

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

Кафедра математической кибернетики и компьютерных наук

ОТЧЕТ ПО ПРАКТИКЕ WORK4

ОТЧЕТ

Студента 3 курса 311 группы
направления 02.03.02 — Фундаментальная информатика и информационные
технологии
факультета КНиИТ
Забоева Максима Владиславовича

Проверил

Старший преподаватель

М. С. Портенко

Саратов 2023

СОДЕРЖАНИЕ

1	Условие задачи	3
2	Практическая часть	4
2.1	Последовательная реализация метода Гаусса	4
2.2	Параллельная реализация метода Гаусса	7
3	Результат запусков	12
3.1	Характеристики компьютера	12
3.2	Таблица результатов	12
4	Тестовые запуски	13
4.1	Последовательная реализация	13
4.2	Параллельная реализация	13

1 Условие задачи

Задайте элементы больших матриц и векторов при помощи датчика случайных чисел. Отключите печать исходных матрицы и вектора и печать результирующего вектора (закомментируйте соответствующие строки кода). Проведите вычислительные эксперименты, результаты занесите в таблицу 1.

Таблица 1. Время выполнения последовательного и параллельного алгоритмов Гаусса решения систем линейных уравнений и ускорение

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм	
			Время	Ускорение
1	10			
2	100			
3	500			
4	1000			
5	1500			
6	2000			
7	2500			
8	3000			

Рисунок 1 – Время выполнения последовательного и параллельного алгоритмов Гаусса решения систем линейных уравнений и ускорение

2 Практическая часть

2.1 Последовательная реализация метода Гаусса

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

Основной идеей метода Гаусса является приведение матрицы 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 x_n , после этого из предпоследнего уравнения становится возможным определение переменной x_{n-1} и т.д.

Код программы:

```
#include "omp.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>

int* pSerialPivotPos; // The Number of pivot rows selected at the
// iterations
int* pSerialPivotIter; // The Iterations, at which the rows were pivots
// Function for simple initialization of the matrix
// and the vector elements
void DummyDataInitialization(double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables
    for (i = 0; i < Size; i++) {
        pVector[i] = i + 1;
        for (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) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i = 0; i < Size; i++) {
        pVector[i] = rand() / double(1000);
        for (j = 0; j < Size; j++) {
            if (j <= i)
                pMatrix[i * Size + j] = rand() / double(1000);
            else
                pMatrix[i * Size + j] = 0;
        }
    }
}

// Function for memory allocation and definition of the objects elements
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);
    // 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);
}

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

// Finding the pivot row
int FindPivotRow(double* pMatrix, int Size, int Iter) {
    int PivotRow = -1; // The index of the pivot row
    int MaxValue = 0; // The value of the pivot element
    int i; // Loop variable
    // Choose the row, that stores the maximum element
    for (i = 0; i < Size; i++) {
        if ((pSerialPivotIter[i] == -1) &&
            (fabs(pMatrix[i * Size + Iter]) > MaxValue)) {
            PivotRow = i;
            MaxValue = fabs(pMatrix[i * Size + Iter]);
        }
    }
}

