

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИТМО
Дисциплина: Архитектура ЭВМ

Отчет
по домашней работе №5
«OpenMP»

Выполнил: Крайнов Данил Алексеевич
студ. гр. М3139

Санкт-Петербург
2020

Цель работы: знакомство со стандартом распараллеливания команд OpenMP.

Инструментарий и требования к работе: использован C++.

Теоретическая часть

OpenMP

OpenMP – это стандарт для распараллеливания программ.

Распараллеливание программ — это способ ускорить время работы той или иной программы, к которой это распараллеливание применимо. Для того, чтобы была возможность применить OpenMP, должны быть выполнены следующие условия:

- в программе должен быть участок кода, который может быть распараллелен, то есть выполнен не последовательно, а параллельно, с использованием дополнительных вычислительных мощностей (например, вложенные циклы или многократно повторяющиеся действия в одном цикле, обладающие свойством ассоциативности (например, многократное умножение переменной на элементы массива))
- компьютер, на котором запущена данная программа, должен обладать соответствующими техническими характеристиками (в нашем случае — несколько логических ядер процессора)

Как работает OpenMP:

Задачи распределяются и описываются между потоками с помощью специальных директив на стадии компиляции (в случае C++ - pragma)

Для описания такой директивы в C++ необходимо написать `#pragma omp ...` перед кодом, который должен выполняться несколькими потоками. На месте многоточия используется соответствующий синтаксис, который определяется структурой кода и тем, что в нем происходит.

Синтаксис OpenMP (использованный в решении):

- `parallel`: создание нескольких нитей. После `parallel` и начинается само распараллеливание кода, то есть распределение задач на установленное число потоков. Для этого основной поток (он есть всегда, `master thread`) создает подчиненные потоки (`slave`) так, что суммарное число потоков равно установленному заранее количеству. Клоз означает, что последующая часть кода будет распределена между всеми созданными потоками-нитеями
- `for`: означает, что будет распараллелен кусок кода, описанный в цикле `for`
- `schedule(scheduleType)`: принимает на вход параметр `scheduleType`, который определяет, как именно будет распределена работа между потоками
- `private(var1, var2, var3, ..., varN)`: определяет класс переменных, переданных в качестве аргументов. В данном случае это означает, что для каждого потока будет создан свой экземпляр каждой из переменных `var1, ..., varN`
- `shared(var1, var2, ..., varN)`: также определяет класс переменных. Переменные, переданные в качестве аргументов, будут являться общими для каждого из потоков.
- `reduction(operation:var)`: принимает на вход операцию (сложение `+`, умножение `*`, максимум `max`, минимум `min` и так далее (операции обладают

ассоциативностью)), которую нужно будет выполнить со всеми созданными для потоков экземплярами var. То есть в результате работы кода в оригинальный var будет записан результат выполнения var operation var operation var ... над всеми ее экземплярами (в случае умножения — все экземпляры перемножатся и будет записан результат умножения всех var, в случае максимума — максимум из всех экземпляров var, и так далее)

- `omp_set_num_threads(numThreads)`: устанавливает число нитей, которое будет использовано по умолчанию (default) равным переданному значению `numThreads` во всех участках кода, где происходит распараллеливание

- `omp_set_dynamic(arg)`: переданный `arg` разрешает или запрещает динамическое (то есть в ходе работы программы) изменение числа нитей. В нашем случае передаем `arg = 0`, запрещая такое изменение, чтобы проверить время работы при постоянном использовании того или иного числа потоков

Подробнее о параметре `scheduleType` и его возможных значениях:

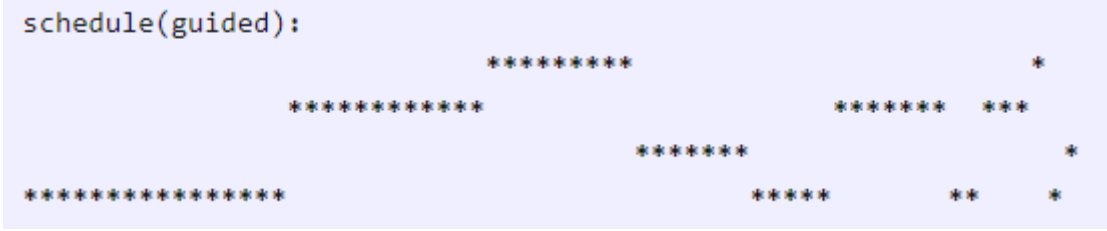
- `static`: итерации поровну распределяются между нитями последовательно (то есть блоками)

```
schedule(static):
*****
*****
*****
*****
```

