

ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
Навчально-науковий інститут економіки, управління, права та
інформаційних технологій
Кафедра інформаційних систем та технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавр

на тему: **«Порівняння підходів до паралельного програмування на загальній пам'яті (TPL/OpenMP) та розподіленій пам'яті (MPI) на прикладі класичних обчислювальних задач»**

Виконав: здобувач вищої освіти
за освітньою програмою
Інформаційні управляючі системи
спеціальності 126 Інформаційні
системи та технології
ступеня вищої освіти бакалавр
групи 126ІСТ_бд_2022
Насоненко Олександр Вячеславович
Керівник: Поночовний Юрій
Леонідович
Рецензент: Муравльов Володимир
Вячеславович

Полтава – 2026 року

ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
Навчально-науковий інститут економіки, управління, права та
інформаційних технологій
Кафедра інформаційних систем та технологій

Освітня програма Інформаційні управляючі системи
Спеціальність 126 Інформаційні системи та технології
Рівень вищої освіти перший (бакалаврський)

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ **Юрій УТКІН**
«10» червня 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ

Насоненку Олександрю Вячеславовичу

1. Тема роботи:
«Порівняння підходів до паралельного програмування на загальній пам'яті (TPL/OpenMP) та розподіленій пам'яті (MPI) на прикладі класичних обчислювальних задач»,
керівник роботи д.т.н., професор, професор кафедри інформаційних систем та технологій Поночовний Юрій Леонідович.
Затверджено наказом закладу вищої освіти від «10» квітня 2026 року № 383 ст.
2. Строк подання здобувачем вищої освіти роботи «08» червня 2026 року.
3. Вихідні дані до роботи: стандарти та специфікації паралельного програмування (стандарт OpenMP 4.5, стандарт MPI-4.1), дані офіційних сайтів <https://www.openmp.org/>, <https://www.mpi-forum.org/>, <https://learn.microsoft.com/> (для MS-MPI та TPL), <https://mpi4py.readthedocs.io/> (для mpi4py), <https://docs.python.org/> (для Python); середовища розробки та виконання: Dev-C++ 6.3 з компілятором TDM-GCC 64-bit, Visual Studio Code з .NET SDK, Microsoft MPI 10.1.3, Python 3.11 з бібліотеками mpi4py та numpy.
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Розділ 1. Теоретичні основи паралельних обчислень та моделей пам'яті
Розділ 2. Реалізація паралельних алгоритмів засобами TPL, OpenMP та MPI
Розділ 3. Програмна реалізація та обґрунтування результатів дослідження
5. Перелік графічного матеріалу: схеми, рисунки, графіки, діаграми за темою та об'єктом дослідження.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання отримав
Оцінювання економічної ефективності результатів дослідження	Дивнич О. Д., к. е. н., доцент, доцент кафедри економіки та публічного управління	01.04.2026 р.	29.05.2026 р.

7. Дата видачі завдання «10» червня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Вибір і затвердження теми роботи	02.06.2025 р. – 04.06.2025	
2	Складання і затвердження розгорнутого плану та завдання на кваліфікаційну роботу	05.06.2025 р. – 10.06.2025 р.	
3	Опрацювання джерел інформації	11.06.2025 р. – 15.06.2025 р.	
4	Збір, вивчення і обробка інформації, необхідної для виконання роботи	16.06.2025 р. – 20.06.2025 р.	
5	Виконання теоретико-методологічного розділу роботи	08.09.2025 р. – 01.11.2025 р.	
6	Виконання дослідницько-аналітичного розділу роботи	19.01.2026 р. – 14.02.2026 р.	
7	Виконання проєктно-рекомендаційного розділу роботи	16.02.2026 р. – 31.03.2026 р.	
8	Оцінювання економічної ефективності результатів дослідження	01.04.2026 р. – 29.05.2026 р.	
9	Оформлення тексту роботи	01.06.2026 р. – 06.06.2026р.	
10	Попередній захист роботи на кафедрі	08.06.2026 р.	
11	Доопрацювання роботи з урахуванням зауважень і пропозицій	09.06.2026 р. – 10.06.2026 р.	
12	Нормоконтроль	11.06.2026 р. – 15.06.2026 р.	
13	Захист кваліфікаційної роботи	16.06.2026 р. - 18.06.2026 р.	

Здобувач вищої освіти

Олександр НАСОНЕНКО

Керівник роботи

Юрій ПОНОЧОВНИЙ

ЗМІСТ

ВСТУП	6
РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА МОДЕЛЕЙ ПАМ'ЯТІ.....	9
1.1 Архітектури паралельних обчислювальних систем і класифікація підходів до розпаралелювання.....	9
1.2 Модель паралелізму на загальній пам'яті і технології її реалізації	12
1.3 Модель паралелізму на розподіленій пам'яті і стандарт MPI	15
1.4 Огляд класичних обчислювальних задач для порівняльного аналізу .	19
РОЗДІЛ 2 РЕАЛІЗАЦІЯ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ ЗАСОБАМИ TPL, OPENMP ТА MPI	23
2.1 Середовище розробки та методика вимірювання продуктивності в умовах псевдопаралельного виконання.....	23
2.2 Реалізація задач засобами TPL та OpenMP (загальна пам'ять)	26
2.3 Реалізація задач засобами MS-MPI та mpi4py (розподілена пам'ять) .	29
2.4 Порівняльний аналіз результатів вимірювань	32
РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОБҐРУНТУВАННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ	35
3.1 Аналіз отриманих залежностей і практичні рекомендації щодо вибору моделі розпаралелювання.....	35
3.2 Опис програмного комплексу та інструкція користувача	38
3.3 Техніко-економічне обґрунтування ефективності розробленої системи	42
ВИСНОВКИ.....	46
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	48
ДОДАТКИ.....	53

ВСТУП

Актуальність. Сучасний етап розвитку комп'ютерних технологій характеризується повсюдним переходом до багатоядерних процесорів: навіть бюджетний ноутбук сьогодні містить від двох до восьми обчислювальних ядер, а серверні платформи налічують десятки і сотні ядер. Проте здатність ефективно використовувати ці ресурси залежить не від апаратного забезпечення, а від застосованої моделі паралельного програмування. За даними аналітичних оглядів [1, 2], значна частина прикладного програмного забезпечення досі виконується у переважно послідовному режимі, не використовуючи потенціал наявних ядер, що призводить до неефективного використання апаратних ресурсів і збільшення часу виконання обчислювально інтенсивних задач.

Паралельне програмування реалізується двома принципово різними підходами: на основі загальної пам'яті (OpenMP для C++ та TPL для C#/.NET) та на основі розподіленої пам'яті з передаванням повідомлень (стандарт MPI). Кожен підхід має свої переваги, обмеження і типові сфери застосування, однак систематичного кількісного порівняння їх продуктивності на одних і тих самих задачах в умовах обмеженого апаратного паралелізму – зокрема на персональних комп'ютерах із псевдопаралельним режимом виконання – у доступній літературі недостатньо [3, 4]. Це формує актуальну науково-практичну проблему: як обрати оптимальний підхід до розпаралелювання, не маючи доступу до суперкомп'ютерного кластера, і як оцінити реальну ефективність кожного підходу в умовах звичайного офісного ноутбука.

Метою кваліфікаційної роботи є порівняльний аналіз підходів до паралельного програмування на загальній пам'яті (TPL/OpenMP) та розподіленій пам'яті (MPI) шляхом реалізації та вимірювання продуктивності класичних обчислювальних задач на платформі з обмеженим апаратним паралелізмом і формулювання практичних рекомендацій щодо вибору моделі розпаралелювання.

Відповідно до мети роботи необхідно вирішити наступні *завдання*:

1. Дослідити теоретичні основи паралельних обчислень: класифікацію архітектур, моделі загальної та розподіленої пам'яті, технології OpenMP, TPL та MPI, а також закони Амдала і Густафсона–Барсіса як аналітичний інструментарій оцінки прискорення;
2. Обрати та обґрунтувати набір класичних обчислювальних задач (пошук простих чисел, множення матриць, обчислення числа π методом Монте-Карло), що охоплюють різні шаблони паралелізму;
3. Реалізувати дванадцять паралельних програм засобами OpenMP (C++), TPL (C#), MS-MPI (C++) та mpi4py (Python) і провести вимірювання прискорення та ефективності при кількості потоків/процесів 1, 2, 4 та 8;
4. Провести порівняльний аналіз отриманих результатів, виявити закономірності поведінки кожного підходу залежно від класу задачі та режиму виконання;
5. Сформулювати практичні рекомендації щодо вибору моделі розпаралелювання та виконати техніко-економічне обґрунтування ефективності розробленого програмного комплексу.

Об'єкт дослідження – процеси паралельного виконання обчислювальних задач на багатоядерних процесорах в умовах загальної та розподіленої пам'яті.

Предмет дослідження – порівняльна продуктивність підходів TPL, OpenMP, MS-MPI та mpi4py на класичних обчислювальних задачах в умовах псевдопаралельного виконання на офісному ноутбуці.

Методи досліджень – дослідження базується на методах порівняльного аналізу програмних систем, вимірювання продуктивності (бенчмаркінгу) з медіанним усередненням результатів, обчислення показників прискорення та ефективності згідно із законом Амдала, а також на методах декомпозиції даних і паралельного програмування із застосуванням моделей fork-join та передавання повідомлень.

Інформаційна база – наукові статті та монографії з паралельного програмування, офіційна документація стандартів OpenMP і MPI, технічна

документація Microsoft щодо TPL та MS-MPI, документація бібліотеки mpi4py, а також результати власних вимірювань продуктивності, отримані в ході виконання роботи [5, 6, 7].

Практична значущість – розроблений програмний комплекс із дванадцяти паралельних реалізацій та скрипту автоматичної візуалізації результатів може бути використаний як навчальний стенд для дослідження паралельних алгоритмів. Сформована матриця вибору підходу до розпаралелювання і кількісно підтверджені рекомендації дозволяють розробникам-початківцям обґрунтовано обирати між TPL, OpenMP та MPI залежно від типу задачі, мови програмування та доступного апаратного середовища без необхідності доступу до спеціалізованої обчислювальної інфраструктури.

Результати роботи апробовані в рамках наукової конференції здобувачів вищої освіти за результатами науково-дослідної роботи у 2025-2026 рр.

Структура кваліфікаційної роботи логічно пов'язана з задачами досліджень і містить перелік умовних позначень, вступ, три розділи основної частини, висновки, список використаних джерел. Загальний обсяг текстової частини кваліфікаційної роботи складає 52 сторінки формату A4. Вона містить 12 рисунків і 13 таблиць.

РОЗДІЛ 1

ТЕОРЕТИЧНІ ОСНОВИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА МОДЕЛЕЙ ПАМ'ЯТІ

1.1 Архітектури паралельних обчислювальних систем і класифікація підходів до розпаралелювання

З розвитком мікроелектроніки та переходом до багатоядерних процесорів паралельні обчислення перестали бути привілеєм суперкомп'ютерних центрів і стали невід'ємною частиною звичайного програмного забезпечення. Сучасний персональний комп'ютер є паралельною системою – навіть офісний ноутбук із двоядерним процесором здатний одночасно виконувати кілька незалежних потоків інструкцій. Для ефективного використання таких можливостей розробнику необхідно розуміти класифікацію архітектур паралельних систем і відповідні моделі організації пам'яті [1].

Загальноприйнятою основою для класифікації паралельних обчислювальних систем є таксономія Флінна, запропонована ще у 1966 р. і розширена у 1972 р. [2]. Відповідно до неї, архітектуру системи визначає поєднання кількості потоків команд та потоків даних, що обробляються одночасно. Виокремлюють чотири класи, які наведено у таблиці 1.1.

Таблиця 1.1 – Класифікація архітектур паралельних систем за таксономією Флінна

Клас	Потоки команд	Потоки даних	Приклади
SISD	Один	Один	Однопроцесорний ПК (послідовна архітектура)
SIMD	Один	Багато	Векторні процесори, GPU, AVX-інструкції
MISD	Багато	Один	Систолічні та конвеєрні системи
MIMD	Багато	Багато	Багатоядерні CPU, кластери, хмарні вузли

Практичний інтерес для задач даної роботи представляє клас MIMD, оскільки саме до нього належить переважна більшість сучасних паралельних

систем [2]. Усередині класу MIMD принципово важливою є класифікація за організацією пам'яті, запропонована Джонсоном (рисунок 1.1). Вона поділяє системи на два типи: з загальною (розподіленою) пам'яттю (GMSV, Global Memory – Shared Variables) та з розподіленою пам'яттю (DMMP, Distributed Memory – Message Passing) [3].

У системах GMSV усі процесорні ядра мають прямий доступ до єдиного адресного простору. Таке рішення спрощує програмування – обмін даними між потоками відбувається через спільні змінні – проте породжує потребу в синхронізації (м'ютекси, атомарні операції) для уникнення перегонових станів. Масштабованість таких систем обмежена: при великій кількості ядер виникає конкуренція за шину пам'яті [3]. Саме на цій моделі базуються технології OpenMP та TPL (Task Parallel Library).

Системи DMMP, навпаки, будуються з вузлів із локальною пам'яттю, де кожен процес не має безпосереднього доступу до пам'яті іншого. Обмін здійснюється явним пересиланням повідомлень. Це підвищує масштабованість і усуває проблему доступу до спільної пам'яті, але накладає на розробника відповідальність за декомпозицію даних і комунікаційні патерни [3]. Стандарт MPI (Message Passing Interface) є де-факто еталонним інструментом для програмування систем із розподіленою пам'яттю.



