САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Карпенко Андрей Сергеевич

студ. гр. М3138

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 – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций. Позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran.

Основной поток порождает дочерние потоки по мере необходимости. Программирование путем вставки директив компилятора в ключевые места исходного кода программы. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для расспараллеливания участков кода.

В модели с разделяемой памятью взаимодействие потоков происходит через разделяемые переменные. При неаккуратном обращении с такими переменными в программе могут возникнуть ошибки соревнования (race condition). Такое происходит из-за того, что потоки выполняются параллельно и соответственно последовательность доступа к разделяемым переменным может быть различна от одного запуска программы к другому.

Модель с разделяемой памятью:

- Все потоки имеют доступ к глобальной разделяемой памяти
- Данные могут быть разделяемые и приватные
- Разделяемые данные доступны всем потокам
- Приватные — только одному
- Синхронизация требуется для доступа к общим данным

Синтаксис. В основном конструкции OpenMP – это директивы компилятора Для C/C++ директивы имеют следующий вид (рисунок №1):

```
#pragma omp конструкция [условие [условие]...]
```

Рисунок №1 – основная конструкция

```
#pragma omp parallel [другие директивы]
```

Директива parallel указывает, что структурный блок кода должен быть выполнен параллельно в несколько потоков (нитей, threads). Каждый из созданных потоков выполнит одинаковый код содержащийся в блоке, но не одинаковый набор команд. В разных потоках могут выполняться различные

ветви или обрабатываться различные данные, что зависит от таких операторов как if-else или использования директив распределения работы.

`#pragma omp parallel for num_threads(th)` – директива parallel for. Эта директива ставится непосредственно перед циклом for и указывает на то, что разные итерации указанного цикла не зависят одна от другой, и, следовательно, могут выполняться параллельно. Он позволяет не фиксировать заранее число потоков, и это число может определяться при выполнении программы. `num_threads(th)` – обозначает количество тредов.

`#pragma parallel for schedule(тип, размер)`
Здесь тип – одно из слов static, dynamic, guided. Размер — явно указанное натуральное число. Некоторые типы позволяют не указывать размер, некоторые даже запрещают его указывать.

1. Static. Если размер не указан, все множество итераций цикла делится на примерно равные части (по числу входящих в них итераций) в соответствии с числом потоков и каждому потоку назначается своя часть. Такой вариант обеспечивает минимальные затраты на планирование, но может страдать несбалансированностью нагрузки по потокам в случае, если разные итерации выполняются за существенно разное время.
2. Dynamic. Части всего множества итераций (опять количество итераций в каждой части равно указанному размеру, а если он не указан, то 1 по умолчанию) назначаются каждому потоку по его запросу. Это означает, что после начального распределения частей очередная часть достается тому потоку, который раньше закончил свою работу.
3. Guided. Размер определяет минимальный размер порции итераций, а реальный размер пропорционален оставшемуся объему работы (по числу итераций), деленному на число потоков. Как и в случае dynamic, полученные порции итераций цикла назначаются потокам по их запросу.

Вариант 7. Алгоритм будет приводить матрицу к треугольному виду с помощью элементарных преобразований: для каждого столбца под главной диагональю получаем нули, для этого находим в данном столбце ненулевой элемент, меняем его строку со строкой на главной диагонали (при этом знак определителя изменится на противоположный) и вычитаем из всех строк ниже данную строку, так чтобы получились нули. После приведения к треугольному виду находим произведение элементов, стоящих на главной диагонали – это и будет определитель.

