UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

PARALELIZAÇÃO DE KNN EM C

Exercício Programa II

FILIPE BISON DE SOUZA - 14570653 - Turma 94
GABRIELA ALCAIDE - 14746492 - Turma 94
RODRIGO GONÇALVES CARDOSO - 14658330 - Turma 94

SÃO PAULO
2024

FILIPE BISON DE SOUZA - 14570653 - Turma 94
GABRIELA ALCAIDE - 14746492 - Turma 94
RODRIGO GONÇALVES CARDOSO - 14658330 - Turma 94

PARALELIZAÇÃO DE KNN EM C

Exercício Programa II

Trabalho apresentado para o Exercício Programa II da disciplina de Organização e Arquitetura de Computadores II (OACII), ministrada pelo professor Dr. Clodoaldo Aparecido de Moraes Lima.

SÃO PAULO
2024

SUMÁRIO

1. PROPOSTA

Este projeto tem como objetivo transcrever o algoritmo de KNN, anteriormente desenvolvido em MIPS, para a Linguagem C. Além disso, pretende-se paralelizar o código, utilizando OpenMP, a fim de otimizar o desempenho do programa.

2. COMO COMPILAR

- Código sequencial: gcc -o knns knn_sequencial.c

- Código paralelizado da seção 3.2:
  gcc -o knn1 knn_paralelo_1_todas_paralelizacoes_plausiveis.c -fopenmp

- Código paralelizado da seção 3.3: gcc -o knn2 knn_paralelo_2_paralelizar_tudo_menos_calcular_distancias_e_encontrar_k_menores.c -fopenmp

- Código paralelizado da seção 3.4: gcc -o knn3 knn_paralelo_3_paralelizar_exclusivamente_laco_de_criar_YTest.c -fopenmp

3. COMO EXECUTAR (PASSANDO PARÂMETROS)

- Código sequencial: ./knns.exe valor_largura valor_altura valor_k

- Código paralelizado da seção 3.2:
  ./knn1.exe valor_largura valor_altura valor_k

- Código paralelizado da seção 3.3:
  ./knn2.exe valor_largura valor_altura valor_k

- Código paralelizado da seção 3.4:
  ./knn3.exe valor_largura valor_altura valor_k

Vale ressaltar que, ao executar o programa (conforme comando especificado na linha acima), será realizado um KNN para cada um dos 08 arquivos de teste disponibilizados.

Destaca-se que todos os YTest entregues foram gerados com os seguintes argumentos: w = 3; h = 1; k = 2. Ou seja, foi executado “./nome_programa 3 1 2” para gerar os valores de YTest

4. CÓDIGO SEQUENCIAL EM C

4.1. Medição do tempo

A medição de tempo era essencial nesse projeto, para realizarmos comparações entre os conjuntos de teste e entre o sequencial e o paralelizado.

Inicialmente, utilizamos a função clock() da biblioteca padrão C para mensurar o tempo de execução das operações. No entanto, verificou-se que a precisão dessa abordagem era insuficiente para conjuntos de teste menores, resultando em tempos medidos como zero devido à baixa resolução do método. Para solucionar esse problema, exploramos alternativas como clock_gettime(), amplamente utilizada em sistemas Unix-like. Contudo, ao realizar testes em diferentes ambientes, optamos por continuar no sistema operacional Windows, onde encontramos um método baseado na função QueryPerformanceCounter() para obter maior precisão na medição de tempo.

```
double obtemTempoEmSegundos() {
    LARGE_INTEGER frequency, start;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);
    return (double)start.QuadPart / frequency.QuadPart;
}
```

A função obterTempoEmSegundos, baseada na QueryPerformanceCounter(), foi escolhida por sua capacidade de medir intervalos de tempo com alta resolução, essencial para operações rápidas, possibilitando uma medição para conjuntos menores.

A medição foi limitada ao intervalo em que é feita a criação da matriz do conjunto de teste e a construção do array yTest, destacando a execução do núcleo do algoritmo para cada conjunto de treino. Essa abordagem garante uma análise mais precisa do desempenho das operações críticas, eliminando possíveis interferências de outras partes do código, como a leitura de arquivos, a qual, não será paralelizada.

4.2. Leitura do arquivo

Na etapa de leitura do arquivo, adotou-se uma abordagem que combina a contagem prévia de valores com a alocação dinâmica de memória. Inicialmente, uma função percorre o arquivo para contar o número de valores, garantindo que a memória necessária seja alocada de forma eficiente. Essa contagem também assegura que arquivos com tamanhos variáveis possam ser processados de maneira fácil.

```c
int contarLinhasArquivo(const char* nomeArquivo) {

    FILE* arquivo = fopen(nomeArquivo, "r");

    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo!\n");
        return -1;
    }

    int linhas = 0;
    double valor;

    while (fscanf(arquivo, "%lf", &valor) == 1) {  // Conta cada valor lido como uma linha
        linhas++;
    }

    fclose(arquivo);  // Fecha o arquivo após a contagem
    return linhas;
}
```

