Лабораторная работа №4	M3139	2023
OpenMP	Гришечкин Павел Аверьянович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: Стандарт OpenMP 2.0. Язык C++. Компилятор g++ версии 8.1.0.

Описание конструкций OpenMP для распараллеливания команд.

В ходе решения задачи были использованы следующие конструкции OpenMP:

- `#pragma omp parallel`
- `#pragma omp for nowait`
- `#pragma omp critical`

Parallel

Данная конструкция является обозначением части программы, которая будет выполняться многопоточно

For

Данная конструкция ставится перед for-ом. Данный for будет “распараллелен”, то есть каждый поток будет исполнять свой набор итераций и эти наборы не пересекаются.

Флаг nowait позволяет потокам, которые закончили свою часть итераций исполнять код дальше, не дожидаясь завершения всех итераций.

Critical

Данная конструкция гарантирует, что соответствующий блок кода будет выполнен только одним потоком одновременно.

Описание работы программы

(Все фрагменты кода взяты из файла hard.cpp)

```
const short int L = 255;
const short int M = 4;
const uint8_t C0 = 0;
const uint8_t C1 = 84;
const uint8_t C2 = 170;
const uint8_t C3 = 255;
const int RUN_COUNT = 1000;
```

Листинг кода 1 – Константы

L – количество оттенков

M – число порогов

C%i – оттенок, соответствующий iму классу

RUN_COUNT – количество запусков программы (для вычисления среднего времени работы)

```
int main(int number_of_arguments, char *args[]) {
    assert("Incorrect number of arguments" && number_of_arguments == 4);
    int number_of_threads;
    try {
        number_of_threads = stoi(args[1]);
    } catch (...) {
        cerr << "Incorrect arguments";
        return 0;
    }
    assert("Incorrect number of threads" && number_of_threads >= -1);
    ...
```

Листинг кода 2 – Проверка корректности аргументов

```
ifstream input_file(args[2], ios_base::binary);
input_file.exceptions(ifstream::failbit);

try {
    input_file.seekg(0, ifstream::end);
    long long size = input_file.tellg();
    input_file.seekg(0, ifstream::beg);

    auto *data = new uint8_t[size];
    try {
        input_file.read((char*) data, size);
```

Листинг кода 3 – Ввод данных из файла

В приведенном выше фрагменте кода описан ввод всех байт входных данных.

Метод seekg перемещает указатель чтения файла на указанную позицию (либо с начала, либо с конца)

Метод telling возвращает позицию указателя.

В результате работы кода в массиве data будут содержаться все байты входных данных.

```
string p5;
p5 += data[0];
p5 += data[1];
p5 += data[2];
assert(p5 == "P5\n");

string w, h;
int pos = 3;
while (data[pos] != ' ') w += data[pos++];
pos++;
while (data[pos] != '\n') h += data[pos++];

try {
    count_of_elements = stoi(w) * stoi(h);
} catch (...) {
    cerr << "Incorrect data";
    input_file.close();
    return 0;
}

pos++;
string s255;
s255 += data[pos++];
s255 += data[pos++];
s255 += data[pos++];
s255 += data[pos++];
assert(s255 == "255\n");
assert(pos + count_of_elements == size);
```

Листинг кода 4 – Проверка корректности входного файла

В фрагменте кода выше проверяется соответствие входного файла стандарту из т/з.

В результате работы кода pos будет указывать на первый байт массива оттенков.

Переменная count_of_elements хранит количество элементов в массиве оттенков.

Алгоритм без использования OpenMp

```
void solution_without_parallel(uint8_t* data, int begin_pos, int size) {
    long long gist[L + 1];
    for (long long & i : gist) {
        i = 0;
    }
    for (int i = begin_pos; i < size; i++) {
        gist[data[i]]++;
    }

    for (int i = 0; i < L + 1; i++) {
        prefsum[i + 1] = prefsum[i] + gist[i];
        prefsum_m[i + 1] = prefsum_m[i] + gist[i] * i;
    }
    ...
```

