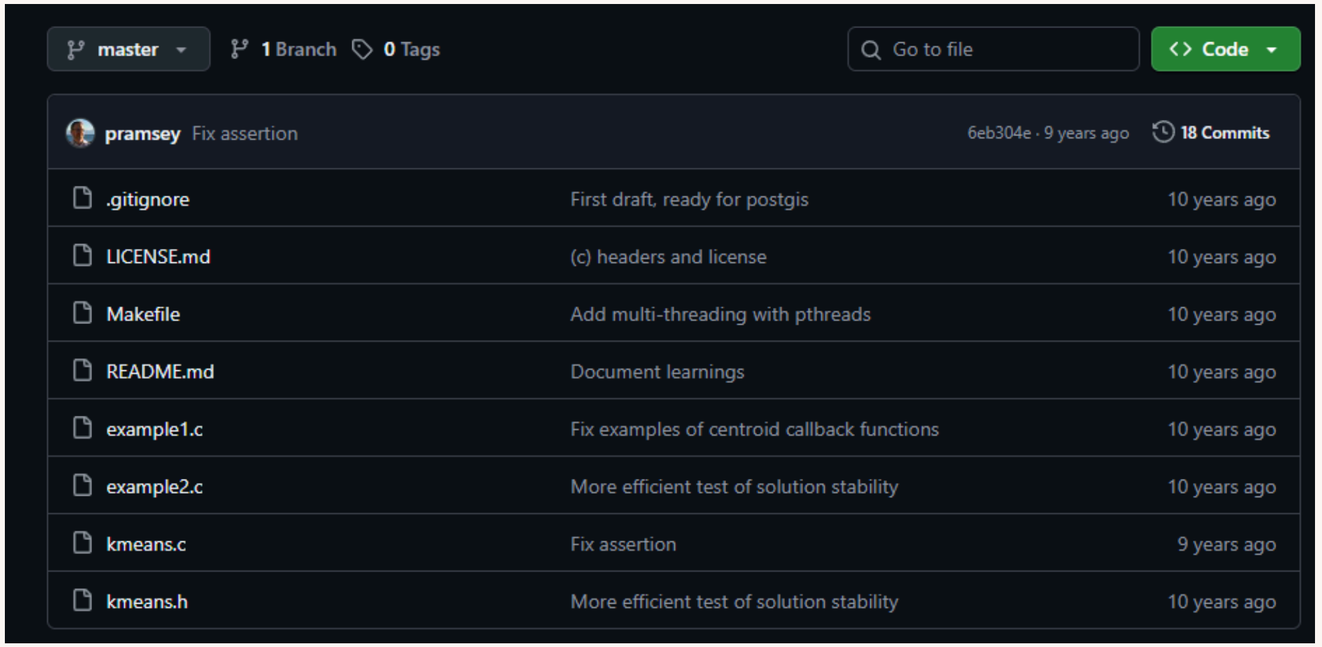
Implementação Paralela de uma Técnica de Inteligência Artificial

Diego Marchioni, Gabriel Rajão, Luiza Dias,
Sophia Carrazza

GITHUB DO PROJETO

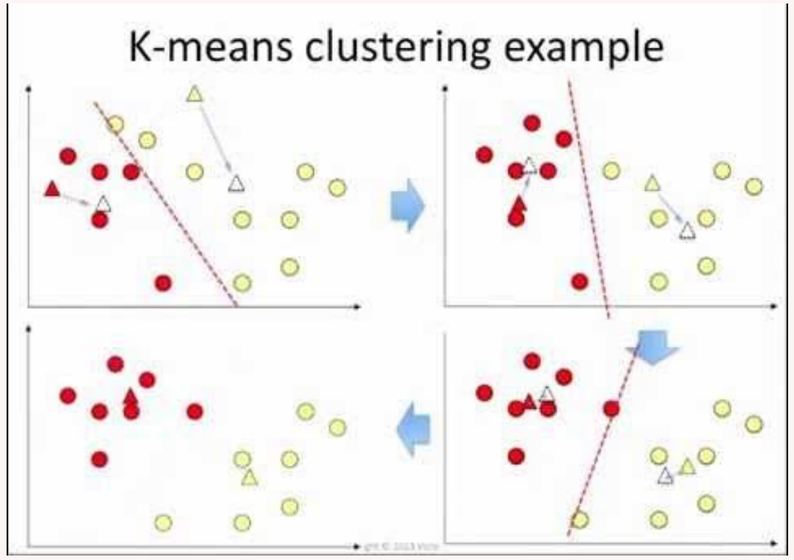
https://github.com/gabrielrajao/kmeansParalelizado

