Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт вычислительной математики и информационных технологий

Кафедра прикладной математики

Направление подготовки: 01.03.02 – Прикладная математика и информатика

КУРСОВАЯ РАБОТА ПО ДИСЦИПЛИНЕ
«ОПЕРАЦИОННЫЕ СИСТЕМЫ»

Студент группы 09-812
«____» _____2020 г. подпись А.Н. Васильев

Научный руководитель
ассистент кафедры
прикладной математики
"___"_____ 2020 г. подпись Д.Х. Гиниятова

Казань 2020

Содержание

Постановка задачи

Дана система n линейных алгебраических уравнений (СЛАУ):

$$A\vec{x}=\vec{b},$$

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix},\ \vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix},\ \vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad (1)$$

где A – невырожденная матрица коэффициентов с диагональным преобладанием размерности $n \times n$, $\vec{b}$– вектор свободных членов размерности n, $\vec{x}$ – неизвестный вектор размерности n

или

$$\begin{cases} a_{11}x_1 + \cdots + a_{11}x_n = b_1, \\ \cdots \\ a_{n1}x_1 + \cdots + a_{nn}x_n = b_n \end{cases}$$

и дано начальное приближение $x^0 = \{x_1^0, x_2^0, \cdots, x_n^0\}$. Условие существования единственного решения системы (1) считается выполненным. Элементы матрицы и компоненты векторов являются вещественными числами.

Найти решение системы (1) методом нижней релаксации с некоторой заданной точностью ε. Реализовать последовательный и параллельный алгоритмы решения СЛАУ методом нижней релаксации на языке C++. Для параллельной реализации использовать технологию OpenMP. Выполнить сравнение скорости вычисления последовательной и параллельной программ, провести анализ результатов счета и сформулировать выводы.

Описание метода решения

Метод нижней релаксации – одна из разновидностей метода Гаусса-Зейделя для решения СЛАУ, который приводит к более быстрой сходимости за счёт итерационного параметра ω.

Рассмотрим систему *(1)*. Мы можем разбить матрицу A на несколько компонент: диагональная матрица D, нижняя треугольная и верхняя треугольная матрицы L и U. Тогда A можно представить в виде: $A = L + D + U$, где

$$L = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}, \quad D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix},$$

$$U = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

В результате, СЛАУ может быть переписана в таком виде:

$$(D + \omega L)x = \omega b - [\omega U + (\omega - 1)D]x,$$

где число ω – релаксационный параметр, который может иметь значения $0 < \omega < 1$.

Этот метод будет сходиться при некоторых условиях, а именно: матрица A обладает диагональным преобладанием, отмечается скорость повышения сходимости, если матрица будет симметрична и положительно определена.

Метод нижней релаксации, как известно, итерационный метод, который находит значение x в левой части, используя прошлое значение x в правой части.

$$x^{k+1} = (D + \omega L)^{-1}(\omega b - [\omega U + (\omega - 1)D]x^k) = L_\omega x^k + C,$$

где x^k – это k-тое приближение x. Отсюда очевидно следует, что каждый элемент вектора x может быть вычислен по формуле:

$$x_i^{k+1} = (1 - \omega)x_i^k + \frac{\omega}{a_{ii}}(b_i - \sum_{j<i} a_{ii}x_j^{k+1} - \sum_{j>i} a_{ii}x_j^k),\ i = 1,2,\dots,n \quad (2)$$

Последовательная реализация метода

- Для данного метода используется функция *double* LOW_SOR_NP*, в начале которой создается вектор *phi*. Его нужно инициализировать значением вектора *initial*. Именно он будет служить решением СЛАУ.
- Чтобы записать время начала работы нужно переменной *start* присвоить результат работы функции *omp_get_wtime()*. Сразу после мы используем функцию *void recal_correct_nparralel* чтобы вычислить точность текущего решения и запоминаем значение в переменной *ostatok*. Для того чтобы подсчитать точность решения используется норма Фробениуса.

$$\|A * phi - b\| = \left[\sum_i (A[i] * phi - b[i])^2\right]^{1/2}$$

- Затем следует цикл, работающий до тех пор, пока текущая точность больше требуемой. В нём для каждого элемента вектора *phi* применяется формула Фробениуса *(2)*, сразу после вычисляется точность и проверяется условие цикла. В конце работы цикла в переменную *end* фиксируется время с помощью функции *omp_get_wtime()*. А время работы *time* находим как разность переменных *end* и *start.*
- Возвращается вектор *phi*.

