Задача 1 - Метод K-Средних

Реализуйте параллельную версию метода k-средних с использованием технологии OpenMP. Используйте как несколько потоков, так и векторизацию. Обоснуйте выбранную стратегию распараллеливания.

Разобьем функцию Kmeans на 4 части (инициализация центроидов, поиск ближайшего центроида для точек и 1й и 2й цикл пересчета центроидов) и замерим время выполнения каждой из них:

```
vector<size_t> KMeans(const Points& data, size_t K) {
    size_t data_size = data.size();
    size_t dimensions = data[0].size();
    vector<size_t> clusters(data_size);

    Points centroids(K);

    for (size_t i = 0; i < K; ++i) {
        centroids[i] = data[UniformRandom(data_size - 1)];
    }

    bool converged = false;
    while (!converged) {
        converged = true;
        for (size_t i = 0; i < data_size; ++i) {
            size_t nearest_cluster = FindNearestCentroid(centroids, data[i]);
            if (clusters[i] != nearest_cluster) {
                clusters[i] = nearest_cluster;
                converged = false;
            }
        }
        if (converged) {
            break;
        }

        vector<size_t> clusters_sizes(K);
        centroids.assign(K, Point(dimensions));
        for (size_t i = 0; i < data_size; ++i) {
            for (size_t d = 0; d < dimensions; ++d) {
                centroids[clusters[i]][d] += data[i][d];
            }
            ++clusters_sizes[clusters[i]];
        }

        for (size_t i = 0; i < K; ++i) {
            if (clusters_sizes[i] != 0) {
                for (size_t d = 0; d < dimensions; ++d) {
                    centroids[i][d] /= clusters_sizes[i];
                }
            } else {
                centroids[i] = GetRandomPosition(centroids);
            }
        }
    }
    return clusters;
}
```

Получим следующие результаты:

1. ~0%
2. 97.23%
3. 2.77%
4. ~0%

Вывод: не имеет смысла оптимизировать цикл 1, т. к. время его выполнения очень мало (имеем единственный не вложенный цикл, а также как правило кол-во кластеров значительно меньше кол-ва точек), к тому же

функция rand() не является реентерабельной (потокобезопасной), т.к хранит seed в глобальной переменной. Цикл 4 также не имеет смысл оптимизировать.

```
for (size_t i = 0; i < K; ++i) {
    if (clusters_sizes[i] != 0) {
        for (size_t d = 0; d < dimensions; ++d) {
            centroids[i][d] /= clusters_sizes[i];
        }
    } else {
        centroids[i] = GetRandomPosition(centroids);
    }
}
```

Цикл 2 оптимизировать также не получится, т. к. могут возникнуть гонки по данным (в разных циклах происходит обращение к одним и тем же ячейкам массива), поэтому необходимо объявить массивы атомарными примерно так:

```
#pragma omp parallel for schedule(static)
for (size_t i = 0; i < data_size; ++i) {
    for (size_t d = 0; d < dimensions; ++d) {
        #pragma omp atomic
        centroids[clusters[i]][d] += data[i][d];
    }
    #pragma omp atomic
    ++clusters_sizes[clusters[i]];
}
```

что скорее всего не сильно улучшит результат, а возможно даже ухудшит, т. к. может возникнуть конкуренция за доступ к данным. Оставим без оптимизации.

Оптимизируем цикл 3. Данный цикл занимает более 97% времени работы программы. Опимизируем его под параллельное исполнение:

```
#pragma omp parallel for schedule(static)
for (size_t i = 0; i < data_size; ++i) {
    size_t nearest_cluster = FindNearestCentroid(centroids, data[i]);
    if (clusters[i] != nearest_cluster) {
        clusters[i] = nearest_cluster;
        converged = false;
    }
}
```

в данном цикле нет конкурентного доступа к данным и нет вызова потоконебезопасных функций.

В функциях Distance и GetRandomPosition можно предложить компилятору использовать векторизацию.

