САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OpenMP»

Выполнил(а): ДЗЕСТЕЛОВ ХЕТАГ АРТУРОВИЧ

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

Параллельное программирование (ПП) является широким ответвлением в проектировании вычислительных систем. ПП требуется в тех случаях, когда требуется уменьшить время вычислений за счёт распределения вычислений между несколькими “вычислителями” – будь то несколькими ядрами процессора или несколькими компьютерами. При разработке требуется учитывать такие факторы, как цену оборудования, его производительность, затраты на разработку и поддержания соответствующего ПО. Последнее предусматривает и отладку программ при их написании. Под разные цели требуются разные принципы и стандарты.

OpenMP – стандарт для распараллеливания программ на языках C, C++, Fortan. Представляет из себя набор директив компилятора, библиотечных методов и переменных, предназначенных для программирования многопоточных приложений на системах с общей память (DSM-системы, в которых процессор имеет прямой доступ к памяти другого процессора).

Преимуществом стандарта OpenMP является простота разработки и отладки приложений. Распараллеливание заключается в выделении тех участка кода, вычисление которых может производиться параллельно. Компилятор интерпретирует специальные пометки и вставляет в соответствующие места программы библиотечные вызовы для распараллеливания участков кода. В связи с этим, написание и отладка программы – это в первую очередь разработка и отладка последовательного алгоритма, а далее разметка участков кода на соответствующие блоки. Ещё одни преимуществом стандарта является сравнительно низкий порог вхождения для специалистов, которые

отлично владеют языками программирования и имеют представление об основных принципах распараллеливания. Недостатками является проблематичность организации взаимодействия потоков, что приводит к частым ошибкам (**race condition** – условия гонок, о ней позже) и ограниченность DSM-системами.

За разметку программы на участки ответственен сам программист, он же и должен следить за синхронизацию потоков и их взаимную зависимость. В такой модели взаимодействие потоков происходит через разделяемые переменные. Неаккуратное проектирование программы может привести к ошибки соревнования (**race condition**) – ситуации, когда последовательность доступа к общим переменным у параллельных потоков непредсказуема (может отличаться при различных запусках программы).

Перейдём непосредственно к инструментам программирования и модели OpenMP. Существуют основной поток (**master**) и порожденные им (**slave**), которые появляются на некоторый промежуток по мере необходимости. В основном конструкции OpenMP – это директивы компилятора. Для C/C++ директивы имеют следующий вид:

#pragma omp <конструкция> [<условие> [<условие>]...]

(стоит отметить, что т. к. конструкции OpenMP является директивами, то на некорректную конструкцию компилятор ответит её игнорированием, выполняя её последовательно). Так, например, конструкция ***#pragma omp parallel*** помечает блок кода, как “параллельный регион”, именно здесь директивы вставят соответствующие библиотечные конструкции, которые распараллелят соответствующий участок. Нам интересна конструкция для параллельного цикла ***#pragma omp for***. Это конструкция создает (в зависимости от её условий) один или несколько потоков, в которых исполняются итерации цикла. Все переменные, объявленные внутри тела цикла, по умолчанию является локальными в каждом потоке. Барьером конструкции (место, доходя до которого, поток ожидает, пока остальные производят вычисления) по умолчанию является конец цикла. **Средствами синхронизации** потоков является критические

секции, атомарные операции, точки синхронизации и т. д. Рассмотрим основные из них: **критические секции** – блоки кода, которые могут выполняться одновременно только одним из потоков (остальные, готовые к этому блоку, ожидают своей очереди); **атомарная операция** похожа на критическую секцию, т. е. гарантируется корректная работа с общей переменной (на время выполнения блокируется доступ к данной переменной всеми потоками, кроме одного), некоторые атомарные операции напрямую заменяются командами процессора, что ускоряет их работу по сравнению с критическим секциями; **барьеры** – точки синхронизации потоков (как описано выше с концом цикла).

