###### 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 ЛАБОРАТОРНОЙ РАБОТЫ

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»

студента 2 курса, 22208 группы

**Лебедева Антона Андреевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Кудинов

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc18443921)

[ЗАДАНИЕ 3](#_Toc18443922)

[ГРАФИКИ 4](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ 6](#_Toc18443924)

[Приложение Листинг 7](#_Toc18443925)

# ЦЕЛЬ

Разработать и исследовать параллельные программы решения СЛАУ методом простой итерации с применением одной из библиотек, реализующих стандарты OpenMP.

# ЗАДАЧА

# Написать последовательную программу, 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.

# 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.

# Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.ГРАФИКИ

Как видно из графика наиболее эффективным для данной задачи планированием Schedule является static с разбиением на фрагменты по 10 итераций. Самым неэффективным является метод Guided с разбиением на фрагменты по 500 итераций.

**ЗАКЛЮЧЕНИЕ**

# В результате выполнения работы была освоена технология OMP, распараллелено решение СЛАУ методом простой итерации, замерено ускорение и эффективность программы, проведено исследование по использованию целесообразности использования разных параметров Schedule в зависимости от размера вычисляемых данных и количества потоков.

Было выяснено, что наиболее эффективным вариантом реализации программы является создание одной параллельной секции #pragma omp parallel, охватывающей весь итерационный алгоритм. Это связано с тем, что программа не затрачивает каждый раз время на выделение нужного количества потоков, как в первом варианте. При использовании Shedule удалось добиться незначительного ускорения с помощью метода Static с разбиением на фрагменты по 10 итераций. Исходя из графика мы можем сделать вывод, что основная форма планирования является эффективной.

# ЛИСТИНГ

# Вариант 1

#include <omp.h>

#include <cmath>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <iostream>

#include <vector>

using matrix\_cont = std::vector<std::vector<double>>;

const std::size\_t N = 1200;

const double e = 0.000001;

double t = 0.001;

matrix\_cont MatrixBuilder() {

    matrix\_cont matrix(N, std::vector<double>(N));

#pragma omp parallel for

    for (std::size\_t i = 0; i < N; ++i) {

        for (std::size\_t j = 0; j < N; ++j) {

            if (j == i) {

                matrix[i][j] = 2.0;

            } else {

                matrix[i][j] = 1.0;

            }

        }

    }

    return matrix;

}

void RandomVectorX(double\* x) {

    srand(time(NULL));

#pragma omp parallel for

    for (std::size\_t i = 0; i < N; ++i) {

        x[i] = rand() % 10;

    }

}

void AMultX(matrix\_cont& matrix, double\* x, double\* b) {

#pragma omp parallel for

    for (std::size\_t i = 0; i < N; ++i) {

        for (std::size\_t j = 0; j < N; ++j) {

            b[i] += x[j] \* matrix[i][j];

        }

    }

}

void Sub(double\* x1, double\* b) {

#pragma omp parallel for

    for (std::size\_t i = 0; i < N; ++i) {

        x1[i] -= b[i];

    }

}

void MultT(double\* x1) {

#pragma omp parallel for

    for (std::size\_t i = 0; i < N; ++i) {

        x1[i] \*= t;

    }

}

double Module(double\* u) {

    double a = 0;

#pragma omp parallel for reduction(+ : a)

    for (std::size\_t i = 0; i < N; ++i) {

        a += (u[i] \* u[i]);

    }

    return sqrt(a);

}

void SearchX(matrix\_cont& matrix, double\* x, double\* b) {

    std::size\_t count = 0;

    double x0[N] = {0};

    double x1[N] = {0};

    double u = 0;

    double uOld = 0;

    double bNorm = 0;

    AMultX(matrix, x0, x1);

    Sub(x1, b);

    bNorm = Module(b);

    do {

        if (u > uOld && (count % 5 == 0)) {

            t = (-t);

        }

        uOld = u;

        MultT(x1);

        Sub(x0, x1);

