Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

В. В. Старченко

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ НА МОВІ JAVA

Навчальний посібник

З дисципліни «Паралельне програмування»

ЧНУ
ВИДАВНИЦТВО
Миколаїв – 2026

УДК 004.43Java(075.8)
С 77

Рекомендовано до друку Вченою радою Чорноморського національного університету імені Петра Могили (протокол № 8 від 30 червня 2025 р.)

Рецензенти:
Бондаренко Д. О., канд. техн. наук, доцент, зав. кафедри інформаційних систем Харківського національного економічного університету ім. Семена Кузнеця.
Михелєв І. Л., канд. техн. наук, доцент, зав. кафедри інформаційних систем і технологій Національного університету кораблебудування імені адмірала Макарова.

Старченко В. В.

С 77 Паралельне програмування на мові Java: навч. посіб. з дисципліни «Паралельне програмування». Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2026. 160 с.

ISBN 978-966-336-428-5

У посібнику розглянуті особливості розробки програмного забезпечення на мові Java для паралельних та розподілених комп'ютерних систем. Посібник призначено для студентів спеціальностей «Комп'ютерні науки» та «Комп'ютерна інженерія», а також може бути корисним студентам спеціальностей інших напрямів підготовки галузі знань «Інформаційні технології».

УДК 004.43Java(075.8)

ISBN 978-966-336-428-5

ЗМІСТ

ВСТУП

Сучасні обчислювальні системи стають все більш складними. Збільшується кількість ядер, впроваджуються кластерні архітектури. Традиційне послідовне програмування, хоч і залишається фундаментальним, часто не дозволяє повною мірою використати потенціал наявної апаратної платформи для розв'язання ресурсомістких задач. В умовах постійного зростання обсягів даних та вимог до швидкості їх обробки, паралельне програмування набуває особливої актуальності. Воно дозволяє розподілити обчислювальне навантаження між декількома процесорами, ядрами або навіть окремими комп'ютерами, значно прискорюючи виконання програм.

Курс «Паралельне програмування» присвячений вивченню основних концепцій, принципів та інструментів розробки ефективних паралельних програм з використанням можливостей мови Java. Мова Java, завдяки вбудованій підтримці багатопроцесності[1] та багатій бібліотеці concurrent–класів, є потужним інструментом для створення паралельних застосунків.

Цей навчальний посібник має на меті надати студентам та розробникам необхідні теоретичні знання та практичні навички для розуміння та реалізації паралельних алгоритмів. Посібник охоплює широкий спектр тем, від фундаментальних понять паралелізму до конкретних прикладів реалізації паралельних алгоритмів на мові Java.

Структура посібника побудована таким чином, щоб послідовно ввести читача в світ паралельного програмування:

Розділ «Головні поняття паралельного програмування» закладає теоретичний фундамент курсу. У ньому детально розглянуто умови та різноманітні шляхи досягнення паралелізму в обчислювальних системах, різні рівні паралелізму, що можуть бути присутні в інформаційній системі (від рівня інструкцій до рівня задач).

Також представлені різні режими паралельних обчислень (SIMD, MIMD тощо), основні парадигми паралельного програмування (наприклад, паралелізм даних, паралелізм задач), типові помилки, з якими стикаються розробники паралельних програм (стан перегонів за даними, взаємне блокування), поняття інформаційного графа програми

[1] У описі мови Java використовуються два терміни thread і stream, які зазвичай перекладаються єдиним словом, як "потік", але мають принципово різні значення. Під терміном thread розуміється потік команд, тобто процес виконання програми, а під терміном stream – потік даних, що використовуються для обрахунків за допомогою інструкцій програми. Тому далі потік команд (thread) будемо позначати словом "процес", потік даних (stream) – власно словом "потік".

як засобу візуалізації залежностей, методи оцінки максимально досяжного паралелізму для конкретної задачі та ключові етапи розробки ефективних паралельних алгоритмів.

Розділ «Паралельні архітектури» знайомить з основними типами паралельних архітектур, починаючи з класифікації Флінна, що групує архітектури за потоками інструкцій та даних. У ньому розглянуті принципи роботи конвеєрів, різні архітектури пам'яті (спільна, розподілена), а також особливості побудови та використання обчислювальних кластерів для великомасштабних паралельних обчислень.

Розділ «Основи багатопроцесності в Java» є практичним ядром посібника, присвяченим реалізації паралелізму засобами мови Java. У ньому описані основні стани життєвого циклу процесу в Java, розглянуті різні способи створення та запуску процесів, а також методи управління їх виконанням. Особлива увага приділена питанням синхронізації процесів для забезпечення коректної роботи зі спільними даними, а також програмним механізмам міжпроцесної комунікації в Java.

Розділ «Приклади реалізації паралельних алгоритмів» демонструє застосування вивчених концепцій та інструментів для реалізації класичних паралельних алгоритмів. У ньому розглянуто метод графа «Операції–операнди» для аналізу паралелізму в алгоритмах, а також представлені паралельні реалізації популярних обчислювальних методів, таких як метод Монте–Карло для імітаційного моделювання, метод найменших квадратів для апроксимації даних та метод розрахунку визначеного інтегралу. Ці приклади допоможуть закріпити теоретичні знання та отримати практичний досвід у розробці паралельних програм.

Засвоєння матеріалу цього посібника дозволить розробникам ефективно використовувати багатоядерні процесори та розподілені системи, створюючи високопродуктивне програмне забезпечення для розв'язання складних обчислювальних задач.

РОЗДІЛ 1
ГОЛОВНІ ПОНЯТТЯ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

В сучасному світі, де продуктивність обчислювальних систем відіграє ключову роль, розуміння принципів паралелізму стає невід'ємною частиною компетенції будь–якого розробника програмного забезпечення. Цей розділ закладає теоретичну основу для вивчення паралельного програмування. Розглядаються ключові умови та фундаментальні підходи до досягнення паралелізму в обчислювальних системах. Аналізуються різні рівні, на яких може бути реалізований паралелізм, від апаратного до розподілених систем, а також основні режими виконання паралельних розрахунків.

Особлива увага приділяється розгляду домінуючих парадигм паралельного програмування, таких як паралелізм даних та паралелізм задач, з акцентом на їх застосування в контексті мови Java. Обговорюються типові помилки, що виникають при розробці паралельних програм, включаючи умови перегонів за даними та взаємні блокування процесів.

Представлено методику аналізу потенціалу розпаралелювання програм за допомогою інформаційних графів. Обговорюються теоретичні моделі, зокрема ті, що сформульовані як закони Амдала та Густавсона–Барсіса, для оцінки максимально досяжного прискорення. На завершення, окреслюються ключові етапи системного підходу до розробки ефективних паралельних алгоритмів.

Опанування цих концепцій є критично важливим для розуміння принципів побудови високопродуктивних та надійних паралельних додатків на мові Java.

1.1 Умови досягнення паралелізму в паралельному програмуванні

Паралельне програмування дозволяє ефективно використовувати ресурси сучасних обчислювальних систем, які складаються з багатоядерних процесорів, графічних прискорювачів та кластерів. Для досягнення ефективного паралелізму необхідно враховувати низку умов, що забезпечують ефективну організацію обчислень. Розглянемо основні умови досягнення паралелізму.

1.1.1 Надлишковість елементів обчислювальної мережі

Для досягнення паралелізму важливо, щоб обчислювальна система мала надлишковість елементів, які можуть одночасно виконувати обчислення. Ця надлишковість забезпечується шляхом:

а) використання багатоядерних процесорів;

б) застосування кластерів, що складаються з багатьох обчислювальних вузлів;

в) інтеграції графічних процесорів (GPU) та інших спеціалізованих пристроїв.

Надлишковість елементів дозволяє виконувати кілька завдань одночасно, зменшуючи загальний час обчислень.

1.1.2 Незалежність функціонування окремих пристроїв ЕОМ

Для ефективного паралелізму обчислювальні пристрої повинні функціонувати незалежно один від одного. Це передбачає:

а) відсутність взаємних блокувань між пристроями під час виконання завдань;

б) мінімізацію взаємодії між пристроями, що дозволяє скоротити час очікування даних;

в) використання асинхронних методів обробки даних.

Незалежність функціонування сприяє підвищенню продуктивності обчислювальної системи за рахунок одночасного виконання декількох задач.

1.1.3 Можливість дублювання обчислювальних пристроїв ЕОМ

Дублювання обчислювальних пристроїв забезпечує можливість виконання ідентичних завдань на різних пристроях одночасно. Це досягається завдяки:

а) створенню резервних пристроїв для підвищення надійності системи;

б) використанню методів поділу завдань між кількома однаковими пристроями;

в) застосуванню векторних процесорів або матричних обчислювальних блоків для одночасної обробки великих обсягів даних.

Дублювання пристроїв також сприяє забезпеченню відмовостійкості системи.

1.1.4 Використання спеціалізованих обчислювальних пристроїв

Спеціалізовані обчислювальні пристрої, такі як GPU[2], TPU[3] , FPGA[4] та ASIC[5], дозволяють значно прискорити виконання певних типів завдань. Вони забезпечують:

а) високий рівень паралелізму завдяки великій кількості обчислювальних ядер;

б) оптимізацію обчислень для конкретних алгоритмів (наприклад, машинного навчання, обробки графіки);

в) зниження енергоспоживання у порівнянні з процесорами загального призначення.

Використання спеціалізованих пристроїв є провідною тенденцією розвитку сучасних паралельних обчислень.

1.2 Шляхи досягнення паралелізму

Ефективне паралельне програмування вимагає не лише розуміння синтаксису та API мови Java, але й глибокого усвідомлення принципів проєктування паралельних систем.

Досягнення високої продуктивності в паралельних застосунках залежить від п’яти основних чинників:

1) правильної декомпозиції задач;
2) уникнення взаємних блокувань;
3) мінімізації конфліктів під час доступу до спільних ресурсів;
4) ефективного балансування навантаження та зменшення накладних витрат.

1.2.1 Розбивка задачі на підзадачі

Ключова умова досягнення паралелізму обрахунків – можливість розділити основну задачу на незалежні або частково залежні підзадачі. Цей процес називається **декомпозицією задачі**. Декомпозиція задач є фундаментальним етапом розробки паралельних програм. Існує два основні підходи до розбивки: **декомпозиція за даними** (data parallelism) та **декомпозиція за завданнями** (task parallelism):

---

[2] Graphics Processing Unit – графічний процесор, англ.
[3] Tensor Processing Unit – тензорний блок обробки, англ.
[4] Field-Programmable Gate Array – програмовна користувачем логічна інтегральна схема, англ.
[5] Application-Specific Integrated Circuit – інтегральна схема для специфічного застосування, англ.

а) **декомпозиція за даними** передбачає розподіл великого набору даних між різними процесами, де кожен процес виконує однакові операції над своєю частиною даних. Цей підхід особливо ефективний для операцій над масивами, матрицями та колекціями;

б) **декомпозиція за завданнями** полягає в розбитті алгоритму на незалежні логічні блоки, які можуть виконуватися паралельно. Цей підхід підходить для конвеєрної обробки та ситуацій, де різні частини алгоритму виконують різні операції.

1.2.1.1 Критерії ефективної декомпозиції

При розбивці задач необхідно враховувати наступні критерії:

а) **гранулярність** – розмір підзадач повинен бути достатньо великим, щоб накладні витрати на створення процесів не перевищували користь від паралелізації;

б) **незалежність** – підзадачі повинні мати мінімальні залежності одна від одної;

в) **збалансованість** – обсяг роботи повинен бути рівномірно розподілений між процесами;

г) **локальність даних** – кожен процес повинен працювати з даними, що знаходяться в його локальній пам'яті.

1.2.2 Відсутність взаємних блокувань

Для досягнення ефективного паралелізму необхідно уникати **взаємоблокувань (deadlocks)**. Взаємне блокування виникає, коли два або більше процесів нескінченно чекають один на одного для звільнення ресурсів. Це одна з найсерйозніших проблем у паралельному програмуванні, оскільки вона може призвести до повного зависання програми. Для уникнення цього необхідно:

а) дотримуватись порядку захоплення ресурсів;

б) використовувати механізми *timeout* для очікування блокування;

в) уникати необов'язкових синхронізацій.

1.2.3 Мінімізація конфліктів через доступ до ресурсів

Конфлікти, або змагання за доступ до ресурсів (race conditions) виникають, коли кілька процесів одночасно намагаються отримати доступ до спільних ресурсів. Розрізняють наступні типи конфліктів:

а) **Read–Write конфлікти** – один процес читає, інший записує;

б) **Write–Write конфлікти** – декілька процесів одночасно записують;

в) **False sharing** – конфлікти на рівні кеш–ліній процесора.

Щоб уникнути змагань за ресурси, необхідно:

а) Використовувати синхронізовані колекції (наприклад, ConcurrentHashMap);

б) застосовувати конструкції synchronized, Lock або ReentrantLock для захисту критичних секцій;

в) проєктувати алгоритми без спільного стану (stateless algorithms).

1.2.4 Балансування навантаження

Нерівномірний розподіл роботи між процесами призводить до ситуації, коли одні процеси перевантажені, а інші простоюють. Це значно знижує ефективність паралелізації. Тому для ефективного використання обчислювальних ресурсів задачі повинні бути рівномірно розподілені між процесами. Це називається **балансуванням навантаження**. У Java для цього можна:

а) використовувати інструмент ForkJoinPool для автоматичного розподілу задач;

б) контролювати розмір задачі під час декомпозиції;

в) аналізувати продуктивність за допомогою профілювання.

1.2.5 Зменшення накладних витрат

Накладні витрати в паралельних програмах виникають через:

а) створення та знищення процесів;

б) синхронізацію між процесами;

в) переключення контексту;

г) конфлікти кешу процесора;

ґ) передачу даних між процесами.

Для їх зменшення в Java рекомендується:

д) використовувати пул процесів (наприклад, Executors.newFixed ThreadPool);

е) уникати надмірного створення дрібних задач;

є) переходити до асинхронного програмування, якщо це можливо (наприклад, CompletableFuture);

ж) мінімізувати кількість та розмири критичних секцій.

1.3 Рівні паралелізму в інформаційній системі

Сучасні обчислювальні системи, від персональних комп'ютерів до великих серверів і кластерів, використовують паралелізм на різних рівнях для підвищення продуктивності та ефективності (рис. 1.1). Паралелізм означає одночасне виконання кількох дій або обчислень. Розуміння різних рівнів, на яких може існувати паралелізм, є ключовим для розробки ефективних паралельних програм.

Для розробника, що використовує Java, ключовими є **паралелізм на рівні програми** (створення та управління процесами, синхронізація) та **паралелізм на рівні макроопераций** (використання паралельних Streams, бібліотек). Однак, розуміння всіх рівнів паралелізму дає цілісне уявлення про те, як досягається висока продуктивність сучасних комп'ютерів, і допомагає приймати обґрунтовані рішення при проєктуванні, реалізації та оптимізації паралельних застосунків. Тому розглянемо основні рівні паралелізму, що зустрічаються в інформаційних системах, починаючи від найвищого рівня абстракції (операційна система) і закінчуючи найнижчим (апаратне забезпечення процесора).

Рисунок 1.1 – Рівні паралелізму в інформаційній системі

1.3.1 Паралелізм на рівні завдань операційної системи (Task–Level Parallelism)

Це найвищий рівень паралелізму, де операційна система (ОС) керує одночасним виконанням кількох незалежних програм або процесів. Кожен процес є окремою сутністю зі своїм власним адресним простором, ресурсами (файлами, сокетами) та станом виконання.

1.3.1.1 Реалізація

ОС використовує **планувальник завдань** (scheduler) для розподілу часу центрального процесора (CPU) між активними процесами.

а) на **багатоядерних** або багатопроцесорних системах ОС може призначати різні процеси для виконання на різних фізичних ядрах, досягаючи таким чином **справжнього паралелізму**;

б) на **одноядерних** системах ОС створює ілюзію паралелізму за допомогою техніки **багатозадачності з поділом часу** (time–sharing multitasking). Вона швидко перемикає контекст (стан) процесора між різними процесами, виділяючи кожному невеликий квант часу. Це створює **конкурентність** (concurrency), а не істинний паралелізм, але дозволяє всім програмам просуватися у виконанні.

1.3.1.2 Приклади

Користувач одночасно працює з текстовим редактором, слухає музику через медіаплеєр, завантажує файли через веббраузер, поки у фоні працюють системні служби (наприклад, антивірус або служба оновлення).

1.3.1.3 Значення для програміста Java

Хоча цей рівень паралелізму керується переважно ОС, розробник повинен його враховувати. Програми можуть конкурувати за ресурси системи (CPU, пам’ять, дисковий ввід/вивід). Також програми можуть взаємодіяти між собою через механізми міжпроцесної взаємодії (Inter–Process Communication – IPC), такі як канали (pipes), сокети або спільна пам’ять.

1.3.2 Паралелізм на рівні програми (Program–Level Parallelism)

Цей рівень стосується розбиття **однієї програми** на кілька процесів (threads), які можуть виконуватися паралельно або конкурентно в межах одного (батьківського) процесу для вирішення спільної задачі.

1.3.2.1 Реалізація

Програміст явно проєктує та реалізує паралельну логіку за допомогою засобів мови програмування або спеціалізованих бібліотек. Головний механізм реалізації паралелізму на рівні програми в Java – **процеси (Threads)**. Вони виконуються в межах батьківського процесу – прикладної програми – і ділять з нею спільний адресний простір (пам’ять), що полегшує обмін даними між ними (через спільні змінні та об’єкти). Однак це призводить до необхідності використання

механізмів синхронізації (блокування, семафори, атомарні операції) для запобігання станів перегонів за даними (race conditions) та забезпечення цілісності спільних даних. У Java для цього використовуються ключове слово synchronized, класи Lock, Semaphore, атомарні класи з пакету java.util.concurrent тощо. Іноді батьківський процес може породжувати нові (дочірні) процеси для виконання підзадач. Це забезпечує кращу ізоляцію, але ускладнює обмін даними.

У Java основні інструменти для роботи з цим рівнем паралелізму – це клас java.lang.Thread, інтерфейс java.lang.Runnable та потужний пакет java.util.concurrent, що надає високорівневі абстракції, такі як пули процесів ExecutorService, фреймворк Fork/Join, паралельні колекції та засоби синхронізації.

1.3.2.2 Приклади

а) багатопотоковий вебсервер, де кожен новий запит клієнта обробляється за допомогою окремого процесу;

б) програма для наукових обчислень, яка розбиває велику матрицю або обчислювальну сітку на частини та обробляє їх паралельно на різних ядрах CPU;

в) графічний інтерфейс користувача (GUI), де один процес (Event Dispatch Thread у Swing або JavaFX Application Thread) відповідає за обробку подій та оновлення інтерфейсу, а інші процеси виконують тривалі фонові завдання (наприклад, завантаження даних з мережі).

1.3.2.3 Значення для програміста Java

Це **основний рівень фокусу** для курсу паралельного програмування. Саме тут вивчаються техніки декомпозиції задач, створення та управління процесами, синхронізації доступу до спільних ресурсів, обробки помилок у багатопроцесному середовищі та оптимізації паралельних алгоритмів.

1.3.3 Паралелізм на рівні макрооперацій (Macro–Operation Level Parallelism)

Цей рівень займає проміжне положення між паралелізмом на рівні програми та паралелізмом на рівні машинних команд. Він передбачає паралельне виконання великих обчислювальних блоків, функцій або ітерацій циклу в межах одного процесу чи програми. Часто асоціюється з **паралелізмом даних** (data parallelism), коли одна й та сама операція застосовується паралельно до різних елементів великої структури даних (наприклад, масиву).

1.3.3.1 Реалізація

а) **паралельні цикли** – розбиття ітерацій циклу (for, while) для паралельного виконання на кількох процесах;

б) **паралельні бібліотеки** – використання спеціалізованих бібліотек, які надають паралельні версії стандартних алгоритмів (наприклад, сортування, перетворення Фур'є, операції лінійної алгебри);

в) **векторні інструкції (SIMD)** – використання спеціальних інструкцій процесора (Single Instruction, Multiple Data), які дозволяють одній команді виконувати ту саму операцію одночасно над кількома елементами даних, упакованими в широкі регістри (наприклад, SSE, AVX в архітектурі x86);

г) **функціональне програмування** – сучасні мови, включаючи Java, пропонують засоби функціонального програмування, які спрощують вираження паралелізму даних. Наприклад, Java Streams API дозволяє легко перетворити послідовний потік обробки колекції на паралельний за допомогою методу .parallelStream() або .parallel();

г) **автоматична паралелізація компілятором** – деякі компілятори (особливо для мов C/C++ та Fortran) можуть автоматично аналізувати код (наприклад, цикли) і генерувати паралельний код для виконання на багатоядерних процесорах або з використанням SIMD–інструкцій.

1.3.3.2 Приклади

а) обробка великого зображення, де кожен процес обробляє свою частину пікселів;

б) множення двох великих матриць за допомогою паралельного алгоритму;

в) використання `IntStream.range(0, N).parallel().map(i-> foo(i)).sum()` у Java для паралельного обчислення суми результатів функції `foo()` для діапазону значень.

1.3.3.3 Значення для програміста Java

Розуміння цього рівня дозволяє ефективно використовувати високорівневі інструменти такі як Java Streams API або бібліотеки для паралельних обчислень, для прискорення обробки великих обсягів даних без необхідності вручну керувати процесами та синхронізацією на низькому рівні.

1.3.4 Паралелізм на рівні машинних команд (Instruction–Level Parallelism – ILP)

Цей рівень паралелізму реалізується **апаратно** всередині кожного ядра процесора і дозволяє виконувати кілька машинних інструкцій одночасно або в конвеєрному режимі, навіть якщо програма написана як послідовний потік команд. Мета ILP – максимально завантажити виконавчі пристрої процесора в кожному тактовому циклі.

1.3.4.1 Реалізація

Сучасні процесори використовують комплекс апаратних технік:

а) **конвеєризація (Pipelining)** – виконання інструкції розбивається на кілька послідовних етапів (наприклад, вибірка інструкції, декодування, виконання, доступ до пам'яті, запис результату). Конвеєр дозволяє різним інструкціям перебувати на різних етапах виконання одночасно. Якщо конвеєр має *N* стадій, в ідеалі він може завершувати одну інструкцію за кожен тактовий цикл після заповнення;

б) **суперскалярність (Superscalar Execution)** – ядро процесора містить кілька **паралельних виконавчих пристроїв** (наприклад, кілька АЛП – арифметико–логічних пристроїв, блоків для операцій з плаваючою комою, блоків завантаження/збереження даних). Це дозволяє процесору вибирати, декодувати та виконувати **кілька інструкцій** за один тактовий цикл, якщо між ними немає залежностей і є вільні виконавчі пристрої;

в) **щвиконання поза порядком (Out–of–Order Execution – OoOE)** – процесор може динамічно змінювати порядок виконання інструкцій відносно того, як вони йдуть у програмному коді. Це робиться для того, щоб обійти залежності даних (коли інструкція чекає на результат попередньої) і знайти незалежні інструкції, які можна виконати раніше на вільних виконавчих пристроях. Спеціальні апаратні механізми (наприклад, буфер перевпорядкування, регістри перейменування) гарантують, що результати будуть коректними, ніби інструкції виконувались послідовно;

г) **передбачення переходів (Branch Prediction)** – щоб конвеєр не простоював при зустрічі з інструкціями умовного або безумовного переходу (які змінюють порядок виконання), процесор намагається вгадати, куди відбудеться перехід (або чи відбудеться він взагалі), і починає спекулятивно виконувати інструкції з передбаченої гілки. Якщо передбачення виявилося правильним, час не втрачається. Якщо ні – результати спекулятивного виконання скасовуються, і конвеєр перезавантажується з правильної адреси.

1.3.4.2 Значення для програміста Java

Цей рівень паралелізму значною мірою **прозорий** для програміста. Його реалізація прихована в апаратному забезпеченні CPU. Однак **компілятори** (включаючи Just–In–Time (JIT) компілятор Java Virtual Machine) відіграють важливу роль. Вони генерують машинний код, намагаючись оптимізувати його таким чином, щоб апаратні механізми ILP могли ефективно працювати (наприклад, розгортаючи цикли, перевпорядковуючи інструкції, уникаючи зайвих залежностей даних). Знання про існування ILP може іноді допомогти написати код, який краще піддається оптимізації (наприклад, структуруючи цикли або зменшуючи кількість умовних переходів у критичних секціях).

1.3.5 Паралелізм на рівні машинних слів і арифметичних операцій (Bit–Level Parallelism)

Цей, найнижчий, рівень паралелізму стосується здатності процесора обробляти дані певної розрядності (ширини машинного слова) за одну операцію або використовувати спеціальні інструкції для паралельної обробки менших блоків даних до рівня окремих бітів.

1.3.5.1 Реалізація:

а) **ширина машинного слова:** основний аспект цього рівня – це розрядність архітектури процесора (наприклад, 8–бітна, 16–бітна, 32–бітна, 64–бітна). Процесор з більшою розрядністю може оперувати з більшими числами за одну інструкцію або обробляти більше даних одночасно. Наприклад, 64–бітний процесор може додати два 64–бітних цілих числа за один такт, тоді як 32–бітному процесору для цього знадобиться кілька операцій (наприклад, додавання молодшої та старшої частин з урахуванням переносу). Збільшення розрядності було одним з основних шляхів підвищення продуктивності у початковій фазі історичного розвитку комп'ютерів;

б) **Single Instruction, Multiple Data (SIMD)** – інструкції SIMD також можна розглядати на цьому рівні. Вони дозволяють одній інструкції застосовувати операцію (наприклад, додавання, множення) одночасно до кількох менших елементів даних (наприклад, чотирьох 32–бітних чисел або восьми 16–бітних чисел), які «упаковані» в один широкий SIMD–регістр (128–бітний, 256–бітний або навіть 512–бітний). Це форма паралелізму даних на рівні бітів/байт всередині регістрів;

в) **бітові операції** – деякі алгоритми (наприклад, у криптографії, стисненні даних, графіці) інтенсивно використовують побітові логічні

операції (AND, OR, XOR, зсуви), які процесор виконує дуже швидко над усіма бітами машинного слова паралельно.

1.3.5.2 Приклади:

а) виконання арифметичних операцій над 64–бітними числами (long в Java) на 64–бітній архітектурі;

б) використання SIMD–інструкцій для прискорення обробки мультимедійних даних (відео, аудіо), наукових обчислень, криптографії;

в) швидке обчислення контрольних сум або хеш–функцій за допомогою побітових операцій.

1.3.5.3 Значення для програміста Java

Вибір правильних типів даних (наприклад, використання long там, де потрібна 64–бітна арифметика на / відповідній платформі) може мати значення для продуктивності. JVM та JIT–компілятор можуть автоматично використовувати SIMD–інструкції для оптимізації певних операцій (наприклад, копіювання масивів, деяких циклів), особливо в новіших версіях Java.

1.4 Режими паралельних обрахунків

Паралельні обчислення можуть бути організовані у різних режимах залежно від завдань, які вирішуються, та архітектури обчислювальної системи. Режими паралельних обчислень забезпечують різні підходи до організації обчислювальних процесів. Вибір відповідного режиму залежить від типу завдання, архітектури системи та вимог до продуктивності й масштабованості. Розглянемо основні режими паралельних обчислень: багатозадачний режим, режим поділу часу та розподілені обчислення.

1.4.1 Багатозадачний режим

Багатозадачний режим передбачає виконання декількох завдань одночасно на одній обчислювальній системі. Основні характеристики цього режиму:

а) **чергування виконання завдань** – система швидко перемикається між завданнями, створюючи ілюзію одночасного виконання;

б) **застосування багатоядерних процесорів** – завдання можуть виконуватися паралельно на різних ядрах;

в) **оптимізація використання ресурсів** – цей режим дозволяє уникати простоїв обчислювальної системи.

Багатозадачний режим широко використовується в сучасних операційних системах для забезпечення роботи кількох програм одночасно.

1.4.2 Режим поділу часу

Режим поділу часу забезпечує ефективне використання ресурсів системи за допомогою розподілу обчислювального часу між кількома завданнями. Основні аспекти цього режиму:

а) **квантування часу** – кожне завдання отримує невеликий проміжок часу (квант) для виконання;

б) **пріоритетність завдань** – система може встановлювати пріоритети для завдань, забезпечуючи виконання критично важливих процесів;

в) **підвищення інтерактивності** – режим поділу часу забезпечує швидке реагування на запити користувачів у багатозадачних системах.

Цей режим активно використовується у багатокористувацьких системах, де потрібно одночасно обслуговувати велику кількість користувачів. На рис. 1.2 наведено часову шкалу з чітким розподілом часу між чотирма процесами (P1, P2, P3, P4) в рамках двох раундів виконання. У першому раунді кожен процес отримує два часових інтервали, а в другому – лише P1 і P2 отримують по одному інтервалу.

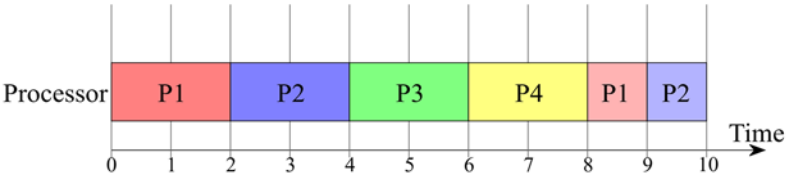


Рисунок 1.2 – Розподіл часу між чотирма процесами

1.4.3 Розподілені обчислення

Розподілені обчислення передбачають виконання завдань на кількох взаємопов'язаних обчислювальних вузлах, які можуть бути географічно розподілені. Основні характеристики цього режиму:

а) **мережеве з'єднання** – обчислювальні вузли обмінюються даними через мережу;

б) **масштабованість** – кількість обчислювальних вузлів можна збільшувати для підвищення продуктивності;

в) **висока надійність** – у разі відмови одного з вузлів система може

продовжувати роботу на інших вузлах.

Розподілені обчислення широко використовуються для вирішення великих наукових, технічних і комерційних завдань, таких як моделювання клімату, обробка великих даних та машинне навчання.

1.5 Парадигми паралельного програмування

Парадигми паралельного програмування визначають основні підходи до організації паралельних обчислень. Вони сприяють ефективному вибору інструментів для організації обчислювальних процесів у сучасних системах. Вибір парадигми залежить від специфіки задачі та доступної апаратної архітектури, що дозволяє досягати оптимального співвідношення продуктивності та складності реалізації.

1.5.1 Означення поняття парадигми

Парадигма у контексті паралельного програмування визначається як концептуальний підхід до структурування та виконання обчислювальних процесів, що дозволяє ефективно використовувати доступні ресурси. Розглянемо головні парадигми паралельного програмування, такі як **паралелізм даних** і **паралелізм завдань**.

1.5.2 Парадигма паралелізму даних

Паралелізм даних полягає у розподілі даних між кількома обчислювальними елементами для одночасної обробки. Основні риси цієї парадигми:

а) переваги:

1) ефективне використання ресурсів за рахунок розпаралелювання масивів великих даних;

2) простота реалізації для задач із чітко вираженою структурою даних;

3) підвищення продуктивності при обробці великих обсягів даних;

б) недоліки:

1) високі витрати на синхронізацію даних між обчислювальними елементами;

2) складність масштабування для задач із нерівномірним розподілом навантаження.

1.5.3 Парадигма паралелізму завдань

Паралелізм завдань передбачає одночасне виконання незалежних завдань на різних обчислювальних елементах. Основні риси цієї парадигми:

а) переваги:

1) гнучкість у розподілі завдань між обчислювальними елементами;
2) зниження залежності від структури даних, що обробляються;
3) можливість виконання різнорідних завдань одночасно;

б) недоліки:

1) складність координації виконання завдань із взаємозалежностями;
2) необхідність розробки ефективних методів балансування навантаження;
3) обмеження продуктивності через залежність між завданнями.

1.6 Типи помилок програміста

У процесі розробки програмного забезпечення програмісти стикаються з різними типами помилок. Кожна категорія помилок має свої характерні риси, причини виникнення та способи уникнення. Розуміння типів помилок програмування та їх причин є важливою складовою успішної розробки програмного забезпечення. Завдяки аналізу характерних рис, умов виникнення та способів уникання програмісти можуть значно підвищити якість і надійність своїх програм. Розглянемо основні категорії помилок, з якими стикаються програмісти на мові Java, приділяючи особливу увагу тим, що є специфічними для паралельних систем.

1.6.1 Синтаксичні помилки

Характерні риси

Синтаксичні помилки є порушеннями граматичних правил мови програмування Java. Вони виникають, коли код написано не відповідно до специфікації мови. Прикладами можуть бути друкарські помилки в ключових словах, пропущені крапки з комою в кінці інструкцій, невідповідність дужок або фігурних дужок, використання змінної до її оголошення, неправильний синтаксис виклику методу тощо.

Умови виникнення

Ці помилки виявляються на етапі компіляції коду. Компілятор Java (javac) аналізує вихідний код і, якщо знаходить синтаксичні

невідповідності, генерує повідомлення про помилку та припиняє процес компіляції. Виконуваний байт–код (.class файл) не створюється, доки всі синтаксичні помилки не будуть виправлені.

Способи уникання:

а) **уважність при написанні коду** – простий, але ефективний метод;

б) **використання інтегрованих середовищ розробки (IDE)** – сучасні IDE (Eclipse, IntelliJ IDEA, VS Code з відповідними плагінами) надають потужні інструменти для виявлення синтаксичних помилок у реальному часі, підсвічуючи їх безпосередньо в редакторі коду. Вони також пропонують автодоповнення коду, що зменшує ймовірність друкарських помилок;

в) **дотримання стандартів кодування** – узгоджений стиль кодування полегшує читання та виявлення помилок;

г) **регулярна компіляція** – не чекайте завершення написання великого блоку коду перед компіляцією. Часта компіляція дозволяє виявляти помилки на ранніх етапах.

1.6.2 Помилки часу виконання (Runtime Errors)

Характерні риси

Помилки часу виконання (також відомі як винятки – exceptions) виникають під час виконання програми, вже після її успішної компіляції. Вони сигналізують про непередбачені ситуації, які заважають нормальному виконанню програми. Типовими прикладами в Java є:

а) NullPointerException – спроба доступу до члена об'єкта за посиланням, яке має значення null;

б) ArrayIndexOutOfBoundsException – спроба доступу до елемента масиву за індексом, що виходить за межі допустимого діапазону;

в) ArithmeticException – Арифметична помилка, наприклад, ділення на нуль;

г) ClassCastException – спроба неправильного приведення типу об'єкта;

ґ) OutOfMemoryError – недостатньо пам'яті для створення нових об'єктів. Хоча OutOfMemoryError формально є помилкою (Error), а не винятком (Exception), її часто розглядають у цій категорії з точки зору програміста.

