## 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.

|  |  |  |
| --- | --- | --- |
|  | #pragma omp parallel for | #pragma omp parallel |
| 1 | 66 | 67 |
| 2 | 36 | 38 |
| 3 | 28 | 29 |
| 4 | 25 | 25 |
| 5 | 23 | 23 |
| 6 | 22 | 21 |
| 7 | 20 | 20 |
| 8 | 18 | 18 |
| 9 | 17 | 16 |
| 10 | 15 | 17 |
| 11 | 15 | 15 |
| 12 | 15 | 15 |

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 <string>

#include <cmath>

#include <chrono>

#include <fstream>

#include <vector>

#include <omp.h>

#include "operations.h"

using std::vector;

const std::size\_t VEC\_SIZE = 300;

const std::size\_t MAT\_SIZE = 90000;

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

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

    double\* x = new double[size];

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

    FillZero(x, size);

    FillZero(final\_x, size);

    double res = EPSILON;

    double norm\_x\_squared = 0;

    double 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 schedule(guided) 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 = Ax

            }

            final\_x[i] -= b[i];                                 // final\_x = Ax - b

            norm\_x\_squared += final\_x[i] \* final\_x[i];          // ||Ax - b|| ^ 2

            final\_x[i] \*= THAU;                                 // final\_x = t(Ax - b)

            final\_x[i] = x[i] - final\_x[i];                     // final\_x = x - t(Ax - b)

        }

        #pragma omp parallel for schedule(guided)

        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) {

    double\* A = new double[MAT\_SIZE];

    double\* u = new double[VEC\_SIZE];

    double\* b = new double[VEC\_SIZE];

    FillMatrix(A, VEC\_SIZE);

    FillVectU(u, VEC\_SIZE);

    FillVectB(b, u, A, VEC\_SIZE);

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

    {

        omp\_set\_num\_threads(numThreads);

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

        auto start = clock.now();

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

        auto end = clock.now();

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

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

        delete[] x;

        }

    delete[] A;

    delete[] u;

    delete[] b;

}

**var2.cpp**

#include <iostream>

#include <string>

#include <cmath>

#include <chrono>

#include <fstream>

#include <thread>

#include <vector>

#include <omp.h>

#include "operations.h"

using std::vector;

const std::size\_t VEC\_SIZE = 300;

const std::size\_t MAT\_SIZE = 90000;

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

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

    double\* x = new double[size];

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

    FillZero(x, size);

    FillZero(final\_x, size);

    double res = EPSILON;

    double norm\_x\_squared = 0;

    double 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 = Ax

                }

                final\_x[i] -= b[i];                                 // final\_x = Ax - b

                norm\_x\_squared += final\_x[i] \* final\_x[i];          // ||Ax - b|| ^ 2

                final\_x[i] \*= THAU;                                 // final\_x = t(Ax - b)

                final\_x[i] = x[i] - final\_x[i];                     // final\_x = x - t(Ax - b)

            }

            #pragma omp for

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

                x[i] = final\_x[i];

            }

            FillZero(final\_x, size);

            #pragma omp single

            {

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

                norm\_x\_squared = 0;

                iter++;

            }

        }

    }

    std::cout << iter << "\n";

    delete[] final\_x;

    return x;

}

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

    double\* A = new double[MAT\_SIZE];

    double\* u = new double[VEC\_SIZE];

    double\* b = new double[VEC\_SIZE];

    FillMatrix(A, VEC\_SIZE);

    FillVectU(u, VEC\_SIZE);

    FillVectB(b, u, A, VEC\_SIZE);

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

    {

        omp\_set\_num\_threads(numThreads);

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

        auto start = clock.now();

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

        auto end = clock.now();

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

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

        delete[] x;

    }

    delete[] A;

    delete[] u;

    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;

    }

}