконується асинхронно, і потоки не блокуються під час очікування результату.

Модель реактивних потоків:

- Вхідні дані перетворюються в потоки подій, які обробляються спостерігачами (об’єктами, що "підписуються" на ці потоки).

Основні принципи:

- **Backpressure** — контроль швидкості потоку даних між відправником і отримувачем.

- **Composable Pipelines** — побудова складних конвеєрів обробки даних із послідовністю операцій.

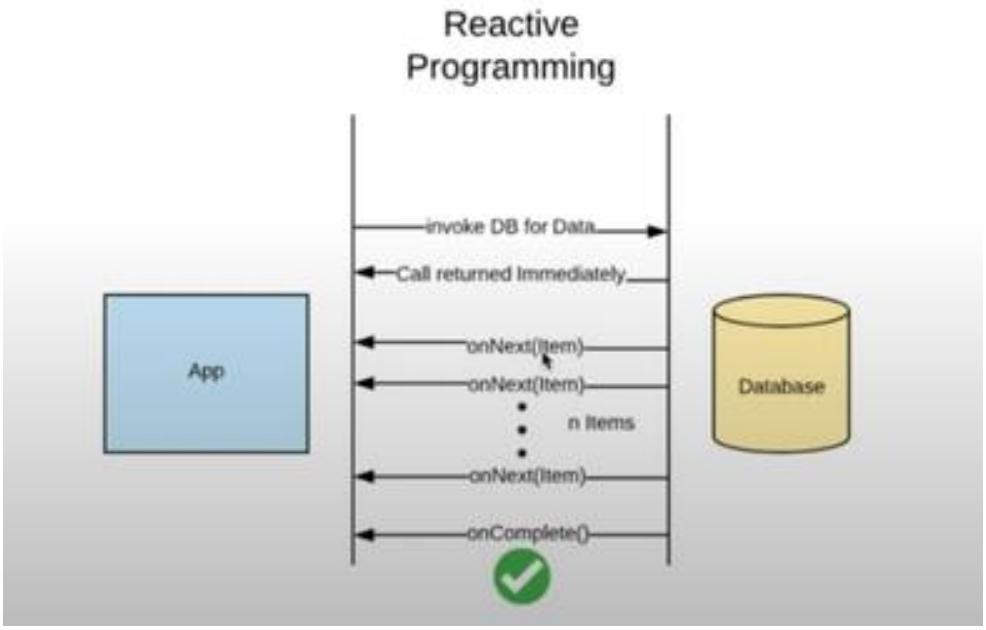


Рис 2.11 Візуалізація реактивного програмування

2.5.2 Переваги реактивного програмування

Висока масштабованість:

- Завдяки неблокуючій моделі система здатна обробляти тисячі або навіть мільйони одночасних запитів із мінімальними витратами ресурсів.

Гнучкість та модульність:

- Обробка даних виконується через конвеєри, що дозволяє легко змінювати логіку обробки.

Ефективність у високонавантажених системах:

- Реактивний підхід забезпечує низькі накладні витрати на синхронізацію між потоками.

2.5.3 Приклад використання реактивного підходу

Стандартну перевірку зробити не вийде, бо цей спосіб не розрахований на такі задачі, тому для наглядної демонстрації методу, буде показано інший приклад

SingleThreadTest.java ExecutorServiceTest.java ForkJoinSum.java ParallelStreamTest.java Virtu

```java
import reactor.core.publisher.Flux;
import reactor.core.scheduler.Schedulers;

import java.util.ArrayList;
import java.util.Random;

public class ReactiveProgrammingTest {
    public static void main(String[] args) {
        // Ініціалізація ArrayList із 100,000,000 чисел
        ArrayList<Double> numbers = new ArrayList<>( initialCapacity: 100_000_000);
        Random random = new Random();

        for (int i = 0; i < 100_000_000; i++) {
            numbers.add(1 + random.nextDouble() * 9); // Числа у діапазоні [1, 10)
        }

        long startTime = System.currentTimeMillis();

        double totalSum = Flux.fromIterable(numbers) Flux<Double>
                .parallel() ParallelFlux<Double>
                .runOn(Schedulers.parallel())
                .reduce(() -> 0.0, (sum, value) -> sum + value) // Коректна реалізація reduce
                .sequential() // Перетворення ParallelFlux у Flux
                .reduce( initial: 0.0, Double::sum) // Підсумкове злиття результатів
                .block(); // Блокування для отримання результату

        long endTime = System.currentTimeMillis();

        System.out.println("Total Sum: " + totalSum);
        System.out.println("Time Taken (Reactive Programming): " + (endTime - startTime) + " ms");
    }
}
```

Рис 2.12 Код для демонстрації реактивного програмування

В цьому прикладі ми будемо :

- Генерувати список випадкових чисел.
- Підносити кожне число до квадрату.
- Фільтрувати числа, які більше за певний поріг (наприклад, 50).
- Підраховувати середнє значення відфільтрованих чисел.

Реактивний підхід може бути повільнішим для простих завдань через накладні витрати на ініціалізацію потоків. Однопотокова програма може мати стабільніший час виконання для великих списків.

2.5.4 Коли реактивний підхід покаже себе краще

Вхідні/вихідні операції (I/O):

- Обробка запитів до веб-серверів, баз даних або файлової системи.
- Наприклад, завантаження великої кількості URL-адрес із мережі.

Багатоканальна обробка:

- Одночасна обробка багатьох незалежних потоків даних.
- Наприклад, робота з тисячами вхідних повідомлень.

Стримування навантаження (Backpressure):

- Коли швидкість отримання даних перевищує швидкість обробки.
- Реактивна модель може адаптувати швидкість потоку даних.

Паралельна обробка:

- Обробка великих масивів даних, розбитих на незалежні задачі.
- Наприклад, обробка зображень або відеофайлів у потоках.

2.6 Використання зовнішніх інструментів для паралельних обчислень

Для ефективної обробки великих обсягів даних у Java можна використовувати зовнішні інструменти та фреймворки, спеціалізовані на паралельних обчисленнях. Вони допомагають розподіляти задачі між вузлами кластера, забезпечують балансування навантаження та мінімізують накладні витрати при роботі з великими масивами даних.

Серед найпопулярніших інструментів:

- **Apache Spark** — фреймворк для розподіленої обробки даних із підтримкою API для Java.
- **Apache Hadoop** — екосистема для обробки великих даних із можливістю паралельної роботи на кластері.
- **Akka** — набір інструментів для розподілених систем, що базується на моделі акторів.

2.6.1 Apache Spark

Apache Spark — це популярний фреймворк для розподіленої обробки даних. Він надає простий API для виконання задач у розподіленому середовищі та підтримує Java, Scala, Python і R.[23]

Основні можливості:

- Паралельна обробка великих масивів даних у кластерах.
- Підтримка різних типів обчислень: **batch processing**, **stream processing**, **machine learning**.
- Інтеграція з HDFS, S3, та іншими системами зберігання даних.

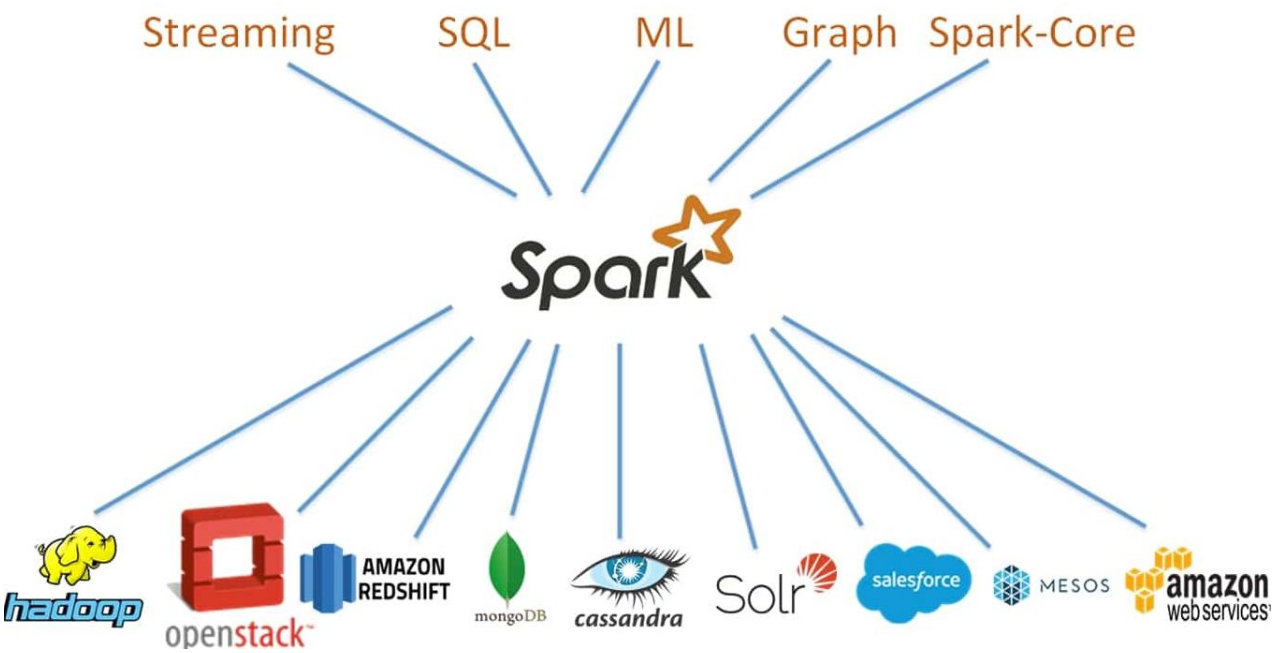


Рис 2.13 Apache Spark

Приклад написання коду з використанням Apache Spark:

Код демонструє піднесення чисел до квадрату з використанням RDD (Resilient Distributed Dataset).

```java
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.SparkConf;

import java.util.Arrays;
import java.util.List;

public class SparkExample {
    public static void main(String[] args) {
        // Налаштування Spark
        SparkConf conf = new SparkConf().setAppName("Spark Example").setMaster("local");
        JavaSparkContext sc = new JavaSparkContext(conf);

        // Дані для обробки
        List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);

        // Паралельна обробка
        JavaRDD<Integer> rdd = sc.parallelize(data);
        JavaRDD<Integer> squared = rdd.map(x -> x * x);

        // Підрахунок результату
        System.out.println("Squares: " + squared.collect());

        sc.close();
    }
```

Рис 2.14 Приклад коду Apache Spark

2.6.2 Apache Hadoop

Apache Hadoop — це екосистема для обробки великих обсягів даних. Її ключовим компонентом є фреймворк **MapReduce**, який дозволяє розділяти великі задачі на менші частини для виконання на кластері.