Умови виникнення

Ці помилки виникають через логічні недоліки, непередбачені вхідні дані, проблеми з ресурсами (пам'ять, файлова система, мережа) або

специфічні умови виконання, які не були враховані під час розробки. Наприклад, NullPointerException виникне, якщо метод очікує об'єкт, але отримує null і намагається викликати його метод.

Способи уникання:

а) **обробка винятків** – використання блоків try–catch–finally для перехоплення та обробки потенційних винятків, що дозволяє програмі відновити роботу або завершитися коректно;

б) **перевірка вхідних даних** – валідація даних, отриманих від користувача, з файлів або мережі, перед їх використанням;

в) **захисне програмування (Defensive Programming)** – додавання перевірок у коді для запобігання помилковим ситуаціям (наприклад, перевірка на null перед використанням об'єкта, перевірка меж масиву);

г) **керування ресурсами** – коректне використання ресурсів, наприклад, закриття файлів або мережевих з'єднань у блоці finally або за допомогою конструкції try–with–resources;

ґ) **ретельне тестування** – тестування програми з різними вхідними даними, включаючи граничні та некоректні випадки, для виявлення потенційних помилок часу виконання.

1.6.3 Логічні (алгоритмічні) помилки

Характерні риси

Логічні помилки є одними з найскладніших для виявлення. Програма з логічною помилкою компілюється успішно і може виконуватися без генерації винятків, але вона видає неправильний результат або поводиться не так, як очікувалося. Помилка полягає в неправильній реалізації алгоритму або невірному розумінні вимог. Прикладами можуть бути використання неправильної формули, помилка на одиницю (off–by–one error) в циклах або індексах, неправильні умови в операторах if або while, неврахування всіх можливих випадків.

Умови виникнення

Логічні помилки виникають через:

а) неправильне розуміння поставленої задачі або вимог;

б) помилки при проєктуванні алгоритму;

в) неправильна реалізація коректного алгоритму;

г) некоректне використання математичних операцій або логічних виразів.

Вони можуть проявлятися лише за певних вхідних даних або умов виконання, що ускладнює їх виявлення.

Способи уникання:

а) **ретельне проєктування та аналіз алгоритмів** – перед написанням коду переконайтеся, що ви чітко розумієте задачу та маєте коректний алгоритм її вирішення;

б) **модульне програмування** – розбиття складної задачі на менші, простіші модулі (методи, класи), які легше реалізувати та тестувати;

в) **написання тестів** – створення автоматизованих тестів (unit tests, integration tests), які перевіряють коректність роботи коду для відомих вхідних даних та очікуваних результатів. Може бути корисна методологія тест–орієнтованої розробки (Test Driven Development – TDD);

г) **відлагодження (Debugging)** – використання відлагоджувачів (debuggers) для покрокового виконання коду, перевірки значень змінних та відстеження процесу виконання програми;

г) **рецензування коду (Code Review)** – залучення інших розробників до перевірки коду допомагає знайти логічні помилки, які автор міг пропустити;

д**) використання тверджень (Assertions)** – додавання тверджень (assert) для перевірки інваріантів та припущень у коді під час розробки та тестування.

1.6.4 Помилки синхронізації (Synchronization Errors)

Цей тип помилок є специфічним для паралельного та багатопроцесного програмування.

Характерні риси

Помилки синхронізації виникають, коли кілька робочих процесів взаємодіють некоректно, зазвичай при доступі до спільних ресурсів (змінних, об'єктів, файлів). Основні типи помилок синхронізації:

а) **стан перегонів за даними (Race Condition)** – результат виконання залежить від непередбачуваного порядку (таймінгу) доступу процесів до спільних даних. Часто виникає, коли операція «читання–модифікація–запис» над спільним ресурсом не є атомарною;

б) **взаємне блокування (Deadlock)** – ситуація, коли два або більше процесів нескінченно чекають один на одного, оскільки кожен процес утримує ресурс, необхідний іншому процесу, і чекає на ресурс, утримуваний цим іншим процесом;

в) **живе блокування (Livelock)** – процеси активно виконують дії, реагуючи на стан інших процесів, але система в цілому не просувається до корисного результату. Процеси «надто ввічливі» і постійно поступаються один одному ресурсом;

г) **голодування (Starvation)** – один або кілька процесів постійно

не можуть отримати доступ до необхідних ресурсів або процесорного часу, тоді як інші процеси продовжують працювати.

Умови виникнення:

а) одночасний несинхронізований доступ кількох процесів до спільних змінюваних даних (*shared mutable state*);

б) неправильне використання механізмів синхронізації (ключове слово synchronized, об'єкти Lock, семафори, монітори);

в) некоректний порядок захоплення блокувань (призводить до deadlock);

г) складні залежності між процесами та ресурсами.

Ці помилки часто важко відтворити та діагностувати, оскільки вони залежать від таймінгу виконання процесів, який може змінюватися від запуску до запуску.

Способи уникання:

а) **мінімізація спільних змінюваних даних** – намагайтеся проєктувати систему так, щоб процеси працювали з незалежними даними або використовували незмінювані (immutable) об'єкти;

б) **правильне використання синхронізації** – використовуйте synchronized блоки/методи, явні блокування (java.util.concurrent.locks.Lock), атзмінні омарні (java.util.concurrent.atomic), процесобезпечні колекції (java.util.concurrent) для захисту доступу до спільних ресурсів;

в) **уникнення взаємних блокувань** – дотримуйтесь фіксованого порядку захоплення блокувань у всіх процесах. Використовуйте блокування з таймаутом (Lock.tryLock());

г) **використання високорівневих конструкцій** – застосовуйте інструменти з пакету java.util.concurrent (наприклад, ExecutorService, BlockingQueue, CountDownLatch, CyclicBarrier), які інкапсулюють складну логіку синхронізації;

ґ) **ретельне тестування паралельності** – використовуйте спеціалізовані інструменти та методики для тестування багатопроцесного коду під навантаженням, щоб виявити потенційні випадки *race conditions* та *deadlocks*;

д) **аналіз коду** – суттєво можуть допомогти виявити потенційні проблеми синхронізації також *статичний аналіз коду* та його *рецензування*.

1.6.5 Помилки комунікації

Ці помилки є особливо актуальними для розподілених систем, але можуть виникати і при міжпроцесної взаємодії в одній програмі,

особливо якщо використовуються складні протоколи обміну повідомленнями.

Характерні риси

Помилки комунікації стосуються проблем в обміні даними або сигналами між різними процесами, процесами чи вузлами системи. Основними типами помилок комунікації є наступні:

а) втрата повідомлень;

б) дублювання повідомлень;

в) порушення порядку доставки повідомлень (якщо порядок важливий);

г) пошкодження даних під час передачі;

ґ) таймаути при очікуванні відповіді;

д) некоректна серіалізація або десеріалізація об'єктів, що передаються;

е) проблеми з буферизацією (переповнення або недостатність буфера).

Умови виникнення:

а) ненадійність мережі (в розподілених системах);

б) помилки в реалізації протоколу комунікації;

в) несумісність форматів даних між відправником і одержувачем;

г) перевантаження системи або окремих її компонентів;

ґ) помилки в логіці обробки повідомлень (наприклад, неправильна обробка таймаутів або повторних передач).

Способи уникання:

а) **використання надійних протоколів** – застосування стандартних надійних протоколів (наприклад, TCP для мережевої взаємодії) або реалізація механізмів підтвердження доставки та повторної передачі на рівні застосунку;

б) **контроль цілісності даних** – використання контрольних сум або інших механізмів для виявлення пошкодження даних;

в) **обробка таймаутів та помилок** – реалізація логіки для обробки ситуацій, коли відповідь не надходить вчасно або коли виникають помилки передачі;

г) **ідемпотентність операцій** – проєктування обробників повідомлень таким чином, щоб повторне отримання та обробка одного й того ж повідомлення не призводили до некоректних результатів;

ґ) **управління процесом та буферизація** – коректне налаштування розмірів буферів та реалізація механізмів управління процесом даних (*flow control*) для запобігання перевантаженню;

д) **стандартизація форматів даних** – використання узгоджених форматів (наприклад, JSON, XML, Protocol Buffers) та механізмів

серіалізації;

е) **моніторинг та логування** – детальний моніторинг та логування комунікаційних процесів для полегшення діагностики проблем.

1.7 Інформаційний граф програми

Для ефективного розпаралелювання програми необхідно розуміти її внутрішню структуру та залежності між її частинами. Одним з ключових інструментів для аналізу цих залежностей є **інформаційний граф програми** (також відомий як граф залежностей за даними). Цей граф візуалізує потік даних та послідовність виконання операцій, що дозволяє ідентифікувати потенційні можливості для паралельного виконання.

1.7.1 Означення

Інформаційний граф програми – це орієнтований граф $G = (V, E)$,

де V – множина **вершин**, що представляють операції (обчислення, інструкції, виклики функцій або більші блоки коду) програми;

E – множина **дуг** (орієнтованих ребер), що представляють залежності за даними між операціями.

Дуга $(u, v) \in E$ від вершини u до вершини v означає, що операція v використовує результат, обчислений операцією u. Це накладає обмеження на порядок виконання: операція u має бути завершена перед тим, як операція v зможе розпочатися. Така залежність називається **потоковою залежністю** (flow dependency або read–after–write, RAW).

1.7.2 Призначення інформаційного графу

Аналіз інформаційного графу допомагає вирішити наступні завдання:

а) **виявлення паралелізму** – операції, між якими не існує шляху в графі (прямого чи опосередкованого через інші вершини), є незалежними і можуть потенційно виконуватися паралельно;

б) **ідентифікація критичного шляху** – найдовший шлях у графі від вхідних до вихідних даних визначає мінімальний час виконання програми, навіть за наявності необмеженої кількості процесорів. Оптимізація операцій на критичному шляху є пріоритетною;

в) **декомпозиція задачі** – граф допомагає розділити програму на менші незалежні або слабко залежні підзадачі, які можна призначити різним процесам або процесорам;

г) **планування виконання** – інформація про залежності

використовується для побудови оптимального розкладу виконання задач на паралельній архітектурі.

1.7.3 Приклад побудови інформаційного графу

Розглянемо простий фрагмент коду, що обчислює вираз $d = (x + y) \times (z/2) - 1$:

Алгоритм 1 Обчислення виразу $d = (x + y) \times (z/2) - 1$

Крок 1: op1: $a \leftarrow x + y$;

Крок 2: op2: $b \leftarrow z/2$;

Крок 3: op3: $c \leftarrow a * b$;

Крок 4: op4: $d \leftarrow c - 1$;

Представимо ці операції вершинами графу $V = \{op1, op2, op3, op4\}$. Визначимо залежності за даними:

а) $op3$ використовує результат a з $op1$. Отже, існує дуга ($op1$,$op3$);

б) $op3$ використовує результат b з $op2$. Отже, існує дуга ($op2$,$op3$);

в) $op4$ використовує результат c з $op3$. Отже, існує дуга ($op3$,$op4$).

Операції $op1$ та $op2$ не залежать одна від одної і можуть виконуватися паралельно. Операція $op3$ залежить від результатів $op1$ та $op2$. Операція $op4$ залежить від результату $op3$.

Графічно це можна зобразити, як наведено на рис. 1.3.

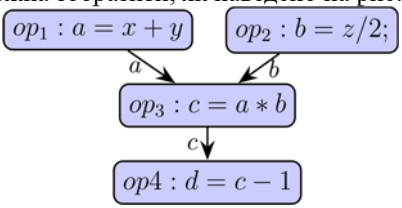


Рисунок 1.3 – Приклад інформаційного графу для обчислення виразу $d = (x + y) \times (z/2) - 1$

З цього графу видно, що $op1$ та $op2$ можуть виконуватися паралельно. Після їх завершення може виконуватися $op3$, а потім $op4$. Критичний шлях проходить через $op1 \rightarrow op3 \rightarrow op4$ або $op2 \rightarrow op3 \rightarrow op4$, залежно від того, яка з операцій ($op1$ чи $op2$) виконується довше.

1.7.4 Приклад: Ідеально послідовна модель

Розглянемо алгоритм, де кожна наступна операція безпосередньо залежить від результату попередньої. Наприклад, послідовне застосування функцій:

Алгоритм 2 Послідовне обчислення композиції функцій
Крок 1: op1: $y1 \leftarrow f1(x)$;
Крок 2: op2: $y2 \leftarrow f2(y1)$;
Крок 3: op3: $y3 \leftarrow f3(y2)$;
Крок 4: op4: $y4 \leftarrow f4(y3)$.

Тут $op2$ залежить від $op1$, $op3$ залежить від $op2$, $op4$ залежить від $op3$. Жодні дві операції не можуть виконуватися паралельно. Інформаційний граф для такого випадку є лінійним ланцюгом (рис. 1.4).

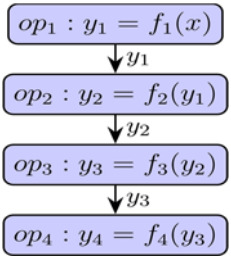


Рисунок 1.4 – Інформаційний граф для ідеально послідовної моделі.

Така структура представляє повну відсутність паралелізму на рівні цих операцій. Критичний шлях охоплює всі операції.

1.7.5 Приклад: Ідеально паралельна модель

Тепер розглянемо випадок, коли виконується одна й та сама операція над різними незалежними даними. Класичним прикладом є поелементна обробка масиву:

Алгоритм 3 Поелементна обробка масиву:
Крок 1: **for** $i \leftarrow 1$ to N **do**;
Крок 2: $y[i] \leftarrow f(x[i])$;
Крок 3: **end for**.

Якщо функція f не має побічних ефектів і кожна ітерація циклу працює з унікальними елементами $x[i]$ та $y[i]$, то всі ітерації (операції обчислення $f(x[i])$) є повністю незалежними одна від одної.

Нехай op_i позначає операцію $y[i] = f(x[i])$. Тоді для $N = 4$ інформаційний граф матиме вигляд, показаний на рис. 1.5.

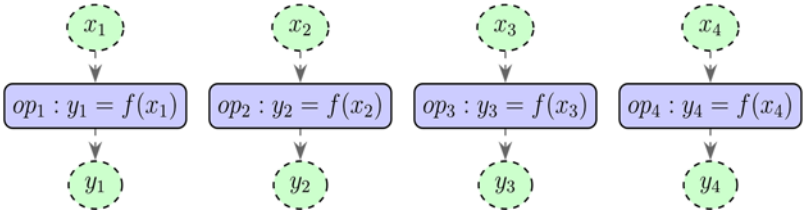


Рисунок 1.5 – Інформаційний граф для ідеально паралельної моделі (показано опціональні зв'язки з вхідними/вихідними даними)

У цьому графі відсутні дуги між основними операціями $op1, op2, op3, op4$. Це вказує на те, що вони можуть виконуватися повністю паралельно. Така структура ідеально підходить для розпаралелювання за даними (*data parallelism*).

1.7.6 Зв'язок з паралельним програмуванням на Java

В контексті Java, вершини інформаційного графу можуть відповідати:

а) окремим інструкціям або методам;

б) завданням, що реалізують інтерфейси Runnable або Callable.

Залежності між вершинами визначають порядок запуску та синхронізації процесів. Наприклад:

в) незалежні операції (як у ідеально паралельній моделі) можуть бути виконані паралельно за допомогою ExecutorService (наприклад, через submit() або invokeAll()) або *Parallel Streams* в Java 8+;

г) залежна операція (як $op3$ у першому прикладі або всі операції у послідовній моделі) повинна чекати на завершення тих операцій, від яких вона залежить. Це можна реалізувати за допомогою Future.get(), CompletableFuture або інших механізмів синхронізації (наприклад, CountDownLatch, CyclicBarrier);

ґ) послідовні ланцюги операцій (як у послідовній моделі) зазвичай виконуються в одному потоці, оскільки паралелізм тут не принесе користі;

д) використання спільних змінних (*shared mutable state*) між процесами також створює неявні залежності, які не завжди очевидні з простого аналізу процесу даних.

Ці залежності вимагають використання механізмів синхронізації (synchronized, Lock, атомарні змінні) для уникнення станів перегонів за даними (*race conditions*), що, в свою чергу, може впливати на порядок виконання і обмежувати паралелізм.

Розуміння інформаційного графу допомагає програмісту на Java обирати адекватні засоби з java.util.concurrent для ефективної реалізації паралельних алгоритмів.

1.7.7 Обмеження моделі

Важливо пам'ятати, що інформаційний граф представляє *потенційний* паралелізм, заснований на залежностях за даними. Реальний паралелізм та прискорення залежать від багатьох факторів, найбільш важливими з яких є наступні:

а) накладні витрати на створення процесів, управління ними та синхронізацію;

б) кількість доступних процесорних ядер;

в) балансування навантаження між процесами;

г) вплив кеш–пам'яті та архітектури системи пам'яті;

ґ) динамічна поведінка програми (наприклад, умовні переходи), яка може змінювати граф залежностей під час виконання.

Незважаючи на ці обмеження, інформаційний граф залишається потужним концептуальним інструментом для аналізу та проєктування паралельних програм.

1.8 Оцінка максимально досяжного паралелізму

Паралельне програмування дозволяє значно підвищити продуктивність обчислень шляхом використання декількох процесорних ядер або процесорів одночасно. Однак не всі алгоритми можуть бути ефективно розпаралелені. Тому одним з найважливіших питань при розробці паралельних програм є визначення теоретичних меж прискорення, яке може бути досягнуто під час розпаралелювання даного конкретного алгоритму. Розуміння цих меж допомагає програмістам приймати обґрунтовані рішення щодо архітектури програми та ефективного використання обчислювальних ресурсів. Розглянемо основні метрики для оцінки ефективності паралельних програм та фундаментальні закони, що визначають максимально досяжний паралелізм.

1.8.1 Основні поняття та метрики

1.8.1.1 Час виконання послідовної та паралельної програм

Позначимо:

а) T_1 – час виконання програми на одному процесорі (послідовне виконання);

б) T_p – час виконання програми на p процесорах (паралельне виконання);

в) p – кількість процесорів (ядер).

1.8.1.2 Коефіцієнт прискорення (Speedup)

Коефіцієнт прискорення Sp визначає, у скільки разів паралельна програма виконується швидше за послідовну:

$$S_p = \frac{T_1}{T_p} \tag{1.1}$$

Ідеальне (лінійне) прискорення досягається, коли $Sp = p$, тобто програма на p процесорах виконується у p разів швидше.

Розрізняють такі типи прискорення:

а) ***лінійне прискорення***: $Sp = p$ – ідеальний випадок;

б) ***сублінійне прискорення***: $Sp < p$ – найбільш поширений випадок;

в) ***суперлінійне прискорення***: $Sp > p$ – рідкісний випадок, можливий через ефекти кешування.

1.8.1.3 Коефіцієнт ефективності використання процесорів

Ефективність Ep показує, наскільки ефективно використовуються процесори:

$$E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p} \tag{1.2}$$

Ефективність вимірюється у долях від одиниці або у відсотках:

а) $Ep = 1$ (100 %) – ідеальна ефективність;

б) $Ep < 1$ – реальна ефективність (завжди менше 100 %);

Чим ближче Ep до 1, тим краще використовуються ресурси.

1.8.2 Закон Амдала

1.8.2.1 Формулювання закону

Закон Амдала, сформульований Джином Амдалом[6] у 1967 році, встановлює теоретичну верхню межу прискорення паралельної програми.

[6] Gene Myron Amdahl (16.11.1922 – 10.11.2015) – американський вчений в галузі обчислювальної техніки, математик і підприємець, найбільше відомий своєю роботою в компанії IBM.

Нехай програма складається з двох частин:

а) послідовна частина з часткою *f* від загального часу виконання;

б) паралельна частина з часткою (1 –*f*) від загального часу виконання.

Тоді час виконання на *p* процесорах:

$$T_p = T_1 \left(f + \frac{1 - f}{p} \right) \tag{1.3}$$

А максимальне прискорення згідно з законом Амдала:

$$S_p = \frac{1}{f + \frac{1-f}{p}} \tag{1.4}$$

1.8.2.2 Граничний випадок

При $p \to \infty$отримуємо максимально можливе прискорення:

$$S_{\max} = \lim_{p \to \infty} S_p = \frac{1}{f} \tag{1.5}$$

Це означає, що навіть при нескінченній кількості процесорів прискорення обмежене послідовною частиною програми.

1.8.2.3 Практичні наслідки закону Амдала:

а) якщо $f = 0{,}1$ (10 % послідовного коду), то $S_{\max} = 10$;.

б) якщо $f = 0{,}05$ (5 % послідовного коду), то $S_{\max} = 20$;

в) якщо $f = 0{,}01$ (1 % послідовного коду), то $S_{\max} = 100$.

1.8.3 Закон Густавсона–Барсіса

1.8.3.1 Обмеження закону Амдала

Закон Амдала припускає, що розмір задачі залишається постійним при збільшенні кількості процесорів. Однак на практиці часто збільшують розмір задачі пропорційно до доступних обчислювальних ресурсів.

1.8.3.2 Формулювання закону Густавсона–Барсіса

Джон Густавсон[7] та Едвін Барсіс[8] у 1988 році запропонували альтернативний підхід, який враховує масштабування розміру задачі.

[7] John L. Gustafson – американський вчений-комп'ютерник і бізнесмен, відомий головним чином своєю роботою в області високопродуктивних обчислень.

[8] Edwin H. Barsis. – американський вчений-комп'ютерник, співробітник Sandia National Laboratories.

Припустимо, що при фіксованому часі виконання T на p процесорах:

а) послідовна частина залишається постійною: s;

б) паралельна частина масштабується: $p \cdot (T - s)$.

Тоді прискорення згідно з законом Густавсона–Барсіса:

$$S_p = s + p \cdot (1 - s) = p - s \cdot (p - 1) \quad (1.6)$$

де s – частка послідовного коду від загального часу виконання на p процесорах.

1.8.3.3 Переваги закону Густавсона–Барсіса

Закон Густавсона–Барсіса:

а) дає більш оптимістичний прогноз прискорення;

б) краще відповідає реальним сценаріям використання;

в) показує, що при масштабуванні задачі можна досягти майже лінійного прискорення.

1.8.4 Порівняння законів Амдала та Густавсона–Барсіса

Для більшої наочності головні характеристики законів Амдала та Густавсона–Барсіса наведемо у табл. 1.1.

Таблиця 1.1 – Порівняння законів Амдала та Густавсона–Барсіса

Характеристика	***Закон Амдала***	***Закон Густавсона–Барсіса***
Розмір задачі	Фіксований	Масштабується
Обмеження прискорення	$\frac{1}{f}$	$p - s(p - 1)$
Застосування	Фіксовані задачі	Масштабовані задачі
Перспектива	Песимістична	Оптимістична

1.8.5 Практичні рекомендації

1.8.5.1 Вибір метрики оцінки:

а) використовуйте закон Амдала для задач фіксованого розміру;

б) використовуйте закон Густавсона–Барсіса для масштабованих задач;

в) завжди вимірюйте реальну продуктивність на цільовому обладнанні.

1.8.5.2 Оптимізація паралельного коду:

а) мінімізуйте послідовні ділянки коду;
б) зменшуйте накладні витрати на синхронізацію;
в) балансуйте навантаження між процесами;
г) враховуйте особливості архітектури процесора.

1.9 Основні етапи розробки паралельних алгоритмів

Розробка ефективних паралельних програм вимагає більше, ніж просто знання синтаксису паралельних конструкцій мови програмування (як–от процеси, завдання чи синхронізаційні примітиви в Java). Необхідний систематичний підхід до аналізу проблеми та проєктування алгоритму, який може ефективно використовувати доступні паралельні ресурси. Цей процес зазвичай поділяють на чотири ключові етапи, які часто є ітеративними та взаємопов'язаними.

Рисунок 1.6 – Класичний підхід до проєктування паралельних алгоритмів за моделлю Яна Фостера

Класичний підхід до проєктування паралельних алгоритмів, часто асоційований з моделлю Яна Фостера1 1Ian T. Foster – американський вчений–комп'ютерник, професор University of Chicago і співробітник Argonne National Laboratory , включає наступні чотири етапи (рис. 1.6):

а) декомпозиція (Partitioning / Decomposition);
б) комунікація (Communication);
в) агломерація (Agglomeration / Grouping);
г) відображення (Mapping / Assignment).

Хоча вони представлені послідовно, на практиці розробник часто повертається до попередніх етапів, щоб внести корективи на основі аналізу, зробленого на пізніших етапах. Наприклад, детальний аналіз характеристик комунікацій може виявити необхідність переглянути

початкову декомпозицію або стратегію агломерації, а вибір стратегії відображення (наприклад, *господар–працівник* чи *децентралізована*) може вплинути на вимоги до комунікацій та агломерації.

Розуміння цих етапів дозволяє розробникам приймати обґрунтовані рішення щодо структури паралельного алгоритму та потенційних компромісів. Тому розглянемо їх більш детально.

1.9.1 Декомпозиція (Partitioning / Decomposition)

Мета: Розбити обчислювальну задачу та/або дані, над якими вона працює, на якомога більшу кількість дрібних незалежних частин (задач або блоків даних). На цьому етапі фокус робиться на виявленні максимального потенційного паралелізму в проблемі, ігноруючи поки що обмеження цільової платформи.

Існують два основних типи декомпозиції:

а) **декомпозиція за задачами (Task Decomposition)** – проблема розбивається на набір функціонально відмінних завдань, які можуть виконуватися паралельно. Наприклад, в графічному редакторі одне завдання може обробляти введення користувача, інше – оновлювати екран, третє – виконувати складні обчислення у фоні;

б) **декомпозиція за даними (Data Decomposition)** – проблема розбивається шляхом поділу основної структури даних, над якою виконуються обчислення. Зазвичай одні й ті ж операції застосовуються до різних частин даних паралельно. Класичний приклад – обробка великого масиву або матриці, де кожен процес обробляє свою частину.

Ключовим результатом цього етапу є ідентифікація ***гранулярності*** (розміру) потенційних паралельних завдань. На цьому етапі бажано отримати якомога дрібнішу гранулярність, щоб забезпечити максимальну гнучкість на наступних етапах.

1.9.2 Комунікація (Communication)

Мета: Визначити необхідні взаємодії (комунікації) між задачами, ідентифікованими на етапі декомпозиції. Завдання рідко бувають повністю незалежними. Їм часто потрібно обмінюватися даними або синхронізувати свої дії.

На цьому етапі аналізуються:

а) **потоки даних:** які дані повинні передаватися між задачами?

б) **залежності:** які задачі залежать від результатів інших задач?

в) **синхронізація:** в яких точках виконання задачі повинні чекати одна одну?

Важливою задачею цього етапу є визначення структури та обсягу комунікацій, оскільки саме комунікаційні витрати часто стають основним обмежуючим фактором для продуктивності паралельних програм.

Комунікації можна класифікувати за кількома ознаками:

1.9.2.1 Класифікація за дистанцією

а) **локальні комунікації** – обмін даними відбувається між невеликою кількістю «сусідніх» задач. Це типово для алгоритмів, де декомпозиція даних створює логічну сітку або сусідство (наприклад, обробка зображень, скінченні різниці). Кожна задача взаємодіє лише з обмеженим числом партнерів;

б) **глобальні комунікації** – взаємодія охоплює велику кількість або всі задачі в системі. Прикладами є операції збору даних в одній точці, розсилки даних усім задачам або глобальна синхронізація (бар'єри). Такі комунікації можуть бути дорогими з точки зору часу та ресурсів мережі;

в) **неструктуровані (Іррегулярні) комунікації** – патерни обміну даними не є регулярними і можуть залежати від вхідних даних або проміжних результатів обчислень. Це характерно для задач на графах, розріджених матрицях, адаптивних методах.

1.9.2.2 Класифікація за структурою (патерном):

а) **точка–точка (Point–to–point)** – передача даних між двома конкретними задачами (відправником та отримувачем). Це базовий тип комунікації;

б) **колективні (Collective)** – скоординований обмін даними між групою задач. Основні типи колективних операцій:

в) *розсилка (Broadcast)* – одна задача надсилає однакові дані всім іншим задачам у групі;

г) *збір (Gather)* – кожна задача з групи надсилає дані одній визначеній задачі–отримувачу;

ґ) *розсіювання (Scatter)* – одна задача розподіляє різні частини свого буфера даних між різними задачами групи;

д) *всі–до–всіх (All–to–all)* – кожна задача надсилає окреме повідомлення кожній іншій задачі в групі (модифікації: «*allgather*» – кожна збирає дані від усіх, «*allreduce*» – редукція з розсилкою результату всім);

е) *редукція / Згортка (Reduction)* – дані від усіх задач у групі комбінуються за допомогою асоціативної операції (сума, максимум, мінімум, *логічне і/або* тощо) для отримання єдиного результату на одній або всіх задачах;

є) *бар'єр (Barrier)* – точка синхронізації, де жодна задача не може продовжити виконання, доки всі задачі групи не досягнуть цієї точки.

1.9.2.3 Класифікація за динамікою

а) **статичні комунікації** – партнери по комунікації та структура обмінів відомі заздалегідь (на етапі компіляції або на самому початку виконання) і не змінюються протягом роботи алгоритму. Характерно для багатьох алгоритмів з регулярною структурою даних (наприклад, обробка щільних матриць);

б) **динамічні комунікації** – структура комунікацій змінюється під час виконання програми. Партнери та обсяги даних можуть залежати від проміжних результатів обчислень. Це типово для адаптивних алгоритмів, задач балансування навантаження (наприклад, «крадіжка роботи» – work stealing), деяких алгоритмів на графах.

1.9.2.4 Класифікація за часом / синхронізацією

а) **синхронні комунікації (Synchronous)** – операції надсилання та отримання даних вимагають узгодження між задачами. Виклик операції (наприклад, «*send*») блокується доти, доки відповідна операція («*receive*») не буде ініційована на іншій стороні, і передача даних може безпечно відбутися (або гарантується, що буфер відправника можна перевикористовувати). Це спрощує логіку програми, але може призводити до простоїв, якщо одна із задач чекає на іншу;

б) **асинхронні комунікації (Asynchronous)** дозволяють задачі ініціювати операцію передачі даних (надсилання або отримання) і негайно продовжити виконання інших обчислень, не чекаючи завершення самої передачі. Завершення комунікації перевіряється пізніше за допомогою окремих операцій (наприклад, «*test*», «*wait*»). Це дозволяє ефективно перекривати обчислення та комунікації, що потенційно збільшує продуктивність, але значно ускладнює програмування через необхідність керування буферами та станом незавершених операцій. Часто реалізуються за допомогою *неблокуючих* викликів.

Розуміння цих характеристик комунікацій є критично важливим для вибору правильних примітивів синхронізації та передачі даних в Java та для оцінки потенційних вузьких місць продуктивності.

1.9.3 Агломерація (Agglomeration / Grouping)

Мета: об'єднати (агломерувати) дрібні задачі та/або комунікації, визначені на попередніх етапах, у більші одиниці. Цей етап є кроком від ідеалізованого максимального паралелізму до більш практичної реалізації, враховуючи реальні накладні витрати.

Можливі причини для виконання агломерації є наступними:

а) **зменшення комунікаційних витрат** – кожна комунікація має певні накладні витрати (латентність). Об'єднання багатьох дрібних повідомлень в одне велике може бути ефективнішим;

б) **збільшення гранулярності** – дуже дрібні задачі можуть мати накладні витрати на створення, планування та виконання, які перевищують корисну роботу. Агломерація збільшує розмір задач, покращуючи співвідношення обчислень до накладних витрат;

в) **покращення локальності даних** – групування задач, що працюють з одними й тими ж або близькими даними, може покращити використання кеш–пам'яті;

г) **масштабованість програмного забезпечення** – керування мільйонами дуже дрібних задач може бути складнішим, ніж керування тисячами більших задач.

На цьому етапі відбувається пошук компромісу: агломерація зменшує накладні витрати, але також може зменшити потенційний паралелізм і ускладнити балансування навантаження. Наприклад, якщо об'єднати занадто багато задач, може виявитися, що роботи вистачить лише для невеликої кількості процесорів/процесів.

1.9.4 Відображення (Mapping / Assignment)

Мета: призначити (відобразити) агломеровані задачі на конкретні паралельні ресурси (наприклад, процесорні ядра, процеси Java).

Основні цілі відображення:

а) **максимізація використання ресурсів** – забезпечити, щоб більшість процесорів/ядер були зайняті корисною роботою (балансування навантаження);

б) **мінімізація комунікаційних витрат** – якщо це можливо, задачі, що часто взаємодіють між собою, слід розміщувати на одному процесорі або на процесорах, що знаходяться близько один до одного (наприклад, мають спільний кеш або швидке з'єднання).

Існують два основних підходи до відображення:

а) **статичне відображення** – розподіл задач між процесами/процесорами визначається на початку виконання програми і не змінюється. Це підходить для задач з добре передбачуваною структурою та часом виконання;

б) **динамічне відображення** – розподіл задач відбувається під час виконання програми. Це дозволяє адаптуватися до змін навантаження та доступності ресурсів. Приклади включають пули процесів (ExecutorService в Java) та алгоритми «крадіжки роботи» (work–stealing).

Вибір стратегії відображення залежить від характеристик задач, комунікаційної структури та особливостей цільової паралельної платформи. В Java часто використовуються інструменти, що реалізують динамічне відображення (наприклад, ForkJoinPool).

