САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

# Отчет

по домашней работе №5

**«OpenMP»**

Выполнил: Утюжников Дмитрий Александрович

студ. гр. М3139

Санкт-Петербург

2020

**Цель работы:** знакомство со стандартом распараллеливания команд OpenMP.

**Инструментарий и требования к работе:** рекомендуется использовать C, C++. Возможно использовать Python и Java.

## Теоретическая часть

OpenMP — это библиотека для параллельного программирования вычислительных систем с общей память. Потоки создаются в рамках единственного процесса и имеют свою собственную память. Кроме того, все потоки имеют доступ к памяти процесса (см. рисунок 1).

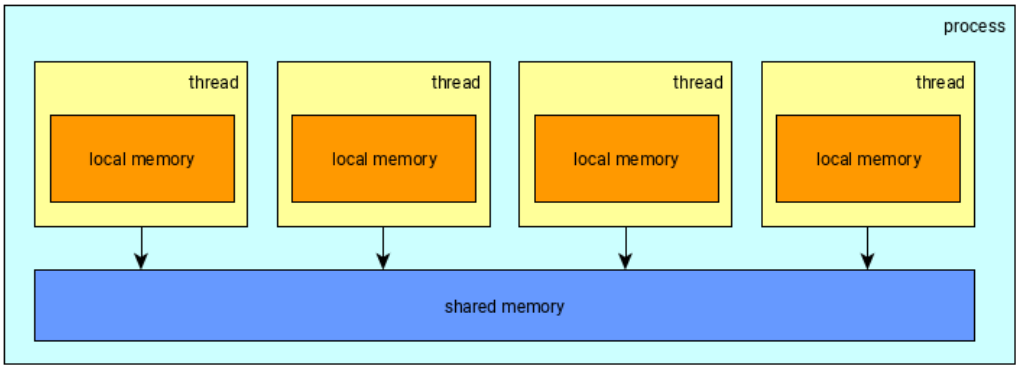


Рисунок 1 - модель памяти в OpenMP

Параллельная область задаётся директивой #pragma omp parallel, процесс порождает ряд потоков. Их число можно задать явно num_threads (кол-во потоков), однако по умолчанию будет создано столько потоков, сколько в системе вычислительных ядер. Границы параллельной области выделяются фигурными скобками (см. рисунок 2). Директивы parallel могут быть вложены, поэтому могут создаваться вложенные потоки.

```
    // 1 thread
    int num = 4;
#pragma omp parallel num_threads(num) {
    // 4 threads
}
    // 1 thread
```

Рисунок 2 - Создание параллельной области

Поток, который существует всё время работы, называется master, остальные потоки открываются и закрываются.

Все переменные, созданные до директивы parallel являются общими для всех потоков. Переменные, созданные внутри потока, являются локальными и доступны только текущему потоку. При изменении общей переменной одновременно несколькими потоками возникает состояние гонок (нет определенности, какой-либо конкретный порядок записи) — это проблема. Для её решения проблемы существует директива critical. В критической секции в один момент времени может находиться только один поток, остальные ожидают ее освобождения. Для ряда операций более эффективно использовать директиву atomic, чем критическую секцию (см. рисунок 3). Она ведет себя также, но работает чуть быстрее.

```
    double determinant;
#pragma omp parallel for num_threads(threads)
    for (int i = 0; i < size; i++) {
#pragma omp atomic
        determinant *= matrix[i][i];
    }
```

Рисунок 3 - Пример использования atomic

На рисунке 3 приведен алгоритм подсчета произведения элементов главной диагонали матрицы. Чтобы произведение было верно посчитано, необходимо использовать директиву atomic, иначе результат умножения не будет корректным.

Если мы хотим, чтобы все потоки выполнили некоторую часть алгоритма, а потом приступили к следующей части, то следует использовать директиву barrier. Поток завершит часть вычислений до директивы barrier и будет ждать остальные потоки (см. рисунок 4). Сначала все потоки увеличат x на единицу, а потом выведут новое значение.

```
#pragma omp parallel {
    #pragma omp atomic
        x++;
    #pragma omp barrier
    #pragma omp critical {
        cout << x;
    }
}
```

Рисунок 4 - Пример использования barrier

Самый популярный способ распределения задач в OpenMP — параллельный цикл. Для распараллеливания используется #pragma omp parallel for (см. рисунок 5).

```
#pragma omp parallel for num_threads(threads)
        for (int j = i + 1; j < size; j++) {
            double k = matrix[j][i] / matrix[i][i];
            for (int y = i; y < size; y++) {
                matrix[j][y] -= k * matrix[i][y];
            }
        }
    }
```

Рисунок 5 - Пример использования parallel for

Чтобы параллельно вычислить сумму, произведение элементов, можно использовать директиву reduction (оператор: список). Список перечисляет имена общих переменных. У переменных должен быть скалярный тип.

