Методичні вказівки
до виконання самостійної роботи з дисципліни «Програмне забезпечення розподілених інформаційних систем та паралельних обчислень» зі спеціальності «Інженерія програмного забезпечення»

Міністерство освіти і науки України
Вінницький національний технічний університет

Методичні вказівки до виконання самостійної роботи з дисципліни «Програмне забезпечення розподілених інформаційних систем та паралельних обчислень» зі спеціальності «Інженерія програмного забезпечення»

Вінниця
ВНТУ
2025

Рекомендовано до видання Радою з якості освіти Вінницького національного технічного університету Міністерства освіти і науки України (протокол № від 2025 р.)

Рецензенти:
О. К. Колесницький, кандидат технічних наук, професор
О. В. Войцеховська, кандидат технічних наук, доцент

Методичні вказівки до виконання самостійної роботи з дисципліни «Програмне забезпечення розподілених інформаційних систем та паралельних обчислень» зі спеціальності «Інженерія програмного забезпечення». [Електронний ресурс] / уклад. О.М. Хошаба, В.П.Майданюк. – Вінниця : ВНТУ, 2025. – 51 с.

У методичних вказівках наведено основні теоретичні дані до виконання самостійної роботи з дисципліни «Програмне забезпечення розподілених інформаційних систем та паралельних обчислень». Методичні вказівки розроблено відповідно до навчальної програми дисципліни і орієнтовано на студентів магістерського рівня й аспірантів спеціальності «Інженерія програмного забезпечення».

ЗМІСТ

ВСТУП

Сучасна інформатика і прикладні галузі знань дедалі більше орієнтуються на використання розподілених та паралельних обчислень. Зростання обсягів даних, необхідність забезпечення високої доступності інформаційних сервісів, потреба у масштабуванні обчислювальних ресурсів і гарантуванні стійкості до відмов зумовлюють переосмислення підходів до побудови програмного забезпечення. Якщо ще кілька десятиліть тому домінували централізовані системи з одним потужним сервером або мейнфреймом, то сьогодні основу інфраструктури складають десятки, сотні й навіть тисячі вузлів, об’єднаних у єдину систему, здатну динамічно перерозподіляти ресурси. У цих умовах зростає роль дисциплін, спрямованих на формування у здобувачів вищої освіти знань про архітектури, принципи й інструменти створення розподілених систем та паралельних обчислювальних середовищ. Традиційні централізовані підходи поступаються місцем розподіленим і паралельним архітектурам, здатним забезпечити масштабованість, відмовостійкість і високу продуктивність. У цих умовах підготовка фахівців, які володіють як фундаментальними знаннями про природу розподілених процесів, так і практичними навичками роботи з сучасними інструментами, набуває особливої актуальності.

Методичні вказівки до виконання самостійної роботи з дисципліни «Програмне забезпечення розподілених інформаційних систем та паралельних обчислень» розроблені з метою надати здобувачам вищої освіти цілісне уявлення про архітектурні моделі, алгоритмічні основи й технологічні платформи, які формують сучасний ландшафт обчислювальних систем. У них поєднано класичні теоретичні положення (моделі паралелізму, алгоритми консенсусу, теореми CAP і PACELC) з аналізом найактуальніших практик (контейнеризація, оркестрація, потокова обробка даних, GPU-кластери).

Зміст методичних вказівок структуровано за темами, що дозволяє поступово формувати у здобувачів як фундаментальні знання, так і практичні компетентності. Особлива увага приділяється балансуванню між академічними моделями й реальними індустріальними технологіями - від OpenMP, MPI та CUDA до Docker, Kubernetes, Apache Kafka й Spark.

ТЕМА 1. ВСТУП ДО РОЗПОДІЛЕНИХ І ПАРАЛЕЛЬНИХ СИСТЕМ

Мета вивчення теми: Сформувати цілісне розуміння еволюції розподілених і паралельних систем - від мейнфреймів та кластерів до хмар, edge/fog і блокчейну - та закласти базу для інженерних рішень, що враховують ключові обмеження: затримки, узгодженість і відмовостійкість. Навчити студентів застосовувати теореми CAP і PACELC для обґрунтованого вибору архітектур і технологій.

1.1 Завдання до самостійної роботи

1. Простежити основні етапи розвитку: мейнфрейми → SMP/MPP → кластери → grid → хмари → edge/fog → p2p/блокчейн.
2. Розмежувати поняття паралельних (спільна/роздільна пам’ять) і розподілених систем; окреслити типові сценарії застосування кожних.
3. Ввести й уніфікувати метрики: затримка/пропускна здатність, доступність (availability), узгодженість (consistency), толерантність до розділення мережі (partition tolerance), MTBF/MTTR, SLO/SLA (на базовому рівні).
4. Розібрати природу затримок у мережі та пам’яті; скласти «latency-бюджет» для простої служби (теоретично).
5. Класифікувати типи відмов (crash, мережеві, часткові) та їхній вплив на дизайн.
6. Формально опанувати CAP: що означають C/A/P, чому всі три одночасно недосяжні за наявності розділення, як це відбивається на виборі БД/сховищ.
7. Опанувати PACELC: компроміс PA/C за наявності розділення та EL/C за його відсутності; навчитися читати ці компроміси в архітектурних рішеннях.
8. Навчитися зіставляти вимоги продукту (затримки, доступність, узгодженість) з класами систем (CA/CP/AP; EL/EC) та робити аргументований попередній вибір.
9. Розглянути приклади систем із різними пріоритетами (узгодженість-перш, доступність-перш, затримка-перш) без прив’язки до конкретних вендорів.
10. Підготувати короткий конспект з визначеннями і типовими компромісами для подальших тем курсу.

1.2 Короткі теоретичні відомості

Еволюція обчислювальних систем починається з мейнфреймів і монолітних застосунків, де вся логіка та дані зосереджені в одному

обчислювальному центрі й масштабування відбувається переважно «вгору» - додаванням потужніших процесорів і пам’яті. Подальший розвиток привів до SMP/MPP-архітектур і кластерів з багатьох дешевших вузлів, об’єднаних швидкими мережами; згодом з’явилися grid-підходи з федерацією розрізнених ресурсів і високою гетерогенністю. Хмарні обчислення додали еластичність, автоматизацію та моделі IaaS/PaaS/SaaS, а edge/fog-парадигми повернули частину обчислень «ближче до джерела даних», щоб зменшити затримки і навантаження на магістральні канали. Окремою віхою стали p2p-мережі та блокчейн. Це приклад децентралізованих систем без єдиного довіреного координатора, де незмінний журнал подій і протоколи консенсусу забезпечують спільний стан між рівнозначними вузлами.

Паралельні та розподілені системи вирішують різні, хоча й частково перетинні завдання. Паралельні обчислення - це насамперед одночасне виконання багатьох операцій у межах одного вузла зі спільною пам’яттю (потоки, OpenMP) або в тісно зв’язаних вузлах із роздільною пам’яттю та швидким інтерконектом (процеси, MPI), а також на прискорювачах GPGPU (CUDA/OpenCL).

Розподілені системи - це сукупність незалежних вузлів, що взаємодіють мережею й повинні працювати коректно попри часткові відмови й непередбачувані затримки. На практиці сучасні платформи поєднують обидва світи: наприклад, аналітика великих даних на Spark використовує міжвузловий розподіл і внутрішньовузловий паралелізм.

Ключовими величинами, що визначають поведінку таких систем, є затримка (час відповіді одиничної операції) і пропускна здатність (кількість оброблених операцій за секунду). Важливо спостерігати не лише середні значення, а й «хвости» розподілу - p95/p99, у яких ховається досвід користувача.

Доступність системи зручно оцінювати співвідношенням:

$$A \approx MTBF / (MTBF + MTTR),$$

де MTBF - середній час між відмовами,

MTTR - середній час відновлення.

Для паралельних обчислень важливо використання Закону Амдала (граничне прискорення обмежене непаралелізованою часткою програми) та Закону Густафсона (зростання задачі дозволяє краще використовувати більшу кількість процесорів). У розподіленому середовищі кожний додатковий мережевий стрибок, кворум, шифрування або колекція сміття додають мілісекунди, які множаться у складених шляхах запиту.

Моделі відмов формують вимоги до проєктування. Crash-stop і crash-recovery описують зникнення або перезапуск вузла; проти них працюють журнали змін і реплікація. Помилки пропуску (omission) та таймаути підказують необхідність повторних спроб із експоненційним backoff і ідемпотентних операцій. Розділення мережі (partition) - випадок, коли підмножини вузлів не бачать одна одну - найкритичніше для протоколів

узгодженості.

У деяких сценаріях можливі навіть «візантійські» збої, коли вузли поводяться довільно або зловмисно, що потребує BFT-протоколів. Техніки підвищення стійкості включають реплікацію (leader/follower, multi-leader), кворуми читання/запису, дедуплікацію, а для складних бізнес-процесів - саги замість глобальних розподілених транзакцій.

Під «узгодженістю» маємо на увазі спостережувану семантику доступу до спільних даних за наявності паралелізму та реплікації. Лінійризовність гарантує, що кожна операція виглядає миттєвою у деякий момент між викликом і відповіддю, надаючи «найсильнішу» модель для об'єктів. Серіалізовність забезпечує еквівалентність до деякого послідовного порядку транзакцій. Послідовна узгодженість слабша за лінійризовність, але зберігає єдиний порядок на рівні програми. Зрештою узгоджені системи (eventual consistency) допускають тимчасові розбіжності між репліками, але гарантують збіжність; практичні покращення тут - «read-your-writes», монотонні читання, каузальна узгодженість або обмежена застарілість.

Теорема CAP формалізує базовий компроміс для реплікованих систем під час розділення мережі: неможливо одночасно гарантувати сувору узгодженість (C) та доступність (A), якщо приймаємо, що розділення (P) може статися. У реальному світі P - не опція, а даність, тому вибір робиться між A і C саме в момент P: у класі CP ми жертвуємо доступністю, блокуючи або відхиляючи частину запитів, зате зберігаємо строгість даних; у класі AP ми дозволяємо вузлам відповідати автономно, мирячись із тимчасовими розбіжностями, що пізніше усуваються. Комбінація CA має сенс лише за відсутності P, тобто у моделях, далеких від практики георозподілених систем.

Принцип PACELC доповнює CAP і нагадує, що компроміси існують не лише під час розділення.

Якщо розділення сталося (P), ми так само обираємо між доступністю (A) та узгодженістю (C), але коли його немає (Else), все одно стоїть вибір між затримкою (L) та узгодженістю (C). Звідси типові профілі систем:

PA/EL - за розділення пріоритет доступності, у «спокійний час» пріоритет низьких затримок (зазвичай із слабшими гарантіями читання);

PC/EC - за розділення пріоритет строгій узгодженості, а за нормальних умов - також пріоритет узгодженості, навіть якщо це коштує додаткових мілісекунд на координацію.

Практичне читання CAP/PACELC починається з вимог домену: фінансові транзакції, резервування та інвентар зазвичай тяжіють до C-перш підходів (PC/EC або CP), стрічки подій, лічильники та кеші - до A-перш (PA/EL або AP). Географічна реплікація підсилює ціну координації, отже сильні гарантії часто означають готовність прийняти більші затримки. Вибір між синхронною та асинхронною реплікацією напряму перемикає важелі: синхронність підвищує узгодженість і хвости затримок,

асинхронність покращує доступність і latency за ціною тимчасової неузгодженості. Існують також «тоновані» механізми: налаштовувані кворуми, кеші з TTL, read-repair, outbox/inbox-патерни, ідемпотентні команди та саги, що дозволяють тонко підлаштовувати компроміси під конкретний сценарій.

Нарешті, зв'язок із паралельними обчисленнями створює міст до наступних тем курсу. Внутрішньовузлова паралельність (SIMD/SIMT, багатоядерність, GPU) підвищує пропускну здатність, але не усуває мережевих затримок і проблем консенсусу; міжвузлова паралельність (MPI, розподілені фреймворки на кшталт Spark) поєднує обмін повідомленнями з реплікацією та кворумами, де обмеження CAP/PACELC проявляються під час shuffle-стадій, чекпойнтів і потокової обробки.

Тому дизайн сучасних систем майже завжди є поєднанням: ми вибудовуємо потрібний рівень узгодженості там, де ціна помилки неприйнятна, і приймаємо зрештою узгоджені механізми там, де головує затримка й доступність, використовуючи паралелізм для продуктивності, а розподілені протоколи - для надійності.

1.3 Контрольні питання

1. Чим відрізняються паралельні обчислення від розподілених систем? Наведіть по одному прикладу для кожного підходу.
2. Які ключові етапи еволюції: мейнфрейми → SMP/MPP → кластери → grid → хмари → edge/fog → p2p/блокчейн? Що було драйвером переходів?
3. У чому різниця між клієнт-серверною, багаторівневою та p2p-архітектурами?
4. Що таке блокчейн у контексті p2p-систем? Яку роль відіграє консенсус?
5. Поясніть відмінність між latency і throughput. Чому важливі показники p95/p99?
6. Які джерела додають затримку до запиту в розподіленій системі (мережа, кворуми, шифрування, GC тощо)?
7. Сформулюйте та поясніть показники MTBF і MTTR. Як з них оцінити доступність системи?
8. Чим відрізняються моделі відмов: crash-stop, crash-recovery, omission/timeout і візантійська?
9. Які техніки підвищують відмовостійкість: реплікація, кворуми R/W, ідемпотентність, повторні спроби з backoff, дедуплікація?
10. Як синхронна та асинхронна реплікації впливають на затримку й узгодженість?
11. Поясніть: лінеаризовність, серіалізовність і послідовна узгодженість. У чому різниця між ними?
12. Що таке eventual consistency? Які існують «послаблення» читань: read-your-writes, monotonic reads, causal consistency, bounded staleness?

13. Чим «узгодженість» у CAP відрізняється від «C» в ACID-транзакціях?
14. Сформулюйте теорему CAP. Які саме властивості маються на увазі під C, A та P?
15. Чому у практичних системах P вважається даністю? Коли й чому доводиться обирати між C та A?
16. Наведіть приклад сценарію, де виправданий вибір CP, і приклад, де виправданий AP.
17. Чому комбінація CA зазвичай нереалістична у реальному світі?
18. Сформулюйте принцип PACELC. Що означають пари PA/EL і PC/EC?
19. Наведіть приклад системного рішення, що тяжіє до PA/EL, і приклад для PC/EC.
20. Сформулюйте закон Амдала. Як він обмежує прискорення при збільшенні кількості процесорів?
21. Як закон Густафсона інтерпретує масштабування задачі? Чим його інтуїція відрізняється від Амдала?
22. Які ролі відіграють OpenMP, MPI та CUDA у паралельних обчисленнях? Де кожен підхід доречніший?
23. Обчисліть доступність A, якщо MTBF = 2000 год, MTTR = 2 год. Як зміниться A, якщо зменшити MTTR удвічі?
24. Для програми з паралелізованою часткою p = 0,9 на N = 8 ядрах знайдіть теоретичне прискорення за Амдалом. Як зміниться результат при N = 32?
25. Сервіс глобально реплікований у двох регіонах. Які наслідки для затримки й узгодженості матиме перехід від асинхронної до синхронної реплікації? Як це «читається» через PACELC?
26. Уявіть платіжну систему та стрічку новин. Для кожної оберіть профіль CAP/PACELC і коротко обґрунтуйте.
27. Під час мережевого розділення частина запитів «зависає». Яку стратегію оберете: відхиляти запис (CP) чи відповідати з локальних даних (AP)? Які ризики в кожному випадку?
28. Запропонуйте «latency-бюджет» для простої операції читання (клієнт → API → кеш → БД з реплікацією). Де шукатимете найбільші резерви оптимізації?
29. Наведіть приклад вимоги продукту, що змушує обрати сильнішу модель узгодженості (лініаризовність/серіалізовність). Чим доведеться пожертвувати?
30. Які практики допомагають пом’якшити наслідки eventual consistency на рівні бізнес-логіки (saga, outbox/inbox, компенсуючі дії, TTL-кеші, read-repair)?

