Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
Факультет комп'ютерних наук
Кафедра інженерії програмного забезпечення

Звіт
з лабораторної роботи №1
з дисципліни «Теорія паралельних обчислень»
на тему «Паралельні алгоритми розв'язування заповнених систем лінійних рівнянь»

Виконали ст. гр. ІПЗм-22-6:
Миронюк С.А.,
Сєнічкін І.О.

Перевірив викладач:
доц. Кобзєв В.Г.

Харків, 2023

ПАРАЛЕЛЬНІ АЛГОРИТМИ РОЗВ'ЯЗУВАННЯ ЗАПОВНЕНИХ СИСТЕМ ЛІНІЙНИХ РІВНЯНЬ

Мета – навчитися створювати і аналізувати паралельні алгоритми розв'язування заповнених систем лінійних рівнянь та оцінювати показники їх прискорення у порівнянні з послідовними алгоритмами.

Індивідуальне завдання:

Система лінійних алгебраїчних рівнянь (СЛАР) задана матрицею коефіцієнтів *A* та вектором вільних членів *B*. Побудувати графи послідовного та паралельного алгоритмів розв'язання заданої СЛАР одним з наступних методів:

а) матричним методом з використанням приєднаної матриці,

б) матричним методом з використанням елементарних перетворень рядків,

в) методом Крамера,

г) методом Гауса,

д) методом LU-розкладання матриці коефіцієнтів *A*.

Реалізувати алгоритми програмно. Обчислити значення показників прискорення та ефективності розпаралелювання.

Надано варіант: а) матричним методом з використанням приєднаної матриці.

Дані для лабораторної роботи: значення синіх пікселів зображення «Вхід до ХНУРЕ». Рядок – починаючи с 60-го пікселя, строка – з 1-го. Задані розміри матриць; 150×150, 300×300, 500×500, 1000×1000. Стовпчик вільних членів – 6-ій з кінця рисунку 1. У випадку недостатньої інформації на фотографії, нам потрібно додати ще одну копію цієї фотографії. Зазначена фотографія відображена на рисунку 1.

Рисунок 1 – Зображення для виконання завдання

Хід роботи:

Система лінійних алгебраїчних рівнянь має вигляд:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 ; \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 ; \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m , \end{cases}$$

або в матричній формі:

$$A\,X = B,$$

де

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}; \qquad X = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix}; \qquad B = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_m \end{pmatrix}.$$

Розширена матриця системи має вигляд

$$\overline{A} = (A \mid B) = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} & b_m \end{array}\right).$$

Розв'язком системи називається матриця-стовпець, яка обертає матричне рівняння $A\,X = B$ у тотожність.

Матричний метод полягає у розв'язанні матричного рівняння $X = A^{-1}\,B$.

Реалізація методу полягає в знаходженні оберненої матриці і множенні її на стовпець вільних членів. Використовується для невироджених (det $A \neq 0$) квадратних систем.

Оберненою до квадратної матриці A називається матриця A^{-1} така, що

$$A\,A^{-1} = A^{-1}A = E\,.$$

Для того щоб квадратна матриця A мала обернену, необхідно та достатньо,щоб матриця коефіцієнтів A була невиродженою.

Приєднаною до квадратної матриці A називається матриця:

$$\widetilde{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} A_{11} & A_{21} & \ldots & A_{n1} \\ A_{12} & A_{22} & \ldots & A_{n2} \\ \ldots & \ldots & \ldots & \ldots \\ A_{1n} & A_{2n} & \ldots & A_{nn} \end{pmatrix},$$

де A_{ij} – алгебраїчні доповнення елементів a_{ij} матриці A.

