Московский авиационный институт

(Национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра «Вычислительная математика и программирование»

Курсовая работы

по курсу «Численные методы»

Тема: «Распараллеливание вычислительных алгоритмов решения задач линейной алгебры»

Выполнил: Наседкин Г.К.

Группа: М8О-405Б-20

Проверил: доц. Иванов И. Э.

Дата:

Оценка:

Москва, 2023

Оглавление

Задание

Примернить методы параллельного программирования к методом решения задач линейной алгебры.

Теоретические сведения

Для решения задачи распараллеливания алгоритмов решения задач линейной алгебры были использованы две библиотеки:

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Thread - Поток класса представляет собой одиночный поток выполнения. Потоки позволяют одновременно выполнять несколько функций. Потоки начинают выполнение сразу после создания связанного объекта потока, начиная с функции верхнего уровня, предоставленной в качестве аргумента конструктора.

Параллельные вычисления — это тип вычислений, при котором одновременно выполняется множество операций или процессов. Большие проблемы часто можно разделить на более мелкие, которые затем могут быть решены одновременно, и, как следствие, большая проблема будет решена при таком подходе гораздо быстрее.

Как это устроено в компьютере:

Процесс на компьютере — это любое отдельно запущенное приложение. Например, открытый браузер, антивирусная программа, Skype и др. — все это отдельные процессы на компьютере.

Каждый отдельный процесс способен существовать отдельно друг от друга в том смысле, что независимо потребляет ресурсы устройства — оперативная память и время процессора.

Один процесс может порождать несколько потоков (Threads), которые будут выполняться одновременно и параллельно. Важно отметить, что все потоки одного процесса будут исполнять отдельные части кода одной программы, деля между собой общие ресурсы, выделенные процессу.

Современные компьютеры имеют продвинутые процессоры, у которых есть по несколько ядер. Каждое отдельное ядро способно обработать минимум один поток. Раньше было так: одно ядро имело один ядерный поток, а один поток — это выполнение одной команды. Теперь научились

делать ядра многопоточными, а это означает, что одно ядро может одновременно выполнять несколько команд в соответствии с числом ядерных потоков.