ТЕМА 2. АРХІТЕКТУРНІ МОДЕЛІ

Мета вивчення теми: Сформувати системне розуміння основних архітектурних моделей розподілених систем - клієнт-серверної, багаторівневої (n-tier/мікросервіси), рівнозначних p2p-мереж, а також гібридних і федеративних підходів та навчити обґрунтовано обирати модель під конкретні вимоги (латентність, узгодженість, доступність, масштабованість, відмовостійкість, безпека, вартість і відповідність політикам даних).

2.1 Завдання до самостійної роботи

1. Дати чіткі визначення моделей: клієнт-сервер, 2-/3-/n-tier, мікросервіси, p2p, гібридні (cloud/edge, on-prem/cloud) і федеративні (federated data/systems).
2. Розмежувати тонкий/товстий клієнт, BFF (Backend-for-Frontend), синхронну (HTTP/gRPC) та асинхронну (MQ/Kafka) взаємодію.
3. Пояснити типові шари багаторівневих систем (presentation, API, business, data), їхні обов'язки й точки масштабування; показати відмінність моноліту і мікросервісів.
4. Розглянути p2p-мережі: типи оверлеїв (DHT, gossip), маршрутизацію, узгодження стану, NAT-traversal, стійкість до збоїв; окреслити місце блокчейну як окремого класу p2p-архітектур.
5. Показати гібридні та федеративні моделі: multi-cloud, edge/fog, георозподіл; принципи data locality, data sovereignty, політики доступу та резидентності.
6. Навчити зіставляти вимоги домену з архітектурою через призму CAP/PACELC (C/A/P та L/C-компроміси), а також через нефункціональні вимоги (SLO/SLA, MTBF/MTTR).
7. Порівняти моделі за критеріями: латентність/пропускна здатність, узгодженість даних (strong/eventual/causal), еластичність, спостережуваність, вартість володіння (TCO).
8. Розглянути патерни: CQRS, Event Sourcing, Saga, Circuit Breaker, Retry/Backoff та ідемпотентність - і де вони доречні в кожній моделі.
9. Окреслити ризики й контрольні заходи безпеки: довірчі межі, автентифікація/авторизація, шифрування в транзиті/на зберіганні, багатотенантність.
10. Надати типові case-study: низька затримка та локальні оновлення → edge + кеші (гібридна), висока узгодженість транзакцій → 3-tier/мікросервіси з сильними гарантіями, децентралізований обмін файлами чи координація без центру → p2p.
11. Запропонувати шаблон матриці вибору архітектури (вимоги → критерії → оцінка → рішення) для подальших практичних робіт.
12. Відпрацювати на практиці мінімальні артефакти: контекстна діаграма

(C4), послідовності взаємодій (sequence), схема розгортання для обраної моделі.

2.2 Короткі теоретичні відомості

Архітектурні моделі визначають, як розподіляються обов'язки, дані й координація між компонентами системи, а отже - якими будуть компроміси між затримкою, узгодженістю, доступністю, масштабованістю, вартістю та керованістю.

У клієнт-серверній парадигмі логіка та дані зосереджені на сервері, а клієнт здебільшого відповідає за інтерфейс і мінімальну обробку; товстий клієнт переносить частину бізнес-логіки «на край», зменшуючи тиск на мережу, але ускладнюючи оновлення та контроль версій. Перехід до багаторівневих систем розділяє презентаційний, прикладний і дата-рівні, додає прошарок API та кросс-сервісні політики, дозволяє незалежно масштабувати вузькі місця й застосовувати різні технології зберігання під конкретні шаблони доступу.

Еволюційним розвитком n-tier стали мікросервіси, де кожна бізнес-можливість інкапсулюється власним сервісом із чіткими контрактами, власником даних і автономним циклом релізів. Це зменшує зв'язність і підвищує стійкість до часткових збоїв, але вводить накладні витрати на мережеву взаємодію, трасування й узгодженість між сервісами. Синхронні виклики (HTTP/gRPC) простіші концептуально, проте ланцюжки залежностей збільшують p99-затримки; асинхронні шини повідомлень (черги, потоки подій) покращують розв'язність у часі, дають природну буферизацію та «справедливість» навантаження, однак зсувають систему до зрештою узгоджених моделей і вимагають ідемпотентності обробників, ретраїв з backoff та дедуплікації.

У таких системах узгодженість проєктують як локальну властивість меж сервісу: всередині - серіалізовність або лінеаризовність, назовні - компроміс між швидкістю та стійкістю до розділення мережі за CAP/PACELC; транзакції, що охоплюють кілька сервісів, замінюють сагами й компенсаційними діями, а читання оптимізують через CQRS та кеші з керованою застарілістю.

Рівнозначні p2p-системи усувають центральний координатор і покладаються на оверлейні мережі, де вузли автономно знаходять одне одного, маршрутизують запити та реплікують стан. DHT-структури забезпечують детерміноване адресування та масштабовані пошуки, gossip-протоколи - швидке розповсюдження метаданих і детекцію збоїв, а NAT-traversal і relaying роблять зв'язність практичною в мережах із проміжними пристроями. Відсутність центру покращує цензуростійкість і усуває «єдину точку відмови», але робить нормою часткові відмови та динаміку складу мережі, тож система зазвичай тяжіє до AP-профілю CAP і PA/EL у PACELC, приймаючи зрештою узгодженість як стандарт. Блокчейн - окремий клас p2p-архітектур, у яких незмінний журнал подій формують

протоколи консенсусу; PoW/PoS/BFT гарантують єдиний порядок транзакцій за відсутності довіри, проте платять затримкою підтвердження та обмеженим throughput, тому така архітектура виправдана там, де важливі децентралізована довіра, відкритий доступ і перевірюваність.

Гібридні моделі поєднують центр і край: частина обробки виконується на edge/fog-вузлах ближче до джерел даних для зменшення латентності й трафіку, тоді як агрегування, довготривале зберігання й аналітика відбуваються в хмарі або дата-центрі. Ключем є політика розміщення стану та алгоритми узгодження: від простих кешів із TTL і фоновим read-repair до каузальної узгодженості чи CRDT-структур, що дозволяють безконфліктне злиття конвергентних реплік. У таких системах архітектор формує «бюджет затримки» на шляхах запиту, відокремлює інтерактивні операції з жорсткими SLO від батчевих, визначає межі, де потрібна сильна узгодженість, і зони, де прийнятна сталева або каузальна, та встановлює механізми деградації під час розділень: локальний режим роботи, черги на відкладений запис, ідемпотентні команди та відновлення після reconnection. Типовим є використання CDN і edge-кешів для статичного та напівдинамічного контенту, локальних функцій для швидких перевірок і централізованих сервісів для «єдиного джерела правди».

Федеративні моделі описують співпрацю автономних доменів під спільними протоколами без передачі контролю над даними. На прикладі федеративних систем керування ідентичностями домени погоджують взаємну довіру через стандартні механізми (OIDC/SAML), у федеративних даних - організовують запити «на місці», забезпечуючи суверенність і відповідність регуляціям, а у федеративному навчанні - передають градієнти або параметри замість сирих даних. Тут першочерговими стають політики узгодженості на міжорганізаційних границях, аудит, договірні SLO і процедури розв’язання конфліктів; глобальні транзакції рідко можливі, тому проектна дисципліна спирається на івент-сорсинг, немиттєву конвергенцію та чіткі SLA на латентність між доменами. Федерації часто комбінують multi-cloud і on-prem кластери, де PACELC підказує: у нормі доводиться вибирати між додатковою затримкою глобальної координації (EC) і слабшими, але швидшими гарантіями (EL), а під час розділень - між відмовою операцій і тимчасовими розбіжностями.

Незалежно від моделі, дизайн має виходити з нефункціональних вимог та характеристик навантаження. Якщо домінує інтерактивність і вимога до низьких p99, доцільно мінімізувати синхронні ланцюжки викликів, виносити блокуючі операції в асинхронні черги та робити сервіси безстанними для горизонтального масштабування за балансувальниками. Якщо критична цілісність транзакцій, корисно зменшити кількість меж консистентності, централізувати записи або застосувати алгоритми консенсусу з кворумами, погодившись на додаткові мілісекунди координації. Для систем зі спорадичним підключенням або георозподілом краще працюють моделі з локальними оновленнями, конфлікт-вільними

структурами даних і суворою ідемпотентністю команд. Усі моделі потребують спостережуваності: кореляційні ідентифікатори, розподілене трасування, метрики й логування роблять видимими «довгі хвости» та дозволяють своєчасно керувати ретраями, таймаутами й обмеженням навантаження. Безпека перетинає кожен шар - від шифрування в транзиті та на зберіганні, через нульову довіру між сервісами та least privilege, до багатотенантності й контрольованої резидентності даних у федераціях і гібридних розгортаннях. Зріла практика завершується матрицею вибору архітектури: формулюванням вимог, відображенням їх на критерії, зіставленням із CAP/PACELC і явним оформленням рішень та компромісів, що забезпечують передбачувану поведінку системи за нормальних умов і під час збоїв.

2.3 Контрольні питання

1. У чому відмінність між клієнт-серверною, багаторівневою (n-tier) та мікросервісною архітектурами?
2. Тонкий vs товстий клієнт: переваги, недоліки, типові сценарії застосування.
3. Які ролі виконують шари presentation, API, business, data у n-tier? Де зазвичай виникають «вузькі місця»?
4. Коли мікросервіси виправдані порівняно з добре модульованим монолітом? Наведіть критерії прийняття рішення.
5. Поясніть патерн BFF (Backend-for-Frontend). У яких випадках він потрібен?
6. Синхронна (HTTP/gRPC) vs асинхронна (черги/стріми) взаємодія: як кожна впливає на p95/p99-затримки та зв’язність компонентів?
7. Що таке ідемпотентність у міжсервісних інтеграціях і чому вона критична для ретраїв з backoff?
8. Порівняйте 2PC і Saga для розподілених транзакцій: коли що обрати?
9. Що таке DHT? Які властивості (детерміноване адресування, масштабованість) вона забезпечує?
10. Як працюють gossip-протоколи і для чого вони використовуються (розповсюдження стану, детекція збоїв)?
11. Які є підходи до NAT-traversal у p2p (STUN/TURN/relay) і коли без relay не обійтися?
12. Який профіль за CAP/PACELC притаманний більшості p2p-систем і чому?
13. Блокчейн як p2p-архітектура: як вибір консенсусу (PoW/PoS/BFT) впливає на latency, throughput та узгодженість?
14. Наведіть приклади задач, де edge/fog суттєво зменшує латентність і трафік до центру.
15. Поясніть принципи data locality і data sovereignty. Як вони впливають на розміщення сервісів і даних?

16. Що таке «локальний режим» під час розділень мережі? Які політики узгодження після reconnection доцільні?
17. CRDT: коли такі структури корисні в edge-сценаріях і які гарантії вони дають?
18. Як скласти «latency-бюджет» для запиту в гібридній системі? Які ділянки зазвичай домінують у p99?
19. Чим федеративна система відрізняється від централізованої з точки зору власності на дані та управління?
20. Як організовується довіра між доменами у федераціях ідентичностей (OIDC/SAML)?
21. Що таке «запити на місці» (query-in-place) у федерації даних і коли вони необхідні?
22. Федеративне навчання (federated learning): які переваги та обмеження щодо приватності й узгодженості?
23. Які SLO/SLA варто фіксувати між автономними доменами у федерації?
24. Де доцільно забезпечувати сильну узгодженість (linearizability/serializability) у мікросервісній системі, а де достатньо eventual/causal?
25. Як PACELC допомагає вибирати між EL (нижча латентність) і EC (сильніша узгодженість) у «мирний час»?
26. Наведіть по одному прикладу систем, орієнтованих на PC/EC та PA/EL, і коротко обґрунтуйте вибір.
27. Для чого потрібні circuit breaker, bulkhead і rate limiting? Як вони взаємодіють із ретраями?
28. Які практики спостережуваності критичні для розподілених архітектур: кореляційні ID, розподілене трасування, метрики, логування?
29. Які межі довіри (trust boundaries) слід явно виділяти в клієнт-серверних, p2p та федеративних моделях?
30. Як шифрування в транзиті/на зберіганні та принцип найменших привілеїв реалізуються у multi-tenant системах?
31. Вам потрібно побудувати стрічку подій з мільйонами користувачів: яку архітектуру оберете (мікросервіси + стріми, p2p, гібрид) і чому?
32. Переїзд із моноліту на мікросервіси: які кроки міграції, які ризики для узгодженості даних і як їх мінімізувати (CQRS, Saga, ідемпотентність)?

ТЕМА 3. БЛОКЧЕЙН ЯК МОДЕЛЬ P2P-АРХІТЕКТУРИ

Мета роботи: Сформувати цілісне розуміння блокчейну як p2p-архітектури репліkованої стан-машини з незмінним журналом подій; пояснити, як принципи побудови (криптографічні хеші, ланцюжки блоків, Merkle-дерева, підписи, мережевий gossip) поєднуються з протоколами консенсусу (PoW, PoS, BFT-родини) для досягнення безпеки (safety), живучості (liveness) і керованої фінальності; навчити зіставляти ці принципи з реальними системами (Bitcoin, Ethereum) та архітектурою dApps і робити обґрунтований інженерний вибір з урахуванням продуктивності, узгодженості, доступності, масштабованості й безпеки.

3.1 Завдання до самостійної роботи

1. Дати точні визначення блокчейну як розподіленого журналу та p2p-накладної мережі: блок, заголовок, хеш-ланцюг, Merkle-дерево, транзакція, підпис, mempool, вузол-ретранслятор, повний/легкий вузол.
2. Пояснити модель загроз і властивості безпеки: цілісність, незмінність, псевдоанонімність, Sybil-атаки та механізми їх стримування.
3. Розкрити принципи консенсусу: цілі (safety vs liveness), фінальність (ймовірнісна та економічна/детермінована), вибір лідера/блок-продюсера, правила вибору гілки (fork choice).
4. Порівняти PoW, PoS і BFT-підходи: джерело «вартості»/ресурсу, умови безпеки, стійкість до розділень мережі, енергоспоживання, затримки й пропускна здатність.
5. Пояснити, як профілі CAP/PACELC проявляються в різних блокчейнах:
a. Nakamoto-консенсус: PA/EL із зрештою узгодженим станом (реорги під час розділень),
b. BFT-сімейство: PC/EC з пріоритетом строгої узгодженості й можливою втратою доступності під час розділення.
6. Розібрати архітектуру Bitcoin: UTXO-модель, формат блоку, складність і ретаргетинг, час блоку та «кількість підтверджень» як метрика фінальності, комісії та інцентиви, чинники orphan-rate (пропагація/розмір блоку).
7. Розібрати архітектуру Ethereum: account-based модель, EVM, газ і ринок комісій, події/логі, PoS-валідація (слоти, епохи, фінальність), вплив вибору fork-rule на затримки та узгодженість.
8. Порівняти UTXO vs account-based моделі з погляду паралелізації, простоти валідації, конфліктів і моделювання стану.
9. Сформувати уявлення про dApps: компоненти (смарт-контракт, фронтенд, гаманець, RPC/провайдер, вузол), токени (ERC-20/721), схеми авторизації/підпису транзакцій.
10. Окреслити основи масштабування: L2-ролапи (optimistic/zk), канали/стейт-канали, сайдчейни, шардинг; їхні гарантії узгодженості та

вплив на затримку.

