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 №4 по курсу «Анализ алгоритмов»

Тема Параллельные вычисления на основе нативных потоков

Студент Морозов Д.В.

Группа ИУ7-52Б

Оценка (баллы)

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

Москва — 2023 г.

Оглавление

Введение

Многопоточность — это форма параллельной обработки или разделения задач на части для одновременной обработки. Вместо отправки большой задачи на одно ядро, многопоточные программы разбивают её на несколько частей или потоков.

Целью данной лабораторной работы является описание и реализация последовательной и параллельной версии алгоритма DBSCAN.

Задачи данной лабораторной:

1) описание алгоритма DBSCAN;

2) реализация последовательной и параллельной версии алгоритма;

3) сравнение затрат реализаций алгоритма по времени выполнения.

1 Аналитическая часть

1.1 Алгоритм DBSCAN

Алгоритм DBSCAN (Density Based Spatial Clustering of Applications with Noise), плотностный алгоритм для кластеризации пространственных данных с присутствием шума, был предложен Мартином Эстер, Гансом-Питером Кригель и коллегами в 1996 году как решение проблемы разбиения данных на кластеры произвольной формы [1].

Идея, положенная в основу алгоритма, заключается в том, что внутри каждого кластера наблюдается типичная плотность точек (объектов), которая заметно выше, чем плотность снаружи кластера, а также плотность в областях с шумом ниже плотности любого из кластеров. Ещё точнее, что для каждой точки кластера её соседство заданного радиуса должно содержать не менее некоторого числа точек, это число точек задаётся пороговым значением. Перед изложением алгоритма дадим необходимые определения.

Определение 1. *Eps-соседство точки* p, обозначаемое как $N_{eps}(p)$, определяется как множество документов, находящихся от точки p на расстояния не более Eps. Поиска точек, чьё $N_{eps}(p)$ содержит хотя бы минимальное число точек ($MinPt$) не достаточно, так как точки бывают двух видов: ядровые и граничные.

Определение 2. Точка p *непосредственно плотно-достижима* из точки q (при заданных Eps и $MinPt$), если $p \in N_{eps}(q)$ и $|N_{eps}(q)| \geq MinPt$.

Определение 3. Точка p *плотно-достижима* из точки q (при заданных Eps и $MinPt$), если существует последовательность точек $p_1, ..., p_n$ такая, что при всех i p_{i+1} непосредственно плотно-достижима из p_i.

Определение 4. Точка p *плотно-связана* с точкой q (при заданных Eps и $MinPt$), если существует точка o такая, что p и q плотно-достижимы из неё.

Определение 5. *Кластер* — это не пустное множество плотно-связанных точек. В каждом кластере содержится хотя бы $MinPt$ объектов.

Определение 6. *Шум* — это множество точек, которые не принад-

лежат ни одному кластеру.

Алгоритм DBSCAN для заданных значений параметров Eps и MinPt исследует кластер следующим образом: сначала выбирает случайную точку, являющуюся ядровой, в качестве затравки, затем помещает в кластер саму затравку и все точки, плотно-достижимые из неё.

Алгоритм в общем виде.

DBSCAN

Вход: множество точек документов $\mathcal{D}$, ε и MinPt.
Выход: множество кластеров $\mathcal{C} = \{C_j\}$.
Шаг 1. Установить всем элементам множества $\mathcal{D}$ флаг «не посещён».
Присвоить текущему кластеру $\mathcal{C}_j$ нулевой номер, $j := 0$.
Множество шумовых документов Noise := $\oslash$.
Шаг 2. Для каждого $d_i \in \mathcal{D}$ такого, что флаг(d_i) = «не посещён», выполнить:
Шаг 3. флаг(d_i) := «посещён»;
Шаг 4. $N_i := N_\varepsilon(d_i) = \{q \in \mathcal{D} | dist(d_i, q) \leq \varepsilon\}$
Шаг 5. Если$|N_i| <$ MinPt, то
Noise := Noise + $\{d_i\}$
иначе
номер следующего кластера j := j + 1;
EXPANDCLUSTER($d_i, N_i, \mathcal{C}, \varepsilon, MinPt$);