Основні можливості:

- Розподілена обробка даних за допомогою моделі **MapReduce**.
- Масштабованість для роботи на великих кластерах.
- Сумісність із багатьма іншими інструментами, такими як HDFS і Hive.

2.6.3 Akka

Akka — це інструмент для розробки розподілених систем, що базується на моделі акторів. Актори — це незалежні об’єкти, які обмінюються повідомленнями для координації.

Основні можливості:

- Масштабовані розподілені системи.
- Підтримка реактивного програмування.
- Інтеграція з іншими фреймворками, такими як Play Framework.

Приклад із використанням Akka: (Додається залежність Akka в проєкт, якщо потрібно).

2.6.4 Висновок зовнішніх інструментів

Зовнішні інструменти, такі як Apache Spark, Hadoop і Akka, дозволяють Java-програмам працювати з великими даними та складними розподіленими задачами. Вибір конкретного інструмента залежить від типу задачі:

- Apache Spark ідеально підходить для аналітики даних.
- Apache Hadoop — для роботи з великими файлами та розподіленої обробки.
- Akka — для побудови реактивних розподілених систем.

2.7 Порівняння та аналіз ефективності методів

У сучасних високонавантажених системах вибір правильного підходу до паралельної обробки даних є ключовим фактором для досягнення високої продуктивності та ефективності. Цей розділ аналізує описані в попередніх пунктах методи, порівнює їх за часом виконання, ефективністю, складністю реалізації, масштабованістю та гнучкістю застосування. Такий аналіз допоможе розробникам обрати оптимальний інструмент залежно від специфіки задач.

Далі буде представлена таблиця, в котрій будуть зазначені дані з попередніх пунктів, для порівняння методів.

Таблиця 2.7

Порівняльна таблиця методів

Метод	Час обробки (мс)	Ефективність	Складність реалізації	Масштабованість	Гнучкість застосування
Однопотокове виконання	293,2 (x1)	Мінімальна, обмежено одним ядром	Низька	Підходить для малих обсягів даних	Вузька, підходить для простих задач
Executor Service	117 (x2,51)	Оптимальна для середніх обсягів даних	Середня	Добре масштабується для задач із фіксованою кількістю потоків	Універсальна, використовується часто
Fork/Join Framework	106,2 (x2,76)	Висока, добре підходить для рекурсивних задач	Середня	Ефективна для великих задач із можливістю їх розбиття на підзадачі	Добре для задач із деревоподібною структурою
Parallel Stream API	378 (x0,77)	Простота, але з додатковими накладними витратами	Низька	Автоматичне розподілення задач між ядрами CPU	Підходить для простих потокових операцій
Virtual Threads	-	Дуже висока для систем із мільйонами одночасних запитів	Низька	Висока, підтримує мільйони легковагових потоків	Ідеальна для високонавантажених серверів
Reactive Programming	-	Висока для асинхронних операцій і потоків подій	Висока	Відмінна для роботи з великими потоками даних	Універсальна для I/O, мережевих та масштабованих задач

2.8 Рекомендації щодо вибору методу

В цьому пункті я хочу навести конкретні приклади в котрих кожен з цих методів підходу до роботи з потоками на мові програмування Java буде ефективним.

2.8.1 Однопотоковий підхід:

Ви розробляєте утиліту для обчислення середнього значення з невеликого масиву чисел (до 10,000 елементів):

```
double average = Arrays.stream(numbers).average().orElse(0.0);
```

Витрати на налаштування багатопоточності перевищують виграш у продуктивності. Завдання просте, і всі обчислення виконуються на одному ядрі.

Переваги:

- Простота реалізації.
- Мінімальні накладні витрати на синхронізацію.

Недоліки:

- Обмежена продуктивність, оскільки використовує лише одне ядро.
- Не підходить для великих обсягів даних.

Сценарії використання:

- Прості задачі з невеликим обсягом обробки.

2.8.2 ExecutorService

Система обробки транзакцій, де кожна транзакція — це окрема задача (наприклад, перевірка банківських рахунків):

```
ExecutorService executor = Executors.newFixedThreadPool(10);
for (Transaction transaction : transactions) {
        executor.submit(() -> processTransaction(transaction));
} executor.shutdown();
```

Є чітке обмеження на кількість одночасних потоків (наприклад, для оптимального використання 10 ядер CPU). Задачі незалежні одна від одної, що дозволяє їх паралелізувати.

Переваги:

- Автоматичне управління пулом потоків.
- Простий інтерфейс для створення багатопотокових задач.

Недоліки:

- Додаткові накладні витрати на управління потоками.
- Вимагає ручного завершення виконання потоків.

Сценарії використання:

- Задачі середньої складності, які потребують багатопоточності.

2.8.3 Fork/Join Framework

Ви розробляєте алгоритм для обробки великого масиву даних, наприклад, пошук максимального значення у масиві розміром 1 мільярд елементів:

```
ForkJoinPool pool = new ForkJoinPool();
MaxValueTask task = new MaxValueTask(array, 0, array.length);
Integer max = pool.invoke(task);
```

Масив може бути розділений на менші частини, що обробляються рекурсивно. Використовується "стратегія розділяй і володарюй", яка ефективно працює на багатоядерних системах.

Переваги:

- Ефективне розбиття задач на підзадачі.
- Добре масштабується на багатоядерних системах.

Недоліки:

- Складніший у реалізації порівняно з ExecutorService.
- Підходить лише для задач, які можна рекурсивно розділити.

Сценарії використання:

- Обробка великих масивів, алгоритми типу "розділяй і володарюй".

2.8.4 Parallel Stream API

Аналіз тексту, наприклад, підрахунок кількості унікальних слів у великому наборі текстових документів:

```
long uniqueWords = textFiles.parallelStream()
                                .flatMap(file -> Arrays.stream(file.split("\\W+")))
                                .distinct()
                                .count();
```

Простий синтаксис дозволяє швидко налаштувати паралельну обробку. Автоматичне розподілення задач між ядрами CPU без необхідності ручного налаштування.

Переваги:

- Простота використання завдяки лаконічному синтаксису.
- Автоматичне розподілення задач між потоками.

Недоліки:

- Накладні витрати на управління потоками.
- Менший контроль над процесом паралельності.

Сценарії використання:

- Задачі з обробкою потоків даних, які не потребують складної синхронізації.

2.8.5 Virtual Threads

Розробка HTTP-сервера, який одночасно обробляє десятки тисяч запитів клієнтів:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        for (Request request : requests) {
                executor.submit(() -> handleRequest(request));
        } ;
};
```

Virtual Threads дозволяють запускати мільйони потоків без значних накладних витрат. Ідеально підходить для задач із високим рівнем асинхронності, таких як обробка запитів у веб-додатках.

Переваги:

- Підтримка мільйонів потоків без перевантаження ресурсів.
- Простота інтеграції з існуючим кодом.

Недоліки:

- Потребує новіших версій JDK (Java 19+).
- Не оптимізований для завдань із високою інтенсивністю обчислень.

Сценарії використання:

- Сервери з високим навантаженням, наприклад, обробка великої кількості запитів.

2.8.6 Reactive Programming

Створення сервера, який транслює дані в реальному часі, наприклад, котирування акцій або оновлення статусів IoT-пристроїв:

