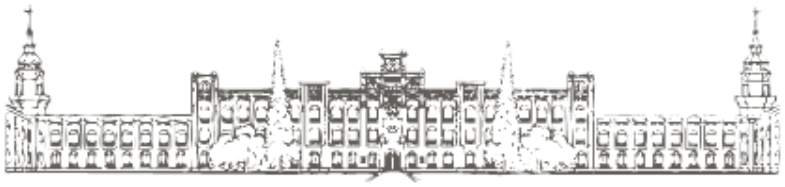

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Технології паралельних обчислень. Курсова робота»
Тема: «Алгоритм множення матриць (стрічковий)»

Керівник:
ст. викладач К.П. Нестеренко

«Допущено до захисту»
_____

«__» _____ 2025 р.

Захищено з оцінкою
_____

Члени комісії:

_____ Інна СТЕЦЕНКО

_____ Костянтин НЕСТЕРЕНКО

Виконавець:
Голяка Д.В.
студент групи ІП-21
залікова книжка № 07
_____

«26» березня 2025 р.

Київ – 2025

ЗАВДАННЯ

Курсова робота на тему «Алгоритм множення матриць (стрічковий)» спрямована на реалізацію та дослідження ефективності обчислювальних методів множення матриць, зокрема стрічкового підходу. У межах роботи необхідно реалізувати два варіанти послідовних алгоритмів – класичний та імітаційний, а також два паралельні варіанти на основі стрічкового обміну: з частковою передачею стовпців і з повною передачею матриці. Реалізовані алгоритми мають бути перевірені на коректність, після чого необхідно провести експериментальне дослідження продуктивності з використанням різних розмірів вхідних матриць та конфігурацій обчислювального середовища. Особливу увагу слід приділити аналізу прискорення, що досягається при переході до багатопотокового виконання, а також дослідженню масштабованості алгоритмів залежно від кількості доступних процесорних ядер. Очікуваним результатом є обґрунтовані висновки щодо доцільності використання паралельних рішень у задачах великої розмірності.

АНОТАЦІЯ

Пояснювальна записка курсової роботи складається з п’яти розділів, містить 4 таблиці, 8 рисунків та 8 джерел – загалом 50 сторінок.

Курсова робота присвячена дослідженню ефективності стрічкового алгоритму множення матриць у послідовних та паралельних обчислювальних середовищах.

Мета роботи – реалізація кількох варіантів алгоритму множення матриць, зокрема паралельних модифікацій із частковою та повною передачею даних, та проведення експериментального порівняльного аналізу їх продуктивності.

У розділі 1 розглянуто теоретичну основу класичних і відомих паралельних алгоритмів множення матриць, наведено псевдокоди реалізацій та зроблено аналітичний огляд.

Розділ 2 присвячений реалізації двох послідовних алгоритмів, їх верифікації та дослідженню швидкодії при різних розмірностях.

У розділі 3 обґрунтовано вибір платформи .NET та інструментів для реалізації паралельних обчислень, зокрема бібліотеки System.Threading.Tasks та структури BlockingCollection.

У розділі 4 описано проєктування, реалізацію та тестування двох паралельних варіантів стрічкового алгоритму: з частковим обміном стовпців та з повною передачею матриці.

У розділі 5 проведено порівняння ефективності реалізацій, досліджено прискорення та масштабованість залежно від кількості ядер процесора.

Програмне забезпечення розгорнуто на платформі .NET 9.0 з використанням локального середовища запуску та Docker-контейнерів для обмеження ресурсів.

КЛЮЧОВІ СЛОВА: МНОЖЕННЯ МАТРИЦЬ, ПАРАЛЕЛЬНИЙ АЛГОРИТМ, C#, .NET, БАГАТОПОТОКОВІСТЬ, THREADING, ПРИСКОРЕННЯ, DOCKER, СТРІЧКОВИЙ АЛГОРИТМ, BLOCKINGCOLLECTION, TPL.

ЗМІСТ

ВСТУП

У сучасних умовах розвитку інформаційних технологій обчислювальні ресурси відіграють ключову роль у забезпеченні ефективності програмних систем. Із зростанням обсягів оброблюваних даних та потребою у швидкому виконанні складних обчислень, традиційні послідовні алгоритми вже не можуть забезпечити необхідну продуктивність. Натомість паралельні обчислення дозволяють ефективно використовувати багатоядерні процесори, підвищуючи швидкодію та масштабованість програмних рішень. Завдяки паралельним підходам до обробки даних з'являється можливість суттєво зменшити час виконання ресурсоємних задач.

Однією з фундаментальних операцій в обчислювальній математиці та чисельному моделюванні є множення матриць. Ця операція лежить в основі багатьох прикладних задач – від обробки зображень і симуляцій фізичних процесів до алгоритмів машинного навчання. Одним із варіантів реалізації є стрічковий алгоритм множення матриць, який передбачає циклічну передачу стовпців між потоками. Такий підхід дає можливість ефективно адаптувати алгоритм до паралельного виконання, знижуючи витрати на синхронізацію та обмін даними. У свою чергу, дослідження подібних алгоритмів є актуальним для вдосконалення сучасних обчислювальних методів.

У даній курсовій роботі реалізовано чотири варіанти алгоритму множення матриць: класичний послідовний, імітований послідовний (із реалізацією стрічкової схеми), паралельний з частковим обміном (передача стовпців між потоками) та паралельний з повним обміном (передача всієї матриці). У ході роботи здійснено експериментальні дослідження часу виконання алгоритмів із різною розмірністю матриць, проведено оцінку прискорення та впливу кількості ядер на продуктивність.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Множення матриць є фундаментальною операцією в лінійній алгебрі, яка знаходить широке застосування в різних галузях, включаючи наукові обчислення, комп'ютерну графіку та машинне навчання. Ефективна реалізація цієї операції є критично важливою для підвищення продуктивності обчислювальних систем. У цьому розділі розглянуто послідовні та паралельні алгоритми множення матриць.

1.1 Класичний послідовний алгоритм

Класичний алгоритм множення матриць обчислює кожен елемент результуючої матриці як суму добутків відповідних елементів рядка першої матриці та стовпця другої [1]. Для двох матриць A розміру $n \times m$ та B розміру $m \times p$, елемент матриці C розміру $n \times p$ обчислюється за формулою 1.1:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} * b_{kj} \quad (1.1)$$

Псевдокод цього алгоритму виглядає наступним чином:

```
Function classical_matrix_multiplication
  Arguments: Matrix A (n x m), Matrix B (m x p)
  Create Matrix C (n x p) initialized to zero
  For i from 1 to n
    For j from 1 to p
      For k from 1 to m
        C[i][j] = C[i][j] + A[i][k] * B[k][j]
  Return Matrix C
```

1.2 Імітований послідовний алгоритм

Імітований послідовний алгоритм реалізує стрічкову схему множення матриць, але виконується у звичайному послідовному середовищі без використання потоків. Ідея полягає в тому, що кожен рядок матриці A множиться не на фіксований набір стовпців матриці B, а проходить циклічний зсув індексу стовпця. Це дозволяє імітувати поведінку стрічкового паралельного алгоритму, що корисно для відлагодження логіки або проведення порівняльного аналізу.

```
Function sequential_stripe_simulation
    Arguments: Matrix A (n x n), Matrix B (n x n)
    Create Matrix C (n x n) initialized to zero

    For i from 0 to n - 1
        row = A[i]
        colIndex = i

        For iter from 0 to n - 1
            column = B.getColumn(colIndex)
            C[i][colIndex] = dot_product(row, column)
            colIndex = (colIndex - 1 + n) mod n

    Return Matrix C
```

1.3 Паралельний стрічковий алгоритм

Цей підхід передбачає обробку кожного рядка матриці окремим потоком, із поступовим обміном стовпців або всієї матриці B [1]. У межах цієї роботи реалізовано два варіанти такого підходу: один із частковою передачею стовпців, інший – із передачею повної матриці B.

