

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 пошуку оптимальних значень методом градієнтного
спуску та його паралельна реалізація мовою програмування C++

Керівник:

проф. Стеценко Інна Вячеславівна

«Допущено до захисту»

«__» _____ 2024 р.

Захищено з оцінкою

Члени комісії:

Виконавець:

Шляхтун Денис Михайлович

студент групи ІП-14

залікова книжка № ІП-1431

«25» квітня 2024 р.

Інна СТЕЦЕНКО

Антон ДИФУЧИН

Київ – 2024

ЗАВДАННЯ

Основним завданням курсової роботи є паралельна реалізація алгоритму пошуку оптимальних значень методом градієнтного спуску, що забезпечує прискорення не менше 1,2 при достатньо великій складності обчислень.

1. Виконати огляд існуючих реалізацій алгоритму, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.

2. Виконати розробку послідовного алгоритму пошуку оптимальних значень методом градієнтного спуску мовою програмування C++. Опис алгоритму забезпечити у вигляді псевдокоду. Провести тестування алгоритму та зробити висновок про коректність розробленого алгоритму. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.

3. Виконати розробку паралельного алгоритму пошуку оптимальних значень методом градієнтного спуску мовою програмування C++. Опис алгоритму забезпечити у вигляді псевдокоду. Забезпечити ініціалізацію даних при будь-якому великому заданому параметрі кількості даних.

4. Виконати тестування алгоритму, що доводить коректність результатів обчислень. Тестування алгоритму обов'язково проводити на великій кількості даних. Коректність перевіряти порівнянням з результатами послідовного алгоритму.

5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.

6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення не менше 1,2.

7. Дослідити вплив параметрів паралельного алгоритму на отримуване прискорення. Один з таких параметрів – це кількість підзадач, на які поділена задача при розпаралелюванні її виконання.

8. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

АНОТАЦІЯ

У курсовій роботі здійснюється розробка паралельного алгоритму пошуку оптимальних значень методом градієнтного спуску мовою програмування C++.

Досліджено алгоритм, розроблено псевдокод, переглянуто різні відомі рішення по розробці паралельного алгоритму. За основу паралельності взято природу градієнтного спуску, що залежить від початкової точки пошуку.

Здійснено розробку градієнтного спуску та його модифікації з обмеженням пошуку значень по відрізу. Розроблено код мовою програмування C++. Для реалізації багатопоточності було використано бібліотеку oneTBB від Intel.

Досягнуто прискорення в 3.72 рази при використанні 12 потоків процесору Intel® Core™ i7-10750H. Також здійснено порівняння використання різної кількості потоків.

Ключові слова: паралельний алгоритм, багатопоточна технологія, Gradient Descent, oneTBB.

ЗМІСТ

ВСТУП.....	7
1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ.....	8
1.1 Математичні основи градієнтного спуску	8
1.2 Послідовний алгоритм градієнтного спуску	9
1.3 Відомі паралельні реалізації алгоритму	9
2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ	10
2.1 Проєктування послідовного алгоритму	10
2.2 Реалізація послідовного алгоритму.....	11
2.3 Тестування та аналіз швидкодії.....	13
3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС	18
4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ.....	20
4.1 Проєктування паралельного алгоритму	20
4.2 Реалізація паралельного алгоритму	20
4.3 Тестування та аналіз швидкодії.....	21
5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ	25
ВИСНОВКИ	28
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	29
ДОДАТКИ.....	31

Додаток А. Лістинг коду реалізації градієнтного спуску.....	31
Додаток Б. Лістинг коду реалізації градієнтного спуску з обмеженням відрізками.....	33
Додаток В. Лістинг коду прикладу тестування алгоритму	36
Додаток Г. Лістинг коду додаткових засобів, використаних для реалізації алгоритму	40

ВСТУП

Паралельні обчислення набувають усе більше актуальності в сучасному світі. Цей тренд обумовлюється низкою чинників. Сучасні комп'ютери оснащені процесорами, що мають два й більше ядер, а багатопоточні алгоритми можуть використовувати весь потенціал обчислювальної потужності, водночас скоротивши час виконання. Також, експоненційно зростає кількість даних та виникає потреба їх обробки. Багатопоточність може значно прискорити обробку великих масивів даних, що робить її незамінною для таких задач, як штучний інтелект, машинне навчання та наукові моделювання. У паралельних обчисленнях є і низка обмежень. Не всі задачі піддаються ефективному паралельному виконанню. Розробка та програмування паралельних програм може бути складнішим завданням, ніж програмування послідовних програм[1].

Гرادієнтний спуск – це поширений алгоритм оптимізації, який використовується для навчання машинних моделей. Він працює шляхом ітеративного регулювання параметрів моделі для мінімізації функції втрат, що робить його корисним для задач машинного навчання та глибокого навчання. Але сфера застосування ГС не обмежується лише цими галузями, його також використовують в інженерії, робототехніці, іграх та інших сферах[2].

У рамках курсової роботи розроблено послідовну та паралельну реалізацію алгоритму пошуку оптимальних значень методом градієнтного спуску мовою програмування C++.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

1.1 Математичні основи градієнтного спуску

Градiєнтний спуск – це iтеративний алгоритм оптимiзацiї, що використовується для пошуку локального мiнiмуму або максимуму функцiї. Метод зазвичай використовується в машинному навчаннi та глибокому навчаннi для мiнiмiзацiї функцiї втрат[2].

Градiєнтний спуск можна застосувати лише на функцiях, якi є диференцiйовними, адже для обрахунку градiєнту потрібна похiдна функцiї. Також функцiя повинна бути опуклою, щоб не було залежностi вiд початкової точки, адже при наявностi понад одного локального мiнiмуму алгоритм не обов'язково буде знаходити глобальний мiнiмум.

Градiєнт у випадку багатовимiрної функцiї – це вектор часткових похiдних по кожнiй зi змiнних. Градiєнт для n -вимiрної функцiї $f(x)$ на заданiй точцi p рахується вiдповiдно до формули 1.1[2].

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix} \quad (1.1)$$

Алгоритм градiєнтного спуску iтеративно обчислює наступну точку. Вiд значення поточної точки вiднiмається градiєнт на поточнiй позицiї, помножений на швидкiсть навчаннi. Цей процес записаний у формулi 1.2[2].

$$p_{n+1} = p_n - \eta \nabla f(p_n) \quad (1.2)$$

Параметр η , швидкiсть навчаннi, регулює розмiр кроку. Чим менший цей параметр, тим довше сходиться градiєнтний спуск, можуть пройти всi iтерацiї до досягнення оптимальної точки. Чим бiльший параметр швидкостi навчаннi, тим бiльший шанс, що алгоритм не досягне оптимальної точки, «перестрибне її», або взагалi розiйдеться[2].

1.2 Послідовний алгоритм градієнтного спуску

На основі математичних основ можна скласти псевдокод алгоритму. Вхідними параметрами є функція обчислення градієнту `gradient`, початкова точка `startPosition`, швидкість навчання `learnRate`, кількість ітерацій алгоритму `nIterations`.

```
func gradientDescent(gradient, startPosition, learnRate, nIterations):
    currentPosition = startPosition
    repeat for nIterations:
        currentPosition = currentPosition – learnRate * gradient(currentPosition)
    return currentPosition
```

1.3 Відомі паралельні реалізації алгоритму

Поширені паралельні реалізації алгоритму базуються на основі стохастичного градієнтного спуску.

Реалізація Hogwild! є методом паралельного стохастичного градієнтного спуску, який пропонує необмежений доступ до пам'яті, тобто оновлення параметрів моделі відбувається асинхронно. Різні потоки можуть одночасно оновлювати параметри, не чекаючи один на одного[3].

Downpour SGD запускає кілька копій моделі паралельно на підмножинах тренувальних даних. Ці моделі надсилають свої оновлення на сервер параметрів, який розташований на кількох машинах. Кожна машина відповідає за зберігання та оновлення частини параметрів моделі. Однак, оскільки копії моделі не спілкуються одна з одною, наприклад, не діляться вагами чи оновленнями, їх параметри постійно знаходяться під загрозою розходження, що ускладнює збіжність[4].

EASGD зв'язує параметри робочих моделей асинхронного SGD з еластичною силою, тобто центральною змінною, яка зберігається сервером параметрів. Це дає змогу локальним змінним відхилятися далі від центральної змінної, що в теорії дозволяє більше дослідження простору параметрів. Вони емпірично показують, що цей збільшений обсяг дослідження призводить до покращення продуктивності, знаходячи нові локальні оптимуми[5].

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Проектування послідовного алгоритму

За основу можна взяти псевдокод із пункту 1.2. Також можливо доповнити його параметром `tolerance`, який задає мінімально можливе переміщення за ітерацію, `abs()` у контексті псевдокоду значить модуль числа.

```
func gradientDescentPoint(gradient, startPosition, learnRate, nIterations, tolerance):
```

```
    currentPosition = startPosition
```

```
    repeat for nIterations:
```

```
        diff = - learnRate * gradient(currentPosition)
```

```
        if abs(diff) < tolerance:
```

```
            break repeat
```

```
        currentPosition = currentPosition + diff
```

```
    return currentPosition
```

Такий псевдокод алгоритму дає змогу шукати градієнт по заданій точці, для уникнення знаходження лише локальних мінімумів при виконанні роботи будуть надаватися масиви випадкових точок фіксованого розміру для послідовного ітеративного пошуку оптимуму за допомогою градієнтного спуску.

