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 Многопоточное программирование

Студент Романов С.К.

Группа ИУ7-55Б

Оценка (баллы) _____

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

Москва — 2023 г.

Оглавление

ВВЕДЕНИЕ

Многопоточность – способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой.

Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразование.

Для распараллеливания может быть рассмотрена задача кластеризации пространственных данных.

Алгоритм DBSCAN (Density Based Spatial Clustering of Applications with Noise), был предложен Мартином Эстер, Гансом-Питером Кригель и коллегами в 1996 году как решение проблемы разбиения (изначально пространственных) данных на кластеры произвольной формы. Большинство алгоритмов, производящих плоское разбиение, создают кластеры по форме близкие к сферическим, так как минимизируют расстояние документов до центра кластера. Авторы DBSCAN экспериментально показали, что их алгоритм способен распознать кластеры различной формы.

Цель лабораторной работы:

– Изучение и реализация параллельных вычислений

Задачи лабораторной работы:

1. Проанализировать последовательный и параллельный варианты алгоритма DBSCAN,
2. Определить средства программной реализации алгоритмом,
3. Реализовать разработанные алгоритмы,
4. Провести анализ затрат работы программы по времени, выяснить влияющие на них характеристики,
5. Создать отчёт, содержащий:
   – актуальность исследования;
   – характеристики предложенной реализации (по времени);
   – результаты тестирования;
   – Выводы.

В ходе работы будут затронуты следующие темы:

1. Параллелизация вычислений.
2. Алгоритм DBSCAN.
3. Многопоточность.

1 Аналитическая часть

1.1 Алгоритм DBSCAN

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 $N_\varepsilon(p)$, определяется как множество документов, находящихся от точки p на расстояния не более ε: $N_\varepsilon(p) = \{q \in D | dist(p,q) \leq \varepsilon\}$. Поиска точек, чьё $N_\varepsilon(p)$ 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 ε и MinPt), если $p \in N_\varepsilon(p)$ и $|N_\varepsilon(p)| \geq MinPt$ (рис. 1.1)

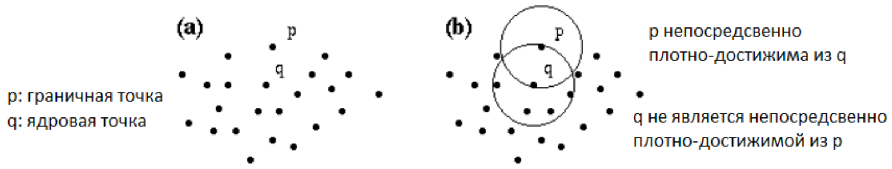

Рисунок 1.1 – Пример точек, находящихся в отношении непосредственно плотной достижимости

Определение 3. Точка p плотно-достижима из точки q (при заданных ε и MinPt), если существует последовательность точек $q = p_1, p_2, ..., p_n = p : p_{i+1}$ непосредственно плотно-достижимы из p_i. Это отношение транзитивно, но не симметрично в общем случае, однако симметрично для двух ядровых точек.

Определение 4. Точка p плотно-связана с точкой q (при заданных ε и MinPt), если существует точка $o : p$ и q плотно-достижимы из o (при заданных ε и MinPt), (рис. 1.2)

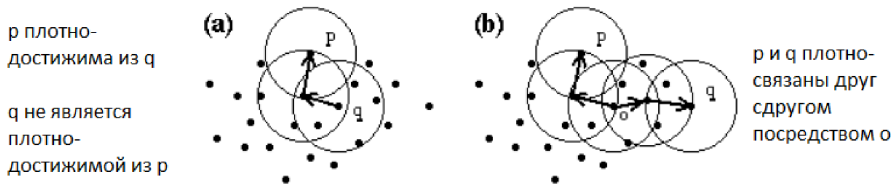

Рисунок 1.2 – Пример точек, находящихся в отношении плотной связанности.

Теперь мы готовы дать определения кластеру и шуму.

Определение 5. Кластер $\mathcal{C}_j$ (при заданных ε и MinPt) – это не пустое подмножество документов, удовлетворяющее следующих условиям:

1. $\forall p, q$: если $p \in \mathcal{C}_j$ и q плотно-достижима из p(при заданных ε и MinPt), то $q \in \mathcal{C}_j$,
2. $\forall p, q \in \mathcal{C}_j$: p плотно связана с q (при заданных ε и MinPt);

Итак, кластер – это множество плотно-связанных точек. В каждом кластере содержится хотя бы MinPt документов.

Шум – это подмножество документов, которые не принадлежат ни одному кластеру: $\{p \in D | \forall j : p \notin \mathcal{C}_j, j = \overline{1, |\mathcal{C}|}\}$

Алгоритм DBSCAN для заданных значений параметров ε 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 := $\varnothing$.
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. $|N_{ik}| \geq MinPt$, то $N_i := N_i + N_{ik}$;
Шаг 7. Если $\nexists p : d_k \in C_p, p = \overline{1, |\mathcal{C}|}$, то $C_j := C_j = \{d_k\}$;

1.2 Параллельный алгоритм DBSCAN

Параллельный алгоритм DBSCAN заключается в том, что точки помещаются в непересекающиеся d-мерные клетки с длиной стороны $\varepsilon/\sqrt{d}$ по их координатам (рис. 1.3(b)). Ячейки обладают тем свойством, что все точки внутри клетки находятся на расстоянии ε друг от друга и будут принадлежать к одному и тому же кластеру. Затем (рис. 1.3(c)) мы отмечаем основные точки, после чего мы генерируем кластеры для основных точек следующим образом:

Мы создаем граф содержащий по одной вершине на ядровую ячейку (ячейка, содержащая не менее одной ядровой точки), и соединить две вершины, если ближайшая пара ядровых точек из двух ячеек находится на расстоянии ε. Далее будем ссылаться на него как как на клеточный граф. Сам этот шаг проиллюстрирован на рисунке 1.3(d). Затем находим компоненты связности графа ячеек для назначения идентификаторов кластеров точкам в ядровых ячейках и назначаем идентификаторы кластеров для граничных точек. В конце возвращаем список кластеров.

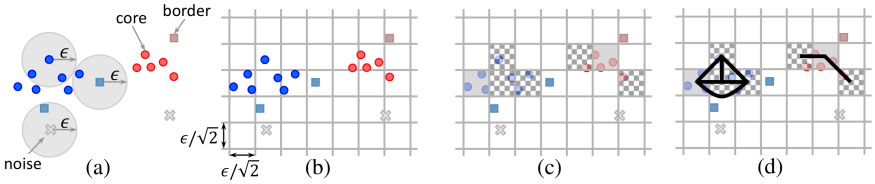

Рисунок 1.3 – Параллельный алгоритм DBSCAN

Алгоритм в общем виде.

DBSCAN

Вход: множество индексированных точек $\mathcal{P}$, ε и MinPt.
Выход: множество кластеров $\mathcal{C} = \{C_j\}$.
Шаг 1. Разбить множество $\mathcal{P}$ точек на множество ячеек $\mathcal{G}$
Шаг 2. coreFlags := MarkCore($\mathcal{G}, \mathcal{P}, \varepsilon, MinPt$)
Шаг 3. $\mathcal{C}$:= ClusterCore($\mathcal{G}, \mathcal{P}, coreFlags, \varepsilon, MinPt$)
Шаг 4. ClusterBorder($\mathcal{G}, \mathcal{P}, coreFlags, \mathcal{C}, \varepsilon, MinPt$)
Шаг 5. Вернуть $\mathcal{C}$

MarkCore

Вход: множество индексированных точек $\mathcal{P}$, множество ячеек $\mathcal{G}$ ε и MinPt.
Выход: список ядровых точек $coreFlags$.

```
1. coreFlags := {0, ..., 0}                    \\ Длина равна размеру P
2. Параллельно ∀g ∈ G
3.       Если len(g) ≥ minPt
4.             Для ∀p ∈ g
5.                   coreFlags[p] := 1
6.             иначе
7.                   для ∀p ∈ g
8.                         count := len(g)
9.                         для ∀h ∈ g.NeighborCells(ε)
10.                              count := count + RangeCount(p, ε, h)
11.                        Если count ≥ minPt
12.                              coreFlags[p] := 1
13. Вернуть coreFlags
```