Após a contagem, o arquivo é reaberto para leitura, e os valores são armazenados em um array alocado dinamicamente. Essa estratégia evita desperdício de memória e garante que os dados sejam carregados com precisão, permitindo um processamento subsequente eficiente e confiável.

```c
// Função para ler os dados de um arquivo e recebe um ponteiro para um vetor
// que será preenchido com os valores do arquivo
void ler_arquivo(const char *nome_arquivo, double **valores, int *tamanho) {
    *tamanho = contarLinhasArquivo(nome_arquivo);  // Obtém a quantidade de linhas

    if (*tamanho <= 0) {
        *valores = NULL;
        return;
    }

    FILE* arquivo = fopen(nome_arquivo, "r"); // Abre o arquivo novamente para leitura dos valores

    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo!\n");
        *valores = NULL;
        return;
    }

    *valores = (double*)malloc(sizeof(double)* *tamanho);  // Aloca o array com o tamanho exato

    for (int i = 0; i < *tamanho; i++) {
        fscanf(arquivo, "%lf", &(*valores)[i]);  // Lê cada valor do arquivo
    }

    fclose(arquivo);  // Fecha o arquivo após a leitura
}
```

4.3. Criar Matrizes

No algoritmo do KNN é necessário utilizar as matrizes com os dados do conjunto, tanto para o de treinamento quanto para o de testes. Para isso foi criada uma só função que realiza essa lógica para ambos casos.

A função, nomeada criar_matriz, é responsável por definir, alocar e popular uma matriz em que tem as dimensões igual a largura (w) por valor obtido por meio do cálculo de (tamanho do conjunto - largura (w) - altura (h) +1). É também verificado se esse valor é positivo, o que valida os parâmetros. Caso não forem a execução é interrompida.

```c
// Função para criar uma matriz baseada no vetor de entrada
double **criar_matriz(double *dados, int tamanho, int largura, int altura, int *linhas_matriz) {
    *linhas_matriz = tamanho - largura - altura + 1;
    if (*linhas_matriz <= 0) {
        fprintf(stderr, "Dimensões inválidas para criação da matriz\n");
        exit(EXIT_FAILURE);
    }

    double **matriz = (double **)malloc(*linhas_matriz * sizeof(double *));
```

Após a alocação, a matriz é populada com os valores do conjunto de dados fornecido. Cada elemento é preenchido com base em um deslocamento calculado a

partir da posição atual na matriz e na largura definida, garantindo que os dados sejam organizados de forma sequencial e adequada ao modelo.

```
    for (int i = 0; i < *linhas_matriz; i++) {
        matriz[i] = (double *)malloc(largura * sizeof(double));
        for (int j = 0; j < largura; j++) {
            matriz[i][j] = dados[i + j];
        }
    }

    return matriz;
}
```

4.4. Fase de treino

Na fase de treino é utilizada a função criar matriz apresentada anteriormente para criar a matriz de treino, que será utilizada para na fase de testes escolher qual intervalo do conjunto será utilizado na escolha da previsão.

Além da criação da matriz de treino, é gerado o vetor yTreino, que armazena as previsões correspondentes a cada linha da matriz. Para isso, foi implementada a função criar_yTreino, responsável por alocar dinamicamente um vetor de tamanho igual ao número de linhas da matriz de treino. O vetor é então populado com os valores provenientes do conjunto de treino (xTrain), utilizando um deslocamento calculado a partir da largura e altura especificadas. Esse deslocamento garante que cada valor de yTrain corresponda corretamente ao intervalo de dados relacionado na matriz de treino, estabelecendo a base necessária para o processo de aprendizado do modelo.

4.5. Fase de Testes knn

Na fase de testes do algoritmo KNN, o objetivo principal é realizar previsões para os dados do conjunto de teste com base no conjunto de treino previamente construído. Esse processo, realizado na função knn, envolve a criação de matrizes para o conjunto de teste, o cálculo de previsões utilizando os vizinhos mais próximos e, por fim, a avaliação do erro obtido.

```c
// Função principal do algoritmo KNN
void knn(const char* nomeConjunto, double **matrizTrain, int linhasTrain, double *yTrain,
                int tamanhoTrain, double *xTest, int tamanhoTest, int largura, int altura, int k) {

    // Variáveis para medição do tempo
    double inicio, fim;
    double tempo_total;

    inicio = obterTempoEmSegundos(); // Início da contagem de tempo

    int linhasTest;
    double **matrizTest = criar_matriz(xTest, tamanhoTest, largura, altura, &linhasTest);
    double *yTest = criar_YTest(matrizTrain, linhasTrain, matrizTest, linhasTest, yTrain, largura, k);

    fim = obterTempoEmSegundos(); // Fim da contagem de tempo

    tempo_total = ((double)(fim - inicio));
    printf("Tempo total de execução do teste %s: %.20f segundos\n", nomeConjunto, tempo_total);

    salvar_YTest_em_arquivo(nomeArquivoY(nomeConjunto), yTest, linhasTest);

    double erro = calcular_erro_absoluto_medio(xTest, yTest, linhasTest, largura, altura);
    printf("Erro absoluto médio do conjunto de dados %s: %f\n", nomeConjunto, erro);

    for (int i = 0; i < linhasTest; i++) free(matrizTest[i]);
    free(matrizTest);
    free(yTest);
}
```

4.5.1. Criar Y Test

Nesta etapa, é chamada a função “criar_matriz”, já explicada anteriormente, para estruturar a matriz do conjunto de teste a partir dos dados de entrada. Em seguida, a função “criar_YTest” é utilizada para gerar o vetor de previsões yTest.