Альтернативний спосіб пошуку градієнтного спуску – додавання параметру відрізка, на якому шукається оптимальний параметр. Нехай лівий край відрізка – `intervalStart`, а правий край – `intervalEnd`.

```

func gradientDescentInterval(gradient, startPosition, learnRate, nIterations,
tolerance, intervalStart, intervalEnd):
    currentPosition = startPosition
    repeat for nIterations:
        diff = - learnRate * gradient(currentPosition)
        if abs(diff) < tolerance:
            break repeat
        if currentPosition + diff < intervalStart
            currentPosition = intervalStart
            break repeat
        else if currentPosition + diff > intervalEnd
            currentPosition = intervalEnd
            break repeat
        currentPosition = currentPosition + diff
    return currentPosition

```

Для уникнення знаходження лише локального мінімуму будуть ітеративно перевірятися багато відрізків підряд.

2.2 Реалізація послідовного алгоритму

Повна програмна реалізація розміщена на GitHub за посиланням: <https://github.com/Denys-Shliakhtun/ConcurrentGradientDescent>.

Для реалізації та тестування алгоритму було здійснено розробку допоміжних структур та функцій для спрощення написання коду (додаток Г).

Функції `sum` та `average` здійснюють обрахунок відповідно суми та середнього значення вектору чисел заданого типу.

Структура `Point` – точка, що містить вектор координат та при наявності покажчик на значення градієнту, якщо ця точка є результатом алгоритму градієнтного спуску. Структура містить конструктори за замовчуванням, копіювання та за вектором координат. Також структура містить методи знаходження суми координат двох точок, знаходження середини між двома

точками та перевантаження оператора додавання для знаходження суми координат двох точок.

Структура `Interval` – відрізок, що містить точку початку та точку кінця. Структура містить конструктори за замовчуванням та за точками початку та кінця. Структура містить метод знаходження середини відрізка.

Функція `generateInitialPoints` генерує вектори випадкових точок у межах заданого відрізка. Для функції можна задати кількість векторів та кількість точок у кожному векторі.

Функція `testResult` перевіряє, чи містить вектор точок результуючу точку із заданою точністю `tolerance`. Результат градієнтного спуску не завжди повністю співпадає з очікуваним результатом залежно від його значення `tolerance` і загалом порівняння чисел з плаваючою крапкою може бути неправильним, якщо здійснюється перевірка рівності.

Реалізація звичайного алгоритму наведена в додатку А. Розглянемо функцію `gradientDescentPoint`. Загалом вона є прямою інтерпретацією псевдокоду, внутрішній цикл відповідає за перебір кожної координати точки і `toleranceCount` перевіряє, чи по всіх координатах значення кроку занадто мале для продовження.

Функція `linearGradientDescentPoint` ітеративно по розміру вектору наданих точок шукає результат алгоритму для кожної початкової точки, додає їх у єдиний вектор `results` та повертає його.

Реалізація алгоритму із застосуванням відрізків наведена у додатку Б. Розглянемо функцію `gradientDescentInterval`. На додачу до `toleranceCount` додався `outOfRangeCount`, який слідкує, щоб при досягненні межі відрізка алгоритм проходив ще одну ітерацію, а якщо на наступну ітерацію знову здійснився вихід із межі, то алгоритм закінчує роботу.

Функція `linearGradientDescentInterval` на додачу до ітерування кожного відрізка загалом формує вектор `intervals` заданої кількості рівних відрізків шляхом розбивання початкового відрізка функцією `breakIntervalIntoEqualParts` та

кожен відрізок з `intervals` проходить алгоритм, початковою точкою вважається середина відрізка.

2.3 Тестування та аналіз швидкодії

Для тестування задамо функції та знайдемо їхні мінімуми. Нехай перша функція – функція 2.1.

$$f(x) = (x - 1)^2 + \cos(x^2) \quad (2.1)$$

Графік функції зображено на рисунку 2.1 за допомогою GeoGebra.

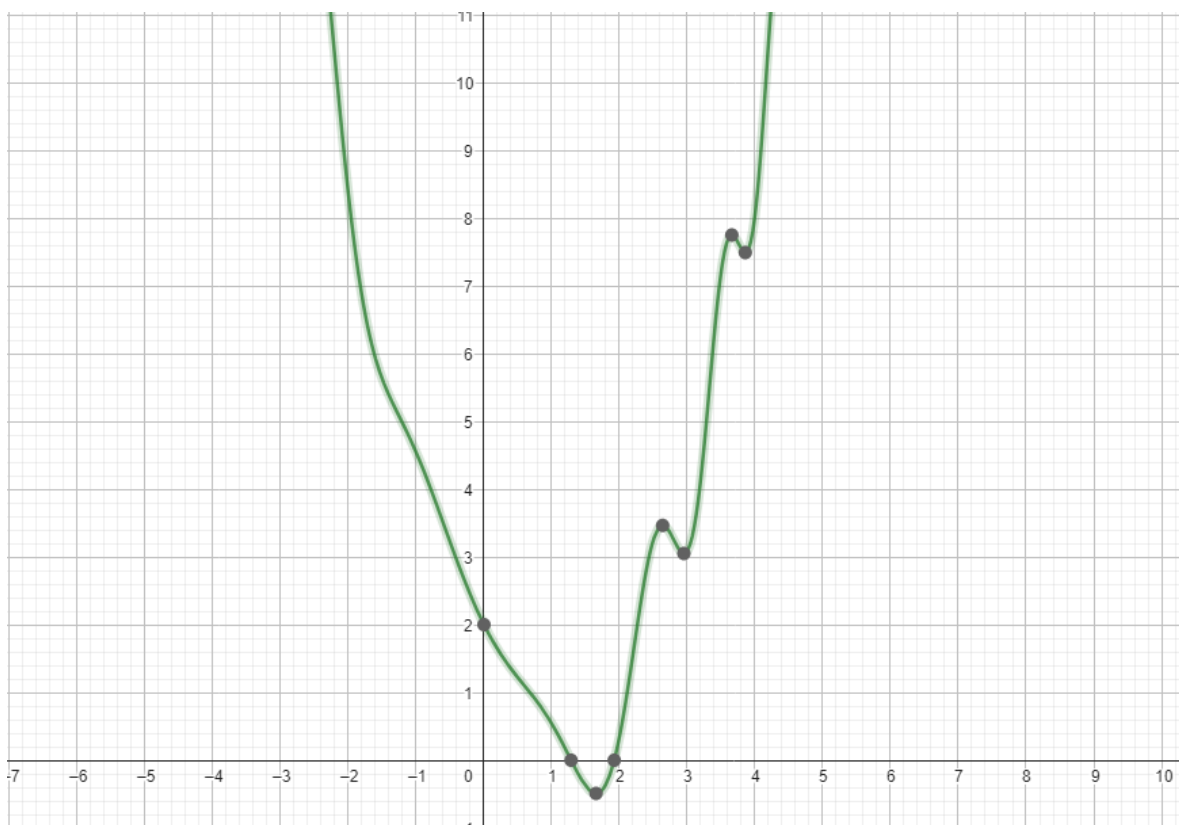


Рисунок 2.1 – Графік функції 2.1

Глобальним мінімумом функції є -0.49 у точці $x = 1.65383$.

При тестуванні буде виводитися не лише позиція мінімуму, а і досягнуте значення градієнту, адже нульове значення градієнту означає, що точка знаходиться в точці екстремуму.

Функція обчислення градієнту – `f2d`, наведена у додатку В.

Проведемо тестування функції `linearGradientDescentPoint` з 10 випадковими точками в інтервалі $[-10; 20]$ (рисунок 2.2).

```

F:\university\VI_semester\mul x + -
Initial pos:  -6.000  12.000  7.000  17.000  13.000  15.000  -1.000  10.000  13.000  -1.000
Test passed!

All points:    1.654  11.841  6.790  16.872  12.858  14.895  1.654  9.812  12.858  1.654
Gradient:     -0.000  0.000  0.000  0.000  0.000  0.000  -0.000  0.000  0.000  -0.000
Press any key to continue . . . |

```

Рисунок 2.2 – Результат тестування функції
linearGradientDescentPoint

Значення градієнту для всіх точок 0 і було знайдено хоча б один результат, що співпадає з глобальним мінімумом, тому тестування можна вважати успішним.