EXPANDCLUSTER

Вход: текущая точка d_i, его ε-соседство N_i, текущий кластер C_j и ε, MinPt.
Выход: кластер C_j
Шаг 1. $C_j := C_j + \{d_i\}$;
Шаг 2. Для всех документов $d_k \in N_i$:
Шаг 3. Если флаг(d_k) = «не посещён», то
Шаг 4. флаг(d_k) := «посещён»;
Шаг 5. $N_{ik} := N_\varepsilon(d_k)$;
Шаг 6. Если $|N_{ik}| \geq MinPt$, то $N_i := N_i + N_{ik}$;
Шаг 7. Если $\nexists p : d_k \in C_p, p = \overline{1, |\mathcal{C}|}$, то $C_j := C_j = \{d_k\}$;

2 Конструкторская часть

2.1 Алгоритм DBSCAN

На рисунке 2.1 приведена схема алгоритма DBSCAN, на рисунках 2.2–2.5 приведены схемы вспомогательных подпрограмм.

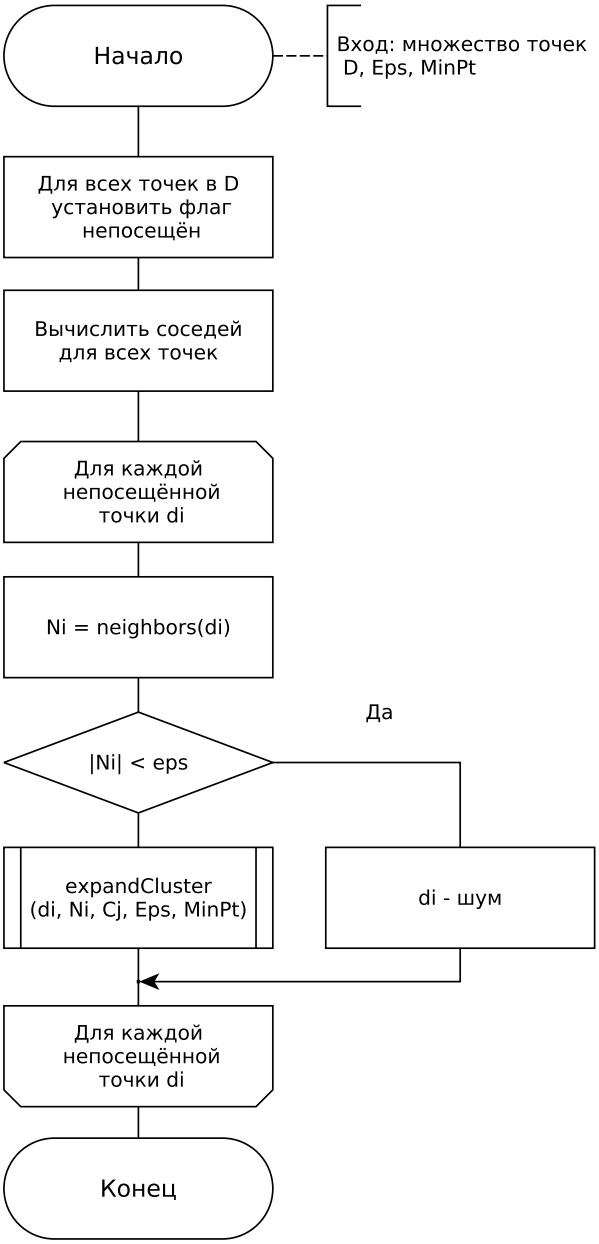

Рисунок 2.1 – Схема алгоритма DBSCAN

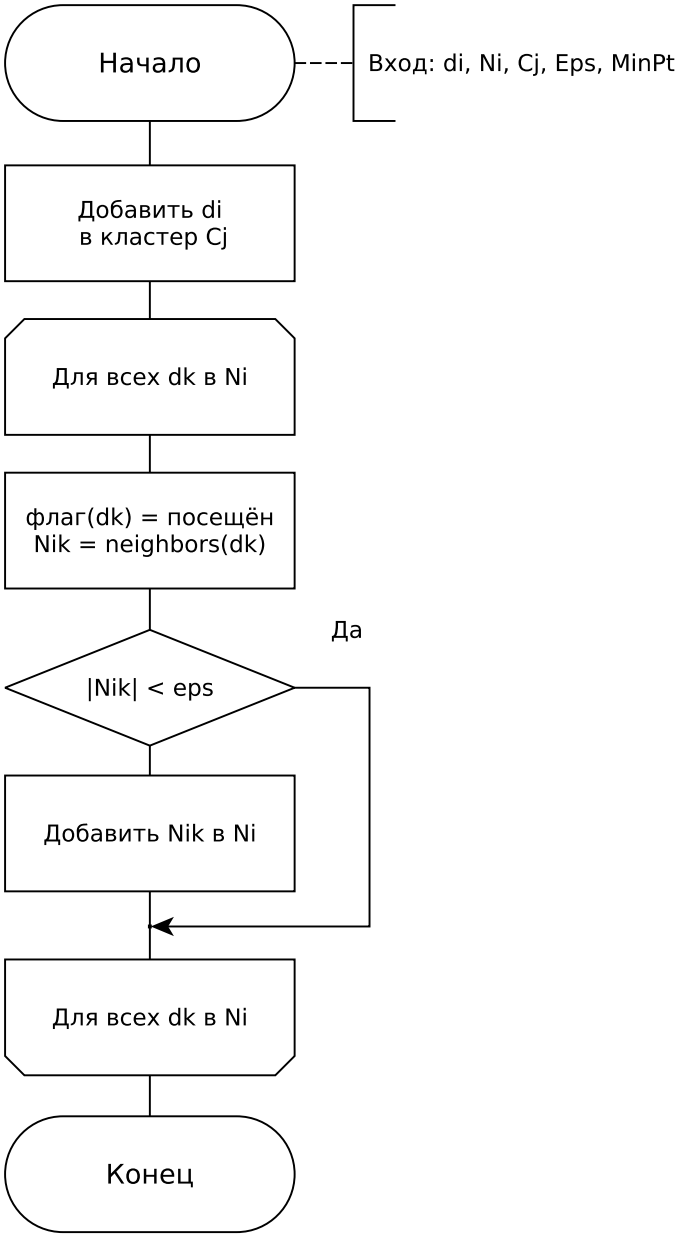

Рисунок 2.2 – Функция ExpandCluster

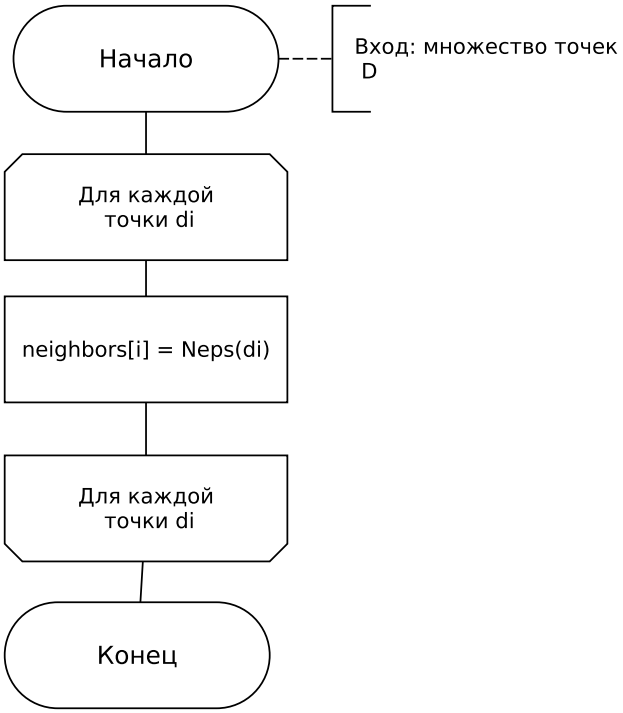

Рисунок 2.3 – Функция последовательного вычисления соседей

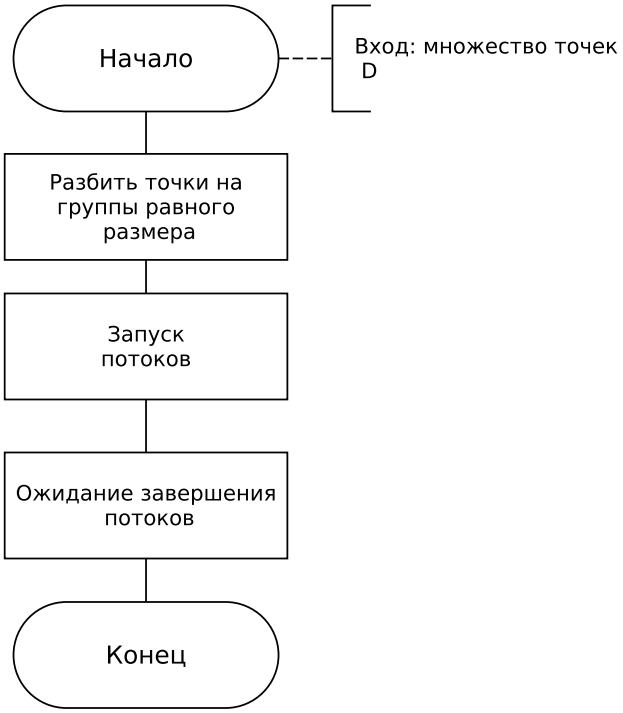

