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 09-811

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_2020 г. подпись В.А. Лапинский

Научный руководитель

ассистент кафедры  
прикладной математики

"\_\_\_"\_\_\_\_\_\_\_\_\_ 2020 г. подпись А.Г. Маркина

Казань 2020

**Содержание**

[Постановка задачи 3](#_Toc40200277)

[Описание метода решения 4](#_Toc40200278)

[Результаты вычислений 6](#_Toc40200279)

[Заключение 9](#_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 - это разновидность метода Гаусса-Зейделя для решения СЛАУ, который приводит к более быстрой сходимости.

Дана система *(1)*. Метод будет сходиться при условии, что матрица *A* обладает строгим диагональным преобладание, скорость сходимости повыситься, если матрица в добавок будет симметрична и положительно определена. Разобьем матрицу *A* на несколько компонент: диагональная матрица *D*, строго нижняя треугольная и строго верхняя треугольная матрицы *L* и *U*. Тогда *A* можно представить в виде: , где

, ,

В результате, СЛАУ может быть переписана в виде:

*,*

где число – релаксационный параметр, .

Метод нижней релаксации – это итерационный метод, который находит значение в левой части, используя прошлое значение в правой части.

*,*

где – это *k*-тое приближение . Следовательно, каждый элемент вектора может быть вычислен по формуле:

, (2)

Последовательная реализация метода

Для последовательного вычисления метода нижней релаксации используется функция *type\* low\_sor\_non\_parralel(type\*\* A, type\* b, int size, type omega, type\* initial, type convergence\_criteria, double& time)*, где *type* – это тип чисел, которые хранит матрица и вектор, *A* – квадратная матрица, *b* – вектор, *size* – размерность матрицы *A* и вектора *b*, *omega* – это релаксационный параметр, *initial* – вектор начальной инициализации, *convergence\_criteria* – точность с которой будет находиться решение, *time* – переменная для неявного результата, времени работы программы. В качестве результата функция возвращает вектор *x*, который является решение СЛАУ.

В начале функции создается вектор *phi*, который будет являться решением и результатом функции. Вектор *phi* инициализируем значениями из вектора *initial*. В переменную *start* записываем время начала работы, с помощью функции *omp\_get\_wtime()*. Далее используем функцию void *recompute\_accuracy\_non\_parralel(type\*\* A, type\* phi, type\* b, int size)* вычисляем точность текущего решения и записываем его в глобальную переменную *residual*. Используется норма Фробениуса для подсчета точности решения:

Далее идет цикл, который работает до тех пор, пока текущая точность больше необходимой. В цикле к каждому элементу вектора *phi* применяется формула *(2)*, после этого вычисляется текущая точность, аналогично, как это было сделано ранее и проверяется условие цикла. После окончания работы цикла в переменную *end* записывается время с помощью функции *omp\_get\_wtime()*, затем в переменную *time* записываем разность *end – start*. Переменная time будет хранить время затраченное на поиск решения в секундах. Функция возвращает вектор *phi* и заканчивает свою работу.