Згідно з цим методом обернена матриця знаходиться за формулою:

$$A^{-1} = \frac{1}{\det A}\widetilde{A} = \frac{1}{\det A}\begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{pmatrix}^{T} = \frac{1}{\det A}\begin{pmatrix} A_{11} & A_{21} & \ldots & A_{n1} \\ A_{12} & A_{22} & \ldots & A_{n2} \\ \ldots & \ldots & \ldots & \ldots \\ A_{1n} & A_{2n} & \ldots & A_{nn} \end{pmatrix}.$$

Дана програма була виконана на пристрої DESKTOP-IDO8GRU з процесором Intel(R) Core(TM) i5-7600 CPU, який включає 4 ядра, базовая тактовая частота 3500 МГц, объем кэш памяти 3 уровня (L1 – 256 КБ, L2 – 1,0 МБ, L3 – 6 МБ).

Програмна реалізація алгоритму розв’язання СЛАР матричним методом на мові програмування Python.

Даний код працює із зображеннями, виконує їх обробку та вирішує системи лінійних рівнянь матричним методом, порівнюючи ефективності послідовного та паралельного розв'язання систем лінійних рівнянь для різних розмірів матриць.

В ході виконання роботи при об’єднанні вертикально двох зображень з’ясувалося, що ми отримали виродженою матрицю. Було прийнято рішення при об’єднані зображень другу частину зображення приєднати вертикально з переворотом.

Основні кроки коду:

1. Імпортуються необхідні бібліотеки:

PIL (Python Imaging Library): from PIL import Image – бібліотека для роботи із зображеннями у Python.

NumPy: import numpy as np – бібліотека для виконання числових операцій у Python. concurrent.futures.ThreadPoolExecutor: from concurrent.futures import ThreadPoolExecutor – частина модуля concurrent.futures, що надає пул потоків для паралельної обробки.

scipy.linalg.lstsq: from scipy.linalg import lstsq – функція в модулі scipy.linalg для вирішення лінійних матричних рівнянь.

time: import time – бібліотека надає різні функції, пов'язані з часом.

2. Обробка зображення:

Для об’єднання та обрізання зображень визначено дві функції (glue_images_vertically_with_flip та crop_image).

Зображення завантажуються та трансформуються за допомогою бібліотеки PIL.

3. Матричне та векторне перетворення:

Функції (image_to_matrix_float та image_to_vector_float) перетворюють зображення відповідно на матриці та вектори чисел з плаваючою комою.

```python
from PIL import Image                    # Python Imaging Library, використовується для роботи з зображеннями
import time                              # Надає різноманітні функції, пов’язані з часом
import numpy as np                       # бібліотека для числових операцій у Python
from concurrent.futures import ThreadPoolExecutor   # частина модуля concurrent.futures для паралельної обробки
from scipy.linalg import lstsq                  # розв’язок лінійного матричного рівняння

# Функція об'єднання двох зображень вертикально з переворотом другого зображення
def glue_images_vertically_with_flip(image1, image2):  # склеює їх разом вертикально та перевертає друге зображення горизонтально перед вставленням
    width = max(image1.width, image2.width)
    height = image1.height + image2.height
    result = Image.new('RGB', (width, height), 'white')
    result.paste(image1, (0, 0))

    # Перевернути друге зображення перед приклеюванням
    image2_flipped = image2.transpose(method=Image.FLIP_LEFT_RIGHT)

    result.paste(image2_flipped, (0, image1.height))
    return result

# Функція обрізки зображення за заданими координатами та розмірами
def crop_image(image, x, y, width, height):
    return image.crop((x, y, x + width, y + height))    # обрізає зображення на основі заданих координат (x, y) і розмірів (ширина, висота)

# Перетворення зображення в матрицю чисел типу float
def image_to_matrix_float(image):
    width, height = image.size
    matrix = np.zeros((height, width), dtype=float)

    for i in range(height):                # перетворює зображення на матрицю чисел з плаваючою комою, вилучаючи синій канал
        for j in range(width):
            color = image.getpixel((j, i))
            blue = color[2]
            matrix[i][j] = blue

    return matrix

# Перетворення зображення в вектор чисел типу float
def image_to_vector_float(image):          # перетворює зображення на вектор чисел з плаваючою комою, вилучаючи синій канал
    width, height = image.size
    vector = np.zeros(height, dtype=float)
```