1.9.4.1 Основні стратегії відображення та планування

Окрім поділу на статичне та динамічне, існують поширені архітектурні патерни (стратегії) для організації взаємодії та розподілу роботи між паралельними сутностями:

а) **господар–працівник (Master–Worker / Manager–Worker)** – одна сутність («господар» або «менеджер») відповідає за розподіл завдань між іншими сутностями («працівниками»). Працівники виконують завдання та повертають результати господареві, який може агрегувати їх або видавати нові завдання:

1) ***переваги:***
   - простота реалізації та централізований контроль;
   - легко реалізувати динамічне балансування навантаження, якщо завдання незалежні (господар видає нове завдання вільному працівнику);

2) ***недоліки:***
   - господар може стати вузьким місцем (якщо генерація/розподіл завдань або збір результатів є складним) та єдиною точкою відмови;
   - масштабованість обмежена можливостями господаря;

3) ***особливості реалізації в Java:***
   - може бути реалізовано за допомогою головного процесу, який подає завдання до ExecutorService;

б) **ієрархічні (Hierarchical)** – структура управління має кілька рівнів. Наприклад, головний господар може роздавати великі шматки роботи координаторам проміжних рівнів, які, в свою чергу, розбивають їх на менші завдання для працівників нижчого рівня:

1) ***переваги:***
   - покращена масштабованість порівняно з чистою моделлю «господар–працівник» за рахунок розподілу функцій керування;
   - може краще відповідати ієрархії апаратного забезпечення (MNUA–вузли, кластери);

2) ***недоліки:***
   - більша складність реалізації та комунікацій;

3) ***особливості реалізації в Java:***

– ієрархічну організацію черг та механізм «крадіжки роботи» використовує ForkJoinPool, що має риси цієї моделі;

в) **децентралізовані / Рівноправні (Decentralized / Peer–to–Peer)** – всі сутності (потоки, процеси) є рівноправними, немає єдиного центрального координатора. Задачі можуть передаватися безпосередньо між сутностями. Балансування навантаження часто досягається динамічно, наприклад, за допомогою «крадіжки роботи», коли вільні процеси «крадуть» завдання з черг інших процесів:

1) ***переваги:***

– висока потенційна масштабованість та відмовостійкість (немає єдиної точки відмови);

– добре підходить для задач, де комунікації переважно локальні або нерегулярні;

2) ***недоліки:***

– складніша координація та комунікації;

– реалізація ефективного динамічного балансування навантаження може бути нетривіальною;

– важче відстежувати глобальний стан системи;

3) ***особливості реалізації в Java:***

– яскравим прикладом використання децентралізованого балансування навантаження через *work–stealing* є ForkJoinPool;

– часто є децентралізованими також моделі на основі акторів;

г) **конвеєрні (Pipelined)** – задача розбивається на послідовність етапів (стадій). Дані ПРОХОДЯТЬ через ці етапи послідовно, причому кожен етап виконується окремою сутністю (робочим процесом). Кілька одиниць даних можуть одночасно перебувати на різних етапах конвеєра:

1) ***переваги:***

– ефективно для потокової обробки даних або задач, що мають природну послідовну структуру етапів;

– дозволяє спеціалізувати процеси/процесори на конкретних етапах;

– забезпечує високу пропускну здатність після заповнення конвеєра;

2) ***недоліки:***

– продуктивність обмежується найповільнішим етапом конвеєра;

– може бути складно збалансувати навантаження між етапами;

– висока латентність для першого елемента даних, що проходить весь конвеєр;

3) ***особливості реалізації в Java:***

– може бути реалізовано за допомогою черг (наприклад, BlockingQueue) між процесами, кожен з яких відповідає за свій етап;

– потужні інструменти для побудови конвеєрів обробки даних надають реактивні бібліотеки (RxJava, Project Reactor).

Вибір конкретної стратегії відображення та планування залежить від структури задачі, характеристик комунікацій, вимог до масштабованості та відмовостійкості, а також від можливостей цільової платформи та інструментів (наприклад, бібліотек Java).

Контрольні питання та завдання

Загальні поняття та умови паралелізму

1. Що розуміється під паралельним програмуванням і яка його роль у сучасних обчислювальних системах?
2. Назвіть та коротко опишіть ключові умови досягнення паралелізму в обчислювальних системах.
3. Поясніть концепцію надлишковості елементів обчислювальної мережі як умову досягнення паралелізму. Наведіть приклади.
4. Чому незалежність функціонування окремих пристроїв ЕОМ є важливою для ефективного паралелізму?
5. Як можливість дублювання обчислювальних пристроїв ЕОМ сприяє досягненню паралелізму та відмовостійкості?
6. Яку роль відіграють спеціалізовані обчислювальні пристрої (GPU, TPU, FPGA, ASIC) у досягненні паралелізму?

Шляхи досягнення паралелізму

1. Що таке декомпозиція задачі в контексті паралельного програмування?
2. Поясніть різницю між декомпозицією за даними (data parallelism) та декомпозицією за завданнями (task parallelism). Наведіть приклади для кожного підходу.
3. Які критерії ефективної декомпозиції задач ви знаєте?
4. Що таке взаємне блокування (deadlock) у паралельних програмах? Які існують методи запобігання взаємним блокуванням?
5. Поясніть поняття «змагання за доступ до ресурсів» (race conditions). Які типи конфліктів розрізняють?
6. Які існують способи мінімізації конфліктів через доступ до спільних ресурсів у Java?
7. Чому балансування навантаження є важливим для ефективності

паралельних програм? Які інструменти для цього існують в Java?

8. Що таке накладні витрати в паралельних програмах? Які існують способи їх зменшення в Java?

Рівні паралелізму в інформаційній системі

1. Назвіть та коротко опишіть основні рівні паралелізму в інформаційній системі.

2. Що таке паралелізм на рівні завдань операційної системи (Task–Level Parallelism)? Як він реалізується на одноядерних та багатоядерних системах?

3. Поясніть паралелізм на рівні програми (Program–Level Parallelism). Які основні механізми його реалізації в Java?

4. Що таке паралелізм на рівні макрооперацій (Macro–Operation Level Parallelism)? Наведіть приклади його реалізації.

5. Поясніть концепцію паралелізму на рівні машинних команд (Instruction–Level Parallelism – ILP). Які апаратні техніки використовуються для його реалізації?

6. Що таке паралелізм на рівні машинних слів і арифметичних операцій (Bit–Level Parallelism)? Як ширина машинного слова та інструкції SIMD сприяють цьому рівню паралелізму?

Режими паралельних обрахунків

1. Які основні режими паралельних обрахунків ви знаєте?

2. Опишіть багатозадачний режим виконання паралельних обрахунків.

3. Поясніть режим поділу часу. Які його основні аспекти та де він переважно використовується?

4. Що таке розподілені обчислення? Назвіть їхні основні характеристики та сфери застосування.

Парадигми паралельного програмування

1. Дайте визначення поняттю «парадигма» в контексті паралельного програмування.

2. Опишіть парадигму паралелізму даних. Назвіть її переваги та недоліки.

3. Опишіть парадигму паралелізму завдань. Назвіть її переваги та недоліки.

4. В яких випадках доцільно використовувати паралелізм даних, а в яких – паралелізм завдань?

Типи помилок програміста в паралельних системах

1. Які основні категорії помилок можуть виникати при розробці програмного забезпечення на Java?
2. Опишіть синтаксичні помилки: їхні риси, умови виникнення та способи уникання.
3. Що таке помилки часу виконання (runtime errors/exceptions)? Наведіть приклади типових винятків у Java.
4. Наведіть умови виникнення помилок часу виконання та способи їх уникання.
5. Поясніть, що таке логічні (алгоритмічні) помилки. Чому їх важко виявляти?
6. Які специфічні помилки характерні для паралельних програм?

Теоретичні моделі та аналіз

1. Як використовуються інформаційні графи для аналізу потенціалу розпаралелювання програм?
2. Поясніть суть закону Амдала. Для оцінки чого він використовується?
3. Поясніть суть закону Густавсона–Барсіса. Чим його підхід відрізняється від закону Амдала?
4. Наведіть ключові етапи системного підходу до розробки ефективних паралельних алгоритмів?

РОЗДІЛ 2
ПАРАЛЕЛЬНІ АРХІТЕКТУРИ

Ефективність паралельної програми нерозривно пов'язана з архітектурою апаратного забезпечення, на якому вона виконується. Розуміння основних типів паралельних архітектур дозволяє:

а) **обирати правильні програмні моделі та інструменти** – різні архітектури краще підходять для різних підходів до паралелізму (наприклад, спільна пам'ять для процесів Java, розподілена пам'ять для обміну повідомленнями);

б) **оптимізувати продуктивність** – знання про те, як процесори взаємодіють з пам'яттю та між собою, допомагає уникати «проблемної ділянки» (bottlenecks) і максимально використовувати ресурси системи;

в) **передбачати проблеми масштабованості** – розуміння обмежень конкретної архітектури дозволяє оцінити, наскільки добре програма буде працювати при збільшенні кількості процесорів або обсягу даних.

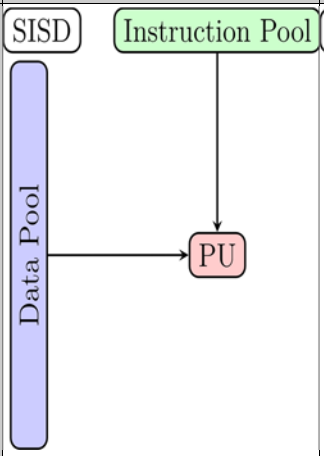
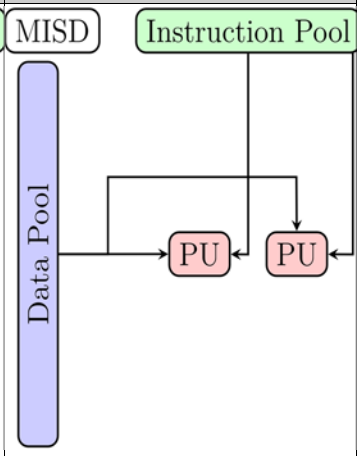
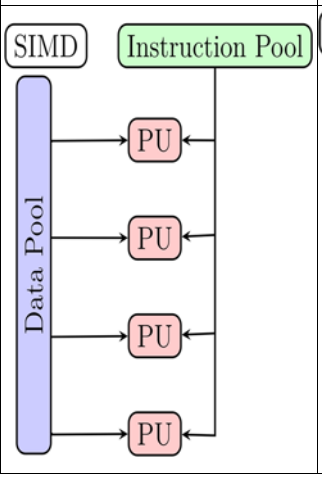
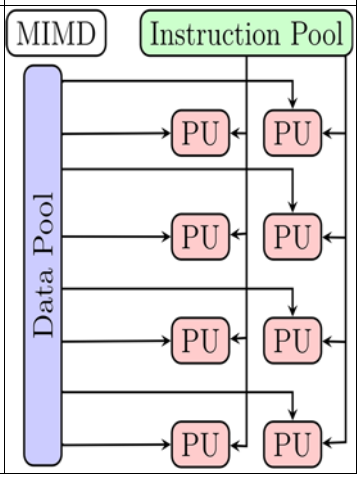
У цьому розділі ми розглянемо фундаментальні концепції та класифікації паралельних архітектур, які є основою для сучасних багатоядерних процесорів, графічних прискорювачів та кластерних систем, на яких часто виконуються Java–додатки.

2.1 Класифікація паралельних архітектур М. Флінна

Однією з найвідоміших і найпростіших класифікацій комп'ютерних архітектур є **таксономія Флінна** (Flynn's Taxonomy), запропонована Майклом Флінном[9] у 1966 році. Вона базується на кількості потоків інструкцій (Instruction Stream) та потоків даних (Data Stream), які система може обробляти одночасно (табл. 2.1).

[9] Flynn, Michael J – американський науковець.

Таблиця 2.1 – Класифікація Флінна

	Єдиний потік інструкцій *Single Instruction*	***Множинний потік інструкцій*** *Multiple Instruction*
Єдиний потік даних *Single Data*		
Множинний потік даних *Multiple Data*		

Для ефективного паралельного програмування на Java необхідно розуміти, на якій архітектурі (або комбінації архітектур) буде виконуватися програма. Класифікація Флінна є одною з фундаментальних архітектурних концепцій. Тому розглянемо її більш детально.

1) **SISD (Single Instruction, Single Data – Єдиний потік інструкцій, Єдиний потік даних):**

а) це класична послідовна архітектура фон Неймана;

б) один процесор виконує один потік інструкцій над одним потоком даних;

в) паралелізм на рівні інструкцій (ILP) може існувати всередині процесора (конвеєризація, суперскалярність), але на програмному рівні це послідовне виконання;

г) ***Приклад:*** традиційні одноядерні процесори минулого;

2) **SIMD (Single Instruction, Multiple Data – Єдиний потік інструкцій, множинний потік даних):**

а) одна інструкція застосовується одночасно до багатьох елементів даних;

б) добре підходить для завдань з високим рівнем паралелізму даних (data parallelism), таких як обробка зображень, векторні обчислення, наукові симуляції;

в) у Java пряма робота з SIMD–інструкціями обмежена, але деякі бібліотеки та JVM можуть використовувати їх «під капотом» для оптимізації (наприклад, у векторних операціях або через Project Panama / Vector API у нових версіях Java);

г) ***Приклади:*** Векторні процесори, мультимедійні розширення процесорів (MMX, SSE, AVX), графічні процесори (GPU);

3) **MISD (Multiple Instruction, Single Data – Множинний потік інструкцій, єдиний потік даних):**

а) кілька інструкцій одночасно обробляють один потік даних;

б) цей тип архітектури зустрічається дуже рідко на практиці;

в) ***Теоретичні приклади:*** системи з надлишковим виконанням для підвищення надійності (наприклад, у космічних апаратах, де кілька процесорів виконують однакові обчислення над тими ж даними для виявлення помилок);

4) **MIMD (Multiple Instruction, Multiple Data – Множинний потік інструкцій, множинний потік даних):**

а) кілька процесорів (ядер) можуть одночасно виконувати різні потоки інструкцій над різними потоками даних;

б) це найгнучкіший і найпоширеніший тип паралельної архітектури сьогодні;

в) саме на архітектури типу MIMD орієнтована більшість засобів паралельного програмування в Java (процеси Thread, ExecutorService, ForkJoinPool);

г) ***Приклади:*** багатоядерні процесори (CPU), багатопроцесорні системи (SMP), кластери, розподілені системи.

MIMD–архітектури, в свою чергу, поділяються за способом організації доступу до пам'яті.

2.2 Конвеєрна обробка (Pipelining)

Перш ніж розглядати паралелізм на рівні багатьох процесорів чи ядер, важливо зрозуміти, як сучасні процесори досягають високої продуктивності навіть при виконанні одного потоку команд. Ключовою технікою тут є конвеєризація.

2.2.1 Загальний принцип конвеєра

Ідея конвеєра (pipeline) полягає у розбитті складного процесу (наприклад, виконання однієї машинної інструкції) на послідовність простіших етапів (стадій). Кожен етап виконується спеціалізованим апаратним блоком. Замість того, щоб чекати повного завершення однієї інструкції перед початком наступної, конвеєр дозволяє різним інструкціям перебувати на різних етапах виконання одночасно.

Уявіть собі складальну лінію на заводі: кожен робітник виконує одну операцію, і багато виробів одночасно перебувають на лінії на різних стадіях готовності. Аналогічно, процесорний конвеєр обробляє потік інструкцій.

Основна мета конвеєризації – збільшити пропускну здатність (throughput), тобто кількість інструкцій, що виконуються за одиницю часу, навіть якщо час виконання однієї окремої інструкції (latency) може трохи зрости через накладні витрати на поділ етапів.

2.2.2 Конвеєр команд (Instruction Pipeline)

Це найпоширеніший тип конвеєра в сучасних CPU. Процес виконання інструкції розбивається на типові етапи, наприклад:

а) **вибірка інструкції (Instruction Fetch – IF)** – зчитування наступної інструкції з пам'яті (або кешу);

б) **декодування інструкції (Instruction Decode – ID)** – визначення типу операції та операндів;

в) **виконання (Execute – EX)** – виконання арифметико–логічної операції;

г) **доступ до пам'яті (Memory Access – MEM)** – читання або запис даних в пам'ять (якщо потрібно);

ґ) **запис результату (Write Back – WB)** – запис результату операції в регістр.

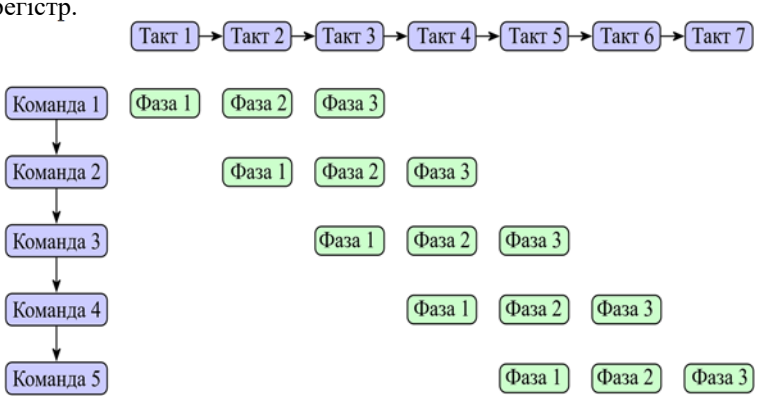


Рисунок 2.1 – Конвеєр команд

У п'ятиступеневому конвеєрі (рис. 2.1), в ідеальному випадку, кожний тактовий цикл нова інструкція може входити в конвеєр, і інша інструкція завершує своє виконання. Таким чином, хоча одна інструкція все ще проходить 5 етапів, середня швидкість виконання наближається до однієї інструкції за такт:

а) паралелізм на рівні інструкцій (англ. Instruction Level Parallelism – ILP): Конвеєр команд є основним механізмом реалізації ILP. Сучасні процесори можуть мати значно глибші конвеєри (10–30+ етапів) та використовувати додаткові техніки ILP, такі як суперскалярність (кілька конвеєрів, що працюють паралельно) та виконання поза порядком (out–of–order execution);

б) проблеми (Hazards): Робота конвеєра може порушуватися через залежності між інструкціями:

1) **структурні конфлікти (Structural Hazards)** – два етапи потребують одного ресурсу одночасно;

2) **конфлікти по даних (Data Hazards)** – інструкція потребує результат попередньої, яка ще не завершилася. Вирішується за допомогою перенаправлення даних (forwarding) або зупинки конвеєра (stalls);

3) **конфлікти по керуванню (Control Hazards)** – інструкції умовного/безумовного переходу змінюють потік виконання, роблячи завантажені в конвеєр інструкції непотрібними. Вирішується за допомогою передбачення переходів (branch prediction).

2.2.3 Конвеєр даних (Data Pipeline / Vector Pipeline)

Цей тип конвеєра застосовує послідовність операцій (етапів) до потоку даних. Кожен етап виконує певну обробку над елементом даних, після чого передає його на наступний етап (рис. 2.2).

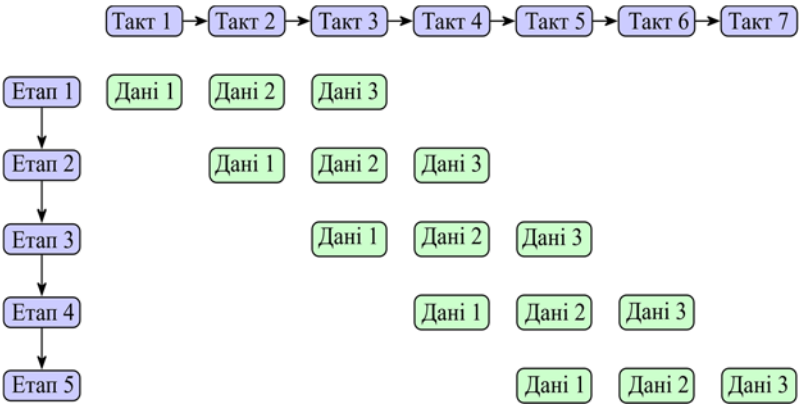


Рисунок 2.2 – Конвеєр даних

Підрахуємо коефіцієнт прискорення для конвеєрної архітектури, наведеної на рис. 2.2. Класичний процесор, побудований на архітектурі фон Неймана виконає п'ятиетапний обрахунок трьох наборів даних за $T_1 = 3 \times 5 = 15$ тактів. У випадку конвеєрної архітектури перший набір даних буде обраховано за п'ять тактів. Цей процес називається **заповненням конвеєра**, а кількість етапів обробки даних – **глибиною конвеєра**. Позначимо її літерою d. У нашому випадку $d = 5$. Після того, як буде обраховано перший набір даних, результати обрахунку кожного наступного набору будуть сходити з конвеєра на кожному наступному такті. Тобто, у випадку архітектури, наведеної на рис. 2.2, для цього знадобиться ще два такти: $T_n = 5 + 2 = 7$. Тоді коефіцієнт прискорення $K = \frac{T_1}{T_n} = \frac{15}{7} \approx 2{,}14$. Тобто, розрахунок той самої задачі за допомогою конвеєрної архітектури буде виконаний більш ніж у два рази швидше.

Узагальнюючи вищенаведене, можна вивести формулу для розрахунку коефіцієнта прискорення для конвеєру даних:

$$K = \frac{T_1}{T_n} = \frac{d \cdot n}{d + n - 1}, \tag{2.1}$$

де d – глибина конвеєра;

n – кількість наборів даних, що обробляються за один такт.

Конвеєри даних тісно пов'язані з архітектурами SIMD. Векторні інструкції (наприклад, AVX) або операції на GPU часто реалізуються за допомогою конвеєрів, де кожен етап виконує частину операції над елементами вектора або потоку даних.

Прикладами застосування систем з конвеєрною архітектурою є обробка сигналів, рендеринг графіки (де вершини та пікселі проходять через конвеєр обробки), деякі алгоритми фільтрації.

2.2.4 Значення для програміста Java

Конвеєр команд значною мірою є прозорим для Java–програміста. JVM транслює байт–код в машинні інструкції, а апаратний конвеєр процесора виконує їх максимально швидко. Розуміння його існування допомагає усвідомити, що послідовний код всередині одного процесу також виконується з високим ступенем внутрішнього паралелізму на рівні інструкцій. Хоча прямий вплив на код Java обмежений, надмірно складна логіка з частими переходами та залежностями по даних теоретично може дещо знижувати ефективність конвеєра (хоча сучасні процесори дуже добре оптимізують це).

Конвеєр даних стає більш релевантним при роботі з великими масивами даних та використанні технологій, що спираються на SIMD. Наприклад, при використанні Java Vector API (з Project Panama) або при роботі з бібліотеками для GPU–обчислень (JCUDA, TornadoVM тощо), ефективність може залежати від того, наскільки добре дані та операції над ними «вкладаються» в модель конвеєрної обробки даних, яку використовує відповідний апаратний прискорювач (CPU SIMD–юніти або GPU).

2.3 Архітектури пам'яті

Спосіб, у який процесори (ядра) отримують доступ до пам'яті, є ключовим фактором, що впливає на модель програмування та продуктивність.

2.3.1 Архітектури зі спільною пам'яттю (Shared Memory)

У системах зі спільною пам'яттю всі процесори (ядра) мають доступ до єдиного адресного простору. Це означає, що будь–який процесор може напряму читати та записувати дані в будь–яку комірку спільної пам'яті.

Модель програмування: зазвичай використовується модель

процесів (threads), де процеси спілкуються та синхронізуються через спільні змінні в пам'яті. Стандартні процеси Java (java.lang.Thread) та інструменти з java.util.concurrent (наприклад, synchronized, volatile, Lock, AtomicInteger) призначені саме для цієї моделі.

Переваги:

– відносно проста модель програмування для багатьох завдань (дані легко розділяти між процесами);

– швидкий обмін даними між процесорами (якщо дані знаходяться близько);

Недоліки:

– **проблема узгодженості кешів (Cache Coherence)** – кожне ядро зазвичай має свій кеш L1/L2. Якщо одне ядро змінює дані у своєму кеші, інші ядра повинні бачити ці зміни. Апаратні механізми (протоколи типу MESI) забезпечують узгодженість, але це може призводити до затримок;

– **пропускна здатність пам'яті** – шина пам'яті може стати «проблемною ділянкою» при великій кількості ядер, що одночасно звертаються до пам'яті;

– **масштабованість** – створення систем зі спільною пам'яттю з дуже великою кількістю процесорів є складним та дорогим.

Системи зі спільною пам'яттю поділяються на:

а) **UMA (Uniform Memory Access – Рівномірний доступ до пам'яті)** – час доступу до будь–якої комірки пам'яті однаковий для всіх процесорів. Типовий приклад – симетричні мультипроцесори (SMP) та сучасні багатоядерні процесори в настільних комп'ютерах та ноутбуках;

б) **NUMA (Non–Uniform Memory Access – Нерівномірний доступ до пам'яті)** – система складається з кількох вузлів, кожен з яких має свою «локальну» пам'ять та процесори. Процесор може отримати доступ до пам'яті будь–якого вузла, але доступ до локальної пам'яті значно швидший, ніж до «віддаленої» пам'яті іншого вузла. Використовується у потужних серверах та високопродуктивних обчислювальних системах.

Важливо для Java. На NUMA–системах розташування процесів Java та даних, які вони обробляють, може суттєво впливати на продуктивність. JVM та операційна система намагаються оптимізувати це, але іноді потрібне явне керування прив'язкою процесів до ядер (affinity).

2.3.2 Архітектури з розподіленою пам'яттю (Distributed Memory)

У системах з розподіленою пам'яттю кожен процесор має свою власну локальну пам'ять, недоступну напряму іншим процесорам. Адресні простори процесорів не перетинаються.

Модель програмування. Взаємодія між процесорами відбувається шляхом явного обміну повідомленнями (Message Passing) через мережеве з'єднання. Процесори надсилають та отримують повідомлення, що містять дані.

Стандарти. Найвідомішим стандартом для обміну повідомленнями є MPI (Message Passing Interface).

Важливо для Java. Хоча стандартна Java не має вбудованої підтримки MPI на рівні мови, існують бібліотеки та фреймворки для розподілених обчислень, які реалізують схожі концепції (наприклад, Akka, Hazelcast, Apache Spark, які дозволяють Java–програмам працювати на кластерах).

Переваги:

а) **висока масштабованість** – легко об'єднувати велику кількість вузлів (комп'ютерів) у єдину систему (кластер);

б) немає проблеми узгодженості кешів у глобальному сенсі (тільки в межах одного вузла, якщо він багатоядерний).

Недоліки:

а) **складніша модель програмування** – програміст повинен явно керувати передачею даних між процесорами;

б) **високі затримки (latency)** при комунікації через мережу порівняно з доступом до спільної пам'яті;

в) **пропускна здатність мережі** може стати обмежуючим фактором.

Приклади: Комп'ютерні кластери, масово–паралельні процесори (MPP), мережі робочих станцій (Grid).

2.3.3 Гібридні архітектури

Сучасні високопродуктивні системи часто є гібридними. Наприклад, **кластер** може складатися з багатьох **вузлів**, кожен з яких є багатоядерною (MIMD/Shared Memory, часто NUMA) системою. В межах одного вузла процеси можуть спілкуватися через спільну пам'ять, а між вузлами використовується обмін повідомленнями. Це дозволяє поєднувати переваги обох підходів.

2.4 Приклади сучасних паралельних архітектур

а) **багатоядерні процесори (Multicore CPUs):**

– найпоширеніший приклад MIMD/Shared Memory (зазвичай UMA або NUMA);

– кілька ядер на одному кристалі, часто зі спільним кешем L3 та доступом до спільної оперативної пам'яті;

– основна платформа для застосування стандартних Java–процесів та інструментів з пакету java.util.concurrent;

– ***важливі аспекти:*** узгодженість кешів, локальність даних (data locality), хибний поділ (false sharing);

б) **графічні процесори (GPUs):**

– архітектура типу SIMD (або точніше SIMT – Single Instruction, Multiple Threads);

– мають тисячі простих обчислювальних ядер, оптимізованих для масового паралелізму даних;

– мають власну високошвидкісну пам'ять (GDDR);

– програмуються за допомогою спеціальних моделей (CUDA, OpenCL);

– ***використання з Java*** можливе через спеціальні бібліотеки (наприклад, JCUDA, Aparapi, TornadoVM), які дозволяють виконувати обчислювально- інтенсивні частини Java–коду на GPU. Ключовим аспектом продуктивності є передача даних між основною пам'яттю (RAM) та пам'яттю GPU;

в) **кластери:**

– системи з розподіленою пам'яттю (MIMD/Distributed Memory), що складаються з окремих комп'ютерів (вузлів), об'єднаних мережею;

– ***платформи для Java:*** Apache Hadoop, Apache Spark, Apache Flink, Hazelcast та інші дозволяють створювати розподілені Java–додатки, що працюють на кластерах. Вони абстрагують низькорівневий обмін повідомленнями, надаючи вищорівневі API.

2.5 Вплив архітектури на програмування в Java

Розуміння архітектури безпосередньо впливає на те, як ми пишемо та оптимізуємо паралельні Java–програми:

а) **спільна пам'ять (багатоядерні CPU):**

1) **синхронізація** – необхідність використання synchronized, Lock, volatile, атомарних класів для коректного доступу до спільних даних та уникнення станів гонки за даними (race conditions);

2) **узгодженість кешів та модель пам'яті Java (JMM)** – розуміння JMM допомагає писати код, який буде коректно працювати на різних архітектурах (гарантії volatile та synchronized щодо видимості змін);

3) **локальність даних** – значно підвищити продуктивність може розміщення даних близько до ядра, яке їх обробляє (особливо на NUMA). Уникнення «хибного поділу» (false sharing), коли різні процеси модифікують незалежні дані, що випадково опинилися в одному кеш–рядку;

4) **вибір пулу процесів** – розмір пулу (ThreadPoolExecutor, ForkJoinPool) часто залежить від кількості доступних ядер;

б) **розподілена пам'ять (Кластери, Фреймворки типу Spark):**

1) **серіалізація** – дані, що передаються між вузлами, повинні бути серіалізовані. Ефективність серіалізації впливає на продуктивність;

2) **вартість комунікації** – мережева передача даних значно повільніша за доступ до локальної пам'яті. Алгоритми повинні мінімізувати обмін даними між вузлами (перемішування/shuffle у Spark є дорогою операцією);

3) **обробка відмов** – у розподілених системах вища ймовірність відмови окремих вузлів. Фреймворки надають механізми для забезпечення відмовостійкості;

в) **гетерогенні системи (CPU + GPU):**

1) **передача даних** – копіювання даних між пам'яттю CPU та GPU є основною проблемною ділянкою. Необхідно мінімізувати такі передачі та максимізувати обчислення на GPU для кожного переданого блоку даних;

2) **вибір завдань** – не всі завдання добре підходять для GPU. Найкраще GPU справляються з масовими, незалежними обчисленнями над великими масивами даних (SIMD/SIMT).

2.6 Класифікація паралельних архітектур В. Хендлера

Класифікація, розроблена Вольфгангом Хендлером (Wolfgang Hфndler) в Університеті Ерлангена–Нюрнберга в 1977 році, відома як Ерлангенська система класифікації (Erlangen Classification Scheme, ECS). На відміну від більш первісної та простішої класифікації Флінна, яка фокусується лише на потоках команд та даних, класифікація Хендлера пропонує більш детальну трирівневу модель для опису паралелізму в комп'ютерних системах. Вона враховує паралелізм на рівні керування програмою, арифметико–логічних пристроїв та обробки бітів.

2.6.1 Ерлангенська система класифікації (ECS)

Класифікація Хендлера описує комп'ютерну архітектуру за допомогою трьох пар характеристик, що відображають ступінь паралелізму та конвеєризації на трьох різних рівнях обробки:

а) **рівень пристрою керування програмою (ПКП, англ. Program Cont– rol Unit, PCU)** – характеризує кількість пристроїв керування, що працюють паралельно;

б) **рівень арифметико–логічного пристрою (АЛП, англ. Arithmetic and Logic Unit, ALU)** – характеризує кількість арифметико–логічних пристроїв, що контролюються кожним ПКП і працюють паралельно;

в) **рівень елементарної логічної схеми (ЕЛС, англ. Elementary Logic Circuit, ELC)** – характеризує кількість біт, що обробляються паралельно в одному АЛП (тобто розрядність АЛП).

Кожен рівень описується двома значеннями:

1) k – кількість паралельно працюючих блоків (ПКП);

2) k' – кількість блоків (ПКП), що використовуються в режимі конвеєра (macro–pipelining);

3) d – кількість АЛП, що контролюються кожним ПКП і працюють паралельно;

4) d' – кількість АЛП, які можуть бути конвеєризовані або працювати в режимі ланцюга (chaining) під керуванням одного ПКП;

5) w – розрядність АЛП (кількість біт, що обробляються паралельно одним АЛП);

6) w' – кількість конвеєрних стадій на рівні обробки бітів в АЛП (arithmetic pipelining).

Таким чином, повна формула класифікації Хендлера має вигляд:

$$t(\mathrm{ECS}) = (k \times k', d \times d', w \times w'),$$

або, іноді, для спрощення, якщо конвеєризація на певному рівні відсутня або не розглядається, використовують скорочену форму:

$$t(\mathrm{ECS}) = (k, d, w),$$

де k – кількість процесорних блоків керування (ПКП);

d – кількість арифметико–логічних пристроїв (АЛП) в кожному ПКП;

w – розрядність АЛП/тракту даних (кількість біт).

І для конвеєрних варіантів:

k' – глибина конвеєра на рівні ПКП (макроконвеєр);

d' – глибина конвеєра на рівні АЛП (ланцюжок АЛП);

w' – глибина конвеєра на бітовому рівні в АЛП (арифметичний конвеєр).

2.6.2 Детальний опис компонентів

2.6.2.1 Рівень ПКП (*k, k'*)

Рівень ПКП *k*: Показує, скільки незалежних програм або потоків команд може виконуватися одночасно. Наприклад, у багатопроцесорній системі *k* відповідає кількості процесорів.

Рівень ПКП *k′*: Відображає можливість конвеєрної обробки на рівні цілих програм або їх великих блоків. Це може бути реалізовано, наприклад, шляхом передачі результатів одного ПКП на вхід іншого.

2.6.2.2 Рівень АЛП (d, d')

Рівень АКП *d*: Кількість АЛП, що працюють паралельно під керуванням одного ПКП. Це типово для векторних процесорів або процесорів з кількома виконавчими пристроями (наприклад, суперскалярні архітектури, де $d > 1$).

Рівень АКП *d′*: Описує конвеєризацію на рівні АЛП. Наприклад, якщо один АЛП виконує множення, а інший додавання, і вони можуть працювати послідовно над потоком даних (ланцюжок), це враховується у *d′*.

2.6.2.3 Рівень ЕЛС (w, w')

Рівень ЕЛС *w*: Розрядність тракту даних АЛП. Для 64–розрядного процесора $w = 64$.

Рівень ЕЛС *w′*: Глибина конвеєра всередині самого АЛП. Наприклад, операція множення з плаваючою комою може бути розбита на декілька стадій (нормалізація, множення мантис, додавання експонент, округлення), і *w′* показує кількість таких стадій.

