Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №5 по курсу "Анализ алгоритмов"

Тема Организация асинхронного взаимодействия потоков

Студент Морозов Д.В.

Группа ИУ7-52Б

Оценка (баллы)

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2023 г.

Оглавление

Введение

Целью данной лабораторной работы является реализация асинхронного взаимодействия потоков на примере конвейерной обработки данных в алгоритме сжатия Хаффмана.

Задачи данной лабораторной:

1) описание алгоритма Хаффмана;
2) реализация однопоточной и многопоточной версии алгоритма;
3) сравнение затрат реализаций по времени выполнения.

1 Аналитическая часть

В этом разделе представлено описание алгоритма.

1.1 Сжатие Хаффмана

Одним из основных методов, используемыми для сжатия данных в базах данных в настоящее время является кодирование Хаффмана на основе кодового бинарного дерева. Код Хаффмана является неравномерным и префиксным. Неравномерность кода означает, что символы имеют разную длину кодового слова (размер кода), коды символов, которые чаще встречаются в тексте, имеют меньший размер, а коды редко встречающихся символов, имеют больший размер. Префиксный называется такая кодировка, в которой ни один код не является началом другого кода, таким образом выполняется условие Фано и достигается однозначность при декодировании [1].

Метод сжатия информации на основе двоичных кодирующих деревьев был предложен Д. А. Хаффманом в 1952. Идея алгоритма состоит в том, чтобы наиболее часто встречающиеся символы имели более короткие коды, символы, встречающиеся реже всего, имели очень длинный код. Алгоритм сжатия данных Хаффмана, как канонический, так и его адаптивные версии, обладают достаточно высокой эффективностью и лежат в основе многих других методов, используемых в алгоритмах сжатия данных. Каждому символу в кодируемом тексте присваивается вес, равный частоте его появления. Затем формируется бинарное дерево кодировки, в котором каждый символ является листом. Путь вправо — добавление единицы в текущий код, путь влево — добавление нуля [1].

Алгоритм сжатия можно описать с помощью 3 этапов.

1) Создание конечного узла для каждого символа и добавление его в очередь приоритетов.

2) Пока в очереди больше одного узла: удаление из очереди двух узлов с наивысшим приоритетом (самой низкой частотой); создание нового

внутреннего узла с этими двумя узлами в качестве дочерних элементов и частотой, равной сумме частот обоих узлов; добавление нового узла в очередь приоритетов.

3) Обход сформированного дерева и формирование кода Хаффмана.

Итак, строка aabacdab будет закодирована в 00110100011011 (0|0|11|0|100|011|0|11), с помощью следующих кодов:

— a — 0;

— b — 11;

— c — 100;

— d — 011.

2 Конструкторская часть

В этом разделе приведена схема алгоритма.

2.1 Алгоритм Хаффмана

На рисунке 2.1 приведена схема алгоритма Хаффмана.

Рисунок 2.1 – Схема алгоритма Хаффмана

3 Технологическая часть

В данном разделе представлены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к программному обеспечению

Используемое программное обеспечение должно предоставлять возможность измерения процессорного времени и работы с потоками.

3.2 Средства реализации

Для реализации данной лабораторной работы был выбран язык программирования C++ [2] и среда разработки CLion, которая позволяет замерять процессорное время с помощью пакета `<ctime>` [3] и работать с потоками, используя класс std::thread [4].

3.3 Реализация алгоритма

В листингах 3.1 и 3.2–3.4 приведены реализации классов заявки и конвейера, в листингах 3.5–3.8 приведены раелизации этапов алгоритма Хаффмана.

Листинг 3.1 – Класс Request

```
class Request {
public:
    Request(string srcData) : srcData{srcData} {}
    string srcData;
    string result;
    unordered_map<char, int> freq;
    priority_queue<shared_ptr<Node>, vector<shared_ptr<Node>>,
       NodeComparator> pq;
    unordered_map<char, string> huffmanCode;

    static shared_ptr<Request> createRequest() {
        return
           make_shared<Request>(StringGenerator::genString(10));
    }
};
```