Рисунок 2 – Частина коду

4. Операції лінійної алгебри:

Дві функції (solve_matrix_equation та solve_matrix_equation_parallel) визначено для розв’язування лінійних систем рівнянь за допомогою можливостей лінійної алгебри та паралельної обробки NumPy.

```
    for i in range(height):
        color = image.getpixel((0, i))
        blue = color[2]
        vector[i] = blue

    return vector

# Друк матриці
def print_matrix(matrix):
    for row in matrix:
        print(row)               # призначена для друку матриць

# Друк вектора
def print_vector(vector):
    print(vector)               # призначена для друку векторів

# Послідовний метод розв'язання системи лінійних рівнянь
def solve_matrix_equation(matrix, vector):       # розв‘язок системи лінійних рівнянь
    try:
        result_vector = np.linalg.solve(matrix, vector)
        return result_vector
    except np.linalg.LinAlgError:
        print("Матриця є одиничною. Не вміє розв’язувати систему лінійних рівнянь.")
        return None

# Паралельний метод розв'язання системи лінійних рівнянь
def solve_matrix_equation_parallel(matrix, vector):
    try:
        result_vector = lstsq(matrix, vector)[0]
        return result_vector
    except np.linalg.LinAlgError:
        print("Матриця є одиничною. Не вміє розв’язувати систему лінійних рівнянь.")
        return None
```

Рисунок 3 – Код розв'язання системи лінійних рівнянь послідовним і паралельним методами

5. Виконання основного коду:

Основний блок коду використовує ThreadPoolExecutor для розпаралелювання обробки різних розмірів матриці.
Для кожного розміру матриці в matrix_sizes (150, 300, 500, 1000):
Зображення завантажується та обробляється.
З зображень витягуються матриці та вектори.
Рішення лінійної системи обчислюються послідовно та паралельно.
Час виконання для обох методів вимірюється та друкується.
Результати обох методів друкуються.

```python
try:    # Основний блок коду обгорнено блоком try-except для перехоплення IOError винятків
    # Розміри матриць, для яких буде виконано розв'язання систем
    matrix_sizes = [150, 300, 500, 1000]  # розміри матриць

    # вказуємо кількість потоків у ThreadPoolExecutor
    with ThreadPoolExecutor(max_workers=4) as executor:
        for size in matrix_sizes:
            # Завантаження зображення "knure.jpg"
            image_knure = Image.open("images/knure.jpg")  # він завантажує зображення, виконує перетворення та витягує матричні та векторні представлення
            start_time = time.time()

            # Об'єднання та обрізка зображень с переворотом второго изображения
            image_glued_knure = glue_images_vertically_with_flip(image_knure, image_knure)
            image_knure_matrix = crop_image(image_glued_knure, 60, 0, size, size)
            image_knure_vector = crop_image(image_glued_knure, 1163, 0, 1, size)

            # Перетворення зображень у числові представлення
            matrix_float = image_to_matrix_float(image_knure_matrix)
            vector_float = image_to_vector_float(image_knure_vector)

            # Перетворення у NumPy масиви
            matrix = np.array(matrix_float, dtype=float)
            vector = np.array(vector_float, dtype=float)

            # Розв'язання системи лінійних рівнянь (послідовний метод)
            start_time_sequential = time.time()
            result_vector = solve_matrix_equation(matrix, vector)
            end_time_sequential = time.time()
            elapsed_time_sequential = (end_time_sequential - start_time_sequential) * 1000  # у мілісекундах

            # Розв'язання системи лінійних рівнянь (паралельний метод)
            start_time_parallel = time.time()
            result_vector_parallel = executor.submit(solve_matrix_equation_parallel, matrix, vector)
            end_time_parallel = time.time()
            elapsed_time_parallel = (end_time_parallel - start_time_parallel) * 1000  # у мілісекундах


            # Виведення часу виконання
            print("Розмір матриці {}: ".format(size))
            print("Час виконання послідовного методу: {} мілісекунд".format(float(elapsed_time_sequential)))
```

