Projeto PCD: K-means 1D (naive) com Paralelização Progressiva

Profs. Álvaro e Denise (Turmas I e N)

O algoritmo K-Means realiza uma operação de agrupamento ("clusterização") para mineração de dados, ou seja, permite agrupar amostras de um dado conjunto em grupos homogêneos. O método particiona um conjunto de *n* observações (pontos) em *k* grupos, onde cada ponto será associado ao grupo cuja média seja a mais próxima. A distância Euclidiana é geralmente a métrica adotada para medir a proximidade. A "clusterização" é um problema *NP-hard*, mas existem algoritmos heurísticos eficientes que podem rapidamente encontrar um ótimo local. Nesta implementação, a aplicação recebe como entrada as coordenadas (1D) de *k* centróides iniciais e um conjunto de dados. O K-Means realiza um processo iterativo, no qual os pontos são reagrupados de acordo com a menor distância Euclidiana entre eles e os centróides. Em seguida, o centróide de cada partição é recalculado tomando a média de todos os pontos da partição, e todo o procedimento é repetido até que nenhum centróide seja alterado e nenhum ponto seja atribuído a outro grupo. Ao final, o algoritmo retorna as coordenadas dos *k* centróides finais.

Objetivo

Implementar o **k-means em 1 dimensão** (pontos `X[i]` e centróides `C[c]`), medir **SSE** e **desempenho**, e **paralelizar** o núcleo do algoritmo em três etapas independentes:

1. **OpenMP (CPU memória compartilhada)**
2. **CUDA (GPU)**
3. **MPI (memória distribuída)**

Entradas e saídas

- **Entradas (CSV, 1 coluna, sem cabeçalho):**
  - `dados.csv` com **N** valores (pontos).
  - `centroides_iniciais.csv` com **K** valores.
- **Saídas:**
  - No terminal: **iterações**, **SSE (*Sum of Squared Errors*, ou Soma dos Erros Quadráticos, em português) final**, **tempo total** (ms).
  - Arquivos: `assign.csv` (N linhas, índice do cluster por ponto) e `centroids.csv` (K linhas com centróides finais).

Algoritmo (base “naive”)

Itere até `max_iter` ou até variar pouco o SSE (`eps`):

1. **Assignment:** para cada ponto, escolher o centróide mais próximo (minimiza $(x_i - c)^2$); acumular **SSE**.
2. **Update:** para cada cluster, **média** dos pontos atribuídos. Se um cluster ficar vazio, **copie** `X[0]` (estratégia simples).

Etapa 0 — Versão sequencial (baseline)

- Executar a versão **sequencial** (fornecida mais abaixo).
- Coletar: **SSE por iteração**, **tempo total**, **iterações**.
- Salvar esses números: serão a **linha de base** para speedup.

Etapa 1 — OpenMP (CPU)

Meta: paralelizar as funções de *assignment* e *update* na CPU.

O que paralelizar

- **Assignment:** laço `for (i=0; i<N; ++i)`.
- **Update:**
  - opção A (mais simples): usar **acumuladores por thread** (`sum_thread[c]`, `cnt_thread[c]`) e **reduzir** após a região paralela;
  - opção B: usar `#pragma omp critical` e verificar impactos no desempenho.

Medições

- **Escalonamento em threads:** T ∈ {1, 2, 4, 8, 16, ...}.
- **Speedup** = tempo_serial / tempo_OpenMP.
- **Afinar:** *schedule* (`static` vs `dynamic`) e *chunk size*.
- **Validação:** SSE não deve **aumentar** ao longo das iterações (pode ficar igual se convergiu).

Dica de compilação

```
gcc -O2 -fopenmp -std=c99 kmeans_1d_omp.c -o kmeans_1d_omp -lm
```

Etapa 2 — CUDA (GPU)

Meta: mover o **assignment** para a GPU; o **update** pode ser feito na GPU (com atomics) ou no host (copiando `assign`).