Листинг кода 5 – Вычисление гистограммы и предподсчет префиксных сумм

Переменная begin_pos указывает на начало массива оттенков

Переменная size равна размеру входных данных

Так как не гарантируется возможность хранения двух изображений в памяти, то итоговый результат будет записан в массив входных данных.

```
double max_delta = 0;
    int ans[3];
    double delta;
    size_t m1;
    size_t m2;
    size_t m3;
    size_t m4;
    for (short int f0 = 0; f0 < L - M; f0++) {
        for (short int f1 = f0 + 1; f1 < (short int) (L - M + 1); f1++) {
            for (short int f2 = f1 + 1; f2 < (short int) (L - M + 2); f2++)
{
                m1 = calc_m(0, f0);
                m2 = calc_m(f0 + 1, f1);
                m3 = calc_m(f1 + 1, f2);
                m4 = calc_m(f2 + 1, L);

                delta = (double) (m1 * m1) / calc_q(0, f0);
                delta += (double) (m2 * m2) / calc_q(f0 + 1, f1);
                delta += (double) (m3 * m3) / calc_q(f1 + 1, f2);
                delta += (double) (m4 * m4) / calc_q(f2 + 1, L);

                if (delta > max_delta) {
                    max_delta = delta;
                    ans[0] = f0;
                    ans[1] = f1;
                    ans[2] = f2;
                }
            }
        }
    }
```

Листинг кода 6 – Вычисление межкластерной дисперсии и порогов

```
#define calc_q(left, right) (prefsum[right + 1] - prefsum[left])
#define calc_m(left, right) (prefsum_m[right + 1] - prefsum_m[left])
```

Листинг кода 7 – Выражения calc_m и calc_q

Алгоритм Оцу заключается в поиске порогов таких, что межклассовая дисперсия классов, образованных этими порогами, была максимальна.

Изначальная формула дисперсии (анализируемого значения) имеет следующий вид:

$$\sigma^2 = \sum_{i=1}^{4} q_i * m_i^2$$

$$q_i = \sum \frac{p_i}{N}$$

$$m_i = \sum \frac{i * p_i}{q_i * N}$$

$$p_i = \sum \frac{n_i}{N}$$

Так как N – константа, то можно вынести N и упростить анализируемое выражение:

$$\sigma_1^2 = \sum_{i=1}^{4} \left(\sum j * p_j\right)^2 / \sum p_j$$

Для быстрого вычисления суммы на отрезке можно использовать префиксные суммы.

В листинге кода 6:

- prefsum[i] = $\sum_{j=0}^{i-1} n_j$
- prefsum_m[i] = $\sum_{j=0}^{i-1} n_j * j$

Тогда итоговая формула анализируемого значения будет иметь следующий вид:

$$\sigma_2^2 = \sum_{i=1}^{4} \frac{(prefsum\_m_{r_i} - prefsum\_m_{l_i-1})^2}{prefsum_{r_i} - prefsum_{l_i-1}}$$

В листинге кода 6 перебираются все возможные значения порогов, и ищется тот, у которого межклассовая дисперсия максимальна.

```
for (int i = begin_pos; i < size; i++) {
    if (data[i] < ans[0]) data[i] = C0;
    else if (data[i] < ans[1]) data[i] = C1;
    else if (data[i] < ans[2]) data[i] = C2;
    else data[i] = C3;
}
```

Листинг кода 8 – Перезапись заданной картинки

Если флаг rewrite равен true, то массив входных данных будет изменен на массив выходных данных.

Изображение 1 – Сравнение изначального изображения с полученным

Как можно видеть на изображении сверху, алгоритм успешно разбивает картинку на 4 цветовых класса.

Описание распараллеливания с использованием OpenMp

```
void solution_with_parallel(uint8_t* data, int begin_pos, int size, bool
rewrite) {
    long long gist[L + 1];

#pragma omp parallel
    {
#pragma omp for
        for (short int i = 0; i < L + 1; i++) {
            gist[i] = 0;
        }
        long long thread_gist[L + 1];
        for (short int i = 0; i < L + 1; i++) {
            thread_gist[i] = 0;
        }

#pragma omp for nowait
        for (int i = begin_pos; i < size; i++) {
            thread_gist[data[i]]++;
        }

#pragma omp critical
        {
            for (short int i = 0; i < L + 1; i++) {
                gist[i] += thread_gist[i];
            }
        };
```

Листинг кода 9 – Распараллеленное нахождение гистограммы

Для каждого потока создается массив thread_gist, в котором подсчитывается количество каждого оттенка для данного потока.

