###### 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, 22202 группы

**Бальчинова А.С.**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

В.А. Перепёлкин

Новосибирск 2024

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc18443921)

[ЗАДАНИЕ 3](#_Toc18443922)

[ОПИСАНИЕ РАБОТЫ 4-6](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ 6](#_Toc18443924)

[Приложение. 7](#_Toc18443925)-12

**ЦЕЛЬ**

Ознакомиться со стандартом OpenMP, изучить его основные директивы и проверить эффективность его применения на практике.

# 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)

1. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер. 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.
2. На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы.

# ОПИСАНИЕ РАБОТЫ

Было написано два варианта распараллеливания итерационного алгоритма:

• 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.

Затем были произведены замеры времени работы итерационного алгоритма при использовании различного числа процессорных ядер.

Построены графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Число потоков | 1 | 2 | 4 | 8 | 16 |
| Время выполнения 1 | 35,910 | 18,060 | 9,661 | 5,064 | 4,027 |
| Ускорение S | 1 | 1,988 | 3,717 | 7,09 | 8,917 |
| Эффективность E, % | 100 | 98.5 | 92.9 | 88,6 | 55,7 |
| Время выполнения 2 | 35,300 | 17,737 | 9,146 | 4,895 | 3,849 |
| Ускорение, раз | 1 | 1,99 | 3,859 | 7,21 | 9,171 |
| Эффективность, % | 100 | 99,5 | 96,4 | 90,12 | 57,32 |

# 

# 

# Как можно видеть из графиков, ускорение в обоих вариантах растёт примерно одинаково.

# Падение эффективности распараллеливания также примерно одинаковое. Второй вариант слегка превосходит первый по скорости.

# Возможно это связано с тем, что для реализиции второго варианта некоторые операции пришлось вынести в тело цикла, тем самым уменьшив время на передачу аргументов. Довольно резкое падение эффективности на 16 потоках может являться результатом гипертрединга.

# ЗАКЛЮЧЕНИЕ

В процессе выполнения лабораторной работы мы познакомились с OpenMP, научились применять его директивы на практике, а также сделали вывод об эффективности его использования.

# ПРИЛОЖЕНИЕ 1 (Вариант 1)

#include <iostream>

#include <chrono>

#include <cmath>

#include <omp.h>

#define N 15360

using namespace std;

using chrono::high\_resolution\_clock;

double \*initMatrix() {

    int i, j;

    double \*A = new double[N \* N];

#pragma omp parallel for private(i, j)

    for (i = 0; i < N; i++) {

        for (j = 0; j < N; j++) {

            if (i == j) {

                A[i \* N + j] = 2.0;

            }

            else {

                A[i \* N + j] = 1.0;

            }

        }

    }

    return A;

}

double \*initVector(double value) {

    double \*res = new double[N];

    int i;

#pragma omp parallel for private(i)

    for (i = 0; i < N; i++) {

        res[i] = value;

    }

    return res;

}

void mult(const double \*matrix, const double \*vect, double \*result) {

    int i, j;

#pragma omp parallel for private(i, j)

    for (i = 0; i < N; i++) {

        result[i] = 0.0;

        for (j = 0; j < N; j++) {

            result[i] += matrix[i \* N + j] \* vect[j];

        }

    }

}

void mult(double \*a, const double tau) {

    int i;

#pragma omp parallel for private(i)

    for (i = 0; i < N; i++) {

        a[i] \*= tau;

    }

}

void sub(double \*a, const double \*b) {

    int i;

#pragma omp parallel for private(i)

    for (i = 0; i < N; i++) {

        a[i] -= b[i];

    }

}

double Norm(const double \*u) {

    double result = 0.0;

    int i;

#pragma omp parallel for reduction(+:result) private(i)

    for (i = 0; i < N; i++) {

        result += u[i] \* u[i];

    }

    return sqrt(result);

}

struct Context {

    double \*A;

    double \*x;

    double \*b;

    double \*v;

    double tau;

    const double epsilon;

    double Norm\_b;

    Context(const double epsilon) : epsilon(epsilon) {

        A = initMatrix();

        x = initVector(0.0);

        b = initVector(N + 1.0);

        v = initVector(0.0);

        Norm\_b = Norm(b);

        tau = 0.0001;

    }

    ~Context() {

        delete[] A;

        delete[] x;

        delete[] b;

        delete[] v;