Essa função calcula as distâncias entre cada linha da matriz de teste e todas as linhas da matriz de treino, utilizando a função “calcular_distancias”. As distâncias são armazenadas no vetor “indices”, e a função “encontrar_k_menores” identifica os índices das k menores distâncias, representando os vizinhos mais próximos. Com esses índices, os valores correspondentes de yTrain são utilizados para calcular a previsão média, que é armazenada no vetor yTest.

```c
double * criar_YTest(double **matriz_treino, int linhas_treino, double **matriz_teste,
                     int linhas_teste, double *y_treino, int largura, int k) {
    double *y_teste = (double *)malloc(linhas_teste * sizeof(double));
    for (int i = 0; i < linhas_teste; i++) {
        double *linha_teste = matriz_teste[i];
        double *distancias = (double *)malloc(linhas_treino * sizeof(double));

        calcular_distancias(matriz_treino, linhas_treino, largura,
                            linha_teste, linhas_treino, distancias);

        int *indices = (int *)malloc(k * sizeof(int));
        encontrar_k_menores(distancias, linhas_treino, k, indices);

        double soma = 0.0;
        for (int j = 0; j < k; j++) {
            soma += y_treino[indices[j]];
        }
        y_teste[i] = soma / k;

        free(distancias);
        free(indices);
    }
    return y_teste;
}
```

- **Cálculo de distâncias**

O cálculo de distâncias é realizado utilizando a fórmula da distância euclidiana. Cada diferença entre os valores das mesmas colunas das matrizes de treino e teste é elevada ao quadrado, e os resultados são somados para gerar a distância total. Esse processo é repetido para todas as linhas da matriz de treino, resultando em um vetor de distâncias correspondente a cada linha do conjunto de teste.

```c
void calcular_distancias(double **matriz_treino, int linhas_treino, int largura,
                         double *linha_teste, int n, double *distancias) {
    for (int i = 0; i < n; i++) {
        double soma = 0.0;
        for (int j = 0; j < largura; j++) {
            double diff = matriz_treino[i][j] - linha_teste[j];
            soma += diff * diff;
        }
        distancias[i] = soma;
    }
}
```

- **Definir K menores distâncias**

Após o cálculo das distâncias, a função “encontrar_k_menores” seleciona os índices das k menores distâncias, que representam os vizinhos mais próximos. Para evitar repetições, os valores já selecionados no vetor de distâncias são substituídos por infinito, de forma que não serão mais selecionados.

```
void encontrar_k_menores(double *distancias, int n, int k, int *indices) {
    for (int i = 0; i < k; i++) {
        double menor = INFINITY;
        int indice_menor = -1;
        for (int j = 0; j < n; j++) {
            if (distancias[j] < menor) {
                menor = distancias[j];
                indice_menor = j;
            }
        }
        indices[i] = indice_menor;
        distancias[indice_menor] = INFINITY; // Marcar como usado
    }
}
```

4.5.2. Salvar em arquivo

O vetor yTest, contendo as previsões geradas, é salvo em um arquivo por meio da função salvar_YTest_em_arquivo.

```
salvar_YTest_em_arquivo(nomeArquivoY(nomeConjunto), yTest, linhasTest);
```

Essa função cria um arquivo com nome baseado no conjunto de dados de teste, substituindo "xtest" por "yTest", para facilitar a identificação dos resultados.

```
char* nomeArquivoY(const char* nomeConjunto){
    char *ptr = NULL;
    char *nomeArquivoY = strdup(nomeConjunto); // Copiar a string original
    if ((ptr = strstr(nomeArquivoY, "xtest"))) memcpy(ptr, "yTest", 5);
    //trocar a ocorrencia de xTest no nome do arquivo por yTest
    return nomeArquivoY;
}
```

Cada valor de yTest é gravado no arquivo, garantindo que os resultados possam ser analisados posteriormente.

```c
void salvar_YTest_em_arquivo(const char *nome_arquivo, double *yTest, int tamanho) {
    FILE *arquivo = fopen(nome_arquivo, "w");
    if (!arquivo) {
        perror("Erro ao criar o arquivo");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < tamanho; i++) {
        fprintf(arquivo, "%lf\n", yTest[i]);
    }

    fclose(arquivo);
    printf("Resultados de yTest salvos em '%s'.\n", nome_arquivo);
}
```

4.5.3. Cálculo de erro absoluto médio

Após a alocação, a matriz é populada com os valores do conjunto de dados fornecido. Cada elemento é preenchido com base em um deslocamento calculado a partir da posição atual na matriz e na largura definida, garantindo que os dados sejam organizados de forma sequencial e adequada ao modelo.

```c
double calcular_erro_absoluto_medio(double *xTest, double *y_previsto,
                                    int tamanhoXTest, int largura, int altura) {
    double soma = 0.0;

    for (int i = 0; i < tamanhoXTest; i++) {
        soma += fabs(xTest[i + largura + altura] - y_previsto[i]);
    }

    return soma / tamanhoXTest;
}
```

4.6. Função Principal

A função principal do programa (main) é responsável por configurar os parâmetros iniciais, gerenciar a leitura e alocação de memória para os conjuntos de dados, e executar o algoritmo KNN para diferentes conjuntos de teste. Este processo é organizado em etapas bem definidas:

4.6.1. Leitura dos Argumentos e Configuração Inicial

No início, a função verifica se os argumentos necessários foram fornecidos corretamente. São esperados três parâmetros além do nome do programa: largura,

altura e k, que definem as dimensões da matriz e o número de vizinhos a serem considerados no algoritmo KNN.