Нам потребуется условия конструкций: **num_threads, reduction, schedule**:

- **num_threads** – условие для явного задания количества потоков, которые будут выполнять параллельную область.
- **reduction(<оператор>: <переменные>)** – это условие гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы или произведения.
- **schedule –** условие для задание принципа, по которому будут распределяться итерации цикла между потоками:
  - **static** – итерации равномерно распределяются по потокам;
  - **dynamic** – итерации распределяются пакетами заданного размера (как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую)
  - **guided** – аналогичен предыдущему, за исключением того, что размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось
  - **runtime** – тип распределения определяется в момент выполнения программы

Практическая часть

Для решения поставленной задачи (**вариант №7**) был использован алгоритм приведения матрицы к треугольному виду через прямой ход Гаусса (как известно, определитель матрицы равен произведению элементов треугольной матрицы на главной диагонали).

Асимптотика алгоритма $\mathcal{O}(n^3)$:

- первый внешний цикл отвечает за зануление текущего столбца:
  - первый внутренний цикл отвечает за **нахождение ненулевого элемент**а (n итераций), по которому будет производиться зануление.
  - второй внутренний цикл отвечает за **зануление всех строк, ниже рассматриваемой** (n – i итераций при рассмотрении i-й строки)
    - при этом каждое зануление производит n арифметических действий (по всем **столбцам в пределах одной строки**) по соответствующей формуле.
- второй внешний цикл считает произведение на главной диагонали.

Распараллелить можно такие любые циклы, выполнение итераций в которых могут происходить независимо друг от друга:

- поиск ненулевого элемента.
- цикл зануления в пределах одной строки.
- цикл зануления столбца
- общее произведение элементов на главной диагонали.

Распараллеливание **поиска ненулевого элемента** бессмысленно, распараллеливание **зануления конкретной строки** потребует создание большого кол-ва потоков, что является недешевой операцией, т. е. только замедлит программу. Существенное ускорение даст распараллеливание **зануления всех строк, ниже рассматриваемой.** Применим конструкцию для цикла и укажем соответствующие условия. Аналогично можем распараллелить просчёт произведения, что, однако, существенно не

ускорит программу (в силу того, что основной вклад даёт именно зануление строк), однако хорошо демонстрирует работу редукции.

По результатам тестирования на случайно сгенерированных данных (матрицы 500x500 и 2000x2000) по нескольким запускам написанной программы имеем следующие результаты (см. рисунок 1 и рисунок 2). Как видно, в лучшем случае применение OpenMP ускорило программу в 4 раза.