Рисунок 4 – Код розв'язання системи лінійних рівнянь послідовним і паралельним методами

```
-6.59590210e+00  1.76536263e+01 -2.38306435e+01  1.17676196e+01
-2.66332265e+00 -2.10308714e+01  2.91492792e+01 -1.08154818e+01
-9.69210842e+00  2.32813549e+01 -2.82310612e+01  1.93529934e+01
-7.34233688e+00  3.34959962e+00  6.11141947e+00 -2.36952545e+00
-9.53267971e+00  5.89128727e+00 -8.25762167e-01 -1.59290575e+00
-5.50335770e+00  1.42013369e+01 -7.77371374e+00  2.18825230e+00
-5.27686436e+00  1.62730416e+01 -9.06120899e+00 -2.62889675e+00
 2.82031418e+00  1.61953557e+01 -1.41939271e+01 -2.92432145e+00
 7.89073206e+00 -1.53134075e+01  1.60007468e+01 -5.14133788e+00
 9.12589466e-03 -2.58084925e+00 -4.12466708e+00  5.81610150e+00
 2.28979484e-02 -9.43738246e+00  1.71505599e+00  3.26506777e+00
-1.42107970e+01  1.47132667e+01 -9.40856559e-01 -8.12738922e+00
 1.53497305e+00 -1.95928870e+00  1.08983843e+01 -1.12042522e+01
 5.36650889e+00  6.13451996e+00 -9.90754288e+00  1.89473648e+01
-1.57806213e+01  1.06386905e+01 -1.24789400e+00 -7.79082476e+00
-1.36397447e+00  2.07449542e+00 -2.41359544e+00  1.94049603e+01
-3.26163579e+01  3.22123125e+01 -1.90511977e+01  2.91037321e-02
 1.45775812e+01 -2.94041401e-01 -1.65142045e+01 -4.87150322e+00
 3.52129663e+01 -4.34505297e+01  3.24388837e+01 -1.06736693e+01
-2.00750900e+00 -2.91445728e+00 -1.40686732e+01  2.66668641e+01
-1.86848210e+01 -1.12070182e+00  2.46428417e+01 -2.21739780e+01
-2.26607387e+00  2.11834535e+01 -1.05278280e+01 -1.10375373e+01
 2.42403070e+01 -1.53040718e+01  5.14244212e+00  2.49532443e+00
-7.16017349e+00  1.74136916e+01 -4.77581287e+01  5.06708902e+01
-3.07284137e+01  3.21631770e+01 -3.57021244e+01  2.00100037e+01
-8.34837807e+00  3.31088398e+00  7.27435237e+00 -9.66321077e+00
 6.55338121e+00 -4.93612236e+00 -9.72066942e+00  9.60832422e+00
-7.96382531e+00  1.89010896e+01 -1.89837397e+01 -2.26196894e+00
 1.82457101e+01 -1.06725422e+01 -4.98119542e+00  4.63020670e+00
-8.96034731e-01  2.82216082e+00 -6.08008184e+00  1.02751756e+01
-1.39487057e+01  1.45190880e+01 -9.86166625e+00 -1.93856272e+00
 1.86503585e+00 -7.90011452e-01  5.63903971e+00  3.40412282e+00
-1.86487589e+00 -2.02964891e+00 -3.67233437e+00  8.51095210e+00
 6.70392051e+00 -3.58200126e+00 -1.04139872e+01  8.28613398e+00
 1.39018826e+01 -3.23103512e+01  3.16799502e+01 -1.82055650e+01
 3.14625131e+00 -3.47593925e+00 -1.73227822e+00  2.78349648e+00]

Process finished with exit code 0
```