Листинг 3.2 – Класс Conveyor (часть 1)

```
class Conveyor {
public:
    Conveyor(){}

    void runParallel(size_t cntRequests) {
        for (size_t i = 0; i < cntRequests; i++) {
            auto r = Request::createRequest();
            q1.push(r);
        }

        this->threads[0] = std::thread(&Conveyor::stage1, this);
        this->threads[1] = std::thread(&Conveyor::stage2, this);
        this->threads[2] = std::thread(&Conveyor::stage3, this);

        for (int i = 0; i < THRD_CNT; i++) {
            this->threads[i].join();
        }

        for (size_t i = 0; i < cntRequests; i++) {
            auto r = processedRequests[i];
            Huffman::printInfo(r);
        }
    }
```

Листинг 3.3 – Класс Conveyor (часть 2)

```
private:
    void stage1() {
        while (!this->q1.empty()) {
            auto r = q1.front();
            Huffman::stage1(r);

            q2.push(r);
            q1.pop();
        }
    }

    void stage2() {
        do {
            if (!this->q2.empty()) {
                auto r = q2.front();
                Huffman::stage2(r);

                q3.push(r);
                q2.pop();
            }
        } while(!q1.empty() || !q2.empty());
    }

    void stage3() {
        do {
            if (!this->q3.empty()) {
                auto r = q3.front();
                Huffman::stage3(r);

                processedRequests.push_back(r);
                q3.pop();
            }
        } while(!q1.empty() || !q2.empty() || !q3.empty());
    }

private:
    std::thread threads[THRD_CNT];
    std::vector<std::shared_ptr<Request>> processedRequests;
```

Листинг 3.4 – Класс Conveyor (часть 3)

```
    std::queue<std::shared_ptr<Request>> q1;
    std::queue<std::shared_ptr<Request>> q2;
    std::queue<std::shared_ptr<Request>> q3;
};
```

Листинг 3.5 – 1-й этап алгоритма Хаффмана

```
void stage1(shared_ptr<Request> r) {
    for (char ch: r->srcData) {
        r->freq[ch]++;
    }

    for (auto pair: r->freq) {
        r->pq.push(make_shared<Node>(pair.first, pair.second,
            nullptr, nullptr));
    }
}
```

Листинг 3.6 – 2-й этап алгоритма Хаффмана

```
void stage2(shared_ptr<Request> r) {
    while (r->pq.size() > 1) {
        shared_ptr<Node> left = r->pq.top(); r->pq.pop();
        shared_ptr<Node> right = r->pq.top(); r->pq.pop();

        int sum = left->freq + right->freq;
        r->pq.push(make_shared<Node>('\0', sum, left, right));
    }
}
```

Листинг 3.7 – 3-й этап алгоритма Хаффмана

```
void stage3(shared_ptr<Request> r) {
    encode(r->pq.top(), "", r->huffmanCode);

    for (char ch: r->srcData) {
        r->result += r->huffmanCode[ch];
    }
}
```

Листинг 3.8 – Формирование кодов Хаффмана

```
1  void encode(shared_ptr<Node> root, string str,
       unordered_map<char, string> &huffmanCode) {
2      if (root == nullptr)
3          return;
4
5      if (!root->left && !root->right) {
6          huffmanCode[root->ch] = str;
7      }
8
9      encode(root->left, str + "0", huffmanCode);
10     encode(root->right, str + "1", huffmanCode);
11 }
```

4 Исследовательская часть

В данном разделе произведено сравнение однопоточной и многопоточной версии алгоритма.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени:

— операционная система — Ubuntu 22.04.1 Linux x86_64;

— оперативная память — 8 ГБ;

— процессор — AMD Ryzen 5 3550H [5].

Замеры проводились на ноутбуке, включенном в сеть электропитания. Во время замеров ноутбук не был нагружен сторонними приложениями.

4.2 Время выполнения алгоритма

На рисунке 4.1 представлен график, иллюстрирующий зависимость времени работы однопоточной и многопоточной версии алгоритма от количества заявок.