Практическая часть

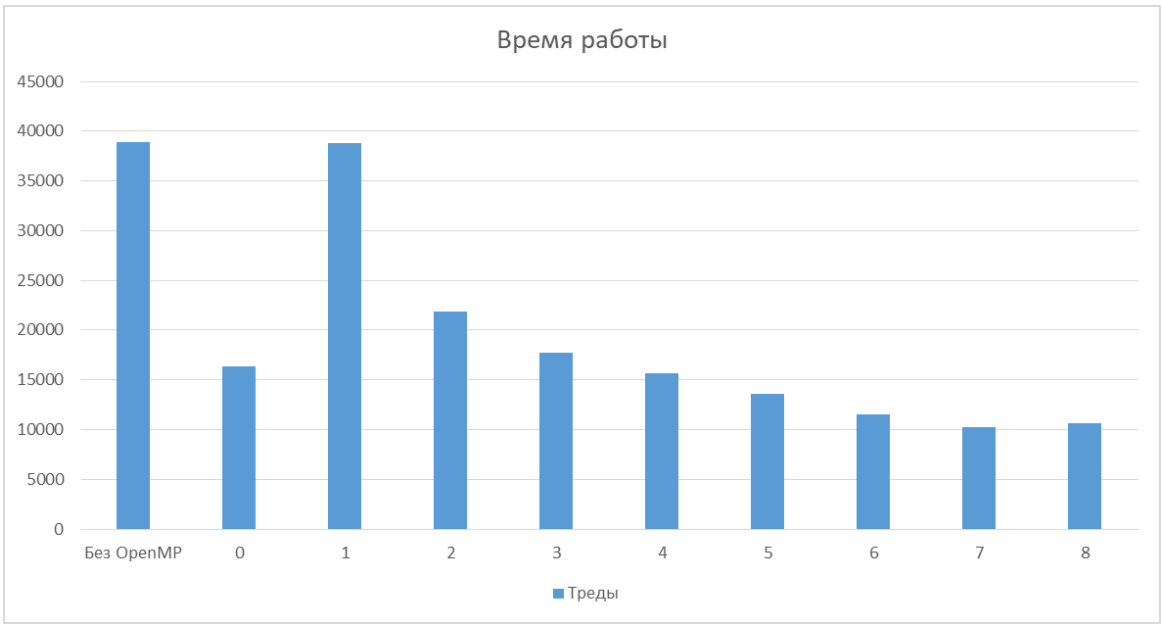

Рисунок №2 – время работы программы (в мс) с разными тредами

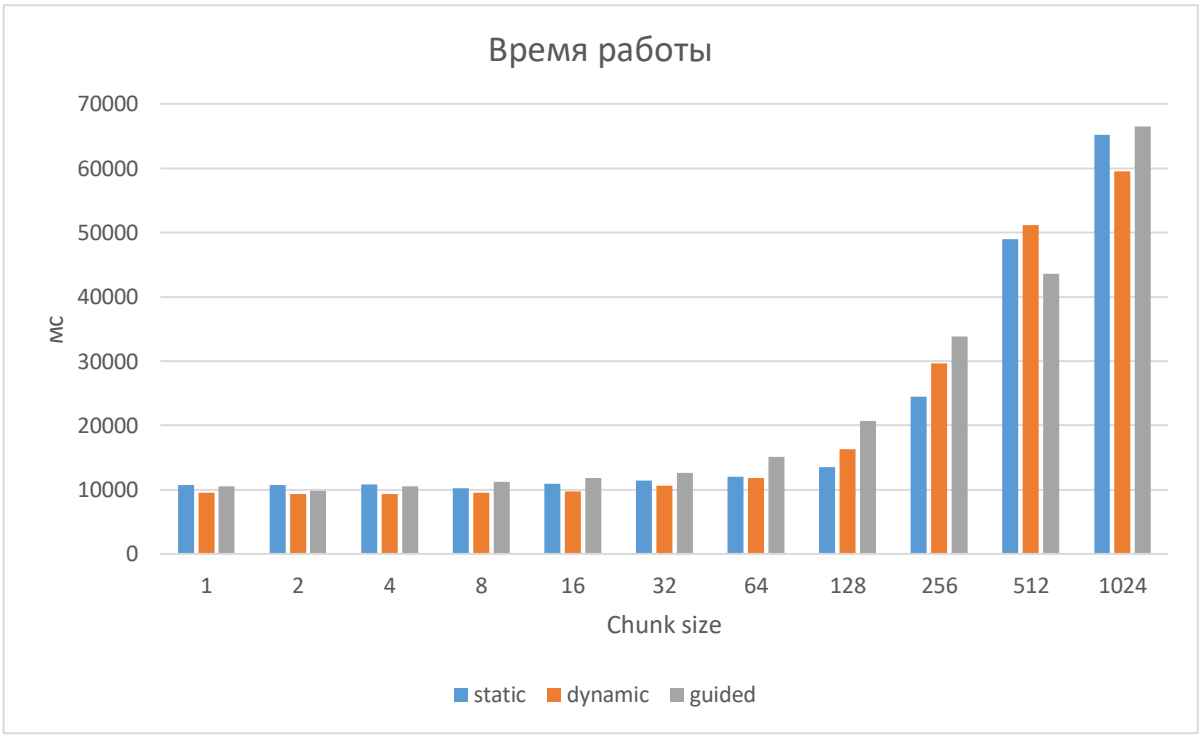

Рисунок №3 – разные режимы работы

Описание работы программы

Метод determinant – обычный метод для нахождения определителя, omp_determinant – метод для нахождения определителя с OpenMP, create – метод по созданию случайной матрицы заданного размера, read_matrix – чтение матрицы из заданного файла.

Листинг

Main.cpp

```
#include <iostream>
#include <vector>
#include <fstream>
#include <omp.h>
using namespace std;
int th;
float determinant(vector <vector <float>> a);
float omp_determinant(vector <vector <float>> a);
vector <vector <float>> read_matrix(string file);
vector <vector <float>> create(int n);
int main(int argc, char* argv[]) {
    float res;
    int max = atoi(argv[1]);
    th = 0;
    vector <vector <float>> a = read_matrix(argv[2]);
    double start = omp_get_wtime();
    res = determinant(a);
    double time = omp_get_wtime() - start;
    if (argc > 3) {
        ofstream out;
        out.open(argv[3]);
        out << "Determinant: ";
        out << res;
        out << endl;
        out.close();
    }
    else {
        printf("Determinant: % g\n", res);
    }
    printf("\nWithout OpenMP % f ms\n", time * 1000);
    for (int i = 0; i <= max; i++) {
        th = i;
        start = omp_get_wtime();
        omp_determinant(a);
        time = omp_get_wtime() - start;
        printf("\nTime(%i thread(s)) : % f ms\n", i, time * 1000);
    }

    return 0;
}

float determinant(vector<vector<float>> a) {
    int n = a.size();
    float res = 1;
    for (int i = 0; i < n; i++) {
        int cur = -1;
        for (int j = i; j < n; j++) {
            if (a[i][j] != 0) {
```

```
                cur = j;
                break;
            }