Рисунок 2.4 – Функция параллельного вычисления соседей: основной поток

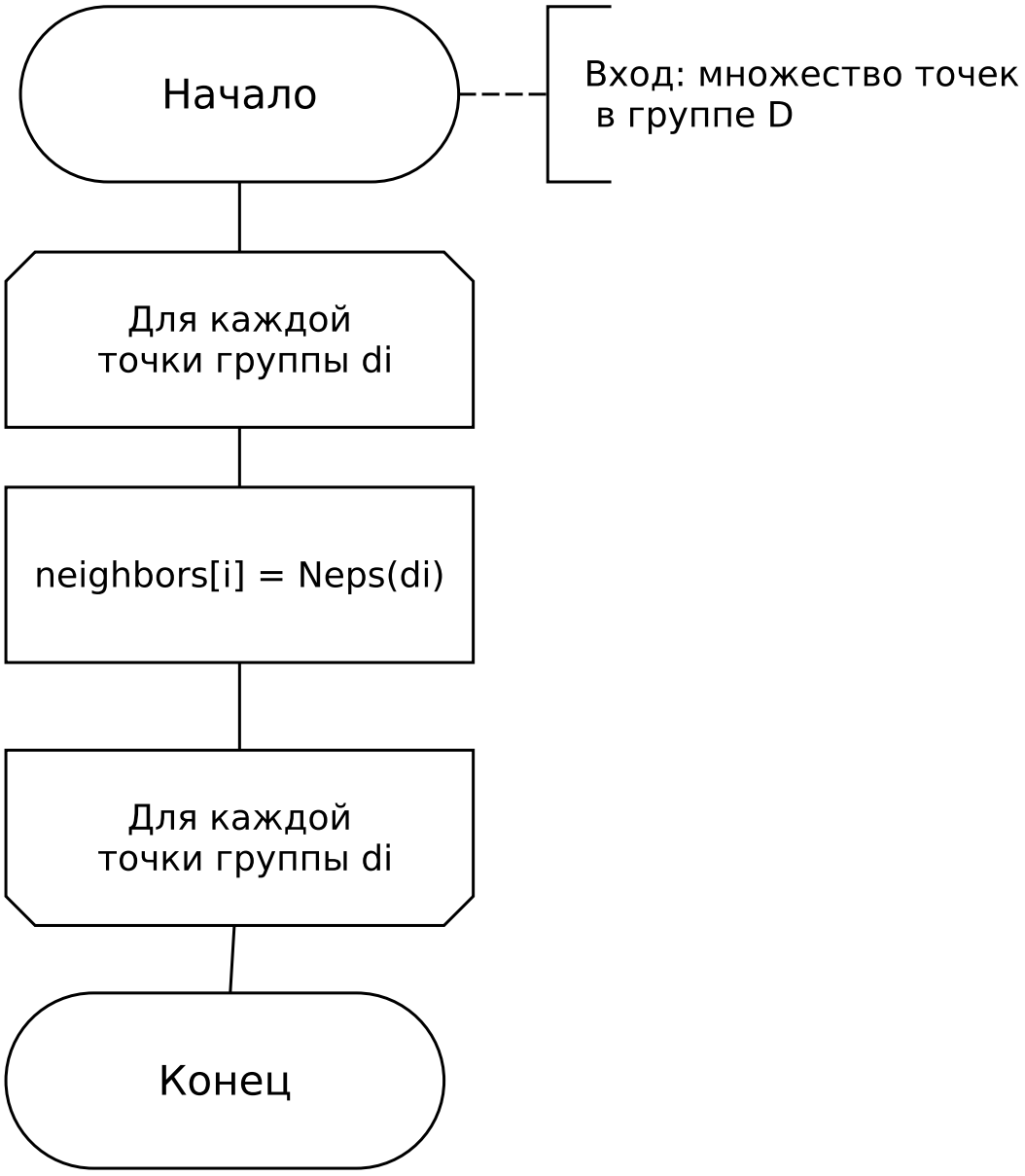

Рисунок 2.5 – Функция параллельного вычисления соседей: вспомогательные потоки

2.2 Функциональные требования

К программе предъявляются следующие требования:

— считывание из файла координат точек на плоскости;

— вывод в файл номеров кластеров точек;

— выполнение замеров времени выполнения реализаций алгоритмов.

