

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

Крутько Олександр Олександрович

**ПОРІВНЯЛЬНИЙ АНАЛІЗ КОНКУРЕНТНИХ (CONCURRENT) І
ПАРАЛЕЛЬНИХ (PARALLEL) МОДЕЛЕЙ ПРОГРАМУВАННЯ:
ПРОДУКТИВНІСТЬ, МАСШТАБОВАНІСТЬ І ЗРУЧНІСТЬ
ВИКОРИСТАННЯ**


кваліфікаційна робота


здобувача вищої освіти другого (магістерського) рівня

освітньої програми «Мультимедійні системи»

за спеціальністю F2 «Інженерія програмного забезпечення»

Особистий підпис 

Науковий керівник , Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри , Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

АНОТАЦІЯ

Тема: Порівняльний аналіз конкурентних (concurrent) і паралельних (parallel) моделей програмування: продуктивність, масштабованість і зручність використання.

Спеціальність: F2 «Інженерія програмного забезпечення».

Установа: ЛНУ імені Тараса Шевченка, 2026 р.

Магістерська робота містить: 103 с., 17 рис., 2 табл., 32 джерел.

Об'єкт дослідження – процеси розробки та виконання програмного забезпечення в умовах конкурентного (concurrent) та паралельного (parallel) обчислення.

Предмет дослідження – моделі конкурентного та паралельного програмування, їхні властивості, архітектурні механізми, підходи до організації взаємодії та синхронізації, а також показники продуктивності, масштабованості й зручності використання у практичних програмних системах..

Мета дослідження – є проведення порівняльного аналізу сучасних моделей конкурентного та паралельного програмування з погляду продуктивності, масштабованості та зручності використання.

Результати дослідження – структуроване розуміння того, як сучасні підходи до конкурентного та паралельного програмування впливають на ефективність програмних систем, методичні основи для прийняття рішень під час проектування програмного забезпечення.

Ключові слова: ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ, ОДНОЧАСНІСТЬ, МОДЕЛІ КОНКУРЕНТНОСТІ, ПРОДУКТИВНІСТЬ, МАСШТАБОВАНІСТЬ, ВИСОКОПРОДУКТИВНІ ОБЧИСЛЕННЯ

ABSTRACT

Topic: Comparative Analysis of Concurrent and Parallel Programming Models: Performance, Scalability, and Usability..

Speciality: F2 "Software engineering".

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2026.

Master's thesis consists of: 103 pp., 17 im., 2 tables, 32 sources

Object of the study – the processes of software development and execution under concurrent and parallel computing conditions..

Subject of the study – concurrent and parallel programming models, their properties, architectural mechanisms, approaches to interaction and synchronization, as well as performance, scalability, and usability indicators in practical software systems.

Purpose of the research is to conduct a comparative analysis of modern concurrent and parallel programming models in terms of performance, scalability, and usability.

Research results include a structured understanding of how modern approaches to concurrent and parallel programming influence the efficiency of software systems, as well as methodological foundations for decision-making during software design.

Keywords: PARALLEL PROGRAMMING, CONCURRENCY, CONCURRENCY MODELS, PERFORMANCE, SCALABILITY, HIGH-PERFORMANCE COMPUTING.

ЗМІСТ

ВСТУП	5
Розділ 1. Теоретичні основи одночасних та паралельних моделей програмування	10
1.1. Еволюція підходів до одночасності та паралелізму в програмних системах	10
1.2. Класифікація моделей одночасного та паралельного програмування	13
1.3. Аналіз ключових парадигм та моделей паралельного програмування	16
1.4. Проблеми взаємодії, синхронізації та узгодженості даних у різних моделях	27
1.5. Критерії оцінювання ефективності: продуктивність, масштабованість, зручність використання	30
Розділ 2. Аналіз сучасних моделей одночасного та паралельного програмування	35
2.1. Потоківі, конкурентні та акторні моделі: принципи, переваги та обмеження	35
2.2. Моделі з розподіленою пам'яттю і комунікацією: BSP, BPRAM, Orca, Manta, occam- рі	39
2.3. Автоматизовані та апаратно-орієнтовані підходи: Jade, Cyrus, GPU-структури BACQ	43
2.4. Порівняльний аналіз моделей та узагальнення їх характеристик	47
Розділ 3. Експериментальна оцінка моделей одночасного та паралельного програмування	49
3.1. Методика експериментальних досліджень та засоби реалізації прототипів	49
3.2. Реалізація тестових застосунків: моделі потоків, акторів, паралельних черг та розподілених викликів	54
3.3. Оцінювання ефективності моделей та інтерпретація експериментальних результатів	64
ВИСНОВКИ	75
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	78
ДОДАТКИ	82
Додаток А Порівняльна таблиця різних моделей	82
Додаток Б. Код C# для множення двох матриць	85
Додаток В. Код множення матриць на Python	90
Додаток Г. Код на C# для сортування великого масиву	95
Додаток Д. Код на Python для сортування великого масиву	99

ВСТУП

Сучасний стан розвитку програмних систем характеризується стрімким зростанням вимог до їхньої продуктивності, масштабованості та здатності ефективно використовувати ресурси обчислювальних платформ різного рівня – від настільних процесорів до багатовузлових кластерів і прискорювачів на основі GPU. У цих умовах постає фундаментальна потреба у глибокому розумінні моделей конкурентного (одночасного) та паралельного програмування, які лежать в основі побудови високопродуктивних обчислювальних застосунків. Обрана тема є актуальною, оскільки саме конкуренція за ефективне використання апаратних можливостей і зменшення витрат синхронізації визначає успіх більшості сучасних інженерних, наукових та промислових програмних рішень. Одночасність і паралелізм перестали бути ознаками виключно високопродуктивних систем: вони стали необхідними для масштабованості будь-якого сучасного застосунку, зокрема вебсервісів, розподілених систем, IoT-платформ та програм другого рівня реального часу.

Проблема в загальному вигляді полягає в тому, що, попри наявність великої кількості моделей та підходів – від потоків та акторів до GPU-керованих черг, транзакційної пам'яті та розподілених середовищ – розробники стикаються з труднощами вибору оптимальної моделі для конкретного завдання. Різні підходи демонструють неоднакові характеристики щодо затримок, пропускної здатності, енергоспоживання, складності конфігурації, вартості синхронізації та зручності програмування. Нерідко моделі, розроблені для певної апаратної або прикладної галузі, поводяться неефективно поза своїм початковим контекстом. Аналіз двадцяти наукових статей, індексованих у Scopus, показує, що наукова спільнота пропонує широкий спектр інноваційних підходів, але не подає цілісної порівняльної картини, доступної інженеру-практику. Описи моделей, таких як Cyrus, Orca, Jade, ossam-pi чи Manta, демонструють суттєві переваги в певних нішах, але потребують уніфікованої оцінки за критеріями продуктивності,

масштабованості й практичної застосовності у звичайних умовах промислової розробки.

Аналізуючи останні дослідження науково-метричної бази Scopus з обраної методики можна констатувати, що дослідники активно розробляють оптимізації на рівні компілятора, структури даних та архітектурні рішення, спрямовані на підвищення паралелізму і зменшення накладних витрат синхронізації. Наприклад, моделі Jade або ossam-pi [25; 29] демонструють ефективність завдяки суворим правилам доступу до даних, тоді як Cygus [4] фокусується на високопаралельному відтворенні виконання, а BACQ [23] забезпечує масштабованість для GPU-орієнтованих систем. Розподілена спільна пам'ять Orca підтверджує можливість ефективного збереження абстракції спільного доступу без суттєвих втрат продуктивності [20], а система Manta демонструє потенціал швидкісних віддалених викликів у розподілених середовищах [16]. Разом з тим, незважаючи на широкий спектр теоретичних і практичних напрацювань, відсутня узагальнена модель прийняття рішень, яка б дозволяла обґрунтовано вибрати відповідний механізм одночасності або паралельності залежно від вимог конкретної задачі.

Об'єкт дослідження – процеси розроблення та виконання програмного забезпечення в умовах конкурентного (concurrent) та паралельного (parallel) обчислення, що забезпечують ефективне використання багатоядерних, багатовузлових та гетерогенних обчислювальних систем.

Предмет дослідження – моделі конкурентного та паралельного програмування, їхні властивості, архітектурні механізми, підходи до організації взаємодії та синхронізації, а також показники продуктивності, масштабованості й зручності використання у практичних програмних системах.

У такому контексті **метою** кваліфікаційної роботи є проведення порівняльного аналізу сучасних моделей одночасного та паралельного програмування з погляду продуктивності, масштабованості та зручності використання, а також формування практичних рекомендацій щодо їх

застосування у програмних системах, що можуть бути реалізовані стандартними засобами мов програмування загального призначення. Досягнення цієї мети передбачає систематизацію теоретичних основ, оцінювання моделей на основі експериментів, а також розроблення прототипів, доступних для реалізації в умовах навчальних і промислових середовищ.

Для досягнення поставленої мети необхідно розв'язати кілька послідовних завдань.

1. Здійснити аналіз міжнародних наукових досліджень, зокрема відібраних публікацій Scopus, щоб визначити ключові тренди та інноваційні механізми у сфері одночасності й паралелізму.
2. Розробити узагальнену класифікацію моделей, що включатиме потокові моделі, актори, транзакційну пам'ять, GPU-підходи та розподілені середовища.
3. Сформулювати критерії оцінювання продуктивності та масштабованості, включно з часом виконання, пропускну здатністю, накладними витратами синхронізації та апаратною незалежністю.
4. Провести експериментальне дослідження та реалізувати низку тестових прототипів за допомогою доступних технологій, таких як C#, Python, Java або Go, що дозволить отримати практичні результати, придатні для порівняльного аналізу.
5. Розробити узагальнені рекомендації щодо вибору моделей конкурентності та паралелізму для різних типів програмних систем, включно з реальними прикладами використання.

Наукова новизна дослідження полягає у комплексному порівняльному аналізі сучасних моделей конкурентного та паралельного програмування з урахуванням не лише теоретичних характеристик, а й експериментально підтверджених показників продуктивності та масштабованості, отриманих на основі реалізації прототипів у доступних мовах програмування. На відміну від наявних у літературі досліджень, які зазвичай описують окремі моделі,

системи або оптимізації для вузьких застосувань, у роботі здійснюється узагальнення та систематизація різномірних підходів – від потоково-орієнтованих і акторних моделей до транзакційної пам'яті, GPU-структур даних і розподілених моделей спільної пам'яті. Новизна також полягає у формуванні практичних рекомендацій щодо вибору моделі одночасності або паралелізму для типових завдань інженерії програмного забезпечення, що ґрунтуються на експериментальних даних, отриманих із використанням прототипів, реалізованих стандартними засобами .NET, Java, Python або Go. Таким чином, робота усуває прогалину між теоретичними підходами та інженерною практикою, пропонуючи практично застосовну, методично обґрунтовану структуру для прийняття рішень.

Методи дослідження кваліфікаційної роботи включають системний аналіз наукових джерел для узагальнення теоретичних основ сучасних моделей конкурентного та паралельного програмування; порівняльний аналіз для встановлення відмінностей, переваг і недоліків моделей за критеріями продуктивності, масштабованості та складності використання; моделювання та розроблення експериментальних прототипів із використанням доступних мов програмування для отримання емпіричних показників; експериментально-статистичні методи для вимірювання часу виконання, пропускну здатності, накладних витрат синхронізації та ефективності використання ресурсів; а також методи програмного профілювання та аналізу продуктивності для уточнення характеристик поведінки моделей під різним навантаженням. Додатково використовуються елементи інженерії програмного забезпечення – проектування архітектурних рішень, побудова тестових сценаріїв та верифікація результатів – що забезпечує практичну спрямованість проведеного дослідження.

Отримані результати дозволяють сформулювати структуроване розуміння того, як сучасні підходи до одночасного та паралельного програмування впливають на ефективність програмних систем, а також створити методичні основи для прийняття рішень під час проектування програмного забезпечення,

що масштабуватиметься та ефективно використовуватиме ресурси сучасних обчислювальних платформ.

Висловлюємо подяку адміністрації університету та особисто директору Інституту математики та інформаційних технологій Могильному Г.А. за можливість доступу до реферативної бази SCOPUS.

Розділ 1. Теоретичні основи одночасних та паралельних моделей програмування

1.1. Еволюція підходів до одночасності та паралелізму в програмних системах

Еволюція підходів до одночасності, конкурентності та паралелізму в програмних системах є результатом тривалого розвитку як апаратного забезпечення, так і програмних моделей, що забезпечують ефективне використання обчислювальних ресурсів. Перші комп'ютерні системи будувалися навколо однопоточкового виконання, де програма виконувалася послідовно і взаємодія з апаратурою була синхронною. Однак із появою багатозадачності в операційних системах виникла потреба в абстракціях, що дозволяють виконувати окремі процеси паралельно або принаймні створювати ілюзію паралельного виконання. Ранні підходи були переважно орієнтовані на процеси й контекстне перемикавання, що давало можливість виконувати кілька незалежних програм одночасно, але ще не забезпечувало ефективного паралелізму в межах однієї задачі.

Подальший розвиток моделей одночасності почався зі створення поточкових систем, які дозволили програмам розділяти виконання на кілька незалежних гілок, що працюють у межах одного процесу. Запропоновані механізми синхронізації – семафори, м'ютекси, монітори – сформували основу класичної конкурентності, але одночасно створили суттєві проблеми, пов'язані з гонками даних, взаємоблокуваннями та складністю відлагодження. Ці труднощі стимулювали інтенсивні дослідження у галузі формальних моделей програмування. Значний вплив на подальший розвиток здійснили роботи Лесли Лампорта, який сформулював фундаментальні уявлення про причинність подій, логічний час і послідовність у розподілених системах [13; 14], а також Тоні Хоара з його моделлю Communicating Sequential Processes (CSP), що запропонувала канали й обмін повідомленнями як альтернативу спільній пам'яті. CSP заклала теоретичну основу для багатьох сучасних мов та

фреймворків, таких як ossam-pi, Go та сучасні реалізації акторної моделі [9; 10].

У другій половині 1980-х та 1990-х років одночасність активно досліджувалася в контексті розподілених систем, коли з'явилися моделі розподіленої спільної пам'яті, зокрема Orca, що дозволила програмам працювати із глобальними об'єктами без явного керування передаванням даних. У цей же період формуються паралельні моделі для високопродуктивних обчислень, такі як Bulk Synchronous Parallel (BSP) або PRAM (Parallel Random Access Memory), які дозволили формально визначати характеристики продуктивності у великих паралельних системах [31]. Паралельно досліджувались механізми автоматичного вилучення паралелізму, на основі яких була створена система Jade, що аналізує доступ до даних і формує залежності між діями програми без явного втручання програміста. Із появою архітектур із підтримкою апаратного паралелізму та прискорювачів, таких як GPU, стало актуальним дослідження паралельних структур даних, зокрема Boundary-Aware Concurrent Queue, що забезпечує масштабований доступ у масивно-паралельних середовищах [23].

Початок XXI століття ознаменувався новим етапом розвитку, коли багатоядерні процесори стали стандартом навіть у побутових пристроях, а паралелізм із розкоші високопродуктивних систем перетворився на необхідність для будь-якого сучасного програмного забезпечення. Це спричинило пошук моделей, які є одночасно продуктивними та зручними для програмування. Акторна модель (модель акторів), що набирає популярності завдяки системам Akka/Erlang та Orleans/Swift [26], стала привабливою завдяки відмовостійкості та простоті масштабування. Паралельно розвивались моделі транзакційної пам'яті, спрямовані на зменшення складності керування конкурентним доступом. Дистанційні виклики, оптимізовані в таких системах, як Manta [16], відкрили можливості для швидкого виконання розподілених задач без значних витрат на мережеву взаємодію.

У сучасних програмних системах одночасність і паралелізм більше не сприймаються як окремі технічні прийоми, а інтегруються на всіх рівнях – від апаратного забезпечення та операційних систем до мов програмування й фреймворків. Еволюція моделей продовжується під впливом хмарних обчислень, мікросервісної архітектури, подієво-орієнтованих систем та великомасштабних платформ оброблення даних. Фокус зміщується з локального паралелізму до глобальної оркестрації обчислень, де зручність використання стає не менш важливою, ніж продуктивність. Усе це свідчить про те, що розвиток підходів до одночасності та паралелізму є безперервним процесом, який відображає потреби програмної інженерії та постійний пошук рівноваги між швидкодією, масштабованістю й простотою моделювання.

1.2. Класифікація моделей одночасного та паралельного програмування

Класифікація моделей одночасного та паралельного програмування охоплює широкий спектр підходів, які відрізняються принципами організації виконання, способом взаємодії між обчислювальними сутностями та механізмами керування доступом до даних. Потреба в такій класифікації виникла через значну різноманітність сучасних архітектур, де поєднуються багатоядерні процесори, графічні прискорювачі, розподілені середовища та хмарні платформи. Кожен підхід пропонує унікальний спосіб структурування паралельних обчислень, що дозволяє ефективно вирішувати певний клас задач, проте не завжди може бути універсальним. Тому класифікація допомагає визначити межі застосовності тієї чи іншої моделі та оцінити її потенціал щодо продуктивності, масштабованості й зручності програмування.

Для початку розглянемо різницю між одночасним (послідовним) та паралельним підходами, що винесено в тему кваліфікаційної роботи.

На рис. 1.1 показано дві моделі для конкурентного та паралельного піходу.



Рис. 1.1 Моделі послідовного та паралельного підходу.

На рис. 1.1 зверху зображено паралельну модель програмування, а внизу – конкурентну, послідовну. У паралельній моделі можливе одночасне виконання різних задач різними ядрами процесора, тоді як у послідовній моделі завдання виконуються по черзі одним ядром. Треба зазначити, що термін конкурентна або послідовна відноситься скоріше до логіки програми, а не до фізичної реалізації.

Однією з найпоширеніших груп моделей є потокові або thread-based системи, які ґрунтуються на поділі задачі на окремі потоки виконання в межах одного процесу. Вони забезпечують гнучкість і дають можливість реалізувати паралелізм на рівні системного API, але потребують ручного керування синхронізацією за допомогою м'ютексів, семафорів, блокувань та атомарних операцій. Висока складність керування залежностями та ризики взаємоблокувань роблять цей підхід більш придатним для досвідчених розробників, але він залишається базовою основою для багатьох сучасних систем. Інший важливий напрям – моделі, орієнтовані на обмін повідомленнями, що представлені насамперед акторною моделлю. У межах цих підходів кожен об'єкт або актор працює незалежно й взаємодіє з іншими через передачу повідомлень, що істотно спрощує проектування паралельних програм та зменшує ризики некоректного доступу до спільних даних.

Окрему групу становлять моделі, що використовують абстракції спільної пам'яті, але намагаються мінімізувати складність взаємодії між потоками. До таких систем належить транзакційна пам'ять, яка застосовує ідеї із теорії баз даних для вирішення конфліктів доступу. Транзакційний підхід дозволяє програмісту описувати операції над даними як атомарні блоки, знімаючи з нього відповідальність за керування блокуваннями. Подібні концепції лежать в основі деяких сучасних високорівневих мов та фреймворків, які приховують від розробника низькорівневі деталі синхронізації. Поряд із цим значного розвитку набули моделі з розподіленою пам'яттю, такі як Bulk Synchronous Parallel (BSP) чи Orca, які передбачають поділ задачі між ізольованими обчислювальними вузлами з чітко визначеними

фазами обміну даними. Вони добре підходять для великих кластерних систем і забезпечують спрогнозовану продуктивність.

Суттєве місце у класифікації займають моделі, орієнтовані на гетерогенні обчислення, насамперед GPU-орієнтовані підходи, де паралелізм досягається завдяки масивному виконанню багатьох однотипних операцій. Для таких систем розробляються спеціальні паралельні структури даних, як-от Boundary-Aware Concurrent Queue, що оптимізовані для специфічної пам'яті та архітектури обчислювальних блоків графічних процесорів. Паралельно існують автоматизовані моделі, зокрема Jade або Cyrus, що намагаються виявляти паралелізм автоматично, аналізуючи залежності між даними або фіксуючи взаємодії в системі. Завдяки цьому паралельне виконання стає доступнішим без суттєвого ускладнення програмного коду.

Сучасна класифікація моделей паралелізму не є фіксованою та продовжує розширюватися під впливом розвитку апаратного забезпечення, розподілених платформ і хмарних сервісів. Усе частіше моделі поєднують елементи різних підходів, утворюючи гібридні системи, які одночасно використовують потоки, актори, GPU-ядра та механізми транзакційної пам'яті. Така інтеграція дозволяє забезпечити баланс між продуктивністю та простотою моделювання, що стає вирішальним фактором для сучасної інженерії програмного забезпечення. У підсумку класифікація моделей одночасності та паралелізму є ключовим інструментом для аналізу їхньої ефективності, оскільки допомагає вибрати оптимальний підхід відповідно до вимог конкретної задачі та особливостей обчислювального середовища.

1.3 Аналіз ключових парадигм та моделей паралельного програмування

Вибір правильної моделі паралельного програмування є стратегічно важливим для досягнення високої продуктивності та ефективності на сучасних обчислювальних системах. Від цього вибору залежить не лише швидкість виконання програми, але й складність її розробки, портативність та можливість подальшого масштабування. Сучасний ландшафт паралельних архітектур – від багатоядерних процесорів до великомасштабних гетерогенних кластерів та реконфігурованих систем – породив різноманіття програмних моделей, кожна з яких пропонує свої унікальні переваги та компроміси.

Проведемо порівняльний аналіз ключових парадигм паралельного програмування. Аналіз охоплює як усталені стандарти, такі як MPI та OpenMP, що є основою для високопродуктивних обчислень, так і високорівневі абстрактні моделі, спрямовані на спрощення розробки, та спеціалізовані підходи для новітніх архітектур, таких як графічні процесори (GPU) та програмовані логічні інтегральні схеми (FPGA).

Основою для різноманітних моделей паралельного програмування є фундаментальні архітектурні відмінності обчислювальних систем та підходи до керування паралелізмом. Дихотомії "спільна проти розподіленої пам'яті" та "явне проти неявного управління" визначають ключові компроміси, на яких будується кожна модель. Розуміння цих базових концепцій дозволяє системно оцінити переваги та недоліки конкретних реалізацій, від низькорівневих інтерфейсів до високорівневих мов.

Архітектури зі спільною пам'яттю надають усім процесорам доступ до єдиного адресного простору. Моделі, що базуються на цій концепції, такі як OpenMP, дозволяють процесам взаємодіяти шляхом читання та запису у спільні змінні. Це природно відображається на сучасні багатоядерні процесори, де ядра мають спільний доступ до оперативної пам'яті.

На противагу цьому, системи з розподіленою пам'яттю складаються з вузлів, кожен з яких має власний приватний адресний простір. Взаємодія між процесами на різних вузлах відбувається виключно через явну передачу повідомлень мережею. Стандартом де-факто для таких систем є Інтерфейс передачі повідомлень (MPI).