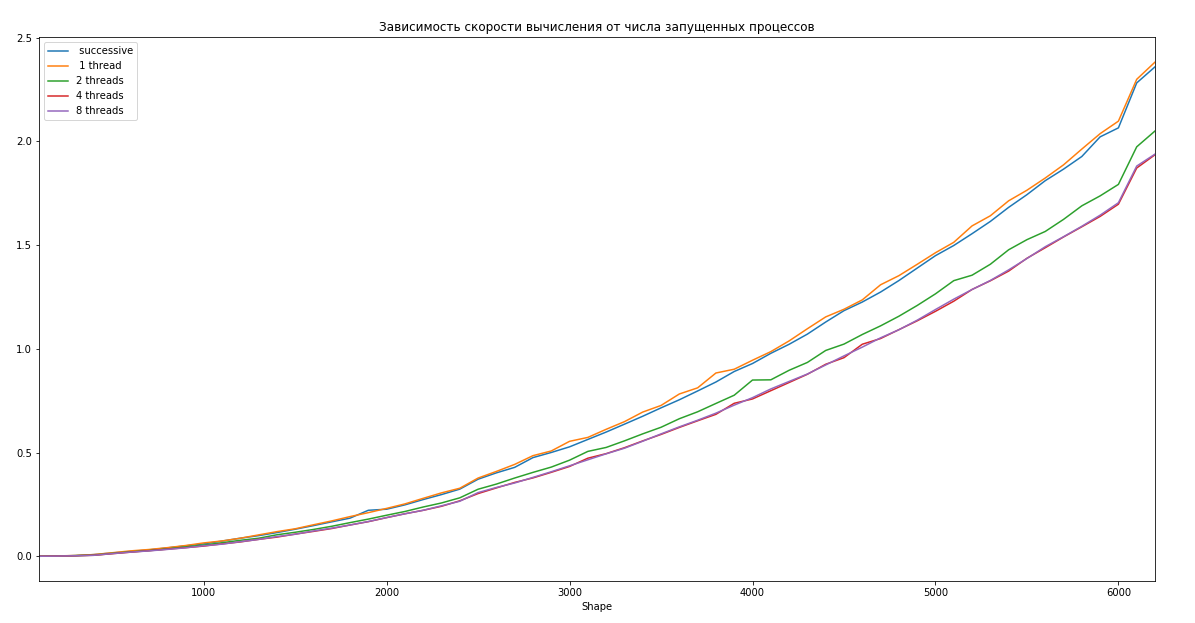
Параллельная реализация метода

Для параллельного вычисления метода нижней релаксации используется функция *type\* low\_sor(type\*\* A, type\* b, int size, type omega, type\* initial, type convergence\_criteria, int number\_threads, double& time)*. Смысл входных параметров аналогичен последовательной реализации, за исключением дополнительного параметра *number\_threads*, он отвечает за число потоков на которых будет выполняться функция. Внутренняя реализация отличается лишь использование функции *void recompute\_accuracy(type\*\* A, type\* phi, type\* b, int size, int number\_threads)* вместо *void recompute\_accuracy\_non\_parralel(type\*\* A, type\* phi, type\* b, int size)*, которая использует параллельные вычисления. Такое решение по распараллеливанию было сделано, потому что в методе не так много участков, где его можно применить, так цикл проверки точности распараллелить нельзя, цикл применения формулы *(2)* нельзя, так как каждая следующая итерация зависит от предыдущей, циклы для нахождения сумм нельзя, так как внутри них происходят “легкие” вычисления, что распараллеливание их приводит к ухудшению времени работы из-за накладных расходов связанных с параллельными вычислениями.