1.3.1 Стрічковий алгоритм із частковою передачею

Кожен потік отримує відповідний рядок матриці A та стовпець B. Після обчислення часткового добутку потік передає стовпець наступному потоку в кільці, а сам отримує новий стовпець від попереднього.

```
Function parallel_stripe_partial_exchange
    Arguments: Matrix A (n x n), Matrix B (n x n)
    Create Matrix C (n x n) initialized to zero

    Create array of mailboxes[0..n-1] to exchange columns between threads

    For each thread i in parallel
        rowA = A.getRow(i)
        colB = B.getColumn(i)
```

```
    currentColumn = (index: i, data: colB)
    resultRow = array of zeros with length n

    For iteration from 0 to n - 1
      dot = 0
      For k from 0 to n - 1
        dot += rowA[k] * currentColumn.data[k]
      resultRow[currentColumn.index] = dot

      If iteration < n - 1:
        nextIndex = (i + 1) mod n
        mailboxes[nextIndex].Add(currentColumn)
        currentColumn = mailboxes[i].Take()

    Set C[i] = resultRow

  Wait for all threads to complete
  Return Matrix C
```

1.3.2 Стрічковий алгоритм із повною передачею

У цьому варіанті кожен потік отримує повну копію матриці B, виконує обчислення всіх потрібних елементів результату для свого рядка, і не виконує передачу даних під час роботи.

```
Function parallel_stripe_full_exchange
  Arguments: Matrix A (n x n), Matrix B (n x n)
  Create Matrix C (n x n) initialized to zero

  For each thread i in parallel
    rowA = A.getRow(i)
    resultRow = array of zeros with length n
    currentColIndex = i

    For iteration from 0 to n - 1
      colB = B.getColumn(currentColIndex)
```

```
        dot = 0
        For k from 0 to n - 1
            dot += rowA[k] * colB[k]
        resultRow[currentColIndex] = dot

        currentColIndex = (currentColIndex - 1 + n) mod n

    Set C[i] = resultRow

Wait for all threads to complete
Return Matrix C
```

Ці два варіанти дозволяють порівняти ефективність різних стратегій обміну даними: зменшення обміну (часткова передача) проти спрощення логіки (повна передача).

1.4 Додаткові паралельні алгоритми

Окрім стрічкового підходу, існують інші відомі паралельні реалізації алгоритму множення матриць, що базуються на блокових або рекурсивних стратегіях.

1.4.1 Метод Кеннона

Алгоритм Кеннона базується на двовимірному розбитті матриць на блоки. Він мінімізує обсяг комунікацій, попередньо зсуваючи блоки A ліворуч, а B – вгору. Після цього на кожній ітерації виконується множення локальних блоків із подальшим зсувом блоків для наступних ітерацій. Це забезпечує ефективну паралелізацію при малій кількості пересилань [2].

1.4.2 Метод Фокса

Цей підхід також ґрунтується на блочному множенні. На кожному кроці блок одного з рядків A транслюється у відповідний рядок процесорної сітки, а блоки B зсуваються вниз. Локальне множення виконується на кожному процесорі. Алгоритм добре підходить для реалізації на синхронізованих системах [3].

1.4.3 Метод Штрассена

Цей алгоритм зменшує кількість множень за рахунок рекурсивного поділу матриць на підматриці та виконання лише семи замість восьми множень. Паралельна реалізація можлива на кожному рівні рекурсії. Основна перевага — зменшення теоретичної складності до $O(n^{2.81})$, проте є складність у реалізації та чутливість до розміру матриць [4].

У результаті аналізу відомих реалізацій алгоритмів множення матриць було вирішено зосередити увагу на реалізації чотирьох варіантів: двох послідовних та двох паралельних. Класичний послідовний алгоритм обрано як базову реалізацію, що дозволяє оцінити абсолютну ефективність паралельних підходів. Імітований послідовний варіант реалізує стрічкову логіку без багатопотоковості та дає змогу проаналізувати вплив схеми обчислень окремо від паралелізму.

Серед паралельних рішень розглянуто два варіанти стрічкового алгоритму: з частковою передачею стовпців та з передачею всієї матриці. Це дозволяє дослідити вплив схеми обміну даними на продуктивність: порівняння між оптимізацією комунікацій (часткова передача) та спрощенням обчислювальної логіки (повна передача) є особливо цінним з практичної точки зору.

Таке поєднання дозволяє комплексно охопити різні аспекти реалізації: алгоритмічну структуру, паралельність, ефективність обміну та масштабованість – що і стало головним критерієм вибору для подальшого дослідження.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

У цьому розділі розглянуто реалізацію двох послідовних алгоритмів множення матриць — класичного та імітаційного, а також здійснено їх перевірку на коректність та аналіз швидкодії. Основна мета – отримати еталонні результати для подальшого порівняння з паралельними реалізаціями.

2.1 Реалізація послідовних алгоритмів

У цьому підрозділі наведено реалізацію двох послідовних варіантів алгоритму множення матриць: класичного та імітаційного (повторює логіку стрічкового обміну без використання потоків). Обидва алгоритми реалізовано мовою програмування C# згідно з псевдокодами, поданими у розділі 1.

2.1.1 Класичний алгоритм

Класичний алгоритм реалізує базовий трьохцикловий підхід до множення матриць. На кожній ітерації обчислюється елемент C[i][j] як скалярний добуток i-го рядка матриці A та j-го стовпця матриці B. Нижче наведено вихідний код головного методу, повну версію класу можна переглянути у Додатку А.

```
public Matrix Multiply(Matrix a, Matrix b)
{
      int n = a.Rows;
      int m = b.Cols;
      int s = a.Cols;
      double[,] resultData = new double[n, m];

      for (int i = 0; i < n; i++)
      {
         for (int j = 0; j < m; j++)
         {
            double sum = 0;
            for (int k = 0; k < s; k++)
            {
               sum += a[i, k] * b[k, j];
            }

            resultData[i, j] = sum;
         }
      }

      return new Matrix(resultData);
   }
```

2.1.2 Імітаційний алгоритм

Даний алгоритм реалізує послідовну симуляцію стрічкового обчислення. Кожен рядок матриці A обчислюється з послідовним циклічним зміщенням індексу стовпця матриці B. Таким чином імітується обмін стовпців між потоками, що пізніше буде реалізовано в паралельній версії. Така структура максимально наближена до логіки паралельного стрічкового обчислення з обміном стовпців між потоками. Метод, що виконує множення наведено нижче. Повна версія наведена у Додатку Б.

```
public Matrix Multiply(Matrix a, Matrix b)
{
    int n = a.Rows;
    int s = a.Cols;
    int m = b.Cols;

    double[,] resultData = new double[n, m];

    for (int i = 0; i < n; i++)
    {
       double[] rowA = a.GetRow(i);
       int currentColIndex = i;

       for (int iter = 0; iter < m; iter++)
       {
          double dot = 0;
          for (int k = 0; k < s; k++)
          {
             dot += rowA[k] * b[k, currentColIndex];
          }

          resultData[i, currentColIndex] = dot;
          currentColIndex = (currentColIndex - 1 + m) % m;
       }
    }

    return new Matrix(resultData);
  }
```

2.2 Верифікація результатів

Для перевірки правильності реалізації обох послідовних алгоритмів була проведена верифікація результатів шляхом порівняння їх із результатами, отриманими за допомогою сторонньої математичної бібліотеки Math.NET

Numerics, яка є надійним і точним інструментом для чисельних обчислень у середовищі .NET [5].

У методі VerifyAll генеруються випадкові матриці заданого розміру. Вони перемножуються як за допомогою реалізованих алгоритмів (класичного та імітаційного), так і за допомогою Math.NET. Далі результати порівнюються на предмет рівності з допустимою похибкою. Повний вихідний код системи верифікації наведено у Додатку В.

```
internal static void VerifyAll(int dimension)
{
    Matrix matrixA = Matrix.GenerateRandomMatrix(dimension, dimension);
    Matrix matrixB = Matrix.GenerateRandomMatrix(dimension, dimension);
    Matrix verifiedResult = MultiplyWithMathNet(matrixA, matrixB);

    Matrix classicalResult = ClassicalMultiplier.Multiply(matrixA, matrixB);
    VerifyEquality(classicalResult, verifiedResult, "Classical multiplier");

    Matrix stripeMultiplier = StripeMultiplier.Multiply(matrixA, matrixB);
    VerifyEquality(stripeMultiplier, verifiedResult, "Stripe multiplier");
}
```

Для підтвердження коректності були виконані тести при розмірності матриць 128 та 1028 (рисунки 2.1 та 2.2). В обох випадках усі значення результатів збігалися з очікуваними, що свідчить про правильність реалізацій.

```
+ | Classical multiplier has successfully been verified
+ | Stripe multiplier has successfully been verified
```

Рисунок 2.1 – Результат верифікації на розмірності 128 елементів

```
+ | Classical multiplier has successfully been verified
+ | Stripe multiplier has successfully been verified
```

Рисунок 2.2 – Результат верифікації на розмірності 1028 елементів

2.3 Аналіз швидкодії

З метою оцінки продуктивності реалізованих послідовних алгоритмів було проведено експерименти з різними розмірами квадратних матриць. Для кожного алгоритму та розмірності виконувалося 20 прогонів, після чого обчислювалося середнє значення часу виконання. Такий підхід дозволяє зменшити вплив

флуктуацій, спричинених системними процесами, і отримати об’єктивну оцінку швидкодії.

У таблиці 2.1 наведено результати вимірювання часу виконання як класичного, так і імітаційного послідовного алгоритму. Для наочності ці результати візуалізовано на рисунку 2.3, де зображено залежність часу виконання від розмірності вхідної матриці.

