МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Факультет комп’ютерних наук

Кафедра програмної інженерії

ЗВІТ

з лабораторної роботи №1

з курсу: «Теорія паралельних обчислень»

на тему: «Паралельні алгоритми розв’язування заповнених систем лінійних рівнянь»

Варіант №11

Виконала:

ст. гр. ІПЗм-19-2

Михневич Тетяна

Перевірив:

доц. каф. ПІ

Кобзєв В.Г.

Харків 2020

**Мета роботи**

Навчитися створювати і аналізувати паралельні алгоритми розв’язування заповнених систем лінійних рівнянь та оцінювати показники їх прискорення у порівнянні з послідовними алгоритмами.

**Завдання**

Картинка для виконання лабораторної роботи:



Значення зеленого у кожному пікселі картинки, починаючи з лівого верхнього кута – коефіцієнти матриці. Стовбець вільних членів – значення зеленого у кожному пікселі стовбця, починаючи з правого верхнього кута.

Три варіанта матриць:

10x10

100x100

500x500

Рішення системі лінійних рівнянь буде виконуватися матричним методом з використанням приєднаної матриці.

**Хід роботи**

Алгоритм виконання лабораторної роботи.

1. Сформувати матрицю відповідного розміру для вирішення СЛАР. Для цього пройти по кожному пікселю зображення та взяти значення G (зеленого кольору). Для програмування алгоритму будемо використовувати мову програмування C#.

Відповідна функція наведена нижче.

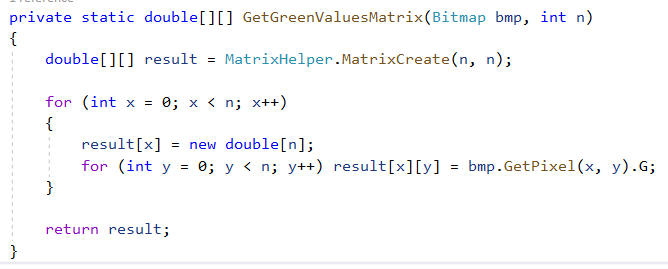


Рисунок 1 – Формування матриці певного розміру зеленого кольору

1. Сформувати стовбець вільних членів відповідного розміру. Для цього пройти по кожному пікселю правого стовбця зображення та взяти значення G.

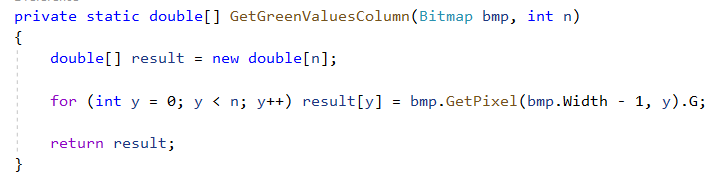


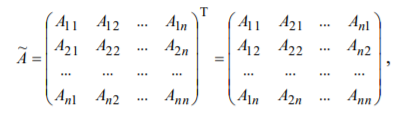
Рисунок 2 – Формування стовпця вільних членів певного розміру зеленого кольору

1. Запрограмувати вирішення системи послідовним алгоритмом – матричним методом з використанням приєднаної матриці.

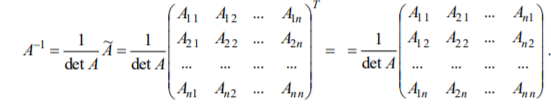
Реалізація методу полягає в знаходженні оберненої матриці і множенні її на стовпець вільних членів. Використовується для невироджених (det A≠ 0) квадратних систем.

Оберненою до квадратної матриці A називається матриця така, що

Для того щоб квадратна матриця A мала обернену, необхідно та достатньо, щоб матриця A була невиродженою. Приєднаною до квадратної матриці A називається матриця



де Aij – алгебраїчні доповнення елементів aij матриці A. Згідно з цим методом обернена матриця знаходиться за формулою



Код наведено у додатку А.

1. Розпаралелити отриманий послідовний алгоритм. Код наведено у додатку А.
2. Виконаємо програму для різних розмірів матриці (див. рис. 3).



