НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: «Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями»

Виконав (-ла): студент (-ка) 4-го курсу, групи ІП-05
(шифр групи)

Бондарчук Артем Дмитрович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доц. каф. ОТ, к. т. н., Корочкін О. В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) доц. каф. ОТ, Волокита А. М. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент доц. каф. СПСКС, к. т. н., Тарасенко-Клятченко О.В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____
(підпис)

Київ – 2024 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

ЗАВДАННЯ

на бакалаврський дипломний проєкт студента

Бондарчука Артема Дмитровича

1. Тема проєкту Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями
Керівник проєкту доцент, к. т. н. Корочкін Олександр Володимирович ,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 27 травня 2024 року № 2112-с

2. Термін здачі студентом закінченого проєкту *6 червня 2024 р.*

3. Вихідні дані до проєкту *технічна документація. теоретичні дані.*

4. Зміст пояснювальної записки (перелік завдань, які потрібно розробити)

Розділ 1. Огляд структур та сфер застосування високопродуктивних комп’ютерних систем

Розділ 2. Розробка програмно-апаратного комплексу

Розділ 3. Розробка програмного забезпечення

Розділ 4. Тестування програми

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи (структурна схема), алгоритм дій програмного забезпечення (принципова схема), діаграма класів (функціональна схема).

6. Консультанти розділів проєкту

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання 15 грудня 2023р.

Календарний план

№ п/п	Назва етапу дипломного проєкту	Строк виконання етапу проєкту	Примітка
1.	*Затвердження теми проєкту*	*10.12.2023-15.12.2023*	
2.	*Вивчення та аналіз завдання*	*15.12.2023-15.03.2024*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2024-25.03.2024*	
4.	*Розробка структур окремих підсистем*	*25.03.2024-5.04.2024*	
5.	*Програмна реалізація системи*	*5.04.2024-15.04.2024*	
6.	*Оформлення пояснювальної записки*	*15.04.2024-20.05.2024*	
7.	*Захист програмного продукту*	*24.05.2024*	
8.	*Передзахист*	*6.06.2024*	
9.	*Захист*	*20.06.2024*	

Студент _____ Артем БОНДАРЧУК
(підпис)

Керівник проєкту _____ Олександр КОРОЧКІН
(підпис)

АНОТАЦІЯ

Розроблений дипломний проєкт має на меті створення високопродуктивного програмно-апаратного комплексу для операцій з комплексними матрицями. За останні роки спостерігається збільшення актуальності використання багатопотокових обчислювальних систем у зв'язку з великим обсягом матричних операцій у різних галузях діяльності.

Методика, запропонована у цьому проекті, передбачає створення комп'ютерної системи на основі наявних комплектуючих та розробку програмного забезпечення для ефективного вирішення матричних операцій за допомогою паралельних обчислень. Використання багатопотоковості дозволить досягти максимальної ефективності при вирішенні задач, пов'язаних з операціями над комплексними матрицями.

Ключові слова: високопродуктивний програмно-апаратний комплекс, багатоядерний процесор, паралельне програмування, матричні операції.

ANNOTATION

The aim of this thesis project is to create a high-performance hardware and software system for operations with complex matrices. In recent years, there has been an increase in the relevance of using multi-threaded computing systems due to the large volume of matrix operations in various industries.

The methodology proposed in this project involves the creation of a computer system based on existing components and the development of software for the efficient solution of matrix operations using parallel computing. The use of multithreading will allow to achieve maximum efficiency in solving problems related to operations on complex matrices.

Keywords: high-performance hardware and software complex, multi-core processor, parallel programming, matrix operations.

Довідки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпляр	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.466500.001 ОА***	Високопродуктивний програмно-	1		
			апаратний комплекс для операцій			
			з комплексними матрицями			
			Опис альбому			
	А4	***ІАЛЦ.466500.002 ТЗ***	Високопродуктивний програмно-	3		
			апаратний комплекс для операцій			
			з комплексними матрицями			
			Технічне завдання			
	А4	***ІАЛЦ.466500.003 ПЗ***	Високопродуктивний програмно-	73		
			апаратний комплекс для операцій			
			з комплексними матрицями			
			Пояснювальна записка			
	А4	***ІАЛЦ.466500.004 Д1***	Структурна схема системи	1		
			(структурна схема)			
	А4	***ІАЛЦ.466500.005 Д2***	Алгоритм дій програмного	1		
			забезпечення			
			(принципова схема)			
	А4	***ІАЛЦ.466500.006 Д3***	Діаграма класів	1		
			(функціональна схема)			
	А4	***ІАЛЦ.466500.007 Д4***	Текст програмного коду	9		

					ІАЛЦ.466500.001 ОА		
Зм	*Лист*	*№ докум.*	*Підп.*	*Дата*			
Розроб		Бондарчук А. Д.			*Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Корочкін О. В.					
						КПІ ім. Ігоря Сікорського ФІОТ, ІП-05	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями*»

Київ – 2024

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку високопродуктивного програмно-апаратного комплексу для операцій з комплексними матрицями.

Областю застосування цієї системи є розв'язання математичних задач, які включають обробку та аналіз складних даних у формі матриць, таких як аналіз зображень та звуку, обчислення фізичних моделей у науці та інженерії, обробка сигналів у сфері телекомунікацій, та багато інших.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання на розробку високопродуктивного програмно-апаратного комплексу для операцій з комплексними матрицями для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка програмно-апаратного комплексу для скорочення часу розв'язання наданих векторно-матричних операцій.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами для розробки даного дипломного проекту є статті з мережі Інтернет, науково-технічна література з комп’ютерних технологій та паралельного програмування, офіційні документації і наукові статті.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система повинна відповідати наступним вимогам:

- Забезпечувати швидке та ефективне виконання обчислень з комплексними матрицями.
- Мати оптимальне співвідношення ціни та якості компонентів, забезпечуючи при цьому стабільну та безвідмовну роботу.

- Забезпечувати гнучкість та можливість подальшого розвитку системи шляхом заміни або оновлення окремих компонентів без необхідності повної переробки архітектури.

5.2. Вимоги до програмного забезпечення

Операційна система Microsoft Windows 10 і вище (рекомендовано)

5.3. Вимоги до апаратної частини

Вимоги до центрального процесора: не нижче моделей Intel® Core™ i5 або AMD® Ryzen™ 5.

Обсяг оперативної пам'яті (RAM): не менше 8 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.12.2023-15.12.2023
Вивчення та аналіз завдання	15.12.2023-15.03.2024
Розробка архітектури та загальної структури системи	15.03.2024-25.03.2024
Розробка структур окремих частин системи	25.03.2024-5.04.2024
Програмна реалізація системи	5.04.2024-15.04.2024
Виправлення помилок	15.04.2024-20.05.2024
Оформлення пояснювальної записки	25.04.2024

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями»

Київ – 2024

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ВПКС	Високопродуктивна комп’ютерна система
ЦОС	Цифрова обробка сигналів
FFT	Швидке перетвореня Фур’є (англ. Fast Fourier Transform)
ПАК	Програмно-апаратний комплекс
RDMA	Remote Direct Memory Access
кБ	Кілобайт (1000 байт)
МБ	Мегабайт (1024 кБ)
CPU	Центральний процесор (англ. Central Processing Unit)
GPU	Графічний процесор (англ. Graphical Processing Unit)
ВПО	Високопродуктивні обчислення
ОС	Операційна система
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
RAM	Оперативна пам’ять (англ. Random Access Memory)
TDP	Максимальне теоретичне виділення тепла (англ. thermal design power)
AMD	Advanced Micro Devices, Inc
SSD	Твердотільний накопичувач (англ. SSD, solid-state drive)
OpenMP	Open Multi-Processing
API	(Application Programming Interface) інтерфейс прикладного програмування

ВСТУП

Операції з комплексними матрицями є необхідними у багатьох галузях, включаючи фізику, математику, комп'ютерну графіку та програмування. Вектори та матриці відіграють ключову роль у вирішенні різноманітних завдань, від комп'ютерної графіки до військових розрахунків. Проте, виконання складних обчислень, пов'язаних з матрицями, може вимагати значного часу або використання високопродуктивних обчислювальних систем. На звичайному комп'ютері такі завдання можуть займати дні, тоді як багатоядерні системи здатні виконувати ті ж обчислення за лічені секунди. Для підвищення ефективності необхідно організувати паралельні обчислення та максимально використовувати доступні ресурси процесора.

Метою цього дипломного проекту є створення високопродуктивного програмно-апаратного комплексу для операцій з комплексними матрицями, спрямованого на значне скорочення часу виконання таких завдань. Для досягнення цієї мети поставлені наступні завдання:

- Дослідити апаратні компоненти, щоб визначити, який процесор та інші складові найкраще підходять для вирішення складних задач;
- Проаналізувати бібліотеки та мови програмування, які можуть бути використані для створення складних систем;
- Вибрати найбільш ефективне та універсальне програмне забезпечення для вирішення поставленої задачі;
- Розробити програму та проаналізувати її продуктивність при виконанні матричних операцій.

Досягнувши цих цілей, ця робота має на меті зробити внесок у розвиток високопродуктивних обчислювальних систем, здатних ефективно обробляти складні матричні операції - навички, необхідної для інженерних і наукових застосувань.

РОЗДІЛ 1. ОГЛЯД СТРУКТУР ТА СФЕР ЗАСТОСУВАННЯ ВИСОКОПРОДУКТИВНИХ КОМП'ЮТЕРНИХ СИСТЕМ

Сучасні ВПКС відіграють ключову роль у виконанні складних обчислень, особливо у сферах, де необхідно працювати з великою кількістю даних та виконувати паралельні обчислення. Ці системи дозволяють знижувати час виконання задач, підвищуючи їх ефективність та точність. У цьому розділі буде проведено огляд основних структур ВПКС, включаючи кластерні системи, багатоядерні системи та графічні процесори. Крім того, буде розглянуто різні сфери застосування високопродуктивних обчислювальних систем, зокрема, у галузі квантової механіки, машинного навчання, комп'ютерної графіки та інших наукових і промислових сферах.

1.1 Кластерні системи

Кластерні обчислювальні системи є однією з найпотужніших технологій, що дозволяє значно підвищити продуктивність обчислювальних процесів. Вони представляють собою колекцію незалежних комп'ютерів або вузлів, об'єднаних у мережу для роботи над спільними завданнями [1]. Вузли можуть бути фізичними машинами або віртуальними екземплярами на хмарній інфраструктурі. Основна мета кластерних систем — досягнення високої продуктивності і масштабованості шляхом розподілу обчислювального навантаження між кількома вузлами, що дозволяє виконувати паралельні обчислення і зменшити час виконання завдань.

1.1.1 Архітектура кластерних систем

Архітектура кластерних систем складається з декількох ключових компонентів, кожен з яких виконує свою специфічну функцію:

1) Вузли є основними елементами кластерної системи. Вони можуть бути фізичними серверами або віртуальними машинами, розташованими як у локальній мережі, так і у віддалених дата-центрах. Кожен вузол має власні процесори, оперативну пам’ять, накопичувачі даних і операційну систему, що дозволяє йому виконувати обчислювальні завдання самостійно або у взаємодії з іншими вузлами.
2) Високошвидкісна мережа є критично важливою для забезпечення ефективної комунікації між вузлами. Використовуються технології, такі як InfiniBand, RDMA або високошвидкісні Ethernet-з’єднання, які забезпечують низьку затримку і високу пропускну здатність передачі даних [2].
3) Балансувальники навантаження розподіляють вхідні запити або завдання серед доступних вузлів, запобігаючи перевантаженню окремих вузлів і забезпечуючи оптимальне використання ресурсів. Це дозволяє рівномірно розподілити обчислювальне навантаження і мінімізувати час очікування завдань.
4) Системи зберігання даних в кластерних системах забезпечують високошвидкісний доступ до даних, використовуючи розподілені файлові системи або бази даних. Це дозволяє забезпечити доступність даних для всіх вузлів кластеру і ефективно обробляти великі обсяги інформації.

Приклад архітектури кластерної системи зображено на рис. 1.1.

1.1.2 Переваги кластерних систем

Кластерні системи мають кілька важливих переваг:

1) Паралельна обробка в кластерних системах дозволяє значно скоротити час виконання складних обчислень. Наприклад, задачі, які потребують значних обчислювальних ресурсів, такі як моделювання фізичних процесів або аналіз великих обсягів даних, можуть бути виконані набагато швидше за рахунок розподілу навантаження між кількома вузлами [3].

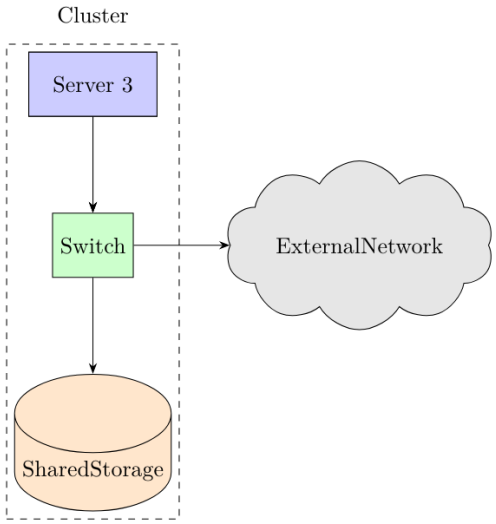


Рисунок 1.1 – Архітектура кластерної системи

2) Кластерні системи легко масштабуються, тобто можна додавати або видаляти вузли в кластері в залежності від поточних потреб. Це дозволяє адаптувати систему до змінних вимог і забезпечувати необхідний рівень продуктивності без значних витрат [4].
3) Завдяки розподіленню завдань серед кількох вузлів, система може продовжувати роботу навіть у разі виходу з ладу одного з них. Це забезпечує безперервність сервісу і мінімізує ризик втрати даних або простою системи.
4) Балансування навантаження та оптимізація використання ресурсів дозволяють уникати простоїв і забезпечувати максимальну продуктивність системи. Це особливо важливо в умовах високих обчислювальних навантажень і обмежених ресурсів.

1.1.3 Види кластерних систем

Кластерні системи поділяються на кілька видів залежно від їхнього призначення:

1) Обчислювальні кластери призначені для виконання великих обчислювальних завдань, таких як наукові обчислення, моделювання

фізичних процесів або обробка даних у реальному часі. Такі кластери зазвичай використовуються в наукових дослідженнях, інженерії та обробці великих обсягів даних [5].

2) Кластери зберігання використовуються для управління великими обсягами даних та забезпечення високої швидкості доступу до них. Це можуть бути розподілені файлові системи або бази даних, які забезпечують збереження і доступ до даних для всіх вузлів кластеру.
3) Кластери високої доступності забезпечують безперервну роботу критично важливих додатків, мінімізуючи час простою. Такі кластери використовуються для підтримки сервісів, які повинні бути доступними 24/7, наприклад, у фінансових установах, медичних закладах або в електронній комерції.

1.1.4 Застосування кластерних систем

Кластерні системи використовуються для виконання великих обчислювальних завдань, таких як моделювання клімату, біологічні дослідження, обробка астрономічних даних, геномні дослідження і моделювання фізичних явищ. Висока продуктивність і паралельна обробка дозволяють науковцям проводити складні розрахунки і аналізувати великі обсяги даних за короткий час.

Кластерні системи застосовуються для аналізу великих обсягів фінансових даних, прогнозування ризиків, алгоритмічної торгівлі та обробки транзакцій у реальному часі. Це дозволяє фінансовим установам швидко реагувати на зміни на ринку і приймати обґрунтовані рішення.

У галузі індустрії медіа та розваг кластерні системи корисні для обробки відео, рендерингу графіки, проведення симуляцій у віртуальній реальності та інших обчислювально інтенсивних завдань. Висока продуктивність кластерів дозволяє значно скоротити час обробки медіа-контенту і підвищити якість кінцевого продукту.

У медицині кластерні системи використовуються для обробки медичних зображень, моделювання біологічних процесів, аналізу геномних даних та інших обчислювальних завдань. Це дозволяє медичним працівникам отримувати точні і швидкі результати, що сприяє покращенню діагностики і лікування пацієнтів.

У промисловості кластерні системи застосовуються для оптимізації виробничих процесів, моделювання складних механічних систем, аналізу великих обсягів даних з виробничих ліній та інших задач. Це дозволяє підвищити ефективність виробництва і зменшити витрати.

Таким чином, кластерні системи є ефективним рішенням для реалізації високопродуктивних обчислень у різних галузях, що дозволяє суттєво підвищити продуктивність і надійність обчислювальних процесів. Вони забезпечують можливість виконання складних обчислювальних завдань у найкоротші терміни, підвищуючи ефективність і конкурентоспроможність організацій та наукових установ.

1.2 Багатоядерні системи

Багатоядерні системи стали невід'ємною частиною сучасних комп'ютерних архітектур, значно підвищуючи продуктивність і ефективність обчислень. Їх поява була обумовлена фізичними обмеженнями одноядерних процесорів, які досягли межі своїх можливостей щодо підвищення швидкості та зменшення розмірів транзисторів. Основна ідея багатоядерних систем полягає у використанні декількох ядер на одному процесорі для паралельної обробки завдань, що дозволяє значно прискорити виконання складних обчислювальних задач.

1.2.1 Архітектура багатоядерних процесорів

Багатоядерний процесор являє собою інтегральну схему, яка містить два або більше ядер, що взаємодіють між собою. Кожне ядро здатне виконувати незалежні обчислення, що дозволяє обробляти кілька задач одночасно [6]. Наприклад, під час перегляду відео та одночасного використання іншого додатку, один процесор може обробляти відеопотік, тоді як інший займатиметься іншими завданнями.

Архітектура багатоядерних процесорів передбачає ефективну комунікацію між ядрами, що дозволяє розподіляти завдання та обмінюватися обробленими даними через спільну шину даних (рис. 1.2). Це забезпечує високу швидкість і ефективність обчислень порівняно з одноядерними процесорами. Крім того, сучасні багатоядерні процесори включають додаткові компоненти, такі як спільний кеш пам'яті, який забезпечує швидкий доступ до часто використовуваних даних, і контролери пам'яті, що дозволяють ефективно керувати доступом до оперативної пам'яті.

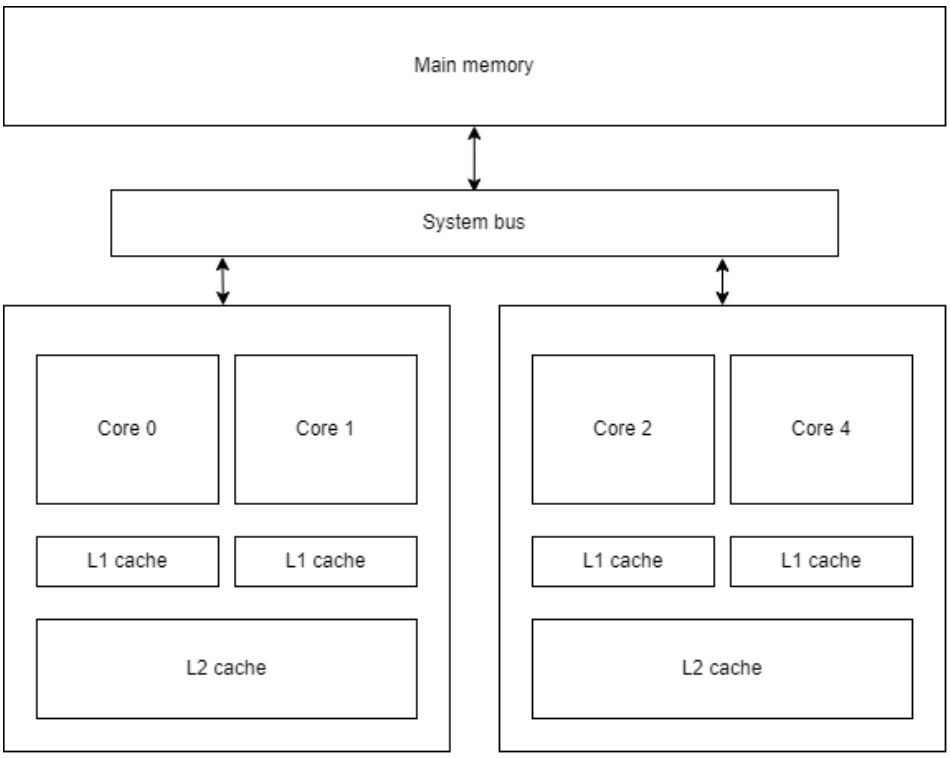


Рисунок 1.2 – Схема архітектури багатоядерного процесора

1.2.2 Переваги багатоядерних систем

Багатоядерні процесори забезпечують вищу продуктивність завдяки можливості одночасного виконання кількох потоків завдань. Це особливо важливо для додатків, які вимагають високих обчислювальних ресурсів, таких як обробка відео, наукові розрахунки та моделювання. Наприклад, в області комп'ютерної графіки та обробки відео багатоядерні процесори дозволяють значно прискорити рендеринг зображень і обробку відеопотоків.

Використання декількох ядер дозволяє знижувати енергоспоживання, оскільки кожне ядро працює на нижчих частотах, що зменшує тепловиділення та підвищує загальну енергоефективність системи. Це особливо важливо для мобільних пристроїв та серверів, де ефективне використання енергії має критичне значення.