Рисунок 1.1 – Класифікація Флінна та Джонсона: зв'язок між типами паралелізму та моделями пам'яті

Важливим аспектом є також класифікація режимів виконання паралельних задач. Режим розподілу часу (псевдопаралелізм) передбачає, що кількість потоків перевищує кількість фізичних ядер: ОС по чергово надає кожному потоку квант процесорного часу. При цьому реального одночасного виконання немає, проте проявляються усі ефекти паралелізму – необхідність синхронізації, взаємного виключення, обробки станів гонитви. Цей режим є саме тим, у якому проводитиметься дослідження даної роботи, оскільки платформою слугує офісний ноутбук з обмеженою кількістю ядер [1]. Справжній паралелізм (синхронні паралельні обчислення) реалізується тоді, коли кількість задіяних ядер відповідає кількості потоків, і кожен потік виконується фізично незалежно.

Для оцінки теоретичної ефективності паралельних алгоритмів використовують два фундаментальних закони. Закон Амдала (1967) визначає верхню межу прискорення S при фіксованому розмірі задачі. Якщо α – частка послідовних операцій алгоритму, а p – кількість процесорів, то:

$$S = \frac{1}{\alpha + \frac{1-\alpha}{p}} \leq \frac{1}{\alpha}. \quad (1.1)$$

Це означає, що навіть нескінченне збільшення кількості процесорів не дає прискорення, більшого за $1/\alpha$. Наприклад, якщо 10% коду є послідовним, максимальне прискорення не перевищить 10 разів, незалежно від кількості ядер [4].

Закон Густафсона–Барсіса (1988) пропонує інший погляд: при збільшенні кількості ядер користувачі схильні збільшувати обсяг задачі, залишаючи час виконання сталим. У такому формулюванні масштабоване прискорення:

$$S_p = p + (1 - p), \quad (1.2)$$

зростає лінійно з кількістю процесорів p , і паралельна частина домінує над послідовною. Таким чином, закон Амдала є більш актуальним при дослідженні

прискорення на фіксованій задачі (що відповідає умовам цієї роботи), тоді як закон Густафсона краще відображає практику наукових обчислень із масштабованим навантаженням [4].

Узагальнюючи, вибір між моделлю загальної та розподіленої пам'яті визначається не лише архітектурою цільової системи, а й характером самої задачі: ступенем її декомпозиції, інтенсивністю обміну даними між підзадачами та допустимими накладними витратами на синхронізацію або комунікацію [3]. Саме цей вибір є центральним предметом дослідження даної роботи.

1.2 Модель паралелізму на загальній пам'яті і технології її реалізації

Модель паралельного програмування на загальній пам'яті є природним способом використання ресурсів багатоядерного процесора, де всі обчислювальні ядра мають спільний доступ до єдиного адресного простору. Ця модель передбачає, що паралельні гілки виконання – потоки (threads) – можуть безпосередньо читати та записувати спільні змінні, що спрощує обмін даними порівняно з передаванням повідомлень. Водночас спільне використання пам'яті породжує проблему узгодженості: якщо два потоки одночасно модифікують одну і ту саму змінну без координації, результат є непередбачуваним. Такий стан називають гонитвою за даними (race condition) [8].

Базовим механізмом захисту від гонитви є м'ютекс (mutex, mutual exclusion) – примітив синхронізації, який гарантує, що у будь-який момент часу лише один потік може перебувати в критичній секції – фрагменті коду, що працює зі спільним ресурсом [9]. Поряд із м'ютексом використовуються атомарні операції – неподільні операції читання-зміни-запису, що не потребують явного блокування і є ефективнішими для простих лічильників та прапорців. Неправильне використання блокувань може призвести до взаємного блокування (deadlock): ситуації, коли два або більше потоків нескінченно очікують один на

одного. Проектування з урахуванням цих ефектів є ключовою частиною розробки паралельних програм для загальної пам'яті [9].

Двома основними технологіями, що реалізують описану модель і використовуватимуться в роботі, є OpenMP та TPL (.NET Task Parallel Library). Принципова відмінність між ними полягає у рівні абстракції та цільовій мові програмування, що унаочнює таблиця 1.2.

Таблиця 1.2 – Порівняльна характеристика OpenMP та TPL як інструментів паралелізму на загальній пам'яті

Характеристика	OpenMP	TPL (.NET)
Мова	C, C++, Fortran	C#, VB.NET, F#
Механізм	Директиви препроцесора (#pragma)	Класи та методи (Parallel.For, Task)
Керування потоками	Автоматичне (fork-join)	Через пул потоків .NET ThreadPool
Синхронізація	critical, atomic, barrier	lock, Interlocked, Monitor
Рівень абстракції	Низький–середній	Середній–високий
Середовище	GCC/MSVC/Clang	.NET Framework / .NET 6+

OpenMP (Open Multi-Processing) – це стандарт, що підтримується провідними виробниками апаратного та програмного забезпечення і реалізований у компіляторах GCC, Clang, MSVC та Intel C++ [10]. Поточна версія стандарту – 6.0, яка розширює підтримку гетерогенних обчислень і AI-навантажень. Програміст позначає секції коду для паралельного виконання за допомогою директив #pragma omp, решту бере на себе виконавче середовище. Модель виконання OpenMP будується на принципі fork-join: головний потік («майстер») породжує команду робочих потоків на початку паралельної секції, а після її завершення всі потоки збираються назад. Найуживанішою конструкцією є розпаралелювання циклу:

```
#pragma omp parallel for reduction(+:sum) schedule(static)
for (int i = 0; i < N; i++) { sum += f(i); }
```

Тут клауза `reduction` автоматично створює приватну копію змінної `sum` для кожного потоку та об'єднує їх після циклу, уникаючи гонитви. Клауза `schedule(static)` рівномірно розподіляє ітерації між потоками. Ключовою перевагою OpenMP є інкрементальне розпаралелювання: послідовний код залишається працездатним, якщо директиви ігнорувати (або компілювати без прапорця `-fopenmp`), що суттєво спрощує відлагодження [10].

TPL (Task Parallel Library) – бібліотека, що входить до складу .NET починаючи з версії 4.0, і є рекомендованим способом написання паралельного коду на платформі Microsoft [11]. Центральним поняттям TPL є завдання (Task) – одиниця роботи, яку виконавче середовище ставить у чергу пулу потоків і диспетчеризує по доступних ядрах. Клас `Parallel` надає зручні методи для паралельних циклів – `Parallel.For` та `Parallel.ForEach`, а також `Parallel.Invoke` для одночасного запуску довільного набору дій. На відміну від OpenMP, TPL не потребує спеціального прапорця компілятора і природно поєднується з механізмом `async/await` для асинхронного програмування [12]. Бібліотека автоматично масштабує кількість потоків відповідно до навантаження системи і не вимагає від розробника явного визначення кількості ядер.

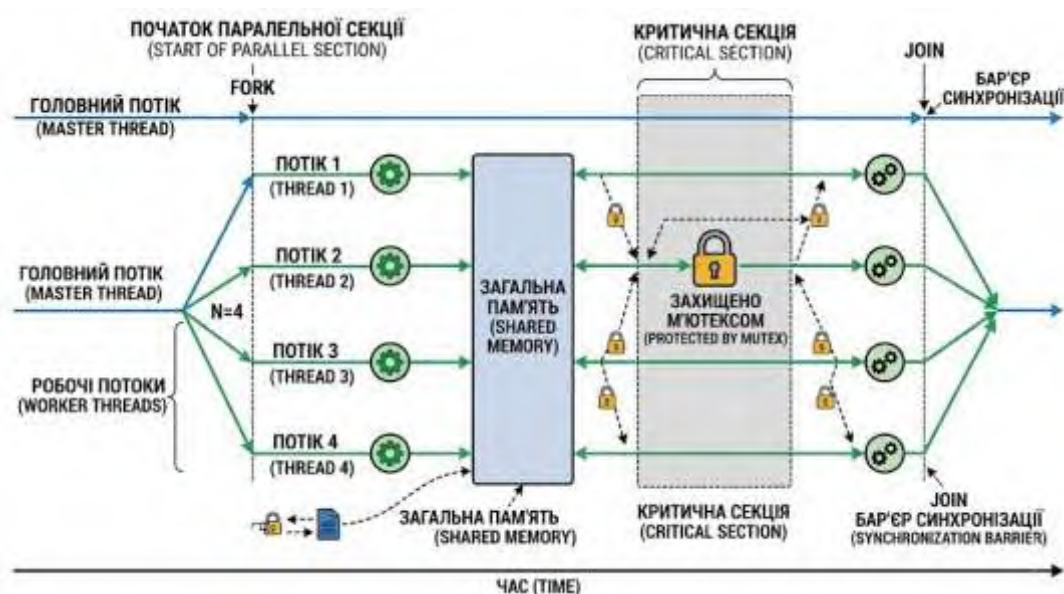


Рисунок 1.2 – Модель fork-join та взаємодія потоків у моделі загальної пам'яті

Спільним для обох технологій є базовий принцип вимірювання ефективності. Прискорення S при використанні p потоків визначається як:

$$S = \frac{T_1}{T_p}, \quad (1.3)$$

де T_1 – час виконання послідовного алгоритму,

T_p – час виконання паралельного алгоритму на p потоках.

В умовах псевдопаралелізму (один або два фізичних ядра) значення S для задач з великими накладними витратами на синхронізацію може бути меншим за 1, що свідчить про неефективність розпаралелювання для даного класу задач. Саме такі ситуації і становлять науковий інтерес у даній роботі [7].

Отже, OpenMP та TPL реалізують одну й ту саму концептуальну модель – паралелізм на загальній пам'яті – але на різних мовах і з різним ступенем автоматизації. OpenMP є ближчим до апаратури і зручнішим для роботи в C++, тоді як TPL пропонує вищий рівень абстракції в межах екосистеми .NET. Порівняння їх продуктивності на одних і тих самих обчислювальних задачах є одним із ключових завдань практичного розділу.

1.3 Модель паралелізму на розподіленій пам'яті і стандарт MPI

Модель розподіленої пам'яті є альтернативним підходом до організації паралельних обчислень, принципово відмінним від моделі загальної пам'яті, розглянутої у попередньому підрозділі. У системах із розподіленою пам'яттю кожен процес має власний ізольований адресний простір і не може безпосередньо звертатися до пам'яті інших процесів. Єдиним способом обміну даними є явне передавання повідомлень (message passing): процес-відправник пакує дані в повідомлення та надсилає його адресату, який явно виконує операцію отримання. Така модель усуває проблему гонитви за даними та не

потребує механізмів синхронізації на рівні спільної пам'яті, натомість виводить на перший план накладні витрати на комунікацію між процесами [15].

Стандартом де-факто для реалізації цієї моделі є MPI (Message Passing Interface) – специфікація програмного інтерфейсу, що визначає набір функцій для комунікації між паралельними процесами [16]. Перша версія стандарту вийшла у 1994 р.; відтоді він неодноразово розвивався: MPI-3.1 (2015) додав неблокуючі колективні операції, MPI-4.0 (2021) – підтримку великих лічильників і персистентні колективи, MPI-4.1 (2023) – уточнення та корекції, а MPI-5.0 (2025) – стандартизований бінарний інтерфейс ABI [16]. Переваги MPI – висока переносимість, масштабованість від одного ПК до суперкомп'ютерних кластерів з мільйонами ядер, а також добра ізоляція процесів.

Кожен MPI-процес ідентифікується цілочисельним рангом (rank) у межах комунікатора (communicator) – об'єкта, що визначає групу процесів, які беруть участь у спільних операціях. Стандартний комунікатор MPI_COMM_WORLD охоплює всі запущені процеси. MPI підтримує два класи комунікацій [17]. Точкові операції (point-to-point) здійснюються між конкретною парою процесів: MPI_Send / MPI_Recv у блокуючому варіанті та MPI_Isend / MPI_Irecv у неблокуючому. Колективні операції залучають усі процеси в комунікаторі одночасно і є ключовим інструментом для ефективної реалізації паралельних алгоритмів. Найважливіші з них подано у таблиці 1.3.

Таблиця 1.3 – Основні колективні операції MPI та їх призначення

Операція	Функція	Призначення
Broadcast	MPI_Bcast	Розсилання даних від кореневого процесу всім
Scatter	MPI_Scatter	Розподіл масиву між процесами (по частині кожному)
Gather	MPI_Gather	Збір частин масиву з усіх процесів до кореневого
Reduce	MPI_Reduce	Глобальна редукція (сума, мін., макс.) до одного процесу
AllReduce	MPI_Allreduce	Глобальна редукція з доставкою результату всім
Barrier	MPI_Barrier	Синхронізація всіх процесів у точці бар'єру

Типова структура MPI-програми є незмінною: після ініціалізації середовища (MPI_Init) кожен процес з'ясовує свій ранг та загальну кількість

процесів, потім виконує свою частину роботи та обмінюється проміжними результатами, а наприкінці кореневий процес збирає підсумки і завершує роботу через MPI_Finalize. Саме така схема – ініціалізація, декомпозиція даних, локальне обчислення, комунікація, агрегація – становить канонічну модель паралельного алгоритму на розподіленій пам'яті [17].