3 Технологическая часть

3.1 Средства реализации

Используемое программное обеспечение должно предоставлять возможность выполнения замеров времени и работы с потоками.

Для выполнения 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].

3.2 Реализация алгоритмов

В листингах 3.1–3.9 приведены класс DBScan, реализация алгоритма и вспомогательных функций.

Листинг 3.1 – Классс DBScan (часть 1)

```
1  class DBScan {
2      public:
3      const vector<shared_ptr<Point>> points;
4      vector<int> clusterIndexes;
5
6      DBScan(vector<shared_ptr<Point>> points, size_t
           minPointsInCluster, double eps, int cntThreads) :
7      points(points), minPointsInCluster(minPointsInCluster),
           eps(eps), threads(cntThreads) {
8          this->cntThreads = cntThreads;
9
10         for (size_t i = 0; i < points.size(); i++) {
11             clusterIndexes.push_back(UNVISITED);
12         }
13     }
14
15     void runSerial();
16     void runParallel();
```

Листинг 3.2 – Классс DBScan (часть 2)

```
private:
    const size_t minPointsInCluster;
    const double eps;
    int cntThreads;
    int curPointIndex;
    int curClusterID;
    vector<thread> threads;
    vector<vector<int>> pointGroups;
    map<int, vector<int>> neighbors;
    mutex m;
    void initNeighbors();
    void serialCalcNeighbors();
    void parallelCalcNeighbors();
    void formPointGroups();
    int run();
    int expandCluster();
    vector<int> getNeighbors(const shared_ptr<Point> point);
    void calcGroupNeighbors(int groupNumber);
};
```

Листинг 3.3 – Функция последовательного вычисления соседей

```
    void DBScan::serialCalcNeighbors() {
        for (int i = 0; i < points.size(); i++) {
            neighbors[i] = getNeighbors(points[i]);
        }
    }
```

Листинг 3.4 – Функция параллельного вычисления соседей: основной поток

```
    void DBScan::parallelCalcNeighbors() {
        formPointGroups();
        for (int i = 0; i < cntThreads; i++) {
            this->threads[i] =
                thread(&DBScan::calcGroupNeighbors, this, i);
        }
        for (int i = 0; i < cntThreads; i++) {
            this->threads[i].join();
        }
    }
```

Листинг 3.5 – Функция параллельного вычисления соседей: вспомогательные потоки

```
1 void DBScan::calcGroupNeighbors(int groupNumber) {
2     for (size_t i = 0, n = pointGroups[groupNumber].size(); i <
        n; i++) {
3         vector<int> curNeighbors =
              getNeighbors(points[pointGroups[groupNumber][i]]);
4         m.lock();
5         neighbors[i] = move(curNeighbors);
6         m.unlock();
7     }
8 }
```

Листинг 3.6 – Основная функция алгоритма

```
1  int DBScan::run() {
2      curClusterID = 0;
3      for (size_t i = 0, n = points.size(); i < n; i++) {
4          if (clusterIndexes[i] == UNVISITED) {
5              curPointIndex = i;
6              if (expandCluster() != -1) {
7                  curClusterID++;
8              }
9          }
10     }
11
12     return 0;
13 }
```

Листинг 3.7 – Функция expandCluster (часть 1)

```
1  int DBScan::expandCluster() {
2      vector<int> seeds = getNeighbors(points.at(curPointIndex));
3      if (seeds.size() + 1 < minPointsInCluster) {
4          clusterIndexes[curPointIndex] = NOISE;
5          return -1;
6      }
7      clusterIndexes[curPointIndex] = curClusterID;
8      for (size_t i = 0, cntSeeds = seeds.size(); i < cntSeeds;
          ++i) {
9          clusterIndexes[seeds[i]] = curClusterID;
10     }
```