Сучасні високопродуктивні кластери є ієрархічними: вони складаються з вузлів, кожен з яких є багатоядерною системою зі спільною пам'яттю. Для ефективного використання таких архітектур необхідні гібридні моделі, що поєднують передачу повідомлень між вузлами (MPI) та багатопоточність у межах одного вузла (наприклад, OpenMP або pthreads). Як показано в дослідженні паралельного обчислення ліній току, такий підхід дозволяє зменшити обсяг мережових комунікацій та надлишкових операцій вводу-виводу завдяки використанню спільного кешу на вузлі [27].

Існують також спроби абстрагувати апаратну модель пам'яті від програміста за допомогою систем розподіленої спільної пам'яті (Distributed Shared Memory, DSM). Такі системи, як Orca, емулюють спільний адресний простір на апаратурі з розподіленою пам'яттю. Об'єктна модель Orca з механізмом "передачі функцій" (function shipping) часто виявляється більш ефективною за сторінкові DSM-системи, оскільки надсилає менше повідомлень та даних, передаючи лише операцію та її параметри замість повного стану об'єкта [22].

Явний паралелізм покладає на програміста повну відповідальність за управління паралельними процесами, включаючи їх створення, синхронізацію та комунікацію. Моделі, такі як MPI, надають розробнику максимальний контроль над апаратними ресурсами, що потенційно дозволяє досягти найвищої продуктивності. Однак це вимагає значних зусиль, глибокого розуміння архітектури та призводить до складного, багатослівного коду [24].

Неявний паралелізм, навпаки, має на меті приховати складність управління паралельними потоками. У цьому підході компілятор або середовище виконання автоматично виявляють можливості для паралельного

виконання та керують ним. Наприклад, мова Jade зберігає семантику послідовної програми, а паралелізм виявляється автоматично на основі наданих програмістом специфікацій доступу до даних. Перевагою такого підходу є значне спрощення розробки, підвищення портативності та детермінізм виконання. Проте, автоматичне управління може вносити додаткові накладні витрати на аналіз залежностей та обмежувати гнучкість оптимізації [25].

Ці фундаментальні дихотомії формують спектр компромісів, на основі яких розробляються практичні моделі програмування, що детально розглядаються в наступних розділах.

Історично модель передачі повідомлень стала стандартом для великомасштабних паралельних обчислень, оскільки вона прямо відображає фізичну архітектуру кластерних систем. Її найпоширенішою реалізацією є стандарт MPI, який забезпечує портативність та високу продуктивність. Стратегічна важливість цієї моделі полягає в тому, що вона слугує не лише самостійним інструментом, але й основою для багатьох сучасних гібридних та високорівневих систем, які використовують MPI як надійний транспортний рівень.

MPI базується на кількох ключових концепціях. Програма складається з набору процесів, кожен з яких має власний ізольований адресний простір. Процеси об'єднані в групи, що називаються комунікаторами, і взаємодіють виключно шляхом надсилання та отримання повідомлень. Стандарт визначає два основні типи комунікацій: точка-точка (між двома процесами) та колективні (між усіма процесами в комунікаторі, наприклад, розсилка або редукація). Сильні сторони MPI: програміст повністю контролює передачу даних, що дозволяє досягти високої продуктивності шляхом тонкої оптимізації; модель добре масштабується на тисячі й більше процесорів; MPI забезпечує переносимість коду між різними HPC-платформами.

Розробка MPI-програм вимагає від програміста вручну керувати всіма аспектами комунікації та синхронізації. Програміст несе повну

відповідальність за правильну організацію комунікацій для уникнення ситуацій, коли процеси нескінченно очікують один на одного.

Роль MPI як фундаментального шару для інших систем підтверджується кількома джерелами. Наприклад, система OMPC (OpenMP Cluster) використовує MPI як базовий транспортний рівень для передачі даних між вузлами кластера [30]. Реалізація TMPI розроблена для оптимізації виконання MPI-програм на машинах зі спільною пам'яттю шляхом перетворення процесів у потоки [28]. Дослідження гібридних моделей використовує реалізацію "MPI-only" як базовий варіант для порівняння ефективності [27].

Необхідність гібридних моделей зумовлена архітектурою сучасних HPC-кластерів, які складаються з багатоядерних вузлів. Використання лише MPI (один процес на ядро) на таких системах є неефективним, оскільки ігнорує можливість швидкого обміну даними через спільну пам'ять у межах одного вузла. Гібридний підхід, що поєднує MPI для міжвузлової комунікації та OpenMP (або pthreads) для паралелізму всередині вузла, дозволяє краще відповідати цій ієрархічній структурі.

Переваги гібридного підходу наочно продемонстровані в дослідженні паралельного обчислення ліній току [27], де порівнювалися реалізації MPI-only та MPI-hybrid:

- зменшення надлишкових операцій вводу-виводу: у гібридній моделі потоки на одному вузлі спільно використовують кеш даних, завантажуючи кожен блок даних лише один раз, тоді як у моделі MPI-only кожен процес завантажує дані окремо;
- зниження обсягу мережевих комунікацій: оскільки кожен вузол містить більше даних, ймовірність того, що обчислення продовжиться в межах того ж вузла, зростає, що зменшує потребу в передачі даних між вузлами;
- зменшення вузьких місць: у моделі MPI-only, коли багато процесів одночасно обробляють один блок даних, процес-власник цього блоку стає вузьким місцем. У гібридній моделі кілька потоків можуть одночасно працювати з цим блоком у спільній пам'яті, що усуває цей недолік.

Таким чином, гібридний підхід дозволяє більш ефективно використовувати ієрархічну структуру пам'яті та обчислювальних ресурсів сучасних суперкомп'ютерів.

Архітектурна залежність та складність програмування, притаманні MPI та гібридним моделям, стимулювали розробку абстракцій вищого рівня, призначених для перекладання тягаря управління паралелізмом з програміста на середовище виконання.

Складність розробки, властива низькорівневим підходам, стала рушійною силою для створення моделей програмування вищого рівня абстракції. Їхня головна мета – знизити когнітивне навантаження на розробника, перекладаючи відповідальність за управління комунікацією та синхронізацією на середовище виконання. Такі моделі прагнуть спростити процес написання коду, підвищити його портативність та надійність, намагаючись при цьому зберегти високу продуктивність.

Паралелізм на основі задач (Task-Based Parallelism) представляє обчислення у вигляді орієнтованого ациклічного графа (Directed Acyclic Graph, DAG), де вузли – це обчислювальні задачі, а ребра – залежності по даних між ними. Середовище виконання динамічно планує виконання задач на доступних ресурсах, як тільки їхні залежності стають задоволеними.

OpenMP Cluster (OMPC) є прикладом розширення знайомої моделі OpenMP для роботи на кластерних системах. OMPC намагається забезпечити простоту моделей зі спільною пам'яттю, таких як OpenMP, використовуючи при цьому масштабованість MPI. Ця модель приховує складність MPI-комунікацій за директивами OpenMP, дозволяючи програмістам "вивантажувати" задачі на віддалені вузли кластера аналогічно до того, як вони вивантажують їх на GPU. Порівняння продуктивності показує, що OMPC демонструє конкурентоспроможність, значно випереджаючи синхронні реалізації MPI, але може поступатися більш зрілим асинхронним середовищам виконання задач, таким як StarPU, через додаткові накладні витрати на абстракцію [30].

Jade – це мова неявного паралелізму, яка зберігає семантику послідовної програми. Її ключова особливість полягає у використанні специфікацій доступу до даних, які програміст додає до коду. Середовище виконання аналізує ці специфікації, щоб автоматично виявити залежності між задачами та виконати їх паралельно. Цей підхід забезпечує важливі переваги, такі як детермінізм (результат не залежить від розкладу виконання) та портативність. Водночас він має недоліки: накладні витрати на динамічну перевірку доступу та потенційні проблеми з вибором правильної гранулярності задач [25].

Розглянемо ще один підхід: об'єктно-орієнтований паралелізм (ООР). Поєднання об'єктно-орієнтованого програмування (ООР) з паралелізмом мотивоване тим, що механізми інкапсуляції та абстракції можуть спростити розробку складних паралельних систем, приховуючи деталі синхронізації та комунікації всередині об'єктів [24].

Orca є прикладом того, як принципи ООР можуть бути застосовані для спрощення паралельного програмування. Це портативна об'єктно-орієнтована система розподіленої спільної пам'яті (DSM), де спільні дані інкапсулюються в об'єкти. Замість того, щоб програміст керував передачею повідомлень, Orca автоматично підтримує когерентність об'єктів. Система використовує протокол оновлення з передачею функцій (function shipping): замість передачі повного стану зміненого об'єкта мережею передається лише операція та її параметри. Це робить Orca значно ефективнішою за сторінкові DSM-системи, оскільки вона надсилає менше повідомлень та даних [21].

Моделі на основі взаємодіючих процесів (CSP та Актори) представляють паралельну програму як сукупність незалежних процесів (або акторів), що взаємодіють через канали або асинхронні повідомлення.

Occam- π є мовою, що базується на формалізмі Communicating Sequential Processes (CSP) та пі-численні. Вона розроблена для програмування масивно-паралельних реконфігурованих архітектур, таких як Ambria. На відміну від класичного CSP, який моделює лише статичні мережі, пі-числення дозволяє

динамічно створювати та змінювати канали зв'язку, що робить Оссам-рі гнучким інструментом для програмування реконфігурованих систем [32].

Хоча високорівневі моделі спрощують розробку загальноцільових програм, досягнення пікової продуктивності на новітніх апаратних платформах вимагає ще більшої спеціалізації програмних підходів.

Для повного використання унікальних можливостей сучасних апаратних прискорювачів, таких як графічні процесори (GPU) та програмовані логічні інтегральні схеми (FPGA), необхідна розробка спеціалізованих моделей програмування. Ці моделі часто відмовляються від високого рівня абстракції на користь прямого доступу до апаратних особливостей, вимагаючи від програміста глибокого розуміння специфіки платформи для досягнення пікової продуктивності.

Програмування графічних процесорів (GPU) оптимізована для масивно-паралельної обробки даних (SIMD/SIMT). Модель програмування CUDA від NVIDIA надає набір абстракцій для ефективного використання цієї архітектури.

Ключові абстракції CUDA:

- Ієрархія потоків: обчислення організовані у вигляді сітки (grid) з блоків (blocks), кожен з яких містить потоки (threads). Потоки в межах одного блоку можуть синхронізуватися та обмінюватися даними через швидкодію спільну пам'ять. Потоки виконуються групами по 32, що називаються варпами (warps).

- Ієрархія пам'яті: CUDA надає доступ до різних типів пам'яті: глобальної (великий обсяг, висока затримка), спільної (швидка, обмежена одним блоком) та приватної для кожного потоку.

Обчислювальні ядра (kernels): програміст визначає функції-ядра, які виконуються на GPU паралельно тисячами потоків [7].

Аналіз реалізації алгоритму PCM на CUDA демонструє переваги GPU для задач з високим ступенем паралелізму даних. Порівняння з реалізацією на багатоядерних CPU з використанням OpenMP показує, що GPU є надзвичайно

ефективним, коли всі потоки у варпі виконують однаковий шлях коду. Якщо ж у кодї є розгалуження, що змушують потоки одного варпу виконувати різні інструкції, відбувається серіалізація виконання, що знижує продуктивність. CPU, навпаки, надає більшу гнучкість, оскільки кожне ядро є незалежним і може виконувати довільний код [7].

Програмування реконфігурованих архітектур (FPGA) пропонує максимальну гнучкість, дозволяючи створювати спеціалізовані апаратні конвеєри для конкретних алгоритмів. Однак традиційне програмування FPGA за допомогою мов опису апаратури (HDL) є складним і трудомістким процесом. Високорівневий синтез (High-Level Synthesis, HLS) значно спрощує цей процес, дозволяючи генерувати HDL-код з мов високого рівня, таких як C++.

Для подальшого підвищення продуктивності та створення високоефективних паралельних структур на FPGA був запропонований підхід, що базується на функціональних патернах, таких як MapReduce. Ці патерни реалізовані у вигляді C++ шаблонів, які використовуються в середовищі HLS.

Принцип роботи: програміст описує алгоритм як композицію функціональних операторів (Map, Reduce, ZipWith), застосованих до потоків даних. Бібліотека шаблонів автоматично генерує апаратну реалізацію у вигляді глибоко конвеєризованої архітектури.

Цей підхід дозволяє створювати високопродуктивні паралельні конвеєри з ініціалізаційним інтервалом (Initiation Interval, II) – кількістю тактів між початком послідовних ітерацій у конвеєрі – рівним 1. Це означає, що конвеєр може приймати нові дані в кожному такті, досягаючи максимальної пропускної здатності. При цьому програміст абстрагується від низькорівневих деталей реалізації, таких як керування станами скінченного автомату чи синхронізація даних [1].

Однак, наскільки точно теоретичні моделі та високорівневі абстракції відповідають реальній продуктивності на конкретному обладнанні, залишається важливим питанням, що потребує окремого розгляду.

Вибір моделі паралельного програмування є складним багатовимірним компромісом, що залежить від багатьох факторів. Не існує універсального рішення, і кожна модель пропонує свій баланс між продуктивністю, складністю розробки, портативністю та масштабованістю. Цей розділ систематизує та порівнює проаналізовані моделі за ключовими критеріями, щоб надати цілісне уявлення про їхні сильні та слабкі сторони.

Розглянуті моделі можна умовно розташувати на спектрі від низькорівневих до високорівневих. На низькому рівні знаходяться такі моделі, як MPI та CUDA. Вони надають програмісту максимальний контроль над апаратними ресурсами, дозволяючи досягати пікової продуктивності за рахунок тонкої оптимізації комунікацій, розподілу даних та управління пам'яттю. Однак ціною за такий контроль є надзвичайно висока складність розробки, великий обсяг коду та необхідність глибоких знань про цільову архітектуру.

На високому рівні знаходяться моделі, що мають на меті спростити розробку шляхом абстракції. Jade, OMPC та функціональні патерни для HLS є яскравими прикладами. Хоча OMPC та Jade обидві абстрагують комунікації, вони представляють дві різні філософії: OMPC розширює знайомий синтаксис спільної пам'яті (OpenMP) на розподілене середовище, тоді як Jade переосмислює контракт між програмістом і середовищем виконання, використовуючи специфікації доступу до даних для виявлення паралелізму, неявного в послідовному коді. Ці абстракції значно прискорюють розробку та підвищують портативність, але можуть вносити додаткові накладні витрати та обмежувати гнучкість оптимізації.

Гібридні моделі (MPI+OpenMP) та розширення на кшталт OMPC є спробою знайти золоту середину. Вони поєднують знайомий і відносно простий синтаксис (наприклад, директиви OpenMP) з потужністю низькорівневих реалізацій (наприклад, MPI як транспортний рівень), намагаючись запропонувати баланс між продуктивністю та простотою використання.

Для аналізу та проектування паралельних алгоритмів часто використовуються абстрактні обчислювальні моделі, такі як BSP (Bulk Synchronous Parallel) та BPRAM. Вони надають формальний апарат для оцінки складності алгоритмів, враховуючи вартість комунікацій та синхронізації. Однак виникає питання про їхню практичну цінність для точного прогнозування реального часу виконання на конкретному обладнанні.

Дослідження [12] продемонструвало, що прогнози, зроблені на основі моделей BSP та BPRAM, можуть суттєво відрізнятися від реальних показників продуктивності. На п'яти різних паралельних платформах були зафіксовані похибки у прогнозуванні, що сягали 200%. Причини таких розбіжностей полягають у тому, що моделі не враховують багатьох аспектів реальних систем:

- складність ієрархії пам'яті та ефекти кешування;
- конкуренцію за мережеві ресурси та топологію мережі;
- накладні витрати операційної системи та середовища виконання.

Аналіз джерел дозволяє виділити кілька ключових тенденцій, що визначають сучасний стан та майбутній розвиток паралельного програмування:

1. Гібридизація: для ефективного використання сучасних ієрархічних суперкомп'ютерів, що складаються з багатоядерних вузлів, комбінування моделей (наприклад, MPI для міжвузлової взаємодії та OpenMP/threads для паралелізму всередині вузла) стало не просто можливістю, а необхідністю. Це дозволяє оптимально використовувати як розподілену, так і спільну пам'ять.

2. Абстракція: спостерігається стійкий рух у напрямку моделей вищого рівня, які приховують складність управління паралелізмом від програміста. Парадигми, засновані на задачах (task-based), як-от OMPC, або на неявному паралелізмі, як-от Jade, знижують когнітивне навантаження на розробника, автоматизуючи планування, синхронізацію та передачу даних. Це спрощує розробку, хоча іноді й ціною певних накладних витрат.

3. Спеціалізація: поява нових апаратних архітектур, таких як GPU та FPGA, стимулює розробку вузькоспеціалізованих моделей програмування (CUDA, HLS з функціональними патернами). Ці моделі надають прямий доступ до унікальних можливостей апаратури, що є критичним для досягнення максимальної продуктивності в таких галузях, як машинне навчання, наукові симуляції та обробка сигналів.

Майбутнє паралельного програмування, ймовірно, полягатиме у співіснуванні різноманітних моделей, кожна з яких займатиме свою нішу. Ключовим викликом стане розробка інструментів та середовищ, що дозволять легко та ефективно поєднувати різні моделі в рамках однієї складної програми, забезпечуючи як високу продуктивність, так і продуктивність розробників.

1.4. Проблеми взаємодії, синхронізації та узгодженості даних у різних моделях

Проблеми взаємодії, синхронізації та узгодженості даних є визначальними для будь-якої моделі одночасного та паралельного програмування, оскільки саме вони визначають коректність роботи програмних систем та їх здатність ефективно масштабуватися. У випадку традиційних потокових моделей головна складність полягає в керуванні доступом до спільної пам'яті. Потоки, що виконуються паралельно, можуть змагатися за доступ до тих самих даних, створюючи неконтрольовані конфлікти, що призводять до гонок даних, непередбачуваних станів або втрати інформації. Семафори, м'ютекси, блокування та інші механізми синхронізації лише частково розв'язують ці проблеми, часто породжуючи інші ризики, зокрема взаємоблокування, голодування потоків або надмірні накладні витрати, які значно погіршують продуктивність. Такі ризики роблять багатопотокове програмування складним навіть для досвідчених фахівців і вимагають від розробника глибокого розуміння внутрішніх механізмів операційних систем.

У моделях, що ґрунтуються на обміні повідомленнями, частина цих проблем знімається завдяки ізоляції станів окремих сутностей. Акторна модель забезпечує, що кожен актор працює зі своєю локальною пам'яттю, а взаємодія з іншими здійснюється виключно через черги повідомлень. Проте й тут існують труднощі, пов'язані з узгодженістю порядку повідомлень, затримками доставки та визначенням гарантій доставки, що є критично важливими для розподілених систем. У таких випадках виникає потреба у спеціальних протоколах, наприклад, гарантованої доставки, і механізмах оброблення помилок, що збільшують складність моделі. Канальні системи, як-от Communicating Sequential Processes, також потребують чіткої координації порядку взаємодії, оскільки неправильне проектування каналів може призвести до блокувань або порушення узгодженості.

У розподілених системах проблеми синхронізації та узгодженості даних стають ще складнішими через відсутність глобального годинника, непередбачуваність мережових затримок і ризики часткових збоїв окремих вузлів. Тут на перший план виходить CAP-теорема, яка стверджує, що одночасно можна досягти лише двох із трьох властивостей: узгодженості, доступності або стійкості до розділення. Моделі узгодженості, як-от послідовна, причинна або Eventual Consistency (з якою працюють NoSQL-бази даних), вибираються відповідно до компромісів CAP-теореми. Найкритичніші системи використовують алгоритми сильного консенсусу, наприклад, Raft або Paxos, які забезпечують узгодженість ціною підвищеної латентності.

Моделі на кшталт Orca або BSP пропонують механізми узгодження станів через глобальні бар'єри або реплікацію даних, але такі підходи мають свої недоліки – збільшення латентності, необхідність у складних алгоритмах узгодження та ризик виникнення конфліктів під час реплікації. Мережеві виклики, навіть оптимізовані як у системі Manta, все одно залишаються джерелом додаткових накладних витрат, які обмежують масштабованість. Питання узгодженості вирішуються через модель послідовної узгодженості, causal consistency або eventual consistency, але жодна з них не є універсальною, і вибір залежить від специфіки системи.

У моделях автоматичного вилучення паралелізму та в апаратно-орієнтованих паралельних середовищах також виникають проблеми з узгодженістю даних. Наприклад, Jade аналізує залежності між операціями, але може некоректно оцінити взаємодію у випадках складних патернів доступу, що призводить до втрати паралелізму або помилкових конфліктів. У GPU-середовищах, де сотні та тисячі потоків працюють над спільними масивами даних, критично важливими є правильне структурування пам'яті та уникнення таких явищ, як конфлікти банків пам'яті або розсинхронізація між блоками обчислень. Розроблені структури, зокрема Boundary-Aware Concurrent Queue, частково розв'язують ці проблеми, але вимагають глибокого розуміння внутрішньої архітектури GPU.

Отже, незалежно від моделі, основні проблеми взаємодії та узгодженості даних залишаються одними з найскладніших у сфері одночасного та паралельного програмування. Кожен підхід пропонує власний набір механізмів, які частково зменшують труднощі, але водночас створюють нові виклики. Сучасні тенденції спрямовані на пошук балансів між простотою моделювання, мінімізацією накладних витрат синхронізації та забезпеченням високої продуктивності, що підкреслює важливість подальших досліджень у цьому напрямі.

1.5. Критерії оцінювання ефективності: продуктивність, масштабованість, зручність використання

Оцінювання ефективності моделей одночасного та паралельного програмування є ключовим етапом аналізу їх придатності для розв'язання різних класів задач у сфері інженерії програмного забезпечення. Визначення чітких критеріїв дозволяє не лише порівнювати моделі між собою, а й розуміти їхні межі застосування та потенційні переваги у конкретних обчислювальних умовах. У центрі цього оцінювання зазвичай знаходяться три фундаментальні характеристики: продуктивність, масштабованість та зручність використання. Кожен із цих критеріїв має власну специфіку, залежить від архітектури моделі та середовища виконання і може по-різному впливати на загальну ефективність програмної системи.

Продуктивність є найважливішим і найбільш вимірюваним критерієм, оскільки саме вона відображає здатність моделі забезпечувати мінімальний час виконання задачі або максимальну пропускну здатність системи. На продуктивність впливають накладні витрати на синхронізацію, контекстне перемикання, обмін повідомленнями, реплікацію стану та інші операції, які залежать від конкретної моделі. Наприклад, традиційні поточкові моделі демонструють високий потенціал продуктивності за правильного керування блокуваннями, але можуть втрачати переваги через часті конфлікти доступу. Акторні моделі ефективні у випадках великої кількості незалежних задач, проте їхня продуктивність може знижуватися через затримки у чергах повідомлень. Розподілені системи, зокрема Orca чи Manta, залежать від мережових комунікацій, які створюють латентність і впливають на загальну швидкодію. GPU-орієнтовані підходи демонструють високу продуктивність лише в умовах масивного паралелізму, тоді як дрібні задачі можуть виконуватися повільніше через витрати на передавання даних.