- `dynamic`: наборы итераций фиксированного размера случайным образом распределяются между потоками

```
schedule(dynamic):
*  **  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
```

- guided: работает так же, как и dynamic, но размер наборов итераций изменяется в ходе выполнения кода и рассчитывается пропорционально: количество оставшихся итераций / количество потоков

- auto: 

ЭТОТ параметр переключает выбор scheduleType на компилятор

- runtime: откладывает выбор scheduleType до времени работы программы (до runtime)

-default: при отсутствии клоза schedule в прагме выставляется дефолтный scheduleType

Таким образом, OpenMP позволяет равномерно распределить нагрузку на несколько вычислительных мощностей, что позволяет проводить не последовательные, а параллельные вычисления / операции, тем самым сильно сокращая время работы программы на входных данных значительного размера.

Практическая часть

Выполнен вариант 7 (нахождение определителя матрицы).

Для нахождения определителя использован метод Гаусса — приведение матрицы к треугольному виду и произведение элементов главной диагонали в качестве ответа.

`maxElem` хранит расположение максимального по модулю элемента в столбце

Если он не равен текущему, то меняем местами строки матрицы, не забывая поменять знак определителя (`swar` происходит за $O(1)$, так как матрица хранится как массив указателей на массивы — то есть свапаем указатели).

Если на главной диагонали оказался нулевой элемент, то определитель матрицы равен 0 и дальнейшие вычисления бессмысленны (`return 0`), иначе приводим матрицу к треугольному виду. Здесь и используем первое распараллеливание:

```
#pragma omp parallel for schedule(static) private(j, k, tmp)
    for (j = i + 1; j < n; j++) {
        tmp = array[j][i] / array[i][i];
        for (k = 0; k < n; k++) {
            array[j][k] -= tmp * array[i][k];
        }
    }
```

Цикл `for` распараллелен между потоками (нет `shared(i)`, так как на практике оказалось, что время работы программы увеличивается в 1.5 раза, а переменная `i` на данном участке кода и так является глобальной). Для каждого потока создадим приватные переменные `j`, `k`, `tmp`, чтобы разные потоки обрабатывали разные части нашей матрицы.

После приведения матрицы к треугольному виду считаем ответ — произведение элементов на главной диагонали. Для этого так же воспользуемся

возможностями OpenMP для проведения параллельных вычислений, так как умножение обладает ассоциативностью, и ответ — произведение всех произведений, то есть `reduction(*:ans)`:

```
#pragma omp parallel for schedule(static) reduction(*:ans)
    for (i = 0; i < n; i++) {
        ans *= array[i][i];
    }
    return ans;
```

В `main` из командной строки парсится количество потоков `numThreads`, после чего выставляются соответствующие параметры:

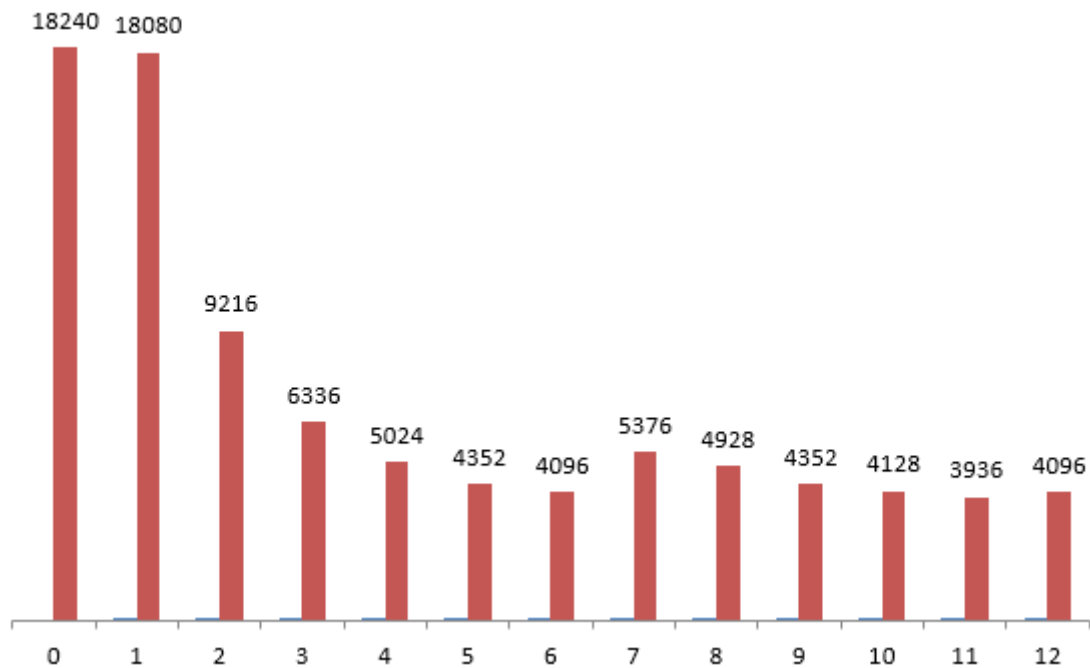
```
omp_set_dynamic(0);
omp_set_num_threads(numThreads);
```

