Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

ДИПЛОМНА РОБОТА

на тему: “Системний аналіз продуктивності мов програмування на різних апаратно-програмних платформах”

Виконала:
студентка групи КН-41
Галагуз Тетяна Олександрівна

Науковий керівник:
Старший викладач
Ляшук Тарас Григорович

.

Рівне-2026
м. Рівне

ЗМІСТ

СПИСОК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

CPU	Central Processing Unit
I/O	Input/Output
SIMD	Single Instruction, Multiple Data
JVM	Java Virtual Machine
IL	Intermediate Language
CLR	Common Language Runtime
JIT	Just-In-Time
GIL	Global Interpreter Lock
RAM	Random Access Memory
TPL	Task Parallel Library
GC	Garbage Collector

ВСТУП

Вибір мови програмування є одним з визначальних рішень при розробці програмного забезпечення, оскільки він безпосередньо впливає на характеристики виконання програм – час роботи, використання оперативної пам'яті та навантаження на процесор. Різні мови реалізують принципово відмінні моделі виконання: від прямої компіляції до машинного коду (C++), до інтерпретації з динамічним управлінням типами (Python), із проміжними підходами на базі байт-коду з JIT-компіляцією (Java, C#). Зазначені відмінності призводять до того, що одна і та сама задача, реалізована різними мовами, може демонструвати суттєво різні результати вимірювань продуктивності навіть за ідентичних умов виконання [1].

Проблема порівняльного аналізу продуктивності мов програмування є предметом досліджень упродовж тривалого часу. Однак більшість існуючих робіт зосереджена або на вузькоспеціалізованих навантаженнях, або на окремих аспектах – наприклад, енергоспоживанні чи пропускній здатності – без урахування впливу апаратно-програмної платформи як окремого чинника. Зокрема, Pereira et al. у дослідженні, опублікованому в журналі Science of Computer Programming [1], проаналізували 27 мов програмування за показниками часу виконання, використання пам'яті та споживання енергії. Результати показали, що мови з різними моделями виконання утворюють стійкі групи за продуктивністю, проте самі по собі ці виміри не відповідають на питання про причинно-наслідковий зв'язок між середовищем виконання та отриманими характеристиками. Подальший розвиток цього питання знайшов відображення у роботі van Kempen et al. [2], де було продемонстровано, що результати порівняльних вимірювань суттєво залежать від методологічних рішень – зокрема від кількості активних ядер процесора, режимів прогріву JIT-компілятора та активності підсистеми пам'яті. Без контролю цих параметрів результати можуть бути систематично зміщені на користь тієї чи іншої мови, незалежно від її реальних характеристик виконання.

Окремим аспектом є залежність результатів бенчмаркінгу від операційної системи та апаратного забезпечення. Відомо, що середовище виконання CLR (.NET/C#) демонструє відмінні показники на платформі Windows порівняно з Linux через ступінь інтеграції з системними викликами операційної системи. JVM (Java), натомість, проявляє відносно стабільне міжплатформне виконання, хоча і з відмінностями, зумовленими різними реалізаціями збирача сміття та порогами JIT-компіляції [3]. Окрім того, дослідження, присвячені компіляції Python-коду [4], засвідчують, що показники виконання інтерпретованих мов змінюються нелінійно залежно від типу навантаження (CPU-bound, memory-bound, I/O-bound) та конфігурації апаратної платформи. Ці факти вказують на те, що коректний системний аналіз продуктивності потребує контрольованого середовища та відтворюваної методики, а не лише набору ізольованих вимірювань.

Таким чином, проблема полягає у відсутності комплексної, методологічно обґрунтованої порівняльної характеристики продуктивності мов C++, Java, C# та Python у розрізі різних типів навантажень і апаратно-програмних платформ. Саме це зумовлює актуальність та практичне значення даної роботи.

Актуальність дослідження. Вибір мови програмування в умовах обмежених обчислювальних ресурсів або специфічних вимог до часу відгуку безпосередньо впливає на проектні рішення при розробці програмного забезпечення. Наявні порівняльні дослідження не забезпечують відтворюваної методики оцінювання продуктивності у розрізі апаратно-програмних платформ, що унеможливлює застосування їх результатів у практиці проектування. Дана робота спрямована на усунення цього недоліку шляхом розробки та застосування системної методики бенчмаркінгу [1, 2].

Мета дослідження полягає у вимірюванні та порівнянні показників продуктивності програм, реалізованих мовами C++, Java, C# та Python, при виконанні алгоритмів різноманітних типів навантажень на різних апаратно-програмних платформах. Дослідження спрямоване на аналіз залежності результатів вимірювань від типу мови програмування, середовища виконання та

апаратної конфігурації, задля формулювання обґрунтованих висновків щодо умов застосовності кожної з мов.

Завдання дослідження:

1. Дослідити теоретичні основи продуктивності програмного забезпечення та особливості виконання програм у компільованих, JIT-орієнтованих та інтерпретованих мовах.
2. Проаналізувати наявні інструменти та методики вимірювання продуктивності програм.
3. Спроєктувати систему бенчмаркінгу з визначенням тестових алгоритмів, що відповідають CPU-bound, memory-bound, I/O-bound та системним навантаженням.
4. Розробити та імплементувати набір тестових програм мовами C++, Java, C# та Python з ідентичними алгоритмами та вхідними даними.
5. Здійснити експеримент з вимірювання продуктивності на визначених апаратно-програмних платформах із дотриманням методики усереднення та відтворюваності результатів.
6. Систематизувати та проаналізувати отримані результати з метою виявлення закономірностей залежності продуктивності від мови, типу навантаження та платформи.

Об'єктом дослідження є процес виконання програм, реалізованих мовами C++, Java, C# та Python, на різних апаратно-програмних платформах.

Предметом дослідження є показники продуктивності програм – час виконання, використання процесора та оперативної пам'яті, пропускна здатність – у залежності від мови програмування, типу навантаження та апаратно-програмної платформи.

Методи дослідження. У роботі використовуються теоретичні методи – аналіз науково-технічної літератури та систематизація характеристик середовищ виконання. Емпіричні методи охоплюють проведення контрольованого обчислювального експерименту, вимірювання показників продуктивності з

багаторазовим повторенням та статистичне усереднення результатів, а також узагальнення та порівняльний аналіз отриманих даних.

Практичне значення дослідження полягає у формуванні доказової бази для обґрунтованого вибору мови програмування, залежно від типу задачі та апаратно-програмної платформи. Отримані результати можуть бути використані при проєктуванні програмних систем з вимогами до продуктивності, а також у навчальному процесі при вивченні архітектур мов програмування та методів бенчмаркінгу.

Структура роботи. Робота складається зі вступу, …

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ АНАЛІЗУ ПРОДУКТИВНОСТІ МОВ ПРОГРАМУВАННЯ

1.1. Продуктивність програмного забезпечення та її основні характеристики

Поняття продуктивності програмного забезпечення не зводиться до єдиної числової характеристики. У загальному розумінні продуктивність визначається як обсяг корисної роботи, виконаної обчислювальною системою за одиницю часу при заданих витратах ресурсів [5]. Водночас у контексті порівняльного аналізу мов програмування важливо розмежовувати два суміжні поняття: швидкодію (speed) та ефективність використання ресурсів (resource efficiency). Швидкодія характеризує абсолютний час виконання конкретного обчислювального завдання, тоді як ефективність відображає відношення виконаної роботи до витрачених ресурсів – процесорного часу, обсягу пам'яті та енергії. Ці два аспекти не завжди корелюють між собою: програма може виконуватись швидко, водночас надмірно навантажуючи процесор або займаючи значний обсяг оперативної пам'яті, що у певних системних контекстах є небажаним [1]. Отже, повноцінна оцінка продуктивності передбачає одночасний аналіз кількох вимірювань, а не орієнтацію лише на час виконання.

Характеристики продуктивності програми суттєво залежать від того, який ресурс обчислювальної системи є вузьким місцем при її виконанні. За цим критерієм навантаження прийнято класифікувати на кілька типів, що схематично зображено на рисунку 1.1.

Рисунок 1.1 – Класифікація типів обчислювальних навантажень за ресурсом-обмежувачем

Завдання типу CPU-bound (обчислювально-обмежені) характеризуються тим, що час їх виконання визначається насамперед швидкістю процесора. Процесор при цьому завантажений близько до 100 %, а продуктивність зростає пропорційно до підвищення тактової частоти або збільшення кількості задіяних ядер [6]. Типовими прикладами є сортування масивів великого розміру, матричні обчислення, криптографічні операції та чисельне інтегрування. Для задач типу I/O-bound (введення-виведення-обмежені) визначальним чинником є швидкість підсистеми введення-виведення – дискової або мережевої. У цьому випадку процесор більшу частину часу перебуває в стані очікування завершення операції читання або запису, а підвищення його тактової частоти практично не впливає на загальний час виконання [6]. Завдання типу memory-bound (пам'яттю-обмежені) обмежені пропускною здатністю підсистеми пам'яті: продуктивність визначається швидкістю передавання даних між оперативною пам'яттю та процесорними кешами, а не обчислювальною потужністю ядер [5]. Так,

програми, що виконують випадковий доступ до великих масивів даних, зазвичай є memory-bound навіть на процесорах з високою тактовою частотою. Окремо виділяють системні задачі – ті, що пов'язані з безпосередньою взаємодією з операційною системою: створення процесів і потоків, робота з дескрипторами файлів, виклики системних функцій. Їх продуктивність залежить від реалізації планувальника, накладних витрат на перемикання контексту та особливостей конкретної операційної системи, що робить цей клас навантажень чутливим до платформи виконання.

Зазначена класифікація має пряме практичне значення при проєктуванні системи бенчмаркінгу: тестові алгоритми повинні охоплювати всі чотири типи навантажень, оскільки мови програмування можуть демонструвати якісно різну поведінку залежно від того, який ресурс є обмежуючим. Програма, що виконується швидше за конкурента у CPU-bound задачах, може поступатися йому при роботі з інтенсивним введенням-виведенням.

Для кількісного опису продуктивності в дослідженнях використовується набір стандартизованих метрик, кожна з яких відображає окремий аспект поведінки програми під час виконання. Першою і найпоширенішою метрикою є час виконання, який, однак, розподіляється на два принципово різних виміри. Wall time (астрономічний час, або час «настінного годинника») – це повний проміжок часу від запуску програми до її завершення, що вимірюється незалежним годинником і включає усі затримки: очікування введення-виведення, блокування, перемикання контексту та роботу інших процесів у системі [7]. CPU time (процесорний час) – це час, протягом якого процесор фактично виконував інструкції даного процесу, не включаючи часу очікування. Співвідношення між цими двома метриками є діагностичним показником типу навантаження: якщо CPU time значно менший за wall time, це свідчить про те, що програма витрачає суттєву частину часу на очікування зовнішніх ресурсів, тобто є I/O-bound [7]. У контексті порівняльного бенчмаркінгу мов програмування wall time є більш репрезентативним показником з точки зору

кінцевого користувача, проте він чутливий до системного шуму і потребує статистичного усереднення за багатьма запусками [2].

Завантаженість процесора (CPU load) характеризує частку часу, протягом якої процесор перебував у активному стані виконання інструкцій. Ця метрика дозволяє з'ясувати, чи є процесор вузьким місцем при виконанні програми, і є необхідною для інтерпретації результатів часових вимірювань. Використання пам'яті оцінюється через розмір купи (heap) – динамічно виділеної пам'яті в адресному просторі процесу – та через загальний обсяг оперативної пам'яті (RAM), що займає процес у системі. Мови з керованою пам'яттю (Java, C#) мають додаткові накладні витрати, пов'язані з роботою збирача сміття, що може проявлятись як нерівномірне зростання heap під час виконання [1].

Пропускна здатність (throughput) визначається як кількість операцій або обсяг даних, що оброблюється за одиницю часу. Ця метрика є особливо релевантною для навантажень з потоковою обробкою даних та для оцінювання ефективності паралельного виконання. Затримка (latency) характеризує час від початку виконання однієї операції до її завершення і є показником, що визначає реактивність системи для кожного окремого запиту. Взаємозв'язок між throughput і latency є нелінійним: зростання throughput, як правило, супроводжується збільшенням latency через зростання черг запитів і конкуренцію за ресурси, що наочно відображається на відповідному графіку залежності цих двох метрик (рисунок 1.2).

У даній роботі основними вимірюваними метриками є wall time як інтегральна характеристика часу виконання, CPU time для оцінювання реального обчислювального навантаження на процесор, а також використання оперативної пам'яті. Throughput розглядається у задачах I/O-bound характеру, де він є більш інформативним показником порівняно з абсолютним часом виконання.

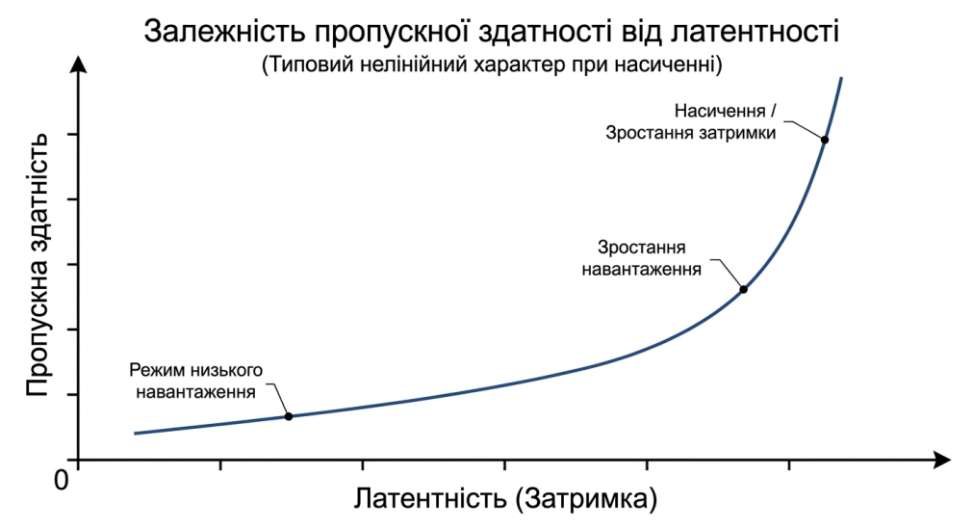

Рисунок 1.2 – Типова залежність між затримкою та пропускною здатністю системи

Сукупне застосування зазначених метрик забезпечує багатовимірну картину продуктивності, необхідну для коректного порівняльного аналізу мов програмування на різних апаратно-програмних платформах.

1.2. Особливості виконання програм у різних мовах

Принципові відмінності у характеристиках продуктивності мов програмування зумовлені насамперед моделлю виконання, яку реалізує кожна з них. Під моделлю виконання розуміється спосіб перетворення вихідного коду на інструкції, що безпосередньо виконуються процесором: цей процес може відбуватись повністю до запуску програми, поступово під час її роботи або покроково при виконанні кожного рядка коду. Від обраної моделі залежать як абсолютні показники швидкодії, так і поведінка програми під час старту, прогріву та тривалого виконання [8].

У мовах з компіляцією до машинного коду, до яких належить C++, перетворення вихідного тексту програми на машинні інструкції відбувається повністю до запуску – у фазі компіляції. Компілятор (наприклад, GCC або MSVC) аналізує увесь код та застосовує широкий спектр оптимізацій: інлайнінг функцій, розгортання циклів, векторизацію з використанням SIMD-інструкцій,

усунення мертвого коду та оптимізацію звернень до пам'яті [5]. Результатом є виконуваний файл у машинному коді цільової архітектури, що виконується безпосередньо процесором без будь-яких проміжних рівнів інтерпретації. Це забезпечує мінімальні накладні витрати під час виконання та детерміновану поведінку у часі. Водночас статична компіляція позбавляє програму можливості адаптуватись до поведінки під час виконання: оптимізації приймаються на підставі аналізу коду, а не реальних профілів навантаження [8].

Якісно іншу модель реалізують мови з JIT-компіляцією, зокрема Java та C#. Вихідний код цих мов спочатку компілюється у проміжне байт-кодове представлення – байт-код JVM для Java та Intermediate Language (IL) для C# – яке не є машинним кодом конкретного процесора. Виконання програми починається з інтерпретації байт-коду, а середовище виконання (JVM або CLR) паралельно збирає статистику звернень до методів і ділянок коду [9]. Коли певний метод або цикл перевищує встановлений поріг частоти викликів, він передається JIT-компілятору, який перетворює байт-код на оптимізований машинний код з урахуванням реального профілю виконання. У HotSpot JVM цей процес організований у кілька рівнів: рівень C1 забезпечує швидку початкову компіляцію з базовими оптимізаціями, а рівень C2 – глибоку оптимізацію гарячих методів після накопичення достатнього обсягу профільних даних [9]. Аналогічну багаторівневу схему реалізує CLR (.NET/C#).

Практичним наслідком такої архітектури є наявність фази прогріву (warmup): протягом початкового часу виконання JIT-програми працюють повільніше за статично скомпільовані аналоги, оскільки частина ресурсів процесора витрачається на компіляцію, а не на виконання корисної роботи. Дослідження Traini et al. [3] показали, що лише 74,7 % запусків Java-програм досягають стабільного стану продуктивності, а в 43,5 % пар (JVM, бенчмарк) стабільний стан не досягається стійко від запуску до запуску. Це робить коректний прогрів середовища обов'язковою умовою при бенчмаркінгу JIT-мов. Після завершення прогріву JIT-компілятор здатний досягати показників, близьких до статично скомпільованого коду, а в окремих сценаріях – навіть

перевищувати їх завдяки адаптивним оптимізаціям на основі реального профілю [9].

Принципово відмінну поведінку демонструє Python у своїй стандартній реалізації CPython. Вихідний код транслюється у байт-код, який виконується інтерпретатором покроково без компіляції у машинний код. Кожна інструкція байт-коду вимагає відповідного виклику інтерпретатора, що вносить постійні накладні витрати на кожну операцію [4]. Суттєвим чинником є також динамічна система типізації Python: оскільки тип кожного об'єкта визначається під час виконання, інтерпретатор змушений щоразу перевіряти тип операнда перед виконанням операції, що унеможливлює низку оптимізацій, доступних компільованим мовам [4]. Окремим структурним обмеженням є Global Interpreter Lock (GIL) – глобальний м'ютекс, який дозволяє виконувати байт-код лише одному потоку одночасно незалежно від кількості доступних ядер процесора [10]. Це робить багатопотокове виконання CPU-bound задач у CPython практично неефективним. Починаючи з Python 3.13, GIL є необов'язковим компонентом, проте цей режим залишається експериментальним і широко не застосовується [10].

Узагальнення описаних відмінностей наведено у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика моделей виконання програм

Характеристика	C++	Java / C#	Python (CPython)
Модель виконання	Компіляція до машинного коду	Байт-код + JIT-компіляція	Інтерпретація байт-коду
Фаза компіляції	До запуску (AOT)	До запуску (байт-код) + під час виконання (JIT)	До запуску (байт-код)
Фаза прогріву	Відсутня	Присутня (C1→C2 у JVM)	Відсутня (немає JIT)
Управління пам'яттю	Ручне / RAII	Автоматичне (GC)	Автоматичне (GC)
Багатопотоковість	Повноцінна	Повноцінна	Обмежена (GIL)
Залежність від платформи	Бінарний файл під конкретну архітектуру	Переносимий байт-код (JVM/CLR)	Переносимий байт-код
Типова відносна продуктивність (CPU-bound)	Найвища	Середня–висока (після прогріву)	Найнижча

Наведені відмінності мають безпосередній вплив на методику проведення порівняльних вимірювань продуктивності. Для коректного зіставлення результатів необхідно враховувати фазу прогріву JIT-середовищ, виключати час першого запуску із вимірювань для Java та C#, а також усереднювати результати за достатньою кількістю повторень для зниження впливу недетермінізму JIT-компілятора [2, 3].

1.3. Вплив апаратно-програмної платформи

Результати вимірювань продуктивності програм залежать не лише від мови програмування та алгоритму, а й від апаратно-програмної платформи, на

якій здійснюється виконання. Під апаратно-програмною платформою розуміється сукупність характеристик обчислювальної системи, що включає архітектуру процесора, конфігурацію підсистеми пам'яті та операційну систему з її планувальником задач та реалізацією системних викликів. Зміна будь-якого з цих компонентів може призвести до відчутного відхилення у значеннях вимірюваних метрик навіть при незмінному програмному коді та вхідних даних [11].

Процесор є первинним ресурсом, що визначає швидкодію CPU-bound задач. Однак із зростанням тактової частоти процесорів різниця між швидкістю виконання обчислень і швидкістю доступу до оперативної пам'яті стала принциповою: доступ до RAM займає від 50 до 100 наносекунд, тоді як виконання однієї інструкції займає частки наносекунди [11]. Для усунення цього дисбалансу застосовується ієрархія процесорних кешів. Кеш рівня L1 є найшвидшим – латентність доступу становить менше 1 нс при ємності від 32 до 128 КБ, і він є індивідуальним для кожного ядра. Кеш L2 є більшим (256 КБ – 2 МБ) і повільнішим (3–5 нс), також здебільшого виділений кожному ядру окремо. Кеш L3 є спільним для всіх ядер процесора, його об'єм може становити від 4 до 64 МБ і більше, а латентність – від 10 до 20 нс [11]. Така ієрархія наведена на рисунку 1.3.