2.6.3 Приклади класифікації архітектур за Хендлером

Розглянемо, як деякі відомі архітектурні класи (включаючи класифікацію Флінна) можуть бути представлені за допомогою ECS:

а) **SISD (Single Instruction, Single Data stream)** – традиційний послідовний комп'ютер.

$$t(\mathrm{SISD}) = (1 \times 1, 1 \times 1, w \times 1 \quad w'),$$

або спрощено: (1,1,*w*). Наприклад, для старого 8–бітного мікропроцесора без конвеєризації АЛП: (1,1,8). Для сучасного процесора з 64–бітною архітектурою та конвеєризованим АЛП: ($1 \times 1{,}1 \times 1{,}64 \times w'$);

б) **SIMD (Single Instruction, Multiple Data stream)** – векторні процесори або масивно–паралельні процесори з одним пристроєм

керування.

$$t(\text{SIMD}) = (1 \times 1, d \times 1 \;\; d', w \times 1 \;\; w'),$$

або спрощено: (1,*d*,*w*). Тут *d* > 1. Приклад: процесор ILLIAC IV мав *k* = 1,*d* = 64,*w* = 64. Таким чином, (1,64,64). Сучасні GPU також мають характеристики SIMD (або точніше SIMT – Single Instruction, Multiple Threads), де один блок керування керує багатьма обчислювальними ядрами;

в) **MISD (Multiple Instruction, Single Data stream)** – цей клас є досить рідкісним. Теоретично, це може бути система, де декілька інструкцій виконуються над одним потоком даних, можливо, в конвеєрному режимі.

$$t(\text{MISD}) = (k \times k', 1 \times 1, w \times 1 \;\; w'),$$

або спрощено: (*k*,1,*w*), де *k* > 1. Прикладом може бути конвеєрний процесор, де різні стадії конвеєра виконують різні інструкції над тим самим потоком даних. Наприклад, якщо є *k*′ стадій обробки даних різними інструкціями: (1 × *k*′,1 × 1,*w* × *w*′);

г) **MIMD (Multiple Instruction, Multiple Data stream)** – багатопроцесорні системи, де кожен процесор виконує свою програму над своїми даними:

$$t(\text{MIMD}) = (k \times 1 \;\; k', d \times 1 \;\; d', w \times 1 \;\; w'),$$

або спрощено: (*k*,*d*,*w*), де *k* > 1. Якщо *d* = 1, то це класичний мультипроцесор, де кожен процесор є SISD–типу. Наприклад, двоядерний процесор: (2,1,64). Кластер з 16 вузлів, кожен з яких є 64–бітним процесором: (16,1,64). Якщо кожен процесор в MIMD системі є векторним (SIMD–типу), то *d* > 1;

г) **конвеєрний процесор (Pipelined Processor)** – процесор з конвеєризацією на рівні АЛП:

$$t(\text{Pipeline ALU}) = (1 \times 1, 1 \times d', w \times w'),$$

де *d*′ – кількість функціональних блоків в ланцюжку;

w′ – глибина конвеєра всередині одного АЛП.

Наприклад, Cray–1 мав (1,1 × *d*′, 64 × *w*′), де *d*′ відображало можливість ланцюгової обробки (chaining) між різними функціональними пристроями;

д) **асоціативний процесор (Associative Processor)** має *d* АЛП, що працюють паралельно над даними, які вибираються за вмістом:

$$t(\text{Assoc. Proc.}) = (1, d, 1),$$

де *w* = 1, оскільки операції часто виконуються побітово або над невеликими полями.

2.6.4 Приклад детального розбору: Intel Pentium 4

Розглянемо гіпотетичний приклад для процесора типу Pentium 4 з технологією Hyper–Threading:

1) $k = 1$ – фізично один процесорний кристал;

2) $k' = 2$ – завдяки Hyper–Threading, операційна система «бачить» два логічних процесори. Це можна інтерпретувати як форму макроконвеєризації або швидкого перемикання контексту на рівні ПКП;

3) $d \approx 2…3$ – Pentium 4 мав декілька виконавчих блоків (для цілочисельних операцій, операцій з плаваючою комою, MMX/SSE інструкцій), які могли працювати паралельно (суперскалярність);

4) d' – глибина конвеєра для послідовності операцій (наприклад, FPU мав власний конвеєр);

5) $w = 32$ (для основних регістрів загального призначення) або $w = 64/128$ (для SSE/SSE2 регістрів);

6) w' – дуже глибокий конвеєр, наприклад, NetBurst архітектура мала конвеєр до 20–31 стадій для цілочисельних операцій.

Таким чином, для Pentium 4 з Hyper–Threading, спрощена класифікація могла б бути:

$$t = (1 \times 2, \approx 3 \times d', word\_size \times w'),$$

де *word_size* залежить від типу операції (32, 64, 128), а d' та w' відображають глибину відповідних конвеєрів.

2.6.5 Переваги та недоліки класифікації Хендлера

2.6.5.1 Переваги

а) **деталізація** – надає значно більше деталей про архітектуру, ніж класифікація Флінна, враховуючи паралелізм на трьох різних рівнях та конвеєризацію;

б) **гнучкість** – дозволяє описувати широкий спектр архітектур, включаючи складні гібридні системи;

в) **структурованість** – трирівнева модель (ПКП, АЛП, ЕЛС) є логічною та добре відображає ієрархію обробки в комп'ютері;

г) **врахування конвеєризації** – явно включає параметри (k', d', w') для опису конвеєрної обробки на кожному рівні, що є важливим аспектом сучасних процесорів.

2.6.5.2 Недоліки

1) **Складність** – класифікація може бути складною для розуміння та застосування, особливо через велику кількість параметрів;

2) **неоднозначність інтерпретації** – іноді буває складно однозначно визначити значення параметрів k',d',w' для конкретних архітектур, оскільки вони можуть залежати від режиму роботи або конкретної конфігурації;

3) **обмеженість для деяких сучасних архітектур** – хоча класифікація Хендлера є більш детальною, деякі аспекти сучасних архітектур (наприклад, гетерогенні системи з різними типами ядер, складні ієрархії кеш–пам'яті, мережеві аспекти в розподілених системах) можуть не повністю охоплюватися;

4) **менш поширена** – порівняно з класифікацією Флінна, класифікація Хендлера є менш відомою та рідше використовується в загальноосвітній літературі, хоча вона є важливою для спеціалістів з архітектури комп'ютерів.

2.6.6 Порівняння з класифікацією Флінна

Класифікація Флінна (1966 р., розширена 1972 р.) базується на поняттях потоку команд (Instruction Stream) та потоку даних (Data Stream), виділяючи чотири основні класи: SISD, SIMD, MISD, MIMD.

Таблиця 2.2 – Порівняння класифікацій Флінна та Хендлера

Аспект	***Класифікація Флінна***	***Класифікація Хендлера (ECS)***
Рівні деталізації	Один рівень (потоки команд/даних)	Три рівні (ПКП, АЛП, ЕЛС)
Паралелізм	Кількість потоків команд і даних	Кількість паралельних блоків на кожному рівні
Конвеєризація	Не враховується явно (хоча MISD може інтерпретуватися як форма конвеєра)	Явно враховується на кожному з трьох рівнів (k',d',w')
Розрядність даних	Не враховується явно	Враховується (w)
Складність	Проста, легко запам'ятовується	Більш складна, більше параметрів
Застосування	Широко використовується для загального опису	Більш специфічна, для детального аналізу архітектур

Класифікація Хендлера може розглядатися як розширення або уточнення класифікації Флінна. Вона забезпечує трирівневий опис, що враховує паралелізм та конвеєризацію на рівнях пристроїв керування програмою, арифметико–логічних пристроїв та бітової обробки. Хоча вона є складнішою за класифікацію Флінна, її детальність дозволяє краще зрозуміти внутрішню структуру та потенціал продуктивності різноманітних паралельних систем. Наприклад, категорія SIMD за Флінном може бути деталізована за Хендлером як (1,*d*,*w*), де *d* показує кількість паралельних АЛП, а *w* – їх розрядність. Аналогічно, MIMD може бути (*k*,1,*w*) для багатопроцесорної системи з послідовними процесорами, або (*k*,*d*,*w*) для системи, де кожен з *k* процесорів сам є SIMD–подібним (*d* > 1). Для більшої наочності головні аспекти обох класифікацій зведені у порівняльну табл. 2.2.

2.7 Обчислювальні кластери

Сучасні наукові та інженерні задачі часто вимагають величезних обчислювальних ресурсів, які перевищують можливості одного, навіть найпотужнішого, комп'ютера. Для вирішення таких задач використовують *обчислювальні кластери*.

Обчислювальний кластер (англ. *compute cluster*) – це сукупність окремих комп'ютерів (які називаються **вузлами** або **нодами**), з'єднаних між собою високошвидкісною мережею і працюючих разом як єдина обчислювальна система. Основна ідея полягає в тому, щоб розподілити велику задачу на менші підзадачі, які можуть виконуватися паралельно на різних вузлах кластера, а потім об'єднати результати.

Використання кластерів дозволяє досягти:

– **високої продуктивності (High Performance Computing, HPC)** – сумарна обчислювальна потужність кластера може значно перевищувати потужність окремих комп'ютерів;

– **високої доступності (High Availability, HA)** – у разі відмови одного вузла, інші вузли можуть продовжити роботу, забезпечуючи безперебійність сервісу (хоча це більше стосується кластерів доступності, а не чисто обчислювальних);

– **масштабованості (Scalability)** – можливість нарощувати обчислювальну потужність шляхом додавання нових вузлів до кластера;

– **економічної ефективності** – часто побудова кластера з багатьох стандартних комп'ютерів є економічно вигіднішою, ніж придбання

одного суперкомп'ютера еквівалентної потужності.

У контексті паралельного програмування, кластери є основною платформою для виконання застосунків, що використовують моделі розподіленої пам'яті, такі як MPI, або розподілені фреймворки, такі як Apache Spark.

2.7.1 Архітектура обчислювального кластера

Типовий обчислювальний кластер складається з наступних основних компонентів:

2.7.1.1 Вузли (Nodes)

Це окремі комп'ютери, що складають кластер. Зазвичай виділяють кілька типів вузлів:

1) **обчислювальні вузли (Compute Nodes)** – основна частина кластера, призначена безпосередньо для виконання паралельних задач користувачів. Зазвичай вони не мають власних дисків великої ємності (використовують мережеве сховище) і оптимізовані для обчислень;

2) **головний вузол (Head Node / Master Node / Login Node)** – точка входу для користувачів. На ньому відбувається компіляція програм, постановка задач у чергу, моніторинг стану кластера. Зазвичай має доступ до всіх необхідних ресурсів кластера;

3) **вузли зберігання даних (Storage Nodes)** – комп'ютери, що надають доступ до спільного дискового простору для всіх вузлів кластера (наприклад, через NFS[10], Lustre[11], CephFS[12]);

4) **сервісні вузли (Service Nodes)** можуть виконувати специфічні функції, наприклад, моніторинг, керування живленням тощо.

2.7.1.2 Мережеве з'єднання (Interconnect)

Критично важливий компонент, що забезпечує комунікацію між вузлами кластера. Від його пропускної здатності та затримок (latency) значною мірою залежить ефективність паралельних програм, особливо тих, що потребують інтенсивного обміну даними. Поширені технології мережевих з'єднань:

[10] Мережева файлова система (NFS) – це протокол розподіленої файлової системи, який дозволяє користувачам у мережі отримувати доступ до файлів і каталогів на віддалених серверах так, ніби вони локальні.

[11] Lustre – це паралельна розподілена файлова система з відкритим кодом, яка в основному використовується для високопродуктивних обчислень та наукових застосувань.

[12] CephFS – гнучке та легкомасштабоване петабайтне сховище файлів з відкритим кодом.

1) **gigabit Ethernet / 10Gb Ethernet** – стандартна та відносно недорога технологія, підходить для задач з не дуже інтенсивним обміном даними;

2) **InfiniBand** – високошвидкісна мережа з низькими затримками, спеціально розроблена для HPC. Забезпечує значно кращу продуктивність для багатьох паралельних застосунків;

3) **omni–Path** – конкурент InfiniBand від Intel (його розвиток на даний час припинено).

2.7.1.3 Система зберігання даних (Storage System)

Оскільки обчислювальні вузли зазвичай не мають власних великих дисків, необхідна спільна система зберігання даних, доступна з усіх вузлів. Це дозволяє всім процесам паралельної задачі працювати з одними й тими ж вхідними та вихідними файлами. Найбільш типові рішення є наступними:

а) **Network File System (NFS)** – простий та поширений протокол для надання спільного доступу до файлів по мережі;

б) **паралельні файлові системи (Lustre[11], GPFS/Spectrum Scale, CephFS[12])** – більш складні та продуктивні системи, розроблені спеціально для HPC, здатні забезпечити високу пропускну здатність при одночасному доступі з багатьох вузлів.

2.7.1.4 Програмне забезпечення (Software Stack)

Для функціонування кластера як єдиної системи необхідний набір програмного забезпечення:

1) **операційна система** – зазвичай на всіх вузлах встановлюється однакова ОС, найчастіше один із дистрибутивів Linux (наприклад, CentOS, Ubuntu Server, Scientific Linux);

2) **кластерне проміжне ПЗ (Cluster Middleware)** – засоби для керування кластером, моніторингу вузлів, розгортання ПЗ;

3) **система керування ресурсами та чергами завдань (Resource Manager / Job Scheduler)** – наприклад, Slurm, PBS/OpenPBS, Torque, LSF. Це ПЗ розподіляє ресурси кластера (процесори, пам'ять) між задачами користувачів, ставить задачі в чергу та запускає їх на виконання на вільних вузлах;

4) **бібліотеки паралельного програмування** – реалізації MPI (MPICH, OpenMPI), бібліотеки для роботи з розподіленими даними тощо;

5) **компілятори та інструменти розробки** – компілятори C/C++/Fortran (GCC, Intel Compilers), компілятор Java (JDK), відладчики, профілювальники.

2.7.2 Типи кластерів

Хоча основний фокус цього курсу на обчислювальних кластерах, варто згадати й інші типи:

1) **кластери високої продуктивності (High–Performance Computing, HPC).** Основне призначення – максимальна швидкість обчислень для складних наукових та інженерних задач. Саме цей тип є предметом нашого розгляду;

2) **кластери високої доступності (High–Availability, HA)** – призначені для забезпечення безперебійної роботи критично важливих сервісів (наприклад, вебсерверів, баз даних). У разі відмови одного вузла, його функції перебирає інший;

3) **кластери для балансування навантаження (Load Balancing)** – розподіляють вхідні запити (наприклад, до вебсайту) між кількома вузлами для запобігання перевантаженню одного сервера та покращення часу відгуку.

Часто кластери можуть поєднувати риси різних типів.

2.7.3 Моделі програмування для кластерів

Для написання програм, що ефективно виконуються на кластерах, використовуються спеціальні моделі та технології паралельного програмування.

2.7.3.1 Message Passing Interface (MPI)

Це стандарт *де–факто* для програмування на кластерах з розподіленою пам'яттю. MPI – це специфікація бібліотеки передачі повідомлень, яка дозволяє процесам, що виконуються на різних вузлах кластера, обмінюватися даними. Програміст явно керує відправленням та отриманням повідомлень між процесами. Існують реалізації MPI для C, C++, Fortran, а також для Java (про це нижче).

2.7.3.2 Фреймворки для великих даних (MapReduce, Spark)

Для обробки дуже великих обсягів даних були розроблені спеціалізовані фреймворки, такі як Apache Hadoop (з його моделлю MapReduce) та Apache Spark. Вони працюють поверх кластера і надають програмісту більш високорівневий інтерфейс для розподіленої обробки даних, приховуючи багато деталей низькорівневої комунікації та відмовостійкості. Java є однією з основних мов програмування для цих фреймворків.

2.7.3.3 Паралельне програмування на Java в кластерному середовищі

Хоча Java традиційно більше асоціюється з багатопотоковістю на одній машині (спільна пам'ять), існує кілька способів використання Java для програмування на обчислювальних кластерах (розподілена пам'ять):

а) **використання MPI з Java.** Існують бібліотеки, що реалізують прив'язку (binding) стандарту MPI до мови Java. Найвідоміша – **MPJ Express**. Вона дозволяє писати MPI–програми майже так само, як на C чи C++, використовуючи об'єктно–орієнтовані можливості Java. Програми на MPJ Express компілюються як звичайні Java–програми та запускаються за допомогою спеціальних скриптів MPJ Express, які взаємодіють із системою запуску процесів кластера (наприклад, через Hydra або спільно зі Slurm/PBS).

б) **використання розподілених фреймворків:**

1) **Apache Spark** – має потужний Java API (а також Scala, Python, R) для розподіленої обробки даних та машинного навчання на кластерах. Spark бере на себе керування розподілом задач та даних по вузлах кластера;

2) **Hazelcast, Apache Ignite, Infinispan** – це розподілені платформи даних в пам'яті (*In–Memory Data Grids*), які надають розподілені структури даних (Map, Queue, Set тощо) та обчислювальні можливості (наприклад, виконання коду на вузлах, де зберігаються дані). Вони можуть бути використані для побудови розподілених Java–застосунків;

3) **Akka Clusters** – фреймворк, що реалізує модель акторів для побудови розподілених, стійких до відмов систем на Java та Scala. Дозволяє створювати кластери з акторів, що можуть взаємодіяти через мережу;

в) **пряме використання мережевих сокетів.** Можна реалізувати взаємодію між процесами на різних вузлах за допомогою стандартних засобів Java для роботи з мережею (java.net.Socket, java.nio). Це найбільш низькорівневий підхід, що вимагає від програміста самостійної реалізації протоколів взаємодії, обробки помилок, синхронізації тощо.

Вибір конкретного підходу залежить від типу задачі, вимог до продуктивності, наявних інструментів та досвіду розробників.

2.7.4 Переваги та недоліки використання кластерів

Переваги:

– висока сукупна продуктивність;

– можливість вирішення задач, непосильних для одного комп'ютера;

– масштабованість (можливість нарощування потужності);

– потенційно краще співвідношення ціна/продуктивність порівняно з суперкомп'ютерами;

– гнучкість у конфігурації.

Недоліки:

– складність адміністрування та налаштування;

– складність розробки та відлагодження паралельних програм для розподіленої пам'яті;

– накладні витрати на комунікацію між вузлами можуть обмежувати ефективність;

– необхідність спеціального ПЗ (системи керування ресурсами, бібліотеки MPI тощо);

– енергоспоживання та охолодження можуть бути значними.

Контрольні питання та завдання

Загальні питання

1. Чому розуміння паралельних архітектур є важливим для ефективності паралельної програми?
2. Які фундаментальні концепції та класифікації паралельних архітектур розглядаються в цьому розділі?

Класифікація паралельних архітектур М. Флінна

1. На чому базується таксономія Флінна для класифікації комп'ютерних архітектур?
2. Опишіть архітектуру SISD (Single Instruction, Single Data). Наведіть приклад.
3. Що таке паралелізм на рівні інструкцій (ILP) в контексті SISD архітектури?
4. Опишіть архітектуру SIMD (Single Instruction, Multiple Data). Для яких завдань вона добре підходить?
5. Як у Java реалізується робота з SIMD–інструкціями? Наведіть приклади.

6. Опишіть архітектуру MISD (Multiple Instruction, Single Data). Чому вона рідко зустрічається на практиці? Наведіть теоретичний приклад.

7. Опишіть архітектуру MIMD (Multiple Instruction, Multiple Data). Чому вона є найпоширенішою сьогодні?

8. Які засоби паралельного програмування в Java орієнтовані на архітектури типу MIMD?

9. Наведіть приклади MIMD–архітектур.

10. За яким принципом поділяються MIMD–архітектури?

11. Зобразіть та поясніть схематично класифікацію Флінна.

Конвеєрна обробка (Pipelining)

1. У чому полягає ідея конвеєра (pipeline) в обробці команд?

2. Яка основна мета конвеєризації?

3. Назвіть та опишіть типові етапи конвеєра команд (Instruction Pipeline).

4. Поясніть, як працює п'ятиступеневий конвеєр команд і як він досягає середньої швидкості виконання однієї інструкції за такт.

5. Що таке паралелізм на рівні інструкцій (ILP) у контексті конвеєра команд? Які додаткові техніки ILP можуть використовувати сучасні процесори?

6. Назвіть та опишіть проблеми (hazards), що можуть порушувати роботу конвеєра. Як вони вирішуються?

7. Поясніть принцип роботи конвеєра даних (Data Pipeline / Vector Pipeline).

8. Наведіть приклад розрахунку коефіцієнта прискорення для конвеєрної архітектури з глибиною конвеєра $d = 5$ та $n = 3$ наборами даних.

9. Напишіть та поясніть узагальнену формулу для обрахунку коефіцієнта прискорення для конвеєра даних.

10. З якими архітектурами тісно пов'язані конвеєри даних? Наведіть приклади їх застосування.

11. Яке значення має конвеєр команд для Java–програміста? Чи може програміст прямо впливати на його ефективність?

12. Коли конвеєр даних стає релевантним для Java–програміста? Наведіть приклади.

Архітектури пам'яті

1. Яким чином спосіб доступу процесорів до пам'яті впливає на модель програмування та продуктивність?
2. Опишіть архітектури зі спільною пам'яттю (Shared Memory). Яка модель програмування зазвичай використовується?
3. Які інструменти Java призначені для моделі зі спільною пам'яттю?
4. Назвіть переваги архітектур зі спільною пам'яттю.
5. Назвіть недоліки архітектур зі спільною пам'яттю.
6. Поясніть різницю між UMA (Uniform Memory Access) та NUMA (Non–Uniform Memory Access). Наведіть приклади.
7. Чому NUMA–системи є важливими для Java–програмістів і як можна оптимізувати продуктивність на таких системах?
8. Опишіть архітектури з розподіленою пам'яттю (Distributed Memory). Яка модель програмування використовується?
9. Назвіть найвідоміший стандарт для обміну повідомленнями в системах з розподіленою пам'яттю?
10. Як Java–програми можуть працювати на системах з розподіленою пам'яттю? Наведіть приклади бібліотек та фреймворків.
11. Назвіть переваги архітектур з розподіленою пам'яттю.
12. Назвіть недоліки архітектур з розподіленою пам'яттю.
13. Наведіть приклади систем з розподіленою пам'яттю.
14. Що таке гібридні архітектури пам'яті? Наведіть приклад.

Приклади сучасних паралельних архітектур

1. Опишіть багатоядерні процесори (Multicore CPUs) як приклад паралельної архітектури. Які аспекти є важливими при програмуванні для них?
2. Опишіть графічні процесори (GPUs) як приклад паралельної архітектури (SIMD/SIMT). Як можливе їх використання з Java?
3. Який ключовий аспект продуктивності при використанні GPU з Java?
4. Опишіть кластери як приклад паралельної архітектури. Які платформи для Java дозволяють створювати розподілені Java–застосунки для кластерів?

Вплив архітектури на програмування в Java

1. Як архітектура зі спільною пам'яттю (багатоядерні CPU) впливає на синхронізацію, узгодженість кешів та модель пам'яті Java (JMM)?

2. Поясніть важливість локальності даних та уникнення хибного розділення (false sharing) на багатоядерних CPU.

3. Як кількість доступних ядер впливає на вибір розміру пулу процесів в Java?

4. Які аспекти важливі при програмуванні для архітектур з розподіленою пам'яттю (кластери, фреймворки типу Spark) в Java?

5. Які основні проблеми виникають при програмуванні для гетерогенних систем (CPU + GPU) на Java?

Класифікація паралельних архітектур В. Хендлера

1. Як називається класифікація, розроблена Вольфгангом Хендлером, і чим вона відрізняється від класифікації Флінна?

2. На яких трьох рівнях обробки класифікація Хендлера (ECS) враховує паралелізм?

3. Якими трьома парами характеристик описується комп'ютерна архітектура в ECS? Назвіть кожну характеристику та її значення.

4. Напишіть повну та спрощену формулу класифікації Хендлера.

5. Детально опишіть компоненти рівня ПКІ (k, k').

6. Детально опишіть компоненти рівня АЛП (d, d').

7. Детально опишіть компоненти рівня ЕЛС (w, w').

8. Наведіть приклади класифікації архітектур SISD, SIMD, MISD, MIMD за Хендлером (спрощена та повна форма, де доречно).

9. Як можна класифікувати конвеєрний процесор (Pipelined Processor) та асоціативний процесор (Associative Processor) за Хендлером?

10. Наведіть гіпотетичний приклад детального розбору класифікації процесора Intel Pentium 4 з технологією Hyper–Threading за Хендлером.

11. Назвіть переваги класифікації Хендлера.

12. Назвіть недоліки класифікації Хендлера.

13. Порівняйте класифікацію Флінна та Хендлера за основними аспектами (рівні деталізації, паралелізм, конвеєризація, розрядність даних, складність, застосування).

Обчислювальні кластери

1. Дайте означення обчислювального кластера. Яка основна ідея їх використання?

2. Які переваги дає використання обчислювальних кластерів?

3. У якому контексті паралельного програмування кластери є основною платформою?

РОЗДІЛ 3
ОСНОВИ БАГАТОПРОЦЕСНОСТІ В JAVA

Багатопроцесність (англ. **Multithreading**) – це технологія виконання кількох процесів в рамках одного програмного застосунку. Кожен процес виконується незалежно, але використовує спільні ресурси програми, такі як пам'ять і файлові дескриптори. У Java багатопроцесність є вбудованою властивістю мови, що дозволяє створювати багатопроцесні програми. Розуміння основ багатопроцесності в Java дозволяє створювати швидкий та ефективний програмний код. Проте використання процесів вимагає уважного підходу для уникнення потенційних проблем, таких як умови змагання за спільні ресурси чи взаємне блокування.

Використання багатопроцесності має наступні переваги:

– **покращена швидкість реагування на події (Improved Responsiveness).** Багатопроцесні програми можуть продовжувати виконувати певні завдання, навіть якщо одне з них очікує завершення операції вводу/виводу;

– **підвищена продуктивність обчислень (Enhanced Performance).** Завдяки паралельному виконанню обчислювальних задач програма може працювати ефективніше на багатоядерних процесорах;

– **спільне використання ресурсів обчислювальної системи (Resource Sharing).** Процеси можуть обмінюватися інформацією, використовуючи спільну пам'ять, що зменшує накладні витрати на створення окремих процесів;

– **паралелізм виконання обчислень (Concurrency).** Багатопроцесність дозволяє одночасно виконувати кілька незалежних завдань, що покращує загальну продуктивність.

Введемо головні поняття, багатопроцесності:

– **програмний застосунок (Program).** Одиниця виконання програми, яка має власний простір пам'яті. У системі кожен програмний застосунок виконується незалежно;

– **процес (Thread).** Легковагова одиниця виконання, що є частиною програмного застосунку і розділяє його ресурси. У Java процеси створюються за допомогою класу Thread або інтерфейсу Runnable;

– **задача (Task).** Послідовність інструкцій, яку виконує процес під час своєї активності. У Java важливо розрізняти процес, як виконавця, та задачу, як послідовність дій, яка цим процесом виконується;

– **паралельність (Concurrency).** Здатність виконувати кілька завдань одночасно шляхом переключення між ними. У Java це реалізується через API для багатопроцесності;

– **паралелізм (Parallelism).** Реальне одночасне виконання кількох завдань на різних ядрах процесора. У Java це досягається через Fork/Join Framework або Parallel Streams;

– **умови змагання за доступ до спільних даних (Race Condition).** Ситуація, коли кілька процесів одночасно намагаються змінити спільний ресурс, що призводить до некоректного результату;

– **синхронізація (Synchronization).** Механізм, який забезпечує послідовний доступ до спільних ресурсів, наприклад, через ключове слово synchronized у Java;

– **взаємне блокування (Deadlock).** Ситуація, коли два або більше процесів чекають один на одного для звільнення ресурсу, через що вони залишаються заблокованими.

3.1 Основні стани життєвого циклу процесу

Головною умовою ефективного багатопроцесного програмування є глибоке розуміння **життєвого циклу** процесу в Java.

Життєвий цикл процесу в Java визначається кількома станами:

а) **Новий (New).** Процес створюється, але ще не запускається. Це відбувається після створення об’єкта класу Thread;

б) **Готовий (Runnable).** Процес готовий до виконання і чекає на виділення процесорного часу;

в) **Виконання (Running).** Процес виконується на процесорі;

г) **Очікування (Waiting).** Процес чекає на сигнал або іншу подію для продовження виконання;

ґ) **Часове очікування (Timed Waiting).** Процес чекає певний проміжок часу;

д) **Завершений (Terminated).** Процес закінчив своє виконання.

На рис. 3.1 наведена діаграма життєвого циклу процесу в Java.

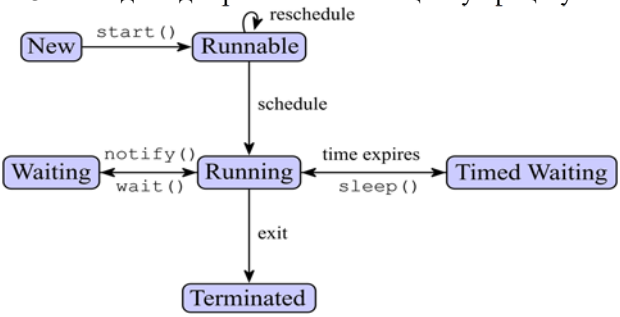


Рисунок 3.1 – Життєвий цикл процесу в Java

3.2 Створення та запуск процесів в Java

У Java існує кілька способів створення і запуску процесів, кожен з яких має свої переваги і недоліки. Основні підходи включають використання класу Thread або реалізацію інтерфейсу Runnable. Вибір залежить від конкретного сценарію використання та вимог до програми.

3.2.1 Створення процесів за допомогою класу Thread

Клас Thread є базовим механізмом для створення процесів у Java. Для створення нового процесу можна створити клас, який наслідує Thread, та перевизначити метод run().

3.2.1.1 Приклад реалізації

У лістингу 3.1 наведено приклад створення процесу MyThread оператором new. Клас MyThread є нащадком батьківського класу Thread. Задача, що виконується процесом MyThread описана у методі run().

```
class MyThread extends Thread {
@Override
public void run() {
for (int i = 0; i < 5; i++) {
System.out.println("Thread: " + i);
try {
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}

public class Main {
public static void main(String[] args) {
MyThread thread = new MyThread();
thread.start();
}
}
```

Лістинг 3.1 – Наслідування класу Thread

У методі main() створюється примірник thread класу MyThread. Під час створення примірник thread отримує усі необхідні ресурси. Після створення примірника thread, він запускається на виконання методом start().

3.2.1.2 Переваги та недоліки

Переваги:

– проста реалізація для невеликих задач;

– можливість використовувати методи класу Thread безпосередньо.

Недоліки:

– обмеження через відсутність множинного наслідування в Java: клас може наслідувати лише один клас;

– менш гнучкий підхід порівняно з використанням Runnable.

3.2.2 Створення процесів за допомогою інтерфейсу Runnable

Інтерфейс Runnable дозволяє визначити задачу, яку потім виконує процес. Це досягається шляхом імплементації методу run().

3.2.2.1 Приклад реалізації

У лістингу 3.2 наведено приклад створення задачі MyRunnable оператором new. Клас MyRunnable є імплементацією батьківського інтерфейсу Runnable. Цей інтерфейс вимагає опису дій, які будуть

виконані процесом під час виконання задачі. Такий опис здійснюється у методі run().

```
class MyRunnable implements Runnable {
@Override
public void run() {
for (int i = 0; i < 5; i++) {
System.out.println("Runnable: " + i);
try {
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}

public class Main {
public static void main(String[] args) {
Thread thread = new Thread(new MyRunnable());
thread.start();
}
}
```

Лістинг 3.2 – Імплементація інтерфейсу Runnable

У методі main() створюється примірник thread класу Thread. Під час створення примірник процесу thread отримує вже створену задачу MyRunnable. Після створення примірника thread, він запускається на виконання методом start().

3.2.2.2 Переваги та недоліки

Переваги:

– дозволяє створювати більш гнучкі програми, оскільки клас задачі може наслідувати інші класи;

– більш придатний для багаторазового використання коду.

Недоліки:

– потрібно явно створювати об'єкт Thread для запуску задачі.

3.2.3 Порівняння способів створення процесів

Наслідування Thread дозволяє дещо пришвидшити розробку коду завдяки більш тісної інтеграції процесу та виконуваної ним задачі.

Імплементація Runnable дозволяє відділити код задачі від процесу, що її виконує (рис. 3.2). Це може додати більшої гнучкості програмному коду під час розробки та покращити його супроводження у подальшому.

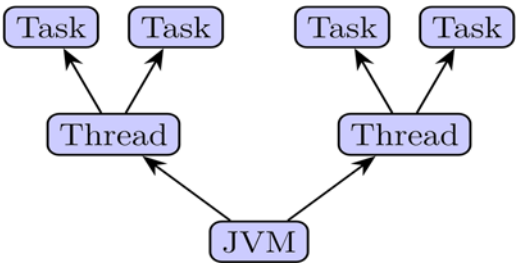


Рисунок 3.2 – Рівні паралелізму у Java з точки зору прикладного програміста

Для більшої наочності результати порівняння обох способів створення процесів наведено у табл. 3.1.

Таблиця 3.1 – Порівняння способів створення процесів

Критерій	***Наслідування Thread***	***Імплементація Runnable***
Наслідування	Дозволяє наслідувати	Можна наслідувати
	лише один клас	інші класи
Гнучкість	Менш гнучкий підхід	Більш гнучкий підхід
Повторне використання коду	Обмежене	Більш зручне

Для більшої гнучкості та повторного використання коду рекомендується використовувати Runnable, тоді як Thread може бути корисним для швидкої реалізації простих задач.

3.3 Методи управління процесами

Клас Thread надає низку методів для управління процесами. Найважливіші з них є наступними:

1) start() – переводить процес зі стану *New* у *Runnable*. Тобто запускає процес;

2) run() – містить код, який виконується процесом. Послідовність інструкцій задачі;

3) sleep(milliseconds) – переводить процес у стан очікування (*Timed Waiting*) на заданий час;

4) join() – змушує текучий процес чекати завершення виконання дочірнього процесу;

5) interrupt() – переривает виконання процесу;

6) isAlive() – перевіряє, чи знаходиться процес у стані виконання.

3.3.1 Приклад управління процесами

Лістинг 3.3 демонструє використання методів sleep() та join() для синхронізації процесів.

```
class MyRunnable implements Runnable {
@Override
public void run() {
for (int i = 0; i < 5; i++) {
System.out.println(Thread.currentThread().getName() + " – " + i);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
System.out.println("Thread Interrupted");
}
}
}
}

public class Main {
public static void main(String[] args) {
Thread thread1 = new Thread(new MyRunnable(), "Thread 1");
Thread thread2 = new Thread(new MyRunnable(), "Thread 2");
thread1.start();
thread2.start();
try {
thread1.join();
thread2.join();
} catch (InterruptedException e) {
System.out.println("Main Thread Interrupted");
}
System.out.println("All threads completed");
}
}
```

Лістинг 3.3 – Управління процесами за допомогою методів sleep() і join()

У батьківському методі main() створюються два дочірніх процеси thread1 і thread2. Потім відбувається їх запуск методом start(). Після цього, за допомогою послідовних викликів методу join() для кожного дочірнього процесу, батьківський процес переводиться до стану чекання закінчення роботи обох цих процесів. Оскільки під час роботи дочірніх процесів може виникнути виключна ситуація InterruptedException, то виклики методу join() дочірніх процесів охоплюються блоком try–catch.

3.4 Синхронізація процесів. Програмний монітор

У паралельному програмуванні однією з ключових проблем є координація доступу кількох процесів до спільних ресурсів. Неконтрольований одночасний доступ може призвести до стану перегонів (*race condition*) та пошкодження даних. Для вирішення цієї проблеми були розроблені різні механізми синхронізації, і одним з найважливіших та найпоширеніших є **програмний монітор** (або просто *монітор*).

Концепція монітора була запропонована Пером Брінчем Хансеном[13] та Тоні Гоаром[14] на початку 1970–х років як високорівнева конструкція для безпечного та структурованого керування доступом до спільних даних у конкурентному середовищі. Монітор інкапсулює спільні дані разом з процедурами (методами), які оперують цими даними, та забезпечує необхідну синхронізацію для процесів, що викликають ці процедури.