Код

Matrix.h

```
#pragma once
#include <vector>
#include <fstream>
#include <iostream>
#include <string>
#include <cmath>
#include <omp.h>
using namespace std;

struct Rect { size_t y, x, h, w; };

////////////////////////////////////////////////////////
// Class
////////////////////////////////////////////////////////

template <typename _T>
class Matrix {
public:
    vector<vector<_T>> data;
    pair<size_t, size_t> dim;

    Matrix() : Matrix((int)0) {} // default constructor
    Matrix(size_t, size_t=0); // create matrix nxm
    Matrix(size_t, size_t, _T); // create matrix nxm with default value
    Matrix(pair<size_t, size_t> p) : Matrix(p.first, p.second) {}
    Matrix(Matrix<_T>&); // copy matryx
    Matrix(char*); // load matrix from file
    Matrix(vector<_T>, size_t=0, size_t=0); // load matrix from vector
    Matrix(vector<vector<_T>>); // load matrix from vector of vector
    vector<_T>& operator[](size_t); // get line
    void printM(); // print matrix
```

```
    void recordM(char*); // record matrix to file

    void sumL(size_t, size_t, _T, size_t=0, size_t=0); // adds the first row to the second row
with a coefficient

    void subL(size_t i, size_t j, _T coef, size_t from=0, size_t to=0) { this->sumL(i, j, -
coef, from, to); }

    void sumC(size_t, size_t, _T, size_t=0, size_t=0); // adds the first row to the second row
with a coefficient

    void subC(size_t i, size_t j, _T coef, size_t from=0, size_t to=0) { this->sumC(i, j, -
coef, from, to); }

    void swapL(size_t, size_t); // swap lines

    void swapC(size_t, size_t); // swap colomns

    pair<size_t, size_t> max(size_t, size_t, size_t, size_t); // max in rect

    size_t maxC(size_t, size_t=0, size_t=0); // max in one colomn

    size_t maxL(size_t, size_t=0, size_t=0); // max in one line

    Matrix<_T> T();

    double norm1();

    double norm2();

    double normC();

    double normL();

    void resize(size_t, size_t);

    void copy(Matrix<_T>, size_t=0, size_t=0, Rect={0, 0, 0, 0});

    bool isSquare();

    bool isSymmetrical();

};


////////////////////////////////////////////////////////////

// Realization

////////////////////////////////////////////////////////////


template <typename _T>

Matrix<_T>::Matrix(size_t n, size_t m) {

    m = (m == 0) ? n : m;

    this->dim = {n, m};

    this->data.resize(n, vector<_T>(m));

}


template <typename _T>

Matrix<_T>::Matrix(Matrix<_T>& m) {

    this->dim = m.dim;
```

```
    this->data = m.data;
}

template <typename _T>
Matrix<_T>::Matrix(size_t n, size_t m, _T value) {
    this->dim = {n, m};
    this->data.resize(n, vector<_T>(m, value));
}

template <typename _T>
Matrix<_T>::Matrix(char* s) {
    ifstream input(s);
    input >> this->dim.first >> this->dim.second;
    this->data.resize(this->dim.first, vector<_T>(this->dim.second));
    for (size_t i = 0; i < this->dim.first; i++)
        for (size_t j = 0; j < this->dim.second; j++)
            input >> this->data[i][j];
}

template <typename _T>
Matrix<_T>::Matrix(vector<_T> v, size_t n, size_t m) {
    n = (n == 0) ? 1        : n;
    m = (m == 0) ? v.size() : m;
    this->dim = {n, m};
    this->data.resize(n, vector<_T>(m));
    // #pragma omp parallel for
    for (size_t i = 0; i < n; i++)
        // #pragma omp parallel for
        for (size_t j = 0; j < m; j++)
            this->data[i][j] = v[i * m + j];
}

template <typename _T>
Matrix<_T>::Matrix(vector<vector<_T>> v) {
    this->dim = {v.size(), v.size() == 0 ? 0 : v[0].size()};
    this->data = v;
}
```

```
template <typename _T>
vector<_T>& Matrix<_T>::operator[](size_t i) { return this->data[i]; }

template <typename _T>
void Matrix<_T>::printM() {
    cout.precision(3);
    for (size_t i = 0; i < this->dim.first; i++) {
        for (size_t j = 0; j < this->dim.second; j++)
            cout << this->data[i][j] << '\t';
        cout << '\n';
    }
}

template <typename _T>
void Matrix<_T>::recordM(char* s) {
    ofstream output(s);
    output.precision(4);
    output << this->dim.first << ' ' << this->dim.second << '\n';
    for (size_t i = 0; i < this->dim.first; i++) {
        for (size_t j = 0; j < this->dim.second; j++)
            output << this->data[i][j] << '\t';
        output << '\n';
    }
}

template <typename _T>
void Matrix<_T>::sumL(size_t i, size_t j, _T coef, size_t from, size_t to) {
    to = (to == 0) ? this->dim.second : to;
    size_t _from;
    // #pragma omp parallel for shared(from, to, coef, i, j, data) private(_from)
    for (_from = from; _from < to; _from++)
        this->data[j][_from] += coef * this->data[i][_from];
}

template <typename _T>
void Matrix<_T>::sumC(size_t i, size_t j, _T coef, size_t from, size_t to) {
    to = (to == 0) ? this->dim.first : to;
    // #pragma omp parallel for
```

```
    for (size_t _from = from; _from < to; _from++)
        this->data[_from][j] += coef * this->data[_from][i];
}

template <typename _T>
void Matrix<_T>::swapL(size_t i, size_t j) {
    swap(this->data[j], this->data[i]);
}
template <typename _T>
void Matrix<_T>::swapC(size_t i, size_t j) {
    // #pragma omp parallel for
    for (size_t k = 0; k < this->dim.first;  k++) swap(this->data[k][j], this->data[k][i]);
}

template <typename _T>
pair<size_t, size_t> Matrix<_T>::max(size_t fromC, size_t toC, size_t fromL, size_t toL) {
    pair<size_t, size_t> res {fromL, fromC}; // {line, colomn}
    for (size_t i; fromC < toC; fromC++)
        for (i = fromL; i < toL; i++)
            if (abs(this->data[res.first][res.second]) < abs(this->data[i][fromC]))
                res = {i, fromC};
    return res; // {line, colomn}
}

template <typename _T>
size_t Matrix<_T>::maxC(size_t Colomn, size_t from, size_t to) {
    to = (to == 0) ? this->dim.second : to;
    return this->max(Colomn, Colomn + 1, from, to).first;
}

template <typename _T>
size_t Matrix<_T>::maxL(size_t Line, size_t from, size_t to) {
    to = (to == 0) ? this->dim.first : to;
    return this->max(from, to, Line, Line + 1).second;
}

template <typename _T>
Matrix<_T> Matrix<_T>::T() {
```

```
    Matrix<_T> res(this->dim.second, this->dim.first);
    // #pragma omp parallel for
    for (size_t i = 0; i < res.dim.first; i++)
        // #pragma omp parallel for
        for (size_t j = 0; j < res.dim.second; j++)
            res[i][j] = this->data[j][i];
    return res;
}

template <typename _T>
double Matrix<_T>::norm1() {
    double res = 0;
    for (size_t i = 0; i < this->dim.first; i++)
        for (size_t j = 0; j < this->dim.second; j++)
            res = std::max(res, this->data[i][j]);
    return res;
}

template <typename _T>
double Matrix<_T>::norm2() {
    double res = 0;
    for (size_t i = 0; i < this->dim.first; i++)
        for (size_t j = 0; j < this->dim.second; j++)
            res += pow(this->data[i][j], 2);
    return pow(res, 0.5d);
}

template <typename _T>
double Matrix<_T>::normC() {
    double res = 0, cur = 0;
    for (size_t i = 0; i < this->dim.first; i++) {
        cur = 0;
        for (size_t j = 0; j < this->dim.second; j++)
            cur += abs(this->data[i][j]);
        res = std::max(cur, res);
    }
    return res;
}
```

```
template <typename _T>
double Matrix<_T>::normL() {
    double res = 0;
    for (size_t i = 0; i < this->dim.first; i++) {
        for (size_t j = 0; j < this->dim.second; j++)
            res = std::max(abs(this->data[i][j]), res);
    }
    return res;
}

template <typename _T>
void Matrix<_T>::resize(size_t n, size_t m) {
    this->dim = {n, m};
    this->data.resize(n);
    // #pragma omp parallel for
    for (size_t i = 0; i < n; i++) this->data[i].resize(m);
}

template <typename _T>
void Matrix<_T>::copy(Matrix<_T> M, size_t y, size_t x, Rect rect) {
    rect.h = rect.h == 0 ? M.dim.first : rect.h;
    rect.w = rect.w == 0 ? M.dim.second : rect.w;
    // #pragma omp parallel for
    for (int i = 0; i < rect.h; i++) {
        // #pragma omp parallel for
        for (int j = 0; j < rect.w; j++) {
            this->data[y + i][x + j] = M[rect.y + i][rect.x + j];
        }
    }
}

template <typename _T>
bool Matrix<_T>::isSquare() { return dim.first == dim.second; }

template <typename _T>
bool Matrix<_T>::isSymmetrical() {
    if (!isSquare()) return false;
```

```
    for (size_t i = 0; i < dim.first; i++)
        for (size_t j = i + 1; j < dim.second; j++)
            if (data[i][j] != data[j][i])
                return false;
    return true;
}

////////////////////////////////////////////////////////
// Operators
////////////////////////////////////////////////////////

template <typename _T>
Matrix<_T> operator*(Matrix<_T> left, Matrix<_T> right) {
    if (left.dim.second != right.dim.first) {
        cerr << "Dimension error: can't multiply matrix " << left.dim.first << "x" <<
                left.dim.second << " with matrix " << right.dim.first << "x" <<
right.dim.second << "\n";
        exit(1);
    }
    Matrix<_T> res(left.dim.first, right.dim.second);
    // #pragma omp parallel for
    for (size_t i = 0; i < left.dim.first; i++)
        // #pragma omp parallel for
        for (size_t j = 0; j < right.dim.second; j++)
            // #pragma omp parallel for
            for (size_t k = 0; k < right.dim.first; k++)
                res[i][j] += left[i][k] * right[k][j];
    return res;
}

template <typename _T>
Matrix<_T> operator-(Matrix<_T> m) {
    Matrix<_T> res(m.dim);
    // #pragma omp parallel for
    for (size_t i = 0; i < res.dim.first; i++)
        // #pragma omp parallel for
        for (size_t j = 0; j < res.dim.second; j++)
            res[i][j] = -m[i][j];
```

```
    return res;
}

template <typename _T>
Matrix<_T> operator+(Matrix<_T> left, Matrix<_T> right) {
    if (left.dim.first != right.dim.first && left.dim.second != right.dim.second) {
        cerr << "Dimension error: can't sum matrix " << left.dim.first << "x" <<
left.dim.second
             << " with matrix " << right.dim.first << "x" << right.dim.second << "\n";
        exit(1);
    }
    Matrix<_T> res(left.dim);
    // #pragma omp parallel for
    for (size_t i = 0; i < res.dim.first; i++)
        // #pragma omp parallel for
        for (size_t j = 0; j < res.dim.second; j++)
            res[i][j] = left[i][j] + right[i][j];
    return res;
}
template <typename _T>
Matrix<_T> operator-(Matrix<_T> left, Matrix<_T> right) { return left + (-right); }

template <typename _T>
Matrix<_T> operator*(Matrix<_T> left, _T right) {
    Matrix<_T> res(left.dim);
    // #pragma omp parallel for
    for (size_t i = 0; i < left.dim.first; i++)
        // #pragma omp parallel for
        for (size_t j = 0; j < left.dim.second; j++)
            res[i][j] = left[i][j] * right;
    return res;
}
template <typename _T>
Matrix<_T> operator*(_T left, Matrix<_T> right) { return right * left; }



////////////////////////////////////////////////////////
// functions
```

```
////////////////////////////////////////////////////////////

template <typename _T>
Matrix<_T> E(size_t n) {
    Matrix<_T> res(n);
    // #pragma omp parallel for
    for (size_t i = 0; i < n; i++) res[i][i] = 1;
    return res;
}
```