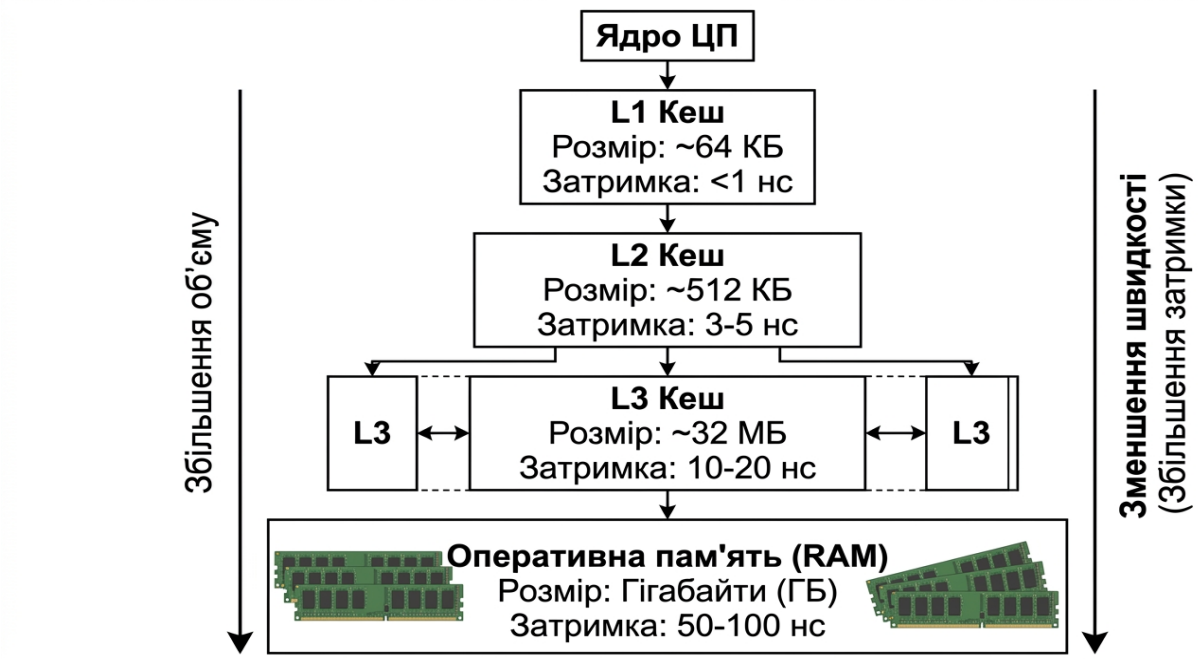

Рисунок 1.3 – Ієрархія пам'яті процесора: кеші L1/L2/L3 та оперативна пам'ять

Програми, що демонструють хорошу просторову та часову локальність даних, ефективно використовують кеш-пам'ять і виконуються значно швидше, ніж алгоритми з випадковим доступом до великих масивів. Дослідження показують, що для ідентичної кількості процесорних інструкцій програма з випадковим доступом до пам'яті може генерувати на два порядки більше промахів кешу порівняно з програмою, що послідовно проходить масив [11]. Цей ефект є суттєвим для мов з управлінням пам'яттю через збирач сміття (Java, C#, Python): алокатори об'єктів у купі можуть розміщувати пов'язані об'єкти некомпактно, що погіршує використання кешу порівняно з ручним керуванням пам'яттю у C++.

Кількість фізичних ядер процесора визначає ступінь можливого паралелізму. Для CPU-bound задач, що допускають багатопотокове виконання, збільшення кількості ядер пропорційно зменшує wall time, проте для однопотокових задач визначальною залишається тактова частота. Оскільки Python (CPython) через GIL обмежений одним потоком для байт-коду, він

практично не отримує переваги від багатоядерних процесорів у CPU-bound задачах, на відміну від C++, Java та C# [10].

Оперативна пам'ять є вузьким місцем для задач класу memory-bound. При промаху кешу останнього рівня (L3) процесор змушений очікувати даних з DRAM, що призводить до простою конвеєра. Пропускна здатність шини пам'яті та латентність доступу до DRAM визначають верхню межу продуктивності таких задач незалежно від швидкості процесора. Для програм на Java та Python з динамічним виділенням об'єктів у купі тиск на підсистему пам'яті додатково зростає через роботу збирача сміття, що періодично виконує обхід купи і може призводити до пауз у виконанні програми [1].

Операційна система є третім суттєвим чинником платформи. Її планувальник задач, реалізація системних викликів та накладні витрати на перемикання контексту безпосередньо впливають на продуктивність у I/O-bound та системних задачах. Порівняльні вимірювання засвідчують, що планувальник Linux (Completely Fair Scheduler, CFS) у більшості сценаріїв забезпечує нижчі затримки для інтерактивних і паралельних задач, ніж планувальник Windows NT, особливо на системах з великою кількістю ядер [12]. Окрім того, компілятори та бібліотеки часто генерують різний код залежно від цільової операційної системи: MSVC на Windows та GCC/Clang на Linux можуть застосовувати різні набори оптимізацій навіть для ідентичного вихідного коду, що вносить додаткову змінну у результати міжплатформних порівнянь [12]. Для Java та C# вплив ОС є менш прямим – JVM і CLR абстрагують більшість платформозалежних аспектів, – однак реалізація потоків та I/O залишається залежною від примітивів операційної системи.

Зведений вплив розглянутих факторів платформи на показники продуктивності наведено у таблиці 1.2.

Таблиця 1.2 – Вплив компонентів апаратно-програмної платформи на показники продуктивності

Компонент платформи	Впливає на	Тип задач	Характер впливу
Тактова частота CPU	Час виконання інструкцій	CPU-bound	Прямо пропорційний
Кількість ядер CPU	Wall time (паралельні задачі)	CPU-bound (багатопотокові)	Пропорційне зменшення (для масштабованих алгоритмів)
Кеш L1/L2 (на ядро)	Затримка доступу до «гарячих» даних	CPU-bound, Memory-bound	Впливає на локальність доступу до даних
Кеш L3 (спільний)	Частота промахів при міжпотоковому доступі	Memory-bound, паралельні	Зменшує кількість звернень до оперативної пам’яті
Об'єм та пропускна здатність RAM	Максимальна продуктивність memory-bound	Memory-bound	Обмежує пропускну здатність
Операційна система	Планування потоків, системні виклики	I/O-bound, системні	Формує накладні витрати виконання
Компілятор/версія ОС	Якість машинного коду	CPU-bound	Визначає рівень оптимізацій

Таким чином, повноцінне порівняння продуктивності мов програмування вимагає чіткої фіксації всіх компонентів платформи, оскільки зміна будь-якого з них здатна суттєво вплинути на результати і зробити порівняння некоректним. Саме тому методика дослідження, описана у розділі 3, передбачає детальну документацію апаратної конфігурації та версій програмного забезпечення для кожного середовища виконання.

РОЗДІЛ 2. АНАЛІЗ ІНСТРУМЕНТІВ ТА СЕРЕДОВИЩ ДОСЛІДЖЕННЯ

2.1. Особливості мов програмування

У даному дослідженні для порівняльного аналізу продуктивності обрано чотири мови: C++, Java, C# та Python. Вибір зумовлений тим, що ці мови є популярними та представляють принципово різні підходи до управління пам'яттю, системи типів та організації паралелізму, що, у свою чергу, формує їх характерні профілі продуктивності в задачах різних класів. Узагальнену порівняльну характеристику цих мов наведено у таблиці 2.1, а детальний розгляд особливостей кожної з них подано нижче.

C++ є мовою зі статичною системою типів та ручним управлінням пам'яттю, що реалізується через ідіому RAII (Resource Acquisition Is Initialization). Суть RAII полягає у прив'язці часу існування ресурсу до часу існування об'єкта: ресурс захоплюється у конструкторі та звільняється у деструкторі, який викликається детерміновано у момент виходу об'єкта за межі видимості [13]. Це виключає необхідність у збирачі сміття та пов'язаних з ним пауз виконання, забезпечуючи передбачуваний часовий профіль. C++ реалізує принцип нульових витрат абстракцій (zero-overhead abstractions): конструкції мови – шаблони, вбудовані функції, перевантаження операторів – компілюються без накладних витрат під час виконання, якщо вони не використовуються програмою [8]. Мова надає пряме управління розміщенням даних у пам'яті, що дозволяє програмісту формувати структури, оптимальні для кешу процесора. Повноцінна підтримка багатопотоковості без системних обмежень забезпечує масштабованість на багатоядерних системах. Стандарти C++17 та C++20 суттєво розширили стандартну бібліотеку та можливості паралельного програмування.

Java є об'єктно-орієнтованою мовою зі статичною типізацією та автоматичним управлінням пам'яттю через збирач сміття. Ключовою архітектурною особливістю є принцип «Write Once, Run Anywhere»: вихідний код компілюється у байт-код JVM, який є платформонейтральним і виконується

на будь-якій системі, де встановлено сумісну JVM [14]. Усі об'єкти Java розміщуються у купі і управляються збирачем сміття з поколіннями (generational GC): молоде покоління (Young Generation) збирається часто і швидко, старе (Old Generation) – рідше, але з більшими паузами. Для операцій з примітивними типами Java використовує значення на стеку, що знижує тиск на збирач сміття, однак усі об'єкти є посилальними типами. Повноцінна підтримка багатопотоковості через стандартну бібліотеку Java Concurrency дозволяє ефективно задіювати всі ядра процесора, що принципово відрізняє Java від Python у паралельних сценаріях.

C# є мовою платформи .NET зі статичною типізацією та автоматичним управлінням пам'яттю через CLR. Суттєвою відмінністю від Java є наявність справжніх типів-значень (value types): структури (struct) у C# розміщуються на стеку або інлайново в інших об'єктах без виділення пам'яті у купі, що знижує навантаження на збирач сміття у задачах з великою кількістю дрібних об'єктів [13]. Мова підтримує як посилальні типи (class), так і типи-значення (struct), що дає програмісту явний контроль над розміщенням даних. CLR забезпечує крос-платформне виконання починаючи з .NET Core (2016) та об'єднаної платформи .NET 5+, що дозволяє запускати програми на C# на Windows, Linux та macOS. Підтримка async/await та бібліотека TPL (Task Parallel Library) забезпечують розвинені засоби паралельного та асинхронного програмування.

Python у стандартній реалізації CPython є динамічно типізованою мовою з автоматичним управлінням пам'яттю через підрахунок посилань (reference counting) з доповненням у вигляді збирача циклів. На відміну від трьох попередніх мов, Python не має статичної типізації: типи змінних визначаються під час виконання, що спрощує розробку, але унеможливлює низку оптимізацій на рівні компілятора та збільшує накладні витрати на кожну операцію [4]. Стандартна бібліотека Python охоплює широкий спектр функцій, а екосистема сторонніх бібліотек (NumPy, Pandas, SciPy) реалізована переважно у C/C++, що дозволяє досягати прийнятної продуктивності для обчислювальних задач за рахунок делегування критичних операцій до скомпільованого коду. Проте у

задачах, де обчислення відбуваються безпосередньо в Python-коді (тобто без звернення до нативних бібліотек), продуктивність CPython є суттєво нижчою порівняно з іншими трьома мовами [1].

Таблиця 2.1 – Порівняльна характеристика мов програмування як інструментів дослідження

Характеристика	C++	Java	C#	Python (CPython)
Система типів	Статична	Статична	Статична	Динамічна
Управління пам'яттю	RAII / ручне	GC (generational)	GC (generational)	Підрахунок посилань + GC
Паузи GC	Відсутні	Присутні (minor/major)	Присутні (generational)	Короткі (при циклах)
Типи-значення на стеку	Так (усі примітиви та struct)	Часткові (тільки примітиви)	Так (struct)	Ні (всі об'єкти у купі)
Повноцінна багатопотоковість	Так	Так	Так	Обмежена (GIL)
Крос-платформність	Потребує перекомпіляції	Так (JVM)	Так (.NET Core/.NET 5+)	Так (інтерпретатор)
Типова область застосування	Системне ПЗ, HPC, ігри	Корпоративні системи, Android	Windows-застосунки, ігри (Unity)	Data Science, скриптинг

Наведені характеристики безпосередньо впливають на те, як кожна з мов поводиться у задачах різних типів навантажень. Зокрема, відсутність GC у C++ забезпечує детерміноване виконання без пауз, але перекладає відповідальність за управління пам'яттю на програміста. Наявність типів-значень у C# дає перевагу перед Java у задачах зі значним числом дрібних об'єктів, оскільки знижує тиск на збирач сміття. Python, незважаючи на свою гнучкість, несе постійні накладні

витрати, пов'язані з динамічною типізацією та інтерпретацією байт-коду, що визначає його характерне місце у результатах порівняльного аналізу. Середовища виконання кожної з цих мов розглядаються у підрозділі 2.2.

2.2. Середовища виконання програм

Середовище виконання програми визначає конкретний шлях перетворення вихідного коду на послідовність дій процесора. У контексті порівняльного бенчмаркінгу це поняття охоплює не лише сам механізм виконання, а й компілятор або інтерпретатор з його налаштуваннями, версію середовища, конфігурацію управління пам'яттю та умови прогріву. Схематично шлях від вихідного коду до виконання для чотирьох досліджуваних мов зображено на рисунку 2.1.

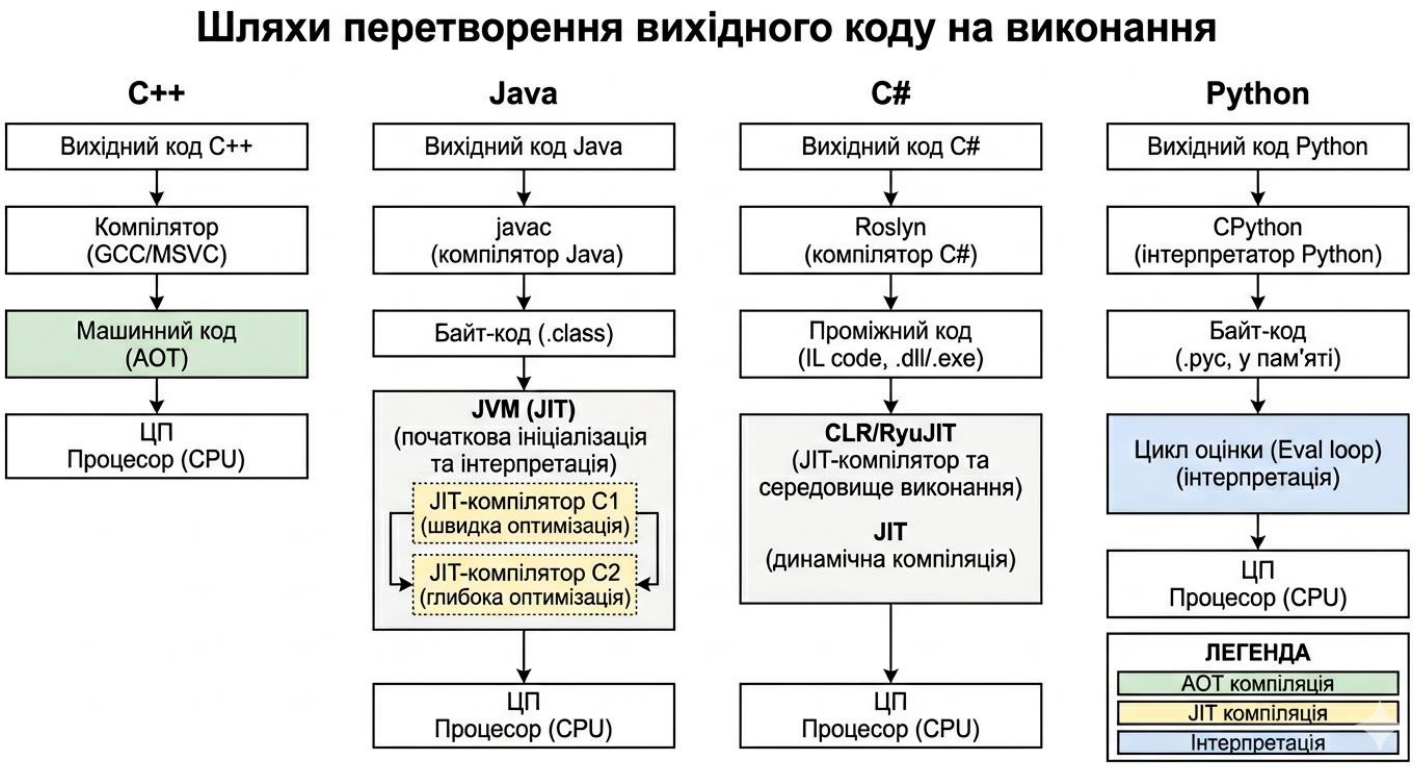

Рисунок 2.1 – Шляхи перетворення вихідного коду на виконання для C++, Java, C# та Python

Найбільш прямим шляхом є компіляція C++ до машинного коду. У даному дослідженні на платформі Linux застосовується GCC (GNU Compiler Collection), на платформі Windows – MSVC (Microsoft Visual C++). Обидва компілятори підтримують ієрархію рівнів оптимізації: для збірок релізу використовуються прапори -O2 (GCC) та /O2 (MSVC), що забезпечують повний набір стандартних оптимізацій – інлайнінг функцій, усунення мертвого коду, виключення підвиразів – без агресивного розгортання циклів, яке характерне для -O3 і може збільшувати розмір коду [15]. Принципова відмінність між двома компіляторами полягає у наборі внутрішніх оптимізаційних проходів і стратегії векторизації: GCC та MSVC генерують різний машинний код навіть для ідентичного вихідного тексту з однаковим рівнем оптимізації. Саме тому результати C++ на Windows і Linux у цьому дослідженні розглядаються окремо, а не як одна серія вимірювань.

Java і C# поділяють спільну архітектурну ідею – виконання через байт-код із JIT-компіляцією – проте реалізують її незалежно. Середовищем виконання Java є HotSpot JVM у складі OpenJDK 21 (LTS). HotSpot застосовує багаторівневу JIT-компіляцію: на початку виконання методи інтерпретуються або компілюються компілятором C1 з мінімальними затратами часу, а після досягнення порогу частоти викликів (за замовчуванням близько 10 000 звернень) передаються компілятору C2, який виконує глибоку оптимізацію на основі зібраного профілю [9]. Для C# аналогічну роль виконує CLR платформи .NET 8 (LTS) з JIT-компілятором RyuJIT. Незважаючи на схожість концепцій, між JVM та CLR є суттєва практична відмінність: CLR підтримує справжні типи-значення (struct), які розміщуються на стеку або інлайново в масивах без звернення до купи, що знижує навантаження на збирач сміття у певних класах задач. Крім того, поведінка збирачів сміття різна: HotSpot за замовчуванням використовує G1GC з паузами типу «stop-the-world», тоді як CLR застосовує паралельне фонове збирання, яке у .NET 8 суттєво скорочує тривалість пауз [13].

Спільною методичною вимогою для обох JIT-середовищ є необхідність фази прогріву перед початком вимірювань. Дослідження Traini et al. [3]

фіксують, що без прогріву лише 74,7 % запусків Java-програм досягають стабільного стану продуктивності. Тому у методиці, описаній у розділі 3, перші ітерації кожного тесту для Java та C# виключаються з розрахунку і слугують виключно для прогріву JIT-компілятора.

Принципово відмінним є середовище виконання CPython для Python. CPython реалізує стекову віртуальну машину, в якій виконання програми зводиться до покрокового диспетчингу байт-кодових інструкцій у циклі обчислення _PyEval_EvalFrameDefault() [15]. Оскільки стандартний CPython не виконує JIT-компіляцію, фаза прогріву для Python відсутня у тому розумінні, в якому вона є для Java та C#. Разом з тим, починаючи з версії 3.11, у CPython впроваджено адаптивну спеціалізацію опкодів: при повторному виконанні певної інструкції з передбачуваними типами операндів вона замінюється спеціалізованою версією, що скорочує витрати на перевірку типів. Це вносить невелику нелінійність у перші кілька десятків ітерацій Python-тестів і також враховується при обробці результатів. У даному дослідженні застосовується CPython 3.12.

Таким чином, чотири досліджувані мови представляють три якісно різних механізми виконання: статична компіляція з повними оптимізаціями до запуску (C++), адаптивна JIT-компіляція з фазою прогріву (Java, C#) та інтерпретація байт-коду без JIT (Python). Ця відмінність у механізмах є одним з ключових чинників, що визначають характер результатів, отриманих у розділі 4, і має прийматись до уваги при порівнянні показників між мовами.

2.3. Апаратно-програмні платформи

Для забезпечення відтворюваності та порівнянності результатів бенчмаркінгу необхідно чітко визначити апаратно-програмні платформи, на яких проводиться дослідження. Зміна будь-якого параметра платформи між серіями вимірювань здатна внести систематичне зміщення у результати, що унеможливить коректне зіставлення мов програмування між собою.

З точки зору апаратної конфігурації центральним компонентом є процесор. Для даного дослідження обрано процесор з кількістю фізичних ядер, достатньою для виконання як однопотокових, так і багатопотокових тестових задач. Тактова частота процесора безпосередньо визначає продуктивність у CPU-bound задачах, тоді як розмір і ієрархія кешів L1/L2/L3 суттєво впливають на результати у задачах з інтенсивним доступом до пам'яті. Для коректності вимірювань на час проведення тестів вимикаються механізми динамічного масштабування частоти (Turbo Boost / Precision Boost), оскільки вони вносять недетермінізм у часові виміри.

Обсяг та швидкість оперативної пам'яті є визначальними для задач класу memory-bound. Недостатній обсяг RAM призводить до використання файлу підкачки, що кардинально змінює профіль виконання і робить вимірювання нерепрезентативними. У даному дослідженні обсяг оперативної пам'яті достатній для розміщення всіх тестових структур даних у RAM без звернення до диску. Для JVM додатково фіксується розмір купи через параметри запуску, щоб виключити вплив динамічного розширення heap на часові показники.

Підсистема зберігання даних (SSD або HDD) безпосередньо впливає на результати у задачах класу I/O-bound. Різниця у пропускній здатності між HDD (~100–200 МБ/с) та NVMe SSD (~3000–7000 МБ/с) є настільки суттєвою, що може повністю перекрити відмінності між мовами програмування у цьому класі задач. Тому тип накопичувача фіксується як обов'язковий параметр середовища і зазначається у звіті про умови проведення експерименту.