Завдяки розподілу завдань між кількома ядрами, багатоядерні системи є більш надійними. У разі збою одного ядра, інші ядра можуть продовжувати роботу, що підвищує стійкість системи до помилок. Це особливо важливо для критичних систем, де надійність і безперервність роботи є ключовими вимогами.

1.2.3 Виклики та проблеми багатоядерних систем

Розробка програмного забезпечення для багатоядерних систем вимагає спеціалізованих знань та навичок. Програмісти повинні забезпечити ефективну синхронізацію потоків і уникати станів гонки, що ускладнює процес розробки. Це потребує використання спеціальних інструментів і методологій для розробки паралельних програм, таких як OpenMP, MPI та інші.

Не всі задачі можуть бути легко розділені на паралельні потоки. Для досягнення максимальної продуктивності необхідно оптимізувати розподіл завдань між ядрами, що часто є складним завданням. Деякі обчислювальні

задачі мають високу залежність між етапами обробки, що ускладнює їх паралелізацію.

Швидка і ефективна комунікація між ядрами є ключовим фактором для досягнення високої продуктивності. Проблеми зі взаємодією між ядрами можуть значно знижувати ефективність системи. Це включає проблеми з управлінням кеш-пам'яттю, синхронізацією доступу до спільних ресурсів і оптимізацією пропускної здатності шини даних.

1.2.4 Підсумок щодо багатоядерних систем

Багатоядерні системи є важливим кроком у розвитку комп'ютерних технологій, що дозволяють значно підвищити продуктивність і ефективність обчислень. Вони мають численні переваги, такі як висока продуктивність, енергоефективність та надійність, але також потребують вирішення низки технічних викликів, пов'язаних з програмуванням і оптимізацією використання ресурсів.

У контексті розробки високопродуктивного ПАК для операцій з комплексними матрицями, використання багатоядерних систем дозволить значно прискорити обробку великих обсягів даних та підвищити ефективність обчислень. Зокрема, паралельна обробка матричних операцій може забезпечити значне скорочення часу обробки і покращити продуктивність системи в цілому.

1.3 Графічні процесори

GPU відіграють ключову роль у високопродуктивних обчислювальних системах, забезпечуючи значні переваги у паралельних обчисленнях. На відміну від CPU, які оптимізовані для послідовних обчислень, GPU призначені для виконання великої кількості простих операцій паралельно, що робить їх ідеальними для задач, які можна розбити на безліч незалежних частин.

Основною відмінністю GPU від GPU є їх архітектура. Графічні процесори складаються з великої кількості ядер, які здатні обробляти дані одночасно (рис. 1.3). Це дозволяє GPU ефективно виконувати масово-паралельні обчислення. Наприклад, сучасні GPU можуть містити тисячі ядер, що дозволяє їм виконувати мільйони операцій одночасно [8].

Використання GPU у високопродуктивних обчисленнях дозволяє досягти значного прискорення для таких задач, як обробка зображень, моделювання фізичних процесів, аналіз великих обсягів даних та машинне навчання. Наприклад, NVIDIA, один з провідних виробників графічних процесорів, активно розвиває рішення для ВПО та штучного інтелекту, що дозволяє зменшити час на обчислення з місяців до днів або навіть годин [9][10].

Одним з ключових аспектів використання GPU є їх здатність до масово-паралельних обчислень. Це означає, що завдання, які можуть бути розподілені на безліч незалежних частин, можуть виконуватись набагато швидше на GPU, ніж на традиційних CPU.

GPU використовуються для швидкої обробки та рендерингу графіки, що необхідно в ігровій індустрії, а також для обробки відео у реальному часі.

Моделювання фізичних процесів, таких як динаміка рідин та кліматичні моделі, виграють від використання GPU завдяки їх здатності обробляти великі обсяги даних паралельно.

Тренування нейронних мереж включає обробку великих обсягів даних, що робить GPU незамінним інструментом для цієї задачі. Завдяки високій пропускній здатності пам'яті та здатності обробляти мільйони паралельних операцій, GPU значно прискорюють процес навчання моделей.

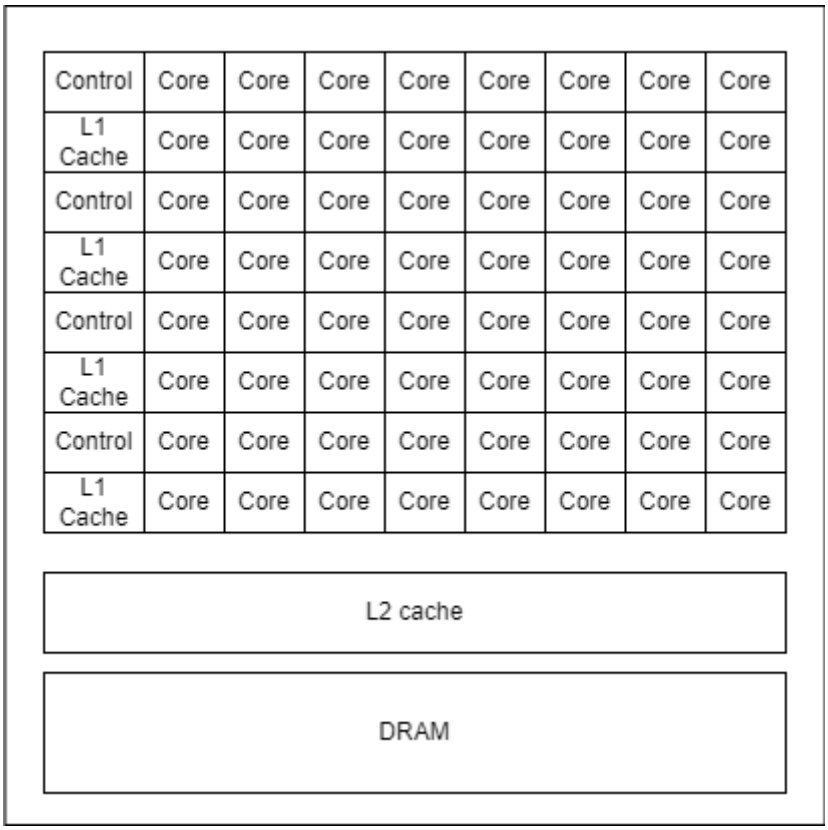


Рисунок 1.3 – Схема архітектури графічного процесора

Для ефективного використання GPU необхідні спеціалізовані програмні інструменти та платформи. Серед найбільш популярних можна виділити:

- CUDA: Програмна платформа від NVIDIA, що дозволяє розробникам писати паралельні програми для графічних процесорів. CUDA включає в себе компілятори, бібліотеки та інструменти для оптимізації обчислень [11].
- OpenCL: Відкрита платформа для розробки програм, що можуть виконуватись на різних типах процесорів, включаючи GPU. OpenCL забезпечує універсальність та сумісність з різними виробниками апаратного забезпечення [12].

Графічні процесори є невід'ємною частиною сучасних високопродуктивних обчислювальних систем. Їх архітектура дозволяє ефективно виконувати масово-паралельні обчислення, що відкриває нові

можливості для різноманітних наукових та інженерних задач. Використання GPU у поєднанні з спеціалізованим програмним забезпеченням, таким як CUDA та OpenCL, дозволяє досягти значного прискорення обчислювальних процесів, що є критично важливим для сучасних наукових досліджень та інновацій.

1.4 Сфери застосування множення комплексних матриць

1.4.1 Використання у квантовій механіці

У квантовій механіці множення комплексних матриць відіграє центральну роль у математичному описі квантових систем. Квантові стани системи представлені векторами у гільбертовому просторі, а оператори, що діють на ці стани, часто представлені у вигляді матриць. Наприклад, оператор Гамільтона, який визначає енергетичні стани системи, є матрицею, що взаємодіє з вектором стану. Рівняння Шредінгера, яке описує еволюцію квантового стану у часі, можна записати в матричній формі, що робить обчислення ефективнішими за допомогою матричного множення [19].

Комплексні числа, які складають елементи цих матриць, дозволяють квантовій механіці моделювати суперпозиції станів і інтерференційні ефекти, що є суттєвими для точного опису квантових явищ. Обчислення ймовірностей, пов'язаних із вимірюванням різних квантових станів, також базується на множенні комплексних матриць. Наприклад, ймовірність переходу системи з одного стану в інший залежить від матричних елементів операторів переходу.

Діагоналізація матриць у квантовій механіці є важливим процесом для знаходження власних значень і власних векторів, які представляють можливі результати вимірювань фізичних величин і відповідні квантові стани. Це є критично важливим для аналізу спектрів енергій систем і визначення динаміки квантових процесів.

Симуляції квантових систем, що використовують множення комплексних матриць, дозволяють вченим моделювати поведінку матеріалів на атомному рівні, досліджувати хімічні реакції і розробляти нові квантові технології, включаючи квантові комп’ютери.

1.4.2 Використання у цифровій обробці сигналів

Цифрова обробка сигналів є однією з ключових сфер, де множення комплексних матриць відіграє критичну роль. У ЦОС основними завданнями є аналіз, модифікація та синтез сигналів, які можуть бути аудіо, відео чи іншими формами даних. Комплексні матриці та операції з ними використовуються для реалізації таких важливих алгоритмів, як швидке перетворення Фур'є, фільтрація та модуляція сигналів

FFT є фундаментальним алгоритмом у ЦОС, який дозволяє швидко і ефективно перетворювати сигнал з часової області в частотну і навпаки. Це перетворення базується на множенні комплексних чисел і матриць, що робить процеси аналізу та обробки сигналів значно швидшими. В обробці зображень FFT використовується для стиснення даних, видалення шуму та відновлення зображень.

Множення комплексних матриць також використовується в адаптивних фільтрах, які застосовуються для видалення шуму з сигналів у реальному часі. Адаптивні фільтри автоматично налаштовуються відповідно до характеристик сигналу та шуму, що дозволяє ефективно виділяти корисний сигнал. Ці фільтри є невід'ємною частиною багатьох комунікаційних систем, медичних приладів і систем обробки аудіо та відео.

Модуляція сигналів, яка використовується для передачі даних через різні комунікаційні канали, також базується на операціях з комплексними числами. Такі методи модуляції, як квадратурна амплітудна модуляція і ортогональна частотна модуляція, широко застосовуються в сучасних системах зв'язку, включаючи мобільні телефони, Wi-Fi та інші бездротові технології [20].

Цифрова обробка сигналів знаходить застосування в численних галузях. У медичній сфері, наприклад, ЦОС використовується для обробки електрокардіограм та магнітно-резонансних зображень. У промисловості ЦОС застосовується для контролю та моніторингу процесів у реальному часі. У сфері розваг і мультимедіа ЦОС забезпечує обробку аудіо та відео для створення високоякісного контенту.

1.4.3 Застосування в наукових дослідженнях

У сфері обчислювальної фізики, множення комплексних матриць використовується для моделювання фізичних систем. Зокрема, у теорії електромагнітного поля, електродинаміки та інших областях фізики обчислення з використанням комплексних чисел дозволяють описати хвильові процеси, розповсюдження електромагнітних хвиль та інші явища. Такі моделі допомагають вченим краще розуміти поведінку складних фізичних систем і передбачати їхню еволюцію.

У хімії та біології, особливо в молекулярній динаміці, множення комплексних матриць використовується для аналізу структур біомолекул і їхніх взаємодій. При моделюванні білків і нуклеїнових кислот, матричні обчислення допомагають визначити енергетичні мінімуми, передбачити структуру молекул і їхню стабільність. Такі обчислення є критично важливими для розробки нових лікарських препаратів і вивчення біологічних процесів на молекулярному рівні.

Також важливим напрямком застосування є астрофізика, де комплексні матриці використовуються для аналізу даних, отриманих з телескопів та інших астрономічних приладів. Для обробки сигналів від радіотелескопів і дослідження космічного випромінювання застосовуються методи, засновані на множенні комплексних матриць. Це дозволяє астрономам вивчати структуру і динаміку космічних об'єктів, аналізувати спектри випромінювання і досліджувати фундаментальні властивості Всесвіту.

Комплексні матриці також відіграють важливу роль у сфері фінансової математики, де вони використовуються для моделювання та аналізу фінансових ринків. Зокрема, у стохастичних моделях, які описують рух цін на фінансові інструменти, використовуються комплексні числа для визначення ймовірностей та ризиків. Це допомагає фінансовим аналітикам приймати обґрунтовані рішення щодо інвестицій і управління ризиками.

Завдяки своїй універсальності та ефективності, комплексні матриці є незамінним інструментом у багатьох галузях науки та техніки. Їх використання забезпечує високий рівень точності та продуктивності в обчисленнях, що є критичним для вирішення складних задач і досягнення нових наукових та інженерних висот.

ВИСНОВОК ДО РОЗДІЛУ 1

Проведений детальний огляд структур високопродуктивних комп'ютерних систем, який показав що вони базуються на використанні кластерних системи, багатоядерних процесорах та графічних процесорах.

Кластерні системи складаються з кількох обчислювальних вузлів, з'єднаних мережею для досягнення високої продуктивності. Вони дозволяють розподіляти обчислювальні завдання між вузлами, що підвищує ефективність та скорочує час виконання складних задач. Кластери широко використовуються в наукових дослідженнях, обробці великих даних та моделюванні фізичних процесів завдяки їх гнучкості та масштабованості.

Багатоядерні процесори є основою сучасних обчислювальних систем. Вони дозволяють виконувати декілька потоків інструкцій одночасно, що значно підвищує продуктивність системи. Використання багатоядерних процесорів є особливо ефективним для задач, що можуть бути паралелізовані, таких як обробка зображень, машинне навчання та наукові обчислення. Розвиток багатоядерних технологій від таких виробників, як Intel та AMD, продовжує сприяти зростанню продуктивності комп'ютерних систем.

Графічні процесори спеціалізовані на паралельних обчисленнях і призначені для виконання великої кількості простих операцій одночасно. Завдяки їхній архітектурі з тисячами ядер, GPU можуть значно прискорити обчислювальні завдання, що потребують масово-паралельної обробки, включаючи обробку зображень, моделювання фізичних процесів та машинне навчання. Інструменти та платформи, такі як CUDA від NVIDIA та OpenCL, надають можливості для розробки програмного забезпечення, що використовує переваги GPU.

Також було розглянуто застосування множення комплексних матриць у різних сферах. Ці операції знаходять широке використання у наукових дослідженнях, техніці та промисловості. Зокрема, у квантовій механіці,

обчислювальній фізиці, хімії, біології та астрофізиці матричні обчислення допомагають моделювати складні системи, аналізувати структури та передбачати їх поведінку. У фінансовій математиці вони використовуються для моделювання та аналізу фінансових ринків, що сприяє прийняттю обґрунтованих рішень щодо інвестицій та управління ризиками.

Таким чином, розглянуті у першому розділі структури високопродуктивних комп'ютерних систем демонструють різні підходи до підвищення продуктивності обчислень. Кожна з цих структур має свої унікальні переваги та сфери застосування, що дозволяє вирішувати широкий спектр задач від наукових досліджень до обробки великих даних та машинного навчання.

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМНО-АПАРАТНОГО КОМПЛЕКСУ

Для створення високопродуктивного програмно-апаратного комплексу для операцій з комплексними матрицями необхідно ретельно підійти до вибору як апаратної, так і програмної складової системи. Основні компоненти, які визначають продуктивність ПАК, включають центральний процесор, оперативну пам’ять, графічний процесор, материнську плату, накопичувачі та блок живлення. Вибір мови програмування також відіграє ключову роль, оскільки він впливає на ефективність реалізації алгоритмів, зручність розробки та підтримки програмного забезпечення, а також на можливість використання паралельних обчислень для досягнення максимальної продуктивності.

2.1 Розробка апаратної складової

При розробці високопродуктивного ПАК для операцій з комплексними матрицями ми керуємося обмеженим бюджетом у розмірі 100 тис. грн. Наша мета полягає в максимізації продуктивності та ефективності системи, враховуючи цільові характеристики дипломного проекту. У цьому контексті обираються компоненти, що найкращим чином відповідають вимогам та потребам нашого проекту, з урахуванням фінансових обмежень.

2.1.1 Порівняння та вибір процесора ПАК

Intel Corporation і Advanced Micro Devices, Inc. (AMD) - це дві з найбільших та впливових компаній у сфері виробництва процесорів і комп'ютерної техніки.

Intel є однією з найстаріших компаній у галузі, заснованою в 1968 році. Вона спеціалізується на виробництві широкого спектру процесорів для різних

ринків, включаючи серверні, настільні та мобільні пристрої. Intel відома своїми інноваціями та технологічними розробками, такими як архітектура x86, яка використовується в більшості комп'ютерів по всьому світу [13].

AMD, заснована в 1969 році, також відома своїми процесорами, конкуруючи з Intel у багатьох сегментах ринку. Компанія випускає широкий спектр процесорів для різних потреб, включаючи високопродуктивні процесори для ігор і серверів, а також енергоефективні рішення для мобільних і вбудованих пристроїв. AMD відома своєю архітектурою Zen, яка отримала позитивні відгуки за продуктивність і вартість [14].

Створимо таблицю характеристик процесорів у таблиці 2.1.

Таблиця 2.1 Характеристики процесорів для ПАК.

Параметр	AMD Ryzen Threadripper PRO 7995WX	AMD Ryzen 9 7950X	Intel Xeon Platinum 8470	Intel Core i9-14900KS
Вартість, грн	527,353	23,899	370,499	32,999
Роз’єм	sTR5	AM5	FCLGA4677	FCLGA1700
Базова частота, ГГц	2.5	4.5	2.0	3.2
Турбо частота, ГГц	5.1	5.7	3.8	6.2
Кількість ядер	96	16	52	24
Кількість потоків	192	32	104	32
TDP, Вт	350	170	350	150

AMD Ryzen Threadripper PRO 7995WX виділяється надзвичайною кількістю ядер та потоків (96/192), що робить його ідеальним для надважких паралельних обчислень. Високий обсяг кешу (L3 384 МБ) додатково підвищує

продуктивність при роботі з великими обсягами даних. Однак, висока ціна (527,353₴) робить його менш привабливим з точки зору співвідношення ціна/якість.

Ryzen 9 7950X з 16 ядрами та 32 потоками, цей процесор має високі тактові частоти (до 5.7 ГГц) і значно менший TDP (170 Вт). Він є чудовим варіантом для менших бюджетів, забезпечуючи відмінну продуктивність за відносно низьку ціну (23,899₴).

Intel Xeon Platinum 8470 підходить для серверних та обчислювальних систем з великим обсягом оперативної пам'яті (до 4 ТБ). Його 52 ядра та 104 потоки забезпечують високу продуктивність, але висока ціна (370,499₴) та порівняно низькі тактові частоти можуть бути обмеженням для деяких задач.

Intel Core i9-14900KS має 24 ядра (8 високопродуктивних і 16 енергоефективних) (рис. 2.1), що дозволяє досягати високих частот (до 6.2 ГГц). Він забезпечує відмінне співвідношення ціна/якість (32,999₴), з помірним TDP (150 Вт типове, 253 Вт макс.).

Рисунок 2.1 – Структура процесора Intel Core i9-14900KS

З урахуванням різних факторів, таких як кількість ядер, частоти, TDP та ціна, найкращим варіантом для створення високопродуктивного програмно-апаратного комплексу є Intel Core i9-14900KS. Він забезпечує високу продуктивність завдяки поєднанню високопродуктивних та енергоефективних ядер, відмінні тактові частоти та доступну ціну. Цей процесор є оптимальним

вибором для завдань, що потребують інтенсивних обчислень, та забезпечує найкраще співвідношення ціни до якості та ефективності.

Процесор Intel Core i9-14900KS демонструє видатні результати у різних тестах продуктивності. Зокрема, він досягає таких показників у обчислювальних операціях [15]:

- Операції з цілими числами: 222,865 операцій в пам'яті / сек;
- Операції з плаваючою комою: 157,544 операцій в пам'яті / сек;
- Пошук простих чисел: 267 мільйонів простих чисел/сек;
- Випадкове сортування рядків: 97,770 тисяч рядків/сек;
- Шифрування даних: 52 341 МБ/сек;
- Стиснення даних: 847 048 кБ/сек;

Вибір процесора значно впливатиме на загальну продуктивність системи, тому правильний вибір допоможе досягти максимальної ефективності при обробці комплексних матричних операцій.

2.1.2 Розробка структури ПАК

Наш бюджет для розробки ПАК складає 100 тис. грн. Цей бюджет обмежує наш вибір комплектуючих, проте ми прагнемо максимально оптимізувати його використання, щоб забезпечити максимальну продуктивність системи за доступними коштами. У цьому контексті обираються компоненти, що найкращим чином відповідають вимогам та потребам нашого проекту, з урахуванням фінансових обмежень.

Для початку розглянемо процесор. Обираючи Intel Core i9-14900KS (рис.2.2), ми беремо до уваги його високі характеристики, такі як кількість ядер, частоти роботи, а також співвідношення ціни та якості. Цей процесор забезпечить високу продуктивність і ефективність обчислень з оптимальним використанням бюджету.