Час виконання паралельної MPI-програми включає не лише обчислювальний час, а й час на комунікацію. Загальний час виконання T_p визначається як:

$$T_p = T_{comp} + T_{comm} = \frac{T_1}{p} + \alpha + \beta \cdot m, \quad (1.4)$$

де T_1 – час послідовного обчислення,

p – кількість процесів,

α – латентність передавання повідомлення (час установлення зв'язку),

β – пропускна здатність каналу зв'язку,

m – розмір повідомлення в байтах.

У випадку одного фізичного вузла (що відповідає умовам даної роботи) α і β визначаються затримкою міжпроцесної взаємодії всередині однієї операційної системи, яка може бути порівнянною за величиною з корисним обчислювальним часом для малих задач [15].

Для виконання даної роботи на платформі Windows використовується реалізація MS-MPI (Microsoft MPI) версії 10.1.3 – офіційна безкоштовна реалізація стандарту MPI від Microsoft, що підтримує Windows 10/11 і Windows Server 2016-2022 [18]. MS-MPI встановлюється у вигляді двох компонентів: виконавчого середовища (MSMpiSetup.exe) та SDK для розробників (msmpisdk.msi), після чого програми компілюються у Dev-C++ з підключенням бібліотек msmpi.lib / msmpicxx.lib. Запуск MPI-програм здійснюється командою: mpiexec -n <N> program.exe, де N – кількість процесів. В умовах одного ПК всі

процеси запускаються на тих самих фізичних ядрах, що й означає псевдопаралельний режим роботи.

Як альтернативна платформа для реалізацій із більш зручним синтаксисом використовується Python із бібліотекою `mpi4py` [19]. Бібліотека `mpi4py` надає об'єктно-орієнтований інтерфейс до MPI, максимально наближений до стандарту C/Fortran, і підтримує як передавання довільних Python-об'єктів через механізм серіалізації (`comm.send` / `comm.recv`), так і швидке передавання буферів NumPy-масивів із продуктивністю, близькою до C (`Comm.Send` / `Comm.Recv` з великої літери) [19]. Запуск здійснюється аналогічно: `mpirun -n 4 python script.py`.

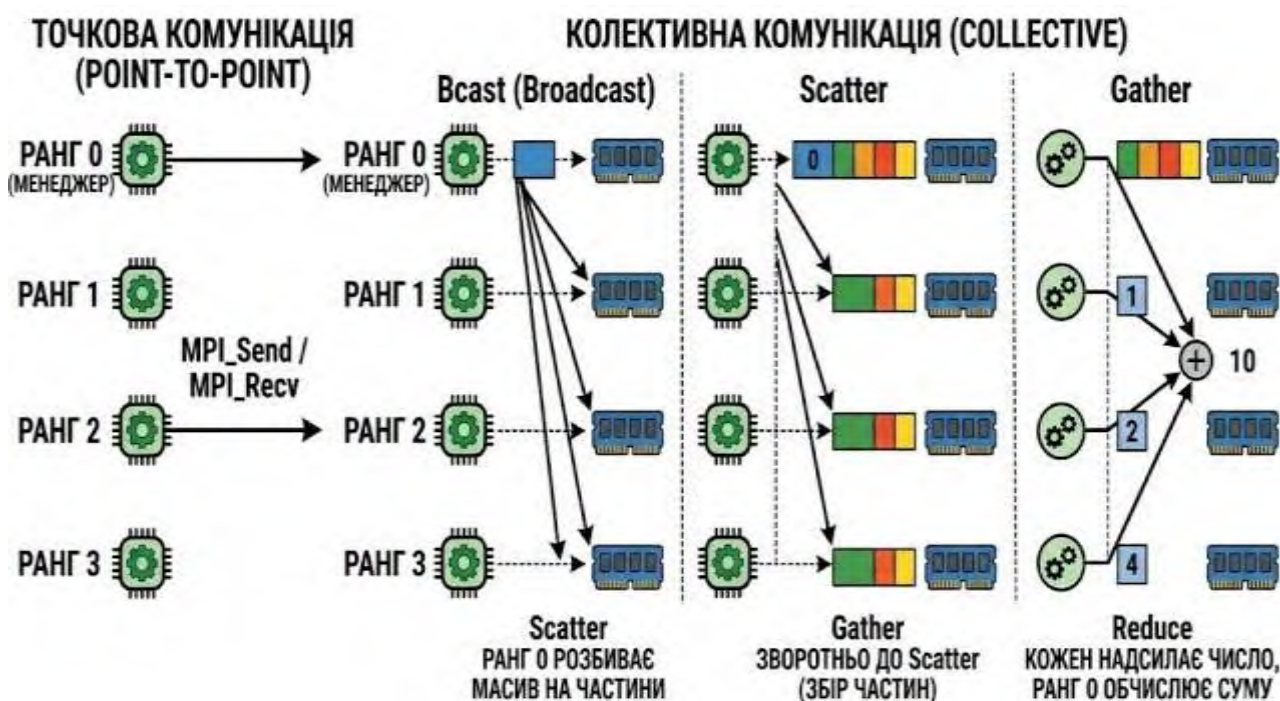


Рисунок 1.3 – Схема комунікаційних операцій MPI: точкові та колективні

Порівнюючи підходи MPI (C++/MS-MPI та Python/`mpi4py`), варто зазначити, що C++-варіант дає нижчий рівень накладних витрат за рахунок компіляції в нативний код, тоді як Python-варіант є значно зручнішим для прототипування та наочнішим для аналізу алгоритму. Обидва варіанти реалізуватимуться в практичному розділі для тих самих обчислювальних задач, що дозволить оцінити вплив мови реалізації на виміряне прискорення в умовах псевдопаралельного виконання.

1.4 Огляд класичних обчислювальних задач для порівняльного аналізу

Вибір конкретних обчислювальних задач для порівняльного аналізу паралельних підходів є критично важливим методологічним рішенням: задачі мають охоплювати різні шаблони розпаралелювання, бути достатньо простими для реалізації на кількох платформах і при цьому репрезентувати реальні класи прикладних проблем. У даній роботі для дослідження обрано три класичні задачі: пошук простих чисел методом решета Ератосфена, множення матриць та обчислення числа π методом Монте-Карло. Кожна з них є представником окремого шаблону паралелізму – data-parallel, reduction та embarrassingly parallel відповідно – що дозволяє дослідити широкий спектр поведінки алгоритмів в умовах як загальної, так і розподіленої пам'яті [22].

Першою задачею є пошук простих чисел у діапазоні від 2 до N методом решета Ератосфена. Алгоритм послідовно відсіює кратні числа для кожного знайденого простого, тобто по своїй суті є ітераційним і має виражені залежності між ітераціями: результат відсіву на кроці k залежить від усіх попередніх простих. Це робить задачу класичним прикладом data-parallel шаблону з частковими залежностями між підзадачами [23]. Паралельна реалізація можлива шляхом розподілу діапазону між потоками або процесами – кожен обробляє свій сегмент масиву – однак потребує механізму синхронізації або комунікації для передачі знайдених простих між учасниками обчислення. Саме наявність такої залежності дозволяє дослідити накладні витрати на синхронізацію в TPL/OpenMP та на комунікацію в MPI, що є одним із ключових аспектів порівняльного аналізу.

Другою задачею є множення матриць розміром $N \times N$. Це одна з найбільш вивчених паралельних задач, що є ядром багатьох чисельних методів – від розв'язання систем лінійних рівнянь до нейромережових обчислень [24]. Кожен елемент результуючої матриці C обчислюється незалежно:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj}. \quad (1.5)$$

Ця незалежність робить задачу ідеальним представником шаблону data-parallel без залежностей між підзадачами. Стандартна схема розпаралелювання – розподіл рядків матриці A між потоками або процесами – мінімізує необхідну комунікацію: кожен учасник потребує лише свою частину A та повну матрицю B . В умовах MPI це реалізується за допомогою MPI_Scatter (для розподілу A) та MPI_Bcast (для розсилання B), що дозволяє дослідити ефективність колективних операцій на практиці [25]. Порівняння прискорення цієї задачі із задачею решета дозволить кількісно оцінити вплив залежностей між підзадачами на загальну ефективність розпаралелювання.

Третьою задачею є обчислення числа π методом Монте-Карло. Метод базується на статистичній оцінці: якщо рівномірно кидати випадкові точки у квадрат зі стороною 1, то частка точок, що потрапляють у вписане коло, наближається до $\pi/4$. Оцінка числа π визначається як:

$$\pi \approx 4 \cdot \frac{M}{N}, \quad (1.6)$$

де M – кількість точок, що потрапили у коло,
 N – загальна кількість точок.

Кожна із N спроб є повністю незалежною від інших, що робить задачу класичним прикладом embarrassingly parallel шаблону – найбільш сприятливого для розпаралелювання [26]. Агрегація часткових сум M від кожного потоку або процесу виконується одноразовою операцією зведення (reduction), яка в OpenMP реалізується клаузою reduction, у TPL – через Interlocked або локальні суматори, а в MPI – через MPI_Reduce. Метод Монте-Карло широко використовується у фінансовому моделюванні, фізичних симуляціях та машинному навчанні [27], що підтверджує практичну цінність дослідження саме цієї задачі.

Для систематизації відмінностей між задачами з точки зору паралелізму у таблиці 1.4 наведено їх порівняльну характеристику за ключовими параметрами.

Таблиця 1.4 – Порівняльна характеристика обраних задач за параметрами паралелізму

Параметр	Решето Ератосфена	Множення матриць	Метод Монте-Карло
Шаблон паралелізму	Data-parallel (з залежностями)	Data-parallel (без залежностей)	Embarrassingly parallel
Залежності між підзадачами	Є (часткові)	Відсутні	Відсутні
Тип комунікації (MPI)	Point-to-point + Broadcast	Scatter + Broadcast + Gather	Reduce
Тип синхронізації (OpenMP/TPL)	Critical / lock	Відсутня	Reduction / Interlocked
Обчислювальна складність	$O(N \log \log N)$	$O(N^3)$	$O(N)$
Очікуване прискорення	Помірне	Високе	Максимально близьке до лінійного

Як видно з таблиці 1.4, задачі суттєво відрізняються за характером комунікаційного та синхронізаційного навантаження. Метод Монте-Карло є «ідеальною» паралельною задачею і слугуватиме базовою лінією для оцінки максимально досяжного прискорення в умовах конкретної платформи.

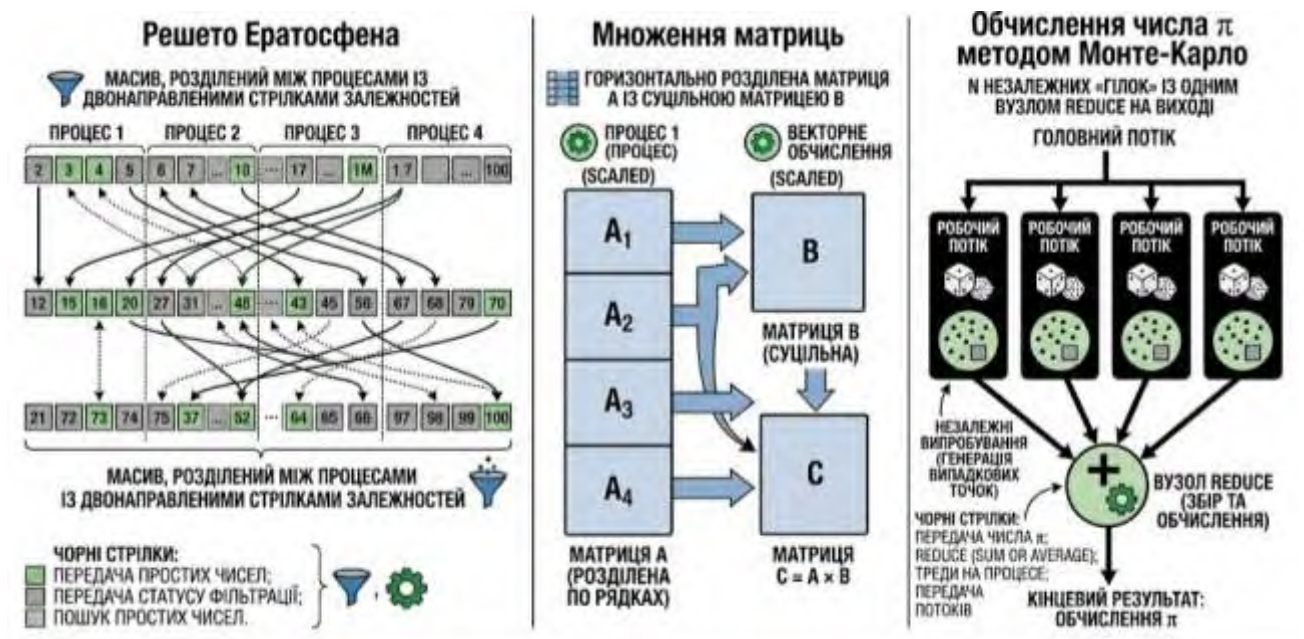


Рисунок 1.4 – Шаблони розпаралелювання трьох обраних задач

Множення матриць є репрезентативним для широкого класу інженерних задач і чутливим до пропускної здатності каналу комунікації при великих N . Решето Ератосфена, завдяки залежностям між ітераціями, дозволить виявити різницю між підходами у здатності справлятися з алгоритмічними обмеженнями на паралелізацію [22].

Таким чином, обрана сукупність задач утворює репрезентативний набір для порівняльного аналізу: вона охоплює три принципово різних шаблони паралелізму, три рівні комунікаційного навантаження та широкий діапазон очікуваних прискорень. Реалізація кожної з них засобами TPL, OpenMP та MPI дозволить отримати обґрунтовані кількісні дані для висновків про доцільність застосування кожного підходу залежно від природи задачі та доступної апаратної платформи.

