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 WORK5

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

Решите систему линейных уравнений согласно варианту параллельным методом Гаусса.

2.

$$x_1+2x_2+3x_3+4x_4+5x_5=2,$$
$$2x_1+3x_2+7x_3+10x_4+13x_5=12,$$
$$3x_1+5x_2+11x_3+16x_4+21x_5=17,$$
$$2x_1-7x_2+7x_3+7x_4+2x_5=57,$$
$$x_1+4x_2+5x_3+3x_4+10x_5=7.$$

2 Практическая часть

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]);
			}
		}
		//printf("Local thread (id = %i) pivot row : %i\n", omp_get_thread_num(), ThreadPivotRow.PivotRow);
#pragma omp critical
		{
			if (ThreadPivotRow.MaxValue > MaxValue) {
				MaxValue = ThreadPivotRow.MaxValue;
				PivotRow = ThreadPivotRow.PivotRow;
			}
		} // pragma omp critical
	}// pragma omp parallel
	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) {
        pMatrix[0] = 1;
        pMatrix[1] = 2;
        pMatrix[2] = 3;
        pMatrix[3] = 4;
        pMatrix[4] = 5;
        pMatrix[5] = 2;
        pMatrix[6] = 3;
        pMatrix[7] = 7;
        pMatrix[8] = 10;
        pMatrix[9] = 13;
        pMatrix[10] = 3;
        pMatrix[11] = 5;
        pMatrix[12] = 11;
        pMatrix[13] = 16;
        pMatrix[14] = 21;
        pMatrix[15] = 2;
        pMatrix[16] = -7;
        pMatrix[17] = 7;
        pMatrix[18] = 7;
        pMatrix[19] = 2;
        pMatrix[20] = 1;
        pMatrix[21] = 4;
        pMatrix[22] = 5;
        pMatrix[23] = 3;
        pMatrix[24] = 10;

        pVector[0] = 2;
        pVector[1] = 12;
        pVector[2] = 17;
        pVector[3] = 57;
        pVector[4] = 7;
}
// 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 computational process termination
void ProcessTermination(double* pMatrix, double* pVector, double*
        pResult) {
        delete[] pMatrix;
        delete[] pVector;
        delete[] pResult;
}

// Back substation
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;
}
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]);
}
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();
```

```
        printf("Initial Matrix \n");
        PrintMatrix(pMatrix, Size, Size);
        printf("Initial Vector \n");
        PrintVector(pVector, Size);
        ParallelResultCalculation(pMatrix, pVector, pResult, Size);
        finish = omp_get_wtime();
        duration = finish - start;
        // Testing the result
        //TestResult(pMatrix, pVector, pResult, Size);
        printf("\nResult\n");
        PrintVector(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 Тестовые запуски

```
Initial Matrix
 1.0000  2.0000  3.0000  4.0000  5.0000
 2.0000  3.0000  7.0000 10.0000 13.0000
 3.0000  5.0000 11.0000 16.0000 21.0000
 2.0000 -7.0000  7.0000  7.0000  2.0000
 1.0000  4.0000  5.0000  3.0000 10.0000
Initial Vector
 2.0000 12.0000 17.0000 57.0000  7.0000
Result
 3.0000 -5.0000  4.0000 -2.0000  1.0000
 Time of execution: 0.003716
```