ClusterCore

Вход: множество индексированных точек $\mathcal{P}$, множество ячеек $\mathcal{G}$,
флаги ядровых ячеек $coreFlags$, ε и MinPt.
Выход: Кластеры $\mathcal{C}$.
1. uf := $UnionFind()$ \\Инициализация Union-find структуры
2. Параллельно $\forall g \in \mathcal{G}$
3. Для $\forall h \in g.NeighborCells()$
4. Если $g > h$ && $uf.Find(g) \neq uf.Find(h)$
5. Если $Connected(g, h)$
6. $uf.Link(g, h)$
7. для $\forall p \in g$
8. $\mathcal{C} := \{-1, ..., -1\}$ Длина множества $\mathcal{P}$
9. Параллельно $\forall g \in \mathcal{G}$
10. $\forall p \in g : coreFlags[p] = 1$
11. $\mathcal{C}[p] := uf.Find(g)$
12. Вернуть $\mathcal{C}$

ClusterBorder

Вход: множество индексированных точек $\mathcal{P}$, множество ячеек $\mathcal{G}$,
кластеры $\mathcal{C}$, флаги ядровых ячеек $coreFlags$, ε и MinPt.
Выход: Обновленные кластеры $\mathcal{C}$
1. Параллельно $\forall g \in \mathcal{G} : len(g) < minPt$
2. Для $\forall p \in cell\, g : coreFlags[p] = 0$
3. Для $\forall h \in g \cup g.NeighborCells(\varepsilon)$
4. Для $\forall q \in cell\, h : coreFlags[q] = 1$
5. Если $dist(p, q) \leq \varepsilon$
6. $\mathcal{C}[p] := \mathcal{C}[p] \cup \mathcal{C}[q]$
7. Вернуть $\mathcal{C}$

Вывод

В данном разделе были рассмотрены алгоритмы DBSCAN и его распараллеленного аналога.

2 Конструкторская часть

2.1 Разработка алгоритмов

На рисунках 2.1, 2.2 представлена схема линейного алгоритма DBSCAN.

На рисунках 2.3, 2.4, 2.5, 2.6, 2.7, 2.8 представлена схема параллельного алгоритма DBSCAN.

