МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP

студента 2 курса, группы 23201

Смирнова Гордея Андреевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.С. Матвеев

Новосибирск 2025

СОДЕРЖАНИЕ

ЦЕЛЬ

Практическое освоение методов распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области.

ЗАДАНИЕ

1. Последовательную программу из лабораторной работы 1, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP. Реализовать два варианта программы:

- Вариант 1: для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for,
- Вариант 2: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.
  Уделить внимание тому, чтобы при запуске программы на различном числе OpenMP-потоков решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

2. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: от 1 до числа доступных в узле. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
3. На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы.

ОПИСАНИЕ РАБОТЫ

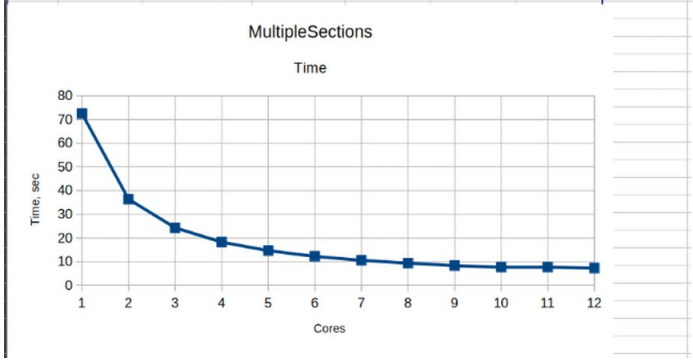
На языке C++ была написана OpenMP-программа для решения СЛАУ итерационным методом минимальных невязок. Были произведены замеры производительности программы при запуске на 1-12 потоках процессора. Результаты замеров приведены ниже. Исходный код программы приведён в приложениях.

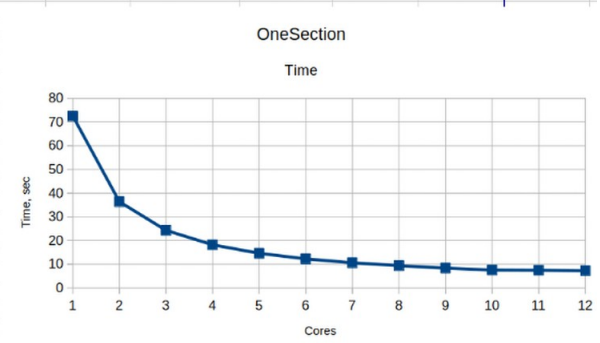
MultipleSections

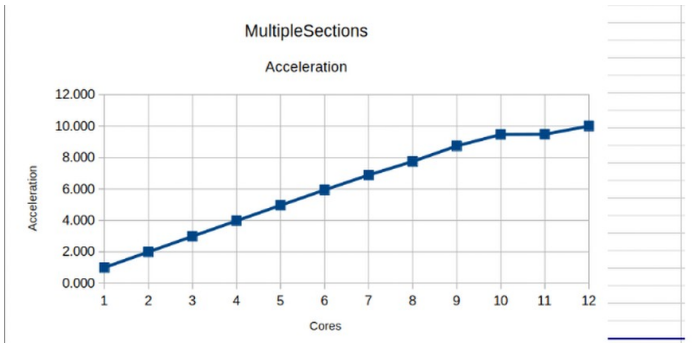
Cores	Time, sec	Acceleration	Efficiency
1	72.5	1.000	1.000
2	36.34	1.995	0.998
3	24.28	2.986	0.995
4	18.24	3.975	0.994
5	14.62	4.959	0.992
6	12.21	5.938	0.990
7	10.53	6.885	0.984
8	9.36	7.746	0.968
9	8.3	8.735	0.971
10	7.66	9.465	0.946
11	7.65	9.477	0.862
12	7.25	10.000	0.833