Проведемо тестування функції linearGradientDescentInterval з 10 інтервалами у межах відрізка [-10; 20] (рисунок 2.3).

```

F:\university\VI_semester\mul x + -
Test passed!

All points:    -7.000  -4.000  -1.000  1.654  2.950  6.312  9.487  12.360  15.514  18.471
Gradient:     -29.353 -12.303  -2.317  -0.000  0.000  0.000  0.000  0.000  -0.000  0.000
Press any key to continue . . . |

```

Рисунок 2.3 – Результат тестування функції
linearGradientDescentInterval

Значення градієнту є 0 не для всіх точок, бо не у всіх інтервалах є локальний мінімум, а за межі свого інтервалу точки не виходять. У результаті тестування знайдено і локальні, і глобальні мінімуми, тому тестування можна вважати успішним.

Аналогічно до попереднього тестування проведемо тестування для функції двох змінних (функція 2.2).

$$f(x) = (x_1 - 1)^2 + \cos(x_1^2) + (x_2 + 1)^2 + \cos(x_2^2) \quad (2.2)$$

Графік функції зображено на рисунку 2.1 за допомогою GeoGebra.

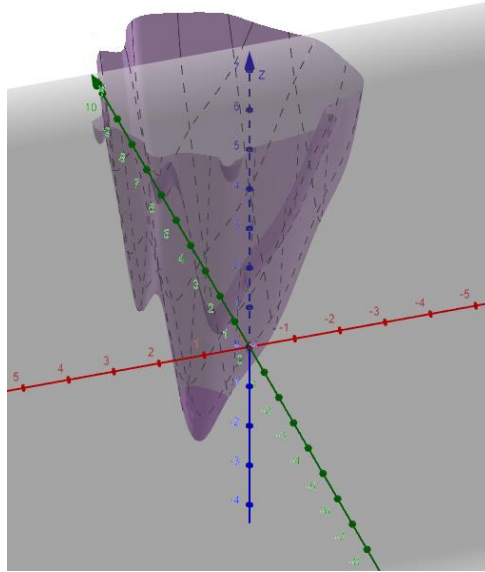


Рисунок 2.4 – Графік функції

Глобальним мінімумом функції є $(1.65383, -1.65383)$.

Функція обчислення градієнту – `f3d`, наведена у додатку В.

Проведемо тестування функції `linearGradientDescentPoint` з 10 випадковими точками в інтервалі $[(-10, -20); (20, 10)]$ (рисунок 2.5).

```

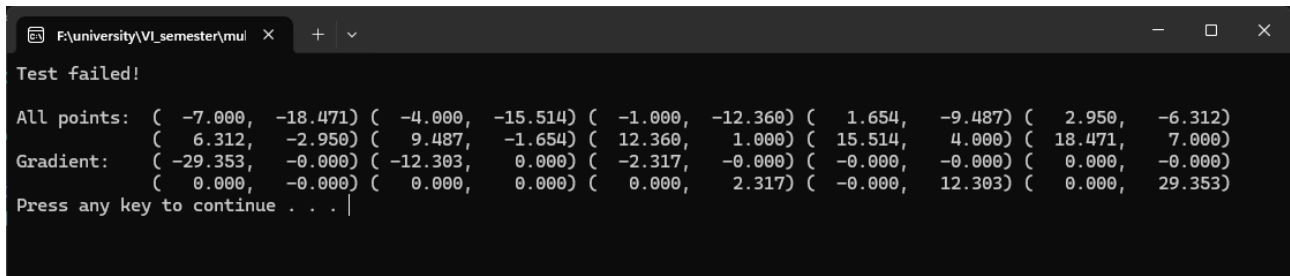
F:\university\VI_semester\mul x + v - □ x
Initial pos: ( -6.000, 2.000) ( 7.000, 7.000) ( 13.000, 5.000) ( -1.000, 0.000) ( 13.000, -11.000)
             ( -1.000, -1.000) ( 5.000, -6.000) ( 18.000, -14.000) ( -2.000, -6.000) ( -9.000, 3.000)
Test passed!

All points: ( 1.654, -1.654) ( 6.790, -1.654) ( 12.858, -1.654) ( 1.654, -1.654) ( 12.858, -11.017)
            ( 1.654, -1.654) ( 4.593, -5.795) ( 17.954, -14.026) ( 1.654, -5.795) ( 1.654, -1.654)
Gradient: ( -0.000, 0.000) ( 0.000, 0.000) ( 0.000, 0.000) ( -0.000, 0.000) ( 0.000, 0.000)
           ( -0.000, 0.000) ( 0.000, -0.000) ( -0.000, 0.000) ( -0.000, -0.000) ( -0.000, 0.000)
Press any key to continue . . . |
  
```

Рисунок 2.5 – Результат тестування функції

`linearGradientDescentPoint`

Проведемо тестування функції `linearGradientDescentInterval` з 10 інтервалами у межах відрізка $[(-10, -20); (20, 10)]$ (рисунок 2.6).



```

F:\university\VI_semester\mul x + -
Test failed!
All points: ( -7.000, -18.471) ( -4.000, -15.514) ( -1.000, -12.360) ( 1.654, -9.487) ( 2.950, -6.312)
            ( 6.312, -2.950) ( 9.487, -1.654) ( 12.360, 1.000) ( 15.514, 4.000) ( 18.471, 7.000)
Gradient:   ( -29.353, -0.000) ( -12.303, 0.000) ( -2.317, -0.000) ( -0.000, -0.000) ( 0.000, -0.000)
            ( 0.000, -0.000) ( 0.000, 0.000) ( 0.000, 2.317) ( -0.000, 12.303) ( 0.000, 29.353)
Press any key to continue . . . |

```

Рисунок 2.6 – Результат тестування функції
linearGradientDescentInterval

Тестування провалилося, адже спосіб обрахування відрізків не враховує природу поведінки при більшій кількості вимірів, коли по факту кількість зон, по яким виконується градієнтний спуск, повинна зростати експоненційно відносно вхідного параметру кількості, щоб займати повністю вказаний простір, тобто у даному випадку потрібно формувати 100 таких зон. Надалі в роботі алгоритм з використанням відрізків буде застосовуватися лише для одновимірних функцій.

Тестування часу виконання здійснювалося процесором Intel® Core™ i7-10750H, що має 6 ядер та 12 потоків[11], з обмеженням Turbo Boost у 3.5 GHz для запобігання перегріванню при виконанні алгоритму. Було здійснено виміри часу виконання функції linearGradientDescentPoint для однієї та двох змінних, linearGradientDescentInterval. Перед вимірюванням виконувалося 20 запусків алгоритму, потім вираховувався середній час за наступні 20 запусків. Були здійснені заміри різної кількості ітерацій алгоритму.

Результати заміру часу виконання функцій зображено на рисунку 2.7.

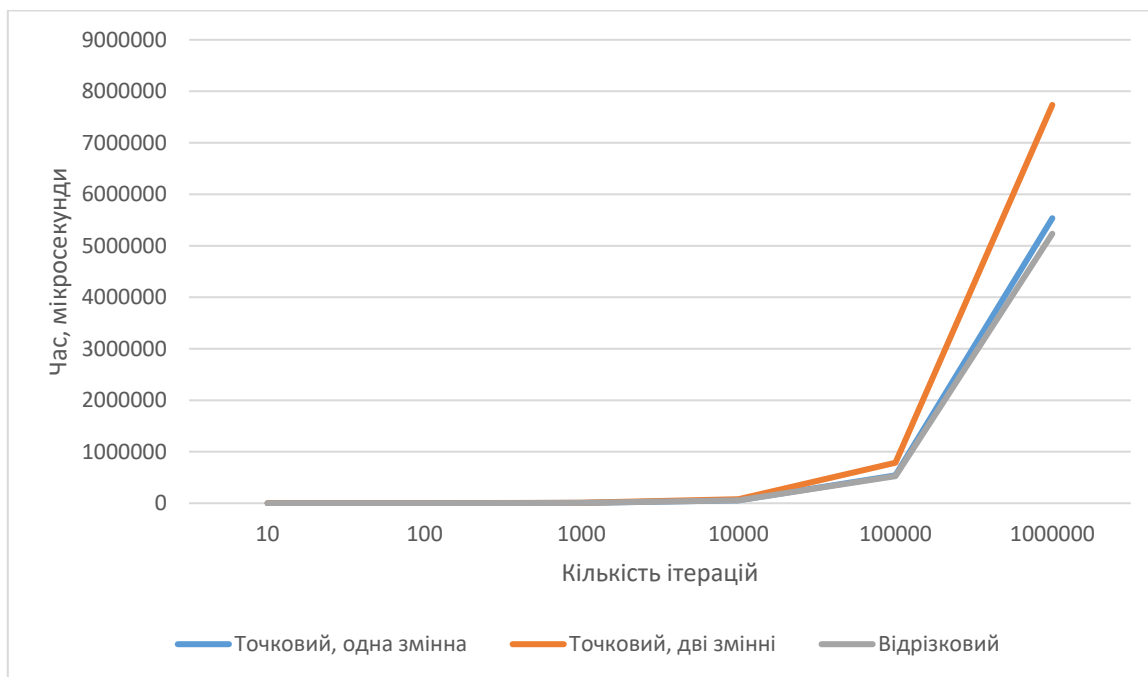


Рисунок 2.7 – Результати замірів часу виконання лінійного алгоритму

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

Існують різні варіації бібліотек, які дозволяють використання пулу потоків, як окремі бібліотеки від окремих розробників, так і складові великих проєктів. Boost – відомий проєкт, що містить різного роду бібліотеки, зокрема простий пул потоків[6]. QThreadPool – частина фреймворку Qt[7].