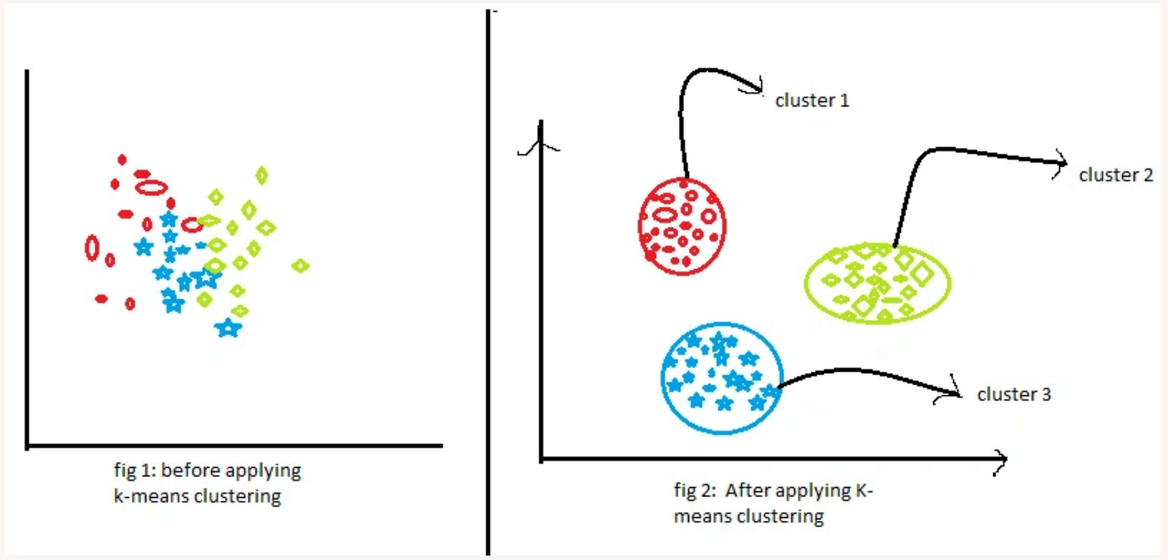
Código Escolhido



https://github.com/pramsey/kmeans

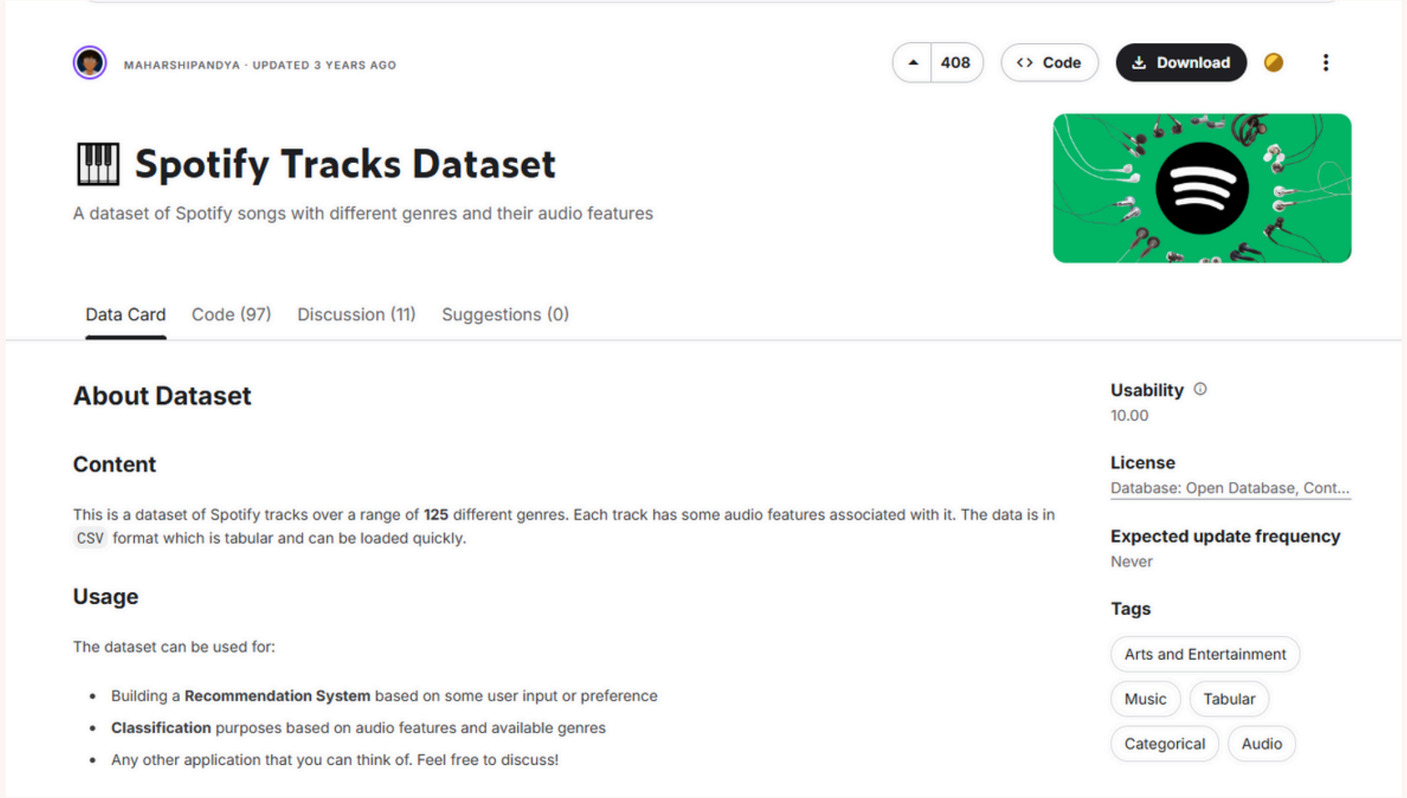
Código Escolhido





Para relembrar: Algoritmo de **agrupamento** (clustering) de aprendizado de máquina **não supervisionado** que agrupa dados em clusters (k)

Base de Dados



MAHARSHIPANDYA · UPDATED 3 YEARS AGO

408 Code Download

Spotify Tracks Dataset

A dataset of Spotify songs with different genres and their audio features

Data Card Code (97) Discussion (11) Suggestions (0)

About Dataset

Content

This is a dataset of Spotify tracks over a range of **125** different genres. Each track has some audio features associated with it. The data is in `CSV` format which is tabular and can be loaded quickly.

Usage

The dataset can be used for:

- Building a **Recommendation System** based on some user input or preference
- **Classification** purposes based on audio features and available genres
- Any other application that you can think of. Feel free to discuss!

Usability
10.00

License
Database: Open Database, Cont...

