**ФГБОУ ВО   
Уфимский университет науки и технологий**

**Кафедра ВМиК**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 90 |  |  |  |  |  |  |  |  |  |  |  |  |
| 80 |  |  |  |  |  |  |  |  |  |  |  |  |
| 70 |  |  |  |  |  |  |  |  |  |  |  |  |
| 60 |  |  |  |  |  |  |  |  |  |  |  |  |
| 50 |  |  |  |  |  |  |  |  |  |  |  |  |
| 40 |  |  |  |  |  |  |  |  |  |  |  |  |
| 30 |  |  |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

Параллельное вычисление произведения матриц с использованием «OpenMP»

**ОТЧЕТ**

**по лабораторной работе**

**по** Технологиям параллельного программирования

(*наименование дисциплины*)

|  |
| --- |
| Лабораторная работа 3 |
| (обозначение документа) |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Группа |  |  | Фамилия, И., О. | Подпись | Дата | Оценка |
| МО-325Б |  |
|  |  |
| Студент | | | Шарыгин М.С. |  |  |  |
| Преподаватель | | | Спеле В.В. |  |  |  |
| Принял | | |  |  |  |  |

**Уфа 2025 г****.**

Содержание

[1 Цель работы 3](#_Toc192729347)

[2 Практическая часть 4](#_Toc192729348)

[2.1 Программная реализация последовательного алгоритма умножения двух квадратных матриц 4](#_Toc192729349)

[2.2 Программная реализация трёх алгоритмов распараллеливания 5](#_Toc192729350)

[2.3 Проверка результатов работы алгоритмов 7](#_Toc192729351)

[2.4 Выбор оптимального алгоритма 8](#_Toc192729352)

[2.5 Подсчёт времени работы последовательной и наилучшей параллельной программ 8](#_Toc192729353)

[2.6 Вычисление ускорения и эффективности 9](#_Toc192729354)

[2.7 Построение графиков ускорения и эффективности 9](#_Toc192729355)

[2.8 Определение пиковой производительности одного ядра и всей многоядерной системы 10](#_Toc192729356)

[2.9 Определение реальной производительности 11](#_Toc192729357)

[2.10 Построение графиков эффективности «» и «» 12](#_Toc192729358)

[2.11 Определение 13](#_Toc192729359)

[2.12 Модифицирование параллельной программы 13](#_Toc192729360)

[3 Вывод 15](#_Toc192729361)

[4 Приложение 16](#_Toc192729362)

[4.1 Код программы 16](#_Toc192729363)

# Цель работы

Приобрести навыки распараллеливания вложенных циклов с использованием директив «OpenMP». Исследовать ускорение, эффективность и производительность многопоточных реализаций алгоритмов решения задачи матричного умножения.

# Практическая часть

## Программная реализация последовательного алгоритма умножения двух квадратных матриц

Для начала реализуем возможность ввода количества повторов умножения пользователем, как показано на рисунке 2.1.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.1 – Ввод количества повторов

Затем создадим три статических двумерных массива (рисунок 2.2):

* массив «A» для первой матрицы;
* массив «B» для второй матрицы;
* массив «C» для результирующей матрицы.

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.2 – Создание статических двумерных массивов

На рисунке 2.3 показана реализация циклами «for» и функцией «rand()» заполнения массивов случайными числами из отрезка .

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.3 – Заполнение массивов случайными числами

Затем напишем последовательный алгоритм умножения двух квадратных матриц «A» и «B», который будет повторяться «q»; время выполнения будет подсчитываться с помощью функции «omp\_get\_wtime», которая будет фиксировать время в начале и конце умножения матриц. Затем реализуем подсчёт квадрата евклидовой нормы результирующей матрицы (рисунок 2.4).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.4 – Умножение матриц и подсчет квадрата евклидовой нормы

## Программная реализация трёх алгоритмов распараллеливания

Теперь реализуем алгоритмы распараллеливания путём добавления директив «OpenMP» в разные циклы последовательной программы, как показано на рисунках 2.5, 2.6 и 2.7.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.5 – Первый алгоритм распараллеливания

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.6 – Второй алгоритм распараллеливания

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.7 – Третий алгоритм распараллеливания

## Проверка результатов работы алгоритмов

Отладим работу программы, для чего сверим полученные квадраты евклидовых норм последовательного и распараллеленных алгоритмов: так как результаты совпадают, то распараллеленные алгоритмы работают корректно (рисунок 2.8).