Оскільки у відкритих джерелах не було наведено різницю у швидкодії різних пулів потоків, то було обрано проєкт на основі додаткових можливостей. Отже, для розробки паралельного алгоритму було обрано бібліотеку Intel® oneAPI Threading Building Blocks (oneTBB)[8], що спрощує роботу з додавання паралельності до складних програм. oneTBB є частиною інструментів oneAPI, що зокрема містять інструменти для машинного та глибокого навчання. Основною причиною вибору саме цієї бібліотеки є потенціальна можливість розвинути алгоритм градієнтного спуску для навчання нейронної мережі. Бібліотека орієнтується на масштабованому програмуванні на великих обсягах даних.

Відповідно до документації[9] бібліотека oneTBB містить набір загальних паралельних алгоритмів, наприклад сортування та паралельний цикл; граф потоку, що дозволяє здійснювати графовий паралелізм; контейнери, що можуть використовуватися декількома потоками, наприклад вектор і черга; локальні сховища потоків; планувальник завдань, призначений для інтенсивних паралельних обчислень.

Для виконання роботи обрані класи `task_arena` та `task_group`, що є частиною планувальника завдань[10].

`task_arena` – аналог пулу потоків, що дозволяє керувати набором завдань та запускати їх паралельно.

`task_group` – це клас, який використовується для створення та управління групами завдань. Група завдань - це колекція завдань, які виконуються

одночасно. `task_group` дає змогу синхронізувати та координувати виконання завдань, а також обробляти результати їх виконання.

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

4.1 Проєктування паралельного алгоритму

У послідовному алгоритмі подається на вхід масив початкових точок або формується масив відрізків, по яким ітеративно виконується алгоритм градієнтного спуску, тому ідея паралельного виконання полягає у одночасному пошуку оптимумів за різними початковими точками та відрізками замість ітеративного проходження. Задачі градієнтного спуску є незалежними, адже не використовують спільних даних, тому і відсутня необхідність у синхронізації. Однак варто зазначити, що фактично паралельність та наявність масиву вхідних параметрів застосовується не просто для пошуку оптимуму, а для уникнення того, що знайдеться локальний мінімум.

Функції `gradientDescentPoint` та `gradientDescentInterval`, псевдокод яких описаний у розділі 2.1, повністю відповідають паралельному алгоритму, змінюється лише зовнішня функція перебору усіх заданих точок чи відрізків.

4.2 Реалізація паралельного алгоритму

Реалізація звичайного алгоритму наведена у додатку А. Для спрощення вигляду та розуміння було написано структуру точки, реалізація наведена у додатку Г.

Функція `concurrentGradientDescentPoint` містить об'єкти класів `task_arena` та `task_group`, описані у 3 розділі. Для об'єкту `task_arena`, тобто для пулу потоків задається максимальна кількість потоків. На відміну від послідовного алгоритму, замість ітеративного обрахунку градієнтного спуску здійснюється додавання задач у пул потоків шляхом їх запуску об'єктом `task_group`. Після додавання задач здійснюється очікування виконання усіх задач об'єктом `task_group`.

Реалізація алгоритму із застосуванням відрізків наведена у додатку Б. Для спрощення вигляду та розуміння було написано структуру відрізка, реалізація наведена у додатку Г.

Аналогічно, на відміну від послідовного алгоритму, функція `concurrentGradientDescentInterval` додає задачі виконання функції `gradientDescentInterval` для кожного відрізка до `task_arena`, а не просто ітеративно запускає алгоритм градієнтного спуску, потім очікує виконання задач перед поверненням результату.

Загалом кількість підзадач для обох реалізацій алгоритму, що можуть виконуватися паралельно, зводиться до кількості початкових точок або відрізків.

4.3 Тестування та аналіз швидкодії

Проведемо тестування функції `concurrentGradientDescentPoint` з 10 випадковими точками в інтервалі $[-10; 20]$ з функцією 2.1 (рисунок 4.1).

```

F:\university\VI_semester\mul x + -
Initial pos: -6.000 12.000 7.000 17.000 13.000 15.000 -1.000 10.000 13.000 -1.000
Test passed!
All points: 1.654 11.841 6.790 16.872 12.858 14.895 1.654 9.812 12.858 1.654
Gradient: -0.000 0.000 0.000 0.000 0.000 0.000 -0.000 0.000 0.000 -0.000
Press any key to continue . . . |

```

Рисунок 4.1 – Результат тестування функції
`concurrentGradientDescentPoint`

Проведемо тестування функції `concurrentGradientDescentInterval` з 10 інтервалами у межах відрізка $[-10; 20]$ з функцією 2.1 (рисунок 4.2).

```

F:\university\VI_semester\mul x + -
Test passed!
All points: -7.000 -4.000 -1.000 1.654 2.950 6.312 9.487 12.360 15.514 18.471
Gradient: -29.353 -12.303 -2.317 -0.000 0.000 0.000 0.000 0.000 -0.000 0.000
Press any key to continue . . . |

```

Рисунок 4.2 – Результат тестування функції
`concurrentGradientDescentInterval`

Проведемо тестування функції `concurrentGradientDescentPoint` з 10 випадковими точками в інтервалі $[(-10, -20); (20, 10)]$ з функцією 2.2 (рисунок 4.3).

```

F:\university\VI_semester\mul x + - □ ×
Initial pos: ( -6.000, 2.000) ( 7.000, 7.000) ( 13.000, 5.000) ( -1.000, 0.000) ( 13.000, -11.000)
              ( -1.000, -1.000) ( 5.000, -6.000) ( 18.000, -14.000) ( -2.000, -6.000) ( -9.000, 3.000)
Test passed!
All points: ( 1.654, -1.654) ( 6.790, -1.654) ( 12.858, -1.654) ( 1.654, -1.654) ( 12.858, -11.017)
             ( 1.654, -1.654) ( 4.593, -5.795) ( 17.954, -14.026) ( 1.654, -5.795) ( 1.654, -1.654)
Gradient: ( -0.000, 0.000) ( 0.000, 0.000) ( 0.000, 0.000) ( -0.000, 0.000) ( 0.000, 0.000)
           ( -0.000, 0.000) ( 0.000, -0.000) ( -0.000, 0.000) ( -0.000, -0.000) ( -0.000, 0.000)
Press any key to continue . . . |

```

Рисунок 4.3 – Результат тестування функції
concurrentGradientDescentPoint

Результати співпадають із тестуванням послідовного алгоритму. Необхідності в тестуванні на більшому розмірі вхідних параметрів немає, адже задачі, що виконуються в різних потоках, не використовують спільних даних та не потребують синхронізації, тому й не може бути помилок, спричинених неправильною взаємодією між потоками.

Тестування швидкодії проводилося аналогічно до того, що описане в пункті 2.3. Зі змін можна виділити заміри часу для пулу потоків із максимальною багатопоточністю у 12, 9, 6 і 3 потоки. Тестування як для послідовного, так і для паралельного алгоритмів проводилося з 20 підзадачами, що не є кратним кількості потоків, але є більше за кількість потоків. Підзадачі мають на увазі кількість точок або відрізків, що є параметрами градієнтного спуску.

Для більш наочного відображення різниці між різною кількістю потоків було взято за 100% час 12 потоків і відносно нього порівняно заміри іншої кількості потоків. Наприклад, 200% означає, що алгоритм виконувався вдвічі довше, ніж на 12 потоках.

На рисунку 4.4 зображено різницю швидкодії при виконанні функції concurrentGradientDescentPoint з однією змінною.