Expected update frequency
Never

Tags
Arts and Entertainment
Music Tabular
Categorical Audio

https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset

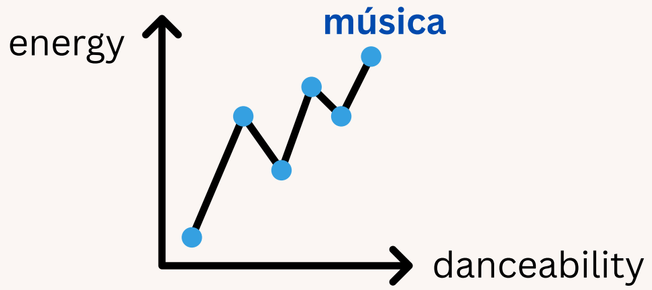
Objetivo de Resultado

Identificar os clusters de músicas em sentimento, de acordo com a tupla:

- **danceability (Dançabilidade):** Quão adequada uma faixa é para dançar (0.0 a 1.0).
- **energy (Energia):** Uma medida de intensidade e atividade (0.0 a 1.0).

datasetFilt.csv > data

```
1   danceability,energy
2   0.676,0.461
3   0.42,0.166
4   0.438,0.359
5   0.266,0.0596
```



Aplicação Paralela

```c
static void pt_centroid(const Pointer *objs, const int *clusters,
                        size_t num_objs, int cluster, Pointer centroid) {
    int i;
    int num_cluster = 0;
    double sumx = 0.0, sumy = 0.0;
    point **pts = (point **)objs;
    point *center = (point *)centroid;

    /* PARALELIZAÇÃO: Loop paralelo com reduction */
    #ifdef _OPENMP
    #pragma omp parallel for reduction(+:sumx,sumy,num_cluster) schedule(guided)
    #endif
    for (i = 0; i < num_objs; i++) {
        if (clusters[i] != cluster || objs[i] == NULL)
            continue;
        sumx += pts[i]->x;
        sumy += pts[i]->y;
        num_cluster++;
    }

    if (num_cluster > 0) {
        center->x = sumx / num_cluster;
        center->y = sumy / num_cluster;
    }
}
```

Reduction: recalcula a posição do centróide pela média dos seus pontos, OpenMP distribui essa tarefa de soma entre os núcleos do processador, otimizando o desempenho em grandes volumes de dados.

Vantagens:

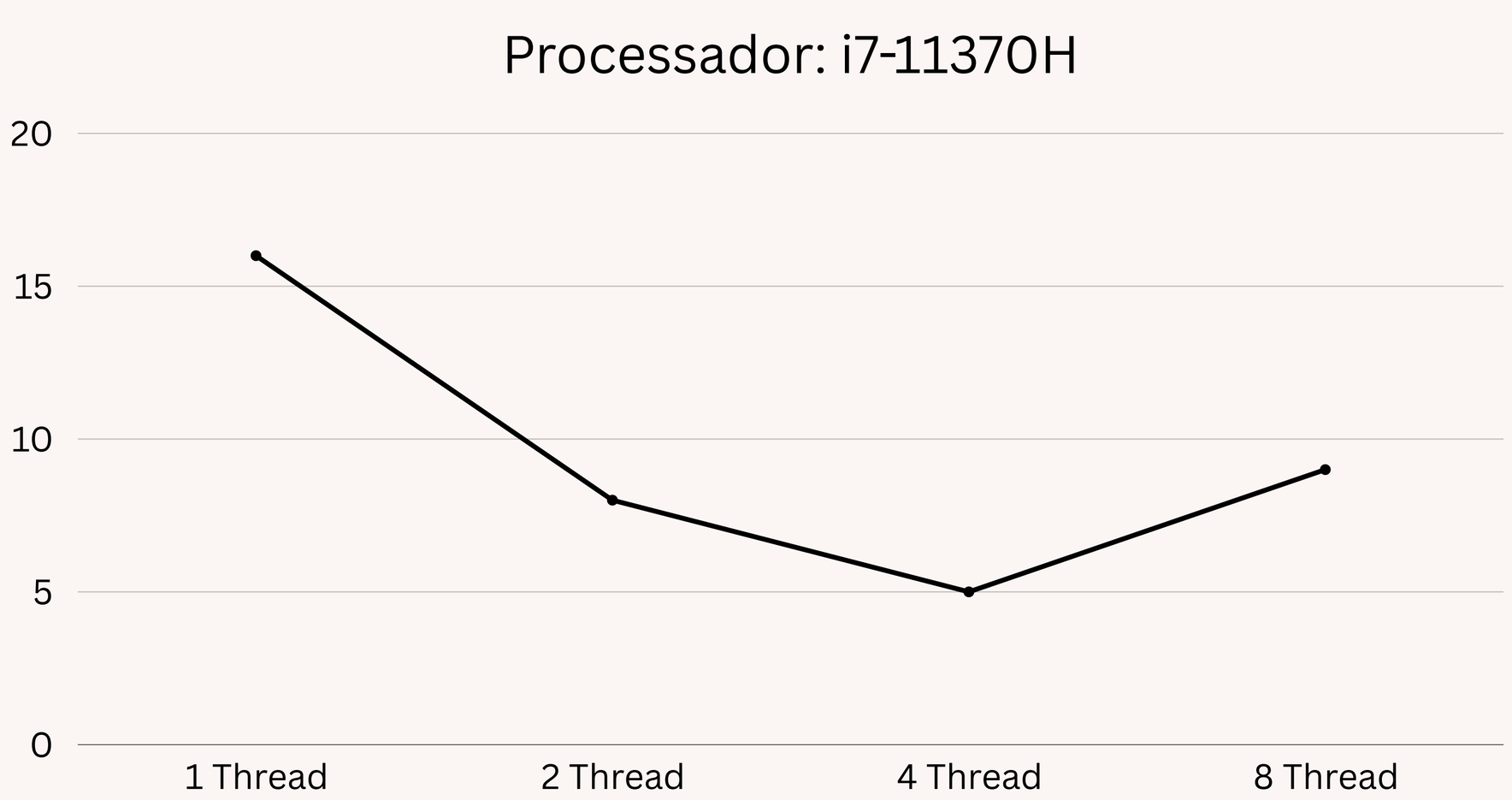
- Veloz e possui escalabilidade alta
- Seguro (garante que não ocorre "Condição de Corrida")

