###### 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, 20204 группы

**Дронова Дениса Юоьевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук

А. Ю. Власенко

Новосибирск 2022

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc99151371)

[ЗАДАНИЕ 3](#_Toc99151372)

[ХОД РАБОТЫ 4](#_Toc99151373)

[ЗАКЛЮЧЕНИЕ 5](#_Toc99151374)

[Приложение 1. *Таблицы и графики* 6](#_Toc99151375)

[Приложение 2. *Код программы (без schedule)* 7](#_Toc99151376)

[Приложение 3. *Код программы (c schedule)* 11](#_Toc99151377)

# ЦЕЛЬ

Ознакомиться с принципами работы OpenMP. Реализовать программу решения системы линейных алгебраических уравнений, используя функционал OpenMP. Оценить эффективность и ускорение программ при различном числе потоков. Определить наиболее подходящий способ разделения данных между процессами (#pragma omp for schedule(…)).

# ЗАДАНИЕ

1. Распараллелить программу (последовательную из лабораторной работы №1) с помощью OpenMP при условии установки директивы #pragma omp parallel на весь итерационный алгоритм.
2. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 12, 16. Исходные данные выбрать так, чтобы программа работала на одном ядре не менее 30 секунд.
3. Провести исследование на определение оптимальных параметров #pragma omp for schedule при фиксированном размере задачи и количестве потоков.

# ХОД РАБОТЫ

Из лабораторной работы была позаимствована программа последовательного алгоритма решения СЛАУ методом сопряженных градиентов, которая впоследствии была распараллелена при помощи директив OpenMP. Параллельная программа была протестирована на корректность решения при различном числе потоков и различных размерах матрицы. Был подобран параметр N=1400 так, что на одном потоке время решения составило ~40 секунд. Были произведены замеры времени исполнения программы при числе потоков равному 1, 2, 4, 8, 12, 16. Исходя из полученных результатов были построены таблица и графики времени исполнения, эффективности, ускорения в зависимости от числа потоков (см. Приложение 1). Также программа была запущена при неизменном числе потоков (4), но при различных параметрах #pragma omp for schedule(…) (static, dynamic, guided). Результаты замеров времени были отображены в таблице.

|  |  |
| --- | --- |
| **Параметр** | **Время** |
| static | 9,831032 |
| dynamic | 21,566581 |
| guided | 10,191126 |

# **ЗАКЛЮЧЕНИЕ**

После анализа всех полученных после тестирования данных можно сделать вывод о том, что использование директив OpenMP для распараллеливания является более быстрым в реализации методом ускорения вычислений. Однако можно проследить тенденцию снижения ускорения и эффективности с увеличением числа потоков. Такая тенденция объясняется ограниченным числом ядер на узле, так как при создании большого числа потоков они лишь будут больше простаивать в очереди, ожидая от процессора разрешения на выполнение своих задач. Также порождение большого числа потоков менее эффективно в силу специфики OpenMP, а именно задействования при вычислениях не более чем одного узла (отсутствие межузлового общения). Таким образом накладывается ограничение на число используемых программой ядер. По результатам анализа программ с разными параметрами #pragma omp for schedule(…) можно сделать вывод о том, что в условиях исследуемого алгоритма наиболее оптимальным параметром является static. Такой вывод вытекает из однородности данных, распределяемых между потоками. Предположительно параметр dynamic показывает наихудший результат по времени исполнения из-за накладных расходов на аналитические расчеты, необходимые при принятии решений на распределение итераций между потоками, что в рамках задачи является нецелесообразным.

# Приложение 1. *Таблицы и графики*

|  |  |  |  |
| --- | --- | --- | --- |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 38,80720 | 100% | 1 |
| 2 | 19,72659 | 98% | 1,967253739 |
| 4 | 10,04609 | 97% | 3,862916604 |
| 8 | 5,44348 | 89% | 7,129114049 |
| 12 | 4,29299 | 75% | 9,039665818 |
| 16 | 5,69908 | 43% | 6,809381656 |