Параллельная реализация метода

- В этом методе для вычисления нижней релаксации используется функция *double* LOW_SOR.* По смыслу реализации функций *void recal_correct* и *recal_correct_nparralel* не отличаются. Но в последняя использует параллельные вычисления. Стоит отметить, что здесь не очень много места для распараллеливания.
- Нельзя этого сделать в цикле проверки точности, в цикле применения формулы *(2)*, потому что это итерационный метод, а в местах нахождения сумм, распараллеливание процессов приводит к повышению времени работы.

Результаты вычислений

Работа выполнялась на восьмиядерном процессоре Intel Core i5-9300H CPU, 2.40GHz, в расположении которого имеется 8 потоков.

Вычисления производились при $\omega = 0.9$. Эта величина была найдена при помощи функции *omega_prov()*.

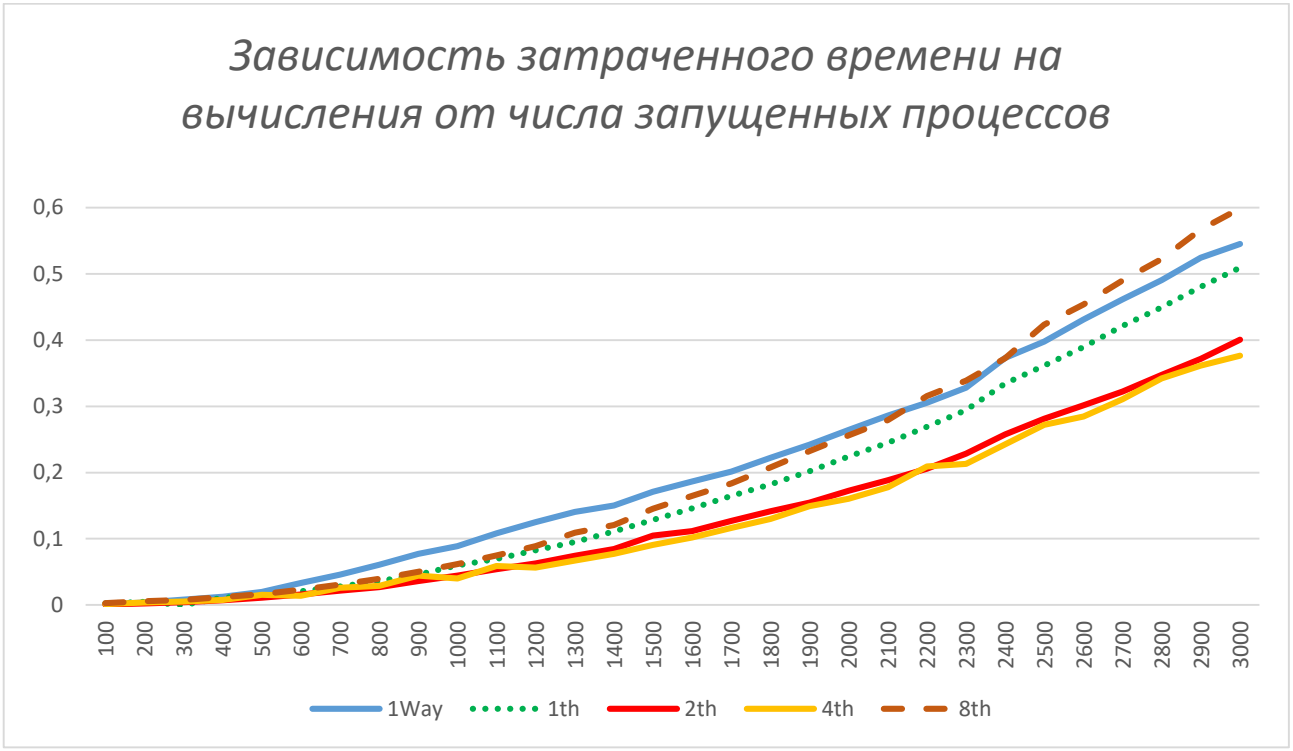

Рис. 1 – Зависимость затраченного времени на вычисления от числа запущенных процессов

По графику видно, что время работы последовательной программы почти такое же как у параллельной, однако в дальнейшем это время начинает расти. Целесообразным становится применять параллельный алгоритм. Как следствие производительность увеличивается. Работу двухпоточных и четырехпоточных программ можно назвать эквивалентными. Акцентирую внимание на том, что при дальнейшем увеличении числа потоков скорость работы падает и смысл использовать большее число нитей падает.