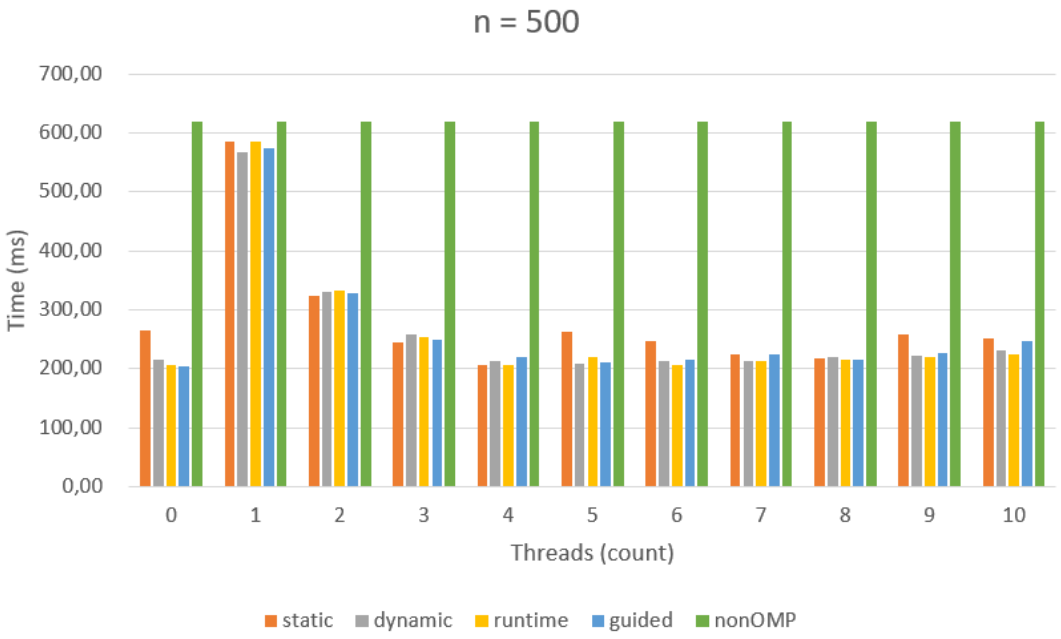

Рисунок №1 – Матрица 500x500

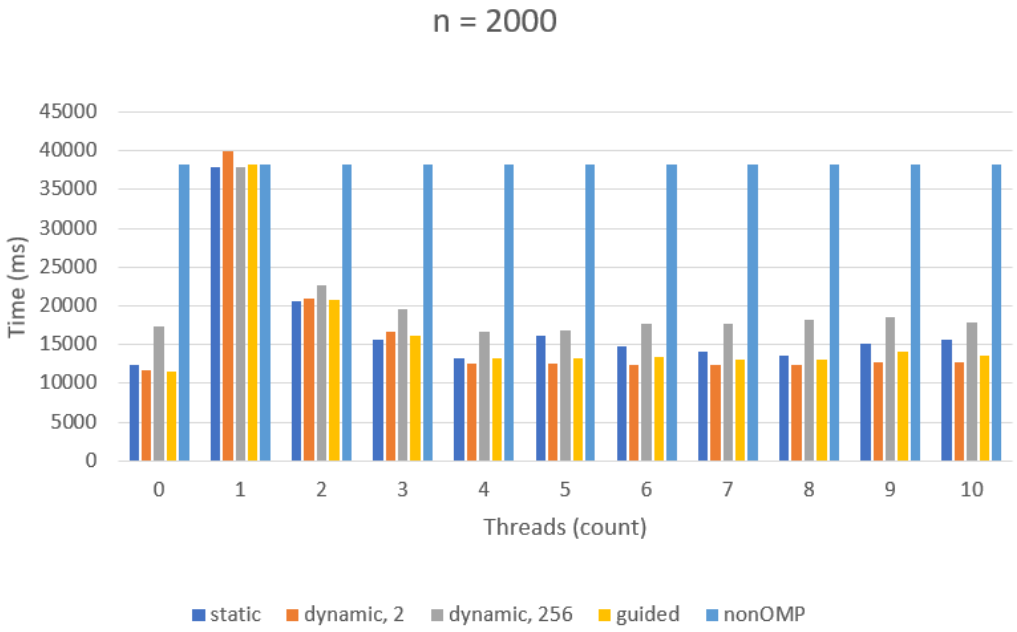

Рисунок №2 – Матрица 2000x2000

Программа представляет из себя ввод данных через файловую систему, обработку ошибок указания параметров запуска просчёт и замер времени исполнения при использовании OpenMP (и без), вывод результатов замера в консоль, вывод результата вычисления определителя в консоль или файл.