        }
        if (cur == -1) {
            return 0;
        }
        if (cur != i) {
            swap(a[i], a[cur]);
            res = -res;
        }
        for (int j = i + 1; j < n; j++) {
            float proportion = a[j][i] / a[i][i];
            for (int k = i + 1; k < n; k++) {
                a[j][k] -= a[i][k] * proportion;
            }
        }
    }
    for (int i = 0; i < n; i++) {
        res *= a[i][i];
    }
    return res;
}

float omp_determinant(vector<vector<float>> a) {
    int n = a.size();
    float res = 1;
    for (int i = 0; i < n; i++) {
        int cur = -1;
        for (int j = i; j < n; j++) {
            if (a[i][j] != 0) {
                cur = j;
                break;
            }
        }
        if (cur == -1) {
            return 0;
        }
        if (cur != i) {
            swap(a[i], a[cur]);
            res = -res;
        }
#pragma omp parallel for num_threads(th)
        for (int j = i + 1; j < n; j++) {
            float proportion = a[j][i] / a[i][i];
            for (int k = i + 1; k < n; k++) {
                a[j][k] -= a[i][k] * proportion;
            }
        }
    }
    for (int i = 0; i < n; i++) {
```

```
        res *= a[i][i];
    }
    return res;
}

vector <vector <float>> create(int n) {
    vector<vector<float>> a(n, vector <float>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i][j] = float(rand()) / float(RAND_MAX) * 100;
        }
    }
    return a;
}

vector <vector <float>> read_matrix(string file) {
    ifstream in;
    in.open(file);
    if (!in.is_open()) {
        cout << "Opening file error";
        exit(0);
    }
    int n;
    in >> n;
    vector <vector <float>> a(n, vector <float>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            in >> a[i][j];
        }
    }
    in.close();
    return a;
}
```