Esses argumentos são convertidos para inteiros usando a função atoi, e, em seguida, uma verificação adicional assegura que os valores sejam positivos. Caso contrário, uma mensagem de erro é exibida e a execução do programa é encerrada.

```c
int main(int argc, char *argv[])
{
    if (argc != 4) //o nome do programa é um argumento
    {
        fprintf(stderr, "Formato de compilação: <programa> <largura> <altura> <k>\n");
        return EXIT_FAILURE;
    }

    double *xTrain, *xTest;
    int tamanhoTrain, tamanhoTest;
    int linhasTrain = 0;

    // Obtém os valores de largura, altura e k a partir dos argumentos
    // E converte para inteiros
    int largura = atoi(argv[1]);
    int altura = atoi(argv[2]);
    int k = atoi(argv[3]);

    // Verificação adicional para valores inválidos
    if (largura <= 0 || altura <= 0 || k <= 0)
    {
        fprintf(stderr, "Erro: largura, altura e k devem ser números inteiros positivos.\n");
        return EXIT_FAILURE;
    }
```

4.6.2. Definição e Leitura dos Conjuntos de Dados

Uma lista com os nomes dos arquivos que contêm os dados dos conjuntos de teste é definida. Esses arquivos representam diferentes tamanhos de conjuntos de teste, variando desde pequenas até grandes quantidades de dados, permitindo medir o desempenho do algoritmo em diversos cenários.

```c
const char *nomesArquivosTest[] = {
    "dados_xtest_10.txt",
    "dados_xtest_30.txt",
    "dados_xtest_50.txt",
    "dados_xtest_100.txt",
    "dados_xtest_1000.txt",
    "dados_xtest_100000.txt",
    "dados_xtest_1000000.txt",
    "dados_xtest_10000000.txt"
    };
```

O conjunto de treinamento é lido a partir do arquivo "dados_xtrain.txt". Utilizando as funções já descritas, os dados são processados para criar: a matriz de treino, através da função criar_matriz, e o vetor yTrain, utilizando a função criar_yTreino.

```
int totalArquivos = sizeof(nomesArquivosTest) / sizeof(nomesArquivosTest[0]);
// Leitura do conjunto de treino e a partir dele construção da matriz de treino e do y de treino
ler_arquivo("dados_xtrain.txt", &xTrain, &tamanhoTrain);
double **matrizTrain = criar_matriz(xTrain, tamanhoTrain, largura, altura, &linhasTrain);
double *yTrain = criar_yTreino(xTrain, linhasTrain, largura, altura);
```

4.6.3. Processamento dos Conjuntos de Teste

Para cada arquivo na lista de conjuntos de teste:

```
for (int i = 0; i < totalArquivos; i++) {
    double *xTest = NULL; // Inicializa xTest para cada arquivo
    int tamanhoTest = 0;

    // Lê o arquivo atual
    ler_arquivo(nomesArquivosTest[i], &xTest, &tamanhoTest);

    knn(nomesArquivosTest[i], matrizTrain, linhasTrain, yTrain, tamanhoTrain,
        xTest, tamanhoTest, largura, altura, k);


    free(xTest); // Libera a memória alocada para xTest, pois ele é reiniciado no for
}
```

1. Os dados são lidos e armazenados na variável xTest usando a função ler_arquivo.
2. O algoritmo KNN é executado para esse conjunto, realizando a fase de testes para cada um dos arquivos.
3. A memória alocada para o conjunto de teste (xTest) é liberada após cada iteração, garantindo uma gestão eficiente da memória.

4.6.4. Liberação de Memória

Após processar todos os conjuntos de teste, as estruturas de dados criadas para o conjunto de treino são liberadas: as linhas da matriz de treino (matrizTrain) e o vetor yTrain e os dados brutos do conjunto de treino (xTrain).

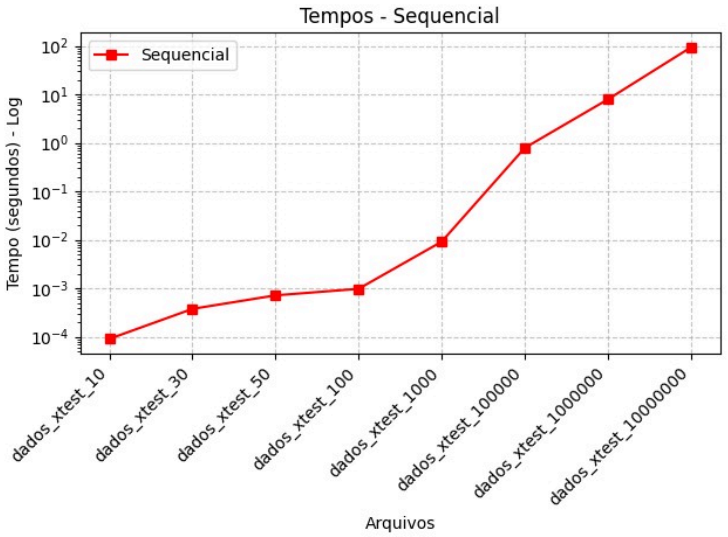
```
for (int i = 0; i < linhasTrain; i++) free(matrizTrain[i]);
free(matrizTrain);
free(yTrain);
free(xTrain);
return 0;
```

4.7. Validação de desempenho

Com esse código foram levantados os seguintes tempos relativos a cada conjunto de teste.

Tamanho do conjunto dos arquivos	Tempo gasto em segundos
10	0.000090700108557939 53s
30	0.000377200078219175 34s
50	0.000716800335794687 27s
100	0.000977799762040376 66s
1.000	0.009182199835777282 70s
100.000	0.789483200293034320 00s
1.000.000	7.906629700213670700 00s
10.000.000	93.712580299936235000 00s

Com isso foi plotado o gráfico abaixo, porém com o log do tempo, para possibilitar visualizar melhor a curva mesmo com a alta variedade de tempos.