LinearSolver.h

```
#pragma once
#include "../Matrix.h"
#include <cstdint>
#include <thread>

////////////////////////////////////////////////////////////
// Class
////////////////////////////////////////////////////////////

template <typename T> class LinearSolver {
private:
    Matrix<T> Iter(Matrix<T>& m, Matrix<T> b, double epsilon, bool flag);
public:
    // LU
    pair<Matrix<T>, vector<pair<size_t, size_t>>> LUdecompos(Matrix<T>& m);
    Matrix<T> LUsolve(Matrix<T>& m, Matrix<T>& b);
    double detM(Matrix<T>& m);
    Matrix<T> inverseM(Matrix<T>& m);

    // TMA
    bool CheckCondition(Matrix<T>& m);
    Matrix<T> TMAsolve(Matrix<T>& m, Matrix<T>& b);

    // Iter
    Matrix<T> SimpleIter(Matrix<T>& m, Matrix<T>& b, double epsilon);
    Matrix<T> Zaydel(Matrix<T>& m, Matrix<T>& b, double epsilon);
```

```
    // Jacobi
    T t(Matrix<T>& m);
    pair<vector<T>, Matrix<T>> Jacobi(Matrix<T>& m, double epsilon);

    // QR
    pair<Matrix<T>, Matrix<T>> QRdecompose(Matrix<T>& m);
    pair<vector<T>, vector<pair<T, T>>> QRsolve(Matrix<T>& m, double epsilon);
};

////////////////////////////////////////////////////////////
// Realization
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
// LU
template <typename T>
pair<Matrix<T>, vector<pair<size_t, size_t>>> LinearSolver<T>::LUdecompos(Matrix<T>& m) {
    if (!m.isSquare()) {
        cerr << "LU decomposition working only for square matrix\n"; exit(1);
    }

    Matrix<T> LUm(m);
    vector<pair<size_t, size_t>> P;

    for (size_t j = 0, i; j < m.dim.first - 1; j++) {
        size_t k = LUm.maxC(j, j);
        if (k != j) {
            LUm.swapL(j, k);
            P.emplace_back(j, k);
        }
        #pragma omp parallel for shared(j, m, LUm) private(i)
        for (i = j + 1; i < m.dim.first; i++) {
            LUm[i][j] /= LUm[j][j];
            LUm.subL(j, i, LUm[i][j], j + 1);
        }
    }
    return {LUm, P};
}
```

```
template <typename T>
Matrix<T> LinearSolver<T>::LUsolve(Matrix<T>& m, Matrix<T>& b) {
    if (!m.isSquare()) {
        cerr << "LU decomposition working only for square matrix\n"; exit(1);
    }
    if (m.dim.first != b.dim.second) {
        cerr << "The dimension of the left side does not coincide with the dimension of the
right side\n"; exit(1);
    }

    Matrix<T> res(b);
    pair<Matrix<T>, vector<pair<size_t, size_t>>> resLU = LUdecompos(m);

    size_t k;
    #pragma omp parallel for shared(resLU, res, b) private(k)
    for (k = 0; k < b.dim.first; k++) {
        for (size_t i = 0; i < resLU.second.size(); i++)
            swap(res[k][resLU.second[i].first], res[k][resLU.second[i].second]);
        for (size_t i = 0; i < res.dim.second; i++)
            for (size_t j = 0; j < i; j++)
                res[k][i] -= res[k][j] * resLU.first[i][j];
        for (size_t i = res.dim.second - 1; i != UINT64_MAX; i--) {
            for (size_t j = i + 1; j < res.dim.second; j++)
                res[k][i] -= res[k][j] * resLU.first[i][j];
            res[k][i] /= resLU.first[i][i];
        }
    }
    return res;
}

template <typename T>
double LinearSolver<T>::detM(Matrix<T>& m) {
    pair<Matrix<T>, vector<pair<size_t, size_t>>> resLU = LUdecompos(m);
    T res = resLU.first[0][0];
    for (size_t i = 1; i < m.dim.first; i++)
        res *= resLU.first[i][i];
    return res * ((resLU.second.size() % 2) ? -1 : 1);
```

```
}

template <typename T>
Matrix<T> LinearSolver<T>::inverseM(Matrix<T>& m) {
    Matrix<T> e = E<T>(m.dim.second);
    return LUsolve(m, e);
}
////////////////////////////////////////////////////////

////////////////////////////////////////////////////////
// TMA
template <typename T>
bool LinearSolver<T>::CheckCondition(Matrix<T>& m) {
    bool res = true;
    int counter = 0;
    for (size_t i = 1; i < m.dim.second - 1; i++) {
        counter += (abs(m[1][i]) > abs(m[0][i]) + abs(m[2][i])) ? 1 : -1;
        res = (res && abs(m[1][i]) >= abs(m[0][i]) + abs(m[2][i]));
    }
    return res && (counter >= 0);
}

template <typename T>
Matrix<T> LinearSolver<T>::TMAsolve(Matrix<T>& m, Matrix<T>& b) {
    if (!CheckCondition(m)) cout << "Warning! for this matrix, the method is unstable!\n";
    size_t N = m.dim.second;
    Matrix<T> res(b);

    vector<T> P(N);
    P[0] = -m[2][0] / m[1][0]; P[N - 1] = 0;
    for (size_t i = 1; i < N - 1; i++)
        P[i] = -m[2][i] / (m[1][i] + m[0][i] * P[i - 1]);

    size_t k;
    #pragma omp parallel for shared(m, b, P, res, N) private(k)
    for (k = 0; k < b.dim.first; k++) {
        vector<T> Q(N);
        Q[0] = b[k][0] / m[1][0];
```

```
        for (size_t i = 1; i < N - 1; i++)
            Q[i] = (b[k][i] - m[0][i] * Q[i - 1]) / (m[1][i] + m[0][i] * P[i - 1]);
        Q[N - 1] = (b[k][N - 1] - m[0][N - 1] * Q[N - 2]) / (m[1][N - 1] + m[0][N - 1] * P[N -
2]);

        res[k][N - 1] = Q[N - 1];
        for (size_t i = N - 2; i != UINT64_MAX; i--)
            res[k][i] = res[k][i + 1] * P[i] + Q[i];
    }
    return res;
}
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
// Iter
template <typename T>
Matrix<T> LinearSolver<T>::Iter(Matrix<T>& m, Matrix<T> b, double epsilon, bool flag) {
    size_t N = m.dim.second;
    Matrix<T> alpha(m);

    T coef;
    for (size_t i = 0; i < N; i++) {
        coef = alpha[i][i];
        alpha[i][i] = 0;
        for (size_t j = 0; j < N; j++) {
            alpha[i][j] /= -coef;
        }
        for (size_t n = 0; n < b.dim.first; n++) {
            b[n][i] /= coef;
        }