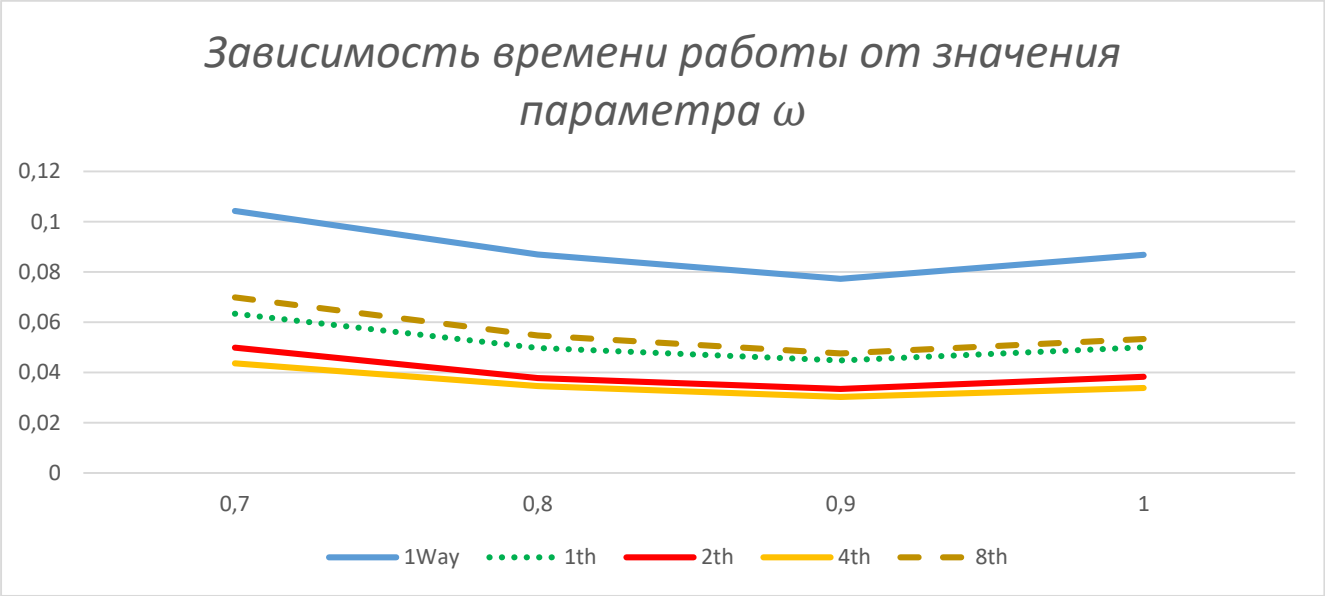

Рис. 2 – Зависимость времени работы от значения параметра ω

На данном графике изображена зависимость времени работы от значений параметра ω для последовательной и параллельной реализации с уже известными потоками. Матрица A была фиксированного размера – 1000. Чтобы получить боле точные значения релаксационного параметра шаг установили равным 0.01. По графику точно видно, что минимум достигается при $\omega = 0.9$.

Заключение

В рамках курсовой работы были изучены теоретические основы методов решения СЛАУ, а также параллельного программирования OpenMP. Были реализованы последовательная и параллельная версии метода нижней релаксации. В ходе экспериментов был установлен наиболее оптимальный параметр релаксации. Результаты скорости вычисления говорят о том, что при больших размерностях СЛАУ имеет смысл распараллеливать программу, однако отметим, что целесообразность такого использования зависит от типов задач, специфичности алгоритмов и технологий.

Список использованных источников

1. Глазырина Л.Л, Карчевский М.М Введение в численные методы. - Казань: Казан. ун-т, 2017. - 122 с.

2. Successive over-relaxation // en.wikipedia.org URL: https://en.wikipedia.org/wiki/Successive_over-relaxation

3. Антонов А.С. Параллельное программирование с использование технологии OpenMP. - М: изд-во МГУ, 2009. - 77 с.