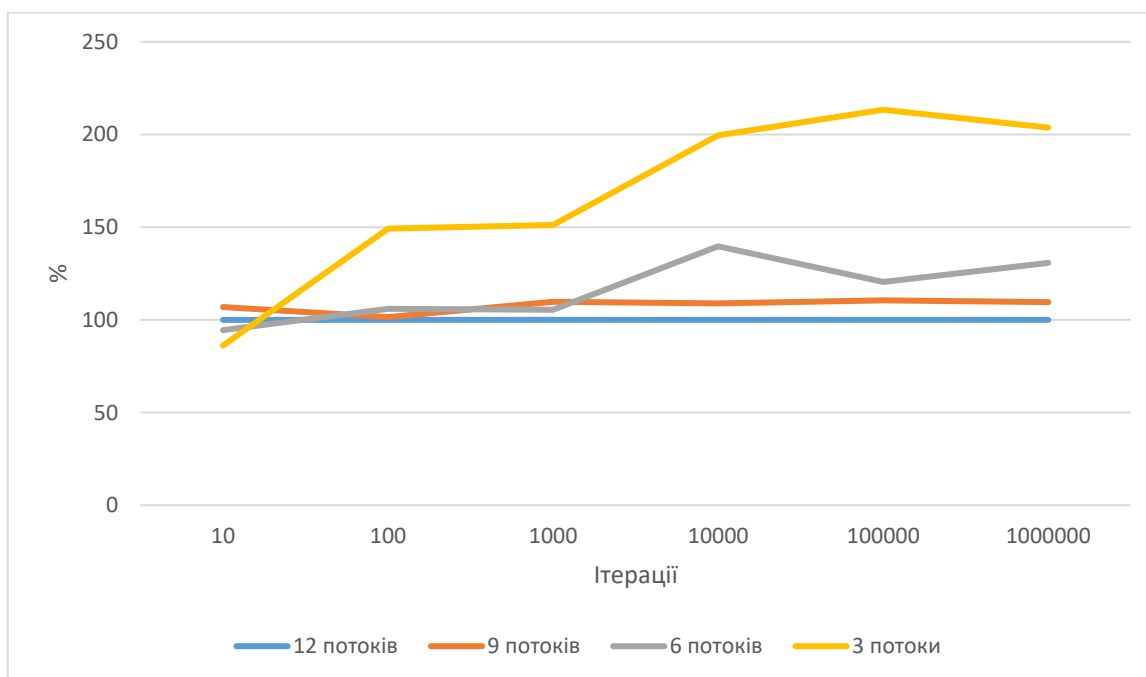


Рисунок 4.4 – Співвідношення часу виконання функції
`concurrentGradientDescentPoint` з однією змінною

На рисунку 4.5 зображено різницю швидкодії при виконанні функції `concurrentGradientDescentPoint` з двома змінними.

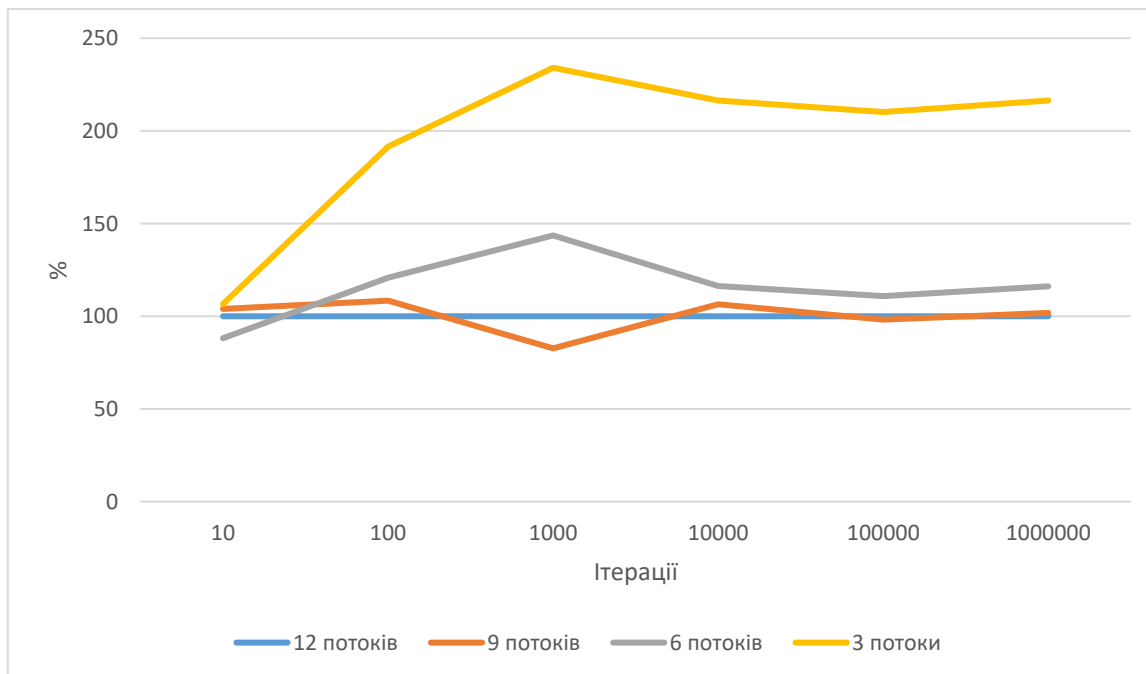


Рисунок 4.5 – Співвідношення часу виконання функції
`concurrentGradientDescentPoint` з двома змінними

На рисунку 4.6 зображено різницю швидкодії при виконанні функції `concurrentGradientDescentInterval`.

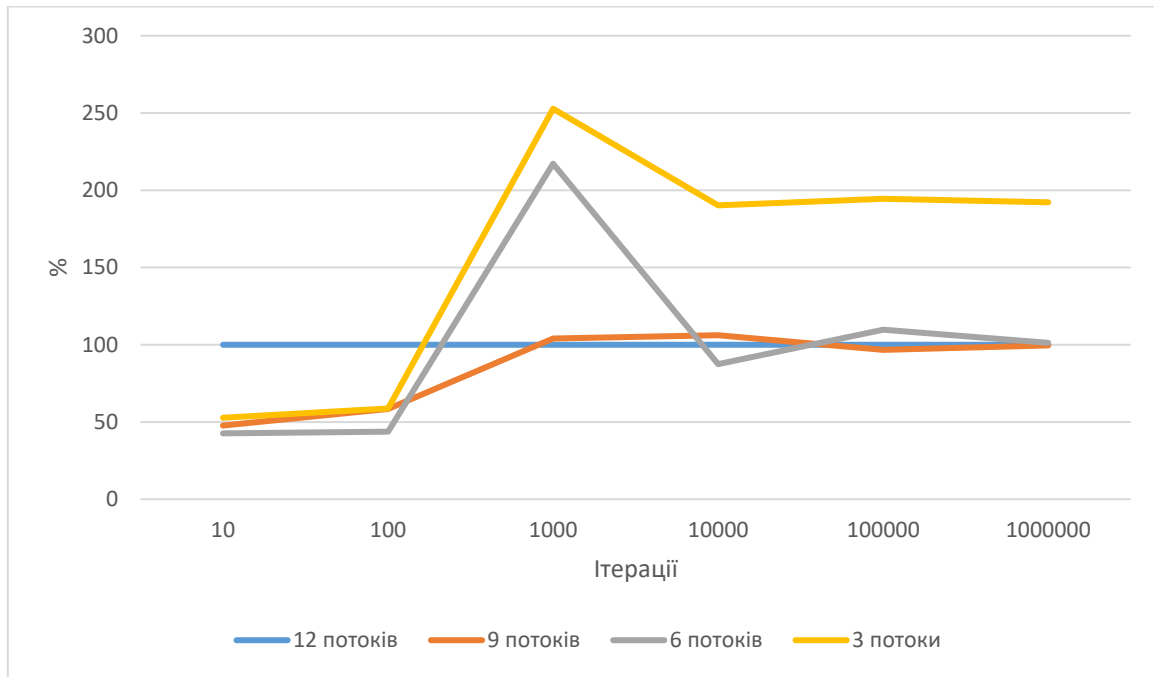


Рисунок 4.6 – Співвідношення часу виконання функції
concurrentGradientDescentInterval

З графіків можна зробити висновки, що збільшення кількості потоків не означає лінійне скорочення часу виконання, однією з причин може бути наявність лише 6 фізичних ядер процесора, максимальна кількість потоків у 12 досягається технологією Intel® Hyper-Threading[11].

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Підсумуємо отримані результати з пунктів 2.3 та 4.3.

У таблиці 5.1 представлені результати вимірів послідовного та паралельного алгоритму на 12 потоках із застосуванням функції однієї змінної.

Таблиця 5.1 – Результати вимірів пошуку оптимуму функції однієї змінної

Кількість ітерацій	Час послідовного алгоритму, мікросекунд	Час паралельного алгоритму, мікросекунд
10	65	72
100	585	268
1 000	8 029	2 471
10 000	56 747	16 343
100 000	541 316	156 210
1 000 000	5 535 720	1 551 640

У таблиці 5.2 представлені результати вимірів послідовного та паралельного алгоритму на 12 потоках із застосуванням функції двох змінних.

Таблиця 5.2 – Результати вимірів пошуку оптимуму функції двох змінних

Кількість елементів	Час послідовного алгоритму, мікросекунд	Час паралельного алгоритму, мікросекунд
10	102	76
100	864	308
1 000	9 016	2 798
10 000	78 020	20 914
100 000	788 233	216 559
1 000 000	7 735 065	2 070 970

У таблиці 5.3 представлені результати вимірів послідовного та паралельного алгоритму на 12 потоках із застосуванням обмеження по відрізьку.

Таблиця 5.3 – Результати вимірів пошуку оптимуму функції з обмеженням по відрізьку

Кількість елементів	Час послідовного алгоритму, мікросекунд	Час паралельного алгоритму, мікросекунд
10	111	218
100	670	628
1 000	5 709	1 652
10 000	55 038	16 042
100 000	524 790	153 220
1 000 000	5 234 085	1 511 490

На рисунку 5.1 відображено прискорення паралельного алгоритму відносно послідовного.