Рисунок – Фрагмент результатів обох методів для кожного розміру матриці

Результати виводяться на екран, включаючи розмір матриці, час виконання послідовного та паралельного методів.

```
C:\Users\MSA\anaconda3\python.exe C:\Users\MSA\PycharmProjects\TeoriaParallelSolution-Lab01Sequintiol\main.py
Розмір матриці 150:
Час виконання послідовного методу: 6.824970245361328 мілісекунд
Час виконання паралельного методу: 0.9784698486328125 мілісекунд
Розмір матриці 300:
Час виконання послідовного методу: 2.9141902923583984 мілісекунд
Час виконання паралельного методу: 0.9782314300537109 мілісекунд
Розмір матриці 500:
Час виконання послідовного методу: 11.722087860107422 мілісекунд
Час виконання паралельного методу: 0.9753704071044922 мілісекунд
Розмір матриці 1000:
Час виконання послідовного методу: 78.0949592590332 мілісекунд
Час виконання паралельного методу: 0.9772777557373047 мілісекунд

Process finished with exit code 0
```

Рисунок – Час виконання для вирішення лінійних систем послідовно та паралельно для кожного розміру матриці

Таблиця 1 – Час виконання алгоритмів розв'язання СЛАУ для матриць різних розмірів

Розмір матриці	Послідовний метод	Паралельний метод	Прискорення, $S = \frac{T_{sequential}}{T_{parallel}}$	Ефективність, $E = \frac{S}{число\ потоків}$
	Час виконання послідовного методу, мс	Час виконання паралельного методу, мс		
150×150	6,825	0,97	6,98	1,745
300×300	2,914	0,97	2,98	0,745
500×500	11,722	0,97	12,02	3,005
1000×1000	78,098	0,98	79,94	19,985

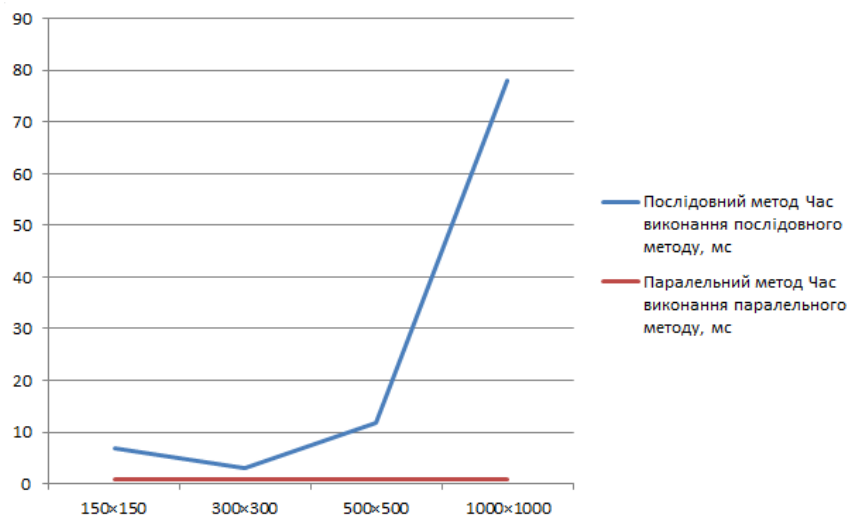

Рисунок 8. Графік залежності часу виконання від розміру матриці для паралельної і послідовної реалізації алгоритму рішення СЛАУ матричним методом

Висновок: У ході роботи була створена послідовна і паралельна реалізація алгоритму розв’язання СЛАР матричним методом. За результатами роботи паралельного методу отримали прискорення часу виконання розрахунків. Проведено порівняння ефективності послідовного та паралельного розв'язання систем лінійних рівнянь для різних розмірів матриць Паралельна реалізація була швидше за послідовну для матриці 150×150 майже в 7 разів, для 300×300 – в 3 рази, для 500×500 – в 12 раз, а для матриці 1000×1000 – в 80 разів. Завдяки використанню багатопотоковості та розподілених обчислень вдалося досягти збільшення продуктивності та швидкості обробки даних.