Було розглянуто теоретичні засади паралельних обчислень: класифікацію архітектур за таксономією Флінна та Джонсона, закони Амдала і Густафсона–Барсіса як аналітичний інструментарій оцінки прискорення, принципи функціонування та програмні засоби моделей загальної пам'яті (OpenMP, TPL) і розподіленої пам'яті (MPI, mpi4py), а також обґрунтовано вибір трьох класичних задач – решета Ератосфена, множення матриць і методу Монте-Карло – що охоплюють різні шаблони паралелізму і забезпечують репрезентативну основу для практичного порівняльного дослідження в умовах псевдопаралельного виконання.

РОЗДІЛ 2

РЕАЛІЗАЦІЯ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ ЗАСОБАМИ TPL, OPENMP ТА MPI

2.1 Середовище розробки та методика вимірювання продуктивності в умовах псевдопаралельного виконання

Практична частина дослідження виконується на офісному ноутбуку з процесором Intel Core i5 10-го покоління (2 фізичні ядра, 4 логічних потоки завдяки технології Hyper-Threading), оперативною пам'яттю 8 ГБ DDR4 та операційною системою Windows 11. Ця конфігурація є типовою для студентського та офісного середовища і свідомо обрана як платформа дослідження: вона дозволяє виявити реальну поведінку паралельних алгоритмів в умовах обмеженого апаратного паралелізму, коли запуск 4 або більше потоків/процесів неминуче призводить до псевдопаралельного виконання з розподілом процесорного часу між учасниками [29]. Саме такі умови є найбільш поширеними на практиці для більшості розробників-початківців і формують реалістичний контекст для порівняння підходів TPL, OpenMP та MPI.

Для реалізації алгоритмів на мові C++ з підтримкою OpenMP та MS-MPI використовується інтегроване середовище розробки Dev-C++ версії 6.3 (TDM-GCC 64-bit). Компілятор TDM-GCC підтримує стандарт OpenMP 4.5 і активується прапорцем `-fopenmp` у налаштуваннях проєкту. Для підключення MS-MPI версії 10.1.3 до проєкту Dev-C++ необхідно вказати шляхи до заголовних файлів (`%MSMPI_INC%`) та статичних бібліотек (`msmpi.lib`, `msmpisxx.lib`) у розділі параметрів компілятора і лінкера відповідно [30]. MPI-програми запускаються не безпосередньо з середовища, а через командний рядок за допомогою утиліти `mpiexec`: `mpiexec -n N program.exe`; де `N` задає кількість паралельних процесів (1, 2 або 4 у даному дослідженні). Оскільки всі процеси виконуються на одному вузлі, комунікація між ними відбувається через механізм

міжпроцесної взаємодії операційної системи, а не через мережу, що мінімізує латентність порівняно з кластерним середовищем, але все одно вносить вимірювані накладні витрати [29].

Для реалізацій на Python використовується інтерпретатор Python 3.14 із бібліотеками `mpi4py`, `numpy` та стандартним модулем `multiprocessing`. Середовищем розробки слугує Visual Studio Code з розширенням Python. Запуск `mpi4py`-програм аналогічний до C++: `mpiexec -n N python script.py`. Бібліотека `multiprocessing` застосовується для реалізацій на загальній пам'яті в Python як функціональний аналог TPL/OpenMP, оскільки через глобальне блокування інтерпретатора (GIL) стандартні потоки Python не забезпечують справжнього паралелізму для CPU-навантажених задач [31]. Налаштування використовуваних середовищ та бібліотек зведено у таблиці 2.1.

Таблиця 2.1 – Конфігурація середовищ розробки та виконання

Компонент	Інструмент / Версія	Платформа	Прапорці / Команда запуску
C++ компілятор	TDM-GCC 64-bit 10.3	Dev-C++ 6.3	-fopenmp -O2
OpenMP	OpenMP 4.5 (вбудований у GCC)	C++	-fopenmp
MS-MPI	Microsoft MPI 10.1.3	C++	msmpi.lib, mpiexec -n N
Python	CPython 3.11	VS Code	–
mpi4py	3.1.x	Python	mpiexec -n N python script.py
multiprocessing	стандартна бібліотека	Python	Pool(processes=N)

Методика вимірювання продуктивності є уніфікованою для всіх реалізацій і платформ. Для кожної задачі та кожного значення кількості потоків/процесів ($N = 1, 2, 4$) виконується не менше п'яти запусків; фіксується медіанний час виконання для усунення випадкових викидів, пов'язаних із фоновою активністю ОС. Час вимірюється якомога «ближче» до обчислювального ядра: в C++ використовується `omp_get_wtime()` (для OpenMP) або `MPI_Wtime()` (для MPI), у Python – `time.perf_counter()`. Такий підхід виключає з вимірювання час ініціалізації середовища та введення-виведення даних [32].

На основі отриманих значень часу обчислюються два показники ефективності. Прискорення визначається за допомогою виразу (1.3). Ефективність використання обчислювальних ресурсів визначається як:

$$E_p = \frac{S_p}{p} \times 100\%, \quad (2.1)$$

де p – кількість задіяних потоків або процесів.

Значення $E_p < 100 \%$ свідчить про наявність накладних витрат на синхронізацію або комунікацію; у режимі псевдопаралелізму ($p >$ кількості фізичних ядер) можливе $E_p < 25 \%$, що є нормальним і очікуваним результатом [33].



Рисунок 2.1 – Схема вимірювання часу виконання паралельної програми

Усі вихідні коди організовані в єдиному каталозі проекту з підкаталогами за технологією (openmp/, mpi_cpp/, mpi_python/) та задачею (sieve/, matmul/, montecarlo/). Кожен підкаталог містить файл із послідовною реалізацією (*_seq) та паралельними варіантами, що забезпечує коректне порівняння між

послідовним і паралельним виконанням на ідентичному алгоритмічному ядрі [30]. Детальна структура програмного комплексу та приклади командного рядка описані у наступному підрозділі.

2.2 Реалізація задач засобами TPL та OpenMP (загальна пам'ять)

У даному підрозділі описано практичну реалізацію трьох обчислювальних задач із застосуванням OpenMP (C++, Dev-C++ з компілятором TDM-GCC 64-bit) та TPL (C#, VS Code, .NET 10). Усі шість програм використовують єдину методику: один прогрівний запуск без урахування результатів, після якого проводяться основні вимірювання при кількості потоків 1, 2, 4 і 8. Результати автоматично зберігаються у файли формату JSON. За базовий час T_1 приймається час послідовного виконання (1 потік), відносно якого обчислюються прискорення S та ефективність E .

Задача 1 – Пошук простих чисел (діапазон до 5000000, метод пробного ділення). В OpenMP-версії зовнішній цикл перевірки розпаралелено директивою `#pragma omp parallel for reduction(+:count) schedule(dynamic, 1000)`. Клауза *reduction* забезпечує потокобезпечне накопичення лічильника без явних блокувань, а *schedule(dynamic, 1000)* покликаний компенсувати нерівномірність навантаження – перевірка великих чисел коштує більше, ніж малих [22]. У TPL-версії використано *Parallel.For* з локальними агрегатами потоків та *Interlocked.Add* для фінальної редукції [23]. Отримані результати виявили повну відсутність прискорення для OpenMP: час виконання при 1, 2, 4 і 8 потоках склав відповідно 1942, 1933, 1961 і 1943 мс, тобто $S \approx 1,00$ у всіх випадках. TPL показала помірне прискорення: 2271,82 мс $\rightarrow$ 1306,26 мс $\rightarrow$ 1063,18 мс $\rightarrow$ 1272,62 мс при 1, 2, 4, 8 потоках, тобто $S = 1,74$ при 2 потоках і $S = 2,14$ при 4. При 8 потоках прискорення TPL знизилося до 1,79, що свідчить про насичення.

Задача 2 – Множення матриць 1024×1024 . Використано *ijk*-алгоритм із оптимізацією кешового доступу: цикл по k розміщено перед циклом по j , що забезпечує послідовний доступ до пам'яті [24]. В OpenMP-версії застосовано *#pragma omp parallel for schedule(static)* для рівномірного розподілу рядків. У TPL-версії паралельний прохід по рядках реалізовано через *Parallel.For* [23]. Ця задача показала найкраще прискорення серед трьох: OpenMP – 4000, 2173, 1686, 1629 мс при $S = 1,84/2,37/2,46$; TPL – 8878, 4731, 4069, 4218 мс при $S = 1,88/2,18/2,11$ для 2, 4, 8 потоків відповідно. Зазначимо, що TPL у режимі 1 потоку виконує задачу вдвічі повільніше за OpenMP/C++ (8879 мс проти 4000 мс), що пояснюється накладними витратами керованого середовища .NET та різницею у генерованому машинному коді.

Задача 3 – Обчислення числа π методом Монте-Карло (100 млн точок).
Оцінка числа π :

$$\pi \approx 4 \cdot \frac{N_{\text{hits}}}{N_{\text{total}}}, \quad (2.2)$$

де N_{hits} – кількість точок у чверті одиничного кола.

В OpenMP-версії для кожного потоку реалізовано власний лінійний конгруентний генератор із унікальним початковим значенням, щоб уникнути конкуренції за стан генератора [25]. У TPL-версії кожна ітерація отримує окремий екземпляр *System.Random*. Незважаючи на те, що Монте-Карло вважається «*embarrassingly parallel*» задачею з мінімальними залежностями між ітераціями [25], обидві реалізації не показали очікуваного прискорення: OpenMP – 1693, 1755, 1829, 1732 мс ($S < 1$ для всіх варіантів із кількома потоками); TPL – 2155, 2380, 1311, 1318 мс ($S = 1,64$ лише при 4 і 8 потоках, а при 2 потоках – уповільнення). Причина – вузьке місце в генерації псевдовипадкових чисел та значні накладні витрати на управління потоками відносно корисного часу кожної ітерації. Зведені результати всіх шести вимірювань подано у таблиці 2.2.

Таблиця 2.2 – Зведені результати вимірювань прискорення та ефективності (OpenMP і TPL)

Задача	Метод	Потоків	Час, мс	S	E, %
Прості числа	OpenMP	1	1942,00	1,0000	100,00
Прості числа	OpenMP	2	1933,00	1,0047	50,23
Прості числа	OpenMP	4	1961,00	0,9903	24,76
Прості числа	OpenMP	8	1943,00	0,9995	12,49
Прості числа	TPL	1	2271,82	1,0000	100,00
Прості числа	TPL	2	1306,26	1,7392	86,96
Прості числа	TPL	4	1063,18	2,1368	53,42
Прості числа	TPL	8	1272,62	1,7852	22,31
Матриці 1024 ²	OpenMP	1	4000,00	1,0000	100,00
Матриці 1024 ²	OpenMP	2	2173,00	1,8408	92,04
Матриці 1024 ²	OpenMP	4	1686,00	2,3725	59,31
Матриці 1024 ²	OpenMP	8	1629,00	2,4555	30,69
Матриці 1024 ²	TPL	1	8878,66	1,0000	100,00
Матриці 1024 ²	TPL	2	4731,17	1,8766	93,83
Матриці 1024 ²	TPL	4	4068,93	2,1821	54,55
Матриці 1024 ²	TPL	8	4217,70	2,1051	26,31
Монте-Карло	OpenMP	1	1693,00	1,0000	100,00
Монте-Карло	OpenMP	2	1755,00	0,9647	48,23
Монте-Карло	OpenMP	4	1829,00	0,9256	23,14
Монте-Карло	OpenMP	8	1732,00	0,9775	12,22
Монте-Карло	TPL	1	2154,79	1,0000	100,00
Монте-Карло	TPL	2	2380,16	0,9053	45,27
Монте-Карло	TPL	4	1310,77	1,6439	41,10
Монте-Карло	TPL	8	1317,70	1,6353	20,44

Графічне порівняння прискорення для всіх задач і методів наведено на рисунку 2.2.

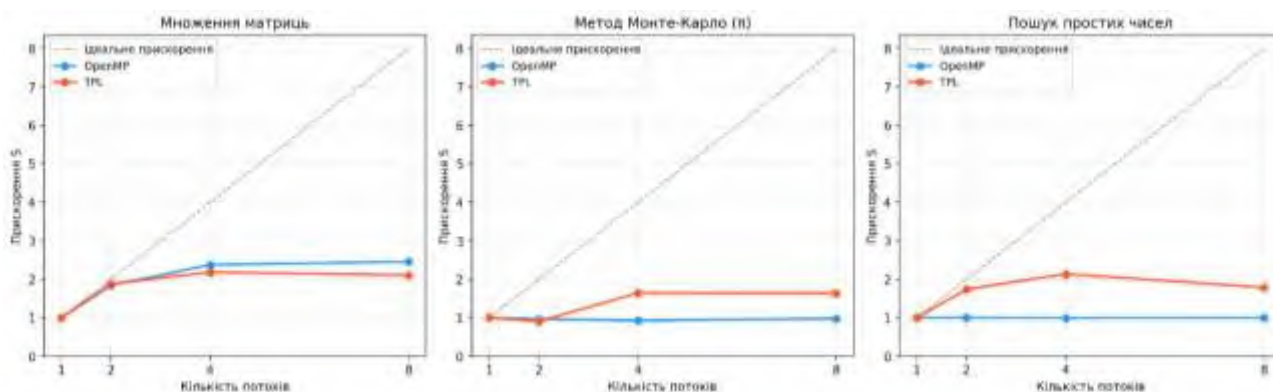


Рисунок 2.2 – Прискорення паралельних алгоритмів (OpenMP vs TPL) для трьох задач

Загальна тенденція, що простежується в даних таблиці 2.1, реальне прискорення на офісному ноутбучі суттєво поступається теоретичному. Найбільш виражений ефект дає множення матриць: OpenMP досягає $S = 2,46$ при 8 потоках, TPL – $S = 2,19$ при 4. Задача пошуку простих чисел виявилася несподівано: TPL показала помірне прискорення, тоді як OpenMP з динамічним плануванням у цій конфігурації не дав жодного виграшу – накладні витрати на управління потоками в режимі псевдопаралелізму перевищили корисний ефект. Детальний аналіз цих результатів та їх порівняння з MPI-реалізаціями проводиться у підрозділі 2.4.