    }

    double normAlpha = alpha.normC();
    Matrix<T> res(b.dim);

    Matrix<T> was;
    size_t k;
```

```
    #pragma omp parallel for shared(res, N, b, epsilon, flag, alpha, normAlpha) private(k,
was)
    for (k = 0; k < b.dim.first; k++) {
        Matrix<T> cur(b[k], N, 1), betha(b[k], N, 1);
        do {
            was = cur;
            if (flag) {
                cur = betha + alpha * cur;
            } else {
                for (size_t i = 0; i < N; i++) {
                    cur[i][0] = betha[i][0];
                    for (size_t j = 0; j < i; j++) {
                        cur[i][0] += alpha[i][j] * cur[j][0];
                    }
                    for (size_t j = i; j < N; j++) {
                        cur[i][0] += alpha[i][j] * was[j][0];
                    }
                }
            }
        } while ((cur - was).normC() * (normAlpha / (1 - normAlpha)) > epsilon);
        for (size_t i = 0; i < N; i++) {
            res[k][i] = cur[i][0];
        }
    }
    return res;
}

template <typename T>
Matrix<T> LinearSolver<T>::SimpleIter(Matrix<T>& m, Matrix<T>& b, double epsilon) {
    return Iter(m, b, epsilon, true);
}
template <typename T>
Matrix<T> LinearSolver<T>::Zaydel(Matrix<T>& m, Matrix<T>& b, double epsilon) {
    return Iter(m, b, epsilon, false);
}
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
```

```
// Jacobi
template <typename T>
T LinearSolver<T>::t(Matrix<T>& m) {
    T res = 0;
    for (size_t i = 0; i < m.dim.first; i++)
        for (size_t j = i + 1; j < m.dim.first; j++)
            res += m[i][j] * m[i][j];
    return pow(res, 0.5d);
}

template <typename T>
pair<vector<T>, Matrix<T>> LinearSolver<T>::Jacobi(Matrix<T>& m, double epsilon) {
    size_t N = m.dim.second;
    if (!m.isSymmetrical()) { cerr << "Jacobi working only for simmetric matrix\n"; exit(1); }

    Matrix<T> alpha(m), u = E<T>(N), uT = E<T>(N), resM = E<T>(N);
    double phi;
    do {
        for (size_t i = 0; i < N; i++)
            for (size_t j = i + 1; j < N; j++) {
                phi = (alpha[i][i] == alpha[j][j]) ? M_PI_4 : 0.5 * atan(2 * alpha[i][j] /
(alpha[i][i] - alpha[j][j]));
                u[i][i] = cos(phi); u[i][j] = -sin(phi);
                u[j][i] = sin(phi); u[j][j] =  cos(phi);

                uT[i][i] =  cos(phi); uT[i][j] = sin(phi);
                uT[j][i] = -sin(phi); uT[j][j] = cos(phi);

                std::thread first([&](){ alpha = uT * alpha * u; }),
                            second([&](){ resM = resM * u; });
                first.join();
                second.join();

                u[i][i] = T(1); u[i][j] = T(0);
                u[j][i] = T(0); u[j][j] = T(1);
                uT[i][i] = T(1); uT[i][j] = T(0);
                uT[j][i] = T(0); uT[j][j] = T(1);
            }
```

```
    } while (t(alpha) > epsilon);
    vector<T> resV(N);
    for (size_t i = 0; i < N; i++) resV[i] = alpha[i][i];
    return {resV, resM};
}
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
// QR
template <typename T>
pair<Matrix<T>, Matrix<T>> LinearSolver<T>::QRdecompose(Matrix<T>& m) {
    size_t N = m.dim.second;
    Matrix<T> Q = E<T>(N), R(m), v(N, 1);
    vector<Matrix<T>*> H(N - 1, nullptr);
    std::thread thr([&](){
        for (size_t i = 0; i < H.size(); i++) {
            while (H[i] == nullptr) {}
            Q = Q * (*(H[i]));
        }
    });

    Matrix<T>* newH = nullptr;
    for (size_t i = 0; i < N - 1; i++) {
        T temp = 0;
        for (size_t j = i; j < N; j++) {
            v[j][0] = R[j][i];
            temp = temp + R[j][i] * R[j][i];
        }
        v[i][0] += ((R[i][i] < 0) ? -pow(temp, 0.5) : pow(temp, 0.5));
        newH = new Matrix<T>;
        *newH = - (v * v.T()) * (T)(2.d / (v.T() * v)[0][0]);
        for (size_t j = 0; j < v.dim.first; j++) newH->data[j][j] += 1;
        H[i] = newH;
        R = (*(H[i])) * R;
        v[i][0] = 0;
    }
    thr.join();
    for (size_t i = 0; i < H.size(); i++) delete H[i];
```

```
    return {Q, R};
}

template <typename T>
pair<vector<T>, vector<pair<T, T>>> LinearSolver<T>::QRsolve(Matrix<T>& m, double epsilon) {
    size_t N = m.dim.second;
    Matrix<T> A(m);
    vector<pair<T, T>> resC(N);
    vector<T> resR(N);
    bool flag;
    size_t indexC, indexR;
    do {
        indexC = 0; indexR = 0;
        flag = true;
        pair<Matrix<T>, Matrix<T>> qr = QRdecompose(A);
        A = qr.second * qr.first;
        for (size_t i = 0; i < N; i++) {
            T sum = (T)0;
            for (size_t j = i + 1; j < N; j++) sum = sum + A[j][i] * A[j][i];
            if (pow(sum, 0.5) > epsilon) {
                T b = A[i][i] + A[i + 1][i + 1], c = A[i + 1][i] * A[i][i + 1];
                T d = pow(b, 2) - 4 * c;
                pair<T, T> sqr1 = {-b / 2.d,   pow(d, 0.5) / 2.d},
                           sqr2 = {-b / 2.d, - pow(d, 0.5) / 2.d};
                flag = flag && (max(hypot(sqr1.first - resC[indexC].first, sqr1.second -
resC[indexC].second),
                                    hypot(sqr2.first - resC[indexC + 1].first, sqr2.second -
resC[indexC + 1].second)) < epsilon);
                resC[indexC++] = sqr1;
                resC[indexC++] = sqr2;
                i += 1;
            } else {
                resR[indexR++] = A[i][i];
            }
        }
    } while (!flag);
    resR.resize(indexR);
    resC.resize(indexC);
```

```
    return {resR, resC};
}
//////////////////////////////////////////////////////////
```