.

Відповді на контрольні питання:

1. Поясніть, у чому полягає суть алгоритму викреслювання стовпців, як виконується оцінка коефіцієнта прискорення цього алгоритму у разі блокового розподілу даних. Спробуйте оцінити коефіцієнт прискорення у разі блочно-циклічного розподілу даних в припущенні відсутності часу на обмін даними між процесами.

Алгоритм викреслювання стовпців використовується для розв'язання систем лінійних рівнянь та зменшення обчислювальної складності задачі. Суть алгоритму полягає в тому, щоб послідовно виключати стовпці з матриці лінійної системи, використовуючи вже знайдені розв'язки для зменшення розмірності системи.

Нехай необхідно вирішити таку задачу:

$$x_1=c_1$$
$$x_2=c_2+a_{21}x_1$$
$$x_3=c_3+a_{31}x_1+a_{32}x_2$$
$$x_4=c_4+a_{41}x_1+a_{42}x_2+a_{43}x_3$$

З системи неважко помітити, що якщо відомо x1, то всі вирази

$$c_i^1 = a_{i1} + c_1^i, \quad i = 2, \dots n,$$

де, n – вимірність вектора x, можуть бути обчислені паралельно, тобто незалежно один від одного.

В свою чергу, якщо відомо x_1 і x_2, то вирази також можуть обчислюватись паралельно.

$$c_i^2 = a_{i2} + c_1^i, \quad i = 3, \dots n.$$

Вважаючи, що одна одиниця часу еквівалентна часу виконання однієї операції незалежно від її типу, можна визначити коефіцієнт прискорення $К_{ПР}$ для цього алгоритму. Зрозуміло, що для оцінки $К_{ПР}$ досить лише оцінити кількість неодночасно виконуваних операцій у послідовному й у паралельному варіантах алгоритму викреслювання стовпців. При такому оцінюванні можна не враховувати час, необхідний для встановлення зв’язків і обміну даними між процесорами під час розрахунків.

Для послідовного алгоритму загальне число операцій дорівнює:

$$N_1 = \sum_{j=1}^{n-1} 2j = 2(0{,}5(n-1)n) = n^2 - n.$$

Для паралельного варіанта алгоритму викреслювання стовпців, кожному з p процесорів системи найпростіше надати масив даних довжиною $q = n/p$ (блоковий розподіл даних), і вважати, що кожен процесор виконує обчислення тільки в межах цих даних. При такій організації обчислень, кількість неодночасно виконуваних операцій для оцінки величин максимум буде дорівнювати $2q$. Після того як всіма процесорами були

зроблені всі необхідні операції, процесори обмінюються один з одним отриманими результатами.

Таким чином, для такого варіанта алгоритму викреслювання стовпців загальна кількість неодночасно виконуваних операцій, яка необхідна для розв'язання поставленої задачі, обумовлюється співвідношенням:

$$N_p = 2(n-1)n \ / \ p = \frac{2}{p}(n^2 - n).$$

Остаточно звідси витікає:

$$K_{\text{ПР}} = N_1 \ / \ N_p = p/2.$$

Оцінка коефіцієнта прискорення у разі блочно-циклічного розподілу даних в припущенні відсутності часу на обмін даними між процесами:

В разі блочно-циклічного розподілу даних, кожен процес отримує певний блок даних та самостійно працює з ним. Якщо при цьому відсутній час на обмін даними між процесами, то кожен процес може виконувати свою частину роботи паралельно.
Оцінка коефіцієнта прискорення у цьому випадку буде аналогічною.