11. Розглянути типові ризики та експлойти: 51%/супермаджоритарні атаки, selfish mining, MEV/реорганізації, реентрансі-вразливість, нескінченні лупи газу - та інженерні контрзаходи.
12. Навчитися читати й будувати прості діаграми потоків: від створення транзакції у гаманці до включення в блок і настання фінальності.
13. Уміти оцінювати продуктивність: як інтервал блоку, розмір блоку/газ-ліміт, мережевий gossip і топологія впливають на TPS, p95/p99-латентність і orphan-rate.
14. Провести зіставлення вимог домену з вибором ланцюга й консенсусу: де потрібна швидка фінальність (BFT/PoS), де прийнятні ймовірнісні підтвердження (PoW), де виправдані L2.
15. Сформувати набір практичних артефактів для лабораторної: шаблон мінімального смарт-контракту, план розгортання локальної мережі (devnet/testnet), чек-лист безпеки транзакцій і газ-бюджету.

3.2 Короткі теоретичні відомості

Блокчейн - це p2p-архітектура реплікованої стан-машини, у якій усі вузли підтримують спільний, незмінний журнал подій. Дані організовані у блоки: заголовок містить хеш попереднього блоку, корінь Merkle-дерева транзакцій, мітку часу та службові поля; завдяки криптографічним хешам змінити будь-який запис «всередині» ланцюга без помітної перетасовки всіх наступних блоків практично неможливо.

Транзакції підписуються приватними ключами й розповсюджуються мережею за принципом gossip; кожен вузол перевіряє валідність і тимчасово зберігає їх у mempool, а повні вузли (full nodes) додатково зберігають увесь ланцюг і виконують повну перевірку правил консенсусу. Дерево Merkle дозволяє компактно доводити включення транзакції в блок (inclusion proof), що використовують легкі клієнти (SPV) без завантаження всієї історії.

Оскільки мережа асинхронна й відмови часткові - нормальні, блокчейн потребує протоколу консенсусу, який тримає баланс між безпекою (safety: відсутність розходжень у історії підтверджених станів) і живучістю (liveness: система продовжує роботу попри затримки та відмови), надаючи модель фінальності - від імовірнісної до детермінованої.

У Bitcoin застосовано Nakamoto-консенсус на основі Proof-of-Work: майнери змагаються у пошуку блоку, підбираючи nonce так, щоб хеш заголовка був нижчим за ціль (difficulty target). Імовірність знайти блок пропорційна обчислювальній потужності; найдовша (точніше, «найважча») гілка вважається канонічною. Через конкуренцію та мережеві затримки часом виникають короткі розгалуження (orphan/stale blocks), які зникають після «перегону» однієї з гілок - тому фінальність у PoW є ймовірнісною: що більше «підтверджень» має транзакція (наступних

блоків поверх неї), то нижчий ризик реорганізації. Параметри на кшталт інтервалу блоків і розміру блоку впливають на компроміс між затримкою підтвердження, пропускною здатністю та ймовірністю розгалужень: швидке зростання блоків або надто часті блоки підвищують orphan-rate через повільнішу пропагацію. Модель обліку в Bitcoin - UTXO: транзакції «витрачають» виходи попередніх транзакцій і створюють нові виходи; така форма природно децентралізує перевірки, зменшує глобальний стан і спрощує паралелізацію валідації.

Ethereum перейшов від PoW до Proof-of-Stake: право пропонувати блок у певному часовому слоті надається валідатору, який поставив (stake) токени; інші валідатори голосують (attest) за бачений стан. Поєднання правил вибору гілки (наприклад, LMD-GHOST) з фіналізацією епох (Casper FFG) забезпечує швидку економічну фінальність: після достатньої кворумної кількості голосів «поворот назад» потребує значних штрафів (slashing) і стає економічно недоцільним. Рахункова (account-based) модель спрощує програмованість: стан описується балансами та сховищем акаунтів, які змінюються при виконанні байткоду віртуальної машини (EVM). Обчислювальна робота й доступ до сховища оплачуються «газом», що обмежує ресурси транзакції та захищає від зловживань; ринок комісій регулює навантаження. Розгалуження тут теж можливі через мережеву асинхронність, але механізми атестацій і фіналізації зменшують їхній ефект на прикладну семантику.

Розмаїття консенсусів у p2p-системах відбиває різні компроміси. У читанні через CAP/PACELC більшість блокчейнів за мережевого розділення обирають або доступність, допускаючи тимчасову розбіжність історій із подальшою конвергенцією (PA у CAP; EL у «мирний час» - мінімізація затримки на шкоду найсуворішій узгодженості), або пріоритезують строгість порядку за ціною можливої втрати доступності під час розділення (PC; EC у нормальному режимі). Родина BFT-протоколів (PBFT і нащадки), характерна для невеликих permissioned-мереж, дає детерміновану фінальність за кілька раундів голосування, але масштабованість і мережеві витрати обмежують її у відкритих мільйонних мережах. PoW забезпечує відкриту участь і стійкість до Sybil-атак через вартість обчислень, проте має високу енергетичну ціну й більшу латентність фінальності; PoS знижує «вартість доступу», покладається на економічні стимули та покарання, зберігаючи живучість при менших затримках, однак вимагає ретельного дизайну проти «нічого-на-кошт» та централізації стейку.

dApps (децентралізовані застосунки) будуються як поєднання смарт-контрактів, що живуть у ланцюгу та виконуються детерміновано всіма вузлами, і клієнтських застосунків, які взаємодіють із мережею через гаманці та RPC-провайдери. Контракт визначає правила зміни стану, події для спостереження, а також інтерфейси; поширені стандарти токенів (наприклад, ERC-20 для взаємозамінних і ERC-721 для невзаємозамінних

активів) уніфікують взаємодію екосистеми. Життєвий цикл транзакції включає підпис користувачем, потрапляння до mempool, відбір продюсером блока з урахуванням комісій і обмежень газу, включення в блок, поширення блока мережею та досягнення фінальності за правилами протоколу; затримки тут визначаються як локальними чергами, так і пропагацією та параметрами консенсусу. Для масштабування поверх базового шару застосовують L2-рішення: optimistic і zk-rollups агрегують транзакції поза ланцюгом із публікацією стиснених доказів у L1, зменшуючи вартість і латентність для користувача при різних гарантіях безпеки; додатково використовують канали стану, сайдчейни й шардинг. Водночас з'являються нові вектори ризику: MEV і реорганізації, уразливості смарт-контрактів (реентрансі, переповнення, неконтрольовані зовнішні виклики), атаки на мережеву синхронізацію; інженерні контрзаходи охоплюють формальну верифікацію критичного коду, обмеження прав доступу, багаторівневий моніторинг і дисципліну оновлень. Сукупно блокчейн демонструє, як p2p-мережа без центрального координатора може досягати спільного стану, коли криптографія, економічні стимули та мережеві протоколи узгоджуються в єдиний механізм із чіткими, хоч і небезкоштовними, гарантіями.

3.3 Контрольні питання

1. Дайте визначення блокчейну як реплікованої стан-машини в p2p-мережі.
2. Що містить заголовок блоку і яку роль відіграє хеш ланцюжка?
3. Для чого використовують Merkle-дерево і як працює доказ включення (Merkle proof)?
4. Чим відрізняються повний вузол (full node), легкий клієнт (SPV) і валідатор/майнер?
5. Опишіть життєвий цикл транзакції: підпис → mempool → включення в блок → фінальність.
6. Консенсус і властивості коректності
7. Поясніть різницю між safety і liveness у протоколах консенсусу.
8. Чим відрізняється детермінована фінальність від ймовірнісної? Наведіть приклади.
9. Яку роль відіграють правила вибору гілки (fork choice rule)?
10. Які загрози має нейтралізувати механізм стійкості до Sybil-атак?
11. Які компроміси за CAP/PACELC типово роблять публічні блокчейни?
12. Опишіть принцип Proof-of-Work та поняття «складності» і ретаргетингу.
13. Чому фінальність у Bitcoin є ймовірнісною? Як пов'язані «кількість підтверджень» і ризик реорганізації?
14. Що таке orphan/stale-блоки і чому їхня частка зростає при великих блоках або коротких інтервалах?

15. Розкрийте UTXO-модель: як валідуються витрати і чому вона полегшує паралельну перевірку?
16. Як формується ринок комісій та які фактори впливають на пріоритет транзакцій у mempool?
17. У чому полягає Proof-of-Stake в Ethereum: слоти, епохи, атестації, фіналізація (Casper FFG)?
18. Як правило LMD-GHOST впливає на вибір канонічної гілки й латентність фінальності?
19. Поясніть account-based модель і як EVM змінює стан мережі.
20. Що таке «газ», для чого він потрібен і як обмежує ресурси виконання?
21. Які ризики PoS (напр., «nothing-at-stake», централізація стейку) і як їм протидіють (slashing тощо)?
22. Назвіть основні компоненти dApp: смарт-контракт, фронтенд, гаманець, RPC-провайдер; поясніть їхні ролі.
23. Які стандарти токенів (ERC-20/721) і які інтерфейси вони визначають?
24. Перерахуйте типові уразливості смарт-контрактів (реентрансі, неконтрольовані call-и, переповнення тощо) та способи їхнього запобігання.
25. Що таке MEV і як він впливає на порядок транзакцій та користувацьку латентність?
26. Яку роль відіграють оракули та які ризики вони вводять у модель довіри?
27. Поясніть різницю між L2-ролапами: optimistic vs zk-rollups (довіра, латентність, пропускна здатність).
28. Чим відрізняються канали/стейт-канали, сайдчейни та шардинг за гарантіями безпеки і узгодженості?
29. Як параметри блоку (розмір/час) і швидкість пропагації впливають на TPS, p95/p99 і orphan-rate?
30. Які метрики варто відстежувати для продуктивності блокчейну на рівні користувача та мережі?
31. Як працює мережевий gossip і які стратегії зменшують затримку розповсюдження блоків/транзакцій?
32. Назвіть і опишіть атаки: 51%/супермаджоритарна, selfish mining, eclipse-атака.
33. Які механізми захищають від спаму/флуду транзакцій у публічних мережах?
34. Вам потрібна швидка фінальність для фінансових розрахунків: який тип консенсусу/ланцюга оберете і чому?
35. Для ігрового dApp із мікроплатежами: яку комбінацію L1/L2 обрати для балансу вартості, латентності та безпеки?
36. У мережі збільшили розмір блоку при незмінному інтервалі: спрогнозуйте вплив на orphan-rate і p99-латентність.
37. Порівняйте UTXO й account-based для сценарію з великою кількістю дрібних платежів і взаємодіючих смарт-контрактів.

ТЕМА 4. ОСНОВИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Мета роботи: Сформувати фундаментальне розуміння паралельних обчислень як бази для продуктивних розподілених систем: від моделей виконання (процеси, потоки) і механізмів синхронізації до аналітичних моделей PRAM/BSP та законів масштабування (Амдала, Густафсона). Навчити проєктувати коректні (без гонок і дедлоків) та ефективні паралельні алгоритми з прогнозованою продуктивністю.

4.1 Завдання до самостійної роботи

1. Розмежувати процеси й потоки: адресний простір, IPC vs спільна пам’ять, накладні витрати створення/перемикання, моделі розміщення (NUMA).
2. Упорядкувати модель пам’яті та причини гонок: види видимості, кеш-кохерентність, бар’єри пам’яті; що таке data race, atomic, lock-free / wait-free.
3. Опанувати механізми синхронізації: м’ютекси, семафори, бар’єри, умови (CV), читачі/писачі, спінлоки; шаблони producer–consumer, pipeline, work stealing.
4. Визначити й запобігати дедлокам, livelock, starvation; правила порядку захоплення та таймаути.
5. Ввести метрики продуктивності: work, span (critical path), speedup S(N), efficiency E(N), scalability, parallel overhead, granularity, load balance.
6. Розібрати PRAM-модель і її варіанти (EREW/CREW/CRCW): як обмеження доступу до пам’яті впливають на складність й алгоритмічні техніки (pad, partition, prefix-sum).
7. Розібрати BSP-модель: супер-кроки, параметри p, g, L, оцінка вартості $W + H\cdot g + S\cdot L$; відображення BSP на практику (MPI, колективи, батчинг, мінімізація синхронізацій).
8. Пояснити Закон Амдала: формула $S(N)=1/((1-p)+p/N)$, межі прискорення, «вузьке горлечко», вплив серіальної частки та оверхедів.
9. Пояснити Закон Густафсона: $S_G(N)=N-(1-p)(N-1)$, сильне vs слабке масштабування, чому збільшення задачі змінює прогноз продуктивності.
10. Застосувати закони до реальних сценаріїв: коли виграє сильне масштабування (фіксований обсяг), коли – слабке (фіксований час/якість результату).
11. Показати мапінг моделей на інструменти: OpenMP (директиви, редукції, scheduling), MPI (message passing, колективи, топології), GPU (SIMT, ієрархія потоків/пам’яті).
12. Навчитися будувати latency-throughput бюджет для паралельного ядра: вибір розміру блоку/пакета, мінімізація синхронізацій, уникнення помилок кешування (false sharing).

13. Відпрацювати методику вимірювань: мікробенчмарки, прогрів, реплікації, довірчі інтервали; базові профайлери/трейсери; інтерпретація p95/p99.
14. Підготувати набір навчальних артефактів: паралельні prefix-sum, map–reduce, матриця×вектор з аналізом PRAM/BSP-вартості; чек-лист безпеки (race-free) і продуктивності (баланс/кеш/IO).

4.2 Короткі теоретичні відомості

Паралельні обчислення вивчають, як виконувати багато операцій одночасно, зберігаючи коректність і прогнозовану продуктивність. Базовими сутностями виконання є процеси та потоки. Процес має власний адресний простір і взаємодіє з іншими процесами через механізми міжпроцесної комунікації (канали, сокети, спільні сегменти), що додає накладних витрат, але забезпечує ізоляцію. Потоки розділяють пам'ять одного процесу й перемикаються швидше, проте саме спільна пам'ять породжує гонки даних: два або більше виконань одночасно читають/записують ту саму змінну без належної синхронізації. Щоб уникати гонок, застосовують атомарні операції, блокування (м'ютекси, RW-локи), семафори, бар'єри, умови (condition variables), а також неблокуючі структури.