3.4.1 Основні властивості монітора

Монітор характеризується двома основними властивостями:

а) **взаємне виключення (Mutual Exclusion)** – в будь–який момент часу не більше одного процесу може виконувати код всередині монітора (тобто викликати його методи). Якщо один процес вже знаходиться всередині монітора, будь–який інший процес, що намагається увійти до монітора, буде заблокований доти, доки перший процес не вийде з монітора. Це гарантує, що спільні дані, інкапсульовані монітором, не будуть одночасно модифіковані кількома

[13] Пер Брінч Хансен (Per Brinch Hansen), (13.11.1938 – 31.07.2007) – американський дослідник датського походження, відомий своєю роботою в області операційних систем, паралельного програмування та паралельних і розподілених обчислень.

[14] Чарлз Ентоні Річард Гоар (Charles Antony Richard Hoare), 11.12.1934 – англійський вчений, що працює в галузі інформатики та обчислювальної техніки.

процесами;

б) **умовна синхронізація (Conditional Synchronization)** – монітори надають механізм, за допомогою якого процеси можуть тимчасово призупинити своє виконання всередині монітора, якщо певна умова ще не виконана, і чекати, доки ця умова не стане істинною. Це реалізується за допомогою *змінних умови* (*condition variables*).

3.4.2 Змінні умови

Змінна умови – це об'єкт, пов'язаний з монітором, який дозволяє процесам чекати на виконання певних умов.

Основні операції над змінними умови є наступними:

а) wait() – процес, що викликає цю операцію, звільняє блокування монітора і переходить у стан очікування на цій змінній умови. Він залишатиметься заблокованим, доки інший процес не викличе signal() або signalAll() для цієї ж змінної умови. Коли процес пробуджується, він повинен знову спробувати захопити блокування монітора, перш ніж продовжити виконання;

б) notify() – пробуджує *один* з процесів, що очікують на цій змінній умови. Якщо таких процесів немає, операція не має ефекту. Який саме процес, з тих, що чекають, буде пробуджено, зазвичай невідомо;

в) notifyAll() – пробуджує *усі* процеси, що очікують на цій змінній умови. Кожен пробуджений процес намагатиметься знову захопити монітор.

Важливо зазначити, що після пробудження процес, що викликав wait(), повинен *перевірити умову* ще раз, оскільки:

– інший процес міг захопити монітор першим і змінити стан так, що умова знову стала хибною;

– можливі так звані «помилкові пробудження» (spurious wakeups), коли процес пробуджується без явного виклику signal()/signalAll().

Тому очікування на умову зазвичай реалізується в циклі:

```
// Процес вже володіє блокуванням монітора
while (!condition) {
conditionVariable.wait(); // або await() для Condition
}
// Умова виконана, можна продовжувати
```

Лістинг 3.4 – Шаблон очікування на умову

3.4.3 Реалізація моніторів у Java

Java надає два основних способи реалізації концепції монітора:
1) вбудовані монітори;
2) явні блокування та умови.
Розглянемо їх більш детально.

3.4.3.1 Вбудовані монітори (Intrinsic Locks)

Кожен об'єкт у Java має асоційований з ним вбудований монітор (також відомий як *intrinsic lock* або *monitor lock*). Доступ до цього монітора контролюється за допомогою ключового слова synchronized:

1) **взаємне виключення** – ключове слово synchronized може використовуватися для блоків коду або для цілих методів;

```
// Синхронізований блок
synchronized (someObject) {
 // Код, що виконується з взаємним виключенням
 // на моніторі примірника класу someObject
}

// Синхронізований метод екземпляра
public synchronized void synchronizedMethod() {
 // Код методу виконується з взаємним виключенням
 // на моніторі поточного примірника класу (this)
}

// Синхронізований статичний метод
public static synchronized void staticSynchronizedMethod() {
 // Код методу виконується з взаємним виключенням
 // на моніторі примірника класу (MyClass.class)
}
```

Лістинг 3.5 – Використання ключового слова synchronized

Коли процес входить у synchronized блок або метод, він намагається захопити вбудований монітор відповідного об'єкта. Якщо монітор вже захоплений іншим процесом, поточний процес блокується:

1) **умовна синхронізація** – клас java.lang.Object надає методи wait(), notify() та notifyAll(), які відповідають операціям над змінними умови. Ці методи можна викликати *тільки* з коду, який виконується всередині synchronized блоку або методу для об'єкта, чий монітор використовується. Кожен об'єкт Java має *один* неявний набір очікуючих процесів (*condition queue*), пов'язаний з його вбудованим монітором.

Обмеження вбудованих моніторів:

1) неможливо перервати процес, що очікує на захоплення монітора;
2) неможливо спробувати захопити монітор без блокування (*timed/try lock*);
3) всі умови пов'язані з одним набором очікуючих процесів, що може бути неефективно, якщо потрібно чекати на різні умови. notify() пробуджує довільний процес, що може бути не той, який потрібен;
4) блокування завжди є рекурсивним (процес, що захопив монітор, може повторно його захоплювати).

3.4.3.2 Явні блокування та умови (java.util.concurrent.locks)

Пакет java.util.concurrent.locks, введений у Java 5, надає більш гнучкий та потужний механізм блокувань, який дозволяє реалізувати монітори з розширеними можливостями:

а) **взаємне виключення** – інтерфейс Lock (з основною реалізацією ReentrantLock) надає методи для явного захоплення (lock()) та звільнення (unlock()) блокування. Схема його використання наведена у лістингу 3.6;

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

Lock lock = new ReentrantLock();

// ...

lock.lock(); // Захопити блокування блокується(, якщо зайнято)
try {
 // Критична секція: доступ до спільних ресурсів
} finally {
 lock.unlock(); // Завжди звільняти блокування у finally блоці!
}
```

Лістинг 3.6 – Схема використання ReentrantLock

Інтерфейс Lock також надає додаткові можливості:

1) tryLock() – спробувати захопити блокування без блокування;
2) tryLock(long time, TimeUnit unit) – спробувати захопити блокування протягом заданого часу;
3) lockInterruptibly() – захопити блокування, але дозволити переривання процесу під час очікування;

б) **умовна синхронізація** – інтерфейс Condition надає механізм змінних умови, аналогічний wait()/notify()/notifyAll(), але пов'язаний з конкретним об'єктом Lock. Об'єкт Condition створюється за допомогою методу newCondition() об'єкта Lock. Він має наступні методи:

1) await() – аналог wait(). Процес звільняє Lock і чекає;
2) signal() – аналог notify(). Пробуджує один процес, що очікує на

цій умові;

3) signalAll() – аналог notifyAll(). Пробуджує всі процеси, що очікують на цій умові.

Перевагою Condition є можливість мати *кілька* змінних умови, пов'язаних з одним блокуванням Lock. Це дозволяє створювати більш ефективні та зрозумілі конструкції для складних умов синхронізації (наприклад, окремі умови для генераторів та споживачів у задачі обмеженого буфера). Приклад використання Lock та Condition наведений у лістингу 3.7.

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class BoundedBuffer {
 final Lock lock = new ReentrantLock();
 final Condition notFull = lock.newCondition(); // Умова: буфер не повний
 final Condition notEmpty = lock.newCondition(); // Умова: буфер не порожній

 final Object[] items = new Object[100];
 int putptr, takeptr, count;

 public void put(Object x) throws InterruptedException {
 lock.lock();
 try {
 while (count == items.length) {
 System.out.println("Buffer full, producer waiting...");
 notFull.await(); // Чекати, чекати вивільнення порожнього місця
 }
 items[putptr] = x;
 if (++putptr == items.length) putptr = 0;
 ++count;
 System.out.println("Produced item, signaling consumer.");
 notEmpty.signal(); // Повідомити споживача, що є елемент
 } finally {
 lock.unlock();
 }
 }

 public Object take() throws InterruptedException {
 lock.lock();
 try {
 while (count == 0) {
 System.out.println("Buffer empty, consumer waiting...");
 notEmpty.await(); // Чекати, появи нового елементу
 }
 Object x = items[takeptr];
 if (++takeptr == items.length) takeptr = 0;
 --count;
 System.out.println("Consumed item, signaling producer.");
 notFull.signal(); // Повідомити споживача, що є вільне місце
 return x;
 } finally {
 lock.unlock();
 }
 }
}
```

Лістинг 3.7 – Приклад використання Lock та Condition

3.5 Програмні механізми синхронізації

Паралельне програмування є важливим аспектом сучасної розробки програмного забезпечення, особливо з розповсюдженням багатоядерних процесорів та розподілених систем. У паралельному програмуванні процеси часто потребують доступу до спільних ресурсів (даних або пристроїв). Без належного контролю одночасний доступ кількох процесів до одного ресурсу може призвести до помилок, відомих як *стан перегонів* (race condition), пошкодження даних або непередбачуваної поведінки програми. Механізми синхронізації призначені для координації дій процесів, забезпечення взаємного виключення доступу до критичних секцій коду та організації очікування певних подій.

Мова Java надає потужний набір програмних інструментів для синхронізації, які представлені у пакеті java.util.concurrent.

В цьому розділі розглянуто ключові програмні механізми синхронізації: атомарні змінні, засувки, бар'єри, ExecutorService, Future та ForkJoinPool, які дозволяють розробникам контролювати виконання паралельних задач, керувати процесами та ефективно організовувати обчислення.

3.5.1 Атомарні змінні (Atomic Variables)

3.5.1.1 Концепція, призначення та принцип роботи

Атомарні змінні надають механізм для виконання простих операцій над змінними (наприклад, інкремент, декремент, оновлення) як єдиної, неподільної (атомарної) операції без необхідності використання блокувань (synchronized або Lock).

Це досягається за допомогою низькорівневих атомарних інструкцій процесора, таких як *Compare–And–Swap* (CAS).

Принцип роботи CAS: Операція CAS приймає три операнди: адресу пам'яті (V), очікуване старе значення (A) та нове значення (B). Операція атомарно оновлює значення в пам'яті V до B, *тільки якщо* поточне значення V дорівнює очікуваному значенню A. В іншому випадку нічого не змінюється. Операція повертає інформацію про те, чи відбулося оновлення.

3.5.1.2 Основні атомарні класи

У Java атомарні змінні представлені класами в пакеті java.util.concurrent.atomic, такими як:

а) AtomicInteger – атомарні операції з цілими числами;

б) AtomicLong – атомарні операції з довгими цілими числами;

в) AtomicBoolean – атомарні операції з булевими значеннями;

г) AtomicReference<T> – атомарні операції з об'єктними посиланнями.

3.5.1.3 Основні методи атомарних змінних

Атомарні класи надають наступні ключові методи:

а) get() – отримання поточного значення;

б) set(value) – встановлення нового значення;

в) getAndSet(newValue) – атомарна заміна з поверненням старого значення;

г) compareAndSet(expected, update) – умовна заміна значення;

ґ) incrementAndGet() – атомарне збільшення з поверненням нового значення;

д) getAndIncrement()– атомарне збільшення з поверненням старого значення.

3.5.1.4 Переваги та недоліки використання атомарних змінних

Переваги:

а) **продуктивність** – у ситуаціях з низьким та середнім рівнем міжпроцесних конфліктів атомарні змінні часто працюють швидше за блокування, оскільки уникають призупинення процесів;

б) **запобігання взаємним блокуванням (*deadlocks*)** – оскільки блокування не використовуються, ризик виникнення взаємних блокувань, пов'язаних з неправильним порядком захоплення блокувань, відсутній;

в) **запобігання *race conditions*** – під час використання атомарних змінних гарантується неподільність операцій, що з ними виконуються;

г) **простота використання** – використання атомарних змінних не потребує явного керування блокуваннями.

Недоліки:

– інструмент створено для роботи тільки з даними, що мають просту структуру.

3.5.1.5 Приклад використання AtomicInteger

У лістингу 3.8 продемонстровано безпечне інкрементування лічильника counter кількома процесами.

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounterExample {
 private final AtomicInteger counter = new AtomicInteger(0);
 private final AtomicInteger procNumber = new AtomicInteger(0);

 public void increment() {
 int value1 = procNumber.incrementAndGet();
 int value2 = counter.incrementAndGet(); // Атомарна операція
 System.out.println("Proc: " + value1 + " – " + value2);
 procNumber.decrementAndGet();
 }

 public int getValue() {
 return counter.get();
 }

 public static void main(String[] args) {
 AtomicCounterExample example = new AtomicCounterExample();
 for (int i = 0; i < 10; i++) {
 new Thread(() –> example.increment()).start();
 }
 while (example.procNumber.get() > 0) {
 try {
 Thread.sleep(1);
 } catch (InterruptedException ex) {
 System.err.println(ex.toString());
 }
 }
 System.out.println("Result: " + example.getValue());
 }
}
```

Лістинг 3.8 – Використання AtomicInteger для лічильника

У методі main() створюється примірник класу AtomicCounterExample, у якому використовуються два лічильника типу AtomicInteger: counter – як рахівник чисел, і procNumber – як рахівник робочих процесів. Далі, у циклі for створюються та запускаються на виконання 10 анонімних процесів з задачами, що викликають метод increment() для конкурентного збільшення рахівника counter. На початку і в кінці роботи методу increment() виконується також збільшення та зменшення рахівника робочих процесів procNumber відповідно. Це дозволяє за допомогою циклу while у головному методі main() відслідкувати момент закінчення роботи дочірніх процесів.

3.5.2 Засувка (CountDownLatch)

3.5.2.1 Концепція, призначення та принцип роботи

Засувка (CountDownLatch) – це механізм синхронізації, який дозволяє одному або декільком процесам чекати, доки не завершиться певний набір операцій, що виконуються іншими процесами.

Основний клас у Java – java.util.concurrent.CountDownLatch. CountDownLatch ініціалізується певним цілим числом (лічильником). Процеси, які повинні чекати, викликають метод await(). Цей метод блокує процес доти, доки лічильник не досягне нуля. Інші процеси, завершивши свою частину роботи, викликають метод countDown(), який зменшує лічильник на одиницю. Коли лічильник стає нулем, усі процеси, що очікували на await(), розблоковуються і продовжують виконання. Після того, як лічильник досяг нуля, засувка більше не може бути скинута або використана повторно.

3.5.2.2 Основні методи

а) CountDownLatch(int count) – конструктор з початковим значенням лічильника;

б) await() – блокує поточний процес до досягнення лічильником нуля;

в) await(long timeout, TimeUnit unit) – очікування з тайм–аутом;

г) countDown() – зменшує лічильник на одиницю;

д) getCount() – повертає поточне значення лічильника.

3.5.2.3 Сценарії використання

а) **очікування готовності сервісів** – застосовується для запуску основної частини завдання після завершення ініціалізації. Головний процес чекає, поки всі сервіси ініціалізуються;

б) **координація початку виконання** – коли всі процеси чекають сигналу для одночасного старту;

в) **збір результатів** – коли головний процес чекає завершення роботи всіх дочірніх робочих процесів.

3.5.2.4 Приклади використання CountDownLatch

У лістингу 3.9 продемонстровано сценарій використання CountDownLatch для збору результатів паралельних обрахунків.

```
import java.util.concurrent.CountDownLatch;

class StopLatchedThread extends Thread {
 private final CountDownLatch stopLatch;
 private final int procNumber;

 public StopLatchedThread( CountDownLatch s, int n) {
 stopLatch = s;
 procNumber = n;
 }

 @Override
 public void run() {
 System.out.println( "Thread(" + procNumber + "): running");
 stopLatch.countDown();
 }
 }

public class StopLatchedThreadDemo {
 public static void main(String[] args) {
 CountDownLatch stopLatch = new CountDownLatch( 3);
 for (int i = 0; i < 3; ++i) {
 (new StopLatchedThread( stopLatch, i + 1)).start();
 }
 try {
 stopLatch.await();
 }
 catch( InterruptedException e) {
 System.err.println( e.toString());
 }
 System.out.println( "Main process stoped");
 }
 }
```

Лістинг 3.9 – Використання CountDownLatch для збору результатів паралельних обрахунків

Під час створення об'єкту CountDownLatch, його рахівнику привласнюється значення кількості дочірніх процесів. Після цього дочірні процеси створюються та запускаються на виконання, а головний процес переводиться до стану чекання за допомогою методу await(). Усі дочірні процеси перед закінченням виконання свого коду за допомогою методу countDown() зменшують значення рахівника на одиницю. Після того, як значення рахівника досягне нуля головний процес виходить зі стану чекання та продовжує виконання свого коду. Це може бути, наприклад, збирання результатів обрахунків дочірніх процесів у єдиний звіт.

У лістингу 3.10 продемонстровано сценарій використання CountDownLatch для одночасного запуску багатьох паралельних процесів.

```
import java.util.concurrent.CountDownLatch;

class LatchedThread extends Thread {
 private final CountDownLatch startLatch;
 private final int procNumber;

 public LatchedThread( CountDownLatch s, int n) {
 startLatch = s;
 procNumber = n;
 System.out.println( "Thread(" + procNumber + "): created");
 }

 @Override
 public void run() {
 System.out.println( "Thread(" + procNumber + "): started");
 try {
 startLatch.await();
 System.out.println( "Thread(" + procNumber + "): running");
 }
 catch(InterruptedException e) {
 System.err.println( e.toString());
 }
 System.out.println( "Thread(" + procNumber + "): stoped");
 }
 }

public class CountDownLatchDemo {
 public static void main(String[] args) {
 System.out.println( "Main process started");
 CountDownLatch startLatch = new CountDownLatch( 1);
 for (int i = 0; i < 4; ++i) {
 (new LatchedThread(startLatch, i+1)).start();
 }
 try {
 Thread.sleep( 200);
 }
 catch( InterruptedException e) {
 System.err.println( e.toString());
 }
 startLatch.countDown();
 System.out.println( "Main process stoped");
 }
 }
```

Лістинг 3.10 – Використання CountDownLatch для одночасного запуску багатьох паралельних процесів

У цьому випадку під час створення об’єкту CountDownLatch, його рахівнику привласнюється значення одиниці. Дочірні процеси створюються, запускаються на виконання та зупиняються на виконанні

методу await(). У потрібний момент часу головний процес виконує метод countDown(), і усі дочірні процеси, які були зупинені на методі await() одночасно відновлюють свою роботу. Такий сценарій використовується у системах реального часу, коли потрібно одночасно отримати та обробити декілька процесів даних, що надходять з різних джерел.

3.5.3 Phaser

3.5.3.1 Концепція фазування

Phaser – це гнучкий механізм синхронізації, який дозволяє координувати виконання процесів через декілька фаз. На відміну від CountDownLatch, Phaser може бути повторно використаний та підтримує динамічну реєстрацію учасників.

3.5.3.2 Основні поняття

а) **фаза** – етап виконання, через який проходять процеси;
б) **учасник** – процес, зареєстрований у Phaser;
в) **прибуття** – момент, коли процес сигналізує про завершення поточної фази;
г) **просування** – перехід до наступної фази після прибуття всіх учасників до кінця попередньої фази.

3.5.3.3 Основні методи:

1) Phaser() – створення примірника Phaser без учасників;
2) Phaser(int parties) – створення примірника Phaser з заданою кількістю учасників;
3) register() – реєстрація нового учасника;
4) arrive() – сигнал про завершення фази без очікування інших;
5) arriveAndAwaitAdvance() – сигнал і очікування завершення фази;
6) arriveAndDeregister() – завершення участі у синхронізації;
7) getPhase() – отримання номера поточної фази.

3.5.3.4 Приклад використання Phaser

У лістингу 3.11 продемонстровано створення трьох дочірніх процесів, кожен з яких виконує роботу, що складається з трьох фаз. Координація виконання фаз здійснюється головним процесом.

```
import java.util.concurrent.Phaser;

class Worker implements Runnable {
 private final int id;
 private final Phaser phaser;

 Worker(int id, Phaser phaser) {
 this.id = id;
 this.phaser = phaser;
 System.out.println("Працівник " + id + " створений.");
 }

 @Override
 public void run() {
 for (int phase = 0; phase < 3; phase++) {
 doWork(phase);
 phaser.arriveAndAwaitAdvance(); // Завершуємо фазу і чекаємо інших
 }
 phaser.arriveAndDeregister(); // Завершуємо участь
 System.out.println("Працівник " + id + " завершив свою роботу.");
 }

 private void doWork(int phase) {
 try {
 System.out.println("Працівник " + id + " почав роботу у фазі " + phase);
 Thread.sleep((long) (Math.random() * 2000));
 System.out.println("Працівник " + id + " завершив роботу у фазі " + phase);
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }
 }
}

public class PhaserExample {
 public static void main(String[] args) {
 int numberOfWorkers = 3;
 Phaser phaser = new Phaser(numberOfWorkers + 1); // +1 для головного процесу

 // Запуск робочих процесів
 for (int i = 1; i <= numberOfWorkers; i++) {
 final int workerId = i;
 (new Thread( new Worker(workerId, phaser))).start();
 }

 // Головний процес координує фази
 for (int phase = 0; phase < 3; phase++) {
 System.out.println("=== Фаза " + phase + " ===");
 phaser.arriveAndAwaitAdvance(); // Очікуємо завершення фази
 System.out.println("Фаза " + phase + " завершена\n");
 }
 phaser.arriveAndDeregister(); // Головний процес завершує участь
 System.out.println("Головний процес завершив свою роботу.");
 }

}
```

Лістинг 3.11 – Приклад використання Phaser для багатофазної обробки

3.5.4 Бар'єри (CyclicBarrier)

3.5.4.1 Концепція, призначення та принцип роботи

Бар'єр (CyclicBarrier) – це механізм синхронізації, який дозволяє групі процесів чекати один одного в певній точці виконання перед тим, як продовжити роботу. Основний клас у Java – java.util.concurrent.CyclicBarrier.

На відміну від CountDownLatch, CyclicBarrier може бути повторно використаний після досягнення бар'єру всіма процесами.

CyclicBarrier ініціалізується кількістю процесів (*parties*) *N*, які мають досягти бар'єра. Кожен процес, досягнувши точки синхронізації, викликає метод await(). Цей метод блокує процес доти, доки всі *N* процесів не викличуть await(). Після того, як останній процес викликає await(), бар'єр «зламується», всі процеси розблоковуються і можуть продовжити виконання. Після того, як всі процеси пройшли бар'єр, він автоматично скидається і може бути використаний знову для наступної точки синхронізації.

CyclicBarrier часто застосовується в паралельних алгоритмах, що складаються з кількох фаз, де всі процеси повинні завершити поточну фазу перед початком наступної (наприклад, паралельне сортування, ітеративні обчислення в наукових задачах).

3.5.4.2 Основні характеристики

1) **циклічність** – бар'єр автоматично скидається після використання;

2) **фіксована кількість учасників** – кількість процесів, що синхронізуються встановлюється при створенні;

3) **можливість виконання дії** – у момент досягнення бар'єру усіма процесами, перед його «зламом», бар'єр може виконати заздалегідь визначену дію. Це може бути корисно для виконання підготовчих дій для наступного етапу обчислень;

4) **обробка виключень** – бар'єр підтримує переривання та тайм–аути.

3.5.4.3 Основні методи

а) CyclicBarrier(int parties) – створення бар'єру для заданої кількості процесів;

б) CyclicBarrier(int parties, Runnable barrierAction) – створення бар'єру з дією при досягненні бар'єру;

в) await() – очікування досягнення бар'єру всіма процесами;

г) await(long timeout, TimeUnit unit) – очікування досягнення бар'єру з тайм–аутом;

д) getNumberWaiting() – отримання кількості процесів, що очікують;

е) reset() – скидання бар'єру до початкового стану.

3.5.4.4 Приклад використання CyclicBarrier

У лістингу 3.12 продемонстровано типовий сценарій використання CyclicBarrier.

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

class Worker extends Thread {
 CyclicBarrier barrier;
 int workerNumber = 0;

 Worker( CyclicBarrier b, int n) {
 barrier = b;
 workerNumber = n;
 System.out.println( "Worker(" + workerNumber + "): created");
 }

 @Override
 public void run() {
 System.out.println( "Worker(" + workerNumber + "): start");
 try {
 barrier.await();
 System.out.println( "Worker(" + workerNumber + "): end");
 }
 catch (InterruptedException | BrokenBarrierException e) {
 }
 }
 }

public class CyclicBarrierDemo {
 public static void main(String[] args) {
 int workersNumber = 5;
 CyclicBarrier barrier = new CyclicBarrier( workersNumber, () -> {
 System.out.println( "CyclicBarrier Action");
 });

 for (int i = 0; i < workersNumber; i++) {
 new Worker( barrier, i).start();
 }
 }
 }
```

Лістинг 3.12 – Приклад використання CyclicBarrier

Під час створення примірника barrier типу CyclicBarrier його конструктору передається кількість процесів (workersNumber), яких він повинен очікувати і дія, яку він повинен виконати під час «зламу». Далі створюються та запускаються на виконання дочірні процеси типу Worker. У методі run() кожного дочірнього процесу передбачений виклик методу await() циклічного бар'єру.

3.5.5 Future

3.5.5.1 Концепція, призначення та принцип роботи

Future представляє собою результат асинхронного обчислення. Цей інтерфейс надає методи для перевірки, чи завершилось обчислення, очікування його завершення та отримання результату. Future є універсальним механізмом для роботи з асинхронними операціями, який дозволяє контролювати виконання завдань та обробляти їх результати.

3.5.5.2 Основні методи

Інтерфейс Future<V> включає наступні основні методи:

а) boolean cancel(boolean mayInterruptIfRunning) – спроба скасувати виконання завдання;

б) boolean isCancelled() – перевірка, чи було завдання скасовано;

в) boolean isDone() – перевірка, чи завершилось виконання завдання (успішно, з винятком або було скасовано);

г) V get() – очікування результату завдання (блокує поточний процес);

ґ) V get(long timeout, TimeUnit unit) – очікування результату завдання з обмеженням часу.

3.5.5.3 CompletableFuture

В Java 8 був представлений клас CompletableFuture<T>, який реалізує інтерфейси Future<T> та CompletionStage<T> і надає широкі можливості для асинхронного програмування. А саме:

- комбінування декількох асинхронних завдань;
- трансформація та обробка результатів;
- обробка винятків;
- умовне виконання завдань;
- створення ланцюжків залежних операцій.

3.5.5.4 Приклад використання Future

Лістинг 3.13 демонструє використання Future для отримання результатів асинхронних обчислень.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class FutureTaskDemo {

 public static void main(String[] args) {
 int a = 2;
 int b = 3;

 Callable task = () -> {
 return a + b;
 };
 FutureTask<Integer> future = new FutureTask<>(task);
 new Thread(future).start();
 try {
 System.out.println(future.get());
 } catch (InterruptedException | ExecutionException ex) {
 System.err.println(ex.toString());
 }
 }

}
```

Лістинг 3.13 – Використання Future для отримання результатів асинхронних обчислень

3.5.5.5 Приклад використання CompletableFuture

У лістингу 3.14 показані основні операції з CompletableFuture, такі як створення, трансформація, отримання та поєднання результатів, обробка виключень.

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureExample {
 public static void main(String[] args) {
 // Приклад 1: Створення CompletableFuture, що повертає значення
 CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
 System.out.println("Running task in thread: " + Thread.currentThread().getName());
 return "Hello from supplyAsync";
 });

 // Приклад 2: Використання thenApply для перетворення результату CompletableFuture
 CompletableFuture<String> future2 = future1.thenApply(result -> {
 System.out.println("Transforming result in thread: " + Thread.currentThread().getName());
 return result + ", transformed with thenApply";
 });

 // Приклад 3: Використання thenAccept для отримання результату CompletableFuture
```

```
CompletableFuture<Void> future3 = future2.thenAccept(result -> {
System.out.println("Consuming result in thread: " + Thread.currentThread().getName());
System.out.println("Final Result: " + result);
});

// Приклад 4: Використання дескриптора для обробки винятків
CompletableFuture<String> future4 = CompletableFuture.supplyAsync(() -> {
System.out.println("Running task with exception in thread: " + Thread.currentThread().getName());
throw new RuntimeException("Exception occurred");
});

CompletableFuture<String> future5 = future4.handle((result, exception) -> {
if (exception != null) {
System.out.println("Handling exception in thread: " + Thread.currentThread().getName());
return "Fallback result due to exception: " + exception.getMessage();
} else {
return result;
}
});

// Приклад 5: Склеювання двох CompletableFutures
CompletableFuture<String> future6 = CompletableFuture.supplyAsync(() -> "First part");
CompletableFuture<String> future7 = CompletableFuture.supplyAsync(() -> " Second Part");
CompletableFuture<String> combinedFuture = future6.thenCombine(future7, (part1, part2) -> part1
+ part2);

try {
// Чекання завершення обчислень та друк результатів
System.out.println("Result of future1: " + future1.get());
System.out.println("Result of future2: " + future2.get());
future3.get();
System.out.println("Result of future5 (exception handled): " + future5.get());
System.out.println("Result of combinedFuture: " + combinedFuture.get());
} catch (InterruptedException | ExecutionException ex) {
System.err.println(ex.toString());
}
}

}
```

Лістинг 3.14 – Демонстрація основних операцій з CompletableFuture

У лістингу 3.14 показано порядок використання наступних методів:

1) supplyAsync() – створює примірник CompletableFuture, який асинхронно виконує завдання та повертає результат;

2) thenApply() – трансформує результат CompletableFuture після його завершення;

3) thenAccept() – приймає результат CompletableFuture після завершення його обчислення;

4) handle() – обробляє винятки, які можуть виникнути під час виконання CompletableFuture;

5) thenCombine() – поєднує результати двох примірників CompletableFuture;

6) get() – блокує головний процес, доки CompletableFuture не завершиться та не поверне результат.

3.5.6 ExecutorService

3.5.6.1 Концепція, призначення та принцип роботи

ExecutorService є одним із фундаментальних інтерфейсів у Java для керування процесами. Він є розширенням базового інтерфейсу Executor і представляє собою сервіс, який може виконувати задачі асинхронно. ExecutorService забезпечує більш високий рівень абстракції над процесами, звільняючи розробника від необхідності ручного створення та керування процесами.

3.5.6.2 Основні методи

Інтерфейс ExecutorService включає наступні основні методи:

1) void execute(Runnable command) – передає завдання для виконання, не повертаючи результат;

2) <T> Future<T> submit(Callable<T> task) – передає завдання для виконання і повертає Future, який представляє результат виконання завдання;

3) Future<?> submit(Runnable task) – передає завдання для виконання без результату, але повертає Future, який можна використовувати для контролю виконання;

4) <T> Future<T> submit(Runnable task, T result) – передає завдання для виконання і повертає Future з заданим результатом після завершення завдання;

5) <T> List<Future<T> > invokeAll(Collection<? extends Callable<T> > tasks) – виконує всі завдання з колекції і повертає список з Future;

6) <T> T invokeAny(Collection<? extends Callable<T> > tasks) – виконує завдання з колекції і повертає результат одного успішно завершеного завдання;

7) void shutdown() – ініціює завершення роботи ExecutorService, але дозволяє вже запущеним завданням завершитись;

8) List<Runnable> shutdownNow() – намагається зупинити всі активні завдання і повертає список завдань, що очікували виконання;

9) boolean awaitTermination(long timeout, TimeUnit unit) – блокує поточний процес, доки всі завдання не будуть завершені або не мине вказаний час очікування.

3.5.6.3 Типи пулів процесів

Пакет java.util.concurrent.Executors містить фабричні методи для створення різних типів ExecutorService:

а) newSingleThreadExecutor() – створює ExecutorService, який використовує один процес для виконання завдань;

б) newFixedThreadPool(int nThreads) – створює пул з фіксованою кількістю процесів;

в) newCachedThreadPool() – створює пул процесів, який створює нові процеси за потребою, але перевикористовує раніше створені процеси, якщо вони доступні;

г) newScheduledThreadPool(int corePoolSize) – створює пул процесів, який може планувати виконання завдань через певний проміжок часу.

3.5.6.4 Приклад використання

У лістингу 3.15 показано простий приклад використання ExecutorService для паралельного обчислення.