`Output` используется в случае, если в командной строке передали файл вывода. Далее идет стандартный ввод из файла (строки `n = 2000` и `array[i][j] = rand() % 10` закомментированы, они были использованы для проверки времени работы программы при больших входных данных ($O(n^3)$ при `n = 2000`) и построения наглядных графиков).

Время работы вычисляется как разность `end - start`, где `start = GetTickCount()` до вызова функции `getAns()`, а `end = GetTickCount()` после выполнения `getAns()`. Далее идет вывод с закрытием файла при необходимости.

Графики

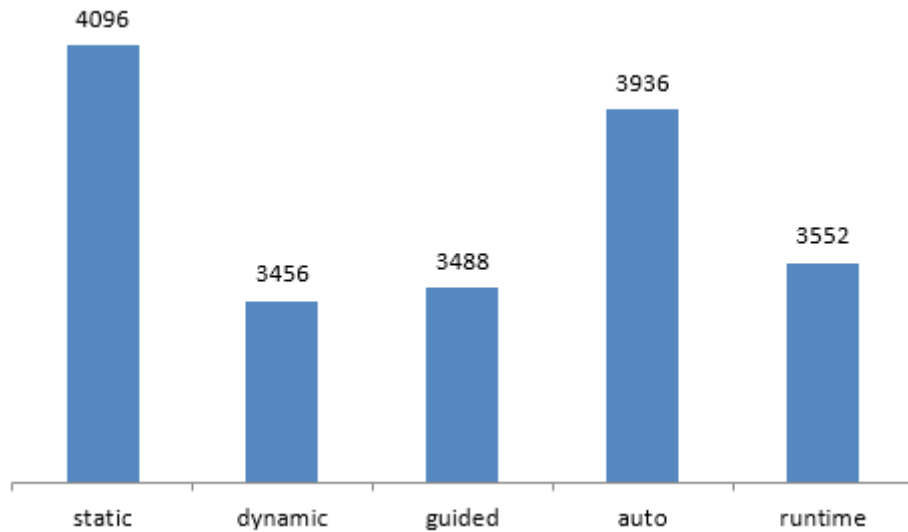
Время работы от числа потоков



Представлена зависимость времени работы программы от значения первого аргумента командной строки. От 0 до 12, так как у моего процессора 12 логических ядер (рассмотрение больших чисел не имеет смысла, так как результат будет схож с одним из описанных на графике).

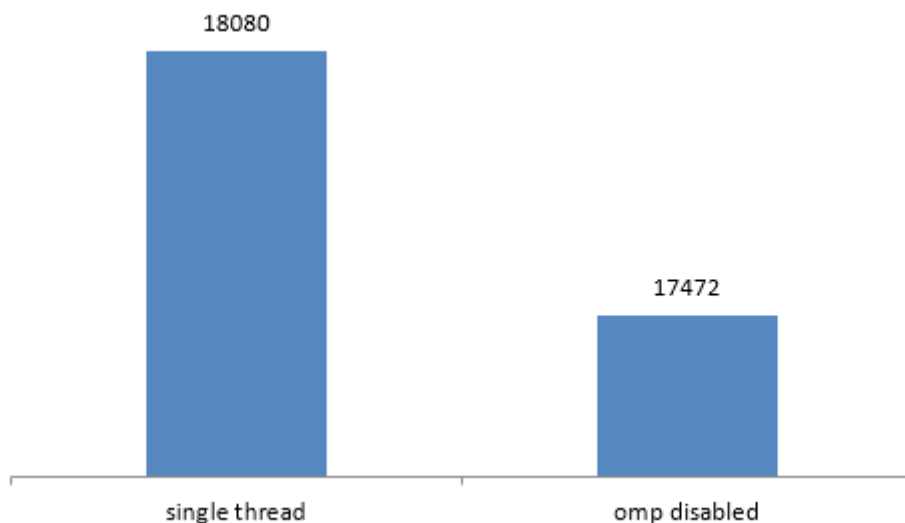
Можно заметить, что при подключении второго потока, время работы сокращается вдвое (что предельно логично). Подобная логика прослеживается примерно до 6 потоков. Далее увеличение числа потоков не несет практического смысла, что видно по графику. Напротив, время работы программы даже немного дольше, что связано с более длительным распределением и созданием нитей, и возвращается к оптимальному только при использовании 10-12 ядер. Но, при больших входных ($n \geq 5000$) данных использование большего числа ядер дало бы более видимый результат.