# Результаты вычислений

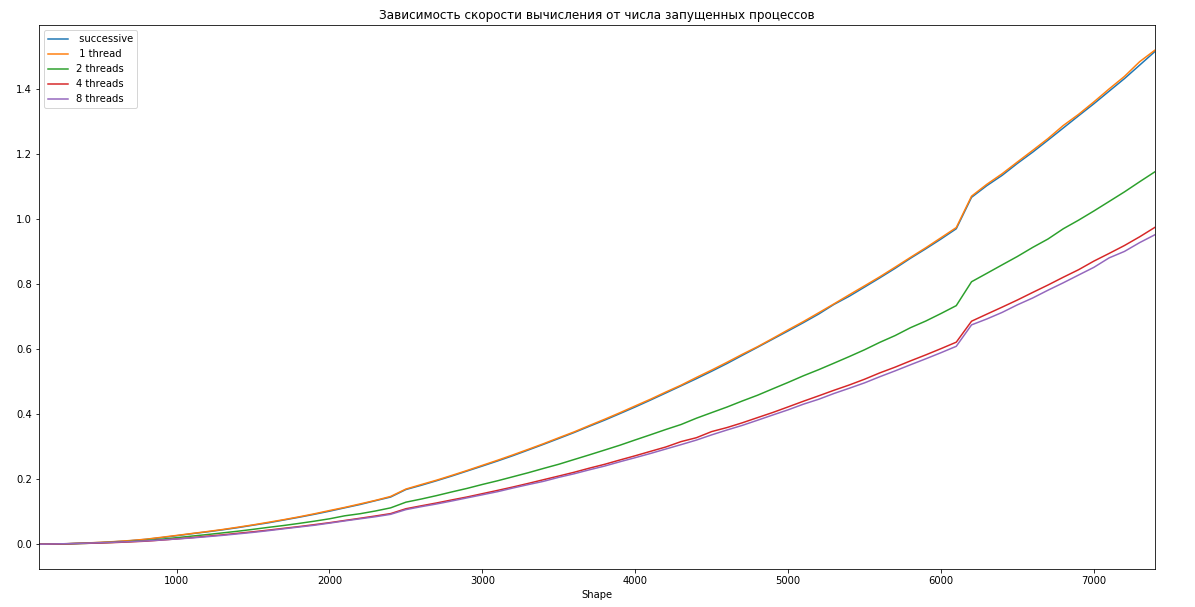
Вычисления были проведены на двух процессорах *AMD A8-7410*, который имеет 4 ядра и 4 потока и изготовлен по 28 нм техпроцессу, максимальная частота составляет 2500 MHz, и *AMD Ryzen 5 2600*, который имеет 6 ядер и 6 потоков и изготовлен по 12 нм техпроцессу, максимальная частота составляет 3400 MHz.

Вычисления производились при , была найдена опытным путем при помощи функции *void omega\_test()*, и вектором инициализации служил нулевой вектор.



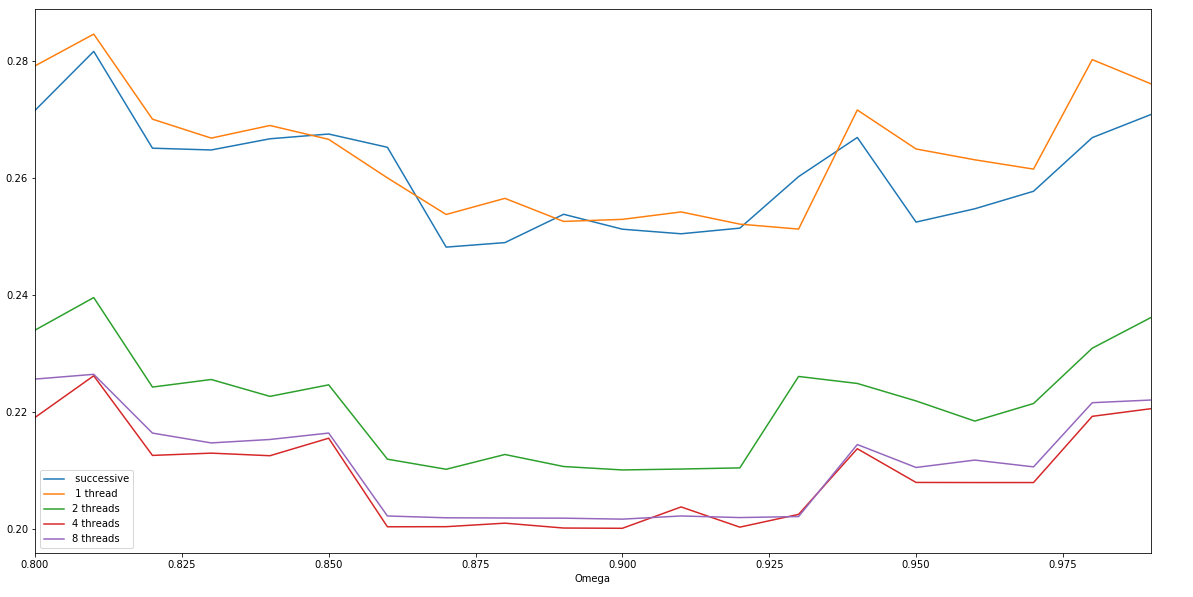
*Рисунок 1 – Зависимость скорости вычисления от числа запущенных процессов на процессоре AMD A8-7410*