Листинг 3.8 – Функция expandCluster (часть 2)

```
    while (seeds.size() > 0) {
        int curSeed = seeds.back();
        seeds.pop_back();
        vector<int> seedNeighbors =
           getNeighbors(points.at(curSeed));

        size_t cntNeighbors = seedNeighbors.size();
        if (cntNeighbors + 1 >= minPointsInCluster) {
            for (size_t i = 0; i < cntNeighbors; ++i) {
                int curNeighbor = seedNeighbors[i];
                if (clusterIndexes[curNeighbor] == UNVISITED ||
                   clusterIndexes[curNeighbor] == NOISE) {
                    if (clusterIndexes[curNeighbor] == NOISE) {
                        seeds.push_back(curNeighbor);
                    }
                    clusterIndexes[curNeighbor] = curClusterID;
                }
            }
        }
    }

    return 0;
}
```

Листинг 3.9 – Функция getNeighbors

```
vector<int> DBScan::getNeighbors(const shared_ptr<Point> point) {
    vector<int> neighborIndexes;
    for (size_t i = 0, n = points.size(); i < n; i++) {
        double distance = point->dist(points[i]);
        if (distance <= eps && distance >= 1e-8) {
            neighborIndexes.push_back(i);
        }
    }

    return neighborIndexes;
}
```

3.3 Пример работы программы

На рисунках 3.1–3.3 приведены результаты работы программы при различных значениях *Eps* и *MinPt*.