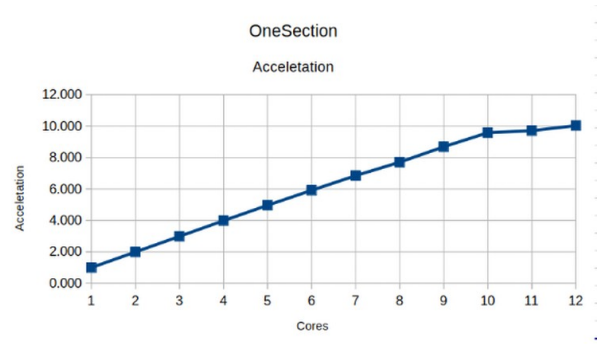
OneSection

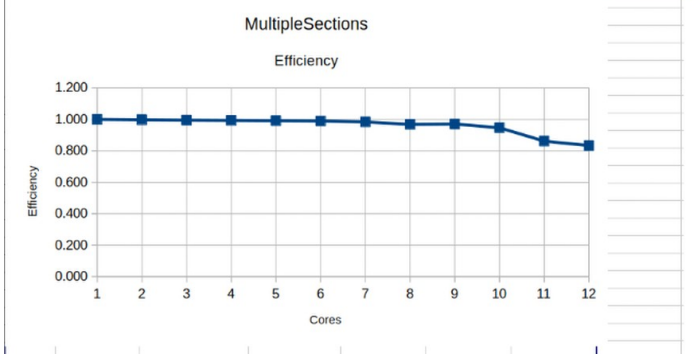
Cores	Time, sec	Acceleration	Efficiency
1	72.5	1.000	1.000
2	36.37	1.993	0.997
3	24.28	2.986	0.995
4	18.2	3.984	0.996
5	14.61	4.962	0.992
6	12.26	5.914	0.986
7	10.59	6.846	0.978
8	9.41	7.705	0.963
9	8.35	8.683	0.965
10	7.57	9.577	0.958
11	7.47	9.705	0.882
12	7.23	10.028	0.836

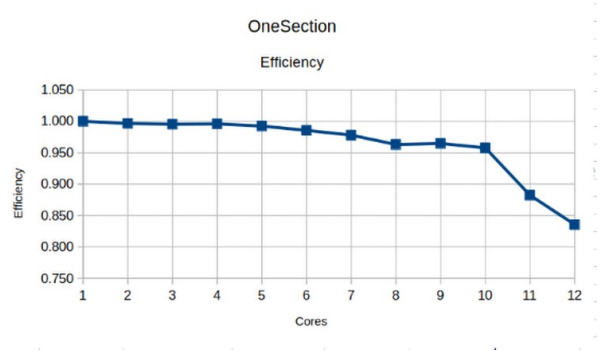












По результатам замеров реализации программы с одной/несколькими параллельными секциями практически не отличаются друг от друга – это означает, что в данной ситуации порождение новых потоков доставляет пренебрежимо малые дополнительные расходы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были изучены основные функции библиотеки OpenMP и получены базовые знания о параллельных вычислениях.

ПРИЛОЖЕНИЯ

Приложение 1: *Заголовочные файлы solvers*

```
// NonParallel.h
#pragma once

#include "SLE_Solver.h"

class NonParallel : public SLE_Solver {
 public:
    explicit NonParallel(const fvector& matA, const fvector& vecB);
    void solve() override;

 private:
    float norm(const fvector& x) override;
    void computeY() override;
    void computeTau() override;
    void computeX() override;
};


// MultipleSectionsOMP.h
#pragma once

#include "SLE_Solver.h"

class MultipleSectionsOMP : public SLE_Solver {
 public:
    explicit MultipleSectionsOMP(const fvector& matA, const fvector& vecB);
    void solve() override;

 private:
    float norm(const fvector& x) override;
    void computeY() override;
    void computeTau() override;
    void computeX() override;
};


// OneSectionOMP.h
#pragma once

#include "SLE_Solver.h"

class OneSectionOMP : public SLE_Solver {
 public:
    explicit OneSectionOMP(const fvector& matA, const fvector& vecB);
    void solve() override;
```

```
 private:
    float norm(const fvector& x) override;
    void computeY() override;
    void computeTau() override;
    void computeX() override;

    float numerator, denominator;
};
```

Приложение 2: *Остальные заголовочные файлы*

```
// BinIO.h
#pragma once

#include <string>
#include <vector>

class BinIO {
 public:
    using fvector = std::vector<float>;

    static fvector readVecFromBin(const std::string& filename);
    static void writeVecToBin(const fvector& vec,
                              const std::string& filename);
};


// SLE_Solver.h
#pragma once

#include <string>
#include <vector>

class SLE_Solver {
 public:
    using fvector = std::vector<float>;

    explicit SLE_Solver(const fvector& matA, const fvector& vecB);
    virtual ~SLE_Solver() = default;

    virtual void solve() = 0;
    [[nodiscard]] float getDiff(const std::string& vecXbin) const;
    [[nodiscard]] fvector getActualX() const;

 protected:
    const fvector& matA, vecB;
    const size_t N;
```

```
    fvector vecX, vecY;
    float tau;

    static constexpr float EPSILON = 1e-5;

 private:
    virtual float norm(const fvector& x) = 0;
    virtual void computeY() = 0;
    virtual void computeTau() = 0;
    virtual void computeX() = 0;
};
```