Desenho mínimo

- **Kernel de assignment:** 1 *thread* por ponto `i`.
  - Cada thread varre **K** centróides, calcula `d = (X[i]-C[c])^2`, guarda o melhor e escreve `assign[i]`.
  - (Opcional) carregar `C` em **memória constante**.
- **SSE:** reduzir no host somando os erros por ponto (ou fazer redução em blocos).
- **Update:**
  - opção A (mais simples): copiar `assign` para CPU e calcular médias no host;
  - opção B: usar **atomics** em `sum[c]` e `cnt[c]` na GPU e depois dividir.

Medições

- **Tamanho de bloco** (p.ex., 128, 256, 512) × **grid**;
- **Tempos**: H2D/D2H, *kernel*, total;
- **Throughput**: pontos/s; **speedup** vs. serial e vs. OpenMP.

Dica de compilação

```
nvcc -O2 kmeans_1d_cuda.cu -o kmeans_1d_cuda
```

Etapa 3 — MPI (distribuída)

Meta: distribuir os **N** pontos entre **P** processos; centróides são **globais** a cada iteração.

Passos por iteração

1. **Broadcast** (ou inicialização compartilhada): todos os processos têm `C`.
2. **Assignment local:** cada processo calcula `assign_local` e `SSE_local` para seu bloco de pontos.
3. **Redução global:**
   - somar `SSE_local` → `SSE_global` com `MPI_Reduce`;
   - somar `sum_local[c]` e `cnt_local[c]` para todos os clusters com `MPI_Allreduce`;
   - cada processo atualiza `C` com os resultados globais.
4. Próxima iteração até convergir.

Medições

- **Strong scaling:** P ∈ {1, 2, 4, 8, ...}.
- **Tempo de comunicação:** destacar o custo de `Allreduce`.

- **Speedup** vs. serial e OpenMP.

Dica de compilação/execução

```
mpicc -O2 -std=c99 kmeans_1d_mpi.c -o kmeans_1d_mpi -lm
mpirun -np 4 ./kmeans_1d_mpi dados.csv centroides_iniciais.csv [args...]
```

Conjuntos de teste sugeridos (1D)

- **Pequeno:** N=10^4, K=4
- **Médio:** N=10^5, K=8
- **Grande:** N=10^6, K=16 (se houver memória) Gere dados com mistura de faixas (ex.: perto de 0, 10, 20, 30) para facilitar a verificação visual.

O que entregar

1. **Código no github**: `serial/`, `openmp/`, `cuda/`, `mpi/` (cada pasta com `README.md` de como compilar/rodar).
2. **Relatório curto (4–6 págs po etapa)**:
    - Ambiente (CPU/GPU/RAM/rede; versões de compilador).
    - Gráficos: **tempo**, **speedup**, **pontos/s** por etapa.
    - Para MPI: curva de **speedup** e comentário sobre custo de `Allreduce`.
    - Para CUDA: impacto de **block size** e custo de **transferência**.
    - Para OpenMP: efeito de **nº de threads** e de *schedule*.
    - Seções de **validação** (SSE por iteração, convergência, igualdade de resultados entre versões dentro de tolerância).
    - Análise de resultados e conclusões
    - Referências bibliográficas: apresente uma pequena revisão bibliográfica e compare seus resultados com outros encontrados na literatura.

Critérios de avaliação

- **Desempenho e análise** (speedup, tempos de execução, eficiência e gargalos por arquitetura): **30%**
- **Corretude e reprodutibilidade** (SSE consistente, convergência, demonstração da corretude da execução): **30%**
- **Relatório** (clareza, gráficos, referências bibliográficas, análises e conclusões): **20%**
- **Qualidade do código e organização**: **10%**
- **Extra** (implementação e/ou análises não sugeridas no enunciado e que melhorem a qualidade científica do trabalho): **10%**

Dicas rápidas

- Padronize **parâmetros** (N, K, `max_iter`, `eps`) entre as versões para comparar.
- Fixe uma **semente** ao gerar dados (quando aplicável) para repetibilidade.