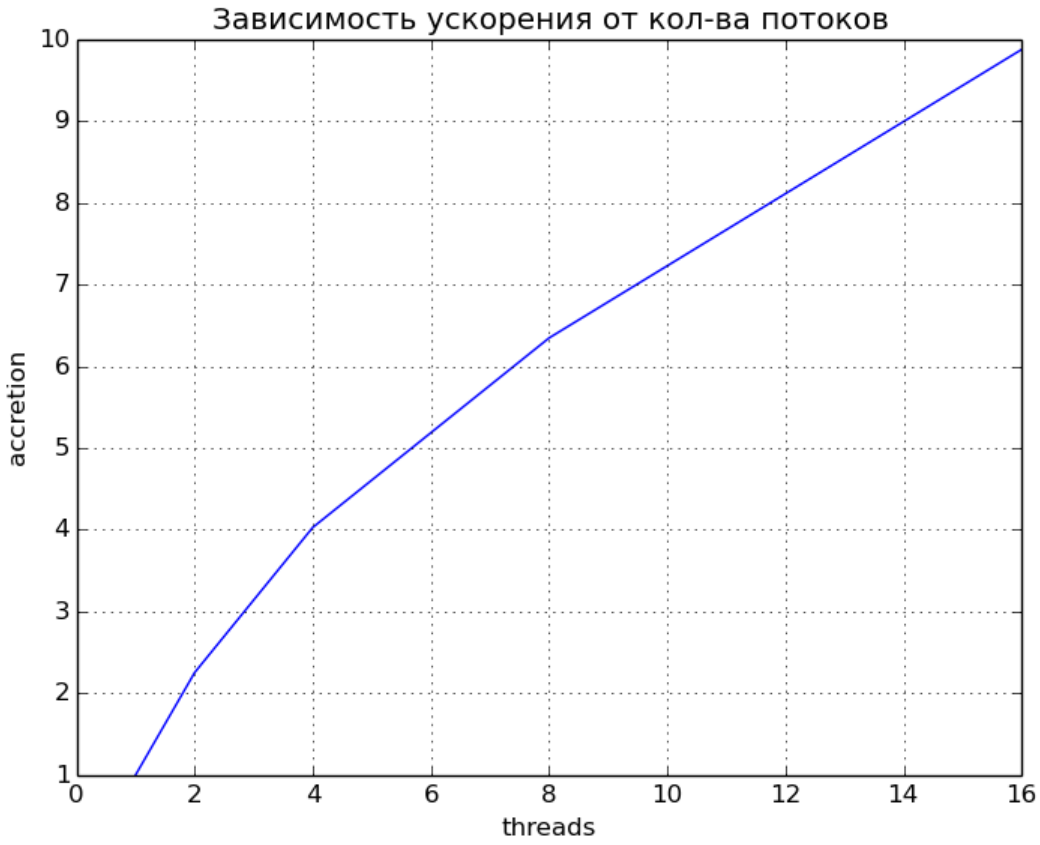
```
#pragma omp simd reduction(+:distance_sqr)
for (size_t i = 0; i < dimensions; ++i) {
    distance_sqr += (point1[i] - point2[i]) * (point1[i] - point2[i]);
}

#pragma omp simd
for (size_t d = 0; d < dimensions; ++d) {
    new_position[d] = (centroids[c1][d] + centroids[c2][d] + centroids[c3][d]) / 3;
}
```

Запустим программу с разными входными параметрами. Получим следующие результаты:

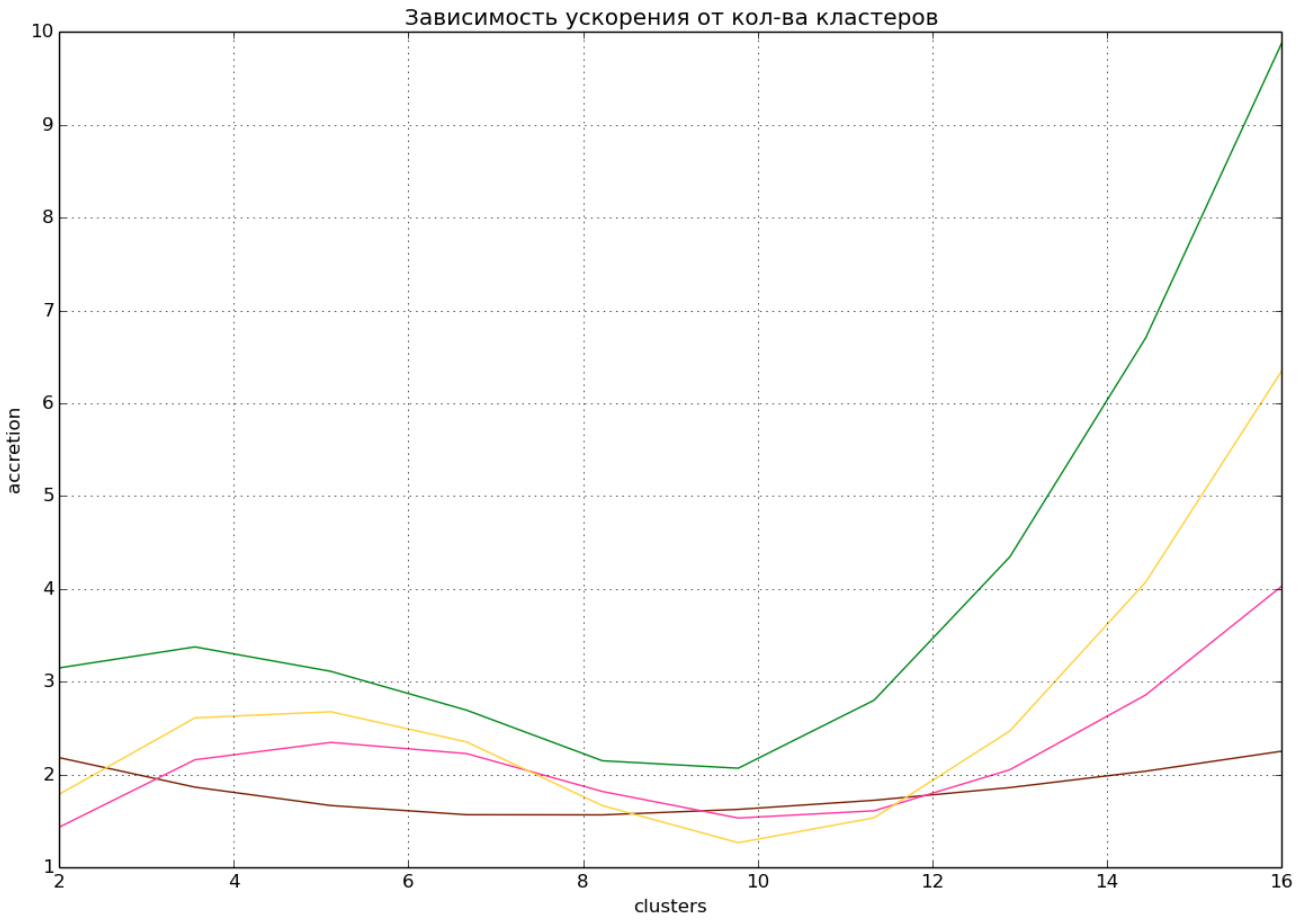
размерность данных	2				4				16			
кол-во точек / кол-во кластеров	1 000	10 000	100 000	1 000 000	1 000	10 000	100 000	1 000 000	1 000	10 000	100 000	1 000 000
2	0:00.00 0:00.03 0:00.00 0:00.14 0:00.12	0:00.03 0:00.06 0:00.02 0:00.03 0:00.24	0:00.31 0:00.25 0:00.24 0:00.26 0:00.72	0:04.01 0:02.99 0:02.43 0:02.35 0:01.94	0:00.02 0:00.23 0:00.00 0:00.09 0:00.10	0:00.07 0:00.08 0:00.05 0:00.06 0:00.21	0:01.02 0:00.62 0:00.55 0:01.01 0:00.71	0:07.25 0:04.54 0:03.89 0:04.43 0:05.48	0:01.99 0:00.04 0:00.10 0:00.08 0:00.20	0:04.62 0:00.19 0:00.27 0:00.39 0:00.25	0:03.76 0:02.13 0:02.51 0:02.51 0:01.44	0:25.94 0:12.10 0:18.31 0:14.51 0:08.26
4	0:00.00 0:00.00 0:00.00 0:00.01 0:00.13	0:00.07 0:00.10 0:00.04 0:00.04 0:00.38	0:00.50 0:00.37 0:00.48 0:00.29 0:00.32	0:08.45 0:05.60 0:04.07 0:03.30 0:02.65	0:00.02 0:00.05 0:00.00 0:00.17 0:00.27	0:00.23 0:00.13 0:00.09 0:00.11 0:00.65	0:00.99 0:00.78 0:00.48 0:00.98 0:00.67	0:15.48 0:09.63 0:08.17 0:06.48 0:04.99	0:01.74 0:00.04 0:00.19 0:00.10 0:00.18	0:03.28 0:00.21 0:00.35 0:00.53 0:00.40	0:03.62 0:01.74 0:01.62 0:02.27 0:01.05	1:01.25 0:34.09 0:26.75 0:22.53 0:18.27
8	0:00.01 0:00.01 0:00.00 0:00.01 0:00.40	0:00.12 0:00.17 0:00.14 0:00.05 0:00.41	0:01.49 0:00.96 0:00.58 0:00.47 0:00.60	0:29.83 0:18.11 0:10.55 0:07.53 0:05.55	0:00.02 0:00.14 0:00.01 0:00.02 0:00.35	0:00.26 0:00.15 0:00.12 0:00.10 0:00.55	0:02.32 0:01.36 0:00.85 0:00.86 0:03.14	0:46.22 0:24.98 0:20.01 0:12.13 0:07.27	0:01.96 0:00.03 0:00.05 0:00.13 0:00.14	0:03.05 0:00.34 0:00.28 0:00.46 0:00.32	1:10.04 0:41.28 0:25.11 0:16.12 0:13.36	0:40.15 0:25.46 0:20.78 0:21.98 0:18.04
16	0:00.01 0:00.01 0:00.00 0:00.01 0:00.27	0:00.41 0:00.31 0:00.17 0:00.12 0:01.27	0:11.61 0:06.10 0:03.29 0:02.00 0:02.24	2:30.89 1:20.08 0:44.08 0:25.77 0:17.84	0:00.08 0:00.05 0:00.02 0:00.14 0:00.63	0:00.65 0:00.32 0:00.19 0:00.18 0:00.75	0:06.65 0:02.84 0:01.64 0:01.69 0:00.96	5:19.90 2:46.18 1:33.44 0:55.28 0:47.08	0:01.66 0:00.07 0:00.20 0:00.04 0:00.32	0:03.14 0:00.97 0:01.35 0:00.63 0:00.70	2:24.20 1:19.57 0:45.03 0:27.23 0:18.37	23:55.04 10:37.01 5:55.73 3:46.08 2:25.19