Масштабованість є другим ключовим критерієм, який визначає здатність моделі ефективно використовувати зростаючу кількість

обчислювальних ресурсів без суттєвих втрат продуктивності. Моделі потоків масштабуються переважно вертикально, залежно від кількості ядер у системі, але погано розширюються на розподілені середовища через високі витрати синхронізації. Натомість акторна модель та підходи, засновані на обміні повідомленнями, демонструють кращу горизонтальну масштабованість завдяки ізоляції станів і можливості розподіляти акторів між вузлами. Такі системи, як BSP або сучасні GPU-моделі, оптимізовані саме для великомасштабних обчислень і дають змогу прогнозувати поведінку програми при збільшенні кількості потоків чи ядер. Автоматизовані моделі, зокрема Jade чи Cygus, також орієнтовані на масштабованість, але їх ефективність залежить від якості аналізу залежностей, який може стати обмеженням у складних сценаріях.

Зручність використання є менш формалізованим, але не менш значущим критерієм, оскільки він безпосередньо впливає на продуктивність розробника та ймовірність помилок у програмі. Моделі, що вимагають ручного керування блокуваннями або складної координації потоків, мають високу когнітивну складність і потребують значних зусиль для підтримки та відлагодження. У цьому контексті більш привабливими виявляються акторні та функціональні моделі, що мінімізують спільний стан і пропонують декларативний стиль опису паралельної поведінки. Розподілені моделі, навіть попри свою масштабованість, часто мають низьку зручність через складність у керуванні зв'язками між вузлами, обробленні відмов та узгодженості реплік. GPU-моделі, хоча й забезпечують високу продуктивність, вимагають від розробника глибоких знань специфіки апаратної архітектури, що обмежує їх широке використання поза високопродуктивними обчислювальними застосунками.

Таким чином, оцінювання моделей паралельного та одночасного програмування потребує комплексного підходу, що враховує особливості кожного критерію та їх взаємозв'язок. Висока продуктивність може бути досягнута на шкоду зручності, а хороша масштабованість не гарантує

мінімальної латентності чи простоти реалізації. Саме тому порівняльний аналіз моделей, що враховує як теоретичні, так і експериментальні аспекти, є необхідним для створення практичних рекомендацій щодо вибору оптимального підходу в інженерії програмного забезпечення.

Продуктивність (Performance)

Продуктивність описує, наскільки швидко система обробляє заданий обсяг роботи.

Метрики та формули представлено формулами (1.1)-(1.3):

Час відгуку (Response Time):

$$RT = \frac{\sum_{i=1}^n t_i}{n}, \quad (1.1)$$

де t_i - час відгуку для i -го запиту, n - кількість запитів.

Пропускна здатність (Throughput):

$$TP = \frac{N}{T}, \quad (1.2)$$

де N – кількість оброблених операцій, T – час.

Використання ресурсів (CPU, RAM):

$$U = \frac{R_{used}}{R_{total}} \cdot 100\%, \quad (1.3)$$

Методика: проводяться навантажувальні тести (load testing), вимірюється середній час відгуку, кількість транзакцій за секунду, рівень використання ресурсів.

Масштабованість (Scalability)

Масштабованість описує здатність системи ефективно нарощувати продуктивність при збільшенні обсягу роботи або додаванні ресурсів (процесорів, серверів), здатність системи ефективно працювати при зростанні навантаження.

Метрики та формули надано у формулах (1.4) та (1.5):

Коефіцієнт масштабованості:

$$S = \frac{P_n}{P_1 \cdot n}, \quad (1.4)$$

де P_n – продуктивність при вузлах, P_1 – продуктивність при одному вузлі.

Еластичність (Elasticity):

$$E = \frac{C_{actual}}{C_{expected}}, \quad (1.5)$$

де C_{actual} – фактична продуктивність після масштабування, $C_{expected}$ – теоретично очікувана.

Методика: моделюється збільшення кількості користувачів або обсягу даних, вимірюється відхилення від лінійного зростання продуктивності.

Зручність використання (Usability)

Зручність використання (юзабіліті) описує, наскільки ефективно, результативно та приємно користувачі можуть досягати визначених цілей при роботі з системою, наскільки легко користувачі взаємодіють із системою. Визначається через Юзабіліті-тестування та опитувальники.

Метрики та формули: (1.6)-(1.8).

Час виконання завдання:

$$T_{avg} = \frac{\sum_{i=1}^n t_i}{n}, \quad (1.6)$$

Коефіцієнт успішності (Success Rate):

$$SR = \frac{N_{success}}{N_{total}} \cdot 100\% \quad (1.7)$$

Коефіцієнт помилок (Error Rate):

$$ER = \frac{N_{errors}}{N_{total}} \cdot 100\% \quad (1.8)$$

Суб'єктивна оцінка (SUS – System Usability Scale): анкетування користувачів, середній бал за шкалою 0–100.

Методика: юзабіліті-тестування з реальними користувачами, вимірювання часу виконання завдань, кількості помилок, анкетування.

Формулу інтегральної оцінки наведено формолою (1.9):

$$IE = w_p \cdot P + w_s \cdot S + w_u \cdot U, \quad (1.9)$$

де:

- IE – інтегральна оцінка ефективності (0–1 або 0–100%).
- F – нормалізоване значення продуктивності.
- S – нормалізоване значення масштабованості.
- U – нормалізоване значення зручності використання.
- w_u – вагові коефіцієнти (сума дорівнює 1).

Розділ 2. Аналіз сучасних моделей одночасного та паралельного програмування

2.1. Потокові, конкурентні та акторні моделі: принципи, переваги та обмеження

Потокові, конкурентні та акторні моделі становлять фундамент сучасних підходів до організації паралельного виконання програм, забезпечуючи різні способи структуризації взаємодії між обчислювальними одиницями та керування спільним станом.

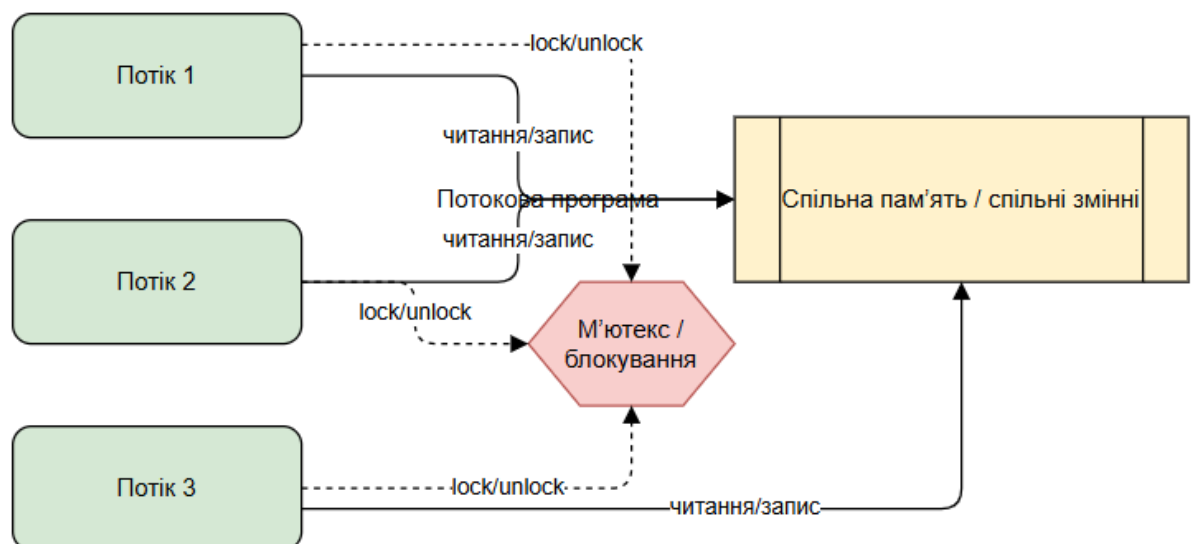


Рис. 2.1 Потокова модель.

Потокова модель (рис 2.1) є найбільш традиційною і ґрунтується на використанні потоків операційної системи, які виконуються паралельно в межах одного процесу та мають спільний доступ до пам'яті. Основний принцип цієї моделі полягає в поділі задачі на декілька незалежних шляхів виконання, кожен з яких може обробляти власну частину даних. Водночас необхідність синхронізації між потоками через блокування, семафори та інші примітиви створює значні ризики взаємоблокувань, гонок даних і надмірних накладних витрат. Перевагою потокової моделі є прямий контроль над

виконанням та низький оверхед взаємодії у випадку правильно організованої синхронізації, проте її обмеженнями виступають висока складність реалізації, складність відлагодження та низька масштабованість у великих багатоядерних або розподілених системах.

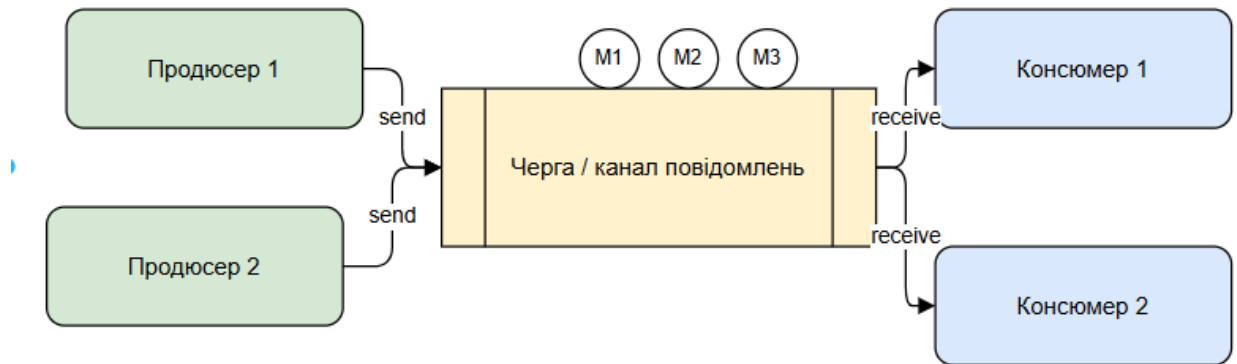


Рис. 2.2. Конкурентна модель

Конкурентні моделі програмування (рис. 2.2) розширюють ідейну основу потокового підходу, пропонуючи абстракції, що дозволяють взаємодіяти між обчислювальними сутностями без прямого доступу до спільної пам'яті. До таких моделей належать підходи, що передбачають взаємодію через канали, черги або події. У межах цих моделей процеси або задачі можуть виконуватися одночасно, але їхня взаємодія керується зовнішніми механізмами, які координують порядок оброблення даних та усувають більшість ризиків, пов'язаних зі спільною пам'яттю. Основним принципом є декомпозиція системи на незалежні компоненти, що спілкуються за допомогою структурованих інтерфейсів. Це знижує ймовірність конфліктів доступу, а також підвищує передбачуваність та відмовостійкість програм. Проте конкурентні моделі часто вимагають складного проектування каналів або черг, потребують додаткових ресурсів на передачу повідомлень і можуть бути чутливими до затримок, особливо в розподілених або мережових середовищах. Незважаючи на це, вони залишаються одним із

найефективніших підходів для побудови модульних, масштабованих програмних систем.

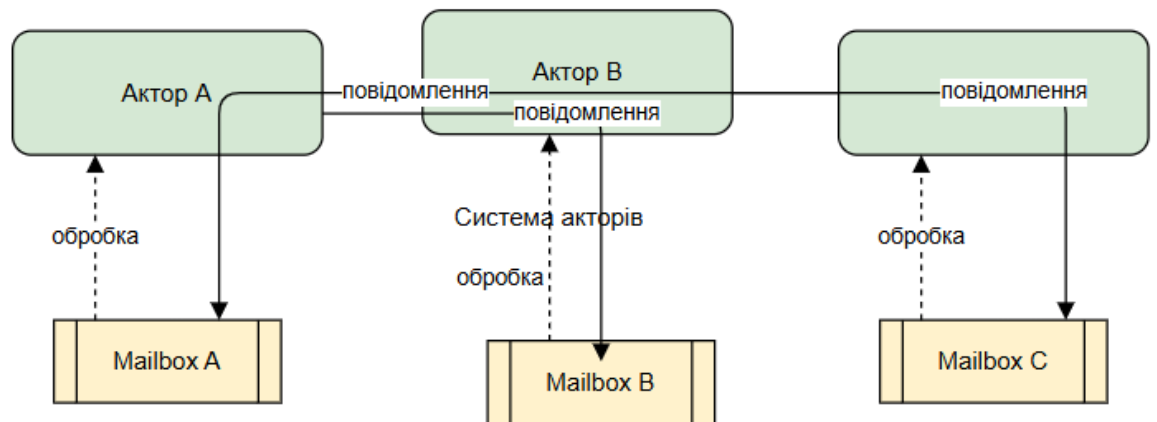


Рис. 2.3. Акторна модель.

Акторна модель (рис. 2.3) представляє собою окремий клас конкурентних моделей і є однією з найбільш перспективних парадигм для масштабованих і розподілених систем. Вона базується на концепції акторів - незалежних обчислювальних сутностей, кожна з яких має власний стан, чергу повідомлень та набір поведінок для реагування на ці повідомлення. Актори не ділять спільної пам'яті, а взаємодіють виключно через асинхронний обмін повідомленнями. Такий принцип забезпечує високу ізоляцію стану, що практично усуває ризики гонок даних і зменшує потребу в блокуваннях. Перевагою акторної моделі є здатність до ефективного горизонтального масштабування, оскільки актори можуть розміщуватися на різних вузлах мережі або на різних потоках без істотних змін у логіці системи. Крім того, акторна модель характеризується високою відмовостійкістю: збій одного актора не впливає на роботу інших, а спеціальні механізми нагляду дозволяють автоматизувати відновлення.

Разом із тим, акторна модель має і свої обмеження. Асинхронність взаємодії ускладнює визначення причинно-часових залежностей та логічний порядок оброблення повідомлень. Система може зіткнутися з непередбачуваними латентностями або зміною порядку доставки повідомлень, що ускладнює забезпечення сильної узгодженості стану. Крім того, моделі обміну повідомленнями можуть мати значні накладні витрати в умовах інтенсивного трафіку або великої кількості дрібних повідомлень. Акторні системи також вимагають спеціальних фреймворків або середовищ виконання, таких як Akka, Orleans або Erlang/OTP, що може підвищити поріг входження для розробників.

У підсумку потокові, конкурентні та акторні моделі демонструють різні компроміси між продуктивністю, гнучкістю та зручністю використання. Потокова модель пропонує максимальний контроль, але створює високу когнітивну складність та потребу в точному керуванні синхронізацією. Конкурентні моделі з каналами або подіями знижують ризики пов'язані зі спільною пам'яттю та підвищують модульність, проте вимагають точного проектування алгоритмів взаємодії. Акторна модель забезпечує ізоляцію, масштабованість і відмовостійкість, але може створювати нові виклики, пов'язані з латентністю та узгодженістю. Порівняння цих підходів дозволяє визначити, які моделі є більш придатними для тих чи інших умов - від багатоядерних процесорів до великих розподілених систем та високонавантажених сервісів.

2.2. Моделі з розподіленою пам'яттю і комунікацією: BSP, BPRAM, Orca, Manta, ossam-pi

Моделі з розподіленою пам'яттю та комунікацією (рис. 2.4) становлять окремий клас підходів до організації паралельних обчислень, у яких обчислювальні вузли мають власний локальний стан, а координація здійснюється через явний обмін повідомленнями або через спеціальні абстракції спільних об'єктів. Їх поява була зумовлена потребою ефективно використовувати багатовузлові системи, кластери та мережі робочих станцій, де відсутня апаратна підтримка глобальної спільної пам'яті. На відміну від класичних поточкових моделей, орієнтованих на єдиний адресний простір, моделі з розподіленою пам'яттю і комунікацією намагаються формально описати взаємодію між процесами, а також надати програмісту прогнозовані характеристики продуктивності та масштабованості. До таких моделей належать Bulk Synchronous Parallel (BSP), BPRAM, а також конкретні системи на основі абстракцій спільних об'єктів і віддалених викликів, зокрема Orca, Manta та мова ossam-pi, яка спирається на ідеї процесів, що взаємодіють через

Кожен вузол має власну локальну пам'ять і обчислення.

Координація виконується через мережу: передавання повідомлень, віддалені виклики, узгодження спільних об'єктів і бар'єри синхронізації.

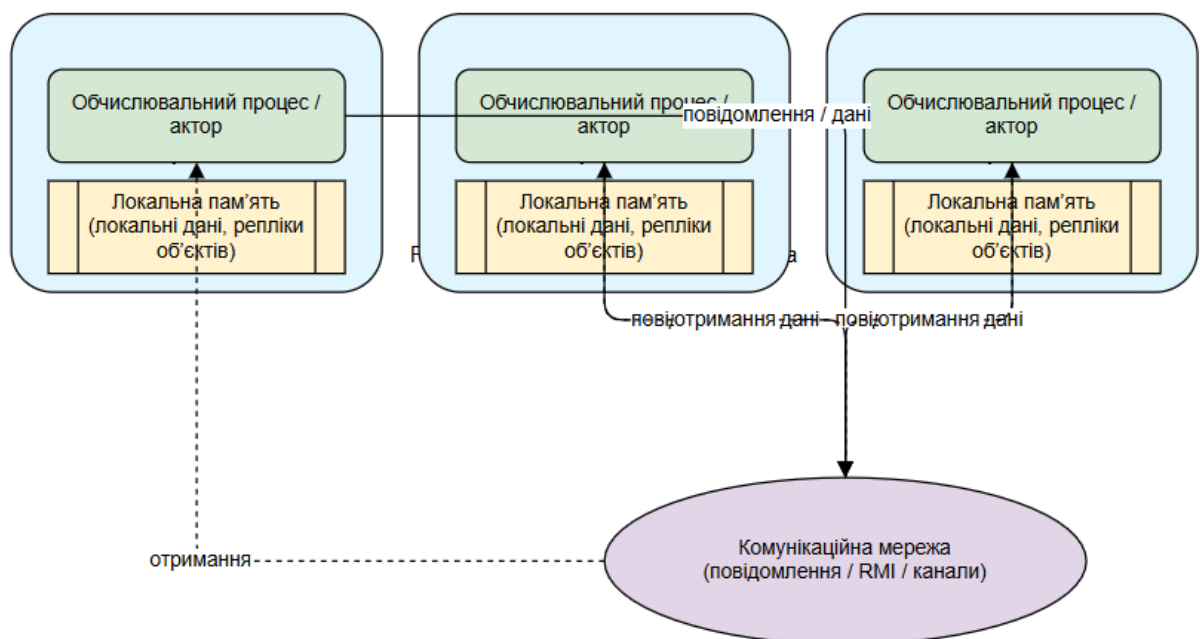


Рис. 2.4. Узагальнена модель з розподіленою пам'яттю та комунікацією

Модель Bulk Synchronous Parallel пропонує структурований підхід до паралельного програмування, у якому обчислення поділяються на послідовність суперетапів. Кожен суперетап складається з локальних обчислень на окремих процесах, фази комунікації, де відбувається обмін даними між процесами, та глобального бар'єра синхронізації. Така організація дозволяє формально оцінювати час виконання, оскільки враховуються три основні компоненти: затрати на обчислення, витрати на комунікацію та вартість синхронізації. BSP суттєво спрощує аналіз алгоритмів для великих кластерів і надає програмісту модель, у якій можна передбачати масштабованість, плануючи розподіл задач між вузлами. Водночас жорсткі бар'єри синхронізації можуть призводити до простоїв, коли швидші процеси очікують на повільніші, що зменшує ефективність у випадках нерівномірного навантаження.

Модель BPRAM розвиває ідеї паралельних абстракцій, поєднуючи уявлення про блоковий розподіл пам'яті та випадковий доступ, що дозволяє досліджувати складність паралельних алгоритмів за умов наявності розподілених ресурсів. У цій моделі розподіл даних між вузлами та структура доступу до них стають ключовими параметрами, від яких залежить ефективність роботи системи. BPRAM використовується насамперед як аналітичний інструмент для оцінювання теоретичної масштабованості та продуктивності алгоритмів у розподіленому середовищі, надаючи можливість порівнювати різні підходи до розподілу даних і організації комунікації. Хоча BPRAM менш поширена як практична платформа розробки, вона відіграє важливу роль у розумінні меж продуктивності паралельних програм на кластерах.

Система Orca реалізує модель об'єктно-орієнтованої розподіленої спільної пам'яті, у якій програміст працює не з низькорівневими повідомленнями, а з абстракціями спільних об'єктів. Кожен об'єкт логічно є спільним для всієї системи, але фізично може бути реплікований або розподілений між вузлами. Доступ до об'єктів здійснюється через

високорівневі операції, а система сама відповідає за узгодження стану, синхронізацію та мінімізацію мережових витрат. Такий підхід наближає програмну модель до традиційного уявлення про спільну пам'ять, але фактично працює в умовах розподілених ресурсів. Перевагою Orca є зменшення когнітивної складності для розробника, оскільки більшість деталей комунікації прихована, однак це може супроводжуватися накладними витратами на прозору реплікацію та узгодження, особливо в умовах інтенсивної взаємодії.

Архітектура Manta представляє ще один важливий напрям - оптимізацію віддалених викликів методів у розподілених системах. Вона фокусується на досягненні дуже низької латентності для RMI-операцій, поєднуючи ефективні транспортні протоколи з оптимізованими механізмами серіалізації та десполітизації даних. Програміст у такій системі працює з віддаленими об'єктами, викликаючи їхні методи так, ніби вони є локальними, тоді як Manta відповідає за маршрутизацію та передачу повідомлень між вузлами. Основна перевага полягає в тому, що віддалені виклики інтегруються в об'єктно-орієнтовану модель програмування, зменшуючи розрив між локальними і розподіленими обчисленнями. Водночас ефективність такого підходу сильно залежить від мережевої інфраструктури, а також від характеру навантаження, оскільки надмірна кількість дрібних RMI-операцій може зменшити загальну продуктивність через мережеві затримки.