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.8 – Проверка корректности работы распараллеленных алгоритмов

## Выбор оптимального алгоритма

Пусть размерности матриц будут . Найдем такие «q», при которых время выполнения каждого алгоритма будет примерно 10 секунд, и выберем тот алгоритм, который окажется производительнее остальных. По результатам запусков, представленных в таблице 2.1, можно сделать вывод, что алгоритм 2 является наиболее производительным.

|  |  |  |
| --- | --- | --- |
| **Алгоритм** | **q** | **T** |
| **1** | 100 | 10.8004 |
| **2** | 20000 | 10.816 |
| **3** | 7000 | 11.5915 |

Таблица 2.1 – Выбор наиболее производительного алгоритма

## Подсчёт времени работы последовательной и наилучшей параллельной программ

Заполним таблицу 2.2 с временами выполнения последовательной и параллельной программ при разных «N» и «q» таким образом, чтобы время выполнения первой [последовательной программы] было не менее 10 секунд.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **q** |  |  |
| **5** | 35000000 | 12.8598 | 376.541 |
| **10** | 4000000 | 11.6465 | 49.581 |
| **50** | 25000 | 10.3002 | 2.32114 |
| **100** | 3500 | 12.5957 | 2.39422 |
| **200** | 400 | 10.5151 | 1.65261 |
| **500** | 25 | 10.5052 | 1.58007 |

Таблица 2.2 – Время выполнения программ

## Вычисление ускорения и эффективности

Для вычисления ускорения необходимо разделить время выполнения программы на одном ядре на время выполнения программы на «p» ядрах: . А отношение ускорения к количеству ядер – эффективность: . В таблице 2.3 представлены вычисленные ускорения и эффективности для результатов, полученных в предыдущем пункте [2.5].

|  |  |  |
| --- | --- | --- |
| **N** | **S** | **E** |
| **5** | 0.0341525 | 0.0042690625 |
| **10** | 0.234898 | 0.02936225 |
| **50** | 4.43756 | 0.554695 |
| **100** | 5.26088 | 0.65761 |
| **200** | 6.36272 | 0.79534 |
| **500** | 6.64857 | 0.83107125 |

Таблица 2.3 – Ускорение и эффективность

## Построение графиков ускорения и эффективности

При небольших «N» ускорение и эффективность оказываются очень маленькими, так как вычисления проходят достаточно быстро и время, которое затрачивается на создание параллельной области, превышает время вычислений. Но при увеличении «N» ускорение и эффективность также увеличиваются, так как в данном случае время вычислений больше времени, которое необходимо на создание параллельной области.

## Определение пиковой производительности одного ядра и всей многоядерной системы

Пиковой (теоретической) производительностью называется максимальное количество команд или операций, которое может теоретически выполнять вычислительная система в единицу времени при условии постоянной и полной загрузки всех ее исполнительных устройств.

Пиковая производительность чаще всего измеряется в количестве выполняемых в секунду операций с плавающей точкой – «Flops». Формула для расчета пиковой производительности «» имеет вид: , где

* «𝑝» – количество процессоров или ядер вычислительной системы;
* «𝑛» – теоретическое количество операций с плавающей точкой, которое может выполнять процессор или ядро за 1 такт;
* «𝑣» – тактовая частота, на которой работает процессор (ядро).

Вычисление «» и «»:

## Определение реальной производительности

Реальной производительностью называется количество команд или операций, которое выполняет вычислительная система в единицу времени для конкретного алгоритма (программы). Чтобы вычислить один элемент результирующей матрицы «C», необходимо выполнить скалярное умножения строки матрицы «A» на столбец матрицы B – требуется совершить «N» операций умножения и столько же операций сложения. Тогда для вычисления полной матрицы C требуется «» вещественных операций.

Тогда если за время «T» осуществляется «q» матричных умножений, получим, что реальную производительность можно выразить формулой: .

Чтобы оценить эффективность использования вычислительной системы при выполнении последовательного и параллельного алгоритмов, введем коэффициенты «» и «»: . В таблице 2.4 представлены полученные результаты.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** |  |  |  |  |
| **5** | 680414936.469 | 23237841.297 | 0.0103722 | 0.00004428 |
| **10** | 686901644.271 | 161352130.856 | 0.0104711 | 0.000307455 |
| **50** | 606784334.285 | 2692642408.471 | 0.00924976 | 0.0051308 |
| **100** | 555745214.637 | 2923707929.931 | 0.00847173 | 0.00557109 |
| **200** | 608648514.993 | 3872662031.574 | 0.00927818 | 0.00737931 |
| **500** | 594943456.574 | 3955520957.932 | 0.00906926 | 0.0075372 |

Таблица 2.4 – Реальная производительность

## Построение графиков эффективности «» и «»

На одном ядре увеличение размерности матриц приводит к постепенному ухудшению эффективности из-за роста объема вычислений и ухудшения использования памяти. На «p = 8» ядрах увеличение размерности матриц позволяет лучше распределить нагрузку между ядрами, улучшить использование памяти и достичь более высокой производительности за счет параллелизма.

## Определение

Определим размерность матриц такую, что при любом 𝑁 < параллельная версия программы будет работает медленнее последовательной, а при 𝑁 > будет иметься ускорение при использовании параллельной программы. После проведения дополнительных запусков получено, что .

## Модифицирование параллельной программы

Добавим спецификатор «if» директивы «parallel» так, как показано на рисунке 2.9. Теперь параллельная область будет создаваться только в том случае, если .