Рисунок 3 – Виконання програми для різних розмірів матриці

1. Заміряємо час виконання послідовно та паралельно та занесемо його у таблицю:

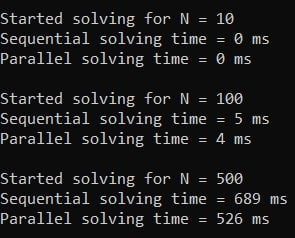


Рисунок 4 – Результат виконання програми

|  |  |  |
| --- | --- | --- |
|  | Послідовно, мс | Паралельно, мс |
| 10x10 | 0 | 0 |
| 100x100 | 5 | 4 |
| 500x500 | 689 | 526 |

На малих об’ємах даних витрати на створювання потоків дорівнюють витратам на виконання корисної роботи, тому алгоритми працюють однаково за часом. На великих об’ємах вже стає видно перевагу від розпаралелювання обчислень.

1. Обчислити значення показників прискорення та ефективності розпаралелювання за допомогою закону Амдала.

Прискорення S (як відношення до часу виконання послідовного алгоритму) за законом Амдала задається рівнянням:

де: p— частина яку можна виконувати послідовно; 1 - p частина, яка виконується паралельно; n — кількість процесорів.

У нашому випадку n = 2, p = 0.2:

Обчислити значення показників прискорення та ефективності розпаралелювання за допомогою закону Густавсона.

1. де g — частка послідовних обчислень в програмі, p — кількість процесорів.

У нашому випадку p = 2, g = 0.2:

**Висновки**

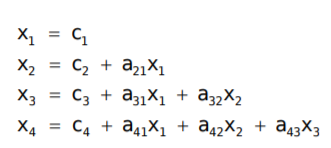
У результаті виконання лабораторної роботи ми навчилися створювати і аналізувати паралельні алгоритми розв’язування заповнених систем лінійних рівнянь та оцінювати показники їх прискорення у порівнянні з послідовними алгоритмами. Був реалізований послідовно та паралельно матричний метод з використанням приєднаної матриці. На малих об’ємах даних витрати на створювання потоків дорівнюють витратам на виконання корисної роботи, тому алгоритми працюють однаково за часом. На великих об’ємах вже стає видно перевагу від розпаралелювання обчислень.

**Контрольні запитання**

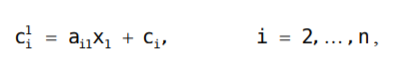
(запитання 1, 3, 5)

***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 блочноциклічного розподілу даних в припущенні відсутності часу на обмін даними між процесами.***

Нехай необхідно вирішити таку задачу:



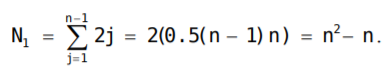
З системи неважко помітити, що якщо відомо x1, то всі вирази

 (1)

де, n – вимірність вектора x, можуть бути обчислені паралельно, тобто незалежно один від одного. В свою чергу, якщо відомо x1 і x2, то вирази

 (2)

також можуть обчислюватись паралельно. Вважаючи, що одна одиниця часу еквівалентна часу виконання однієї операції незалежно від її типу, можна визначити коефіцієнт прискорення КПР для цього алгоритму. Зрозуміло, що для оцінки КПР 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:



Для паралельного варіанта алгоритму викреслювання стовпців, кожному з р 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. Неважко помітити, що при такій організації обчислень, кількість неодночасно виконуваних операцій для оцінки величин (1) або (2) 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:

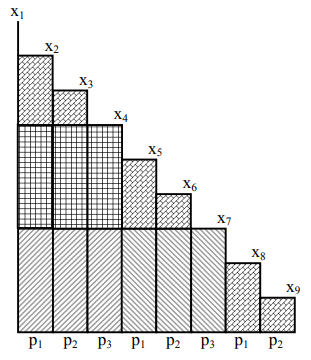


Остаточно звідси витікає:



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

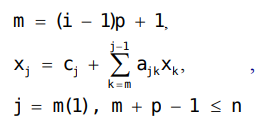
Для уникнення ефекту Гайдна всі дані вже блочно-циклічним способом розподілу розрізаються на h = [n / p] смуг, кожна з яких шириною p. Графічний приклад розподілення задач для 3-ьох процесорів:



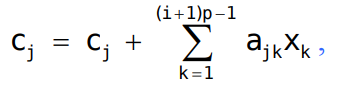
Тоді алгоритм можна побудувати у такий спосіб:

Для всіх i=1(1)h виконується така послідовність кроків

1. Послідовний алгоритм:

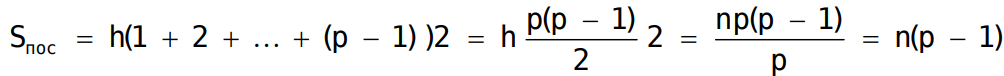


1. Паралельний крок:

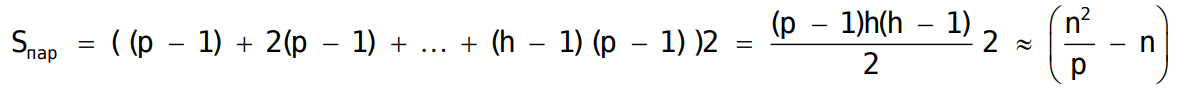


де j = m (1), m + p – 1.

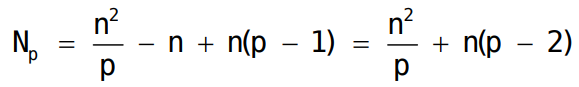
Якщо ж провести підрахунок кількості неодночасно виконуваних операцій, необхідних для роботи алгоритму, то для послідовних кроків їх кількість Sпос дорівнює:



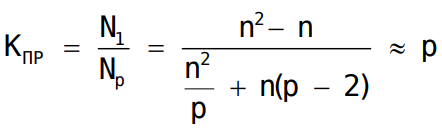
Відповідно для паралельних кроків ця кількість буде:



Таким чином сумарна кількість операцій:



Коефіцієнт прискорення:



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

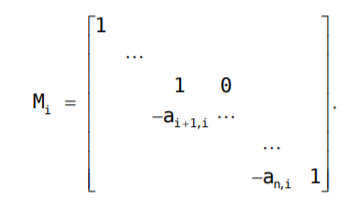
У цьому алгоритмі використовується той факт, що лінійну рекурентну задачу можна звести до розв’язання системи рівнянь вигляду:

(3)

де x і c – вектори вимірністю n, а А – строго нижньотрикутна матриця. Тоді з (3) витікає, що:

(4)

У свою чергу в (4) L - E = Mn \* Mn-1 \* … \* M1, де:



Безсумнівно, що добуток Mic можна виконати за допомогою процедури логарифмічного підсумовування. А це означає, що для алгоритму рекурентного добутку коефіцієнт прискорення виражений через **оцінку складності алгоритму**, буде:



де запис y = O(x), означає буквально таке: існує така позитивна константа α, що для всіх x, більших від деякого доволі великого значення x0, виконується нерівність y ≤ αx.

ДОДАТОК А

КОД ПОСЛІДОВНОЇ ПРОГРАМИ

public static class SequentialMatrixSolver

{

public static double[] Solve(double[][] matrix, double[] b)

{

var inversedMatrix = MatrixInverse(matrix);

var res = MatrixProduct(inversedMatrix, b);

return res;

}

private static double[][] MatrixInverse(double[][] matrix)

{

int n = matrix.Length;

double[][] result = MatrixHelper.MatrixDuplicate(matrix);

int[] perm;

int toggle;

double[][] lum = MatrixDecompose(matrix, out perm, out toggle);

if (lum == null)

throw new Exception("Unable to compute inverse");

double[] b = new double[n];

for (int i = 0; i < n; ++i)

{

for (int j = 0; j < n; ++j)

{

if (i == perm[j])

b[j] = 1.0;

else

b[j] = 0.0;

}

double[] x = HelperSolve(lum, b);

for (int j = 0; j < n; ++j)

result[j][i] = x[j];

}

return result;

}

private static double[] HelperSolve(double[][] luMatrix, double[] b)

{

// before calling this helper, permute b using the perm array

// from MatrixDecompose that generated luMatrix

int n = luMatrix.Length;

double[] x = new double[n];

b.CopyTo(x, 0);

for (int i = 1; i < n; ++i)