Aplicação Paralela

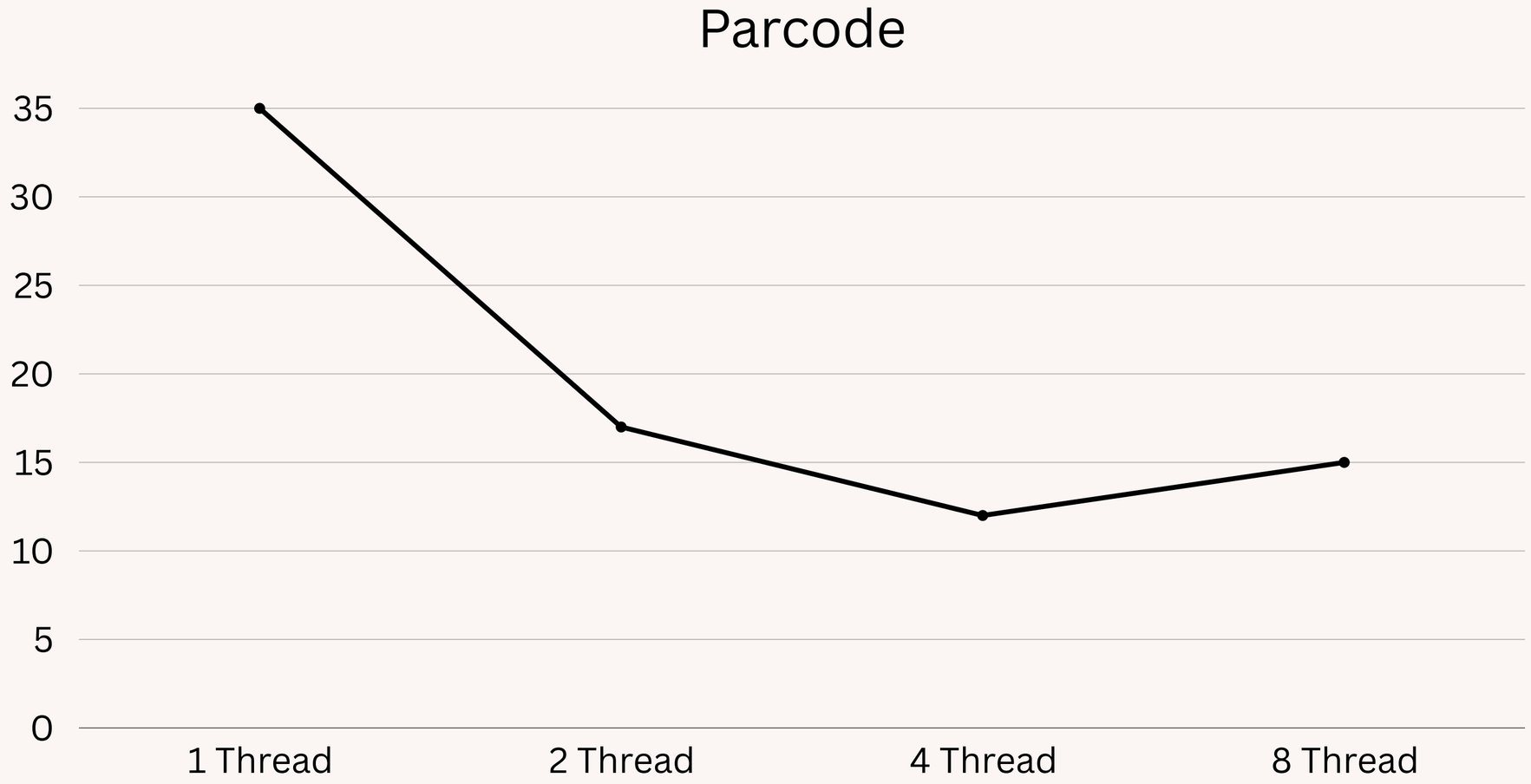
```
#pragma omp parallel for schedule(guided, 128)
#endif
for (i = 0; i < config->num_objs; i++) {
    double distance, curr_distance;
    int cluster, curr_cluster;
    Pointer obj;

    assert(config->objs != NULL);
    assert(config->num_objs > 0);
    assert(config->centers);
    assert(config->clusters);

    obj = config->objs[i];
    if (!obj) {
        config->clusters[i] = KMEANS_NULL_CLUSTER;
        continue;
    }

    /* Inicializa com a distância para o primeiro cluster */
    curr_distance = (config->distance_method)(obj, config->centers[0]);
    curr_cluster = 0;

    /* Verifica todos os clusters e encontra o mais próximo */
    for (cluster = 1; cluster < config->k; cluster++) {
        distance = (config->distance_method)(obj, config->centers[cluster]);
        if (distance < curr_distance) {
            curr_distance = distance;
            curr_cluster = cluster;
        }
    }

    /* Armazena o cluster mais próximo */
    config->clusters[i] = curr_cluster;
}
```

Balanceamento: atribui cada ponto de dados ao seu cluster mais próximo, o OpenMP, permite que todos os núcleos da CPU trabalhem ao mesmo tempo para classificar os pontos, o que resulta em um ganho de performance drástico.