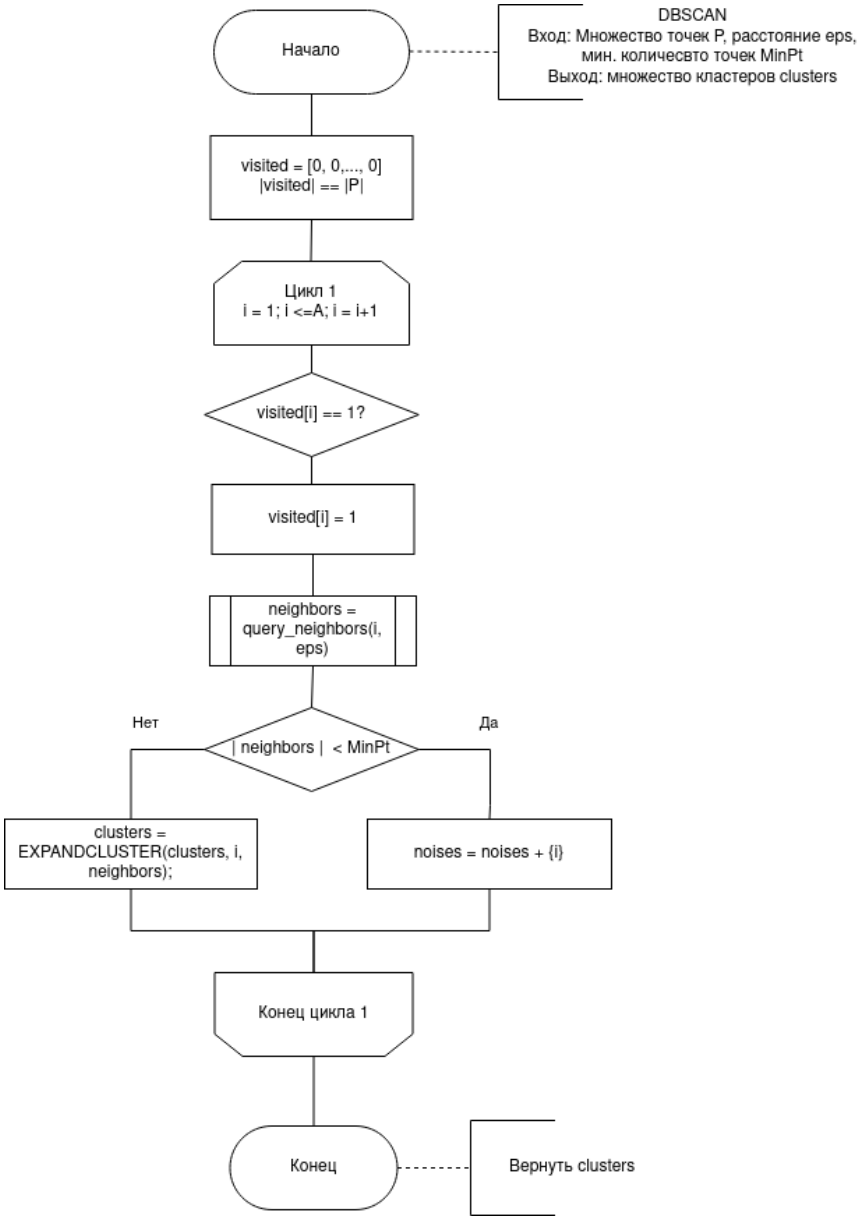

Рисунок 2.1 – Линейный алгоритм DBSCAN

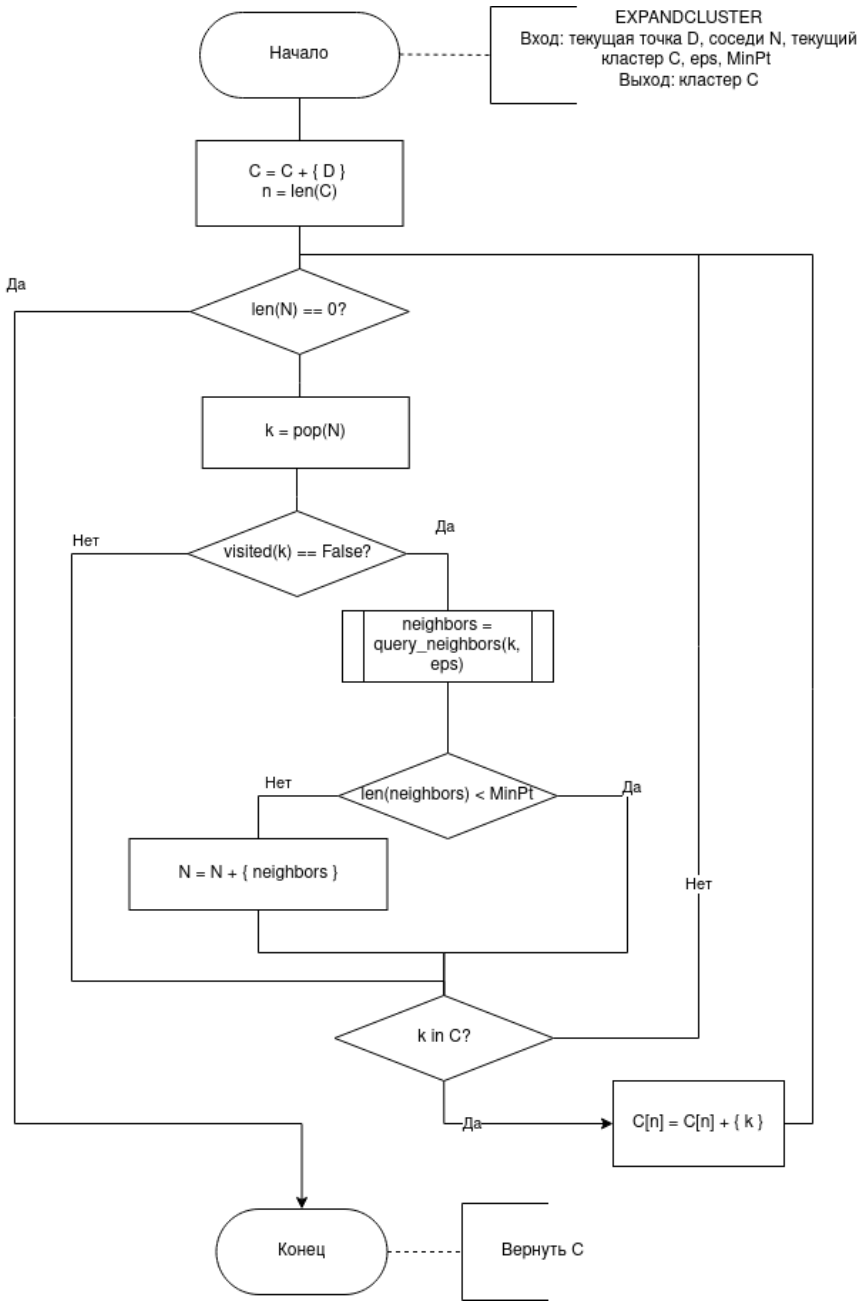

Рисунок 2.2 – Функция EXPANDCLUSTERS линейного алгоритма DBSCAN

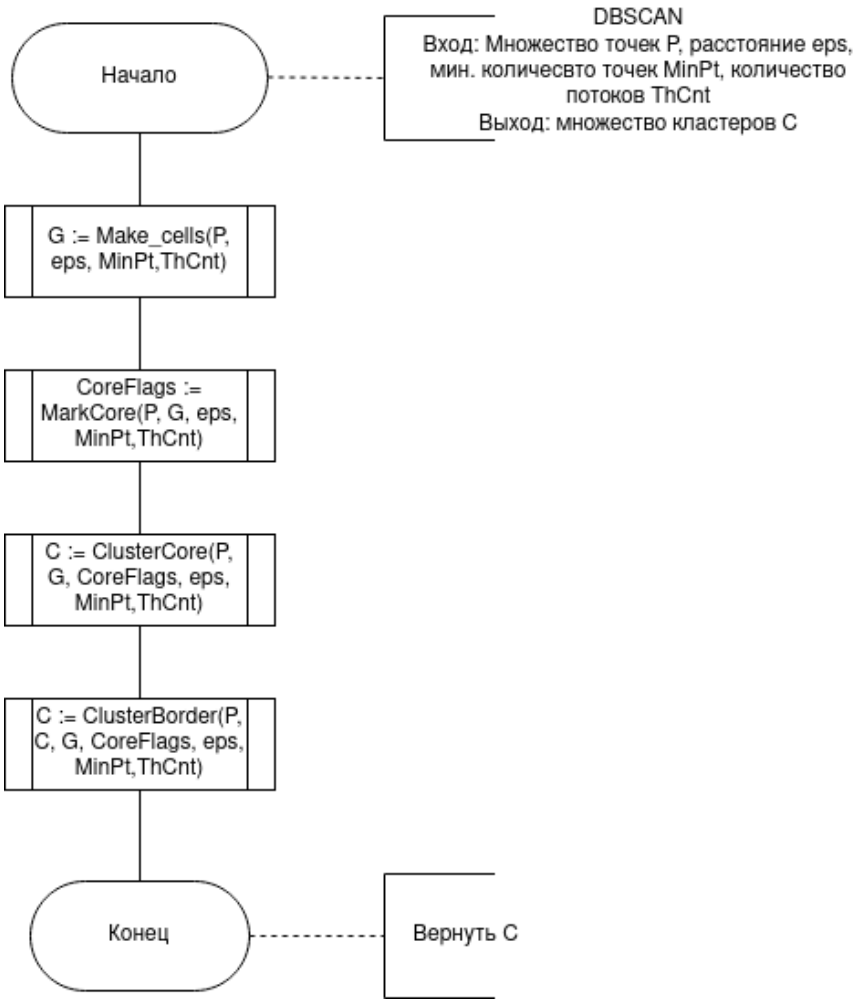

Рисунок 2.3 – Параллельный алгоритм DBSCAN

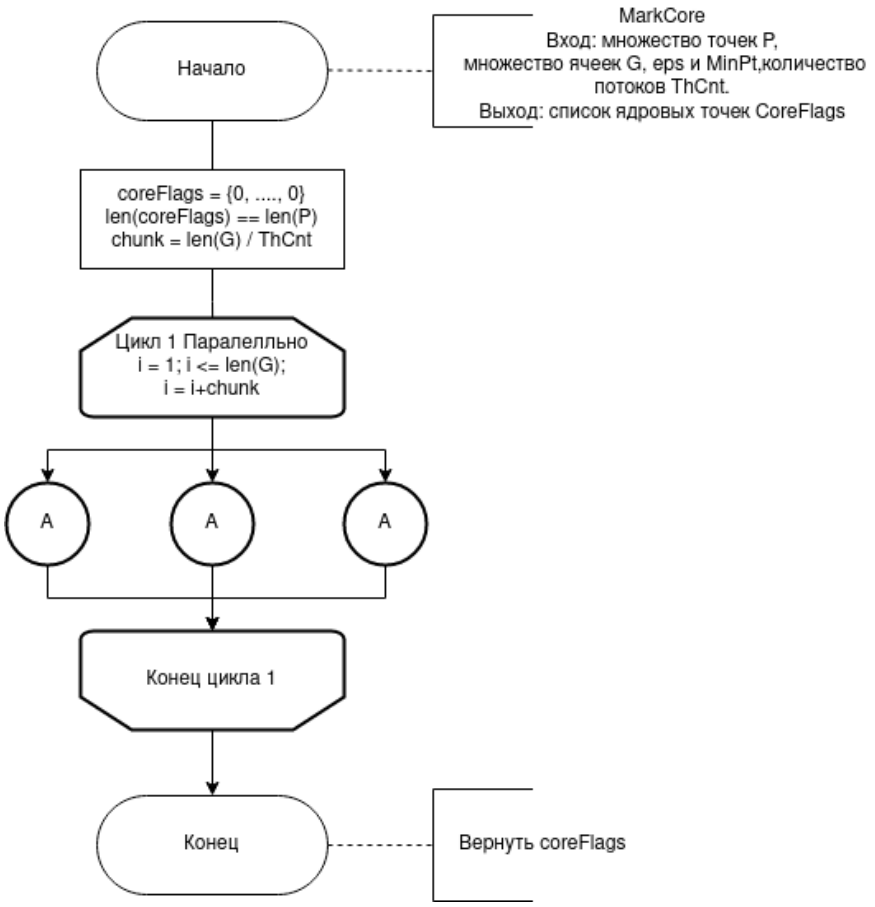

Рисунок 2.4 – Функция markCore параллельного алгоритма DBSCAN, часть 1

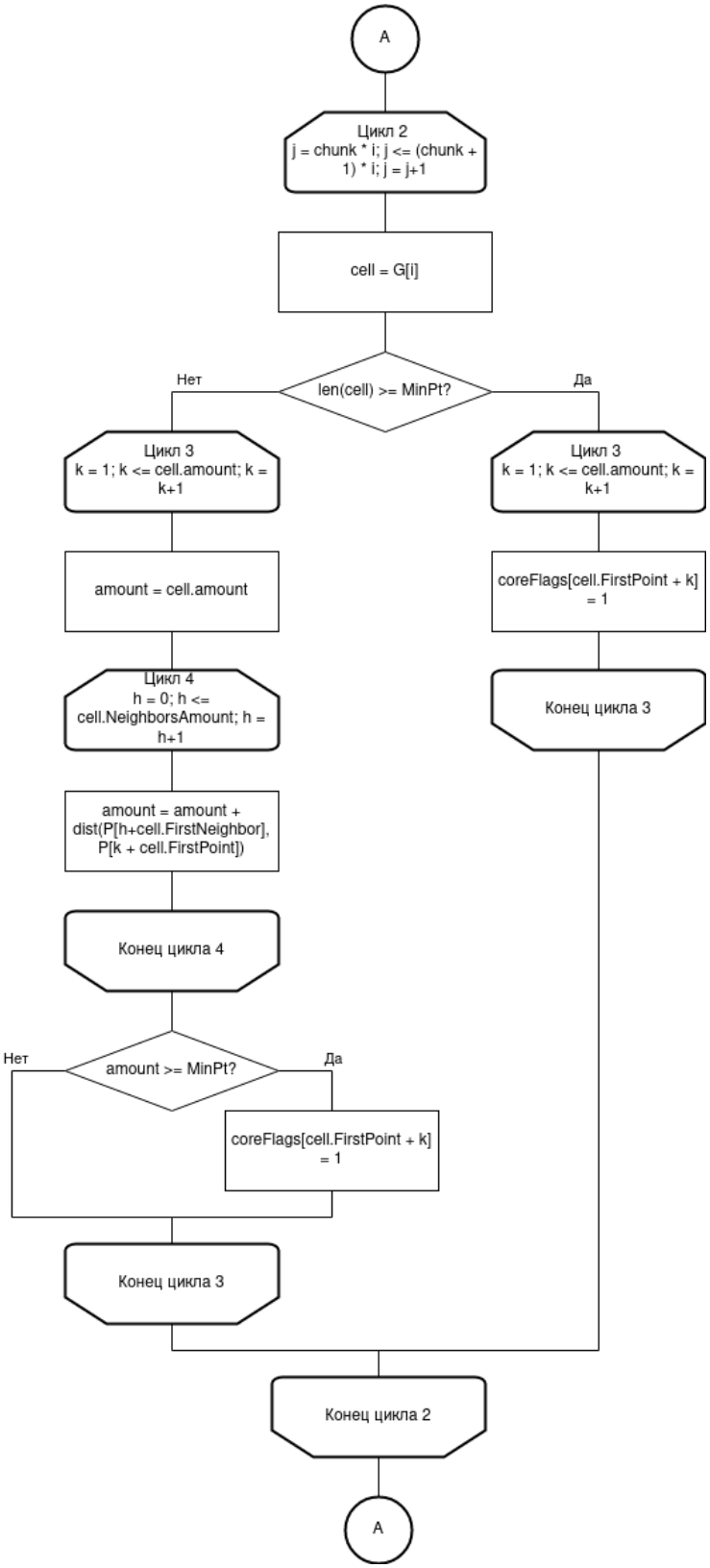

Рисунок 2.5 – Функция markCore параллельного алгоритма DBSCAN, часть 2

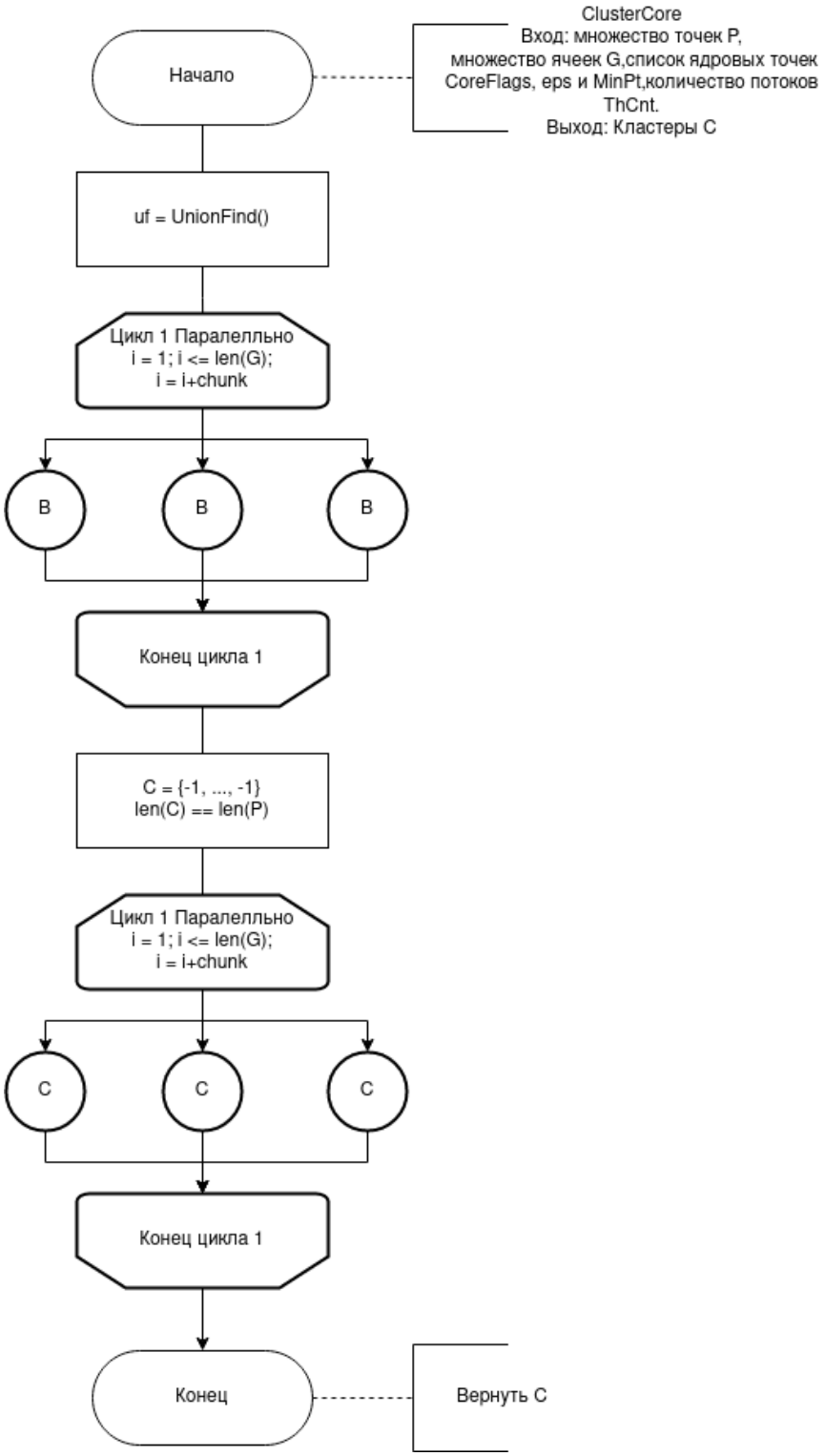

Рисунок 2.6 – Функция clusterCore параллельного алгоритма DBSCAN, часть 1

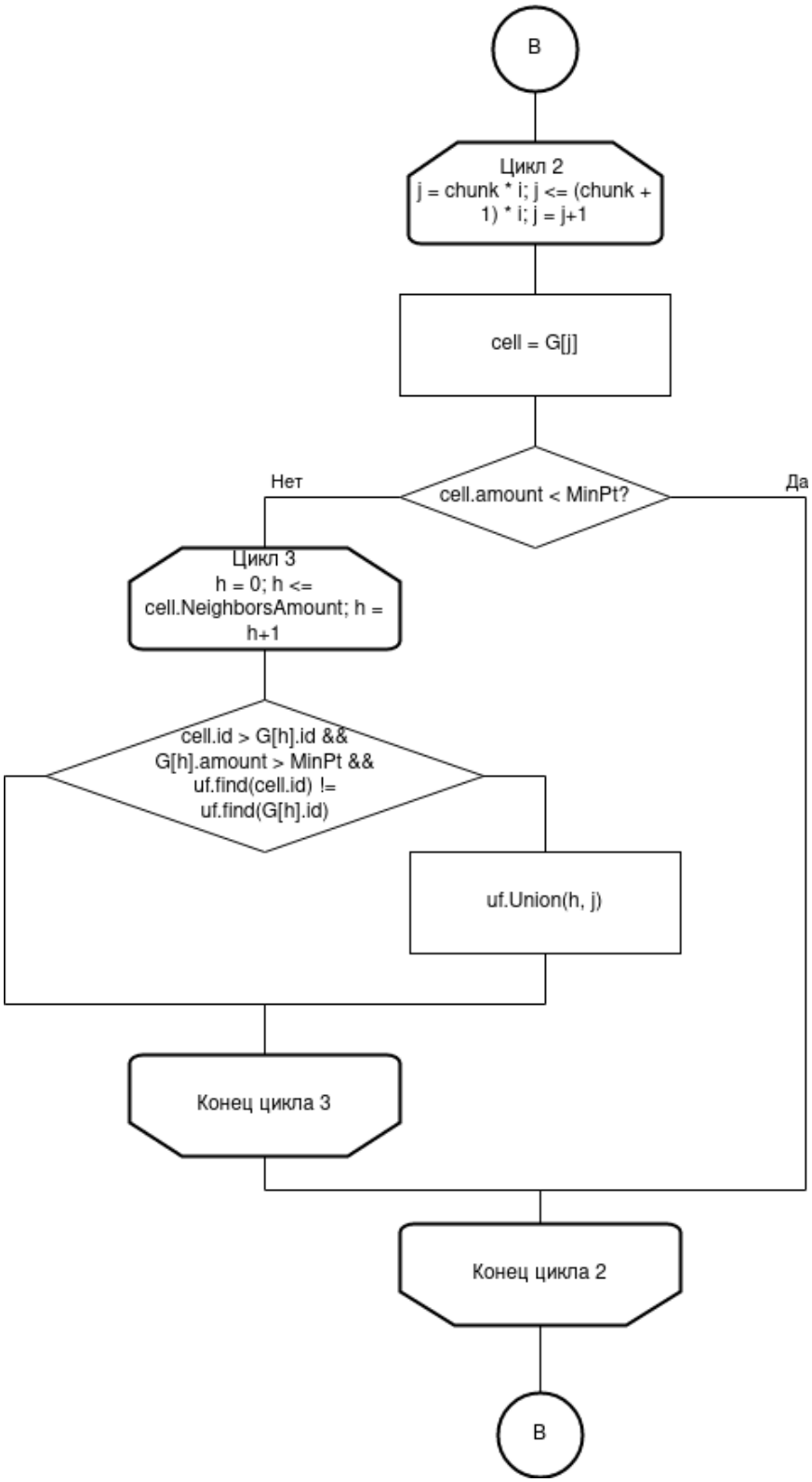

Рисунок 2.7 – Функция clusterCore параллельного алгоритма DBSCAN, часть 2

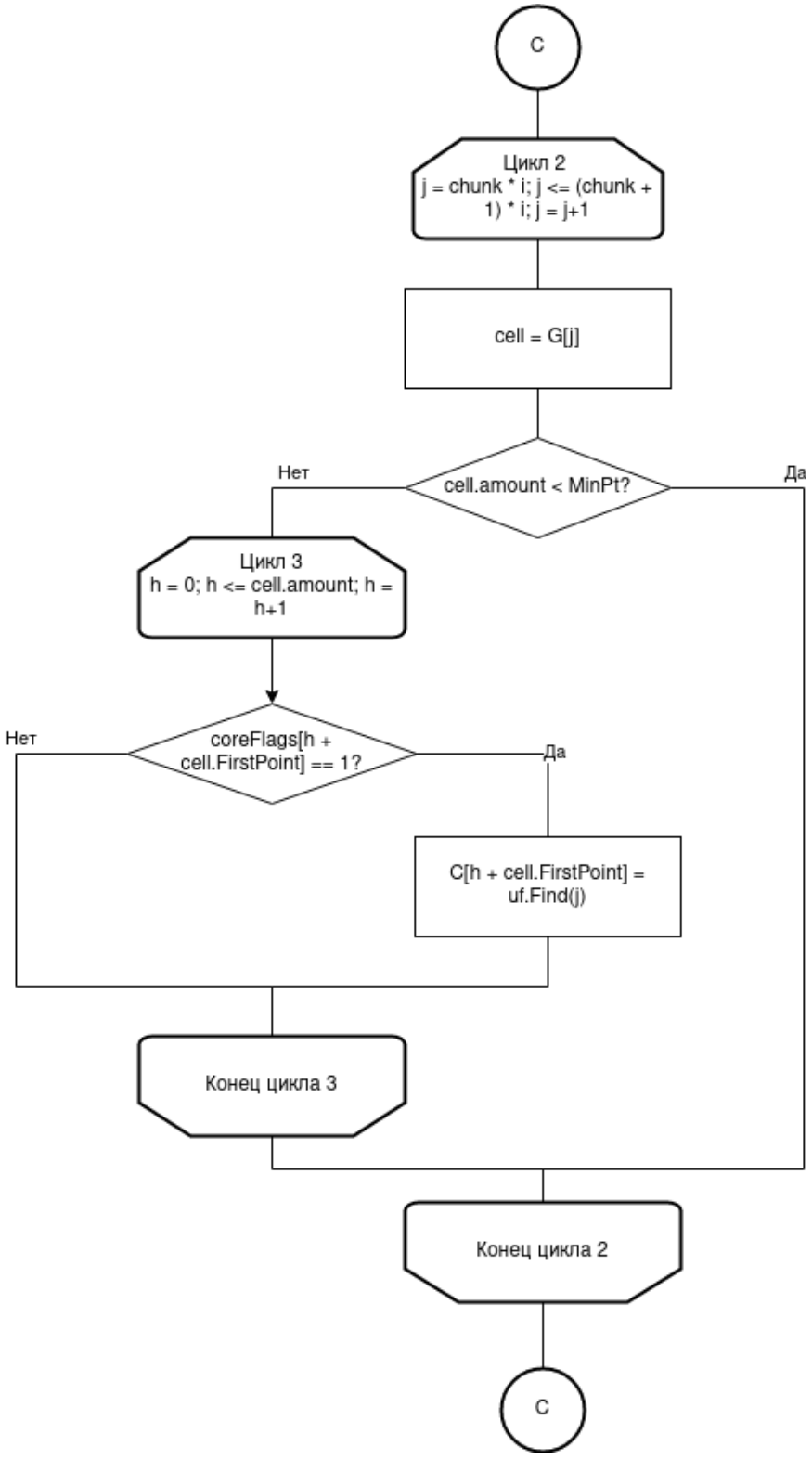

Рисунок 2.8 – Функция clusterCore параллельного алгоритма DBSCAN, часть 3

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы алгоритмов DBSCAN, последовательного и параллельного.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и сами реализации алгоритмов.

3.1 Требования к программному обеспечению

К программе предъявляется ряд условий:

– На вход подается путь к файлу, содержащий некие двумерные данные, а также числа eps и MinPt, определяющие определяющие характеристики алгоритма;

– На выход ПО должно выводить результат алгоритма DBSCAN в графическом виде;

– ПО должно замерять время работы алгоритмов;

3.2 Средства реализации

Для реализации данной лабораторной работы необходимо установить следующее программное обеспечение:

- Rust Programming Language v1.64.0 - язык программирования
- Criterion.rs v0.4.0 - Средство визуализации данных
- LaTeX - система документооборота

3.3 Реализация алгоритмов

В следующих листингах представлены следующие алгоритмы:

1. В листингах 3.1 и 3.2 представлен последовательный алгоритм DBSCAN.
2. В листингах 3.3, 3.4, 3.5, 3.6 и 3.7 представлен параллельный алгоритм DBSCAN.

3.3.1 Последовательный алгоритм DBSCAN

Листинг 3.1 – Последовательный алгоритм DBSCAN

```
pub fn run(&mut self)
{
    let len = self.dataset.lock().unwrap().labels_amount();
    for col_a in 0..len {
        for col_b in col_a + 1..len {
            self.focus = Focus(col_a, col_b);
            self.visited = vec![false; self.dataset.lock().unwrap().records_amount()];
            self.clusters.push((vec![], col_a, col_b));
            for i in 0..self.visited.len() {
                if self.visited[i] {
                    continue;
                }
                self.visited[i] = true;
                let mut neighbors = self.dataset.lock().unwrap().query_neighbors(
                    i,
                    self.eps,
                    self.focus.clone().into(),
                );

                if neighbors.len() < self.min_pts {
                    self.noises.push(i);
                } else {
                    self.expand_cluster(i, &mut neighbors);
                }
            }
        }
    }
}
```

Листинг 3.2 – Последовательный алгоритм DBSCAN, расширение кластеров

```