Таблиця 2.1 – Середні заміри часу послідовних алгоритмів

Кількість елементів	Час класичного алгоритму, мілісекунд	Час імітаційного алгоритму, мілісекунд
64	1.56	1.57
128	4.75	4.09
256	38.95	29.03
512	371.67	403.00
1024	7877.34	6254.05
1536	16006.95	15099.41
2048	126735.81	75985.13
2560	263384.42	151062.62
3072	446932.37	270252.72
3584	764320.81	443090.97

Згідно з даними таблиці 2.1, при малій розмірності матриць різниця у часі виконання між алгоритмами незначна. Проте при збільшенні розміру імітаційний варіант починає випереджати класичний, що пов’язано з ефективнішим доступом до пам’яті завдяки зчитуванню рядка матриці A у вигляді одновимірного масиву.

Водночас, починаючи з розмірності 512, час виконання обох алгоритмів зростає експоненційно, що узгоджується з їх теоретичною складністю $O(n^3)$. Особливо помітним це стає при розмірах понад 1024, де час обчислень досягає десятків і сотень секунд. Зростання обсягу обчислень при збереженні одного

потоку виконання призводить до різкого погіршення продуктивності, що створює обґрунтовану необхідність переходу до паралельних обчислень.

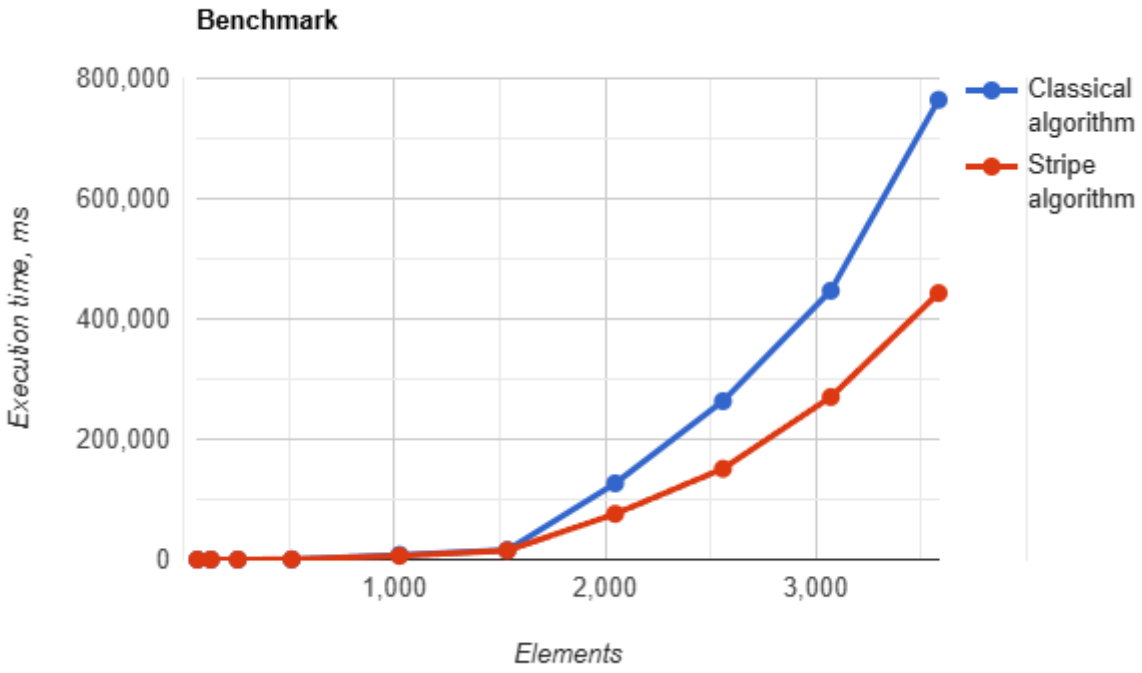

Рисунок 2.3 – Залежність часу виконання від розмірності матриць

У результаті реалізації та аналізу двох послідовних варіантів алгоритму множення матриць встановлено, що імітаційний підхід, попри однакову обчислювальну складність, демонструє кращу продуктивність через оптимізацію доступу до пам’яті. Обидві реалізації успішно пройшли верифікацію на базових тестах, що підтвердило правильність обчислень. Швидкодія обох алгоритмів суттєво погіршується з ростом розмірності, що створює підґрунтя для розробки паралельних рішень, які будуть розглянуті у наступних розділах.

3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

Для реалізації паралельних алгоритмів множення матриць у межах даної роботи було обрано платформу .NET, яка забезпечує зручні, продуктивні та кросплатформні засоби для розробки високорівневих обчислювальних задач. Завдяки наявності інтегрованих бібліотек для багатопотоковості та управління задачами, .NET є придатною платформою для реалізації алгоритмів, що інтенсивно використовують ресурси процесора [6].

Основним інструментом організації паралельних обчислень у роботі є .NET Task Parallel Library (TPL) – високорівнева бібліотека, що дозволяє ефективно створювати, запускати та координувати паралельні задачі. Клас Task надає зручну модель асинхронного виконання, дозволяючи запускати задачі у вигляді окремих обчислювальних потоків. Для контролю ресурсоємних обчислень використовується спеціальний параметр TaskCreationOptions.LongRunning, який сигналізує планувальнику задач про необхідність виділення окремого потоку для кожного обчислення [7].

Для синхронізації обміну даними між потоками в реалізації стрічкового алгоритму з частковим обміном застосовано BlockingCollection<T> - потоко-безпечну чергу, яка дозволяє організувати передавання даних між задачами без необхідності ручного блокування [8]. Це забезпечує безпечну взаємодію між потоками та дозволяє побудувати модель «producer-consumer», яка імітує обмін частинами матриць між обчислювальними вузлами.

Керування життєвим циклом задач реалізовано за допомогою масиву задач, який дозволяє одночасно запускати всі потоки та синхронно очікувати завершення обчислень методом Task.WaitAll. Такий підхід гарантує цілісність результатів, а також дозволяє легко масштабувати кількість потоків у залежності від розмірності вхідних даних.

Таким чином, обрана сукупність інструментів – .NET TPL, параметри LongRunning, механізм BlockingCollection<T> та колективне очікування задач –

дозволяє створювати надійні та масштабовані паралельні алгоритми, що добре підходять для обробки великих матричних структур.

4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

4.1 Проєктування

Паралельні реалізації алгоритму множення матриць базуються на принципі розподілу обчислень між потоками, де кожен потік відповідає за формування окремого рядка результатної матриці. Таке розбиття дозволяє організувати незалежну роботу потоків без конфліктів за спільні ресурси. При цьому структура обчислень наслідує логіку послідовного стрічкового алгоритму, зберігаючи циклічну схему зміщення стовпців на кожній ітерації.

Загальна модель роботи полягає у створенні множини робітників, кожен з яких виконує свою частину завдання в межах виділеного потоку. Потоки працюють паралельно та незалежно один від одного, а координація між ними відбувається за допомогою структур синхронізації. Це дозволяє реалізувати механізм передачі даних – у випадку з частковим обміном – або повного копіювання адреси другої матриці – у випадку повної передачі матриці.

Взаємодія між потоками організована згідно з кільцевою топологією: кожен потік передає певний обсяг даних наступному у визначеній послідовності. Це дозволяє рівномірно розподілити навантаження та уникнути блокувань. При цьому взаємодія є контрольованою та безпечною, що дозволяє забезпечити цілісність результатів у багатопоточному середовищі.

Життєвий цикл обчислення включає послідовні етапи: створення потоку, передача вхідних даних, виконання обчислень, передача проміжних результатів (у разі потреби), завершення роботи та передача результату до головного керуючого процесу. Завершення роботи всіх потоків сигналізує про готовність остаточної результатної матриці.

Управління паралельними задачами виконується централізовано, за допомогою ініціалізації потоків у керуючому компоненті. Потоки запускаються одночасно та очікуються до завершення за допомогою механізмів синхронізації.

Така модель дозволяє легко масштабувати обчислення залежно від доступної кількості апаратних ресурсів.

Загальна архітектура паралельного обчислення є розширюваною та придатною для подальших оптимізацій, таких як адаптація до середовищ з розподіленою пам’яттю або підтримка більш складних схем комунікацій. В межах цієї роботи обрана модель дозволяє повноцінно дослідити вплив багатопоточності на ефективність алгоритмів множення матриць.

4.2 Реалізація

У цьому підрозділі представлено реалізацію двох варіантів паралельного стрічкового алгоритму множення матриць, що базуються на принципі розподілу обчислень між потоками. Обидві реалізації використовують обрану платформу .NET та засоби паралельного програмування, описані в розділі 3.

Перший варіант передбачає часткову передачу даних між потоками, реалізовану у вигляді послідовної пересилки стовпців у кільцевій топології. Другий варіант використовує повну передачу – кожен потік отримує всю матрицю B, що дозволяє виконувати обчислення ізольовано без потреби взаємодії з іншими потоками під час виконання.