```

```

        }
    }
    return PivotRow;
}

// Column elimination
void SerialColumnElimination(double* pMatrix, double* pVector,
    int Pivot, int Iter, int Size) {
    double PivotValue, PivotFactor;
    PivotValue = pMatrix[Pivot * Size + Iter];
    for (int i = 0; i < Size; i++) {
        if (pSerialPivotIter[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 SerialGaussianElimination(double* pMatrix, double* pVector, int
    Size) {
    int Iter; // The number of the iteration of the Gaussian
    // elimination
    int PivotRow; // The number of the current pivot row
    for (Iter = 0; Iter < Size; Iter++) {
        // Finding the pivot row
        PivotRow = FindPivotRow(pMatrix, Size, Iter);
        pSerialPivotPos[Iter] = PivotRow;
        pSerialPivotIter[PivotRow] = Iter;
        SerialColumnElimination(pMatrix, pVector, PivotRow, Iter, Size);
    }
}

// Back substitution
void SerialBackSubstitution(double* pMatrix, double* pVector,
    double* pResult, int Size) {
    int RowIndex, Row;
    for (int i = Size - 1; i >= 0; i--) {
        RowIndex = pSerialPivotPos[i];
        pResult[i] = pVector[RowIndex] / pMatrix[Size * RowIndex + i];
        for (int j = 0; j < i; j++) {
            Row = pSerialPivotPos[j];
            pVector[Row] -= pMatrix[Row * Size + i] * pResult[i];
            pMatrix[Row * Size + i] = 0;
        }
    }
}

// Function for the execution of Gauss algorithm
void SerialResultCalculation(double* pMatrix, double* pVector, double* pResult, int Size) {
    // Memory allocation
    pSerialPivotPos = new int[Size];
    pSerialPivotIter = new int[Size];
    for (int i = 0; i < Size; i++) {
        pSerialPivotIter[i] = -1;
    }
    // Gaussian elimination
    SerialGaussianElimination(pMatrix, pVector, Size);
    // Back substitution
    SerialBackSubstitution(pMatrix, pVector, pResult, Size);
    // Memory deallocation
    delete[] pSerialPivotPos;
}

```

```

        delete[] pSerialPivotIter;
    }
    // Function for computational process termination
    void ProcessTermination(double* pMatrix, double* pVector, double*
        pResult) {
        delete[] pMatrix;
        delete[] pVector;
        delete[] pResult;
    }
    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("Serial 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();
        SerialResultCalculation(pMatrix, pVector, pResult, Size);
        finish = clock();
        duration = (finish - start) / CLOCKS_PER_SEC;
        // Printing the result vector
        //printf("\n Result Vector: \n");
        PrintVector(pResult, Size);
        // Printing the execution time of Gauss method
        printf("\n Time of execution: %f\n", duration);
        // Computational process termination
        ProcessTermination(pMatrix, pVector, pResult);
        return 0;
    }
}

```

2.2 Параллельная реализация метода Гаусса

Код программы:

```

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <math.h>
#include <iostream>
#include <omp.h>
using namespace std;
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;
// Finding the pivot row
int ParallelFindPivotRow(double* pMatrix, int Size, int Iter) {
    int PivotRow = -1; // The index of the pivot row

```

```

    double MaxValue = 0; // The value of the pivot element
    int i; // Loop variable
    // Choose the row, that stores the maximum element
#pragma omp parallel
    {
        TThreadPivotRow ThreadPivotRow;
        ThreadPivotRow.MaxValue = 0;
        ThreadPivotRow.PivotRow = -1;
#pragma omp for
        for (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 Iter; // The number of the iteration of the Gaussian
    // elimination
    int PivotRow; // The number of the current pivot row
    for (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);
    }
}

void DummyDataInitialization(double* pMatrix, double* pVector, int Size) {
    int i, j; // Loop variables
    for (i = 0; i < Size; i++) {
        pVector[i] = i + 1;
    }
}

