МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

ЧИСЛЕННОЕ РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ МЕТОДОМ ИСКЛЮЧЕНИЯ ГАУССА И ИТЕРАЦИОННЫМИ МЕТОДАМИ

ОТЧЕТ О ПРАКТИКЕ

Студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Аношкина Андрея Алексеевича

Проверил
Старший преподаватель ______________ М. С. Портенко

Саратов 2024

СОДЕРЖАНИЕ

1 Work 05

Задание

Решите систему линейных уравнений согласно варианту параллельным методом Гаусса.

$$\begin{cases} x_1 + x_2 + 4x_3 + 4x_4 + 9x_5 + 9 = 0 \\ 2x_1 + 2x_2 + 17x_3 + 17x_4 + 82x_5 + 146 = 0 \\ 2x_1 + 3x_3 - x_4 + 4x_5 + 10 = 0 \\ x_2 + 4x_3 + 12x_4 + 27x_5 + 26 = 0 \\ x_1 + 2x_2 + 2x_3 + 10x_4 - 37 = 0 \end{cases}$$

Определение задачи решения системы линейных уравнений

Множество n линейных уравнений:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

называется системой линейных уравнений или линейной системой.

В более кратком (матричном) виде система может быть представлена как $Ax = b$, где $A = (a_{ij})$ есть вещественная матрица размера $n \times n$, а вектора b и x состоят из элементов.

Под задачей решения системы линейных уравнений для заданных матрицы A и вектора b обычно понимается нахождение значения вектора неизвестных x, при котором выполняются все уравнения системы.

Метод Гаусса

— Основная идея: приведение матрицы A к верхнему треугольному виду с помощью эквивалентных преобразований.
— Эквивалентные преобразования:
  – умножение уравнения на ненулевую константу;
  – перестановка уравнений;
  – суммирование уравнения с любым другим уравнением системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе — прямой ход метода Гаусса — исходная система линейных уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду.

На обратном ходе метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной , после этого из предпоследнего уравнения становится возможным определение переменной x_{n-1} и т. д.

Прямой ход метода Гаусса

— На итерации i, $0 \leq i < n$, метода производится исключение неизвестной i для всех уравнений с номерами k, больших i ($i \leq k < n$). Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу (a_{ki}/a_{ii}), чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым.

— Все необходимые вычисления определяются при помощи соотношений:

$$\begin{cases} a'_{kj} = a_{kj} - (a_{ki}/a_{ii})\dot{a}_{ij} \\ b'_k = b_k - (a_{ki}/a_{ii})\dot{b}_i \\ i \leq j < n, i < k \leq n, o \leq i < n \end{cases}$$

Обратный ход метода Гаусса

После приведения матрицы коэффициентов к треугольному виду становится возможным определение значений неизвестных:

— Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_n.

— Из предпоследнего уравнения становится возможным определение переменной x_{n-1}, и т. д.

В общем виде, выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$\begin{cases} x_n = b_n/a_{nn}, \\ x_i = (b_i - \sum_{j=i+1}^{n} a_{ij}x_j)/a_{ii}, i = n-1, n-2, \ldots, 0. \end{cases}$$

Выбор ведущего элемента

Описанный алгоритм применим, только если ведущие элементы отличны от нуля, т. е. $a_{ii} \neq 0$.

— Рассмотрим k-й шаг алгоритма. Пусть $s = max|a_{kk}|, |a_{k+1k}|, \ldots, |a_{nk}|$

— Тогда переставим s-ю и k-ю строки матрицы (выбор ведущего элемента по столбцу).

— В итоге получаем систему $PAx = Pb$, где P — матрица перестановки.