У подальших підрозділах наведено основні фрагменти коду та пояснення до кожного з варіантів реалізації.

4.2.1 Паралельний алгоритм із частковою передачею

У даній реалізації паралельного стрічкового алгоритму використовується часткова передача даних між потоками у вигляді послідовної пересилки стовпців матриці B. Кожен потік обробляє один рядок матриці A та отримує на початку лише один стовпець матриці B, відповідний своєму індексу. Після кожної ітерації обчислень цей стовпець передається наступному потоку за кільцем, а сам потік отримує новий стовпець від попереднього. Таким чином забезпечується поступове заповнення кожного рядка результатної матриці.

Управління передачею організоване через потокобезпечні структури, які дозволяють синхронізувати обмін між потоками без прямого блокування. Кожен

потік працює незалежно, отримуючи на вхід власний рядок, початковий стовпець та посилання на засоби обміну. По завершенню усіх ітерацій обчислений рядок записується у відповідну позицію результатної матриці.

Нижче наведено фрагмент коду, який демонструє запуск обчислень у потоках (див. Додаток Г):

```
for (int i = 0; i < n; i++)
{
   double[] rowA = a.GetRow(i);
   double[] colData = b.GetColumn(i);
   Column initialColumn = new(i, colData);

   StripeWorker stripeWorker = new(i, rowA, m, mailboxes, initialColumn, (index,
computedRow) =>
   {
      for (int j = 0; j < m; j++)
      {
         resultData[index, j] = computedRow[j];
      }
   });

   tasks[i] = Task.Factory.StartNew(stripeWorker.Execute,
TaskCreationOptions.LongRunning);
}
```

Основна логіка обчислень розміщена в окремому робітнику – класі StripeWorker. Він відповідає за обчислення одного рядка матриці результату та організацію обміну даними з іншими потоками. Ключовий фрагмент реалізації методу Execute наведено нижче (див. Додаток Г):

```
internal void Execute()
{
   double[] resultRow = new double[_iterations];
   Column currentColumn = _initialColumn;

   for (int iteration = 0; iteration < _iterations; iteration++)
   {
      double dot = _row
         .Select((rowValue, idx) => rowValue * currentColumn.Data[idx])
         .Sum();

      resultRow[currentColumn.Index] = dot;

      if (iteration >= _iterations - 1)
         continue;

      int nextWorkerIndex = (_workerIndex + 1) % _mailboxes.Length;
      _mailboxes[nextWorkerIndex].Add(currentColumn);
```

```
            currentColumn = _mailboxes[_workerIndex].Take();
        }

        _onCompleted(_workerIndex, resultRow);
    }
```

У кожній ітерації потік обчислює скалярний добуток власного рядка з поточним стовпцем. Результат записується у відповідну позицію результатного рядка. Після цього стовпець передається наступному потоку, а сам потік отримує новий стовпець для наступного обчислення. Завершення обчислень фіксується шляхом виклику делегата, що передає готовий рядок матриці назад у центральну координуючу структуру.

Завершення всіх обчислень координується методом Task.WaitAll, після чого збирається повна результатна матриця. Такий підхід дозволяє ефективно реалізувати стрічкову модель множення з контрольованим обміном, зберігаючи коректність обчислень навіть для великих обсягів даних.

4.2.2 Паралельний алгоритм із повною передачею

У цій версії паралельного стрічкового алгоритму кожен потік виконує обчислення свого рядка незалежно від інших, використовуючи копію матриці B. Такий підхід виключає необхідність обміну даними між потоками під час виконання, що спрощує управління та мінімізує витрати на синхронізацію.

Після запуску кожен потік виконує послідовне обчислення всіх елементів свого рядка, використовуючи механізм циклічного зміщення індексу стовпця. Це дозволяє зберегти стрічкову схему обчислення, аналогічну до попереднього варіанту, але без передачі стовпців.

Нижче наведено фрагмент запуску потоків, аналогічний до попереднього варіанту, однак з передачею повної матриці (див. Додаток Д):

```
for (int i = 0; i < n; i++)
{
    double[] rowA = a.GetRow(i);

    BulkWorker worker = new(i, rowA, b, m, (index, computedRow) =>
    {
        for (int j = 0; j < m; j++)
        {
            resultData[index, j] = computedRow[j];
        }
```

```
        });

        tasks[i] = Task.Factory.StartNew(worker.Execute, TaskCreationOptions.LongRunning);
    }
```

Весь обчислювальний процес відбувається локально в межах кожного потоку, як показано у ключовому фрагменті коду виконавця (див. Додаток Д):

```
internal void Execute()
{
    double[] resultRow = new double[_iterations];
    int currentColIndex = _workerIndex;

    for (int iter = 0; iter < _iterations; iter++)
    {
        double dot = _row
            .Select((rowValue, idx) => rowValue * _matrixB[idx, currentColIndex])
            .Sum();

        resultRow[currentColIndex] = dot;
        currentColIndex = (currentColIndex - 1 + _iterations) % _iterations;
    }

    _onCompleted(_workerIndex, resultRow);
}
```

На відміну від варіанту з частковою передачею, тут немає взаємодії між потоками після їхнього запуску. Це робить реалізацію більш прямолінійною, проте збільшує загальне споживання пам’яті.

Обидва підходи реалізують ту саму логіку множення, однак відрізняються організацією взаємодії та управлінням ресурсами.

4.3 Тестування

З метою перевірки коректності реалізованих паралельних алгоритмів було проведено верифікацію результатів множення матриць. Усі результати були зіставлені з еталонним обчисленням, виконаним за допомогою бібліотеки Math.NET Numerics, яка забезпечує достовірність математичних операцій у середовищі .NET [5].

Для верифікації використовувалися однакові випадково згенеровані квадратні матриці, які передавались на вхід як послідовним, так і паралельним реалізаціям. Результати обчислень кожного варіанту порівнювались із базовим результатом, і для кожного з них фіксувалося, чи є матриці тотожними з урахуванням допустимої похибки.

Фрагмент коду, що здійснює верифікацію всіх реалізацій, наведено нижче (див. Додаток В):

```
internal static void VerifyAll(int dimension)
{
    Matrix matrixA = Matrix.GenerateRandomMatrix(dimension, dimension);
    Matrix matrixB = Matrix.GenerateRandomMatrix(dimension, dimension);
    Matrix verifiedResult = MultiplyWithMathNet(matrixA, matrixB);

    Matrix classicalResult = ClassicalMultiplier.Multiply(matrixA, matrixB);
    VerifyEquality(classicalResult, verifiedResult, "Classical multiplier");

    Matrix stripeMultiplier = StripeMultiplier.Multiply(matrixA, matrixB);
    VerifyEquality(stripeMultiplier, verifiedResult, "Stripe multiplier");

    Matrix parallelStripeMultiplier = ParallelStripeMultiplier.Multiply(matrixA, matrixB);
    VerifyEquality(parallelStripeMultiplier, verifiedResult, "Parallel stripe multiplier
multiplier");

    Matrix parallelBulkMultiplier = ParallelBulkMultiplier.Multiply(matrixA, matrixB);
    VerifyEquality(parallelBulkMultiplier, verifiedResult, "Parallel bulk multiplier
multiplier");
}
```

Результати верифікації (рисунки 4.1 та 4.2) підтвердили коректність реалізації всіх алгоритмів, включаючи обидві паралельні версії. Перевірку було виконано для декількох розмірностей, зокрема 512×512 та 2048×2048, що дозволяє стверджувати про стабільність реалізацій як при помірних, так і великих об’ємах даних.

```
+ | Classical multiplier has successfully been verified
+ | Stripe multiplier has successfully been verified
+ | Parallel stripe multiplier multiplier has successfully been verified
+ | Parallel bulk multiplier multiplier has successfully been verified
```

Рисунок 4.1 – Результат верифікації для матриць розмірності 512

```
+ | Classical multiplier has successfully been verified
+ | Stripe multiplier has successfully been verified
+ | Parallel stripe multiplier multiplier has successfully been verified
+ | Parallel bulk multiplier multiplier has successfully been verified
```

Рисунок 4.2 – Результат верифікації для матриць розмірності 2048

Таким чином, усі реалізовані варіанти множення матриць успішно пройшли верифікацію. Це свідчить про правильність обчислень і дозволяє перейти до аналізу продуктивності, що буде розглянуто у наступному розділі.

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

У цьому розділі проведено повноцінний експериментальний аналіз продуктивності реалізованих алгоритмів множення матриць. Основну увагу зосереджено на оцінці ефективності двох паралельних стрічкових реалізацій, їх порівнянні з послідовними варіантами, дослідженні прискорення та масштабованості залежно від розмірності вхідних даних і кількості доступних процесорних ядер. Всі вимірювання виконано в контрольованому середовищі, а результати наведено у вигляді таблиць та графіків.