Мова ossam-рі поєднує ідеї класичної мови ossam, заснованої на моделі Communicating Sequential Processes, з розширеннями для підтримки динамічних процесів і каналів, запозиченими з π -числення. У ній паралелізм є явним: програміст описує процеси, що виконуються незалежно, і канали, через які вони обмінюються повідомленнями. Особливістю ossam-рі є жорстка дисципліна комунікації, що дозволяє компілятору виконувати глибокий аналіз та гарантувати відсутність певних класів помилок. Така модель добре пристосована до реконфігурованих архітектур і систем, де структура взаємодії може змінюватися під час виконання. Проте суворі правила та специфічний

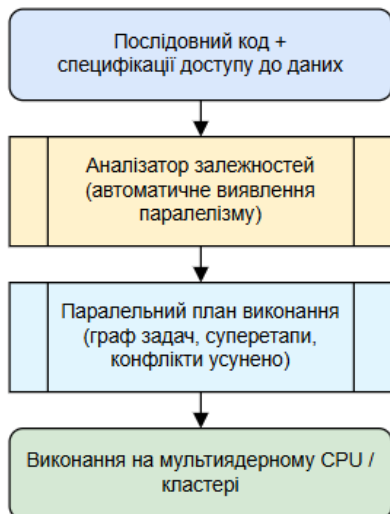
синтаксис збільшують поріг входження для розробників, що обмежує поширення мови у широкій практиці.

Узагальнюючи, моделі з розподіленою пам'яттю і комунікацією - BSP, BPRAM, Orca, Manta, ossam-pi - демонструють різні рівні абстракції та підтримки програміста: від формальних моделей для аналізу алгоритмів до практичних систем розподілених об'єктів та мов програмування. Вони надають засоби керування комунікацією і узгодженістю стану в умовах відсутності єдиного адресного простору, що робить їх ключовими для проектування масштабованих кластерних та розподілених систем. Водночас вибір конкретної моделі пов'язаний із компромісами між продуктивністю, складністю реалізації, вимогами до апаратної платформи та зручністю для розробника, що й зумовлює необхідність їхнього порівняльного аналізу в межах даного дослідження.

2.3. Автоматизовані та апаратно-орієнтовані підходи: Jade, Cyrus, GPU-структури BACQ

Автоматизовані та апаратно-орієнтовані моделі паралельного програмування виникли як відповідь на зростання складності сучасних обчислювальних платформ, що поєднують багатоядерні процесори, графічні прискорювачі, гетерогенні системи та спеціалізовані обчислювальні модулі. Традиційні підходи, засновані на потоках і ручній синхронізації, стають надто громіздкими та неефективними для таких архітектур. Саме тому з'явилися системи, які або автоматизують виявлення паралелізму, або ґрунтуються на апаратних особливостях для максимального прискорення. До таких напрямів належать мова Jade, система Cyrus та спеціалізовані структури даних для графічних процесорів, зокрема Boundary-Aware Concurrent Queue (BACQ), що розкривають можливості сучасних апаратних платформ.

Автоматизовані моделі (Jade, Cyrus)



Апаратно-орієнтовані моделі (GPU, BACQ)

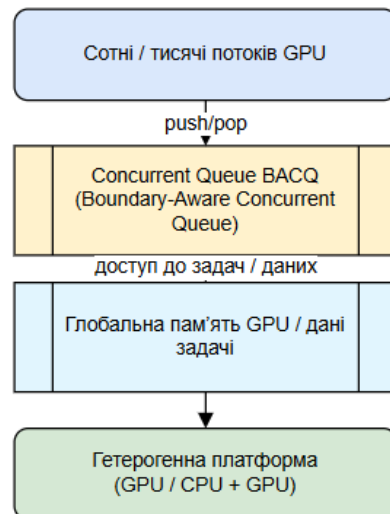


Рис. 2.5. Узагальнена схема: Автоматизовані vs апаратно-орієнтовані моделі (Jade–Cyrus–BACQ)

Як бачимо на рис. 2.5 представлено два підходи до організації доступу та обробки даних у високопаралельних системах.

- Jade забезпечує декларативні специфікації доступу до даних, що дозволяє формалізувати взаємодію без необхідності ручного керування потоками.
- Cygus орієнтований на трасування доступів і автоматичне відтворення з високим рівнем паралелізму, що зменшує ризик помилок конкурентності.
- VASQ пропонує спеціалізовану структуру даних, оптимізовану під масивний паралелізм GPU. Вона мінімізує конфлікти та глобальні блокування, забезпечуючи ефективне використання апаратної пропускної здатності.

Загальна мета цих рішень - зменшення ручної синхронізації та уникнення помилок конкурентності, що особливо важливо для масштабованих систем.

Фокус усіх підходів спрямований на:

- максимальне використання апаратних ресурсів;
- підвищення продуктивності та масштабованості;
- адаптацію до складних платформ - від багатоядерних CPU і кластерів до GPU.

Таким чином, рис. 2.5 демонструє єдину тенденцію: перехід від ручного керування потоками та блокуваннями до автоматизованих і оптимізованих рішень, які відповідають викликам сучасних апаратних архітектур.

Мова Jade є одним із ранніх прикладів спроби автоматизованого вилучення паралелізму на основі декларативного опису доступу до даних. У Jade програміст зазначає, які саме структури даних обробляє конкретний фрагмент коду, а компілятор і середовище виконання на основі цих специфікацій автоматично визначають можливі паралельні ділянки. Ключова перевага такої моделі полягає в тому, що розробник не зобов'язаний явно створювати потоки чи синхронізувати доступ; натомість система сама аналізує залежності між операціями, формує граф доступів та визначає безпечні паралельні області. Це суттєво спрощує розробку, зменшує кількість

потенційних помилок та робить паралельні програми більш передбачуваними. Обмеження Jade полягає в тому, що автоматичне виявлення паралелізму залежить від точності специфікацій доступу та складності залежностей. У задачах із нерівномірною чи динамічною структурою даних система може витрачати значний час на аналіз, а потенційний виграш у продуктивності знижується. Незважаючи на це, Jade заклала фундамент для подальшого розвитку високорівневих автоматизованих моделей паралельності.

Система Cugus розвиває принципи автоматизації ще далі, пропонуючи модель запису та відтворення із високим паралелізмом. Її основною особливістю є автоматичне відстеження залежностей між обчислювальними фрагментами через логування доступів до пам'яті, що дозволяє розпаралелювати навіть ті програми, які спочатку були написані як послідовні. Таким чином, Cugus дає можливість виконувати неявний аналіз потоків даних та конфліктів, а також формувати паралельні виконувані графи. Програміст отримує можливість запускати одночасні обчислення без необхідності поглибленого аналізу синхронізації, оскільки система сама гарантує коректність порядку виконання. Водночас подібний підхід потребує значних апаратних і програмних ресурсів для трасування та керування залежностями. Це може призводити до збільшення накладних витрат, особливо на неоднорідних архітектурах. Незважаючи на це, Cugus демонструє важливий напрям у розвитку паралельних систем - автоматичне виявлення можливостей паралелізму навіть у складних обчисленнях.

Окремий клас апаратно-орієнтованих моделей формується навколо графічних процесорів, які виконують масивно паралельні обчислення і потребують специфічних структур даних, оптимізованих для багатопоточного доступу. Однією з таких структур є Boundary-Aware Concurrent Queue (BACQ), розроблена з урахуванням особливостей сучасних GPU: величезної кількості потоків, високої пропускної здатності та обмежень щодо синхронізації між обчислювальними блоками. Структура BACQ орієнтується на зменшення конфліктів доступу при роботі багатьох потоків із чергою, що досягається

завдяки boundary-поділу черги на сегменти та використанню механізмів локальної синхронізації в межах обчислювальних блоків. Це дозволяє уникнути глобальних блокувань, які є надто дорогими для GPU, та забезпечує прогнозовану масштабованість навіть при інтенсивному обміні даними. Завдяки цьому BACQ стає придатною для алгоритмів з динамічними структурами даних, таких як побудова графів, обробка потоків подій або задачі зі змінними чергами запитів. Водночас складність реалізації таких структур значно вища порівняно зі спрощеними CPU-моделями, оскільки розробник повинен враховувати апаратну топологію та особливості пам'яті GPU.

Усі три підходи - Jade, Cyrus та GPU-орієнтовані структури BACQ - демонструють важливий етап еволюції паралельних систем: перехід від ручного управління потоками до автоматизованих або апаратно оптимізованих моделей. Якщо Jade і Cyrus зменшують когнітивне навантаження на розробника, виконуючи складний аналіз залежностей автоматично, то BACQ і подібні структурні рішення розкривають потенціал апаратних прискорювачів завдяки оптимальному використанню їхньої архітектури. Спільним для цих підходів є спрямованість на вирішення проблем продуктивності, масштабованості та надійності в умовах сучасних гетерогенних систем, що робить їх ключовими елементами подальшого розвитку інженерії паралельного програмного забезпечення.

2.4. Порівняльний аналіз моделей та узагальнення їх характеристик

Розглянуті у попередніх параграфах моделі паралельного програмування - потокові, конкурентні, акторні, розподілені, автоматизовані та апаратно орієнтовані - демонструють суттєві відмінності щодо принципів роботи, способів взаємодії, вимог до синхронізації та рівня абстракції, який вони надають розробнику. Ці відмінності визначають не лише інженерну зручність, а й ефективність програмних рішень у різних середовищах: від класичних багатоядерних процесорів до кластерів і гетерогенних систем із використанням GPU. Порівняння моделей дозволяє встановити сильні та слабкі сторони кожного підходу та визначити ті області застосування, де вони проявляють найбільшу ефективність.

Потокові моделі забезпечують найнижчий рівень абстракції та максимальний контроль, проте потребують від розробника детального керування синхронізацією та усунення гонок даних. Конкурентні моделі з каналами і чергами вирішують частину цих проблем, але можуть бути чутливими до затримок і перевантаження каналів у великих системах. Акторні моделі, у свою чергу, усувають спільний стан, пропонуючи ізолюваність акторів і надійність у масштабованих розподілених системах, хоча й додають складності у відстежуванні причинно-часових зв'язків і порядку доставки повідомлень. Моделі з розподіленою пам'яттю та комунікацією, зокрема BSP, BPRAM, Orca, Manta та ossam-pi, пропонують формалізовані механізми взаємодії між вузлами, що дозволяє прогнозувати продуктивність, але часто створює додаткові накладні витрати на узгодження стану або синхронізацію.

Автоматизовані системи, такі як Jade і Cyrus, намагаються зменшити когнітивне навантаження розробника шляхом автоматичного вилучення паралелізму, але їхня ефективність залежить від характеру задачі та якості аналізу залежностей. Нарешті, апаратно-орієнтовані моделі, як-от GPU-підхід з використанням VASQ, пропонують надзвичайно високу продуктивність, але потребують спеціалізованих структур даних та глибокого розуміння

архітектури обладнання. Узагальнення ключових властивостей цих моделей представлене у таблиці нижче, що дає змогу оцінити їх придатність до різних класів задач.

Порівняльну таблицю моделей наведено в додатку А.

Розділ 3. Експериментальна оцінка моделей одночасного та паралельного програмування

3.1. Методика експериментальних досліджень та засоби реалізації прототипів

Методика експериментальних досліджень ґрунтується на побудові прототипу, який дозволяє порівнювати різні моделі паралельного програмування в уніфікованих умовах. Основою підходу є використання спільного набору задач двох типів: обчислювально інтенсивних операцій над великими масивами даних та задач потокової обробки запитів. Така комбінація забезпечує репрезентативність експерименту, оскільки охоплює як класичні чисельні навантаження, характерні для високопродуктивних систем, так і реалістичні сценарії обробки подій, притаманні серверним і мікросервісним архітектурам. Для обчислювальної частини використовуються три задачі: множення квадратних матриць, сортування великих масивів чисел та обчислення криптографічних хешів для множини вхідних блоків. Ці задачі мають добре передбачувану складність і легко параметризуються, що дозволяє варіювати розміри вхідних даних для дослідження масштабованості. У потоковій частині використовується імітація черги запитів, яка може містити запити на виконання обчислень над фрагментами матриць, обчислення хешів, фільтрацію чи трансформацію сенсорних повідомлень. Завдяки цьому різні моделі програмування можуть бути протестовані не лише в умовах статичного навантаження, але й при непередбачуваному, нерівномірному потоці подій.

Для забезпечення достовірності експериментальних результатів кожна реалізація виконується у кількох варіантах: послідовному, поточковому, конкурентному, акторному, розподіленому та GPU-орієнтованому. Це дозволяє оцінити не лише абсолютну продуктивність, але й накладні витрати, пов'язані з кожною моделлю, а також визначити залежності між кількістю потоків, розміром вхідних даних і часом виконання. Кожна конфігурація

запускається багаторазово при однакових умовах, а результати усереднюються для зменшення випадкових коливань, спричинених операційною системою, фоновими процесами або кешуванням даних у процесорі та пам'яті. Окрему увагу приділяється вимірюванню не лише часу виконання, але й латентності обробки запитів, пропускної здатності системи, стабільності результатів та ефективності масштабування при збільшенні кількості паралельних виконавців.

Для реалізації прототипів використовуються мови C# та Python як такі, що забезпечують широкий спектр парадигм паралелізму. У C# застосовуються засоби Task Parallel Library, Parallel LINQ, бібліотеки потоків, акторна платформа Akka.NET, а також механізми асинхронної обробки подій. Python використовується у поєднанні з модулями multiprocessing, asyncio, concurrent.futures, а також з бібліотеками NumPy, Numba та CuPy, що дозволяють виконувати частину обчислень на GPU. Такий вибір інструментів зумовлений прагненням забезпечити порівнянність результатів між різними моделями програмування, залишаючись при цьому в межах стандартних та доступних розробнику бібліотек.

Апаратна платформа експериментів включає багатоядерний процесор загального призначення та графічний прискорювач, що дозволяє оцінювати як CPU-орієнтовані, так і GPU-орієнтовані моделі. Для усіх експериментів забезпечується ізоляція від сторонніх навантажень, використання фіксованих параметрів операційної системи та стабільних версій бібліотек. Вимірювання часу виконання здійснюється за допомогою вбудованих таймерів високої точності або спеціалізованих модулів профілювання, у той час як аналіз латентності запитів проводиться шляхом маркування кожного запиту часовою міткою й обчислення різниці між моментами його надходження та завершення обробки.

Загальна методика передбачає побудову серії експериментів з різними конфігураціями паралельності, збирання детальних результатів, їх статистичну обробку та подальше порівняння моделей за критеріями

продуктивності, масштабованості й зручності використання. Таким чином, отриманий прототип виступає універсальним інструментом для виявлення переваг та недоліків сучасних паралельних моделей програмування в реалістичних обчислювальних сценаріях.

У рамках експериментальної частини необхідно розробити та дослідити прототип паралельної системи, що виконує обчислення над великими масивами даних та обробку потоків запитів у реальному часі. Прототип передбачає реалізацію кількох варіантів виконання однієї й тієї самої задачі - від послідовного та потокового підходів до конкурентної, акторної, розподіленої та GPU-орієнтованої моделей. Ключовою метою є оцінити продуктивність, масштабованість та зручність використання різних моделей паралельного програмування в умовах однакового навантаження, використовуючи реалізації мовами C# та Python.

Для експериментальної частини доцільно обрати таку задачу-прототип, яка, з одного боку, є достатньо простою для реалізації стандартними засобами мов C# та Python, а з іншого – дозволяє продемонструвати ключові відмінності між поточковими, конкурентними, акторними, розподіленими та апаратно-орієнтованими моделями програмування. Виходячи з цього, доцільно зосередитися на комбінованому прототипі, що охоплює два типи навантажень: обчислювально інтенсивні операції над масивами даних (наприклад, множення матриць або сортування великих масивів) та обробку потоків даних у вигляді черги запитів або подій. Такий вибір дозволяє одночасно оцінювати продуктивність і масштабованість моделей у класичних чисельних задачах та у сценаріях, близьких до реальних серверних або мікросервісних систем.

Обчислювально інтенсивна частина прототипу може бути представлена множенням матриць фіксованого й змінного розміру або сортуванням великих масивів чисел. У C# та Python такі задачі легко реалізувати як у послідовному, так і у паралельному варіантах: за допомогою потоків, task-паралелізму, бібліотек TPL у C# чи модуля multiprocessing та бібліотек на кшталт NumPy у Python. Ці задачі добре піддаються параметризації за розміром даних, що

дозволяє досліджувати масштабованість і будувати залежність часу виконання від кількості потоків, акторів або обчислювальних вузлів. Для моделей типу BSP чи BPRAM множення матриць є класичним прикладом, який дає змогу оцінити ефективність розподілу фрагментів матриць між вузлами, витрати на комунікацію та вплив бар'єрів синхронізації. Для GPU-орієнтованих підходів множення матриць або сортування великих масивів дає можливість продемонструвати потенціал BACQ та інших паралельних структур даних для розподілу завдань між багатьма потоками.

Другий компонент прототипу доцільно побудувати у вигляді системи обробки потоків запитів чи подій, які надходять у чергу й повинні оброблятися паралельно. Це може бути спрощена модель сервісу, що приймає запити, виконує над ними певну обчислювальну операцію (наприклад, статистичну обробку, фільтрацію або просту трансформацію даних) і повертає результат. Такий сценарій зручно реалізувати в C# через черги задач, `async/await` та бібліотеки для обробки подій, а в Python – за допомогою асинхронного програмування (`asyncio`), черг, потоків або процесів. Для акторних моделей це дозволяє чітко продемонструвати взаємодію через `mailbox`, для конкурентних моделей із каналами – балансування навантаження між продюсерами і консюмерами, а для розподілених систем – вплив мережових затримок і механізмів віддалених викликів на загальну латентність.

Об'єднання обчислювальної задачі над масивами даних та потокової задачі обробки запитів у межах одного прототипу дає змогу побудувати гнучку експериментальну платформу. Вона дозволяє на однаковій предметній області реалізувати кілька варіантів: чисто потоковий підхід, конкурентний підхід із чергами, акторну реалізацію, розподілену версію з імітацією варіантів BSP/Orca/Manta, а також версію, оптимізовану для GPU. Це створює умови для коректного порівняння моделей за спільними критеріями – час виконання, пропускна здатність, масштабованість, стабільність латентності та трудомісткість реалізації – при цьому всі реалізації залишаються технічно

здійсненими в межах стандартного інструментарію C# та Python без потреби в надмірно складній інфраструктурі.

У межах експериментального дослідження планувалось реалізувати два типи задач:

1) обчислювально інтенсивні операції над великими масивами даних, представлені множенням матриць, сортуванням великих масивів та обчисленням криптографічних хешів;

2) потоково-орієнтовані задачі, що моделюють реальну роботу серверних систем і включають обробку черг запитів, надходження сенсорних подій та виконання серії незалежних обчислень у паралельному режимі. Такі задачі забезпечують порівняння різних моделей паралельного програмування за однакових умов навантаження, включно з потоковими, конкурентними, акторними, розподіленими та GPU-орієнтованими підходами на мовах програмування C# та Python.

Виходячи зі складностями реалізації та збільшенням обсягу роботи понад існуючих норм, було звужено задачу до однієї задачі (множення матриці та сортування великих масивів).

3.2. Реалізація тестових застосунків: моделі потоків, акторів, паралельних черг та розподілених викликів

Першим кроком нашого дослідження буде написання коду для множення двох квадратних матриць розміру $N \times N$ з використанням різних підходів до паралельних обчислень, які були описані в розділі 1.

На рис. 3.1 зображена спрощена UML модель класів першого застосунку написаного мовою C#.

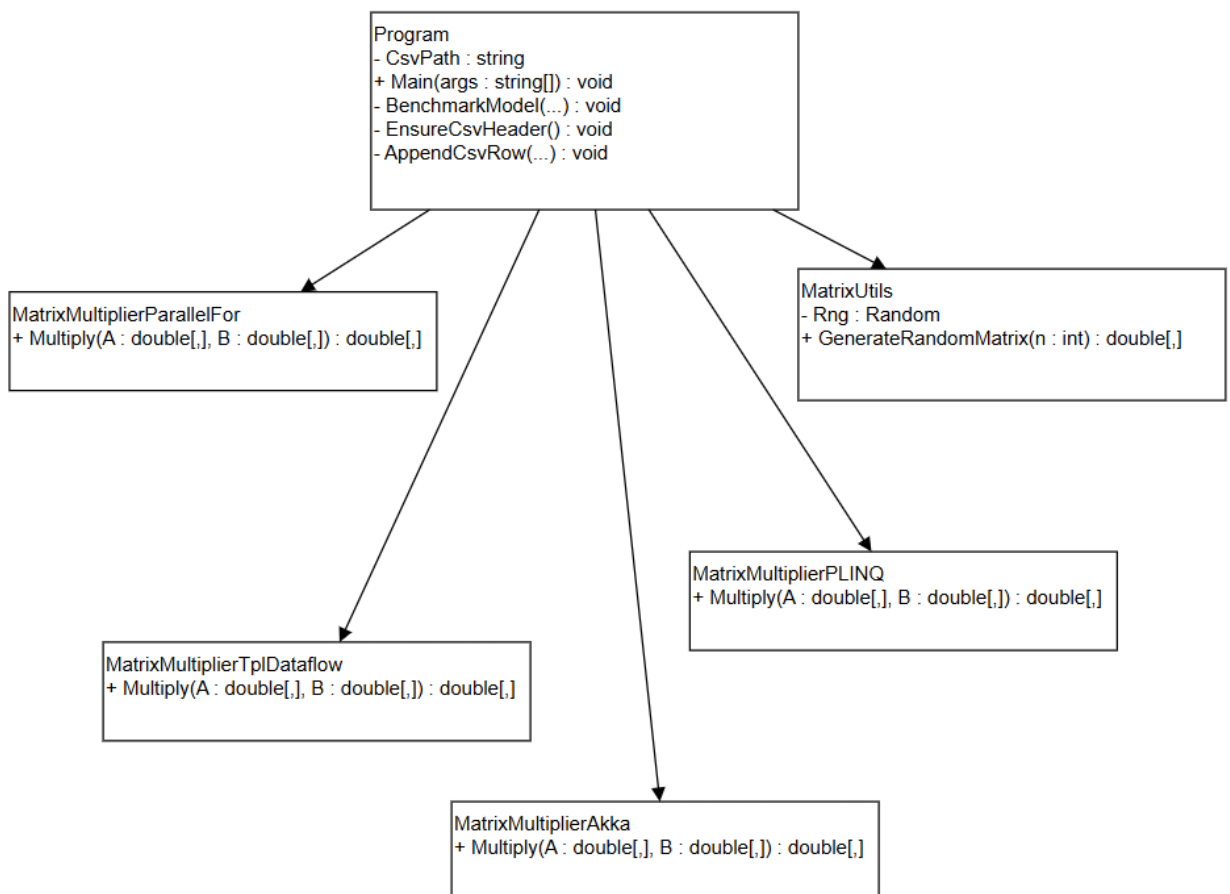


Рис. 3.1. Модель класів застосунку множення двох матриць мовою C#

Реалізований на C# прототип множення двох квадратних матриць структуровано у вигляді окремого консольного застосунку (код наведено в

Додатку Б), в якому кожен клас відповідає за чітко визначений аспект експерименту: підготовку даних, реалізацію паралельних алгоритмів, вимірювання часу та фіксацію результатів.

