## 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**

ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ

студента 2 курса, 22201 группы

**Рабецкого Валерия Дмитриевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель: А.С. Матвеев

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЗАДАНИЕ 3](#_TOC_250003)

[ОПИСАНИЕ РАБОТЫ 3](#_TOC_250002)

[ЗАКЛЮЧЕНИЕ 5](#_TOC_250001)

[Листинг программы 6](#_TOC_250000)

**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: 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.

1. Провести исследование на определение оптимальных параметров

#pragma omp for schedule(...) при некотором фиксированном размере

задачи и количестве потоков.

1. На основании полученных результатов сделать вывод о

целесообразности использования первого или второго варианта

программы.

**ОПИСАНИЕ РАБОТЫ**

Написал последовательную программу из лабораторной работы 1,

реализующую итерационный алгоритм решения системы линейных

алгебраических уравнений вида Ax=b, распараллелил с помощью

OpenMP.

Реализовал 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.

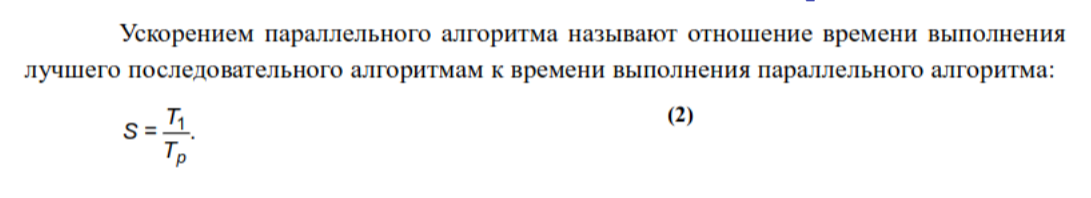
Замерил время работы двух вариантов программы при использовании

различного числа процессорных ядер: от 1 до числа доступных в узле.

Построил графики зависимости времени работы программы,

ускорения и эффективности распараллеливания от числа используемых

ядер.



|  |  |  |
| --- | --- | --- |
|  | Ускорение для 1 варианта | Ускорение для 2 варианта |
| 1 | 1 | 1 |
| 2 | 1,962962963 | 1,944444444 |
| 3 | 2,717948718 | 2,837837838 |
| 4 | 3,212121212 | 3,620689655 |
| 5 | 3,655172414 | 4,038461538 |
| 6 | 3,925925926 | 4,565217391 |
| 7 | 4,416666667 | 5 |
| 8 | 4,608695652 | 5,526315789 |
| 9 | 5,047619048 | 5,25 |
| 10 | 5,3 | 5,526315789 |
| 11 | 5,578947368 | 5,833333333 |
| 12 | 5,578947368 | 4,772727273 |

Эффективность = ускорение / количество потоков

|  |  |  |
| --- | --- | --- |
|  | Эффективность для 1 варианта | Эффективность для 2 варианта |
| 1 | 1 | 1 |
| 2 | 0,981481481 | 0,972222222 |
| 3 | 0,905982906 | 0,945945946 |
| 4 | 0,803030303 | 0,905172414 |
| 5 | 0,731034483 | 0,807692308 |
| 6 | 0,654320988 | 0,760869565 |
| 7 | 0,630952381 | 0,714285714 |
| 8 | 0,576086957 | 0,690789474 |
| 9 | 0,560846561 | 0,583333333 |
| 10 | 0,53 | 0,552631579 |
| 11 | 0,507177033 | 0,53030303 |
| 12 | 0,464912281 | 0,397727273 |

Провести исследование на определение оптимальных параметров

#pragma omp for schedule(...) при некотором фиксированном размере

задачи и количестве потоков.

|  |  |  |  |
| --- | --- | --- | --- |
|  | schedule(static) | schedule(dynamic) | schedule(guided) |
| 1 | 67 | 68 | 66 |
| 2 | 37 | 95 | 36 |
| 3 | 28 | 104 | 28 |
| 4 | 25 | 100 | 24 |
| 5 | 25 | 87 | 22 |
| 6 | 23 | 78 | 22 |
| 7 | 20 | 75 | 21 |
| 8 | 18 | 70 | 21 |
| 9 | 19 | 64 | 20 |
| 10 | 18 | 58 | 22 |
| 11 | 17 | 50 | 26 |
| 12 | 19 | 41 | 36 |

Самым оптимальным параметром для schedule (…) является static.