fn expand_cluster(&mut self, index: usize, neighbors: &mut Vec<usize>)
{
    let n = self.clusters.len() - 1;
    self.clusters[n].0.push(vec![index]);
    let len = self.clusters[n].0.len() - 1;
    while !neighbors.is_empty() {
        let k = neighbors.pop().unwrap();
        if !self.visited[k] {
            self.visited[k] = true;
            let new_neighbors = self.dataset.lock().unwrap().query_neighbors(
                k,
                self.eps,
                self.focus.clone().into(),
            );
            if new_neighbors.len() >= self.min_pts {
                neighbors.extend(new_neighbors);
            }
        }
        if self.not_in_clusters(k) {
            self.clusters[n].0[len].push(k);
        }
    }
}

fn not_in_clusters(&self, index: usize) -> bool
{
    let mut res = true;
    let n = self.clusters.len() - 1;
    for cluster in self.clusters[n].0.iter() {
        if cluster.contains(&index) {
            res = false;
            break;
        }
    }

    res
}
```

3.3.2 Параллельный алгоритм DBSCAN

Листинг 3.3 – Параллельный алгоритм DBSCAN

```
pub fn run(&mut self)
{
    let len = self.dataset.lock().unwrap().labels_amount();
    for col_a in 0..len {
        for col_b in col_a + 1..len {
            self.focus = Focus(col_a, col_b);
            self.make_cells();
            #[cfg(debug_assertions)]
            {
                draw_cells(self);
            }
            self.mark_core();
            self.cluster_core();
            self.cluster_border();
        }
    }
}
```