Как можно увидеть из *рисунка 1* время работы последовательной программы почти такое же как у параллельной при участии только одного потока. Параллельный алгоритм становиться более эффективным уже при участии двух потоков, при участии четырёх потоков наблюдается прирост производительности по сравнению с двумя потоками, время работы при восьми потоках похоже на время при четырёх, это связанно с тем что у процессора только 4 ядра.

**

*Рисунок 2 – Зависимость скорости вычисления от числа запущенных процессов на процессоре AMD Ryzen 5 2600*

Результаты, полученные на *рисунке 2*, аналогичны предыдущему эксперименту, только разность в производительности между параллельной программой при одном, двух и четырёх стала более заметной. Так же и на предыдущем процессоре программа при 8 потоках выполняется с той же скоростью, что и при четырёх, хотя число ядер увеличилось с 4 до 6. Это связанно с тем, что шести ядрам приходиться симулировать работу восьми ядер, что приводит к дополнительным накладным расходам, в результате пропадает ожидаемый прирост производительности.

**

*Рисунок 3 – Зависимость скорости вычисления от значения параметра*

На *рисунке 3* изображено зависимость времени выполнения метода нижней релаксации последовательной и параллельной реализации с 1, 2, 4, 8 потоками от параметра . Эксперимент проводился при фиксированной размерности матрицы *A*, равной 1000. Вычисления происходили с шагом 0.01, с целью получения точного значения параметра . На рисунке можно увидеть, что в процессе вычислений образовалась область минимума между и . Середина этой области равна 0.895, было выбрано значение 0.89, так как при наблюдался скачок, поэтому округлил в меньшую сторону.

# Заключение

В результате мы получили, что не каждую программу можно распараллелить и даже само распараллеливание не всегда может дать ощутимый прирост в производительности. Так же при написании параллельных программ очень сильно важен выбор параметров, так на примере рисунка 2 мы заметили, что можно потерять прирост в производительности не правильным выбором числа потоков.

Метод нижней, в теории, хорошо подходит для распараллеливания, но как оказалось на практике, не так много участков кода можно эффективно распараллелить используя средства библиотеки OpenMP.

Несмотря на все трудности, которые были перечислены выше, параллельная реализация программы работает значительно быстрее чем последовательно, что говорит об эффективности параллельных вычислений и об их важности в написании производительных программ.

# Список использованных источников

1. Successive over-relaxation // en.wikipedia.org URL: https://en.wikipedia.org/wiki/Successive\_over-relaxation (дата обращения: 05.05.2020).

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

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

# Листинг программы

Main.cpp

#include <iostream>

#include <fstream>

#include <omp.h>

#include <stdlib.h> // srand, rand

#include <time.h>

#include "LinalGenerator.h" // generator function

// set data type

#define type double

double\*\* make\_matrix(int size)

{

std::vector<std::vector<type>> generated = generateGoodConditionedMatrix<type>(size);

type\*\* matrix = new type \* [size];

for (int i = 0; i < size; ++i)

matrix[i] = new type[size];

for (int i = 0; i < size; ++i)

for (int j = 0; j < size; ++j)

matrix[i][j] = generated[i][j];

return matrix;

}

type\* make\_random(int size)

// return random fill vector

{

type\* vector = new type[size];

for (int i = 0; i < size; ++i)

vector[i] = type(rand() % size) + 1;

return vector;

}

type\* make\_zeros(int size)

// return zero vector

{

type\* vector = new type[size];

for (int i = 0; i < size; ++i)

vector[i] = 0;

return vector;

}

bool check(type\* a, type\* b, int size, type eps)

// checking result

{

for (int i = 0; i < size; ++i)

if (abs(a[i] - b[i]) > eps)

return false;

return true;