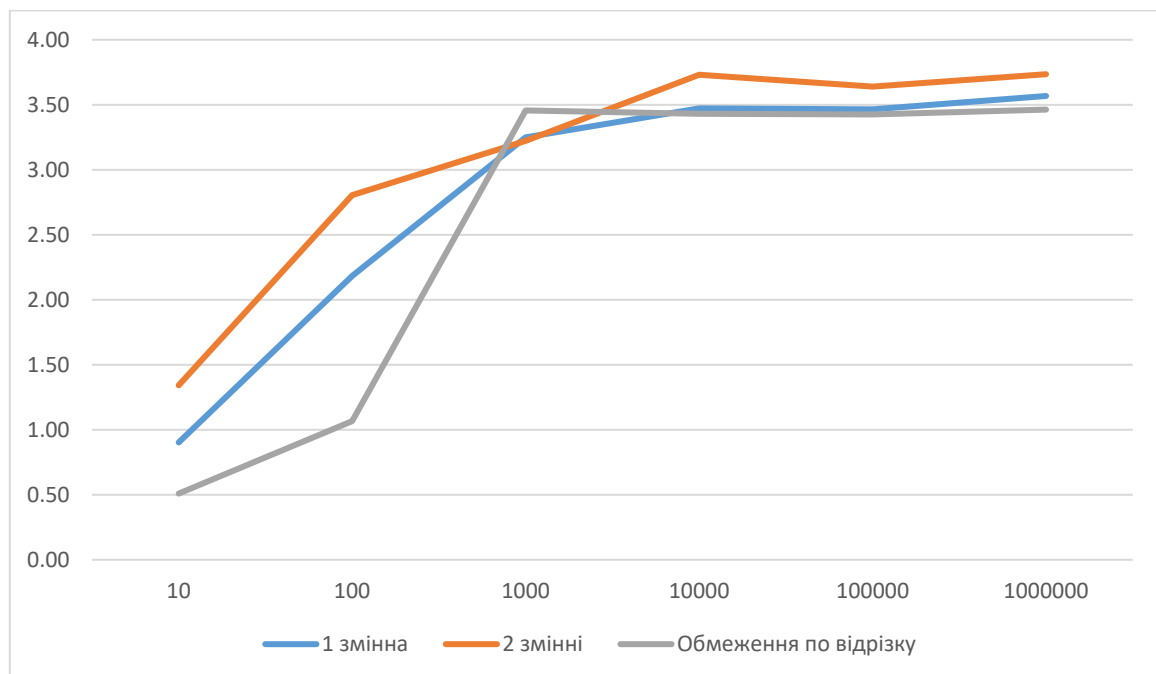


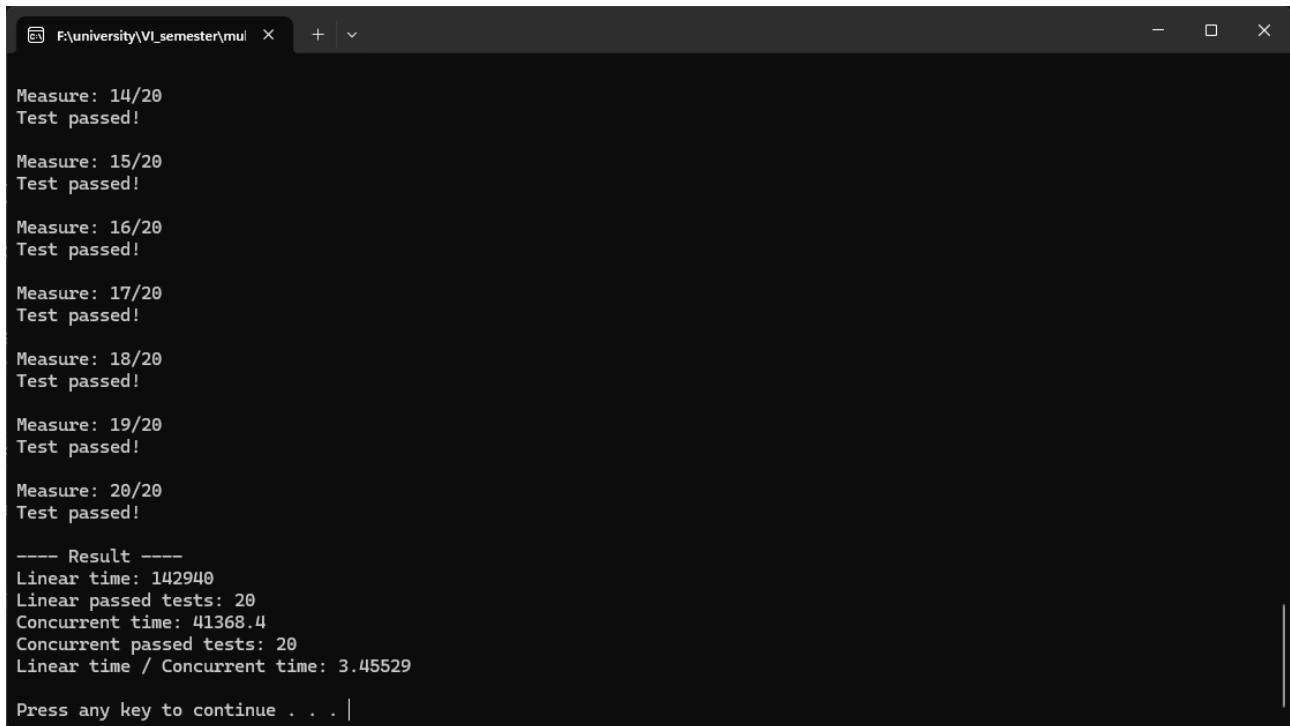
Рисунок 5.1 – Співвідношення часу виконання між послідовним та паралельним алгоритмом

Найбільше прискорення становить 3.73 для пошуку оптимального значення функції двох змінних. Загалом прискорення можна вважати

задовільним починаючи від 100 ітерацій алгоритму, що є доволі гарним показником.

Заміри проводилися відповідно до опису в пунктах 2.3 і 4.3. Код для проведення замірів наведено в додатку В.

Приклад замірів часу виконання алгоритму відображено на рисунку 5.2.



```
F:\university\VI_semester\mul x + v
Measure: 14/20
Test passed!

Measure: 15/20
Test passed!

Measure: 16/20
Test passed!

Measure: 17/20
Test passed!

Measure: 18/20
Test passed!

Measure: 19/20
Test passed!

Measure: 20/20
Test passed!

---- Result ----
Linear time: 142940
Linear passed tests: 20
Concurrent time: 41368.4
Concurrent passed tests: 20
Linear time / Concurrent time: 3.45529

Press any key to continue . . . |
```

Рисунок 5.2 – Результат виконання програми в консолі

ВИСНОВКИ

При виконанні роботи було досліджено, розроблено та реалізовано мовою програмування C++ алгоритм пошуку оптимальних значень методом градієнтного спуску та його паралельну версію.

Спочатку було досліджено алгоритм, переглянуто різні відомі рішення по розробці паралельного алгоритму. З'ясовано, що робота над алгоритмом є актуальною, адже він використовується для задач машинного навчання, глибокого навчання та у сферах інженерії, робототехніки, ігор та інших.

Потім було здійснено розробку послідовного алгоритму та його модифікації з обмеженням пошуку значень по відрізу, зокрема було розроблено псевдокод, код мовою програмування C++ та здійснено успішне тестування. Також було здійснено виміри швидкодії алгоритму.

Для реалізації багатопоточності було обрано бібліотеку oneTBB від Intel, що є частиною великого проєкту, що зокрема містить інструменти для машинного навчання.

На основі вибраної бібліотеки здійснено розробку паралельного алгоритму та досліджено його швидкодію в порівнянні з різною кількістю потоків та ітерацій алгоритму.

При тестуванні було досягнуто максимальне прискорення в 3.72 рази при використанні 12 потоків процесору Intel® Core™ i7-10750H в пошуку оптимуму функції двох змінних, що свідчить про виконання поставленого основного завдання.