В critical блоке все массивы суммируются в один.

```
for (short int i = 0; i < L + 1; i++) {
    prefsum[i + 1] = prefsum[i] + gist[i];
    prefsum_m[i + 1] = prefsum_m[i] + gist[i] * i;
}
```

Листинг кода 10 – Вычисление префиксных сумм

Вычисление префиксных сумм можно оптимизировать до L/2 + L/(2 * <количество потоков>), но т.к. L = 255, то данная оптимизация является незначительной.

```
double max_delta = 0;
short int ans[3];
#pragma omp parallel
{
    double thread_max_delta = 0;
    short int thread_ans[3];
    double delta;
    size_t m1;
    size_t m2;
    size_t m3;
    size_t m4;
    for (short int f0 = 0; f0 < L - M; f0++) {
        for (short int f1 = f0 + 1; f1 < (short int) (L - M + 1); f1++) {
#pragma omp for nowait
            for (short int f2 = f1 + 1; f2 < (short int) (L - M + 2); f2++)
{
                m1 = calc_m(0, f0);
                m2 = calc_m(f0 + 1, f1);
                m3 = calc_m(f1 + 1, f2);
                m4 = calc_m(f2 + 1, L);

                delta = (double) (m1 * m1) / calc_q(0, f0);
                delta += (double) (m2 * m2) / calc_q(f0 + 1, f1);
                delta += (double) (m3 * m3) / calc_q(f1 + 1, f2);
                delta += (double) (m4 * m4) / calc_q(f2 + 1, L);

                if (delta > thread_max_delta) {
                    thread_max_delta = delta;
                    thread_ans[0] = f0;
                    thread_ans[1] = f1;
                    thread_ans[2] = f2;
                }
            }
```

```
        }
    }
#pragma omp critical
    {

        if (max_delta < thread_max_delta) {
            max_delta = thread_max_delta;
            swap(ans, thread_ans);
        }
    };
}
```

Листинг кода 11 – Распараллеленное вычисление порогов

Алгоритм схож с алгоритмом подсчета гистограммы.

Для каждого потока вычисляется максимальная дисперсия, после чего в critical блоке значения потоков объединяются.

Так как нельзя распараллелить все циклы, то будем разбивать на потоки только один из циклов.

Самым оптимальным решением будет разбиение внутреннего цикла на потоки, так как в нем на каждой итерации выполняется меньше всего операций и разбиение на потоки будет более точным.

```
#pragma omp parallel
    {
        #pragma omp for
        for (int i = begin_pos; i < size; i++) {
            if (data[i] < ans[0]) data[i] = C0;
            else if (data[i] < ans[1]) data[i] = C1;
            else if (data[i] < ans[2]) data[i] = C2;
            else data[i] = C3;
        }
    }
```

Листинг кода 12 – Распараллеленная перезапись входных данных

Результаты

Тестирование производилось на процессоре AMD Ryzen 5 3500U

Результат работы при стандартном количестве потоков:

```
77 130 187
Time (default number of threads): 14.7712 ms
```