}

type\* matmul(type\*\* matr, type\* v, int size)

// matrix by vector multiplication

{

type\* result = new type[size];

type temp = 0;

#pragma omp parallel for private(temp)

for (int i = 0; i < size; ++i)

{

temp = 0;

for (int j = 0; j < size; ++j)

temp += v[j] \* matr[i][j];

result[i] = temp;

}

return result;

}

type residual; // used to reduction

void recompute\_accuracy\_non\_parralel(type\*\* A, type\* phi, type\* b, int size)

// Non parralel method

// find accuracy using parallel

// accuracy metric is "The Frobenius norm"

{

type temp;

residual = 0;

for (int i = 0; i < size; ++i)

{

temp = -b[i];

for (int j = 0; j < size; ++j)

temp += A[i][j] \* phi[j];

residual += temp \* temp;

}

residual = sqrt(residual);

}

type\* low\_sor\_non\_parralel(type\*\* A, type\* b, int size,

type omega, type\* initial, type convergence\_criteria, double& time)

// Non parralel method

// function to find solution for Ax=b using lower SOR method

// A - (size x size) matrix

// vector b - right side of the equation

// omega - parameter

// initial - initial vector for x

// residual - accuracy

{

type\* phi = new type[size];

for (int i = 0; i < size; ++i) phi[i] = initial[i];

type sigma;

double start = omp\_get\_wtime(); // set start time

recompute\_accuracy\_non\_parralel(A, phi, b, size); // find accuracy

while (residual > convergence\_criteria)

{

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);

}

recompute\_accuracy\_non\_parralel(A, phi, b, size); // find accuracy

}

double end = omp\_get\_wtime(); // set end time

time = end - start; // compute time

return phi; // return result (x)

}

void recompute\_accuracy(type\*\* A, type\* phi, type\* b, int size, int number\_threads)

// find accuracy using parallel

// accuracy metric is "The Frobenius norm"

{

type temp;

residual = 0;

omp\_set\_num\_threads(number\_threads);

// works more effective than standard reduction

#pragma omp parallel for private(temp) shared(residual) schedule(dynamic)

for (int i = 0; i < size; ++i)

{

temp = -b[i];

for (int j = 0; j < size; ++j)

temp += A[i][j] \* phi[j];

#pragma omp critical

residual += temp \* temp;

}

residual = sqrt(residual);

}

type\* low\_sor(type\*\* A, type\* b, int size,

type omega, type\* initial, type convergence\_criteria, int number\_threads, double& time)

// function to find solution for Ax=b using lower SOR method

// A - (size x size) matrix

// vector b - right side of the equation

// omega - parameter

// initial - initial vector for x

// residual - accuracy

{

type\* phi = new type[size];

for (int i = 0; i < size; ++i) phi[i] = initial[i];

type sigma;

double start = omp\_get\_wtime(); // set start time

recompute\_accuracy(A, phi, b, size, number\_threads); // find accuracy

while (residual > convergence\_criteria)

{

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);

}

recompute\_accuracy(A, phi, b, size, number\_threads); // find accuracy

}

double end = omp\_get\_wtime(); // set end time

time = end - start; // compute time

return phi; // return result (x)

}

void test()

{

// ------------ Подготовительный блок -------------

srand(time(NULL));

// Set parametres

int size = 10000;

type omega = 0.9;

type convergence\_criteria = 0.001;

// Generate matrix

type\*\* A = make\_matrix(size);

// Generate matrix x

type\* x = make\_random(size);

// Set initial vector

type\* initial = make\_zeros(size);

// file pointer

std::fstream fout;

// opens an existing csv file or creates a new file.

fout.open("reportcard.csv", std::ios::out | std::ios::app);

// Table Header

fout << "Shape, successive, 1 thread,2 threads,4 threads,8 threads\n";

double sum\_time = 0;

double time = 0; // time check variable

int n = 20; // number of repeats

// ------------- Вычислительный блок -------------

// Make Table

for (int i = 100; i <= size; i += 100)