Правильне використання примітивів спирається на модель пам'яті: процесорні й компіляторні оптимізації можуть змінювати порядок доступів, тож потрібні бар'єри та правила видимості для гарантій, що запис одного потоку стане спостережним для іншого в належний момент. Водночас надмірна синхронізація шкодить продуктивності: вона збільшує затримки, звужує паралельність і підсилює контенцію. Типові хиби - дедлок (циклічне очікування ресурсів), livelock (постійна активність без прогресу) і starvation (голодування через несправедливий розподіл). Практика вимагає фіксованого порядку захоплення ресурсів, таймаутів, ідемпотентних критичних секцій, а також уникнення «false sharing» - ситуації, коли незалежні змінні потрапляють у той самий кеш-рядок і викликають непотрібну інвалідацію кешів.

Аналітичні моделі дають спільну мову для оцінювання алгоритмів. У PRAM (Parallel Random Access Machine) припускають ідеально синхронні процесори з єдиним абстрактним доступом до пам'яті. Різновиди EREW/CREW/CRCW регулюють можливість одночасних читань/записів: у EREW (Exclusive Read Exclusive Write) заборонені одночасні доступи до однієї комірки; у CREW (Concurrent Read Exclusive Write) дозволені паралельні читання, але не записи; у CRCW (Concurrent Read Concurrent Write) дозволені і паралельні читання, і записи, причому потрібне правило розв'язання колізій запису (наприклад, «перемагає мінімальний індекс»). Хоч модель і спрощена, вона дисциплінує мислення: наприклад, побудова префіксних сум чи паралельного сортування вимагає або уникати

конфліктів пам’яті (EREW), або свідомо їх розв’язувати (CRCW). Для наближення до реальності використовують BSP (Bulk-Synchronous Parallel): обчислення розбиваються на супер-кроки з локальною роботою, комунікацією та глобальною синхронізацією. Вартість оцінюють як $W+H\cdot g+S\cdot L$, де

W - сумарна локальна робота,

H - обсяг переданих повідомлень,

g - «ціна» одиниці передавання,

S - кількість супер-кроків,

L - вартість бар’єра. Ця модель напряму мапується на практику MPI: мінімізуйте S (рідкісні синхронізації), агрегуйте дані для зменшення

H, топологічно «наближайте» взаємодіючі процеси, використовуйте колективні операції з високоефективними реалізаціями.

Продуктивність характеризують час послідовного виконання T_1, паралельного на N виконавцях T_N, прискорення $S(N)=T_1$ і ефективність $E(N)=S(N)/N$. У практичному аналізі корисна теорема Брента: якщо сумарна робота дорівнює T_1, а критичний шлях - T_∞, то за достатньої кількості процесорів час близький до $\Theta(\max(T1/N,T_\infty))$; звідси видно, що поганий баланс навантаження або довга послідовна ділянка знищують масштабування.

Два базові закони описують межі масштабування. Закон Амдала припускає фіксований обсяг роботи й частку p, що паралелізується: $S(N)=(1-p)+p/N_1$. Навіть мізерна послідовна частина $(1-p)$ жорстко обмежує граничне прискорення: за $N\to\infty$ маємо $Smax \to 1 - pSmax$. У реальних програмах до цього додаються накладні витрати паралелізації: створення потоків, синхронізація, комунікації, кеш-промахи - ефективне p зменшується, а крива насичення приходить раніше. Закон Густафсона підходить інакше: збільшуємо розмір задачі зі зростанням N, тримаючи час приблизно сталим; отримуємо $SG(N)=N-(1-p)(N-1)$. Це показує, що для класів проблем, де природно росте паралельна частина (наприклад, точність сітки, кількість сценаріїв, розмір даних), слабке масштабування може бути ефективним, навіть якщо сильне масштабування (за фіксованого обсягу) швидко впирається в межі Амдала. Розумний дизайн поєднує обидва погляди: для заданого SLA шукаємо мінімальний N, що виконує дедлайн, і водночас оцінюємо, як зміниться якість чи деталізація рішення, якщо збільшити задачу при наявних ресурсах.

Перенесення теорії в інженерію починається з вибору моделі й інструментів. Для спільної пам’яті зручно використовувати OpenMP: директиви паралелізації циклів, редукції, політики розкладу (static/dynamic/guided) й обережне керування локальністю даних на NUMA. Для роздільної пам’яті підходить MPI: явні повідомлення точка-точка, колективи (broadcast/allreduce/alltoall), віртуальні топології та перекриття комунікації з обчисленнями. Для масово-паралельних підзадач ефективні GPU в моделі SIMT: правильна організація блоків/ниток, коалесцинг

доступів до глобальної пам’яті, використання спільної (shared) пам’яті, мінімізація дивергенції гілок.

Незалежно від платформи, потрібно будувати «латентнісно-пропускний бюджет» ядра: збалансувати гранулярність задач, мінімізувати кількість синхронізацій і комунікацій, вирівнювати навантаження (work stealing, динамічний розклад) та берегти кеші (вирівнювання структур, падінг проти false sharing). Коректність верифікується інваріантами й інструментами пошуку гонок/дедлоків, а продуктивність - репрезентативними мікро- і макробенчмарками з прогрівом, кількома прогоном і аналізом хвостів p95/p99, бо саме вони визначають користувацький досвід і стабільність конвеєрів.

Таким чином, процеси, потоки й синхронізація дають мікробудівельні блоки; PRAM і BSP - мову оцінювання та проєктування; закони Амдала й Густафсона - компас для рішень про масштабування, які разом дозволяють створювати паралельні програми, що є водночас правильними та ефективними.

4.3 Контрольні питання

1. Чим процес відрізняється від потоку з погляду адресного простору, ізоляції та накладних витрат?
2. Які переваги й ризики спільної пам’яті для потоків? Наведіть приклади.
3. Що таке data race і чим воно відрізняється від логічної гонки (race condition)?
4. Поясніть призначення м’ютекса, семафора, бар’єра та condition variable. Коли який примітив доцільний?
5. У чому різниця між RW-локом і звичайним м’ютексом? Які патерни доступу роблять RW-лок корисним/шкідливим?
6. Що таке atomic операції, а що - lock-free / wait-free структури? Які гарантії вони дають?
7. Поясніть поняття моделі пам’яті і навіщо потрібні бар’єри (memory fences).
8. Охарактеризуйте NUMA. Як політика розміщення даних впливає на продуктивність потоків?
9. Дайте визначення deadlock, livelock, starvation. Назвіть способи запобігання кожному.
10. Що таке false sharing? Як його виявляти й уникати на практиці?
11. Порівняйте синхронні та асинхронні черги в продюсер–консюмер сценаріях: латентність, пропускна здатність, справедливість.
12. Сформулюйте модель PRAM. Для чого вона використовується в аналізі паралельних алгоритмів?
13. Поясніть відмінності EREW, CREW, CRCW. Які конфлікти пам’яті дозволені/заборонені?
14. Яке правило розв’язання колізій записів можна застосовувати в CRCW і як воно впливає на складність?

15. Розгляньте паралельний prefix-sum: для яких варіантів PRAM він реалізується без конфліктів і чому?
16. Які прийоми (pad/partition/replication) використовують, щоб адаптувати алгоритм до EREW?
17. Сформулюйте BSP і поясніть значення параметрів p, g, L.
18. Запишіть вартість BSP-програми у вигляді W+H· g+S· L. Що означають W,H,S?
19. Як зменшити H і S у реальному коді (MPI): приклади технік?
20. Чому балансування навантаження (load balance) критичне в BSP і як його досягати?
21. Наведіть приклад алгоритму, де виграш від агрегації повідомлень (batching) суттєвий.
22. Сформулюйте Закон Амдала та поясніть, чому навіть мала серіальна частка обмежує прискорення.
23. Сформулюйте Закон Густафсона і поясніть відмінність сильного vs слабкого масштабування.
24. Обчисліть прискорення за Амдалом для p=0,92 і N=16. Якою буде ефективність E(N)?
25. Для задачі, що масштабується, при p=0,98 і N=64 оцініть $S_G(N)$ за Густафсоном. Що це означає практично?
26. Як накладні витрати паралелізації (створення потоків, синхронізація, кеш-промахи) змінюють ефективне p у формулах?
27. Які метрики варто збирати під час тестування паралельної програми: T_1,T_N,S(N),E(N),T∞?
28. У чому суть теореми Брента і як вона допомагає оцінити нижню межу часу виконання?
29. Чому важливо аналізувати p95/p99 затримок, а не лише середні значення?
30. Які підходи до розкладу (static/dynamic/guided) в OpenMP і коли кожен доречний?
31. Як уникають блокувальних «гарячих точок» (lock contention) у багатопотокових структурах даних?
32. Назвіть прийоми оптимізації під GPU (SIMT): коалесцинг доступів, використання shared memory, зменшення дивергенції.
33. Є цикл із 10% серіальної роботи. За яких N приріст швидкодії стає практично непомітним? Обґрунтуйте через Амдала.
34. Для алгоритму з нерівномірними підзадачами яку політику планування оберете в OpenMP і чому?
35. Дано BSP-алгоритм із W=10 операцій,H=10^8 байт, S=20, g=5· 10^−9 с/байт, L=5· 10^−4 с. Оцініть загальний час і визначте домінуючу складову.
36. Ви спостерігаєте деградацію масштабування при переході з 16 на 32 ядра. Які три найвірогідніші причини і як їх перевірити?

ТЕМА 5. ТЕХНОЛОГІЇ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Мета вивчення теми: Надати цілісне розуміння й практичні навички використання моделей та інструментів паралельних обчислень на CPU і GPU - OpenMP (спільна пам’ять), MPI (роздільна пам’ять/кластер), GPGPU з CUDA/OpenCL - з акцентом на коректність, продуктивність, масштабування та застосування в HPC і AI.

5.1 Завдання для самостійної роботи

1. Розмістити OpenMP, MPI, GPGPU в «ландшафті» паралельних технологій; визначити, коли доречна кожна наступна модель і їхні комбінації (MPI+OpenMP, MPI+CUDA):
   OpenMP: директиви parallel/for/sections, редукції, atomic/critical/locks, задачний паралелізм (task), політики розкладу (static/dynamic/guided), прив’язка потоків і NUMA, бази offload (target) у 5.x.
   MPI: ранги й комунікатори, p2p (blocking/non-blocking), колективи (broadcast/scatter/gather/allreduce), похідні типи, топології, односпрямовані операції (RMA), персистентні комунікації; бази відмовостійкості.
   GPU/SIMT основи: варпи/вейвфронти, occupancy, ієрархія пам’яті (register/shared/global/const/texture), коалесинг доступів, конфлікти банків, дивергенція гілок.
   CUDA: ядра й параметри запуску (<<<grid, block>>>), використання shared-пам’яті (tiling), атоміки, стріми та події для перекриття копій/обчислень, уніфікована пам’ять, pinned memory, основу multi-GPU (NCCL, peer-to-peer).
   OpenCL: модель платформи (platform/device/context/queue), ядра, буфери/образи, профілювання; акцент на переносимість між вендорами та trade-off «портативність ↔ продуктивність».
2. Скласти продуктивнісну модель для наступних архітектур:
   roofline (обчислювальна інтенсивність), межі ширини пам’яті й латентності; вартість PCIe/NVLink; поєднання з законами Амдала/Густафсона для сильного/слабкого масштабування.
   HPC-патерни: щільна/розріджена лінійна алгебра (GEMM, SpMV), stencil/ПДР, FFT, доменна декомпозиція, halo-обміни; картографування на OpenMP/MPI/CUDA.
   AI-застосування: бібліотеки й прискорювачі (BLAS/cuBLAS, cuDNN, oneDNN), Tensor Cores, змішана точність (FP16/BF16), агрегація градієнтів (NCCL/MPI AllReduce), data/model/pipeline-паралелізм, масштабування тренування.
3. Опанувати такі гібридні підходи: MPI+OpenMP на вузлі, MPI+CUDA між вузлами, GPUDirect RDMA, вибір розміру батчів/тайлів для мінімізації комунікацій.

4. Відпрацювати інструменти профілювання: Nsight Systems/Compute, ncu/nsys, VTune/perf, mpiP/Score-P; побудувати профілі, знайти «гарячі точки», оцінити p95/p99.
5. Забезпечити коректність і числову якість: тестування на гонки/дедлоки, стабільність редукцій (неасоціативність FP, Kahan), відтворюваність експериментів.
6. Провести енерго-продуктивнісний та вартісний аналіз: продуктивність/Вт і продуктивність/$.
7. Підготувати такі лабораторні артефакти: OpenMP: паралельний matrix-vector, prefix-sum, порівняння розкладів. MPI: 2D-stencil із halo-обмінами, AllReduce для суми. CUDA/OpenCL: tiled-matrix-multiply, скан (Blelloch), мікробенчмарк пропускної здатності пам’яті.

5.2 Короткі теоретичні відомості

Технології паралельних обчислень охоплюють низку моделей і інструментів, які по-різному організують одночасне виконання роботи на сучасних апаратних платформах. На вузлах із спільною пам’яттю найприроднішим вибором є OpenMP: компіляторні директиви (parallel, for, sections, task) вмикають багатопотокове виконання фрагментів коду без ручного створення потоків, а механізми редукцій, атомарних операцій та критичних секцій забезпечують коректність доступу до спільних змінних. Ефективність визначається розкладом і гранулярністю: статичний розклад підходить для рівномірного навантаження, динамічний і guided - для нерівномірного; прив’язка потоків до ядер і врахування NUMA мінімізують «міграцію» даних між сокетами. У версіях 5.x з’явився offload (target) на прискорювачі, що дозволяє одним програмним підходом покривати і CPU, і GPU. Втім, надлишкова синхронізація, false sharing та оверсубскрипція потоків швидко «з’їдають» прискорення, тому дизайн циклів і даних (вирівнювання структур, падінг, уникнення гарячих точок) є критичним.

Коли пам’ять роздільна, а обчислення розподілені між багатьма вузлами кластера, використовується MPI - стандарт передачі повідомлень. Кожен процес має власний адресний простір і спілкується через точка-точка виклики (блокуючі та неблокуючі Isend/Irecv) і високорівневі колективні операції (Broadcast, Allreduce, Alltoall тощо). Колективи спираються на ефективні дерева/біноміальні схеми і топології інтерконекту; правильно підібрані вони мінімізують чисті витрати комунікації. У практиці BSP-подібної оцінки це означає скорочення кількості глобальних синхронізацій, агрегацію дрібних повідомлень і збалансовану доменну декомпозицію, щоби уникати простоїв на бар’єрах. Сучасний MPI також підтримує односпрямований доступ до пам’яті (RMA), персистентні комунікації та інтеграцію з прискорювачами, що важливо в гібридних застосунках.