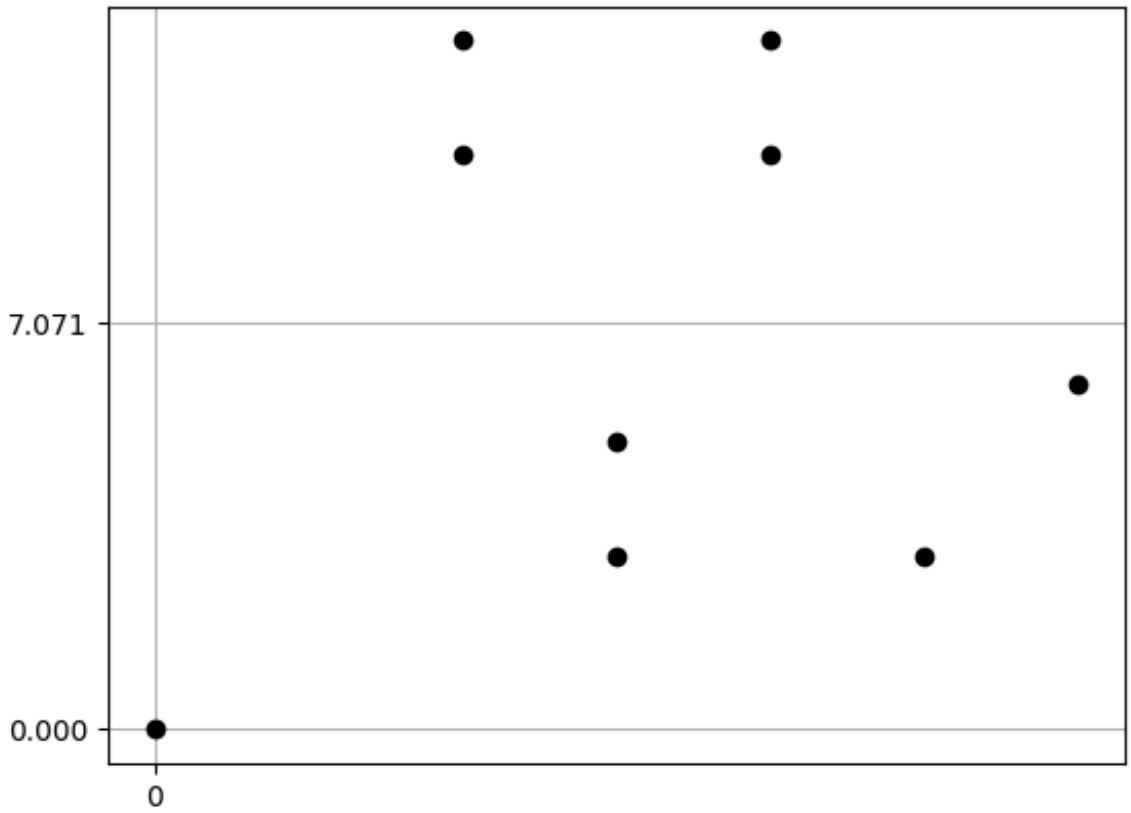

Рисунок 3.1 – Все точки являются шумом

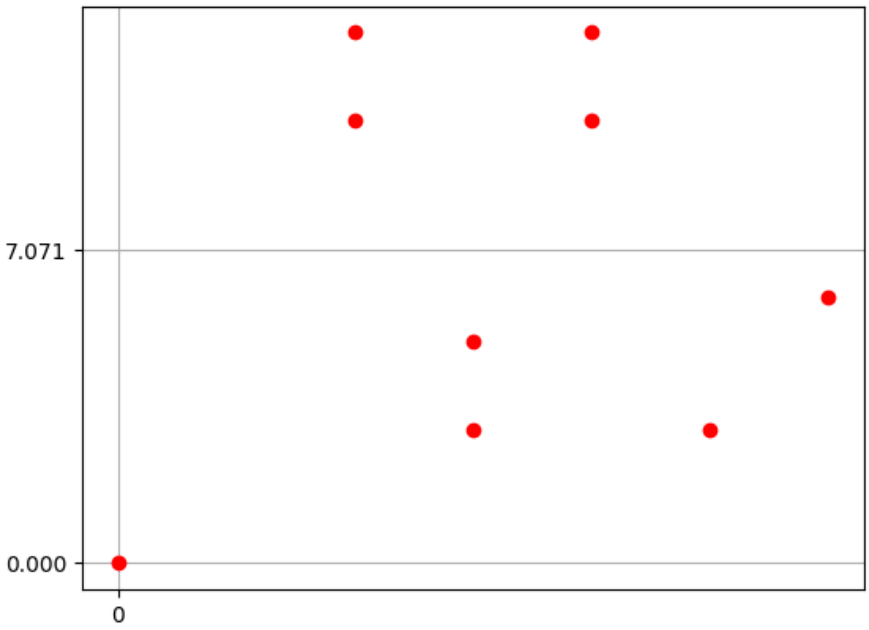

Рисунок 3.2 – 1 кластер

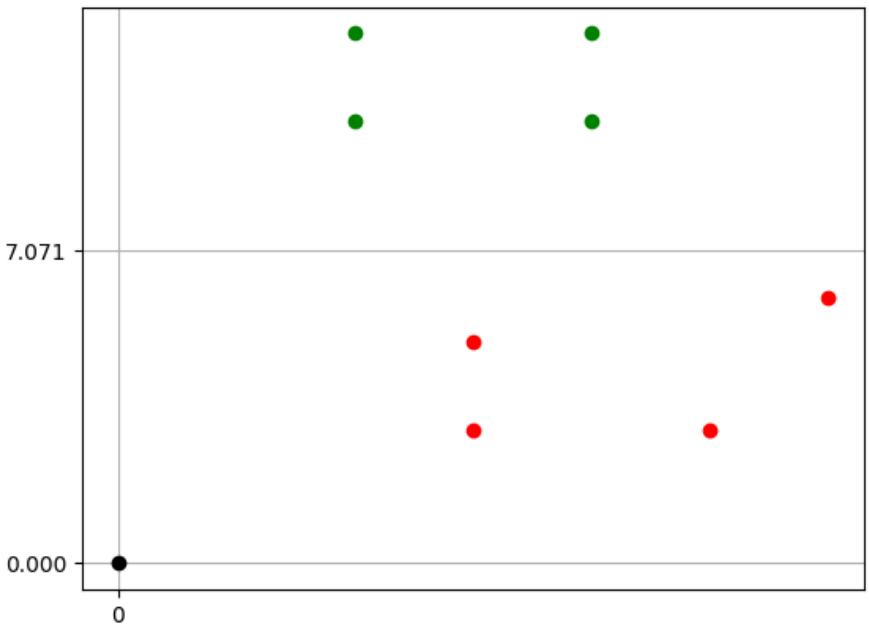

Рисунок 3.3 – 2 кластера

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени:

— операционная система Ubuntu 22.04.1 Linux x86_64;

— оперативная память 8 Гбайт;

— процессор AMD Ryzen 5 3550H, 8 физических ядер, 8 логических ядер [5].

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

4.2 Время выполнения алгоритмов

На рисунке 4.1 представлен график, иллюстрирующий зависимость времени работы последовательной и параллельной версии алгоритма от количества потоков.