Рисунок 2.2 – Процесор Intel Core i9-14900KS

Материнська плата Gigabyte Z790 AORUS ELITE X WIFI7 (рис.2.3) обрана за сумісністю з роз’ємом процесора LGA1700, за можливість підтримувати високу тактову частоту оперативної пам'яті до 8200 МГц, наявністю трьох роз'ємів M.2 для встановлення швидкісних запам’ятовуючих пристроїв та три роз'єми PCIe 5, що надає можливість у розширенні системи ПАК у майбутньому.

В ролі системи охолодження процесора було обрано кулер Deepcool AG620 (рис.2.4), оснащений двома вентиляторами та здатним працювати з TDP до 260 Вт. Цей кулер забезпечить стабільну роботу процесора навіть при високих навантаженнях, забезпечуючи оптимальну температуру роботи

Рисунок 2.3 – Материнська плата Gigabyte Z790 AORUS ELITE X WIFI7

Рисунок 2.4 – Кулер Deepcool AG620

Оперативна пам'ять була обрана з урахування потреби у швидкості. Patriot Memory Viper Venom DDR5 (рис.2.5) відзначається не лише високою тактовою частотою 7400 МГц, але і вражаючою ефективністю у роботі з великими обсягами даних. Ця швидкість дозволить нашій системі миттєво реагувати на

завдання та операції з комплексними матрицями, забезпечуючи швидку та ефективну роботу без затримок.

Рисунок 2.5 – Оперативна пам'ять Patriot Memory Viper Venom DDR5

Потреба у графічному процесорі є не такою великою, оскільки основне навантаження буде на центральний процесор. Отже з урахуванням цього, було обрано ГП MSI GeForce GTX 1650 D6 VENTUS XS OCV3 (рис.2.6). Його невелика ціна, енергоефективність та ефективна система охолодження дозволить зберегти високу продуктивність системи протягом тривалого часу.

Рисунок 2.6 – ГП MSI GeForce GTX 1650 D6 VENTUS XS OCV3

Для ефективного та надійного зберігання даних у системі, було обрано SSD Samsung 990 PRO MZ-V9P1T0BW (рис.2.7), з великим обсягом пам'яті та високою швидкістю читання та запису, що забезпечить швидкий доступ до даних та забезпечить стабільну роботу системи.

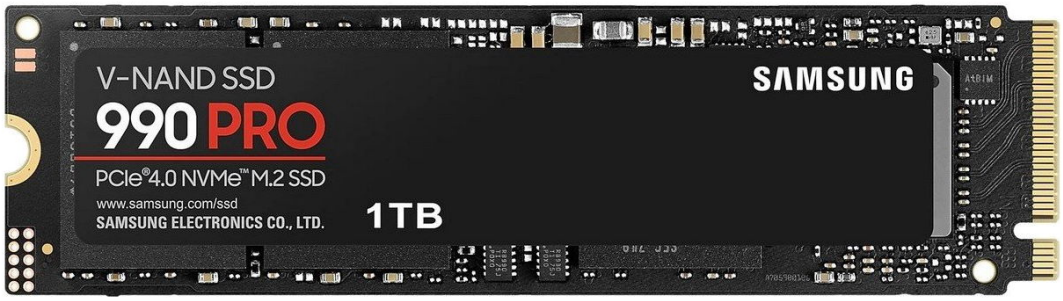


Рисунок 2.7 – SSD Samsung 990 PRO MZ-V9P1T0BW

Блок живлення обирався з розрахунком потужності в декілька разів більшим, ніж потрібно, що надасть можливість встановлення потужніших аналогів компонентів, та вдосконалення системи у майбутньому. Для цього нам підійде Chieftec Polaris PPS-650FC (рис.2.8), зі сертифікацією енергоефективності 80 PLUS Gold, що забезпечить стабільне живлення всіх компонентів системи. Його ефективність дозволить економити електроенергію та знижувати тепловиділення.

Рисунок 2.8 – Блок живлення Chieftec Polaris PPS-650FC

Корпус Cougar Uniface (рис.2.9), з відмінною проходимістю повітря та достатньою кількістю місця для встановлення комплектуючих, забезпечить оптимальну вентиляцію та охолодження всіх складових системи. Його практичний та ергономічний дизайн робить його ідеальним вибором для нашого високопродуктивного ПАК.

Отже, після підбору всіх комплектуючих ми можемо порахувати ціну ПАК. Ціни на кожен компонент вказано у таблиці 2.2.

Рисунок 2.9 – Корпус Cougar Uniface

Таблиця 2.2 Апаратна складова програмно-апаратного комплексу.

Назва компоненту	Бренд та модель	Ціна, грн
Процесор	Intel Core i9-14900KS	32,999
Материнська плата	Gigabyte Z790 AORUS ELITE X WIFI7	14,199
Кулер	Deepcool AG620	2,589
Оперативна пам’ять	Patriot Memory Viper Venom DDR5 2x16Gb 7400 МГц	5,832
Графічний процесор	MSI GeForce GTX 1650 D6 VENTUS XS OCV3	6,799
SSD	Samsung 990 PRO MZ-V9P1T0BW	5,599
Блок живлення	Chieftec Polaris PPS-650FC	3,550
Корпус	Cougar Uniface	2,899

Програмно-апаратний комплекс було зібрано з врахуванням обмеження бюджету у 100 тис. гривень. Загальна вартість комплектуючих складає 74,466 грн, що значно менше нашого бюджету, залишаючи нам простір для можливих надбудов та модифікацій.

2.2 Огляд і вибір засобів для створення програмного забезпечення ПАК

Вибір правильного інструментарію для розробки програмного забезпечення є критичним етапом у створенні високопродуктивного ПАК. Враховуючи специфіку обчислень з комплексними матрицями, необхідно ретельно оцінити можливості різних мов програмування та їх підтримку паралельних обчислень.

2.2.1 Java

Java – це об'єктно-орієнтована мова програмування загального призначення, яка широко використовується для розробки різноманітних застосунків, включаючи наукові та високопродуктивні обчислення. Java має вбудовану підтримку багатопотоковості, що робить її привабливим вибором для реалізації паралельних алгоритмів [16].

Завдяки віртуальній машині Java, програми, написані на цій мові, можуть виконуватися на різних операційних системах і апаратних платформах без необхідності внесення змін до коду.

Багата екосистема мови Java має величезну кількість бібліотек і фреймворків, що спрощує розробку та прискорює процес створення програмного забезпечення.

Автоматичне керування пам'яттю, що включає збирач сміття Java автоматично звільняє пам'ять, що не використовується, зменшуючи ризик виникнення помилок, пов'язаних з керуванням пам'яттю.

Java надає засоби для створення та управління потоками, що дозволяє ефективно використовувати багатоядерні процесори та розподілені системи.

Java підтримує багатопотоковість через клас Thread та інтерфейс Runnable. Для створення нового потоку можна створити клас, що успадковується від Thread, або реалізувати інтерфейс Runnable. Потім об'єкт цього класу передається конструктору Thread, і потік запускається методом start(). Приклад коду ініціалізації потоку показано на рис. 2.10.

```
public class MyThread extends Thread {
    public void run() {
        // Код, який виконуватиметься в окремому потоці
    }
}

MyThread thread = new MyThread();
thread.start();
```

Рисунок 2.10 – Ініціалізація потоку в мові Java

Java надає різноманітні засоби для організації взаємодії між потоками, включаючи:

- Синхронізовані блоки та методи (synchronized) забезпечують взаємне виключення, дозволяючи лише одному потоку одночасно виконувати критичну секцію коду.
- Очікування та сповіщення (wait, notify, notifyAll) дозволяють потокам чекати на певні умови та сповіщати інші потоки про зміну стану.
- Блокування (Lock, ReentrantLock) надають більш гнучкі механізми синхронізації, ніж ключове слово synchronized.
- Семафори (Semaphore) обмежують кількість потоків, які можуть одночасно отримати доступ до ресурсу.
- Бар'єри (CyclicBarrier) дозволяють групі потоків чекати один на одного в певній точці виконання.
- Атомарні змінні (AtomicInteger, AtomicLong тощо) забезпечують атомарні операції над змінними, запобігаючи race conditions.

- Паралельні колекції (ConcurrentHashMap, ConcurrentLinkedQueue тощо) надають потокобезпечні реалізації колекцій для використання в багатопотокових застосунках.

Ці інструменти дозволяють розробникам ефективно координувати роботу потоків, синхронізувати доступ до спільних ресурсів та забезпечувати коректність виконання паралельних алгоритмів.

Java є потужною та універсальною мовою програмування, яка добре підходить для вирішення поставленої нами задачі. Її багатий набір інструментів для багатопотоковості та організації взаємодії потоків дозволяє створювати ефективні та масштабовані паралельні застосунки для обробки комплексних матриць.

2.2.2 C++ (OpenMP)

C++ – це потужна мова програмування загального призначення, яка широко використовується для розробки високопродуктивних застосунків, системного програмного забезпечення, ігор та наукових обчислень. C++ надає низькорівневий доступ до апаратних ресурсів та дозволяє ефективно керувати пам'яттю, що робить його привабливим вибором для реалізації обчислювальних задач, що вимагають високої продуктивності.

OpenMP – це API для паралельного програмування на багатопроцесорних системах з розділеною пам'яттю. OpenMP надає набір директив, функцій та змінних оточення, які дозволяють легко розпаралелювати цикли та інші блоки коду, не вдаючись до складного програмування з використанням потоків [17].

C++ дозволяє досягти високої продуктивності завдяки низькорівневому доступу до апаратних ресурсів та можливості оптимізації коду.

OpenMP підтримується багатьма компіляторами та операційними системами, що забезпечує хорошу портативність програмного забезпечення.

OpenMP надає простий та інтуїтивно зрозумілий спосіб розпаралелювання коду за допомогою директив.

OpenMP дозволяє комбінувати різні моделі паралелізму, такі як розпаралелювання циклів, секцій та задач.

OpenMP дозволяє легко розпаралелювати цикли за допомогою директиви #pragma omp parallel for. Ця директива автоматично розподіляє ітерації циклу між доступними потоками, забезпечуючи ефективне використання багатоядерних процесорів. Приклад ініціалізації потоків у C++ зображено на рис. 2.11.

OpenMP надає різноманітні засоби для організації взаємодії між потоками, включаючи:

- Критичні секції (#pragma omp critical) забезпечують взаємне виключення, дозволяючи лише одному потоку одночасно виконувати критичну секцію коду.
- Бар'єри (#pragma omp barrier) дозволяють групі потоків чекати один на одного в певній точці виконання.
- Редукції (#pragma omp parallel for reduction) дозволяють виконувати операції згортки (наприклад, суму, добуток) над даними, оброблюваними різними потоками.
- Блокування (omp_lock_t, omp_set_lock, omp_unset_lock) надають низькорівневі механізми синхронізації для явного керування доступом до спільних ресурсів.
- Атомарні операції (#pragma omp atomic) забезпечують атомарне виконання операцій над змінними, запобігаючи стану перегонів у потоках.

```
#include <omp.h>

int main() {
    #pragma omp parallel num_threads(4) // Створення 4 потоків
    {
        int thread_id = omp_get_thread_num();
        printf("Hello from thread %d\n", thread_id);
    }
    return 0;
}
```

Рисунок 2.11 – Ініціалізація потоків у мові C++

C++ є потужним інструментом для розробки високопродуктивних ПАК для операцій з комплексними матрицями. Його висока продуктивність, портативність та простота використання роблять його привабливим вибором для цього завдання.

2.2.3 C#

C# – це сучасна об'єктно-орієнтована мова програмування, розроблена компанією Microsoft [18]. C# є однією з основних мов програмування платформи .NET і широко використовується для розробки різноманітних застосунків, включаючи веб-застосунки, десктопні програми, мобільні застосунки та ігри. C# має потужні засоби для роботи з багатопотоковістю, що робить його привабливим вибором для реалізації паралельних алгоритмів.

C# має чистий та виразний синтаксис, що полегшує читання та написання коду. Мова тісно інтегрована з платформою .NET, що надає доступ до величезної кількості бібліотек та інструментів для розробки. Як і Java, C# має збирач сміття, який автоматично звільняє пам'ять, що не використовується.

C# надає багатий набір інструментів для роботи з багатопотоковістю, включаючи класи Thread, Task, Parallel та інші.

Клас Thread дозволяє створювати та керувати потоками виконання. Приклад ініціалізації вказано на рис. 2.12.

Клас Task представляє асинхронну операцію і спрощує роботу з багатопотоковістю, надаючи високорівневі абстракції.

Клас Parallel надає методи для розпаралелювання циклів for та foreach, а також для паралельного виконання запитів LINQ.

```
using System.Threading;

Thread thread = new Thread(() => {
    // Код, який виконуватиметься в окремому потоці
    for (int i = 0; i < 10; i++) {
        // Обчислення для потоку
    }
});
thread.Start();
```

Рисунок 2.12 – Ініціалізація потоку в мові C#

C# пропонує різноманітні механізми для координації та синхронізації потоків, включаючи:

- Ключове слово lock забезпечує взаємне виключення, дозволяючи лише одному потоку одночасно виконувати критичну секцію коду.
- Монітори (Monitor) надають більш низькорівневий контроль над синхронізацією, ніж ключове слово lock.
- Очікування та сповіщення (Wait, Pulse, PulseAll) дозволяють потокам чекати на певні умови та сповіщати інші потоки про зміну стану.
- Семафори (SemaphoreSlim) обмежують кількість потоків, які можуть одночасно отримати доступ до ресурсу.
- Бар'єри (Barrier) дозволяють групі потоків чекати один на одного в певній точці виконання.
- Атомарні операції (Interlocked) забезпечують атомарне виконання операцій над змінними, запобігаючи стану перегонів у потоках.
- Паралельні колекції (ConcurrentDictionary, ConcurrentQueue тощо) надають потокобезпечні реалізації колекцій для використання в багатопотокових застосунках.

- Завдання (Task) є легкою абстракцією для представлення асинхронних операцій, що спрощує роботу з багатопотоковістю.

Зручність мови C#, її простота та багатий набір інструментів для багатопотоковості роблять її гарним вибором для цього завдання.

2.2.4 Python

Python – це високорівнева інтерпретована мова програмування загального призначення, яка відома своєю простотою, читабельністю та великою кількістю бібліотек. Python широко використовується в різних областях, включаючи веб-розробку, аналіз даних, машинне навчання та наукові обчислення. Хоча Python не славиться своєю високою продуктивністю в обчислювальних задачах, він має інструменти для паралельного програмування, які можуть бути корисними для певних сценаріїв.

Синтаксис Python простий та інтуїтивно зрозумілий, що робить його легким для вивчення та використання. Він має величезну кількість бібліотек та фреймворків для різних завдань, включаючи NumPy, SciPy та pandas для наукових обчислень. Інтерактивний режим Python дозволяє швидко експериментувати з кодом та прототипувати рішення. Можливість працювати на різних операційних системах забезпечує хорошу портативність програмного забезпечення.

Python підтримує багатопотоковість через модуль threading. Однак, через глобальне блокування інтерпретатора, справжній паралелізм на рівні потоків обмежений. Для обчислювальних задач, що вимагають високої продуктивності, рекомендується використовувати багатопроцесорність через модуль multiprocessing, який дозволяє обходити обмеження глобального блокування інтерпретатора. Приклад реалізації потоків через цей модуль вказано на рис. 2.13.

```python
import multiprocessing

def worker_function():
    # Код, який виконуватиметься в окремому процесі
    pass

if __name__ == '__main__':
    p = multiprocessing.Process(target=worker_function)
    p.start()
```

Рисунок 2.13 – Ініціалізація потоку в мові Python

Python надає кілька засобів для організації взаємодії між потоками та процесами, включаючи:

- Блокування (Lock, RLock) забезпечують взаємне виключення, дозволяючи лише одному потоку або процесу одночасно отримувати доступ до спільного ресурсу.
- Умовні змінні (Condition) дозволяють потокам або процесам чекати на певні умови та сповіщати інші потоки або процеси про зміну стану.
- Семафори (Semaphore) обмежують кількість потоків або процесів, які можуть одночасно отримати доступ до ресурсу.
- Черги (Queue) дозволяють потокам або процесам обмінюватися даними безпечним та синхронізованим способом.
- Обмін повідомленнями (Pipe) надає низькорівневий механізм для обміну повідомленнями між процесами.
- Менеджери контексту (Manager) дозволяють створювати спільні об'єкти, доступні з різних процесів.

Якщо пріоритетом є простота розробки та наявність великої кількості готових бібліотек для наукових обчислень, то Python може бути гарним варіантом. Однак, через обмеження глобального блокування інтерпретатора, для досягнення високої продуктивності в обчислювальних задачах рекомендується використовувати багатопроцесорність замість багатопотоковості, що не дуже підходить для нашого ПАК.

2.2.5 Підсумок щодо вибору мови програмування

Проведений огляд мов програмування Java, C++ (OpenMP), C# та Python дозволяє зробити висновок, що кожна з них має свої переваги та недоліки в контексті розробки високопродуктивного програмно-апаратного комплексу для операцій з комплексними матрицями.

Java пропонує кросплатформність, багату екосистему та вбудовану підтримку багатопотоковості, що робить її привабливим вибором для розробки складних застосунків.

C++ (OpenMP) забезпечує високу продуктивність та ефективне використання апаратних ресурсів, особливо при використанні директив OpenMP для розпаралелювання обчислень.

C# поєднує в собі сучасність, елегантність та потужну інтеграцію з платформою .NET, надаючи широкий спектр інструментів для розробки різноманітних застосунків.

Python відрізняється простотою, читабельністю та великою кількістю бібліотек, що робить його привабливим вибором для швидкої розробки та прототипування.

Враховуючи вимоги до легкості розширення коду в залежності від кількості ядер у процесора, найбільш підходящою мовою програмування для розробки ПАК є Java. Її вбудована підтримка багатопотоковості та можливість створення пулу потоків, дозволяють легко адаптувати код до різних апаратних конфігурацій.

Крім того, Java має багату екосистему бібліотек для роботи з матрицями та лінійної алгебри, що спростить розробку алгоритмів для операцій з комплексними матрицями.

ВИСНОВОК ДО РОЗДІЛУ 2

Було виконано розробку апаратної складової високопродуктивного ПАК для операцій з комплексними матрицями. На основі комплексного аналізу та порівняння різних варіантів процесорів, враховуючи їх продуктивність, енергоефективність та вартість, було обрано багатоядерний процесор Intel Core i9-14900KS як оптимальний за критерієм ціна/продуктивність.

Розроблено структуру ПАК під обраний процесор, визначено основні компоненти та їх взаємозв'язки. Особливу увагу приділено забезпеченню ефективної взаємодії між апаратною та програмною складовими, що є критичним для досягнення високої продуктивності обчислень в ПАК.

Було проведено детальний аналіз засобів створення програмної складової ПАК. Враховуючи вимоги до продуктивності, зручності розробки та масштабованості, на основі порівняльного аналізу різних мов програмування було обрано мову Java для програмування ПАК.

Таким чином, у цьому розділі було закладено фундамент для подальшої розробки та реалізації високопродуктивного ПАК для операцій з комплексними матрицями. Обраний процесор Intel Core i9-14900KS та мова програмування Java забезпечують необхідну продуктивність та гнучкість для ефективного вирішення поставлених завдань.

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

В цьому розділі проведено розробку програмного забезпечення ПАК для вирішення математичного виразу множення комплексних матриць:

MA = MB * MC

3.1 Розробка паралельного алгоритму

Алгоритм паралельного обчислення виразу для комплексних матриць:

1. $MAx_H = MBx * MCx_H - MBy * MCy_H$
2. $MAy_H = MBx * MCy_H + MBy * MCx_H$

Умовні позначення:

- H = N ÷ P: Розмір блоку матриці, де N – розмірність матриці, P – кількість потоків.
- MAx: Матриця, що містить дійсні частини результату.
- MAy: Матриця, що містить уявні частини результату.
- MBx: Матриця, що містить дійсні частини першого множника (MB).
- MBy: Матриця, що містить уявні частини першого множника (MB).
- MCx: Матриця, що містить дійсні частини другого множника (MC).
- MCy: Матриця, що містить уявні частини другого множника (MC).

3.2 Розробка алгоритмів потоків

Алгоритми потоків реалізують паралельне обчислення виразів для визначення дійсної та уявної частин результуючої матриці MA. Для досягнення максимальної продуктивності, обчислення розподіляються між кількома потоками, кожен з яких працює з частиною даних. Алгоритми потоків описано в таблицях 3.1 – 3.5.