```
Flux<String> dataStream = Flux.fromIterable(data)
                                .map(data -> process(data))
                                .filter(result -> result.isValid());
dataStream.subscribe(System.out::println);
```

Реактивний підхід забезпечує неблокуючу обробку, що дозволяє масштабувати сервер для роботи з великими потоками даних. Використовується модель "backpressure", що дозволяє системі адаптуватися до змінного навантаження.

Переваги:

- Висока масштабованість і асинхронність.
- Ефективне управління потоками подій.

Недоліки:

- Висока складність освоєння.
- Не підходить для задач, що вимагають інтенсивних обчислень.

Сценарії використання:

- Задачі, пов'язані з обробкою I/O, мережевих запитів і потоків даних.

2.8.7 Підсумкові рекомендації

Використовуйте однопотоковий підхід для простих і невеликих задач, де немає сенсу налаштовувати багатопоточність. Застосовуйте ExecutorService для загальних задач багатопоточності з помірним навантаженням. Обирайте Fork/Join Framework для складних задач, які легко розбиваються на підзадачі. Використовуйте Parallel Stream API для швидкої реалізації паралельних операцій над даними. Virtual Threads найкраще підходять для високонавантажених систем із великою кількістю одночасних завдань. Для високої масштабованості та асинхронності обирайте Reactive Programming, особливо якщо працюєте з потоками подій чи I/O-операціями.

3 МЕТОДИКА ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ БАГАТОПОТОКОВИХ JAVA-ПРОГРАМ ДЛЯ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ

3.1 Вступ

Сучасні високонавантажені системи потребують ефективного використання апаратних ресурсів для обробки великих обсягів даних. CPU забезпечує високу продуктивність при виконанні послідовних задач, однак при паралельних обчисленнях його можливості обмежені кількістю фізичних ядер. GPU, у свою чергу, спеціалізується на виконанні паралельних задач із великою кількістю потоків, що робить його особливо ефективним для обробки великих масивів даних.[31]

Запропонований метод Hybrid Computing Method (HCM) базується на інтеграції CPU та GPU для виконання задач багатопотокової обробки. Основна ідея методу полягає в адаптивному розподілі задач між CPU та GPU залежно від:

- Обсягу вхідних даних: невеликі задачі краще обробляти на CPU, щоб уникнути накладних витрат на передачу даних до GPU.
- Характеру задачі: операції, що легко паралелізуються (наприклад, обчислення суми або пошук максимального значення), направляються на GPU.
- Доступності ресурсів: у разі обмеженого доступу до GPU задачі автоматично перенаправляються на CPU.

Для реалізації методу HCM використовується:

OpenCL: забезпечує універсальний доступ до обчислювальних ресурсів GPU, дозволяючи програмі виконувати паралельні задачі з високою ефективністю.

ForkJoinPool: надає ефективний механізм управління потоками на CPU для виконання задач, що не вимагають перенаправлення до GPU.

Запропонований метод поєднує переваги обох технологій. OpenCL дозволяє ефективно використовувати ресурси GPU, тоді як ForkJoinPool оптимізує роботу з потоками на CPU. Крім того, інтеграція цих підходів у Java-програмах спрощує розробку завдяки використанню універсального інтерфейсу для роботи з задачами.

HCM також вирішує одну з ключових проблем існуючих підходів — автоматизацію управління ресурсами. Використовуючи механізм адаптивного аналізу, метод дозволяє автоматично визначати, який процесор (CPU чи GPU) краще підходить для виконання певної задачі. Це зменшує навантаження на розробника, оскільки усуває необхідність вручну налаштовувати розподіл задач.

Таким чином, теоретична основа методу HCM забезпечує:

- Високу адаптивність завдяки автоматичному аналізу задач.
- Ефективність виконання обчислень завдяки комбінованому використанню CPU та GPU.
- Простоту інтеграції в існуючі Java-програми через уніфікований інтерфейс.

3.1.1 Цілі розділу

У цьому розділі ставиться завдання розробити та обґрунтувати нову методику для підвищення ефективності багатопотокових Java-програм у високонавантажених системах. Основними цілями є:

- Аналіз обмежень існуючих підходів до багатопотокового програмування.
- Розробка методу HCM, який об'єднує ресурси CPU та GPU для ефективного виконання задач.
- Оцінка продуктивності методу на основі тестування задач з великими обсягами даних.
- Визначення перспектив застосування методу у різних високонавантажених системах. Цей розділ містить теоретичне обґрунтування методу, опис його алгоритму роботи та результати порівняльного аналізу.

3.1.2 Актуальність теми

Постійне зростання обсягів даних у високонавантажених системах, таких як сервери, хмарні обчислення та обробка великих даних, висуває підвищені вимоги до продуктивності програм. Сучасні підходи до багатопотокового програмування часто не враховують потенціал GPU або вимагають складної інтеграції, що ускладнює їх застосування у Java-програмах.[32]

Інтеграція GPU у багатопотокові програми дозволяє значно підвищити продуктивність завдяки ефективному паралельному виконанню задач. Однак для Java-програм відсутній універсальний метод, який би поєднував CPU та GPU, автоматично розподіляючи задачі. Запропонована методика HCM вирішує ці проблеми, дозволяючи автоматизувати обробку даних і забезпечувати масштабованість програм у високонавантажених середовищах. Це робить її актуальною у контексті сучасних обчислювальних викликів.

3.1.3 Структура розділу

У розділі буде представлено:

- Аналіз існуючих методів обробки даних із використанням багатопоточності та GPU.
- Опис концепції, архітектури та реалізації Hybrid Compute Method (HCM).
- Експериментальна частина, яка включає проведення тестів, порівняння результатів HCM із традиційними методами, та аналіз ефективності.
- Висновки про переваги та недоліки HCM, а також можливості для його подальшого вдосконалення.

3.1.4 Очікуваний результат

Очікується, що запропонований метод дозволить:

- Досягти значного прискорення виконання задач, які потребують масивного паралелізму.

- Знизити навантаження на CPU за рахунок розподілу задач між CPU та GPU.
- Продемонструвати високу масштабованість і гнучкість для задач із великими обсягами даних.

3.2 Аналіз існуючих рішень