Кол-во потоков: - 1 - 2 - 4 - 8 - 16



Как видно из графика, при увеличении кол-ва потоков увеличивается и ускорение работы программы, но рост не линейный, видно, что производная уменьшается, следовательно можно предположить, что при определенном кол-ве потоков рост прекратится. Следуя результатам, описанным выше (более 97% времени программа проводит в цикле, который удалось распараллелить, остальные нет), и руководствуясь законом Амдала, можно предложить, что при кол-ве потоков около 32 прирост производительности должен быть близок к 0.

Построим график зависимости ускорения от кол-ва кластеров. Видно, что при увеличении кол-ва кластеров ускорение увеличивается, при том увеличивается тем быстрее, чем больше кол-во потоков, т. к. удалось распараллелить цикл, который ищет ближайший к точке кластер.



Построим график зависимости ускорения от кол-ва точек (график в логарифмическом масштабе). Важно отметить: как видно из таблицы, при малом кол-ве точек программа выполняется за миллисекунды, что приводит к большой погрешности замеров времени выполнения (результат сильно зависит от загруженности кластера, работы планировщика и. т. д), следовательно на левый «хвост» графиков не стоит обращать внимания.

Видно, что при кол-ве точек от 10 000 и дальнейшем увеличении увеличивается и ускорение, но при этом при кол-ве точек около 1 000 000 переходит в насыщение.