Static (Статическое планирование):

Описание: Все итерации цикла разбиваются на части, и каждая часть присваивается потоку статически, до начала выполнения.

Dynamic (Динамическое планирование):

Описание: Итерации цикла динамически распределяются между потоками во время выполнения. Каждый поток получает новую порцию итераций после завершения предыдущей.

Guided (Управляемое планирование):

Описание: Похоже на динамическое планирование, но порции итераций начинаются с большего размера и уменьшаются по мере продвижения. Это помогает уменьшить накладные расходы на управление в случае, если некоторые итерации выполняются значительно быстрее.

**Листинг программы**

**var1.cpp**

#include <iostream>

#include <cmath>

#include <chrono>

#include <fstream>

#include <omp.h>

#include "operations.h"

const std::size\_t VEC\_SIZE = 2500;

const std::size\_t MAT\_SIZE = 6250000;

const int NUM\_THREADS\_MAX = omp\_get\_max\_threads();

float\* readBinaryFile(const char\* filename) {

    float\* buffer = nullptr;

    std::ifstream file(filename, std::ios::binary);

    if (!file.is\_open()) {

        std::cerr << "Unable to open file: " << filename << std::endl;

        return nullptr;

    }

    file.seekg(0, std::ios::end);

    std::size\_t fileSize = file.tellg();

    file.seekg(0, std::ios::beg);

    buffer = new float[fileSize / sizeof(float)];

    file.read(reinterpret\_cast<char\*>(buffer), fileSize);

    file.close();

    return buffer;

}

void writeFloatArrayToFile(const char\* filename, float\* array, size\_t size) {

    FILE\* file = fopen(filename, "wb");

    if (file != NULL) {

        fwrite(array, sizeof(float), size, file);

        fclose(file);

    } else {

        printf("Не удалось открыть файл для записи\n");

    }

}

float\* Solve(float\*& A, float\*& b, float\* x, int size) {

    float\* final\_x = new float[size];

    FillZero(final\_x, size);

    float res = 1;

    float norm\_x\_squared = 0;

    float norm\_b = 0;

    for (int i = 0; i < size; ++i)

    {

        norm\_b += b[i] \* b[i];

    }

    norm\_b = sqrt(norm\_b);

    int iter = 0;

    while (res >= EPSILON) {

        #pragma omp parallel for reduction(+:norm\_x\_squared)

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

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

                final\_x[i] += A[i \* size + j] \* x[j];

            }

            final\_x[i] -= b[i];

            norm\_x\_squared += final\_x[i] \* final\_x[i];

            final\_x[i] = final\_x[i] \* THAU;

            final\_x[i] = x[i] - final\_x[i];

        }

        #pragma omp parallel for

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

            x[i] = final\_x[i];

        }

        res = sqrt(norm\_x\_squared) / norm\_b;

        norm\_x\_squared = 0;

        FillZero(final\_x, size);

        iter++;

    }

    std::cout << iter << "\n";

    delete[] final\_x;

    return x;

}

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

    float\* A;

    float\* b;

    float\* x;

    A = readBinaryFile("matA.bin");

    b = readBinaryFile("vecB.bin");

    x = new float[VEC\_SIZE];

    for (int numThreads = 1; numThreads <= NUM\_THREADS\_MAX; ++numThreads)

    {

        FillZero(x, VEC\_SIZE);

        omp\_set\_num\_threads(numThreads);

        std::chrono::high\_resolution\_clock clock;

        auto start = clock.now();

        x = Solve(A, b, x, VEC\_SIZE);

        auto end = clock.now();

        auto time = std::chrono::duration\_cast<std::chrono::seconds> (end - start);

        std::cout << time.count() << " s\n";

    }

    writeFloatArrayToFile("myVecX", x, VEC\_SIZE);

    delete[] x;

    delete[] A;

    delete[] b;