Параллельная реализация

Фрагмент кода решения приведен ниже:

```
// SerialGauss.cpp
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <omp.h>

int* pPivotPos; // The Number of pivot rows selected at the iterations
int* pPivotIter; // The Iterations, at which the rows were pivots

typedef struct {
	int PivotRow;
	double MaxValue;
} TThreadPivotRow;

// Function for simple initialization of the matrix
// and the vector elements
void DummyDataInitialization(double* pMatrix, double* pVector, int Size) {
	for (int i = 0; i < Size; ++i) {
		pVector[i] = i + 1.0;
		for (int j = 0; j < Size; ++j)
			if (j <= i)
				pMatrix[i * Size + j] = 1;
			else
				pMatrix[i * Size + j] = 0;
	}
}

// Function for random initialization of the matrix
// and the vector elements
void RandomDataInitialization(double* pMatrix, double* pVector, int Size) {
	srand(unsigned(clock()));
	for (int i = 0; i < Size; ++i) {
		pVector[i] = rand() / double(1000);
```

```
            for (int j = 0; j < Size; ++j)
                if (j <= i)
                    pMatrix[i * Size + j] = rand() / double(1000);
                else
                    pMatrix[i * Size + j] = 0;
    }
}

void MyDataInitialization(double* pMatrix, double* pVector, int Size) {
    pMatrix[0] = 1; pMatrix[1] = 1; pMatrix[2] = 4; pMatrix[3] = 4; pMatrix[4] = 9; pVector[0] = -9;
    pMatrix[5] = 2; pMatrix[6] = 2; pMatrix[7] = 17; pMatrix[8] = 17; pMatrix[9] = 82; pVector[1] = -146;
    pMatrix[10] = 2; pMatrix[11] = 0; pMatrix[12] = 3; pMatrix[13] = -1; pMatrix[14] = 4; pVector[2] = -10;
    pMatrix[15] = 0; pMatrix[16] = 1; pMatrix[17] = 4; pMatrix[18] = 12; pMatrix[19] = 27; pVector[3] =
    ↪ -26;
    pMatrix[20] = 1; pMatrix[21] = 2; pMatrix[22] = 2; pMatrix[23] = 10; pMatrix[24] = 0; pVector[4] = 37;
}

// Function for memory allocation and definition of the objectselements
void ProcessInitialization(double*& pMatrix, double*& pVector, double*& pResult, int& Size) {
    // Setting the size of the matrix and the vector
    do {
        printf("\nEnter size of the matrix and the vector: ");
        scanf_s("%d", &Size);
        printf("\nChosen size = %d \n", Size);
        if (Size <= 0)
            printf("\nSize of objects must be greater than 0!\n");
    } while (Size <= 0);

    Size = 5; // Only for my data

    // Memory allocation
    pMatrix = new double[Size * Size];
    pVector = new double[Size];
    pResult = new double[Size];

    // Initialization of the matrix and the vector elements
    //DummyDataInitialization(pMatrix, pVector, Size);
    //RandomDataInitialization(pMatrix, pVector, Size);
    MyDataInitialization(pMatrix, pVector, Size);
}

// Function for formatted matrix output
void PrintMatrix(double* pMatrix, int RowCount, int ColCount) {
    for (int i = 0; i < RowCount; ++i) {
        for (int j = 0; j < ColCount; ++j)
            printf("%7.4f ", pMatrix[i * RowCount + j]);
        printf("\n");
    }
}
```

```
// Function for formatted vector output
void PrintVector(double* pVector, int Size) {
    for (int i = 0; i < Size; ++i)
        printf("%7.4f ", pVector[i]);
}

// Finding the pivot row
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
    int PivotRow = -1; // The index of the pivot row
    int MaxValue = 0; // The value of the pivot element

    #pragma omp parallel
    {
        TThreadPivotRow ThreadPivotRow;
        ThreadPivotRow.MaxValue = 0;
        ThreadPivotRow.PivotRow = -1;
        // Choose the row, that stores the maximum element
        #pragma omp for
        for (int i = 0; i < Size; ++i) {
            if ((pPivotIter[i] == -1) && (fabs(pMatrix[i * Size + Iter]) > ThreadPivotRow.MaxValue)) {
                ThreadPivotRow.PivotRow = i;
                ThreadPivotRow.MaxValue = fabs(pMatrix[i * Size + Iter]);
            }
        }

        #pragma omp critical
        {
            if (ThreadPivotRow.MaxValue > MaxValue) {
                MaxValue = ThreadPivotRow.MaxValue;
                PivotRow = ThreadPivotRow.PivotRow;
            }
        }
    }
    return PivotRow;
}

// Column elimination
void ParallelColumnElimination(double* pMatrix, double* pVector, int Pivot, int Iter, int Size) {
    double PivotValue, PivotFactor;
    PivotValue = pMatrix[Pivot * Size + Iter];
    #pragma omp parallel for private(PivotFactor) schedule(dynamic, 1)