5. PARALELIZAÇÃO DO CÓDIGO EM C

3.1 Ambiente - SO e Hardware

Utilizou-se o Sistema Operacional Windows de 64 bits (baseado em x64) para compilar e executar o código paralelo. Nesse ponto, surgiram alguns entraves, que foram selecionados como segue.

Para compilação do código paralelo, verificou-se a necessidade da flag -fopenmp, a fim de permitir o uso das diretivas do OpenMP no GCC.

Esta flag, no entanto, não era reconhecida pelo ambiente, que utilizava minGW32 para compilar programas em C. Dessa maneira, foi necessário instalar o minGW64, da MSYS2, software também voltado a compilar programas em C no Windows, e com compatibilidade para OpenMP.

Além da instalação, foram feitos os devidos ajustes nas variáveis de ambiente.

Para rodar os códigos (sequencial e todas as versões paralelizadas) foi utilizada uma máquina com processador 11th Gen Intel(R) Core(TM) i5-1135G7; frequência de 2.40Gz; 16GB de RAM instalada e 15.7GB de RAM utilizável.

Vale ressaltar que, para gerar cada um dos gráficos apresentados nas próximas seções, foram realizadas 10 execuções de cada arquivo em cada cenário analisado (através da passagem de parâmetros via linha de comando 10 vezes) e utilizada a média de tempo para cada arquivo.

3.2 Primeira tentativa: paralelizar todos os trechos passíveis de paralelização

Como tentativa inicial, optou-se por identificar todos os trechos passíveis de paralelização (onde a paralelização não gera conflitos ou erros de cálculo) e aplicar diretivas OpenMP. Conforme orientado pelo professor, sabe-se que essa não será a solução ideal. Ela foi feita apenas a caráter exploratório.

Inicialmente, não foi explicitada a quantidade de threads, de forma a utilizar o padrão do ambiente: 04 threads, conforme verificado nos testes.

3.2.1 Popular matriz

Na função “criar_matriz”, o laço externo da população da matriz foi paralelizado, com a diretiva “#pragma omp parallel for”. Desta forma, as linhas da matriz serão divididas entre as threads, e cada uma delas irá popular as linhas

atribuídas a ela (em todas as suas colunas, uma vez que o laço interno não foi paralelizado). Segue o código.

```
#pragma omp parallel for
for (int i = 0; i < *linhas_matriz; i++) {
        matriz[i] = (double *)malloc(largura * sizeof(double));
        for (int j = 0; j < largura; j++) {
                matriz[i][j] = dados[i + j];
        }
}
```

3.2.2 Calcular distâncias

Na função “calcular_distancias”, o laço externo foi paralelizado, com a diretiva “#pragma omp parallel for”. Desta forma, as linhas da matriz de Treino serão divididas entre as threads, e cada uma delas irá calcular a distância de suas linhas em relação à linha de Teste fixada (dada via parâmetro), considerando todas as colunas (o loop interno não é paralelizado). Segue o código.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
        double soma = 0.0;
        for (int j = 0; j < largura; j++) {
                double diff = matriz_treino[i][j] - linha_teste[j];
                soma += diff * diff;
        }
        distancias[i] = soma;
}
```

3.2.3 Encontrar k menores distâncias

Na função “encontrar_k_menores”, a sequência de buscas pelo 1º menor valor, 2º menor, …, k-ésimo menor valor não foi paralelizada, uma vez que o resultado de cada uma dessas buscas depende das anteriores.

A paralelização foi realizada, então, sobre o laço que itera pelos valores do vetor de distâncias. Para isso, cada thread cria variáveis privadas, para armazenar o menor valor entre os que analisou e seu índice. Foram utilizadas as diretivas “pragma omp parallel”, para definir o início da região paralelizada (criando as

variáveis privadas logo em seguida) e “pragma omp for” para dividir os elementos do vetor de distâncias entre as threads.

Finalmente, cada thread passa por uma região crítica, onde compara o seu menor_local (menor valor entre os que analisou) ao menor geral (menor valor já encontrado por qualquer uma das threads). Para garantir Exclusão Mútua, foi utilizada a diretiva “#pragma omp critical”.

Segue o código.

```
for (int i = 0; i < k; i++) {
        double menor = INFINITY;
        int indice_menor = -1;
        #pragma omp parallel
        {
                double local_menor = INFINITY;
                int local_indice_menor = -1;

                #pragma omp for
                for (int j = 0; j < n; j++) {
                        if (distancias[j] < local_menor) {
                                local_menor = distancias[j];
                                local_indice_menor = j;
                        }
                }

                // Sincroniza e encontra o menor valor global
                #pragma omp critical
                {
                        if (local_menor < menor) {
                                menor = local_menor;
                                indice_menor = local_indice_menor;
                        }
                }
        }
        indices[i] = indice_menor;
        distancias[indice_menor] = INFINITY; // Marcar como usada
```

```
}
```

3.2.4 Criar o YTeste

Na função “criar_YTest”, as linhas do YTeste a serem criadas foram divididas entre as threads (paralelizadas), com a diretiva “#pragma omp parallel for”. Destaca-se aqui que o laço paralelizado realiza chamadas a duas funções que também sofreram paralelização: “calcular_distancias” e “encontrar_k_menores”.

Ressalta-se que percebeu-se desempenho insatisfatório por conta dessa configuração, cenário que será detalhado nas próximas seções, juntamente com sua correção. Uma vez que, na seção atual, o objetivo é descrever a tentativa inicial, a fim de compará-la com os resultados finais, segue o código.