Таблиця 3.1 – Алгоритм роботи потоку Т1

№	Дія потоку	Код операції
1	Введення МВх	
2	Сигнал потокам Т2…Тр про введення даних	S2-1…Sp-1
3	Чекати на введення даних в потоках Т2…Т4	W2-1… W4-1
4	Обчислення 1: $МАх_Н$ = МВх * $МСх_Н$ - МВу * $МСу_Н$	
5	Обчислення 2: $МАу_Н$ = МВх * $МСу_Н$ + МВу * $МСх_Н$	
6	Сигнал потоку Т2 про завершення обчислень.	S2-2
7	Чекати на завершення обчислень $МАх_Н$, $МАу_Н$ в потоках Т2…Тр	W1-2… Wp-2
8	Виведення результату МАх	

Таблиця 3.2 – Алгоритм роботи потоку Т2

№	Дія потоку	Код операції
1	Введення МВу	
2	Сигнал потокам Т1…Тр про введення даних	S1-1…Sp-1
3	Чекати на введення даних в потоках Т1…Т4	W1-1…W4-1
4	Обчислення 1: $МАх_Н$ = МВх * $МСх_Н$ - МВу * $МСу_Н$	
5	Обчислення 2: $МАу_Н$ = МВх * $МСу_Н$ + МВу * $МСх_Н$	
6	Сигнал потоку Т1 про завершення обчислень.	S1-2
7	Чекати на завершення обчислень $МАх_Н$, $МАу_Н$ в потоках Т1…Тр	W1-2… Wp-2
8	Виведення результату МАу	

Таблиця 3.3 – Алгоритм роботи потоку Т3

№	Дія потоку	Код операції
1	Введення МСх	
2	Сигнал потокам Т1…Тр про введення даних	S1-1…Sp-1

Кінець таблиці 3.3

№	Дія потоку	Код операції
3	Чекати на введення даних в потоках T1…T4	W1-1…W4-1
4	Обчислення 1: $MAx_H = MBx * MCx_H - MBy * MCy_H$	
5	Обчислення 2: $MAy_H = MBx * MCy_H + MBy * MCx_H$	
6	Сигнал потокам T1 та T2 про завершення обчислень.	S1-2, S2-2

Таблиця 3.4 – Алгоритм роботи потоку T4

№	Дія потоку	Код операції
1	Введення MCy	
2	Сигнал потокам T1…Tp про введення даних	S1-1…Sp-1
3	Чекати на введення даних в потоках T1…T3	W1-1…W3-1
4	Обчислення 1: $MAx_H = MBx * MCx_H - MBy * MCy_H$	
5	Обчислення 2: $MAy_H = MBx * MCy_H + MBy * MCx_H$	
6	Сигнал потокам T1 та T2 про завершення обчислень.	S1-2, S2-2

Таблиця 3.5 – Алгоритм роботи потоку Tp (p = 5…P)

№	Дія потоку	Код операції
1	Чекати на введення даних в потоках T1…T4	W1-1…W4-1
2	Обчислення 1: $MAx_H = MBx * MCx_H - MBy * MCy_H$	
3	Обчислення 2: $MAy_H = MBx * MCy_H + MBy * MCx_H$	
4	Сигнал потокам T1 та T2 про завершення обчислень.	S1-2, S2-2

Умовні позначення в табл. 3.1 – 3.5 включають:

- S - Сигнал. Вказує, що потік надсилає сигнал іншому потоку для синхронізації дій. Індекс після S вказує на номер потоку, якому надсилається сигнал, та номер сигналу. Наприклад, S2-1 означає, що потік надсилає перший сигнал потоку T2.

- W - Очікування. Вказує, що потік очікує на сигнал від іншого потоку перед тим, як продовжити виконання. Індекс після W вказує на номер потоку, від якого очікується сигнал, та номер сигналу. Наприклад, W4-1 означає, що потік очікує на перший сигнал від потоку T4.
- p - Загальна кількість потоків, задіяних у виконанні алгоритму.
- T - Потік. Вказує на конкретний потік, який виконує дії, описані в таблиці. Наприклад, T1, T2, ..., Tp позначають потоки з номерами від 1 до P.

3.3 Розробка схеми взаємодії потоків

Для ефективної реалізації паралельного алгоритму множення комплексних матриць необхідно розробити схему взаємодії потоків, яка забезпечить коректну синхронізацію та обмін даними між ними.

На рис. 3.1 представлена схема взаємодії потоків при обчисленні виразів MAx_H та MAy_H.

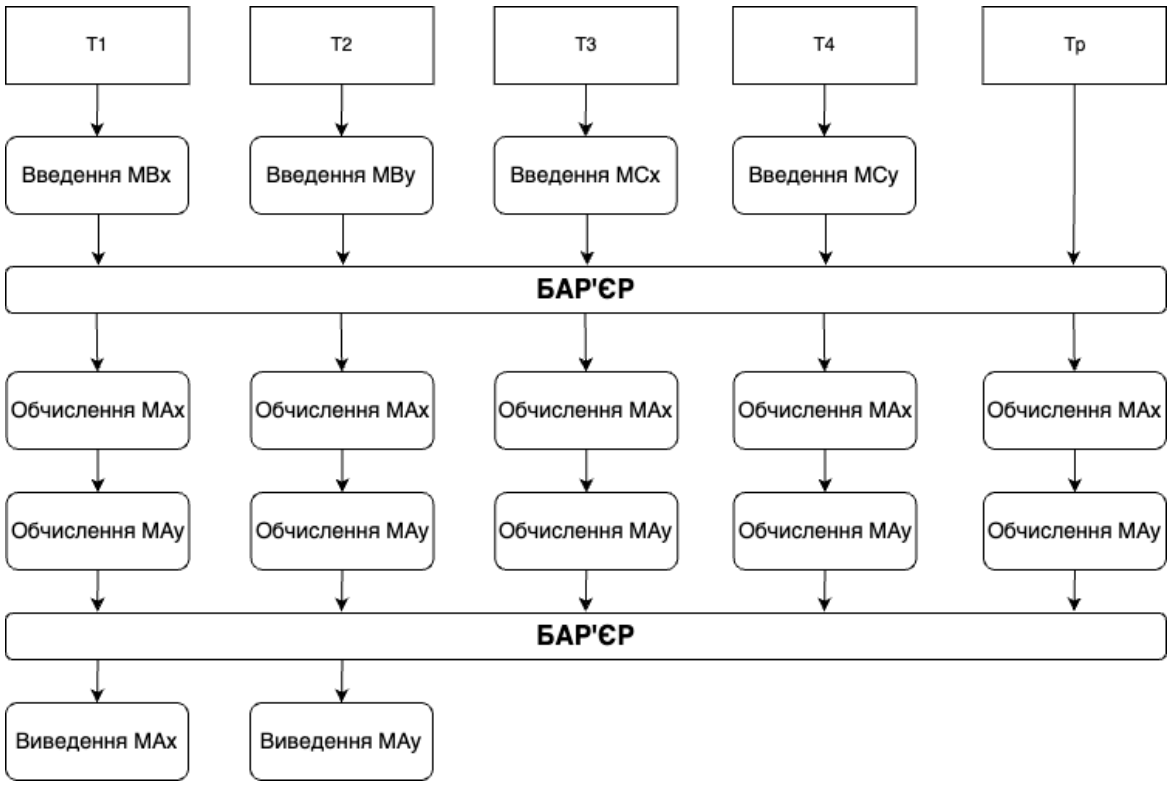


Рисунок 3.1 – Схема взаємодії потоків ПАК

Опис схеми взаємодії потоків:

1. Введення даних:
   - Потік T1 відповідає за введення матриці MBx.
   - Потік T2 відповідає за введення матриці MBy.
   - Потік T3 відповідає за введення матриці MCx.
   - Потік T4 відповідає за введення матриці MCy.
2. Бар'єр синхронізації 1:
   - Всі потоки очікують один на одного перед початком обчислень, щоб гарантувати, що вхідні дані повністю завантажені.
3. Обчислення:
   - Кожен потік паралельно обчислює свою частину результуючих матриць MAx_H та MAy_H.
4. Бар'єр синхронізації 2:
   - Всі потоки знову очікують один на одного після завершення обчислень, щоб гарантувати, що всі частини MAx та MAy пораховані та готові для виведення.
5. Виведення результату:
   - Потоки T1 та T2 збирають результати обчислень з всіх потоків та формують остаточні матриці MAx та MAy.
   - Потік T1 виводить обчислену матрицю MAx.
   - Потік T2 виводить обчислену матрицю MAy.

Запропонована схема взаємодії потоків забезпечує ефективне використання ресурсів багатоядерних систем та дозволяє досягти високої продуктивності при обчисленні виразів з комплексними матрицями.

3.4 Розробка програми

Для реалізації завдання паралельного множення матриць було обрано мову програмування Java. Цей вибір зумовлений декількома факторами: Java

забезпечує потужні засоби для роботи з багатопоточністю, має вбудовані методи синхронізації потоків, має розвинену бібліотеку класів, а також є кросплатформною мовою, що дозволяє запускати програму на різних операційних системах без необхідності внесення змін до коду.

Розробка програми включала створення кількох ключових класів, кожен з яких виконує свою специфічну роль у процесі паралельного обчислення.

Клас MainClass відповідає за запуск програми та координування виконання різних варіантів обчислень. Він містить основний метод main, в якому ініціалізуються дані, запускаються різні реалізації множення матриць і вимірюється час виконання кожного з них.

Клас Data є центральним сховищем даних для всієї програми. Він містить методи для зчитування та запису матриць з файлів, а також для їх зберігання. Крім того, в класі реалізовано методи для множення матриць як у послідовному, так і в паралельному режимах. Клас також забезпечує синхронізацію між потоками через бар'єри.

Клас ComplexMatrix представляє комплексну матрицю і містить методи для роботи з її елементами. Він дозволяє зберігати дійсні та уявні частини матриці, встановлювати та отримувати їх значення, а також виводити матрицю на екран.

Клас ScalableWorker використовує клас Thread для ініціалізації потоків та є гнучким рішенням для розподілу обчислень між кількома потоками. Він дозволяє ефективно масштабувати виконання задачі на різну кількість процесорів або ядер. Цей клас забезпечує обробку вхідних даних, синхронізацію обчислень та збереження результатів у паралельному режимі.

Клас PoolWorker реалізує роботу в межах пулу потоків. Кожен потік виконує завантаження частини вхідних даних, обчислення частини добутку матриць та збереження результатів. Синхронізація між потоками досягається за допомогою механізмів сигналізації та очікування.

Клас SequentialWorker призначений для виконання обчислень як у паралельному режимі з використанням технології IntStream, так і в послідовному. Він завантажує матриці з файлів, виконує їх множення та зберігає результати. Використовуючи IntStream, клас реалізує паралельне множення матриць, що дозволяє порівняти продуктивність з послідовним виконанням.

3.4.1 Розробка програми з ініціалізацією потоків через клас Thread

Для вирішення поставленої задачі було розроблено програму, яка використовує клас Thread для ініціалізації та управління потоками [21]. Цей підхід дозволяє чітко контролювати створення, запуск та завершення роботи кожного потоку, забезпечуючи високу гнучкість у паралельному виконанні обчислень.

На початку програма зчитує вхідні дані з файлів та ініціалізує необхідні матриці. Для паралельного виконання обчислень створюється певна кількість об'єктів класу Thread. Кожен потік відповідає за обробку частини даних або виконання певної задачі. Наприклад, якщо потрібно паралельно обробити рядки матриці, кожен потік може бути відповідальним за обробку одного або кількох рядків.

Клас, який наслідує клас Thread, містить основну логіку роботи потоку. В даній програмі цей клас має назву ScalableWorker. Цей клас визначає метод run, в якому описано всі дії, що виконує потік.

Після створення потоків вони запускаються за допомогою методу start, який викликає метод run для кожного потоку. Потоки починають виконувати свою роботу одночасно, що дозволяє значно прискорити обробку даних.

Кожен потік незалежно зчитує свою частину даних, виконує необхідні обчислення та зберігає результати. Наприклад, один потік може бути

відповідальним за зчитування та обробку дійсних частин матриць, а інший - за обробку уявних частин.

Для забезпечення коректної взаємодії між потоками використовуються механізми синхронізації, такі як методи wait та notify класу Object, або бар'єри. Це гарантує, що всі потоки завершать зчитування даних перед тим, як перейти до обчислень, а також що всі обчислення будуть завершені перед збереженням результатів. Наприклад, можна використовувати бар'єр для того, щоб потоки чекали один на одного після зчитування даних та перед початком обчислень. Це дозволяє синхронізувати їх роботу та уникнути ситуацій, коли один потік починає обчислення, не дочекавшись завершення зчитування даних іншим потоком.

Після завершення обчислень результуючі матриці зберігаються у файли. Запис результатів виконується одним або декількома потоками в залежності від конкретної реалізації програми. Для забезпечення коректного запису даних та уникнення колізій також використовуються механізми синхронізації.

Програма має повний контроль над створенням, запуском, зупинкою та управлінням потоками, що дозволяє реалізувати складні сценарії паралельного виконання.

Можливість створення та налаштування потоків для виконання різних задач забезпечує високу гнучкість у розробці паралельних алгоритмів.

Програма легко адаптується до різної кількості ядер процесора, що дозволяє ефективно використовувати доступні апаратні ресурси.

Використання класу Thread для ініціалізації потоків є ефективним підходом до розробки програм з паралельним виконанням обчислень. Цей метод дозволяє досягти високої продуктивності та ефективності обробки великих обсягів даних, зокрема матриць, завдяки чіткій організації та управлінню потоками.

3.4.2 Розробка програми з використанням паралельного виконання циклів через IntStream

Ще одним способом вирішення завдання було створення реалізації програми, яка використовує технологію IntStream для паралельного виконання циклів [22]. Використання IntStream з паралельними потоками дозволяє значно спростити паралельне обчислення та забезпечити автоматичне розподілення навантаження між процесорами.

Спочатку програма зчитує вхідні дані з файлів та ініціалізує необхідні матриці. Після ініціалізації даних, основна частина обчислень виконується за допомогою IntStream. Ця технологія дозволяє створювати потоки примітивних типів даних, таких як int, long, та double, з можливістю їх послідовної або паралельної обробки.

Для створення паралельного потоку використовується метод IntStream.range(start, end).parallel(), який генерує числовий потік в межах від start до end і виконує його паралельно. Кожен елемент потоку обробляється незалежно в різних потоках, що дозволяє значно прискорити обчислення.

Кожен елемент потоку представляє індекс рядка або стовпця матриці, для якого потрібно виконати обчислення. Для кожного індексу потік викликає лямбда-вираз або метод, що виконує обчислення. Це дозволяє обчислювати суму добутків елементів рядка однієї матриці та стовпця іншої матриці для отримання елемента результуючої матриці. Це обчислення може включати як дійсні, так і уявні частини комплексних чисел.

Технологія IntStream.parallel() автоматично розбиває завдання на окремі підзадачі, які виконуються паралельно на доступних процесорних ядрах. Цей процес базується на використанні сплітераторів (spliterators), які рекурсивно ділять потік даних на менші частини до досягнення розміру, оптимального для паралельного виконання.

Сплітератор дозволяє створювати нові підзадачі з частинами початкового потоку, які можуть виконуватись одночасно. Це забезпечує ефективне розподілення навантаження між всіма доступними процесорними ядрами.

Використання ForkJoinPool забезпечує управління потоками та їх синхронізацію. ForkJoinPool автоматично створює необхідну кількість потоків для виконання підзадач і управляє чергами завдань для оптимізації продуктивності. Він використовує робочі черги для кожного потоку і реалізує алгоритм роботи-викрадення (work-stealing), що дозволяє уникнути ситуацій, коли деякі потоки залишаються без роботи.

Завдяки цим механізмам забезпечується ефективне балансування навантаження та мінімізація витрат на синхронізацію. Це дозволяє уникнути явного використання механізмів синхронізації, таких як wait та notify, або бар'єрів (CyclicBarrier), забезпечуючи коректний запис даних навіть при паралельному виконанні.

Після завершення обчислень результуючі матриці зберігаються у файли. Запис результатів виконується послідовно для забезпечення цілісності даних та уникнення колізій.

Використання IntStream.parallel() дозволяє автоматично оптимізувати розподіл навантаження між доступними процесорами, що забезпечує ефективне використання апаратних ресурсів.

Переваги використання IntStream для паралельного виконання циклів включають:

- Простота використання інтерфейсу IntStream, що забезпечує зручний та зрозумілий синтаксис для паралельного виконання циклів, спрощуючи реалізацію паралельних алгоритмів.
- Оптимізація завдяки виклику методу IntStream.parallel(), що автоматично керує розподілом елементів потоку між потоками. Це дозволяє ефективно

використовувати всі доступні процесори без необхідності явного управління потоками.

- Програма легко адаптується до різної кількості ядер процесора, що забезпечує високу продуктивність при обробці великих обсягів даних.
- Використання алгоритмів роботи-викрадення та управління чергами завдань забезпечує рівномірний розподіл роботи між потоками, мінімізуючи час простою та збільшуючи загальну продуктивність.

У підсумку, використання IntStream для паралельного виконання циклів є ефективним та зручним підходом до реалізації паралельних обчислень у програмах на Java. Цей метод дозволяє досягти високої продуктивності та ефективності обробки даних завдяки автоматичному розподілу навантаження та простоті реалізації.

3.4.3 Розробка програми з використанням Thread Pool і Executors

Іншим способом вирішення завдання було створення реалізації програми з використанням пулу потоків (Thread Pool) та виконавців (Executors). Цей підхід забезпечує ефективне управління потоками та розподіл завдань між ними, що дозволяє значно підвищити продуктивність обчислень при роботі з великими обсягами даних.

Програма починається зі зчитування вхідних даних з файлів та ініціалізації необхідних матриць. Для ефективного управління потоками створюється пул потоків. Пул потоків дозволяє обмежити кількість одночасно працюючих потоків, що знижує накладні витрати на створення та знищення потоків, а також покращує використання системних ресурсів.

Використовується клас Executors, зокрема метод Executors.newFixedThreadPool(int nThreads), який створює пул з фіксованою кількістю потоків. Кількість потоків визначається відповідно до кількості доступних процесорних ядер або обсягу завдання.

Основні обчислення розподіляються між потоками за допомогою виконавців (ExecutorService). Для цього створюються завдання (Runnable або Callable), які представляють окремі частини обчислень, і передаються до виконання в пул потоків. Наприклад, для множення матриць можна створити завдання для обчислення кожного елемента результуючої матриці або для обчислення окремих рядків чи блоків матриць.

Пул потоків управляє розподілом завдань між потоками та їх виконанням. Завдання передаються в чергу, з якої вони автоматично витягуються та виконуються доступними потоками. Кожен потік виконує своє завдання незалежно, що забезпечує паралельність обчислень та зменшує час виконання. Після завершення завдання результати обчислень зберігаються у відповідних частинах результатуючої матриці.

Використовується механізм синхронізації для забезпечення коректної взаємодії між потоками. Завершення всіх завдань у пулі потоків контролюється за допомогою методів shutdown() та awaitTermination(). Метод shutdown() ініціює поступове завершення роботи пулу, при якому поточні завдання виконуються до кінця, але нові завдання не приймаються. Метод awaitTermination() блокує потік, який викликав його, до тих пір, поки всі завдання не завершаться або не мине вказаний час очікування.

Синхронізація результатів досягається за допомогою механізмів блокування, таких як CountDownLatch або CyclicBarrier, які забезпечують, що всі потоки завершили свої обчислення перед тим, як продовжити подальші дії.

Після завершення обчислень результуючі матриці зберігаються у файли. Запис результатів виконується послідовно для забезпечення цілісності даних та уникнення колізій. Використання пулу потоків забезпечує рівномірний розподіл навантаження та мінімізацію накладних витрат на управління потоками.

Пул потоків дозволяє уникнути накладних витрат на створення та завершення роботи потоків, що значно підвищує продуктивність програми.

Використання фіксованого пулу потоків дозволяє оптимально використовувати системні ресурси, запобігаючи надмірному навантаженню на процесор та пам'ять.

Класи Executors та ExecutorService забезпечують зручний інтерфейс для розподілу завдань та управління їх виконанням, що спрощує реалізацію паралельних алгоритмів.

Програма легко адаптується до різної кількості ядер процесора, що забезпечує високу продуктивність при обробці великих обсягів даних.

Таким чином, використання пулу потоків та виконавців у програмі забезпечує ефективне та зручне управління паралельними обчисленнями, що дозволяє досягти високої продуктивності та ефективності при вирішенні складних обчислювальних задач.

ВИСНОВОК ДО РОЗДІЛУ 3

Було виконано розробку програмного забезпечення для паралельного множення комплексних матриць. Для реалізацій програмного забезпечення було використано мову програмування Java.

Розроблено паралельний алгоритм множення матриць, який забезпечує ефективний розподіл обчислень між потоками. Цей підхід дозволяє значно прискорити процес обчислень та використовувати всі доступні апаратні ресурси.

Була розроблена схема взаємодії потоків, яка забезпечує координацію між потоками під час зчитування вхідних даних, виконання обчислень та запису результатів. Використання механізмів синхронізації та сигналізації гарантує правильну послідовність виконання дій.

Для реалізації паралельного множення матриць було розроблено три реалізації програми. Перша реалізація ґрунтується на використанні потоків (наслідування класу Thread), які забезпечують чітке та гнучке розподілення обчислень між потоками, що дозволяє масштабувати виконання на різну кількість процесорів. Друга реалізація використовує пул потоків (Thread Pool), що дозволяє управляти за допомогою технології ThreadPoolExecutor для оптимального та простого розподілу обчислень і синхронізації результатів. Третя реалізація базується на використовуванні паралельних циклів з технологією IntStream.parallel(), що дозволяє паралелізувати виконання циклів на високому рівні абстракції.