5.1 Середовище запуску та методика тестування

Для отримання достовірних та відтворюваних результатів усі експериментальні дослідження виконувалися в контрольованому середовищі з дотриманням однакових умов запуску. Основна частина вимірювань – включно з порівнянням послідовних і паралельних реалізацій, а також дослідженням прискорення – проводилася локально на фізичному пристрої. Перед запуском тестів було відключено інтернет-з'єднання, антивірусне програмне забезпечення, усі зайві фонові служби та додатки, щоб мінімізувати сторонній вплив на результати вимірювання.

Для вимірювання часу використовувався вбудований таймер Stopwatch, що забезпечує високу точність. Замір здійснювався безпосередньо навколо виклику методу множення матриць. Це дозволило ізолювати саме обчислювальний компонент алгоритму від зовнішніх операцій вводу-виводу чи ініціалізації. Відповідний код наведено у Додатку Е.

Для автоматизованого виконання численних незалежних прогонів було створено bash-скрипт, який виконував задану кількість запусків (20 для кожної конфігурації), обчислював середнє значення та відображав результати у зручному форматі. Цей скрипт використовувався для локального тестування та дозволив отримати репрезентативні дані для побудови графіків. Його лістинг наведено у Додатку Ж.

Дослідження впливу кількості ядер процесора виконувалося окремо у контейнеризованому середовищі, що дозволило точно регулювати доступ до обчислювальних ресурсів. Для цього було створено Docker-інфраструктуру з відповідним Dockerfile та конфігурацією compose.yaml, у якій явно задавалась кількість ядер за допомогою параметра cpus. Контейнерна ізоляція забезпечила стабільне середовище для вимірювань при різній кількості ядер. Весь необхідний код розміщено у Додатку Л а автоматизований скрипт запуску тестів — у Додатку К.

Такий підхід дозволив поєднати гнучкість локального запуску з точністю обмеження ресурсів у контейнері, що забезпечило достовірність результатів для всіх аспектів дослідження.

5.2 Порівняння паралельних реалізацій

На цьому етапі проведено порівняння двох реалізованих паралельних стрічкових алгоритмів: із частковою та повною передачею даних. Основною метою було з'ясувати, який з підходів забезпечує кращу швидкодію залежно від розміру вхідних даних.

У таблиці 5.1 наведено усереднені значення часу виконання для обох паралельних варіантів при різних розмірностях квадратних матриць після 20 прогонів.

Таблиця 5.1 – Середній час виконання паралельних реалізацій

Кількість елементів	Час алгоритму із частковою передачею, мілісекунд	Час алгоритму із повною передачею, мілісекунд
64	28.29	20.80
128	124.82	94.62
256	547.00	297.38
512	1089.11	813.36
1024	5413.65	3446.61
1536	14082.91	11233.97
2048	29610.64	51397.53

Продовження таблиці 5.1

Кількість елементів	Час алгоритму із частковою передачею, мілісекунд	Час алгоритму із повною передачею, мілісекунд
2560	56903.27	100188.82
3072	98921.80	172069.59
3584	157936.29	269924.47

Аналіз таблиці дозволяє зробити кілька висновків. При малих та середніх розмірах (до 2048) реалізація з повною передачею демонструє кращі результати. Це пояснюється простішою логікою виконання: потоки не очікують даних одне від одного, що знижує часові витрати на синхронізацію. Проте зі збільшенням розмірності ситуація змінюється: час виконання варіанту з повною передачею стрімко зростає. Це свідчить про зростаючий вплив надлишкового доступу до однакових комірок пам’яті великою кількістю потоків.

На відміну від цього, реалізація з частковою передачею демонструє більш стабільне зростання часу, зберігаючи перевагу при великих обсягах даних. Починаючи з розмірності 2560, ця версія працює помітно швидше, що робить її більш придатною для обчислень з великою розмірністю.

Графік на рисунку 5.1 ілюструє залежність часу виконання обох алгоритмів від розмірності матриць. Чітко видно перетин кривих, який вказує на зміну поведінки при збільшенні розміру вхідних даних.

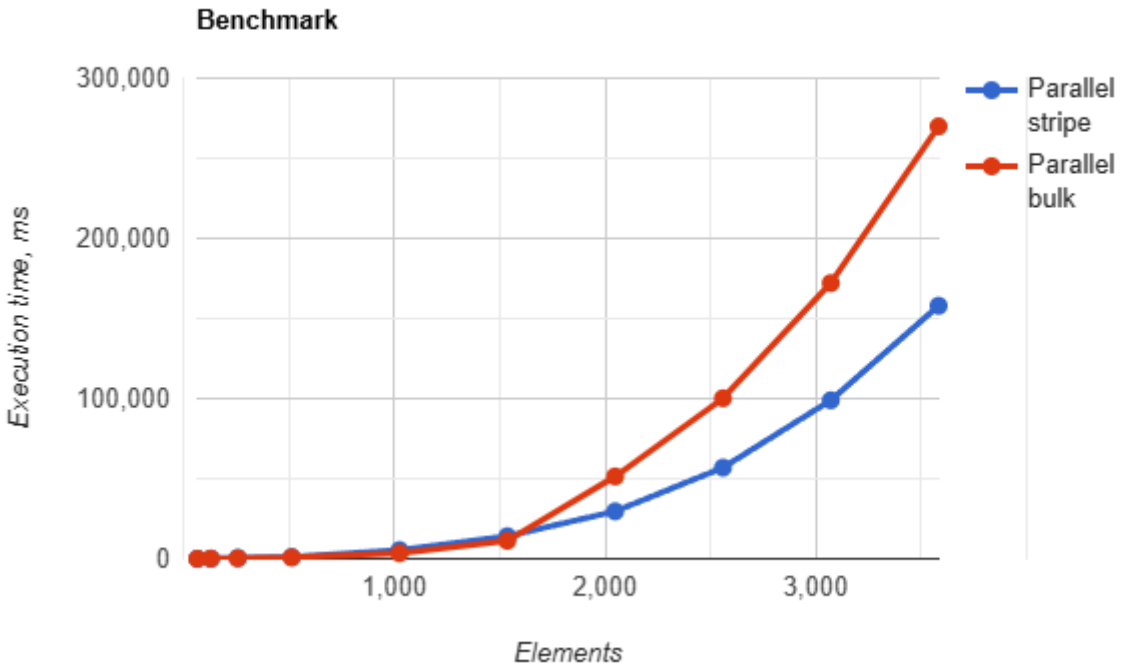

Рисунок 5.1 – Залежність кількості часу від розмірності

Таким чином, при виборі між цими двома підходами варто враховувати розмірність вхідних даних. Для невеликих задач більш доцільним є використання реалізації з повною передачею. Натомість для задач великого масштабу оптимальнішу продуктивність забезпечує реалізація з частковим обміном.

5.3 Дослідження прискорення

У цьому підрозділі досліджується прискорення, яке забезпечується використанням паралельного стрічкового алгоритму з частковою передачею даних у порівнянні з двома послідовними реалізаціями: класичною та імітаційною. Прискорення обчислюється як відношення часу виконання відповідного послідовного алгоритму до часу виконання паралельного алгоритму. Отримані значення дозволяють оцінити ефективність переходу до багатопоточного виконання.

У таблиці 5.2 наведено результати обчислення прискорення для різних розмірностей квадратних матриць.

Таблиця 5.2 – Прискорення алгоритму із частковою передачею

Кількість елементів	Прискорення відносно класичного алгоритму	Прискорення відносно імітованого алгоритму
64	0.06	0.06
128	0.04	0.03
256	0.07	0.05
512	0.34	0.37
1024	1.46	1.16
1536	1.14	1.07
2048	4.28	2.57
2560	4.63	2.66
3072	4.52	2.73
3584	4.84	2.81

Аналіз таблиці свідчить, що при малій розмірності ефект від паралельного виконання майже відсутній або навіть призводить до уповільнення. Це пов'язано з додатковими витратами на створення потоків та організацію обміну. Починаючи з розмірності 1024 спостерігається помітне зростання прискорення, а після 2048 — стабільна перевага паралельного алгоритму.

Варто зазначити, що прискорення відносно імітаційного алгоритму є меншим, оскільки він сам по собі працює ефективніше за класичний. Це зумовлено тим, що він зчитує рядки матриці A один раз і працює з ними як з одновимірними масивами, що покращує використання кешу та зменшує витрати на індексацію.

Графік на рисунку 5.2 ілюструє, як змінюється прискорення залежно від розмірності задачі. З нього видно, що для задач великого масштабу використання паралельного виконання є виправданим і дає значний виграш у часі.