```
#pragma omp parallel for
for (int i = 0; i < linhas_teste; i++) {
        double *linha_teste = matriz_teste[i];
        double *distancias = (double *)malloc(linhas_treino * sizeof(double));
        calcular_distancias(matriz_treino, linhas_treino, largura, linha_teste, linhas_treino, distancias);
        int *indices = (int *)malloc(k * sizeof(int));
        encontrar_k_menores(distancias, linhas_treino, k, indices);
        double soma = 0.0;
        for (int j = 0; j < k; j++) {
                soma += y_treino[indices[j]];
        }
        y_teste[i] = soma / k;
        free(distancias);
        free(indices);
}
```

3.2.5 Calcular o erro absoluto médio

Na função “calcular_erro_absoluto_medio”, foram divididas entre as threads as linhas observadas de xTeste e YPrevisto.

Existe a variável compartilhada “soma”, inicializada com 0. Cada thread recebe uma cópia dela e incrementa os erros calculados nas linhas que avaliou.

Por fim, os valores de “soma” de todas as threads são somados, através da diretiva “#pragma omp parallel for reduction (+:soma)”. Segue o código.

```
double soma = 0.0;
#pragma omp parallel for reduction(+:soma)
for (int i = 0; i < tamanho; i++) {
        soma += fabs(xTest[i + largura + altura] - y_previsto[i]);
}
```

3.2.6 Liberação de memória

Na função “knn”, foi paralelizada a liberação da memória ocupada por cada linha da matriz de Teste. Segue o código:

```
#pragma omp parallel for num_threads(8)
for (int i = 0; i < linhasTest; i++) free(matrizTest[i]);
```

Na função “main”, foi paralelizada a liberação da memória ocupada por cada linha da matriz de Treino. Segue o código:

```
#pragma omp parallel for num_threads(8)
for (int i = 0; i < linhasTrain; i++) free(matrizTrain[i]);
```

3.2.7 População do YTreino

Na função “criar_yTreino”, as linhas a serem populadas foram divididas entre as threads (paralelizadas), com a diretiva “#pragma omp parallel”. Segue o código.