РОЗДІЛ 4. ТЕСТУВАННЯ ПРОГРАМИ

Тестування є важливою частиною процесу розробки програмного забезпечення, що дозволяє перевірити коректність і продуктивність розроблених алгоритмів та їх реалізацій. Буде розглянуто результати тестування програми для множення комплексних матриць, реалізованої у чотирьох варіантах: з використанням потоків (Thread), з паралельним виконанням циклів (IntStream), з пулом потоків (Thread Pool). Також було проведено тестування у послідовному варіанті.

Тестування було проведено на ноутбуці MacBook Air з наступними технічними характеристиками:

- Процесор Apple M1 (8-ядерний, 4 продуктивних ядра і 4 енергоефективних ядра)
- Оперативна пам'ять об’ємом 16 ГБ з тактовою частотою 4266 МГц
- Операційна система macOS Sonoma
- Дискова система SSD 256 ГБ зі швидкістю запису інформації 2139 МБ/с та швидкістю читання 2830 МБ/с.

Такі характеристики дозволяють оцінити ефективність використання різних підходів до паралельного виконання задач на сучасній ARM-архітектурі з достатньою кількістю оперативної пам'яті для обробки великих обсягів даних.

Для кожного з варіантів реалізації програми будуть проведені наступні тести:

- Час виконання: вимірювання часу, необхідного для виконання операцій множення матриць.
- Прискорення: визначення коефіцієнта прискорення порівняно з послідовним варіантом.
- Ефективність: оцінка ефективності використання процесорних ресурсів.

Прискорення ($К_п$) визначається як відношення часу виконання однопотокової програми $Т_1$ до часу виконання програми з використанням $Т_Р$ потоків.

- $К_п = \frac{Т1}{ТР}$

Ефективність ($К_е$) визначається як відношення прискорення до кількості потоків.

- $К_е = \frac{Кп}{P}$

Для кожного типу програми тестування буде проводитися з наступними конфігураціями:

- Розміри матриць (N) 800x800, 1600x1600 та 2400x2400, щоб оцінити масштабованість алгоритмів.
- Кількість потоків (P) 1, 4, 6 та 8, щоб визначити оптимальну конфігурацію для даного обладнання.

4.1 Тестування програми з потоками

Для порівняння швидкості та ефективності програм, було розглянуто результати тестування програми, яка використовує класи, що наслідують Thread. Тестування включало вимірювання часу виконання операцій, обчислення прискорення та ефективності для різних розмірів матриць та кількості потоків.

Таблиці 4.1 – 4.3 показують час виконання програм у мілісекундах відносно кількості задіяних потоків для різних розмірностей матриць. Розмірність всіх матриць складає N на N.

Після отримання даних тестування було побудовано графіки залежності часу, прискорення та ефективності від кількості потоків та розмірностей матриць (див. рисунки 4.1 – 4.3).

Таблиця 4.1 Час виконання програми з потоками (в мілісекундах)

Розмір матриці N	Кількість ядер P			
	1	4	6	8
800	1563	452	384	361
1600	14921	4819	3827	3506
2400	75324	26788	22247	19842

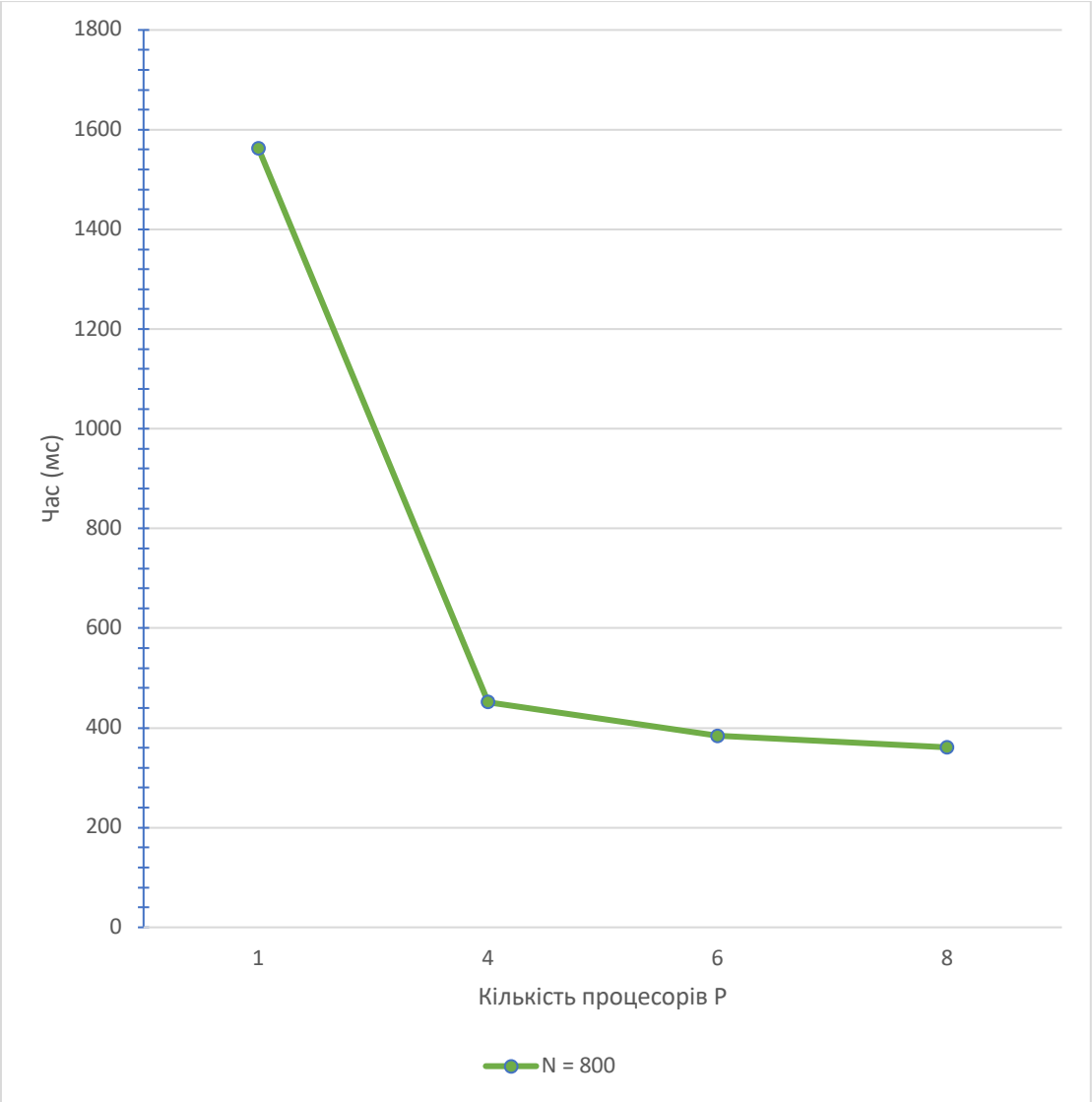


Рисунок 4.1 – графік залежності часу виконання програми з потоками від кількості процесорів для N = 800

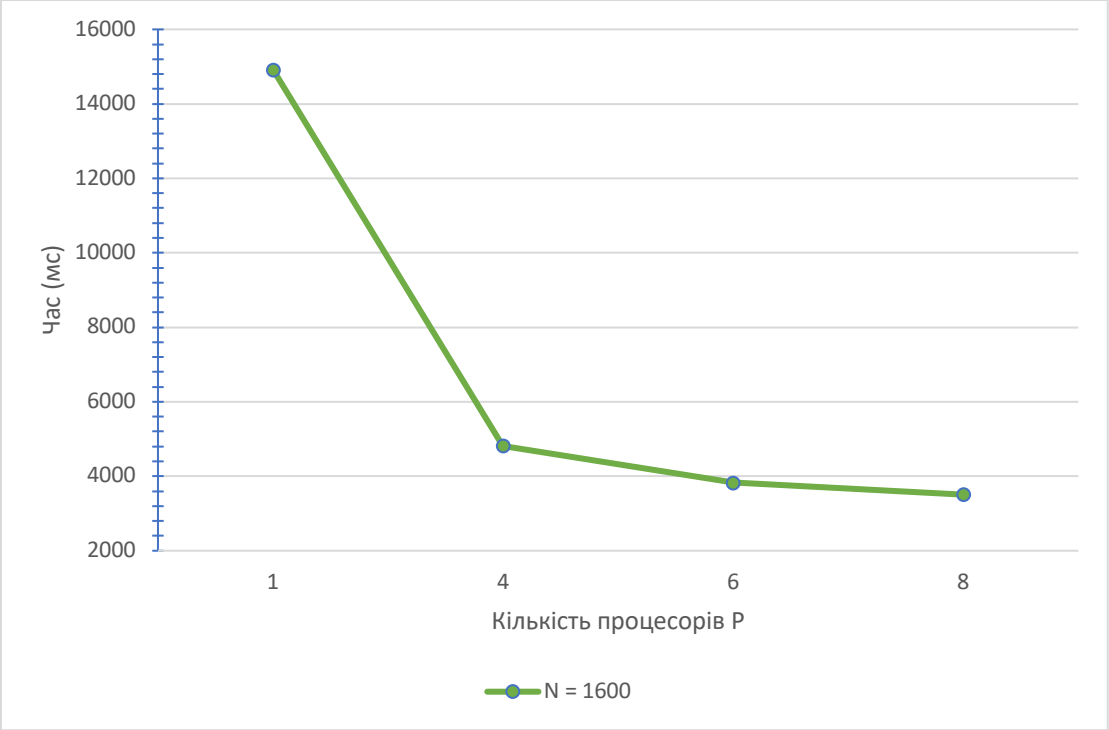


Рисунок 4.2 – графік залежності часу виконання програми з потоками від кількості процесорів для N = 1600

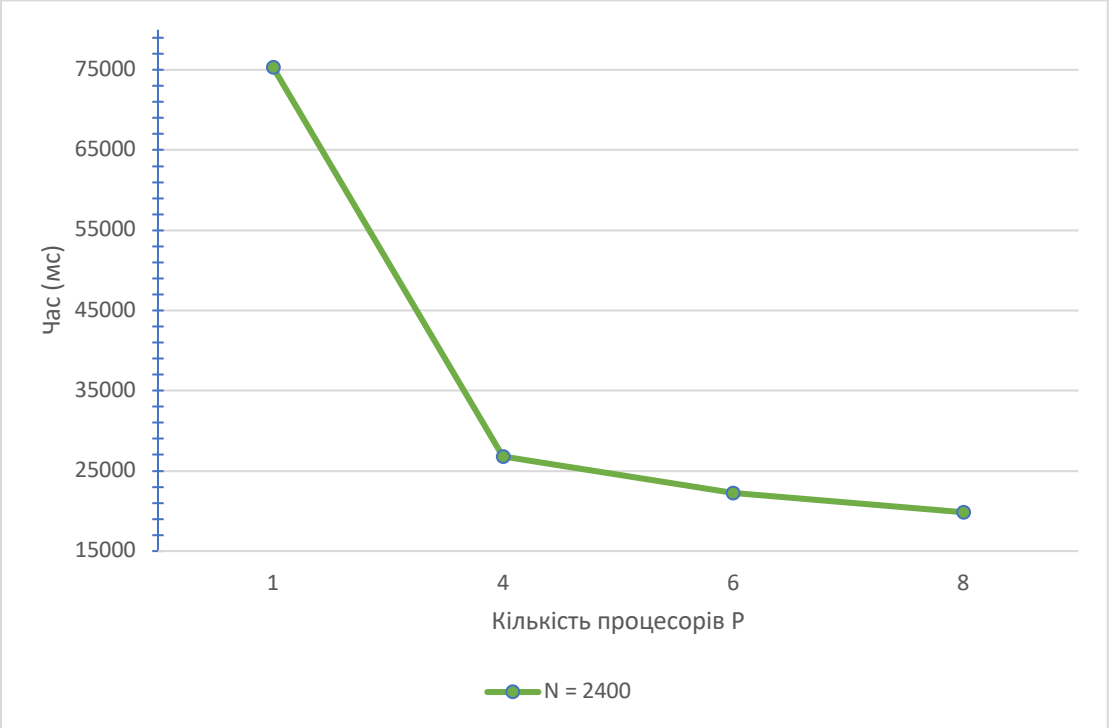


Рисунок 4.3 – графік залежності часу виконання програми з потоками від кількості процесорів для N = 2400

Таблиця 4.2 Коефіцієнти прискорення програми з потоками

Розмір матриці N	Кількість ядер P		
	4	6	8
800	3.46	4.07	4.33
1600	3.10	3.90	4.26
2400	2.81	3.38	3.80

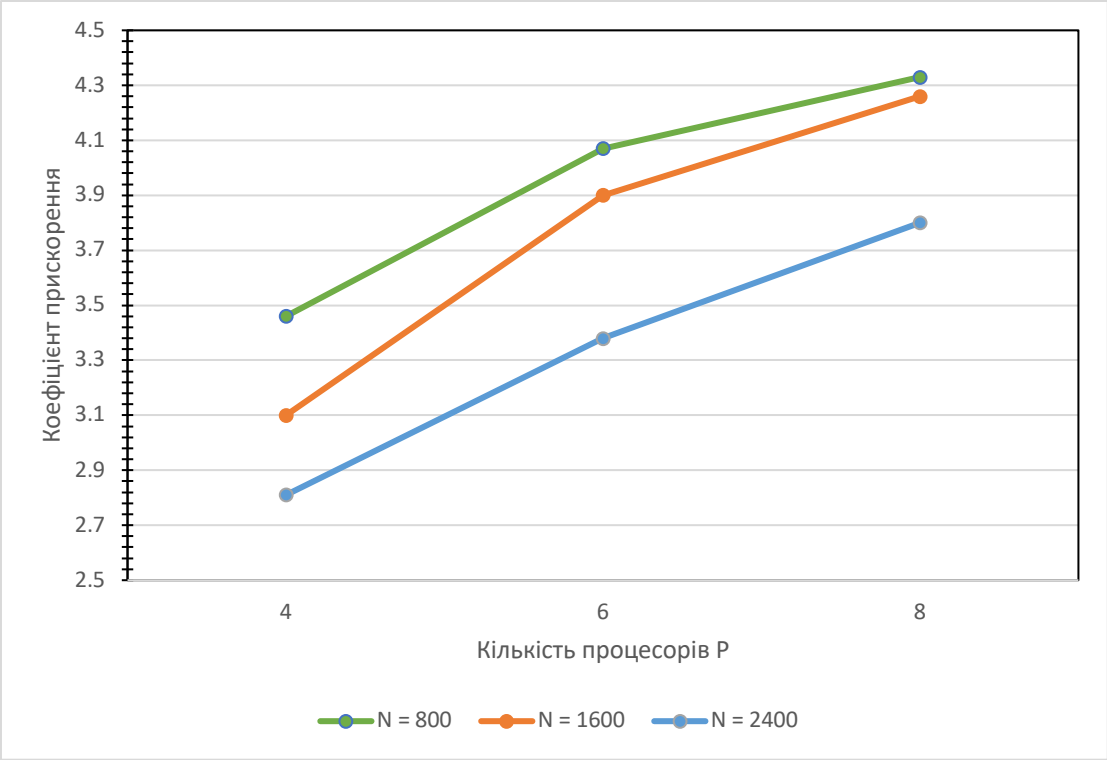


Рисунок 4.4 – графік залежності коефіцієнтів прискорення програми з потоками від кількості процесорів

Таблиця 4.3 Коефіцієнти ефективності програми з потоками

Розмір матриці N	Кількість ядер P		
	4	6	8
800	0.865	0.678	0.541
1600	0.775	0.650	0.533
2400	0.703	0.563	0.475

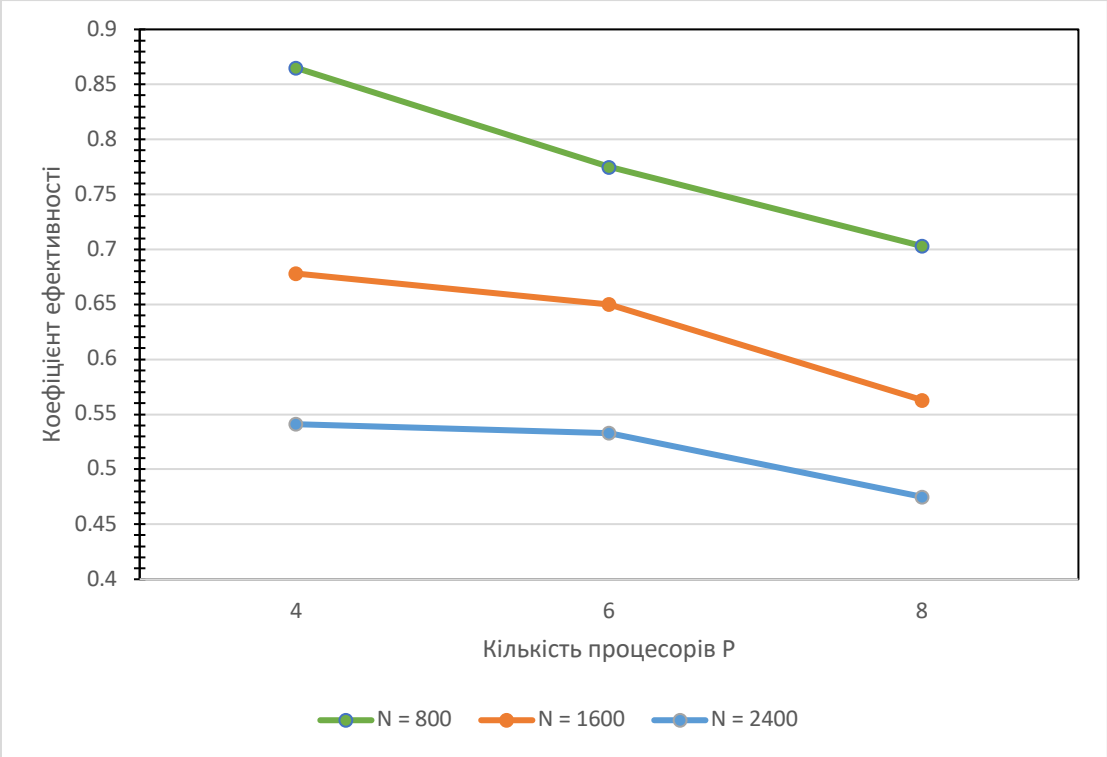


Рисунок 4.5 – графік залежності ефективності програми з потоками від кількості процесорів

Виходячи з отриманих даних можна зробити такі висновки для програми з використанням потоків:

1. Програма показує значне прискорення при збільшенні кількості ядер, особливо для менших матриць.
2. Ефективність зменшується з ростом кількості ядер, що може свідчити про накладні витрати на послідовне введення та виведення даних.
3. Найбільше прискорення і ефективність спостерігаються при використанні 4 ядер, після чого ефективність падає.

4.2 Тестування програми з паралельним виконанням циклів

В цьому підрозділі було проведено огляд результатів тестування програми, яка використовує паралельне виконання циклів з використанням IntStream.parallel().

В таблицях 4.4 – 4.6 вказано час виконання програм у мілісекундах відносно кількості задіяних потоків для різних розмірностей матриць. Розмірність тестованих матриць складає N на N.

Виходячи з даних отриманих після тестування було побудовано графіки залежності часу, прискорення та ефективності від розмірів матриць та кількості задіяних процесорів (див. рисунки 4.6 – 4.11).