Перевага від паралельних обчислень при виконанні алгоритму була відчутна в більшості випадків уже від 100 ітерацій алгоритму, тобто час паралельного алгоритму був менший за послідовний.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Nemirovsky M., Tullsen D. Multithreading Architecture : Springer Cham, 2013, 98 с.
2. Kwiatkowski R. Gradient Descent Algorithm — a deep dive. Towards Data Science, 2021. URL: <https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21> (дата звернення 07.04.2024)
3. Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent / Feng Niu, Benjamin Recht, Christopher Re, Stephen J. Wright. University of Wisconsin-Madison, 2011. 22 с.
4. Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems / Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V, Ng, A. Y. Google Inc, 2012. 9 с.
5. Zhang, S., Choromanska, A., LeCun, Y. Deep learning with Elastic Averaging SGD. Neural Information Processing Systems Conference, 2015. 24 с.
6. thread_pool / Boost. URL: https://www.boost.org/doc/libs/1_84_0/doc/html/boost_asio/reference/thread_pool.html (дата звернення 09.04.2024)
7. QThreadPool / Qt Documentation. URL: <https://doc.qt.io/qt-5/qthreadpool.html> (дата звернення 09.04.2024)
8. Intel® oneAPI Threading Building Blocks / Intel®. URL: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html> (дата звернення 09.04.2024)
9. oneTBB Interfaces / oneAPI Specification. URL: <https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-interfaces.html> (дата звернення 18.04)
10. Task Scheduler / oneAPI Specification. URL: https://spec.oneapi.io/versions/latest/elements/oneTBB/source/task_scheduler.html (дата звернення 09.04.2024)

11. Intel® Core™ i7-10750H Processor / Intel®. URL:
<https://ark.intel.com/content/www/us/en/ark/products/201837/intel-core-i7-10750h-processor-12m-cache-up-to-5-00-ghz.html> (дата звернення
13.04.2024)

ДОДАТКИ

Додаток А. Лістинг коду реалізації градієнтного спуску

```

#include "PointGradientDescent.h"
#include <oneapi/tbb/task_arena.h>
#include <oneapi/tbb/task_group.h>

std::vector<Point> linearGradientDescentPoint(
    Point(*derivative)(Point),
    std::vector<Point>& start_points,
    long double learn_rate,
    int n_iter,
    long double tolerance
)
{
    std::vector<Point> results(start_points.size());
    for (int i = 0; i < start_points.size(); i++)
        results[i] = gradientDescentPoint(derivative, start_points[i],
    learn_rate, n_iter, tolerance);
    return results;
}

std::vector<Point> concurrentGradientDescentPoint(
    int numOfThreads,
    Point(*derivative)(Point),
    std::vector<Point>& start_points,
    long double learn_rate,
    int n_iter,
    long double tolerance
)
{
    tbb::task_group task_group;

    tbb::task_arena::constraints constraints{};
    constraints.set_max_concurrency(numOfThreads);
    tbb::task_arena arena(constraints);

    std::vector<Point> results(start_points.size());

    for (int i = 0; i < start_points.size(); i++)
    {
        arena.execute([&, i] {
            task_group.run([&, i] {
                results[i] = gradientDescentPoint(derivative,
start_points[i], learn_rate, n_iter, tolerance);
            });
        });

    }

    task_group.wait();
    return results;
}

Point gradientDescentPoint(
    Point(*derivative)(Point),
    Point start,
    long double learn_rate,
    int n_iter,
    long double tolerance
)
{
    Point point = start;

```

```
for (int i = 0; i < n_iter; i++)
{
    if (point.grad != nullptr)
        delete point.grad;
    point.grad = new Point(derivative(point));
    int toleranceCount = 0;

    for (int i = 0; i < point.pos.size(); i++)
    {
        long double diff = -1 * learn_rate * point.grad->pos[i];

        if (abs(diff) <= tolerance)
            toleranceCount++;

        point.pos[i] += diff;
    }
    if (toleranceCount == point.pos.size())
        break;
}
return point;
}
```


Додаток Б. Лістинг коду реалізації градієнтного спуску з обмеженням відрізками

```

#include "IntervalGradientDescent.h"
#include <oneapi/tbb/task_arena.h>
#include <oneapi/tbb/task_group.h>

std::vector<Interval> breakIntervalIntoEqualParts(Interval interval, int
num_of_intervals)
{
    std::vector<Interval> intervals;
    Point step;
    for (int i = 0; i < interval.start.pos.size(); i++)
        step.pos.push_back((interval.end.pos[i] - interval.start.pos[i]) /
num_of_intervals);

    intervals.push_back(Interval(interval.start, interval.start + step));
    for (int i = 1; i < num_of_intervals - 1; i++)
        intervals.push_back(Interval(intervals[i - 1].end, intervals[i - 1].end +
step));
    intervals.push_back(Interval(intervals[num_of_intervals - 2].end,
interval.end));

    return intervals;
}

std::vector<Point> linearGradientDescentInterval(
    Point(*derivative)(Point),
    Interval interval,
    int num_of_intervals,
    long double learn_rate,
    int n_iter,
    long double tolerance
)
{
    std::vector<Point> results(num_of_intervals);
    std::vector<Interval> intervals = breakIntervalIntoEqualParts(interval,
num_of_intervals);

    for (int i = 0; i < num_of_intervals; i++)
        results[i] = gradientDescentInterval(
            derivative, intervals[i],
            intervals[i].getHalfwayPoint(),
            learn_rate, n_iter, tolerance);

    return results;
}

std::vector<Point> concurrentGradientDescentInterval(
    int num_of_threads,
    Point(*derivative)(Point),
    Interval interval,
    int num_of_intervals,
    long double learn_rate,
    int n_iter,
    long double tolerance
)
{
    std::vector<Point> results(num_of_intervals);
    std::vector<Interval> intervals = breakIntervalIntoEqualParts(interval,
num_of_intervals);

```

```

tbb::task_group task_group;

tbb::task_arena::constraints constraints{};
constraints.set_max_concurrency(numOfThreads);
tbb::task_arena arena(constraints);

for (int i = 0; i < num_of_intervals; i++)
{
    arena.execute([&, i] {
        task_group.run([&, i] {
            results[i] = gradientDescentInterval(
                derivative, intervals[i],
                intervals[i].getHalfwayPoint(),
                learn_rate, n_iter, tolerance);
        });
    });
task_group.wait();

return results;
}

Point gradientDescentInterval(
    Point(*derivative)(Point),
    Interval interval,
    Point start,
    long double learn_rate,
    int n_iter,
    long double tolerance
)
{
    Point point = start;
    for (int i = 0; i < n_iter; i++)
    {
        if (point.grad)
            delete point.grad;
        point.grad = new Point(derivative(point));
        int toleranceCount = 0;
        int outOfRangeCount = 0;

        for (int i = 0; i < point.pos.size(); i++)
        {
            long double diff = -1 * learn_rate * point.grad->pos[i];

            if (abs(diff) <= tolerance)
                toleranceCount++;

            if (point.pos[i] + diff < interval.start.pos[i])
            {
                if (point.pos[i] < interval.start.pos[i])
                    outOfRangeCount++;
                else
                    point.pos[i] = interval.start.pos[i] - tolerance;
            }
            else if (point.pos[i] + diff > interval.end.pos[i])
            {
                if (point.pos[i] > interval.end.pos[i])
                    outOfRangeCount++;
                else
                    point.pos[i] = interval.end.pos[i] + tolerance;
            }
            else
                point.pos[i] += diff;
        }
    }
}

```

```
        if (toleranceCount == point.pos.size() || outOfRangeCount ==  
point.pos.size())  
            break;  
    }  
    return point;  
}
```

Додаток В. Лістинг коду тестування алгоритму

```

#include "PointGradientDescent.h"
#include "IntervalGradientDescent.h"
#include <iostream>
#include <iomanip>
#include <chrono>
#include "TargetFunctions.h"
using namespace std::chrono;

Point f2d(Point x) { // derivative of (x-1)^2+cos(x^2)
    return Point({ (2 * (x.pos[0] - 1) - 2 * x.pos[0] * sin(x.pos[0] * x.pos[0]))
});
}

Point f3d(Point x) { // derivative of (x1-1)^2+cos(x1^2)+(x2+1)^2+cos(x2^2)
    return Point({
        (2 * (x.pos[0] - 1) - 2 * x.pos[0] * sin(x.pos[0] * x.pos[0])),
        (2 * (x.pos[1] + 1) - 2 * x.pos[1] * sin(x.pos[1] * x.pos[1]))
    });
}

void testPointGradientDescent(
    int numOfThreads,
    Point(*grad)(Point),
    int N_ITER,
    long double LEARN_RATE,
    long double TOLERANCE,
    Interval interval,
    int numOfPoints,
    Point resultPoint)
{
    std::vector<std::vector<Point>> initialPoints = generateInitialPoints(interval,
numOfPoints, 20);

    std::vector<Point> res;
    steady_clock::time_point start, stop;
    std::vector<int> linearTime, concurrentTime;

    int linearPassedTest = 0;
    int concurrentPassedTest = 0;

    std::cout << "\n---- Linear ----\n";

    // Preheat linear
    for (int i = 0; i < 20; i++)
    {
        linearGradientDescentPoint(grad, initialPoints[i], LEARN_RATE, N_ITER,
TOLERANCE);
        std::cout << "Preheat: " << i + 1 << "/20" << std::endl;
    }

    // Measure linear
    for (int i = 0; i < 20; i++)
    {
        start = high_resolution_clock::now();
        res = linearGradientDescentPoint(grad, initialPoints[i], LEARN_RATE, N_ITER,
TOLERANCE);
        stop = high_resolution_clock::now();
        std::cout << "\nMeasure: " << i + 1 << "/20" << std::endl;
        if (testResult(res, resultPoint, 1e-2))
        {
            std::cout << "Test passed!\n";