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

public class ExecutorServiceExample {
 public static void main(String[] args) {
 // Створення пулу з фіксованою кількістю процесів
 ExecutorService executor = Executors.newFixedThreadPool(4);

 // Відправлення завдань на виконання
 for (int i = 0; i < 10; i++) {
 final int taskId = i;
 executor.submit(() -> {
 System.out.println("Виконання завдання " + taskId +
 " в робочому процесі " + Thread.currentThread().getName());
 try {
 // Імітація тривалого обчислення
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }
 return "Результат завдання " + taskId;
 });
 }

 // Завершення роботи ExecutorService
 executor.shutdown();
 try {
 // Очікування завершення всіх завдань
 if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
 executor.shutdownNow();
 }
 } catch (InterruptedException e) {
 executor.shutdownNow();
 Thread.currentThread().interrupt();
 }

 System.out.println("Всі завдання завершено");
 }
}
```

Лістинг 3.15 – Приклад використання ExecutorService

3.5.7 ForkJoinPool

3.5.7.1 Концепція, призначення та принцип роботи

ForkJoinPool є спеціалізованою реалізацією ExecutorService, яка оптимізована для підтримки рекурсивної декомпозиції задач за парадигмою «розділяй і володарюй». Ця парадигма передбачає розбиття великої задачі на менші підзадачі, які можуть виконуватися паралельно, з подальшим об'єднанням результатів.

ForkJoinPool використовує алгоритм «work–stealing» (крадіжка роботи), при якому процеси, які завершили свої завдання, можуть «красти» задачі з черг інших зайнятих процесів. Це дозволяє більш ефективно розподіляти обчислювальне навантаження та максимізувати використання доступних обчислювальних ресурсів.

3.5.7.2 Основні компоненти

Механізм ForkJoinPool включає наступні основні компоненти:

а) ForkJoinPool– спеціалізований пул процесів для виконання ForkJoinTask;

б) ForkJoinTask – абстрактний базовий клас для задач, що виконуються у ForkJoinPool;

в) RecursiveTask<V> – підклас ForkJoinTask, який повертає результат;

г) RecursiveAction – підклас ForkJoinTask, який не повертає результат.

3.5.7.3 Основні методи

Клас ForkJoinTask та його підкласи надають наступні ключові методи:

а) fork() – відправляє задачу на асинхронне виконання у пулі;

б) join() – очікує завершення задачі та повертає результат;

в) invoke() – виконує задачу та повертає результат;

г) invokeAll(ForkJoinTask... tasks) – статичний метод, який виконує всі задачі паралельно.

Клас ForkJoinPool надає методи:

а) static ForkJoinPool commonPool() – повертає загальний пул за замовчуванням;

б) <T> T invoke(ForkJoinTask<T> task) – виконує задачу та повертає результат;

в) void execute(ForkJoinTask<?> task) – асинхронно виконує задачу.

3.5.7.4 Приклад: паралельне обчислення суми масиву

У лістингу 3.16 наведено приклад використання ForkJoinPool для паралельного обчислення суми елементів великого масиву.

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

public class ForkJoinSumExample {

 // Задача для обчислення суми елементів масиву
 static class SumTask extends RecursiveTask<Long> {
 private final int[] array;
 private final int start;
 private final int end;
 private static final int THRESHOLD = 10000; // Поріг розділення задачі

 public SumTask(int[] array, int start, int end) {
 this.array = array;
 this.start = start;
 this.end = end;
 }

 @Override
 protected Long compute() {
 int length = end – start;

 // Якщо розмір задачі досить малий, обчислюємо суму напряму
 if (length <= THRESHOLD) {
 long sum = 0;
 for (int i = start; i < end; i++) {
 sum += array[i];
 }
 return sum;
 }

 // Інакше розділяємо задачу на дві підзадачі
 int middle = start + length / 2;
 SumTask leftTask = new SumTask(array, start, middle);
 SumTask rightTask = new SumTask(array, middle, end);

 // Виконуємо праву підзадачу асинхронно
 rightTask.fork();

 // Виконуємо ліву підзадачу в поточному процесі
 Long leftResult = leftTask.compute();

 // Очікуємо завершення правої підзадачі і отримуємо результат
 Long rightResult = rightTask.join();
```

```
// Складаємо результати обох підзадач
return leftResult + rightResult;
}
}

public static void main(String[] args) {
// Створюємо великий масив для тестування
int size = 100_000_000;
int[] array = new int[size];
Random random = new Random();
for (int i = 0; i < size; i++) {
array[i] = random.nextInt(100);
}

// Створюємо ForkJoinPool або( можна використати commonPool)
ForkJoinPool pool = new ForkJoinPool();

// Створюємо задачу
SumTask task = new SumTask(array, 0, array.length);

// Вимірюємо час виконання
long startTime = System.currentTimeMillis();

// Виконуємо задачу та отримуємо результат
Long result = pool.invoke(task);

long endTime = System.currentTimeMillis();

System.out.println("Сума елементів: " + result);
System.out.println("Час виконання: " + (endTime – startTime) + " мс");

// Перевірка правильності за допомогою послідовного обчислення
long controlSum = 0;
for (int value : array) {
controlSum += value;
}
System.out.println("Контрольна сума: " + controlSum);
System.out.println("Результати співпадають: " + (result == controlSum));

// Завершуємо роботу пулу
pool.shutdown();
}
}
```

Лістинг 3.16 – Приклад використання ForkJoinPool для паралельного обчислення суми елементів великого масиву

3.5.7.5 Використання ForkJoinPool з Java Stream API

Починаючи з Java 8, ForkJoinPool використовується під капотом для паралельної обробки даних за допомогою Stream API. Така обробка відбувається під час виклику методу parallel() для обробки потоку даних. Приклад такої обробки наведено у лістингу 3.17.

```
import java.util.Arrays;
import java.util.concurrent.ForkJoinPool;

public class ParallelStreamExample {
 public static void main(String[] args) {
 int[] array = new int[10_000_000];
 Arrays.setAll(array, i -> i);

 System.out.println("Доступні процесори: " +
 Runtime.getRuntime().availableProcessors());
 System.out.println("Паралелізм ForkJoinPool: " +
 ForkJoinPool.commonPool().getParallelism());

 // Використання паралельного процесу для обробки даних
 long startTime = System.currentTimeMillis();

 long sum = Arrays.stream(array)
 .parallel() // Переключення на паралельну обробку
 .filter(n -> n % 2 == 0) // Фільтрація парних чисел
 .mapToLong(n -> n * n) // Обчислення квадратів
 .sum(); // Обчислення суми

 long endTime = System.currentTimeMillis();

 System.out.println("Результат: " + sum);
 System.out.println("Час виконання: " + (endTime - startTime) + " мс");

 // Порівняння з послідовною обробкою
 startTime = System.currentTimeMillis();

 sum = Arrays.stream(array)
 // Без виклику parallel()
 .filter(n -> n % 2 == 0)
 .mapToLong(n -> n * n)
 .sum();

 endTime = System.currentTimeMillis();

 System.out.println("Результат послідовної обробки: " + sum);
 System.out.println("Час послідовного виконання: " +
 (endTime - startTime) + " мс");
 }
}
```

Лістинг 3.17 – Приклад використання ForkJoinPool з *Java Stream API*

3.5.8 Рекомендації щодо оптимального вибору механізмів синхронізації

а) **атомарні змінні** – призначені для простих операцій зі спільними змінними, лічильників, прапорців;

б) **CountDownLatch** – призначений для одноразової координації, очікування готовності ресурсів;

в) **Phaser** – призначений для складних багатофазних алгоритмів із змінною кількістю учасників;

г) **CyclicBarrier** – призначений для ітеративних алгоритмів, які потребують синхронізації на кожній ітерації;

ґ) **Future** використовується для:

- отримання результатів асинхронних обчислень;
- контролю виконання завдань (перевірка стану, відміна);
- очікування завершення завдань;

д) **ExecutorService** найкраще підходить для:

- виконання незалежних завдань;
- випадків, коли необхідний контроль над кількістю процесів;
- простих паралельних обчислень без складної декомпозиції;
- асинхронного виконання періодичних задач (ScheduledExecutorService);

е) **ForkJoinPool** найкраще підходить для:

- паралельної обробки великих даних;
- задач, які можуть бути розбиті на підзадачі за схемою «розділяй і володарюй»;
- рекурсивної декомпозиції завдань;
- задач з великою кількістю підзадач різного розміру.

Загальні рекомендації:

1) **вибір правильного механізму** – аналізуйте специфіку задачі перед вибором інструменту;

2) **обробка виключень** – завжди обробляйте InterruptedException та інші виключення;

3) **управління ресурсами** – використовуйте конструкцію try-with-resources або явно закривайте ресурси;

4) **тестування** – ретельно тестуйте багатопроцесні програми на різних конфігураціях.

3.5.9 Типові патерни використання

а) **пул процесів з обмеженою кількістю ресурсів** – використовуйте ExecutorServiceз фіксованим розміром пулу, враховуючи доступні ресурси системи:

```
int processors = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(processors);
```

б) **паралельна обробка колекцій** – для простої паралельної обробки колекцій використовуйте Java Stream API з parallel().

```
List<Integer> result = list.parallelStream()
 .map(item –> processItem(item))
 .collect(Collectors.toList());
```

в) **виконання незалежних задач з отриманням результатів** – використовуйте ExecutorService з Future для паралельного виконання незалежних завдань і отримання їх результатів:

```
List<Future<Result>> futures = new ArrayList<>();
for (Task task : tasks) {
 futures.add(executor.submit(() –> processTask(task)));
}

List<Result> results = new ArrayList<>();
for (Future<Result> future : futures) {
 results.add(future.get());
}
```

г) **асинхронні ланцюжки обробки** – використовуйте CompletableFuture для створення ланцюжків асинхронних операцій:

```
CompletableFuture.supplyAsync(() –> fetchData())
 .thenApply(data –> processData(data))
 .thenAccept(result –> saveResult(result))
 .exceptionally(ex –> handleError(ex));
```

г) **паралельна рекурсія** – використовуйте ForkJoinPool для задач, які можуть бути рекурсивно розбиті на підзадачі:

```
class RecursiveTask extends RecursiveTask<Result> {
 @Override
 protected Result compute() {
 if (isSmallEnough()) {
 return computeDirectly();
 }

 RecursiveTask left = new RecursiveTask(/* параметри лівої підзадачі */);
 RecursiveTask right = new RecursiveTask(/* параметри правої підзадачі */);

 left.fork();
 Result rightResult = right.compute();
 Result leftResult = left.join();

 return combine(leftResult, rightResult);
 }
}
```

3.5.10 Порівняння механізмів синхронізації

Для більшої зручності вибору оптимального механізму синхронізації робочих процесів можна скористатися порівняльною табл. 3.2.

Таблиця 3.2 – Порівняння механізмів синхронізації та управління процесами

Характеристика	***Atomic***	***CountDown Latch***	***Phaser***	***Cyclic Barrier***	***Executor Service***	***Future***	***ForkJoin Pool***
Повторне використання	Так	Ні	Так	Так	Так	Ні	Так
Динамічна кількість учасників	Н/Д	Ні	Так	Ні	Так	Н/Д	Так
Багатофазність	Ні	Ні	Так	Обмежено	Ні	Ні	Ні
Блокування процесів	Ні	Так	Так	Так	Ні	Так	Ні
Управління процесами	Ні	Ні	Ні	Ні	Так	Ні	Так
Асинхронність	Ні	Ні	Ні	Ні	Так	Так	Так
Отримання результату	Ні	Ні	Ні	Ні	Через Future	Так	Через Future
Рекурсивні задачі	Ні	Ні	Ні	Ні	Обмежено	Ні	Так
Work–stealing	Ні	Ні	Ні	Ні	Ні	Ні	Так
Складність використання	Низька	Середня	Висока	Середня	Низька	Низька	Середня
Продуктивність	Висока	Середня	Середня	Середня	Висока	Середня	Дуже висока

3.6 Програмні механізми міжпроцесної комунікації

Паралельне програмування часто вимагає взаємодії між різними процесами. Ця взаємодія, відома як **міжпроцесна комунікація** (англ. Inter–Process Communication, IPC), є ключовою для синхронізації дій та обміну даними. У мові Java існує ряд вбудованих механізмів, що полегшують організацію такої комунікації у багатопроцесних застосунках. Цей розділ розглядає два важливих механізми з пакету java.util.concurrent: Exchanger та різні реалізації інтерфейсу BlockingQueue.

3.6.1 Обмінники (Exchanger)

3.6.1.1 Концепція, призначення та принцип роботи

Клас java.util.concurrent.Exchanger<V> надає точку синхронізації, в якій два процеси можуть зустрітися та обмінятися даними. Уявіть собі двох кур'єрів, які мають зустрітися в певному місці, щоб обмінятися пакунками (рис. 3.3). Кожен кур'єр прибуває на місце зустрічі та чекає на іншого. Як тільки обидва кур'єри прибули, вони одночасно обмінюються пакунками і продовжують кожен свій шлях.

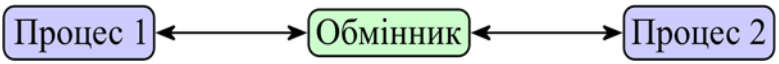


Рисунок 3.3 – Концепція використання обмінника

Саме таку логіку реалізує Exchanger. Кожен процес, що бере участь в обміні, викликає метод exchange(V x):

– процес передає в метод exchange() об'єкт (x), який він хоче віддати;

– процес блокується доти, доки інший процес не викличе метод exchange() на цьому ж екземплярі Exchanger;

– коли другий процес приходить, відбувається обмін: перший процес отримує об'єкт, переданий другим процесом, а другий процес отримує об'єкт, переданий першим;

– метод exchange() повертає об'єкт, отриманий від іншого процесу.

Exchanger є параметризованим типом (Exchanger<V>), де V – це тип даних, якими обмінюються процеси.

3.6.1.2 Приклад використання Exchanger

Розглянемо простий приклад, де два процеси обмінюються рядковими повідомленнями (лістинг 3.18).

```
import java.util.concurrent.Exchanger;

public class ExchangerDemo {

 public static void main(String[] args) {
 Exchanger<String> exchanger = new Exchanger<>();

 // Процес 1
 new Thread(() -> {
 try {
 String messageFromThread1 = "Повідомлення від Процесу 1";
 System.out.println("Процес 1 готує: " + messageFromThread1);
 // Очікує на Процес 2 та обмінюється даними
```

```
String receivedMessage = exchanger.exchange(messageFromThread1);
System.out.println("Процес 1 отримав: " + receivedMessage);
} catch (InterruptedException e) {
Thread.currentThread().interrupt();
System.err.println("Процес 1 перервано.");
}
}).start();

// Процес 2
new Thread(() -> {
try {
String messageFromThread2 = "Повідомлення від Процесу 2";
// Невелика затримка для демонстрації очікування
Thread.sleep(1000);
System.out.println("Процес 2 готує: " + messageFromThread2);
// Очікує на Процес 1 та обмінюється даними
String receivedMessage = exchanger.exchange(messageFromThread2);
System.out.println("Процес 2 отримав: " + receivedMessage);
} catch (InterruptedException e) {
Thread.currentThread().interrupt();
System.err.println("Процес 2 перервано.");
}
}).start();
}
}
```

Лістинг 3.18 – Приклад використання Exchanger

У цьому прикладі кожен процес створює своє повідомлення, викликає exchange(), передаючи його, і блокується. Коли обидва процеси досягають точки обміну, exchanger атомарно обмінює їхні повідомлення, і кожен процес отримує повідомлення від іншого.

3.6.1.3 Застосування Exchanger

Exchanger корисний у сценаріях, де потрібно синхронізувати саме два процеси для парного обміну даними, наприклад:

– обмін буферами між процесами (один заповнює, інший обробляє);

– реалізація деяких алгоритмів паралельного сортування або обробки даних, де потрібен парний обмін;

– координація між двома стадіями конвеєра обробки.

Важливо пам’ятати, що Exchanger розрахований саме на два процеси. Якщо третій процес спробує викликати exchange(), він також заблокується, очікуючи на четвертого партнера для обміну.

Існує також перевантажена версія методу exchange(V x, long timeout, TimeUnit unit), яка дозволяє вказати максимальний час очікування іншого процесу. Якщо час очікування спливає, метод генерує TimeoutException.

3.6.2 Черги (Queue)

3.6.2.1 Концепція, призначення та принцип роботи

Черги є одним з найпоширеніших механізмів для передачі даних між процесами, особливо в моделі «Генератор–Споживач» (Producer–Consumer). У цій моделі один або декілька процесів (генератори) утворюють дані та розміщують їх у спільну чергу, а один або декілька інших процесів (споживачі) вилучають дані з черги для обробки (рис. 3.4).

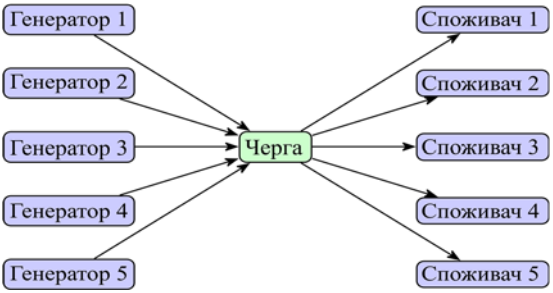


Рисунок 3.4 – Концепція використання черги

Пакет java.util.concurrent надає інтерфейс BlockingQueue<E>, який розширює стандартний java.util.Queue<E> і додає важливу функціональність для паралельного програмування: можливість блокування:

1) **блокування при додаванні елемента** – якщо черга заповнена (має обмежений розмір), процес, що намагається додати елемент за допомогою методу put(E e), буде заблокований доти, доки в черзі не з'явиться вільне місце (тобто інший процес не вилучить елемент);

2) **блокування при вилученні елемента** – якщо черга порожня, процес, що намагається вилучити елемент за допомогою методу take(), буде заблокований доти, доки в черзі не з'явиться новий елемент (тобто інший процес не додасть елемент).

Ці блокуючі операції роблять BlockingQueue ідеальним інструментом для безпечної передачі даних між процесами без необхідності вручну керувати синхронізацією доступу до черги.

3.6.2.2 Основні реалізації BlockingQueue

Існує кілька стандартних реалізацій BlockingQueue в Java:

1) ArrayBlockingQueue<E> – обмежена черга на основі масиву фіксованого розміру. Розмір задається при створенні і не може бути змінений. Елементи обробляються за принципом FIFO (First–In, First–Out). Може бути налаштована на «чесний» (fair) режим, де процеси обслуговуються в порядку їхнього очікування;

2) LinkedBlockingQueue<E> – черга на основі зв'язаного списку. Може бути обмеженою (якщо розмір вказано в конструкторі) або необмеженою (за замовчуванням, розмір обмежений Integer.MAX_VALUE). Зазвичай має вищу пропускну здатність, ніж ArrayBlockingQueue, але споживає більше пам'яті;

3) PriorityBlockingQueue<E> – необмежена черга, яка впорядковує елементи відповідно до їхнього природного порядку або за допомогою штучно наданого інструменту порівняння Comparator. Метод take() повертає елемент з найвищим пріоритетом;

4) SynchronousQueue<E> – черга без внутрішньої ємності. Кожна операція put() повинна чекати на відповідну операцію take() іншого процесу, і навпаки. По суті, це реалізація прямої передачі даних між процесами, схожа на Exchanger, але для одного елемента за раз і без обміну (тільки передача).

3.6.2.3 Приклад використання BlockingQueue (Генератор–Споживач)

Розглянемо класичний приклад з одним генератором та одним споживачем, які використовують ArrayBlockingQueue (лістинг 3.19).

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ThreadLocalRandom;

public class ProducerConsumerQueue {

 public static void main(String[] args) {
 // Створюємо обмежену чергу розміром 10
 BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);

 // ПроцесГенератор–
 Runnable producer = () –> {
 try {
 int value = 0;
 while (true) {
 // Генеруємо дані
 int data = value++;
 System.out.println("Генератор: генерує " + data);
 // Кладемо дані в чергу блокується(, якщо черга повна)
```

```
queue.put(data);
// Імітуємо час на генерацію
Thread.sleep(ThreadLocalRandom.current().nextInt(100, 500));
}
} catch (InterruptedException e) {
Thread.currentThread().interrupt();
System.err.println("Генератора перервано.");
}
};

// ПроцесСпоживач–
Runnable consumer = () -> {
try {
while (true) {
// Беремо дані з черги блокується(, якщо черга порожня)
Integer data = queue.take();
System.out.println("Споживач: обробляє " + data);
// Імітуємо час на обробку
Thread.sleep(ThreadLocalRandom.current().nextInt(500, 1000));
}
} catch (InterruptedException e) {
Thread.currentThread().interrupt();
System.err.println("Споживача перервано.");
}
};

new Thread(producer).start();
new Thread(consumer).start();
}
}
```

Лістинг 3.19 – Приклад Генератор–Споживач з ArrayBlockingQueue

У цьому прикладі генератор безперервно генерує цілі числа і кладе їх у чергу за допомогою put(). Якщо черга заповнена (досягла розміру 10), процес генератора заблокується. Споживач безперервно вилучає числа з черги за допомогою take(). Якщо черга порожня, споживач заблокується. BlockingQueue автоматично керує синхронізацією та блокуванням.

3.6.2.4 Інші методи BlockingQueue

Крім блокуючих put() та take(), інтерфейс BlockingQueue надає інші методи з різною поведінкою:

а) add(E e) – додає елемент до черги, генерує IllegalStateException, якщо черга повна (успадковано від Queue);

б) offer(E e) – додає елемент до черги, повертає false, якщо черга повна (не блокується);

в) offer(E e, long timeout, TimeUnit unit) – намагається додати елемент до черги протягом вказаного часу, повертає false, якщо час сплив, а місце не з'явилося;

г) remove() – вилучає та повертає елемент з голови черги, генерує NoSuchElementException, якщо черга порожня (успадковано від Queue);

ґ) poll() – вилучає та повертає елемент з голови черги, повертає null, якщо черга порожня (не блокується);

д) poll(long timeout, TimeUnit unit) – намагається вилучити елемент з голови черги протягом вказаного часу, повертає null, якщо час сплив, а елемент не з'явився;

е) element() – повертає елемент з голови черги без вилучення, генерує NoSuchElementException, якщо черга порожня (успадковано від Queue);

є) peek() – повертає елемент з голови черги без вилучення, повертає null, якщо черга порожня.

Вибір конкретного методу залежить від необхідної логіки обробки ситуацій повної або порожньої черги.

3.6.2.5 Застосування BlockingQueue

Черги є надзвичайно гнучким механізмом і застосовуються у багатьох сценаріях паралельного програмування:

– реалізація патерну «Генератор–Споживач»;

– управління пулом процесів (ThreadPoolExecutor використовує BlockingQueue для зберігання завдань);

– асинхронна обробка подій;

– передача даних між різними етапами конвеєрної обробки;

– буферизація даних для згладжування різниці у швидкості між процесами.

3.6.3 Порівняння механізмів міжпроцесної комунікації

Exchanger та BlockingQueue надають потужні та зручні засоби для організації міжпроцесної комунікації в Java. Exchanger є специфічним інструментом для синхронного парного обміну даними між двома процесами.

BlockingQueue, навпаки, є більш універсальним механізмом, що ідеально підходить для реалізації моделі «Генератор–Споживач», асинхронної передачі даних та керування процесами, забезпечуючи при цьому безпечність їх роботи та механізми блокування.

Вибір між цими механізмами залежить від конкретної задачі комунікації та синхронізації, яку необхідно вирішити.

Контрольні питання та завдання

Загальні питання

1. Що таке багатопроцесність (multithreading)? Яка головна мета її використання?
2. Назвіть та опишіть основні переваги використання багатопроцесності в програмах.
3. Які потенційні проблеми можуть виникнути при розробці багатопроцесних застосунків?
4. Поясніть різницю між процесом (thread) і процесом (stream) в контексті операційної системи та Java.
5. Що таке «умова змагання» (race condition)? Наведіть приклад коду.
6. Що таке «взаємне блокування» (deadlock)? Опишіть умови, необхідні для його виникнення.
7. Поясніть поняття «спільні ресурси» в контексті багатопроцесності.

Основні стани життєвого циклу процесу

1. Опишіть життєвий цикл процесу в Java.
2. Опишіть стан *New* процесу в Java.
3. Опишіть стан *Runnable* процесу в Java.
4. Опишіть стан *Running* процесу в Java.
5. Опишіть стан *Waiting* процесу в Java.
6. Опишіть стан *Timed Waiting* процесу в Java.
7. Опишіть стан *Terminated* процесу в Java.

Створення та запуск процесів в Java

1. Які існують основні способи створення процесу в Java? Опишіть кожен з них.
2. Які переваги має спосіб створення процесу в Java шляхом успадкування класу Thread?
3. Які недоліки має спосіб створення процесу в Java шляхом успадкування класу Thread?
4. Які переваги має спосіб створення процесу в Java шляхом імплементації інтерфейсу Runnable?
5. Які недоліки має спосіб створення процесу в Java шляхом імплементації інтерфейсу Runnable?
6. Який спосіб створення процесів є кращим: успадкування від класу Thread чи реалізація інтерфейсу Runnable? Поясніть чому.

Методи управління процесами

1. Яка різниця між методами start() та run() класу Thread?
2. Для чого використовується метод join()?
3. Для чого використовується метод sleep()?
4. Для чого використовується метод interupt()?
5. Для чого використовується метод isAlive()?
6. Як можна перервати виконання процесу в Java? Чи є метод stop() безпечним для використання?

Синхронізація процесів. Програмний монітор

1. Що таке синхронізація і для чого вона потрібна?
2. Поясніть механізм роботи ключового слова synchronized (для методів та блоків).
3. Що таке монітор в Java?
4. Поясніть призначення та принцип роботи ключового слова volatile.
5. У чому різниця між synchronized та volatile? Коли що слід використовувати?

Програмні механізми синхронізації

1. Що таке атомарні операції? Наведіть приклади класів з пакету java.util.concurrent.atomic.
2. Опишіть проблему видимості (visibility problem) змінних у багатопроцесному середовищі.
3. Що таке засувка CountDownLatch?
4. Поясніть принцип роботи CountDownLatch.
5. Які існують статичні методи в класі CountDownLatch? Для чого вони призначені?
6. Які існують сценарії використання CountDownLatch?
7. Що таке Phaser?
8. Поясніть концепцію фазування виконання процесу.
9. Які існують статичні методи в класі Phaser? Для чого вони призначені?
10. Що таке бар'єр CyclicBarrier?
11. Поясніть принцип роботи CyclicBarrier.
12. Які існують статичні методи в класі CyclicBarrier? Для чого вони призначені?
13. Які існують сценарії використання CyclicBarrier?
14. Що таке Future?

15.Що таке Runnable та Callable? Яка між ними різниця?

16.Що таке Executor і які проблеми він вирішує?

17.Опишіть ієрархію інтерфейсів: Executor, ExecutorService, ScheduledExecutorService.

18.Які існують статичні методи в класі Executors для створення пулів процесів?

19.Опишіть newFixedThreadPool, newCachedThreadPool та newSingleThreadExecutor.

20.Як отримати результат виконання завдання, переданого до ExecutorService?

21.Як коректно завершити роботу ExecutorService? Опишіть методи
shutdown() та shutdownNow().

22.Що таке ForkJoinPool і які проблеми він вирішує?

23.Опишіть основні компоненти ForkJoinPool.

24.Які існують статичні методи в класі ForkJoinPool? Для чого вони призначені?

Програмні механізми міжпроцесної комунікації

1. Поясніть призначення методів wait(), notify() та notifyAll(). В якому блоці вони мають викликатись?
2. Що таке «втрачений сигнал» (lost wakeup) і як його уникнути?
3. Що таке обмінники Exchanger?
4. Поясніть принцип роботи Exchanger.
5. Які існують статичні методи в класі Exchanger? Для чого вони призначені?
6. Які існують сценарії використання Exchanger?
7. Що таке блокуючі черги BlockingQueue?
8. Поясніть принцип роботи BlockingQueue.
9. Які існують статичні методи в класі BlockingQueue? Для чого вони призначені?
10. Які існують сценарії використання BlockingQueue?

РОЗДІЛ 4
ПРИКЛАДИ РЕАЛІЗАЦІЇ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

У попередніх розділах були розглянуті фундаментальні концепції та інструменти для створення паралельних програм. Тепер настав час перейти до конкретних прикладів, які продемонструють, як ці інструменти використовуються для розв'язання реальних обчислювальних задач, підвищуючи ефективність та швидкість їх вирішення.

Цей розділ має на меті не лише показати готові рішення, але й ефективні підходи до розробки власних паралельних алгоритмів. Розпочнемо з огляду методу розробки паралельних алгоритмів за допомогою графа «операції–операнди». Цей підхід дозволяє візуалізувати залежності між різними частинами обчислювального процесу та ефективно визначати можливості для їх розпаралелювання. Розуміння цього методу є ключовим для систематичного проєктування масштабованих та ефективних паралельних програм.

Далі детально розглянемо паралельні реалізації декількох класичних обчислювальних задач, які часто зустрічаються в наукових та інженерних розрахунках:

а) **ітеративне обчислення визначеного інтегралу** – дослідимо, як можна розбити процес чисельного інтегрування на незалежні підзадачі, що виконуватимуться паралельно, для прискорення отримання результату;

б) **метод Монте–Карло** – це потужний метод, що базується на випадкових випробуваннях, за своєю природою є добре пристосованим до розпаралелювання. Покажемо, як можна значно збільшити кількість випробувань за той самий час, використовуючи багатопроцесність, що призводить до підвищення точності результатів;

в) **метод найменших квадратів** – застосовуваний для апроксимації даних, цей метод також може бути оптимізований шляхом паралельного виконання окремих етапів обчислень, особливо при роботі з великими наборами даних.

Особливу увагу буде приділено практичній реалізації цих алгоритмів з використанням стандартних засобів синхронізації та управління процесами, що надаються бібліотекою *Java Concurrency*. Зокрема, розглянемо приклади використання інструментів CountDownLatch, CyclicBarrier та ExecutorService:

– CountDownLatch буде використаний для ситуацій, де один або декілька процесів очікують завершення виконання інших процесів

перед тим, як продовжити свою роботу;

– CyclicBarrier буде використаний для синхронізації групи процесів у певній точці, дозволяючи їм продовжити подальше виконання лише після того, як усі процеси досягнуть цього бар'єру;

– ExecutorService буде використаний для ефективного управління пулом процесів під час виконання великої кількості асинхронних завдань, спрощуючи процес розробки та підвищуючи продуктивність обчислень.

Вивчення цих прикладів дозволить не тільки краще зрозуміти принципи паралельного програмування, але й отримати практичні навички, необхідні для розробки власних ефективних та масштабованих Java–застосунків. Кожен приклад супроводжуватиметься детальним поясненням коду та аналізом потенційних переваг від розпаралелювання.

4.1 Граф «операції–операнди»

Граф «операції–операнди» (Operation–Operand Graph) – це потужний інструмент для аналізу та оптимізації паралельних алгоритмів. Цей математичний формалізм дозволяє наочно представити залежності між обчислювальними операціями та даними, що в свою чергу допомагає виявити потенційні можливості для паралельного виконання.

4.1.1 Означення та основні поняття

Граф «операції–операнди» $G = (V, E)$ – це орієнтований двочастковий граф,

де $V = V_O \cup V_D$ – множина вершин, розділена на дві підмножини:

V_O – множина вершин–операцій (обчислювальні кроки алгоритму);

V_D – множина вершин–операндів (дані, що обробляються);

$E \subseteq (V_O \times V_D) \cup (V_D \times V_O)$ – множина орієнтованих ребер, що відображають відношення між операціями та операндами.

У графі «операції–операнди» існують два типи ребер:

а) ребра від операндів до операцій $(d, o) \in V_D \times V_O$ – позначають, що операція o використовує операнд d як вхідні дані;

б) ребра від операцій до операндів $(o, d) \in V_O \times V_D$ – позначають, що операція o генерує (обчислює) операнд d.

Приклад графа «операції–операнди» наведено на рис. 4.1.

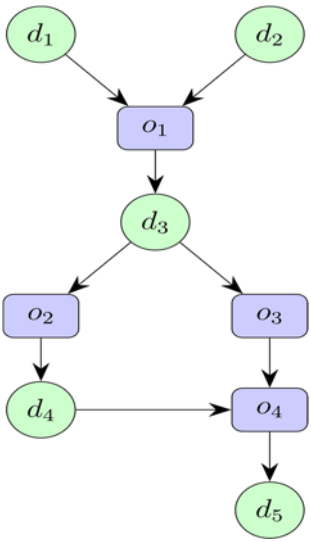


Рисунок 4.1 – Приклад графа «операції–операнди»

4.1.2 Застосування графа «операції–операнди» в паралельному програмуванні

Граф «операції–операнди» є надзвичайно корисним для аналізу можливостей паралелізму в алгоритмах тим, що завдяки графічному представленню інформаційних залежностей, допомагає виявити потенційні можливості для паралельного виконання.

4.1.2.1 Виявлення залежностей між операціями

Дві операції o_i та o_j мають залежність, якщо:

а) існує шлях від o_i до o_j через деякий операнд d (тобто o_i генерує дані, які використовує o_j);

б) існує операнд d, такий що і o_i, і o_j генерують його (конфлікт запису).

Операції, між якими немає залежностей, можуть виконуватися паралельно.

4.1.2.2 Типи залежностей

У паралельному програмуванні розрізняють кілька типів залежностей між операціями:

а) **залежність за даними (data dependency)** – операція o_j використовує дані, згенеровані операцією o_i:

$$o_i \xrightarrow{d} o_j \tag{4.1}$$

б) **антизалежність (anti–dependency)** – операція o_j перезаписує дані, які читає операція o_i:

$$o_i \xrightarrow{anti} o_j \tag{4.2}$$

в) **залежність по виводу (output dependency)** – операції o_i та o_j записують у той самий операнд:

$$o_i \xrightarrow{output} o_j \tag{4.3}$$

Ці залежності утворюють основу для побудови графа залежностей, який використовується для планування паралельного виконання.

4.1.3 Граф залежностей

Граф залежностей операцій $G_D = (V_O, E_D)$ – це орієнтований граф, де V_O – множина вершин–операцій;
$E_D \subseteq V_O \times V_O$ – ребро $(o_i, o_j) \in E_D$ існує, якщо операція o_j залежить від операції o_i.

Граф залежностей можна отримати з графа «операції–операнди» шляхом аналізу шляхів між вершинами–операціями через вершини–операнди.

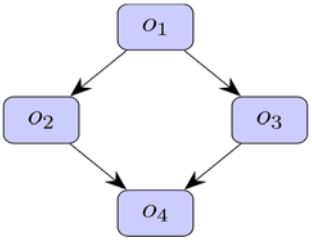


Рисунок 4.2 – Граф залежностей, отриманий з графа «операції–операнди»

4.1.4 Рівні паралелізму та критичний шлях

На основі графа залежностей можна визначити важливі характеристики алгоритму з точки зору паралельного виконання:

4.1.4.1 Рівні паралелізму

Рівень паралелізму – це множина операцій, які можуть виконуватися одночасно.

Формально: **Рівень паралелізму** L_i – це множина операцій, для яких:

а) усі операції–попередники вже виконані (належать до попередніх рівнів $L_0; L_1; \ldots; L_{i-1}$);

б) немає залежностей між операціями всередині одного рівня.

Алгоритм побудови рівнів паралелізму:

Алгоритм 4 Побудова рівнів паралелізму:

Крок 1: $i \leftarrow 0$ ▷ Індекс рівня

Крок 2: $V_{remaining} \leftarrow V_O$ ▷ Операції, які залишилося розподілити

Крок 3: **while** $V_{remaining} \neq \emptyset$ **do**

Крок 4: $L_i \leftarrow \{o \in V_{remaining} \mid \forall o' : (o', o) \in E_D \Rightarrow o' \in \bigcup_j = 0^{i-1} \times L_j\}$

Крок 5: $V_{remaining} \leftarrow V_{remaining} \setminus L_i$

Крок 6: $i \leftarrow i + 1$

Крок 7: **end while**

4.1.4.2 Критичний шлях

Критичний шлях у графі залежностей – це найдовший шлях від початкової до кінцевої операції. Довжина критичного шляху визначає *мінімальний час* виконання алгоритму навіть за умов необмеженого паралелізму. Пошук критичного шляху можна виконати за допомогою алгоритму топологічного сортування та динамічного програмування:

Алгоритм 5 Пошук критичного шляху:

Крок 1: Виконати топологічне сортування графа залежностей

Крок 2: **for** кожна операція o у порядку топологічного сортування **do**

Крок 3: **if** o не має попередників **then**

Крок 4: $length[o] \leftarrow w(o)$ ▷ $w(o)$ – вага (тривалість) операції

Крок 5: **else**

Крок 6: $length[o] \leftarrow w(o) + \max_{o'}:(o', o) \in E_D \times length[o']$

Крок 7: **end if**

Крок 8: **end for**

Крок 9: $CriticalPathLength \leftarrow \max_{o} \in V_O \times length[o]$

4.1.5 Оптимізація паралельних алгоритмів за допомогою графа «операції–операнди»

Граф «операції–операнди» надає наступні можливості для оптимізації паралельних алгоритмів:

а) **мінімізація критичного шляху.** Для прискорення виконання алгоритму можна зосередитись на мінімізації довжини критичного шляху. Цього можна досягти наступними способами:

– розбиття операцій на менші, що дозволяє збільшити рівень паралелізму;

– переставлення операцій для зміни структури залежностей;

– використання більш ефективних алгоритмів для окремих операцій;

б) **балансування навантаження.** При розподілі операцій між виконавцями (процесами) важливо забезпечити рівномірне навантаження. Це можна зробити, виконавши наступні кроки:

– визначити часові навантаження на кожен процес;

– для кожного рівня паралелізму відсортувати операції за часом виконання (за спаданням);

– призначити операції процесам за принципом «найбільші (найдовші) операції – найменш завантаженим процесам».

4.1.6 Приклад застосування методу графа «операції–операнди» для обчислення площі опуклого багатокутника, заданого координатами своїх вершин

4.1.6.1 Постановка задачі

Розглянемо практичну задачу обчислення площі опуклого багатокутника, заданого координатами своїх вершин. Для цього використаємо формулу Шнурівки (Shoelace formula):

$$S = \frac{1}{2}\left|\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right| \tag{4.4}$$

де n – кількість вершин;

(x_i; y_i) – координати i–ї вершини, а індекси беруться за модулем n;

4.1.6.2 Розбиття на операції

Для багатокутника з n вершинами маємо наступні операції:

а) обчислення добутків: $p_i = x_i \times y_i + 1$;

б) обчислення добутків: $q_i = x_i + 1 \times y_i$;

в) обчислення різниць: $d_i = p_i - q_i$;

г) сумування: $sum = \sum_{i-0}^{n-1} d_i$;

д) обчислення модуля: $|sum|$;

е) ділення на 2: $S = \frac{|sum|}{2}$.

4.1.6.3 Побудова графа «Операції–операнди»

Для прикладу розглянемо трикутник з вершинами $A(x_0, y_0)$, $B(x_1, y_1)$, $C(x_2, y_2)$, та побудуємо для нього граф «Операції–операнди» (рис. 4.3).

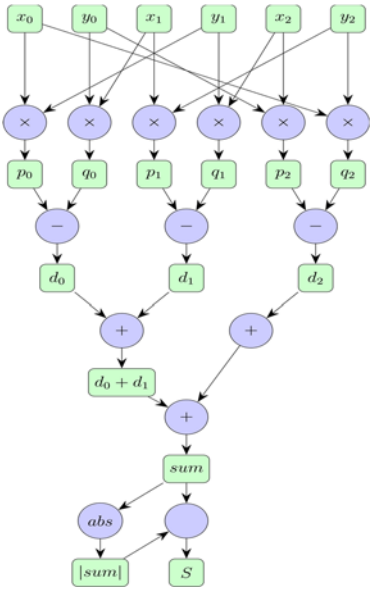


Рисунок 4.3 – Граф «операції–операнди» для обчислення площі трикутника

4.1.6.4 Аналіз паралелізму

З аналізу графа (рис. 4.3) видно, що:
Рівень 1 – всі 6 операцій множення можна виконувати паралельно;
Рівень 2 – 3 операції віднімання можна виконувати паралельно;
Рівень 3 – сумування має ієрархічну структуру;
Рівень 4 – модуль та ділення виконуються послідовно.

4.1.6.5 Оцінка прискорення

Для багатокутника з n вершинами:
– послідовне виконання: $T_1 = 4n + 3$ операції;
– паралельне виконання: $T_p = 4 + \log_2 n + 2$ операції;
– теоретичне прискорення: $S = \frac{4n+3}{4+\log_2 n+2}$.

4.1.6.6 Реалізація на Java