Таблиця 4.4 Час виконання програми з паралельним виконанням циклів (в мілісекундах)

Розмір матриці N	Кількість ядер P			
	1	4	6	8
800	1544	479	461	455
1600	14904	4058	3932	3904
2400	77839	30116	22208	21302

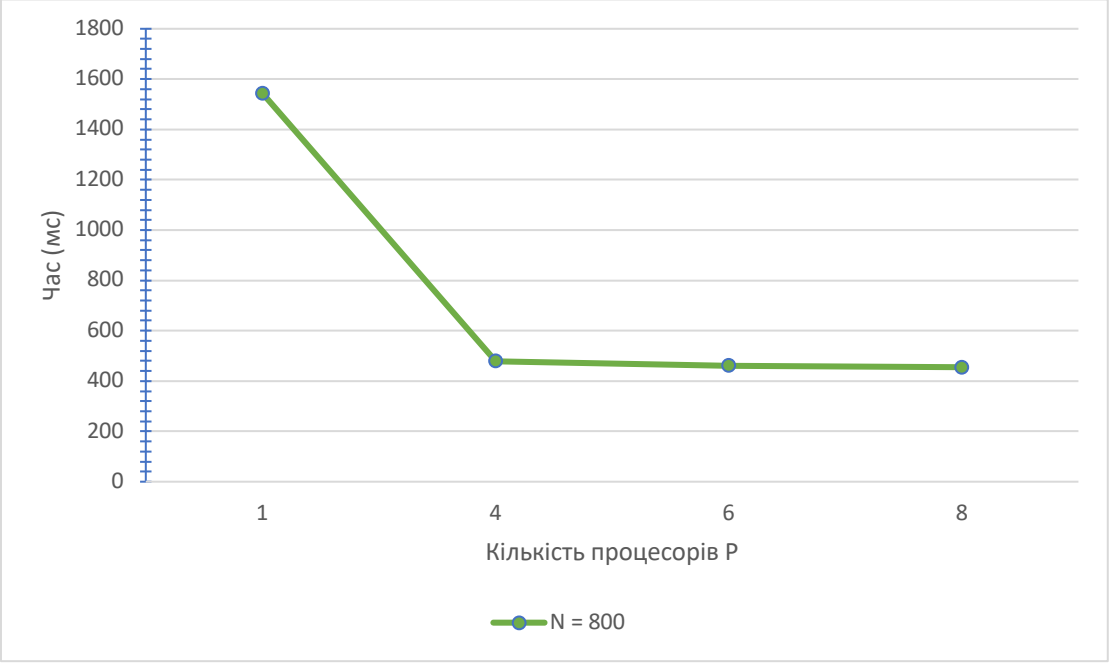


Рисунок 4.6 – графік залежності часу виконання програми з паралельним виконанням циклів від кількості процесорів для N = 800

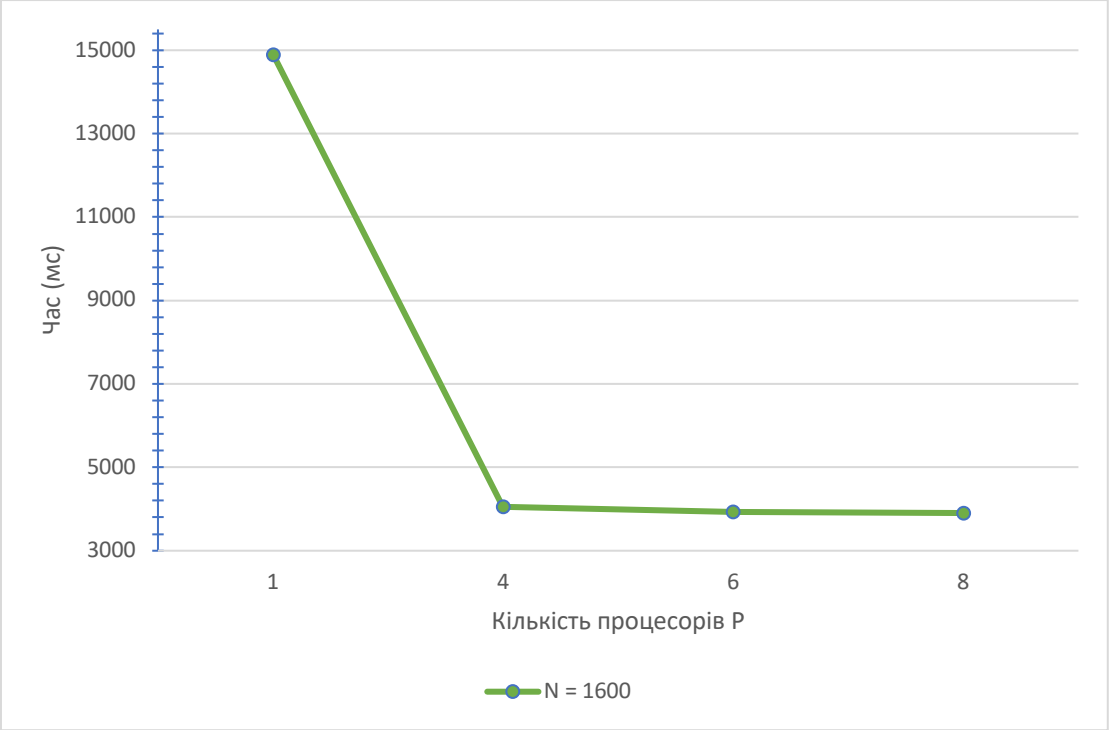


Рисунок 4.7 – графік залежності часу виконання програми з паралельним виконанням циклів від кількості процесорів для N = 1600

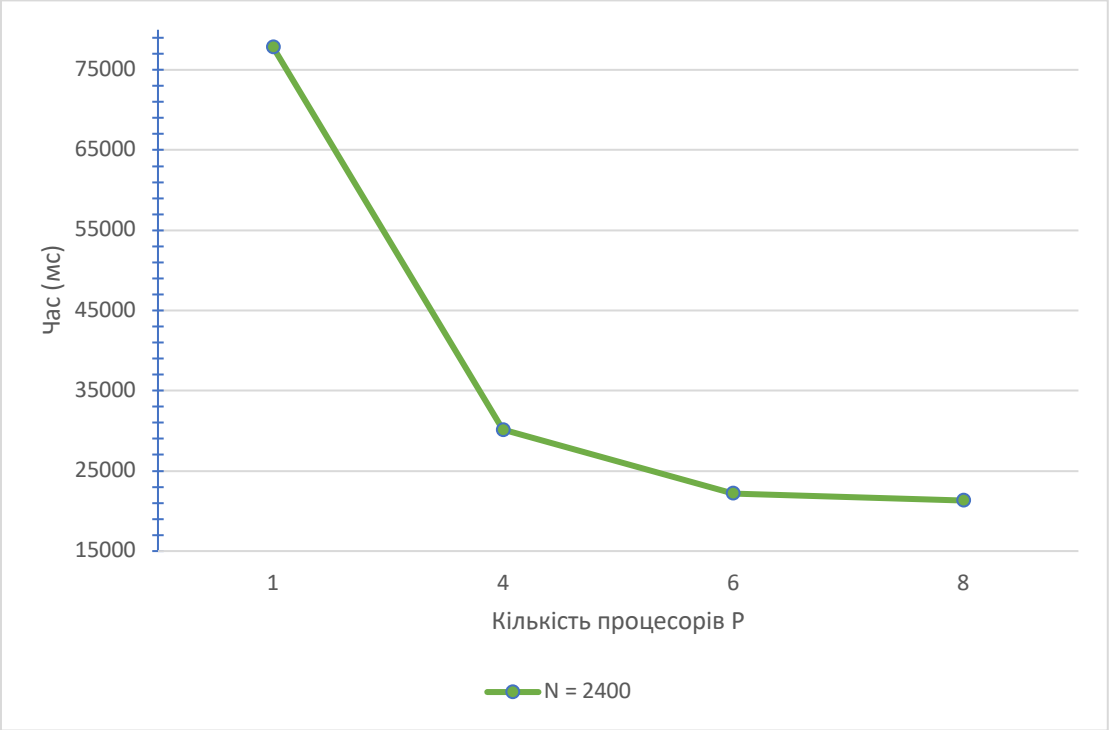


Рисунок 4.8 – графік залежності часу виконання програми з паралельним виконанням циклів від кількості процесорів для N = 2400

Таблиця 4.5 Коефіцієнти прискорення програми з паралельним виконанням циклів

Розмір матриці N	Кількість ядер P		
	4	6	8
800	3.22	3.35	3.39
1600	3.67	3.79	3.82
2400	2.58	3.51	3.65

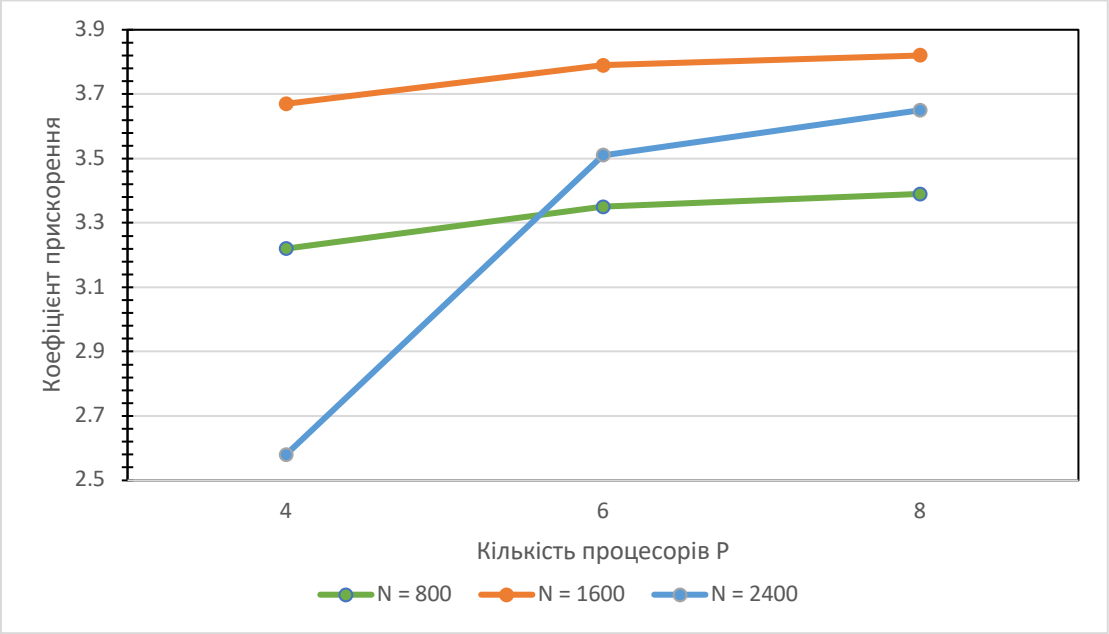


Рисунок 4.9 – графік залежності коефіцієнтів прискорення програми з паралельним виконанням циклів від кількості процесорів

Таблиця 4.6 Коефіцієнти ефективності програми з паралельним виконанням циклів

Розмір матриці N	Кількість ядер P		
	4	6	8
800	0.805	0.558	0.424
1600	0.918	0.632	0.477
2400	0.644	0.585	0.456

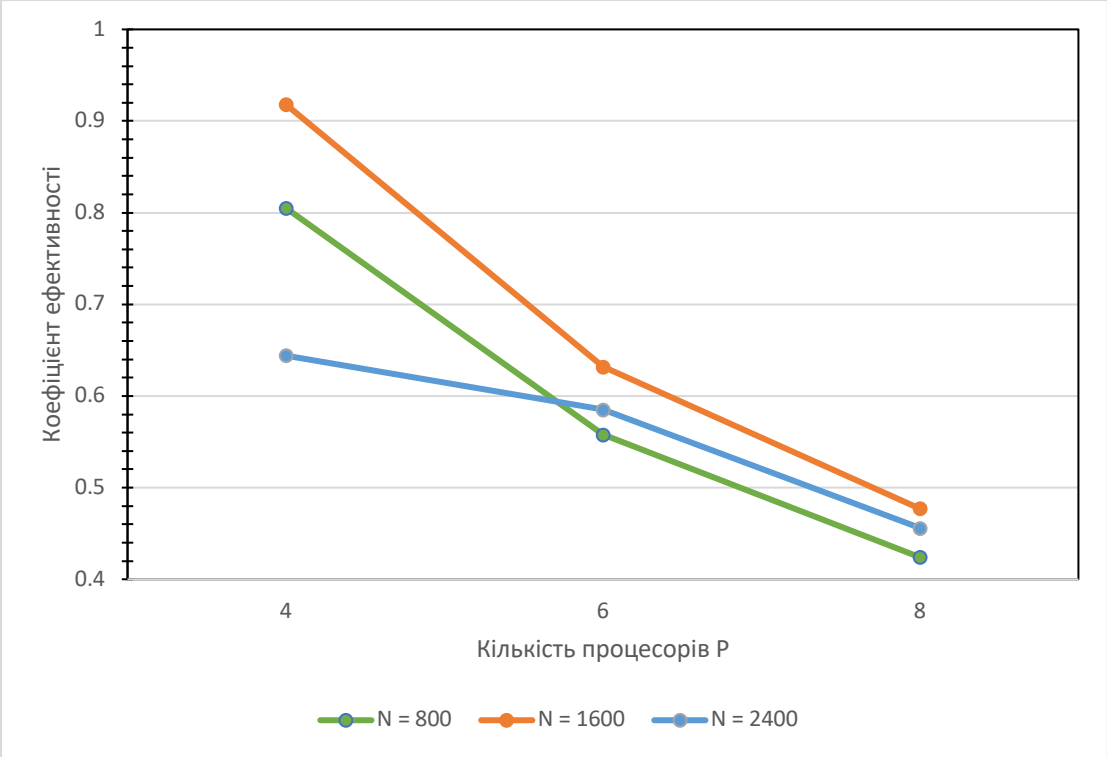


Рисунок 4.10 – графік залежності ефективності програми з паралельним виконанням циклів від кількості процесорів

Після отримання результатів тестування програми ми можемо зробити такі висновки:

1. Програма демонструє стабільне прискорення з збільшенням кількості задіяних ядер.
2. Ефективність вища для меншої кількості потоків, але зменшується при збільшенні ядер, що може свідчити про втрати під час послідовного введення та виведення даних.
3. Найвища ефективність спостерігається при використанні 4 потоків, після чого ефективність зменшується.

4.3 Тестування програми з пулом потоків

Завершальним етапом було розглянуто результати тестування програми, яка використовує пул потоків.

Час виконання у мілісекундах для кожної конфігурації вказано в таблицях 4.7 – 4.9. Розмірність тестованих матриць складає N на N елементів.

Отримані дані від тестування було відображено на графіках залежності часу, прискорення та ефективності від розмірів матриць та кількості задіяних процесорів (див. рисунки 4.11 – 4.15).

Таблиця 4.7 Час виконання програми з пулом потоків (в мілісекундах)

Розмір матриці N	Кількість ядер P			
	1	4	6	8
800	1572	488	407	396
1600	15103	4810	4043	3858
2400	71232	30457	26338	25818

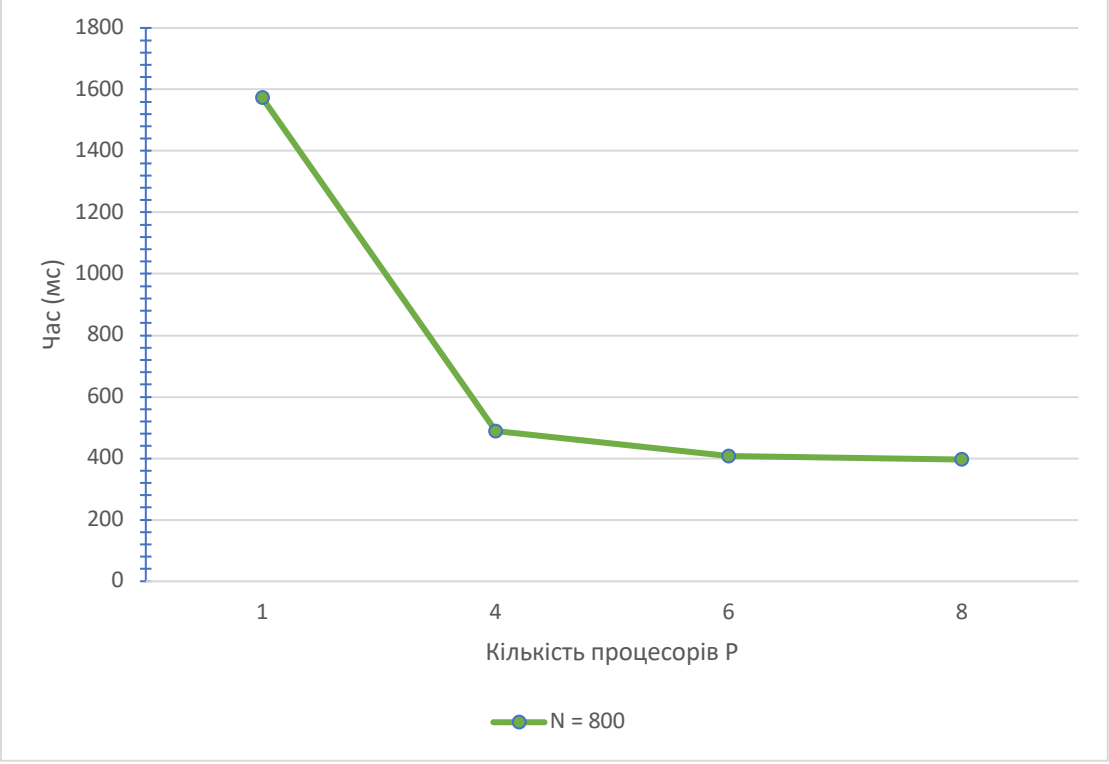


Рисунок 4.11 – графік залежності часу виконання програми з пулом потоків від кількості процесорів для N = 800

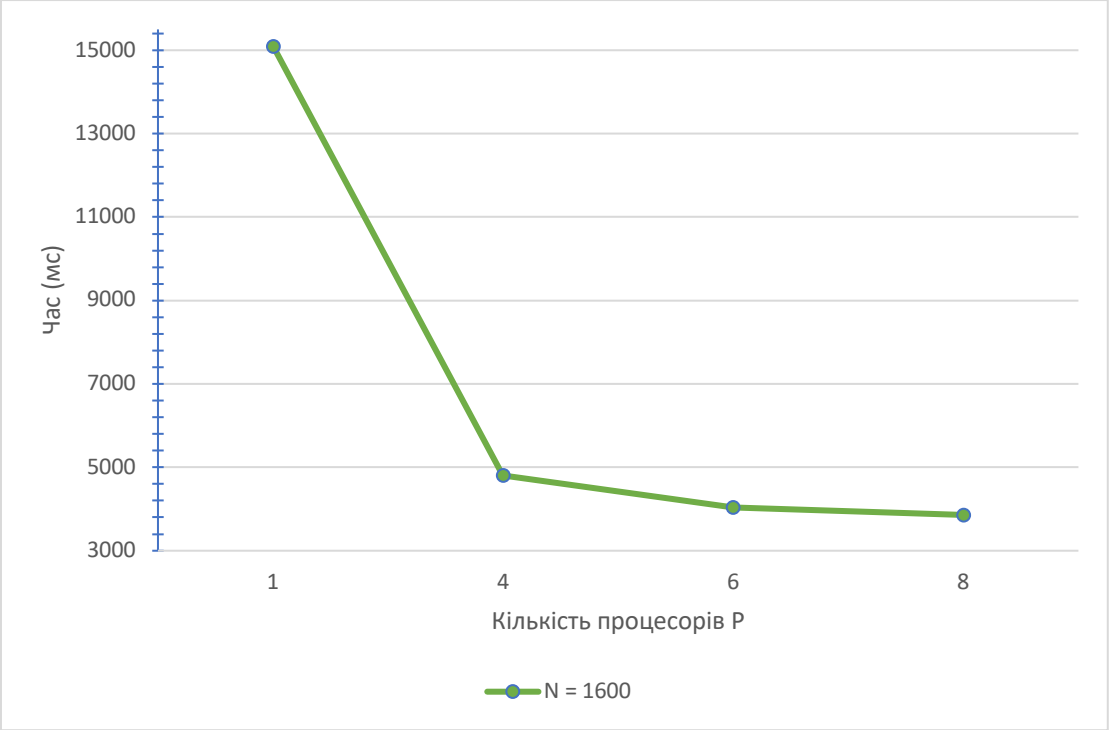


Рисунок 4.12 – графік залежності часу виконання програми з пулом потоків від кількості процесорів для N = 1600

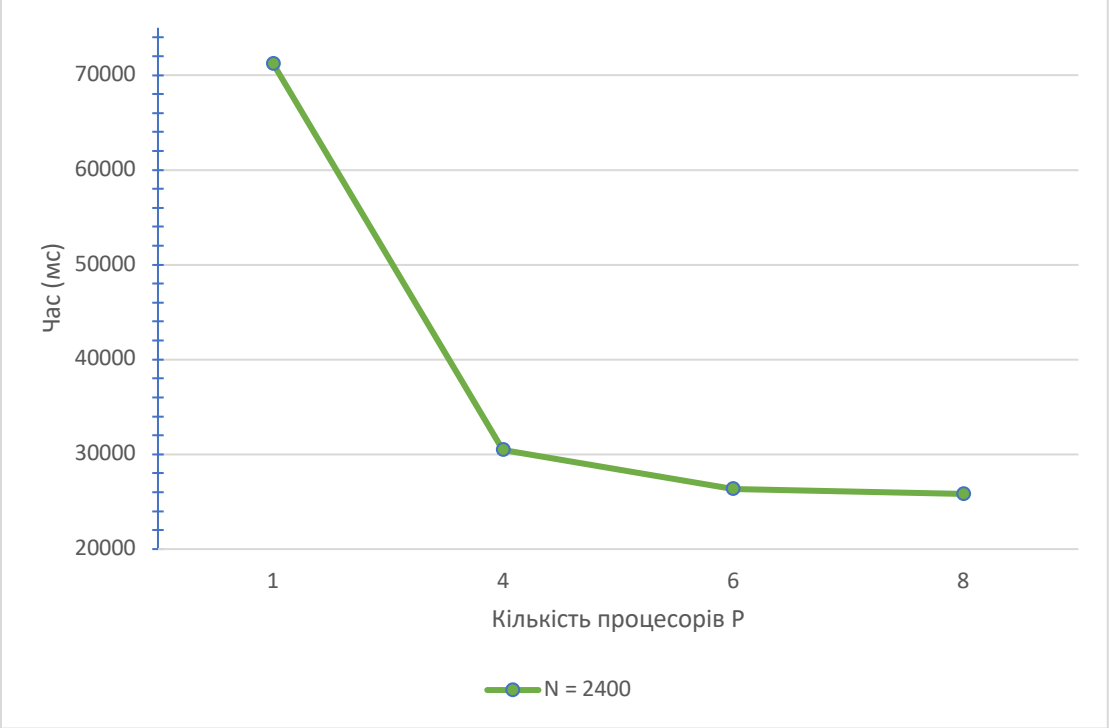


Рисунок 4.13 – графік залежності часу виконання програми з пулом потоків від кількості процесорів для N = 2400