Экспериментальная часть

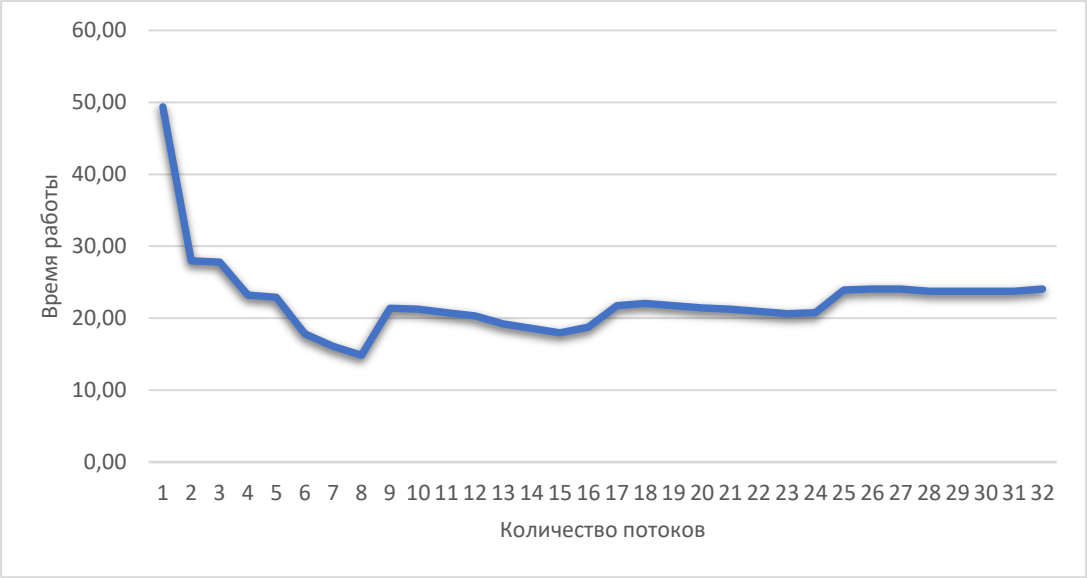

График 1 – Зависимость времени работы от количества потоков

Из графика 1 можно заметить, что время работы быстро убывает при увеличении числа потоков программы до числа потоков процессора (в данном случае 8 потоков), после чего резко возрастает и медленно убывает до 16. После этого время начинает постепенно увеличиваться.

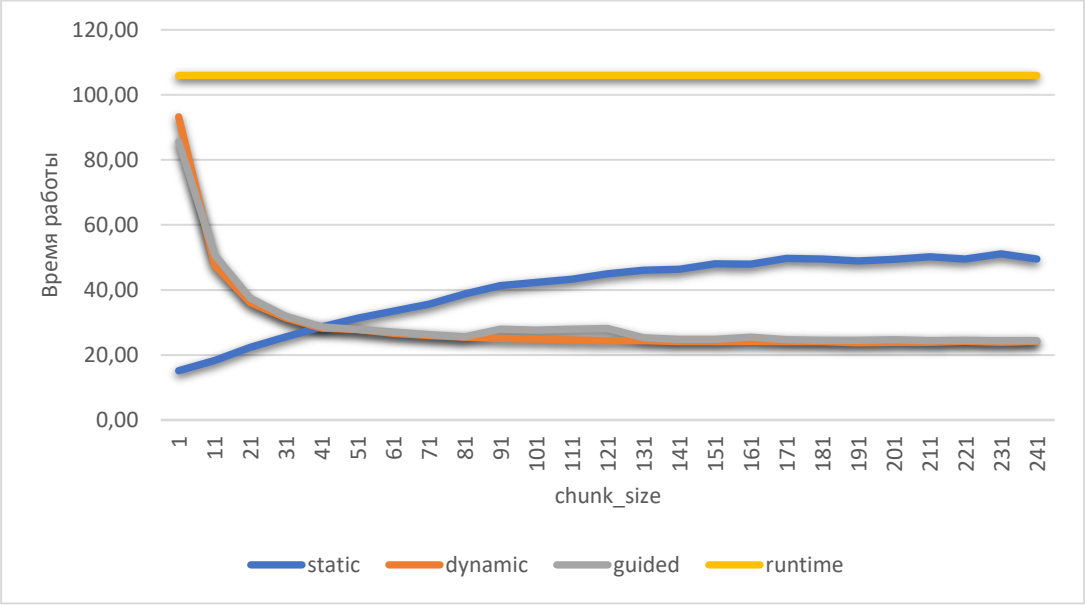

График 2 – Зависимость времени работы от chunk_size

При замерах параметр schedule подставлялся только в цикл для вычисления порогов, так как размеры циклов сильно отличаются.

Schedule позволяет настроить директиву for. У schedule есть 4 режима работы:

- Static – перед выполнением цикла разбивает итерации на блоки в соответствии с chunk_size. Каждому блоку соответствует свой поток.
- Dynamic – перед выполнением цикла разбивает итерации на блоки в соответствии с chunk_size. В процессе выполнения цикла каждый освободившийся поток обращается за новым свободным блоком итераций.
- Guided – перед выполнением цикла разбивает итерации на блоки так, что размер каждого блока примерно равен количеству итераций, деленному на количество потоков. Оставшаяся часть разделяется на блоки, уменьшающейся до chunk_size длины. Эти блоки динамически раздаются потокам.
- Runtime – решение о разбиении на блоки производится во время выполнения программы. Этот режим не зависит от chunk_size.

По графику 2 видно, что минимальное время работы достигается при статическом разбиении на блоки длины, равной 1. Это происходит из-за того, что динамическое сопоставление каждого блока потоку занимает много времени, а при увеличении размера блока какие-то потоки будут выполнять больше операций чем другие.