main.cpp

```
#include "LinearSolver.h"
#include "../lab1/LU/LU.h"
#include "../lab1/TMA/TMA.hpp"
#include "../lab1/Iter/Iter.h"
#include "../lab1/Jacobi/Jacobi.h"
#include "../lab1/QR/QR.h"
#include <chrono>

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    LinearSolver<double> solver;
    Matrix<double> answer;

    char choice; cout << "chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR\n";
cout.flush();
    do { cin >> choice; } while (!('1' <= choice || choice <= '5'));

    switch (choice) {
        case '1': { // LU test
            Matrix<double> m1((char*)"LUm.txt");
            Matrix<double> b1((char*)"LUb.txt");

            const auto start1 = std::chrono::high_resolution_clock::now();
            answer = solver.LUsolve(m1, b1);
            const auto end1 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff1 = end1 - start1;
            cout << "The     parallel time: " << diff1.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer1.txt");

            const auto start2 = std::chrono::high_resolution_clock::now();
            answer = LUsolve(m1, b1);
            const auto end2 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff2 = end2 - start2;
```

```
            cout << "The non parallel time: " << diff2.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer2.txt");

            break;
        }
        case '2': { // TMA test
            Matrix<double> m2((char*)"TMAm.txt");
            Matrix<double> b2((char*)"TMAb.txt");

            const auto start1 = std::chrono::high_resolution_clock::now();
            answer = solver.TMAsolve(m2, b2);
            const auto end1 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff1 = end1 - start1;
            cout << "The     parallel time: " << diff1.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer1.txt");

            const auto start2 = std::chrono::high_resolution_clock::now();
            answer = TMAsolve(m2, b2);
            const auto end2 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff2 = end2 - start2;
            cout << "The non parallel time: " << diff2.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer2.txt");

            break;
        }
        case '3': { // Iter test
            Matrix<double> m3((char*)"Iterm.txt");
            Matrix<double> b3((char*)"Iterb.txt");
            char TheMethod;
            cout << "What TheMethod do you want to use: 1 – SimpleIter, 2 - Zaydel\n";
cout.flush();
            do { cout << ">>> "; cout.flush(); cin >> TheMethod; } while (TheMethod != '1' &&
'2' != TheMethod);
            double epsilon; cout << "with epsilon = "; cout.flush(); cin >> epsilon;

            const auto start1 = std::chrono::high_resolution_clock::now();
            answer = (TheMethod == '1') ? solver.SimpleIter(m3, b3, epsilon) :
solver.Zaydel(m3, b3, epsilon);
            const auto end1 = std::chrono::high_resolution_clock::now();
```

```
            const std::chrono::duration<double> diff1 = end1 - start1;
            cout << "The     parallel time: " << diff1.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer1.txt");

            const auto start2 = std::chrono::high_resolution_clock::now();
            answer = (TheMethod == '1') ? SimpleIter(m3, b3, epsilon) : Zaydel(m3, b3,
epsilon);
            const auto end2 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff2 = end2 - start2;
            cout << "The non parallel time: " << diff2.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer2.txt");

            break;
        }
        case '4': { // Jacobi test
            double epsilon; cout << "with epsilon = "; cout.flush(); cin >> epsilon;
            Matrix<double> m4((char*)"Jacobim.txt");

            const auto start1 = std::chrono::high_resolution_clock::now();
            answer = solver.Jacobi(m4, epsilon).second;
            const auto end1 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff1 = end1 - start1;
            cout << "The     parallel time: " << diff1.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer1.txt");

            const auto start2 = std::chrono::high_resolution_clock::now();
            answer = Jacobi(m4, epsilon).second;
            const auto end2 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff2 = end2 - start2;
            cout << "The non parallel time: " << diff2.count() << " seconds\n"; cout.flush();
            answer.recordM((char*)"answer2.txt");

            break;
        }
        case '5': { // QR test
            double epsilon; cout << "with epsilon = "; cout.flush(); cin >> epsilon;
            Matrix<double> m5((char*)"QRm.txt");
```

```
            const auto start1 = std::chrono::high_resolution_clock::now();
            auto [R1, C1] = solver.QRsolve(m5, epsilon);
            const auto end1 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff1 = end1 - start1;
            cout << "The     parallel time: " << diff1.count() << " seconds\n"; cout.flush();

            std::ofstream answer1("answer1.txt");
            for (size_t i = 0; i < R1.size(); i++) answer1 << R1[i] << '\n';
            for (size_t i = 0; i < C1.size(); i++)
                answer1 << C1[i].first << " + " << C1[i].second << "i\n";

            const auto start2 = std::chrono::high_resolution_clock::now();
            auto [R2, C2] = QRsolve(m5, epsilon);
            const auto end2 = std::chrono::high_resolution_clock::now();
            const std::chrono::duration<double> diff2 = end2 - start2;
            cout << "The non parallel time: " << diff2.count() << " seconds\n"; cout.flush();

            std::ofstream answer2("answer2.txt");
            for (size_t i = 0; i < R2.size(); i++) answer2 << R2[i] << '\n';
            for (size_t i = 0; i < C2.size(); i++)
                answer2 << C2[i].first << " + " << C2[i].second << "i\n";

            break;
        }
    }

    exit(0);
    return 0;
}
```