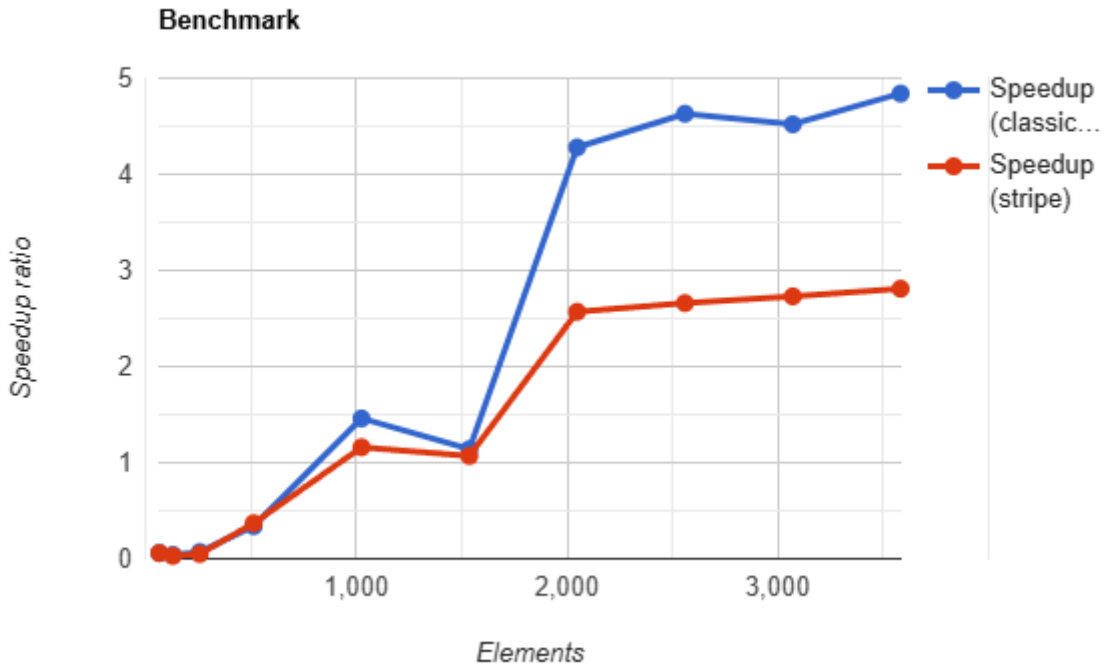

Рисунок 5.2 – Залежність прискорення від розмірності

Загалом, результати експерименту підтверджують, що застосування паралельного обчислення доцільне лише при достатньо великих вхідних даних. У таких випадках досягається істотне скорочення часу виконання, що свідчить про успішність реалізованого підходу.

5.4 Дослідження впливу кількості ядер

Щоб оцінити масштабованість розробленого паралельного алгоритму, було проведено експеримент зі зміною кількості доступних ядер процесора. Для чистоти вимірювання використовувалась фіксована розмірність матриці – 2560×2560, а виконання контролювалося за допомогою Docker-контейнерів.

У таблиці 5.3 наведено середній час виконання для двох алгоритмів: імітованого послідовного та паралельного з частковою передачею. Усі значення усереднено за 20 запусків для уникнення флуктуацій, пов’язаних із системними процесами.

Таблиця 5.3 – Вплив кількості ядер на час виконання

Кількість ядер	Час імітованого алгоритму, мілісекунд	Час алгоритму із частковою передачею, мілісекунд
1	127126.13	620932.92
2	134911.76	579031.14
4	139669.34	283922.93
6	136225.29	193803.56
8	133341.65	150272.78
10	132869.73	135091.39
12	129459.81	126568.61

Як видно з таблиці 5.3, час виконання імітаційного алгоритму залишається практично незмінним при збільшенні кількості ядер, що відповідає його послідовній природі. Натомість паралельний алгоритм демонструє поступове покращення продуктивності зі зростанням числа ядер. Найбільш істотне зменшення часу спостерігається між 2 і 8 ядрами, після чого ефект сповільнюється.

Рисунок 5.3 наочно демонструє залежність часу виконання паралельного алгоритму від кількості ядер.

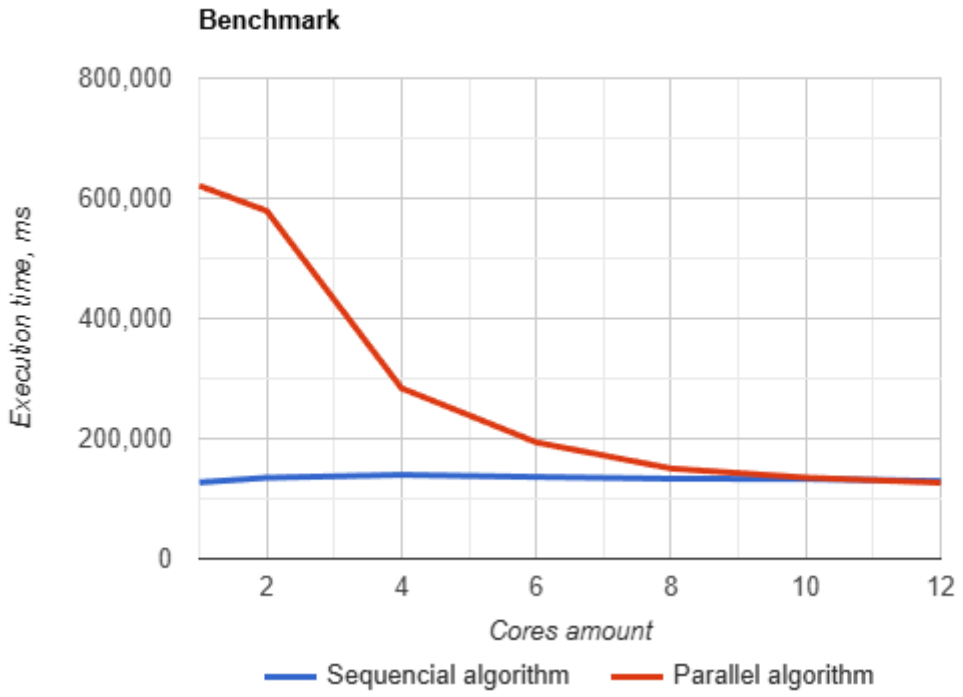

Рисунок 5.3 – Залежність часу виконання від кількості ядер

Результати підтверджують ефективність розробленого паралельного підходу при використанні багатоядерних процесорів. Водночас вони вказують на наявність межі, після якої збільшення кількості ядер не дає суттєвого виграшу через накладні витрати на організацію паралелізму та синхронізацію.

У результаті проведених експериментів встановлено, що реалізація з повною передачею даних є більш ефективною при малих розмірах матриць завдяки простішій логіці виконання. Натомість реалізація з частковим обміном демонструє стабільне масштабування і переважає при великих обсягах даних. Порівняння з послідовними алгоритмами засвідчило, що паралельний підхід стає ефективним починаючи з розмірності 1024×1024. Аналіз масштабованості також підтвердив доцільність багатопоточного виконання до певного порогу, після якого приріст продуктивності зменшується. Таким чином, у цьому розділі підтверджено практичну користь паралельних обчислень і визначено оптимальні умови їх застосування.

ВИСНОВКИ

У ході виконання курсової роботи було реалізовано та досліджено чотири варіанти алгоритму множення матриць: класичний послідовний, імітаційний послідовний, а також два паралельні стрічкові – з частковою та повною передачею даних. У першому розділі було проведено огляд відомих алгоритмів множення матриць, включаючи методи Кеннона, Фокса та Штрассена, а також представлено загальні підходи до їх реалізації. На основі цього аналізу було обґрунтовано доцільність вибору стрічкового алгоритму для подальшого дослідження.

У другому розділі реалізовано два варіанти послідовних алгоритмів, які пройшли верифікацію за допомогою Math.NET Numerics. Було показано, що імітаційний алгоритм демонструє кращу продуктивність за класичний завдяки зменшенню накладних витрат на доступ до пам’яті. Проте обидва алгоритми швидко втрачають ефективність при зростанні розмірності задачі, що підкреслило необхідність переходу до паралельних рішень.

У третьому розділі обґрунтовано вибір технологічної платформи .NET, бібліотеки TPL для багатопотоковості та структури BlockingCollection для безпечного обміну даними між потоками. Обрані інструменти дозволили створити надійне, масштабоване та контрольоване середовище для реалізації паралельного множення матриць.

У четвертому розділі описано проєктування та реалізацію двох паралельних стрічкових алгоритмів. Варіант із частковою передачею забезпечив точну імітацію кільцевого обміну стовпцями між потоками, тоді як варіант із повною передачею спростив логіку за рахунок ізоляції потоків. Обидві реалізації успішно пройшли верифікацію та продемонстрували правильність обчислень.

П’ятий розділ присвячено детальному дослідженню ефективності реалізованих алгоритмів. Зокрема, було встановлено, що реалізація з повною передачею є ефективнішою при невеликих розмірах матриць, тоді як реалізація з частковим обміном переважає при великих обсягах даних, починаючи з розмірності 2560. Максимальне зафіксоване прискорення становило 4.84

відносно класичного алгоритму та 2.81 відносно імітаційного. Дослідження масштабованості вказало на поступове зниження часу виконання паралельного алгоритму зі збільшенням кількості ядер, із найбільш помітним ефектом до 8 ядер.