Exemplo: código sequencial e arquivos de entrada e saída

```
%%writefile centroides_iniciais.csv

10
30
60
90
```

```
Overwriting centroides_iniciais.csv
```

```
%%writefile dados.csv

1
2
3
4
5
6
7
8
4.5
5.5
18
19
20
21
22
23
19.5
20.5
100
125
```

```
Overwriting dados.csv
```

```
%%writefile kmeans_1d_naive.c

/* kmeans_1d_naive.c
   K-means 1D (C99), implementação "naive":
   - Lê X (N linhas, 1 coluna) e C_init (K linhas, 1 coluna) de CSVs sem cabeça
   - Itera assignment + update até max_iter ou variação relativa do SSE < eps.
   - Salva (opcional) assign (N linhas) e centróides finais (K linhas).
```

```
   Compilar: gcc -O2 -std=c99 kmeans_1d_naive.c -o kmeans_1d_naive -lm
   Uso:      ./kmeans_1d_naive dados.csv centroides_iniciais.csv [max_iter=50]
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

/* ---------- util CSV 1D: cada linha tem 1 número ---------- */
static int count_rows(const char *path){
    FILE *f = fopen(path, "r");
    if(!f){ fprintf(stderr,"Erro ao abrir %s\n", path); exit(1); }
    int rows=0; char line[8192];
    while(fgets(line,sizeof(line),f)){
        int only_ws=1;
        for(char *p=line; *p; p++){
            if(*p!=' ' && *p!='\t' && *p!='\n' && *p!='\r'){ only_ws=0; break;
        }
        if(!only_ws) rows++;
    }
    fclose(f);
    return rows;
}

static double *read_csv_1col(const char *path, int *n_out){
    int R = count_rows(path);
    if(R<=0){ fprintf(stderr,"Arquivo vazio: %s\n", path); exit(1); }
    double *A = (double*)malloc((size_t)R * sizeof(double));
    if(!A){ fprintf(stderr,"Sem memoria para %d linhas\n", R); exit(1); }

    FILE *f = fopen(path, "r");
    if(!f){ fprintf(stderr,"Erro ao abrir %s\n", path); free(A); exit(1); }

    char line[8192];
    int r=0;
    while(fgets(line,sizeof(line),f)){
        int only_ws=1;
        for(char *p=line; *p; p++){
            if(*p!=' ' && *p!='\t' && *p!='\n' && *p!='\r'){ only_ws=0; break;
        }
        if(only_ws) continue;

        /* aceita vírgula/ponto-e-vírgula/espaco/tab, pega o primeiro token num
        const char *delim = ",; \t";
        char *tok = strtok(line, delim);
        if(!tok){ fprintf(stderr,"Linha %d sem valor em %s\n", r+1, path); free
        A[r] = atof(tok);
        r++;
        if(r>R) break;
    }
    fclose(f);
    *n_out = R;
    return A;
```

```
}

static void write_assign_csv(const char *path, const int *assign, int N){
    if(!path) return;
    FILE *f = fopen(path, "w");
    if(!f){ fprintf(stderr,"Erro ao abrir %s para escrita\n", path); return; }
    for(int i=0;i<N;i++) fprintf(f, "%d\n", assign[i]);
    fclose(f);
}

static void write_centroids_csv(const char *path, const double *C, int K){
    if(!path) return;
    FILE *f = fopen(path, "w");
    if(!f){ fprintf(stderr,"Erro ao abrir %s para escrita\n", path); return; }
    for(int c=0;c<K;c++) fprintf(f, "%.6f\n", C[c]);
    fclose(f);
}

/* ---------- k-means 1D ---------- */
/* assignment: para cada X[i], encontra c com menor (X[i]-C[c])^2 */
static double assignment_step_1d(const double *X, const double *C, int *assign,
    double sse = 0.0;
    for(int i=0;i<N;i++){
        int best = -1;
        double bestd = 1e300;
        for(int c=0;c<K;c++){
            double diff = X[i] - C[c];
            double d = diff*diff;