Результаты работы программы

Тесты для LU метода

Для матрицы **100x100** и **500** векторов правой части

```
chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR
1
The     parallel time: 0.037411 seconds
The non parallel time: 0.079825 seconds
```

Для матрицы **500x500** и **500** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

1

The time: 0.687726 seconds

The time: 2.2731 seconds

Для матрицы **1000x1000** и **500** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

1

The   parallel time: 3.22049 seconds

The non parallel time: 9.94695 seconds

Тесты для метода прогонки

Для трёхдиагональной матрицы **10000x10000** и **500** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

2

The   parallel time: 0.093647 seconds

The non parallel time: 0.225281 seconds

Для трёхдиагональной матрицы **1000x1000** и **1000** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

2

The   parallel time: 0.023071 seconds

The non parallel time: 0.045992 seconds

Для трёхдиагональной матрицы **100000x100000** и **20** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

2

The   parallel time: 0.044745 seconds

The non parallel time: 0.099345 seconds

Тесты для метода простых итераций

Для матрицы **4x4** и **1** вектора правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

3

What TheMethod do you want to use: 1 - SimpleIter, 2 - Zaydel

>>> 1

with epsilon = 0.0001

The   parallel time: 0.004355 seconds

The non parallel time: 0.001014 seconds

Для матрицы **100x100** и **20** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

3

What TheMethod do you want to use: 1 - SimpleIter, 2 - Zaydel

>>> 1

with epsilon = 0.0001

The   parallel time: 4.87697 seconds

The non parallel time: 13.7048 seconds

Для матрицы **250x250** и **50** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

3

What TheMethod do you want to use: 1 - SimpleIter, 2 - Zaydel

>>> 1

with epsilon = 0.0001

The   parallel time: 124.129 seconds

The non parallel time: 288.886 seconds

Тесты для метода Зейделя

Для матрицы **4x4** и **1** вектора правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

3

What TheMethod do you want to use: 1 - SimpleIter, 2 - Zaydel

>>> 2

with epsilon = 0.0001

The   parallel time: 0.003361 seconds

The non parallel time: 0.000233 seconds

Для матрицы **100x100** и **20** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

3

What TheMethod do you want to use: 1 - SimpleIter, 2 - Zaydel

>>> 2

with epsilon = 0.0001

The   parallel time: 0.02808 seconds

The non parallel time: 0.087236 seconds

Для матрицы **250x250** и **50** векторов правой части

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

3

What TheMethod do you want to use: 1 - SimpleIter, 2 - Zaydel

>>> 2

```
with epsilon = 0.0001

The     parallel time: 0.205931 seconds

The non parallel time: 0.677579 seconds
```

Тесты для метода Якоби

```
Для матрицы 10x10

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

4

with epsilon = 0.0001

The     parallel time: 0.072203 seconds

The non parallel time: 0.023056 seconds

Для матрицы 25x25

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

4

with epsilon = 0.0001

The     parallel time: 4.27976 seconds

The non parallel time: 2.02129 seconds

Для матрицы 50x50

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

4

with epsilon = 0.1

The     parallel time: 36.6522 seconds

The non parallel time: 50.0746 seconds
```

Тесты для QR метода

```
Для матрицы 5x5

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

5

with epsilon = 0.01

The     parallel time: 0.977992 seconds

The non parallel time: 1.24852 seconds

Для матрицы 10x10

chouse test: 1 - LU, 2 - TMA, 3 - Iter, 4 - Jacobi, 5 - QR

5

with epsilon = 0.1

The     parallel time: 9.79842 seconds

The non parallel time: 16.4078 seconds
```

Вывод

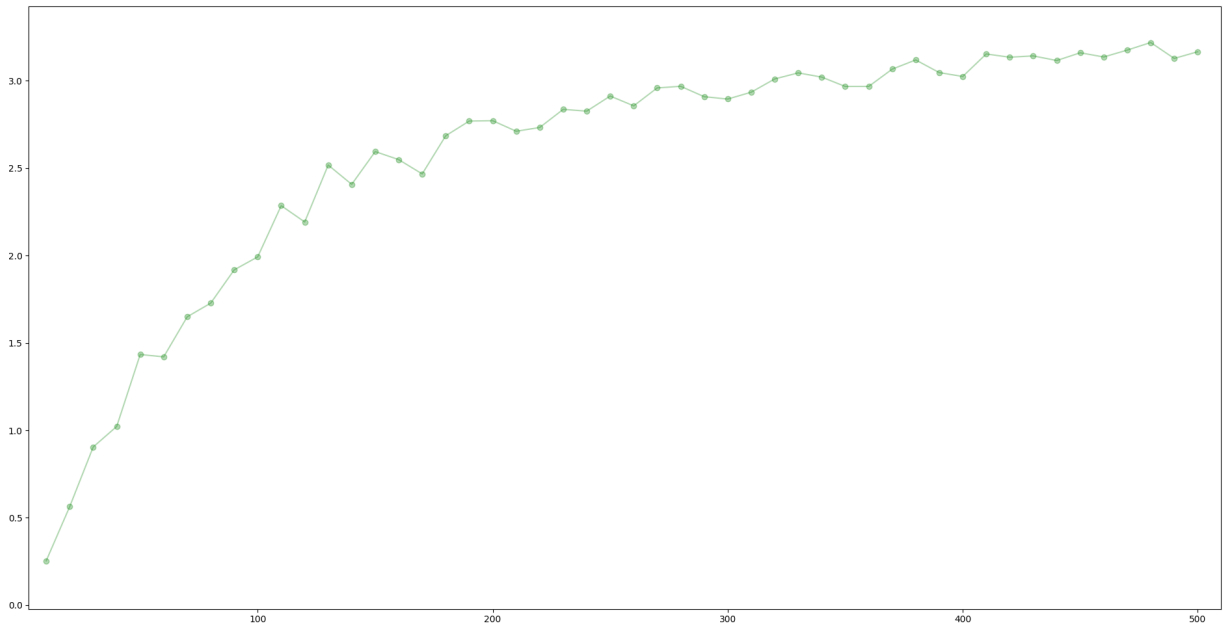
Из результатов можно сделать вывод, что методы параллельного программирования позволяют ускорить методы решения задач линейной алгебры.