Програмна частина платформи визначається операційною системою та версіями системного програмного забезпечення. У даному дослідженні порівнюються дві операційні системи: Windows 11 та Ubuntu Linux 24.04 LTS. Вибір цих двох систем зумовлений їх широким практичним застосуванням і принциповою відмінністю у реалізації планувальника задач, системних викликів та механізмів управління пам'яттю, що дозволяє виявити платформозалежну складову продуктивності. Версії ядра Linux та драйверів фіксуються, оскільки оновлення планувальника CFS, що вносяться між версіями ядра, можуть

змінювати поведінку при багатопотоковому виконанні. На платформі Windows фіксуються версія ОС та пакет оновлень, оскільки патчі безпеки, пов'язані зі вразливостями Spectre та Meltdown, вносять вимірювані накладні витрати на системні виклики залежно від версії.

Перед кожною серією вимірювань виконується стандартна процедура підготовки платформи: закриваються фонові застосунки, вимикається автоматичне оновлення, система переводиться у режим максимальної продуктивності. На Linux додатково встановлюється governor процесора у режим performance замість стандартного powersave. Ці заходи спрямовані на мінімізацію зовнішнього шуму, який, як показано у роботі Tratt [7], може на кілька порядків збільшувати варіативність wall time вимірювань порівняно з більш стабільними апаратними лічильниками.

2.4. Інструменти вимірювання продуктивності

Вибір інструментів вимірювання продуктивності визначається двома різними задачами, що вирішуються у даному дослідженні. Перша – отримання основних кількісних показників виконання програм: часу виконання, використання процесора та оперативної пам'яті. Друга – діагностика причин відхилень у результатах, що потребує більш детальної інформації про поведінку програми на рівні апаратних лічильників. Для кожної з цих задач застосовується власна група інструментів, а їх спільне використання дозволяє отримати повну картину продуктивності.

Основним засобом вимірювання часу виконання є вбудовані таймери мов програмування. Для кожної з досліджуваних мов застосовується стандартний інструмент, вбудований у її стандартну бібліотеку. У C++ використовується std::chrono::high_resolution_clock з бібліотеки <chrono>, що забезпечує наносекундну роздільну здатність і є монотонним годинником, тобто не може зменшуватись між двома послідовними зверненнями [16]. У Java застосовується System.nanoTime(), що також є монотонним таймером з наносекундною

роздільною здатністю і не залежить від системного часу. У C# використовується System.Diagnostics.Stopwatch, що спирається на апаратний лічильник QueryPerformanceCounter на Windows та POSIX clock_gettime на Linux. У Python застосовується time.perf_counter(), який повертає час із найвищою доступною роздільною здатністю на конкретній платформі. Усі чотири інструменти вимірюють wall time – реальний астрономічний час від початку до кінця виконання вимірюваного фрагменту коду, що є основною метрикою у даному дослідженні відповідно до обґрунтування, наведеного у підрозділі 1.1.

Суттєвим методичним моментом є те, що самі виклики таймерів вносять певні накладні витрати. Для тестів з тривалістю виконання понад кілька секунд ці накладні витрати є нехтовно малими. Однак для коротших задач вимірювання обгортається у зовнішній цикл із фіксованою кількістю повторень, а реєструється загальний час циклу з подальшим діленням на кількість ітерацій. Такий підхід, разом із видаленням аномальних значень та усередненням, забезпечує статистично стійкі результати відповідно до рекомендацій Tratt [7].

Для діагностичного аналізу застосовуються профайлери, прив'язані до операційної системи, а не до конкретної мови. На платформі Linux використовується інструмент perf – частина ядра Linux, що надає доступ до апаратних лічильників PMU (Performance Monitoring Unit). За допомогою команди perf stat для будь-якої запущеної програми – незалежно від мови її реалізації – збираються такі показники, як кількість виконаних інструкцій, кількість тактів процесора, промахи кешів L1/L2/L3, промахи гілок та коефіцієнт IPC (інструкцій на такт) [17]. Ці дані дозволяють визначити, чому певна програма виявляється повільнішою або швидшою за очікуване: наприклад, низьке значення IPC у поєднанні з великою кількістю промахів кешу L3 однозначно вказує на задачу класу memory-bound. На платформі Windows аналогічну роль виконує Windows Performance Analyzer (WPA) у складі Windows Performance Toolkit, що також надає доступ до апаратних лічильників через ETW (Event Tracing for Windows).

Моніторинг використання оперативної пам'яті та завантаженості процесора під час виконання тестів здійснюється через бібліотеку psutil – кросплатформну Python-бібліотеку для отримання системних метрик [18]. Вона однаково функціонує на Windows та Linux і дозволяє у реальному часі отримувати RSS (Resident Set Size) – реальний обсяг фізичної пам'яті, виділеної процесу – а також відсоток використання процесора на рівні конкретного процесу. Оскільки psutil запускається у окремому процесі-моніторі паралельно з тестовим процесом, вона не вносить помітних спотворень у вимірювані показники.

Розподіл інструментів за задачами у даному дослідженні таким чином утворює дворівневу схему: вбудовані таймери забезпечують точні часові виміри, що є основним результатом експерименту, а профайлери та системні утиліти слугують інструментами інтерпретації – вони пояснюють отримані результати, але не замінюють основних вимірювань.

РОЗДІЛ 3. ПРОЄКТУВАННЯ СИСТЕМИ БЕНЧМАРКІНГУ

3.1. Обґрунтування вибору тестових алгоритмів

Порівняльний аналіз продуктивності мов програмування втрачає інформативність, якщо тестові задачі не охоплюють різні типи обчислювального навантаження. Як показано у підрозділі 1.1, поведінка програми суттєво залежить від того, який ресурс обчислювальної системи є вузьким місцем – процесор, підсистема пам'яті, підсистема введення-виведення або механізми операційної системи. Мова, що демонструє найкращі результати у CPU-bound задачах, не обов'язково матиме ту саму перевагу у memory-bound сценаріях. Тому набір тестових алгоритмів спроєктовано так, щоб рівномірно представляти всі чотири класи навантажень. Повний перелік алгоритмів наведено у таблиці 3.1.

Таблиця 3.1 – Набір тестових алгоритмів системи бенчмаркінгу

Тип навантаження	Алгоритм	Вхідні дані	Вимірювана метрика
CPU-bound	Quicksort (власна реалізація)	Масив 10^7 випадкових цілих чисел	Час виконання (wall time)
CPU-bound	Обчислення π методом Монте-Карло	10^8 ітерацій	Час виконання (wall time)
Memory-bound	Множення матриць (наївна реалізація)	Матриці 1000×1000 типу double	Час виконання, використання RAM
Memory-bound	Послідовний обхід масиву з підсумовуванням	Масив 10^8 елементів типу double	Час виконання, пропускна здатність

Продовження таблиці 3.1

I/O-bound	Запис та читання файлу	Файл розміром 512 МБ, блок 4 КБ	Час виконання, throughput МБ/с
I/O-bound	Серіалізація масиву у файл та зворотне читання	Масив 10^6 структур	Час виконання
System-level	Створення та завершення потоків	1000 потоків послідовно	Час виконання, кількість потоків/с
System-level	Відкриття та закриття файлових дескрипторів	10^5 операцій open/close	Час виконання

Алгоритми CPU-bound класу спрямовані на максимальне завантаження обчислювального конвеєру процесора з мінімальним зверненням до пам'яті. Quicksort з власною реалізацією – без використання стандартних бібліотечних функцій сортування – обраний тому, що він є однаково виразним алгоритмом на всіх чотирьох мовах і добре відомий у порівняльних дослідженнях продуктивності [19]. Принцип роботи алгоритму наведено на рисунку 3.1. Алгоритм рекурсивно ділить масив на дві частини відносно опорного елемента (pivot): елементи, менші за pivot, переміщуються вліво, більші – вправо, після чого обидві частини сортуються незалежно. Наявність рекурсії та розгалужень навантажує передбачувач переходів процесора і дозволяє виявити відмінності у якості машинного коду, що генерується різними середовищами виконання. Обчислення числа π методом Монте-Карло є суто арифметичним навантаженням без будь-яких звернень до пам'яті, окрім генерації псевдовипадкових чисел, що дозволяє оцінити чисту обчислювальну швидкодію кожної мови.

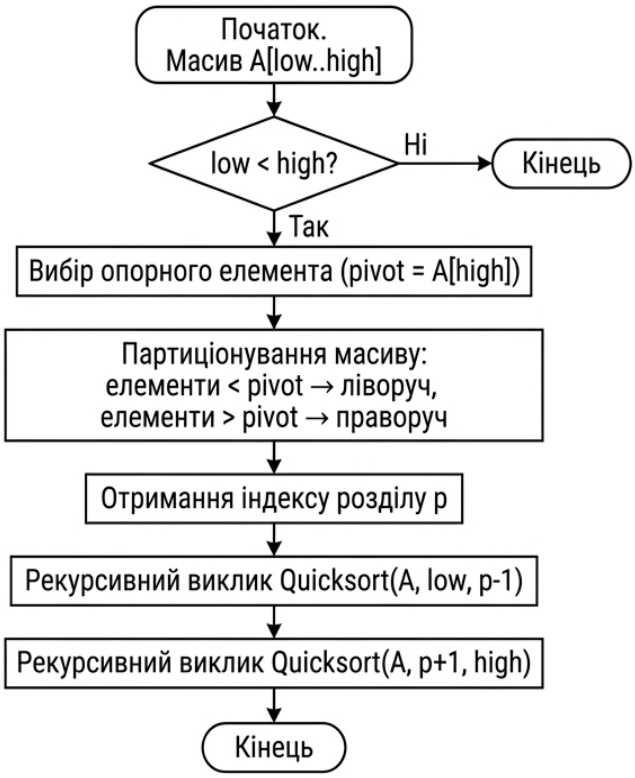

Рисунок 3.1 – Блок-схема алгоритму Quicksort

Memory-bound алгоритми підібрані так, щоб робочий набір даних свідомо перевищував розмір кешу L3 процесора і тиснув на підсистему оперативної пам'яті. Наївна реалізація множення матриць розміру 1000×1000 у стандартному порядку обходу (i-j-k) забезпечує інтенсивний некоаліційований доступ до пам'яті, що є типовим сценарієм для оцінювання пропускної здатності підсистеми пам'яті [20]. Схему алгоритму з позначенням порядку доступу до елементів матриць наведено на рисунку 3.2. При стандартному порядку циклів (i-j-k) доступ до рядків матриці A є послідовним і кеш-дружнім, тоді як доступ до стовпців матриці B є некоаліційованим і генерує значну кількість промахів кешу L3 – саме цей ефект робить задачу чутливою до пропускної здатності пам'яті. Послідовний обхід великого масиву з підсумовуванням, навпаки, є максимально простим з погляду обчислень, але вимагає читання значного обсягу даних з RAM, що дозволяє вимірювати фактичну пропускну здатність пам'яті для кожного середовища виконання.

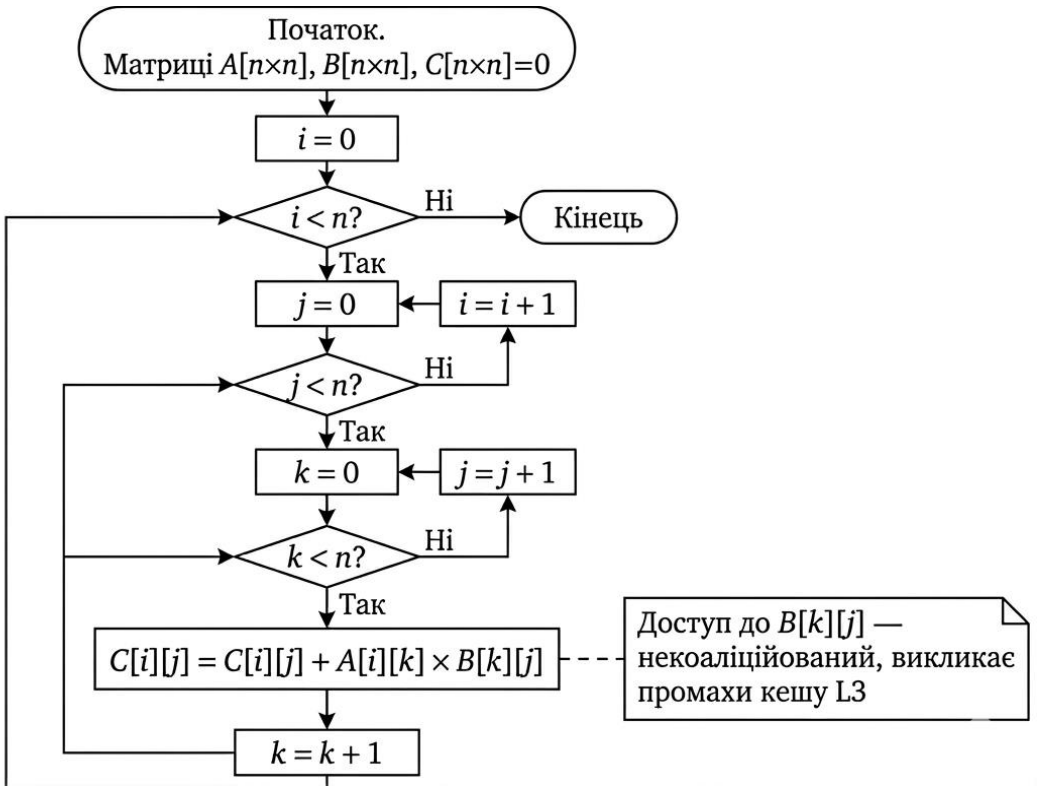

Рисунок 3.2 – Блок-схема наївного алгоритму множення матриць з позначенням порядку доступу до пам'яті

I/O-bound задачі орієнтовані на виявлення відмінностей між мовами у продуктивності блокових операцій з файловою системою. Запис і читання файлу фіксованого розміру виконується побуферно з розміром блоку 4 КБ, що відповідає стандартному розміру сторінки пам'яті і є типовим для реальних застосувань. Серіалізація масиву структур у файл і зворотне читання оцінює не лише продуктивність I/O, а й накладні витрати на перетворення даних, які суттєво відрізняються між мовами.

Задачі системного рівня досліджують накладні витрати на взаємодію з операційною системою, що є найбільш платформозалежним класом навантажень. Послідовне створення і завершення потоків навантажує планувальник ОС і виявляє відмінності між тим, як кожна мова взаємодіє з примітивами потоків операційної системи. Масові операції відкриття і закриття файлових дескрипторів оцінюють накладні витрати на системні виклики, які суттєво відрізняються між Windows та Linux.

Принциповою вимогою до всього набору є ідентичність алгоритмів між мовами: кожен з восьми тестів реалізується однаково на C++, Java, C# та Python з використанням власних конструкцій мови без звернення до зовнішніх обчислювальних бібліотек. Вхідні дані для кожного тесту задаються однаковими параметрами і генеруються з однаковим зерном генератора псевдовипадкових чисел, що забезпечує ідентичність вхідних послідовностей між мовами. Кількість повторень кожного тесту та процедура усереднення результатів описані у підрозділі 3.2.

3.2. Методика проведення експерименту

Коректність порівняльного бенчмаркінгу визначається не лише вибором алгоритмів, а й суворим дотриманням єдиної методики проведення вимірювань. Відсутність стандартизованої процедури є однією з головних причин, через які результати різних досліджень важко зіставити між собою [2]. У даній роботі методика будується навколо кількох взаємопов'язаних принципів, кожен з яких усуває конкретне джерело систематичного або випадкового похибки. Загальний порядок проведення одного бенчмарку наведено на рисунку 3.3.

Рисунок 3.3 – Блок-схема порядку проведення одного бенчмарку

Першим принципом є ідентичність алгоритмів і вхідних даних між мовами. Як зазначено у підрозділі 3.1, кожен тест реалізується власними засобами мови без використання зовнішніх бібліотек. Вхідні дані генеруються один раз з фіксованим зерном генератора псевдовипадкових чисел (seed = 42) і зберігаються у файл, який зчитується кожною мовою перед початком тесту. Це унеможливлює ситуацію, коли різниця у результатах пояснюється різними вхідними послідовностями, а не продуктивністю мови. Для числових задач, де вхідні дані генеруються безпосередньо у коді, кожна мова використовує лінійний конгруентний генератор з ідентичними параметрами.

Другим принципом є обов'язковий прогрів середовищ виконання з JIT-компіляцією. Для Java та C# перші 5 ітерацій кожного тесту виконуються до початку вимірювань і не враховуються у результатах. Це обґрунтовано тим, що у початкових ітераціях JIT-компілятор ще не завершив профілювання гарячих методів і генерацію оптимізованого машинного коду [3]. Дослідження Traini et

al. [3] показують, що навіть після прогріву частина запусків може не досягати стабільного стану продуктивності, тому кількість прогрівних ітерацій встановлена з запасом. Для C++ та Python прогрів у класичному розумінні відсутній, однак перша ітерація все одно виключається, щоб нівелювати ефект холодного кешу операційної системи та сторінкових таблиць.

Третім принципом є статистично достатня кількість повторень. Кожен тест виконується 30 разів після прогріву. Значення 30 є стандартним порогом для застосування центральної граничної теореми, що дозволяє вважати вибіркове середнє наближено нормально розподіленим незалежно від розподілу окремих вимірювань [7]. Для тестів з тривалістю виконання понад 60 секунд кількість повторень знижується до 10, що є компромісом між статистичною якістю та практичним часом проведення експерименту.

Четвертим принципом є видалення аномальних значень перед усередненням. З 30 вимірювань відкидаються 2 найбільших та 2 найменших значення (20% trimmed mean), після чого обчислюється середнє арифметичне та стандартне відхилення по решті 26 вимірюванням. Такий підхід усуває вплив поодиноких викидів, спричинених активністю фонових процесів, плановим збиранням сміття або зовнішніми перериваннями, не відкидаючи при цьому надмірну частку даних. Поряд із середнім значенням фіксується медіана, яка є більш стійкою до асиметричних розподілів, що характерні для JIT-середовищ у перехідному стані.

П'ятим принципом є виключення сторонніх факторів під час проведення вимірювань. Перед кожною серією тестів система переводиться у стандартний стан: закриваються фонові застосунки, вимикається автоматичне оновлення та антивірусне сканування, на Linux встановлюється governor процесора у режим performance. Тести різних мов для одного алгоритму запускаються в одному сеансі роботи системи без перезавантаження, щоб виключити вплив температурного стану процесора. Між серіями тестів витримується пауза 5 секунд для охолодження теплового буфера CPU і скидання стану кешів. Порядок запуску мов у кожній серії рандомізується, щоб виключити систематичну

перевагу мов, що запускаються першими на холодному кеші, або останніми на прогрітому.

Шостим принципом є окреме проведення вимірювань для кожної платформи. Результати на Windows (MSVC + .NET + OpenJDK + CPython) і Linux (GCC + OpenJDK + .NET + CPython) збираються незалежно у різних сеансах. Між платформами не допускається паралельне запускання тестів або спільне використання файлів вхідних даних через мережу, оскільки мережевий I/O вніс би систематичний шум у I/O-bound вимірювання.

Усі результати вимірювань записуються у структуровані CSV-файли з фіксацією метаданих кожного запуску: назва тесту, мова, платформа, номер повторення, wall time у мілісекундах, використання RAM у МБ та CPU load у відсотках. Це забезпечує відтворюваність аналізу і дозволяє повторно обробити дані при зміні методики усереднення.

3.3. Підготовка середовищ виконання

Коректність результатів бенчмаркінгу безпосередньо залежить від того, наскільки точно задокументовано і відтворено умови проведення експерименту. Підготовка середовища виконання охоплює три взаємопов'язані аспекти: конфігурацію компіляторів та інтерпретаторів, апаратні характеристики платформи та версії системного програмного забезпечення. Усі перелічені параметри фіксуються до початку вимірювань і залишаються незмінними протягом усієї серії тестів на кожній платформі.

Дослідження проводиться на двох програмних платформах: Windows 11 та Ubuntu Linux 24.04 LTS. Обидві платформи розгорнуто на одному і тому самому фізичному обладнанні, що виключає вплив апаратних відмінностей на результати міжплатформного порівняння. Апаратна конфігурація тестової машини наведена у таблиці 3.2 та на рисунку 3.4.

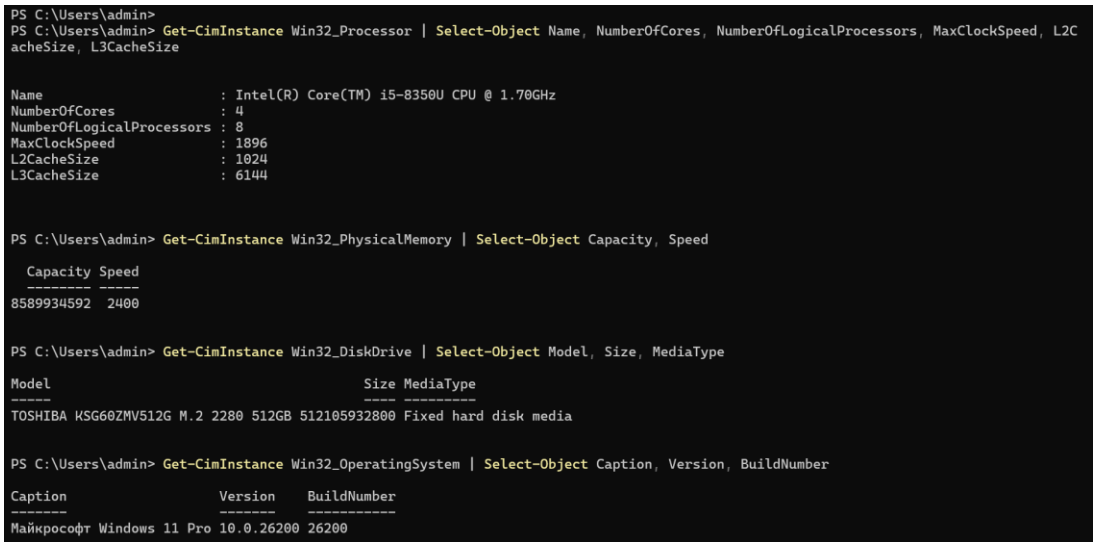

Рисунок 3.4 – Апаратна конфігурація тестової платформи

Таблиця 3.2 – Апаратна конфігурація тестової платформи

Компонент	Характеристика
Процесор	Intel Core i5-8350U, 4 / 8
Базова тактова частота	1,70 ГГц (Turbo Boost до 3,60 ГГц)
Кеш L2	1024 КБ (256 КБ на ядро)
Кеш L3	6144 КБ (спільний)
Оперативна пам'ять	8 ГБ DDR4-2400
Накопичувач	Toshiba KSG60ZMV512G M.2 2280 512 ГБ

Процесор Intel Core i5-8350U належить до мікроархітектури Kaby Lake Refresh і підтримує набір інструкцій до AVX2 включно, що є важливим для оцінки потенціалу автовекторизації у компіляторах C++. Обсяг оперативної пам'яті 8 ГБ достатній для розміщення всіх тестових структур даних у RAM без звернення до файлу підкачки, що перевірялось окремо перед початком кожної серії вимірювань. Накопичувач типу NVMe SSD забезпечує пропускну здатність близько 500 МБ/с для послідовного читання і запису, що є характеристикою середнього рівня для даного класу пристроїв і є репрезентативним для типових робочих станцій.