}

**var2.cpp**

#include <iostream>

#include <cmath>

#include <chrono>

#include <fstream>

#include <omp.h>

#include "operations.h"

const std::size\_t VEC\_SIZE = 2500;

const std::size\_t MAT\_SIZE = 6250000;

const int NUM\_THREADS\_MAX = omp\_get\_max\_threads();

float\* readBinaryFile(const char\* filename) {

    float\* buffer = nullptr;

    std::ifstream file(filename, std::ios::binary);

    if (!file.is\_open()) {

        std::cerr << "Unable to open file: " << filename << std::endl;

        return nullptr;

    }

    file.seekg(0, std::ios::end);

    std::size\_t fileSize = file.tellg();

    file.seekg(0, std::ios::beg);

    buffer = new float[fileSize / sizeof(float)];

    file.read(reinterpret\_cast<char\*>(buffer), fileSize);

    file.close();

    return buffer;

}

void writeFloatArrayToFile(const char\* filename, float\* array, size\_t size) {

    FILE\* file = fopen(filename, "wb");

    if (file != NULL) {

        fwrite(array, sizeof(float), size, file);

        fclose(file);

    } else {

        printf("Не удалось открыть файл для записи\n");

    }

}

float\* Solve(float\*& A, float\*& b, float\* x, int size) {

    float\* final\_x = new float[size];

    FillZero(final\_x, size);

    float res = 1;

    float norm\_x\_squared = 0;

    float norm\_b = 0;

    for (int i = 0; i < size; ++i)

    {

        norm\_b += b[i] \* b[i];

    }

    norm\_b = sqrt(norm\_b);

    int iter = 0;

    #pragma omp parallel

    {

        while (res >= EPSILON) {

            #pragma omp for reduction(+:norm\_x\_squared)

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

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

                    final\_x[i] += A[i \* size + j] \* x[j];

                }

                final\_x[i] -= b[i];

                norm\_x\_squared += final\_x[i] \* final\_x[i];

                final\_x[i] \*= THAU;

                final\_x[i] = x[i] - final\_x[i];

            }

            #pragma omp for

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

                x[i] = final\_x[i];

            }

            #pragma omp single

            {

                res = sqrt(norm\_x\_squared) / norm\_b;

                norm\_x\_squared = 0;

                FillZero(final\_x, size);

                iter++;

            }

            #pragma omp barrier

        }

    }

    std::cout << iter << "\n";

    delete[] final\_x;

    return x;

}

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

    float\* A;

    float\* b;

    float\* x;

    A = readBinaryFile("matA.bin");

    b = readBinaryFile("vecB.bin");

    x = new float[VEC\_SIZE];

    for (int numThreads = 1; numThreads <= NUM\_THREADS\_MAX; ++numThreads)

    {

        FillZero(x, VEC\_SIZE);

        omp\_set\_num\_threads(numThreads);

        std::chrono::high\_resolution\_clock clock;

        auto start = clock.now();

        x = Solve(A, b, x, VEC\_SIZE);

        auto end = clock.now();

        auto time = std::chrono::duration\_cast<std::chrono::seconds> (end - start);

        std::cout << time.count() << " s\n";

    }

    writeFloatArrayToFile("myVecX", x, VEC\_SIZE);

    delete[] x;

    delete[] A;

    delete[] b;

}

**operations.cpp**

#include <iostream>

#include <cmath>

#include <omp.h>

#include "operations.h"

void PrintMat(double\* mat, int size) {

    for (int i = 0; i < size; ++i) {

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

            std::cout << mat[i \* size + j] << "\t";

        }

        std::cout << "\n";

    }

}

void PrintVect(double\* vect, int size) {

    for (int i = 0; i < size; ++i) {

        std::cout << vect[i] << " ";

    }

}

void FillZero(double\*& array, int size) {

    for (int i = 0; i < size; ++i){

        array[i] = 0;

    }

}

void FillMatrix(double\*& matrix, int size) {

    for (int i = 0; i < size; ++i) {

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

            matrix[i \* size + j] = (i == j) ? 2.0 : 1.0;

        }

    }

}

void FillVectU(double\*& u, int size) {

    for (int i = 0; i < size; ++i){

        u[i] = sin((2 \* M\_PI \* i) / size);

    }

}

void FillVectB(double\*& b, double\*& u, double\*& matrix, int size) {

    for (int i = 0; i < size; ++i) {

        double temp = 0.0;

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

            temp += matrix[i \* size + j] \* u[j];

        }

        b[i] += temp;

    }

}