    for (int i = 0; i < Size; ++i)
        if (pPivotIter[i] == -1) {
            PivotFactor = pMatrix[i * Size + Iter] / PivotValue;
            for (int j = Iter; j < Size; ++j)
                pMatrix[i * Size + j] -= PivotFactor * pMatrix[Pivot * Size + j];
            pVector[i] -= PivotFactor * pVector[Pivot];
        }
}
```

```
// Gaussian elimination
void ParallelGaussianElimination(double* pMatrix, double* pVector, int Size) {
    int PivotRow; // The number of the current pivot row
    for (int Iter = 0; Iter < Size; ++Iter) {
        // Finding the pivot row
        PivotRow = ParallelFindPivotRow(pMatrix, Size, Iter);
        pPivotPos[Iter] = PivotRow;
        pPivotIter[PivotRow] = Iter;
        ParallelColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
    }
}

// Back substution
void ParallelBackSubstitution(double* pMatrix, double* pVector, double* pResult, int Size) {
    int RowIndex, Row;
    for (int i = Size - 1; i >= 0; --i) {
        RowIndex = pPivotPos[i];
        pResult[i] = pVector[RowIndex] / pMatrix[Size * RowIndex + i];
        #pragma omp parallel for private(Row)
        for (int j = 0; j < i; ++j) {
            Row = pPivotPos[j];
            pVector[Row] -= pMatrix[Row * Size + i] * pResult[i];
            pMatrix[Row * Size + i] = 0;
        }
    }
}

// Function for the execution of Gauss algorithm
void ParallelResultCalculation(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    // Memory allocation
    pPivotPos = new int[Size];
    pPivotIter = new int[Size];

    for (int i = 0; i < Size; pPivotIter[i] = -1, ++i);

    // Gaussian elimination
    ParallelGaussianElimination(pMatrix, pVector, Size);
    // Back substitution
    ParallelBackSubstitution(pMatrix, pVector, pResult, Size);

    // Memory deallocation
    delete[] pPivotPos;
    delete[] pPivotIter;
}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector, double*
    pResult) {
```

```
    delete[] pMatrix;
    delete[] pVector;
    delete[] pResult;
}

// Function for testing the result
void TestResult(double* pMatrix, double* pVector, double* pResult, int Size) {
    /* Buffer for storing the vector, that is a result of multiplication
    of the linear system matrix by the vector of unknowns */
    double* pRightPartVector;

    // Flag, that shows wheather the right parts vectors are identical or not
    int equal = 0;
    double Accuracy = 1e-2; // Comparison accuracy
    pRightPartVector = new double[Size];
    for (int i = 0; i < Size; ++i) {
        pRightPartVector[i] = 0;
        for (int j = 0; j < Size; ++j)
            pRightPartVector[i] += pMatrix[i * Size + j] * pResult[j];
    }

    for (int i = 0; i < Size; i++)
        if (fabs(pRightPartVector[i] - pVector[i]) > Accuracy)
            equal = 1;

    if (equal == 1)
        printf("\nThe result of the parallel Gauss algorithm is NOT correct. Check your code.");
    else
        printf("The result of the parallel Gauss algorithm is correct.");

    delete[] pRightPartVector;
}
int main() {
    double* pMatrix; // The matrix of the linear system
    double* pVector; // The right parts of the linear system
    double* pResult; // The result vector
    int Size; // The sizes of the initial matrix and the vector
    double start, finish, duration;
    printf("Parallel Gauss algorithm for solving linear systems\n");

    // Memory allocation and definition of objects' elements
    ProcessInitialization(pMatrix, pVector, pResult, Size);

    // The matrix and the vector output
    printf("Initial Matrix \n");
    PrintMatrix(pMatrix, Size, Size);
    printf("Initial Vector \n");
    PrintVector(pVector, Size);

    // Execution of Gauss algorithm
```

```
    start = clock();
    ParallelResultCalculation(pMatrix, pVector, pResult, Size);
    finish = clock();
    duration = (finish - start) / CLOCKS_PER_SEC;

    // Testing the result
    TestResult(pMatrix, pVector, pResult, Size);

    // Printing the result vector
    printf("\n Result Vector: \n");
    PrintVector(pResult, Size);

    // Printing the execution time of Gauss method
    printf("\nTime of execution: %f\n", duration);

    // Computational process termination
    ProcessTermination(pMatrix, pVector, pResult);

    return 0;
}
```