```
public class PolygonArea {
 public static class Point {
 double x, y;
 public Point(double x, double y) {
 this.x = x; this.y = y;
 }
 }

 public static double calculateAreaSequential(Point[] vertices) {
 int n = vertices.length;
 double sum = 0.0;

 for (int i = 0; i < n; i++) {
 int next = (i + 1) % n;
 sum += vertices[i].x * vertices[next].y
 – vertices[next].x * vertices[i].y;
 }

 return Math.abs(sum) / 2.0;
 }
}
```

Лістинг 4.1 – Послідовне обчислення площі багатокутника

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class ParallelPolygonArea extends RecursiveTask<Double> {
 private static final int THRESHOLD = 100;
 private final Point[] vertices;
 private final int start, end;

 public ParallelPolygonArea(Point[] vertices, int start, int end) {
 this.vertices = vertices;
 this.start = start;
 this.end = end;
 }

 @Override
 protected Double compute() {
 if (end – start <= THRESHOLD) {
 return computeDirectly();
 }

 int mid = (start + end) / 2;
 ParallelPolygonArea leftTask =
 new ParallelPolygonArea(vertices, start, mid);
 ParallelPolygonArea rightTask =
 new ParallelPolygonArea(vertices, mid, end);

 leftTask.fork();
 double rightResult = rightTask.compute();
 double leftResult = leftTask.join();
```

```
return leftResult + rightResult;
}

private double computeDirectly() {
double sum = 0.0;
int n = vertices.length;

for (int i = start; i < end; i++) {
int next = (i + 1) % n;
sum += vertices[i].x * vertices[next].y
– vertices[next].x * vertices[i].y;
}

return sum;
}

public static double calculateAreaParallel(Point[] vertices) {
ForkJoinPool pool = new ForkJoinPool();
double sum = pool.invoke(
new ParallelPolygonArea(vertices, 0, vertices.length)
);
pool.shutdown();

return Math.abs(sum) / 2.0;
}
}
```

Лістинг 4.2 – Паралельне обчислення з використанням ForkJoin

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.List;
import java.util.ArrayList;

public class AsyncPolygonArea {

public static double calculateAreaAsync(Point[] vertices) {
int n = vertices.length;
ExecutorService executor = Executors.newFixedThreadPool(
Runtime.getRuntime().availableProcessors()
);

List<CompletableFuture<Double>> futures = new ArrayList<>();

// Розбиваємо обчислення на частини
int chunkSize = Math.max(1, n / 4);

for (int i = 0; i < n; i += chunkSize) {
final int start = i;
final int end = Math.min(i + chunkSize, n);

CompletableFuture<Double> future = CompletableFuture.supplyAsync(() –> {
double partialSum = 0.0;
for (int j = start; j < end; j++) {
int next = (j + 1) % n;
```

```
partialSum += vertices[j].x * vertices[next].y
– vertices[next].x * vertices[j].y;
}
return partialSum;
}, executor);

futures.add(future);
}

// Збираємо результати
double totalSum = futures.stream()
.mapToDouble(CompletableFuture::join)
.sum();

executor.shutdown();
return Math.abs(totalSum) / 2.0;
}
}
```

Лістинг 4.3 – Використання CompletableFuture для паралелізації

```
public class PerformanceBenchmark {

public static void main(String[] args) {
// Генерація тестового багатокутника
Point[] polygon = generateRegularPolygon(1000000, 1000.0);

// Прогрів JVM
for (int i = 0; i < 100; i++) {
PolygonArea.calculateAreaSequential(polygon);
ParallelPolygonArea.calculateAreaParallel(polygon);
}

// Вимірювання часу
long sequentialTime = measureTime(() –>
PolygonArea.calculateAreaSequential(polygon)
);

long parallelTime = measureTime(() –>
ParallelPolygonArea.calculateAreaParallel(polygon)
);

long asyncTime = measureTime(() –>
AsyncPolygonArea.calculateAreaAsync(polygon)
);

System.out.printf("Послідовний алгоритм: %d мс %n", sequentialTime);
System.out.printf("ForkJoin алгоритм: %d мс %n", parallelTime);
System.out.printf("Async алгоритм: %d мс %n", asyncTime);
System.out.printf("Прискорення ForkJoin: %.2fx %n",
(double) sequentialTime / parallelTime);
System.out.printf("Прискорення Async: %.2fx %n",
(double) sequentialTime / asyncTime);
}

private static Point[] generateRegularPolygon(int n, double radius) {
```

```
Point[] vertices = new Point[n];
for (int i = 0; i < n; i++) {
double angle = 2 * Math.PI * i / n;
vertices[i] = new Point(
radius * Math.cos(angle),
radius * Math.sin(angle)
);
}
return vertices;
}

private static long measureTime(Runnable task) {
long start = System.currentTimeMillis();
for (int i = 0; i < 1000; i++) {
task.run();
}
return System.currentTimeMillis() – start;
}
}
```

Лістинг 4.4 – Бенчмарк для порівняння продуктивності

Метод графа «Операції–операнди» демонструє свою ефективність під час аналізу та оптимізації паралельних обчислень. А саме:

а) **візуалізація залежностей** – допомагає зрозуміти структуру алгоритму та можливості паралелізації;

б) **ідентифікація паралелізму** – дозволяє виявити операції, які можна виконувати одночасно;

в) **практична реалізація** – показує різні підходи до паралелізації в Java;

г) **оцінка ефективності** – дає можливість порівняти різні стратегії паралелізації.

Для задачі обчислення площі багатокутника паралелізація дає значне прискорення при великій кількості вершин ($n >> 1000000$), особливо на багатоядерних системах. Вибір конкретного підходу (ForkJoin, CompletableFuture, або інші) залежить від характеристик задачі та доступних ресурсів.

4.2 Паралельне ітеративне обчислення визначеного інтегралу

4.2.1 Постановка задачі

Розглянемо задачу обчислення визначеного інтегралу функції однієї змінної $f(x)$ на інтервалі $[a, b]$:

$$I = \int_a^b f(x)dx \tag{4.5}$$

Ми будемо використовувати ітераційний чисельний метод для досягнення заданої точності ϵ. Популярним підходом є метод трапецій або його модифікації (наприклад, метод Сімпсона).

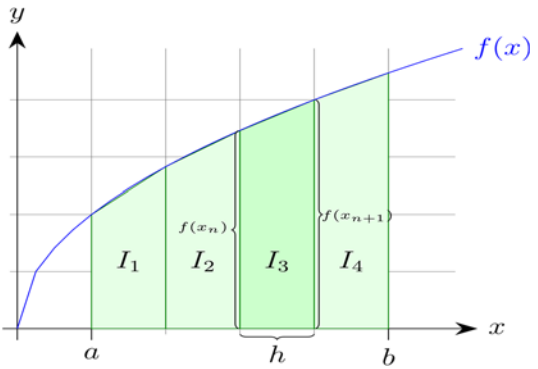


Рисунок 4.4 – Обчислення визначеного інтеграла методом трапецій

За методом трапецій (рис. 4.4) інтеграл наближено обчислюється як сума площ трапецій під графіком функції. Якщо розділити інтервал [*a*, *b*] на *n* рівних підінтервалів довжиною $h = (b - a) / n$, то формула має вигляд:

$$I_n \approx \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)] \quad (4.6)$$

де $x_i = a + i \cdot h$.

Для досягнення заданої точності ϵ використовується ітераційний підхід:

Алгоритм 6 Ітеративне обчислення визначеного інтегралу з заданою точністю ϵ

Крок 1: $n \leftarrow 2$ ▷ Починаємо з невеликої кількості підінтервалів

Крок 2: $I_n \leftarrow \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-1}) + f(x_n)]$

Крок 3: $I_{2n} \leftarrow I_n$ ▷ Готуємось до першій ітерації

Крок 4: **repeat**

Крок 5: $I_n \leftarrow I_{2n}$ ▷ Зберігаємо результат попередньої ітерації

Крок 6: $n \leftarrow 2 \times n$ ▷ Подвоюємо кількість підінтервалів

Крок 7: $I_{2n} \leftarrow \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{2n-1}) + f(x_{2n})]$

Крок 8: **until** $|I_{2n} - I_n| \le \epsilon$ ▷ Якщо різниця менша за ϵ, то I_{2n} вважається результатом

Обчислення $f(x)$ може бути ресурсомістким, а для досягнення високої точності може знадобитися велика кількість підінтервалів n. Це робить задачу гарним кандидатом для паралелізації.

4.2.2 Стратегія паралелізації

Основна ідея полягає в розподілі обчислень суми між кількома процесами. Інтервал $[a, b]$ ділиться на N частин (де N – кількість процесів). Кожен процес обчислює частину інтегралу (суму) на своєму підінтервалі. Потім результати з усіх процесів об'єднуються для отримання загального значення інтегралу I_k для поточного числа підінтервалів k.

Ітераційний процес (перевірка точності та подвоєння n) вимагає синхронізації між процесами або координації з боку головного процесу після завершення кожної ітерації обчислення I_k.

Розглянемо реалізацію цієї стратегії за допомогою різних механізмів синхронізації та управління процесами в Java.

4.2.3 Реалізація алгоритму ітеративного обчислення визначеного інтегралу

Спочатку представимо алгоритм ітеративного обчислення визначеного інтегралу (алгоритм 6) у вигляді задачі Callable (лістинг 4.5).

```
import java.util.concurrent.Callable;

public class IntegrallterationTask implements Callable<Double> {
 int taskNum;
 double lo;
 double hi;
 double epsilon;

 // Конструктор задачі
 IntegrallterationTask(int i, double l, double h, double eps) {
 taskNum = i;
 lo = l;
 hi = h;
 epsilon = eps;
 }

 @Override
 public Double call() throws Exception {
 return Integrallteration( lo, hi, epsilon);
 }

 // Функція користувача для інтегрування
 double userFunction(double a, double b, double x) {
 return(a * x + b);
 }
```

```
// Крок інтегрування
double IntegralStep( double lo, double hi, double step) {
double s = 0.0;

for (double x = lo; x < hi; x += step) {
double f = userFunction( 1, 0, x);
f += userFunction( 1, 0, x + step);
f /= 2;
s += f * step;
}
return( s);
}

// Ітеративний алгоритм інтегрування
double IntegralIteration( double lo, double hi, double epsilon) {
double step = (hi – lo) / 10;
double s1, s2 = 0.0;
do {
s1 = s2;
s2 = IntegralStep( lo, hi, step);
step /= 2;
}
while (Math.abs(s1 – s2) > epsilon);
return( s2);
}
}
```

Лістинг 4.5 – Реалізація алгоритму ітеративного обчислення визначеного інтегралу

Таким чином, для створення об'єкту задачі треба буде використати конструктор IntegralIterationTask(), в якості аргументів якого визначити номер задачі *i*, мінімальну *l* та максимальну *h* межу значень аргументу підінтегральної функції, та точність розрахунку *eps*.

4.2.4 Реалізація обчислення визначеного інтегралу з використанням механізму CountDownLatch

Приклад паралельного обрахунку визначеного інтегралу з використанням механізму синхронізації CountDownLatch наведено у лістингу 4.6.

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

class IntegralDemoCountDownLatch {
 class CountDownLatchTask extends IntegralIterationTask {
 CountDownLatch mainLatch;

 public CountDownLatchTask(int i, double l, double h, double eps, CountDownLatch latch) {
 super(i, l, h, eps);
 mainLatch = latch;
 }
```

```
@Override
public Double call() throws Exception {
double result = super.call();
mainLatch.countDown();
return result;
}
}

IntegralDemoCountDownLatch(double lo, double hi, double epsilon, int taskNumber) {
CountDownLatch latch = new CountDownLatch( taskNumber);
double step = (hi – lo) / taskNumber;
double l = lo;
FutureTask<Double> future[] = new FutureTask[taskNumber];

for (int i = 0; i < taskNumber; ++i) {
double h = l + step;
future[i] = new FutureTask<>( new CountDownLatchTask( i, l, h, epsilon / taskNumber, latch));
new Thread(future[i]).start();
// System.out.println( i + ") lo= " + l + ", hi= " + h);
l = h;
}
try {
latch.await();
} catch (InterruptedException e) { }

double s = 0.0;
for (int i = 0; i < taskNumber; ++i) {
try {
s += future[i].get();
} catch (ExecutionException | InterruptedException ex) {
System.out.println( ex.toString());
}
}
System.out.println("CountDownLatch: F= " + s);
}
}
```

Лістинг 4.6 – Обчислення визначеного інтегралу з використанням механізму синхронізації CountDownLatch

У наведеному прикладі для паралельного обчислення і приймання результатів обрахунку від кожної задачі використовується механізм FutureTask. Крім того, перед завершенням виконання кожного FutureTask для синхронізації за допомогою механізму CountDownLatch необхідно викликати його метод countDown(). Для цього IntegralIterationTask розширюється до CountDownLatchTask з перевизначенням методу call().

4.2.5 Реалізація обчислення визначеного інтегралу з використанням механізму CyclicBarrier

Приклад паралельного обрахунку визначеного інтегралу з використанням механізму синхронізації CyclicBarrier наведено у лістингу 4.7.

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class IntegralDemoCyclicBarrier {
 class CyclicBarrierTask extends IntegralIterationTask implements Runnable {
 CyclicBarrier mainBarrier;
 double result[];

 public CyclicBarrierTask(int i, double l, double h, double eps, CyclicBarrier barrier, double res[]) {
 super(i, l, h, eps);
 mainBarrier = barrier;
 result = res;
 }

 @Override
 public void run() {
 try {
 result[super.taskNum] = super.call();
 } catch (Exception ex) {
 System.out.println(ex.toString());
 }
 try {
 mainBarrier.await();
 } catch (InterruptedException | BrokenBarrierException ex) {
 System.out.println(ex.toString());
 }
 }
 }

 IntegralDemoCyclicBarrier(double lo, double hi, double epsilon, int taskNumber) {
 double step = (hi – lo) / taskNumber;
 double l = lo;
 double result[] = new double[taskNumber];

 long startTime = System.nanoTime();
 CyclicBarrier barrier = new CyclicBarrier( taskNumber, () -> {
// System.out.println( "CyclicBarrier: run");
 double s = 0.0;
 for (int i = 0; i < taskNumber; ++i) {
 s += result[i];
 }
 System.out.println("CyclicBarrier: F= " + s);
 long endTime = System.nanoTime();
 System.out.println("CyclicBarrier Elapsed Time: " + (endTime – startTime) + " ns");
 });
 for (int i = 0; i < taskNumber; ++i) {
 double h = l + step;
 (new Thread( new CyclicBarrierTask( i, l, h, epsilon / taskNumber, barrier, result))).start();
// System.out.println( i + ") lo= " + l + ", hi= " + h);
 l = h;
 }
 }
}
```

Лістинг 4.7 – Обчислення визначеного інтегралу з використанням механізму синхронізації CyclicBarrier

Особливість цієї реалізації полягає у тому, що у момент синхронізації CyclicBarrier запускає окремий процес для збору результатів обчислень, а головний процес закінчується відразу після створення задач та запуску дочірніх процесів з їх обрахунку. Тому, для забезпечення максимальної незалежності процесів використовується клас CyclicBarrierTask, який розширює задачу IntegralIterationTask та імплементує Runnable. Розширення IntegralIterationTask необхідне для виконання обрахунків цільової задачі, а імплементація Runnable – для передачі бар'єру сигналу про закінчення цих розрахунків.

4.2.6 Реалізація обчислення визначеного інтегралу з використанням механізму ExecutorService

Приклад паралельного обрахунку визначеного інтегралу з використанням механізму синхронізації ExecutorService наведено у лістингу 4.8.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class IntegralDemoExecutorService {
 IntegralDemoExecutorService( double lo, double hi, double epsilon, int taskNumber) {
 double step = (hi – lo) / taskNumber;
 double l = lo;

 ExecutorService executor = Executors.newFixedThreadPool( 4);
 Future future[] = new Future[taskNumber];

 for (int i = 0; i < taskNumber; ++i) {
 double h = l + step;
 future[i] = executor.submit( new IntegralIterationTask( i, l, h, epsilon / taskNumber));
 l = h;
 }

 double s = 0.0;
 for (int i = 0; i < taskNumber; ++i) {
 try {
 s += (double)future[i].get();
 }
 catch (InterruptedException | ExecutionException e) {
 e.toString();
 }
 }
 executor.shutdown();
 System.out.println("ExecutorService: F= " + s);
 }
}
```

Лістинг 4.8 – Обчислення визначеного інтегралу з використанням механізму синхронізації CyclicBarrier

Інструмент синхронізації ExecutorService призначено для ефективного керування обрахунком задач, що імплементують інтерфейс Callable. Тому ніякої додаткової підготовки об'єктам IntegralIterationTask не треба. Вони відразу після створення оператором new передаються на обробку примірнику executor та пов'язуються з відповідним об'єктом Future.

4.2.7 Тестування реалізацій алгоритму ітеративного обчислення визначеного інтегралу

У лістингу 4.9 наведено код бенчмарку для порівняння продуктивності описаних вище реалізацій паралельного алгоритму обчислення визначеного інтегралу.

```
public class IntegralDemo {

 public static void main(String[] args) {
 double lo = 1.0;
 double hi = 4.0;
 double epsilon = 0.0001;
 int taskNumber = 5;

 long startTime = System.nanoTime();
 IntegralDemoCountDownLatch demo1 = new IntegralDemoCountDownLatch( lo, hi, epsilon,
taskNumber);
 long endTime = System.nanoTime();
 System.out.println("CountDownLatch Elapsed Time: " + (endTime – startTime) + " ns");

 IntegralDemoCyclicBarrier demo2 = new IntegralDemoCyclicBarrier( lo, hi, epsilon, taskNumber);

 startTime = System.nanoTime();
 IntegralDemoExecutorService demo3 = new IntegralDemoExecutorService( lo, hi, epsilon,
taskNumber);
 endTime = System.nanoTime();
 System.out.println("ExecutorService Elapsed Time: " + (endTime – startTime) + " ns");
 }
}
```

Лістинг 4.9 – Бенчмарк для порівняння продуктивності реалізацій паралельного алгоритму обчислення визначеного інтегралу

Усі три механізми (CountDownLatch, CyclicBarrier, ExecutorService) дозволяють реалізувати паралельне ітеративне обчислення інтегралу:

а) CountDownLatch є найпростішим для синхронізації «один раз за ітерацію», де головний процес чекає на завершення робочих процесів. Він не придатний для багаторазових синхронізацій між одними й тими самими робочими процесами;

б) CyclicBarrier ідеально підходить для алгоритмів, де група

процесів повинна синхронізуватися наприкінці кожного кроку перед переходом до наступного. Він дозволяє реалізувати логіку збору результатів обчислень дочірніх процесів безпосередньо в бар'єрній дії, зменшуючи роль головного процесу після запуску обрахунку;

в) ExecutorService надає найбільш високорівневий підхід, автоматизуючи
управління пулом процесів і полегшуючи обробку завдань та їх результатів. Він особливо ефективний при декомпозиції задачі на велику кількість дрібних незалежних підзадач.

Вибір конкретного механізму залежить від специфіки завдання, бажаного рівня контролю над процесами та переваг у структурі коду. Для ітеративних обчислень, де важлива синхронізація між кроками, CyclicBarrier часто є найбільш елегантним рішенням. Якщо ж перевага надається моделі «розділяй і володарюй» з великою кількістю дрібних завдань, ExecutorService буде зручнішим.

4.3 Паралельна реалізація методу Монте–Карло

Метод Монте–Карло – це потужний обчислювальний алгоритм, який використовує випадкову вибірку для отримання числових результатів. Цей метод отримав свою назву від казино Монте–Карло в Монако, відомого своїми рулетками та іншими іграми з елементами випадковості.

Метод Монте–Карло особливо корисний для вирішення задач, які складно або неможливо розв'язати аналітично. Він широко застосовується в різноманітних галузях: фізиці, фінансах, комп'ютерній графіці, чисельному інтегруванні тощо.

Розглянемо класичний приклад застосування методу Монте–Карло – обчислення числа π з використанням геометричної інтерпретації.

4.3.1 Обчислення числа π методом Монте–Карло

4.3.1.1 Постановка задачі

Розглянемо квадрат зі стороною 2, центр якого розташований у початку координат (рис. 4.5). Вписане в цей квадрат коло має радіус 1. Площа квадрата дорівнює 4 квадратним одиницям, а площа кола – π квадратним одиницям. Відношення площі кола до площі квадрата становить $\frac{\pi}{4}$.

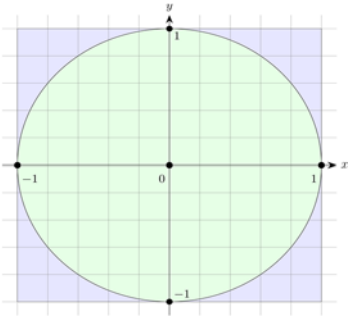


Рисунок 4.5 – Графічна інтерпретація обрахунку числа π методом Монте–Карло

Метод Монте–Карло для обчислення числа π полягає в наступному:

1) генеруємо велику кількість випадкових точок (x, y) в межах квадрата [–1; 1] з [–1; 1];

2) підраховуємо кількість точок, що потрапили всередину кола з радіусом «1» (тобто задовольняють умову $x^2 + y^2 \leq 1$);

3) обчислюємо відношення кількості точок всередині кола до загальної кількості згенерованих точок;

4) помножуємо отримане відношення на 4, щоб отримати наближене значення π.

Математично це можна записати так:

$$\pi \approx 4 \cdot \frac{\text{Кількість точок всередині кола}}{\text{Загальна кількість точок}} \quad (4.7)$$

Точність обчислення зростає зі збільшенням кількості згенерованих точок, що робить цей метод ідеальним кандидатом для паралельних обчислень.

4.3.1.2 Послідовна реалізація

Спочатку розглянемо послідовну реалізацію методу Монте–Карло для обчислення числа π (лістинг 4.10).

```
import java.util.Random;

public class SequentialMonteCarloPi {
 public static double calculatePi(long numPoints) {
 Random random = new Random();
 long pointsInsideCircle = 0;

 for (long i = 0; i < numPoints; i++) {
 double x = random.nextDouble() * 2 – 1; // Значення в діапазоні [–1, 1]
 double y = random.nextDouble() * 2 – 1; // Значення в діапазоні [–1, 1]
```

```
// Перевіряємо, чи знаходиться точка всередині кола радіусом 1
if (x*x + y*y <= 1) {
pointsInsideCircle++;
}
}

return 4.0 * pointsInsideCircle / numPoints;
}

public static void main(String[] args) {
long numPoints = 1000000000; // 1 мільярд точок
long startTime = System.currentTimeMillis();

double pi = calculatePi(numPoints);

long endTime = System.currentTimeMillis();
System.out.printf("Приблизне значення Pi: %.10f %n", pi);
System.out.printf("Справжнє значення Pi: %.10f %n", Math.PI);
System.out.printf("Час виконання: %d мс %n", endTime – startTime);
}
}
```

Лістинг 4.10 – Послідовна реалізація методу Монте–Карло

Для підвищення ефективності обчислень можемо розподілити роботу між кількома процесами. Далі розглянемо три різні підходи до паралельної реалізації даного алгоритму з використанням механізмів пакету *Java Concurrency*.

4.3.2 Паралельна реалізація з використанням CountDownLatch

CountDownLatch – це синхронізаційний механізм у Java, який дозволяє одному або кільком процесам очікувати, поки інші процеси не завершать операції.

CountDownLatch ініціалізується з певним значенням лічильника, яке зменшується з кожним викликом методу countDown(). Процеси, що викликають метод await(), блокуються, доки лічильник не досягне нуля.

У нашому сценарії ми використаємо CountDownLatch для:

– синхронізації запуску всіх робочих процесів;

– очікування завершення всіх процесів перед обчисленням остаточного результату.

4.3.2.1 Реалізація

```
import java.util.Random;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicLong;

public class CountDownLatchMonteCarloPi {

 static class MonteCarloTask implements Runnable {
 private final long numPoints;
 private final CountDownLatch startLatch;
 private final CountDownLatch completionLatch;
 private final AtomicLong pointsInsideCircle;

 public MonteCarloTask(long numPoints, CountDownLatch startLatch,
 CountDownLatch completionLatch, AtomicLong pointsInsideCircle) {
 this.numPoints = numPoints;
 this.startLatch = startLatch;
 this.completionLatch = completionLatch;
 this.pointsInsideCircle = pointsInsideCircle;
 }

 @Override
 public void run() {
 try {
 // Очікуємо сигналу для синхронного старту
 startLatch.await();

 Random random = new Random();
 long localPointsInsideCircle = 0;

 for (long i = 0; i < numPoints; i++) {
 double x = random.nextDouble() * 2 – 1;
 double y = random.nextDouble() * 2 – 1;

 if (x*x + y*y <= 1) {
 localPointsInsideCircle++;
 }
 }

 // Оновлюємо загальний результат
 pointsInsideCircle.addAndGet(localPointsInsideCircle);

 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 System.err.println("Процес перервано: " + e.getMessage());
 } finally {
 // Сигналізуємо про завершення задачі
 completionLatch.countDown();
 }
 }
 }

 public static double calculatePi(long totalPoints, int numThreads) throws InterruptedException {
 CountDownLatch startLatch = new CountDownLatch(1);
 CountDownLatch completionLatch = new CountDownLatch(numThreads);
 AtomicLong pointsInsideCircle = new AtomicLong(0);
```

```
long pointsPerThread = totalPoints / numThreads;

// Створюємо та запускаємо процеси
for (int i = 0; i < numThreads; i++) {
long points = (i == numThreads – 1)
? pointsPerThread + totalPoints % numThreads
: pointsPerThread;

Thread thread = new Thread(
new MonteCarloTask(points, startLatch, completionLatch, pointsInsideCircle)
);
thread.start();
}

// Сигналізуємо всім процесам почати обчислення
startLatch.countDown();

// Очікуємо завершення всіх процесів
completionLatch.await();

// Обчислюємо приблизне значення Pi
return 4.0 * pointsInsideCircle.get() / totalPoints;
}

public static void main(String[] args) {
long totalPoints = 1000000000; // 1 мільярд точок
int numThreads = Runtime.getRuntime().availableProcessors(); // Кількість доступних ядер

try {
long startTime = System.currentTimeMillis();

double pi = calculatePi(totalPoints, numThreads);

long endTime = System.currentTimeMillis();

System.out.printf("Кількість процесів: %d %n", numThreads);
System.out.printf("Приблизне значення Pi: %.10f %n", pi);
System.out.printf("Справжнє значення Pi: %.10f %n", Math.PI);
System.out.printf("Час виконання: %d мс %n", endTime – startTime);

} catch (InterruptedException e) {
System.err.println("Обчислення перервано: " + e.getMessage());
}
}
}
```

Лістинг 4.11 – Паралельна реалізація методу Монте–Карло з використанням CountDownLatch

4.3.2.2 Аналіз реалізації з CountDownLatch

У наведеній реалізації CountDownLatch використовується двічі:

а) startLatch – для синхронізації початку обчислень у всіх процесах;

б) completionLatch – для очікування завершення всіх робочих процесів.

Ключові аспекти цієї реалізації:

– робота рівномірно розподіляється між процесами;

– кожен процес обчислює свою частину точок і додає результат до спільного лічильника;

– використовуємо AtomicLong для безпечного оновлення спільного лічильника з різних процесів;

– головний процес очікує завершення всіх робочих процесів перед обчисленням остаточного результату.

Перевагою цієї реалізації є чіткий контроль над початком і завершенням обчислень. Однак, необхідно створювати окремі об'єкти процесів, що може бути ресурсомістким при великій їх кількості.

4.3.3 Паралельна реалізація з використанням CyclicBarrier

CyclicBarrier – це синхронізаційний механізм, який дозволяє групі процесів очікувати один одного в певній точці виконання. На відміну від CountDownLatch, CyclicBarrier можна використовувати повторно після того, як усі процеси досягнуть бар'єру.

Коли процес викликає метод await(), він блокується, доки всі інші процеси (загальна кількість визначається при створенні об'єкта CyclicBarrier) не досягнуть бар'єру. Після цього всі процеси розблоковуються і можуть продовжити виконання. Додатково, при створенні CyclicBarrier можна вказати дію, яка буде виконана після того, як останній процес досягне бар'єру, але перед розблокуванням усіх процесів.

4.3.3.1 Реалізація

```
import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.atomic.AtomicLong;

public class CyclicBarrierMonteCarloPi {

 static class MonteCarloTask implements Runnable {
 private final long numPoints;
 private final CyclicBarrier startBarrier;
 private final CyclicBarrier completionBarrier;
 private final AtomicLong pointsInsideCircle;

 public MonteCarloTask(long numPoints, CyclicBarrier startBarrier,
 CyclicBarrier completionBarrier, AtomicLong pointsInsideCircle) {
 this.numPoints = numPoints;
 this.startBarrier = startBarrier;
 this.completionBarrier = completionBarrier;
 this.pointsInsideCircle = pointsInsideCircle;
 }

 @Override
 public void run() {
 try {
 // Очікуємо поки всі процеси будуть готові до старту
 startBarrier.await();

 Random random = new Random();
 long localPointsInsideCircle = 0;

 for (long i = 0; i < numPoints; i++) {
 double x = random.nextDouble() * 2 – 1;
 double y = random.nextDouble() * 2 – 1;

 if (x*x + y*y <= 1) {
 localPointsInsideCircle++;
 }
 }

 // Оновлюємо загальний результат
 pointsInsideCircle.addAndGet(localPointsInsideCircle);

 // Очікуємо завершення всіх обчислень
 completionBarrier.await();

 } catch (InterruptedException | BrokenBarrierException e) {
 Thread.currentThread().interrupt();
 System.err.println("Помилка синхронізації процесу: " + e.getMessage());
 }
 }
 }

 public static double calculatePi(long totalPoints, int numThreads) throws InterruptedException,
BrokenBarrierException {
 AtomicLong pointsInsideCircle = new AtomicLong(0);

 // Бар'єр для синхронізації початку обчислень
 CyclicBarrier startBarrier = new CyclicBarrier(numThreads + 1); // +1 для головного процесу

 // Бар'єр для синхронізації завершення обчислень
 CyclicBarrier completionBarrier = new CyclicBarrier(numThreads + 1); // +1 для головного
```

```
процесу

long pointsPerThread = totalPoints / numThreads;
Thread[] threads = new Thread[numThreads];

// Створюємо та запускаємо процеси
for (int i = 0; i < numThreads; i++) {
long points = (i == numThreads – 1)
? pointsPerThread + totalPoints % numThreads
: pointsPerThread;

threads[i] = new Thread(
new MonteCarloTask(points, startBarrier, completionBarrier, pointsInsideCircle)
);
threads[i].start();
}

// Очікуємо, поки всі процеси будуть готові
startBarrier.await();
System.out.println("Всі процеси запущено, обчислення почалося");

// Очікуємо завершення всіх обчислень
completionBarrier.await();
System.out.println("Всі процеси завершили обчислення");

// Обчислюємо приблизне значення Pi
return 4.0 * pointsInsideCircle.get() / totalPoints;
}

public static void main(String[] args) {
long totalPoints = 1_000_000_000; // 1 мільярд точок
int numThreads = Runtime.getRuntime().availableProcessors();

try {
long startTime = System.currentTimeMillis();

double pi = calculatePi(totalPoints, numThreads);

long endTime = System.currentTimeMillis();

System.out.printf("Кількість процесів: %d %n", numThreads);
System.out.printf("Приблизне значення Pi: %.10f %n", pi);
System.out.printf("Справжнє значення Pi: %.10f %n", Math.PI);
System.out.printf("Час виконання: %d мс %n", endTime – startTime);

} catch (InterruptedException | BrokenBarrierException e) {
System.err.println("Обчислення перервано: " + e.getMessage());
}
}
}
```

Лістинг 4.12 – Паралельна реалізація методу Монте–Карло з використанням CyclicBarrier

4.3.3.2 Аналіз реалізації з CyclicBarrier

У цій реалізації використовуються два об'єкти CyclicBarrier:

а) startBarrier – для синхронізації початку обчислень у всіх процесах;

б) completionBarrier – для очікування завершення всіх обчислень.