Листинг программы

Для генерации матриц и векторов использую файл LinalGenerator.h

```
#pragma once
#ifndef LINAL_GENERATOR_H
#define LINAL_GENERATOR_H
#include <vector>
#include <random>

std::random_device rd;
std::mt19937 generator(rd());



// отрезок распределения -- опционален
const int a = 1;
const int b = 100;



template<typename T>
std::vector<T> generateVector(const uint32_t n)
{
    std::vector<T> vec(n);
    std::uniform_int_distribution<int> uniDistribution(a, b);

    for (size_t i = 0; i < vec.size(); ++i) {
        vec[i] = uniDistribution(generator);
    }
```

```
    return vec;
}

template<typename T>
std::vector<std::vector<T>> generateGoodConditionedMatrix(const uint32_t n)
{
    std::vector<std::vector<T>> matrix(n, std::vector<T>(n));
    std::uniform_int_distribution<int> uniDistribution(a, b);

    for (size_t i = 0; i < matrix.size(); ++i) {
        int sum = 0;
        for (size_t j = 0; j < matrix[i].size(); ++j) {
            if (i != j) {
                matrix[i][j] = uniDistribution(generator);
                sum += matrix[i][j];
            }
        }
        if (sum < b) {
            uniDistribution.param(std::uniform_int_distribution<int>::param_type(b - sum +
1, b));
            matrix[i][i] = uniDistribution(generator);
            uniDistribution.param(std::uniform_int_distribution<int>::param_type(a, b));
        }
        else {
            matrix[i][i] = sum + 1;
        }
    }

    return matrix;
```

```
}


#endif // LINAL_GENERATOR_H
```