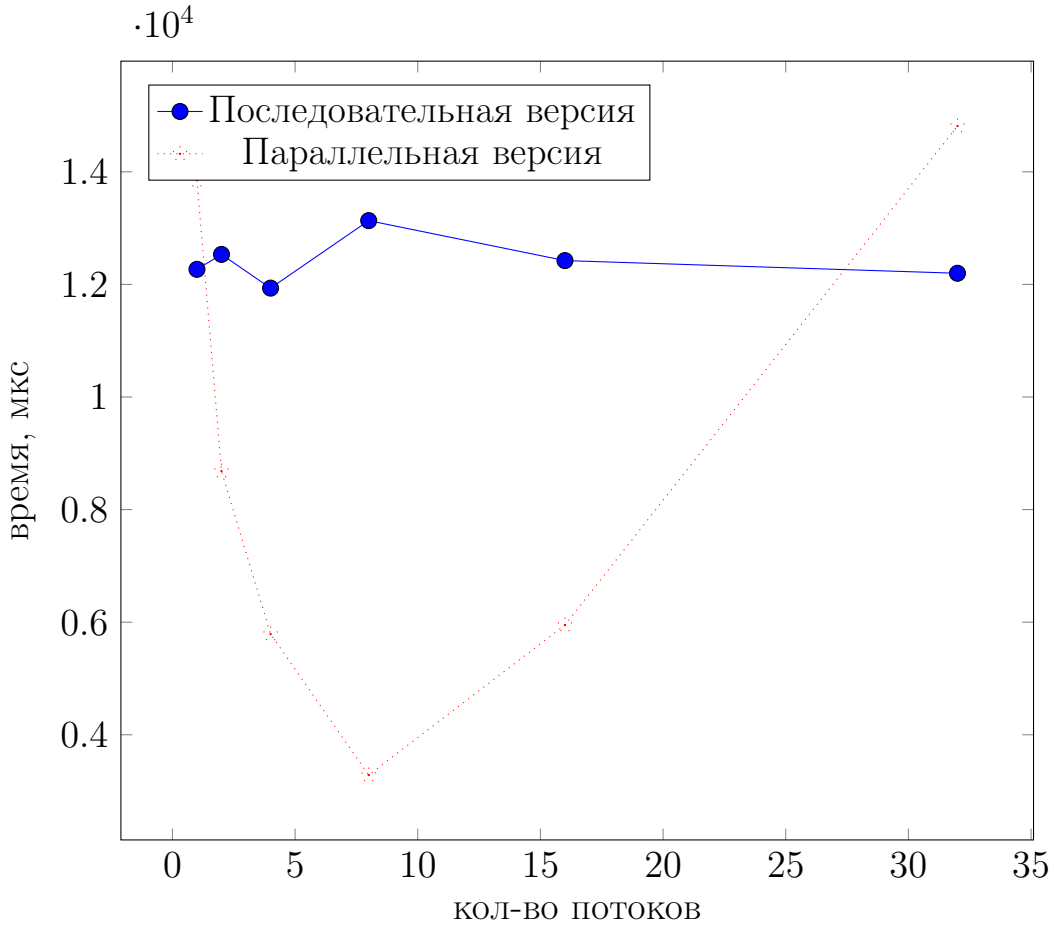

Рисунок 4.1 – Сравнение реализаций алгоритма по времени работы

Вывод

Как видно из полученных данных, параллельная реализация работает быстрее последовательной при количестве рабочих потоков от 2 до 16. Начиная с 8 рабочих потоков, время работы параллельной реализации растёт, при 32 рабочих потоках оно превышает время работы последовательной. Это связано с тем, что процессор не может эффективно обслуживать более 8 потоков.

Заключение

Цель достигнута: были описаны и реализованы последовательная и параллельная версия алгоритма DBSCAN. В ходе выполнения лабораторной работы были решены все задачи:

1) описан алгоритм DBSCAN;

2) реализованы последовательная и параллельная версии алгоритма;

3) проведено сравнение затрат реализаций алгоритма по времени выполнения.

Параллельная реализация алгоритма DBSCAN работает быстрее последовательной при количестве рабочих потоков от 2 до 16. Оптимальное количество потоков для параллельной реализации — 8.

Список использованных источников

1. Большакова Е.И., Клышинский Э.С., Ландэ Д.В., Носков А.А., Пескова О.В., Ягунова Е.В. Автоматическая обработка текстов на естественном языке и компьютерная лингвистика // М.: МИЭМ, —2011. — 272 с.

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).