Tempos com Aplicação Paralela



Tempos com Aplicação Paralela



Aplicação Hibrida

```
MPI_Bcast(&num_objs, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* Aloca memória em todos os processos */
if (rank != 0) {
    pts = calloc(num_objs, sizeof(point));
}

MPI_Bcast(pts, num_objs * 2, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* Configura K-means em todos os processos */
init = calloc(k, sizeof(point));
config.k = k;
config.num_objs = num_objs;
config.max_iterations = 200;
config.distance_method = pt_distance;
config.centroid_method = pt_centroid;
config.objs = calloc(num_objs, sizeof(Pointer));
config.centers = calloc(k, sizeof(Pointer));
config.clusters = calloc(num_objs, sizeof(int));

for (int i = 0; i < num_objs; i++) {
    config.objs[i] = &pts[i];
}

/* Inicializa centróides (apenas master, depois faz broadcast) */
if (rank == 0) {
    srand(42); // seed fixa para reprodutibilidade
    for (int i = 0; i < k; i++) {
        int r = rand() % num_objs;
        init[i] = pts[r];
    }
}

MPI_Bcast(init, k * 2, MPI_DOUBLE, 0, MPI_COMM_WORLD);

for (int i = 0; i < k; i++) {
    config.centers[i] = &init[i];
}
```

MPI_Bcast: é usado para configurar o ambiente, garantindo que no início do algoritmo K-means todos os processos tenham uma cópia idêntica dos dados e dos centróides iniciais.

Aplicação Hibrida

```c
/* Arrays para somas locais e globais */
double *local_sums_x = calloc(k, sizeof(double));
double *local_sums_y = calloc(k, sizeof(double));
int *local_counts = calloc(k, sizeof(int));

double *global_sums_x = calloc(k, sizeof(double));
double *global_sums_y = calloc(k, sizeof(double));
int *global_counts = calloc(k, sizeof(int));

/* Calcula somas locais para cada cluster */
for (i = 0; i < config->num_objs; i++) {
    if (config->objs[i] == NULL) continue;
    int cluster = config->clusters[i];
    if (cluster >= 0 && cluster < k) {
        point *pt = (point *)config->objs[i];
        local_sums_x[cluster] += pt->x;
        local_sums_y[cluster] += pt->y;
        local_counts[cluster]++;
    }
}

MPI_Allreduce(local_sums_x, global_sums_x, k, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(local_sums_y, global_sums_y, k, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(local_counts, global_counts, k, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

/* Atualiza centróides com valores globais */
for (i = 0; i < k; i++) {
    if (global_counts[i] > 0) {
        point *center = (point *)config->centers[i];
        center->x = global_sums_x[i] / global_counts[i];
        center->y = global_sums_y[i] / global_counts[i];
    }
}

free(local_sums_x);
free(local_sums_y);
free(local_counts);
free(global_sums_x);
free(global_sums_y);
free(global_counts);
```