# Приложение 2. *Код программы (без schedule)*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <ctime>

#include <omp.h>

#define N 1400

#define epsilon 10e-5

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

{

    double \*A = (double \*)malloc(N \* N \* sizeof(double));

    double \*b = (double \*)malloc(N \* sizeof(double));

    double \*r = (double \*)malloc(N \* sizeof(double));

    double \*prev\_r = (double \*)malloc(N \* sizeof(double));

    double \*z = (double \*)malloc(N \* sizeof(double));

    double \*X = (double \*)calloc(N, sizeof(double));

    double \*Az = (double \*)malloc(N \* sizeof(double));

    omp\_set\_num\_threads(atoi(argv[1]));

    // INITIALIZING OF VARIABLES

    for (int i = 0; i < N; i++)

    {

        b[i] = (double)(rand() % 100);

        for (int j = i; j < N; j++)

        {

            if (i == j)

            {

                A[i \* N + j] = (double)(rand() % 100) + 200;

            }

            else

            {

                A[i \* N + j] = A[j \* N + i] = (double)(rand() % 100);

            }

        }

    }

    double start = omp\_get\_wtime();

#pragma omp parallel for

    for (int i = 0; i < N; i++)

    {

        X[i] = 0;

    }

#pragma omp parallel for

    for (int i = 0; i < N; i++)

    {

        r[i] = b[i]; // r0 = b - Ax0, x0 - нулеой вектор

        z[i] = r[i]; // z0 = r0

    }

    double square\_b\_len = 0; // square of vector b length

#pragma omp parallel for reduction(+ \

                                   : square\_b\_len)

    for (int i = 0; i < N; i++)

    {

        square\_b\_len += b[i] \* b[i];

    }

    double restrictor = epsilon \* epsilon \* square\_b\_len;

    double temp = 0;

    double alpha = 0;

    double beta = 0;

    int cntr = 0;

    int iterations = 0;

    double sqrR = 0;

    double oldsqrR = 0;

    // double oldsqrR = (r, r);

#pragma omp parallel for reduction(+ \

                                   : oldsqrR)

    for (int i = 0; i < N; i++)

    {

        oldsqrR += r[i] \* r[i];

    }

#pragma omp parallel

    while (cntr < 3)

    {

#pragma omp single

        {

            sqrR = 0;

            temp = 0;

        }

        // A\*z(n)

#pragma omp for

        for (int i = 0; i < N; i++)

        {

            Az[i] = 0;

            for (int j = 0; j < N; j++)

            {

                Az[i] += A[i \* N + j] \* z[j];

            }

        }

#pragma omp for reduction(+ \

                          : temp)

        for (int i = 0; i < N; i++)

        {

            temp += Az[i] \* z[i];

        }

        // alpha(n+1) = (r(n), r(n)) / (Az(n), z(n))

        alpha = oldsqrR / temp;

        // X(n+1) = X(n) + alpha(n+1)\*z(n)

#pragma omp for

        for (int i = 0; i < N; i++)

        {

            X[i] = X[i] + alpha \* z[i];

        }

        //  r(n+1) = r(n) - alpha(n+1)Az(n)

#pragma omp for

        for (int i = 0; i < N; i++)

        {

            r[i] = r[i] - alpha \* Az[i];

        }

        // beta(n+1) = (r(n+1), r(n+1)) / (r(n), r(n))

#pragma omp for reduction(+ \

                          : sqrR)

        for (int i = 0; i < N; i++)

        {

            sqrR += r[i] \* r[i];

        }

#pragma omp single

        {

            beta = sqrR / oldsqrR;

            oldsqrR = sqrR;

        }

        // z(n+1) = r(n+1) + beta(n+1)z(n)

#pragma omp for

        for (int i = 0; i < N; i++)

        {

            z[i] = r[i] + beta \* z[i];

        }

#pragma omp single

        if (sqrR < restrictor)

        {

            cntr++;

        }

#pragma omp single

        {

            iterations++;

        }

    }

    double finish = omp\_get\_wtime();

    double total\_time = (finish - start);

    printf("Iterations count: %d \n", iterations);

    printf("Total time: %f \n", total\_time);

    // Print result

    printf("\n");

    for (int i = 0; i < N; i++)

    {

        printf("%f ", X[i]);

    }

    free(A);

    free(b);

    free(r);

    free(z);

    free(X);

    return 0;