Представим результаты исследований в виде графиков зависимости отношения T_p к T_{np} от размера матрицы, где

T_p – время работы паралельного алгоритма

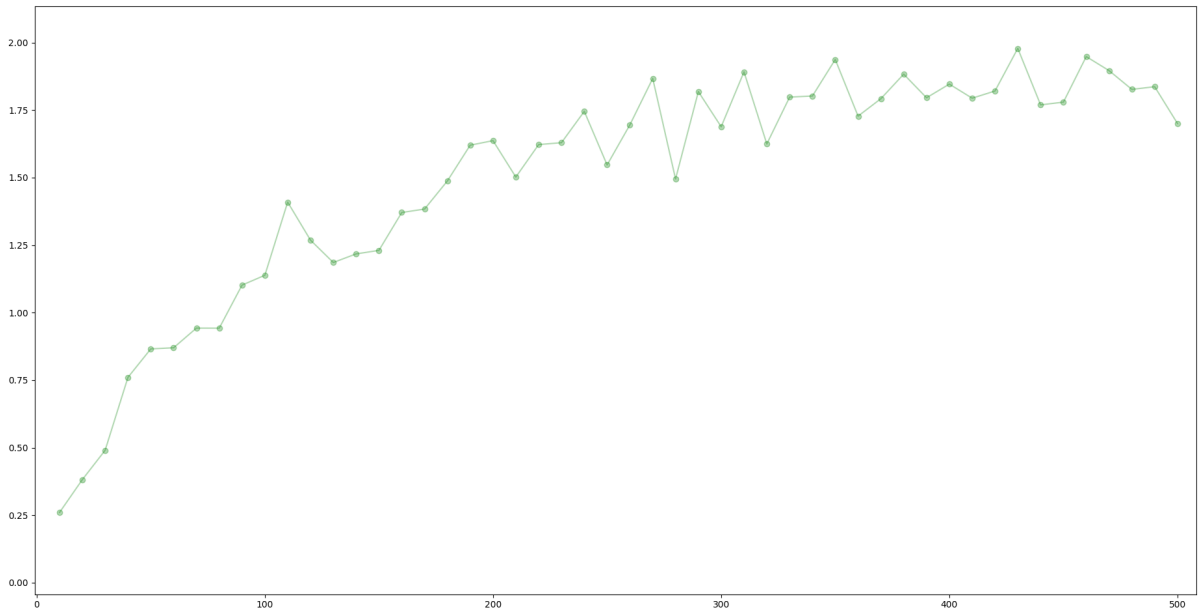
T_{np} – время работы не паралельного алгоритма

LU метод



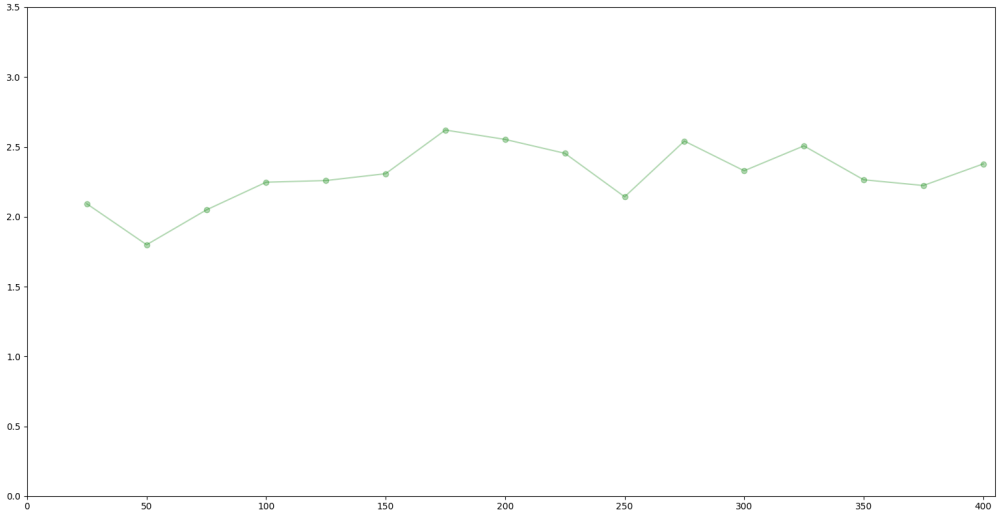
Из графика видно, что прирост скорости происходит только для матриц размером больше 50x50. В пределе прирост скорости равен 200%.

Метод прогонки



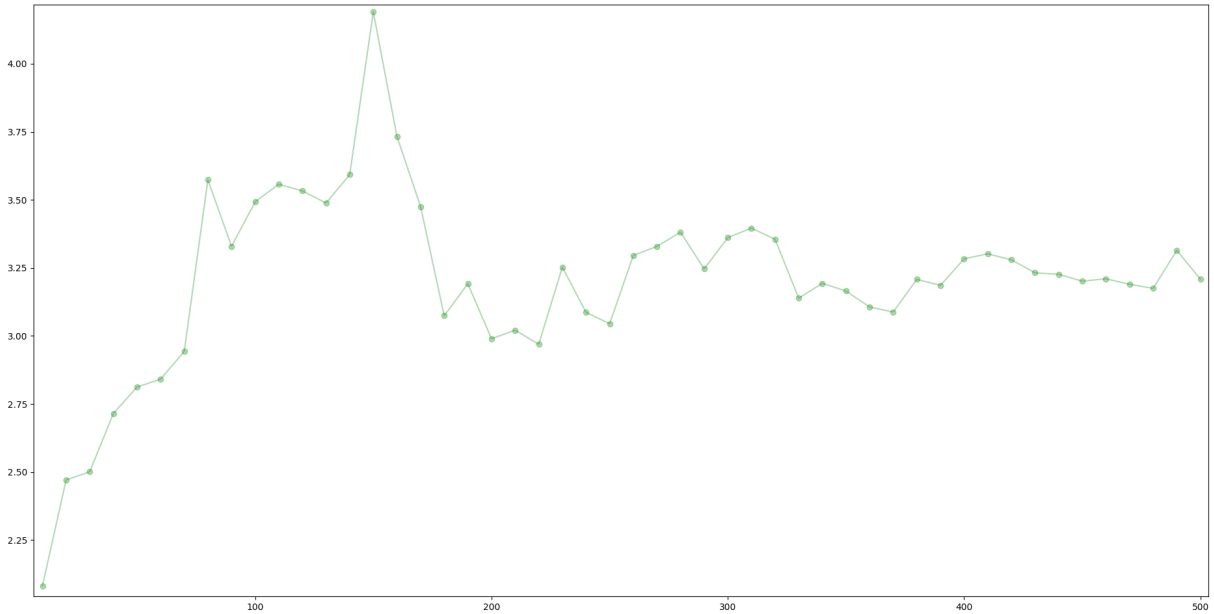
Из графика видно, что прирост скорости происходит только для матриц размером больше 100x100. Для матриц больших 100x100, в среднем, прирост скорости составляет 75-80%.