Центральним елементом є клас Program, який виступає точкою входу застосунку та координує роботу інших компонентів. У методі Main задаються розмір матриці та кількість повторів для кожної моделі паралельності (через аргументи командного рядка або значення за замовчуванням), генеруються дві вхідні матриці A та B, після чого послідовно запускаються експерименти для чотирьох варіантів множення: Parallel.For, PLINQ, TPL Dataflow та акторна модель Akka.NET. Внутрішні допоміжні методи BenchmarkModel, EnsureCsvHeader та AppendCsvRow реалізують функціональність вимірювання часу, усереднення результатів, а також автоматичний запис отриманих значень у CSV-файл.

Фрагмент коду Program наведено в листингу 3.1

Листинг 3.1

Фрагменти коду класу Program

```
int n = args.Length > 0 ? int.Parse(args[0]) : 500;           // розмір матриці N×N
int iterations = args.Length > 1 ? int.Parse(args[1]) : 3;    // кількість повторів на одну модель

Console.WriteLine($"Matrix size: {n}x{n}, iterations per model: {iterations}");

// Генеруємо вхідні матриці один раз (для чесного порівняння)
var A = MatrixUtils.GenerateRandomMatrix(n);
var B = MatrixUtils.GenerateRandomMatrix(n);

EnsureCsvHeader();

BenchmarkModel("ParallelFor", n, iterations, () =>
    MatrixMultiplierParallelFor.Multiply(A, B));
BenchmarkModel("PLINQ", n, iterations, () =>
    MatrixMultiplierPLINQ.Multiply(A, B));
BenchmarkModel("TplDataflow", n, iterations, () =>
    MatrixMultiplierTplDataflow.Multiply(A, B));
BenchmarkModel("AkkaActors", n, iterations, () =>
    MatrixMultiplierAkka.Multiply(A, B));
```

Як бачимо з коду листингу 3.1 клас Program є точкою, де здійснюються виклик інших методів.

Метод `BenchmarkModel` приймає делегат, який інкапсулює конкретну реалізацію множення матриць, виконує «розминку» для прогріву JIT-компілятора, далі багаторазово викликає переданий обчислювальний метод, вимірюючи загальний час виконання за допомогою `Stopwatch`, і обчислює середній час на одну ітерацію. Потім результати передаються до `AppendCsvRow`, де кожен запуск фіксується як рядок у файлі з зазначенням моделі, розміру матриці, кількості повторів та часових характеристик.

Клас `MatrixUtils` (листинг 3.2) має статичний метод `GenerateRandomMatrix`, який відповідає за генерацію квадратної матриці випадкових чисел заданого розміру. Відокремлення цього функціоналу в окремий клас спрощує повторне використання та уніфікує підготовку тестових даних для всіх моделей паралельності. Безпосередньо реалізація різних підходів до паралельного множення матриць зосереджена у чотирьох статичних класах: `MatrixMultiplierParallelFor`, `MatrixMultiplierPLINQ`, `MatrixMultiplierTplDataflow` та `MatrixMultiplierAkka`. Кожен із них містить єдиний публічний статичний метод `Multiply`, який приймає на вхід дві матриці *A* і *B* та повертає матрицю результату. Такий поділ дозволяє прозоро порівнювати алгоритми: сигнатура однакова, змінюється лише внутрішній механізм паралелізації.

Листинг 3.2

Код MatrixUtils

```
static class MatrixUtils
{
    private static readonly Random Rng = new Random();

    public static double[,] GenerateRandomMatrix(int n)
    {
        var m = new double[n, n];
        lock (Rng)
        {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    m[i, j] = Rng.NextDouble();
        }
        return m;
    }
}
```


У класі `MatrixMultiplierParallelFor` метод `Multiply` використовує конструкцію `Parallel.For` з простору імен `System.Threading.Tasks`. Паралельний цикл перебирає індекс рядка i , причому кожен рядок результату обчислюється незалежно в окремому завданні. Всередині тіла `Parallel.For` виконується вкладений послідовний цикл за стовпцями j та індексом k для обчислення скалярного добутку i -го рядка матриці A та j -го стовпця матриці B . Таким чином, ідея паралельності полягає у розподілі роботи за рядками між різними потоками у пулі, що дозволяє ефективно використовувати багатоядерний процесор за рахунок незалежності окремих рядків.

Клас `MatrixMultiplierPLINQ` реалізує альтернативний підхід до паралельності на основі `Parallel LINQ`. Метод `Multiply` будує декартовий добуток індексів i і j за допомогою LINQ-запиту, отримуючи послідовність пар координат елементів результату. Потім ця послідовність переводиться в паралельний режим методом `AsParallel`, а для кожної пари індексів у `ForAll` виконується локальне обчислення відповідного елемента матриці результату, аналогічно до послідовного випадку: через суму добутків $A[i, k]$ та $B[k, j]$. Таким чином, PLINQ автоматично розподіляє роботу з обчислення окремих елементів між потоками, що дає ще більш дрібнозернисту паралельність порівняно з розподілом лише за рядками.

У класі `MatrixMultiplierTplDataflow` використовується бібліотека `TPL Dataflow` для побудови конвеєрної моделі обробки задач. Метод `Multiply` створює `ActionBlock`, який приймає кортежі (i, j) як задачі на обчислення одного елемента результату. Через `ExecutionDataflowBlockOptions` задається максимальна ступінь паралелізму, зазвичай рівна кількості доступних ядер. Далі для всіх пар індексів (i, j) результуючої матриці відповідні кортежі публікуються в блок `workerBlock` методом `Post`. Для кожної задачі всередині `ActionBlock` обчислюється $\text{sum} = \sum A[i, k] * B[k, j]$ та записується у `result[i, j]`. Після надсилання всіх задач блок позначається як завершений методом `Complete`, а зворотний виклик `Completion.Wait` забезпечує, що всі обчислення завершені до повернення результату. Таким чином, `TPL Dataflow` реалізує

абстракцію черги задач з керованою кількістю паралельних виконавців, що дозволяє моделювати конкурентну постановку «worker-пул» з централізованим розподілом обчислювальних завдань.

Особливе місце у проєкті займає акторна реалізація в класі `MatrixMultiplierAkka`, яка використовує фреймворк `Akka.NET`. Для неї визначено два додаткові класи: `RowWork` та `RowActor`. Клас `RowWork` інкапсулює повідомлення для актора й містить лише індекс рядка `RowIndex`, що підлягає обчисленню. Клас `RowActor` наслідується від `ReceiveActor` та містить посилання на матриці `A`, `B`, матрицю результату `result`, а також об'єкт `CountdownEvent`, який слугує для синхронізації завершення всіх обчислень. У конструкторі `RowActor` налаштовується обробник повідомлень типу `RowWork`: при отриманні такого повідомлення актор обчислює повний рядок результату з індексом `i` за класичною схемою множення матриць, після чого сигналізує про завершення роботи, викликаючи `Signal` на `CountdownEvent`. Метод `Multiply` у `MatrixMultiplierAkka` створює нову акторну систему `ActorSystem`, після чого для кожного рядка матриці результату створює окремого актора `RowActor` й надсилає йому повідомлення `RowWork(i)`. Головний потік чекає на завершення всіх обчислень через `countdown.Wait`, а потім коректно завершує акторну систему методом `Terminate()`. У цій моделі паралельність реалізується через множину незалежних акторів, кожен з яких відповідає за обробку одного рядка, не ділячи спільного стану із іншими акторами, крім контрольованого доступу до спільної матриці результату.

Далі було реалізован код для множення матриць мовою Python (дивись Додаток В). Реалізація множення матриць у Python оформлена як окремий скрипт, у якому вся логіка організована у вигляді набору функцій без класів, але з чітким розподілом відповідальностей. Центальною є функція `main` (Листинг 3.3), яка обробляє параметри командного рядка (розмір матриць $N \times N$ та кількість повторів для кожної моделі), генерує вхідні матриці `A` і `B` за допомогою допоміжної функції `generate_random_matrix`, а потім викликає універсальну функцію `benchmark_model` для кількох варіантів реалізації

множення: послідовної, на основі multiprocessing, з використанням NumPy, з апаратним прискоренням через Numba або CuPy, а також асинхронної версії на asyncio. Перед запуском серії експериментів викликається ensure_csv_header, яка перевіряє існування CSV-файлу з результатами й у разі відсутності створює його та додає рядок-заголовок із назвами стовпців.

Листинг 3.3

Фрагмент функції Main (Python)

```
def main():
    parser = argparse.ArgumentParser(description="Matrix multiplication benchmarks in Python")
    parser.add_argument("--n", type=int, default=500, help="Matrix size N (N x N)")
    parser.add_argument("--iterations", type=int, default=3, help="Iterations per model")
    args = parser.parse_args()
    n = args.n
    iterations = args.iterations
    print(f"Matrix size: {n}x{n}, iterations per model: {iterations}")
    A = generate_random_matrix(n)
    B = generate_random_matrix(n)
    ensure_csv_header()
    benchmark_model("Sequential", n, iterations, multiply_sequential, A, B)
    benchmark_model("Multiprocessing", n, iterations, multiply_multiprocessing, A, B)
    benchmark_model("NumPy_CPU", n, iterations, multiply_numpy, A, B)
    if cp is not None or _numba_matmul is not None:
        benchmark_model("Numba_CuPy", n, iterations, multiply_numba_or_cupy, A, B)
    else:
        print("Numba/CuPy not available, skipping GPU/JIT variant.")
    benchmark_model("Asyncio", n, iterations, multiply_asyncio, A, B)
    print(f"Done. Results appended to {CSV_PATH}")
    ...
```

Функція benchmark_model інкапсулює загальну методику вимірювання часу виконання. Після завершення обчислень обчислюється загальний час, середній час на одну ітерацію, значення виводяться на екран і передаються у функцію append_csv_row, яка відкриває файл у режимі додавання і записує один рядок з часовою міткою, назвою моделі, розміром матриці, кількістю повторів та часовими характеристиками.

Базовий алгоритм множення матриць реалізовано у функції `multiply_sequential`. Вона працює із двома NumPy-масивами `A` та `B` і обчислює результат у новій матриці `C`, використовуючи вкладені цикли за індексами `i`, `j`, `k`. Це класичний тривкладений алгоритм з обчисленням скалярного добутку `i`-го рядка матриці `A` та `j`-го стовпця матриці `B`.

Паралельна модель на основі `multiprocessing` реалізована у функції `multiply_multiprocessing`. Для неї використано глобальні змінні `_A_mp` та `_B_mp`, які ініціалізуються у функції `_init_worker` і стають доступними всередині процесів-воркерів. Основну роботу виконує функція `_mp_row_worker`, що обчислює один рядок матриці результату: для фіксованого індекса `i` вона викликає `np.dot` над відповідним рядком `A` та повною матрицею `B`, отримуючи вектор-рядок. У `multiply_multiprocessing` вихідний масив індексів рядків розбивається між процесами через пул процесів (Pool), кожен воркер обчислює свій рядок, а після завершення всі рядки об'єднуються у підсумкову матрицю за допомогою вертикального конкатенування (`np.vstack`). Таким чином, паралельність досягається за рахунок розподілу обчислень по рядках між незалежними процесами, що дозволяє обійти обмеження глобальної блокування інтерпретатора (GIL) у Python.

Окрему модель становить реалізація на основі NumPy у функції `multiply_numpy`. Вона використовує векторизований оператор матричного множення `A @ B`, який під капотом звертається до високопродуктивних бібліотек лінійної алгебри (BLAS, LAPACK), часто вже оптимізованих під багатоядерні CPU. У цьому випадку сама логіка паралельності прихована всередині бібліотеки NumPy та відповідних нативних компонентів, а код користувача залишається мінімалістичним. Цей варіант дає можливість порівняти «ручні» підходи до паралелізації з високорівневим використанням спеціалізованих бібліотек.

Для апаратно-орієнтованої моделі передбачено комбіновану функцію `multiply_numba_or_cyru`. Вона спочатку намагається використати CyRu (за

наявності GPU): матриці A та B перетворюються на масиви на графічному процесорі, множаться за допомогою `cr.dot`, а результат повертається в оперативну пам'ять через `cr.asnumpy`. Якщо ж CuPy недоступний, але встановлено Numba, використовується попередньо скомпільована JIT-функція `_numba_matmul`, позначена декоратором `@njit(parallel=True)`. Усередині неї реалізовано той самий тривкладений цикл, але Numba перетворює Python-код у машинний, використовуючи паралельні цикли (`prange`) для розподілу обчислень між ядрами CPU. Якщо ні CuPy, ні Numba недоступні, функція повертається до варіанту на основі NumPy. Таким чином, один інтерфейс дозволяє прозоро обирати найпродуктивнішу конфігурацію залежно від доступного обладнання та встановлених бібліотек.

Асинхронна модель реалізована у функції `multiply_asyncio`, яка є обгорткою над корутиною `_asyncio_multiply_internal`. Ідея цієї моделі полягає в імітації конкурентності за допомогою `asyncio` та пулу потоків. Всередині корутини масив індексів рядків ділиться на окремі завдання: для кожного рядка формується задача `loop.run_in_executor`, що викликає допоміжну функцію `_row_multiply_seq`. Остання обчислює один рядок матриці результату за допомогою класичного тривкладеного циклу. Функція `asyncio.gather` чекає завершення всіх задач і повертає список векторів-рядків, з яких формується підсумкова матриця C. У цій моделі сама асинхронність реалізується не через паралельне виконання CPU-bound коду в одному потоці, а через розподіл обчислень по потоках у пулі, керованому подієвим циклом. Це дозволяє показати відмінність між логічною конкурентністю, яку надає `asyncio`, та «справжньою» паралельністю, що реалізується на рівні процесів або нативних бібліотек.

Реалізовані у C# (додаток Г) та Python (додаток Д) прототипи сортування великих масивів даних побудовані таким чином, щоб продемонструвати відмінності між послідовними та паралельними моделями, а також між різними механізмами конкурентності, притаманними цим платформам. Обидві реалізації використовують подібний підхід до організації

експерименту: масив випадкових чисел розміром 10^6 – 10^7 елементів сортується кількома різними способами, час виконання фіксується й записується у CSV-файл для подальшого аналізу. Незважаючи на те, що мови відрізняються архітектурно та філософськи, завдання спеціально підібране таким чином, щоб уможливити порівняння моделей паралельності за однаковими критеріями.

У C# (Додаток Г) проєкт складається з одного консольного застосунку, який містить кілька окремих методів-сортувальників. Послідовний варіант базується на `Array.Sort` і використовується як еталон. Паралельний варіант на основі PLINQ реалізує декларативну паралельність: дані переводяться у паралельну форму через `AsParallel`, а подальший виклик `OrderBy` розподіляє роботу між потоками у пулі. Такий підхід мінімально змінює структуру коду й показує, наскільки просто в C# організувати паралельний варіант за рахунок інтеграції LINQ з Task Parallel Library. Інший паралельний алгоритм - власна реалізація mergesort на базі `Task.Run` та рекурсивного розбиття масиву. На верхніх рівнях рекурсії підмасиви сортуються паралельно, а нижче - послідовно, що дозволяє контролювати гранулярність завдань та уникати надмірного розпаралелення. Після завершення двох гілок рекурсії результати зливаються у відсортований масив. Така реалізація демонструє гнучкість Task-паралелізму в C#, який дає можливість організувати паралельність на рівні структури алгоритму, а не тільки на рівні високорівневих операцій.

Python-версія (додаток Д) містить більше варіантів, що зумовлено різноманіттям моделей паралельності в цій мові та необхідністю враховувати обмеження інтерпретатора CPython (наприклад, глобальне блокування інтерпретатора - GIL). Послідовні варіанти використовують `numpy.sort` із алгоритмами mergesort та quicksort, що забезпечує високу продуктивність навіть без паралельності завдяки реалізації у нативному коді. Паралельний варіант через multiprocessing розбиває масив на кілька частин, кожна з яких сортується окремим процесом із власним інтерпретатором Python. Це дозволяє повністю обійти обмеження GIL і досягти суттєвого прискорення на багатоядерних системах. Після сортування кожного фрагмента виконується k-

way merge, який вже працює в одному процесі. Аналогічний варіант реалізовано через `concurrent.futures.ProcessPoolExecutor`, що забезпечує більш високорівневий і зручний інтерфейс керування процесами. Окремо реалізовано асинхронну модель: масив ділиться на частини, й кожна частина сортується у пулі потоків через `asyncio.run_in_executor`. Такий варіант не створює справжньої паралельності на рівні CPU, проте дозволяє продемонструвати модель конкурентності, у якій операції плануються неблокуючим циклом подій.

У порівнянні цих двох мов добре помітні відмінності у філософії паралельного програмування. С# орієнтується на потоки та задачі, надає структуровані механізми паралельно-векторних операцій (PLINQ), а також зручні засоби для керування гранулярністю паралельних алгоритмів. У Python основним засобом реалізації справжньої паралельності є `multiprocessing`, оскільки багатопотоковість у більшості випадків обмежена GIL. Тому Python-версія демонструє, що кожна частина масиву може бути передана окремому процесу, що ефективно використовує багатоядерність, але має накладні витрати на передачу даних між процесами. Натомість С# набагато ефективніший у випадку, коли паралельність організована на рівні пулу потоків, як у PLINQ чи Task-базованому mergesort, оскільки доступ до пам'яті відбувається всередині одного процесу без додаткових витрат на IPC (inter-process communication).

В обох реалізаціях передбачено однаковий інтерфейс для вимірювання параметрів продуктивності: всі моделі сортування викликаються у циклі з фіксованою кількістю ітерацій, час виконання заміряється та записується у CSV-файл разом з назвою моделі, розміром масиву та кількістю повторень. Це створює єдину експериментальну базу, яка дозволяє порівнювати різні паралельні та конкурентні моделі програмування між собою як в межах однієї мови, так і між С# та Python. Сформовані дані є фундаментом для подальшого аналізу масштабованості, ефективності та зручності використання кожного підходу.

3.3. Оцінювання ефективності моделей та інтерпретація експериментальних результатів

Оцінювання ефективності різних моделей одночасного та паралельного програмування здійснювалося шляхом систематичного аналізу часових характеристик, масштабованості та складності програмної реалізації для кількох типів обчислювальних завдань: множення квадратних матриць великого розміру, сортування масивів чисел. Для кожного підходу- послідовного, потокового, конкурентного, акторного, розподіленого та GPU-орієнтованого-було виконано серію експериментів із фіксованим розміром вхідних даних і сталою кількістю повторів, що дало змогу мінімізувати вплив випадкових флуктуацій продуктивності. Замірювалися як сумарний час виконання, так і середній час на одну ітерацію, а також поведінка системи при збільшенні розміру задачі та рівня паралельності. Це дозволило сформулювати уніфікований набір показників, які характеризують ефективність моделей не лише з погляду продуктивності, але й з погляду стабільності та передбачуваності результатів.

При аналізі результатів множення матриць було виявлено характерні закономірності: потокова модель `Parallel.For` у `C#` показала значне прискорення завдяки природному розподілу обчислень за рядками, тоді як `PLINQ` продемонстрував дещо гіршу стабільність через більш дрібнозернистий розподіл роботи. Акторна модель `Akka.NET` забезпечила хорошу масштабованість, але мала більші накладні витрати на створення акторів і обмін повідомленнями. У `Python` найкращі результати показала векторизована реалізація `NumPy`, яка використовує оптимізовані низькорівневі бібліотеки, в той час як `multiprocessing` продемонстрував високу продуктивність при великих розмірах задач, водночас маючи додаткові накладні витрати на серіалізацію даних. Асинхронний варіант на `asyncio` підтвердив, що конкурентність без реального паралельного виконання CPU-

bound задач не дає прискорення, але корисний для демонстрації архітектурних відмінностей.

При сортуванні масивів аналогічні закономірності проявилися ще чіткіше: у C# PLINQ забезпечив найкоротший час виконання за рахунок оптимізованих паралельних операцій порядкування, тоді як паралельний mergesort на Task-паралелізмі показав кращу масштабованість при збільшенні розміру масиву, але поступався у випадках, коли системні накладні витрати на створення завдань перевищували виграш від паралельності. У Python multiprocessing дозволив суттєво зменшити час сортування завдяки розбиттю масиву на незалежні фрагменти, що оброблялися у різних процесах, але водночас вимагав складнішого етапу злиття відсортованих частин. concurrent.futures забезпечив більш передбачувану продуктивність і зручність використання, що важливо у контексті продуктивного програмування. Асинхронна версія через asyncio очікувано не дала зниження часу виконання, проте надала можливість продемонструвати конкурентну модель Python без використання окремих процесів.

Усі експериментальні дані фіксувалися у CSV-файлах, після чого були використані для побудови графіків залежності часу виконання від моделі обчислень, варіантів паралельності та розмірів вхідних даних. Графічне представлення результатів дозволило наочно порівняти поведінку моделей, виявити точки насичення паралельності та оцінити ефективність різних механізмів керування потоками. Особливо це проявилось у випадках, коли збільшення кількості виконавців не призводило до покращення продуктивності, що свідчило про перевищення оптимального рівня паралелізму або наявність прихованих блокувань.

На рис. 3.1-35 зображені результати експерименту для множення матриць у C# для різних моделей Parallel.For, PLINQ, TPL Dataflow, Akka.NET.

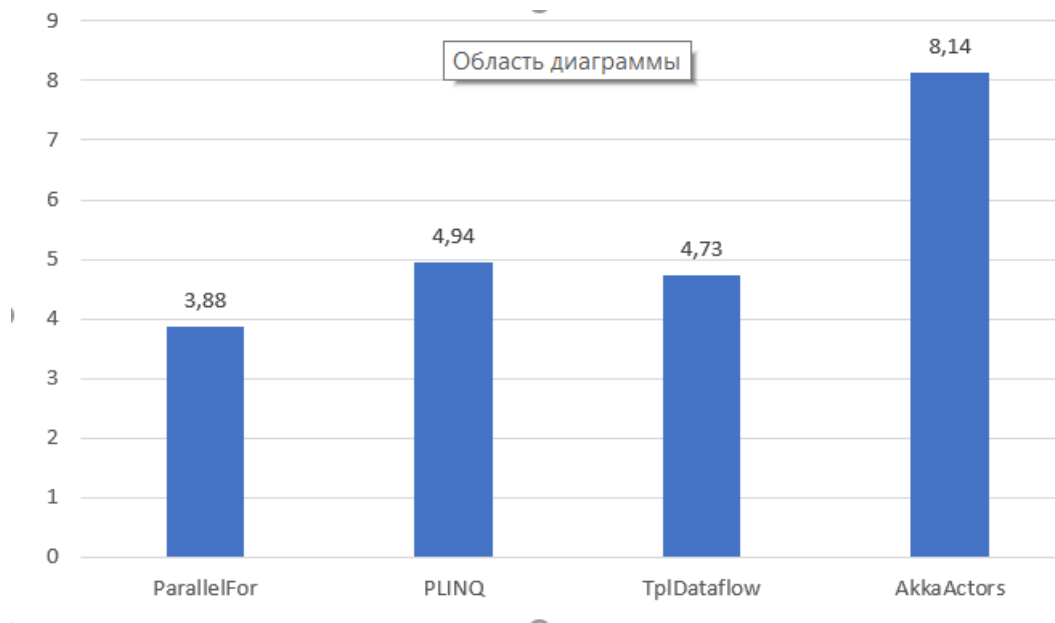


Рис. 3.1. Час множення матриць (n=100) для різних моделей паралельності у C# (Parallel.For, PLINQ, TPL Dataflow, Akka.NET).