На платформі Windows 11 Pro (збірка 26200) встановлено такі середовища виконання (рисунок 3.5): компілятор GCC 13.3 у складі дистрибутиву WinLibs для компіляції програм на C++; Java HotSpot VM 26 як середовище виконання Java; .NET 10.0 як платформа виконання C#; CPython 3.13 як інтерпретатор Python.

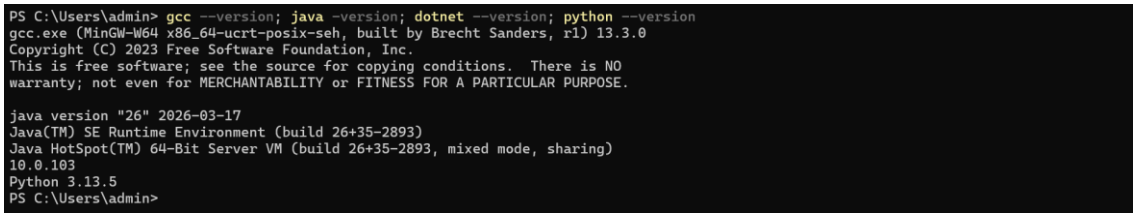

Рисунок 3.5 – Версії середовищ виконання на платформі Windows 11

На платформі Ubuntu Linux 24.04 LTS встановлено такі середовища виконання (рисунок 3.6): компілятор GCC 13.3 зі стандартного репозиторію дистрибутиву; Java HotSpot VM 26, що збігається з версією на Windows; .NET 10.0; CPython 3.13.

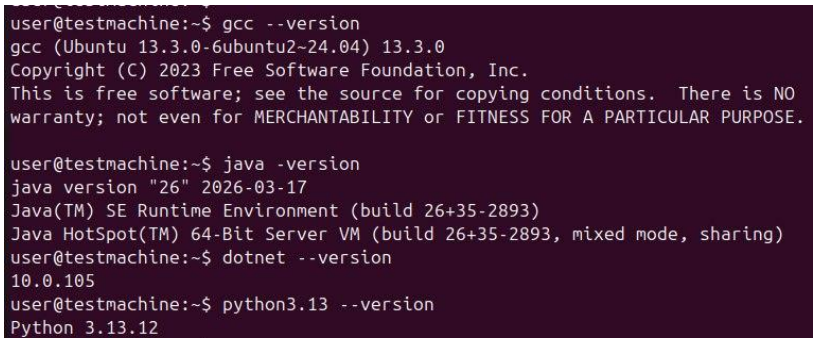

Рисунок 3.6 – Версії середовищ виконання на платформі Ubuntu Linux 24.04 LTS

Версії усіх середовищ виконання є ідентичними на обох платформах, що забезпечує максимальну порівнянність результатів і виключає вплив версійних відмінностей на вимірювані показники продуктивності. Зведену конфігурацію програмних середовищ наведено у таблиці 3.3.

Таблиця 3.3 – Версії середовищ виконання на тестових платформах

Компонент	Windows 11 Pro	Ubuntu Linux 24.04 LTS
Компілятор C++	GCC 13.3 (WinLibs)	GCC 13.3
Java	HotSpot VM 26	HotSpot VM 26
.NET / C#	.NET 10.0	.NET 10.0
Python	CPython 3.13	CPython 3.13
Версія ОС	Windows 11 Pro, збірка 26200	Ubuntu Linux 24.04 LTS

Перед кожною серією вимірювань на обох платформах виконується стандартна процедура підготовки: закриваються фонові застосунки та служби, що не є необхідними для роботи середовища виконання; на Linux встановлюється governor процесора у режим performance командою cpupower frequency-set -g performance; на Windows вмикається план живлення «Висока продуктивність». Механізм динамічного масштабування тактової частоти Intel Turbo Boost залишається увімкненим, оскільки його відключення потребує змін у BIOS і не відповідає типовим умовам реального використання. Для нівелювання теплового дроселювання між серіями тестів витримується пауза 30 секунд.

РОЗДІЛ 4. ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ПРОГРАМ

4.1. CPU-bound задачі

CPU-bound задачі є першим класом навантажень у даному дослідженні. До цього класу належать задачі, продуктивність яких визначається насамперед швидкістю виконання арифметичних та логічних операцій процесором, а не швидкістю доступу до пам'яті або підсистеми введення-виведення. У рамках дослідження CPU-bound клас представлений двома алгоритмами: обчисленням числа π методом Монте-Карло та сортуванням масиву алгоритмом Quicksort. Реалізації алгоритмів для всіх чотирьох мов наведено у додатках А.1–А.4 (Монте-Карло) та А.5–А.8 (Quicksort).

Тест Монте-Карло є суто обчислювальним навантаженням: програма виконує 10^8 ітерацій генерації псевдовипадкових чисел за допомогою власного лінійного конгруентного генератора та підраховує частку точок, що потрапляють у одиничне коло. Звернення до пам'яті зведені до мінімуму – на кожній ітерації задіяні лише регістри процесора, що робить цей тест показовим для оцінки чистої обчислювальної швидкодії середовища виконання. Результати вимірювань наведено на рисунках 4.1 та 4.2.

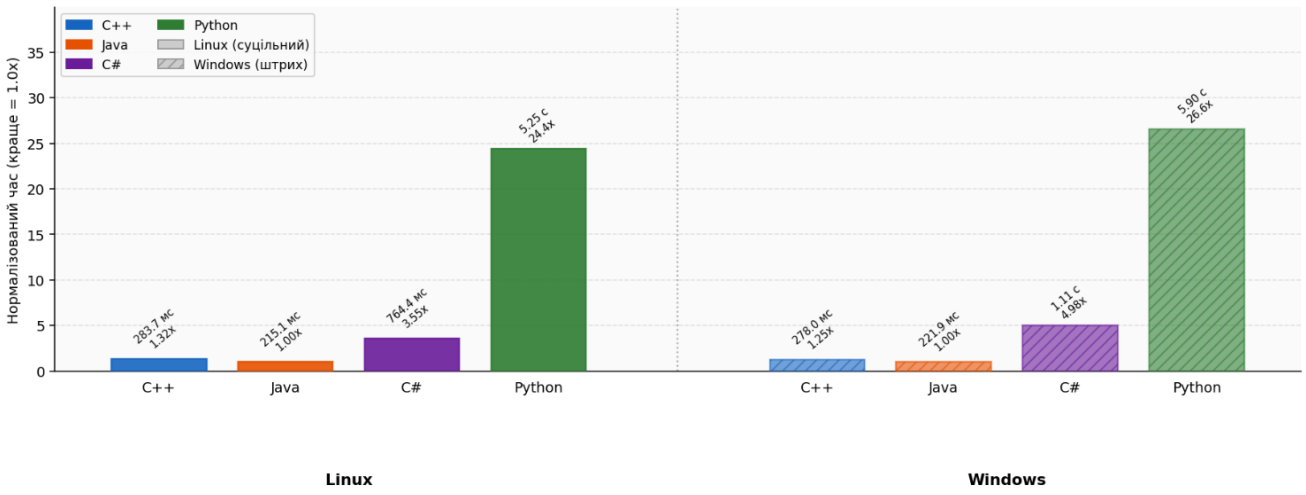

Рисунок 4.1 – Нормалізований час виконання тесту Монте-Карло

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	283.7 мс	282.7 мс	252.2 мс	353.2 мс	17.1 мс	79.9 МБ	99.8%
Java	Linux	**215.1 мс**	216.4 мс	195.5 мс	218.2 мс	4.6 мс	2.1 МБ	100.3%
C#	Linux	764.4 мс	763.2 мс	747.5 мс	780.4 мс	8.7 мс	159.7 МБ	100.1%
Python	Linux	5.25 с	5.25 с	5.21 с	5.35 с	30.1 мс	14.6 МБ	99.6%
C++	Windows	278.0 мс	275.6 мс	261.3 мс	300.6 мс	8.3 мс	4.9 МБ	99.1%
Java	Windows	**221.9 мс**	221.7 мс	208.9 мс	241.6 мс	8.4 мс	2.2 МБ	100.3%
C#	Windows	1.11 с	1.08 с	723.8 мс	1.66 с	348.2 мс	141.7 МБ	98.4%
Python	Windows	5.90 с	5.79 с	5.74 с	6.65 с	242.7 мс	23.3 МБ	99.8%

Рисунок 4.2 – Детальні результати вимірювань тесту Монте-Карло

Як видно з рисунку 4.1, найкращий результат у тесті Монте-Карло демонструє Java: 215,1 мс на Linux та 221,9 мс на Windows. C++ показує дещо вищий час виконання – 283,7 мс на Linux та 278,0 мс на Windows – що є нетиповим результатом для порівняння компільованої мови з JIT-середовищем. Це пояснюється тим, що JIT-компілятор C2 у HotSpot JVM після фази прогріву агресивніше розгортає внутрішній цикл LCG-генератора та краще використовує конвеєр процесора порівняно з GCC на рівні оптимізації -O2, який не застосовує автовекторизацію до циклів з міжітераційними залежностями даних [8]. C# демонструє значно гірший результат – 764,4 мс на Linux та 1,11 с на Windows, що свідчить про менш агресивну оптимізацію RyuJIT для даного типу арифметичного навантаження порівняно з HotSpot C2. Python, як і очікувалось виходячи з аналізу моделі виконання у підрозділі 1.2, відстає від решти мов на один-два порядки: 5,25 с на Linux та 5,90 с на Windows, що у 18–21 разів повільніше за Java. Це відставання зумовлене накладними витратами інтерпретатора CPython на диспетчинг байт-коду та динамічну перевірку типів на кожній з 10^8 ітерацій циклу.

Стандартне відхилення результатів є відносно невеликим для більшості мов: 17,1 мс для C++ на Linux, 4,6 мс для Java на Linux, що свідчить про стабільність вимірювань. Виняток становить Python на Linux – 30,1 мс – що пояснюється нерівномірністю роботи збирача циклічних посилань CPython під час тривалого виконання. Порівняння платформ Windows та Linux для Java

виявляє мінімальну відмінність – 221,9 мс проти 215,1 мс – що підтверджує крос-платформну стабільність HotSpot JVM, зазначену у підрозділі 1.3.

Тест Quicksort навантажує процесор якісно інакше порівняно з Монте-Карло: алгоритм рекурсивно розбиває масив із 10^7 елементів, генеруючи значну кількість розгалужень і непередбачуваних переходів, що інтенсивно навантажує передбачувач переходів процесора. Результати вимірювань наведено на рисунках 4.3 та 4.4.

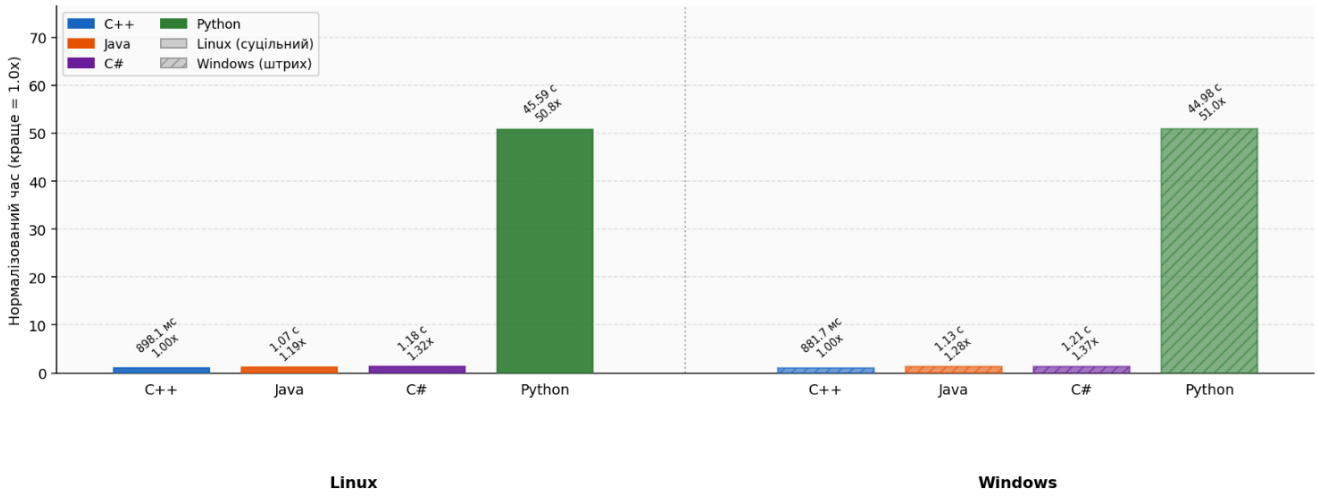

Рисунок 4.3 – Нормалізований час виконання тесту Quicksort

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	**898.1 мс**	882.2 мс	854.1 мс	1.01 с	39.5 мс	79.9 МБ	99.9%
Java	Linux	1.07 с	1.07 с	1.05 с	1.07 с	5.4 мс	81.3 МБ	100.0%
C#	Linux	1.18 с	1.18 с	1.17 с	1.20 с	6.8 мс	157.4 МБ	100.6%
Python	Linux	45.59 с	44.87 с	44.49 с	55.76 с	2.41 с	509.1 МБ	99.8%
C++	Windows	**881.7 мс**	877.8 мс	862.8 мс	912.6 мс	11.0 мс	81.1 МБ	99.6%
Java	Windows	1.13 с	1.12 с	1.07 с	1.25 с	46.7 мс	80.8 МБ	99.8%
C#	Windows	1.21 с	1.19 с	1.16 с	1.39 с	51.3 мс	138.7 МБ	99.7%
Python	Windows	44.98 с	46.68 с	37.78 с	1.1 хв	6.10 с	512.3 МБ	99.6%

Рисунок 4.4 – Детальні результати вимірювань тесту Quicksort

У тесті Quicksort C++ демонструє найкоротший час виконання: 898,1 мс на Linux та 881,7 мс на Windows. Java показує результат, близький до C++: 1,07 с на Linux та 1,13 с на Windows, що становить відставання близько 19–28 % відносно C++. Це узгоджується з очікуваною поведінкою JIT-середовища після

завершення прогріву – HotSpot C2 генерує машинний код якості, наближеної до статично скомпільованого, однак не може повністю усунути накладні витрати на керування об'єктами та перевірки меж масиву, притаманні Java [9]. C# демонструє помітно кращий результат порівняно з Монте-Карло відносно Java: 1,18 с на Linux та 1,21 с на Windows, тобто відставання від C++ становить близько 31–37 %. Відносне покращення C# у Quicksort порівняно з Монте-Карло пояснюється тим, що RyuJIT краще оптимізує код з явними індексними операціями над масивами, ніж щільні арифметичні цикли. Python знову відстає на два порядки: 45,59 с на Linux та 44,98 с на Windows, тобто у 50–51 разів повільніше за C++. Рекурсивна природа алгоритму додатково обтяжує CPython через накладні витрати на кожен виклик функції в інтерпретаторі.

Порівняння платформ у тесті Quicksort виявляє, що результати C++ на Windows та Linux є дуже близькими – різниця менша за 2 %, що підтверджує симетричність компіляторів GCC 13.3 на обох платформах при однаковому рівні оптимізації -O2. Для Java різниця між платформами також є незначною і знаходиться в межах стандартного відхилення. Найстабільнішим у міжплатформному порівнянні є Python – 45,59 с проти 44,98 с – що очікувано, оскільки CPython виконує байт-код незалежно від платформи і його продуктивність визначається насамперед тактовою частотою процесора, а не особливостями операційної системи.

Узагальнюючи результати CPU-bound класу, можна зробити такі висновки. По-перше, для задач з простим арифметичним циклом без складного керування потоком JIT-компілятор HotSpot здатний перевершити статично скомпільований код GCC -O2, що підтверджує адаптивну природу JIT-оптимізації [9]. По-друге, для задач з інтенсивним розгалуженням та рекурсією статична компіляція C++ зберігає перевагу. По-третє, Python у CPU-bound задачах програє решті мов у 18–51 разів залежно від алгоритму, що робить його непридатним для обчислювально інтенсивних задач без залучення нативних бібліотек.

4.2. Memory-bound задачі

Memory-bound задачі є другим класом навантажень у даному дослідженні. Продуктивність програм у цьому класі визначається насамперед пропускною здатністю підсистеми пам'яті, а не обчислювальною потужністю ядер. Клас представлений двома алгоритмами: послідовним обходом масиву з підсумовуванням та наївним множенням матриць. Реалізації для всіх мов наведено у лістингах А.9–А.12 (обхід масиву) та А.13–А.16 (множення матриць).

Тест обходу масиву передбачає послідовне читання масиву з 10^8 елементів типу double із підсумовуванням значень. Алгоритм є максимально простим з обчислювального погляду – на кожній ітерації виконується лише одна операція додавання. Результати вимірювань наведено на рисунках 4.5 та 4.6.

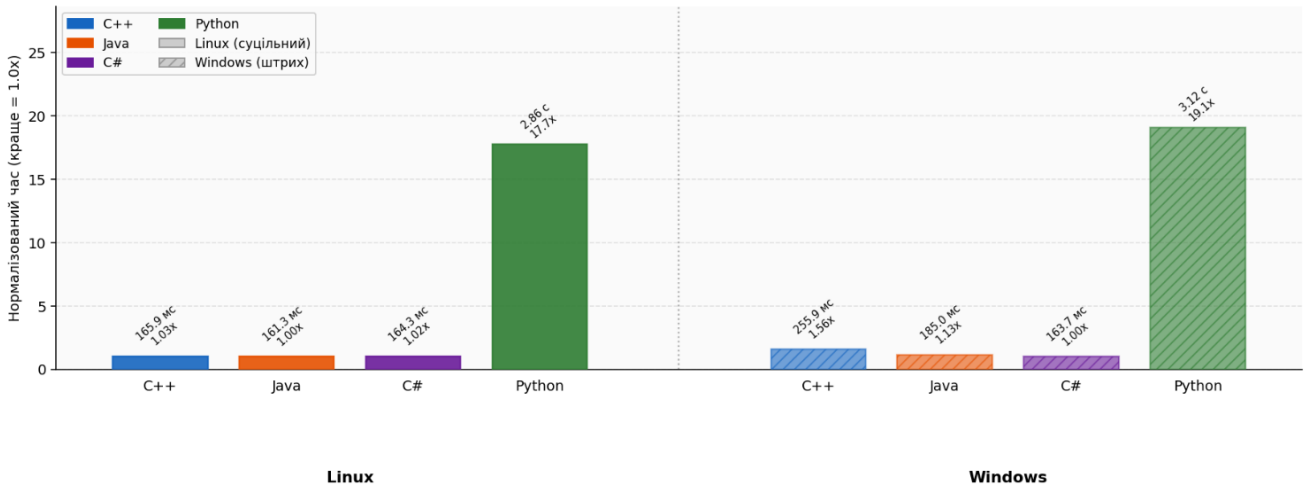

Рисунок 4.5 – Нормалізований час виконання тесту обходу масиву

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	165.9 мс	166.3 мс	154.5 мс	170.6 мс	2.9 мс	766.4 МБ	100.0%
Java	Linux	**161.3 мс**	162.3 мс	143.9 мс	167.1 мс	5.7 мс	765.1 МБ	100.7%
C#	Linux	164.3 мс	165.4 мс	146.8 мс	168.6 мс	4.5 мс	1.53 ГБ	100.0%
Python	Linux	2.86 с	2.85 с	2.74 с	3.11 с	79.1 мс	789.1 МБ	99.8%
C++	Windows	255.9 мс	253.0 мс	199.9 мс	315.1 мс	23.4 мс	767.6 МБ	96.8%
Java	Windows	185.0 мс	178.9 мс	143.5 мс	244.4 мс	26.4 мс	765.2 МБ	98.1%
C#	Windows	**163.7 мс**	157.5 мс	140.8 мс	207.0 мс	14.5 мс	1.52 ГБ	100.6%
Python	Windows	3.12 с	3.07 с	3.01 с	3.53 с	131.0 мс	779.8 МБ	99.6%

Рисунок 4.6 – Детальні результати вимірювань тесту обходу масиву

Результати тесту обходу масиву демонструють принципово іншу картину порівняно з CPU-bound задачами. C++, Java та C# показують дуже близькі результати на Linux: 165,9 мс, 161,3 мс та 164,3 мс відповідно, що знаходиться в межах стандартного відхилення між мовами. Це є прямим підтвердженням того, що при memory-bound навантаженні вузьким місцем є пропускна здатність шини пам'яті, а не швидкість виконання інструкцій – всі три мови однаково обмежені апаратним стелею пропускної здатності RAM [6]. На платформі Windows результати C++ дещо вищі – 255,9 мс проти 165,9 мс на Linux – що пояснюється відмінностями у реалізації управління сторінками пам'яті між ядром Linux та Windows і відповідно різною ефективністю апаратного префетчера при послідовному читанні великих масивів [11].