```



```

        for (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) {
    int i, j; // Loop variables
    srand(unsigned(clock()));
    for (i = 0; i < Size; i++) {
        pVector[i] = rand() / double(1000);
        for (j = 0; j < Size; j++) {
            if (j <= i)
                pMatrix[i * Size + j] = rand() / double(1000);
            else
                pMatrix[i * Size + j] = 0;
        }
    }
}

// Function for memory allocation and definition of the objects elements
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);
    // Memory allocation
    pMatrix = new double[Size * Size];
    pVector = new double[Size];
    pResult = new double[Size];
    // Initialization of the matrix and the vector elements
    RandomDataInitialization(pMatrix, pVector, Size);
    //RandomDataInitialization(pMatrix, pVector, Size);
}

// Function for computational process termination
void ProcessTermination(double* pMatrix, double* pVector, double*
    pResult) {
    delete[] pMatrix;
    delete[] pVector;
    delete[] pResult;
}

// Back substitution
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; i++) {
        pPivotIter[i] = -1;
    }
    ParallelGaussianElimination(pMatrix, pVector, Size);
    ParallelBackSubstitution(pMatrix, pVector, pResult, Size);
    // Memory deallocation
    delete[] pPivotPos;
    delete[] pPivotIter;
}

// 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 = 1.e-6; // 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("The 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 size of the matrix and the vectors
    double start, finish, duration;
    // Data initialization
    ProcessInitialization(pMatrix, pVector, pResult, Size);
    start = omp_get_wtime();
    ParallelResultCalculation(pMatrix, pVector, pResult, Size);
    finish = omp_get_wtime();
    duration = finish - start;
    // Testing the result
    TestResult(pMatrix, pVector, pResult, Size);
}

```

```
    // Printing the time spent by parallel Gauss algorithm
    printf("\n Time of execution: %f\n", duration);
    // Program termination
    ProcessTermination(pMatrix, pVector, pResult);
    return 0;
}
```

3 Результат запусков

3.1 Характеристики компьютера

Процессор — 12th Gen Intel Core i5-12600KF, Базовая скорость 3,70ГГц,
Кол-во ядер 10, Кол-во процессоров 16 (включая 4 энергоэффективных ядра).
16гб Оперативной памяти, скорость 3200МГц

3.2 Таблица результатов

Номер теста	Порядок системы	Последовательный алгоритм	Параллельный алгоритм	
			Время	Ускорение
1	10	0.000000	0.007414	0
2	100	0.001000	0.011167	0,08954956568
3	500	0.083000	0.069435	1,195362569
4	1000	0.610000	0.185726	3,284408214
5	1500	1.958000	0.369328	5,3015206
6	2000	4.610000	0.814302	5,661290283
7	2500	8.841000	1.671254	5,290039695
8	3000	15.246000	3.190156	4,779076634

4 Тестовые запуски

4.1 Последовательная реализация

```
Консоль отладки Microsoft Visual Studio
Serial Gauss algorithm for solving linear systems
Enter size of the matrix and the vector: 100
Chosen size = 100
Time of execution: 0.002000
```

```
Консоль отладки Microsoft Visual Studio
Serial Gauss algorithm for solving linear systems
Enter size of the matrix and the vector: 1000
Chosen size = 1000
Time of execution: 0.589000
E:\games\tests\Parallel\x64\Debug\Parallel.exe (проц
Чтобы автоматически закрывать консоль при остановке
```

```
Консоль отладки Microsoft Visual Studio
Serial Gauss algorithm for solving linear systems
Enter size of the matrix and the vector: 2000
Chosen size = 2000
Time of execution: 4.652000
```

```
Консоль отладки Microsoft Visual Studio
Serial Gauss algorithm for solving linear systems
Enter size of the matrix and the vector: 3000
Chosen size = 3000
Time of execution: 15.873000
```

4.2 Параллельная реализация

```
Enter size of the matrix and the vector: 100
Chosen size = 100
The result of the parallel Gauss algorithm is NOT correct.Check your code.
Time of execution: 0.009806
```

```
Enter size of the matrix and the vector: 1000
Chosen size = 1000
The result of the parallel Gauss algorithm is NOT correct.Check your code.
Time of execution: 0.163259
```

```
Enter size of the matrix and the vector: 2000
```

```
Chosen size = 2000
```

```
The result of the parallel Gauss algorithm is NOT correct.Check your code.
```

```
Time of execution: 0.820508
```

```
Enter size of the matrix and the vector: 3000
```

```
Chosen size = 3000
```

```
The result of the parallel Gauss algorithm is NOT correct.Check your code.
```

```
Time of execution: 3.276685
```