Принцип работы:

1. Для каждой переменной создаются локальные копии в каждом потоке.
2. Локальные копии инициализируются соответственно типу оператора (см. рисунок 6).
3. Над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор. Порядок выполнения операторов не определен.

| Оператор | Исходное значение переменной |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| & | ~0 (каждый бит установлен) |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |

Рисунок 6 - Операторы reduction

По сравнению с начальным кодом (см. рисунок 3) с использованием reduction код примет вид (см. рисунок 7).

```
#pragma omp parallel for reduction (*: determinant)
    for (int i = 0; i < size; i++) {
        determinant *= matrix[i][i];
    }
```

Рисунок 7 - Пример использования reduction

Параллельный цикл позволяет задать опцию schedule, изменяющую алгоритм распределения итераций между потоками.

Опции планирования:

- schedule (static) – статическое планирование. При использовании такой опции итерации цикла будут приблизительно поровну поделены между потоками.
- schedule (static, 5) – блочно-циклическое распределение итераций. Каждый поток получает заданное число итераций в начале цикла, затем процедура распределения продолжается. Планирование выполняется 1 раз.
- schedule (dynamic, 5) – динамическое планирование. По умолчанию параметр опции равен 1. Каждый поток получает заданное число итераций, выполняет их и запрашивает новую порцию. Распределение происходит во время выполнения программы. Распределение зависит от темпов работы потоков и трудоемкости операций
- schedule (guided, 5) – разновидность динамического планирования с изменяемым при каждом последующем распределении числе итераций. Распределение начинается с некоторого начального размера, зависящего от реализации библиотеки до значения, задаваемого в опции (по умолчанию 1).

Описание алгоритма подсчета определителя матрицы методом Гаусса:

1. Приведём матрицу в верхней треугольной с помощью элементарных преобразований.
2. Вычислим произведение элементов на главной диагонали. Значением определителя матрицы будет являться полученный результат.

## Экспериментальная часть

Для реализации параллельного вычисления определителя воспользуемся библиотекой omp.h. Заведём динамический двумерный массив для

хранения матрицы и проинициализируем его. При размерах матрицы 3000x3000 элементов произведем подсчёт времени выполнения вычисления определителя матрицы (см. рисунки 7, 8) с выключенным openmp, с различным числом потоков (от 0 до 16) при одинаковом параметре schedule (static), с одинаковым числом потоков (12 потоков) при различных параметрах schedule (static, dynamic, guided 1, 2, 4…1024).

Для реализации подсчета определителя будем использовать метод Гаусса. Будем идти по столбцам (с 1 по n) и занулять все элементы под главной диагональю. Для столбца i найдём такую строку g с i-ой по n-ую, что i-ый элемент не равен 0. Поменяем строки g и i местами и учтем изменение знака определителя. Далее следует вычесть строку i из всех последующих строк, чтобы получить нули в i-ом столбце в элементах под главной диагональю. И перейдём к следующему столбцу. В итоге получим верхнюю треугольную матрицу и посчитаем определитель, перемножив элементы на главной диагонали.

Можно заметить, что есть возможность распараллелить некоторые части алгоритма, а именно поиск строки g, где i-ый элемент не равен 0, вычитание строки i из всех последующих и подсчёт определителя перемножением элементов.