Листинг 3.4 – Параллельный алгоритм DBSCAN, получение ячеек

```
fn make_cells(&mut self)
{
    let binding = self.dataset.lock().unwrap();

    let vx = &binding.get_data()[self.focus.0];
    let vy = &binding.get_data()[self.focus.1];
    self.pairs = vx
        .iter()
        .cloned()
        .zip(vy.iter().cloned())
        .collect::<Vec<_>>();

    parallel_sort(&mut self.pairs, &pairs_compare_by_x, self.threads_cnt);

    let minx = self.pairs[0].0;
    let mut cur_strip = 0;
    self.strips.push(Strip {
        id: 0,
        amount: 0,
        point_id: 0,
        cells_amount: 0,
        cell_id: 0,
    });
    let eps_len = self.eps / 2f64.sqrt();
    for i in 0..self.pairs.len() {
        if self.pairs[i].0 > (cur_strip + 1) as f64 * (eps_len) + minx {
            parallel_sort(
                &mut self.pairs[self.strips[cur_strip].point_id..i],
                &pairs_compare_by_y,
                self.threads_cnt,
            );
            cur_strip += 1;
            self.strips.push(Strip {
                id: cur_strip,
                amount: 0,
                point_id: i,
                cells_amount: 0,
                cell_id: 0,
            });
        }
        self.strips[cur_strip].amount += 1;
    }

    let mut cell_id = 0;
```

```
        for strip in &mut self.strips {
            self.cells.push(Cell {
                id: cell_id,
                amount: 0,
                point_id: strip.point_id,
                strip_id: strip.id,
                neighbors: vec![],
            });
            strip.cell_id = cell_id;
            strip.cells_amount = 1;
            for i in strip.point_id..(strip.point_id + strip.amount) {
                if self.pairs[i].1 > self.pairs[self.cells[cell_id].point_id].1 + eps_len {
                    cell_id += 1;
                    self.cells.push(Cell {
                        id: cell_id,
                        amount: 0,
                        point_id: i,
                        strip_id: strip.id,
                        neighbors: vec![],
                    });
                    strip.cells_amount += 1;
                }
                self.cells[cell_id].amount += 1;
            }
            cell_id += 1;
        }
        drop(binding);
        self.make_cell_neighbors();
    }
```

Листинг 3.5 – Параллельный алгоритм DBSCAN, получение coreFlags

```
fn mark_core(&mut self) {
    let mut visited = vec![false; self.pairs.len()];
    let n = self.cells.len();
    let chunk_size = n / self.threads_cnt;
    let self_arc = Arc::new(&self);
    let _ = crossbeam::scope(|scope| {
        let mut visited = visited.iter_mut();
        for slice in (0..n).step_by(chunk_size) {
            let self_arc = self_arc.clone();
            let mut visited = visited.by_ref()
                .take(
                    (-(self.cells[slice].point_id as i64)
                        + (self.cells[std::cmp::min(slice + chunk_size, n) - 1].point_id
                            as i64)
                        + (self.cells[std::cmp::min(slice + chunk_size, n) - 1].amount as i64))
                        as usize,
                )
                .collect::<Vec<_>>();
            let k = self.cells[slice].point_id;
            scope.spawn(move |_| {
                for i in slice..std::cmp::min(slice + chunk_size, n) {
                    let cell = &self_arc.cells[i];
                    if cell.amount >= self_arc.min_pts {
                        for i in cell.point_id..(cell.point_id + cell.amount - 1) {
                            *visited[i - k] = true;
                        }
                    } else {
                        for i in cell.point_id..(cell.point_id + cell.amount - 1) {
                            let mut amount = cell.amount;
                            for neighbor in &cell.neighbors {
                                amount += self_arc.range_count(self_arc.pairs[i], *neighbor);
                            }
                            if amount >= self_arc.min_pts {
                                *visited[i - k] = true;
                            }
                        }
                    }
                }
            });
        }
    });
    self.visited = visited.clone();
}
```

Листинг 3.6 – Параллельный алгоритм DBSCAN, получение кластеров

```
fn cluster_core(&mut self)
{
    let mut union_find = UnionFind::new(self.cells.len());
    let mut clusters: Vec<i32> = vec![-1; self.pairs.len()];
    let n = self.cells.len();
    let chunk_size = n / self.threads_cnt;
    let self_arc = Arc::new(&self);
    let union_find = Arc::new(Mutex::new(union_find));
    let _ = crossbeam::scope(|scope| {
        for slice in (0..n).step_by(chunk_size) {
            let self_arc = self_arc.clone();
            let union_find = union_find.clone();
            scope.spawn(move |_| {
                for i in slice..std::cmp::min(slice + chunk_size, n) {
                    let cell = &self_arc.cells[i];
                    if cell.amount >= self_arc.min_pts {
                        for neighbor in &cell.neighbors {
                            let first = union_find.lock().unwrap().find(cell.id);
                            let second = union_find
                                .lock()
                                .unwrap()
                                .find(self_arc.cells[*neighbor].id);
                            if cell.id > self_arc.cells[*neighbor].id
                                && self_arc.cells[*neighbor].amount >= self_arc.min_pts
                                && first != second
                            {
                                union_find.lock().unwrap().union(i, *neighbor);
                            }
                        }
                    }
                }
            });
        }
    });
    #[cfg(debug_assertions)]
    {
        println!("union find done");
    }
    let _ = crossbeam::scope(|scope| {
        let mut clusters = clusters.iter_mut();
        for (i, _) in (0..n).step_by(chunk_size).enumerate() {
            let self_arc = self_arc.clone();
            let union_find = union_find.clone();
            let mut clusters = clusters
```

```
                .by_ref()
                .take(
                    (-(self.cells[i * chunk_size].point_id as i64)
                        + (self.cells[std::cmp::min((i + 1) * chunk_size, n) - 1].point_id
                            as i64)
                        + (self.cells[std::cmp::min((i + 1) * chunk_size, n) - 1].amount
                            as i64)) as usize,
                )
                .collect::<Vec<_>>();
            let k = self.cells[i * chunk_size].point_id;
            scope.spawn(move |_| {
                for j in i * chunk_size..std::cmp::min((i + 1) * chunk_size, n) {
                    let cell = &self_arc.cells[j];
                    if cell.amount >= self_arc.min_pts {
                        for p in cell.point_id..(cell.point_id + cell.amount) {
                            if self_arc.visited[p] {
                                *clusters[p - k] = union_find.lock().unwrap().find(j) as i32;
                            }
                        }
                    }
                }
            });
        }
    });
    #[cfg(debug_assertions)]
    {
        println!("clusters done");
    }
    self.clusters = clusters.to_vec();
}
```

Листинг 3.7 – Параллельный алгоритм DBSCAN, расширение кластеров

```
        let n = self.cells.len();
        let chunk_size = n / self.threads_cnt;
        let self_arc = Arc::new(&self);
        let self_mutex = Arc::new(Mutex::new(self.clusters.clone()));
        let _ = crossbeam::scope(|scope| {
            for slice in (0..n).step_by(chunk_size) {
                let self_arc = self_arc.clone();
                let self_mutex = self_mutex.clone();
                scope.spawn(move |_| {
                    let cell = &self_arc.cells[slice];
                    if cell.amount < self_arc.min_pts {
                        for i in cell.point_id..(cell.point_id + cell.amount) {
                            for neighbor in &cell.neighbors {
                                for p in self_arc.cells[*neighbor].point_id
                                    ..(self_arc.cells[*neighbor].point_id
                                        + self_arc.cells[*neighbor].amount)
                                {
                                    if self_arc.clusters[p] != -1
                                        && self_arc.dst(self_arc.pairs[i], self_arc.pairs[p])
                                            <= self_arc.eps
                                    {
                                        self_mutex.lock().unwrap()[i] = self_arc.clusters[p];
                                        break;
                                    }
                                }
                            }
                        }
                    }
                });
            }
        });
        self.clusters = self_mutex.lock().unwrap().clone();
    }

    fn dst(&self, (x1, y1): (f64, f64), (x2, y2): (f64, f64)) -> f64
```

3.4 Тестовые данные

Одним из недостатков алгоритма DBSCAN является его неодназначность, т.е. нет гарантии в однозначности результата. В связи с этим провести сравнение с неким эталонным результатом не представляется возможным

Вывод

В данном разделе был продемонстрирован исходный код алгоритма DBSACN, последовательного и параллельного.