3. Опишіть модифікацію алгоритму викреслювання стовпців, яка дозволяє практично уникнути впливу ефекту Гайдна на прискорення паралельних обчислень.

Ефект Гайдна визначає, як швидкість виконання завдань масштабується з ростом ресурсів (процесорів). У традиційній постановці ефект Гайдна вказує на те, що збільшення кількості ресурсів (наприклад, процесорів) повинно збільшувати пропорційно швидкість виконання завдань.

Проте, у практиці, існує явище, відоме як "неналежне масштабування" або ефект Амдала, коли реальний приріст продуктивності обмежений часткою програми, яка не може бути паралельною.

Однією з модифікацій алгоритму викреслювання стовпців для уникнення впливу ефекту Гайдна може бути використання динамічного розподілу роботи між процесорами. Замість фіксованого блочного розподілу даних, де кожен процесор отримує певний блок стовпців, можна використовувати динамічний підхід, де процесори динамічно отримують нові блоки для обробки.

Кроки модифікованого алгоритму:

1. Початкове розподіл роботи:

Кожен процесор отримує початковий блок стовпців для обробки.

2. Динамічне викреслювання стовпців:

Після обробки свого блоку, кожен процесор запитує новий блок для обробки.
Процесор продовжує обробку нових блоків, поки не вирішиться вся система лінійних рівнянь.

3. Динамічний розподіл роботи:

Кількість стовпців у новому блоку може динамічно змінюватися в залежності від завдань, які залишилися для вирішення та доступних ресурсів.

Цей підхід дозволяє ефективніше використовувати ресурси, уникаючи статичного розподілу, і може допомогти зменшити вплив ефекту Гайдна. Проте, важливо бути уважним при використанні динамічного розподілу, оскільки може виникнути додатковий наклад на обмін даними між процесорами, і це також може вплинути на загальну ефективність алгоритму.

5. На чому оснований алгоритм рекурентного добутку? Що таке оцінка складності алгоритму і як через неї виражається коефіцієнт прискорення?

Алгоритм рекурентного добутку використовується для обчислення добутку чисел у вигляді рекурентного виразу. Зазвичай, рекурентний добуток визначається виглядом:

$$P(n) = P(n/2) \cdot P(n/2),$$

де $P(n)$ – добуток n чисел.

На практиці, для великих значень n, алгоритм може бути ефективно реалізований за допомогою розбиття задачі на менші підзадачі та використання рекурентної стратегії.

Оцінка складності алгоритму визначає кількість операцій чи ресурсів, які він використовує в залежності від розміру вхідних даних. Для алгоритму рекурентного добутку, оцінка складності визначається кількістю рекурсивних викликів та операцій, які виконуються на кожному рівні рекурсії.

Оцінка складності може бути виражена у вигляді асимптотичної нотації, наприклад, $0(\log n)$, де n – розмір вхідних даних. У випадку алгоритму рекурентного добутку, через рекурентну структуру, зазвичай може бути показано, що він має логарифмічну складність.

Коефіцієнт прискорення в контексті паралельних обчислень визначає, як ефективно використовуються додаткові ресурси (процесори) для розв'язання задачі. Коефіцієнт прискорення $S(p)$ для p процесорів обчислюється відношенням часу виконання на одному процесорі $T(1)$ до часу виконання на p процесорах $T(p)$:

$$S(p)=T(p)T(1)$$

Якщо алгоритм може бути ефективно паралелізований, коефіцієнт прискорення буде більше 1 при збільшенні кількості процесорів. У випадку алгоритму рекурентного добутку, який є рекурсивним, паралельнізація може відбутися, наприклад, через розподіл обчислень на різні частини дерева рекурсії, якщо можливо виконувати обчислення на різних рівнях дерева незалежно. Коефіцієнт прискорення визначає, наскільки ефективно вдається використати паралельні обчислення для розв'язання задачі.