Масово-паралельні прискорювачі (GPGPU) реалізують модель SIMT: тисячі легких ниток об’єднуються у варпи/вейвфронти, виконуючи ті самі інструкції над різними даними. Продуктивність визначається ієрархією пам’яті (регістри → shared/локальна → L2 → глобальна) та здатністю «сховати» латентність доступу через достатню кількість активних варпів (occupancy). Для цього організовують коалесцинг доступів до глобальної пам’яті, кеш-тайлінг зі shared-пам’яттю та мінімізують дивергенцію гілок всередині варпу. У CUDA програміст явно задає розмір сітки й блоків запуску ядра (<<<grid, block>>>), керує потоками (streams) і подіями для перекриття копіювань і обчислень, застосовує уніфіковану пам’ять і pinned-буфери, а для мульти-GPU - peer-to-peer/NVLink і бібліотеки колективів (NCCL) для швидкого AllReduce. OpenCL пропонує схожу обчислювальну модель у більш портативній формі (платформи, контексти, черги команд), що дозволяє таргетувати різних вендорів і навіть CPU; ціна портативності - вищі накладні витрати на узгодження драйверів/компіляції та, часто, менше «тонких» оптимізацій під конкретну архітектуру.

Типові HPC-шаблони добре картографуються на ці інструменти. Щільна лінійна алгебра (GEMM) демонструє «обчислювально щільну» природу й чудово масштабується на GPU завдяки високій інтенсивності операцій; розріджені обчислення (SpMV) обмежені пропускною здатністю пам’яті та потребують ретельного розкладання структур даних (CSR/ELL/HYB) і перепакування для коалесцингу. Обчислення за ПДР (stencil) використовують доменну декомпозицію з halo-обмінами: всередині вузла - OpenMP або CUDA-тайлінг, між вузлами - MPI з накладанням комунікацій на обчислення (overlap). FFT поєднує локальні батчі на GPU з міжвузловими перестановками (all-to-all) - тісна співпраця MPI та прискорювачів тут визначає масштабованість.

У розробці штучного інтелекту прискорення базуються на тих самих принципах. Тензорні ядра і бібліотеки на кшталт cuBLAS/cuDNN/oneDNN реалізують матрично-векторні ядра з високою інтенсивністю, змішана точність (FP16/BF16) підвищує продуктивність і енергоефективність за умови контролю числової стабільності. Розподілене навчання ґрунтується на ефективному AllReduce для агрегації градієнтів (NCCL/MPI), а масштабування до сотень прискорювачів комбінує data-, model- та pipeline-паралелізм, балансує розмір батча й частоту синхронізації, зменшує комунікації через градієнтний компресинг і чекпойнтинг активацій. В end-to-end системах GPUDirect RDMA скорочує копіювання між GPU на різних вузлах, зменшуючи латентність «довгих хвостів» у p99.

Продуктивність програмного забезпечення поширено визначають за допомогою roofline-моделі: гранична швидкодія обмежена або пиковою обчислювальною потужністю пристрою, або «дахом» пропускної здатності пам’яті - тому оптимізація або підвищує обчислювальну інтенсивність (більше FLOP на байт через тайлінг/реюз), або зменшує трафік (кешування, спільна пам’ять, злиття операцій). Межі масштабування

задають і закони Амдала/Густафсона: у сильному масштабуванні критична серіальна частка й накладні витрати синхронізації, у слабкому - правильне «нарощення» задачі так, щоб зберегти ефективність і не втопитися в комунікаціях. У гібридних застосунках найкращі результати дає поєднання MPI для міжвузлової взаємодії з OpenMP або CUDA всередині вузла (MPI+OpenMP, MPI+CUDA): перший рівень відповідає за розподіл доменів і колективи, другий - за щільну експлуатацію кешів чи тензорних блоків. Завершує картину дисципліна спостережуваності: профайлери (Nsight Systems/Compute, VTune, mpiP/Score-P) виявляють «гарячі точки», небаланс, дивергенцію, неперекриті копіювання й некоалесцьовані доступи, а також дозволяють підтвердити, що саме пам’ять, а не обчислення чи синхронізація, лімітує виконання. Разом ці підходи формують практичний інструментарій для побудови коректних, масштабованих і енергоефективних рішень у HPC та AI.

5.3 Контрольні питання

1. У чому відмінність між директивами parallel for, sections і задачним підходом task?
2. Як працюють редукції в OpenMP і які типові помилки при їх використанні?
3. Порівняйте політики розкладу static, dynamic, guided: коли яку обрати?
4. Що таке false sharing і як його уникати в OpenMP-коді?
5. Як прив’язка потоків до ядер (affinity) і NUMA впливають на продуктивність?
6. Коли доцільно застосовувати atomic, а коли - critical або RW-локи?
7. Які обмеження та типові пастки при використанні omp target (offload) на GPU?
8. Поясніть різницю між блокуючими та неблокуючими операціями (Send/Recv vs Isend/Irecv).
9. Для чого потрібні комунікатори й групи в MPI? Наведіть приклад використання.
10. Які варіанти реалізації Allreduce і як вибір алгоритму залежить від топології мережі?
11. У чому переваги односпрямованих операцій (RMA) порівняно з p2p?
12. Як мінімізувати кількість глобальних синхронізацій у BSP-стилі?
13. Що таке персистентні комунікації і коли вони корисні?
14. Як організувати доменну декомпозицію для 2D/3D-stencil із halo-обмінами?
15. Поясніть модель SIMT та поняття warp/вейвфронт. Чому дивергенція гілок шкідлива?
16. Як коалесцинг доступів до пам’яті впливає на пропускну здатність GPU?
17. Для чого використовується shared memory у CUDA і як уникати

конфліктів банків?

18. Порівняйте керування виконанням у CUDA (<<<grid, block>>>, streams, events) та в OpenCL (черги команд).
19. У чому відмінність між pinned і pageable пам’яттю на хості? Коли pinned виправдана?
20. Які підходи існують для multi-GPU (peer-to-peer, NVLink, NCCL) і коли вони дають найбільший виграш?
21. Порівняйте портативність і продуктивність CUDA та OpenCL на прикладі одного ядра.
22. Розшифруйте roofline-модель: що таке операційна інтенсивність і як визначити, чи лімітом є обчислення чи пам’ять?
23. Які метрики збирати під час профілювання GPU (occupancy, memory throughput, warp efficiency)?
24. Як перекривати обчислення та передачі даних (overlap) на GPU і в MPI?
25. Які інструменти профілювання доречні для CPU (OpenMP), GPU (Nsight Systems/Compute) та MPI (mpiP/Score-P)?
26. Як виявити й усунути небаланс навантаження у гібридних застосунках (MPI+OpenMP / MPI+CUDA)?
27. Чому щільна лінійна алгебра (GEMM) добре масштабується на GPU, а SpMV часто упирається в пам’ять?
28. Як організувати FFT у кластері з GPU, щоб мінімізувати all-to-all-витрати?
29. Які типи паралелізму використовують у тренуванні нейромереж: data, model, pipeline?
30. Як працює AllReduce для агрегації градієнтів і чому важлива топологія мережі/бібліотека NCCL?
31. Що таке змішана точність (FP16/BF16), які переваги й ризики для числової стабільності?
32. Які прийоми зменшують комунікаційні витрати в розподіленому навчанні (градієнтний компресинг, чекпойнтинг активацій)?
33. Коли доцільно поєднувати MPI з OpenMP на одному вузлі, а коли - MPI з CUDA?
34. Як GPUDirect RDMA впливає на латентність міжвузлових обмінів?
35. Яку стратегію вибрати для нерівномірного навантаження: динамічний розклад у OpenMP чи балансування доменів у MPI?
36. Які кроки забезпечення коректності в паралельних редукціях з плаваючою комою (наприклад, алгоритм Kahan)?
37. Обчисліть прискорення за Амдалом для OpenMP-циклу з $p=0,95$ на $N=16$. Яка при цьому буде ефективність $E(N)$?
38. Для weak-scaling експерименту за Густафсоном з $p=0,99$ і $N=64$ оцініть $S_G(N)$. Що це означає практично?
39. Є GEMM на GPU з операційною інтенсивністю 20 FLOP/байт, пік пристрою 15 TFLOP/s, пропускна здатність пам’яті 600 GB/s. Якою є верхня межа продуктивності за roofline?

ТЕМА 6. РОЗПОДІЛЕНІ АЛГОРИТМИ ТА КОНСЕНСУС

Мета вивчення теми: Сформувати глибоке розуміння розподілених алгоритмів узгодженості та консенсусу (Paxos, Raft), уміти проєктувати й аналізувати механізми розподіленого блокування без дедлоків, а також будувати надійні процедури fault recovery (журнали, чекпойнти, реконфігурація) з формальними гарантіями safety та liveness під різними моделями відмов і часовими припущеннями.

6.1 Завдання до самостійної роботи

1. Окреслити системну модель: синхронність/часткова асинхронність, мережеві розділення, моделі відмов (crash, crash-recovery, мережеві; огляд BFT як контекст), наслідки FLP-impossibility і роль таймінгових припущень/таймаутів.
2. Розібрати кворуми та реплікацію: majority-кворуми, для умови R+W>N, primary-backup, chain replication, tunable consistency; вплив на latency/throughput і лініаризовність.
3. Для моделі Paxos: ролі (proposer/acceptor/learner), ідея номерів раундів (ballots), фази 1a/1b/2a/2b, інваріанти безпеки; Multi-Paxos для потоку команд, лідерство, лізи, реконфігурація.
4. Для моделі Raft: вибори лідера (term, голосування, heartbeats), AppendEntries і реплікація журналу, commit index, snapshotting/log compaction, joint consensus для зміни складу кворумів; властивості leader completeness і state machine safety.
5. Порівняти Paxos vs Raft: зрозумілість, повідомлення/затримки в нормальному випадку, write-amplification, простота реконфігурації, діагностика збоїв.
6. Розподілене блокування: централізовані сервіси (ZooKeeper/Chubby-підхід), токени «fencing» проти split-brain, lock vs lease, оренда з часовими межами (drift/ск'ю), ідемпотентність критичних секцій.
7. Дедлоки в розподілених системах: необхідні умови (взаємне виключення, утримання й очікування, відсутність примусового відбирання, циклічне очікування), стратегії запобігання/уникнення/виявлення (wait-die, wound-wait, таймаути, wait-for graph, edge-chasing/Chandy-Misra-Haas), локальні vs глобальні детектори.
8. Модель Fault recovery: write-ahead logging, чекпойнти та відновлення стан-машини, rejoin вузлів, ребаланс шард/реплік; семантики доставки (at-least/at-most/exactly-once і їхня практична реалізація через ідемпотентність, дедуплікацію, транзакційні outbox/inbox).
9. Час і впорядкування: логічні годинники (Lamport, vector), причинність, snapshot vs linearizability; вимоги до годинників для ліз (NTP/PTP, bounded drift).

10. Верифікація і тестування: інваріанти безпеки, моделювання (наприклад, TLA+), fault-injection/chaos engineering, методика перевірки кворумних інваріантів і поведінки під мережевими розділеннями.
11. Підготувати практичні артефакти: мінімальна реалізація leader election та log replication з majority-кворумом; лабораторний сервіс distributed lock із fencing-токенами та сценаріями дедлок-ін'єкцій; план disaster/fault recovery (журнали, чекпойнт, відновлення, тестові кейси на split-brain).

6.2 Короткі теоретичні відомості

Розподілені алгоритми спрямовані на те, щоб множина незалежних вузлів, з'єднаних ненадійною мережею, діяла як єдина узгоджена стан-машина. Ключовою рамкою мислення є системна модель: у реальних умовах маємо часткову асинхронність (затримки й швидкості не обмежені наперед, але «переважно добрі»), часткові відмови (crash, crash-recovery), можливі розділення мережі та недосконалі годинники.

Теорема неможливості FLP показує, що у чисто асинхронній моделі детермінований консенсус за наявності хоча б однієї відмови недосяжний; практичні системи виходять із цього, вводячи таймаути, припущення про «достатньо швидкий» канал і механізми повторних спроб. Для реплікованих даних центральною стає робота з кворумами: якщо множини читання й запису перетинаються (класичне $R+W>N$ або majority для $R=W=\lceil (N+1)/2 \rceil$), то зберігається інваріант єдиного «комітованого» префіксу журналу, що і є основою лініаризовності.

Paxos формулює цю ідею в мінімалістичному протоколі: пропонент обирає монотонно зростаючий номер раунду, збирає обіцянки від більшості акцепторів не приймати старші пропозиції (фаза 1), після чого пропонує значення, «прив'язане» до найновішого з уже бачених (фаза 2); інваріант safety гарантується тим, що будь-які два кворуми перетинаються і не можуть узаконити суперечливі значення. У практиці Multi-Paxos фіксує лідера, який багаторазово проводить фазу 2 для послідовності команд, зменшуючи накладні витрати виборів; додаються лізи (часові оренди) для уникнення суперечливих лідерств і механізми реконфігурації складу кворумів без втрати інваріантів. Raft перескладає ті самі ідеї в простіші для реалізації й верифікації кроки: кластери поділені на терми, у кожному з яких через голосування обирається лідер; далі він розповсюджує записи журналу AppendEntries, а коміт відбувається, коли запис закріплено на більшості.

Властивості leader completeness і state machine safety гарантують, що новий лідер не «загубить» підтверджені команди й що порядок застосування команд на всіх репліках однаковий. Snapshotting і log compaction обмежують розмір журналу, а joint consensus дозволяє безпечно змінювати склад кворумів, перекриваючи стару й нову конфігурації. В нормальному режимі обидва підходи дають подібну латентність на запис

(один раунд до більшості), але Raft спрощує діагностику й навчання, тоді як Paxos залишається більш загальним каркасом для різних варіантів оптимізацій.

Розподілене блокування - це застосування консенсусу для взаємного виключення між клієнтами, які спілкуються через ненадійні канали та можуть падати посеред критичної секції. Наївний «лок» поверх центрального сховища легко ламається за split-brain, повторному піднятті клієнта чи мережевих затримках.

Тому промисловий підхід спирається на централізовані координаційні служби (на кшталт ZooKeeper/Chubby), що самі побудовані на кворумній реплікації, і використовує короткоживучі ефермерні вузли та «fencing»-токени: кожне видання лока супроводжується монотонно зростаючим номером, який ресурс перевіряє перед виконанням операції; навіть якщо старий власник «прокинувся» із застарілим лізом, його запит буде відкинуто як такий, що має менший токен. Лізи (lease) з часового боку є більш практичними за «вічні» локи, але вимагають обмеження дрейфу годинників і продуманих таймаутів; критичні секції мають бути ідемпотентними або супроводжуватися дедуплікацією на боці сховища, щоб повтори після таймаутів не спричиняли дубль-ефектів.

Дедлоки у розподіленому світі підкоряються тим самим «кофманівським» передумовам (взаємне виключення, утримання й очікування, відсутність примусового відбирання, циклічне очікування), але виявлення й запобігання ускладнюються фрагментацією знань між вузлами. Практика використовує стратегії запобігання (упорядкування ресурсів, щоб унеможливити цикли), уникнення (протоколи на кшталт wait-die/wound-wait для старшинства транзакцій) та виявлення (глобальний wait-for graph із централізованим або розподіленим edge-chasing, локальні таймаути й пріоритети для розриву циклів). Усе це знову спирається на тайминги й повтори, тож від проектувальника вимагається чітка політика ретраїв, backoff і компенсацій.