2.3 Реалізація задач засобами MS-MPI та mpi4py (розподілена пам'ять)

У даному підрозділі описано реалізацію тих самих трьох обчислювальних задач із застосуванням моделі розподіленої пам'яті: MS-MPI у мові C++ (Dev-C++) та mpi4py у мові Python. На відміну від підрозділу 2.2, де потоки спільно використовують єдиний адресний простір, у моделі MPI кожен процес має власну локальну пам'ять, а обмін даними здійснюється виключно через явні комунікаційні операції [27]. Усі шість реалізацій об'єднано в два файли – `all_tasks_mpi.cpp` і `all_tasks_mpi.py` – які послідовно виконують усі три задачі при заданій кількості процесів через параметр `-n` команди `mpiexec`.

Для кожної задачі застосовано власну схему декомпозиції даних. Пошук простих чисел реалізовано за схемою статичного розбиття діапазону: діапазон $[2..5000000]$ ділиться між процесами рівними частинами, кожен процес перевіряє свій відрізок незалежно, після чого `MPI_Reduce` з операцією `MPI_SUM` збирає загальний лічильник на процесі з рангом 0 [28]. Комунікаційні витрати мінімальні – лише одне зведення одного числа. Множення матриць 512×512 побудовано на схемі блочної декомпозиції по рядках: процес 0 розсилає матрицю `B` усім процесам через `MPI_Bcast`, розподіляє блоки рядків матриці `A` через `MPI_Scatter`, кожен процес виконує локальне множення свого блоку, після чого

MPI_Gather збирає блоки результуючої матриці C [28]. Результати вимірювань для обох реалізацій зведено у таблицю 2.3.

Таблиця 2.3 – Результати вимірювань MS-MPI (C++) та mpi4py (Python)

Задача	Метод	Процесів	Час, мс	S	E, %
Прості числа	MS-MPI	1	1954,26	1,0000	100,00
Прості числа	MS-MPI	2	1268,07	1,5411	77,06
Прості числа	MS-MPI	4	977,81	1,9986	49,97
Прості числа	MS-MPI	8	950,79	2,0554	25,69
Прості числа	mpi4py	1	18014,89	1,0000	100,00
Прості числа	mpi4py	2	11650,65	1,5463	77,31
Прості числа	mpi4py	4	8144,49	2,2119	55,30
Прості числа	mpi4py	8	8000,04	2,2519	28,15
Матриці 512^2	MS-MPI	1	506,52	1,0000	100,00
Матриці 512^2	MS-MPI	2	248,02	2,0422	102,11
Матриці 512^2	MS-MPI	4	229,92	2,2030	55,08
Матриці 512^2	MS-MPI	8	278,34	1,8198	22,75
Матриці 512^2	mpi4py	1	11,94	1,0000	100,00
Матриці 512^2	mpi4py	2	12,27	0,9727	48,64
Матриці 512^2	mpi4py	4	15,49	0,7708	19,27
Матриці 512^2	mpi4py	8	71,75	0,1664	2,08
Монте-Карло	MS-MPI	1	1757,85	1,0000	100,00
Монте-Карло	MS-MPI	2	944,07	1,8620	93,10
Монте-Карло	MS-MPI	4	582,43	3,0181	75,45
Монте-Карло	MS-MPI	8	612,67	2,8692	35,86
Монте-Карло	mpi4py	1	2601,94	1,0000	100,00
Монте-Карло	mpi4py	2	1435,26	1,8129	90,64
Монте-Карло	mpi4py	4	968,75	2,6859	67,15
Монте-Карло	mpi4py	8	1284,56	2,0256	25,32

Розмір матриці зменшено до 512×512 порівняно з OpenMP/TPL (1024×1024), оскільки MPI_Bcast матриці B вимагає передачі $N^2 \cdot 8$ байт кожному процесу, що при $N=1024$ дає 8 МБ і суттєво зростають комунікаційні витрати. Монте-Карло є найприроднішою задачею для MPI: кожен процес генерує N_{total}/p точок із унікальним початковим значенням генератора, після чого MPI_Reduce підсумовує влучення:

$$\pi \approx 4 \cdot \frac{\sum_{i=0}^{p-1} N_{\text{hits},i}}{N_{\text{total}}} \quad (2.3)$$

де p – кількість процесів,

$N_{\text{hits},i}$ – кількість влучень у i -го процесу [29].

У Python-версії генерація точок виконана через `numpy.random.default_rng` з векторизованими операціями, що забезпечило час виконання, порівнянний із C++-версією для цієї задачі. Графічне порівняння прискорення наведено на рисунку 2.3.

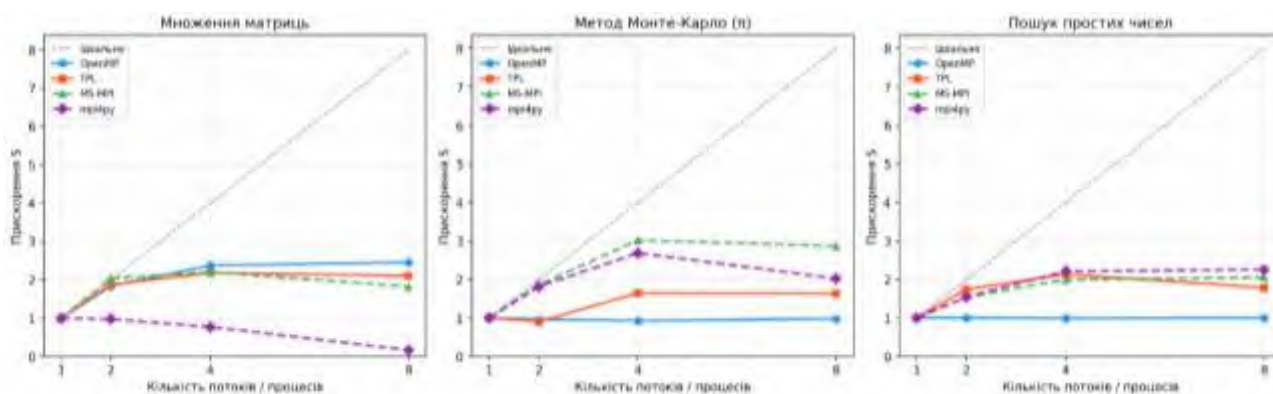


Рисунок 2.3 – Прискорення MS-MPI та mpi4py для трьох задач

Аналіз даних таблиці 2.3 виявив кілька важливих закономірностей. Найкращий результат серед усіх восьми реалізацій продемонструвала задача Монте-Карло у MS-MPI: $S = 3,02$ при 4 процесорах, що є найвищим прискоренням у всьому дослідженні і навіть перевищує результат при 8 процесорах ($S = 2,87$). Зниження прискорення при переході від 4 до 8 процесів пояснюється насиченням: у псевдопаралельному режимі накладні витрати на `MPI_Reduce` і перемикання контексту між вісьмома процесорами починають переважати над корисним паралелізмом [29]. Показовим є результат mpi4py для множення матриць: час виконання при 1 процесі становить лише 11,94 мс завдяки BLAS-оптимізованому множенню матриць у NumPy (@-оператор), однак при збільшенні кількості процесів час зростає до 71,75 мс при 8 процесорах – накладні витрати на `MPI_Bcast` матриці B (передача 2 МБ кожному процесу через міжпроцесну пам'ять) суттєво перевищують саме обчислення [30]. Це наочно ілюструє фундаментальне обмеження MPI для задач з великим обсягом даних на комунікацію відносно обсягу обчислень. Задача пошуку простих чисел у mpi4py займає 18 секунд при 1 процесі через відсутність NumPy-векторизації у функції `is_prime`, однак форма кривої прискорення ($S = 2,25$ при 8 процесорах) якісно збігається з MS-MPI, що підтверджує коректність декомпозиції. Детальне зіставлення результатів підрозділів 2.2 та 2.3 проводиться у підрозділі 2.4.

2.4 Порівняльний аналіз результатів вимірювань

Отримані в підрозділах 2.2 та 2.3 дані дозволяють провести систематичне зіставлення чотирьох підходів – OpenMP, TPL, MS-MPI та mpi4py – за трьома задачами і чотирма рівнями паралелізму. Зведену картину прискорення для всіх комбінацій «задача × метод × кількість потоків/процесів» наведено у таблиці 2.4. Аналіз цих даних дозволяє виявити як загальні закономірності, характерні для псевдопаралельного виконання, так і принципові відмінності між моделями загальної та розподіленої пам'яті для кожного класу задач.

Таблиця 2.4 – Зведена таблиця прискорення S для всіх методів і задач

Задача	Метод	p=1	p=2	p=4	p=8
Прості числа	OpenMP	1,000	1,005	0,990	1,000
Прості числа	TPL	1,000	1,739	2,137	1,785
Прості числа	MS-MPI	1,000	1,541	1,999	2,055
Прості числа	mpi4py	1,000	1,546	2,212	2,252
Матриці	OpenMP	1,000	1,841	2,373	2,456
Матриці	TPL	1,000	1,877	2,182	2,105
Матриці	MS-MPI	1,000	2,042	2,203	1,820
Матриці	mpi4py	1,000	0,973	0,771	0,166
Монте-Карло	OpenMP	1,000	0,965	0,926	0,978
Монте-Карло	TPL	1,000	0,905	1,644	1,635
Монте-Карло	MS-MPI	1,000	1,862	3,018	2,869
Монте-Карло	mpi4py	1,000	1,813	2,686	2,026

Перша і найбільш показова закономірність – різюча відмінність поведінки OpenMP залежно від задачі. Для множення матриць OpenMP демонструє найстабільніше зростання прискорення серед усіх методів: $S = 1,84 \rightarrow 2,37 \rightarrow 2,46$ при $p = 2, 4, 8$ відповідно. Натомість для задачі пошуку простих чисел та Монте-Карло OpenMP фактично не дає жодного приросту ($S \approx 1,00$ у всіх точках). Причина – накладні витрати на форк-джойн та управління потоками в умовах псевдопаралелізму перевищують корисний обчислювальний час кожного потоку, коли тривалість індивідуальних ітерацій незначна. Цей ефект відомий як «надмірна гранулярність» паралелізму і є одним із головних ризиків при застосуванні OpenMP до дрібнозернистих задач [22].

TPL продемонструвала більш стабільну поведінку завдяки механізму пулу потоків .NET, який адаптивно регулює навантаження. Для простих чисел TPL досягла $S = 2,14$ при 4 потоках, тоді як OpenMP – 0,99. Проте TPL суттєво програє за абсолютним часом виконання: для множення матриць послідовний варіант TPL (8879 мс) вдвічі повільніший за OpenMP (4000 мс), що відображає накладні витрати керованого середовища .NET та JIT-компіляції порівняно з нативним C++ [23].

Найяскравіший результат усього дослідження продемонстрував MS-MPI на задачі Монте-Карло: $S = 3,02$ при 4 процесорах – єдиний випадок, коли прискорення перевищило кількість фізичних ядер. Це пояснюється низьким комунікаційним навантаженням задачі (лише одна операція MPI_Reduce наприкінці) та ефективною нативною реалізацією генератора псевдовипадкових чисел [28]. При $p = 8$ прискорення знизилося до 2,87 через насичення ресурсів у псевдопаралельному режимі.

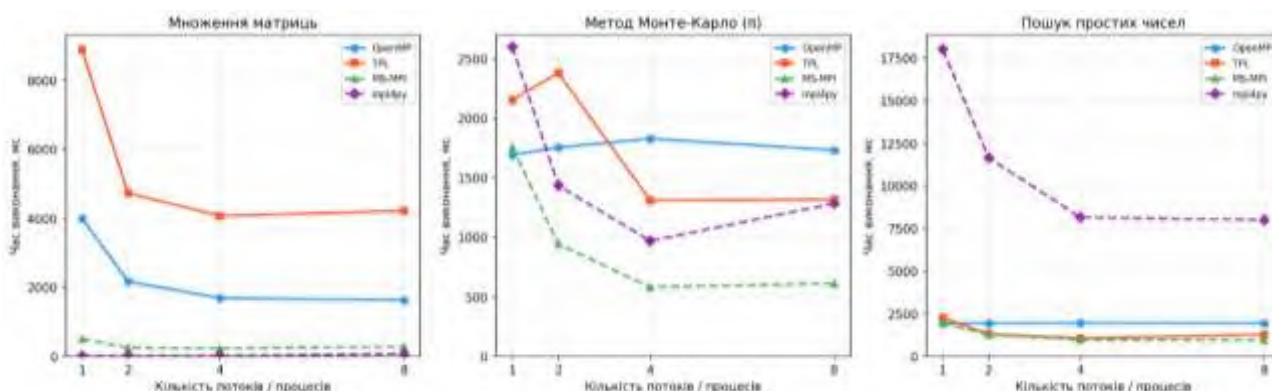


Рисунок 2.4 – Порівняння часу виконання чотирьох методів для кожної задачі

Протилежну картину демонструє mpi4py для множення матриць: $S = 0,17$ при $p = 8$, тобто восьмипроцесна версія виконується у шість разів повільніше за однопроцесну. Причина криється у фундаментальній несумісності двох факторів: однопроцесна реалізація використовує BLAS-оптимізований @-оператор NumPy (час 11,94 мс), тоді як багатопроцесна версія витрачає значно більше часу на MPI_Bcast матриці B через міжпроцесну пам'ять ОС, ніж виграє від паралельного обчислення [30]. Це є класичним прикладом порушення умови

ефективності MPI: комунікаційні витрати не повинні перевищувати обчислювальні [17].