Таблиця 4.8 Коефіцієнти прискорення програми з пулом потоків

Розмір матриці N	Кількість ядер P		
	4	6	8
800	3.22	3.86	3.97
1600	3.14	3.74	3.91
2400	2.34	2.71	2.76

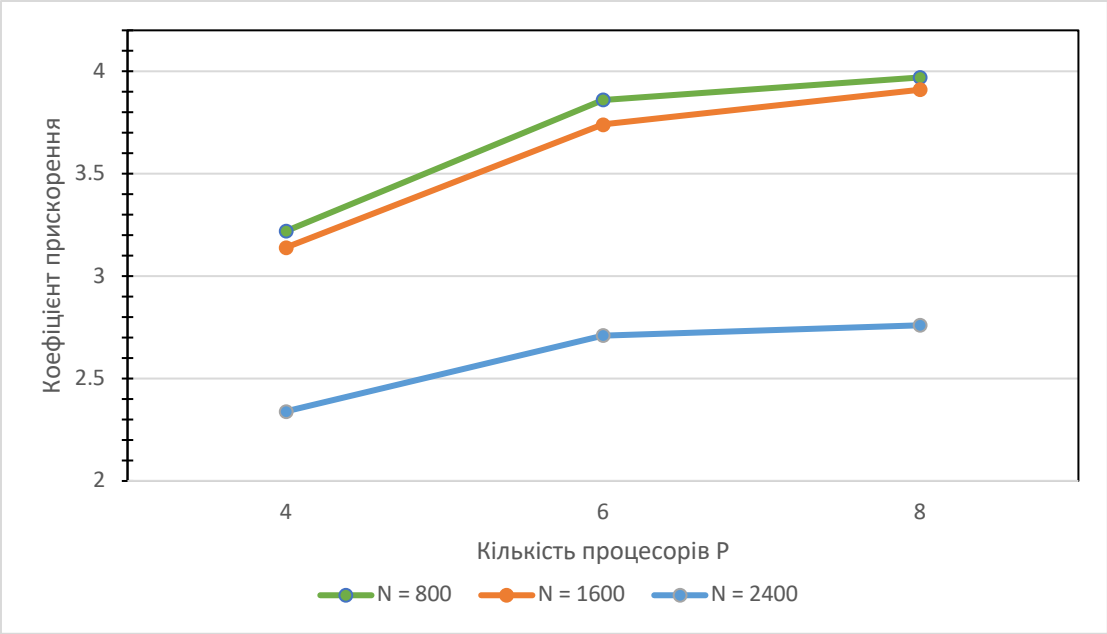


Рисунок 4.14 – графік залежності коефіцієнтів прискорення програми з пулом потоків від кількості процесорів

Таблиця 4.9 Коефіцієнти ефективності програми з пулом потоків

Розмір матриці N	Кількість ядер P		
	4	6	8
800	0.805	0.644	0.496
1600	0.786	0.623	0.489
2400	0.584	0.452	0.345

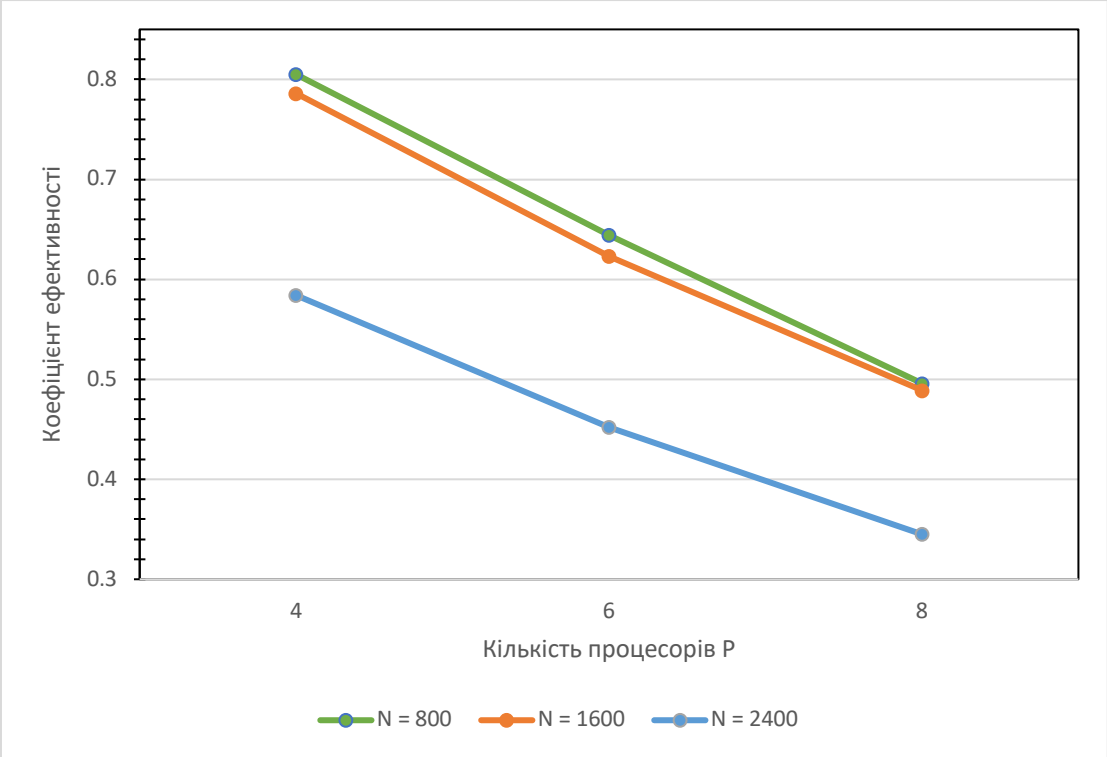


Рисунок 4.15 – графік залежності ефективності програми з пулом потоків від кількості процесорів

Для третьої програми з використанням пулу потоків можна зробити такі висновки:

1. Програма показує значне прискорення при використанні 6 та більше ядер.
2. Ефективність зменшується з збільшенням кількості потоків, що може свідчити про обмеження паралелізації або накладні витрати на введення та виведення даних.
3. Найбільше прискорення і ефективність спостерігаються при використанні 6 потоків, після чого ефективність падає.

4.4 Порівняльний аналіз методів паралелізації

Для визначення найкращого методу паралелізації проведено порівняльний аналіз результатів, отриманих у ході тестування. Таблиця 4.10 показує коефіцієнти ефективності при P = 8, виходячи з чого ми можемо зробити висновки про найбільш ефективну програму для кожного варіанту N.

Таблиця 4.10 Коефіцієнти ефективності методів паралелізації для P = 8

Розмір матриці N	Метод паралелізації		
	Потоки	Паралельне виконання циклів	Пул потоків
800	0.541	0.424	0.496
1600	0.533	0.477	0.489
2400	0.475	0.456	0.345

Для невеликих матриць (N = 800) метод з використанням потоків демонструє найкращу продуктивність, забезпечуючи найвищий коефіцієнт прискорення ($К_п$ = 4.33) та ефективності ($К_е$ = 0.54). Це можна пояснити меншими накладними витратами на створення та управління потоками порівняно з іншими методами.

Зі збільшенням розміру матриці (N = 1600) різниця у продуктивності між методами зменшується. Метод з паралельними циклами показує трохи кращий коефіцієнт прискорення ($К_п$ = 3.82), але метод з потоками все ще залишається конкурентоспроможним ($К_п$ = 3.91).

Для великих матриць (N = 2400) метод з потоками знову виходить на перше місце за продуктивністю, забезпечуючи коефіцієнт прискорення $К_п$ = 3.80 та ефективність $К_е$ = 0.48. Це свідчить про те, що для великих обсягів даних переваги потоків у керуванні та синхронізації переважають над накладними витратами.

Слід зазначити, що ефективність програм знижується зі збільшенням їх кількості понад 4-6. Це пояснюється тим, що тестова система має лише один накопичувальний пристрій, що призводить до послідовного виконання операцій введення/виведення даних, які стають "вузьким місцем" при збільшенні кількості потоків.

ВИСНОВОК ДО РОЗДІЛУ 4

Було проведено тестування трьох реалізацій програмного забезпечення для паралельного множення комплексних матриць. Тестування проводилося на процесорі з кількістю ядер, рівною 8, 16 ГБ оперативної пам'яті та одним накопичувальним пристроєм на 256 ГБ.

Програма з потоками показала значне прискорення при збільшенні кількості потоків. Максимальне значення прискорення 4.33 було досягнуто при N=2400 і P=8, де час виконання зменшився до 19842 мс у порівнянні з 75324 мс для одного потоку. Ефективність зростала до значень більше 0.5 зі збільшенням кількості потоків до 4, після чого спостерігалося поступове зниження до значень більше 0.4 через накладні витрати на управління та ініціалізацію потоків.

Програма з паралельним виконанням циклів досягла максимального прискорення 3.65 при N=2400 і P=8, де час виконання зменшився до 21302 мс відносно 77839 мс з одним потоком. Ефективність досягала 0.91 для меншої кількості потоків, але знижувалася до 0.42 зі збільшенням кількості потоків, що демонструє значні витрати часу на послідовне введення та виведення даних з одним накопичувальним пристроєм.

Програма з використанням пулу потоків продемонструвала максимальне прискорення при N=2400 і P=8, де час виконання зменшився до 25818 мс з 8 задіяними ядрами, порівняно з 71232 мс з одним задіяним ядром. Ефективність зростала до 0.8 при меншій розмірності матриць та 4 задіяних ядрах.

Загалом, програма з використанням потоків демонструє найкращі коефіцієнти ефективності, при 8 задіяних ядрах P, серед усіх варіантів програм. Коефіцієнти ефективності складають $K_e = 0.541$ при $N = 800$, $K_e = 0.533$ при $N = 1600$, $K_e = 0.475$ при $N = 2400$. Ефективність програми може знижуватись через обмеження в одному накопичуючому пристрої тестової системи, через що операції введення/виведення відбуваються послідовно.

ВИСНОВКИ

У даній дипломній роботі було виконано розробку програмно-апаратного комплексу для скорочення часу обчислень операцій з комплексними матрицями. Комплексний підхід до вирішення цієї задачі дозволив досягти значних результатів у прискоренні обчислень, що відкриває нові можливості для оптимізації обчислень у різних наукових та інженерних галузях.

У першому розділі було проведено аналіз різних типів високопродуктивних комп'ютерних систем, включаючи кластерні системи, багатоядерні процесори та графічні процесори. Було визначено основні переваги та недоліки кожної архітектури, а також специфічні сфери їх застосування. Особлива увага була приділена використанню високопродуктивних систем для множення комплексних матриць у таких галузях, як квантова механіка, цифрова обробка сигналів та наукові дослідження.

Другий розділ був присвячений вибору апаратних та програмних засобів для створення високопродуктивного комплексу. Було порівняно різні процесори та обрано оптимальну конфігурацію для реалізації задачі множення комплексних матриць. Також було розглянуто кілька мов програмування (Java, C++, C#, Python) та обрано Java як найбільш підходящу для розробки паралельного програмного забезпечення завдяки її потужним засобам для роботи з багатопотоковістю.

У третьому розділі детально описано процес розробки програмного забезпечення для паралельного множення матриць. Було розроблено три різні реалізації програмного забезпечення:

ScalableWorker забезпечує гнучке розподілення обчислень між потоками з використанням класу Thread.

PoolWorker використовує пул потоків для оптимального розподілу обчислень і забезпечення ефективного використання ресурсів.

SequentialWorker використовує IntStream.parallel() для паралелізації обчислень циклів на високому рівні абстракції.

У четвертому розділі проведено тестування розроблених програм на процесорі з 8 ядрами. Тестування показало значні покращення в часі виконання завдяки використанню багатопотоковості.

Програма з потоками досягла найкращого результату з точки зору часу виконання та ефективності. Було досягнуто максимального прискорення у 4.33 рази при N=2400 і P=8.

Програма з паралельним виконанням циклів продемонструвала коефіцієнт ефективності близько 0.8 для меншої кількості потоків, але коефіцієнт ефективності збільшувся в невеликих обсягах при збільшенні кількості задіяних ядер, досягаючи максимального прискорення у 3.65 раза при N=2400 і P=8.

Програма з пулом потоків продемонструвала результати збільшення прискорення у 3.53 рази при N=2400 і P=8.

Проведена розробка продемонструвала, що використання багатопотоковості та паралельних обчислень дозволяє підвищити продуктивність в більше ніж 3 рази при множенні комплексних матриць. Підхід з використанням потоків виявився найефективнішим, досягнувши прискорення в 4 рази, що свідчить про ефективне використання апаратних ресурсів. Розроблене програмне забезпечення може бути успішно застосоване у різних галузях, де потрібна висока продуктивність обчислень, забезпечуючи значне зниження часу виконання задач.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Understanding Clustering in Cloud Computing [Електронний ресурс] - Режим доступу до ресурсу: https://www.businesstechweekly.com/operational-efficiency/cloud-computing/cluster-cloud-computing/
2. FS NVIDIA Mellanox InfiniBand Switch Overview [Електронний ресурс] - Режим доступу до ресурсу: https://community.fs.com/article/fs-nvidia-mellanox-infiniband-switch-overview.html
3. Parallel processing [Електронний ресурс] - Режим доступу до ресурсу: https://www.techtarget.com/searchdatacenter/definition/parallel-processing
4. High availability and scalability [Електронний ресурс] - Режим доступу до ресурсу:https://www.ibm.com/docs/da/informix-servers/12.10?topic=guide-high-availability-scalability
5. HPC Cluster [Електронний ресурс] - Режим доступу до ресурсу: https://www.purestorage.com/knowledge/what-is-an-hpc-cluster.html
6. Multicore Processor [Електронний ресурс] - Режим доступу до ресурсу: https://www.javatpoint.com/what-is-a-multicore-processor
7. Multicore Programming [Електронний ресурс] - Режим доступу до ресурсу: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/resources/lecture-6-multicore-programming/
8. Exploring the GPU Architecture [Електронний ресурс] - Режим доступу до ресурсу: https://core.vmware.com/resource/exploring-gpu-architecture
9. Nvidia [Електронний ресурс] - Режим доступу до ресурсу: https://dovidnyk.info/index.php/Brand/450
10. High-Performance Computing [Електронний ресурс] - Режим доступу до ресурсу: https://www.nvidia.com/en-us/glossary/high-performance-computing/

11. CUDA Zone [Електронний ресурс] - Режим доступу до ресурсу: https://developer.nvidia.com/cuda-zone
12. Open Computing Language OpenCL [Електронний ресурс] - Режим доступу до ресурсу: https://developer.nvidia.com/opencl
13. Explore Intel's history [Електронний ресурс] - Режим доступу до ресурсу: https://timeline.intel.com/
14. Advanced Micro Devices, Inc. [Електронний ресурс] - Режим доступу до ресурсу: https://www.britannica.com/money/Advanced-Micro-Devices-Inc
15. Intel Core i9-14900KS [Електронний ресурс] - Режим доступу до ресурсу: https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i9-14900KS
16. Java [Електронний ресурс] - Режим доступу до ресурсу: https://www.java.com/en/download/help/whatis_java.html
17. OpenMP [Електронний ресурс] - Режим доступу до ресурсу: https://www.openmp.org/about/openmp-faq/#OMPAPI.General
18. Microsoft [Електронний ресурс] - Режим доступу до ресурсу: https://dovidnyk.info/index.php/Brand/404
19. Quantum Mechanics Must Be Complex [Електронний ресурс] - Режим доступу до ресурсу: https://physics.aps.org/articles/v15/7
20. Wi-Fi networking technology [Електронний ресурс] - Режим доступу до ресурсу: https://www.britannica.com/technology/Wi-Fi
21. Class Thread [Електронний ресурс] - Режим доступу до ресурсу: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html
22. Interface IntStream [Електронний ресурс] - Режим доступу до ресурсу: https://docs.oracle.com/javase%2F8%2Fdocs%2Fapi%2F%2F/java/util/stream/IntStream.html
23. Thread Pools [Електронний ресурс] - Режим доступу до ресурсу: https://docs.oracle.com/javase%2Ftutorial%2Fessential%2Fconcurrency%2F%2F/pools.html

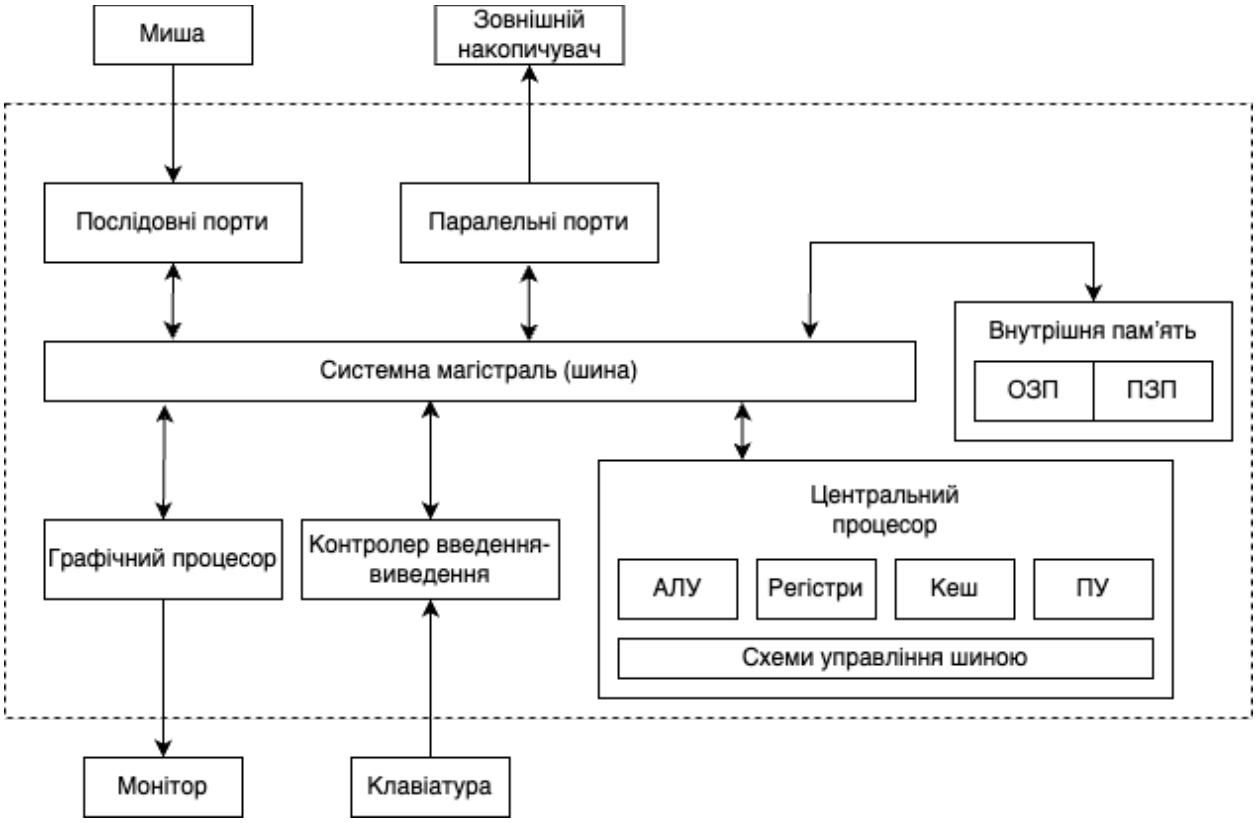
ДОДАТОК 1

Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями

Структурна схема системи (структурна схема)
ІАЛЦ.466500.004 Д1

Аркушів 1

Київ 2024 р



					ІАЛЦ.466500.004 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Бондарчук А.Д.			**Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Корочкін О.В.					
Н. Контр.		Волокита А.М.			**Структурна схема системи (структурна схема)**	КПІ ім. Ігоря Сікорського ФІОТ, ІП-05	
Затвердив		Стіренко С.Г.					