}

# Приложение 3. *Код программы (c schedule)*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <ctime>

#include <omp.h>

#define N 1400

#define epsilon 10e-5

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

{

    double \*A = (double \*)malloc(N \* N \* sizeof(double));

    double \*b = (double \*)malloc(N \* sizeof(double));

    double \*r = (double \*)malloc(N \* sizeof(double));

    double \*prev\_r = (double \*)malloc(N \* sizeof(double));

    double \*z = (double \*)malloc(N \* sizeof(double));

    double \*X = (double \*)calloc(N, sizeof(double));

    double \*Az = (double \*)malloc(N \* sizeof(double));

    omp\_set\_num\_threads(atoi(argv[1]));

    // INITIALIZING OF VARIABLES

    for (int i = 0; i < N; i++)

    {

        b[i] = (double)(rand() % 100);

        for (int j = i; j < N; j++)

        {

            if (i == j)

            {

                A[i \* N + j] = (double)(rand() % 100) + 200;

            }

            else

            {

                A[i \* N + j] = A[j \* N + i] = (double)(rand() % 100);

            }

        }

    }

    double start = omp\_get\_wtime();

#pragma omp parallel for schedule(runtime)

    for (int i = 0; i < N; i++)

    {

        X[i] = 0;

    }

#pragma omp parallel for schedule(runtime)

    for (int i = 0; i < N; i++)

    {

        r[i] = b[i]; // r0 = b - Ax0, x0 - нулеой вектор

        z[i] = r[i]; // z0 = r0

    }

    double square\_b\_len = 0; // square of vector b length

#pragma omp parallel for schedule(runtime) reduction(+ \

                                                    : square\_b\_len)

    for (int i = 0; i < N; i++)

    {

        square\_b\_len += b[i] \* b[i];

    }

    double restrictor = epsilon \* epsilon \* square\_b\_len;

    double temp = 0;

    double alpha = 0;

    double beta = 0;

    int cntr = 0;

    int iterations = 0;

    double sqrR = 0;

    double oldsqrR = 0;

    // double oldsqrR = VectorsMult(r, r);

#pragma omp parallel for schedule(runtime) reduction(+ \

                                                    : oldsqrR)

    for (int i = 0; i < N; i++)

    {

        oldsqrR += r[i] \* r[i];

    }

#pragma omp parallel

    while (cntr < 3)

    {

#pragma omp single

        {

            sqrR = 0;

            temp = 0;

        }

        // A\*z(n)

#pragma omp for schedule(runtime)

        for (int i = 0; i < N; i++)

        {

            Az[i] = 0;

            for (int j = 0; j < N; j++)

            {

                Az[i] += A[i \* N + j] \* z[j];

            }

        }

#pragma omp for schedule(runtime) reduction(+ \

                                           : temp)

        for (int i = 0; i < N; i++)

        {

            temp += Az[i] \* z[i];

        }

        // alpha(n+1) = (r(n), r(n)) / (Az(n), z(n))

        alpha = oldsqrR / temp;

        // X(n+1) = X(n) + alpha(n+1)\*z(n)

#pragma omp for schedule(runtime)

        for (int i = 0; i < N; i++)

        {

            X[i] = X[i] + alpha \* z[i];

        }

        // r(n+1) = r(n) - alpha(n+1)Az(n)

#pragma omp for schedule(runtime)

        for (int i = 0; i < N; i++)

        {

            r[i] = r[i] - alpha \* Az[i];

        }

        // beta(n+1) = (r(n+1), r(n+1)) / (r(n), r(n))

#pragma omp for schedule(runtime) reduction(+ \

                                           : sqrR)

        for (int i = 0; i < N; i++)

        {

            sqrR += r[i] \* r[i];

        }

#pragma omp single

        {

            beta = sqrR / oldsqrR;

            oldsqrR = sqrR;

        }

        // z(n+1) = r(n+1) + beta(n+1)z(n)

#pragma omp for schedule(runtime)

        for (int i = 0; i < N; i++)

        {

            z[i] = r[i] + beta \* z[i];

        }

#pragma omp single

        if (sqrR < restrictor)

        {

            cntr++;

        }

#pragma omp single

        {

            iterations++;

        }

    }

    double finish = omp\_get\_wtime();

    double total\_time = (finish - start);

    printf("Iterations count: %d \n", iterations);

    printf("Total time: %f \n", total\_time);

    free(A);

    free(b);

    free(r);

    free(z);

    free(X);

    return 0;

}