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, группы 22206

***Тропина Никиты Васильевича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

ассистент,

А.А.Ажбаков

Новосибирск 2023

# СОДЕРЖАНИЕ

[ЦЕЛИ 3](#_Toc160174383)

[ЗАДАНИЕ 3](#_Toc160174384)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc160174385)

[Приложение 1. Вариант программы с отдельными параллельными секциями для каждого цикла. 5](#_Toc160174386)

[Приложение 2. Вариант программы с одной параллельной секцией, охватывающей весь итерационный алгоритм. 8](#_Toc160174387)

# ЦЕЛИ

1. Определить эффективность распараллеливания программы с помощью OpenMP от числа используемых ядер.

# ЗАДАНИЕ

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.

На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы.

# ОПИСАНИЕ РАБОТЫ

Программа, реализующая итерационный алгоритм решения систем линейных алгебраических уравнений вида Ax=b, была распараллелена двумя вариантами: для каждого распараллеливаемого цикла создана отдельная параллельная секция #pragma omp parallel for (Приложение 1); весь итерационный алгоритм находится в одной параллельной секции #pragma omp parallel (Приложение 2).

Выполнение программы производилось на вычислительном кластере НОЦ «Газпромнефть-НГУ». Для измерения времени использовалась функция omp\_get\_wtime().

В качестве исходных для тестирования взята модельная задача с заданным решением: Элементы главной диагонали матрицы A равны 2.0, остальные равны 1.0. Все элементы вектора b равны N+1. В этом случае решением системы будет вектор, элементы которого равны 1.0. Начальные значения элементов вектора x равны 0. A – матрица размера N×N. Значение N = 16 384, время исполнения программы без распараллеливания = 47.7 с.

Результаты измерений представлены на диаграммах.

# Приложение 1. Вариант программы с отдельными параллельными секциями для каждого цикла.

#include <cstring>

#include <iostream>

#include <math.h>

#include <omp.h>

enum {

SIZE\_VECTOR = 4096 \* 4,

ZERO\_VALUE = 0

};

const double TAU = 1e-5;

const double EPSILON = 1e-5;

void printVector(const double\* vector, const int size)

{

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

std::cout << vector[i] << " ";

}

std::cout << std::endl;

}

double calculateDeterminant(const double\* matrix, const int size)

{

double determinant = 0;

#pragma omp parallel for reduction(+:determinant)

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

determinant += matrix[i] \* matrix[i];

}

return sqrt(determinant);

}

void fillVectorWithValue(double\* vector, const int size, const int value)

{

#pragma omp parallel for

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

vector[i] = value;

}

}

void vectorSubtraction(const double\* minuend, const double\* subtrahend, double\* result, const int size)

{

#pragma omp parallel for

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

result[i] = minuend[i] - subtrahend[i];

}

}

void multiplyVectorByScalar(double\* vector, double\* result, const int size, const double scalar)

{

#pragma omp parallel for

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

result[i] = vector[i] \* scalar;

}

}

void multiplyMatrixByVector(const double\* matrix, const double\* vector, double\* result, const int size)

{

fillVectorWithValue(result, size, ZERO\_VALUE);

#pragma omp parallel for

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

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

result[i] += matrix[i \* size + j] \* vector[j];

}

}

}

double\* generateZeroVector(const int size)

{

double\* vector = new double[size];

fillVectorWithValue(vector, size, ZERO\_VALUE);

return vector;

}

double\* generateSolutionVector(const int size)

{

double\* solutionVector = generateZeroVector(size);

fillVectorWithValue(solutionVector, size, ZERO\_VALUE);

return solutionVector;

}

double\* generateRightPartVector(const int size)

{

double\* rightPartVector = generateZeroVector(size);

fillVectorWithValue(rightPartVector, size, size + 1);

return rightPartVector;

}

double\* generateSquareMatrix(const int size)

{

double\* matrix = new double[size \* size];

#pragma omp parallel for

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

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

if (i == j)

matrix[i \* size + j] = 2;

else

matrix[i \* size + j] = 1;

}

}

return matrix;

}

bool isAccuracyAchieved(const double numerator, const double denominator)

{

return ((numerator / denominator) < EPSILON);

}

void copyVector(const double\* vector, double\* copy, const int size)

{

memcpy(copy, vector, size \* sizeof(double));

}

double\* iterationMethod(const double\* matrix, double\* rightPartVector, const int size)

{

double\* solutionVector = generateSolutionVector(size);

double\* solutionCopyVector = generateZeroVector(size);

double\* multiplyVector = generateZeroVector(size);

double rightPartDeterminant = calculateDeterminant(rightPartVector, size);

bool run = true;

do {

copyVector(solutionVector, solutionCopyVector, size);

multiplyMatrixByVector(matrix, solutionCopyVector, multiplyVector, size);

vectorSubtraction(multiplyVector, rightPartVector, multiplyVector, size);

multiplyVectorByScalar(multiplyVector, solutionCopyVector, size, TAU);

vectorSubtraction(solutionVector, solutionCopyVector, solutionVector, size);

run = !isAccuracyAchieved(calculateDeterminant(multiplyVector, size), rightPartDeterminant);