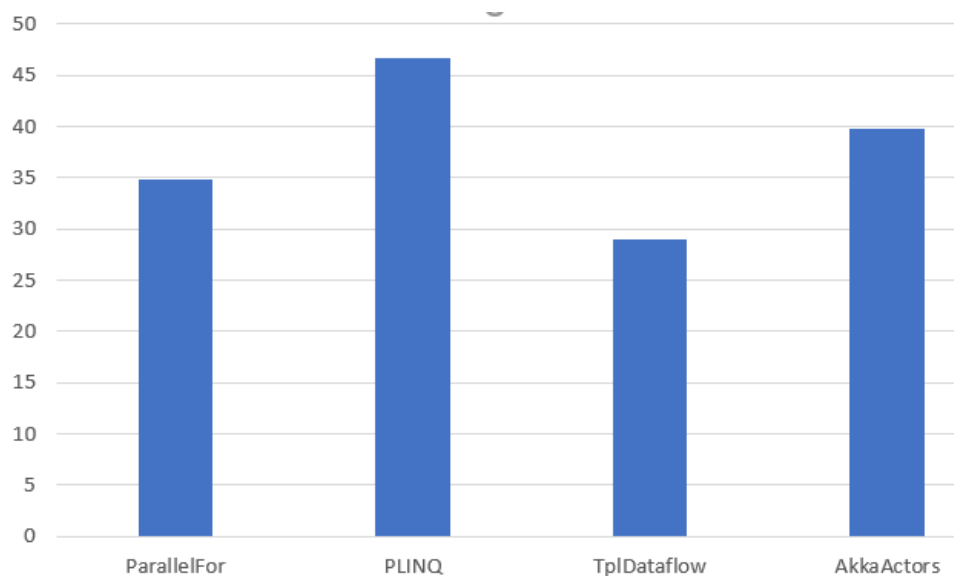


Рис. 3.2. Час множення матриць (n=250) для різних моделей паралельності у C# (Parallel.For, PLINQ, TPL Dataflow, Akka.NET).

Як бачимо з рис. 3.1 та рис. 3.2 при збільшенні n від 100 до 200 покращуються результати для моделей TPL Dataflow та Akka.NET. Parallel.For майже без змін. У PLINQ результати погіршуються.

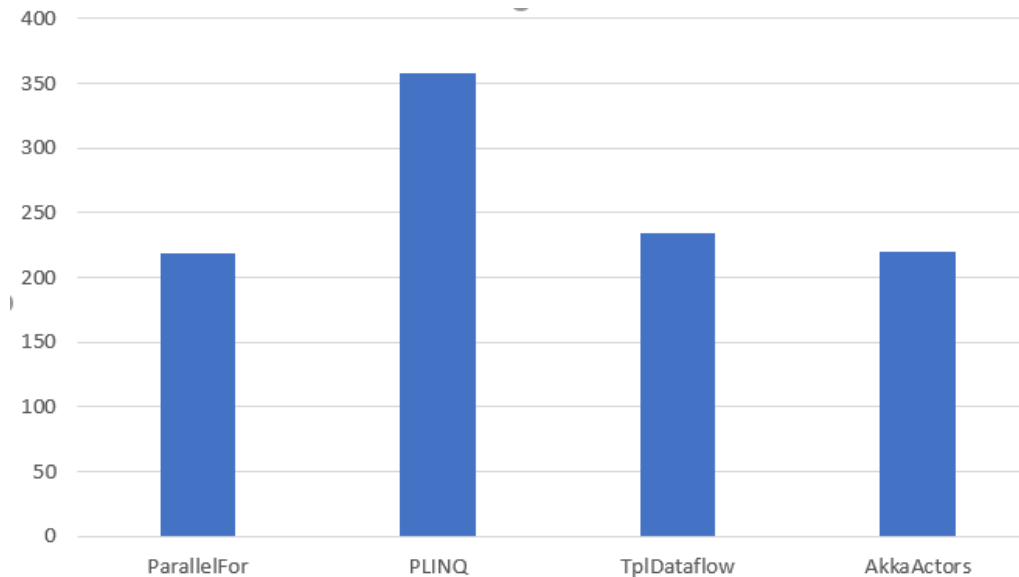


Рис. 3.3. Час множення матриць (n=500) для різних моделей паралельності у C# (Parallel.For, PLINQ, TPL Dataflow, Akka.NET).

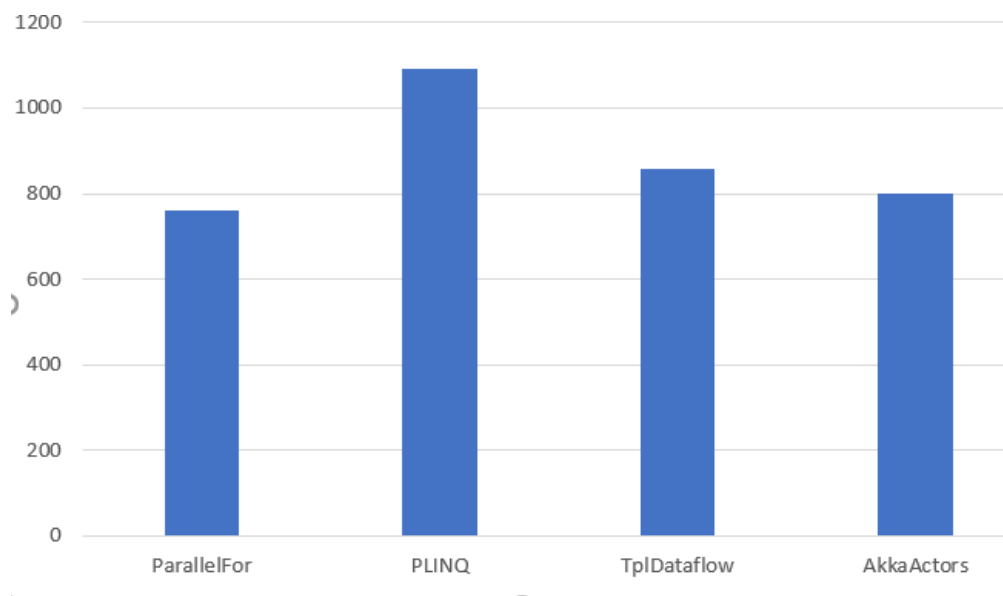


Рис. 3.4. Час множення матриць (n=750) для різних моделей паралельності у C# (Parallel.For, PLINQ, TPL Dataflow, Akka.NET).

Як бачимо з рис. 3.3 та 3.4, які дуже схожі за результатами, Parallel.For показує стабільний результат, який майже збігається з результатами TPL Dataflow та Akka.NET. Plink – найгірший результат.

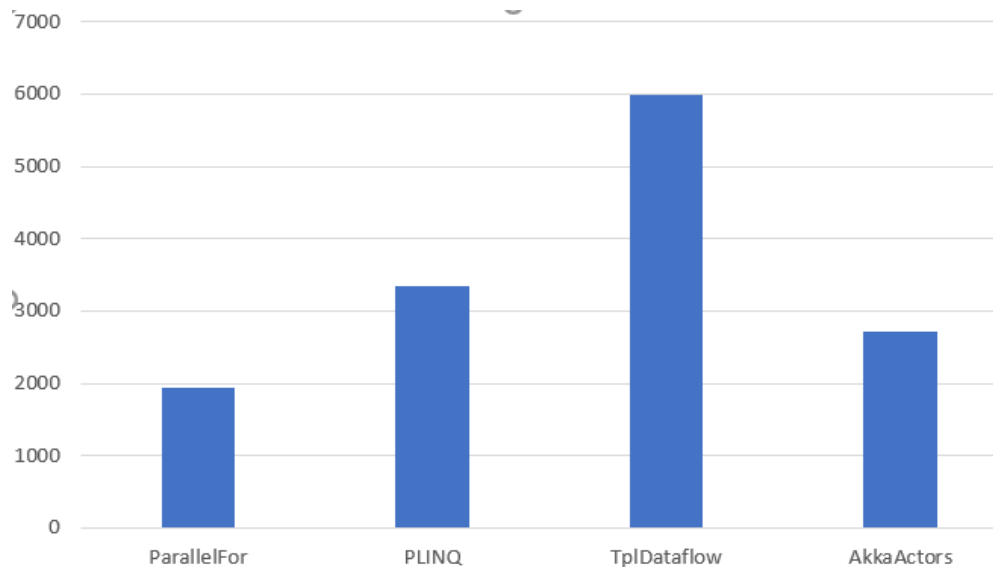


Рис. 3.5. Час множення матриць (n=1000 для різних моделей паралельності у C# (Parallel.For, PLINQ, TPL Dataflow, Akka.NET).

Як бачимо з рис. 3.5 при збільшенні навантаженні лідером став Parallel.For, PLINK покращив результат, а TPL Dataflow показує найгірший результат. На рис. 3.6 зображено результати моніторингу зайнятості процесора та пам'яті під час експерименту при n=1000. Як бачимо процесор завантажений майже постійно, але для пам'яті можна констатувати про відмінності використання для різних моделей, що пояснюється їхніми моделями.

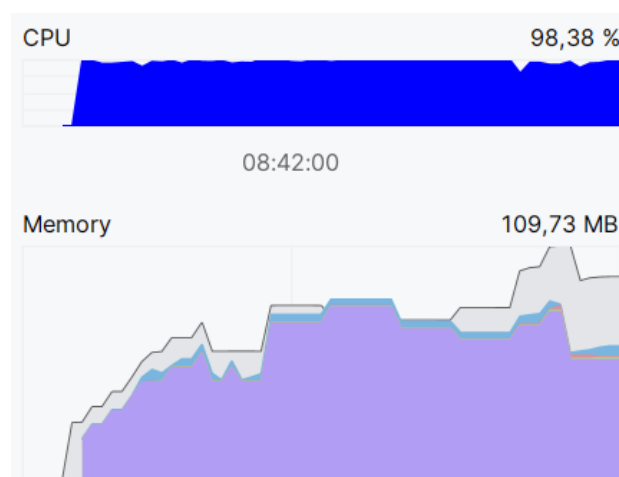


Рис. 3.6. Моніторинг CPU та Memory при t=1000.

У табл. 3.1 приведено результати для тесту множення матриць на Пайтоні. Форма представлення була обрана в силу значної різниці в результатах. Найкращий результат тут безумовно у NumPy_CPU. Multiprocessing показує стабільність щодо залежності «час»-«навантаження», це говорить про масштабованість.

Таблиця 3.1.

Порівняння продуктивності NumPy, multiprocessing та asyncio для задачі множення матриць у Python.

	100	250	500
Sequential	257,89	31874,91	285754,3
Multiprocessing	1489,27	1762,21	2517,14
NumPy_CPU	0,04	1,91	17,48
Asyncio	258,99	29657,25	258493,8

На рис. 3.7. показані результати сортування великих масивів у C# за методами: послідовне сортування, PLINQ та Task-based MergeSort

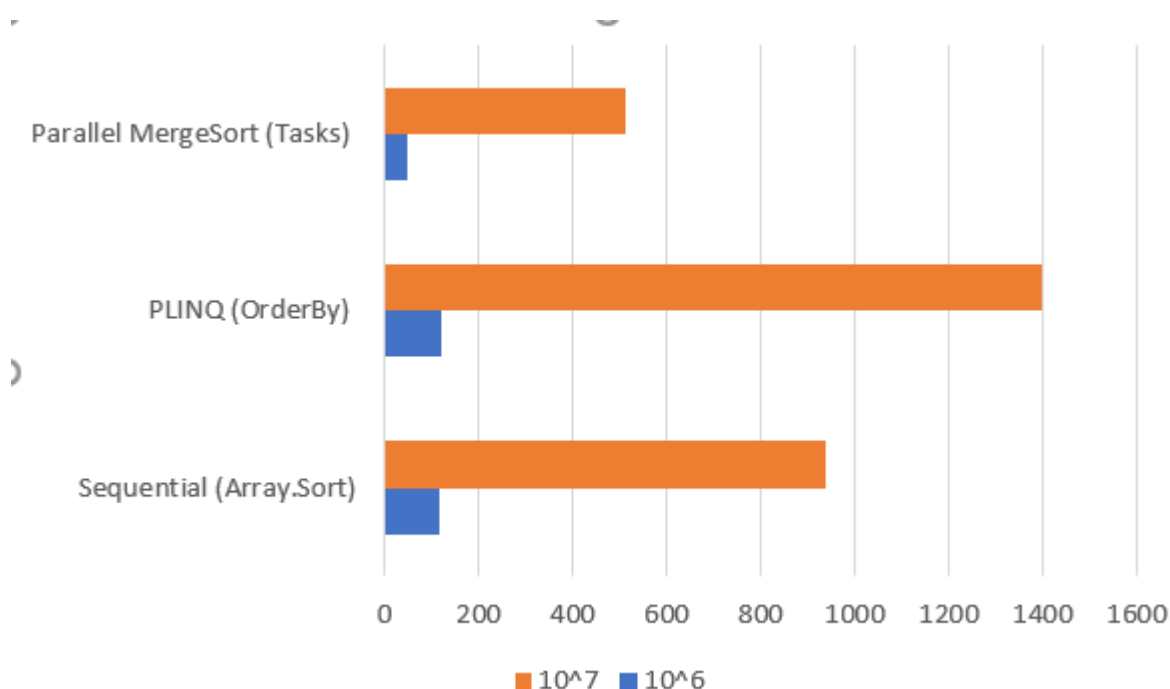


Рис. 3.7 Результати сортування великих масивів у C#: послідовне сортування, PLINQ та Task-based MergeSort.

Як бачимо, метод Paralell MergeSort показує найкращий результат за продуктивністю. Але масштабованість майже однакова для різних методів.

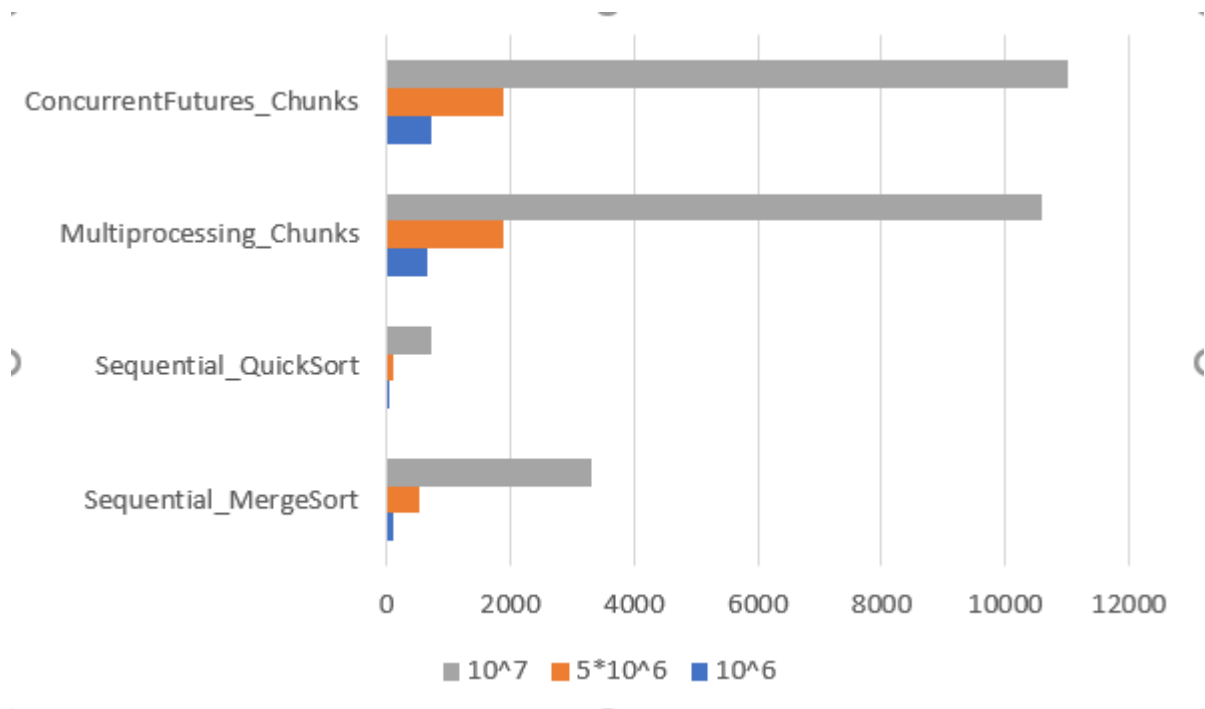


Рис. 3.8. Порівняння multiprocessing і concurrent.futures у Python при сортуванні масивів розміром 10^6 – 10^7 елементів.

Як бачимо з рис. 3.8, де представлено результати для сортування в Python, найкращі результати у методів Sequential.

На рис. 3.9-3.11 показані радіальні діаграми для розглянутих методів за трьома характеристиками, які були оцінені за методикою описаною в п.1.5 цієї кваліфікаційної роботи.

Рис. 3.9 інкапсулює результати експерименту за допомогою мови Python і відповідних бібліотек.

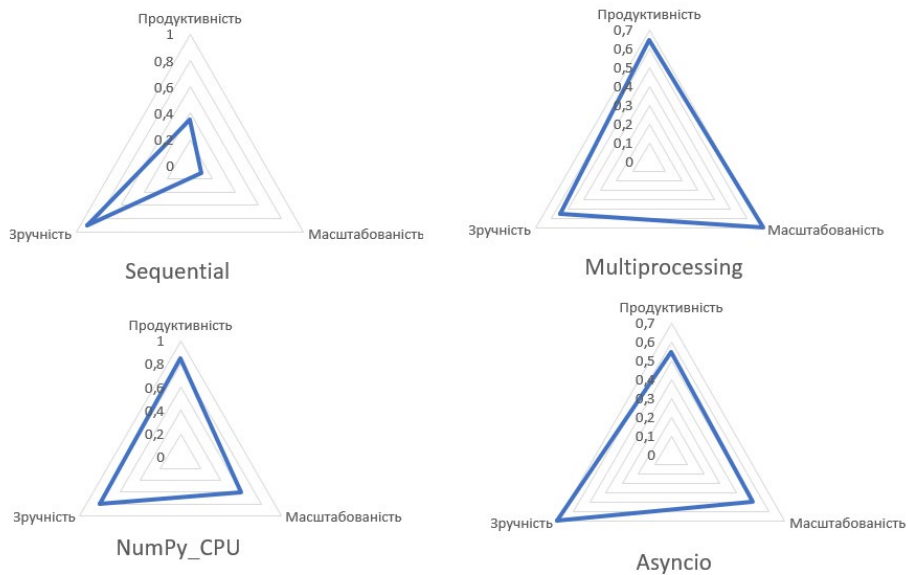


Рис. 3.9. Комплексна оцінка методів (1)

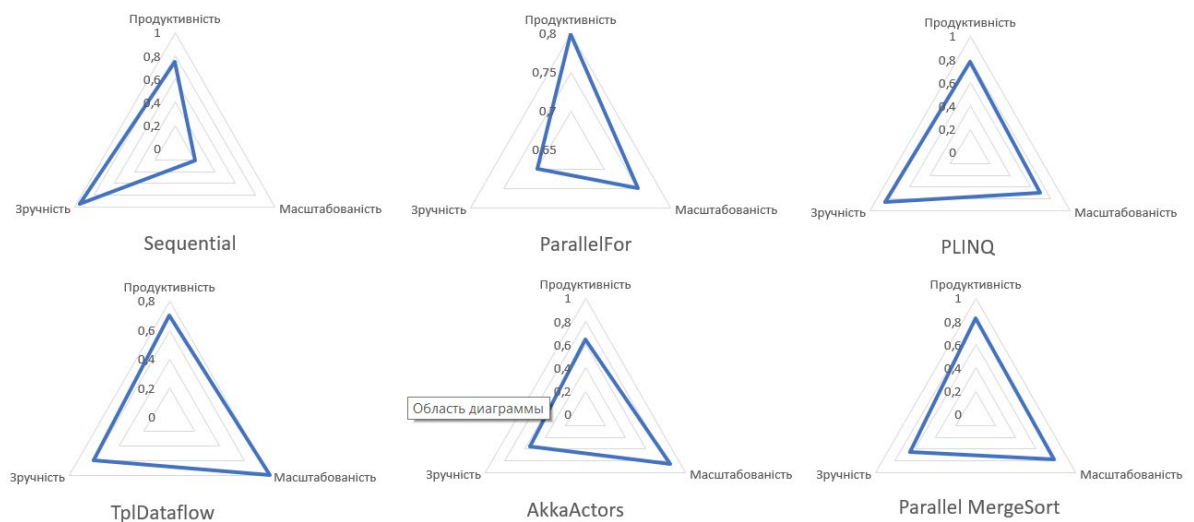


Рис. 3.10. Комплексна оцінка методів (2)

Узагальнюючи отримані результати, можна зазначити, що ефективність паралельних моделей істотно залежить від характеру задачі, накладних витрат на створення і синхронізацію потоків або процесів, а також від того, наскільки природно обчислення розпадаються на незалежні фрагменти. Моделі, орієнтовані на автоматичне розпаралелювання (як-от PLINQ або NumPy), демонструють найвищу продуктивність при мінімальній складності програмування, тоді як моделі низького рівня (Task-паралелізм, multiprocessing) забезпечують більший контроль, але вимагають більш

глибокого розуміння архітектури та особливостей виконання. Таким чином, експериментальна частина підтвердила, що вибір моделі паралельності має базуватися на оцінці структури задачі, необхідного рівня масштабованості та допустимої програмної складності, що є ключовими критеріями у сучасній інженерії програмного забезпечення.