Fault recovery пов’язує консенсус із життєвим циклом вузлів: ціль - відновити коректну, лініаризовну поведінку після збоїв без втрати підтверджених ефектів і без «подвійного застосування». Базовим механізмом є write-ahead logging: перед тим як застосувати команду до стану, її додають у реплікований журнал; після коміту на кворумі реплік команда стає незворотною, і її можна реаплаювати після перезапуску. Періодичні чекпойнти/снепшоти зменшують час відновлення, фіксуючи компактний стан стан-машини; новий або «відсталий» вузол може підтягнути журнал та/або снепшот і знову приєднатися до кворуму, не порушуючи інваріантів. Реконфігурація включає безпечну зміну множини голосуючих учасників, ребаланс шардів і перенесення лідерства з контролем навантаження.

Семантики доставки «at-least-once/at-most-once/exactly-once» у розподілених системах реалізуються не магією транспорту, а комбінацією

ідемпотентності операцій, транзакційного outbox/inbox і дедуплікаційних ключів: транспорт може доставити двічі або не доставити, але система повинна поводитися так, ніби рівно один раз; у стримінгу це часто досягається через коміт офсетів разом зі станом обробника.

Питання часу й упорядкування доповнюється логічними годинниками (Lamport, vector) і прив'язкою до реального часу лише там, де це виправдано (лізи, дедлайни, SLA), з поясненням вимог до синхронізації (NTP/PTP, межі дрейфу). Нарешті, практичні системи валідують описані інваріанти через формальні моделі (наприклад, TLA+ для протоколів), fault-injection і chaos-тести з моделлю збоїв (затримки, drop, розділення), переконуючись, що safety завжди зберігається, а liveness відновлюється після усунення несприятливих умов. Сукупно це формує дисципліну, у якій консенсус забезпечує єдиний порядок команд, розподілені локи - кероване взаємне виключення без пасток дедлоку, а механізми відновлення - повернення системи до правильної роботи з передбачуваними часовими характеристиками і без втрати даних.

6.3 Контрольні питання

1. Які припущення робить частково асинхронна модель мережі та вузлів? Чим вона відрізняється від синхронної/асинхронної?
2. Сформулюйте наслідок теореми FLP для консенсусу. Як практичні системи обходять її обмеження?
3. Поясніть поняття кворуму. Чому умова R+W>N гарантує перетин кворумів читання/запису?
4. Що таке лініаризовність і чим вона відрізняється від серіалізовності у реплікованих журналах?
5. Як вибір величини кворуму впливає на латентність і доступність під час розділення мережі?
6. Назвіть ролі у Paxos (proposer/acceptor/learner) і їхні обов'язки.
7. У чому суть фаз 1 (prepare/promise) та 2 (accept/accepted)? Які інваріанти safety забезпечуються?
8. Навіщо потрібні монотонні номери раундів (ballots)? Що трапиться без них?
9. Як Multi-Paxos зменшує накладні витрати порівняно з одиничним Paxos?
10. Яка роль ліз (lease) у системах на основі Paxos і які вимоги до годинників вони накладають?
11. Як коректно виконати реконфігурацію (зміну складу кворумів) у Paxos, не порушивши safety?
12. Як відбувається вибір лідера у Raft (term, голосування, таймаути, heartbeats)?
13. Поясніть механізм AppendEntries та умови коміту запису (commit index, majority).

14. Що таке leader completeness і state machine safety у Raft?
15. Для чого потрібні snapshotting і log compaction? Які ризики при їх неправильній реалізації?
16. Як працює joint consensus у Raft під час зміни конфігурації кластера?
17. Які причини «зсуву журналу» (log inconsistency) у реплік і як Raft його виправляє?
18. Порівняйте зрозумілість реалізації, накладні витрати на нормальний запис, діагностику збоїв і реконфігурацію.
19. У яких сценаріях ви б віддали перевагу Paxos, а в яких - Raft? Обґрунтуйте.
20. Чому «наївний» розподілений лок поверх сховища може призвести до split-brain?
21. Що таке fencing token і як він запобігає діям «застарілого» власника лока?
22. Порівняйте lock і lease: переваги, ризики (drift/ск'ю годинників), вибір таймаутів.
23. Як ZooKeeper/Chubby забезпечують надійні локи (епемерні вузли, послідовні ZNode, спостерігачі)?
24. Чому ідемпотентність операцій важлива для повторних спроб після таймаутів?
25. Назвіть чотири необхідні умови дедлоку (Кофман) і наведіть приклади у розподіленому контексті.
26. Поясніть стратегії wait-die та wound-wait. Коли кожна з них доцільніша?
27. Як будується wait-for graph у розподіленому середовищі і чим складне глобальне виявлення циклів?
28. У чому ідея edge-chasing (Chandy–Misra–Haas) і які обмеження методу?
29. Які практичні способи запобігання дедлокам у сервісних системах (упорядкування ресурсів, таймаути, пріоритети)?
30. Розкрийте схему write-ahead logging і застосування журналу під час відновлення після збою.
31. Яку роль відіграють чекпойнти/снепшоти у скороченні часу відновлення?
32. Поясніть семантики at-least-once, at-most-once, exactly-once. Як на практиці досягають «exactly-once»?
33. Що таке outbox/inbox і як ці патерни допомагають із дедуплікацією?
34. Як безпечно «повернути» відсталий вузол у кластер, не порушивши інваріанти узгодженості?
35. Які кроки потрібні для коректної зміни лідера без втрати підтверджених операцій?
36. Поясніть різницю між логічними годинниками Лампорта, векторними годинниками та реальним часом.
37. Де у розподілених протоколах справді потрібен «реальний час», а де достатньо логічних порядків?

ТЕМА 7. ВІРТУАЛІЗАЦІЯ ТА КОНТЕЙНЕРИЗАЦІЯ

Мета вивчення теми: Сформувати практичне й концептуальне розуміння віртуалізації та контейнеризації для побудови відтворюваних, ізольованих і продуктивних середовищ у паралельних та розподілених системах; навчити обґрунтовано обирати між віртуальними машинами та контейнерами, керувати життєвим циклом контейнерів засобами Docker / Docker Compose / Swarm, а також правильно використовувати ресурси CPU, пам'яті, мережі й GPU у високопродуктивних та кластерних сценаріях.

7.1 Завдання до самостійної роботи

1. Розмежувати віртуалізацію (гостьові ОС над гіпервізором типу 1/2) та контейнеризацію (ізоляція рівня ОС через namespaces/cgroups): накладні витрати, ступінь ізоляції, швидкість старту, щільність розміщення.
2. Пояснити будову контейнерного образу (шари, union FS), OCI-стандарти (образ/рантайм), multi-arch образи, multi-stage builds, принцип «immutable infrastructure».
3. Розібрати Docker: build/push/pull/run, мережі (bridge/host/overlay/macvlan), томи (bind/volume/tmpfs), ліміти cgroups v2 (CPU, memory, pids, I/O), healthcheck, restart-політики.
4. Показати, як Docker Compose описує багатосервісні середовища: залежності, мережі, томи, секрети; застосування для локального розроблення й інтеграційних тестів.
5. Опанувати Docker Swarm: поняття swarm-кластера, менеджери/воркери, services/stacks, розміщення (constraints, affinities), rolling updates/rollback, secrets/configs.
6. Порівняти Swarm з повноцінним оркестратором (Kubernetes) за складністю, функціоналом і доречністю для малих/середніх кластерів.
7. Розглянути безпеку контейнерів: rootless, user-namespaces, capabilities, seccomp/AppArmor/SELinux, image-signing та SBOM/сканування вразливостей; контроль доступу до сокета Docker.
8. Пояснити мережеві моделі в контейнерах для HPC/кластерів: overlay і його латентність, host-network для низьких затримок, особливості RDMA/SR-IOV у контейнерах.
9. Використання GPU у контейнерах: NVIDIA Container Toolkit, видимість девайсів, мульти-GPU, ізоляція та керування драйверами; сумісність із CUDA/cuDNN.
10. Налаштувати продуктивність для паралельних обчислень: CPU pinning/cpuset, NUMA-політики, hugepages, file-system options, уникнення «noisy neighbor», оптимізація дискового й мережевого I/O.
11. Запустити MPI в контейнерах (один вузол і мультивузол): відповідність версій MPI, відкриття портів, обмін ключами, overlay vs host-мережа, колективні операції на інтерконекті.

12. Поєднати MPI+OpenMP або MPI+CUDA у контейнеризованих застосунках; врахувати розповсюдження бібліотек (NCCL, OpenMPI UCX) та доступ до міжвузлових транспортів.
13. Організувати стан і дані: проєкція томів, політики збереження, вибір драйверів (local/NFS/CSI), резервування та міграція; робота зі stateful-сервісами у Swarm.
14. Інтегрувати контейнери в CI/CD: кешування шарів, reproducible builds, policy-gate перед публікацією образів у реєстр (Docker Hub/Harbor/GitHub Container Registry).
15. Налаштувати спостережуваність: збір логів (JSON-driver/forwarders), метрик і трас (node-exporter, cAdvisor, OpenTelemetry-агенти), моніторинг стану служб Swarm.
16. Показати типові кейси HPC та AI: контейнеризований GEMM/FFT/Stencil на CPU/GPU; розподілене навчання з NCCL AllReduce (multi-GPU, multi-node) у Swarm; пайплайни даних/мікросервіси з різними профілями ресурсів.
17. Розібрати trade-off VM vs containers у продуктивних середовищах: коли потрібна повна ізоляція апаратури/ядра (VM), а коли критичні швидкий старт і щільність (containers).
18. Підготувати практичні артефакти: мінімальний образ для CPU-й GPU-завдань (multi-stage, slim); docker-compose.yml для багатосервісного стенду; Swarm-stack з rolling update та секретами; інструкцію запуску MPI/CUDA у контейнерах із вимірами p95/p99 латентності.

7.2 Короткі теоретичні відомості

Віртуалізація і контейнеризація - це два споріднені, але різні підходи до ізоляції та відтворюваності середовищ. Віртуальні машини відокремлюють цілу гостьову ОС від апаратури через гіпервізор (типу 1 - напряму над «залізом», типу 2 - поверх хост-ОС); кожен ВМ має власне ядро, драйвери, файлову систему, а отже кращу ізоляцію, підтримку різних ядер і стабільні сценарії «живої міграції» й довготривалих stateful-навантажень.

Ціна використання віртуалізації та контейнеризації - вища латентність ввімкнення, менша щільність розміщення і додаткові оверхеди на віртуалізацію пристроїв (I/O, мережа, сховище), які пом’якшуються паравіртуальними драйверами (virtio) та IOMMU. Контейнери ізолюють процеси на рівні ядра хоста за допомогою namespaces і cgroups: усі контейнери ділять одне ядро, тому стартують швидко, займають мало місця і дозволяють високу щільність; натомість ізоляція слабша (ядро спільне), а залежність від версії й конфігурації ядра хоста вища. Образи контейнерів - багатошарові (union filesystem, здебільшого overlay2), будуються як незмінні артефакти з декларативними рецептами (multi-stage builds), поширюються через реєстри й стандартизовані в рамках OCI (образ

і рантайм, з типовим стеком containerd/runc). Саме незмінність і шаровість забезпечують відтворюваність експериментів і CI/CD-пайплайнів.

Docker надає базові примітиви «build–push–pull–run», мережі й томи. Типові мережеві режими - bridge (NAT і ізоляція), host (мінімальна латентність за рахунок відсутності NAT), overlay (віртуальна L2/L3-мережа поверх кількох хостів); вибір визначається компромісами між простотою, безпекою та p99-латентністю. Для даних застосовують bind-монти (швидко, але менш керовано), іменовані volumes (керовані Docker'ом) і tmpfs (для швидких, ефемерних даних). Cgroups v2 дозволяють лімітувати CPU, пам'ять, pids і I/O, а healthcheck і політики restart керують життєвим циклом процесів усередині контейнера.

Docker Compose описує багатосервісні інсталяції у YAML: сервіси, залежності, мережі, томи, секрети й змінні середовища; це стандартний інструмент локальної розробки, інтеграційних тестів і повторюваних стендів. Коли потрібен кластер, у вбудованому оркестраторі Docker Swarm створюють «рой» із менеджерів і воркерів; стан кластера зберігається в реплікованому сховищі на основі Raft, а запуск ведеться у термінах services і stacks. Swarm підтримує декларативну кількість реплік, глобальні сервіси, розміщення за лейблами й обмеженнями (constraints/affinities), rolling updates/rollback, secrets/configs, ingress та overlay-мережі з VIP або DNSRR, що дозволяє просто керувати оновленнями і стійкістю невеликих/середніх кластерів. Порівняно з Kubernetes Swarm простіший і легший в адмініструванні, але має скромніші можливості планування і меншу екосистему - цього достатньо для навчальних кластерів, edge/відокремлених майданчиків і невеликих продакшн-рішень.

Безпека контейнерів базується на мінімізації привілеїв: rootless-режим, user namespaces, обрізання capabilities, профілі seccomp/AppArmor/SELinux, підпис образів і складання SBOM для аналізу вразливостей. Водночас доступ до Docker socket - критичний: його потрібно різко обмежувати. З погляду продуктивності важливі CPU pinning і cpuset (щоб не мігрувати «гарячі» потоки між ядрами і сокетами), NUMA-політики й hugepages, уважний вибір файлової системи та драйвера сховища, планування I/O і «шумні сусіди». Для мережевих навантажень з низькою латентністю зазвичай обирають host-мережу або апаратне прискорення (SR-IOV/VF, DPDK), адже overlay додає кілька зайвих стрибків і інкапсуляцій.

У паралельних і розподілених системах контейнери надають стандартне, «упаковане» виконуване середовище з чіткими залежностями, що радикально спрощує відтворюваність експериментів і переносимість між центрами обробки даних. Для GPGPU використовують NVIDIA Container Toolkit: у контейнер проброшуються пристрої GPU, драйвер і бібліотеки на хості узгоджуються з бібліотеками в образі (CUDA/cuDNN), а multi-GPU сценарії застосовують peer-to-peer, NVLink і бібліотеки колективів (напр., NCCL) - це дозволяє тренувати нейромережі чи виконувати щільну лінійну алгебру в контейнерах практично без

додаткових накладних витрат.

Для міжвузлових паралельних програм (MPI) найкращі результати дають поєднання контейнерів із host-мережею, правильним «склеюванням» версій MPI/UCX/OFI й доступом до RDMA-транспортів (InfiniBand/RoCE) через SR-IOV або hostdev-пассту: тоді колективні операції не упираються в віртуальні мережеві стекі. Застосовують дві моделі: «mpirun всередині контейнера на кожному вузлі» або інтеграцію зі scheduler'ом (Slurm) і запуск через wrapper, що підкладає контейнер кожному рангові. В обох підходах доведеться відкривати порти рантайму MPI, розшарювати /dev/shm для швидкої сегментної пам'яті й ураховувати, що overlay-мережі збільшують латентність all-to-all/Allreduce. Для stateful-компонентів обирають відповідні драйвери томів (локальні, NFS/CSI), політики збереження і бекап/відновлення; у Swarm є базові інструменти керування секретами та конфігураціями, але складні схеми реплікації сховищ частіше виносять у зовнішні сервіси.

У мікросервісних розподілених системах контейнеризація стандартизує «один сервіс - один образ», полегшує blue/green і canary-оновлення (rolling у Swarm), спрощує трасування й моніторинг (лог-драйвери JSON/forwarders, cAdvisor/node-exporter, агенти OpenTelemetry).