```
#pragma omp parallel
for (int i = 0; i < linhasTrain; i++) {
        yTrain[i] = xTrain[i + largura + altura - 1];
};
```

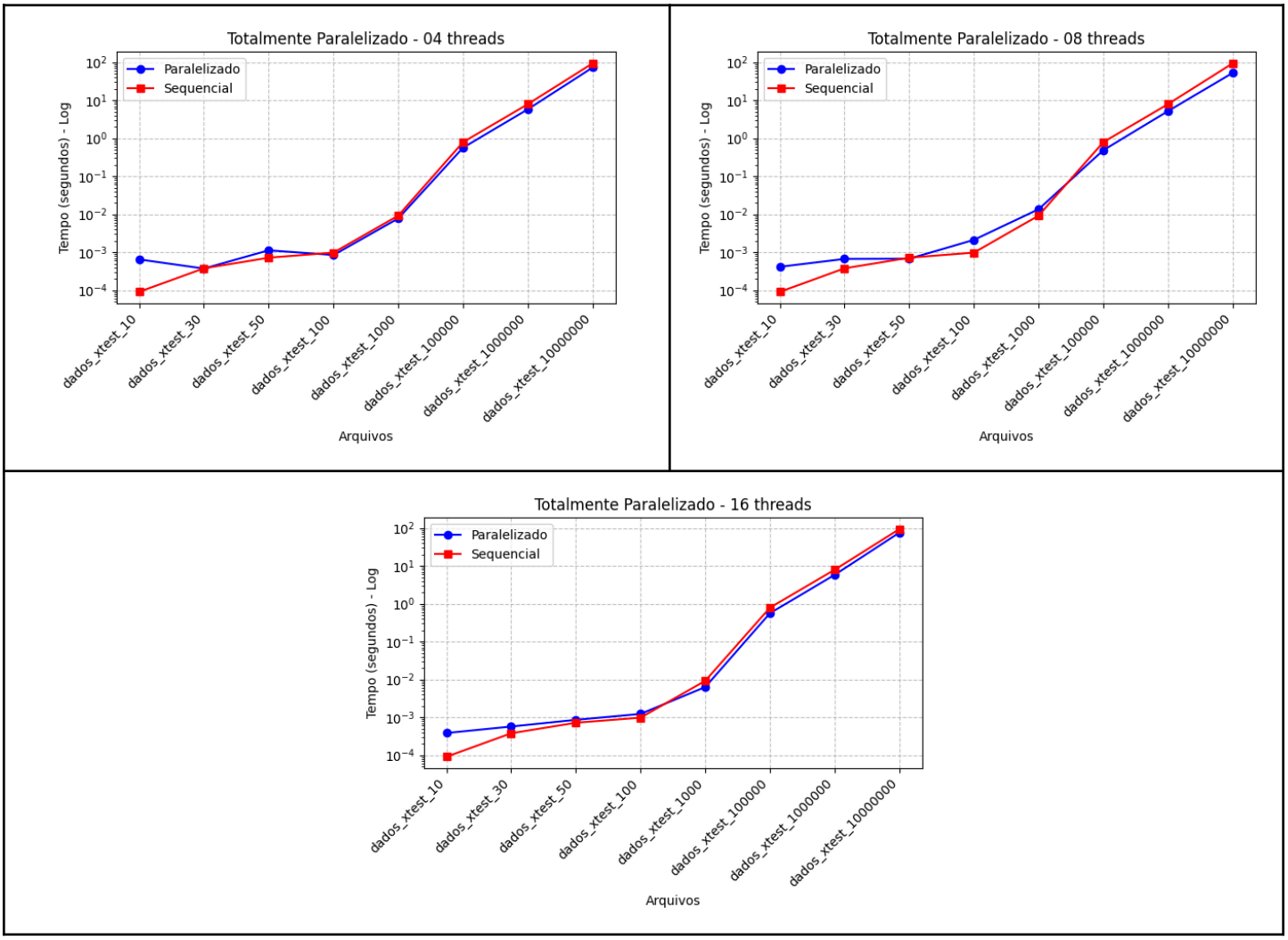
3.2.8 Variação da quantidade de threads e resultados parciais

Conforme citado anteriormente, para executar o código com 04 threads, não são necessárias definições explícitas, por ser o padrão do ambiente. Foram testadas, também, variações nas quantidades de threads (adicionando a cláusula “num_threads()” às diretivas).

Diante do código com paralelização em todos os trechos plausíveis, não foram obtidos ganhos significativos (com nenhuma quantidade de threads), conforme esperado. Não há vantagem em utilizar este código, uma vez que o tempo

é sempre praticamente igual, com exceção do menor arquivo, em que o código paralelizado é mais demorado (devido ao overhead de criação das threads, que não é compensado devido à quantidade relativamente pequena de cálculos).

Abaixo, são apresentados elementos visuais comparativos, para variadas quantidades de threads, do código com paralelização em todos os trechos plausíveis.



A seguir, apresenta-se o tempo de execução do código atual para cada quantidade de threads testada. Percebe-se que o cenário mais rápido, aqui, ocorre com 08 threads.

	Tempo (s) para Arquivo 100.000	Tempo (s) para Arquivo 1.000.000	Tempo (s) para Arquivo 10.000.000
04 threads	0.56	5.79	72.70
08 threads	0.48	5.17	52.55
16 threads	0.56	5.80	75.55

A tabela seguinte apresenta o speedup no tempo de execução do código atual para cada quantidade de threads testada, em relação ao código sequencial. A ideia é verificar se, para 04 threads, há ganho próximo a 4x, por exemplo. Sua análise reforça a ideia de que o código atual não gera ganhos expressivos. Entende-se que a máquina não é dedicada apenas ao programa, porém, ainda assim, o speedup está consideravelmente distante do cenário ideal.

O speedup mais próximo do ideal ocorre com 04 threads, apesar de insatisfatório.

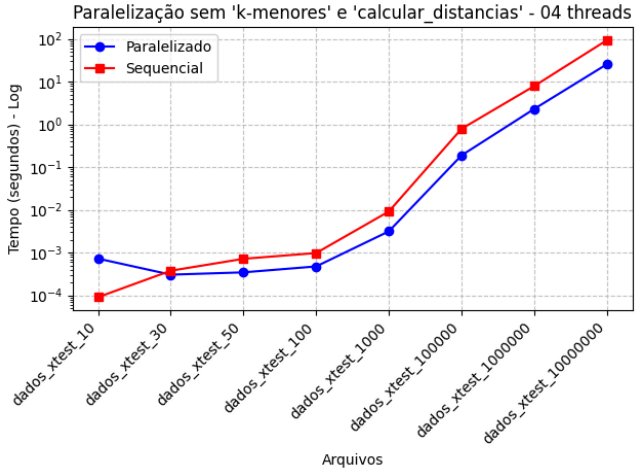
	Speedup para Arquivo 100.000	Speedup para Arquivo 1.000.000	Speedup para Arquivo 10.000.000
04 threads	1.41x (-2.59x do ideal)	1.37x (-2.63x do ideal)	1.29x (-2.71x do ideal)