```

```

        linearPassedTest++;
    }
    else
        std::cout << "Test failed!\n";
    linearTime.push_back(duration_cast<microseconds>(stop - start).count());
}

std::cout << "\n---- Concurrent ----\n";
std::cout << "Threads: " << numOfThreads << std::endl;

// Preheat concurrent
for (int i = 0; i < 20; i++)
{
    concurrentGradientDescentPoint(numOfThreads, grad, initialPoints[i],
LEARN_RATE, N_ITER, TOLERANCE);
    std::cout << "Preheat: " << i + 1 << "/20" << std::endl;
}

// Measure concurrent
for (int i = 0; i < 20; i++)
{
    start = high_resolution_clock::now();
    res = concurrentGradientDescentPoint(numOfThreads, grad, initialPoints[i],
LEARN_RATE, N_ITER, TOLERANCE);
    stop = high_resolution_clock::now();
    std::cout << "\nMeasure: " << i + 1 << "/20" << std::endl;
    if (testResult(res, resultPoint, 1e-2))
    {
        std::cout << "Test passed!\n";
        concurrentPassedTest++;
    }
    else
        std::cout << "Test failed!\n";
    concurrentTime.push_back(duration_cast<microseconds>(stop - start).count());
}

std::cout << "\n---- Result ----\n"
    << "Linear time: " << average(linearTime) << std::endl
    << "Linear passed tests: " << linearPassedTest << std::endl
    << "Concurrent time: " << average(concurrentTime) << std::endl
    << "Concurrent passed tests: " << concurrentPassedTest << std::endl
    << "Linear time / Concurrent time: " << average(linearTime) /
average(concurrentTime) << std::endl;
}

void testIntervalGradientDescent(
    int numOfThreads,
    Point(*grad)(Point),
    int N_ITER,
    long double LEARN_RATE,
    long double TOLERANCE,
    Interval interval,
    int numOfIntervals,
    Point step,
    Point resultPoint)
{
    std::vector<Point> res;
    steady_clock::time_point start, stop;
    std::vector<int> linearTime, concurrentTime;
    std::vector<Interval> intervals;
    intervals.push_back(interval);
    for (int i = 1; i < 20; i++)
    {
        Interval currInterval = intervals[i-1];

```

```

        currInterval.end = currInterval.end + step;
        intervals.push_back(currInterval);
    }

    int linearPassedTest = 0;
    int concurrentPassedTest = 0;

    std::cout << "\n---- Linear ----\n";

    // Preheat linear
    for (int i = 0; i < 20; i++)
    {
        linearGradientDescentInterval(grad, intervals[i], numOfIntervals, LEARN_RATE,
N_ITER, TOLERANCE);
        std::cout << "Preheat: " << i + 1 << "/20" << std::endl;
    }

    // Measure linear
    for (int i = 0; i < 20; i++)
    {
        start = high_resolution_clock::now();
        res = linearGradientDescentInterval(grad, intervals[i], numOfIntervals,
LEARN_RATE, N_ITER, TOLERANCE);
        stop = high_resolution_clock::now();
        std::cout << "\nMeasure: " << i + 1 << "/20" << std::endl;
        if (testResult(res, resultPoint, 1e-2))
        {
            std::cout << "Test passed!\n";
            linearPassedTest++;
        }
        else
            std::cout << "Test failed!\n";
        linearTime.push_back(duration_cast<microseconds>(stop - start).count());
    }

    std::cout << "\n---- Concurrent ----\n";
    std::cout << "Threads: " << numOfThreads << std::endl;

    // Preheat concurrent
    for (int i = 0; i < 20; i++)
    {
        concurrentGradientDescentInterval(numOfThreads, grad, intervals[i],
numOfIntervals, LEARN_RATE, N_ITER, TOLERANCE);
        std::cout << "Preheat: " << i + 1 << "/20" << std::endl;
    }

    // Measure concurrent
    for (int i = 0; i < 20; i++)
    {
        start = high_resolution_clock::now();
        res = concurrentGradientDescentInterval(numOfThreads, grad, intervals[i],
numOfIntervals, LEARN_RATE, N_ITER, TOLERANCE);
        stop = high_resolution_clock::now();
        std::cout << "\nMeasure: " << i + 1 << "/20" << std::endl;
        if (testResult(res, resultPoint, 1e-2))
        {
            std::cout << "Test passed!\n";
            concurrentPassedTest++;
        }
        else
            std::cout << "Test failed!\n";
        for (int i = 0; i < res.size(); i++)
            std::cout << res[i].pos[0] << ' ';
    }

```

```

        std::cout << "\nTime: " << duration_cast<microseconds>(stop - start).count()
<< '\n';
        concurrentTime.push_back(duration_cast<microseconds>(stop - start).count());
    }

    std::cout << "\n---- Result ----\n"
        << "Linear time: " << average(linearTime) << std::endl
        << "Linear passed tests: " << linearPassedTest << std::endl
        << "Concurrent time: " << average(concurrentTime) << std::endl
        << "Concurrent passed tests: " << concurrentPassedTest << std::endl
        << "Linear time / Concurrent time: " << average(linearTime) /
average(concurrentTime) << std::endl;
}

int main() {
    const int N_ITER = 10000;
    const long double LEARN_RATE = 0.01;
    const long double TOLERANCE = 1e-200;
    const int N_THREAD = 3;
    Point(*GRAD_1VAR)(Point) = f2d;
    Point(*GRAD_2VAR)(Point) = f3d;
    const Interval INTERVAL_1VAR(Point({ -10 }), Point({ 20 }));
    const Interval INTERVAL_2VAR(Point({ -10, -20 }), Point({ 20, 10 }));

    // test algorithm with 1 variable
    testPointGradientDescent(N_THREAD, GRAD_1VAR, N_ITER, LEARN_RATE, TOLERANCE,
INTERVAL_1VAR, 20, Point({ 1.65383 }));

    // test algorithm with 2 variables
    //testPointGradientDescent(N_THREAD, GRAD_2VAR, N_ITER, LEARN_RATE, TOLERANCE,
INTERVAL_2VAR, 20, Point({ 1.65383, -1.65383 }));

    // test interval algorithm
    //testIntervalGradientDescent(N_THREAD, GRAD_1VAR, N_ITER, LEARN_RATE, TOLERANCE,
INTERVAL_1VAR, 20, Point({0.5}), Point({ 1.65383 }));

    std::cout << std::endl;
    system("pause");
    return 0;
}

```

Додаток Г. Лістинг коду додаткових засобів, використаних для реалізації алгоритму

```

template<class T>
long double sum(std::vector<T>& vector)
{
    int size = vector.size();

    if (size == 0)
        return 0;

    T sum = 0;
    for (int i = 0; i < size; i++)
        sum += vector[i];

    return sum;
}

template<class T>
long double average(std::vector<T>& vector)
{
    int size = vector.size();

    if (size == 0)
        return 0;

    return sum(vector) / size;
}

struct Point
{
    std::vector<long double> pos;
    Point* grad;

    Point()
    {
        grad = nullptr;
    }

    Point(const Point &p)
    {
        for (int i = 0; i < p.pos.size(); i++)
            pos.push_back(p.pos[i]);
        grad = p.grad;
    }

    Point(const std::vector<long double>& p)
    {
        for (int i = 0; i < p.size(); i++)
            pos.push_back(p[i]);
        grad = nullptr;
    }

    static Point addVectors(Point p1, Point p2)
    {
        Point result;
        for (int i = 0; i < p1.pos.size(); i++)
            result.pos.push_back(p1.pos[i] + p2.pos[i]);
        return result;
    }
}

```



```

    Point operator+(const Point p1)
    {
        return addVectors(*this, p1);
    }

    static Point getHalfwayPoint(Point p1, Point p2)
    {
        Point result;
        for (int i = 0; i < p1.pos.size(); i++)
            result.pos.push_back((p1.pos[i] + p2.pos[i]) / 2);
        return result;
    }
};

struct Interval
{
    Point start;
    Point end;

    Interval()
    {

    }

    Interval(const Point& start, const Point& end)
    {
        this->start = start;
        this->end = end;
    }

    Point getHalfwayPoint()
    {
        return Point::getHalfwayPoint(start, end);
    }
};

std::vector<std::vector<Point>> generateInitialPoints(Interval interval, int
numOfPointsInVector, int numOfVectors) {
    srand(17);
    std::vector<std::vector<Point>> result(numOfVectors);
    for (int i = 0; i < numOfVectors; i++)
    {
        for (int j = 0; j < numOfPointsInVector; j++)
        {
            Point p;
            for (int k = 0; k < interval.start.pos.size(); k++)
                p.pos.push_back(rand()%(int)(interval.end.pos[k]-
interval.start.pos[k])+interval.start.pos[k]);
            result[i].push_back(p);
        }
    }

    return result;
}

bool testResult(std::vector<Point> res, Point answer, long double tolerance)
{
    for (int i = 0; i < res.size(); i++)
    {
        int countRightPos = 0;
        for (int j = 0; j < answer.pos.size(); j++)

```

```
        if (answer.pos[j] - tolerance < res[i].pos[j] && res[i].pos[j] <
answer.pos[j] + tolerance)
            countRightPos++;
        if (countRightPos == answer.pos.size())
            return true;
    }
    return false;
}
```