{

double sum = x[i];

for (int j = 0; j < i; ++j)

sum -= luMatrix[i][j] \* x[j];

x[i] = sum;

}

x[n - 1] /= luMatrix[n - 1][n - 1];

for (int i = n - 2; i >= 0; --i)

{

double sum = x[i];

for (int j = i + 1; j < n; ++j)

sum -= luMatrix[i][j] \* x[j];

x[i] = sum / luMatrix[i][i];

}

return x;

}

private static double[][] MatrixDecompose(double[][] matrix, out int[] perm, out int toggle)

{

// Doolittle LUP decomposition with partial pivoting.

// rerturns: result is L (with 1s on diagonal) and U;

// perm holds row permutations; toggle is +1 or -1 (even or odd)

int rows = matrix.Length;

int cols = matrix[0].Length; // assume square

if (rows != cols)

throw new Exception("Attempt to decompose a non-square m");

int n = rows; // convenience

double[][] result = MatrixHelper.MatrixDuplicate(matrix);

perm = new int[n]; // set up row permutation result

for (int i = 0; i < n; ++i) { perm[i] = i; }

toggle = 1; // toggle tracks row swaps.

// +1 -greater-than even, -1 -greater-than odd. used by MatrixDeterminant

for (int j = 0; j < n - 1; ++j) // each column

{

double colMax = Math.Abs(result[j][j]); // find largest val in col

int pRow = j;

// reader Matt V needed this:

for (int i = j + 1; i < n; ++i)

{

if (Math.Abs(result[i][j]) > colMax)

{

colMax = Math.Abs(result[i][j]);

pRow = i;

}

}

// Not sure if this approach is needed always, or not.

if (pRow != j) // if largest value not on pivot, swap rows

{

double[] rowPtr = result[pRow];

result[pRow] = result[j];

result[j] = rowPtr;

int tmp = perm[pRow]; // and swap perm info

perm[pRow] = perm[j];

perm[j] = tmp;

toggle = -toggle; // adjust the row-swap toggle

}

if (result[j][j] == 0.0)

{

// find a good row to swap

int goodRow = -1;

for (int row = j + 1; row < n; ++row)

{

if (result[row][j] != 0.0)

goodRow = row;

}

if (goodRow == -1)

throw new Exception("Cannot use Doolittle's method");

// swap rows so 0.0 no longer on diagonal

double[] rowPtr = result[goodRow];

result[goodRow] = result[j];

result[j] = rowPtr;

int tmp = perm[goodRow]; // and swap perm info

perm[goodRow] = perm[j];

perm[j] = tmp;

toggle = -toggle; // adjust the row-swap toggle

}

for (int i = j + 1; i < n; ++i)

{

result[i][j] /= result[j][j];

for (int k = j + 1; k < n; ++k)

{

result[i][k] -= result[i][j] \* result[j][k];

}

}

}

return result;

}

private static double[] MatrixProduct(double[][] matrixA, double[] matrixB)

{

int aRows = matrixA.Length;

int aCols = matrixA[0].Length;

int bRows = matrixB.Length;

if (aCols != bRows)

throw new Exception("Non-conformable matrices in MatrixProduct");

double[] result = new double[aRows];

for (int i = 0; i < aRows; ++i) // each row of A

for (int k = 0; k < aCols; ++k) // could use k less-than bRows

result[i] += matrixA[i][k] \* matrixB[k];

return result;

}

}

КОД ПАРАЛЕЛЬНОЇ ПРОГРАМИ

public static class ParallelMatrixSolver

{

public static double[] Solve(double[][] matrix, double[] b)

{

var inversedMatrix = MatrixInverse(matrix);

var res = MatrixProduct(inversedMatrix, b);

return res;

}

private static double[][] MatrixInverse(double[][] matrix)

{

int n = matrix.Length;

double[][] result = MatrixHelper.MatrixDuplicate(matrix);

int[] perm;

int toggle;

double[][] lum = MatrixDecompose(matrix, out perm, out toggle);

if (lum == null)

throw new Exception("Unable to compute inverse");

double[] b = new double[n];

for (int i = 0; i < n; ++i)