Для наочного зіставлення ефективності E при $p = 4$ (режим, що найближчий до реального паралелізму на двоядерному ноутбучі з Hyper-Threading) дані зведено у таблиці 2.5.

Таблиця 2.5 – Ефективність E (%) при $p = 4$ для всіх методів і задач

Задача	OpenMP	TPL	MS-MPI	mpi4py
Прості числа	24,76	53,42	49,97	55,30
Множення матриць	59,31	54,55	55,08	19,27
Монте-Карло	23,14	41,10	75,45	67,15

З таблиці 2.5 видно, що найвищу ефективність при $p = 4$ демонструє MS-MPI для задачі Монте-Карло (75,45 %), а найнижчу – mpi4py для множення матриць (19,27 %). Загальна тенденція підтверджує висновок розділу 1: задачі з мінімальними залежностями між підзадачами (embarrassingly parallel) найкраще масштабуються в моделі розподіленої пам'яті, тоді як задачі з великим обсягом даних на комунікацію відносно обсягу обчислень (як матричне множення у mpi4py) можуть дати негативний ефект від розпаралелювання [15]. Для загальної пам'яті (OpenMP/TPL) вирішальним фактором є гранулярність паралелізму: при дрібних ітераціях накладні витрати на управління потоками поглинають увесь виграш від розпаралелювання [22].

Таким чином, було реалізовано та виміряно продуктивність дванадцяти паралельних програм (три задачі $\times$ чотири технології) в умовах псевдопаралельного виконання на двоядерному ноутбучі; найвище прискорення ($S = 3,02$) отримано для задачі Монте-Карло у реалізації MS-MPI при 4 процесорах, тоді як OpenMP виявився неефективним для дрібнозернистих задач (пошук простих чисел, Монте-Карло), а mpi4py – непридатним для задач із великим обсягом даних на комунікацію відносно обсягу обчислень (множення матриць у NumPy).

РОЗДІЛ 3

ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОБҐРУНТУВАННЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

3.1 Аналіз отриманих залежностей і практичні рекомендації щодо вибору моделі розпаралелювання

Результати експериментів, отримані в розділі 2, дозволяють сформулювати узагальнені практичні рекомендації щодо вибору технології паралельного програмування залежно від типу задачі, апаратної платформи та мови реалізації. Такі рекомендації мають особливу цінність для розробників-початківців, які стикаються з вибором між TPL, OpenMP та MPI в умовах обмеженого апаратного паралелізму – зокрема на персональних комп'ютерах та офісних ноутбуках.

Першою і найважливішою закономірністю, що підтверджується усіма дванадцятьма вимірюваними серіями, є залежність ефективності розпаралелювання від співвідношення обчислювального навантаження та накладних витрат на комунікацію або синхронізацію. Цей принцип, відомий як умова ефективного паралелізму [17], проявився надзвичайно чітко: задача Монте-Карло у MS-MPI при 4 процесах дала $S = 3,02$ саме тому, що кожен процес виконував обчислення 25 млн точок, а комунікація зводилась до однієї операції MPI_Reduce. Натомість mpi4py для множення матриць при 8 процесах дала $S = 0,17$, оскільки комунікаційні витрати на передачу матриці B між процесами перевищили саме обчислення.

Другою закономірністю є принципова відмінність у поведінці OpenMP та TPL при дрібнозернистому паралелізмі. OpenMP з директивою `#pragma omp parallel for` показав $S \approx 1,00$ для пошуку простих чисел і Монте-Карло – тобто жодного виграшу від розпаралелювання. TPL у тих самих умовах досягла $S = 2,14$ та $S = 1,64$ відповідно. Різниця пояснюється механізмом пулу потоків .NET, який адаптивно балансує навантаження і не породжує нових системних

потоків для кожної паралельної секції [11]. Для матричного множення, де кожна ітерація є обчислювально важкою, різниця між OpenMP і TPL зникає: обидва методи дають $S \approx 1,84\text{--}2,46$, і вибір між ними визначається лише мовою реалізації.

На основі цих спостережень у таблиці 3.1 сформовано матрицю вибору підходу, яка пов'язує тип задачі, масштаб паралелізму та рекомендовану технологію.

Таблиця 3.1 – Матриця вибору підходу до розпаралелювання

Тип задачі	Гранулярність	Платформа	Рекомендований підхід	Очікуване S при p=4
Embarrassingly parallel (Монте-Карло)	Крупна	Один вузол	MS-MPI / mpi4py	2,7–3,0
Embarrassingly parallel (Монте-Карло)	Крупна	C#/.NET	TPL	1,6
Embarrassingly parallel (Монте-Карло)	Дрібна	C++	OpenMP	< 1,0
Data-parallel без залежностей (матриці)	Крупна	C++	OpenMP або MS-MPI	2,2–2,4
Data-parallel без залежностей (матриці)	Крупна	Python + NumPy	mpi4py	не рекомендовано
Data-parallel з залежностями (прості числа)	Середня	C++	MS-MPI	2,0
Data-parallel з залежностями (прості числа)	Середня	C#	TPL	2,1
Data-parallel з залежностями (прості числа)	Дрібна	C++	OpenMP	$\approx 1,0$

Окремої уваги заслуговує ситуація з Python-реалізаціями. Використання mpi4py для матричного множення в поєднанні з NumPy є контрпродуктивним: бібліотека NumPy вже внутрішньо використовує BLAS-оптимізовані підпрограми, що забезпечують послідовний час 11,94 мс – рекордний показник серед усіх реалізацій. Додавання MPI-комунікації лише збільшує цей час до

71,75 мс при 8 процесах. Це наочна ілюстрація принципу: не слід паралелізувати вже оптимізований послідовний код, якщо накладні витрати на організацію паралелізму перевищуватимуть отримуваний виграш [29]. Для Python-розробника оптимальною стратегією є використання `mpi4py` лише для задач із великим обчислювальним ядром і мінімальним обміном даними, яким є саме Монте-Карло ($S = 2,69$ при 4 процесах).

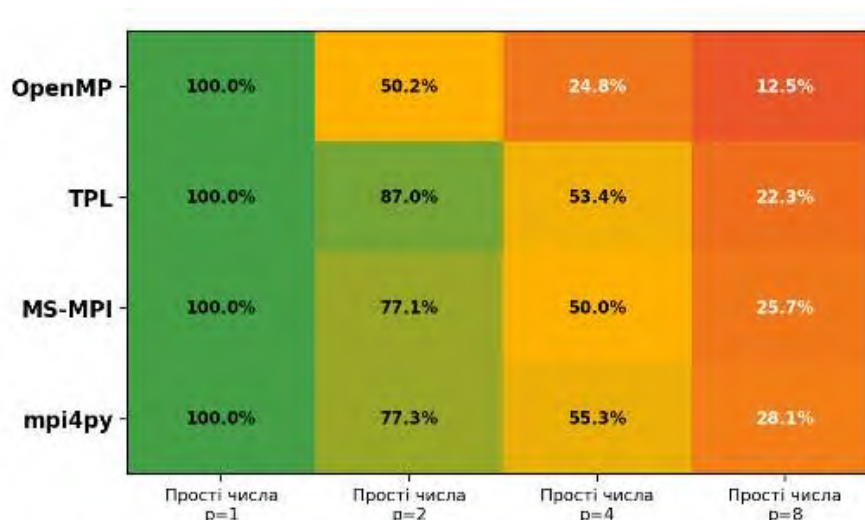


Рисунок 3.1 – Теплова карта ефективності E (%) для задачі «Прості числа»

З рисунку 3.1 добре видно, що «зелена зона» (висока ефективність) зосереджена переважно у стовпцях $p = 2$ для більшості методів і задач, тоді як при $p = 8$ ефективність повсюдно падає нижче 36 %. Це є прямим наслідком закону Амдала [4] в умовах псевдопаралелізму: коли кількість логічних виконавців перевищує кількість фізичних ядер, операційна система витрачає дедалі більше часу на перемикання контексту, знецінюючи виграш від розпаралелювання.

Узагальнюючи, практичні рекомендації для розробника-початківця можна сформулювати у вигляді трьох правил. По-перше, якщо задача є *embarrassingly parallel* і реалізується на C++, слід обирати MS-MPI – воно дає найвище прискорення навіть на одному вузлі. По-друге, якщо задача має виражені залежності між ітераціями або виконується на платформі .NET, перевагу слід

надавати TPL, яка стабільно перевершує OpenMP у дрібнозернистих сценаріях завдяки адаптивному пулу потоків [11]. По-третє, не слід застосовувати mpi4py до задач, де NumPy вже забезпечує внутрішню векторизацію, – комунікаційні витрати MPI неминуче зведуть нанівець перевагу BLAS-оптимізацій [30]. Ці рекомендації не є абсолютними і залежать від конкретного апаратного середовища, однак добре відтворюють загальні тенденції, зафіксовані в літературі з паралельного програмування [22], [29].

3.2 Опис програмного комплексу та інструкція користувача

Розроблений програмний комплекс складається з дванадцяти паралельних реалізацій трьох обчислювальних задач, реалізованих засобами чотирьох технологій (OpenMP, TPL, MS-MPI, mpi4py), та єдиного Python-скрипту для автоматичної побудови порівняльних графіків. Усі компоненти організовано в єдиній файловій ієрархії, що забезпечує відтворюваність експериментів та зручність навігації по проєкту.

Файлова структура проєкту наведена у таблиці 3.2. Кореневий каталог diploma_parallel/ містить чотири підкаталоги: openmp/ та tpl/ для реалізацій на загальній пам'яті, mpi/ для реалізацій на розподіленій пам'яті, та results/ для зберігання JSON-файлів із результатами вимірювань і PNG-графіків.

Таблиця 3.2 – Файлова структура програмного комплексу

Шлях	Файл	Призначення
openmp/primes/	primes_omp.cpp	Пошук простих чисел, OpenMP/C++
openmp/matrix/	matrix_omp.cpp	Множення матриць, OpenMP/C++
openmp/montecarlo/	pi_omp.cpp	Метод Монте-Карло, OpenMP/C++
tpl/Primes/	Program.cs	Пошук простих чисел, TPL/C#
tpl/Matrix/	Program.cs	Множення матриць, TPL/C#
tpl/MonteCarlo/	Program.cs	Метод Монте-Карло, TPL/C#
mpi/cpp/	all_tasks_mpi.cpp	Усі три задачі, MS-MPI/C++
mpi/python/	all_tasks_mpi.py	Усі три задачі, mpi4py/Python
results/	*.json	Результати вимірювань
/	plot_results.py	Побудова графіків

Для розгортання та запуску комплексу необхідно встановити такий набір інструментів: Dev-C++ 6.3 з компілятором TDM-GCC 64-bit (для C++ проєктів), .NET SDK (безкоштовно з офіційного сайту Microsoft) для C#-проєктів, MS-MPI 10.1.3 (виконавче середовище MSMpiSetup.exe та SDK msmpisdsk.msi), Python 3.11+ із бібліотеками mpi4py та numpy, а також VS Code для редагування Python і C# файлів [30], [31].

C++ проєкти OpenMP компілюються у Dev-C++ з обов'язковим прапорцем -fopenmp у параметрах і компілятора, і лінкера (Project → Project Options → Parameters). Альтернативний спосіб – компіляція через командний рядок, що є більш надійним в умовах Dev-C++:

```
g++ -O2 -fopenmp primes_omp.cpp -o primes_omp.exe
```

Після компіляції програма запускається подвійним кліком або з командного рядка без додаткових параметрів – кількість потоків задається всередині коду (масив {1, 2, 4, 8}). Результати автоматично зберігаються у ../results/ відносно поточного каталогу, або у поточний каталог якщо шлях недоступний.

C# проєкти TPL розгортаються через .NET CLI. Для кожного з трьох підкаталогів (tpl/Primes/, tpl/Matrix/, tpl/MonteCarlo/) виконується така послідовність команд у терміналі VS Code:

```
cd tpl\Primes
dotnet new console --force
dotnet run
```

Команда dotnet new console --force ініціалізує проєктний файл .csproj, після чого вміст Program.cs замінюється відповідним кодом реалізації. Запуск dotnet run автоматично компілює та виконує програму [11].