Guided распределение почти совпадает с dynamic распределением. При увеличении размера блока количество обращений на выделение нового блока потоку уменьшается, в следствии чего время уменьшается.

Runtime наименее оптимален.

Результат работы программы без OpenMP:

```
77 130 187
Time without multithreading: 72.34 ms
```

Результат работы программы с OpenMP, включенным 1 потоком:

```
77 130 187
Time (1 thread(s)): 67.03 ms
```

Можно заметить, что выполнение с 1 потоком немного оптимальнее выполнения без разбиения на потоки.

Источники

https://learn.microsoft.com/en-us/cpp/parallel/openmp/openmp-c-and-cpp-application-program-interface?view=msvc-170

Листинг кода

hard.cpp

```cpp
#include "iostream"
#include "omp.h"
#include "fstream"
#include "string"
#include "iomanip"
#include "cassert"

#define calc_q(left, right) (prefsum[right + 1] - prefsum[left])
#define calc_m(left, right) (prefsum_m[right + 1] - prefsum_m[left])

using namespace std;

const short int L = 255;
const short int M = 4;
const uint8_t C0 = 0;
const uint8_t C1 = 84;
const uint8_t C2 = 170;
const uint8_t C3 = 255;

size_t count_of_elements;
size_t prefsum[L + 2];
size_t prefsum_m[L + 2];

void solution_without_parallel(uint8_t* data, int begin_pos, int size) {
    long long gist[L + 1];
    for (long long & i : gist) {
        i = 0;
    }
    for (int i = begin_pos; i < size; i++) {
        gist[data[i]]++;
    }

    for (int i = 0; i < L + 1; i++) {
        prefsum[i + 1] = prefsum[i] + gist[i];
        prefsum_m[i + 1] = prefsum_m[i] + gist[i] * i;
    }

    double max_delta = 0;
    int ans[3];
```

```
    double delta;
    size_t m1;
    size_t m2;
    size_t m3;
    size_t m4;
    for (short int f0 = 0; f0 < L - M; f0++) {
        for (short int f1 = f0 + 1; f1 < (short int) (L - M + 1); f1++) {
            for (short int f2 = f1 + 1; f2 < (short int) (L - M + 2); f2++)
{
                m1 = calc_m(0, f0);
                m2 = calc_m(f0 + 1, f1);
                m3 = calc_m(f1 + 1, f2);
                m4 = calc_m(f2 + 1, L);

                delta = (double) (m1 * m1) / calc_q(0, f0);
                delta += (double) (m2 * m2) / calc_q(f0 + 1, f1);
                delta += (double) (m3 * m3) / calc_q(f1 + 1, f2);
                delta += (double) (m4 * m4) / calc_q(f2 + 1, L);

                if (delta > max_delta) {
                    max_delta = delta;
                    ans[0] = f0;
                    ans[1] = f1;
                    ans[2] = f2;
                }
            }
        }
    }

    for (int i = begin_pos; i < size; i++) {
        if (data[i] < ans[0]) data[i] = C0;
        else if (data[i] < ans[1]) data[i] = C1;
        else if (data[i] < ans[2]) data[i] = C2;
        else data[i] = C3;
    }

    cout << ans[0] << " " << ans[1] << " " << ans[2] << "\n";
}

void solution_with_parallel(uint8_t* data, int begin_pos, int size) {
    long long gist[L + 1];

#pragma omp parallel
    {
#pragma omp for
        for (short int i = 0; i < L + 1; i++) {
            gist[i] = 0;
        }
```

```
        long long thread_gist[L + 1];
        for (short int i = 0; i < L + 1; i++) {
            thread_gist[i] = 0;
        }

#pragma omp for nowait
        for (int i = begin_pos; i < size; i++) {
            thread_gist[data[i]]++;
        }

#pragma omp critical
        {
            for (short int i = 0; i < L + 1; i++) {
                gist[i] += thread_gist[i];
            }
        };
    };

    for (short int i = 0; i < L + 1; i++) {
        prefsum[i + 1] = prefsum[i] + gist[i];
        prefsum_m[i + 1] = prefsum_m[i] + gist[i] * i;
    }

    double max_delta = 0;
    short int ans[3];
    #pragma omp parallel
    {
        double thread_max_delta = 0;
        short int thread_ans[3];
        double delta;
        size_t m1;
        size_t m2;
        size_t m3;
        size_t m4;
        for (short int f0 = 0; f0 < L - M; f0++) {
            for (short int f1 = f0 + 1; f1 < (short int) (L - M + 1); f1++)
{
    #pragma omp for nowait
                for (short int f2 = f1 + 1; f2 < (short int) (L - M + 2);
f2++) {
                    m1 = calc_m(0, f0);
                    m2 = calc_m(f0 + 1, f1);