Приложение 3: ***SLE_Solver.cpp***

```
#include "SLE_Solver.h"

#include <filesystem>
#include <iostream>
#include <string>
#include "BinIO.h"

SLE_Solver::SLE_Solver(const fvector& matA, const fvector& vecB):
matA(matA), vecB(vecB), N(vecB.size()) {
    vecX = fvector(N);
    vecY = fvector(N);
    tau = 0;
}

float SLE_Solver::getDiff(const std::string& vecXbin) const {
    fvector expectedX;
    try {
        expectedX = BinIO::readVecFromBin(vecXbin);
        if (expectedX.size() != N) return -1;
    } catch (const std::runtime_error&) {
        return -1;
    }

    float diff = 0;
    for (size_t i = 0; i < N; i++) {
        diff += std::abs(expectedX[i] - vecX[i]);
    }
    return diff;
}

SLE_Solver::fvector SLE_Solver::getActualX() const {
    return vecX;
}
```

Приложение 4: ***NonParallel.cpp***

```
#include "NonParallel.h"

#include <cmath>

NonParallel::NonParallel(
    const fvector& matA, const fvector& vecB) : SLE_Solver(matA, vecB) {}

float NonParallel::norm(const fvector& x) {
    float result = 0;
    for (size_t i = 0; i < N; i++) {
        result += x[i] * x[i];
    }
    return std::sqrt(result);
}

void NonParallel::computeY() {
    for (size_t i = 0; i < N; i++) {
        float Ax_i = 0;
        for (size_t j = 0; j < N; j++) {
            Ax_i += matA[i * N + j] * vecX[j];
        }
        vecY[i] = Ax_i - vecB[i];
    }
}

void NonParallel::computeTau() {
    float numerator = 0, denominator = 0;
    for (size_t i = 0; i < N; i++) {
        float Ay_i = 0;
        for (size_t j = 0; j < N; j++) {
            Ay_i += matA[i * N + j] * vecY[j];
        }
        numerator += vecY[i] * Ay_i;
        denominator += Ay_i * Ay_i;
    }
    tau = numerator / denominator;
}

void NonParallel::computeX() {
    for (size_t i = 0; i < N; i++) {
        vecX[i] -= tau * vecY[i];
    }
}

void NonParallel::solve() {
    const float normB = norm(vecB);
    while (true) {
        computeY();
        if (norm(vecY) / normB < EPSILON) break;
        computeTau();
        computeX();
    }
```

```
}
```

Приложение 5: *MultipleSections.cpp*

```
#include "MultipleSectionsOMP.h"

#include <cmath>

MultipleSectionsOMP::MultipleSectionsOMP(
    const fvector& matA, const fvector& vecB) : SLE_Solver(matA, vecB) {}


float MultipleSectionsOMP::norm(const fvector& x) {
    float result = 0;
    #pragma omp parallel for default(none) shared(N, x) reduction(+:result)
    for (size_t i = 0; i < N; i++) {
        result += x[i] * x[i];
    }
    return std::sqrt(result);
}

void MultipleSectionsOMP::computeY() {
    #pragma omp parallel for default(none) shared(N, matA, vecB, vecX)
    for (size_t i = 0; i < N; i++) {
        float Ax_i = 0;
        for (size_t j = 0; j < N; j++) {
            Ax_i += matA[i * N + j] * vecX[j];
        }
        vecY[i] = Ax_i - vecB[i];
    }
}

void MultipleSectionsOMP::computeTau() {
    float numerator = 0, denominator = 0;
    #pragma omp parallel for default(none) shared(N, matA, vecY) \
    reduction(+:numerator, denominator)
    for (size_t i = 0; i < N; i++) {
        float Ay_i = 0;
        for (size_t j = 0; j < N; j++) {
            Ay_i += matA[i * N + j] * vecY[j];
        }
        numerator += vecY[i] * Ay_i;
        denominator += Ay_i * Ay_i;
    }
    tau = numerator / denominator;
}

void MultipleSectionsOMP::computeX() {
    #pragma omp parallel for default(none) shared(tau, vecX, vecY)
    for (size_t i = 0; i < N; i++) {
        vecX[i] -= tau * vecY[i];
    }
```

```
}

void MultipleSectionsOMP::solve() {
    const float normB = norm(vecB);
    while (true) {
        computeY();
        if (norm(vecY) / normB < EPSILON) break;
        computeTau();
        computeX();
    }
}
```