Для критичних шляхів із низькою латентністю зменшують довжину ланцюжків синхронних викликів і віддають перевагу host-мережі; для масових неблокуючих черг - overlay достатньо. У HPC/AI-кейсах roofline-аналіз показує, що бенчмарки часто лімітовані не «контейнерністю», а смугою пам'яті/мережі: тож головні виграші дають тайлінг, коалесцинг, змішана точність на GPU, перекриття копій та обчислень (streams/NVLink/GPUDirect RDMA) і правильне співналаштування розмірів батчів та частоти синхронізації градієнтів; сама ж контейнеризація забезпечує керованість, повторюваність і швидке масштабування експериментів.

Таким чином, ВМ доцільні там, де потрібна жорстка ізоляція ядра, різні ОС і складні життєві цикли з live-migration; контейнери - коли критичні щільність, швидкий старт, стандартне пакування й безшовна інтеграція з CI/CD та кластерами. У паралельних і розподілених сценаріях поєднання Docker/Compose/Swarm із дисципліною ресурсного менеджменту (CPU pinning, NUMA, мережа, GPU) дозволяє отримати повторювані, керовані та продуктивні розгортання - від локального ноутбука до невеликого кластера.

7.3 Контрольні питання

1. У чому принципова різниця між віртуалізацією (гіпервізор, гостьова ОС) та контейнеризацією (namespaces/cgroups)?
2. Які компроміси між рівнем ізоляції, щільністю розміщення й часом запуску у ВМ та контейнерів?

3. Коли доцільніше обрати ВМ замість контейнерів (і навпаки)? Наведіть приклади робочих навантажень.
4. Як впливають паравіртуальні драйвери (virtio) та IOMMU на I/O продуктивність у ВМ?
5. Які обмеження накладає спільне ядро хоста для контейнерів (сумісність, модулі, syscalls)?
6. Що таке багатошаровість контейнерного образу та як працює union filesystem (overlay2)?
7. Для чого використовують multi-stage builds і чим корисні multi-arch образи?
8. Які стандарти визначає OCI (image, runtime) і яку роль виконують containerd/runc?
9. Як організувати версіонування та підпис образів (image signing, SBOM) у реєстрі?
10. Робота Docker: запуск, мережі, томи, ресурси. Поясніть цикл build → push → pull → run у Docker.
11. Порівняйте режими мереж: bridge, host, overlay, macvlan - де який доречний?
12. Чим відрізняються bind-mount, named volume і tmpfs з погляду продуктивності та керованості?
13. Як cgroups v2 обмежують CPU, пам’ять, PIDs та I/O? Наведіть приклади налаштувань.
14. Для чого HEALTHCHECK і політики перезапуску (on-failure, always)?
15. Як організувати rootless-режим і навіщо обрізати Linux capabilities?
16. Які ключові секції docker-compose.yml (services, networks, volumes, secrets) та їх призначення?
17. Як описати залежності сервісів та параметризувати конфігурації через .env?
18. Коли Compose достатньо, а коли потрібен оркестратор (Swarm/Kubernetes)?
19. Що таке менеджери й воркери у Swarm і як Raft використовується для зберігання стану?
20. Поясніть різницю між service та stack. Як виконуються rolling updates/rollback?
21. Як працюють розміщення за constraints/affinity та що таке global service?
22. Які опції балансування трафіку (VIP vs DNSRR) є в Swarm і як вони впливають на p99-латентність?
23. Які обмеження Swarm порівняно з Kubernetes важливі для продакшн-кластерів?
24. Які механізми ізоляції варто ввімкнути під час використання механізмів безпеки: user namespaces, seccomp, AppArmor/SELinux, read-only rootfs?
25. Чому доступ до Docker socket небезпечний і як його обмежувати?
26. Як організувати секрети й конфіги у Swarm і чим це краще за змінні середовища?

27. Що таке CPU pinning/cpuset і чому важливі NUMA-політики й hugepages для HPC-навантажень?
28. Які джерела накладних у overlay-мережах і коли варто переходити на host-network?
29. Як уникати «noisy neighbor» у багатоконтейнерному хості (I/O throttling, cgroups, ізоляція дисків/мережі)?
30. Як додати GPU в контейнер (NVIDIA Container Toolkit) і чому важлива сумісність драйверів та бібліотек CUDA/cuDNN?
31. Які підходи до multi-GPU та міжвузлових обчислень (NVLink, NCCL, GPUDirect RDMA) працюють у контейнерах?
32. Як правильно запускати MPI в контейнерах: відповідність MPI/UCX/OFI, відкриття портів, /dev/shm, host-network?
33. Які типові «вузькі місця» для GEMM/FFT/Stencil у контейнерах і як їх профілювати?
34. Як обрати драйвери томів (local, NFS, CSI) та політики збереження даних для stateful-сервісів?
35. Які практики CI/CD знижують час збірки (layer caching, reproducible builds, image scanning перед публікацією)?
36. Як збирати логи, метрики й треси в контейнерних середовищах (JSON log driver, cAdvisor/node-exporter, OpenTelemetry)?
37. У вас сервіс з вимогою ультранизької латентності: які мережеві налаштування контейнера/хоста застосуєте й чому?
38. Під час масштабування в Swarm зростає відсоток timeouts. Які три гіпотези перевірите першими?
39. Образ виріс до 3–4 GB і повільно деплоїться. Які кроки зробите для зменшення розміру та прискорення старту?
40. MPI-застосунок у контейнерах показує деградацію на Alltoall. Які зміни мережевої/кластерної конфігурації спробуєте?

ТЕМА 8. ІНТЕГРАЦІЯ ПАРАЛЕЛЬНИХ І РОЗПОДІЛЕНИХ СИСТЕМ

Мета вивчення теми: Сформувати цілісне розуміння принципів і практик інтеграції паралельних (HPC) та розподілених (Big Data/Cloud) систем: уміти поєднувати інструменти на кшталт MPI/OpenMP/CUDA з фреймворками Spark/Flink/Dask, проектувати гібридні конвеєри даних та обчислень на GPU-кластерах, робити обґрунтований вибір архітектури з огляду на продуктивність, узгодженість, вартість і експлуатаційні ризики.

8.1 Завдання до самостійної роботи

1. Розмежувати класи задач і навантажень: HPC (щільна/розріджена лінійна алгебра, ПДР, CFD) vs Big Data (ETL/ELT, batch/stream, ad-hoc аналітика, ML-пайплайни).
2. Порівняти моделі виконання: SPMD/MPI (жорстка синхронізація, кворуми комунікацій) vs Spark/Flink (RDD/DataFrame/stream, shuffle, fault tolerance).
3. Розібрати патерни інтеграції: in-situ/in-transit аналітика, Lambda/Kappa-архітектури, «Spark → MPI kernel → Spark» (обмін через файлові формати/Arrow/сокети), barrier execution у Spark 3 для запуску узгоджених MPI-робіт.
4. Опрацювати GPU-кластери: топології (PCIe/NVLink), мережі (InfiniBand/RoCE), бібліотеки NCCL/UCX, GPUDirect/GPUDirect RDMA/Storage; планування ресурсів (MIG, пріоритети, preemption).
5. Вибрати середовище виконання: Slurm vs Kubernetes (Volcano/Kube-batch, Spark on K8s, MPI-Operator); сценарії «HPC on K8s» і «Spark on Slurm».
6. Узгодити сховище й формат даних: POSIX-паралельні ФС (Lustre/GPFS) vs об’єктні (S3/Swift); кеші (Alluxio), Apache Arrow/Parquet/ORC для нуль-копійних шляхів і ефективних shuffle/IO.
7. Спроєктувати канал передачі між Big Data ↔ HPC: RDMA/UCX, Arrow Flight, ZeroMQ/Kafka; баланс «низька латентність ↔ простота експлуатації».
8. Врахувати узгодженість та відмовостійкість: де потрібна лініаризовність (метадані/черги), а де достатньо зрештою узгоджених кешів; політики checkpoint/restart, exactly-once у стримінгу.
9. Провести продуктивнісний аналіз: roofline (обчислювальна інтенсивність) + бюджет shuffle/мережі/IO; strong/weak scaling; p95/p99; вартість $/експеримент, Вт/продуктивність.
10. Описати кейси інтеграції: Spark + MPI для великомасштабного ETL → чисельна симуляція → пост-обробка, RAPIDS/Dask + CUDA для GPU-аналітики, розподілене навчання (PyTorch DDP/DeepSpeed) з AllReduce поверх NCCL у багатовузлових середовищах, стримінг Kafka → Spark

Structured Streaming → online-скоринг на GPU-сервісах.

11. Забезпечити контейнеризацію й портативність: Docker/OCI-образи для MPI/CUDA, узгодження версій драйверів і бібліотек, CI/CD з кешем шарів і SBOM.
12. Налаштувати спостережуваність: метрики (throughput/latency, GPU-util, IO, мережа), розподілене трасування (OpenTelemetry), логування, алерти на tail latency та shuffle spill.
13. Пропрацювати безпеку й мультиоренду: ізоляція GPU/CPU/пам'яті, квоти, політики доступу до даних, аудит і відтворюваність (dataset/model versioning).
14. Розробити план експериментів: еталонні мікротести (GEMM/SpMV, shuffle/scan), наскрізні бенчмарки (ETL→HPC→візуалізація), методика вимірювань і регресій.
15. Підготувати практичні артефакти: мінімальний конвеєр Spark ETL → MPI/CUDA-ядро → Spark пост-обробка; запуск мульти-GPU DDP з профілюванням NCCL/мережі; звіт зі strong/weak scaling, roofline-оцінками та рекомендаціями з оптимізації.

8.2 Короткі теоретичні відомості

Інтеграція паралельних (HPC) і розподілених (Big Data/Cloud) систем поєднує дві традиційно різні культури обчислень у єдиний наскрізний конвеєр «дані → обробка → моделювання/навчання → пост-аналіз», де стійкість і еластичність Big Data фреймворків підживлюють продуктивні ядра на базі MPI/OpenMP/CUDA. HPC-задачі будуються за SPMD-парадигмою з жорсткими колективними синхронізаціями, вимогами до низької латентності мережі та передбачуваної пропускної здатності сховищ (Lustre/GPFS), тоді як Big Data (Spark/Flink/Dask) оперує RDD/DataFrame/stream-графами з автоматичним відновленням після збоїв, масштабуванням «за вимогою» і гнучкими обмінами (shuffle) поверх об'єктних або гібридних сховищ (S3/Alluxio + локальні NVMe). Щоб «звести» ці світи, використовують патерни in-situ/in-transit аналітики (обчислювати або поряд із симуляцією, або в каналі її вивантаження), Lambda/Kappa-архітектури для поєднання batch і streaming, а також вставлення HPC-ядер усередину даних-конвеєрів: зокрема, Spark 3 має barrier execution, що дозволяє узгоджено стартувати MPI-роботу у складі job, тоді як обмін даними роблять через колонки Parquet/ORC, нуль-копійні буфери Apache Arrow або високошвидкісні канали (UCX/Arrow Flight/ZeroMQ), мінімізуючи перетворення форматів.

На GPU-кластерах продуктивність визначають топологія (PCIe/NVLink/NVSwitch), мережа (InfiniBand/RoCE з RDMA), бібліотеки NCCL/UCX та здатність приховувати латентність передач (GPUDirect RDMA/Storage, оверлап потоків і копій), тоді як рівень застосунку комбінує data/model/pipeline-паралелізм (PyTorch DDP, DeepSpeed,

Megatron-клас підходів) і тензорні ядра зі змішаною точністю (FP16/BF16) для підвищення FLOP/Вт. Планувальне середовище добирають під домінантну частину конвеєра: Slurm для щільних HPC-ядер із жорсткими вікнами синхронізації та Kubernetes (з Spark on K8s, MPI-Operator, Volcano/Kube-batch) для еластичної оркестрації сервісів, ETL і стрімінгу; у змішаних сценаріях практикують «HPC on K8s» (host-мережа, SR-IOV, NUMA/hugepages, GPU-квоти/MIG) або «Spark on Slurm» (ланчер викликає Spark-екзектори як batch-job, дані прокладаються через загальні ФС).

Критичною є угода про формати та локальність: колоночні зберігання з push-down фільтрами зменшують IO, кеші на вузлах і tiered-storage обмежують дорогі обміни, а Arrow забезпечує уніфіковане подання у пам'яті між Python/Java/C++ без копіювань. Узгодженість і відмовостійкість проектують диференційовано: метадані, черги керування і результати комітять із сильними гарантіями (лініаризовність у координаторах/метасервісах), тоді як масивні проміжні артефакти можуть бути зрештою узгодженими з явними політиками TTL та повторного обчислення; у стримінгу для exactly-once спираються не на «магічний транспорт», а на ідемпотентні приймачі, транзакційний outbox/inbox і ко-коміт з офсетами. Продуктивність читають крізь roofline (де програмний тайлінг і реюз даних підвищують операційну інтенсивність) і бюджет shuffle/мережі/IO; вузькі місця зазвичай у «довгих хвостах» p95/p99 (низькорівневі GC/IO-спайки, перегріті ключі, дисбаланс розділів, неблоковані all-to-all), тож потрібні правильний ключовий простір, перед-агрегація, адаптивний shuffle та топологічно обізнані колективи. Контейнеризація (OCI-образи з узгодженими версіями MPI/CUDA/NCCL, SBOM і сканування) і CI/CD роблять збірки відтворюваними, а безпека й мультиоренда забезпечуються квотами CPU/GPU/пам'яті, політиками доступу до даних і аудитом експериментів (версіонування датасетів/моделей). Сучасні сценарії включають конвеєри «Spark ETL → MPI/CUDA-симуляція → Spark пост-обробка/візуалізація», GPU-аналітику на RAPIDS/Dask з інтеграцією в сховище об'єктів, онлайн-скоринг «Kafka → Spark Structured Streaming → gRPC-сервіси на GPU», а також гібридні робочі навантаження, де edge-попередня обробка зменшує трафік, а центр виконує тяжкі чисельні ядра; успіх усіх цих поєднань тримається на дисципліні форматів і протоколів, грамотному плануванні ресурсів і вимірюванні ефективності не лише в FLOP/s, а й у $/експеримент та Вт/результат.

8.3 Контрольні питання

1. Чим відрізняються класи задач і вимоги до інфраструктури у HPC (SPMD, низька латентність) і Big Data (ETL/shuffle, еластичність)?
2. Поясніть SPMD/MPI vs RDD/DataFrame/stream-підходи. Де кожен з них доречний?
3. Що таке in-situ та in-transit аналітика і коли яку варто обрати?