                    m3 = calc_m(f1 + 1, f2);
                    m4 = calc_m(f2 + 1, L);

                    delta = (double) (m1 * m1) / calc_q(0, f0);
                    delta += (double) (m2 * m2) / calc_q(f0 + 1, f1);
                    delta += (double) (m3 * m3) / calc_q(f1 + 1, f2);
```

```
                    delta += (double) (m4 * m4) / calc_q(f2 + 1, L);

                    if (delta > thread_max_delta) {
                        thread_max_delta = delta;
                        thread_ans[0] = f0;
                        thread_ans[1] = f1;
                        thread_ans[2] = f2;
                    }
                }
            }
        }
    #pragma omp critical
        {
            if (max_delta < thread_max_delta) {
                max_delta = thread_max_delta;
                ans[0] = thread_ans[0];
                ans[1] = thread_ans[1];
                ans[2] = thread_ans[2];
            }
        };
    }

#pragma omp parallel
    {
        #pragma omp for
        for (int i = begin_pos; i < size; i++) {
            if (data[i] < ans[0]) data[i] = C0;
            else if (data[i] < ans[1]) data[i] = C1;
            else if (data[i] < ans[2]) data[i] = C2;
            else data[i] = C3;
        }
    }

    cout << ans[0] << " " << ans[1] << " " << ans[2] << "\n";
}

int main(int number_of_arguments, char *args[]) {
    assert("Incorrect number of arguments" && number_of_arguments == 4);
    int number_of_threads;
    try {
        number_of_threads = stoi(args[1]);
    } catch (...) {
        cerr << "Incorrect arguments";
        return 0;
    }
    assert("Incorrect number of threads" && number_of_threads >= -1);

    cout << fixed << setprecision(5);
```

```
try {
    ifstream input_file(args[2], ios_base::binary);
    input_file.exceptions(ifstream::failbit);

    try {
        input_file.seekg(0, ifstream::end);
        long long size = input_file.tellg();
        input_file.seekg(0, ifstream::beg);

        auto *data = new uint8_t[size];
        try {
            input_file.read((char*) data, size);

            string p5;
            p5 += data[0];
            p5 += data[1];
            p5 += data[2];
            assert(p5 == "P5\n");

            string w, h;
            int pos = 3;
            while (data[pos] != ' ') w += data[pos++];
            pos++;
            while (data[pos] != '\n') h += data[pos++];

            try {
                count_of_elements = stoi(w) * stoi(h);
            } catch (...) {
                cerr << "Incorrect data";
                input_file.close();
                return 0;
            }

            pos++;
            string s255;
            s255 += data[pos++];
            s255 += data[pos++];
            s255 += data[pos++];
            s255 += data[pos++];
            assert(s255 == "255\n");
            assert(pos + count_of_elements == size);

            double begin_time = omp_get_wtime();
            if (number_of_threads == -1) {
                solution_without_parallel(data, pos, size);
            } else {
```

```
                    if (number_of_threads != 0)
omp_set_num_threads(number_of_threads);
                    solution_with_parallel(data, pos, size);
                }
                printf("Time (%i thread(s)): %g ms\n", number_of_threads,
(omp_get_wtime() - begin_time) * 1000);

                ofstream output_file(args[3], ios_base::binary);
                try {
                    output_file.write((char*) data, size);
                } catch (...) {
                    output_file.close();
                    input_file.close();
                    delete[] data;
                    cerr << "Error";
                    return 0;
                }

                output_file.close();
                input_file.close();
                delete[] data;
            } catch (...) {
                input_file.close();
                delete[] data;
                cerr << "Error";
                return 0;
            }
        } catch (...) {
            input_file.close();
            cerr << "Error";
            return 0;
        }
    } catch (...) {
        cout << "Error" << endl;
        return 0;
    }

    return 0;
}
```