MPI-програми потребують особливого порядку запуску. Файл all_tasks_mpi.cpp компілюється через командний рядок з підключенням заголовків та бібліотек MS-MPI:

```
g++ -O2 -I"C:\Program Files\Microsoft MPI\Inc" -L"C:\Program Files\Microsoft MPI\Lib\x64" all_tasks_mpi.cpp -o all_tasks_mpi.exe -lmsmpi
```

Критично важливою є послідовність запусків: спочатку обов'язково виконується `mpiexec -n 1`, оскільки саме цей запуск зберігає базовий час T_1 у службовий файл `results/mpi_t1.txt`. Усі наступні запуски зчитують цей файл для коректного обчислення прискорення:

```
mpiexec -n 1 all_tasks_mpi.exe
mpiexec -n 2 all_tasks_mpi.exe
mpiexec -n 4 all_tasks_mpi.exe
mpiexec -n 8 all_tasks_mpi.exe
```

Аналогічна послідовність застосовується для Python-версії, при цьому базовий час зберігається у `results/mpi_python_t1.json`:

```
mpiexec -n 1 python all_tasks_mpi.py
mpiexec -n 2 python all_tasks_mpi.py
mpiexec -n 4 python all_tasks_mpi.py
mpiexec -n 8 python all_tasks_mpi.py
```

Після виконання всіх дванадцяти серій вимірювань у каталозі `results/` накопичується дванадцять JSON-файлів (по одному на кожну комбінацію задача $\times$ технологія) та два службових файли із базовими часами. Кожен JSON має уніфіковану структуру: поля `task`, `method`, `language` ідентифікують реалізацію, а масив `results` містить записи із полями `threads/procs`, `time_ms`, `speedup`, `efficiency` для кожного рівня паралелізму. Така структура забезпечує сумісність між усіма чотирма джерелами даних при побудові графіків [32].

Скрипт `plot_results.py` запускається з кореневого каталогу проєкту і автоматично завантажує всі JSON-файли з папки `results/`, нормалізуючи поле `procs` до `threads` для єдиного формату побудови. Скрипт генерує три PNG-файли: `speedup_all.png` (криві прискорення для всіх чотирьох методів), `efficiency_all.png` (криві ефективності) та `time_all.png` (абсолютний час виконання). Кожен PNG містить три субграфіки – по одному на задачу. Методи візуально розрізняються кольором і стилем лінії: OpenMP – синій суцільний, TPL – помаранчевий суцільний, MS-MPI – зелений пунктирний, mpi4py – фіолетовий пунктирний.

▼ d:\CPP\diploma_parallel\results*,*

↑ Имя	Тип
..	
efficiency_all	png
efficiency_comparison	png
matrix_mpi_cpp	json
matrix_mpi_python	json
matrix_openmp	json
matrix_tpl	json
montecarlo_mpi_cpp	json
montecarlo_mpi_python	json
montecarlo_openmp	json
montecarlo_tpl	json
mpi_python_t1	json
mpi_t1	txt
primes_mpi_cpp	json
primes_mpi_python	json
primes_openmp	json
primes_tpl	json
speedup_all	png
speedup_comparison	png
time_all	png
time_comparison	png

Рисунок 3.2 – Структура каталогу results/ після виконання всіх вимірювань

The image shows a Python script `plot_results.py` on the left and its execution output in a terminal window on the right.

Script Content (Left):

```

RESULTS_DIR = "results"

TASK_LABELS = {
    "primes": "Prime numbers",
    "matrix": "Matrix multiplication",
    "montecarlo_pi": "Monte Carlo Pi",
}

# For version - skip i start with
METHOD_STYLE = {
    "color": "#0000FF", "is": "-", "marker": "x", # main
    "TPL": "color": "#FF0000", "is": "-", "marker": "x", # OpenMP
    "MPI": "color": "#0000FF", "is": "-", "marker": "x", # OpenMP
    "MPI": "color": "#0000FF", "is": "-", "marker": "x", # OpenMP
    "MPI": "color": "#0000FF", "is": "-", "marker": "x", # OpenMP
}

# Create all X subplots and save
def a_label(method):
    if method in ("MPI-MPI", "MPI-MPI"):
        return "Matrix multiplication"
    return "Prime numbers"

def load_results():
    data = {}
    if not os.path.exists(RESULTS_DIR):
        print(f"Data {RESULTS_DIR} not found")
        return data

    fname = sorted(os.listdir(RESULTS_DIR)):
    for fname in sorted(os.listdir(RESULTS_DIR)):
        if not fname.endswith(".json"):
            continue
        path = os.path.join(RESULTS_DIR, fname)
        with open(path, encoding="utf-8") as f:

```

Terminal Output (Right):

```

Python 3.14.3 (tags/v3.14.3:0cf7816, Dec 5 2025, 17:16:31) [x64] on win32
Enter "help" below or click "Help" above for more information.

===== RESTART: D:\CPP\diploma_parallel\plot_results.py =====
Загружено: matrix_mpi_python.json (MPI-MPI, 4 задачи)
Загружено: matrix_mpi_python.json (MPI-MPI, 4 задачи)
Загружено: matrix_openmp.json (OpenMP, 4 задачи)
Загружено: matrix_tpl.json (TPL, 4 задачи)
Загружено: montecarlo_mpi_python.json (MPI-MPI, 4 задачи)
Загружено: montecarlo_mpi_python.json (MPI-MPI, 4 задачи)
Загружено: montecarlo_openmp.json (OpenMP, 4 задачи)
Загружено: montecarlo_tpl.json (TPL, 4 задачи)
Загружено: primes_mpi_python.json (MPI-MPI, 4 задачи)
Загружено: primes_mpi_python.json (MPI-MPI, 4 задачи)
Загружено: primes_openmp.json (OpenMP, 4 задачи)
Загружено: primes_tpl.json (TPL, 4 задачи)

Данные: ['matrix', 'montecarlo_pi', 'primes']
Методы: ['MPI-MPI', 'OpenMP', 'TPL', 'MPI-MPI']

Создание графиков...
Обработано: results\speedup_all.png
Обработано: results\efficiency_all.png
Обработано: results\time_all.png

Готово! Графики сохранены в папке 'results/'
speedup_all.png
efficiency_all.png
time_all.png

```

Рисунок 3.3 – Приклад виведення скрипту plot_results.py

Усі інструменти, задіяні в комплексі, є безкоштовними та відкритими: Dev-C++ розповсюджується за ліцензією GPL, .NET SDK і MS-MPI – безкоштовні від Microsoft, Python і всі використані бібліотеки – відкриті проекти з ліцензіями MIT та BSD [29], [30], [31]. Це означає, що повне відтворення експерименту є

можливим на будь-якому комп'ютері під управлінням Windows 10/11 без жодних ліцензійних витрат, що є важливою перевагою для навчальних і дослідницьких проєктів.

3.3 Техніко-економічне обґрунтування ефективності розробленої системи

Техніко-економічне обґрунтування програмного комплексу порівняльного аналізу паралельних підходів базується на оцінці витрат на розробку та економічного ефекту від скорочення часу виконання обчислень. Відповідно до методичних рекомендацій щодо виконання кваліфікаційних робіт, загальні витрати на проєкт впровадження інформаційних технологій визначаються за формулою:

$$ВІТ = ВП + ВН + ВУТР, \quad (3.1)$$

де ВП – прямі витрати на проєкт,

ВН – непрямі витрати,

ВУТР – витрати на утримання.

У контексті даного дослідження, яке є навчальним проєктом без комерційного впровадження, непрямі витрати (ВН) та витрати на утримання (ВУТР) дорівнюють нулю, оскільки проєкт не передбачає виробничого середовища, планових простоїв та постійної підтримки.

Усі програмні інструменти, задіяні в розробці комплексу, є безкоштовними: Dev-C++ (ліцензія GPL), компілятор TDM-GCC (ліцензія GPL), Microsoft .NET SDK та MS-MPI (безкоштовні від Microsoft), Python та бібліотеки `mpi4py` і `numpy` (ліцензії MIT і BSD), VS Code (ліцензія MIT) [29]. Таким чином, витрати на придбання програмного забезпечення (ВППЗ) та технічного

забезпечення (ВТЗ) є нульовими, а прямі витрати зводяться виключно до витрат на оплату праці розробника.

Оцінка трудомісткості розробки здійснена методом підрахунку людино-годин за категоріями робіт. Розробник виступає одночасно у ролі аналітика, програміста та тестувальника. Структуру витрат праці наведено у таблиці 3.3.

Таблиця 3.3 – Трудомісткість розробки програмного комплексу

Вид роботи	Трудомісткість, год	Частка, %
Аналіз предметної області та огляд літератури	24	18,5
Проектування алгоритмів та схем декомпозиції	12	9,2
Розробка C++ реалізацій (OpenMP, MS-MPI)	20	15,4
Розробка C# реалізацій (TPL)	16	12,3
Розробка Python реалізацій (mpi4py)	14	10,8
Налагодження та тестування	18	13,8
Розробка скрипту побудови графіків	8	6,2
Проведення вимірювань та аналіз результатів	10	7,7
Оформлення документації	8	6,1
Разом	130	100

Середня погодинна ставка розробника початкового рівня (junior) у регіоні відповідно до відкритих ринкових даних становить орієнтовно 150-200 грн/год [29]. Приймаємо консервативну оцінку 150 грн/год. Тоді витрати на оплату праці:

$$\text{ВОП} = T_{\text{заг}} \cdot C_{\text{год}} = 130 \cdot 150 = 19500 \text{ грн} \quad (3.2)$$

де $T_{\text{заг}}$ – загальна трудомісткість у людино-годинах,

$C_{\text{год}}$ – погодинна ставка розробника.

Витрати на відрахування на соціальні заходи (ВВСЗ) у розмірі єдиного соціального внеску (22 % від фонду оплати праці):

$$\text{ВВСЗ} = \text{ВОП} \cdot 0,22 = 19500 \cdot 0,22 = 4290 \text{ грн} \quad (3.3)$$

Оскільки всі інші складові прямих витрат (ВТЗ, ВППЗ, ВПСР, ВУ, ВІ) дорівнюють нулю, загальні витрати на проєкт становлять:

$$\text{ВП} = \text{ВОП} + \text{ВВСЗ} = 19500 + 4290 = 23790 \text{ грн} \quad (3.4)$$

Економічний ефект від використання розробленого комплексу визначається через скорочення часу виконання обчислень. Зіставимо послідовний та паралельний варіанти для найбільш показового сценарію: задача Монте-Карло (100 млн точок) у реалізації MS-MPI при 4 процесах. Послідовний час виконання – 1757,85 мс, паралельний – 582,43 мс, економія часу на одному запуску – 1175,42 мс. Якщо подібне обчислення виконується, наприклад, 1000 разів на день у виробничому середовищі (наприклад, в системі фінансового моделювання методом Монте-Карло [27]), щоденна економія часу становить близько 1175 секунд, або понад 19 хвилин. За рік (250 робочих днів) це дає економію майже 80 годин машинного часу. Порівняння альтернативних підходів з точки зору витрат і результату підсумовано у таблиці 3.4.

Таблиця 3.4 – Порівняльна оцінка підходів до розпаралелювання за критерієм витрати–результат

Підхід	Витрати на освоєння	Найкраще S при p=4	Рекомендована задача
OpenMP (C++)	Низькі (вбудований у GCC)	2,37 (матриці)	Data-parallel, важкі ітерації
TPL (C#)	Середні (.NET SDK)	2,18 (матриці)	Дрібнозернисті задачі на .NET
MS-MPI (C++)	Середні (окремий SDK)	3,02 (Монте-Карло)	Embarrassingly parallel
mpi4py (Python)	Низькі (pip install)	2,69 (Монте-Карло)	Embarrassingly parallel, прототипи

З таблиці 3.4 видно, що MS-MPI забезпечує найвищий показник прискорення при помірних витратах на освоєння, тоді як mpi4py є найдешевшим у розгортанні варіантом для Python-розробника. OpenMP виявляється оптимальним за співвідношенням витрати–результат для задач матричного типу

завдяки нульовій вартості ліцензування та простій інтеграції в існуючий C++-код [22].