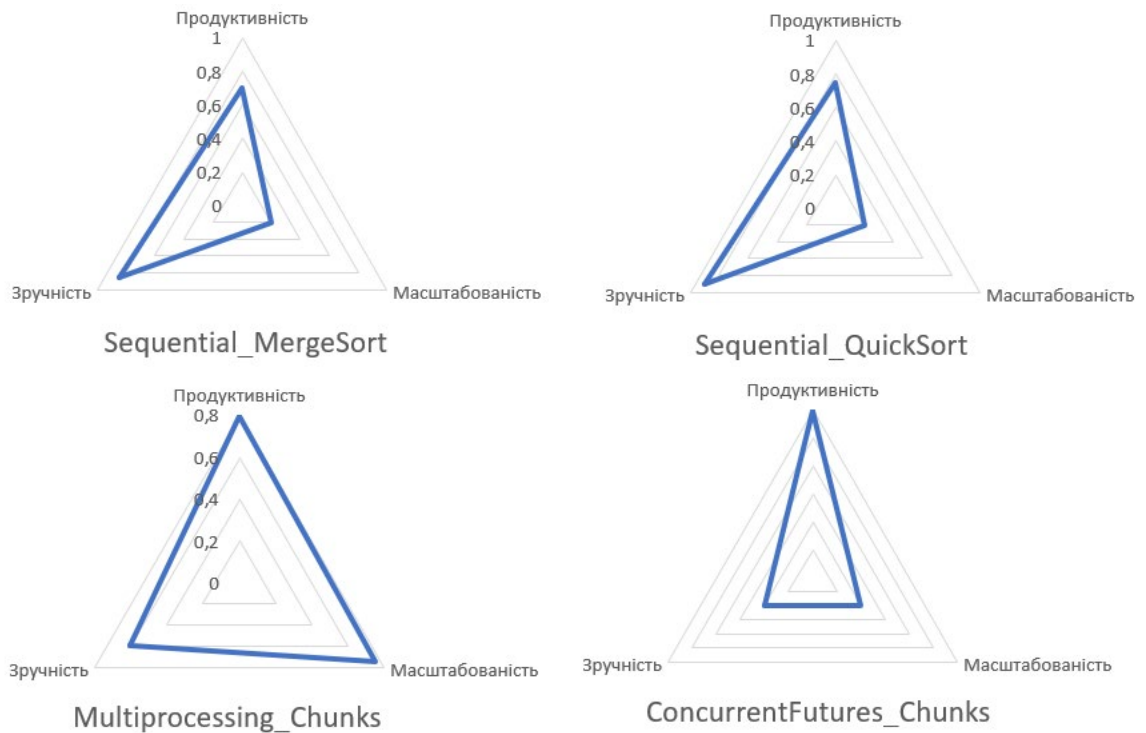


Рис. 3.11. Комплексна оцінка методів (3)

Отримані експериментальні дані дозволяють здійснити орієнтовне оцінювання масштабованості моделей паралельного програмування через зміну часу виконання при збільшенні обсягу обчислень. Масштабованість у даному випадку розуміється як здатність моделі ефективно використовувати додаткові апаратні ресурси - кількість ядер, пропускну здатність пам'яті та можливості GPU. Хоча експерименти не передбачали спеціально контролюваного варіювання числа доступних процесорних ядер, непрямі характеристики можна отримати шляхом порівняння темпів зростання часу виконання при збільшенні розміру задачі. Наприклад, у C# модель `Parallel.For` демонструє майже лінійне масштабування для задач множення матриць, де

робота природно розподіляється за рядками. PLINQ також показав добру масштабованість, але більшого впливу зазнав від накладних витрат планувальника, що спричинило менший ефект при збільшенні обсягу обчислень. Модель Task-based MergeSort масштабувалася найкраще серед алгоритмів сортування, оскільки її рекурсивна структура добре відповідає багаторівневій декомпозиції задачі.

У Python масштабованість виявилася нерівномірною. NumPy реалізує низькорівневе розпаралелювання всередині BLAS-бібліотек, що забезпечує дуже високий потенціал масштабування без будь-яких зусиль зі сторони розробника. Натомість multiprocessing демонструє гарне масштабування при сортуванні та множенні матриць за рахунок виконання кожного фрагмента обчислень у власному процесі, однак накладні витрати на передавання даних і створення процесів обмежують максимальний ефект. concurrent.futures показав подібну картину, але забезпечив кращу стабільність результатів. Моделі, засновані на asyncio, не є масштабованими для обчислень, що використовують CPU, проте можуть розглядатися як модель керування логічною конкурентністю для задач введення/виведення. Таким чином, загальна тенденція свідчить, що масштабованість тісно пов'язана з тим, наскільки природно задача розкладається на незалежні фрагменти та наскільки обрана модель містить додаткові накладні витрати.

Оцінювання зручності використання є суб'єктивним, але може спиратися на кількість написаного коду, його структурованість, складність налагодження та рівень абстракцій, які надає мова програмування чи бібліотека. Якщо аналізувати C#, то PLINQ легко інтегрується у звичайний LINQ-код і потребує мінімальних змін у структурі програми, що робить його найбільш зручним для швидкого прототипування. Parallel.For має більш низький рівень і вимагає від розробника контролю над межами паралельних циклів, але залишається доволі інтуїтивним. Найбільш складним для реалізації є Task-based MergeSort, який потребує ретельного керування глибиною рекурсії та уникнення надмірної кількості задач. Акторна модель Akka.NET

забезпечує високий рівень абстракції й природну ізоляцію стану, проте є громіздкою для простих обчислювальних задач і потребує детального розуміння акторної архітектури.

У Python найвищу зручність забезпечує NumPy, оскільки однооператорне множення матриць і сортування через вбудовані методи поєднують високу продуктивність із мінімальним обсягом коду. `concurrent.futures` також вважається доволі зручним завдяки простому інтерфейсу та автоматичному керуванню пулом процесів. `multiprocessing`, навпаки, є менш зручним через потребу у серіалізації даних та явному створенні процесів, хоча забезпечує ефективну паралельність. `asyncio` найменш зручний для обчислювальних задач, оскільки не дає реальної вигоди у швидкодії, проте може бути використаний для демонстрації концепції конкурентності. Загалом, Python-моделі набагато різноманітніші з точки зору синтаксису й абстракцій, але простіші моделі залишаються також і найпродуктивнішими.

Отже, навіть поверхове оцінювання масштабованості та суб'єктивна оцінка зручності використання дозволяють сформувати повнішу картину ефективності різних моделей паралельного програмування. Хоча для детального аналізу потрібні спеціалізовані інструменти профілювання та контроль апаратних параметрів, уже отримані результати дають змогу зробити обґрунтований висновок про те, які моделі є найбільш раціональними для тих чи інших типів задач, а також наскільки вони зручні для практичного застосування в інженерії програмного забезпечення.

ВИСНОВКИ

У процесі виконання магістерської роботи було досягнуто поставленої мети, яка полягала у здійсненні порівняльного аналізу одночасних і паралельних моделей програмування з позицій продуктивності, масштабованості та зручності використання. Отримані результати підтверджують актуальність дослідження, оскільки сучасні програмні системи дедалі частіше функціонують у багатоядерних, розподілених та гетерогенних обчислювальних середовищах, де ефективне використання паралельності є визначальним чинником продуктивності.

У межах першого завдання роботи було здійснено аналіз міжнародних наукових публікацій, зокрема джерел, індексованих у базі Scopus. Результати аналізу показали, що сучасні дослідження зосереджені на поєднанні високої продуктивності з переносимістю та зручністю програмування. Значну увагу приділено автоматизованому вилученню паралелізму, оптимізаціям на рівні компіляторів, акторним і повідомлювальним моделям, а також спеціалізованим структурам даних для GPU та розподілених середовищ. Це підтвердило, що жодна універсальна модель паралельності не є оптимальною для всіх класів задач, а вибір підходу повинен враховувати характер обчислень і цільову платформу.

У межах другого завдання роботи було розроблено узагальнену класифікацію моделей одночасного та паралельного програмування, яка охоплює потокові та задачні моделі, акторну модель, підходи зі спільною та розподіленою пам'яттю, GPU-орієнтовані рішення та автоматизовані моделі паралелізації. Запропонована класифікація дозволяє систематизувати різноманітні підходи, що використовуються у сучасних програмних системах, та створює методологічну основу для їх порівняння за єдиними критеріями.

У ході виконання третього завдання було сформульовано критерії оцінювання ефективності моделей паралельного програмування. Основними

кількісними показниками визначено час виконання, пропускну здатність та накладні витрати синхронізації, тоді як масштабованість оцінювалася через зміну продуктивності при зростанні обсягу обчислень. Додатково враховувалася апаратна незалежність і можливість перенесення реалізацій між різними платформами. Такий підхід дозволив комплексно оцінити не лише швидкодію, але й практичну придатність моделей у реальних умовах розробки програмного забезпечення.

Четверте завдання було реалізовано шляхом розробки та експериментального дослідження низки прототипів для типових обчислювальних задач, зокрема множення матриць і сортування великих масивів даних. Прототипи було реалізовано мовами C# та Python із використанням різних моделей паралельності, таких як Parallel.For, PLINQ, TPL Dataflow, акторна модель Akka.NET, multiprocessing, concurrent.futures, NumPy та асинхронні підходи. Отримані експериментальні дані продемонстрували суттєві відмінності у продуктивності та масштабованості моделей, а також підтвердили, що автоматизовані й бібліотечні рішення часто забезпечують найкраще співвідношення між швидкодією та складністю реалізації.

У рамках п'ятого завдання було сформульовано узагальнені рекомендації щодо вибору моделей одночасності та паралелізму для різних типів програмних систем. Зокрема, для обчислювально інтенсивних задач зі зрозумілою структурою паралелізму доцільним є використання потокових і задачних моделей або векторизованих бібліотек. Для складних конкурентних систем з асинхронною взаємодією компонентів більш доцільними є акторні та повідомлювальні моделі. Для задач обробки великих масивів даних і чисельних обчислень найефективнішими є GPU-орієнтовані та бібліотечні підходи. Отримані рекомендації підтверджені експериментальними результатами та можуть бути використані як у навчальному процесі, так і в практичній діяльності інженерів програмного забезпечення.

Таким чином, виконане дослідження довело, що ефективно використання одночасності та паралелізму потребує не лише знання окремих технологій, але й системного підходу до вибору моделей програмування. Поєднання теоретичного аналізу, експериментальних досліджень і практичних рекомендацій дозволяє розглядати результати роботи як внесок у методологію проєктування високопродуктивних програмних систем у сучасних обчислювальних середовищах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A Highly Configurable High-Level Synthesis Functional Pattern Library / L. Huang et al. *Electronics*. 2021. Vol. 10, no. 5. P. 532. URL: <https://doi.org/10.3390/electronics10050532> (date of access: 29.11.2025).
2. Binet S. Can Go address the multicore issues of today and the manycore problems of tomorrow?. *Journal of Physics: Conference Series*. 2012. Vol. 368. P. 012017. URL: <https://doi.org/10.1088/1742-6596/368/1/012017> (date of access: 29.11.2025).
3. Concurrent programming in ERLANG. / ed. by A. Joe, A. Joe. 2nd ed. London : Prentice Hall, 1996. 351 p.
4. Cyrus / N. Honarmand et al. *ACM SIGARCH Computer Architecture News*. 2013. Vol. 41, no. 1. P. 193–206. URL: <https://doi.org/10.1145/2490301.2451138> (date of access: 29.11.2025).
5. Dijkstra E. W. Cooperating Sequential Processes. *The Origin of Concurrent Programming*. New York, NY, 1968. P. 65–138. URL: https://doi.org/10.1007/978-1-4757-3472-0_2 (date of access: 29.11.2025).
6. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*. 2001. Vol. 23, no. 6. P. 747–775. URL: <https://doi.org/10.1145/506315.506317> (date of access: 29.11.2025).
7. Hemnani M. Parallel processing techniques for high performance image processing applications. *2016 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, Bhopal, India, 5–6 March 2016. 2016. URL: <https://doi.org/10.1109/sceecs.2016.7509316> (date of access: 29.11.2025).
8. Hennessy J. L., Patterson D. A. Computer Architecture: A Quantitative Approach. Elsevier Science & Technology Books, 2017. 936 p.

9. Hoare C. A. R. Communicating sequential processes. *Communications of the ACM*. 1983. Vol. 26, no. 1. P. 100–106.
URL: <https://doi.org/10.1145/357980.358021> (date of access: 29.11.2025).
10. Hoare C. A. Towards a Theory of Parallel Programming. *Programming Methodology*. New York, NY, 1978. P. 202–214.
URL: https://doi.org/10.1007/978-1-4612-6315-9_16 (date of access: 29.11.2025).
11. Interpreting the Data: Parallel Analysis with Sawzall / R. Pike et al. *Scientific Programming*. 2005. Vol. 13, no. 4. P. 277–298.
URL: <https://doi.org/10.1155/2005/962135> (date of access: 29.11.2025).
12. Juurlink B. H. H., Wijshoff H. A. G. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*. 1998. Vol. 16, no. 3. P. 271–318.
URL: <https://doi.org/10.1145/290409.290412> (date of access: 29.11.2025).
13. Lamport L. Real-Time Model Checking Is Really Simple. *Lecture Notes in Computer Science*. Berlin, Heidelberg, 2005. P. 162–175.
URL: https://doi.org/10.1007/11560548_14 (date of access: 29.11.2025).
14. Lamport L. Time, Clocks, and the Ordering of My Ideas About Distributed Systems. *Lecture Notes in Computer Science*. Berlin, Heidelberg, 2006. P. 578. URL: https://doi.org/10.1007/11864219_51 (date of access: 29.11.2025).
15. Lea D. A Java fork/join framework. *the ACM 2000 conference*, San Francisco, California, United States, 3–4 June 2000. New York, New York, USA, 2000. URL: <https://doi.org/10.1145/337449.337465> (date of access: 29.11.2025).
16. Manta IDE. *GitHub*. URL: <https://github.com/manta-ide/manta>.
17. Margara A., Cugola G. High-Performance Publish-Subscribe Matching Using Parallel Hardware. *IEEE Transactions on Parallel and Distributed Systems*. 2014. Vol. 25, no. 1. P. 126–135.
URL: <https://doi.org/10.1109/tpds.2013.39> (date of access: 29.11.2025).

18. Moreira J. E., Midkiff S. P., Gupta M. From flop to megaflops. *ACM Transactions on Programming Languages and Systems*. 2000. Vol. 22, no. 2. P. 265–295. URL: <https://doi.org/10.1145/349214.349222> (date of access: 29.11.2025).
19. Narlikar G. J., Bluelloch G. E. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*. 1999. Vol. 21, no. 1. P. 138–173. URL: <https://doi.org/10.1145/314602.314607> (date of access: 29.11.2025).
20. Performance comparison of various STM concurrency control protocols using synchrobench / A. Singh et al. *2017 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, Bangalore, India, 23–24 February 2017. URL: <https://doi.org/10.1109/parcomptech.2017.8068330> (date of access: 29.11.2025).
21. Performance Evaluation of Process Partitioning Using Probabilistic Model Checking / S. Bensalem et al. *Hardware and Software: Verification and Testing*. Cham, 2013. P. 344–358. URL: https://doi.org/10.1007/978-3-319-03077-7_23 (date of access: 29.11.2025).
22. Performance evaluation of the Orca shared-object system / H. E. Bal et al. *ACM Transactions on Computer Systems*. 1998. Vol. 16, no. 1. P. 1–40. URL: <https://doi.org/10.1145/273011.273014> (date of access: 29.11.2025).
23. Polak M. S. H., Troendle D. A., Jang B. Boundary-Aware Concurrent Queue: A Fast and Scalable Concurrent FIFO Queue on GPU Environments. *Applied Sciences*. 2025. Vol. 15, no. 4. P. 1834. URL: <https://doi.org/10.3390/app15041834> (date of access: 29.11.2025).
24. Radenski A. A. Object-oriented programming and parallelism: Introduction. *Information Sciences*. 1996. Vol. 93, no. 1-2. P. 1–7. URL: [https://doi.org/10.1016/0020-0255\(96\)00058-8](https://doi.org/10.1016/0020-0255(96)00058-8) (date of access: 29.11.2025).

25. Rinard M. C., Lam M. S. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*. 1998. Vol. 20, no. 3. P. 483–545. URL: <https://doi.org/10.1145/291889.291893> (date of access: 29.11.2025).
26. Smulovics P. Analysis of Actor Model Frameworks. *Linkedin*. URL: <https://www.linkedin.com/pulse/analysis-actor-model-frameworks-peter-smulovics/>.
27. Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture / D. Camp et al. *IEEE Transactions on Visualization and Computer Graphics*. 2011. Vol. 17, no. 11. P. 1702–1713. URL: <https://doi.org/10.1109/tvcg.2010.259> (date of access: 29.11.2025).
28. Tang H., Shen K., Yang T. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems*. 2000. Vol. 22, no. 4. P. 673–700. URL: <https://doi.org/10.1145/363911.363920> (date of access: 29.11.2025).
29. The occam-pi programming language. *The occam-pi*. URL: <http://occam-pi.org/>.
30. The OpenMP Cluster Programming Model / H. Yviquel et al. *ICPP '22: 51st International Conference on Parallel Processing*, Bordeaux France. New York, NY, USA, 2022. URL: <https://doi.org/10.1145/3547276.3548444> (date of access: 29.11.2025).
31. Valiant L. G. A bridging model for parallel computation. *Communications of the ACM*. 1990. Vol. 33, no. 8. P. 103–111. URL: <https://doi.org/10.1145/79173.79181> (date of access: 03.12.2025).
32. Zain-ul-Abdin, Ahlander A., Svensson B. Programming Real-Time Autofocus on a Massively Parallel Reconfigurable Architecture Using Occam-pi. *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Salt Lake City, UT, USA, 1–3 May 2011. 2011. URL: <https://doi.org/10.1109/fccm.2011.20> (date of access: 29.11.2025).

ДОДАТКИ

Додаток А Порівняльна таблиця різних моделей

Модель	Принцип роботи	Синхронізація	Переваги	Обмеження	Типові сфери застосування
Потокова	Спільна пам'ять, паралельні потоки в одному процесі	Ручні примітиви (mutex, semaphore)	Максимальний контроль, низький оверхед	Гонки даних, взаємоблокування, складність відлагодження	Системне програмування, високопродуктивні локальні застосунки
Конкурентна (канали/черги)	Взаємодія через канали або черги повідомлень	Автоматична, залежно від черг	Менше помилок синхронізації, модульність	Чутливість до затримок, складність балансування	Мікросервіси, реактивні системи, обробка подій
Акторна	Актори з приватним станом і mailbox	Відсутня спільна пам'ять	Масштабованість, відмовостійкість, ізоляція	Нестабільна латентність, порядок доставки не гарантовано	Розподілені сервіси, телеком, real-time обробка
BSP	Суперетапи: локальні обчислення + комунікація + бар'єр	Бар'єрна синхронізація	Прогнозованість продуктивності	Простіювання через нерівномірність	Кластери, HPC, наукові обчислення

Модель	Принцип роботи	Синхронізація	Переваги	Обмеження	Типові сфери застосування
BPRAM	Теоретична модель розподіленої пам'яті	Аналітична	Аналіз масштабованості	Мало практичних реалізацій	Оцінювання алгоритмів, теоретичні дослідження
Orca	Розподілена спільна пам'ять через спільні об'єкти	Прозоре узгодження стану	Спрощена модель порівняно з повідомленнями	Накладні витрати на реплікацію	Розподілені об'єктні системи
Manta	Швидкі RMI, оптимізована серіалізація	Автоматична	Дуже низька латентність RMI	Залежність від мережі та трафіку	Розподілені застосунки з частими викликами методів
occam-pi	Процеси + канали, формальна модель CSP + π -числення	Канальна синхронізація	Строгість, безпечність	Складний синтаксис, високий поріг входження	Вбудовані системи, критичні розподілені застосунки
Jade	Декларативні специфікації доступу до даних	Автоматична	Автовилучення паралелізму	Обмеження складними залежностями	Масивні дані, структуровані обчислення

Модель	Принцип роботи	Синхронізація	Переваги	Обмеження	Типові сфери застосування
Cyrus	Логування доступів + автоматичний граф виконання	Автоматична	Паралелізація послідовного коду	Високі накладні витрати	Динамічні обчислення, реконструкція паралельності
BACQ GPU	Масивний паралелізм, сегментована черга для тисяч потоків	Локальна блокування в межах сегментів	Дуже висока продуктивність	Гетерогенність, складність реалізації	GPU-алгоритми, графи, потокові дані

Додаток Б. Код С# для множення двох матриць

```
using System;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using Akka.Actor;

namespace MatrixBenchmarks
{
    class Program
    {
        private const string CsvPath = "matrix_results.csv";

        static void Main(string[] args)
        {
            int n = args.Length > 0 ? int.Parse(args[0]) : 500;    // розмір матриці N×N
            int iterations = args.Length > 1 ? int.Parse(args[1]) : 3; // кількість повторів на одну модель

            Console.WriteLine($"Matrix size: {n}x{n}, iterations per model: {iterations}");

            // Генеруємо вхідні матриці один раз (для чесного порівняння)
            var A = MatrixUtils.GenerateRandomMatrix(n);
            var B = MatrixUtils.GenerateRandomMatrix(n);

            EnsureCsvHeader();

            BenchmarkModel("ParallelFor", n, iterations, () => MatrixMultiplierParallelFor.Multiply(A, B));
            BenchmarkModel("PLINQ", n, iterations, () => MatrixMultiplierPLINQ.Multiply(A, B));
            BenchmarkModel("TplDataflow", n, iterations, () => MatrixMultiplierTplDataflow.Multiply(A, B));
            BenchmarkModel("AkkaActors", n, iterations, () => MatrixMultiplierAkka.Multiply(A, B));

            Console.WriteLine("Done. Results appended to " + CsvPath);
        }

        private static void BenchmarkModel(string modelName, int n, int iterations, Func<double[,]>
multiplyFunc)
        {
            Console.WriteLine($"Running {modelName}...");

            // "розминка", щоб прогріти JIT
            multiplyFunc();

            var sw = Stopwatch.StartNew();
            for (int i = 0; i < iterations; i++)
            {
```

```

        var result = multiplyFunc();
        // результат можна використати/перевірити, щоб уникнути оптимізацій,
        // але зазвичай достатньо просто створення.
    }
    sw.Stop();

    double totalMs = sw.Elapsed.TotalMilliseconds;
    double perIterMs = totalMs / iterations;

    Console.WriteLine($"{modelName}: total = {totalMs:F2} ms, per iter = {perIterMs:F2} ms");

    AppendCsvRow(modelName, n, iterations, totalMs, perIterMs);
}

private static void EnsureCsvHeader()
{
    if (!File.Exists(CsvPath))
    {
        var header = "Timestamp,Model,N,Iterations,TotalMs,PerIterationMs";
        File.AppendAllText(CsvPath, header + Environment.NewLine);
    }
}

private static void AppendCsvRow(string modelName, int n, int iterations, double totalMs, double
perIterMs)
{
    var line = $"{DateTime.UtcNow:O},{modelName},{n},{iterations},{totalMs:F4},{perIterMs:F4}";
    File.AppendAllText(CsvPath, line + Environment.NewLine);
}

static class MatrixUtils
{
    private static readonly Random Rng = new Random();

    public static double[,] GenerateRandomMatrix(int n)
    {
        var m = new double[n, n];
        lock (Rng)
        {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    m[i, j] = Rng.NextDouble();
        }
        return m;
    }
}

#region Parallel.For

```

```

static class MatrixMultiplierParallelFor
{
    public static double[,] Multiply(double[,] A, double[,] B)
    {
        int n = A.GetLength(0);
        var result = new double[n, n];

        Parallel.For(0, n, i =>
        {
            for (int j = 0; j < n; j++)
            {
                double sum = 0;
                for (int k = 0; k < n; k++)
                    sum += A[i, k] * B[k, j];
                result[i, j] = sum;
            }
        });

        return result;
    }
}