4 Исследовательская часть

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система: Arch Linux [1] 64-bit.
- Оперативная память: 16 Гб.
- Процессор: 11th Gen Intel® Core™ i5-11320H @ 3.20 ГГц[2].

4.1 Пример выполнения

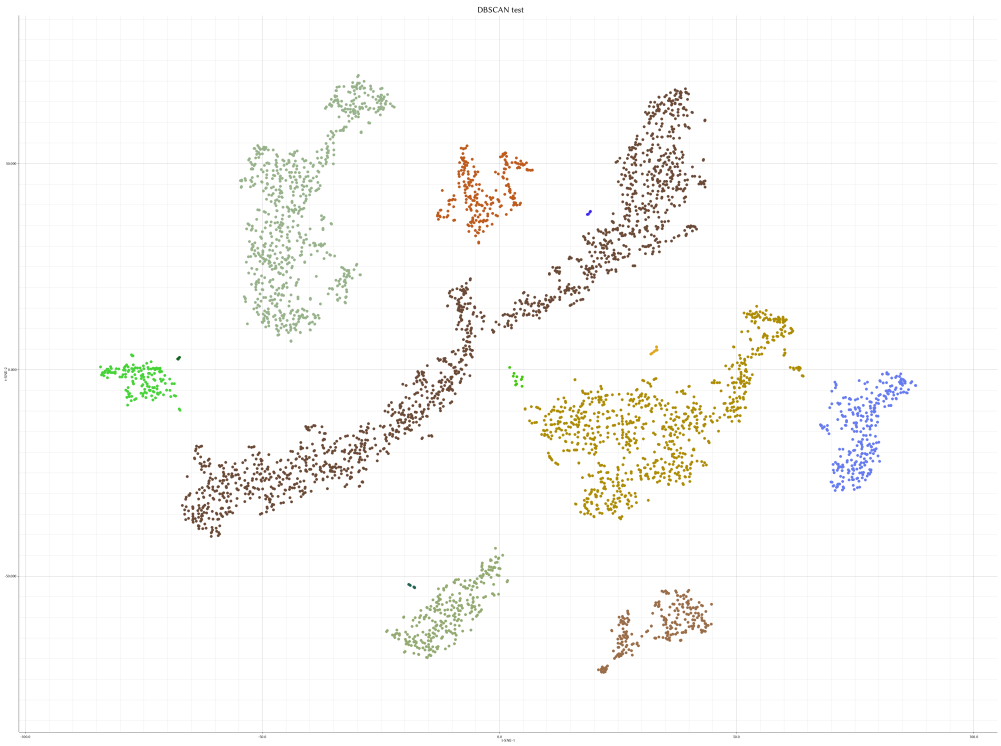

Рисунок 4.1 – Пример выполнения DBSCAN, последовательный алгоритм

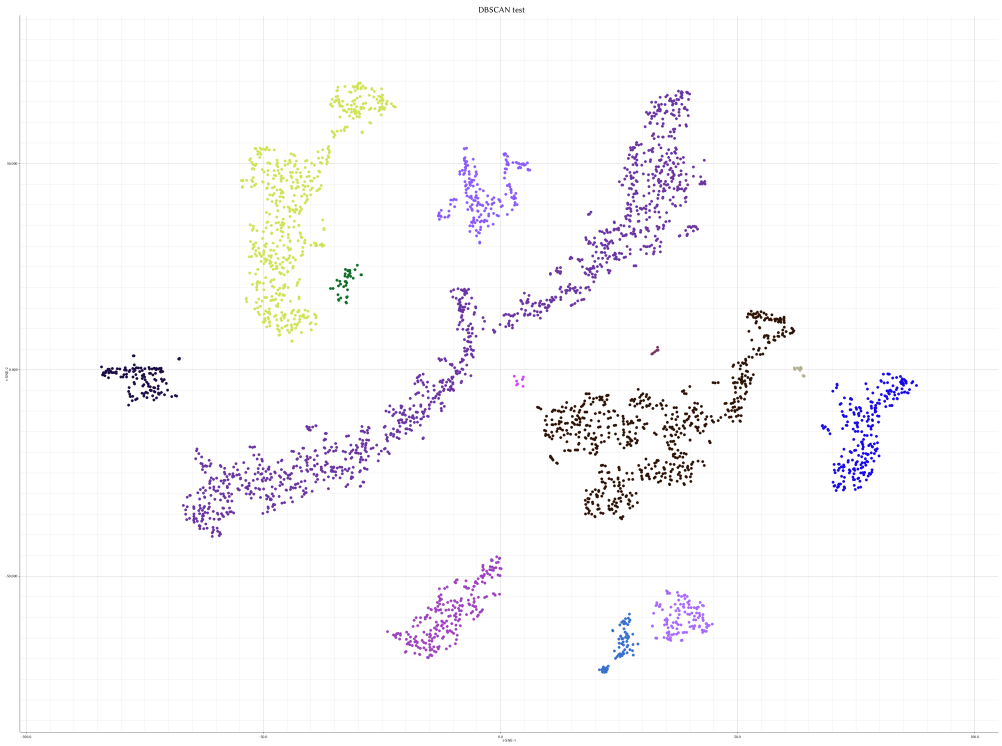

Рисунок 4.2 – Пример выполнения DBSCAN, параллельный алгоритм, 1 поток

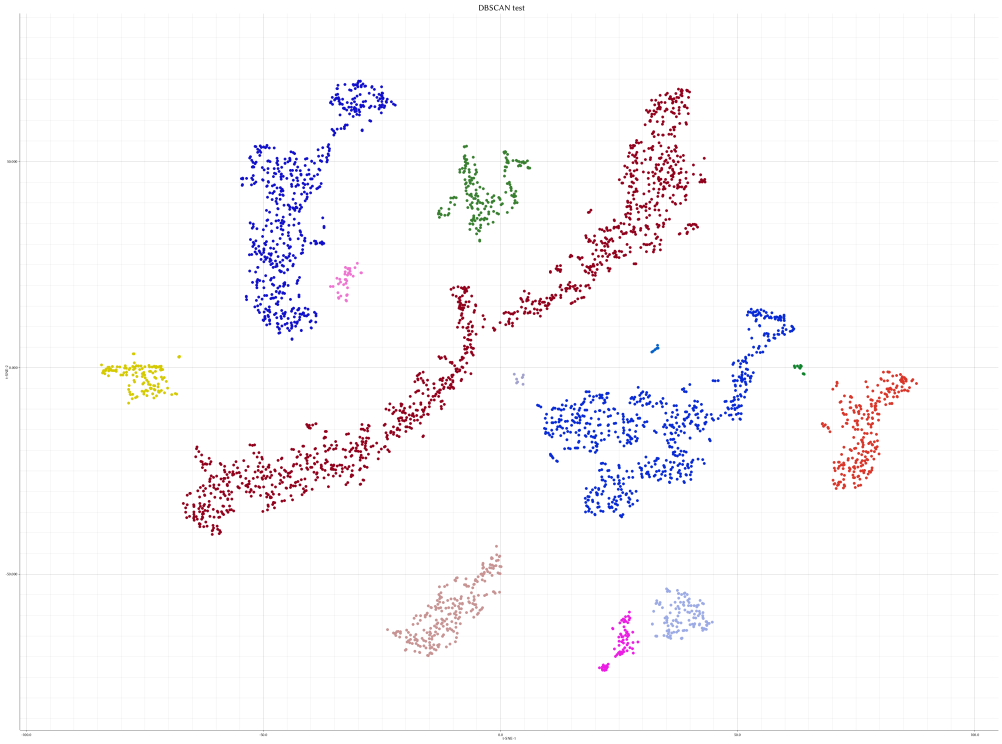

Рисунок 4.3 – Пример выполнения DBSCAN, параллельный алгоритм, 2 потока

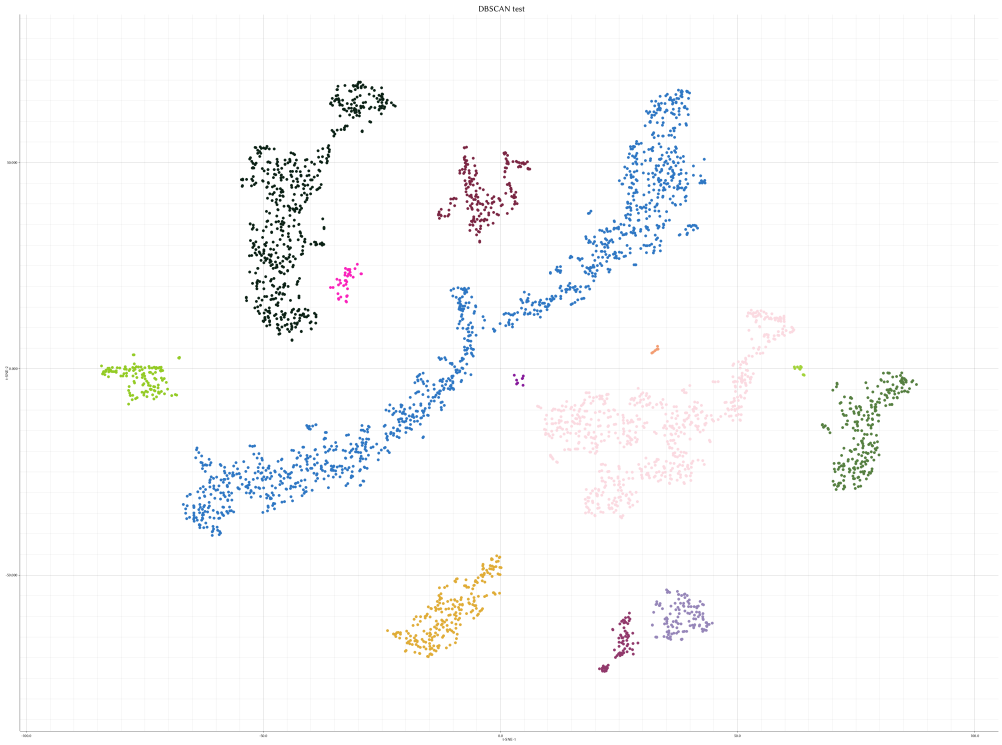

Рисунок 4.4 – Пример выполнения DBSCAN, параллельный алгоритм, 4 потока

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи инструментов замера времени предоставляемых библиотекой Criterion.rs[3]. Пример функции по замеру времени приведен в листинге 4.1. Количество повторов регулируется тестирующей системой самостоятельно, однако ввиду трудоемкости вычислений, количество повторов было ограничено до 25.

Листинг 4.1 – Пример функции замера времени

```
fn dbscan_bench(c: &mut Criterion)
{
    let plot_config = PlotConfiguration::default().summary_scale(AxisScale::Linear);

    let mut group = c.benchmark_group("DBSCAN");
    group.plot_config(plot_config);
    let path = select_file();
    let cols = input_cols();

    let dataset = Dataset::input_dataset(path.clone(), cols.clone(), b',');
    group.sample_size(25);
    let mut serial_model = Model::new(Arc::new(Mutex::new(dataset.clone())));
    serial_model.set_eps(EPS);
    serial_model.set_min_pts(MIN_PTS);
    for size in 0..=6 {
        // group.throughput(Throughput::Bytes(size_of_val(&*m1) as u64 + size_of_val(&*m2) as
            u64));
        let size = 2usize.pow(size);
        let mut parallel_model = ParallelModel::new(Arc::new(Mutex::new(dataset.clone())),
            size);
        parallel_model.set_eps(EPS);
        parallel_model.set_min_pts(MIN_PTS);
        group.bench_with_input(BenchmarkId::new("Parallel", size), &size, |b, _size| {
            b.iter(|| {
                parallel_model.run();
                parallel_model.reset();
                black_box(())
            })
        });
    }

    group.finish();
}
```