Метод простых итераций



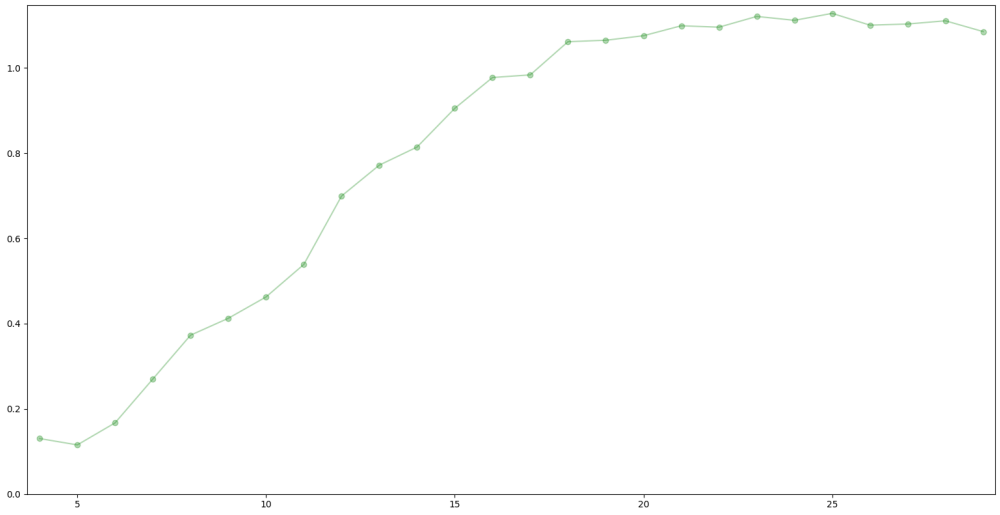
Из графика видно, что средний прирост скорости, в среднем, составляет 100%.

Метод Зейделя



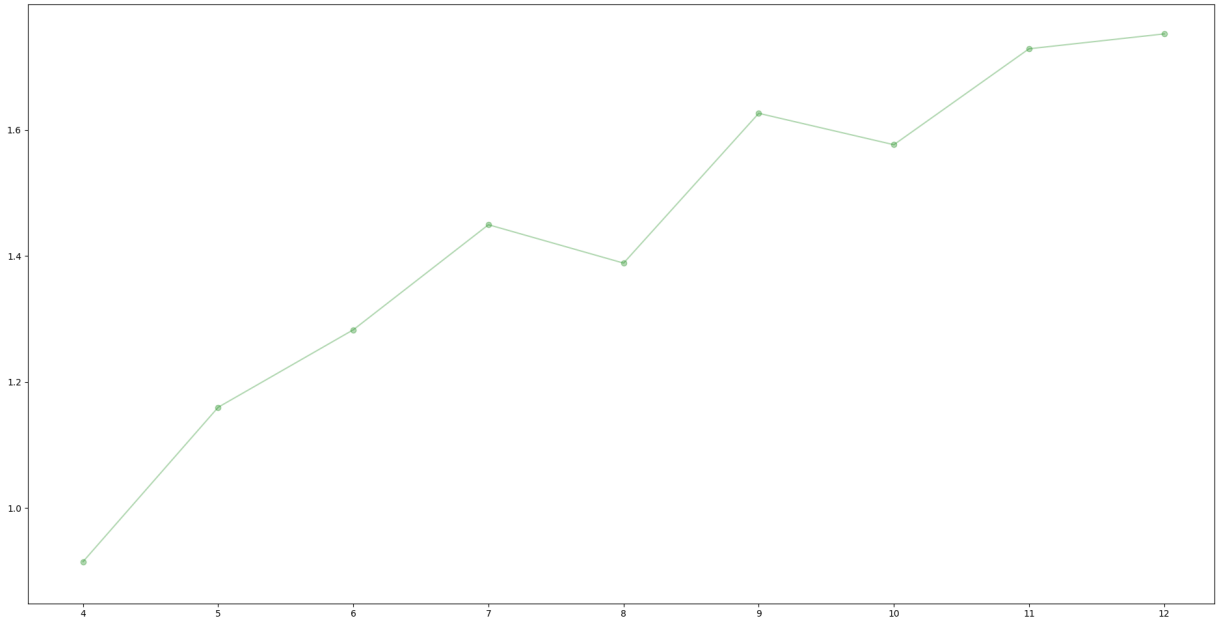
Из графика видно, что прирост скорости происходит для матриц любого размера. Наибольший прирост скорости был достигнут для матриц размером от 100x100 до 200x200. Для матриц рамзмером большим, чем 200x200 средний прирост составляет 225%.

Метод Якоби



Из графика видно, что прирост скорости происходит только для матриц размером больше 17x17. Для матриц больших 17x17, в среднем, прирост скорости составляет 15%

QR Метод



Из графика видно, что прирост скорости происходит только для матриц размером больше 4x4. Для матриц больших 17x17, в среднем, прирост скорости составляет 75%

Из исследований можно сделать вывод, что лучше всего методы параллельного программирования работают для методов итерации Зейделя. Хуже всего методы параллельного программирования удалось реализовать для метода Якоби.

Список литературы

1. Руководство к библиотеке OpenMP
2. Руководство к библиотеке Thread