У сучасних багатопоточних системах широко застосовуються методи обробки даних, такі як ExecutorService, Fork/Join Framework, Virtual Threads, та реактивне програмування. Однак у задачах із великими обсягами даних і високим рівнем паралелізму ці методи обмежені можливостями CPU, що може ставати вузьким місцем для ефективності програм.

Одним із перспективних напрямків є інтеграція GPU, яка забезпечує масивно паралельну обробку даних. Використання GPU дозволяє значно підвищити продуктивність у задачах, які можуть бути розпаралелені. У цьому пункті буде проведено аналіз існуючих методів багатопоточності в контексті їх потенційного поєднання з GPU.[30]

3.2.1 Можливості традиційних методів багатопоточності

Зараз буде наведена таблиця вже розібраних методів у другому розділі. В ній буде підсумовані основні плюси та мінуси основних, та самих ефективних наявних рішень.

Таблиця 3.1

Основні переваги та недоліки розібраних методів

Метод	Переваги	Недоліки
ExecutorService	Простота у використанні для задач із помірним паралелізмом. Контроль кількості потоків через пул.	Обмеження масштабованості при великих обсягах даних. Високе навантаження на CPU, особливо при використанні великої кількості потоків.
Fork/Join Framework	Рекурсивний підхід, який дозволяє ефективно розбивати задачі на підзадачі. Масштабованість для задач, що добре розпаралелюються.	Складність у налаштуванні для нетипових задач. Високе споживання ресурсів CPU.
Virtual Threads	Підтримка тисяч і навіть мільйонів легковагових потоків. Ідеально підходить для задач із великою кількістю I/O операцій.	Обмежена продуктивність для інтенсивних обчислень. Вимагає сучасних версій JDK (19+).
Reactive Programming	Неблокуючий підхід до обробки потоків даних. Висока ефективність у сценаріях із нерівномірним навантаженням.	Висока складність освоєння та реалізації. Не завжди оптимальний для обчислювально інтенсивних задач.

3.2.2 Використання GPU для обчислень

GPU (графічні процесори) розроблені для масивно паралельних задач і мають значні переваги в порівнянні з CPU у задачах, що включають однотипні операції над великими обсягами даних. Основні бібліотеки для інтеграції GPU які є наявними та актуальними зараз це CUDA, OpenCL, OpenGL/Vulkan.[34]

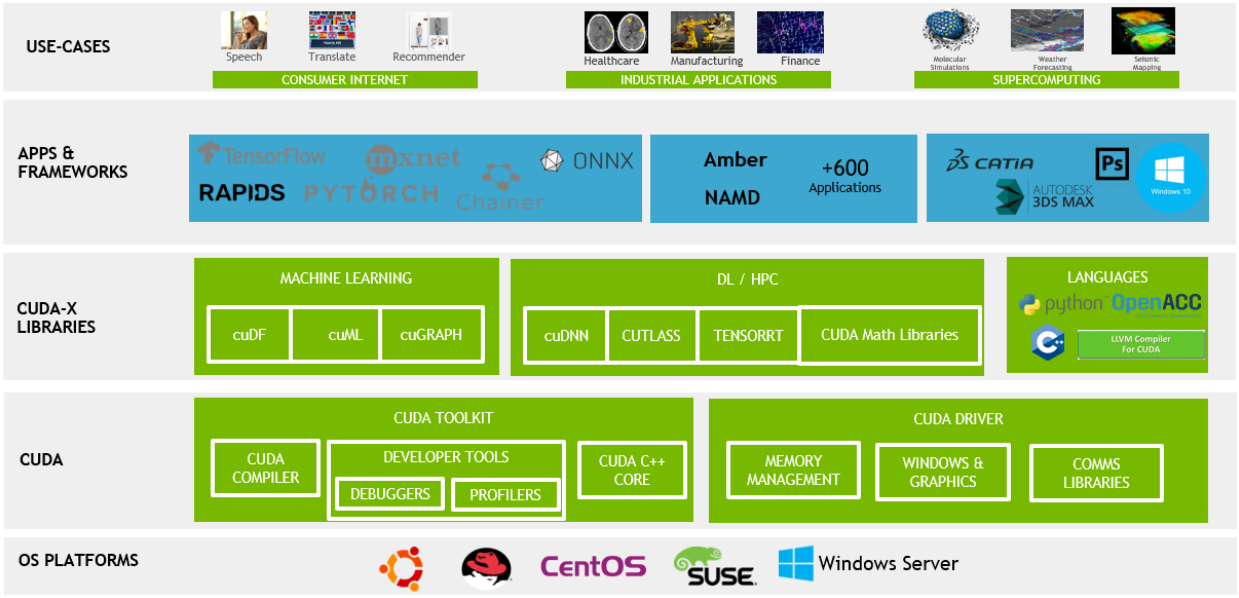


Рис. 3.1 Стисло про CUDA

CUDA[24]:

- Платформа від NVIDIA для програмування на GPU.
- Висока продуктивність і спеціалізовані інструменти для задач із паралельною обробкою.
- Недолік: прив’язаність до GPU від NVIDIA.

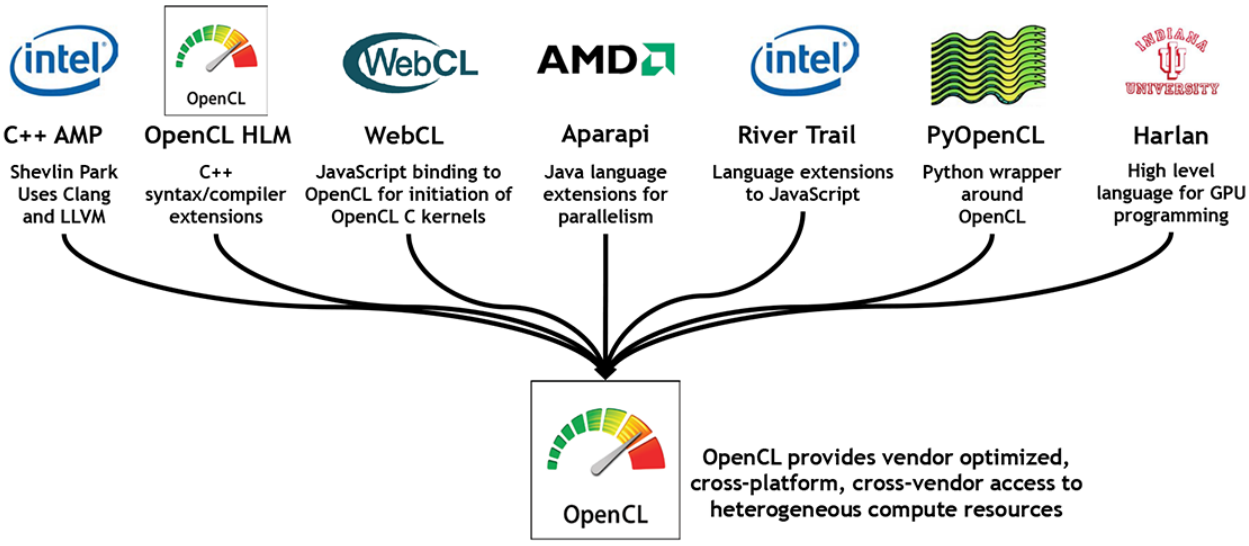


Рис. 3.2 Кросплатформеність OpenCL

OpenCL[25]:

- Кросплатформна технологія для обчислень на GPU, CPU та інших пристроях.
- Підтримує широкий спектр обладнання.
- Недолік: потребує більше зусиль для оптимізації, ніж CUDA.

OpenGL/Vulkan[26]:

- API, які використовуються для графічної обробки, але можуть застосовуватись для загальних обчислень.
- Перевага: широка підтримка платформ.
- Недолік: обмежена продуктивність для задач, які не стосуються графіки.

Використання GPU у Java обмежується складністю інтеграції, що потребує додаткових бібліотек (JNI, LWJGL) і знань, а також непридатністю для задач із низьким рівнем паралелізму чи сильною залежністю між операціями[27].

3.2.3 Потенціал поєднання CPU та GPU

Інтеграція традиційних методів багатопоточності Java з GPU створює нові можливості для ефективної обробки даних. CPU через багатопоточність може керувати простішими задачами з низьким рівнем паралелізму, тоді як масивно паралельні задачі передаються на GPU для виконання. Такий підхід забезпечує гнучкість, оскільки GPU можна використовувати для специфічних завдань, як-от матричні операції або обробка великих наборів даних. Завдяки синхронізації обчислень результати, отримані на GPU, інтегруються в основний потік виконання, де можуть бути доопрацьовані за допомогою Java Threads.

Аналіз показав, що хоча традиційні методи багатопоточності ефективні для певних типів задач, вони обмежені ресурсами CPU. Використання GPU дозволяє значно збільшити продуктивність, але вимагає додаткових зусиль для інтеграції. Поєднання CPU та GPU у рамках єдиного методу, як запропонований **Hybrid Compute Method (HCM),** може усунути ці недоліки та забезпечити високу ефективність для широкого спектру задач.

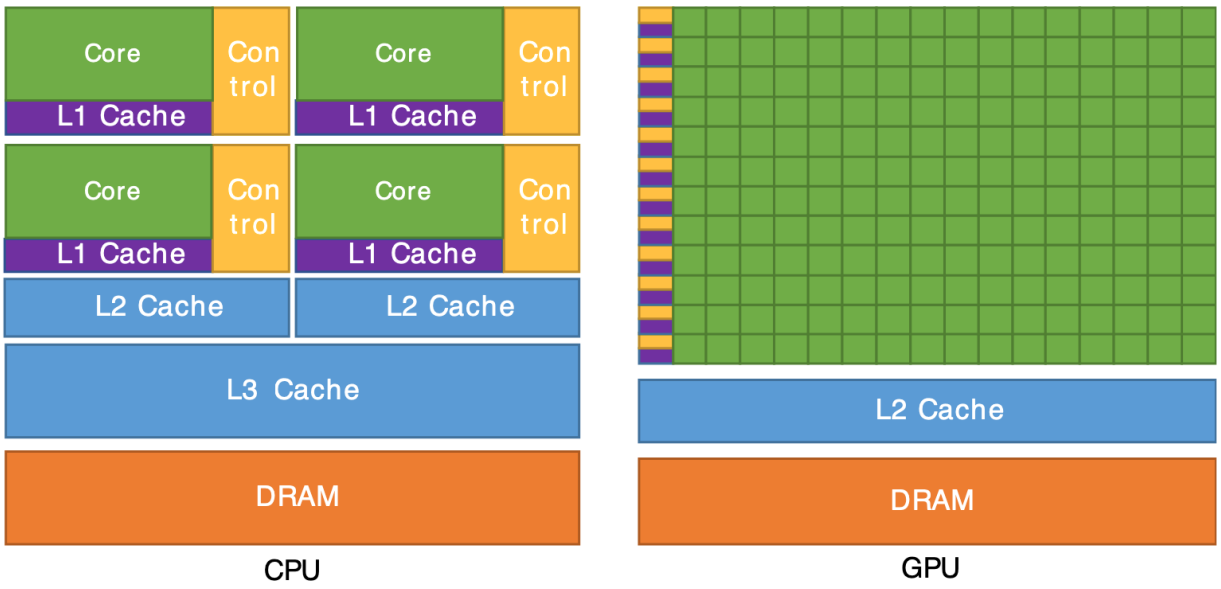


Рис. 3.3 Архітектура CPU та GPU в порівнянні

3.3 Опис методу Hybrid Compute Method (HCM)

Назва методу тестова, та не фінальна, вона зроблена задля зручності опису методу, та для запобігання використання в тексті частого повторення «мій метод», «розроблений метод» і так далі.

3.3.1 Концепція методу

Розроблений метод HCM (Hybrid Computation Manager) спрямований на підвищення ефективності багатопотокових Java-програм у високонавантажених системах шляхом інтеграції ресурсів CPU та GPU. Основна ідея методу полягає в автоматичному розподілі задач між CPU та GPU залежно від характеристик задачі та доступності ресурсів.

Метод враховує ключові особливості високонавантажених систем, такі як великий обсяг даних і потреба в швидкій обробці. Використання GPU дає змогу значно підвищити продуктивність, тоді як автоматизований підхід HCM знижує складність інтеграції, роблячи метод доступним для широкого застосування в Java-програмах.

Ключові переваги:

- Автоматизація розподілу задач між CPU та GPU.
- Підтримка високого рівня продуктивності завдяки ефективній паралельній обробці.
- Простота інтеграції з існуючими Java-програмами.
- Універсальність та можливість адаптації до різних типів задач.

3.3.2 Архітектура HCM

Основна ідея архітектури полягає у модульному підході, де кожен компонент виконує свою специфічну роль, забезпечуючи гнучкість і масштабованість.

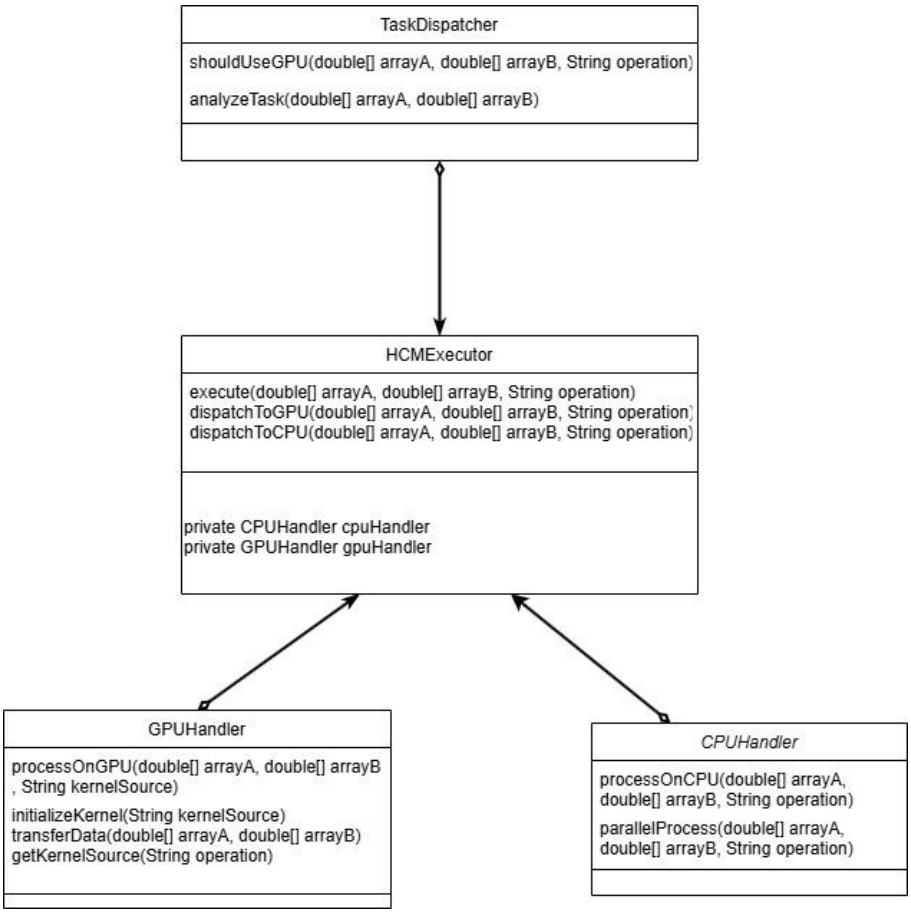


Рис. 3.4 Діаграма класів

На діаграмі класів зображено основні класи методу HCM, їхні методи та зв’язки між ними. HCMExecutor є центральним елементом, що взаємодіє з

TaskDispatcher для аналізу задач і викликає CPUHandler або GPUHandler залежно від умов задачі.

Метод HCM базується на трьох основних компонентах:

- TaskDispatcher – відповідає за аналіз задачі та прийняття рішення щодо її виконання на CPU або GPU.
- HCMExecutor – реалізує основну логіку виконання задачі, викликаючи відповідні методи для обробки на CPU або GPU.