} while (run);

delete[] solutionCopyVector;

delete[] multiplyVector;

return solutionVector;

}

int main()

{

double time = 0;

int nIterations = 10;

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

double\* matrix = generateSquareMatrix(SIZE\_VECTOR);

double\* rightPartVector = generateRightPartVector(SIZE\_VECTOR);

double start = omp\_get\_wtime();

double\* solutionVector = iterationMethod(matrix, rightPartVector, SIZE\_VECTOR);

double finish = omp\_get\_wtime();

time += finish - start;

delete[] matrix;

delete[] solutionVector;

delete[] rightPartVector;

}

std::cout << "TIME: " << time / nIterations << std::endl;

}

# Приложение 2. Вариант программы с одной параллельной секцией, охватывающей весь итерационный алгоритм.

#include <cstring>

#include <iostream>

#include <math.h>

#include <omp.h>

enum {

SIZE\_VECTOR = 4096,

ZERO\_VALUE = 0

};

const double TAU = 1e-5;

const double EPSILON = 1e-5;

void printVector(const double\* vector, const int size)

{

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

std::cout << vector[i] << " ";

}

std::cout << std::endl;

}

double calculateDeterminant(const double\* matrix, const int size)

{

double determinant = 0;

#pragma omp parallel for reduction(+:determinant)

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

determinant += matrix[i] \* matrix[i];

}

return sqrt(determinant);

}

void fillVectorWithValue(double\* vector, const int size, const int value)

{

#pragma omp for

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

vector[i] = value;

}

}

void vectorSubtraction(const double\* minuend, const double\* subtrahend, double\* result, const int size)

{

#pragma omp for

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

result[i] = minuend[i] - subtrahend[i];

}

}

void multiplyVectorByScalar(double\* vector, double\* result, const int size, const double scalar)

{

#pragma omp for

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

result[i] = vector[i] \* scalar;

}

}

void multiplyMatrixByVector(const double\* matrix, const double\* vector, double\* result, const int size)

{

fillVectorWithValue(result, size, ZERO\_VALUE);

#pragma omp for

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

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

result[i] += matrix[i \* size + j] \* vector[j];

}

}

}

double\* generateZeroVector(const int size)

{

double\* vector = new double[size];

fillVectorWithValue(vector, size, ZERO\_VALUE);

return vector;

}

double\* generateSolutionVector(const int size)

{

double\* solutionVector = generateZeroVector(size);

fillVectorWithValue(solutionVector, size, ZERO\_VALUE);

return solutionVector;

}

double\* generateRightPartVector(const int size)

{

double\* rightPartVector = generateZeroVector(size);

fillVectorWithValue(rightPartVector, size, size + 1);

return rightPartVector;

}

double\* generateSquareMatrix(const int size)

{

double\* matrix = new double[size \* size];

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

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

if (i == j)

matrix[i \* size + j] = 2;

else

matrix[i \* size + j] = 1;

}

}

return matrix;

}

bool isAccuracyAchieved(const double numerator, const double denominator)

{

return ((numerator / denominator) < EPSILON);

}

void copyVector(const double\* vector, double\* copy, const int size)

{

memcpy(copy, vector, size \* sizeof(double));

}

double\* iterationMethod(const double\* matrix, double\* rightPartVector, const int size)

{

double\* solutionVector = generateSolutionVector(size);

double\* solutionCopyVector = generateZeroVector(size);

double\* multiplyVector = generateZeroVector(size);

double rightPartDeterminant = calculateDeterminant(rightPartVector, size);

bool run = true;

#pragma omp parallel

{

do {

copyVector(solutionVector, solutionCopyVector, size);

multiplyMatrixByVector(matrix, solutionCopyVector, multiplyVector, size);

vectorSubtraction(multiplyVector, rightPartVector, multiplyVector, size);

multiplyVectorByScalar(multiplyVector, solutionCopyVector, size, TAU);

vectorSubtraction(solutionVector, solutionCopyVector, solutionVector, size);

#pragma omp single

run = !isAccuracyAchieved(calculateDeterminant(multiplyVector, size), rightPartDeterminant);

} while (run);

}

delete[] solutionCopyVector;

delete[] multiplyVector;

return solutionVector;

}

int main()

{

double time = 0;

int nIterations = 10;

for (int i = 0; i < nIterations; i++)

{

double\* matrix = generateSquareMatrix(SIZE\_VECTOR);

double\* rightPartVector = generateRightPartVector(SIZE\_VECTOR);

double start = omp\_get\_wtime();

double\* solutionVector = iterationMethod(matrix, rightPartVector, SIZE\_VECTOR);

double finish = omp\_get\_wtime();

time += finish - start;

delete[] matrix;

delete[] solutionVector;

delete[] rightPartVector;

}

std::cout << "TIME: " << time / nIterations << std::endl;

}