График, показывающий время работы последовательного и параллельного алгоритмов в зависимости от количества потоков

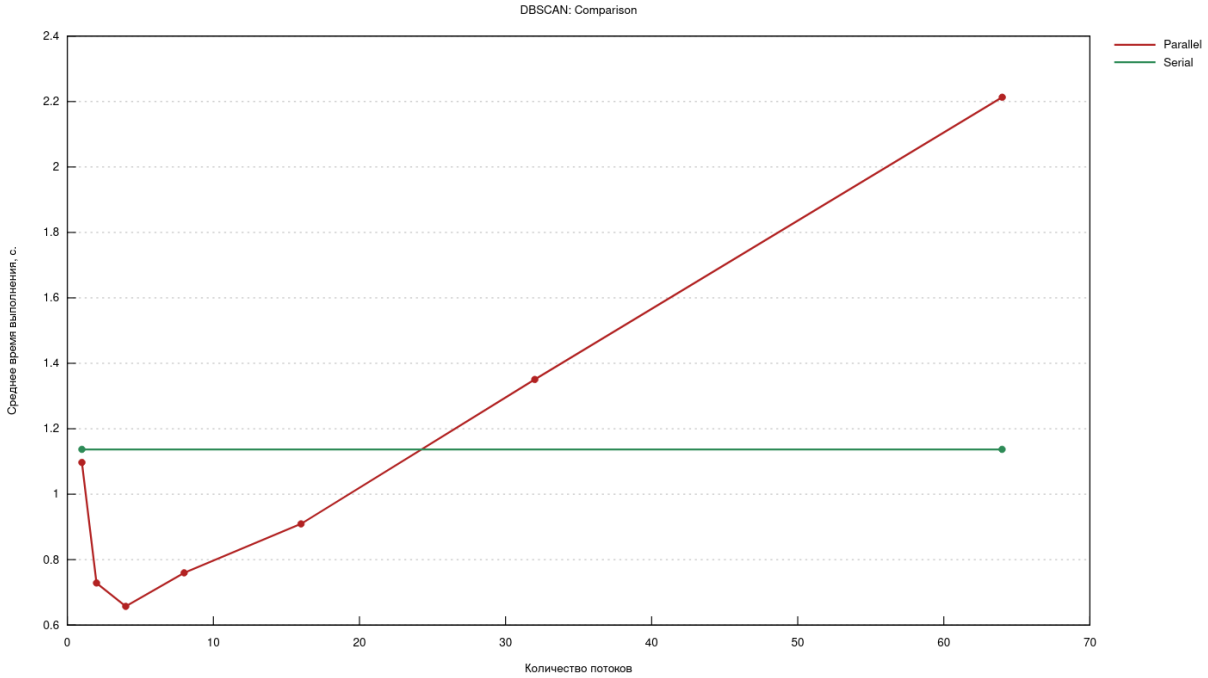

Рисунок 4.5 – Время работы DBSCAN

Как видно из полученных данных, параллельная реализация работает быстрее при количестве рабочих потоков от 1 до 16. Слегка более быстрая работа паралелльного алгоритма может быть объяснена небольшими оптимизациями, которые были реализованы в ходе написания паралельного алгоритма. Затраты, связанные с созданием поток(-а/-ов) при этом несущественны. Начиная с 8 рабочих потоков, процессорное время работы параллельной реализации растёт, и, при 32 рабочих потоках, время работы последовательного алгоритма превышает время работы параллельного. Такой рост связан с тем, что процессор не может эффективно обслуживать более 8 потоков, что объясняет полученные результаты.

Вывод

В результате эксперимента было установлено, что параллельная реализация алгоритма DBSCAN работает быстрее, чем последовательная при количестве рабочих потоков от 1 до 16 рабочих потоках. Это связано с тем, что использование более 8 рабочих потоков приводит к излишним временным затратам на диспетчеризацию потоков.

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы:

1. Были изучены и реализованы 2 версии алгоритма DBSCAN: последовательный и параллельный;
2. Был сделан сравнительный анализ алгоритмов на основе экспериментальных данных.
3. Был подготовлен отчёт по лабораторной работе, содержащий: особенности реализации многопоточной модели; временные характеристики предложенной реализации; результаты работы алгоритма;

Время работы последовательного алгоритма DBSCAN больше паралелльного алгоритма, однако при неверном расчете количества потоков параллельный алгоритм значительно проигрывает своему аналогу.

Цели лабораторной работы по изучению и исследованию параллельных вычислений были достигнуты

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Arch Linux [Электронный ресурс]. — Дата обращения: 19.10.2022. Режим доступа: `https://archlinux.org/`.
2. Процессор Intel® Core™ i5-11320H [Электронный ресурс]. — Дата обращения: 19.10.2022. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu.html`.
3. Criterion [Электронный ресурс]. — Дата обращения: 19.10.2022. Режим доступа: `https://github.com/bheisler/criterion.rs`.