Время работы от значения параметра schedule



Можно заметить, что наиболее оптимальными являются `dynamic` и `guided`. Это логично, так как при приведении матрицы к треугольному виду с каждой итерацией по i уменьшается количество итераций по j , из-за чего статичный размер набора итераций для потоков не так выгоден на последних итерациях по i . Можно заметить, что именно какой-то из этих параметров выбирается системой при использовании типа `runtime`. Также можно заметить, что компилятор (при выборе `auto`) использует параметр `static`.

Разница времени работы с одним потоком и отключенным `omp`



Можно заметить, что при отключенном `omp` (для этого просто

закомментировал все строки, связанные с omp: прагмы, omp_set...), наблюдается небольшой выигрыш по времени. Это связано с тем, что независимо от числа потоков, программе требуется время для выполнения кода из omp и распределения данных между потоками (пусть он и один) и постоянного обновления наборов итераций для этого единственного потока.

Таким образом, OpenMP позволяет значительно ускорить время работы программы, что особенно ощущается на входных данных больших размеров и высокой асимптотической сложности программы. Однако, выбор параметров schedule, числа потоков также сильно влияет на время, из чего можно сделать вывод, что нужно внимательно подходить к их выбору для достижения оптимального результата. При этом для разных входных данных, разных характеристиках техники эти параметры будут отличаться.

Листинг

main.cpp

```
#include <iostream>
#include <vector>
#include <omp.h>
#include <fstream>
#include <sysinfoapi.h>

using namespace std;

const float eps = 1e-6;

int n, numThreads, i, j, k;
vector<float *> array;

float getAns() {
    float ans = 1, maxElem, tmp;
    for (i = 0; i < n - 1; i++) {
        maxElem = i;
        for (j = i + 1; j < n; j++) {
            if (abs(array[j][i]) > abs(array[i][i])) {
                maxElem = j;
            }
        }
        if (maxElem != i) {
            ans = -ans;
            swap(array[i], array[maxElem]);
        }
        if (abs(array[i][i]) > eps) {
#pragma omp parallel for schedule(static) private(j, k, tmp)
            for (j = i + 1; j < n; j++) {
                tmp = array[j][i] / array[i][i];
                for (k = 0; k < n; k++) {
                    array[j][k] -= tmp * array[i][k];
                }
            }
        } else {
            return 0;
        }
    }
#pragma omp parallel for schedule(static) reduction(*:ans)
    for (i = 0; i < n; i++) {
        ans *= array[i][i];
    }
    return ans;
}

int main(int argc, char *argv[]) {
    ios::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
```

```

{
    string s = argv[1];
    for (i = 0; i < (int) s.size(); i++) {
        numThreads = numThreads * 10 + s[i] - '0';
    }
}
omp_set_dynamic(0);
omp_set_num_threads(numThreads);
freopen(argv[2], "r", stdin);
ofstream output;
if (argc == 4) {
    output.open(argv[3]);
}
cin >> n;
//n = 2000;
array.resize(n);
for (i = 0; i < n; i++) {
    array[i] = new float[n];
}
srand(1);
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        cin >> array[i][j];
        //array[i][j] = rand() % 10;
    }
}
float start, end, ans;
start = GetTickCount();
ans = getAns();
end = GetTickCount();
if (output.is_open()) {
    output << "Determinant: " << ans << '\n';
    output.close();
} else {
    cout << "Determinant: " << ans << '\n';
}
cout << "Time: " << end - start;
}

```

CmakeList.txt

```

cmake_minimum_required(VERSION 3.17)
project(hw5)

set(CMAKE_CXX_STANDARD 17)
find_package(OpenMP)
if (OPENMP_FOUND)
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
    set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} $
{OpenMP_EXE_LINKER_FLAGS}")

```

```
else()  
    message(FATAL_ERROR "OpenMP is not found!")  
endif ()  
add_executable(hw5 main.cpp)
```