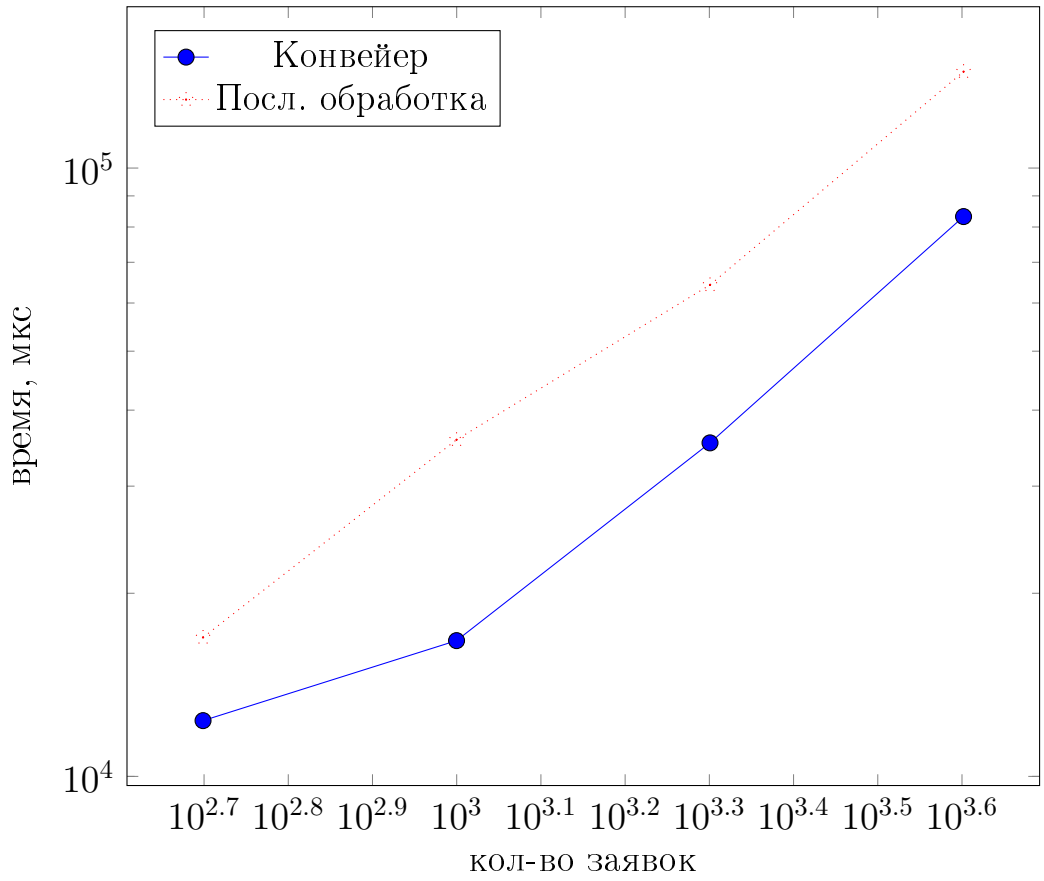

Рисунок 4.1 – Сравнение реализаций алгоритма

Вывод

При количестве заявок 1000 конвейерная обработка превосходит последовательную примерно в 2 раза.

Заключение

Цель достигнута: было реализовано асинхронное взаимодействие потоков на примере конвейерной обработки данных в алгоритме сжатия Хаффмана. В ходе выполнения лабораторной работы были решены все задачи:

1) описан алгоритм Хаффмана;

2) реализована однопоточная и многопоточная версия алгоритма;

3) произведено сравнение затрат реализаций по времени выполнения.

При количестве заявок 1000 конвейерная обработка превосходит последовательную примерно в 2 раза.

Список использованных источников

1. Виноградова М.С., Ткачева О.С. Сжатие данных. Алгоритм Хаффмана // Modern European Researches. —2022. —Vol. 1, № 3.

2. Документация по языку C++ | Microsoft Learn [Электронный ресурс]. Режим доступа: `https://learn.microsoft.com/ru-ru/cpp/cpp/?view=msvc-150` (дата обращения: 30.09.2022).

3. <ctime> | Microsoft Learn [Электронный ресурс]. Режим доступа: `https://learn.microsoft.com/en-us/cpp/standard-library/ctime?redirectedfrom=MSDN&view=msvc-160` (дата обращения: 30.09.2022).

4. std::thread [Электронный ресурс]. Режим доступа: `https://en.cppreference.com/w/cpp/thread/thread` (дата обращения: 30.09.2022).

5. AMD Ryzen™ 5 3550H [Электронный ресурс]. Режим доступа: `https://www.amd.com/en/products/apu/amd-ryzen-5-3550h` (дата обращения: 30.09.2022).