{

// Find b = Ax

// Перевычисляем каждый раз вектор b, так как меняем размерность

type\* b = matmul(A, x, i);

fout << i; // Put Shape

sum\_time = 0;

for (int r = 0; r < n; ++r)

{

type\* result\_np = low\_sor\_non\_parralel(A, b, i, // matrix A and vector b

omega, initial, convergence\_criteria, time); // computetion parameters

if (!check(result\_np, x, i, convergence\_criteria)) // check results

{

std::cout << "Different results!!!\n";

for (int k = 0; k < i; ++k)

std::cout << result\_np[k] << " " << x[k] << std::endl;

return;

}

sum\_time += time;

delete[] result\_np;

}

time = sum\_time / n;

fout << ", " << time; // Put OneWay time

for (int j = 1; j < 9; j\*=2)

{

sum\_time = 0;

for (int r = 0; r < n; ++r)

{

type\* result = low\_sor(A, b, i, // matrix A and vector b

omega, initial, convergence\_criteria, // computetion parameters

j, time); // checkin parameters

if (!check(result, x, i, convergence\_criteria)) // check results

{

std::cout << "Different results!!!\n";

for (int k = 0; k < i; ++k)

std::cout << result[k] << " " << x[k] << std::endl;

return;

}

sum\_time += time;

delete[] result;

}

time = sum\_time / n;

fout << ", " << time; // Put parralel time

}

fout << "\n"; // end table string

std::cout << i << std::endl; // Выводим счетчик для того чтобы наблюдать за процессом тестирования

delete[] b;

}

// Clear memory

for (int i = 0; i < size; ++i) delete[] A[i];

delete[] A;

delete[] x;

delete[] initial;

}

void omega\_test()

{

// ------------ Подготовительный блок -------------

srand(time(NULL));

// Set parametres

int size = 1000;

type convergence\_criteria = 0.001;

// Generate matrix

type\*\* A = make\_matrix(size);

// Generate matrix x

type\* x = make\_random(size);

// Set initial vector

type\* initial = make\_zeros(size);

// Find b = Ax

type\* b = matmul(A, x, size);

// file pointer

std::fstream fout;

// opens an existing csv file or creates a new file.

fout.open("reportcard\_omega.csv", std::ios::out | std::ios::app);

// Table Header

fout << "Omega, , 1 thread,2 threads,4 threads,8 threads\n";

double time = 0; // time check variable

// ------------- Вычислительный блок -------------

// Make Table

for (type omega = 0.8; omega <= 1; omega += 0.01)

{

fout << omega; // Put Omega

type\* result\_np = low\_sor\_non\_parralel(A, b, size, // matrix A and vector b

omega, initial, convergence\_criteria, time); // computetion parameters

if (!check(result\_np, x, size, convergence\_criteria)) // check results

{

std::cout << "Different results!!!\n";

for (int k = 0; k < size; ++k)

std::cout << result\_np[k] << " " << x[k] << std::endl;

return;

}

delete[] result\_np;

fout << ", " << time; // Put OneWay time

for (int j = 1; j < 9; j\*=2)

{

type\* result = low\_sor(A, b, size, // matrix A and vector b

omega, initial, convergence\_criteria, // computetion parameters

j, time); // checkin parameters

if (!check(result, x, size, convergence\_criteria)) // check results

{

std::cout << "Different results!!!\n";

for (int k = 0; k < size; ++k)

std::cout << result[k] << " " << x[k] << std::endl;

return;

}

fout << ", " << time; // Put parralel time

delete[] result;

}

fout << "\n"; // end table string

std::cout << omega << std::endl; // Выводим omega для того чтобы наблюдать за процессом тестирования

}

// Clear memory

for (int i = 0; i < size; ++i) delete[] A[i];

delete[] A;

delete[] x;

delete[] initial;

delete[] b;

}

int main()

{

test();

}

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