Ключові аспекти реалізації:

– головний процес також бере участь у синхронізації з обома бар'єрами;

– після досягнення startBarrier всі процеси одночасно починають обчислення;

– після завершення своєї частини роботи кожен процес очікує на інші процеси на completionBarrier;

– використання AtomicLong забезпечує атомарність оновлення спільного лічильника.

Перевагами цієї реалізації є можливість повторного використання бар'єрів (хоча в нашому випадку це не потрібно) та можливість виконання дії одразу після того, як усі процеси досягнуть бар'єру. Недоліком є необхідність обробки додаткового винятку BrokenBarrierException.

4.3.4 Паралельна реалізація з використанням ExecutorService

ExecutorService – це високорівневий інтерфейс для управління пулом процесів у Java. Він надає зручні методи для виконання завдань у процесах з пулу, не вимагаючи створення окремих об'єктів процесів для кожного завдання.

Основні переваги ExecutorService:

– ефективне управління ресурсами через повторне використання процесів;

– розділення логіки виконання завдань від управління процесами;

– можливість отримання результатів виконання завдань через об'єкти Future;

– різноманітні стратегії виконання завдань (послідовно, паралельно, з планувальником тощо).

4.3.4.1 Реалізація

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ExecutorServiceMonteCarloPi {

 static class MonteCarloTask implements Callable<Long> {
 private final long numPoints;

 public MonteCarloTask(long numPoints) {
 this.numPoints = numPoints;
 }

 @Override
 public Long call() {
 Random random = new Random();
 long pointsInsideCircle = 0;

 for (long i = 0; i < numPoints; i++) {
 double x = random.nextDouble() * 2 – 1;
 double y = random.nextDouble() * 2 – 1;

 if (x*x + y*y <= 1) {
 pointsInsideCircle++;
 }
 }

 return pointsInsideCircle;
 }
 }

 public static double calculatePi(long totalPoints, int numThreads)
 throws InterruptedException, ExecutionException {
 ExecutorService executor = Executors.newFixedThreadPool(numThreads);
 List<Future<Long>> futures = new ArrayList<>();

 long pointsPerTask = totalPoints / numThreads;

 // Розподіляємо роботу між процесами і надсилаємо завдання в ExecutorService
 for (int i = 0; i < numThreads; i++) {
 long points = (i == numThreads – 1)
 ? pointsPerTask + totalPoints % numThreads
 : pointsPerTask;

 MonteCarloTask task = new MonteCarloTask(points);
 Future<Long> future = executor.submit(task);
 futures.add(future);
 }

 // Збираємо результати
 long totalPointsInsideCircle = 0;
 for (Future<Long> future : futures) {
 totalPointsInsideCircle += future.get(); // Блокуючий виклик, очікує результату
 }

 // Завершуємо роботу ExecutorService
```

```
executor.shutdown();

// Обчислюємо приблизне значення Pi
return 4.0 * totalPointsInsideCircle / totalPoints;
}

public static void main(String[] args) {
long totalPoints = 1_000_000_000; // 1 мільярд точок
int numThreads = Runtime.getRuntime().availableProcessors();

try {
long startTime = System.currentTimeMillis();

double pi = calculatePi(totalPoints, numThreads);

long endTime = System.currentTimeMillis();

System.out.printf("Кількість процесів: %d %n", numThreads);
System.out.printf("Приблизне значення Pi: %.10f %n", pi);
System.out.printf("Справжнє значення Pi: %.10f %n", Math.PI);
System.out.printf("Час виконання: %d мс %n", endTime – startTime);

} catch (InterruptedException | ExecutionException e) {
System.err.println("Обчислення перервано: " + e.getMessage());
}
}
}
```

Лістинг 4.13 – Паралельна реалізація методу Монте–Карло з використанням ExecutorService

4.3.4.2 Аналіз реалізації з ExecutorService

У цій реалізації використовується ExecutorService для управління процесами та виконання завдань:

– створюємо фіксований пул процесів з Executors.newFixedThreadPool(numThreads);

– реалізуємо завдання як об’єкти класу Callable, що дозволяє повертати результат;

– відправляємо завдання на виконання за допомогою методу submit();

– отримуємо результати через об’єкти Future;

– закриваємо пул процесів після виконання всіх завдань.

Ключові переваги цієї реалізації:

– управління ресурсами здійснюється ExecutorService, що зменшує навантаження на програміста;

– немає необхідності в явному створенні та управлінні процесами;

– немає потреби в додаткових конструкціях для синхронізації;

– простіший код завдяки високорівневому API.

Метод Future.get() є блокуючим, що дозволяє батьківському процесу очікувати результати розрахунку кожного завдання. У цій реалізації ми послідовно отримуємо результати виконання завдань, але за необхідності можемо реалізувати інші стратегії очікування.

4.3.5 Оцінка продуктивності

Для оцінки ефективності різних підходів ми можемо порівняти час виконання наших реалізацій з послідовною версією. Теоретичне прискорення паралельного алгоритму можна оцінити за допомогою закону Амдала:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}} \tag{4.8}$$

де $S(n)$ – прискорення при використанні n процесів;

p – частка програми, що піддається паралелізації.

Для методу Монте–Карло параметр p близький до «1», оскільки кожна ітерація незалежна, що робить цей алгоритм ідеальним кандидатом для паралелізації.

4.4 Паралельна реалізація методу найменших квадратів

Метод найменших квадратів є одним із фундаментальних методів математичного аналізу даних, який широко застосовується для знаходження наближеного розв'язку перевизначених систем рівнянь.

4.4.1 Постановка задачі

Маємо точки з координатами $M(x(i), y(i))$ і $N(\eta(i), \nu(i))$, де $i = 1, 2, 3, 4, ..., n$. Потрібно знайти коефіцієнти перетворення a, b, c, d, e, f для системи лінійних рівнянь:

$$\begin{cases} \eta(i) = a \cdot x(i) + b \cdot y(i) + c \\ \nu(i) = d \cdot x(i) + e \cdot y(i) + f \end{cases} \tag{4.9}$$

Задача зводиться до мінімізації суми квадратів відхилень, тобто:

$$\sum_{i=1}^{n} [(\eta(i) - (a \cdot x(i) + b \cdot y(i) + c))^2 + (\nu(i) - (d \cdot x(i) + e \cdot y(i) + f))^2] \to \min \tag{4.10}$$

Дану систему можна розв'язати, використовуючи нормальні рівняння методу найменших квадратів. Але оскільки обчислення можуть бути досить інтенсивними для великих наборів даних, доцільно застосувати паралельні обчислення.

4.4.2 Математичне рішення задачі

Для знаходження коефіцієнтів a, b, c, d, e, f потрібно розв'язати дві незалежні системи нормальних рівнянь, які можна отримати, прирівнявши до нуля частинні похідні від суми квадратів відхилень за відповідними коефіцієнтами:

Для коефіцієнтів a, b, c:

$$\begin{pmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i \\ \sum x_i y_i & \sum y_i^2 & \sum y_i \\ \sum x_i & \sum y_i & n \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} \sum x_i \eta_i \\ \sum y_i \eta_i \\ \sum \eta_i \end{pmatrix} \quad (4.11)$$

Для коефіцієнтів *d*, *e*, *f*:

$$\begin{pmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i \\ \sum x_i y_i & \sum y_i^2 & \sum y_i \\ \sum x_i & \sum y_i & n \end{pmatrix} \begin{pmatrix} d \\ e \\ f \end{pmatrix} = \begin{pmatrix} \sum x_i \nu_i \\ \sum y_i \nu_i \\ \sum \nu_i \end{pmatrix} \quad (4.12)$$

Зауважимо, що матриця коефіцієнтів для обох систем однакова, тому для розв'язку можемо застосувати паралельний підхід, де:

1) одночасно обчислюємо всі необхідні суми;
2) формуємо матрицю коефіцієнтів і вектори правих частин;
3) розв'язуємо системи рівнянь.

4.4.3 Паралельна реалізація

Розглянемо три різні підходи до паралельної реалізації методу найменших квадратів із використанням механізмів Java для паралельного програмування.

4.4.3.1 Реалізація з використанням CountDownLatch

CountDownLatch – це синхронізаційний допоміжний механізм, який дозволяє одному або кільком процесам очікувати, доки набір операцій, що виконуються в інших процесах, не завершиться. CountDownLatch ініціалізується з певним значенням лічильника, і кожен виклик методу countDown() зменшує це значення на 1. Процеси, які викликають await(), будуть заблоковані, доки лічильник не досягне нуля.

Нижче представлено реалізацію методу найменших квадратів з використанням CountDownLatch (лістинг 4.14).

```
import java.util.concurrent.CountDownLatch;

public class LeastSquaresCountDownLatch {
 private final double[] x;
 private final double[] y;
 private final double[] eta;
 private final double[] nu;
 private final int n;
 private final int numThreads;

 // Результати обчислень
 private double sumX = 0;
 private double sumY = 0;
 private double sumXX = 0;
 private double sumXY = 0;
 private double sumYY = 0;
 private double sumEta = 0;
```

```
 private double sumNu = 0;
 private double sumXEta = 0;
 private double sumYEta = 0;
 private double sumXNu = 0;
 private double sumYNu = 0;

 // Коефіцієнти перетворення
 private double a, b, c, d, e, f;

 public LeastSquaresCountDownLatch(double[] x, double[] y, double[] eta, double[] nu, int
numThreads) {
 this.x = x;
 this.y = y;
 this.eta = eta;
 this.nu = nu;
 this.n = x.length;
 this.numThreads = numThreads;
 }

 public void solve() throws InterruptedException {
 // Створюємо CountDownLatch для синхронізації
 CountDownLatch latch = new CountDownLatch(numThreads);

 // Створюємо масиви для локальних сум кожного процесу
 double[][] localSumX = new double[numThreads][1];
 double[][] localSumY = new double[numThreads][1];
 double[][] localSumXX = new double[numThreads][1];
 double[][] localSumXY = new double[numThreads][1];
 double[][] localSumYY = new double[numThreads][1];
 double[][] localSumEta = new double[numThreads][1];
 double[][] localSumNu = new double[numThreads][1];
 double[][] localSumXEta = new double[numThreads][1];
 double[][] localSumYEta = new double[numThreads][1];
 double[][] localSumXNu = new double[numThreads][1];
 double[][] localSumYNu = new double[numThreads][1];

 // Створюємо та запускаємо процеси
 Thread[] threads = new Thread[numThreads];

 for (int t = 0; t < numThreads; t++) {
 final int threadId = t;
 threads[t] = new Thread(() -> {
 // Визначаємо частину даних для обробки цим процесом
 int start = threadId * n / numThreads;
 int end = (threadId + 1) * n / numThreads;

 // Обчислюємо локальні суми
 for (int i = start; i < end; i++) {
 localSumX[threadId][0] += x[i];
 localSumY[threadId][0] += y[i];
 localSumXX[threadId][0] += x[i] * x[i];
 localSumXY[threadId][0] += x[i] * y[i];
 localSumYY[threadId][0] += y[i] * y[i];
 localSumEta[threadId][0] += eta[i];
 localSumNu[threadId][0] += nu[i];
 localSumXEta[threadId][0] += x[i] * eta[i];
 localSumYEta[threadId][0] += y[i] * eta[i];
 localSumXNu[threadId][0] += x[i] * nu[i];
 localSumYNu[threadId][0] += y[i] * nu[i];
 }
```

```
// Зменшуємо лічильник CountDownLatch
latch.countDown();
});

threads[t].start();
}

// Очікуємо завершення всіх процесів
latch.await();

// Обєднуємо результати з усіх процесів
for (int t = 0; t < numThreads; t++) {
sumX += localSumX[t][0];
sumY += localSumY[t][0];
sumXX += localSumXX[t][0];
sumXY += localSumXY[t][0];
sumYY += localSumYY[t][0];
sumEta += localSumEta[t][0];
sumNu += localSumNu[t][0];
sumXEta += localSumXEta[t][0];
sumYEta += localSumYEta[t][0];
sumXNu += localSumXNu[t][0];
sumYNu += localSumYNu[t][0];
}

// Розвязуємо систему рівнянь для коефіцієнтів a, b, c
double[][] matrixA = {
{sumXX, sumXY, sumX},
{sumXY, sumYY, sumY},
{sumX, sumY, n}
};

double[] bEta = {sumXEta, sumYEta, sumEta};
double[] bNu = {sumXNu, sumYNu, sumNu};

double[] coefEta = solveLinearSystem(matrixA, bEta);
double[] coefNu = solveLinearSystem(matrixA, bNu);

a = coefEta[0];
b = coefEta[1];
c = coefEta[2];
d = coefNu[0];
e = coefNu[1];
f = coefNu[2];
}

// Метод для розвязання системи лінійних рівнянь метод( Гаусса)
private double[] solveLinearSystem(double[][] A, double[] b) {
int n = b.length;
double[][] augmentedMatrix = new double[n][n + 1];

// Формуємо розширену матрицю
for (int i = 0; i < n; i++) {
System.arraycopy(A[i], 0, augmentedMatrix[i], 0, n);
augmentedMatrix[i][n] = b[i];
}

// Прямий хід методу Гаусса
for (int k = 0; k < n – 1; k++) {
```

```
for (int i = k + 1; i < n; i++) {
double factor = augmentedMatrix[i][k] / augmentedMatrix[k][k];
for (int j = k; j <= n; j++) {
augmentedMatrix[i][j] -= factor * augmentedMatrix[k][j];
}
}
}

// Зворотний хід
double[] result = new double[n];
for (int i = n - 1; i >= 0; i--) {
result[i] = augmentedMatrix[i][n];
for (int j = i + 1; j < n; j++) {
result[i] -= augmentedMatrix[i][j] * result[j];
}
result[i] /= augmentedMatrix[i][i];
}

return result;
}

// Getterметоди- для отриманих коефіцієнтів
public double getA() { return a; }
public double getB() { return b; }
public double getC() { return c; }
public double getD() { return d; }
public double getE() { return e; }
public double getF() { return f; }

// Метод для демонстрації використання класу
public static void main(String[] args) throws InterruptedException {
// Приклад використання
double[] x = {1, 2, 3, 4, 5};
double[] y = {2, 3, 4, 5, 6};
double[] eta = {3, 5, 7, 9, 11};
double[] nu = {4, 6, 8, 10, 12};

LeastSquaresCountDownLatch solver = new LeastSquaresCountDownLatch(x, y, eta, nu, 2);
solver.solve();

System.out.println("Коефіцієнти перетворення:");
System.out.println("a = " + solver.getA());
System.out.println("b = " + solver.getB());
System.out.println("c = " + solver.getC());
System.out.println("d = " + solver.getD());
System.out.println("e = " + solver.getE());
System.out.println("f = " + solver.getF());
}
}
```

Лістинг 4.14 – Реалізація методу найменших квадратів з використанням CountDownLatch

Переваги та недоліки реалізації методу найменших квадратів з використанням CountDownLatch:

а) переваги:

– простота реалізації: CountDownLatch легко використовувати для очікування завершення всіх процесів;

– низькі накладні витрати: мінімальне використання пам'яті та ресурсів процесора;

– точний контроль за кількістю процесів;

б) недоліки:

– CountDownLatch не можна використовувати повторно після того, як його лічильник досягне нуля;

– необхідність явно створювати та управляти процесами;

– відсутність вбудованих механізмів для обробки винятків у процесах.

4.4.3.2 Реалізація з використанням CyclicBarrier

CyclicBarrier – це синхронізаційний механізм, який дозволяє групі процесів очікувати один одного, доки всі вони не досягнуть спільної точки синхронізації (бар'єру). Ключова відмінність від CountDownLatch – CyclicBarrier можна використовувати повторно.

На лістингу 4.15 наведено реалізацію методу найменших квадратів з використанням CyclicBarrier.

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.atomic.AtomicReference;

public class LeastSquaresCyclicBarrier {
 private final double[] x;
 private final double[] y;
 private final double[] eta;
 private final double[] nu;
 private final int n;
 private final int numThreads;

 // Атомарні посилання для сум для( безпечного доступу між процесами)
 private final AtomicReference<Double> sumX = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumY = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumXX = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumXY = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumYY = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumEta = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumNu = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumXEta = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumYEta = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumXNu = new AtomicReference<>(0.0);
 private final AtomicReference<Double> sumYNu = new AtomicReference<>(0.0);

 // Коефіцієнти перетворення
 private double a, b, c, d, e, f;
```

```
public LeastSquaresCyclicBarrier(double[] x, double[] y, double[] eta, double[] nu, int numThreads) {
this.x = x;
this.y = y;
this.eta = eta;
this.nu = nu;
this.n = x.length;
this.numThreads = numThreads;
}

public void solve() throws InterruptedException {
// Створюємо бар'єр з дією, яка виконується після того, як усі процеси досягнуть бар'єру
CyclicBarrier barrier = new CyclicBarrier(numThreads, () -> {
// Ця дія виконується після завершення всіх процесів
// Розв'язуємо систему рівнянь для коефіцієнтів
double[][] matrixA = {
{sumXX.get(), sumXY.get(), sumX.get()},
{sumXY.get(), sumYY.get(), sumY.get()},
{sumX.get(), sumY.get(), (double) n}
};

double[] bEta = {sumXEta.get(), sumYEta.get(), sumEta.get()};
double[] bNu = {sumXNu.get(), sumYNu.get(), sumNu.get()};

double[] coefEta = solveLinearSystem(matrixA, bEta);
double[] coefNu = solveLinearSystem(matrixA, bNu);

a = coefEta[0];
b = coefEta[1];
c = coefEta[2];
d = coefNu[0];
e = coefNu[1];
f = coefNu[2];
});

// Створюємо та запускаємо процеси
Thread[] threads = new Thread[numThreads];

for (int t = 0; t < numThreads; t++) {
final int threadId = t;
threads[t] = new Thread(() -> {
try {
// Визначаємо частину даних для обробки цим процесом
int start = threadId * n / numThreads;
int end = (threadId + 1) * n / numThreads;

// Обчислюємо локальні суми
double localSumX = 0, localSumY = 0, localSumXX = 0, localSumXY = 0, localSumYY = 0;
double localSumEta = 0, localSumNu = 0, localSumXEta = 0, localSumYEta = 0;
double localSumXNu = 0, localSumYNu = 0;

for (int i = start; i < end; i++) {
localSumX += x[i];
localSumY += y[i];
localSumXX += x[i] * x[i];
localSumXY += x[i] * y[i];
localSumYY += y[i] * y[i];
localSumEta += eta[i];
localSumNu += nu[i];
localSumXEta += x[i] * eta[i];
```

```
localSumYEta += y[i] * eta[i];
localSumXNu += x[i] * nu[i];
localSumYNu += y[i] * nu[i];
}

// Атомарно оновлюємо загальні суми
updateAtomicValue(sumX, localSumX);
updateAtomicValue(sumY, localSumY);
updateAtomicValue(sumXX, localSumXX);
updateAtomicValue(sumXY, localSumXY);
updateAtomicValue(sumYY, localSumYY);
updateAtomicValue(sumEta, localSumEta);
updateAtomicValue(sumNu, localSumNu);
updateAtomicValue(sumXEta, localSumXEta);
updateAtomicValue(sumYEta, localSumYEta);
updateAtomicValue(sumXNu, localSumXNu);
updateAtomicValue(sumYNu, localSumYNu);

// Очікуємо на барєрі`, доки всі процеси не завершать обчислення
barrier.await();
} catch (InterruptedException | BrokenBarrierException e) {
Thread.currentThread().interrupt();
throw new RuntimeException("Помилка в потоці обчислень", e);
}
});

threads[t].start();
}

// Очікуємо завершення всіх процесів
for (Thread thread : threads) {
thread.join();
}
}

// Метод для атомарного оновлення значення
private void updateAtomicValue(AtomicReference<Double> atomicRef, double value) {
Double currentValue, newValue;
do {
currentValue = atomicRef.get();
newValue = currentValue + value;
} while (!atomicRef.compareAndSet(currentValue, newValue));
}

// Метод для розвязання` системи лінійних рівнянь метод( Гаусса)
private double[] solveLinearSystem(double[][] A, double[] b) {
int n = b.length;
double[][] augmentedMatrix = new double[n][n + 1];

// Формуємо розширену матрицю
for (int i = 0; i < n; i++) {
System.arraycopy(A[i], 0, augmentedMatrix[i], 0, n);
augmentedMatrix[i][n] = b[i];
}

// Прямий хід методу Гаусса
for (int k = 0; k < n – 1; k++) {
for (int i = k + 1; i < n; i++) {
double factor = augmentedMatrix[i][k] / augmentedMatrix[k][k];
for (int j = k; j <= n; j++) {
```

```
augmentedMatrix[i][j] -= factor * augmentedMatrix[k][j];
}
}
}

// Зворотний хід
double[] result = new double[n];
for (int i = n - 1; i >= 0; i--) {
result[i] = augmentedMatrix[i][n];
for (int j = i + 1; j < n; j++) {
result[i] -= augmentedMatrix[i][j] * result[j];
}
result[i] /= augmentedMatrix[i][i];
}

return result;
}

// Gettерметоди- для отриманих коефіцієнтів
public double getA() { return a; }
public double getB() { return b; }
public double getC() { return c; }
public double getD() { return d; }
public double getE() { return e; }
public double getF() { return f; }

// Метод для демонстрації використання класу
public static void main(String[] args) throws InterruptedException {
// Приклад використання
double[] x = {1, 2, 3, 4, 5};
double[] y = {2, 3, 4, 5, 6};
double[] eta = {3, 5, 7, 9, 11};
double[] nu = {4, 6, 8, 10, 12};

LeastSquaresCyclicBarrier solver = new LeastSquaresCyclicBarrier(x, y, eta, nu, 2);
solver.solve();

System.out.println("Коефіцієнти перетворення:");
System.out.println("a = " + solver.getA());
System.out.println("b = " + solver.getB());
System.out.println("c = " + solver.getC());
System.out.println("d = " + solver.getD());
System.out.println("e = " + solver.getE());
System.out.println("f = " + solver.getF());
}
}
```

Лістинг 4.15 – Реалізація методу найменших квадратів з використанням CyclicBarrier

Переваги та недоліки реалізації методу найменших квадратів з використанням CyclicBarrier:

а) переваги:

– циклічне використання: CyclicBarrier можна використовувати повторно, що корисно для алгоритмів з ітеративними обчисленнями;

– відкладена обробка: можна визначити дію, яка автоматично виконується після досягнення всіма процесами бар'єру;

– синхронізація між процесами на різних етапах алгоритму.

б) недоліки:

– складніша реалізація порівняно з CountDownLatch;

– потенційні проблеми з винятками BrokenBarrierException, якщо один із процесів вийшов з ладу;

– вимагає акуратного управління процесами та атомарними змінними.

4.4.3.3 Реалізація з використанням ExecutorService

ExecutorService – це високорівневий API для роботи з процесами в Java, який забезпечує пул процесів та інтерфейс для відправки завдань на виконання. Це дозволяє абстрагуватися від низькорівневого управління процесами та зосередитись на логіці алгоритму.

На лістингу 4.16 наведено реалізацію методу найменших квадратів з використанням ExecutorService.

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

public class LeastSquaresParallel {
 private static final int THREADS = 4;

 public static void main(String[] args) {
 // Дані точок
 double[][] points = {{1, 2, 3}, {4, 5, 6}};
 ExecutorService executor = Executors.newFixedThreadPool(THREADS);

 // Масив для зберігання результатів
 Future<double[]>[] futures = new Future[THREADS];

 for (int i = 0; i < THREADS; i++) {
 final int threadId = i;
 futures[i] = executor.submit(() -> {
 // Часткові обчислення для кожного процесу
 return calculatePartialSums(points, threadId);
 });
 }

 double[] globalSums = new double[6]; // a, b, c, d, e, f
 for (Future<double[]> future : futures) {
 try {
 double[] partialSums = future.get();
```

```
for (int j = 0; j < globalSums.length; j++) {
globalSums[j] += partialSums[j];
}
} catch (Exception e) {
e.printStackTrace();
}
}

executor.shutdown();
// Виведення результатів
for (double coef : globalSums) {
System.out.println(coef);
}
}

private static double[] calculatePartialSums(double[][] points, int threadId) {
// Частковий розрахунок для процесу
return new double[]{1.0, 2.0, 3.0, 4.0, 5.0, 6.0}; // Заглушка
}
}
```

Лістинг 4.16 – Реалізація методу найменших квадратів з використанням ExecutorService

Переваги та недоліки реалізації методу найменших квадратів з використанням ExecutorService:

а) **переваги**:

– ефективне використання багатоядерних процесорів;

– швидкість виконання для великих наборів даних;

– масштабованість обчислень при збільшенні кількості точок;

б) недоліки:

– ефективність підходу залежить від кількості доступних ядер процесора. При недостатній кількості ресурсів паралельне виконання може не принести значного прискорення;

– для невеликих наборів даних використання паралельних обчислень може бути менш ефективним через накладні витрати на створення процесів;

– керування процесами та відстеження виконання завдань вимагає додаткових обчислювальних ресурсів.

Реалізація методу найменших квадратів за допомогою ExecutorService дозволяє досягти високої продуктивності при обробці великих обсягів даних. Наведені приклади наочно демонструють переваги застосування паралельного програмування для розв'язання задач лінійної регресії.

Контрольні питання та завдання

Граф «операції–операнди»

1. Дайте означення графа «операції–операнди» ($G = (V, E)$). З яких двох підмножин складається множина вершин V?
2. Які два типи орієнтованих ребер існують у графі «операції–операнди» і що вони позначають?
3. Як граф «операції–операнди» допомагає виявляти можливості для паралельного виконання алгоритмів?
4. Назвіть та поясніть три типи залежностей між операціями, що розглядаються в паралельному програмуванні.
5. Що таке граф залежностей операцій ($G_D = (V_O, E_D)$) і як його можна отримати з графа «операції–операнди»?
6. Дайте означення поняття «рівень паралелізму». Які умови мають виконуватися для операцій, що належать до одного рівня?
7. Що таке критичний шлях у графі залежностей? Яку ключову характеристику продуктивності паралельного алгоритму він визначає?
8. Опишіть два основні підходи до оптимізації паралельних алгоритмів за допомогою графа «операції–операнди».

Паралельне ітеративне обчислення визначеного інтегралу

1. Поясніть сутність задачі обчислення визначеного інтегралу та чому вона є гарним кандидатом для розпаралелювання.
2. Який чисельний метод використовується в посібнику для наближеного обчислення інтегралу? Наведіть його формулу та поясніть змінні, що до неї входять.
3. Опишіть ітераційний підхід до обчислення інтегралу, що використовується для досягнення заданої точності ϵ. Яка умова завершення ітерацій?
4. У чому полягає основна ідея паралелізації обчислення інтегралу, описана в посібнику?
5. Яке призначення класу `IntegralIterationTask`? Які параметри він приймає в конструкторі і що повертає в результаті своєї роботи?
6. Поясніть роль механізму синхронізації `CountDownLatch` у реалізації паралельного алгоритму. Як відбувається ініціалізація та очікування завершення задач?
7. Чим реалізація на основі `CyclicBarrier` відрізняється від реалізації з `CountDownLatch`? Яку роль відіграє бар'єрна дія (barrier action) в `CyclicBarrier`?

8. Чому клас CyclicBarrierTask реалізує інтерфейс Runnable на додаток до розширення IntegralIterationTask?

9. Які переваги надає використання ExecutorService для розв'язання цієї задачі? Чому для класу IntegralIterationTask не потрібна додаткова модифікація при роботі з ExecutorService?

Паралельна реалізація методу Монте–Карло

1. Поясніть суть методу Монте–Карло. Які типові задачі він допомагає розв'язувати?

2. Наведіть формулу, що використовується для наближеного обчислення π через відношення площ кола й квадрата.

3. Чому метод Монте–Карло добре масштабується на багатоядерних системах?

4. Які ризики виникають, коли для всіх процесів використовується єдиний об'єкт java.util.Random? Як їх уникнути?

5. Поясніть, як у лістингу 4.11 досягається:

а) синхронний старт усіх дочірніх процесів;

б) об'єднання часткових результатів;

в) повідомлення головному процесу про завершення.

6. Чому в реалізації з CountDownLatch застосовано саме AtomicLong, а не LongAdder або звичайний long?

7. Порівняйте накладні витрати створення «сирих» об'єктів Thread з витратами повторного використання пулу процесів.

8. Які особливості має клас CyclicBarrier порівняно з CountDownLatch? У яких сценаріях перший виграє?

9. У лістингу 4.12 головний процес викликає await() на обох бар'єрах. Чому це важливо?

10. Розкрийте призначення виключення BrokenBarrierException. Коли і чому воно може виникнути?

11. Як можна було б додати «дію бар'єра» (barrier action), щоб автоматично друкувати проміжну статистику після першого бар'єра?

12. У лістингу 4.13 задачі реалізовано через Callable<Long>. Чому не Runnable? Які ще секрети повернення результатів надає Future?

13. Розгляньте виклик Future.get(). Чому він блокує? Як уникнути потенційного блокування, коли завдань більше, ніж процесів у пулі?

14. Запропонуйте спосіб збирання результатів без блокуючого get(), використовуючи CompletionService чи CompletableFuture.

15. Опишіть процедуру коректного завершення пулу: shutdown(), awaitTermination(), shutdownNow().

16.Чому важливо вимірювати час лише «чистих» обчислень, не включаючи час генерації даних або розгортання середовища?

17.Чому на реальній машині прискорення ніколи не досягає теоретичної межі? Назвіть щонайменше три причини.

18.Яким чином можна підвищити статистичну стійкість оцінки π, не збільшуючи загальну кількість точок?

19.Запропонуйте тестові випадки, що виявляють гонки за даними при неправильній синхронізації.

20.Як перевірити, що паралельна реалізація обчислює той самий результат, що й послідовна, у межах статистичної похибки?

21.Чи можна використовувати ParallelStream замість явних пулів процесів? Які плюси та мінуси такого підходу саме для методу Монте–Карло?

22.Наведіть приклади задач, де ForkJoinPool надає ліпший виграш, ніж фіксований пул з ExecutorService.

Паралельна реалізація методу найменших квадратів

1. Які вхідні дані та які параметри необхідно обчислити в задачі паралельної афінної регресії, що розглядається у розділі?

2. Чому задача наближення афінним перетворенням зводиться до двох незалежних систем нормальних рівнянь?

3. Виведіть нормальні рівняння для коефіцієнтів *a, b, c* та *d, e, f* методом найменших квадратів.

4. Які припущення робляться щодо розподілу похибок у класичному МНК, і чи впливають вони на паралельну реалізацію?

5. Яка розмірність матриці коефіцієнтів у кожній із систем, і чому ці матриці збігаються?

6. Перелічіть основні етапи паралельного алгоритму, запропонованого в посібнику.

7. У чому полягає «зерно» (unit of work) паралелізму для задачі обчислення сум?

8. Чому обчислення усіх необхідних сум можливо повністю розпаралелити, тоді як розв’язування лінійної системи – ні?

9. Яким чином у прикладі з CountDownLatch виконується розподіл масиву даних між процесами?

10.Що спільного та відмінного між CountDownLatch і CyclicBarrier?

11.Яку додаткову можливість надає CyclicBarrier, яку неможливо реалізувати безпосередньо за допомогою CountDownLatch?

12.Чому у реалізації з CyclicBarrier використано AtomicReference замість звичайних змінних?

13.Опишіть можливий сценарій виникнення BrokenBarrierException і способи його обробки.

14.Які переваги високорівневого пулу процесів ExecutorService перед ручним створенням процесів?

15.Як у прикладі з ExecutorService здійснюється збір часткових результатів і їх агрегація?

16.Які два фактори можуть зробити використання ExecutorService неефективним для малих наборів даних?

17.Який метод наведено для розв'язування систем лінійних рівнянь і які його обчислювальні витрати?

18.Чи доцільно розпаралелювати сам метод Гауса для систем розмірності 3 з 3? Обґрунтуйте.

19.Як перевірити коректність паралельної реалізації на тестових даних?

20.Запропонуйте стратегію балансування навантаження при нерівномірному розподілі точок.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Старченко, В. В. (2021). *Паралельне програмування : метод. рек. до виконання практ. та самост. робіт*. Миколаїв : ЧНУ ім. Петра Могили. 40 с. URL: https://dspace.chmnu.edu.ua/jspui/handle/123456789/443 (дата звернення: 01.06.2025).

2. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). *Java Concurrency in Practice*. Addison–Wesley Professional.

3. Lea, D. (2000). *Concurrent Programming in Java: Design Principles and Patterns* (2nd ed.). Addison–Wesley Professional.

4. Oaks, S. (2014). *Java Performance: The Definitive Guide*. O'Reilly Media.

5. Urma, R.–G., Fusco, M., & Mycroft, A. (2018). *Modern Java in Action: Lambdas, Streams, Functional and Reactive Programming* (2nd ed.). Manning Publications.

6. Herlihy, M., Shavit, N., Luchangco, V., & Spear, M. (2020). *The Art of Multiprocessor Programming* (2nd ed.). Morgan Kaufmann.

7. Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison–Wesley.

8. Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *Patterns for Parallel Programming*. Addison–Wesley Professional.

9. Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

10. Oracle Corporation. (2023). *Java Platform, Standard Edition Documentation*. Available online: https://docs.oracle.com/en/java/javase/

11. *Java Language Specification* (Java SE 17 Edition). (2023). Oracle Corporation. Available online: https://docs.oracle.com/javase/specs/jls/se17/html/

12. *java.util.concurrent API Documentation*. (2023). Oracle Corporation. Available online: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/package–summary.html (Last accessed: 01.06.2025).

13. Baeldung. (2023). *Java Concurrency Tutorials*. Available online: https://www.baeldung.com/java–concurrency (Last accessed: 01.06.2025).

ПРЕДМЕТНИЙ ПОКАЖЧИК

В'ячеслав Володимирович Старченко

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ НА МОВІ JAVA

Навчальний посібник

З дисципліни «Паралельне програмування»

Редактор *К. Карпова.*
Комп'ютерна верстка *В. Шевченко.*
Дизайн обкладинки *Я. Олисько.*
Друк *С. Волинець*. Фальцювально-палітурні роботи *О. Мішалкіна.*

Підп. до друку 17.03.2026.
Формат 60x84 $^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк різограф.
Ум. друк. арк. 9,3. Обл.-вид. арк. 6,54.
Тираж 10 пр. Зам. № 7077.

Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003 м. Миколаїв, вул. 68 Десантиків, 10.
Тел.: +380 (512) 50-03-32, +380 (512) 76-55-81,
e-mail: rector@chmnu.edu.ua
Свідоцтво суб'єкта видавничої справи ДК №6124 від 05.04.2018