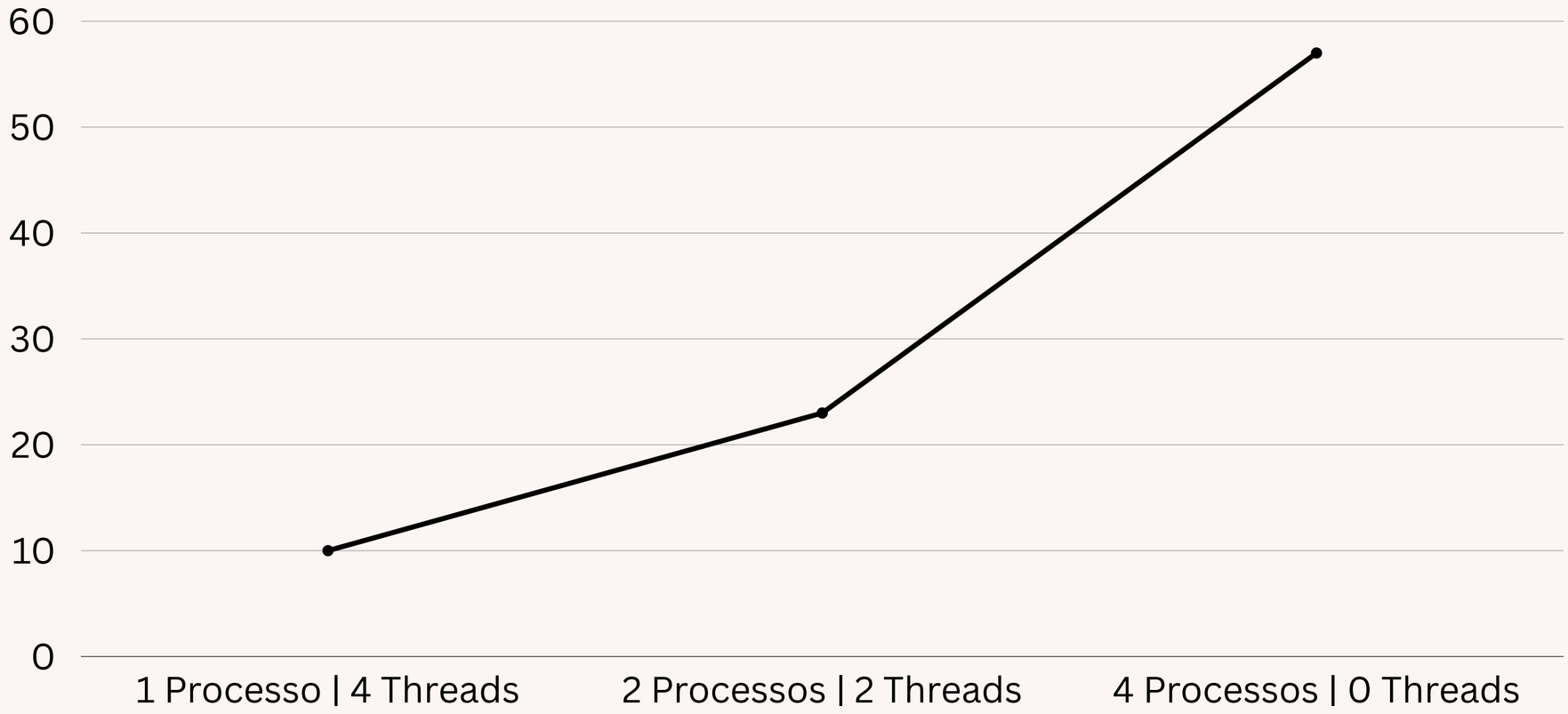
MPI_Allreduce: na etapa de recalcular os centróides, cada processo calcula somas parciais e contagens de pontos para cada cluster. Para obter o centróide global correto, esses valores parciais de todos os processos precisam ser somados.

Aplicação Hibrida

```c
kmeans_result kmeans_mpi(kmeans_config *config, int rank, int size) {
    int iterations = 0;
    int *clusters_last;
    size_t clusters_sz = sizeof(int) * config->num_objs;
    int converged = 0;

    if (!config->max_iterations)
        config->max_iterations = KMEANS_MAX_ITERATIONS;

    clusters_last = calloc(clusters_sz, 1);
    memset(config->clusters, 0, clusters_sz);

    while (!converged && iterations < config->max_iterations) {
        memcpy(clusters_last, config->clusters, clusters_sz);

        /* Atualiza clusters localmente */
        update_r(config);

        /* PARALELIZAÇÃO MPI: Agrega centróides entre processos */
        mpi_aggregate_centroids(config, rank, size);

        /* Verifica convergência localmente */
        int local_converged = (memcmp(clusters_last, config->clusters, clusters_sz) == 0);

        /* Verifica convergência global */
        MPI_Allreduce(&local_converged, &converged, 1, MPI_INT, MPI_LAND, MPI_COMM_WORLD);

        iterations++;
    }

    free(clusters_last);
    config->total_iterations = iterations;

    if (converged)
        return KMEANS_OK;
    else
        return KMEANS_EXCEEDED_MAX_ITERATIONS;
}
```

MPI_Allreduce: verifica se a convergência foi atingida em todos os processos simultaneamente, o algoritmo só termina quando há um consenso global de que os clusters não estão mais mudando.

Tempos com Aplicação Hibrida



Obrigado pela atenção!