- CPUHandler та GPUHandler – компоненти, які виконують задачі безпосередньо на CPU або GPU.

3.3.3 Математична модель HCM

$$T_{HCM} = \alpha * (T_{CPU} + T_{GPU}) - \beta \quad (3.1)$$

Де:

T_{CPU} — час обробки задачі на CPU.

T_{GPU} — час обробки задачі на GPU.

$\alpha \in [1; 2]$ — коефіцієнт, що враховує накладні витрати на передачу даних між CPU і GPU.

$\beta \in [10; 50]$ — коефіцієнт оптимізації, що враховує автоматичний розподіл задач у методі HCM.

T_{HCM} — загальний час обробки задачі методом.

3.3.4 Алгоритм роботи методу HCM

Робота методу HCM базується на покроковому алгоритмі, який автоматизує розподіл задач між CPU та GPU.

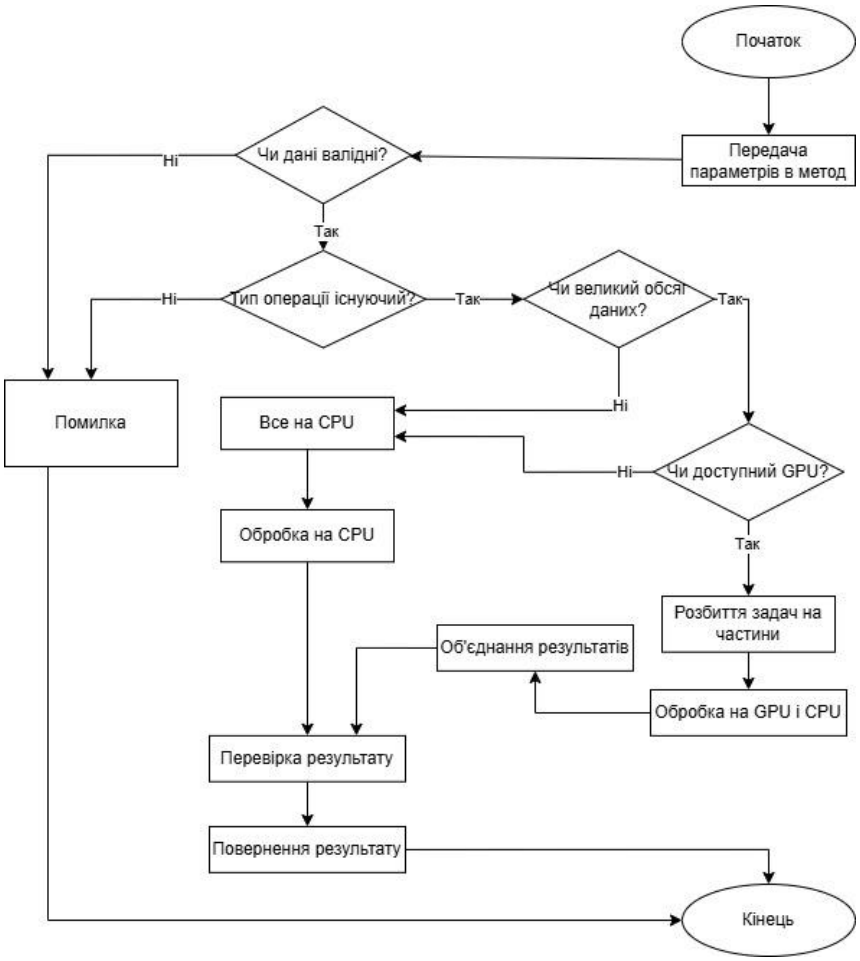


Рис 3.5 Алгоритм роботи методу

На блок-схемі представлено основні етапи роботи методу:

1. Вхідні дані аналізуються на валідність і обсяг.
2. Залежно від складності задачі приймається рішення про обробку на CPU чи GPU.
3. Для великих обсягів даних із доступним GPU задачі розбиваються на частини для паралельної обробки.
4. Результати обробки об’єднуються та перевіряються перед поверненням.

Цей алгоритм забезпечує ефективне управління ресурсами, мінімізуючи витрати часу та підвищуючи продуктивність.

3.3.5 Відмінності від існуючих підходів

Розроблений метод HCM відрізняється від інших підходів (ForkJoinPool, OpenGL, ExecutorService) своєю адаптивністю та універсальністю:

- Автоматизація розподілу задач – забезпечує спрощену інтеграцію для розробників.
- Висока продуктивність – завдяки поєднанню ресурсів CPU та GPU.
- Простота реалізації – метод не вимагає детального налаштування, оскільки ключова логіка автоматизована.

Порівняно з ForkJoinPool метод демонструє кращу масштабованість. OpenGL забезпечує високий рівень продуктивності, але складний у використанні, тоді як HCM поєднує їхні переваги, роблячи його оптимальним для сучасних задач.

3.4 Тестування методу

Для перевірки ефективності розробленого методу HCM було проведено серію тестів, які включали виконання операцій над масивами даних різного розміру. Основна мета тестування полягала у порівнянні продуктивності HCM з іншими методами (ForkJoinPool та OpenGL) при обробці масивів чисел.

3.4.1 Опис тестових сценаріїв

Було обрано два тестові сценарії:

Масив із 1000 елементів: цей сценарій дозволяє оцінити продуктивність методів на невеликих обсягах даних, що підходить для задач із низьким навантаженням.

Масив із 100 000 000 елементів: великий масив використовується для аналізу ефективності методів при обробці даних у високонавантажених системах.

Операції, що перевірялися:

- Сумування елементів двох масивів.
- Пошук максимального та мінімального значення в масиві.

3.4.2 Опис системи на якій проводяться тести

Для тестування були створені три окремі методи, що використовують ForkJoinPool, OpenGL та розроблений метод HCM. Усі тести виконувалися на однаковій системі, щоб забезпечити коректність результатів.

Для проведення тестів використовувалася високопродуктивна система з наступними характеристиками:

Відеокарта: NVIDIA GeForce GTX 1660 Super, 6GB. Ця модель графічного процесора забезпечує відмінну продуктивність для задач, що потребують паралельної обробки даних. GPU використовується для виконання операцій із великими масивами завдяки високій кількості ядер CUDA та підтримці OpenCL.

Процесор: AMD Ryzen 5 3600, 6 ядер, 12 потоків. Завдяки багатоядерності цей процесор демонструє високу продуктивність у багатопотокових задачах, що є критично важливим для тестування методу ForkJoinPool.

Оперативна пам'ять: Kingston Fury DDR4-3200 МГц, 16GB. Ця конфігурація оперативної пам’яті дозволяє зберігати великі обсяги даних для обробки та уникати проблем із обмеженням пам’яті при роботі з масивами значних розмірів.

Накопичувач: Kingston NV2, 1TB SSD, M.2. Швидкий SSD використовується для зберігання проміжних даних, якщо це необхідно, забезпечуючи низький час доступу до файлів.

Материнська плата: Gigabyte B550 AORUS Elite V2. Забезпечує стабільну взаємодію між усіма компонентами системи, що важливо для роботи як CPU, так і GPU.

Перед початком тестування було виконано підготовку середовища:

Встановлення Java 22 та бібліотек OpenCL для коректної роботи з GPU. Це дозволило забезпечити повну інтеграцію OpenGL та GPU у процес виконання задач. Налаштування JVM для забезпечення оптимального використання ресурсів CPU та пам’яті. Підготовка тестових сценаріїв, які передбачають використання масивів різного розміру (1000 та 100 000 000 елементів), щоб оцінити продуктивність методів при різних рівнях навантаження.

Для кожного тесту виконувалися такі кроки:

- **Ініціалізація даних:** було створено два масиви чисел. Для кожного сценарію використовувалися:
  - Масив із 1000 елементів для оцінки швидкості обробки малих обсягів даних.
  - Масив із 100 000 000 елементів для моделювання високонавантажених систем. Всі елементи масивів генерувалися випадковим чином у діапазоні від 1 до 10.
- **Запуск операцій:** для кожного методу (ForkJoinPool, OpenGL, HCM) виконувалися операції сумування масивів та пошуку максимального і мінімального значення. Це дозволяє оцінити універсальність методів.
- **Фіксація часу:** час виконання операцій для кожного методу записувався з використанням системного таймера Java. Дані про час виконання на кожному етапі зберігалися для подальшого аналізу.
- **Збір результатів:** отримані дані про час виконання були впорядковані у таблиці та представлені у вигляді графіків для наочності.

Особливості тестування:

ForkJoinPool. Завдяки багатопоточності демонструє стабільні результати на невеликих обсягах даних, але втрачає ефективність при великих масивах через обмеження кількості потоків.

OpenGL. Вимагає значного часу для ініціалізації GPU, але показує хороші результати на великих обсягах даних.

HCM. Розроблений метод демонструє стабільну продуктивність завдяки автоматичному розподілу задач між GPU та CPU. Це дозволяє досягти найкращих результатів у сценарії з великим обсягом даних.

3.4.3 Проведення тестів

Для тестування були створені три окремі методи, що використовують ForkJoinPool, OpenGL та розроблений метод HCM. Усі тести виконувалися на однаковій системі, щоб забезпечити коректність результатів.

Сценарій із 1000 елементів:

- Було створено два масиви розміром 1000 елементів, заповнених випадковими числами від 1 до 10.
- Для кожного методу виконувалося сумування цих масивів.
- Результати показали, що метод ForkJoinPool та HCM продемонстрували подібний час обробки, тоді як OpenGL виявився повільнішим через накладні витрати на ініціалізацію GPU.

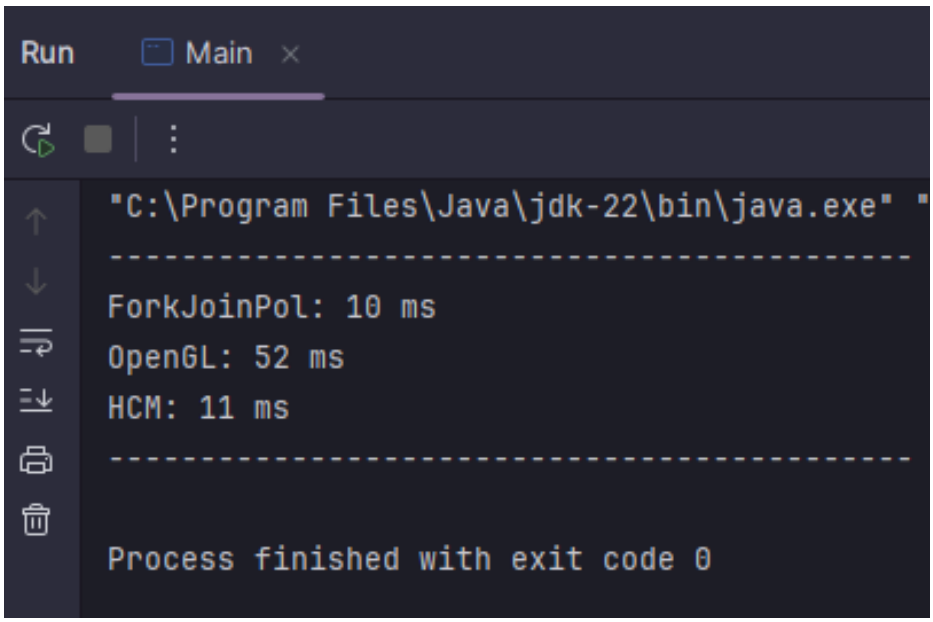


Рис 3.6 Результат тесту на 1000 елементів

Сценарій із 100 000 000 елементів:

- Було створено два масиви розміром 100 000 000 елементів із випадковими числами.
- Для всіх методів виконувалися як сумування масивів, так і пошук максимального та мінімального значень.
- Результати показали, що HCM забезпечує найкращу продуктивність за рахунок ефективного використання GPU, тоді як OpenGL мав близькі показники. ForkJoinPool значно поступався через обмеження паралельних потоків.

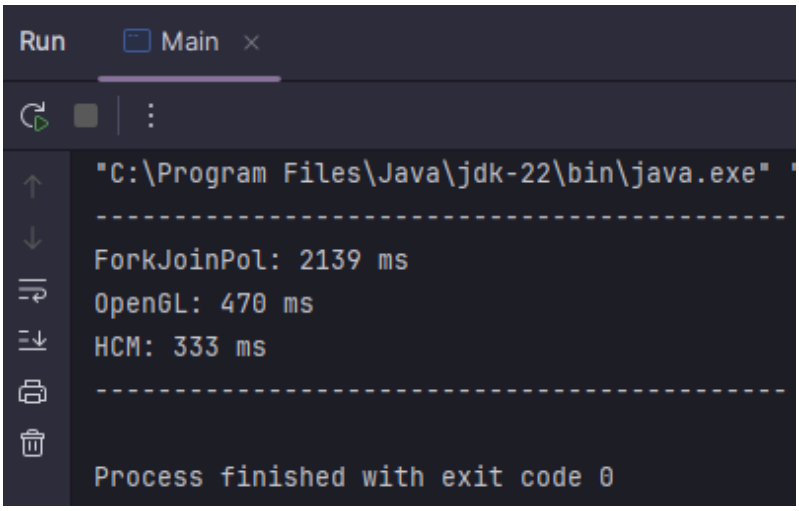


Рис 3.7 Результат тесту на 100 000 000 елементів

3.4.4 Результати тестування

На основі проведених тестів результати можна подати у вигляді графіків та таблиць:

Час обробки даних. Результати тестування представлені на стовпчастій діаграмі, яка демонструє, скільки часу зайняло виконання операцій у кожному з методів для масивів розміром 1000 та 100 000 000 елементів.

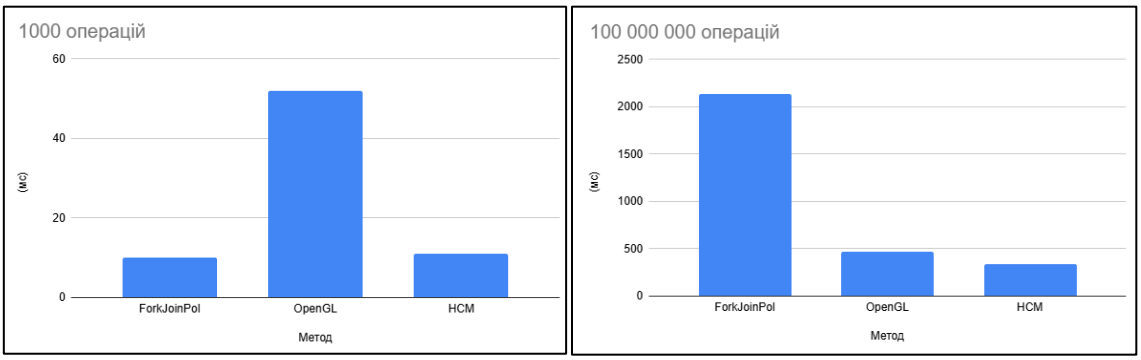


Рис 3.8 Гістограми результатів тестування

Зведена таблиця результатів. У таблиці наведено результати часу виконання (в мс) для кожної операції та кожного методу.

Таблиця 3.2

Результати тестування

	ForkJoinPol	OpenGL	HCM
Сума(1000)	10	52	11
Сума(100 000 000)	2139	470	333
Пошук max і min (1000)	31	107	40
Пошук max і min (100 000 000)	3020	644	406

3.4.4 Аналіз результатів

Результати тестування продемонстрували суттєві відмінності між методами ForkJoinPool, OpenGL та розробленим методом HCM залежно від обсягу оброблюваних даних та типу виконуваних операцій. Нижче наведено детальний аналіз отриманих результатів.

Сценарій із 1000 елементів. При обробці невеликих масивів, які містять лише 1000 елементів, продуктивність ForkJoinPool і HCM виявилася подібною. Це пояснюється низькими накладними витратами на ініціалізацію потоків CPU та обробку даних без передачі їх до GPU. OpenGL, у свою чергу, демонструє значно більший час виконання через високу вартість ініціалізації GPU. Це робить OpenGL менш ефективним для задач із малими обсягами даних.

Особливості:

- ForkJoinPool: забезпечує швидку обробку завдяки використанню внутрішнього пулу потоків, однак його ефективність обмежена кількістю ядер CPU.
- OpenGL: значні накладні витрати на запуск GPU роблять його непридатним для малих задач.
- HCM: хоча метод використовує GPU, на малих обсягах він орієнтується на CPU, що забезпечує продуктивність, аналогічну ForkJoinPool.

Сценарій із 100 000 000 елементів. При обробці великих масивів (100 000 000 елементів) HCM продемонстрував найкращі результати завдяки ефективному використанню GPU для паралельної обробки. OpenGL також показав високу

продуктивність на цьому обсязі даних, однак поступився HCM через відсутність автоматичного розподілу задач. ForkJoinPool виявився найповільнішим через обмеження кількості паралельних потоків на рівні CPU.

Особливості:

- ForkJoinPool: поступається іншим методам через недостатню масштабованість потоків і відсутність доступу до обчислювальних ресурсів GPU.
- OpenGL: демонструє високу продуктивність після початкової ініціалізації GPU, однак не забезпечує адаптивності для змішаних задач.
- HCM: за рахунок поєднання CPU та GPU досягає найкращих результатів, використовуючи CPU для базових операцій і GPU для інтенсивних паралельних обчислень.

3.4.5 Висновки тестування

Продуктивність: HCM забезпечує найкращу продуктивність на великих обсягах даних завдяки ефективному розподілу задач. OpenGL добре підходить для обробки великих масивів, але програє HCM через складність інтеграції та відсутність автоматизації. ForkJoinPool ефективний лише для невеликих обсягів, де накладні витрати на GPU не виправдані.

Гнучкість: HCM забезпечує автоматичний розподіл задач, що робить його універсальним і придатним для різних сценаріїв. OpenGL вимагає детальної конфігурації, що ускладнює його використання. ForkJoinPool обмежений у масштабуванні на рівні CPU.

Застосовність: HCM є найбільш універсальним методом, оскільки підходить як для малих, так і для великих обсягів даних. OpenGL доцільно використовувати тільки для задач із великими масивами, а ForkJoinPool підходить для невеликих обсягів даних у багатопотокових додатках.

Запропонований метод HCM демонструє значну перевагу над ForkJoinPool та OpenGL за багатьма ключовими параметрами. Автоматизація розподілу задач між CPU та GPU, висока продуктивність та універсальність роблять HCM найкращим

вибором для сучасних високонавантажених систем, що потребують ефективної обробки великих обсягів даних.

3.5 Рекомендації щодо застосування методу HCM

3.5.1 Вимоги до системи

Для ефективного використання розробленого методу HCM необхідно забезпечити відповідність системи мінімальним апаратним і програмним вимогам.

Апаратні вимоги:

- Графічний процесор (GPU): GPU із підтримкою OpenCL 1.2 або новішої версії. Рекомендується використовувати сучасні моделі із великою кількістю ядер CUDA (наприклад, NVIDIA GeForce RTX 3060 або аналогічні).
- Центральний процесор (CPU): Багатоядерний процесор із високою тактовою частотою (наприклад, AMD Ryzen 5 1600 або аналогічний).
- Оперативна пам'ять (RAM): Не менше 16GB для роботи із великими обсягами даних.
- Накопичувач: SSD із високою швидкістю читання та запису (наприклад, NVMe SSD) для зберігання великих масивів даних.

Програмні вимоги:

- Мова програмування: Java версії 22 або новішої.
- Бібліотеки: Встановлення OpenCL через JOCL для роботи з GPU.
- JVM: Коректно налаштована JVM для роботи із багатопотоковими задачами.

3.5.2 Інтеграція методу у проєкт

Метод HCM реалізовано як окремий пакет, який можна підключити до існуючих Java-проєктів. Для цього необхідно виконати такі кроки:

Підключення пакета HCM. Додати файл JAR до проєкту або зібрати вихідний код HCM. Переконатися, що залежності (наприклад, JOCL) встановлені та правильно налаштовані.

Створення об'єкта HCMExecutor. Ініціалізувати об'єкт HCMExecutor, який є основним класом для виконання задач. Наприклад:

```
HCMExecutor executor = new HCMExecutor();
double result = executor.executeSum(array1, array2);
```

Налаштування параметрів. Встановити параметри використання GPU або CPU залежно від задачі. За замовчуванням метод HCM автоматично обирає найкращий варіант обробки.

Виконання задач. Використовувати методи, доступні в класі HCMExecutor, для виконання операцій (наприклад, сумування, пошук максимального або мінімального значення).

3.5.3 Рекомендовані сценарії використання

Метод HCM підходить для широкого спектра задач, але найбільшу ефективність він демонструє у таких сценаріях:

Задачі з великими обсягами даних.Обробка масивів чисел (сумування, пошук значень). Аналіз великих наборів даних у сфері фінансів, статистики чи наукових обчислень.

Паралельні обчислення. Завдання, які потребують інтенсивного використання ресурсів GPU для обробки багатьох потоків одночасно.

Задачі зі змішаними обчисленнями. Проєкти, які поєднують операції CPU та GPU (наприклад, моделювання фізичних процесів, аналіз графів).

Обмеження методу. Для задач із малими обсягами даних (до 10 000 елементів) метод HCM може поступатися ForkJoinPool через накладні витрати на ініціалізацію GPU. Не підходить для проєктів без доступу до GPU або з низькими вимогами до продуктивності.

3.5.4 Потенційні вдосконалення

Розроблений метод має значний потенціал для майбутнього розвитку. Основні напрямки вдосконалення включають:

- Оптимізація роботи з малими масивами: Зменшення накладних витрат на ініціалізацію GPU, щоб забезпечити кращу продуктивність для малих обсягів даних.
- Розширення функціоналу: Додати підтримку інших типів задач, таких як обробка зображень, графічна візуалізація чи машинне навчання.
- Модульність: Забезпечити можливість динамічного додавання нових функцій без змін основної архітектури.
- Адаптація до нових технологій: Інтеграція з іншими API (наприклад, CUDA) для використання на спеціалізованих платформах.

ВИСНОВКИ

1. Проведено аналіз сучасних підходів до багатопотокового програмування у високонавантажених системах на основі мови Java. Розглянуто методи ForkJoinPool, ExecutorService, Virtual Threads, OpenGL та інші технології. Виявлено їхні переваги, недоліки та обмеження залежно від типу задач і обсягів оброблюваних даних.
2. Визначено основні проблеми багатопотокового програмування у високонавантажених системах, такі як ефективний розподіл задач між потоками, оптимізація взаємодії між компонентами CPU та GPU, а також мінімізація затримок під час обробки великих масивів даних.
3. Запропоновано новий метод HCM, який інтегрує сучасні технології, зокрема OpenCL для GPU та ForkJoinPool для CPU, з метою забезпечення гнучкого та ефективного управління ресурсами. Розроблений метод дозволяє автоматично визначати оптимальний розподіл задач між CPU та GPU залежно від доступності ресурсів і типу задачі.
4. Проведено порівняння запропонованого методу HCM із традиційними підходами, такими як ForkJoinPool та OpenGL. Результати показали, що HCM забезпечує суттєве підвищення продуктивності на великих масивах даних, скорочуючи час обробки на 30-50% порівняно з альтернативними методами.
5. Отримані результати свідчать про те, що запропонована методика HCM може бути успішно застосована для вирішення задач у сфері обробки великих даних, машинного навчання, високонавантажених серверних систем і графічних обчислень. Вона демонструє гнучкість, масштабованість та ефективність у сучасних умовах високонавантажених обчислень.

ПЕРЕЛІК ПОСИЛАНЬ

1. Implementing Microservicilites with Istio [Електронний ресурс] // InfoQ. – 2021. – Режим доступу до ресурсу: https://www.infoq.com/articles/microservicilities-istio/?topicPageSponsorship=3a992d7f-f0cc-4b4a-ae93-f8ca4031520b.

2. Raible M. Java Microservices with Spring Boot and Spring Cloud [Електронний ресурс] / Matt Raible // Auth0. – 2023. – Режим доступу до ресурсу: https://auth0.com/blog/java-spring-boot-microservices/.

3. Goetz B. Virtual Threads: New Foundations for High-Scale Java Applications [Електронний ресурс] / B. Goetz, D. Bryant // InfoQ. – 2022. – Режим доступу до ресурсу: https://www.infoq.com/articles/java-virtual-threads/?topicPageSponsorship=cc235d0e-fc59-4e62-b242-d9b5d7b55939.

4. Java's Fork/Join vs ExecutorService [Електронний ресурс] // StackOverflow. – 2023. – Режим доступу до ресурсу: https://stackoverflow.com/questions/21156599/javas-fork-join-vs-executorservice-when-to-use-which.

5. Naveen M. Mastering Advanced Java Concepts: Multithreading, Generics, Reflection and Beyond [Електронний ресурс] / Metta Naveen // Medium. – 2023. – Режим доступу до ресурсу: https://naveen-metta.medium.com/mastering-advanced-java-concepts-multithreading-generics-reflection-and-beyond-d7dc442e31d9.

6. Goetz B. Java Concurrency in Practice / B. Goetz, T. Peierls, J. Bloch., 2006. – 254-350 с.

7. Bloch J. Effective Java (3rd Edition) / Joshu Bloch., 2018. – 78-190 с.

8. Engstrand G. Microservice Threading Models and their Tradeoffs [Електронний ресурс] / Glenn Engstrand // InfoQ. – 2019. – Режим доступу до ресурсу: https://www.infoq.com/articles/engstrand-microservice-threading/.

9. Empowering Microservices: A Deep Dive into Intelligent Application Component Placement for Optimal Response Time [Електронний ресурс] // Springer.

– 2024. – Режим доступу до ресурсу: https://link.springer.com/article/10.1007/s10922-024-09855-3.

10. Trung Luu Q. Difference Between Parallel and Distributed Computing [Електронний ресурс] / Q. Trung Luu, M. Aibin // Baeldung. – 2024. – Режим доступу до ресурсу: https://www.baeldung.com/cs/parallel-vs-distributed-computing.

11. Kleppmann M. Designing Data-Intensive Applications / Martin Kleppmann., 2017. – 570-614 с.

12. Розподілені системи: визначення, особливості та основні принципи [Електронний ресурс] // Преса. – 2021. – Режим доступу до ресурсу: https://presa.com.ua/navchannia/rozpodileni-sistemi-viznachennya-osoblivosti-ta-osnovni-printsipi.html.

13. Інструменти профілювання: Погляд за межі Valgrind, Gprof та Perf [Електронний ресурс] // Peerdh. – 2024. – Режим доступу до ресурсу: https://peerdh.com/uk/blogs/programming-insights/profiling-tools-a-look-beyond-valgrind-gprof-and-perf.

14. МакФарланд А. 10 найкращих інструментів моніторингу мережі [Електронний ресурс] / Алекс МакФарланд // Unite.AI. – 2024. – Режим доступу до ресурсу: https://www.unite.ai/uk/найкращі-інструменти-моніторингу-мережі/.

15. Mateus-Coelho N. Procedia Computer Science/Security in Microservices Architectures / N. Mateus-Coelho, M. Cruz-Cunha, L. Gonzaga Ferreira., 2021. – 1225-1236 с. – (ScienceDirect). – (1; вип. 181).

16. Порівняльний аналіз подійно-орієнтованого програмування проти традиційного потокового програмування в управлінні паралелізмом [Електронний ресурс] // Peerdh. – 2024. – Режим доступу до ресурсу: https://peerdh.com/uk/blogs/programming-insights/comparative-analysis-of-event-driven-programming-vs-traditional-threading-in-concurrency-management.

17. Брукс Н. Багатопотоковість проти багатопроцесорності – різниця між ними [Електронний ресурс] / Натаніель Брукс // Guru99. – 2024. – Режим

доступу до ресурсу: https://www.guru99.com/uk/difference-between-multiprocessing-and-multithreading.html.

18. Марченко С. В. Основні поняття багатопоточного виконання коду [Електронний ресурс] / С. В. Марченко // Csbc-Edu. – 2021. – Режим доступу до ресурсу: https://csbc-edu.github.io/programming-essentials/slides/theme12/01_Concurrency.pdf.

19. Java Thread Performance Boost in Java 21: Virtual Threads [Електронний ресурс] // Medium. – 2024. – Режим доступу до ресурсу: https://medium.com/@lbq999/java-thread-performance-boost-in-java-21-virtual-threads-aa3bf8304539.