Результат работы

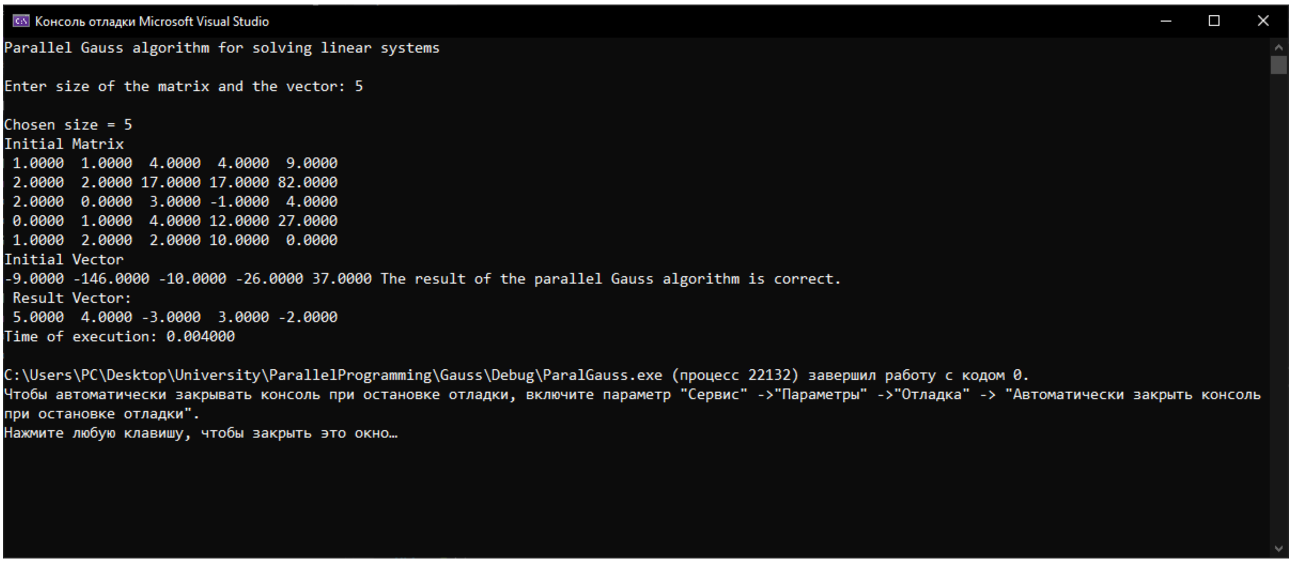

Рисунок 1 – Work-5

Характеристики устройства

Процессор: Intel(R) Core(TM) i5-10400F

Ядер: 6

Оперативная память: 16 Гб