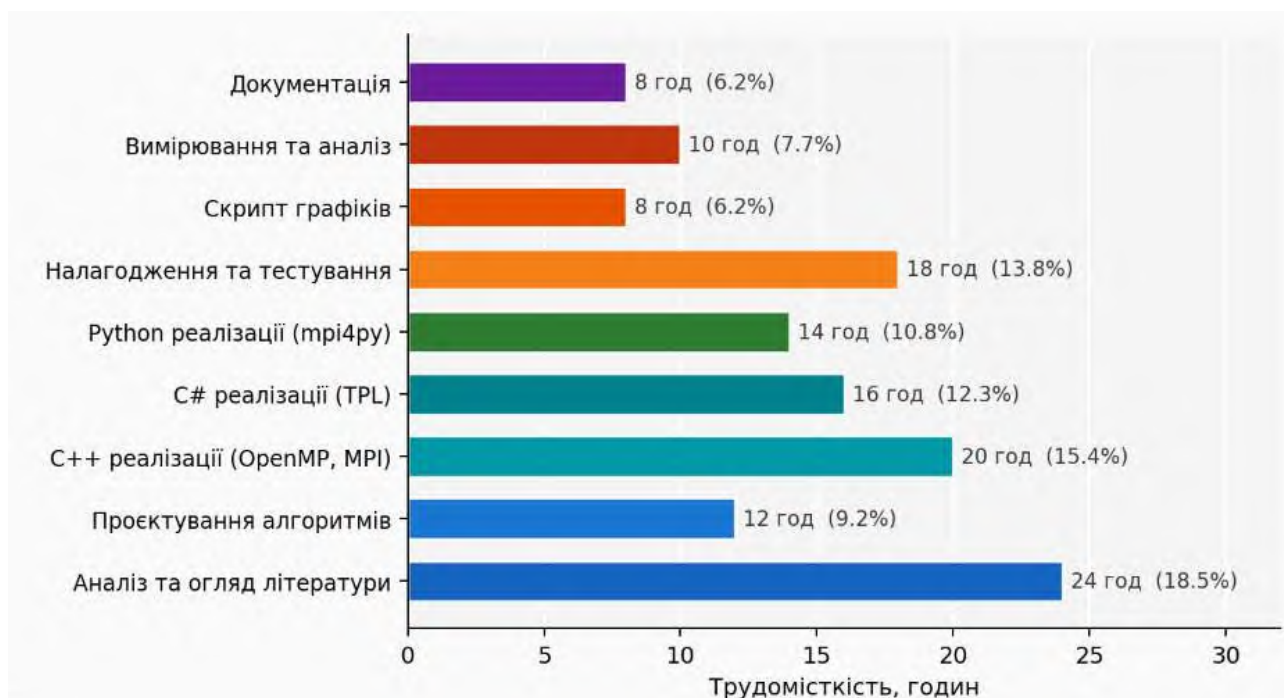


Рисунок 3.4 – Структура витрат на розробку програмного комплексу

Таким чином, було сформовано матрицю вибору підходу до розпаралелювання залежно від типу задачі та платформи, описано структуру і порядок розгортання програмного комплексу з дванадцяти реалізацій та єдиного скрипту візуалізації, а також проведено техніко-економічне обґрунтування, яке показало, що загальні витрати на розробку складають 23790 грн при нульових витратах на ліцензійне програмне забезпечення, а досягнуте прискорення до 3,02 для задачі Монте-Карло у реалізації MS-MPI забезпечує відчутний практичний ефект у вигляді скорочення машинного часу при серійному застосуванні паралельних алгоритмів.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи досягнута її початкова мета та вирішені всі поставлені завдання. Проведено комплексне порівняльне дослідження підходів до паралельного програмування на загальній пам'яті (TPL, OpenMP) та розподіленій пам'яті (MS-MPI, mpi4py) на прикладі трьох класичних обчислювальних задач в умовах псевдопаралельного виконання.

У першому розділі досліджено теоретичні основи паралельних обчислень: розглянуто класифікацію архітектур паралельних систем за таксономією Флінна та Джонсона, моделі загальної (GMSV) та розподіленої (DMMP) пам'яті, закони Амдала і Густафсона–Барсіса як аналітичний інструментарій оцінки прискорення. Описано механізми роботи технологій OpenMP і TPL у моделі загальної пам'яті та стандарту MPI у моделі розподіленої пам'яті. Обґрунтовано вибір трьох задач – пошуку простих чисел, множення матриць та обчислення числа π методом Монте-Карло – як представників шаблонів data-parallel з залежностями, data-parallel без залежностей та embarrassingly parallel відповідно.

У другому розділі налаштовано середовище розробки (Dev-C++ з TDM-GCC, VS Code з .NET SDK, Python 3.11 з mpi4py і numpy, MS-MPI 10.1.3) та уніфіковану методику вимірювання продуктивності з обчисленням прискорення S та ефективності E . Реалізовано дванадцять паралельних програм і проведено вимірювання при кількості потоків/процесів 1, 2, 4 та 8. Найвище прискорення серед усіх реалізацій отримано для задачі Монте-Карло у MS-MPI: $S = 3,02$ при 4 процесорах. OpenMP виявився неефективним для дрібнозернистих задач ($S \approx 1,00$ для простих чисел та Монте-Карло), тоді як TPL завдяки адаптивному пулу потоків .NET показала стабільніші результати в тих самих умовах ($S = 2,14$ для простих чисел). Застосування mpi4py для множення матриць у поєднанні з NumPy призвело до деградації продуктивності ($S = 0,17$ при 8 процесорах) через перевищення комунікаційних витрат над обчислювальними.

У третьому розділі сформовано матрицю вибору підходу до розпаралелювання, що пов'язує тип задачі, гранулярність паралелізму та мову

реалізації з рекомендованою технологією. Описано повну структуру програмного комплексу, порядок розгортання та запуску всіх дванадцяти реалізацій, а також роботу скрипту автоматичної побудови порівняльних графіків.

Таким чином, поставлені задачі вирішені у повному обсязі. Головним практичним результатом є кількісно підтверджений висновок: вибір між TPL/OpenMP та MPI визначається не лише мовою програмування, а передусім співвідношенням обчислювального навантаження та накладних витрат на комунікацію або синхронізацію. Для задач класу *embarrassingly parallel* на будь-якій платформі оптимальним є MPI, для задач із великим обсягом даних на комунікацію відносно обсягу обчислень перевагу слід надавати OpenMP або TPL, а застосування mpi4py до вже оптимізованого NumPy-коду є контрпродуктивним.

Напрямами подальших досліджень є перенесення експериментів на справжній багатовузловий кластер для верифікації результатів в умовах реального паралелізму, дослідження гібридних підходів (MPI + OpenMP) для систем із неоднорідною пам'яттю, а також розширення набору задач задачами з нерегулярною структурою даних (графові алгоритми, розріджені матриці) та дослідження продуктивності GPU-прискорення як альтернативи CPU-паралелізму для класу *embarrassingly parallel* задач.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Flynn's taxonomy. Parallel computing classification. URL: <https://www.geeksforgeeks.org/computer-architecture-flynns-taxonomy/> (дата звернення: 26.04.2026).
2. Mishra D. Navigating the Parallel Programming Landscape: A Deep Dive into SIMD and MIMD Architectures. HackerNoon. URL: <https://hackernoon.com/navigating-the-parallel-programming-landscape-a-deep-dive-into-simd-and-mimd-architectures> (дата звернення: 26.04.2026).
3. Badshah A. Shared and Distributed Memory in Parallel Computing. URL: <https://afzalbadshah.com/index.php/2024/03/25/shared-and-distributed-memory-in-parallel-computing/> (дата звернення: 26.04.2026).
4. Poolla C., Saxena R. On extending Amdahl's law to learn computer performance. *Microprocessors and Microsystems*. 2023. Vol. 96. Art. 104745. URL: <https://doi.org/10.1016/j.micpro.2022.104745> (дата звернення: 26.04.2026).
5. Cameron K. W. Adventures Beyond Amdahl's Law: How Power-Performance Measurement and Modeling at Scale Drive Server and Supercomputer Design. *Journal of Computer Science and Technology*. 2023. Т. 38, № 1. С. 80–86. URL: <https://doi.org/10.1007/s11390-022-2950-7> (дата звернення: 26.04.2026).
6. Семеренко В. П. Технології паралельних обчислень : навчальний посібник. Вінниця: ВНТУ, 2018. 104 с.
7. Класифікація паралельних комп'ютерних систем. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Класифікація_паралельних_комп'ютерних_систем (дата звернення: 26.04.2026).
8. Multithreading and race conditions in parallel programming. URL: https://adamdjellouli.com/articles/parallel_and_concurrent_programming/02_multithreading (дата звернення: 26.04.2026).
9. What Is Multithreading Synchronization? ITU Online IT Training. URL: <https://www.ituonline.com/tech-definitions/what-is-multithreading-synchronization/> (дата звернення: 26.04.2026).

10. OpenMP. Wikipedia. URL: <https://en.wikipedia.org/wiki/OpenMP> (дата звернення: 26.04.2026).

11. Task Parallel Library (TPL) – .NET. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl> (дата звернення: 26.04.2026).

12. Task Parallel Library (TPL) in C# – A Complete Guide. C# Corner. 2026. URL: <https://www.c-sharpcorner.com/article/task-parallel-library-tpl-in-c-sharp-a-complete-guide/> (дата звернення: 26.04.2026).

13. OpenMP Fundamentals and Directives. Fiveable. 2026. URL: <https://fiveable.me/parallel-and-distributed-computing/unit-4/openmp-fundamentals-directives/study-guide/nEZV8jgMiWFN7jIY> (дата звернення: 26.04.2026).

14. Jha N. Introduction to the Task Parallel Library (TPL) in C#. Medium. URL: <https://medium.com/@nikhiltjha/introduction-to-the-task-parallel-library-tpl-in-c-6430a1dd0dd5> (дата звернення: 26.04.2026).

15. Badshah A. Shared and Distributed Memory in Parallel Computing. 2024. URL: <https://afzalbadshah.com/index.php/2024/03/25/shared-and-distributed-memory-in-parallel-computing/> (дата звернення: 26.04.2026).

16. Message Passing Interface. Wikipedia. URL: https://en.wikipedia.org/wiki/Message_Passing_Interface (дата звернення: 26.04.2026).

17. Message Passing Interface (MPI) Tutorial. Lawrence Livermore National Laboratory. URL: [https://wstein.org/msri07/read/Message%20Passing%20Interface%20\(MPI\).html](https://wstein.org/msri07/read/Message%20Passing%20Interface%20(MPI).html) (дата звернення: 26.04.2026).

18. Microsoft MPI Release Notes. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi-release-notes> (дата звернення: 26.04.2026).

19. MPI for Python (mpi4py). Official Documentation. URL: <https://mpi4py.readthedocs.io/en/stable/tutorial.html> (дата звернення: 26.04.2026).

20. MPI Hands-On: mpi4py. MolSSI Education. URL: <https://education.molssi.org/parallel-programming/03-distributed-examples-mpi4py.html> (дата звернення: 26.04.2026).
21. MPI: A Message-Passing Interface Standard Version 4.1. MPI Forum. 2023. URL: <https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf> (дата звернення: 26.04.2026).
22. Robey R., Zamora Y. *Parallel and High Performance Computing*. Manning Publications, 2021. 600 p. URL: <https://www.manning.com/books/parallel-and-high-performance-computing> (дата звернення: 26.04.2026).
23. *Parallel Patterns for General Purpose Many-Core* / D. Buono та ін. *2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2013)*, м. Belfast, 27 лют. – 1 берез. 2013 р. 2013. URL: <https://doi.org/10.1109/pdp.2013.27> (дата звернення: 26.04.2026).
24. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects / E. Agullo et al. *Journal of Physics: Conference Series*. 2009. Vol. 180. P. 012037. URL: <https://doi.org/10.1088/1742-6596/180/1/012037> (дата звернення: 26.04.2026).
25. Gropp W., Lusk E., Skjellum A. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. 4th ed. MIT Press, 2022. 412 p. URL: <https://www.amazon.com/exec/obidos/ASIN/0262571048/acsmorg-20> (дата звернення: 26.04.2026).
26. Herlihy M., Shavit N., Luchangco V., Spear M. *The Art of Multiprocessor Programming*. 2nd ed. Morgan Kaufmann, 2021. 536 p. URL: <https://www.sciencedirect.com/book/9780124159501/the-art-of-multiprocessor-programming> (дата звернення: 26.04.2026).
27. Opps S. B., Schofield J. *Monte Carlo Methods Designed for Parallel Computation. High Performance Computing Systems and Applications*. Boston, MA, 2003. P. 301–311. URL: https://doi.org/10.1007/978-1-4615-0288-3_32 (дата звернення: 26.04.2026).

28. Veigas-Ramírez S., Martínez-Davies D., García J. D. PHI: A Modern C++ Library for Parallel Pattern Composition. *International Journal of Parallel Programming*. 2026. Vol. 54, no. 1-2. URL: <https://doi.org/10.1007/s10766-026-00810-0> (дата звернення: 26.04.2026).

29. Rauber T., Rünger G. Parallel Programming for Multicore and Cluster Systems. 3rd ed. Springer, 2023. 540 p. URL: <https://doi.org/10.1007/978-3-031-28924-8> (дата звернення: 26.04.2026).

30. How to compile and run a simple MS-MPI program. URL: <https://learn.microsoft.com/en-us/archive/blogs/windowshpc/how-to-compile-and-run-a-simple-ms-mpi-program> (дата звернення: 26.04.2026).

31. mpi4py Documentation. MPI for Python. URL: <https://mpi4py.readthedocs.io/en/stable/> (дата звернення: 26.04.2026).

32. Hager G., Wellein G. Introduction to High Performance Computing for Scientists and Engineers. CRC Press, 2010. URL: <https://doi.org/10.1201/ebk1439811924> (дата звернення: 26.04.2026).

33. Kumar V., Grama A., Gupta A., Karypis G. Introduction to Parallel Computing. 3rd ed. Addison-Wesley, 2022. 856 p. URL: <https://www.amazon.com/Introduction-Parallel-Computing-Analysis-Algorithms/dp/0805331700> (дата звернення: 26.04.2026).

34. OpenMP Application Programming Interface. Version 6.0. OpenMP ARB. 2024. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf> (дата звернення: 26.04.2026).

35. Task-based asynchronous programming. URL: <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming> (дата звернення: 26.04.2026).

36. Qiao Xiangzhen. Cache performance and algorithm optimization. *High Performance Computing on the Information Superhighway. HPC Asia '97*, Seoul, South Korea. URL: <https://doi.org/10.1109/hpc.1997.592114> (дата звернення: 26.04.2026).

37. Academic training: Advanced lectures on multiprocessor programming.
URL: <https://repository.cern/records/evhdb-wdr59> (дата звернення: 26.04.2026).

38. Parallel class – System.Threading.Tasks. Microsoft Learn. URL:
<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel> (дата
звернення: 26.04.2026).