{

for (int j = 0; j < n; ++j)

{

if (i == perm[j])

b[j] = 1.0;

else

b[j] = 0.0;

}

double[] x = HelperSolve(lum, b);

for (int j = 0; j < n; ++j)

result[j][i] = x[j];

}

return result;

}

private static double[] HelperSolve(double[][] luMatrix, double[] b)

{

// before calling this helper, permute b using the perm array

// from MatrixDecompose that generated luMatrix

int n = luMatrix.Length;

double[] x = new double[n];

b.CopyTo(x, 0);

for (int i = 1; i < n; ++i)

{

double sum = x[i];

for (int j = 0; j < i; ++j)

sum -= luMatrix[i][j] \* x[j];

x[i] = sum;

}

x[n - 1] /= luMatrix[n - 1][n - 1];

for (int i = n - 2; i >= 0; --i)

{

double sum = x[i];

for (int j = i + 1; j < n; ++j)

sum -= luMatrix[i][j] \* x[j];

x[i] = sum / luMatrix[i][i];

}

return x;

}

private static double[][] MatrixDecompose(double[][] matrix, out int[] perm, out int toggle)

{

// Doolittle LUP decomposition with partial pivoting.

// rerturns: result is L (with 1s on diagonal) and U;

// perm holds row permutations; toggle is +1 or -1 (even or odd)

int rows = matrix.Length;

int cols = matrix[0].Length; // assume square

if (rows != cols)

throw new Exception("Attempt to decompose a non-square m");

int n = rows; // convenience

double[][] result = MatrixHelper.MatrixDuplicate(matrix);

perm = new int[n]; // set up row permutation result

for (int i = 0; i < n; ++i) { perm[i] = i; }

toggle = 1; // toggle tracks row swaps.

// +1 -greater-than even, -1 -greater-than odd. used by MatrixDeterminant

CalculateEachColumn(result, n, perm, toggle);

return result;

}

private static void CalculateEachColumn(double[][] result, int n, int[] perm, int toggle)

{

Parallel.For(0, n - 1, j => // each column

{

double colMax = Math.Abs(result[j][j]); // find largest val in col

int pRow = j;

// reader Matt V needed this:

for (int i = j + 1; i < n; ++i)

{

if (Math.Abs(result[i][j]) > colMax)

{

colMax = Math.Abs(result[i][j]);

pRow = i;

}

}

// Not sure if this approach is needed always, or not.

if (pRow != j) // if largest value not on pivot, swap rows

{

double[] rowPtr = result[pRow];

result[pRow] = result[j];

result[j] = rowPtr;

int tmp = perm[pRow]; // and swap perm info

perm[pRow] = perm[j];

perm[j] = tmp;

toggle = -toggle; // adjust the row-swap toggle

}

if (result[j][j] == 0.0)

{

// find a good row to swap

int goodRow = -1;

for (int row = j + 1; row < n; ++row)

{

if (result[row][j] != 0.0)

goodRow = row;

}

if (goodRow == -1)

throw new Exception("Cannot use Doolittle's method");

// swap rows so 0.0 no longer on diagonal

double[] rowPtr = result[goodRow];

result[goodRow] = result[j];

result[j] = rowPtr;

int tmp = perm[goodRow]; // and swap perm info

perm[goodRow] = perm[j];

perm[j] = tmp;

toggle = -toggle; // adjust the row-swap toggle

}

for (int i = j + 1; i < n; ++i)

{

result[i][j] /= result[j][j];

for (int k = j + 1; k < n; ++k)

{

result[i][k] -= result[i][j] \* result[j][k];

}

}

});

}

private static double[] MatrixProduct(double[][] matrixA, double[] matrixB)

{

int aRows = matrixA.Length;

int aCols = matrixA[0].Length;

int bRows = matrixB.Length;

if (aCols != bRows)

throw new Exception("Non-conformable matrices in MatrixProduct");

double[] result = new double[aRows];

Parallel.For(0, aRows, i => // each row of A

{

for (int k = 0; k < aCols; ++k) // could use k less-than bRows

result[i] += matrixA[i][k] \* matrixB[k];

});

return result;

}

}