    }

};

bool isCloseEnough(Context &cont) {

    mult(cont.A, cont.x, cont.v);   //v = A\*x^n

    sub(cont.v, cont.b);            //v = A\*x^n - b

    double res = Norm(cont.v) / cont.Norm\_b;

    return res < cont.epsilon;

}

void next(Context &cont) {

    mult(cont.A, cont.x, cont.v);   //v = A\*x^n

    sub(cont.v, cont.b);            //v = A\*x^n - b

    mult(cont.v, cont.tau);         //v = tau\*(A\*x^n - b)

    sub(cont.x, cont.v);            //x = x - tau\*(A\*x^n - b)

}

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

    const double epsilon = pow(10, -9);

    Context cont = Context(epsilon);

    high\_resolution\_clock::time\_point begin = high\_resolution\_clock::now();

    while (!isCloseEnough(cont)) {

        next(cont);

    }

    high\_resolution\_clock::time\_point end = high\_resolution\_clock::now();

    cout << "Time diff = " << chrono::duration\_cast<chrono::milliseconds>(end - begin).count() << "[ms]" <<endl;

}

# ПРИЛОЖЕНИЕ 2 (Вариант 2)

#include <iostream>

#include <chrono>

#include <cmath>

#include <omp.h>

#define N 15360

using namespace std;

using chrono::high\_resolution\_clock;

double \*initMatrix() {

    double \*A = new double[N \* N];

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            if (i == j) {

                A[i \* N + j] = 2.0;

            }

            else {

                A[i \* N + j] = 1.0;

            }

        }

    }

    return A;

}

double \*initVector(double value) {

    double \*res = new double[N];

    for (int i = 0; i < N; i++) {

        res[i] = value;

    }

    return res;

}

void mult(const double \*matrix, const double \*vect, double \*result) { //Matrix \* Vector

    int i, j;

#pragma omp for private(i, j)

    for (i = 0; i < N; i++) {

        result[i] = 0.0;

        for (j = 0; j < N; j++) {

            result[i] += matrix[i \* N + j] \* vect[j];

        }

    }

}

void mult(double \*a, const double tau) {

    int i;

#pragma omp for private(i)

    for (i = 0; i < N; i++) {

        a[i] \*= tau;

    }

}

void sub(double \*a, const double \*b) {

    int i;

#pragma omp for private(i)

    for (i = 0; i < N; i++) {

        a[i] -= b[i];

    }

}

double Norm(const double \*u) {  //Non-parallel function

    double result = 0.0;

    for (int i = 0; i < N; i++) {

        result += u[i] \* u[i];

    }

    return sqrt(result);

}

struct Context {

    double \*A;

    double \*x;

    double \*b;

    double \*v;

    double tau;

    const double epsilon;

    double Norm\_b;

    Context(const double epsilon) : epsilon(epsilon) {

        A = initMatrix();

        x = initVector(0.0);

        b = initVector(N + 1.0);

        v = initVector(0.0);

        Norm\_b = Norm(b);

        tau = 0.0001;

    }

    ~Context() {

        delete[] A;

        delete[] x;

        delete[] b;

        delete[] v;

    }

};

void next(Context &cont) {

    mult(cont.A, cont.x, cont.v);   //v = A\*x^n

    sub(cont.v, cont.b);            //v = A\*x^n - b

    mult(cont.v, cont.tau);         //v = tau\*(A\*x^n - b)

    sub(cont.x, cont.v);            //x = x - tau\*(A\*x^n - b)

}

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

    const double epsilon = pow(10, -9);

    Context cont = Context(epsilon);

    double result = 1.0;

    high\_resolution\_clock::time\_point begin = high\_resolution\_clock::now();

    #pragma omp parallel

    {

        do {

            next(cont);

            mult(cont.A, cont.x, cont.v);

            sub(cont.v, cont.b);

        #pragma omp single

            result = 0.0;

        #pragma omp for reduction(+:result)

            for (int i = 0; i < N; i++) {

                result += cont.v[i] \* cont.v[i];

            }

        #pragma omp single

            {

                result = sqrt(result);

                result /= cont.Norm\_b;

            }

        }

        while (result >= cont.epsilon);

    }

    high\_resolution\_clock::time\_point end = high\_resolution\_clock::now();

    cout << "Time diff = " << chrono::duration\_cast<chrono::milliseconds>(end - begin).count() << "[ms]" <<endl;

}