Таким чином, поставлену мету досягнуто. Розроблено, протестовано та проаналізовано стрічковий алгоритм множення матриць у послідовному та паралельному виконанні. Отримані результати дозволяють зробити висновок про ефективність багатопотокового підходу для задач великої розмірності, а також визначити оптимальні умови його застосування залежно від характеристик обчислювального середовища.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Matrix multiplication in parallel using Open MPI // Anjana’s Tech Blog. URL: https://anjanavk.wordpress.com/2011/01/08/matrix-multiplication-in-parallel-using-open-mpi/ (дата звернення: 19.03.2025).

2. Cannon’s algorithm for distributed matrix multiplication // OpenGenus IQ. URL: https://iq.opengenus.org/cannon-algorithm-distributed-matrix-multiplication (дата звернення: 20.03.2025).

3. MPI Matrix-Matrix Multiplication – Fox’s Algorithm. San Diego State University. URL: https://edoras.sdsu.edu/~mthomas/sp17.605/lectures/MPI-MatMatMult.pdf (дата звернення: 20.03.2025).

4. Earnest R. Comparison of Strassen’s Algorithm with the Standard Matrix Multiplication Algorithm. UMBC, 2010. URL: https://web.archive.org/web/20100612150812/http://www.mc2.umbc.edu/docs/earnest.pdf (дата звернення: 21.03.2025).

5. Math.NET Numerics – Numerical Foundation for .NET. URL: https://numerics.mathdotnet.com/ (дата звернення: 21.03.2025).

6. Introduction to .NET Core // Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/core/introduction (дата звернення: 22.03.2025).

7. ThreadPoolTaskScheduler.cs – .NET Runtime Source Code // GitHub. URL: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Threading/Tasks/ThreadPoolTaskScheduler.cs (дата звернення: 22.03.2025).

8. System.Collections.Concurrent.BlockingCollection<T> Class // Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1?view=net-9.0 (дата звернення: 22.03.2025).

ДОДАТКИ

Додаток А. Послідовний класичний алгоритм

```
using MatrixCompute.Core.Abstractions;
using MatrixCompute.Core.Models;

namespace MatrixCompute.Core.Multipliers.Classical;

public class ClassicalMultiplier : IMultiplier
{
   public Matrix Multiply(Matrix a, Matrix b)
   {
      if (a.Cols != b.Rows)
      {
         throw new ArgumentException("Invalid matrix dimensions. A.Cols must equal
B.Rows.");
      }

      int n = a.Rows;
      int m = b.Cols;
      int s = a.Cols;
      double[,] resultData = new double[n, m];

      for (int i = 0; i < n; i++)
      {
         for (int j = 0; j < m; j++)
         {
            double sum = 0;
            for (int k = 0; k < s; k++)
            {
               sum += a[i, k] * b[k, j];
            }

            resultData[i, j] = sum;
         }
      }

      return new Matrix(resultData);
   }
}
```

Додаток Б. Послідовний імітований алгоритм

```
using MatrixCompute.Core.Abstractions;
using MatrixCompute.Core.Models;

namespace MatrixCompute.Core.Multipliers.Stripe;

public class StripeMultiplier : IMultiplier
{
   public Matrix Multiply(Matrix a, Matrix b)
   {
      if (a.Cols != b.Rows)
      {
         throw new ArgumentException("Invalid matrix dimensions. A.Cols must equal
B.Rows.");
      }

      int n = a.Rows;
      int s = a.Cols;
      int m = b.Cols;

      if (m != n)
      {
         throw new NotSupportedException("Stripe algorithm requires the number of
columns in B " +
                                         "to equal the number of rows in A.");
      }

      double[,] resultData = new double[n, m];

      for (int i = 0; i < n; i++)
      {
         double[] rowA = a.GetRow(i);
         int currentColIndex = i;

         for (int iter = 0; iter < m; iter++)
         {
            double dot = 0;
            for (int k = 0; k < s; k++)
            {
               dot += rowA[k] * b[k, currentColIndex];
            }

            resultData[i, currentColIndex] = dot;
            currentColIndex = (currentColIndex - 1 + m) % m;
         }
      }

      return new Matrix(resultData);
   }
}
```

Додаток В. Верифікація обчислення

```
using MathNet.Numerics.LinearAlgebra.Double;
using MatrixCompute.Core.Multipliers.Classical;
using MatrixCompute.Core.Multipliers.ParallelBulk;
using MatrixCompute.Core.Multipliers.ParallelStripe;
using MatrixCompute.Core.Multipliers.Stripe;
using Matrix = MatrixCompute.Core.Models.Matrix;

namespace MatrixCompute.Runner.Utils;

internal static class Verifier
{
   private static readonly ClassicalMultiplier ClassicalMultiplier = new();
   private static readonly StripeMultiplier StripeMultiplier = new();
   private static readonly ParallelStripeMultiplier ParallelStripeMultiplier = new();
   private static readonly ParallelBulkMultiplier ParallelBulkMultiplier = new();

   internal static void VerifyAll(int dimension)
   {
      Matrix matrixA = Matrix.GenerateRandomMatrix(dimension, dimension);
      Matrix matrixB = Matrix.GenerateRandomMatrix(dimension, dimension);
      Matrix verifiedResult = MultiplyWithMathNet(matrixA, matrixB);

      Matrix classicalResult = ClassicalMultiplier.Multiply(matrixA, matrixB);
      VerifyEquality(classicalResult, verifiedResult, "Classical multiplier");

      Matrix stripeMultiplier = StripeMultiplier.Multiply(matrixA, matrixB);
      VerifyEquality(stripeMultiplier, verifiedResult, "Stripe multiplier");

      Matrix parallelStripeMultiplier = ParallelStripeMultiplier.Multiply(matrixA, matrixB);
      VerifyEquality(parallelStripeMultiplier, verifiedResult, "Parallel stripe multiplier
multiplier");

      Matrix parallelBulkMultiplier = ParallelBulkMultiplier.Multiply(matrixA, matrixB);
      VerifyEquality(parallelBulkMultiplier, verifiedResult, "Parallel bulk multiplier
multiplier");
   }

   private static void VerifyEquality(Matrix calculatedResult, Matrix verifiedResult, string
multiplier)
   {
      Console.WriteLine(AreMatricesEqual(calculatedResult, verifiedResult)
         ? $"+ | {multiplier} has successfully been verified"
         : $"- | {multiplier} hasn't been verified");
   }

   private static Matrix MultiplyWithMathNet(Matrix a, Matrix b)
   {
      DenseMatrix matrixA = DenseMatrix.OfArray(a.Data);
      DenseMatrix matrixB = DenseMatrix.OfArray(b.Data);
      DenseMatrix result = matrixA * matrixB;
```

```
        return new Matrix(result.ToArray());
    }

    private static bool AreMatricesEqual(Matrix a, Matrix b, double tolerance = 1e-9)
    {
        if (a.Rows != b.Rows || a.Cols != b.Cols)
        {
            return false;
        }

        for (int i = 0; i < a.Rows; i++)
        {
            for (int j = 0; j < a.Cols; j++)
            {
                if (Math.Abs(a[i, j] - b[i, j]) > tolerance)
                {
                    return false;
                }
            }
        }

        return true;
    }
}
```

Додаток Г. Паралельний алгоритм із частковою передачею

Клас ParallelStripeMultiplier

```
using System.Collections.Concurrent;
using MatrixCompute.Core.Abstractions;
using MatrixCompute.Core.Models;

namespace MatrixCompute.Core.Multipliers.ParallelStripe;

public class ParallelStripeMultiplier : IMultiplier
{
   public Matrix Multiply(Matrix a, Matrix b)
   {
      if (a.Cols != b.Rows)
      {
         throw new ArgumentException("Invalid matrix dimensions. A.Cols must equal
B.Rows.");
      }

      int n = a.Rows;
      int s = a.Cols;
      int m = b.Cols;

      if (m != n)
      {
         throw new NotSupportedException("Stripe algorithm requires the number of
columns in B " +
                                         "to equal the number of rows in A.");
      }

      double[,] resultData = new double[n, m];

      BlockingCollection<Column>[] mailboxes = new BlockingCollection<Column>[n];
      for (int i = 0; i < n; i++)
      {
         mailboxes[i] = new BlockingCollection<Column>(1);
      }

      Task[] tasks = new Task[n];

      for (int i = 0; i < n; i++)
      {
         double[] rowA = a.GetRow(i);
         double[] colData = b.GetColumn(i);
         Column initialColumn = new(i, colData);

         StripeWorker stripeWorker = new(i, rowA, m, mailboxes, initialColumn, (index,
computedRow) =>
         {
            for (int j = 0; j < m; j++)
            {
```

```
                resultData[index, j] = computedRow[j];
            }
        });

        tasks[i] = Task.Factory.StartNew(stripeWorker.Execute,
TaskCreationOptions.LongRunning);
    }

    Task.WaitAll(tasks);
    return new Matrix(resultData);
  }
}
```

