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 3](#_Toc40200277)

[Описание метода решения 4](#_Toc40200278)

[Результаты вычислений 6](#_Toc40200279)

[Заключение 8](#_Toc40200280)

[Список использованных источников 9](#_Toc40200281)

[Листинг программы 10](#_Toc40200282)

# Постановка задачи

Дана система *n*линейных алгебраических уравнений (СЛАУ):

|  |  |
| --- | --- |
| =, | (1) |

где *A* – невырожденная матрица коэффициентов с диагональным преобладанием размерности , – вектор свободных членов размерности *n*, – неизвестный вектор размерности *n*

или

и дано начальное приближение . 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 С++. 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 .

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: , где

, ,

В результате, СЛАУ может быть переписана в таком виде:

*,*

где число – релаксационный параметр, .

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 в левой части, используя прошлое значение в правой части.

*,*

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

, (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.
* 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 . 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 *А* была фиксированного размера – 1000. Чтобы получить боле точные значения релаксационного параметра шаг установили равным 0.01. По графику точно видно, что минимум достигается при .

# 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;

}