ДОДАТОК 2

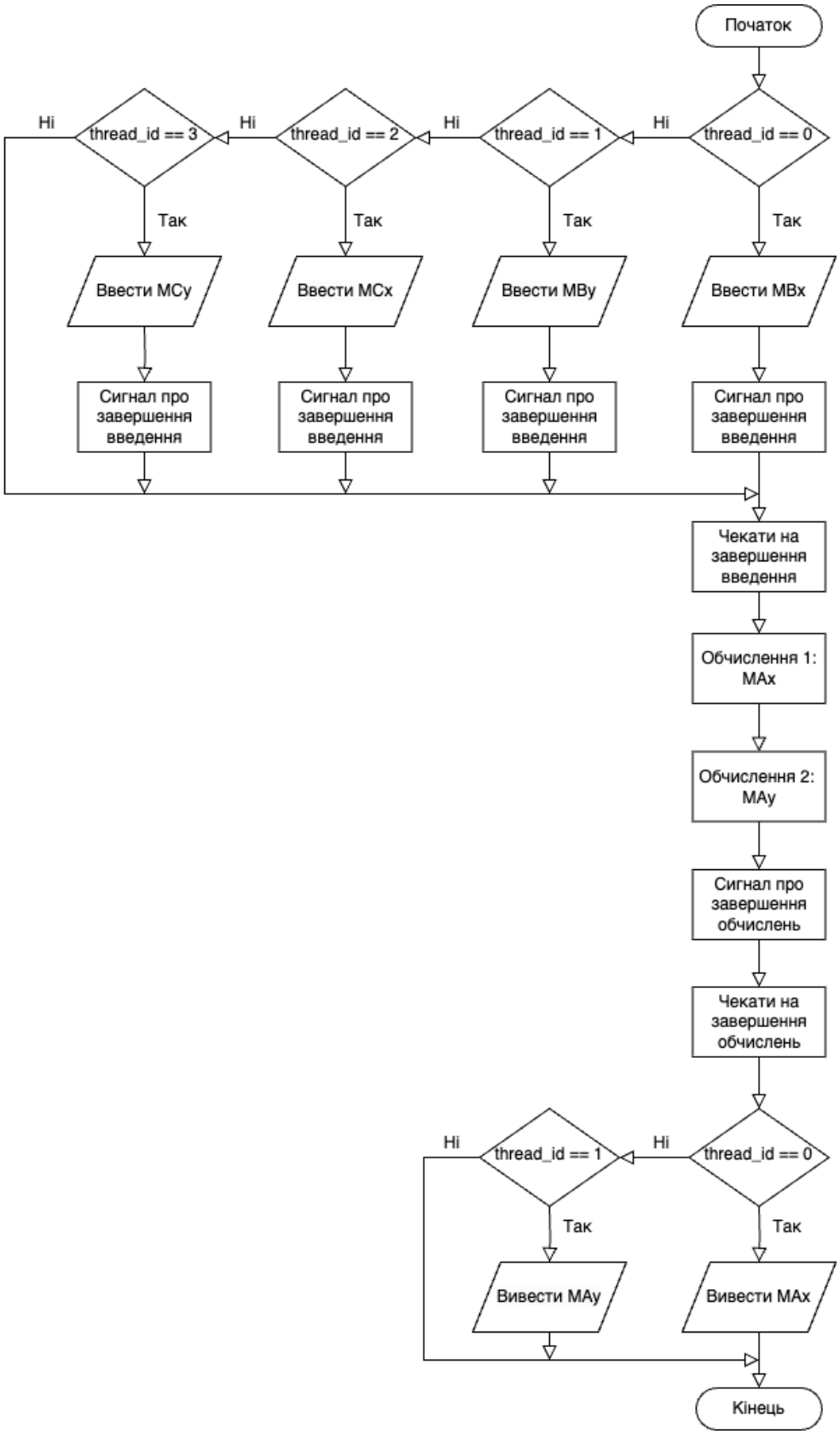
Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями

Алгоритм дій програмного забезпечення (принципова схема)

ІАЛЦ.466500.005 Д2

Аркушів 1

Київ 2024 р



					ІАЛЦ.466500.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Бондарчук А.Д.			**Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями** **Алгоритм дій програмного забезпечення (принципова схема)**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Корочкін О.В.					
Н. Контр.		Волокита А.М.				КПІ ім. Ігоря Сікорського ФІОТ, ІП-05	
Затвердив		Стіренко С.Г.					

ДОДАТОК 3

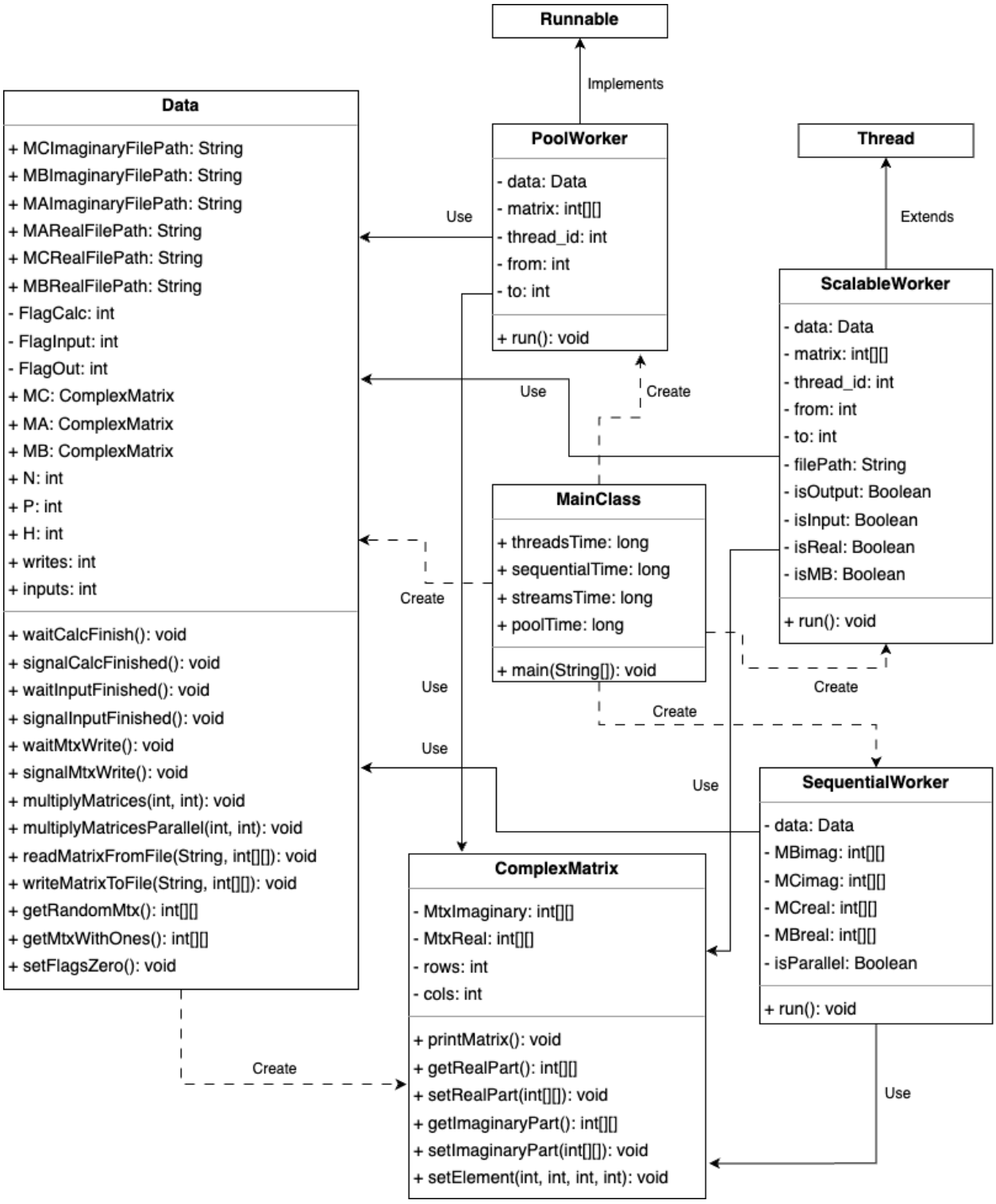
Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями

Діаграма класів (функціональна схема)

ІАЛЦ.466500.006 Д3

Аркушів 1

Київ 2024 р



					ІАЛЦ.466500.006 Д3		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Бондарчук А.Д.			**Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Корочкін О.В.					
Н. Контр.		Волокита А.М.			**Діаграма класів (функціональна схема)**	КПІ ім. Ігоря Сікорського ФІОТ, ІП-05	
Затвердив		Стіренко С.Г.					

ДОДАТОК 4

Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями

Текст програмного коду
ІАЛЦ.466500.007 Д4

Аркушів 1

Київ 2024 р

MainClass.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainClass {
  public static long sequentialTime;
  public static long threadsTime;
  public static long streamsTime;
  public static long poolTime;

  public static void main(String[] args) {
    Data Monitor = new Data();

    // Randomize input data for test purposes
    {
      Monitor.writeMatrixToFile(Data.MBRealFilePath, Monitor.getRandomMtx());
      Monitor.writeMatrixToFile(Data.MBImaginaryFilePath, Monitor.getRandomMtx());
      Monitor.writeMatrixToFile(Data.MCRealFilePath, Monitor.getRandomMtx());
      Monitor.writeMatrixToFile(Data.MCImaginaryFilePath, Monitor.getRandomMtx());
    }

    // Sequential
    {
      long seqTimeStart = System.currentTimeMillis();

      SequentialWorker sequential = new SequentialWorker(Monitor, false);
      sequential.run();

      sequentialTime = System.currentTimeMillis() - seqTimeStart;
      Monitor.setFlagsZero();
    }

    // Threads
    {
      long threadsTimeStart = System.currentTimeMillis();

      ScalableWorker[] workers = new ScalableWorker[Data.P];

      String[] filePaths = { Data.MBRealFilePath, Data.MBImaginaryFilePath,
          Data.MCRealFilePath, Data.MCImaginaryFilePath };
      Boolean[] isReals = { true, false, true, false };

      for (int i = 0; i < Data.P; i++) {
        workers[i] = new ScalableWorker(Monitor,
            i, i < 4 ? filePaths[i] : null,
            i < 4 ? true : false,
            i < 4 ? isReals[i] : false,
            i < 2 ? true : false,
            i < 2 ? true : false);
        workers[i].start();
      }

      Monitor.waitMtxWrite();
      threadsTime = System.currentTimeMillis() - threadsTimeStart;
      Monitor.setFlagsZero();
    }
```

```
    // IntStream
    {
      long streamsTimeStart = System.currentTimeMillis();

      SequentialWorker sequential = new SequentialWorker(Monitor, true);
      sequential.run();

      streamsTime = System.currentTimeMillis() - streamsTimeStart;
    }

    // Thread pool
    {
      long poolTimeStart = System.currentTimeMillis();

      ExecutorService executor = Executors.newFixedThreadPool(Data.P);

      for (int i = 0; i < Data.P; i++) {
        executor.execute(new PoolWorker(Monitor, i));
      }

      executor.shutdown();
      while (!executor.isTerminated()) {
      }

      poolTime = System.currentTimeMillis() - poolTimeStart;

      Monitor.setFlagsZero();
    }

    System.out.println("\nComputation Results:");
    System.out.println("---------------------");
    System.out.println("Versions of computation:");
    System.out.println("1. Sequential (control)");
    System.out.println("2. Threads");
    System.out.println("3. IntStream");
    System.out.println("4. Thread pool");
    System.out.println();
    System.out.println("Time of computation:");
    System.out.println("1. Sequential: " + sequentialTime + " ms");
    System.out.println("2. Threads: " + threadsTime + " ms");
    System.out.println("3. IntStream: " + streamsTime + " ms");
    System.out.println("4. Thread pool: " + poolTime + " ms");
    System.out.println();
    System.out.println("Speedup:");
    System.out.println("Threads: " + (sequentialTime / (double) threadsTime));
    System.out.println("IntStream: " + (sequentialTime / (double) streamsTime));
    System.out.println("Thread pool: " + (sequentialTime / (double) poolTime));
    System.out.println();
    System.out.println("Matrix size N = " + Data.N);
    System.out.println("Number of processors P = " + Data.P);
  }
}
```

Data.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;
```

```
import java.util.stream.IntStream;

public class Data {
  public static int N = 1600;
  public static int P = Runtime.getRuntime().availableProcessors();
  public static int H = N / P;
  public static int inputs = 4;
  public static int writes = 2;

  public static String MBRealFilePath = "./data/MBreal.txt";
  public static String MBImaginaryFilePath = "./data/MBimaginary.txt";

  public static String MCRealFilePath = "./data/MCreal.txt";
  public static String MCImaginaryFilePath = "./data/MCimaginary.txt";

  public static String MARealFilePath = "./data/MAreal.txt";
  public static String MAImaginaryFilePath = "./data/MAimaginary.txt";

  public ComplexMatrix MB;
  public ComplexMatrix MC;
  public ComplexMatrix MA;

  public Data() {
    MB = new ComplexMatrix(N, N);
    MC = new ComplexMatrix(N, N);
    MA = new ComplexMatrix(N, N);
  }

  private int FlagInput = 0;
  private int FlagCalc = 0;
  private int FlagOut = 0;

  public void setFlagsZero() {
    FlagInput = 0;
    FlagCalc = 0;
    FlagOut = 0;
  }

  public synchronized void waitInputFinished() {
    try {
      while (FlagInput < inputs) {
        wait();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public synchronized void signalInputFinished() {
    FlagInput += 1;
    if (FlagInput == inputs) {
      notifyAll();
    }
  }

  public synchronized void waitCalcFinish() {
    try {
      while (FlagCalc < P) {
        wait();
      }
    } catch (Exception e) {
```

```
      e.printStackTrace();
    }
  }

  public synchronized void signalCalcFinished() {
    FlagCalc += 1;
    if (FlagCalc == P) {
      notifyAll();
    }
  }

  public synchronized void waitMtxWrite() {
    try {
      while (FlagOut < writes) {
        wait();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public synchronized void signalMtxWrite() {
    FlagOut += 1;
    if (FlagOut == writes) {
      notifyAll();
    }
  }

  public void multiplyMatrices(int from, int to) {
    for (int i = from; i < to; i++) {
      for (int j = 0; j < N; j++) {
        int realSum = 0, imagSum = 0;
        for (int k = 0; k < N; k++) {
          int realMB = MB.getRealPart()[i][k];
          int imagMB = MB.getImaginaryPart()[i][k];
          int realMC = MC.getRealPart()[k][j];
          int imagMC = MC.getImaginaryPart()[k][j];

          // Calculation 1: MAxH = MBx * MCxH - MBy * MCyH
          realSum += realMB * realMC - imagMB * imagMC;

          // Calculation 2: MAyH = MBx * MCyH + MBy * MCxH
          imagSum += realMB * imagMC + imagMB * realMC;
        }
        MA.setElement(i, j, realSum, imagSum);
      }
    }
  }

  public void multiplyMatricesParallel(int from, int to) {
    IntStream.range(from, to).parallel().forEach(i -> {
      for (int j = 0; j < N; j++) {
        int realSum = 0, imagSum = 0;
        for (int k = 0; k < N; k++) {
          int realMB = MB.getRealPart()[i][k];
          int imagMB = MB.getImaginaryPart()[i][k];
          int realMC = MC.getRealPart()[k][j];
          int imagMC = MC.getImaginaryPart()[k][j];

          // Calculation 1: MAxH = MBx * MCxH - MBy * MCyH
          realSum += realMB * realMC - imagMB * imagMC;
```

```
            // Calculation 2: MAyH = MBx * MCyH + MBy * MCxH
            imagSum += realMB * imagMC + imagMB * realMC;
          }
          MA.setElement(i, j, realSum, imagSum);
        }
      });
    }

    public void readMatrixFromFile(String filename, int[][] matrix) {
      try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        String line;
        int row = 0;
        while ((line = br.readLine()) != null) {
          String[] elements = line.split(" ");
          for (int col = 0; col < elements.length; col++) {
            matrix[row][col] = Integer.parseInt(elements[col]);
          }
          row++;
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

    public void writeMatrixToFile(String filename, int[][] matrix) {
      try (BufferedWriter bw = new BufferedWriter(new FileWriter(filename))) {
        for (int i = 0; i < N; i++) {
          for (int j = 0; j < N; j++) {
            bw.write(matrix[i][j] + " ");
          }
          bw.newLine();
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

    public int[][] getRandomMtx() {
      Random rand = new Random();
      int min = 1, max = 10;

      int[][] mtx = new int[N][N];
      for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
          mtx[i][j] = rand.nextInt((max - min) + 1) + min;
        }
      }
      return mtx;
    }

    public int[][] getMtxWithOnes() {
      int[][] mtx = new int[N][N];
      for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
          mtx[i][j] = 1;
        }
      }
      return mtx;
    }
  }
```

ComplexMatrix.java

```
class ComplexMatrix {
  private int[][] MtxReal;
  private int[][] MtxImaginary;
  private int rows;
  private int cols;

  public ComplexMatrix(int rows, int cols) {
    this.rows = rows;
    this.cols = cols;
    MtxReal = new int[rows][cols];
    MtxImaginary = new int[rows][cols];
  }

  public int[][] getRealPart() {
    return MtxReal;
  }

  public int[][] getImaginaryPart() {
    return MtxImaginary;
  }

  public void setRealPart(int[][] realMtx) {
    MtxReal = realMtx;
  }

  public void setImaginaryPart(int[][] imaginaryMtx) {
    MtxImaginary = imaginaryMtx;
  }

  public void setElement(int row, int col, int real, int imaginary) {
    MtxReal[row][col] = real;
    MtxImaginary[row][col] = imaginary;
  }

  public void printMatrix() {
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
        System.out.print("(" + MtxReal[i][j] + " + " + MtxImaginary[i][j] + "i) ");
      }
      System.out.println("");
    }
  }
}
```

ScalableWorker.java

```
public class ScalableWorker extends Thread {
  private Data data;
  private int thread_id, from, to;
  private String filePath;
  private Boolean isInput, isOutput, isReal, isMB;
  private int[][] matrix;

  public ScalableWorker(Data D, int id, String filePath,
      Boolean isInput, Boolean isReal, Boolean isMB, Boolean isOutput) {
    this.data = D;
    this.thread_id = id;
    this.from = thread_id * Data.H;
    this.to = (thread_id + 1) * Data.H;

    this.filePath = filePath;
```

```
    this.isInput = isInput;
    this.isOutput = isOutput;
    this.isReal = isReal;
    this.isMB = isMB;

    if (this.isInput) {
      this.matrix = new int[Data.N][Data.N];
    }
  }

  @Override
  public void run() {
    try {
      if (isInput) {
        data.readMatrixFromFile(filePath, matrix);
        if (isMB) {
          if (isReal)
            data.MB.setRealPart(matrix);
          else
            data.MB.setImaginaryPart(matrix);
        } else {
          if (isReal)
            data.MC.setRealPart(matrix);
          else
            data.MC.setImaginaryPart(matrix);
        }
        data.signalInputFinished();
      }

      data.waitInputFinished();

      data.multiplyMatrices(from, to);

      data.signalCalcFinished();
      data.waitCalcFinish();

      if (isOutput) {
        if (isReal) {
          data.writeMatrixToFile(Data.MARealFilePath, data.MA.getRealPart());
        } else {
          data.writeMatrixToFile(Data.MAImaginaryFilePath, data.MA.getImaginaryPart());
        }
        data.signalMtxWrite();
      }

    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

PoolWorker.java

```
class PoolWorker implements Runnable {
  private Data data;
  private int thread_id, from, to;
  private int[][] matrix;

  public PoolWorker(Data D, int id) {
    this.data = D;
    this.thread_id = id;
    this.from = thread_id * Data.H;
```

```
    this.to = (thread_id + 1) * Data.H;

    if (this.thread_id <= 3) {
      this.matrix = new int[Data.N][Data.N];
    }
  }

  @Override
  public void run() {
    try {
      if (thread_id == 0) {
        data.readMatrixFromFile(Data.MBRealFilePath, matrix);
        data.MB.setRealPart(matrix);

        data.signalInputFinished();
      } else if (thread_id == 1) {
        data.readMatrixFromFile(Data.MBImaginaryFilePath, matrix);
        data.MB.setImaginaryPart(matrix);

        data.signalInputFinished();
      } else if (thread_id == 2) {
        data.readMatrixFromFile(Data.MCRealFilePath, matrix);
        data.MC.setRealPart(matrix);

        data.signalInputFinished();
      } else if (thread_id == 3) {
        data.readMatrixFromFile(Data.MCImaginaryFilePath, matrix);
        data.MC.setImaginaryPart(matrix);

        data.signalInputFinished();
      }

      data.waitInputFinished();

      data.multiplyMatrices(from, to);

      data.signalCalcFinished();
      data.waitCalcFinish();

      if (thread_id == 0) {
        data.writeMatrixToFile(Data.MARealFilePath, data.MA.getRealPart());
      } else if (thread_id == 1) {
        data.writeMatrixToFile(Data.MAImaginaryFilePath, data.MA.getImaginaryPart());
      }

    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

SequentialWorker.java

```
class SequentialWorker {
  private Data data;
  private Boolean isParallel;
  private int[][] MBreal, MBimag, MCreal, MCimag;

  public SequentialWorker(Data D, Boolean isParallel) {
    this.data = D;
    this.isParallel = isParallel;
    this.MBreal = new int[Data.N][Data.N];
```

```
      this.MBimag = new int[Data.N][Data.N];
      this.MCreal = new int[Data.N][Data.N];
      this.MCimag = new int[Data.N][Data.N];
    }

    public void run() {
      try {
        data.readMatrixFromFile(Data.MBRealFilePath, MBreal);
        data.readMatrixFromFile(Data.MBImaginaryFilePath, MBimag);

        data.MB.setRealPart(MBreal);
        data.MB.setImaginaryPart(MBimag);

        data.readMatrixFromFile(Data.MCRealFilePath, MCreal);
        data.readMatrixFromFile(Data.MCImaginaryFilePath, MCimag);

        data.MC.setRealPart(MCreal);
        data.MC.setImaginaryPart(MCimag);

        if (isParallel)
          data.multiplyMatricesParallel(0, Data.N);
        else
          data.multiplyMatrices(0, Data.N);

        data.writeMatrixToFile(Data.MARealFilePath, data.MA.getRealPart());
        data.writeMatrixToFile(Data.MAImaginaryFilePath, data.MA.getImaginaryPart());

      } catch (Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
```