Споживання оперативної пам'яті у цьому тесті є діагностично важливим показником. C++ утримує 766–767 МБ, що відповідає теоретичному обсягу даних: 10^8 елементів × 8 байт = 800 МБ з незначними накладними витратами на стек та заголовки процесу. C# на обох платформах споживає близько 1,52–1,53 ГБ – майже вдвічі більше за C++. Це пояснюється тим, що CLR виділяє пам'ять під масив типу double[] у керованій купі з додатковими метаданими об'єкта, а також резервує додатковий простір для роботи збирача сміття. Java демонструє аналогічну картину: 765 МБ на Linux та 765 МБ на Windows, що є близьким до C++ завдяки тому, що масив примітивних типів double[] у Java зберігається компактно без boxing.

Значення завантаженості процесора у деяких вимірюваннях перевищує 100 % – зокрема, 100,7 % для C# та 100,6 % для Java. Це є артефактом методики вимірювання: інструмент psutil обчислює завантаженість як відсоток від потужності одного ядра, і значення дещо більше 100% означає що процес на короткі проміжки задіює незначну кількість обчислювальних ресурсів другого ядра – наприклад, для роботи збирача сміття у фоновому потоці. Значення у діапазоні 100,0–100,7 % слід інтерпретувати як однопотокове виконання з мінімальними фоновими активностями середовища виконання.

CPython у тесті обходу масиву відстає від C++ у 17–18 разів: 2,86 с на Linux та 3,12 с на Windows. На відміну від CPU-bound задач, де відставання CPython зумовлене передусім диспетчингом байт-коду, у memory-bound тесті додатковим чинником є те, що CPython зберігає елементи масиву як об'єкти типу float у купі, кожен з яких займає значно більше ніж 8 байт і розміщується некомпактно. Незважаючи на використання модуля array з типом 'd' для компактного зберігання, накладні витрати інтерпретатора на кожну ітерацію циклу залишаються домінуючим чинником [4].

Тест множення матриць є більш обчислювально інтенсивним порівняно з обходом масиву, проте при наївній реалізації з порядком циклів i-j-k доступ до стовпців матриці B є некоаліційованим і генерує значну кількість промахів кешу L3, що переводить задачу у клас memory-bound при великих розмірах матриць [20]. Результати вимірювань наведено на рисунках 4.7 та 4.8.

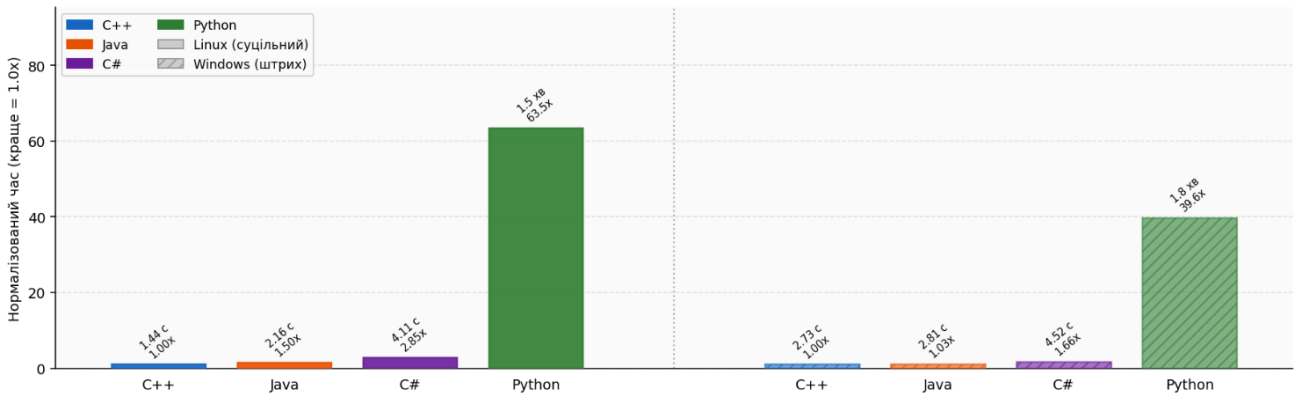

Рисунок 4.7 – Нормалізований час виконання тесту множення матриць

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	**1.44 с**	1.41 с	1.40 с	1.69 с	71.9 мс	79.9 МБ	99.9%
Java	Linux	2.16 с	2.19 с	2.00 с	2.21 с	59.4 мс	26.5 МБ	100.1%
C#	Linux	4.11 с	4.18 с	3.97 с	4.53 с	131.5 мс	68.2 МБ	100.0%
Python	Linux	1.5 хв	1.5 хв	1.5 хв	1.8 хв	4.30 с	128.6 МБ	99.7%
C++	Windows	**2.73 с**	2.77 с	1.72 с	4.32 с	439.1 мс	27.8 МБ	98.9%
Java	Windows	2.81 с	2.82 с	2.42 с	3.02 с	134.6 мс	26.6 МБ	99.6%
C#	Windows	4.52 с	4.43 с	4.30 с	5.05 с	199.3 мс	49.6 МБ	99.7%
Python	Windows	1.8 хв	1.7 хв	1.7 хв	2.2 хв	8.65 с	138.5 МБ	99.5%

Рисунок 4.8 – Детальні результати вимірювань тесту множення матриць

C++ демонструє найкоротший час виконання: 1,44 с на Linux та 2,73 с на Windows. Java показує результат 2,16 с на Linux та 2,81 с на Windows – відставання від C++ становить 50 % на Linux та лише 3 % на Windows. C# виконує тест за 4,11 с на Linux та 4,52 с на Windows, що у 2,85–1,66 рази повільніше за C++ залежно від платформи. Суттєва різниця між платформами для C++ – 1,44 с проти 2,73 с – є найбільшою серед усіх мов у цьому тесті і потребує окремого пояснення: GCC 13.3 на Linux застосовує більш агресивну автовекторизацію внутрішнього циклу множення ніж на Windows, де поведінка компілятора при роботі з UCRT runtime може відрізнятись у стратегії розміщення тимчасових змінних [8].

CPython виконує тест множення матриць за 1,50 хв на Linux та 1,80 хв на Windows, що у 62–40 разів повільніше за C++ відповідно. Таке масштабне відставання зумовлене тим, що наївна реалізація на CPython використовує вкладені списки (list of list) замість компактного масиву, що унеможливлює будь-яку просторову локальність даних. Крім того, кожна операція множення і додавання у внутрішньому циклі вимагає окремого диспетчингу через інтерпретатор, що при 10^9 операціях (1000^3) дає катастрофічне накопичення накладних витрат.

Порівнюючи результати двох memory-bound тестів, слід відзначити, що при послідовному обході масиву різниця між C++, Java та C# практично зникає – всі три мови впираються в однаковий апаратний ліміт пропускної здатності

RAM. Натомість при множенні матриць, де обчислювальна складова є більш значущою, ієрархія мов відновлюється: C++ лідирує, Java наближається до нього, C# відстає. Це підтверджує, що у чисто memory-bound сценаріях вибір мови програмування має значно менший вплив на продуктивність порівняно з CPU-bound задачами.

4.3. I/O-bound задачі

I/O-bound задачі є третім класом навантажень у даному дослідженні. Продуктивність програм у цьому класі визначається швидкістю введення-виведення та накладними витратами на системні виклики ОС. Клас представлений двома алгоритмами: записом файлу розміром 512 МБ блоками по 4 КБ та серіалізацією масиву з 10^6 структур у бінарний файл. Реалізації наведено у лістингах А.17–А.20 (запис файлу) та А.21–А.24 (серіалізація).

Тест запису файлу передбачає послідовний запис 512 МБ даних блоками по 4096 байт через стандартні засоби введення-виведення кожної мови. Результати вимірювань наведено на рисунках 4.9 та 4.10.

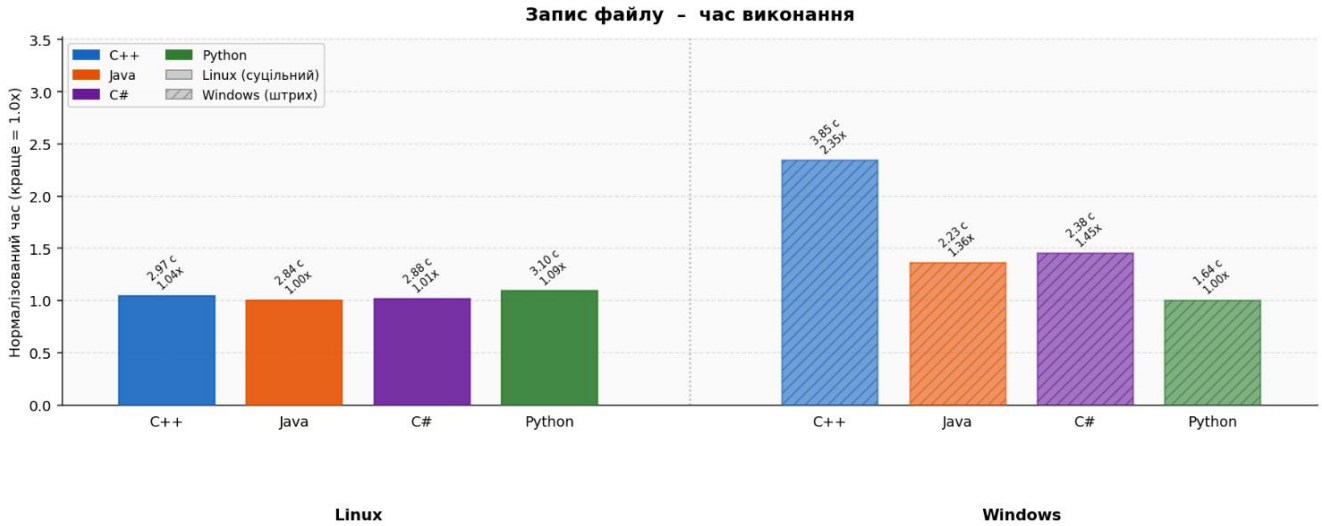

Рисунок 4.9 – Нормалізований час виконання тесту запису файлу

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	2.97 с	2.81 с	1.61 с	4.67 с	680.7 мс	766.4 МБ	26.3%
Java	Linux	**2.84 с**	2.77 с	1.71 с	4.64 с	533.4 мс	2.1 МБ	31.5%
C#	Linux	2.88 с	2.83 с	1.54 с	4.66 с	451.0 мс	1.54 ГБ	28.3%
Python	Linux	3.10 с	2.80 с	1.59 с	5.10 с	904.5 мс	24.2 МБ	35.0%
C++	Windows	3.85 с	3.81 с	2.93 с	5.29 с	552.4 мс	4.7 МБ	92.3%
Java	Windows	2.23 с	2.17 с	1.60 с	3.08 с	394.6 мс	2.2 МБ	86.9%
C#	Windows	2.38 с	2.33 с	1.70 с	3.33 с	382.9 мс	1.52 ГБ	82.5%
Python	Windows	**1.64 с**	1.56 с	1.15 с	2.60 с	379.5 мс	15.7 МБ	58.2%

Рисунок 4.10 – Детальні результати вимірювань тесту запису файлу

Результати тесту запису файлу є найбільш несподіваними серед усіх класів навантажень. На платформі Linux усі чотири мови демонструють близькі результати у діапазоні 2,84–3,10 с, де Java показує найкращий результат – 2,84 с, а CPython найгірший – 3,10 с. Різниця між мовами на Linux є мінімальною і знаходиться в межах стандартного відхилення, що свідчить про те що на Linux вузьким місцем є сама підсистема введення-виведення, а не реалізація буферизації у конкретній мові. Усі чотири мови фактично впираються в однаковий апаратний ліміт послідовного запису NVMe SSD.

На платформі Windows картина кардинально змінюється: C++ стає найповільнішим з результатом 3,85 с, тоді як CPython демонструє найкращий результат – 1,64 с – і випереджає C++ у 2,35 рази. Java та C# показують близькі результати – 2,23 с та 2,38 с відповідно. Таке домінування CPython на Windows пояснюється тим, що стандартна бібліотека Python на Windows використовує Windows API для буферизованого запису з розміром внутрішнього буфера, що краще відповідає характеристикам файлової системи NTFS, тоді як std::ofstream через MSVCRT може виконувати надмірну кількість переходів між простором користувача та ядра при записі блоками 4 КБ [12].

Тест серіалізації передбачає запис масиву з 10^6 структур у бінарний файл та їх зворотне читання. Структура займає 28 байт (4 байти int + 8 байт double + 16 байт рядок). Результати вимірювань наведено на рисунках 4.11 та 4.12.

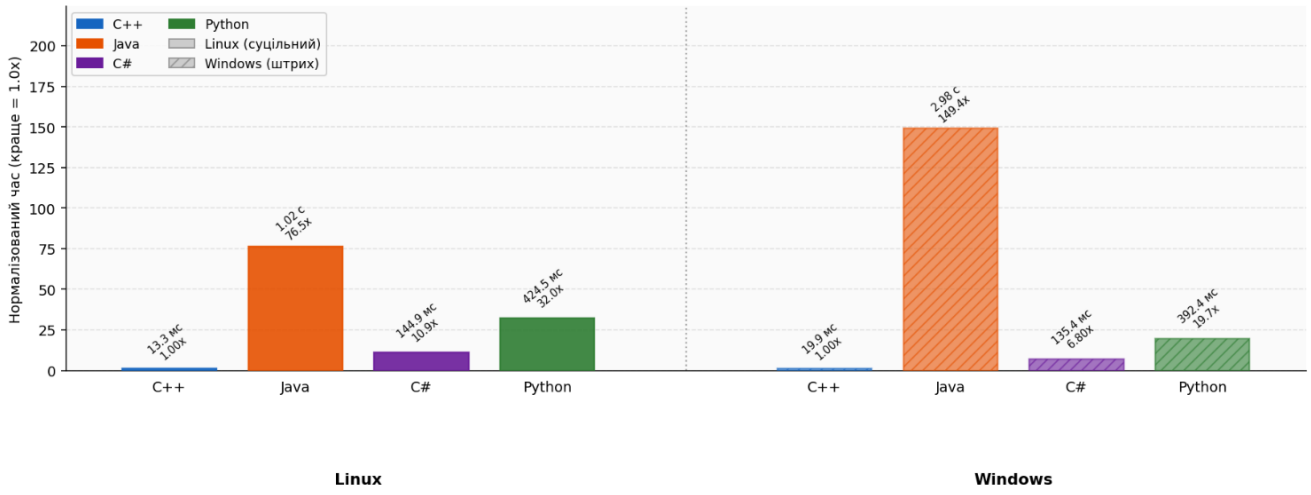

Рисунок 4.11 – Нормалізований час виконання тесту серіалізації

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	**13.3 мс**	13.2 мс	10.8 мс	14.4 мс	0.6 мс	766.4 МБ	98.9%
Java	Linux	1.02 с	1.02 с	982.2 мс	1.04 с	15.1 мс	260.5 МБ	111.4%
C#	Linux	144.9 мс	146.5 мс	123.8 мс	151.3 мс	5.6 мс	108.5 МБ	100.1%
Python	Linux	424.5 мс	426.3 мс	403.0 мс	446.8 мс	12.9 мс	429.0 МБ	99.7%
C++	Windows	**19.9 мс**	17.2 мс	12.2 мс	56.4 мс	8.3 мс	58.2 МБ	95.9%
Java	Windows	2.98 с	2.98 с	2.85 с	3.18 с	73.1 мс	258.7 МБ	103.4%
C#	Windows	135.4 мс	131.6 мс	116.1 мс	191.5 мс	13.8 мс	84.5 МБ	99.8%
Python	Windows	392.4 мс	390.7 мс	381.9 мс	407.7 мс	6.9 мс	394.2 МБ	98.4%

Рисунок 4.12 – Детальні результати вимірювань тесту серіалізації

Результати тесту серіалізації демонструють найбільший розкид між мовами серед усіх восьми тестів. C++ виконує серіалізацію за 13,3 мс на Linux та 19,9 мс на Windows, що на один-два порядки швидше за решту мов. Це пояснюється реалізацією: запис усього масиву з 10^6 структур виконується одним викликом std::ofstream::write() для цілого блоку пам'яті розміром 28 МБ, що фактично зводиться до одного системного виклику write(). Операційна система розміщує ці дані у буфері сторінкового кешу і підтверджує запис без очікування фізичного запису на диск, що і забезпечує надзвичайно малий час виконання [16].

Java демонструє результат 1,02 с на Linux та 2,98 с на Windows – відставання від C++ у 76 та 150 разів відповідно. Незважаючи на те що запис реалізований фрагментами по 4 МБ через власний буфер (~7 системних

викликів), читання виконується без BufferedInputStream – построково по 28 байт, що генерує до 10^6 окремих системних викликів read(). Саме читання є вузьким місцем, а не запис. Суттєва різниця між Linux та Windows – 1,02 с проти 2,98 с – пояснюється тим, що на Linux файлова система ext4 з агресивнішим кешуванням знижує накладні витрати на кожен окремий виклик read(), тоді як на Windows NTFS при великій кількості дрібних читань демонструє вищий overhead [12].

C# показує 144,9 мс на Linux та 135,4 мс на Windows, що є проміжним результатом між C++ та Java. BinaryReader у .NET використовує внутрішню буферизацію при читанні, що суттєво знижує кількість системних викликів порівняно з небуферизованим FileInputStream у Java. CPython виконує серіалізацію за 424,5 мс на Linux та 392,4 мс на Windows через модуль struct, що є прийнятним результатом з огляду на інтерпретовану природу мови – накладні витрати на пакування кожного запису через struct.pack() є домінуючим чинником, а не кількість системних викликів.

Узагальнюючи результати I/O-bound класу, слід відзначити три ключові спостереження. По-перше, реалізація стратегії буферизації має більший вплив на результат ніж вибір мови програмування – один виклик запису великого блоку у C++ дав перевагу у два порядки над посторочним читанням у Java. По-друге, поведінка I/O-bound задач є найбільш платформозалежною серед усіх класів навантажень: різниця між Windows та Linux для окремих мов сягає 2–3 разів, що значно перевищує аналогічні відмінності у CPU-bound та memory-bound тестах. По-третє, CPython у I/O-bound задачах демонструє конкурентоспроможні результати порівняно з компільованими мовами, оскільки накладні витрати інтерпретатора є незначними відносно часу очікування завершення операцій введення-виведення.

4.4. Системні задачі

Системні задачі є четвертим класом навантажень у даному дослідженні. На відміну від попередніх класів, продуктивність у цьому випадку визначається не

характеристиками процесора чи пам'яті, а накладними витратами на взаємодію з операційною системою – зокрема на виклики планувальника, виділення системних ресурсів та переключення контексту. Клас представлений двома тестами: відкриттям та закриттям 10^5 файлових дескрипторів і послідовним створенням та завершенням 1000 потоків. Реалізації для всіх чотирьох мов наведено у лістингах А.25–А.28 (файлові дескриптори) та А.29–А.32 (потоки).

Тест файлових дескрипторів передбачає послідовне виконання 10^5 пар операцій відкриття та закриття тимчасового файлу через стандартні засоби кожної мови. Кожна така пара генерує щонайменше два системні виклики – open та close на Linux або їх еквіваленти на Windows – що робить цей тест показовим для оцінки накладних витрат на взаємодію з ядром ОС. Результати вимірювань наведено на рисунках 4.13 та 4.14.

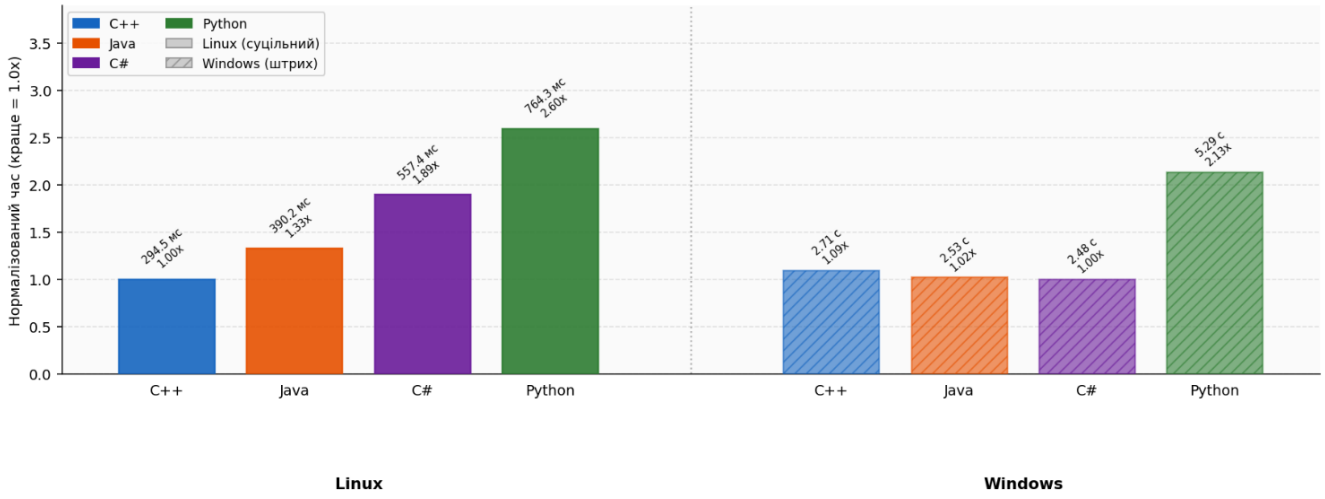

Рисунок 4.13 – Нормалізований час виконання тесту файлових дескрипторів

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	**294.5 мс**	295.0 мс	275.8 мс	297.9 мс	4.4 мс	766.4 МБ	100.0%
Java	Linux	390.2 мс	391.0 мс	373.8 мс	413.3 мс	7.6 мс	77.8 МБ	100.2%
C#	Linux	557.4 мс	555.9 мс	532.6 мс	566.7 мс	6.7 мс	154.7 МБ	100.0%
Python	Linux	764.3 мс	763.6 мс	738.6 мс	807.7 мс	15.9 мс	65.1 МБ	100.0%
C++	Windows	2.71 с	2.54 с	2.45 с	5.80 с	600.1 мс	5.1 МБ	99.7%
Java	Windows	2.53 с	2.51 с	2.46 с	2.70 с	67.4 мс	134.1 МБ	99.7%
C#	Windows	**2.48 с**	2.47 с	2.45 с	2.57 с	26.9 мс	37.0 МБ	99.6%
Python	Windows	5.29 с	5.06 с	5.00 с	6.34 с	377.1 мс	18.3 МБ	99.8%

Рисунок 4.14 – Детальні результати вимірювань тесту файлових дескрипторів

C++ демонструє найкоротший час виконання на Linux – 294,5 мс – що є базовим показником накладних витрат на системні виклики без жодних проміжних рівнів абстракції. Java показує результат 390,2 мс на Linux та 2,53 с на Windows, де відмінність між платформами є найбільш разючою серед усіх мов у цьому тесті – майже у 6,5 разів. C# демонструє 557,4 мс на Linux та 2,48 с на Windows – схожу картину з Java, що свідчить про те що CLR та JVM реалізують відкриття файлів через схожий механізм абстракції над системними викликами, який значно дорожче обходиться на Windows ніж на Linux. CPython показує найгірший результат на Linux – 764,3 мс – проте на Windows демонструє 5,29 с, що є найгіршим результатом серед усіх мов на цій платформі.