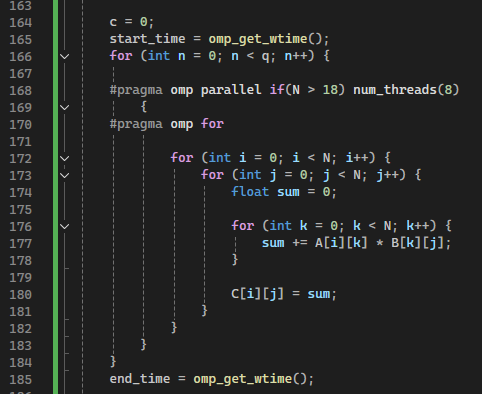


Рисунок 2.9 – Модифицированная программа

Проверим корректность работы программы (рисунок 2.10 и 2.11).

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.10 – Результат при «N = 1»

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2.11 – Результат при «N = 100»

# Вывод

В ходе лабораторной работы мы приобрели навыки распараллеливания вложенных циклов с использованием директив «OpenMP», а также исследовали ускорение, эффективность и производительность многопоточных реализаций алгоритмов решения задачи матричного умножения.

# Приложение

## Код программы

#include <iostream>

#include <omp.h>

using namespace std;

int q;

const int N = 100;

double start\_time, end\_time;

float A[N][N], B[N][N], C[N][N], c;

int main() {

    setlocale(LC\_ALL, "Russian");

    cout << "Введите количество повторов умножения          = ";

    cin >> q;

    cout << endl;

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            A[i][j] = (rand() % 1000 - 500.0) / 1000.0;

            B[i][j] = (rand() % 1000 - 500.0) / 1000.0;

        }

    }

    // Последовательный алгоритм //

    c = 0;

    start\_time = omp\_get\_wtime();

    for (int n = 0; n < q; n++) {

        for (int i = 0; i < N; i++) {

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

                float sum = 0;

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

                    sum += A[i][k] \* B[k][j];

                }

                C[i][j] = sum;

            }

        }

    }

    end\_time = omp\_get\_wtime();

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            c += C[i][j] \* C[i][j];

    cout << "Результат без алгоритма                        = " << c << endl;

    cout << "Время выполнения без алгоритма                 = " << end\_time - start\_time << "\n\n";

    // Алгоритм 1 //

    c = 0;

    start\_time = omp\_get\_wtime();

    for (int n = 0; n < q; n++) {

        for (int i = 0; i < N; i++) {

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

                float sum = 0;

#pragma omp parallel reduction(+:sum) num\_threads(8)

                {

#pragma omp for

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

                        sum += A[i][k] \* B[k][j];

                    }

                }

                C[i][j] = sum;

            }

        }

    }

    end\_time = omp\_get\_wtime();

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            c += C[i][j] \* C[i][j];

    cout << "Результат алгоритма 1                          = " << c << endl;

    cout << "Время выполнения алгоритма 1                   = " << end\_time - start\_time << "\n\n";

    // Алгоритм 2 //

    c = 0;

    start\_time = omp\_get\_wtime();

    for (int n = 0; n < q; n++) {

#pragma omp parallel num\_threads(8)

        {

#pragma omp for

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

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

                    float sum = 0;

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

                        sum += A[i][k] \* B[k][j];

                    }

                    C[i][j] = sum;

                }

            }

        }

    }

    end\_time = omp\_get\_wtime();

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            c += C[i][j] \* C[i][j];

    cout << "Результат алгоритма 2                          = " << c << endl;

    cout << "Время выполнения алгоритма 2                   = " << end\_time - start\_time << "\n\n";

    // Алгоритм 3 //

    c = 0;

    start\_time = omp\_get\_wtime();

    for (int n = 0; n < q; n++) {

        for (int i = 0; i < N; i++) {

#pragma omp parallel num\_threads(8)

            {

#pragma omp for

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

                    float sum = 0;

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

                        sum += A[i][k] \* B[k][j];

                    }

                    C[i][j] = sum;

                }

            }

        }

    }

    end\_time = omp\_get\_wtime();

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            c += C[i][j] \* C[i][j];

    cout << "Результат алгоритма 3                          = " << c << endl;

    cout << "Время выполнения алгоритма 3                   = " << end\_time - start\_time << "\n\n";

    // Применение спецификатора "if" //

    c = 0;

    start\_time = omp\_get\_wtime();

    for (int n = 0; n < q; n++) {

    #pragma omp parallel if(N > 18) num\_threads(8)

        {

    #pragma omp for

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

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

                    float sum = 0;

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

                        sum += A[i][k] \* B[k][j];

                    }

                    C[i][j] = sum;

                }

            }

        }

    }

    end\_time = omp\_get\_wtime();

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            c += C[i][j] \* C[i][j];

    cout << "Результат модифицированного алгоритма 2        = " << c << endl;

    cout << "Время выполнения модифицированного алгоритма 2 = " << end\_time - start\_time << "\n\n";

    return 0;

}