Рисунок 8 – Время выполнения алгоритма с различным числом потоков

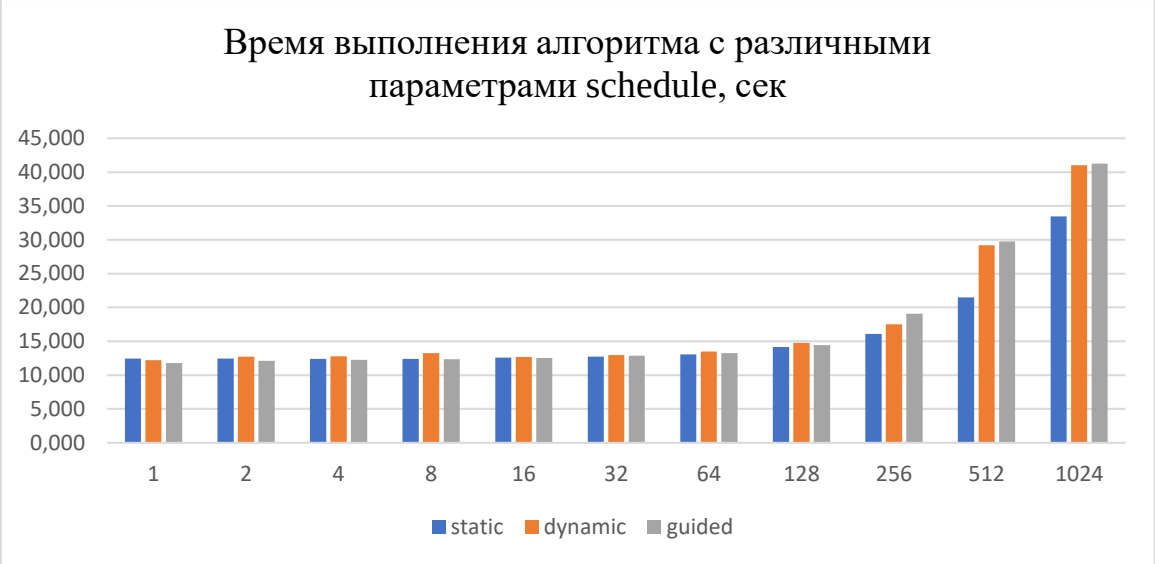


Рисунок 9 - Время выполнения алгоритма с различными параметрами schedule

**Компилятор MinGW-w64.**

**…\hw5\cmake-build-debug> hw5.exe [аргументы]**

# Листинг

## main.cpp

```
#include <iostream>
#include <vector>
#include <ctime>
#include <fstream>
#include <iomanip>
#include "omp.h"

using namespace std;

double determinate_omp(vector <vector <double>> matrix, int threads) {
    int size = matrix.size();
    double determinant = 1;
    for (int i = 0; i < size; i++) {
        int g = -1;
#pragma omp parallel for num_threads(threads)
        for (int j = i; j < size; j++) {
            if (matrix[j][i] != 0 && g == -1) {
                g = j;
            }
        }
        if (g == -1) {
            determinant = 0;
            break;
        }
        if (g != i) {
            swap(matrix[i], matrix[g]);
```

```
            determinant *= -1;
        }
#pragma omp parallel for num_threads(threads)
        for (int j = i + 1; j < size; j++) {
            double k = matrix[j][i] / matrix[i][i];
            for (int y = i; y < size; y++) {
                matrix[j][y] -= k * matrix[i][y];
            }
        }
    }
    for (int i = 0; i < size; i++) {
        determinant *= matrix[i][i];
    }
    return determinant;
}

double determinate_without(vector <vector <double>> matrix);

int parse(string s) {
    if (s[0] == '-') {
        return -1;
    }
    int n = 0;
    for (int i = 0; i < s.length(); i++) {
        n = n * 10 + s[i] - '0';
    }
    return n;
}

int main(int argc, char* argv[]) {
    int threads = parse(argv[1]);
    if (threads < 0) {
        cout << "Num of threads less 0";
        return 0;
    }
    ifstream in(argv[2]);
    if (!in.is_open()) {
        cout << "Input file doesn't open";
        return 0;
    }
    int n;
    in >> n;
    vector <vector <double>> matrix(n, vector <double> (n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            in >> matrix[i][j];
    }
    in.close();
    clock_t clockT = clock();
    double det = determinate_without(matrix);
    double time = ((double) clock() - (double) clockT) / CLOCKS_PER_SEC;
    if (argc > 3) {
        ofstream out(argv[3]);
        if (!out.is_open()) {
```

```
            cout << "Output file doesn't open";
            return 0;
        }
        out << fixed << setprecision(7) << "Determinant: " <<  det <<
endl;
        out.close();
    } else {
        cout << fixed << setprecision(7) << "Determinant: " <<  det <<
endl;
    }
    cout << fixed << setprecision(7) << "Time " << time * 1000 << " ms" <<
endl;
    double start_time;
    start_time = omp_get_wtime();
    determinate_omp(matrix, threads);
    cout << fixed << setprecision(7) << "Time (" << threads << "
thread(s)): "
            << (omp_get_wtime() - start_time) * 1000 << " ms" << endl;
    return 0;
}

double determinate_without(vector <vector <double>> matrix) {
    int size = matrix.size();
    double determinant = 1;
    for (int i = 0; i < size; i++) {
        int g = -1;
        for (int j = i; j < size; j++) {
            if (matrix[j][i] != 0 && g == -1) {
                g = j;
            }
        }
        if (g == -1) {
            determinant = 0;
            break;
        }
        if (g != i) {
            swap(matrix[i], matrix[g]);
            determinant *= -1;
        }
        for (int j = i + 1; j < size; j++) {
            double k = matrix[j][i] / matrix[i][i];
            for (int y = i; y < size; y++) {
                matrix[j][y] -= k * matrix[i][y];
            }
        }
    }
    for (int i = 0; i < size; i++) {
        determinant *= matrix[i][i];
    }
    return determinant;
}
```