На платформі Windows результати усіх мов суттєво погіршуються порівняно з Linux: C++ зростає з 294,5 мс до 2,71 с – у 9,2 рази. Це є найбільш яскравим міжплатформним контрастом у всьому дослідженні і безпосередньо відображає різницю у реалізації файлових системних викликів між ядром Linux та Windows. На Linux виклики open/close є легкими операціями з мінімальним переключенням контексту завдяки архітектурі VFS, тоді як на Windows CreateFile/CloseHandle залучають додаткові рівні перевірок безпеки, реєстрації дескрипторів та взаємодії з менеджером об'єктів ядра NT [12]. Споживання оперативної пам'яті у цьому тесті є мінімальним для всіх мов – від 5,1 МБ для C++ до 154,8 МБ для C# на Linux, що підтверджує що тест навантажує планувальник ОС, а не підсистему пам'яті.

Тест створення потоків передбачає послідовне створення та завершення 1000 потоків, кожен з яких виконує мінімальну роботу – атомарне збільшення лічильника. Час виконання визначається накладними витратами на виділення стеку потоку, реєстрацію у планувальнику ОС та синхронізацію завершення. Результати вимірювань наведено на рисунках 4.15 та 4.16.

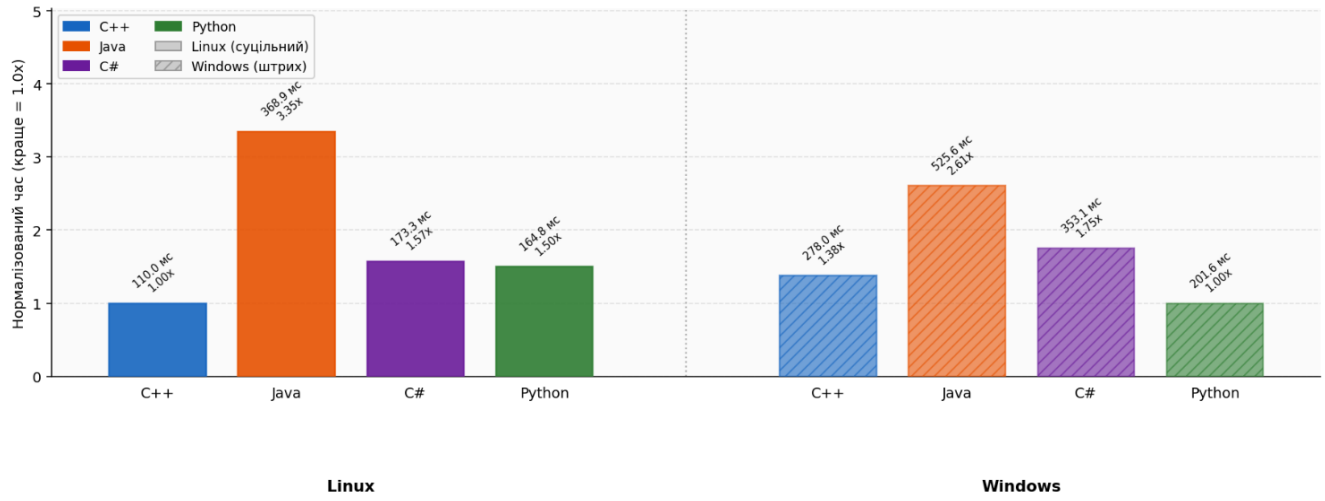

Рисунок 4.15 – Нормалізований час виконання тесту створення потоків

Мова	ОС	Середнє	Медіана	Мін	Макс	Ст.відх.	ОЗУ (сер.)	ЦПУ (сер.)
C++	Linux	**110.0 мс**	111.6 мс	87.0 мс	134.9 мс	11.5 мс	766.4 МБ	109.5%
Java	Linux	368.9 мс	373.9 мс	184.5 мс	393.7 мс	35.8 мс	6.0 МБ	130.9%
C#	Linux	173.3 мс	164.0 мс	113.0 мс	272.6 мс	30.1 мс	134.4 МБ	128.0%
Python	Linux	164.8 мс	142.8 мс	89.9 мс	419.5 мс	68.2 мс	65.1 МБ	119.7%
C++	Windows	278.0 мс	265.3 мс	208.3 мс	367.1 мс	46.6 мс	5.1 МБ	77.5%
Java	Windows	525.6 мс	497.3 мс	209.4 мс	1.07 с	199.3 мс	6.0 МБ	111.5%
C#	Windows	353.1 мс	299.2 мс	190.8 мс	786.6 мс	156.1 мс	67.9 МБ	118.9%
Python	Windows	**201.6 мс**	199.7 мс	194.3 мс	215.1 мс	5.2 мс	18.3 МБ	95.9%

Рисунок 4.16 – Детальні результати вимірювань тесту створення потоків

C++ демонструє найкоротший час виконання: 110,0 мс на Linux та 278,0 мс на Windows. std::thread у MinGW/GCC реалізується безпосередньо через pthread_create на Linux та через відповідний API на Windows без додаткових рівнів абстракції, що забезпечує мінімальні накладні витрати на створення кожного потоку. Java демонструє значно гірший результат – 368,9 мс на Linux та

525,6 мс на Windows, тобто у 3,3 та 1,9 рази повільніше за C++ відповідно. Це пояснюється тим що java.lang.Thread є повноцінним керованим об'єктом JVM: при створенні кожного потоку JVM виділяє стек потоку з власного пулу пам'яті, реєструє об'єкт потоку у системі збирача сміття та виконує ініціалізацію контексту виконання, що є суттєво дорожчою операцією порівняно зі створенням нативного потоку у C++ [9]. C# показує 173,3 мс на Linux та 353,1 мс на Windows – проміжний результат між C++ та Java, що відображає більш легку реалізацію потоків у CLR порівняно з JVM завдяки підтримці типів-значень та меншим накладним витратам на реєстрацію об'єктів потоку.

CPython показує 164,8 мс на Linux та 201,6 мс на Windows, що є несподівано конкурентоспроможним результатом – лише на 50 % повільніше за C++ на Linux. Це пояснюється тим що CPython реалізує потоки через нативні потоки ОС (pthread на Linux, Win32 threads на Windows), а мінімальна робота у кожному потоці – атомарне збільшення лічильника – практично не затримується через GIL, оскільки GIL звільняється між байт-кодовими інструкціями [10]. Таким чином, у даному тесті CPython фактично вимірює накладні витрати ОС на створення потоків, а не накладні витрати інтерпретатора.

Завантаженість процесора у тесті потоків перевищує 100 % для більшості мов: C++ на Linux – 109,5 %, Java – 130,9 %, C# – 128,0 %. Це пояснюється тим що при послідовному створенні потоків ОС короткочасно задіює кілька фізичних ядер одночасно – одне для основного потоку керування і одне або більше для ініціалізації нового потоку та його планування. Інструмент psutil фіксує пікове використання у момент активності кількох ядер, що і дає значення понад 100% відносно потужності одного ядра.

Узагальнюючи результати системного класу навантажень, слід виділити два ключових спостереження. По-перше, саме у цьому класі зафіксована найбільша міжплатформна різниця у всьому дослідженні: для файлових дескрипторів C++ на Windows виявився у 9,2 рази повільнішим ніж на Linux, що безпосередньо відображає архітектурні відмінності між файловими підсистемами Windows NT та Linux. По-друге, у тесті потоків CPython показав

несподівано конкурентоспроможний результат завдяки тому що накладні витрати ОС на створення потоку домінують над накладними витратами інтерпретатора, що є характерною особливістю системних задач на відміну від обчислювальних.

4.5. Зведений порівняльний аналіз

Зведений аналіз результатів дослідження дозволяє узагальнити закономірності продуктивності мов програмування у розрізі всіх класів навантажень та обох платформ. У таблицях 4.1 та 4.2 наведено нормалізований час виконання кожного тесту відносно найкращого результату на відповідній платформі – значення 1,00× відповідає найшвидшій мові у даному тесті, а більші значення показують у скільки разів інші мови поступаються лідеру.

Таблиця 4.1 – Нормалізований час виконання на платформі Linux

Тест	C++	Java	C#	CPython
Монте-Карло	1,32×	1,00×	3,55×	24,4×
Quicksort	1,00×	1,19×	1,31×	50,8×
Обхід масиву	1,03×	1,00×	1,02×	17,7×
Множення матриць	1,00×	1,50×	2,85×	62,5×
Запис файлу	1,05×	1,00×	1,01×	1,09×
Серіалізація	1,00×	76,5×	10,9×	31,9×
Файлові дескриптори	1,00×	1,32×	1,89×	2,59×
Створення потоків	1,00×	3,35×	1,58×	1,50×

Таблиця 4.2 – Нормалізований час виконання на платформі Windows

Тест	C++	Java	C#	CPython
Монте-Карло	1,25×	1,00×	5,00×	26,6×
Quicksort	1,00×	1,28×	1,37×	51,0×
Обхід масиву	1,38×	1,00×	1,47×	16,9×
Множення матриць	1,00×	1,03×	1,66×	39,6×
Запис файлу	2,35×	1,36×	1,45×	1,00×
Серіалізація	1,00×	150×	6,81×	19,7×
Файлові дескриптори	1,09×	1,00×	1,00× (≈)	2,09×
Створення потоків	1,00×	1,89×	1,27×	0,72×

Аналіз таблиць дозволяє виявити стійкі закономірності для кожної мови. C++ лідирує у більшості тестів на обох платформах – зокрема у Quicksort, множенні матриць, серіалізації та створенні потоків – і жодного разу не посідає останнє місце за винятком запису файлу на Windows. Це підтверджує що статична компіляція з оптимізацією -O2 забезпечує стабільно високу продуктивність незалежно від типу навантаження. Єдиним винятком є Монте-Карло, де JIT-компілятор HotSpot після прогріву генерує більш оптимізований машинний код для щільного арифметичного циклу ніж GCC на рівні -O2 [8, 9].

Java демонструє найбільш неоднорідний профіль продуктивності. З одного боку, вона лідирує у Монте-Карло та обході масиву на обох платформах і показує результати близькі до C++ у Quicksort та множенні матриць після прогріву JIT. З іншого боку, серіалізація на Linux виявила критичну проблему реалізації – відсутність буферизації при читанні призвела до відставання у 76,5 разів від C++. Це ілюструє принципову особливість JIT-середовищ: при коректній реалізації вони здатні наближатись до продуктивності статично

скомпільованого коду, проте неоптимальні патерни введення-виведення можуть повністю нівелювати цю перевагу [3].

C# займає стійку третю позицію у більшості CPU-bound та memory-bound тестів, поступаючись як C++ так і Java. Виняток становлять файлові дескриптори на Windows, де C# показує результат на рівні Java, та серіалізація, де внутрішня буферизація BinaryReader забезпечує суттєву перевагу над небуферизованим FileInputStream Java. Порівняно з Java, C# демонструє більш передбачувану поведінку у I/O-bound задачах завдяки кращій буферизації стандартної бібліотеки .NET.

CPython посідає останнє місце у всіх CPU-bound та memory-bound тестах з відставанням від лідера у 17–63 рази залежно від алгоритму. Це відставання є структурним і зумовлене моделлю виконання інтерпретатора, а не якістю реалізації конкретних алгоритмів. Разом з тим у I/O-bound та системних задачах CPython демонструє якісно іншу поведінку: у записі файлу на Windows він є лідером, у створенні потоків на Windows показує найкращий результат (0,72× відносно C++), а у файлових дескрипторах відстає лише у 2–2,6 рази. Це підтверджує відомий практичний принцип: CPython є конкурентоспроможним у задачах де час виконання визначається очікуванням зовнішніх ресурсів, а не обчисленнями в інтерпретаторі [4].

Порівняння платформ виявляє що Linux забезпечує стабільніші та у більшості випадків кращі результати для системних задач – зокрема файлові дескриптори на Linux виконуються у 6–9 разів швидше ніж на Windows для більшості мов. У CPU-bound та memory-bound задачах платформа має менший вплив і різниця між Linux та Windows рідко перевищує 30–50 % для однієї мови. Найбільш платформонейтральними виявились Java у Quicksort та CPython у більшості тестів, що підтверджує крос-платформну стабільність цих середовищ виконання.

ВИСНОВОК

У даній роботі проведено системний аналіз продуктивності мов програмування C++, Java, C# та CPython на апаратно-програмних платформах Windows 11 та Ubuntu Linux 24.04 LTS. Дослідження охопило вісім тестових алгоритмів, що представляють чотири класи обчислювальних навантажень: CPU-bound, memory-bound, I/O-bound та системні задачі. Отримані результати дозволяють сформулювати наступні висновки:

1. У ході теоретичного аналізу встановлено, що модель виконання мови програмування – статична компіляція, JIT-компіляція або інтерпретація байт-коду – є основним чинником, що визначає характер продуктивності, однак не єдиним. Апаратно-програмна платформа, зокрема ієрархія кешів процесора, пропускна здатність підсистеми пам'яті та реалізація системних викликів операційної системи, вносить самостійний і вимірюваний вклад у результати, що підтверджується отриманими даними.

2. Спроєктована система бенчмаркінгу забезпечила відтворювані умови вимірювань: ідентичні алгоритми та вхідні дані для всіх мов, фіксований генератор псевдовипадкових чисел, обов'язковий прогрів JIT-середовищ, 30 робочих ітерацій з відкиданням аномальних значень та усередненням. Застосована методика дозволила мінімізувати вплив системного шуму і отримати статистично стійкі результати.

3. Аналіз результатів CPU-bound тестів показав, що C++ забезпечує найкоротший час виконання у задачах з інтенсивним розгалуженням та рекурсією – зокрема у Quicksort C++ випереджає Java на 19–28% та C# на 31–37%. Водночас у задачах з щільним арифметичним циклом без складного керування потоком HotSpot JVM після завершення прогріву здатний перевершити статично скомпільований код GCC -O2 – Java виявилась швидшою за C++ у тесті Монте-Карло на обох платформах, що підтверджує адаптивну природу JIT-оптимізації. CPython поступився лідерам у 18–51 разів залежно від

алгоритму, що унеможливлює його застосування у обчислювально інтенсивних задачах без залучення нативних бібліотек.

4. У memory-bound задачах встановлено, що при послідовному обході великого масиву різниця між C++, Java та C# практично зникає – усі три мови впираються в однаковий апаратний ліміт пропускної здатності RAM. Це є практично значущим результатом: у сценаріях де вузьким місцем є підсистема пам'яті, вибір між компільованою та JIT-мовою не впливає на продуктивність. При множенні матриць, де обчислювальна складова є більш значущою, ієрархія мов відновлюється на користь C++.

5. Аналіз I/O-bound задач виявив що реалізація стратегії буферизації має більший вплив на результат ніж вибір мови програмування. Запис всього масиву структур одним системним викликом у C++ забезпечив перевагу у 76–150 разів над небуферизованим посторочним читанням у Java, тоді як C# з внутрішньою буферизацією BinaryReader показав проміжний результат. CPython у I/O-bound задачах продемонстрував конкурентоспроможні результати і навіть лідирував у записі файлу на Windows, що підтверджує його придатність для задач де час виконання визначається очікуванням зовнішніх ресурсів.

6. Системні задачі виявили найбільшу платформозалежність серед усіх класів навантажень. Операції з файловими дескрипторами на Windows виявились у 6–9 разів повільнішими ніж на Linux для більшості мов, що безпосередньо відображає архітектурні відмінності між файловими підсистемами Windows NT та Linux. У тесті створення потоків CPython показав несподівано конкурентоспроможний результат завдяки тому що накладні витрати ОС на створення потоку домінують над накладними витратами інтерпретатора.

Таким чином, результати дослідження підтверджують що не існує універсально найпродуктивнішої мови програмування – вибір мови має обґрунтовуватись типом переважаючого навантаження у конкретному застосуванні. C++ є оптимальним вибором для CPU-bound задач з інтенсивним розгалуженням та для системного програмування де критичним є

детермінований час виконання без пауз збирача сміття. Java є конкурентоспроможною альтернативою для тривалих обчислювальних задач де JIT-компілятор має можливість завершити прогрів, а також для крос-платформних застосувань де стабільність міжплатформних результатів є пріоритетом. C# посідає проміжну позицію між C++ та Java і демонструє перевагу у I/O-bound задачах завдяки якісній буферизації стандартної бібліотеки .NET. CPython є придатним вибором для I/O-bound та системних задач, а також як мова інтеграції між нативними бібліотеками, проте є непридатним для обчислювально інтенсивних задач без залучення розширень на C.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Pereira R., Couto M., Ribeiro F., Rua R., Cunha J., Fernandes J. P., Saraiva J. Ranking programming languages by energy efficiency. *Science of Computer Programming*. 2021. Vol. 205. Article 102609. URL: https://doi.org/10.1016/j.scico.2021.102609.

2. van Kempen N., Kwon H.-J., Nguyen D. T., Berger E. D. It's Not Easy Being Green: On the Energy Efficiency of Programming Languages. *arXiv preprint*. 2024. arXiv:2410.05460. URL: https://arxiv.org/abs/2410.05460.

3. Traini L., Cortellessa V., Di Pompeo D., Tucci M. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering*. 2023. Vol. 28. Article 13. URL: https://doi.org/10.1007/s10664-022-10247-x.

4. Oliveira R., Oprescu A., Uta A. An Empirical Study on the Performance and Energy Usage of Compiled Python Code. *arXiv preprint*. 2025. arXiv:2505.02346. URL: https://arxiv.org/abs/2505.02346.

5. Hennessy J. L., Patterson D. A. Computer Architecture: A Quantitative Approach. 6th ed. Cambridge : Morgan Kaufmann, 2019. 936 p.

6. Drepper U. What Every Programmer Should Know About Memory. *Red Hat, Inc.* 2007. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf (дата звернення: 01.04.2026).

7. Tratt L. What Metric to Use When Benchmarking? *Laurence Tratt's Blog*. 2022. URL: https://tratt.net/laurie/blog/2022/what_metric_to_use_when_benchmarking.html (дата звернення: 01.04.2026).

8. Fog A. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. Copenhagen : Technical University of Denmark, 2023. URL: https://www.agner.org/optimize/optimizing_cpp.pdf (дата звернення: 01.04.2026).

9. Lambert J., Casey K., Monahan R. Repositioning Tiered HotSpot Execution Performance Relative to the Interpreter. *arXiv preprint*. 2023. arXiv:2304.06460. URL: https://arxiv.org/abs/2304.06460.

10. Loustau A. et al. Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications. *ACM Computing Surveys*. 2023. Vol. 56, No. 2. URL: https://doi.org/10.1145/3617588.

11. Afzal A., Hager G., Wellein G. Analytic performance model for parallel overlapping memory-bound kernels. *Concurrency and Computation: Practice and Experience*. 2022.

12. Larabel M. Ubuntu Linux Squeezes ~20% More Performance Than Windows 11 On New AMD Zen 4 Threadripper. *Phoronix*. 2023. URL: https://www.phoronix.com/review/threadripper-7995wx-windows-linux (дата звернення: 01.04.2026).

13. Microsoft. Common Language Runtime (CLR) overview. *Microsoft Learn*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/standard/clr (дата звернення: 01.04.2026).

14. Oracle. Java Garbage Collection Basics. *Oracle Technology Network*. URL: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html (дата звернення: 01.04.2026).

15. Zhang Z. et al. Quantifying the interpretation overhead of Python. *Science of Computer Programming*. 2022. Vol. 214. Article 102726.

16. cppreference.com. std::chrono::high_resolution_clock. URL: https://en.cppreference.com/w/cpp/chrono/high_resolution_clock (дата звернення: 01.04.2026).

17. Gregg B. perf: Linux profiling with performance counters. URL: https://www.brendangregg.com/perf.html (дата звернення: 01.04.2026).

18. Rodolà G. psutil documentation. URL: https://psutil.readthedocs.io (дата звернення: 01.04.2026).

19. Hindawi. Improved Matrix Multiplication by Changing Loop Order. *Mobile Information Systems*. 2022. URL: https://doi.org/10.1155/2022/9650652.

20. Sun X. H. et al. The Memory-Bounded Speedup Model and Its Impacts in Computing. *Journal of Computer Science and Technology*. 2023. Vol. 38, No. 1. P. 64–79.

21. Галагуз Т., Кузьмич М., Ляшук Т. Вплив архітектурних особливостей ARM та x86 на продуктивність системних операцій у мовах програмування високого рівня // Proceedings of the 2nd International Scientific and Practical Conference «Current Challenges in Scientific Research». Wroclaw (Poland), 1–3 December 2025. С. 144–147.