Main.cpp

```
#include <iostream>
#include <fstream>
#include <omp.h>
#include <stdlib.h>
#include <time.h>
#include "LinalGenerator.h"
#include <vector>
using namespace std;

double** create_matr(int size)
{
	vector<vector<double>> generated = generateGoodConditionedMatrix<double>(size);
	double** matr = new double * [size];
	for (int i = 0; i < size; i++)
		matr[i] = new double[size];
	for (int i = 0; i < size; i++)
		for (int j = 0; j < size; j++)
			matr[i][j] = generated[i][j];
	return matr;
}

double* create_rand(int size)
{
	double* vector = new double[size];
```

```
	for (int i = 0; i < size; i++)
		vector[i] = double(rand() % size) + 1;
	return vector;
}

double* create_zero(int size)
{
	double* vector = new double[size];
	for (int i = 0; i < size; i++)
		vector[i] = 0;
	return vector;
}

bool check(double* a, double* b, int size, double eps)
{
	for (int i = 0; i < size; i++)
		if (abs(a[i] - b[i]) > eps)
			return false;
	return true;
}

double* mult(double** matr, double* v, int size)
{
	double* res = new double[size];
	double tmp = 0;
#pragma omp parallel for private(tmp)
	for (int i = 0; i < size; i++)
	{
		tmp = 0;
```

```
            for (int j = 0; j < size; j++)
                tmp += v[j] * matr[i][j];
            res[i] = tmp;
        }
        return res;
}

double sigma, ostatok;

void recal_correct_nparralel(double** A, double* phi, double* b, int size)
{
        double tmp;
        ostatok = 0;
        for (int i = 0; i < size; i++)
        {
            tmp = -b[i];
            for (int j = 0; j < size; j++)
                tmp += A[i][j] * phi[j];
            ostatok += tmp * tmp;
        }
        ostatok = sqrt(ostatok);
}

double* LOW_SOR_NP(double** A, double* b, int size, double omega, double*
initial, double crit_shod, double& time)
{
        double* phi = new double[size];
        for (int i = 0; i < size; i++) phi[i] = initial[i];
        double start = omp_get_wtime();
```

```
	recal_correct_nparralel(A, phi, b, size);
	while (ostatok > crit_shod)
	{
		for (int i = 0; i < size; i++)
		{
			sigma = -(A[i][i] * phi[i]);
			for (int j = 0; j < size; j++)
			{
				sigma += A[i][j] * phi[j];
			}
			phi[i] = (1 - omega) * phi[i] + (omega / A[i][i]) * (b[i] - sigma);
		}
		recal_correct_nparralel(A, phi, b, size);
	}
	double end = omp_get_wtime();
	time = end - start;
	return phi;
}

void recal_correct(double** A, double* phi, double* b, int size, int number_threads)
{
	double tmp;
	ostatok = 0;
	omp_set_num_threads(number_threads);
#pragma omp parallel for private(tmp) shared(ostatok) schedule(dynamic)
	for (int i = 0; i < size; i++)
	{
		tmp = -b[i];
		for (int j = 0; j < size; j++)
```

```
                tmp += A[i][j] * phi[j];
#pragma     omp critical
                ostatok += tmp * tmp;
        }
        ostatok = sqrt(ostatok);
}

double* LOW_SOR(double** A, double* b, int size, double omega, double* initial,
double crit_shod, int number_threads, double& time)
{
        double* phi = new double[size];
        for (int i = 0; i < size; i++) phi[i] = initial[i];
        double start = omp_get_wtime();
        recal_correct(A, phi, b, size, number_threads);
        while (ostatok > crit_shod)
        {
                for (int i = 0; i < size; i++)
                {
                        sigma = -(A[i][i] * phi[i]);
                        for (int j = 0; j < size; j++)
                        {
                                sigma += A[i][j] * phi[j];
                        }
                        phi[i] = (1 - omega) * phi[i] + (omega / A[i][i]) * (b[i] - sigma);
                }
                recal_correct(A, phi, b, size, number_threads);
        }
        double end = omp_get_wtime();
        time = end - start;
```

```
	return phi;
}

void prov()
{
	srand(time(NULL));
	int size = 3000;
	double omega = 0.8;
	double crit_shod = 0.001;
	double** A = create_matr(size);
	double* x = create_rand(size);
	double* initial = create_zero(size);

	fstream fout;
	fout.open("otchet.csv", ios::out |ios::app);
	fout << "Shape, OneWay, 1 thread,2 threads,4 threads,8 threads\n";
	double time = 0;

	for (int i = 100; i <= size; i += 100)
	{
		double* b = mult(A, x, i);
		fout << i;

		double* res_np =LOW_SOR_NP(A, b, i,
			omega, initial, crit_shod, time);
		if (!check(res_np, x, i, crit_shod))
		{
			cout << "raznie ress!!!\n";
			for (int k = 0; k < i; ++k)
```

```
                cout << res_np[k] << " " << x[k] << endl;
            return;
        }
        fout << ", " << time;
        delete[] res_np;
        for (int j = 1; j < 9; j *= 2)
        {
            double* res = LOW_SOR(A, b, i,
                omega, initial, crit_shod,
                j, time);
            if (!check(res, x, i, crit_shod))
            {
                cout << "raznie ress!!!\n";
                for (int k = 0; k < i; ++k)
                    cout << res[k] << " " << x[k] << endl;
                return;
            }
            fout << ", " << time;
            delete[] res;
        }
        fout << "\n";
        cout << i << endl;
        delete[] b;
    }
    for (int i = 0; i < size; i++) delete[] A[i];
    delete[] A;
    delete[] x;
    delete[] initial;
}
```

```
void omega_prov()
{

	srand(time(NULL));

	int size = 1000;
	double crit_shod = 0.01;
	double** A = create_matr(size);
	double* x = create_rand(size);
	double* initial = create_zero(size);
	double* b = mult(A, x, size);
	fstream fout;



	fout.open("otchet_omega.csv", ios::out | ios::app);
	fout << "Omega, OneWay, 1 thread,2 threads,4 threads,8 threads\n";
	double time = 0;

	for (double omega = 0.1; omega <= 1; omega += 0.1)
	{
		fout << omega;

		double* res_np = LOW_SOR_NP(A, b, size,
			omega, initial, crit_shod, time);
		if (!check(res_np, x, size, crit_shod))
		{
			cout << "raznie ress!!!\n";
			for (int k = 0; k < size; ++k)
```

```
                cout << res_np[k] << " " << x[k] << endl;
            return;
        }
        delete[] res_np;
        fout << ", " << time;

        for (int j = 1; j < 9; j *= 2)
        {
            double* res = LOW_SOR(A, b, size,
                omega, initial, crit_shod,
                j, time);
            if (!check(res, x, size, crit_shod))
            {
                cout << "raznie ress!!!\n";
                for (int k = 0; k < size; ++k)
                    cout << res[k] << " " << x[k] << endl;
                return;
            }
            fout << ", " << time;
            delete[] res;
        }
        fout << "\n";
        cout << omega << endl;
}
for (int i = 0; i < size; i++)
        delete[] A[i];
delete[] A;
delete[] x;
delete[] initial;
```

```
        delete[] b;
}

int main()
{
        prov();
        //omega_prov();
        return 0;
}
```