        AMultX(matrix, x0, x1);

        Sub(x1, b);

        u = Module(x1);

        u = u / bNorm;

        count++;

    } while (u > e);

    std::cout << "u:" << u << std::endl;

    Sub(x0, x);

    u = Module(x0);

    std::cout << "NormalX: " << u << std::endl;

}

int main(int argc, char\*\* argv) {

    double startTime, endTime;

    double x[N];

    double b[N] = {0};

    startTime = omp\_get\_wtime();

    matrix\_cont matrix = MatrixBuilder();

    RandomVectorX(x);

    AMultX(matrix, x, b);

    SearchX(matrix, x, b);

    endTime = omp\_get\_wtime();

    std::cout << "Time: " << endTime - startTime << std::endl;

    return 0;

}

# Вариант 2

#include <omp.h>

#include <cmath>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <iostream>

#include <vector>

using matrix\_cont = std::vector<std::vector<double>>;

const std::size\_t N = 1200;

const double e = 0.000001;

double t = 0.001;

matrix\_cont MatrixBuilder() {

    matrix\_cont matrix(N, std::vector<double>(N));

#pragma omp for

    for (std::size\_t i = 0; i < N; ++i) {

        for (std::size\_t j = 0; j < N; ++j) {

            if (j == i) {

                matrix[i][j] = 2.0;

            } else {

                matrix[i][j] = 1.0;

            }

        }

    }

    return matrix;

}

void RandomVectorX(double\* x) {

    srand(time(NULL));

#pragma omp for

    for (std::size\_t i = 0; i < N; ++i) {

        x[i] = rand() % 10;

    }

}

void AMultX(matrix\_cont& matrix, double\* x, double\* b) {

#pragma omp for

    for (std::size\_t i = 0; i < N; ++i) {

        for (std::size\_t j = 0; j < N; ++j) {

            double temp = x[j] \* matrix[i][j];

            b[i] += temp;

        }

    }

}

void Sub(double\* x1, double\* b) {

#pragma omp for

    for (std::size\_t i = 0; i < N; ++i) {

        x1[i] -= b[i];

    }

}

void MultT(double\* x1) {

#pragma omp for

    for (std::size\_t i = 0; i < N; ++i) {

        x1[i] \*= t;

    }

}

void Module(double\* u, double& sum) {

#pragma omp for reduction(+ : sum)

    for (std::size\_t i = 0; i < N; ++i) {

        double temp = u[i] \* u[i];

        sum += temp;

    }

#pragma omp master

    sum = sqrt(sum);

}

void SearchX(matrix\_cont& matrix, double\* x, double\* b) {

    std::size\_t count = 0;

    double x0[N] = {0};

    double x1[N] = {0};

    double u = 0;

    double u2 = 0;

    double uOld = 0;

#pragma omp parallel

    {

        AMultX(matrix, x0, x1);

        Sub(x1, b);

        Module(b, u2);

        do {

            if (u > uOld && (count % 5 == 0)) {

#pragma omp master

                t = (-t);

            }

            uOld = u;

            MultT(x1);

            Sub(x0, x1);

            AMultX(matrix, x0, x1);

            Sub(x1, b);

            Module(x1, u);

#pragma omp master

            u = u / u2;

#pragma omp master

            count++;

#pragma omp barrier

        } while (u > e);

#pragma omp master

        std::cout << "u:" << u << std::endl;

        Sub(x0, x);

        Module(x0, u);

    }

    std::cout << "NormalX: " << u << std::endl;

}

int main(int argc, char\*\* argv) {

    double startTime, endTime;

    double x[N];

    double b[N] = {0};

    startTime = omp\_get\_wtime();

    matrix\_cont matrix = MatrixBuilder();

    RandomVectorX(x);

    AMultX(matrix, x, b);

    SearchX(matrix, x, b);

    endTime = omp\_get\_wtime();

    std::cout << "Time: " << endTime - startTime << std::endl;

    return 0;

}