22. Галагуз Т., Ляшук Т. Архітектурні аспекти виконання програм мовами C++, Java, C# та Python // Сучасні інформаційні технології: від теорії до практики (СІнТех 2026) : матеріали І Міжнародної науково-практичної конференції, м. Луцьк, 22–23 травня 2026 р. Луцьк, 2026.

ДОДАТКИ

Додаток А

Лістинги програмного коду тестових алгоритмів

Лістинг А.1 – Обчислення числа π методом Монте-Карло (C++)

```
#include "benchmark_base.h"
#include <cstdint>
#include <iostream>
#include <iomanip>

static constexpr uint64_t LCG_MUL = 636413622384679300 5ULL;
static constexpr uint64_t LCG_ADD = 1442695040888963407ULL;

static inline double lcg_next_double(uint64_t& state) {
    state = state * LCG_MUL + LCG_ADD;
    // 53-bit mantissa: shift right 11 bits, divide by 2^53
    return static_cast<double>(state >> 11) / static_cast<double>(1ULL << 53);
}

static double monte_carlo_pi(uint64_t iterations) {
    uint64_t state  = 42;
    // "+r" — компілятор вважає що asm читає І пише state,
    // тому не може вважати state константою і розгорнути/прорахувати цикл
заздалегідь
    asm volatile("" : "+r"(state));
    uint64_t inside = 0;

    for (uint64_t i = 0; i < iterations; ++i) {
        double x = lcg_next_double(state);
        double y = lcg_next_double(state);
        if (x * x + y * y <= 1.0) ++inside;
    }

    return 4.0 * static_cast<double>(inside) / static_cast<double>(iterations);
}

void run_cpu_monte_carlo() {
    constexpr uint64_t N = 100'000'000;
    double pi_result = 0.0;

    run_benchmark("monte_carlo", /*warmup=*/1, /*iterations=*/30, [&]() {
        pi_result = monte_carlo_pi(N);
        // "r,m" — змушує компілятор реально обчислити pi_result кожну ітерацію,
        // не може закешувати результат між викликами
        asm volatile("" : : "r,m"(pi_result) : "memory");
```

```
    });

    std::cout << std::fixed << std::setprecision(7)
              << "VERIFY: π ≈ " << pi_result << "\n\n";
}
```

Лістинг А.2 – Обчислення числа π методом Монте-Карло (Java)

```
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

public class CpuMonteCarlo {

    private static final long LCG_MUL = 6364136223846793005L;
    private static final long LCG_ADD = 1442695040888963407L;

    private static double monteCarloPi(long iterations) {
        long state  = 42L;
        long inside = 0L;

        for (long i = 0; i < iterations; i++) {
            state = state * LCG_MUL + LCG_ADD;
            double x = (double) (state >>> 11) / (double) (1L << 53);
            state = state * LCG_MUL + LCG_ADD;
            double y = (double) (state >>> 11) / (double) (1L << 53);
            if (x * x + y * y <= 1.0) inside++;
        }

        return 4.0 * inside / (double) iterations;
    }

    public static void run() throws Exception {
        final long N = 100_000_000L;
        double[] result = {0.0};

        BenchmarkRunner.run("monte_carlo", () -> {
            result[0] = monteCarloPi(N);
        });

        System.out.printf("VERIFY: π ≈ %.7f%n%n", result[0]);
    }
}
```

Лістинг А.3 – Обчислення числа π методом Монте-Карло (C#)

```
using System;
```

```
public static class CpuMonteCarlo
{
    private const ulong LcgMul = 6364136223846793005UL;
    private const ulong LcgAdd = 1442695040888963407UL;

    private static double MonteCarloPi(ulong iterations)
    {
        ulong state  = 42UL;
        ulong inside = 0UL;

        for (ulong i = 0; i < iterations; i++)
        {
            state = unchecked(state * LcgMul + LcgAdd);
            double x = (double)(state >> 11) / (double)(1UL << 53);
            state = unchecked(state * LcgMul + LcgAdd);
            double y = (double)(state >> 11) / (double)(1UL << 53);
            if (x * x + y * y <= 1.0) inside++;
        }

        return 4.0 * inside / (double)iterations;
    }

    public static void Run()
    {
        const ulong N = 100_000_000UL;
        double result = 0.0;

        BenchmarkRunner.Run("monte_carlo", () =>
        {
            result = MonteCarloPi(N);
        });

        Console.WriteLine($"VERIFY: π ≈ {result:F7}\n");
    }
}
```

Лістинг А.4 – Обчислення числа π методом Монте-Карло (CPython)

```
from benchmark_runner import run_benchmark

LCG_MUL  = 6364136223846793005
LCG_ADD  = 1442695040888963407
LCG_MASK = 0xFFFFFFFFFFFFFFFF
_53BIT   = float(1 << 53)

# Python is ~100x slower than C++, so we use 10^7 instead of 10^8
N = 10_000_000
```

```
def monte_carlo_pi(iterations: int) -> float:
    state  = 42
    inside = 0
    for _ in range(iterations):
        state = (state * LCG_MUL + LCG_ADD) & LCG_MASK
        x = (state >> 11) / _53BIT
        state = (state * LCG_MUL + LCG_ADD) & LCG_MASK
        y = (state >> 11) / _53BIT
        if x * x + y * y <= 1.0:
            inside += 1
    return 4.0 * inside / iterations


def run() -> None:
    result = [0.0]

    def task():
        result[0] = monte_carlo_pi(N)

    run_benchmark('monte_carlo', task)
    print(f'VERIFY: π ≈ {result[0]:.7f}  (N={N:,})\n')
```

Лістинг А.5 – Сортування масиву алгоритмом Quicksort (C++)

```
#include "benchmark_base.h"
#include <cstring>
#include <fstream>
#include <vector>
#include <iostream>

static int partition(int* arr, int low, int high) {
    int pivot = arr[high];
    int i     = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            ++i;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

static void quicksort(int* arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
```

```
}

void run_cpu_quicksort() {
    constexpr int N = 10'000'000;
    const char*   path = "../input_data/array_10M.bin";

    std::vector<int> original(N);
    {
        std::ifstream f(path, std::ios::binary);
        if (!f) { std::cerr << "ERROR: cannot open " << path << "\n"; return; }
        f.read(reinterpret_cast<char*>(original.data()), N * sizeof(int));
    }

    std::vector<int> work(N);

    run_benchmark("quicksort", /*warmup=*/1, /*iterations=*/30, [&]() {
        std::memcpy(work.data(), original.data(), N * sizeof(int));
        quicksort(work.data(), 0, N - 1);
    });

    std::cout << "VERIFY: first=" << work[0] << "  last=" << work[N - 1] << "\n\n";
}
```

Лістинг А.6 – Сортування масиву алгоритмом Quicksort (Java)

```
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

import java.io.FileInputStream;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.channels.FileChannel;

public class CpuQuicksort {

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i     = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
            }
        }
        int tmp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = tmp;
        return i + 1;
    }
```

```
    private static void quicksort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quicksort(arr, low, pi - 1);
            quicksort(arr, pi + 1, high);
        }
    }

    public static void run() throws Exception {
        final int N    = 10_000_000;
        final String path = "../input_data/array_10M.bin";

        // Read int32 little-endian
        int[] original = new int[N];
        try (FileInputStream fis = new FileInputStream(path);
             FileChannel ch = fis.getChannel()) {
            ByteBuffer buf = ByteBuffer.allocateDirect(N *
4).order(ByteOrder.LITTLE_ENDIAN);
            while (buf.hasRemaining()) ch.read(buf);
            buf.flip();
            for (int i = 0; i < N; i++) original[i] = buf.getInt();
        }

        int[] work = new int[N];

        BenchmarkRunner.run("quicksort", () -> {
            System.arraycopy(original, 0, work, 0, N);
            quicksort(work, 0, N - 1);
        });

        System.out.printf("VERIFY: first=%d  last=%d%n%n", work[0], work[N - 1]);
    }
}
```

Лістинг А.7 – Сортування масиву алгоритмом Quicksort (C#)

```
using System;
using System.IO;
using System.Runtime.InteropServices;

public static class CpuQuicksort
{
    private static int Partition(int[] arr, int low, int high)
    {
        int pivot = arr[high];
        int i     = low - 1;
        for (int j = low; j < high; j++)
        {
            if (arr[j] <= pivot)
```

```
            {
                i++;
                (arr[i], arr[j]) = (arr[j], arr[i]);
            }
        }
        (arr[i + 1], arr[high]) = (arr[high], arr[i + 1]);
        return i + 1;
    }

    private static void Quicksort(int[] arr, int low, int high)
    {
        if (low < high)
        {
            int pi = Partition(arr, low, high);
            Quicksort(arr, low, pi - 1);
            Quicksort(arr, pi + 1, high);
        }
    }

    public static void Run()
    {
        const int N    = 10_000_000;
        const string path = "../input_data/array_10M.bin";

        // Read int32 little-endian
        int[] original = new int[N];
        using (var fs = new FileStream(path, FileMode.Open, FileAccess.Read))
        {
            byte[] bytes = new byte[N * 4];
            fs.ReadExactly(bytes);
            Buffer.BlockCopy(bytes, 0, original, 0, bytes.Length); // host is LE on
Windows
        }

        int[] work = new int[N];

        BenchmarkRunner.Run("quicksort", () =>
        {
            Array.Copy(original, work, N);
            Quicksort(work, 0, N - 1);
        });

        Console.WriteLine($"VERIFY: first={work[0]}  last={work[N - 1]}\n");
    }
}
```

Лістинг А.8 – Сортування масиву алгоритмом Quicksort (CPython)

```
import array
```

```
import sys
from benchmark_runner import run_benchmark

sys.setrecursionlimit(50_000)


def partition(arr: list, low: int, high: int) -> int:
    pivot = arr[high]
    i     = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quicksort(arr: list, low: int, high: int) -> None:
    if low < high:
        pi = partition(arr, low, high)
        quicksort(arr, low, pi - 1)
        quicksort(arr, pi + 1, high)


def run() -> None:
    path = '../input_data/array_10M.bin'
    N    = 10_000_000

    with open(path, 'rb') as f:
        buf = array.array('i')
        buf.fromfile(f, N)
    original = list(buf)

    work: list = []

    def task():
        nonlocal work
        work = original.copy()
        quicksort(work, 0, N - 1)

    run_benchmark('quicksort', task)
    print(f'VERIFY: first={work[0]}  last={work[N - 1]}\n')
```

Лістинг А.9 – Послідовний обхід масиву з підсумовуванням (C++)

```
#include "benchmark_base.h"
#include <vector>
#include <fstream>
#include <iostream>
```

```cpp
#include <iomanip>

void run_mem_array_traverse() {
    constexpr int N = 100'000'000;
    const char*   path = "../input_data/array_100M.bin";

    std::vector<double> arr(N);
    {
        std::ifstream f(path, std::ios::binary);
        if (!f) { std::cerr << "ERROR: cannot open " << path << "\n"; return; }
        f.read(reinterpret_cast<char*>(arr.data()), N * sizeof(double));
    }

    // volatile prevents the compiler from eliminating the summation loop
    volatile double total_sum = 0.0;

    run_benchmark("array_traverse", /*warmup=*/1, /*iterations=*/30, [&]() {
        double s = 0.0;
        const double* p   = arr.data();
        const double* end = p + N;
        while (p != end) s += *p++;
        total_sum = s;
    });

    std::cout << std::fixed << std::setprecision(4)
              << "VERIFY: sum=" << total_sum << "\n\n";
}
```

Лістинг А.10 – Послідовний обхід масиву з підсумовуванням (Java)

```java
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

import java.io.FileInputStream;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public class MemArrayTraverse {

    // volatile prevents JIT from eliminating the summation
    private static volatile double totalSum = 0.0;

    public static void run() throws Exception {
        final int    N    = 100_000_000;
        final String path = "../input_data/array_100M.bin";

        double[] arr = new double[N];
```

```
        // Read float64 little-endian in chunks (avoids double-buffering 800 MB)
        final int CHUNK = 1_000_000;
        byte[] bytes  = new byte[CHUNK * 8];
        ByteBuffer buf = ByteBuffer.wrap(bytes).order(ByteOrder.LITTLE_ENDIAN);

        try (FileInputStream fis = new FileInputStream(path)) {
            int offset = 0;
            while (offset < N) {
                int count = Math.min(CHUNK, N - offset);
                int toRead = count * 8;
                int read = 0;
                while (read < toRead) {
                    int r = fis.read(bytes, read, toRead - read);
                    if (r < 0) break;
                    read += r;
                }
                buf.rewind();
                for (int i = 0; i < count; i++) arr[offset + i] = buf.getDouble();
                offset += count;
            }
        }

        BenchmarkRunner.run("array_traverse", () -> {
            double s = 0.0;
            for (int i = 0; i < N; i++) s += arr[i];
            totalSum = s;
        });

        System.out.printf("VERIFY: sum=%.4f%n%n", totalSum);
    }
}
```

Лістинг А.11 – Послідовний обхід масиву з підсумовуванням (C#)

```
using System;
using System.IO;
using System.Runtime.CompilerServices;

public static class MemArrayTraverse
{
    public static void Run()
    {
        const int    N    = 100_000_000;
        const string path = "../input_data/array_100M.bin";

        double[] arr = new double[N];
        using (var fs = new FileStream(path, FileMode.Open, FileAccess.Read))
        {
            byte[] bytes = new byte[N * 8];
```

```
            fs.ReadExactly(bytes);
            Buffer.BlockCopy(bytes, 0, arr, 0, bytes.Length); // LE on Windows
        }

        double totalSum = 0.0;

        BenchmarkRunner.Run("array_traverse", () =>
        {
            double s = 0.0;
            for (int i = 0; i < N; i++) s += arr[i];
            // Prevent JIT from eliminating the loop
            [MethodImpl(MethodImplOptions.NoInlining)]
            static void Sink(double v) { }
            Sink(s);
            totalSum = s;
        });

        Console.WriteLine($"VERIFY: sum={totalSum:F4}\n");
    }
}
```

Лістинг А.12 – Послідовний обхід масиву з підсумовуванням (CPython)

```
import array
from benchmark_runner import run_benchmark

# 10^8 doubles = ~800 MB; reduce to 10^7 (~80 MB) if memory is tight
def run() -> None:
    path = '../input_data/array_100M.bin'
    n    = 100_000_000

    print(f'Loading {n:,} float64 from {path} ...')
    try:
        with open(path, 'rb') as f:
            buf = array.array('d')
            buf.fromfile(f, n)
    except MemoryError:
        print('WARNING: not enough RAM for 10^8 doubles, reducing to 10^7')
        n = 10_000_000
        with open(path, 'rb') as f:
            buf = array.array('d')
            buf.fromfile(f, n)

    total_sum = [0.0]

    def task():
        s = 0.0
        for v in buf:
            s += v
```

```
        total_sum[0] = s

    run_benchmark('array_traverse', task)
    print(f'VERIFY: sum={total_sum[0]:.4f}\n')
```

Лістинг А.13 – Наївне множення матриць 1000×1000 (C++)

```
#include "benchmark_base.h"
#include <vector>
#include <iostream>
#include <iomanip>
#include <cstdint>

static constexpr uint64_t LCG_MUL = 6364136223846793005ULL;
static constexpr uint64_t LCG_ADD = 1442695040888963407ULL;

static void naive_matmul(const double* __restrict__ A,
                         const double* __restrict__ B,
                               double* __restrict__ C,
                         int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                C[i * n + j] += A[i * n + k] * B[k * n + j];
}

void run_mem_matrix_multiply() {
    constexpr int N = 1000;

    std::vector<double> A(N * N), B(N * N), C(N * N);

    // Initialise A and B outside the measurement loop (seed=42)
    {
        uint64_t state = 42;
        for (int i = 0; i < N * N; ++i) {
            state = state * LCG_MUL + LCG_ADD;
            A[i]  = static_cast<double>(state >> 11) / static_cast<double>(1ULL <<
53);
            state = state * LCG_MUL + LCG_ADD;
            B[i]  = static_cast<double>(state >> 11) / static_cast<double>(1ULL <<
53);
        }
    }

    run_benchmark("matrix_multiply", /*warmup=*/1, /*iterations=*/30, [&]() {
        std::fill(C.begin(), C.end(), 0.0);
        naive_matmul(A.data(), B.data(), C.data(), N);
    });
```

```
    std::cout << std::fixed << std::setprecision(6)
              << "VERIFY: C[0][0]=" << C[0]
              << "  C[999][999]="   << C[N * N - 1] << "\n\n";
}
```

Лістинг А.14 – Наївне множення матриць 1000×1000 (Java)

```
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

public class MemMatrixMultiply {

    private static final long LCG_MUL = 6364136223846793005L;
    private static final long LCG_ADD = 1442695040888963407L;

    public static void run() throws Exception {
        final int N = 1000;

        double[] A = new double[N * N];
        double[] B = new double[N * N];
        double[] C = new double[N * N];

        // Initialise A and B outside measurement (seed=42)
        long state = 42L;
        for (int i = 0; i < N * N; i++) {
            state = state * LCG_MUL + LCG_ADD;
            A[i]  = (double) (state >>> 11) / (double) (1L << 53);
            state = state * LCG_MUL + LCG_ADD;
            B[i]  = (double) (state >>> 11) / (double) (1L << 53);
        }

        BenchmarkRunner.run("matrix_multiply", () -> {
            // Clear C
            java.util.Arrays.fill(C, 0.0);
            // Naive i-j-k
            for (int i = 0; i < N; i++)
                for (int j = 0; j < N; j++)
                    for (int k = 0; k < N; k++)
                        C[i * N + j] += A[i * N + k] * B[k * N + j];
        });

        System.out.printf("VERIFY: C[0][0]=%.6f  C[999][999]=%.6f%n%n",
                          C[0], C[N * N - 1]);
    }
}
```

Лістинг А.15 – Наївне множення матриць 1000×1000 (C#)

```
using System;

public static class MemMatrixMultiply
{
    private const ulong LcgMul = 6364136223846793005UL;
    private const ulong LcgAdd = 1442695040888963407UL;

    public static void Run()
    {
        const int N = 1000;

        double[] A = new double[N * N];
        double[] B = new double[N * N];
        double[] C = new double[N * N];

        // Initialise A and B outside measurement (seed=42)
        ulong state = 42UL;
        for (int i = 0; i < N * N; i++)
        {
            state = unchecked(state * LcgMul + LcgAdd);
            A[i]  = (double)(state >> 11) / (double)(1UL << 53);
            state = unchecked(state * LcgMul + LcgAdd);
            B[i]  = (double)(state >> 11) / (double)(1UL << 53);
        }

        BenchmarkRunner.Run("matrix_multiply", () =>
        {
            Array.Clear(C);
            for (int i = 0; i < N; i++)
                for (int j = 0; j < N; j++)
                    for (int k = 0; k < N; k++)
                        C[i * N + j] += A[i * N + k] * B[k * N + j];
        });

        Console.WriteLine($"VERIFY: C[0][0]={C[0]:F6}  C[999][999]={C[N * N -
1]:F6}\n");
    }
}
```

Лістинг А.16 – Наївне множення матриць 1000×1000 (CPython)

```
from benchmark_runner import run_benchmark

LCG_MUL  = 6364136223846793005
LCG_ADD  = 1442695040888963407
LCG_MASK = 0xFFFFFFFFFFFFFFFF
_53BIT   = float(1 << 53)
```

```
N = 1000


def run() -> None:
    # Initialise A and B outside measurement (seed=42)
    state = 42
    A: list[list[float]] = []
    for i in range(N):
        row = []
        for j in range(N):
            state = (state * LCG_MUL + LCG_ADD) & LCG_MASK
            row.append((state >> 11) / _53BIT)
        A.append(row)

    B: list[list[float]] = []
    for i in range(N):
        row = []
        for j in range(N):
            state = (state * LCG_MUL + LCG_ADD) & LCG_MASK
            row.append((state >> 11) / _53BIT)
        B.append(row)

    C: list[list[float]] = [[0.0] * N for _ in range(N)]

    def task():
        for row in C:
            for k in range(N):
                row[k] = 0.0
        for i in range(N):
            for j in range(N):
                s = 0.0
                Ai = A[i]
                for k in range(N):
                    s += Ai[k] * B[k][j]
                C[i][j] = s

    run_benchmark('matrix_multiply', task)
    print(f'VERIFY: C[0][0]={C[0][0]:.6f}  C[999][999]={C[N-1][N-1]:.6f}\n')
```

Лістинг А.17 – Запис та читання файлу блоками по 4 КБ (C++)

```
#include "benchmark_base.h"
#include <fstream>
#include <vector>
#include <iostream>
#include <cstdio>

#ifdef _WIN32
```

```
#  define WIN32_LEAN_AND_MEAN
#  include <windows.h>
static std::string tmp_path() {
    char buf[MAX_PATH];
    GetTempPathA(MAX_PATH, buf);
    return std::string(buf) + "bench_io_rw.bin";
}
#else
static std::string tmp_path() { return "/tmp/bench_io_rw.bin"; }
#endif

void run_io_file_readwrite() {
    constexpr size_t FILE_SIZE = 512ULL * 1024 * 1024; // 512 MiB
    constexpr size_t BLOCK     = 4096;

    const std::string tmp = tmp_path();

    // Fill block with non-zero pattern
    std::vector<char> wbuf(BLOCK, static_cast<char>(0xAB));
    std::vector<char> rbuf(BLOCK);

    // ── write benchmark ──────────────────────────────────────────
    run_benchmark("file_write", /*warmup=*/1, /*iterations=*/30, [&]() {
        std::ofstream f(tmp, std::ios::binary | std::ios::trunc);
        for (size_t written = 0; written < FILE_SIZE; written += BLOCK)
            f.write(wbuf.data(), static_cast<std::streamsize>(BLOCK));
    });

    // ── read benchmark ───────────────────────────────────────────
    run_benchmark("file_read", /*warmup=*/1, /*iterations=*/30, [&]() {
        std::ifstream f(tmp, std::ios::binary);
        while (f.read(rbuf.data(), static_cast<std::streamsize>(BLOCK))) {}
    });

    std::remove(tmp.c_str());
    std::cout << "VERIFY: temp file removed (" << tmp << ")\n\n";
}
```