Клас StripeWorker

```
using System.Collections.Concurrent;
using MatrixCompute.Core.Models;

namespace MatrixCompute.Core.Multipliers.ParallelStripe;

internal class StripeWorker
{
  private readonly int _workerIndex;
  private readonly double[] _row;
  private readonly int _iterations;
  private readonly BlockingCollection<Column>[] _mailboxes;
  private readonly Column _initialColumn;
  private readonly Action<int, double[]> _onCompleted;

  internal StripeWorker(
    int workerIndex,
    double[] row,
    int iterations,
    BlockingCollection<Column>[] mailboxes,
    Column initialColumn,
    Action<int, double[]> onCompleted)
  {
    _workerIndex = workerIndex;
    _row = row;
    _iterations = iterations;
    _mailboxes = mailboxes;
    _initialColumn = initialColumn;
    _onCompleted = onCompleted;
  }

  internal void Execute()
  {
    double[] resultRow = new double[_iterations];
    Column currentColumn = _initialColumn;

    for (int iteration = 0; iteration < _iterations; iteration++)
    {
```

```
            double dot = _row
                .Select((rowValue, idx) => rowValue * currentColumn.Data[idx])
                .Sum();

            resultRow[currentColumn.Index] = dot;

            if (iteration >= _iterations - 1)
            {
                continue;
            }

            int nextWorkerIndex = (_workerIndex + 1) % _mailboxes.Length;
            _mailboxes[nextWorkerIndex].Add(currentColumn);
            currentColumn = _mailboxes[_workerIndex].Take();
        }

        _onCompleted(_workerIndex, resultRow);
    }
}
```

Додаток Д. Паралельний алгоритм із повною передачею

Клас ParallelBulkMultiplier

```
using MatrixCompute.Core.Abstractions;
using MatrixCompute.Core.Models;

namespace MatrixCompute.Core.Multipliers.ParallelBulk;

public class ParallelBulkMultiplier : IMultiplier
{
   public Matrix Multiply(Matrix a, Matrix b)
   {
      if (a.Cols != b.Rows)
      {
         throw new ArgumentException("Invalid matrix dimensions. A.Cols must equal
B.Rows.");
      }

      int n = a.Rows;
      int s = a.Cols;
      int m = b.Cols;

      if (m != n)
      {
         throw new NotSupportedException("Stripe algorithm requires the number of
columns in B " +
                                         "to equal the number of rows in A.");
      }

      double[,] resultData = new double[n, m];
      Task[] tasks = new Task[n];

      for (int i = 0; i < n; i++)
      {
         double[] rowA = a.GetRow(i);

         BulkWorker worker = new(i, rowA, b, m, (index, computedRow) =>
         {
            for (int j = 0; j < m; j++)
            {
               resultData[index, j] = computedRow[j];
            }
         });

         tasks[i] = Task.Factory.StartNew(worker.Execute,
TaskCreationOptions.LongRunning);
      }

      Task.WaitAll(tasks);
      return new Matrix(resultData);
   }
```

```
}
```

Клас BulkWorker

```
using MatrixCompute.Core.Models;

namespace MatrixCompute.Core.Multipliers.ParallelBulk;

internal class BulkWorker
{
   private readonly int _workerIndex;
   private readonly double[] _row;
   private readonly Matrix _matrixB;
   private readonly int _iterations;
   private readonly Action<int, double[]> _onCompleted;

   internal BulkWorker(int workerIndex, double[] row, Matrix matrixB, int iterations,
Action<int, double[]> onCompleted)
   {
      _workerIndex = workerIndex;
      _row = row;
      _matrixB = matrixB;
      _iterations = iterations;
      _onCompleted = onCompleted;
   }

   internal void Execute()
   {
      double[] resultRow = new double[_iterations];
      int currentColIndex = _workerIndex;

      for (int iter = 0; iter < _iterations; iter++)
      {
         double dot = _row
            .Select((rowValue, idx) => rowValue * _matrixB[idx, currentColIndex])
            .Sum();

         resultRow[currentColIndex] = dot;

         currentColIndex = (currentColIndex - 1 + _iterations) % _iterations;
      }

      _onCompleted(_workerIndex, resultRow);
   }
}
```

Додаток Е. Вимірювання часу виконання

```
using System.Diagnostics;
using MatrixCompute.Core.Abstractions;
using Matrix = MatrixCompute.Core.Models.Matrix;

namespace MatrixCompute.Runner.Utils;

internal sealed class Benchmark
{
    private readonly Stopwatch _sw = new();

    internal void Run(IMultiplier multiplier, int dimension)
    {
        Matrix matrixA = Matrix.GenerateRandomMatrix(dimension, dimension);
        Matrix matrixB = Matrix.GenerateRandomMatrix(dimension, dimension);

        _sw.Restart();
        multiplier.Multiply(matrixA, matrixB);
        _sw.Stop();
        double duration = _sw.Elapsed.TotalMilliseconds;

        Console.WriteLine(duration);
    }
}
```

Додаток Ж. Скрипт для локального тестування

```
#!/bin/bash
set -e

echo "Building project in Release mode..."
dotnet build -c Release

multipliers=(
  "CLASSICAL_MULTIPLIER"
  "STRIPE_MULTIPLIER"
  "PARALLEL_STRIPE_MULTIPLIER"
  "PARALLEL_BULK_MULTIPLIER")
dimensions=(64 128 256 512 1024 1536 2048 2560 3072 3584)
runs=20

dll_path="./bin/Release/net9.0/MatrixCompute.Runner.dll"

for dimension in "${dimensions[@]}"; do
  for multiplier in "${multipliers[@]}"; do
    echo "> MULTIPLIER=${multiplier} and DIMENSION=${dimension}"

    sum=0
    for (( i=1; i<=runs; i++ )); do
      result=$(MULTIPLIER=${multiplier} DIMENSION=${dimension} dotnet ${dll_path})

      echo "Run ${i}: ${result} ms"

      sum=$(awk "BEGIN {print $sum + $result}")
    done

    average=$(awk "BEGIN {printf \"%.2f\", $sum / $runs}")
    echo "Average execution time: ${average} ms"
    echo ""
  done
done
```

Додаток К. Скрипт для контейнеризованого тестування

```
#!/bin/bash
set -e

echo "Building Docker image 'matrix-compute'..."
docker build -t matrix-compute ../../

multipliers=(
  "STRIPE_MULTIPLIER"
  "PARALLEL_STRIPE_MULTIPLIER")
cpu_limits=(1 2 4 6 8 10 12)
runs=20
dimension=2560

for cpus in "${cpu_limits[@]}"; do
  for multiplier in "${multipliers[@]}"; do
    echo "> MULTIPLIER=${multiplier} and CPU limit=${cpus} CPU(s)"

    sum=0
    for (( i=1; i<=runs; i++ )); do
      result=$(docker container run --rm --cpus=${cpus} \
        -e DIMENSION=${dimension} \
        -e MULTIPLIER=${multiplier} \
        matrix-compute)

      echo "Run ${i}: ${result} ms"
      sum=$(awk "BEGIN {print $sum + $result}")
    done

    average=$(awk "BEGIN {printf \"%.2f\", $sum / $runs}")
    echo "Average execution time: ${average} ms"
    echo ""
  done
done
```

Додаток Л. Інфраструктура Docker

Dockerfile

```
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src

COPY ["src/MatrixCompute.Runner/*.csproj", "src/MatrixCompute.Runner/"]
COPY ["src/MatrixCompute.Core/*.csproj", "src/MatrixCompute.Core/"]
COPY ["MatrixCompute.sln", "./"]

RUN dotnet restore

COPY . .

RUN dotnet publish "src/MatrixCompute.Runner/MatrixCompute.Runner.csproj" -c Release
-o /app

FROM mcr.microsoft.com/dotnet/runtime:9.0
WORKDIR /app
COPY --from=build /app .
ENTRYPOINT ["dotnet", "MatrixCompute.Runner.dll"]
```

compose.yaml

```
services:
  matrix-compute:
    build: .
    cpus: 6
    environment:
      - DIMENSION=1024
#     - MULTIPLIER=CLASSICAL_MULTIPLIER
      - MULTIPLIER=STRIPE_MULTIPLIER
#     - MULTIPLIER=PARALLEL_STRIPE_MULTIPLIER
#     - MULTIPLIER=PARALLEL_BULK_MULTIPLIER
#     - MULTIPLIER=MULTIPLIERS_VERIFICATION
#     - VERIFICATION_DIMENSION=512
```