```

#endregion

#region PLINQ

```

static class MatrixMultiplierPLINQ
{
    public static double[,] Multiply(double[,] A, double[,] B)
    {
        int n = A.GetLength(0);
        var result = new double[n, n];

        var indices =
            from i in Enumerable.Range(0, n)
            from j in Enumerable.Range(0, n)
            select new { i, j };

        indices
            .AsParallel()
            .ForAll(p =>
            {
                double sum = 0;
                for (int k = 0; k < n; k++)
                    sum += A[p.i, k] * B[k, p.j];
                result[p.i, p.j] = sum;
            });
    }
}

```

```

        return result;
    }
}

#endregion

#region TPL Dataflow

static class MatrixMultiplierTplDataflow
{
    public static double[,] Multiply(double[,] A, double[,] B)
    {
        int n = A.GetLength(0);
        var result = new double[n, n];

        var options = new ExecutionDataflowBlockOptions
        {
            MaxDegreeOfParallelism = Environment.ProcessorCount
        };

        var workerBlock = new ActionBlock<(int i, int j)>(tuple =>
        {
            int i = tuple.i;
            int j = tuple.j;
            double sum = 0;
            for (int k = 0; k < n; k++)
                sum += A[i, k] * B[k, j];
            result[i, j] = sum;
        }, options);

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                workerBlock.Post((i, j));

        workerBlock.Complete();
        workerBlock.Completion.Wait();

        return result;
    }
}

#endregion

#region Akka.NET Actors

// Повідомлення для актора – який рядок обчислювати
public sealed class RowWork
{
    public int RowIndex { get; }
}

```



```

    public RowWork(int rowIndex) => RowIndex = rowIndex;
}

// Актор, який обчислює один рядок матриці результату
public class RowActor : ReceiveActor
{
    private readonly double[,] _a;
    private readonly double[,] _b;
    private readonly double[,] _result;
    private readonly int _n;
    private readonly CountdownEvent _countdown;

    public RowActor(double[,] a, double[,] b, double[,] result, CountdownEvent countdown)
    {
        _a = a;
        _b = b;
        _result = result;
        _n = a.GetLength(0);
        _countdown = countdown;

        Receive<RowWork>(msg =>
        {
            int i = msg.RowIndex;
            for (int j = 0; j < _n; j++)
            {
                double sum = 0;
                for (int k = 0; k < _n; k++)
                {
                    sum += _a[i, k] * _b[k, j];
                }
                _result[i, j] = sum;
            }

            _countdown.Signal();
        });
    }
}

static class MatrixMultiplierAkka
{
    public static double[,] Multiply(double[,] A, double[,] B)
    {
        int n = A.GetLength(0);
        var result = new double[n, n];
        var countdown = new CountdownEvent(n);

        using (var system = ActorSystem.Create("MatrixSystem"))
        {
            // Створюємо по одному актору на рядок (для простоти)
            for (int i = 0; i < n; i++)

```

```

    {
        var rowActor = system.ActorOf(
            Props.Create(() => new RowActor(A, B, result, countdown)),
            $"rowActor-{i}");

        rowActor.Tell(new RowWork(i));
    }

    // Чекаємо, доки всі рядки будуть обчислені
    countdown.Wait();
    system.Terminate().Wait();
}

return result;
}
}

#endregion
}

```

Додаток В. Код множення матриць на Python

```
#!/usr/bin/env python3
import argparse
import asyncio
import csv
import datetime as dt
import os
import time
from functools import partial
from multiprocessing import Pool, cpu_count

import numpy as np

# Опційні імпорти для GPU
try:
    import numba
    from numba import njit, prange
except ImportError:
    numba = None

try:
    import cupy as cp
except ImportError:
    cp = None

CSV_PATH = "matrix_results_py.csv"

# ----- Утиліти для CSV -----

def ensure_csv_header():
    if not os.path.exists(CSV_PATH):
        with open(CSV_PATH, "w", newline="", encoding="utf-8") as f:
            writer = csv.writer(f)
            writer.writerow(["Timestamp", "Model", "N", "Iterations", "TotalMs", "PerIterationMs"])

def append_csv_row(model_name: str, n: int, iterations: int, total_ms: float, per_iter_ms: float):
    with open(CSV_PATH, "a", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow([
            dt.datetime.utcnow().isoformat(),
            model_name,
            n,
            iterations,
            f"{total_ms:.4f}",
            f"{per_iter_ms:.4f}"
        ])
```

```
)
```

```
def benchmark_model(model_name: str, n: int, iterations: int, func, A, B):  
    """  
    func(A, B) має повертати результат матричного множення.  
    """  
    print(f"Running {model_name}...")  
  
    # "Розминка" для прогріву JIT/кешів  
    func(A, B)  
  
    start = time.perf_counter()  
    for _ in range(iterations):  
        C = func(A, B)  
        # щоб інтерпретатор не викинув результат, можемо мінімально використати C  
        _ = C[0, 0]  
    total = time.perf_counter() - start  
  
    total_ms = total * 1000.0  
    per_iter_ms = total_ms / iterations  
  
    print(f"{model_name}: total = {total_ms:.2f} ms, per iter = {per_iter_ms:.2f} ms")  
    append_csv_row(model_name, n, iterations, total_ms, per_iter_ms)  
  
# ----- Генерація матриць -----  
  
def generate_random_matrix(n: int) -> np.ndarray:  
    return np.random.rand(n, n).astype(np.float64)  
  
# ----- 1. Послідовна реалізація -----  
  
def multiply_sequential(A: np.ndarray, B: np.ndarray) -> np.ndarray:  
    n = A.shape[0]  
    C = np.zeros((n, n), dtype=np.float64)  
    for i in range(n):  
        for j in range(n):  
            s = 0.0  
            for k in range(n):  
                s += A[i, k] * B[k, j]  
            C[i, j] = s  
    return C  
  
# ----- 2. Multiprocessing (розбиття по рядках) -----  
  
# Глобальні змінні для воркерів
```

```
_A_mp = None
_B_mp = None
```

```
def _init_worker(A: np.ndarray, B: np.ndarray):
    global _A_mp, _B_mp
    _A_mp = A
    _B_mp = B
```

```
def _mp_row_worker(i: int) -> np.ndarray:
    # Обчислення i-го рядка: це вектор (1×N)
    global _A_mp, _B_mp
    # Використовуємо NumPy dot для рядка
    row = np.dot(_A_mp[i, :], _B_mp)
    return row
```

```
def multiply_multiprocessing(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    n = A.shape[0]
    # Використовуємо Pool, ініціалізуємо глобальні A,B
    with Pool(processes=cpu_count(), initializer=_init_worker, initargs=(A, B)) as pool:
        rows = pool.map(_mp_row_worker, range(n))
    C = np.vstack(rows)
    return C
```

----- 3. NumPy (CPU, векторизовано) -----

```
def multiply_numpy(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    return A @ B
```

----- 4. Numba / CuPy (GPU якщо є) -----

```
if numba is not None:
    @jit(parallel=True, fastmath=True)
    def _numba_matmul(A, B):
        n = A.shape[0]
        C = np.zeros((n, n), dtype=np.float64)
        for i in prange(n):
            for j in range(n):
                s = 0.0
                for k in range(n):
                    s += A[i, k] * B[k, j]
                C[i, j] = s
        return C
else:
    _numba_matmul = None
```

```

def multiply_numba_or_cupy(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    """
    Якщо доступні cyru + GPU, виконуємо на GPU.
    Якщо cyru немає, але є numba – прискорений CPU-варіант.
    Якщо нічого немає – fallback на numpy.
    """
    # Варіант 1: CuPy (GPU)
    if cp is not None:
        dA = cp.asarray(A)
        dB = cp.asarray(B)
        dC = cp.dot(dA, dB)
        C = cp.asnumpy(dC)
        return C

    # Варіант 2: Numba (CPU JIT)
    if _numba_matmul is not None:
        return _numba_matmul(A, B)

    # fallback
    return multiply_numpy(A, B)

# ----- 5. asyncio (імітація конкурентності) -----

def _row_multiply_seq(i: int, A: np.ndarray, B: np.ndarray) -> np.ndarray:
    # аналог послідовної реалізації для одного рядка
    n = A.shape[0]
    row = np.zeros((n,), dtype=np.float64)
    for j in range(n):
        s = 0.0
        for k in range(n):
            s += A[i, k] * B[k, j]
        row[j] = s
    return row

async def _asyncio_multiply_internal(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    loop = asyncio.get_running_loop()
    n = A.shape[0]
    C = np.zeros((n, n), dtype=np.float64)

    # Використаємо ThreadPoolExecutor за замовчуванням (executor=None)
    tasks = []
    for i in range(n):
        # частина роботи з CPU-bound кодом у треді (імітація конкурентності)
        task = loop.run_in_executor(
            None,

```

```

        partial(_row_multiply_seq, i, A, B)
    )
    tasks.append(task)

rows = await asyncio.gather(*tasks)
for i, row in enumerate(rows):
    C[i, :] = row
return C

def multiply_asyncio(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    return asyncio.run(_asyncio_multiply_internal(A, B))

# ----- main -----

def main():
    parser = argparse.ArgumentParser(description="Matrix multiplication benchmarks in Python")
    parser.add_argument("--n", type=int, default=500, help="Matrix size N (N x N)")
    parser.add_argument("--iterations", type=int, default=3, help="Iterations per model")
    args = parser.parse_args()

    n = args.n
    iterations = args.iterations

    print(f"Matrix size: {n}x{n}, iterations per model: {iterations}")

    # Генеруємо спільні матриці
    A = generate_random_matrix(n)
    B = generate_random_matrix(n)

    ensure_csv_header()

    # 1. Послідовно
    benchmark_model("Sequential", n, iterations, multiply_sequential, A, B)

    # 2. Multiprocessing
    benchmark_model("Multiprocessing", n, iterations, multiply_multiprocessing, A, B)

    # 3. NumPy (CPU)
    benchmark_model("NumPy_CPU", n, iterations, multiply_numpy, A, B)

    # 4. Numba / CuPy (GPU, якщо доступно)
    if cp is not None or _numba_matmul is not None:
        benchmark_model("Numba_CuPy", n, iterations, multiply_numba_or_cupy, A, B)
    else:
        print("Numba/CuPy not available, skipping GPU/JIT variant.")

    # 5. asyncio (імітація конкурентності)

```

```
benchmark_model("Asyncio", n, iterations, multiply_asyncio, A, B)

print(f"Done. Results appended to {CSV_PATH}")

if __name__ == "__main__":
    main()
```


Додаток Г. Код на С# для сортування великого масиву

```
using System;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading.Tasks;

namespace LargeArraySorting
{
    class Program
    {
        private const string CsvPath = "sorting_results.csv";

        static void Main(string[] args)
        {
            int n = args.Length > 0 ? int.Parse(args[0]) : 1_000_000; // Розмір масиву
            int iterations = args.Length > 1 ? int.Parse(args[1]) : 3; // Кількість повторів

            Console.WriteLine($"Array size: {n}, iterations per model: {iterations}");
            Console.WriteLine("Generating random array...");
            double[] baseData = GenerateRandomArray(n);

            EnsureCsvHeader();

            Benchmark("Sequential (Array.Sort)", n, iterations,
                arr => SequentialSort(arr));

            Benchmark("PLINQ (OrderBy)", n, iterations,
                arr => PlinqSort(arr));

            Benchmark("Parallel MergeSort (Tasks)", n, iterations,
                arr => ParallelMergeSortWrapper(arr));

            Console.WriteLine("\nDone. Results appended to " + CsvPath);
        }

        // ----- CSV UTILITIES -----

        private static void EnsureCsvHeader()
        {
            if (!File.Exists(CsvPath))
            {
                File.AppendAllText(CsvPath,
                    "Timestamp,Model,N,Iterations,TotalMs,PerIterationMs" + Environment.NewLine);
            }
        }

        private static void AppendCsvRow(
```

```

        string modelName, int n, int iterations,
        double totalMs, double perIterMs)
    {
        var line = $"{DateTime.UtcNow:O},{modelName},{n},{iterations},{totalMs:F4},{perIterMs:F4}";
        File.AppendAllText(CsvPath, line + Environment.NewLine);
    }

// ----- BENCHMARK -----

private static void Benchmark(
    string modelName,
    int n,
    int iterations,
    Func<double[], double[]> sortFunc)
{
    Console.WriteLine($"{n}=== {modelName} ===");

    // Прорпів
    sortFunc(GenerateRandomArray(50000));

    var sw = Stopwatch.StartNew();

    for (int i = 0; i < iterations; i++)
    {
        double[] data = GenerateRandomArray(n);
        var result = sortFunc(data);

        // Використовуємо елемент, щоб запобігти оптимізації
        if (result.Length > 0 && double.IsNaN(result[0]))
            Console.WriteLine("Impossible, placeholder.");
    }

    sw.Stop();
    double totalMs = sw.Elapsed.TotalMilliseconds;
    double perIterMs = totalMs / iterations;

    Console.WriteLine(
        $"{modelName}: total = {totalMs:F2} ms, per iteration = {perIterMs:F2} ms");

    AppendCsvRow(modelName, n, iterations, totalMs, perIterMs);
}

// ----- ARRAY GENERATION -----

private static double[] GenerateRandomArray(int n)
{
    var rnd = new Random();
    var arr = new double[n];
    for (int i = 0; i < n; i++)

```

```

        arr[i] = rnd.NextDouble();
    return arr;
}

// ----- 1. SEQUENTIAL -----

private static double[] SequentialSort(double[] data)
{
    var copy = new double[data.Length];
    Array.Copy(data, copy, data.Length);

    Array.Sort(copy);
    return copy;
}

// ----- 2. PLINQ -----

private static double[] PlinqSort(double[] data)
{
    return data
        .AsParallel()
        .OrderBy(x => x)
        .ToArray();
}

// ----- 3. PARALLEL MERGESORT -----

private static double[] ParallelMergeSortWrapper(double[] data)
{
    var copy = new double[data.Length];
    Array.Copy(data, copy, data.Length);

    var aux = new double[data.Length];
    int maxDepth = (int)Math.Log2(Environment.ProcessorCount) + 2;

    ParallelMergeSort(copy, aux, 0, data.Length - 1, maxDepth);
    return copy;
}

private static void ParallelMergeSort(double[] arr, double[] aux,
                                     int left, int right, int depthRemaining)
{
    const int SEQUENTIAL_THRESHOLD = 10_000;

    if (right - left <= SEQUENTIAL_THRESHOLD || depthRemaining <= 0)
    {
        Array.Sort(arr, left, right - left + 1);
        return;
    }
}

```

```

    int mid = (left + right) / 2;

    Task leftTask = Task.Run(() =>
        ParallelMergeSort(arr, aux, left, mid, depthRemaining - 1));

    Task rightTask = Task.Run(() =>
        ParallelMergeSort(arr, aux, mid + 1, right, depthRemaining - 1));

    Task.WaitAll(leftTask, rightTask);

    Merge(arr, aux, left, mid, right);
}

private static void Merge(double[] arr, double[] aux, int left, int mid, int right)
{
    int i = left;
    int j = mid + 1;
    int k = left;

    for (int idx = left; idx <= right; idx++)
        aux[idx] = arr[idx];

    while (i <= mid && j <= right)
    {
        if (aux[i] <= aux[j]) arr[k++] = aux[i++];
        else arr[k++] = aux[j++];
    }

    while (i <= mid) arr[k++] = aux[i++];
    while (j <= right) arr[k++] = aux[j++];
}
}
}

```

Додаток Д. Код на Python для сортування великого масиву

```
#!/usr/bin/env python3
import argparse
import asyncio
import csv
import datetime as dt
import heapq
import os
import time
from multiprocessing import Pool, cpu_count
from functools import partial
from concurrent.futures import ProcessPoolExecutor, as_completed

import numpy as np

CSV_PATH = "sorting_results_py.csv"

# ----- CSV helpers -----

def ensure_csv_header():
    if not os.path.exists(CSV_PATH):
        with open(CSV_PATH, "w", newline="", encoding="utf-8") as f:
            writer = csv.writer(f)
            writer.writerow(["Timestamp", "Model", "N", "Iterations", "TotalMs", "PerIterationMs"])

def append_csv_row(model_name: str, n: int, iterations: int, total_ms: float, per_iter_ms: float):
    with open(CSV_PATH, "a", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow([
            dt.datetime.utcnow().isoformat(),
            model_name,
            n,
            iterations,
            f"{total_ms:.4f}",
            f"{per_iter_ms:.4f}",
        ])

def benchmark_model(model_name: str, n: int, iterations: int, func):
    """
    func(arr: np.ndarray) -> np.ndarray
    """
    print(f"\n=== {model_name} ===")

    # Невелика "розминка" для прогріву
    warmup = np.random.rand(100_000)
```

```

func(warmup)

start = time.perf_counter()
for _ in range(iterations):
    data = np.random.rand(n)
    result = func(data)
    # щоб інтерпретатор не викинув результат (мінімальне використання)
    _ = result[0]
total = time.perf_counter() - start

total_ms = total * 1000.0
per_iter_ms = total_ms / iterations
print(f"{model_name}: total = {total_ms:.2f} ms, per iteration = {per_iter_ms:.2f} ms")

append_csv_row(model_name, n, iterations, total_ms, per_iter_ms)

# ----- Генерація -----

def generate_random_array(n: int) -> np.ndarray:
    return np.random.rand(n)

# ----- 1. Послідовні сортування -----

def sort_sequential_mergesort(arr: np.ndarray) -> np.ndarray:
    """
    Послідовне сортування: numpy mergesort (стабільний).
    """
    return np.sort(arr.copy(), kind="mergesort")

def sort_sequential_quicksort(arr: np.ndarray) -> np.ndarray:
    """
    Послідовне сортування: numpy quicksort (швидкий, нестабільний).
    """
    return np.sort(arr.copy(), kind="quicksort")

# ----- 2. Multiprocessing + numpy -----

def _worker_sort_chunk(chunk: np.ndarray, kind: str = "quicksort") -> np.ndarray:
    """
    Функція-воркер: сортує один шматок масиву.
    """
    return np.sort(chunk, kind=kind)

def sort_multiprocessing_chunks(arr: np.ndarray, num_procs: int | None = None) -> np.ndarray:

```

```

"""
Паралельне сортування: ділимо масив на шматки, кожен процес сортує свій шматок,
потім робимо k-way merge (послідовна фаза).
"""

if num_procs is None:
    num_procs = cpu_count()

chunks = np.array_split(arr, num_procs)

with Pool(processes=num_procs) as pool:
    # паралельна фаза
    sorted_chunks = pool.map(_worker_sort_chunk, chunks)

# послідовна фаза: k-way merge
iterables = [chunk.tolist() for chunk in sorted_chunks]
merged_iter = heapq.merge(*iterables)
merged_list = list(merged_iter)
return np.array(merged_list, dtype=arr.dtype)

# ----- 3. concurrent.futures + numpy -----

def _worker_sort_chunk_cf(chunk: np.ndarray, kind: str = "quicksort") -> np.ndarray:
    return np.sort(chunk, kind=kind)

def sort_concurrent_futures_chunks(arr: np.ndarray, num_procs: int | None = None) -> np.ndarray:
    """
    Аналог попереднього, але через concurrent.futures.ProcessPoolExecutor.
    """
    if num_procs is None:
        num_procs = cpu_count()

    chunks = np.array_split(arr, num_procs)
    sorted_chunks = []

    with ProcessPoolExecutor(max_workers=num_procs) as executor:
        futures = [executor.submit(_worker_sort_chunk_cf, chunk) for chunk in chunks]
        for fut in as_completed(futures):
            sorted_chunks.append(fut.result())

    # Порядок у sorted_chunks може бути довільним через as_completed,
    # тому відсортуємо список за першим елементом кожного шматка,
    # щоб відновити приблизний порядок для коректного merge.
    sorted_chunks.sort(key=lambda ch: ch[0] if len(ch) > 0 else float("-inf"))

    iterables = [chunk.tolist() for chunk in sorted_chunks]
    merged_iter = heapq.merge(*iterables)
    merged_list = list(merged_iter)

```

```

return np.array(merged_list, dtype=arr.dtype)

# ----- 4. asyncіo (імітація конкурентності) -----

def _sort_chunk_seq(chunk: np.ndarray, kind: str = "quicksort") -> np.ndarray:
    """
    Звичайне послідовне сортування шматка (CPU-bound),
    яке будемо запускати у thread pool через asyncio.run_in_executor.
    """
    return np.sort(chunk, kind=kind)

async def _async_sort_internal(arr: np.ndarray, num_workers: int | None = None) -> np.ndarray:
    if num_workers is None:
        # для імітації конкурентності достатньо кількох тредів
        num_workers = min(cpu_count(), 8)

    loop = asyncio.get_running_loop()

    chunks = np.array_split(arr, num_workers)

    tasks = []
    for chunk in chunks:
        # передаємо копію chunk, щоб уникнути потенційних side-ефектів
        c = chunk.copy()
        task = loop.run_in_executor(
            None,      # використати дефолтний ThreadPoolExecutor
            partial(_sort_chunk_seq, c)
        )
        tasks.append(task)

    sorted_chunks = await asyncio.gather(*tasks)

    # k-way merge
    iterables = [ch.tolist() for ch in sorted_chunks]
    merged_iter = heapq.merge(*iterables)
    merged_list = list(merged_iter)
    return np.array(merged_list, dtype=arr.dtype)

def sort_asyncio_chunks(arr: np.ndarray) -> np.ndarray:
    """
    Обгортка над asyncio.run, щоб мати такий самий інтерфейс, як в інших функцій.
    """
    return asyncio.run(_async_sort_internal(arr))

# ----- main -----

```



```

def main():
    parser = argparse.ArgumentParser(description="Large array sorting benchmarks (Python)")
    parser.add_argument("--n", type=int, default=1_000_000,
                        help="Array size (e.g., 1000000, 5000000, 10000000)")
    parser.add_argument("--iterations", type=int, default=3,
                        help="Iterations per model")
    args = parser.parse_args()

    n = args.n
    iterations = args.iterations

    print(f"Array size: {n}, iterations per model: {iterations}")

    ensure_csv_header()

    # 1. Послідовні алгоритми
    benchmark_model("Sequential_MergeSort", n, iterations, sort_sequential_mergesort)
    benchmark_model("Sequential_QuickSort", n, iterations, sort_sequential_quicksort)

    # 2. Multiprocessing + numpy
    benchmark_model("Multiprocessing_Chunks", n, iterations, sort_multiprocessing_chunks)

    # 3. concurrent.futures + numpy
    benchmark_model("ConcurrentFutures_Chunks", n, iterations, sort_concurrent_futures_chunks)

    # 4. asyncio (імітація конкурентності у thread pool)
    benchmark_model("Asyncio_Chunks", n, iterations, sort_asyncio_chunks)

    print(f"\nDone. Results appended to {CSV_PATH}")

if __name__ == "__main__":
    main()

```