Лістинг А.18 – Запис та читання файлу блоками по 4 КБ (Java)

```
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

import java.io.*;
import java.nio.file.*;

public class IoFileReadWrite {
```

```
    private static Path tmpPath() {
        return Paths.get(System.getProperty("java.io.tmpdir"), "bench_io_java.bin");
    }

    public static void run() throws Exception {
        final long FILE_SIZE = 512L * 1024 * 1024; // 512 MiB
        final int  BLOCK     = 4096;
        final Path tmp       = tmpPath();

        byte[] wbuf = new byte[BLOCK];
        java.util.Arrays.fill(wbuf, (byte) 0xAB);
        byte[] rbuf = new byte[BLOCK];

        // ── write benchmark ──────────────────────────────────────────
        BenchmarkRunner.run("file_write", () -> {
            try (FileOutputStream fos = new FileOutputStream(tmp.toFile())) {
                long written = 0;
                while (written < FILE_SIZE) {
                    fos.write(wbuf);
                    written += BLOCK;
                }
            }
        });

        // ── read benchmark ───────────────────────────────────────────
        BenchmarkRunner.run("file_read", () -> {
            try (FileInputStream fis = new FileInputStream(tmp.toFile())) {
                //noinspection StatementWithEmptyBody
                while (fis.read(rbuf) != -1) {}
            }
        });

        Files.deleteIfExists(tmp);
        System.out.println("VERIFY: temp file removed (" + tmp + ")\n");
    }
}
```

Лістинг А.19 – Запис та читання файлу блоками по 4 КБ (C#)

```
using System;
using System.IO;

public static class IoFileReadWrite
{
    private static string TmpPath() =>
        Path.Combine(Path.GetTempPath(), "bench_io_csharp.bin");

    public static void Run()
    {
```

```
        const long FILE_SIZE = 512L * 1024 * 1024; // 512 MiB
        const int  BLOCK     = 4096;

        string tmp  = TmpPath();
        byte[] wbuf = new byte[BLOCK];
        Array.Fill(wbuf, (byte)0xAB);
        byte[] rbuf = new byte[BLOCK];

        // ── write benchmark ──────────────────────────────────────────
        BenchmarkRunner.Run("file_write", () =>
        {
            using var fs = new FileStream(tmp, FileMode.Create, FileAccess.Write,
                                          FileShare.None, BLOCK);
            long written = 0;
            while (written < FILE_SIZE)
            {
                fs.Write(wbuf, 0, BLOCK);
                written += BLOCK;
            }
        });

        // ── read benchmark ───────────────────────────────────────────
        BenchmarkRunner.Run("file_read", () =>
        {
            using var fs = new FileStream(tmp, FileMode.Open, FileAccess.Read,
                                          FileShare.Read, BLOCK);
            while (fs.Read(rbuf, 0, BLOCK) > 0) { }
        });

        File.Delete(tmp);
        Console.WriteLine($"VERIFY: temp file removed ({tmp})\n");
    }
}
```

Лістинг А.20 – Запис та читання файлу блоками по 4 КБ (CPython)

```
import os
import tempfile
from benchmark_runner import run_benchmark

FILE_SIZE = 512 * 1024 * 1024  # 512 MiB
BLOCK     = 4096


def run() -> None:
    tmp = os.path.join(tempfile.gettempdir(), 'bench_io_python.bin')
    wbuf = b'\xab' * BLOCK
    rbuf = bytearray(BLOCK)
```

```
    # ── write benchmark ──────────────────────────────────────────────
    def write_task():
        with open(tmp, 'wb') as f:
            written = 0
            while written < FILE_SIZE:
                f.write(wbuf)
                written += BLOCK

    run_benchmark('file_write', write_task)

    # ── read benchmark ───────────────────────────────────────────────
    def read_task():
        with open(tmp, 'rb') as f:
            while f.readinto(rbuf):
                pass

    run_benchmark('file_read', read_task)

    os.remove(tmp)
    print(f'VERIFY: temp file removed ({tmp})\n')
```

Лістинг А.21 – Серіалізація масиву структур у бінарний файл (C++)

```
#include "benchmark_base.h"
#include <fstream>
#include <vector>
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <cstring>
#include <cstdint>

// Layout must match generate_inputs.py: struct.pack('=id16s', id, value, name)
// = : native byte-order, no alignment padding between fields
// i : int32  (4 bytes)
// d : float64 (8 bytes)
// 16s: char[16]
// Total: 28 bytes
#pragma pack(push, 1)
struct Record {
    int32_t id;
    double  value;
    char    name[16];
};
#pragma pack(pop)

static_assert(sizeof(Record) == 28, "Record size mismatch — check packing");

#ifdef _WIN32
```

```
#  define WIN32_LEAN_AND_MEAN
#  include <windows.h>
static std::string tmp_path() {
    char buf[MAX_PATH];
    GetTempPathA(MAX_PATH, buf);
    return std::string(buf) + "bench_ser.bin";
}
#else
static std::string tmp_path() { return "/tmp/bench_ser.bin"; }
#endif

void run_io_serialization() {
    constexpr int N    = 1'000'000;
    const char*   path = "../input_data/structs_1M.bin";

    std::vector<Record> records(N);
    {
        std::ifstream f(path, std::ios::binary);
        if (!f) { std::cerr << "ERROR: cannot open " << path << "\n"; return; }
        f.read(reinterpret_cast<char*>(records.data()), N * sizeof(Record));
    }

    const std::string tmp = tmp_path();
    std::vector<Record> read_back(N);

    // ── write benchmark ──────────────────────────────────────────
    run_benchmark("serialization_write", /*warmup=*/1, /*iterations=*/30, [&]() {
        std::ofstream f(tmp, std::ios::binary | std::ios::trunc);
        f.write(reinterpret_cast<const char*>(records.data()),
                N * static_cast<std::streamsize>(sizeof(Record)));
    });

    // ── read benchmark ───────────────────────────────────────────
    run_benchmark("serialization_read", /*warmup=*/1, /*iterations=*/30, [&]() {
        std::ifstream f(tmp, std::ios::binary);
        f.read(reinterpret_cast<char*>(read_back.data()),
               N * static_cast<std::streamsize>(sizeof(Record)));
    });

    std::remove(tmp.c_str());

    auto trim = [](const char* s, int len) {
        int n = len;
        while (n > 0 && (s[n-1] == '\0' || s[n-1] == ' ')) --n;
        return std::string(s, n);
    };

    std::cout << std::fixed << std::setprecision(6)
              << "VERIFY first: id=" << read_back[0].id
              << "  value="          << read_back[0].value
```

```
              << "  name="           << trim(read_back[0].name, 16) << "\n"
              << "VERIFY last:  id=" << read_back[N-1].id
              << "  value="          << read_back[N-1].value
              << "  name="           << trim(read_back[N-1].name, 16) << "\n\n";
}
```

Лістинг А.22 – Серіалізація масиву структур у бінарний файл (Java)

```
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

import java.io.*;
import java.nio.*;
import java.nio.file.*;

/**
 * Binary record format (little-endian, matches structs_1M.bin):
 *   int32  id     (4 bytes)
 *   float64 value (8 bytes)
 *   char[16] name (16 bytes, zero-padded ASCII)
 *   Total: 28 bytes
 */
public class IoSerialization {

    private static final int RECORD_SIZE = 28;
    private static final int N           = 1_000_000;

    record Record(int id, double value, byte[] name) {}

    private static Path tmpPath() {
        return Paths.get(System.getProperty("java.io.tmpdir"),
"bench_ser_java.bin");
    }

    private static Record[] readStructs(String path) throws IOException {
        Record[] records = new Record[N];
        byte[] buf = new byte[RECORD_SIZE];
        try (FileInputStream fis = new FileInputStream(path)) {
            for (int i = 0; i < N; i++) {
                int read = 0;
                while (read < RECORD_SIZE) {
                    int r = fis.read(buf, read, RECORD_SIZE - read);
                    if (r < 0) throw new EOFException();
                    read += r;
                }
                ByteBuffer bb = ByteBuffer.wrap(buf).order(ByteOrder.LITTLE_ENDIAN);
                int    id    = bb.getInt();
                double value = bb.getDouble();
```

```
                byte[] name  = new byte[16];
                bb.get(name);
                records[i] = new Record(id, value, name);
            }
        }
        return records;
    }

    public static void run() throws Exception {
        final String srcPath = "../input_data/structs_1M.bin";
        final Path   tmp     = tmpPath();

        Record[] records   = readStructs(srcPath);
        Record[] readBack  = new Record[N];
        byte[]   recBuf    = new byte[RECORD_SIZE];

        // Reusable write buffer: 4 MB
        final int CHUNK = 4 * 1024 * 1024 / RECORD_SIZE; // ~145k records
        byte[] wchunk = new byte[CHUNK * RECORD_SIZE];

        // ── write benchmark ──────────────────────────────────────────
        BenchmarkRunner.run("serialization_write", () -> {
            try (FileOutputStream fos = new FileOutputStream(tmp.toFile())) {
                int offset = 0;
                while (offset < N) {
                    int count = Math.min(CHUNK, N - offset);
                    ByteBuffer wb = ByteBuffer.wrap(wchunk, 0, count * RECORD_SIZE)
                                              .order(ByteOrder.LITTLE_ENDIAN);
                    for (int i = 0; i < count; i++) {
                        Record r = records[offset + i];
                        wb.putInt(r.id());
                        wb.putDouble(r.value());
                        wb.put(r.name());
                    }
                    fos.write(wchunk, 0, count * RECORD_SIZE);
                    offset += count;
                }
            }
        });

        // ── read benchmark ───────────────────────────────────────────
        BenchmarkRunner.run("serialization_read", () -> {
            try (FileInputStream fis = new FileInputStream(tmp.toFile())) {
                for (int i = 0; i < N; i++) {
                    int read = 0;
                    while (read < RECORD_SIZE) {
                        int r = fis.read(recBuf, read, RECORD_SIZE - read);
                        if (r < 0) throw new EOFException();
                        read += r;
                    }
```

```
                    ByteBuffer bb =
ByteBuffer.wrap(recBuf).order(ByteOrder.LITTLE_ENDIAN);
                    int    id    = bb.getInt();
                    double value = bb.getDouble();
                    byte[] name  = new byte[16];
                    bb.get(name);
                    readBack[i] = new Record(id, value, name);
                }
            }
        });

        Files.deleteIfExists(tmp);

        System.out.printf("VERIFY first: id=%d  value=%.6f  name=%s%n",
            readBack[0].id(), readBack[0].value(), trimName(readBack[0].name()));
        System.out.printf("VERIFY last:  id=%d  value=%.6f  name=%s%n%n",
            readBack[N-1].id(), readBack[N-1].value(), trimName(readBack[N-
1].name()));
    }

    private static String trimName(byte[] name) {
        int len = name.length;
        while (len > 0 && (name[len-1] == 0 || name[len-1] == ' ')) len--;
        return new String(name, 0, len);
    }
}
```

Лістинг А.23 – Серіалізація масиву структур у бінарний файл (C#)

```
using System;
using System.IO;
using System.Runtime.InteropServices;
using System.Text;

public static class IoSerialization
{
    // Matches structs_1M.bin layout: int32 + float64 + char[16] = 28 bytes (no
padding)
    [StructLayout(LayoutKind.Sequential, Pack = 1)]
    private unsafe struct Record
    {
        public int    Id;
        public double Value;
        public fixed byte Name[16];
    }

    private const int N          = 1_000_000;
    private const int RecordSize  = 28; // 4 + 8 + 16
```

```
    private static string TmpPath() =>
        Path.Combine(Path.GetTempPath(), "bench_ser_csharp.bin");

    public static unsafe void Run()
    {
        const string srcPath = "../input_data/structs_1M.bin";
        string tmp = TmpPath();

        // Read source records
        Record[] records = new Record[N];
        using (var fs = new FileStream(srcPath, FileMode.Open, FileAccess.Read))
        {
            byte[] buf = new byte[RecordSize];
            for (int i = 0; i < N; i++)
            {
                fs.ReadExactly(buf);
                fixed (byte* p = buf)
                {
                    records[i] = *(Record*)p;
                }
            }
        }

        Record[] readBack = new Record[N];

        // ── write benchmark ──────────────────────────────────────────
        BenchmarkRunner.Run("serialization_write", () =>
        {
            using var bw = new BinaryWriter(
                new FileStream(tmp, FileMode.Create, FileAccess.Write,
FileShare.None, 65536));
            for (int i = 0; i < N; i++)
            {
                bw.Write(records[i].Id);
                bw.Write(records[i].Value);
                fixed (byte* p = records[i].Name)
                    for (int b = 0; b < 16; b++) bw.Write(p[b]);
            }
        });

        // ── read benchmark ───────────────────────────────────────────
        BenchmarkRunner.Run("serialization_read", () =>
        {
            using var br = new BinaryReader(
                new FileStream(tmp, FileMode.Open, FileAccess.Read, FileShare.Read,
65536));
            for (int i = 0; i < N; i++)
            {
                readBack[i].Id    = br.ReadInt32();
                readBack[i].Value = br.ReadDouble();
```

```
                fixed (byte* p = readBack[i].Name)
                    for (int b = 0; b < 16; b++) p[b] = br.ReadByte();
            }
        });

        File.Delete(tmp);

        string n0 = TrimName(ref readBack[0]);
        string nL = TrimName(ref readBack[N - 1]);
        Console.WriteLine($"VERIFY first: id={readBack[0].Id}
 value={readBack[0].Value:F6}  name={n0}");
        Console.WriteLine($"VERIFY last:  id={readBack[N-1].Id}  value={readBack[N-
1].Value:F6}  name={nL}\n");
    }

    private static unsafe string TrimName(ref Record r)
    {
        fixed (byte* p = r.Name)
        {
            int len = 0;
            while (len < 16 && p[len] != 0) len++;
            return Encoding.ASCII.GetString(p, len);
        }
    }
}
```

Лістинг А.24 – Серіалізація масиву структур у бінарний файл (CPython)

```
import os
import struct
import tempfile
from benchmark_runner import run_benchmark

# Matches generate_inputs.py: struct.pack('=id16s', id, value, name)
RECORD_FMT  = '=id16s'
RECORD_SIZE = struct.calcsize(RECORD_FMT)  # 28 bytes
N           = 1_000_000


def run() -> None:
    src_path = '../input_data/structs_1M.bin'
    tmp      = os.path.join(tempfile.gettempdir(), 'bench_ser_python.bin')

    # Read source records once
    print('Loading structs_1M.bin ...')
    records: list[tuple] = []
    with open(src_path, 'rb') as f:
        raw = f.read(N * RECORD_SIZE)
    offset = 0
```

```
    for _ in range(N):
        rec = struct.unpack_from(RECORD_FMT, raw, offset)
        records.append(rec)
        offset += RECORD_SIZE
    del raw

    read_back: list[tuple] = []

    # ── write benchmark ──────────────────────────────────────────
    def write_task():
        with open(tmp, 'wb') as f:
            for rec in records:
                f.write(struct.pack(RECORD_FMT, *rec))

    run_benchmark('serialization_write', write_task)

    # ── read benchmark ───────────────────────────────────────────
    def read_task():
        nonlocal read_back
        read_back = []
        with open(tmp, 'rb') as f:
            raw = f.read()
        off = 0
        for _ in range(N):
            read_back.append(struct.unpack_from(RECORD_FMT, raw, off))
            off += RECORD_SIZE

    run_benchmark('serialization_read', read_task)

    os.remove(tmp)

    def trim(b: bytes) -> str:
        return b.rstrip(b'\x00').decode('ascii', errors='replace')

    print(f'VERIFY first: id={read_back[0][0]}  value={read_back[0][1]:.6f}
 name={trim(read_back[0][2])}')
    print(f'VERIFY last:  id={read_back[-1][0]}  value={read_back[-1][1]:.6f}
 name={trim(read_back[-1][2])}\n')
```

Лістинг А.25 – Відкриття та закриття файлових дескрипторів (C++)

```
#include "benchmark_base.h"
#include <cstdio>
#include <iostream>

#ifdef _WIN32
#  define WIN32_LEAN_AND_MEAN
#  include <windows.h>
static std::string tmp_path() {
```

```
    char buf[MAX_PATH];
    GetTempPathA(MAX_PATH, buf);
    return std::string(buf) + "bench_fd.bin";
}
#else
static std::string tmp_path() { return "/tmp/bench_fd.bin"; }
#endif

void run_sys_file_descriptors() {
    constexpr int N = 100'000;
    const std::string tmp = tmp_path();

    // Create the file once
    { std::FILE* f = std::fopen(tmp.c_str(), "wb"); if (f) std::fclose(f); }

    int success_count = 0;

    run_benchmark("file_descriptors", /*warmup=*/1, /*iterations=*/30, [&]() {
        success_count = 0;
        for (int i = 0; i < N; ++i) {
            std::FILE* f = std::fopen(tmp.c_str(), "rb");
            if (f) { std::fclose(f); ++success_count; }
        }
    });

    std::remove(tmp.c_str());
    std::cout << "VERIFY: successful open/close=" << success_count
              << "  (expected " << N << ")\n\n";
}
```

Лістинг А.26 – Відкриття та закриття файлових дескрипторів (Java)

```
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

import java.io.*;
import java.nio.file.*;

public class SysFileDescriptors {

    private static Path tmpPath() {
        return Paths.get(System.getProperty("java.io.tmpdir"), "bench_fd_java.bin");
    }

    public static void run() throws Exception {
        final int  N   = 100_000;
        final Path tmp = tmpPath();
```

```
        // Create the file once
        Files.createFile(tmp);

        int[] success = {0};

        BenchmarkRunner.run("file_descriptors", () -> {
            success[0] = 0;
            for (int i = 0; i < N; i++) {
                try (FileInputStream fis = new FileInputStream(tmp.toFile())) {
                    success[0]++;
                }
            }
        });

        Files.deleteIfExists(tmp);
        System.out.printf("VERIFY: successful open/close=%d  (expected %d)%n%n",
                          success[0], N);
    }
}
```

Лістинг А.27 – Відкриття та закриття файлових дескрипторів (C#)

```
using System;
using System.IO;

public static class SysFileDescriptors
{
    private static string TmpPath() =>
        Path.Combine(Path.GetTempPath(), "bench_fd_csharp.bin");

    public static void Run()
    {
        const int N   = 100_000;
        string    tmp = TmpPath();

        // Create the file once
        File.WriteAllBytes(tmp, Array.Empty<byte>());

        int success = 0;

        BenchmarkRunner.Run("file_descriptors", () =>
        {
            success = 0;
            for (int i = 0; i < N; i++)
            {
                using var fs = File.Open(tmp, FileMode.Open, FileAccess.Read);
                success++;
            }
        });
```

```
        File.Delete(tmp);
        Console.WriteLine($"VERIFY: successful open/close={success}  (expected
{N})\n");
    }
}
```

Лістинг А.28 – Відкриття та закриття файлових дескрипторів (CPython)

```
import os
import tempfile
from benchmark_runner import run_benchmark

N = 100_000


def run() -> None:
    tmp = os.path.join(tempfile.gettempdir(), 'bench_fd_python.bin')
    open(tmp, 'wb').close()

    success = [0]

    def task():
        success[0] = 0
        for _ in range(N):
            f = open(tmp, 'rb')
            f.close()
            success[0] += 1

    run_benchmark('file_descriptors', task)

    os.remove(tmp)
    print(f'VERIFY: successful open/close={success[0]}  (expected {N})\n')
```

Лістинг А.29 – Послідовне створення та завершення потоків (C++)

```
#include "benchmark_base.h"
#include <thread>
#include <atomic>
#include <iostream>

void run_sys_threads() {
    std::atomic<int> counter{0};

    run_benchmark("threads", /*warmup=*/1, /*iterations=*/30, [&]() {
        counter.store(0, std::memory_order_relaxed);
        for (int i = 0; i < 1000; ++i) {
            std::thread t([&]() {
```

```
                counter.fetch_add(1, std::memory_order_relaxed);
            });
            t.join();
        }
    });

    std::cout << "VERIFY: counter=" << counter.load()
              << "  (expected 1000)\n\n";
}
```

Лістинг А.30 – Послідовне створення та завершення потоків (Java)

```
package benchmark.benchmarks;

import benchmark.BenchmarkRunner;

import java.util.concurrent.atomic.AtomicInteger;

public class SysThreads {

    public static void run() throws Exception {
        AtomicInteger counter = new AtomicInteger(0);

        BenchmarkRunner.run("threads", () -> {
            counter.set(0);
            for (int i = 0; i < 1000; i++) {
                Thread t = new Thread(() -> counter.incrementAndGet());
                t.start();
                t.join();
            }
        });

        System.out.printf("VERIFY: counter=%d  (expected 1000)%n%n", counter.get());
    }
}
```

Лістинг А.31 – Послідовне створення та завершення потоків (C#)

```
using System;
using System.Threading;

public static class SysThreads
{
    public static void Run()
    {
        int counter = 0;
```

```
        BenchmarkRunner.Run("threads", () =>
        {
            Interlocked.Exchange(ref counter, 0);
            for (int i = 0; i < 1000; i++)
            {
                var t = new Thread(() => Interlocked.Increment(ref counter));
                t.Start();
                t.Join();
            }
        });

        Console.WriteLine($"VERIFY: counter={counter}  (expected 1000)\n");
    }
}
```

Лістинг А.32 – Послідовне створення та завершення потоків (CPython)

```
import threading
from benchmark_runner import run_benchmark


def run() -> None:
    counter = [0]
    lock    = threading.Lock()

    def worker():
        with lock:
            counter[0] += 1

    def task():
        counter[0] = 0
        threads = []
        for _ in range(1000):
            t = threading.Thread(target=worker)
            t.start()
            t.join()

    run_benchmark('threads', task)
    print(f'VERIFY: counter={counter[0]}  (expected 1000)\n')
```