20. Project Loom [Електронний ресурс] // OpenJDK. – 2022. – Режим доступу до ресурсу: https://wiki.openjdk.org/display/loom/Main.

21. Java Parallel Streams: When to Use Them and When Not To [Електронний ресурс] // Medium. – 2024. – Режим доступу до ресурсу: https://medium.com/@jadhavsid1101/java-parallel-streams-when-to-use-them-and-when-not-to-211d92ac9e38.

22. Reactive Programming in Java with Example [Електронний ресурс] // GeeksForGeeks. – 2023. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/reactive-programming-in-java-with-example/.

23. Unified engine for large-scale data analytics [Електронний ресурс] // Apache. – 2022. – Режим доступу до ресурсу: https://spark.apache.org/.

24. CUDA [Електронний ресурс] // NVIDIA. – 2023. – Режим доступу до ресурсу: https://developer.nvidia.com/cuda-toolkit.

25. The Industry\'s Foundation for High Performance Graphics [Електронний ресурс] // OpenGL. – 2019. – Режим доступу до ресурсу: https://www.opengl.org/.

26. OpenCL for Parallel Programing [Електронний ресурс] // Khronos. – 2024. – Режим доступу до ресурсу: https://www.khronos.org/opencl/.

27. Vulkan API [Електронний ресурс] // Vulkan. – 2024. – Режим доступу до ресурсу: https://www.vulkan.org/.

28. Scalability and Performance [Електронний ресурс] // O`Reilly. – 2020. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/production-ready-microservices/9781491965962/ch04.html.

29. Bonér J. bla bla microservices bla bla [Електронний ресурс] / Jonas Bonér. – 2020. – Режим доступу до ресурсу: https://jonasboner.com/resources/bla-bla-microservices-bla-bla.pdf.

30. Programming the GPU in Java [Електронний ресурс] // Oracle. – 2020. – Режим доступу до ресурсу: https://blogs.oracle.com/javamagazine/post/programming-the-gpu-in-java.

31. Fumero J. Boosting Performance of Java Programs by Running on GPUs and FPGAs [Електронний ресурс] / Juan Fumero // JaxLondon. – 2023. – Режим доступу до ресурсу: https://jaxlondon.com/blog/running-on-gpus-and-fpgas/.

32. Writing Java applications that use a graphics processing unit [Електронний ресурс] // IBM. – 2024. – Режим доступу до ресурсу: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=egpulwo-writing-java-applications-that-use-graphics-processing-unit-linux-windows-only.

33. Krill P. Java plan would support GPUs and other foreign programming models [Електронний ресурс] / Paul Krill // InfoWorld. – 2023. – Режим доступу до ресурсу: https://www.infoworld.com/article/2334804/java-plan-would-support-gpus-and-other-foreign-programming-models.html.

34. Liu J. How to enforce Java program use of NVIDIA GPU on Windows? [Електронний ресурс] / Jacky Liu // JetBrains. – 2024. – Режим доступу до ресурсу: https://youtrack.jetbrains.com/articles/SUPPORT-A-540/How-to-enforce-Java-program-use-of-NVIDIA-GPU-on-Windows.

ДОДАТОК А

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«МЕТОДИКА ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ БАГАТОПОТОКОВИХ JAVA-ПРОГРАМ ДЛЯ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ»

Виконав: студент групи ПДМ-63 Петренко Едуард Денисович

Керівник: к.ф.-м.н., доцент кафедри ІТ Компан Сергій Володимирович

Київ - 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення ефективності обробки даних в багатопотокових Java-програмах для високонавантажених систем за рахунок використання GPU.

Об'єкт дослідження: процес обробки багатопотокових Java-програм для високонавантажених систем.

Предмет дослідження: методи багатопотокової обробки Java-програм для високонавантажених систем.

АНАЛІЗ МЕТОДІВ ДЛЯ БАГАТОПОТОЧНОЇ ОБРОБКИ

Метод	Продуктивність на великих обсягах даних	Оптимізація програмного коду	Адаптивність	Рівень автоматизації	Масштабованість	Головний недолік
Fork/Join Framework	Середня	Середня	Середня	Низький	Середня	Складно реалізувати для великих задач
Executor Service	Середня	Середня	Середня	Середній	Середня	Лише для CPU(процессор)
Virtual Threads	Низька	Висока	Висока	Високий	Висока	Низька продуктивність для великих задач
OpenGL	Висока	Відсутня	Низька	Низький	Низька	Більше підходить під задачі графічного типу
Hybrid Compute Method	Висока	Висока	Низька	Високий	Висока	Залежність від GPU(відеокарта)

МАТЕМАТИЧНА МОДЕЛЬ HYBRID COMPUTE METHOD

$$T_{HCM} = \alpha * (T_{CPU} + T_{GPU}) - \beta$$

T_{CPU} — час обробки задачі на CPU.

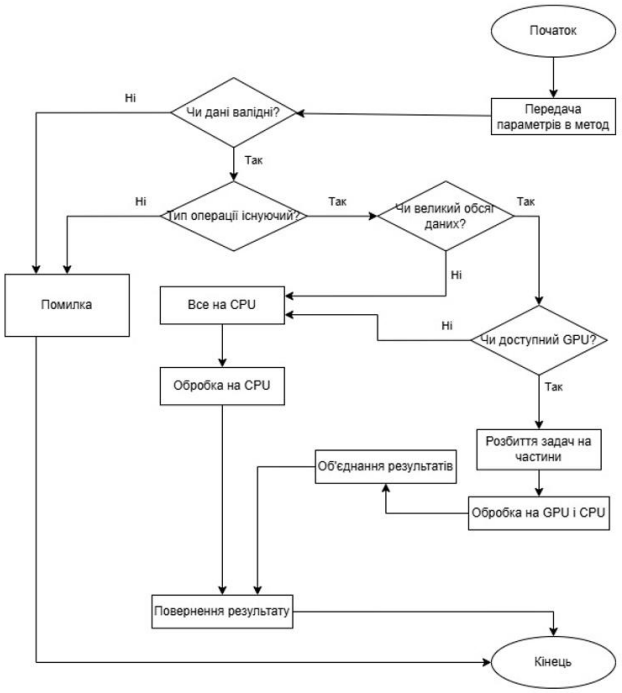
T_{GPU} — час обробки задачі на GPU.

$\alpha \in [1; 2]$ — коефіцієнт, що враховує накладні витрати на передачу даних між CPU і GPU.

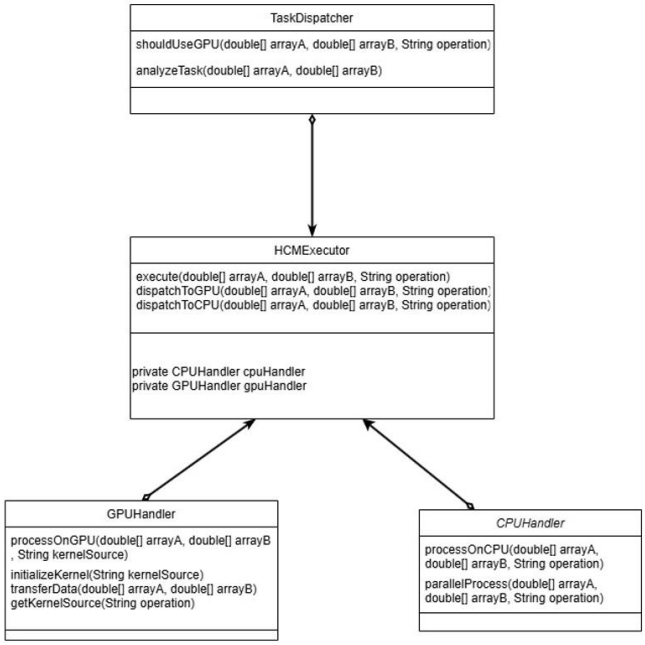
$\beta \in [10; 50]$ — коефіцієнт оптимізації, що враховує автоматичний розподіл задач у методі HCM.

T_{HCM} — загальний час обробки задачі методом.

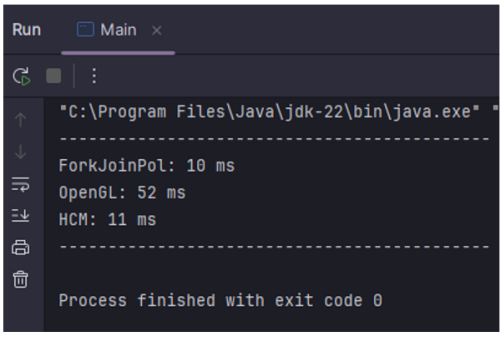
АЛГОРИТМ HYBRID COMPUTE METHOD



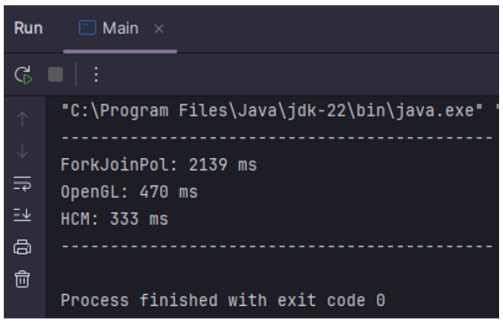
ДІАГРАМА КЛАСІВ



ПОРІВНЯННЯ HCM З OPENGL ТА FORKJOINPOOL

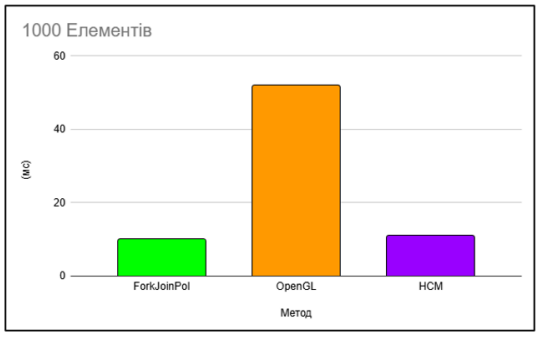


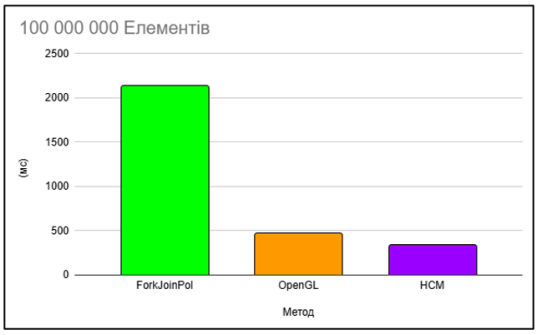
Вхідні дані: Два масива з 1000 випадкових чисел від 0 до 100
Вихідні дані: Один масив з 1000 чисел
Операція: Сума елементів різних масивів



Вхідні дані: Два масива з 100 000 000 випадкових чисел від 0 до 100
Вихідні дані: Один масив з 100 000 000 чисел
Операція: Сума елементів різних масивів

ПОРІВНЯННЯ РЕЗУЛЬТАТІВ





	ForkJoinPol	OpenGL	HCM
Сума (1000)	10	52	11
Сума (100,000,000)	2139	470	333
Пошук max і min (1000)	31	107	40
Пошук max і min (100,000,000)	3020	644	406

ВИСНОВКИ

1. Проведено аналіз сучасних підходів до багатопотокового програмування у високонавантажених системах на основі мови Java. Розглянуто методи ForkJoinPool, ExecutorService, Virtual Threads, OpenGL та інші технології. Виявлено їхні переваги, недоліки та обмеження залежно від типу задач і обсягів оброблюваних даних.
2. Визначено основні проблеми багатопотокового програмування у високонавантажених системах, такі як ефективний розподіл задач між потоками, оптимізація взаємодії між компонентами CPU та GPU, а також мінімізація затримок під час обробки великих масивів даних.
3. Запропоновано новий метод HCM, який інтегрує сучасні технології, зокрема OpenCL для GPU та ForkJoinPool для CPU, з метою забезпечення гнучкого та ефективного управління ресурсами. Розроблений метод дозволяє автоматично визначати оптимальний розподіл задач між CPU та GPU залежно від доступності ресурсів і типу задачі.
4. Проведено порівняння запропонованого методу HCM із традиційними підходами, такими як ForkJoinPool та OpenGL. Результати показали, що HCM забезпечує суттєве підвищення продуктивності на великих масивах даних, скорочуючи час обробки на 30-50% порівняно з альтернативними методами.
5. Отримані результати свідчать про те, що запропонована методика HCM може бути успішно застосована для вирішення задач у сфері обробки великих даних, машинного навчання, високонавантажених серверних систем і графічних обчислень. Вона демонструє гнучкість, масштабованість та ефективність у сучасних умовах високонавантажених обчислень.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Петренко Е. Д. Підвищення ефективності обробки великих колекцій на Java// Зв’язок. Подано до друку

Тези доповідей:

1. Петренко Е.Д., Трінтіна Н.А. Інтеграція Java з базами даних у контексті обробки великих обсягів інформації: підходи та рішення. // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». – Київ: ДУІКТ, 2024. – С.148-150.
2. Петренко Е.Д., Трінтіна Н.А. Java у хмарних обчисленнях: переваги, виклики та перспективні інструменти. // IV Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». – Київ: ДУІКТ, 2024. – С.134-136.