Приложение 6: *OneSection.cpp*

```
#include "OneSectionOMP.h"

#include <cmath>
#include <iostream>
#include <ostream>

OneSectionOMP::OneSectionOMP(
    const fvector& matA, const fvector& vecB) : SLE_Solver(matA, vecB) {
    numerator = denominator = 0;
}

float OneSectionOMP::norm(const fvector& x) {
    float result = 0;
    for (size_t i = 0; i < N; i++) {
        result += x[i] * x[i];
    }
    return std::sqrt(result);
}

void OneSectionOMP::computeY() {
    #pragma omp for
    for (size_t i = 0; i < N; i++) {
        float Ax_i = 0;
        for (size_t j = 0; j < N; j++) {
            Ax_i += matA[i * N + j] * vecX[j];
        }
        vecY[i] = Ax_i - vecB[i];
    }
}

void OneSectionOMP::computeTau() {
    #pragma omp for reduction(+:numerator, denominator)
    for (size_t i = 0; i < N; i++) {
        float Ay_i = 0;
        for (size_t j = 0; j < N; j++) {
            Ay_i += matA[i * N + j] * vecY[j];
        }
        numerator += vecY[i] * Ay_i;
```

```
        denominator += Ay_i * Ay_i;
    }
    #pragma omp single
    {
        tau = numerator / denominator;
        numerator = denominator = 0;
    }
}

void OneSectionOMP::computeX() {
    #pragma omp for
    for (size_t i = 0; i < N; i++) {
        vecX[i] -= tau * vecY[i];
    }
}

void OneSectionOMP::solve() {
    const float normB = norm(vecB);
    bool exit_flag = false;
    #pragma omp parallel default(none) \
    shared(matA, vecB, vecX, vecY, N, tau, normB, exit_flag)
    {
        while (!exit_flag) {
            computeY();

            float normY;
            #pragma omp single
            {
                normY = norm(vecY);
                if (normY / normB < EPSILON) exit_flag = true;
            }

            if (exit_flag) break;

            computeTau();
            computeX();
        }
    }
}
```

Приложение 7: *main.cpp*

```
#include <omp.h>
#include <iostream>
#include <chrono>
#include <string>
#include <vector>

#include "NonParallel.h"
#include "MultipleSectionsOMP.h"
#include "OneSectionOMP.h"
#include "BinIO.h"
```

```
template<typename SolverType>
void solveSLE(const std::vector<float>& matA, const std::vector<float>&
vecB) {
    SolverType solver(matA, vecB);
    const std::string type = typeid(SolverType).name();

    std::cout << "using " << type << " solver..." << std::endl;

    const auto start = std::chrono::system_clock::now();
    solver.solve();
    const auto end = std::chrono::system_clock::now();

    const double duration = std::chrono::duration_cast<
        std::chrono::duration<double>>(end - start).count();
    std::cout << "Done! Time taken: " << duration << " sec" << std::endl;

    BinIO::writeVecToBin(solver.getActualX(), "actualX.bin");

    const float diff = solver.getDiff("vecX.bin");
    const float avg = diff / static_cast<float>(vecB.size());
    std::cout << "Total diff: " << diff
              << " (avg " << avg << ")" << std::endl;
    std::cout << "--------------------------------" <<
                 "--------------------------------" << std::endl;
}

int main() {
    try {
        const std::vector<float> matA = BinIO::readVecFromBin("matA.bin");
        const std::vector<float> vecB = BinIO::readVecFromBin("vecB.bin");

        solveSLE<NonParallel>(matA, vecB);

        const int maxThreads = omp_get_max_threads();

        for (int t = 2; t ⩽ maxThreads; ++t) {
            omp_set_num_threads(t);
            std::cout << "Set " << t << " threads" << std::endl;
            solveSLE<MultipleSectionsOMP>(matA, vecB);
        }

        for (int t = 2; t ⩽ maxThreads; ++t) {
            omp_set_num_threads(t);
            std::cout << "Set " << t << " threads, ";
            solveSLE<OneSectionOMP>(matA, vecB);
        }
    } catch (const std::runtime_error& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Приложение *8: CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.29)
project(lab2)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_FLAGS "-O3 -march=native")

find_package(OpenMP REQUIRED)

set(SRC
        src/BinIO.cpp
        src/SLE_Solver.cpp
        src/solvers/NonParallel.cpp
        src/solvers/MultipleSectionsOMP.cpp
        src/solvers/OneSectionOMP.cpp
)

add_executable(${PROJECT_NAME} ${SRC} src/main.cpp)
target_include_directories(${PROJECT_NAME} PRIVATE
        src/include
        src/include/solvers
)
target_link_libraries(${PROJECT_NAME} PRIVATE OpenMP::OpenMP_CXX)
```