            if(d < bestd){ bestd = d; best = c; }
        }
        assign[i] = best;
        sse += bestd;
    }
    return sse;
}

/* update: média dos pontos de cada cluster (1D)
   se cluster vazio, copia X[0] (estratégia naive) */
static void update_step_1d(const double *X, double *C, const int *assign, int N
    double *sum = (double*)calloc((size_t)K, sizeof(double));
    int *cnt = (int*)calloc((size_t)K, sizeof(int));
    if(!sum || !cnt){ fprintf(stderr,"Sem memoria no update\n"); exit(1); }

    for(int i=0;i<N;i++){
        int a = assign[i];
        cnt[a] += 1;
        sum[a] += X[i];
    }
    for(int c=0;c<K;c++){
        if(cnt[c] > 0) C[c] = sum[c] / (double)cnt[c];
        else           C[c] = X[0]; /* simples: cluster vazio recebe o primeirc
    }
    free(sum); free(cnt);
}
```

```c
static void kmeans_1d(const double *X, double *C, int *assign,
                      int N, int K, int max_iter, double eps,
                      int *iters_out, double *sse_out)
{
    double prev_sse = 1e300;
    double sse = 0.0;
    int it;
    for(it=0; it<max_iter; it++){
        sse = assignment_step_1d(X, C, assign, N, K);
        /* parada por variação relativa do SSE */
        double rel = fabs(sse - prev_sse) / (prev_sse > 0.0 ? prev_sse : 1.0);
        if(rel < eps){ it++; break; }
        update_step_1d(X, C, assign, N, K);
        prev_sse = sse;
    }
    *iters_out = it;
    *sse_out = sse;
}

/* ---------- main ---------- */
int main(int argc, char **argv){
    if(argc < 3){
        printf("Uso: %s dados.csv centroides_iniciais.csv [max_iter=50] [eps=1e
        printf("Obs: arquivos CSV com 1 coluna (1 valor por linha), sem cabeçal
        return 1;
    }
    const char *pathX = argv[1];
    const char *pathC = argv[2];
    int max_iter = (argc>3)? atoi(argv[3]) : 50;
    double eps   = (argc>4)? atof(argv[4]) : 1e-4;
    const char *outAssign   = (argc>5)? argv[5] : NULL;
    const char *outCentroid = (argc>6)? argv[6] : NULL;

    if(max_iter <= 0 || eps <= 0.0){
        fprintf(stderr,"Parâmetros inválidos: max_iter>0 e eps>0\n");
        return 1;
    }

    int N=0, K=0;
    double *X = read_csv_1col(pathX, &N);
    double *C = read_csv_1col(pathC, &K);
    int *assign = (int*)malloc((size_t)N * sizeof(int));
    if(!assign){ fprintf(stderr,"Sem memoria para assign\n"); free(X); free(C);

    clock_t t0 = clock();
    int iters = 0; double sse = 0.0;
    kmeans_1d(X, C, assign, N, K, max_iter, eps, &iters, &sse);
    clock_t t1 = clock();
    double ms = 1000.0 * (double)(t1 - t0) / (double)CLOCKS_PER_SEC;

    printf("K-means 1D (naive)\n");
    printf("N=%d K=%d max_iter=%d eps=%g\n", N, K, max_iter, eps);
    printf("Iterações: %d | SSE final: %.6f | Tempo: %.1f ms\n", iters, sse, ms

    write_assign_csv(outAssign, assign, N);
```

```
    write_centroids_csv(outCentroid, C, K);

    free(assign); free(X); free(C);
    return 0;
}
```

```
Overwriting kmeans_1d_naive.c
```

```
%%shell

gcc -O2 -std=c99 kmeans_1d_naive.c -o kmeans_1d_naive -lm
./kmeans_1d_naive dados.csv centroides_iniciais.csv 50 0.000001 assign.csv cent
cat centroids.csv
```

```
K-means 1D (naive)
N=20 K=4 max_iter=50 eps=1e-06
Iterações: 4 | SSE final: 344.875000 | Tempo: 0.0 ms
6.300000
20.375000
2.900000
112.500000
```