Листинг

Опциональный раздел. Добавляется в отчет, если имеется исходный код (файлы с логическими схемами не указываются в листинге).

main.cpp

```
#include <iostream>
#include <vector>
#include <fstream>

#include <chrono>
#include <random>

using namespace std;

int threads_count = 0;
double findOMP_det(int &n, vector<vector<float>> &matrix) {
    double determinant = 1;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            if (matrix[j][i] != 0) {
                if (i != j) {
                    swap(matrix[i], matrix[j]);
                    determinant *= -1;
                }
                break;
            }
        }

        if (matrix[i][i] != 0) {
#pragma omp parallel for num_threads(threads_count) schedule(static)
            for (int j = i + 1; j < n; j++) {
                float f = -matrix[j][i] / matrix[i][i];
                for (int k = 0; k < n; k++) {
                    matrix[j][k] += f * matrix[i][k];
                }
            }
        }
    }

#pragma omp parallel for num_threads(threads_count) reduction(*:determinant)
schedule(static)
    for (int i = 0; i < n; i++) {
        determinant *= (double) matrix[i][i];
    }

    return determinant;
}
```

```
double findNonOMP_det(int &n, vector<vector<float>> &matrix) {
    double determinant = 1;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            if (matrix[j][i] != 0) {
                if (i != j) {
                    swap(matrix[i], matrix[j]);
                    determinant *= -1;
                }
                break;
            }
        }

        if (matrix[i][i] != 0) {
            for (int j = i + 1; j < n; j++) {
                float f = -matrix[j][i] / matrix[i][i];
                for (int k = 0; k < n; k++) {
                    matrix[j][k] += f * matrix[i][k];
                }
            }
        }
    }

    for (int i = 0; i < n; i++) {
        determinant *= matrix[i][i];
    }

    return determinant;
}

pair<double, double> find_det(bool usingOMP, int n, vector<vector<float>>
matrix) {
    float det;

    auto begin = std::chrono::steady_clock::now();
    det = usingOMP ? findOMP_det(n, matrix) : findNonOMP_det(n, matrix);
    auto end = std::chrono::steady_clock::now();
    auto elapsed_ms = std::chrono::duration_cast<std::chrono::milliseconds>(end
- begin);

    return {det, elapsed_ms.count()};
}

bool read(const string &file, int &n, vector<vector<float>> &matrix) {
    ifstream in(file, ios::in);
    if (!in.is_open()) {
        return false;
    }

    in >> n;
    matrix.resize(n);
    for (int i = 0; i < n; i++) {
        matrix[i].resize(n);
        for (int j = 0; j < n; j++) {
            in >> matrix[i][j];
        }
    }
```

```
    in.close();
    return true;
}

bool write(const string &file, double &ans) {
    ofstream out(file, ios::out);
    if (!out.is_open()) {
        return false;
    }

    out << "Determinant: " << ans << " \n";

    out.close();
    return true;
}

double solve(const int &n, const vector<vector<float>> &matrix) {
    pair<double, double> det = find_det(true, n, matrix);
    pair<double, double> detCheck = find_det(false, n, matrix);

    double delta = fabs(det.first - detCheck.first);
    cerr << (delta / (det.first + detCheck.first) < 0.001 || (isinf(det.first)
&& isinf(detCheck.first)) ? "OK" : "Delta: " + to_string(delta)) << endl;

    cout << "Time (" << (threads_count == 0 ? "virtual processors parameters" :
to_string(threads_count) + " thread(s)") << "): " << det.second << "ms" << endl;
    cout << "Time (without OMP): " << detCheck.second << "ms" << endl;

    return det.first;
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        cout << "Requires <threads_count> <input_file> [<output_file>]";
        return 0;
    }

    threads_count = stoi(argv[1]);
    if (threads_count < 0) {
        cout << "Invalid threads count!";
        return 0;
    }
    int n;
    vector<vector<float>> matrix;

    if (!read(argv[2], n, matrix)) {
        cout << "Reading int input file failed!";
        return 0;
    }

    double det = solve(n, matrix);

    if (argc > 3) {
        if (!write(argv[3], det)) {
            cout << "Writing in output file failed!";
            return 0;
        }
    } else {
        cout << "Determinant: " << det << " \n";
```

```
    }

    return 0;
}
```

Компилятор “minGw w64 6.0”, параметры запуска “-std=c++17 -fopenmp”