4. Порівняйте Lambda- та Kappa-архітектури: сильні/слабкі сторони для гібридних конвеєрів.
5. Чому в інтеграціях часто обирають колоночні формати (Parquet/ORC) замість row-форматів?
6. Які переваги Apache Arrow для нуль-копійних шляхів між мовами/фреймворками?
7. Порівняйте паралельні POSIX-ФС (Lustre/GPFS) та об'єктні сховища (S3). Як це впливає на throughput/latency?
8. Коли доцільні кеші на вузлах (Alluxio/локальні NVMe) і які ризики узгодженості вони вводять?
9. Spark + MPI (та інші поєднання)
10. Що таке barrier execution у Spark 3 і як воно допомагає запускати узгоджені MPI-роботи?
11. Які способи передати дані між Spark і MPI: файли (Parquet), Arrow Flight, сокети/UCX? Плюси/мінуси.
12. Де розташувати «межу» між ETL і чисельним ядром: до чи після нормалізації/перекодування?
13. Як організувати контроль відмов: хто ретраїть - Spark чи MPI-надбудова? Як не дублювати роботу?
14. Поясніть різницю між PCIe, NVLink, NVSwitch і їхній вплив на між-GPU обмін у GPU-кластерах.
15. Навіщо потрібні NCCL/UCX і коли вмикати GPUDirect RDMA/Storage?
16. Що таке MIG на A100/H100 і як він допомагає мультиоренді?
17. Які стратегії паралелізму для DL-тренування (data/model/pipeline) і коли яку обирати?
18. Порівняйте Slurm і Kubernetes (з MPI-Operator/Volcano) для гібридних пайплайнів.
19. «HPC on K8s» vs «Spark on Slurm»: у чому практичні trade-offs?
20. Які налаштування (host-network, SR-IOV, hugepages, NUMA) критичні для низької латентності?
21. Де потрібна лініаризовність (metadata/черги), а де достатньо eventual consistency (кешовані артефакти)?
22. Як досягати exactly-once у стримінгу без «магічного транспорту» (ідемпотентність, outbox/inbox, ко-коміт офсетів)?
23. Які політики checkpoint/restart застосовувати для MPI, Spark і DL-тренування?
24. Поясніть roofline-модель. Як підвищити операційну інтенсивність в ядрах CUDA/MKL?
25. Які метрики p95/p99 варто відстежувати у змішаному пайплайні (shuffle spill, GC-паузи, мережа, GPU-util)?
26. Де типово виникають «гарячі ключі» та як їх мітизувати (перерозбиття, попередня агрегація)?
27. Як оцінити бюджет часу на «ETL → HPC ядро → пост-обробка» і визначити домінуючу ланку?

28. Як забезпечити ізоляцію GPU/CPU/пам’яті для кількох команд (квоти, пріоритети, preemption)?
29. Які підходи до контролю доступу й аудиту даних у змішаних середовищах (dataset/model versioning)?
30. Які ризики несумісності версій CUDA/NCCL/драйвера при контейнеризації і як їх уникати?
31. Чому важливо мати OCI-образи з зафіксованими залежностями та SBOM у CI/CD?
32. Як запускати MPI-роботи в контейнерах на кількох вузлах: вимоги до мережі, /dev/shm, RDMA?
33. Запропонуйте схему «Spark ETL → MPI симуляція → Spark візуалізація» для кліматичної моделі: де стоять чекпойнти?
34. Як побудувати онлайн-скоринг: Kafka → Spark Structured Streaming → gRPC-сервіси на GPU. Де вузькі місця?
35. Яким буде план даних/виконання для RAPIDS/Dask на об’єктному сховищі з NVMe-кешем?
36. Розрахункові/ситуаційні завдання. Ядро має 8 FLOP/байт; пікова пам’ять 800 GB/s, пікова обчислювальна потужність 20 TFLOP/s. Яка теоретична стеля продуктивності?
37. Розрахункові/ситуаційні завдання. AllReduce по 8 GPU через NVLink (200 GB/s лінк-еквівалент) і через IB HDR (200 Gb/s). Як зміниться час на 1 GB градієнтів?
38. Розрахункові/ситуаційні завдання. У Spark spill починається при 70% заповнення пам’яті: які три налаштування/зміни зробите першими?
39. Розрахункові/ситуаційні завдання. Бар’єр-старт MPI-задачі в Spark падає після рестарту екзек’тора. Які кроки діагностики та обхідні рішення?
40. Розрахункові/ситуаційні завдання. Ваше CUDA-ядро лімітоване пам’яттю. Які трансформації даних/тайлінг застосуєте для підвищення інтенсивності?
41. Розрахункові/ситуаційні завдання. Існує два варіанти: більше GPU з меншим батчем vs менше GPU з більшим батчем і рідшими синхронізаціями. Як порахувати $/експеримент і обрати оптимум?

ВИСНОВКИ

Методичні вказівки з дисципліни «Програмне забезпечення розподілених інформаційних систем та паралельних обчислень» є комплексним навчальним матеріалом, у якому гармонійно поєднано теоретичні основи та прикладні аспекти сучасних інформаційних технологій. У процесі опрацювання змісту вказівок чітко простежується прагнення забезпечити пошуковачів не лише знаннями фундаментального характеру, а й практичними інструментами, необхідними для професійної діяльності в умовах стрімкого розвитку цифрових технологій. Структура лекційних занять побудована логічно й послідовно: кожна тема містить формулювання мети й завдань, стислий виклад теоретичних відомостей, контрольні питання та рекомендовану літературу. Такий підхід забезпечує системність викладу, а також орієнтацію на формування у здобувачів як теоретичних знань, так і практичних навичок самостійного аналізу, дослідження й вирішення складних завдань.

Актуальність представлених матеріалів підтверджується тематикою розглянутих питань. У вказівках охоплено широкий спектр проблематики: від еволюції розподілених систем і класичних моделей паралельних обчислень до новітніх технологій контейнеризації, оркестрації та сучасних парадигм обробки великих даних. Це дозволяє аспірантам та здобувачам вищої освіти сформувати цілісне уявлення про галузь, зрозуміти її розвиток у ретроспективі та оцінити перспективи застосування новітніх технологічних підходів. Особливої уваги заслуговує те, що матеріал не обмежуються викладом усталених положень, а вводять у зміст дисципліни аналіз актуальних рішень і практик, які вже сьогодні використовуються у промисловості та наукових дослідженнях. Таким чином, навчальний матеріал спрямований на підготовку спеціалістів, здатних інтегрувати теоретичні знання з інноваційними підходами до побудови високопродуктивних і надійних розподілених систем.

Важливим здобутком методичних вказівок є наявність чітко сформульованих завдань для кожної теми, що визначають очікувані результати навчання. Це не лише забезпечує дидактичну цінність курсу, але й сприяє виробленню у здобувачів уміння застосовувати набуті знання у різних практичних контекстах. Контрольні питання після теоретичного викладу матеріалу створюють передумови для самооцінювання та поглиблення розуміння основних концептів. Вони стимулюють критичне мислення, спонукають до пошуку відповідей у рекомендованій літературі та практичних прикладах, що особливо важливо для аспірантів, які формують власну дослідницьку культуру.

Особливої уваги заслуговує добір літературних джерел, наведений у методичних вказівках. До списків включено як класичні фундаментальні праці провідних науковців у сфері розподілених систем (А. Таненбаум, Л. Ламперт, Дж. Херліхі, Н. Лінч та інші), так і наведені сучасні специфікації,

стандарти та офіційні документації відкритих проєктів (OpenMP, MPI, Docker, Kubernetes, Apache Spark, Ceph, RabbitMQ). Такий підхід створює умови для поєднання академічної фундаментальності з актуальністю сучасних рішень, забезпечуючи здобувачів можливістю орієнтуватися як у класичних концепціях, так і в реальних технологіях, які сьогодні визначають розвиток галузі. При цьому використані джерела є перевіреними й справжніми, що відповідає вимогам академічної доброчесності.

З методологічної точки зору вказівки спрямовані на інтеграцію міждисциплінарних знань. У них поєднано проблематику обчислювальної математики, інженерії програмного забезпечення, систем розподіленої обробки даних, хмарних технологій і високопродуктивних обчислень. Такий синтез знань відповідає сучасному тренду у світовій науці, коли дослідження й практика потребують комплексних компетентностей, що виходять за межі окремих дисциплін. Завдяки цьому аспіранти отримують змогу формувати універсальні навички, необхідні для роботи як у наукових, так і у промислових проєктах.

Не менш значущим є і те, що у методичних вказівках приділено увагу питанням масштабованості та відмовостійкості систем. Розгляд алгоритмів консенсусу, моделей узгодженості та стратегій fault tolerance формує у здобувачів розуміння того, які компроміси закладено в архітектуру сучасних систем і як правильно приймати інженерні рішення в умовах обмежень реального світу. Аналіз технологій оркестрації (Kubernetes), сервісної сітки (Istio, Linkerd), брокерів повідомлень (Kafka, RabbitMQ), розподілених сховищ (HDFS, Ceph) та баз даних нового покоління (NewSQL) створює ґрунтовну основу для професійної підготовки дослідників і практиків.

Окремо слід підкреслити, що матеріали методичних вказівок послідовно підтримують баланс між викладенням фундаментальних ідей (закони Амдала й Густафсона, моделі PRAM і BSP, теореми CAP і PACELC, алгоритми Paxos і Raft) та прикладними аспектами (CUDA, OpenCL, Docker Swarm, Structured Streaming у Spark). Це забезпечує цілісність курсу й формує у студентів здатність поєднувати абстрактні моделі з конкретними технологічними інструментами. Такий підхід розвиває не лише академічну ерудицію, але й практичну готовність до створення та підтримки реальних систем.

Завдяки такому поєднанню матеріал стає універсальним інструментом для підготовки аспірантів, здатних працювати з великими обсягами даних, проєктувати й аналізувати складні розподілені архітектури, проводити власні дослідження та впроваджувати інноваційні рішення у практику.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Tanenbaum A. S., van Steen M. Distributed Systems (3rd ed.). – 2017. – [Електронний ресурс]. – Режим доступу: https://www.distributed-systems.net/index.php/books/ds3/
2. Coulouris G., Dollimore J., Kindberg T., Blair G. Distributed Systems: Concepts and Design (5th ed.). – Pearson, 2011. – [Електронний ресурс]. – Режим доступу: https://www.pearson.com/en-us/pearsonplus/p/9780137521081
3. Kleppmann M. Designing Data-Intensive Applications. – O'Reilly Media, 2017. – 616 p. – [Електронний ресурс]. – Режим доступу: https://dataintensive.net/
4. Armbrust M., Fox A., Griffith R., Joseph A. D., Katz R., Konwinski A., Lee G., Patterson D., Rabkin A., Stoica I., Zaharia M. A View of Cloud Computing. – Communications of the ACM. – 2010. – Vol. 53, No. 4. – P. 50–58. – [Електронний ресурс]. – Режим доступу: https://people.eecs.berkeley.edu/~matei/papers/2010/cacm_above_the_clouds.pdf.
5. Bass L., Clements P., Kazman R. Software Architecture in Practice (3rd ed.). – Addison-Wesley, 2012. – 624 p.
6. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: Doctoral Dissertation. – University of California, Irvine, 2000. – [Електронний ресурс]. – Режим доступу: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
7. Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. – 2008. – 9 p. – [Електронний ресурс]. – Режим доступу: https://cdn.nakamotoinstitute.org/docs/bitcoin.pdf
8. Wood G. Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper). – 2021. – [Електронний ресурс]. – Режим доступу: https://ethereum.org/content/developers/tutorials/yellow-paper-evm/yellow-paper-berlin.pdf
9. Buterin V. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. – 2014. – [Електронний ресурс]. – Режим доступу: https://ethereum.org/en/whitepaper/
10. Amdahl G. M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities // AFIPS Spring Joint Computer Conference. – 1967. – Vol. 30. – C. 483–485. – [Електронний ресурс]. – Режим доступу: https://www3.cs.stonybrook.edu/~rezaul/Spring-2012/CSE613/reading/Amdahl-1967.pdf.
11. Gustafson J. L. Reevaluating Amdahl's Law // Communications of the ACM. – 1988. – 31(5). – C. 532–533. – [Електронний ресурс]. – Режим доступу: https://cacm.acm.org/research/reevaluating-amdahls-law/
12. Valiant L. G. A Bridging Model for Parallel Computation // Communications of the ACM. – 1990. – 33(8). – C. 103–111. –

[Електронний ресурс]. – Режим доступу: https://www.osti.gov/biblio/6502724
13. OpenMP Architecture Review Board. OpenMP Application Programming Interface, Version 6.0. – 2024. – [Електронний ресурс]. – Режим доступу: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf
14. Lamport L. The Part-Time Parliament // ACM Transactions on Computer Systems. – 1998. – 16(2). – С. 133–169. – [Електронний ресурс]. – Режим доступу: https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf
15. Ongaro D., Ousterhout J. In Search of an Understandable Consensus Algorithm (Raft) // USENIX ATC. – 2014. – С. 305–319. – [Електронний ресурс]. – Режим доступу: https://raft.github.io/raft.pdf
16. Ongaro D. Consensus: Bridging Theory and Practice: PhD dissertation. – Stanford University, 2014 (rev. 2016). – 164 p. – [Електронний ресурс]. – Режим доступу: https://web.stanford.edu/~ouster/cgi-bin/papers/ongaro-dissertation.pdf
17. Posenblum M., Garfinkel T. Virtual Machine Monitors: Current Technology and Future Trends // IEEE Computer. – 2005. – Vol. 38, No. 5. – С. 39–47.
18. Barham P., Dragovic B., Fraser K., Hand S., Harris T., Ho A., Neugebauer R., Pratt I., Warfield A. Xen and the Art of Virtualization // Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP). – 2003. – С. 164–177.
19. Kivity A., Kamay Y., Laor D., Lublin U., Liguori A. kvm: the Linux Virtual Machine Monitor // Proceedings of the Ottawa Linux Symposium. – 2007. – С. 225–230.
20. Felter W., Ferreira A., Rajamony R., Rubio J. An Updated Performance Comparison of Virtual Machines and Linux Containers // 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). – 2015. – С. 171–172.
21. Malitsky N., Castain R., Cowan M. Spark-MPI: Approaching the Fifth Paradigm of Cognitive Applications. – arXiv preprint arXiv:1806.01110. – 2018. – [Електронний ресурс]. – Режим доступу: https://arxiv.org/abs/1806.01110
22. Gittens A., Mahoney M. W., et al. Alchemist: An Apache Spark–MPI Interface. – arXiv preprint arXiv:1806.01270. – 2018. – [Електронний ресурс]. – Режим доступу: https://arxiv.org/abs/1806.01270
23. Apache Spark. pyspark. RDD.barrier - Barrier execution mode // PySpark Documentation. – 2025. – [Електронний ресурс]. – Режим доступу: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.barrier.html
24. Apache Software Foundation. [SPARK-24374] SPIP: Support Barrier Execution Mode in Apache Spark // Apache JIRA. – 2018–2025. – [Електронний ресурс]. – Режим доступу: https://issues.apache.org/jira/browse/SPARK-24374.

Електронне навчальне видання

Хошаба Олександр Мирославович,
Володимир Павлович Майданюк

Методичні вказівки до виконання самостійної роботи з дисципліни «Програмне забезпечення розподілених інформаційних систем та паралельних обчислень» зі спеціальності «Інженерія програмного забезпечення»

Рукопис оформив О. Хошаба, В.Майданюк

Видається в авторській редакції

Оригінал-макет виготовила О. Кушнір

Підписано до видання 15.09.2022
Гарнітура Times New Roman.
Зам. № Р2022-071
Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
press.vntu.edu.ua;
Email: irvc.vntu@gmail.com
Свідоцтво суб'єкта видавничої справи
серія ДК No 3516 від 01.07.2009 р.