08 threads	1.65x (-6.35x do ideal)	1.53x (-6.47x do ideal)	1.78x (-6.22x do ideal)
16 threads	1.41x (-14.59x do ideal)	1.36x (-14.64x do ideal)	1.24x (-14.76x do ideal)

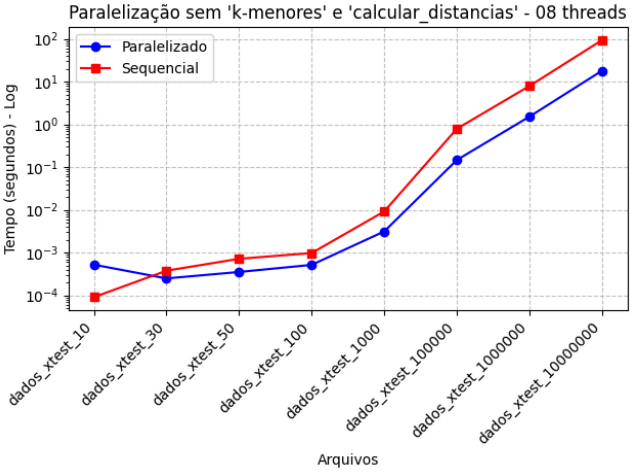
3.3 Segunda tentativa

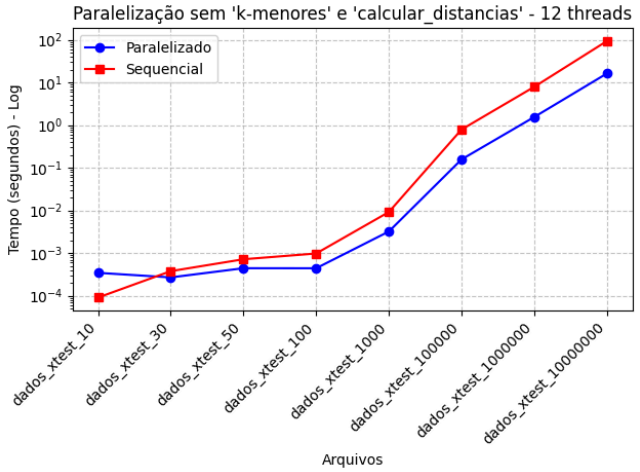
Conforme citado anteriormente, a função “criar_YTeste” foi definida com um laço paralelizado, que chama duas funções nas quais também há paralelização.

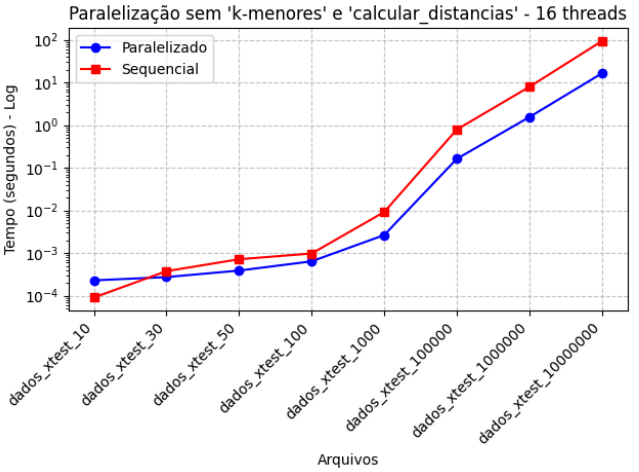
A partir desta observação, a próxima tentativa foi retirar a paralelização das funções chamadas no laço citado: “calcular_distancias” e “encontrar_k_menores”. O laço em si, de “criar_YTeste” permanece sendo paralelizado.

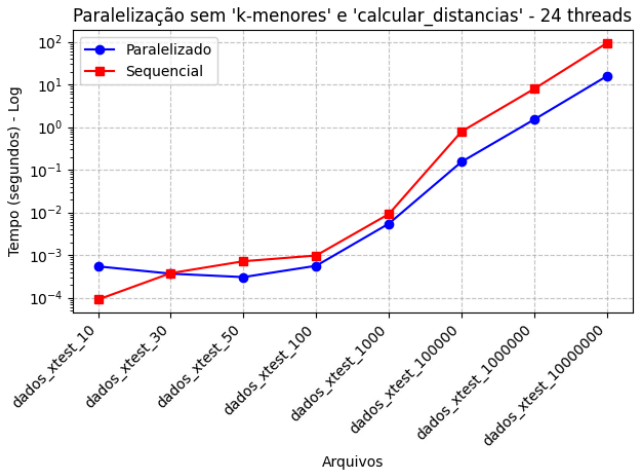
Seguem gráficos para comparação da execução do novo código para variadas quantidades de threads.

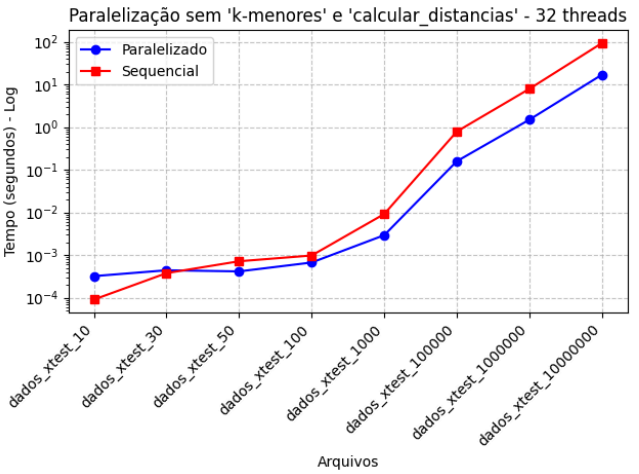

Paralelização sem 'k-menores' e 'calcular_distancias' - 04 threads
Paralelizado
Sequencial
Tempo (segundos) - Log
dados_xtest_10
dados_xtest_30
dados_xtest_50
dados_xtest_100
dados_xtest_1000
dados_xtest_100000
dados_xtest_1000000
dados_xtest_10000000
Arquivos


Paralelização sem 'k-menores' e 'calcular_distancias' - 08 threads
Paralelizado
Sequencial
Tempo (segundos) - Log
dados_xtest_10
dados_xtest_30
dados_xtest_50
dados_xtest_100
dados_xtest_1000
dados_xtest_100000
dados_xtest_1000000
dados_xtest_10000000
Arquivos


Paralelização sem 'k-menores' e 'calcular_distancias' - 12 threads
Paralelizado
Sequencial
Tempo (segundos) - Log
dados_xtest_10
dados_xtest_30
dados_xtest_50
dados_xtest_100
dados_xtest_1000
dados_xtest_100000
dados_xtest_1000000
dados_xtest_10000000
Arquivos


Paralelização sem 'k-menores' e 'calcular_distancias' - 16 threads
Paralelizado
Sequencial
Tempo (segundos) - Log
dados_xtest_10
dados_xtest_30
dados_xtest_50
dados_xtest_100
dados_xtest_1000
dados_xtest_100000
dados_xtest_1000000
dados_xtest_10000000
Arquivos


Paralelização sem 'k-menores' e 'calcular_distancias' - 24 threads
Paralelizado
Sequencial
Tempo (segundos) - Log
dados_xtest_10
dados_xtest_30
dados_xtest_50
dados_xtest_100
dados_xtest_1000
dados_xtest_100000
dados_xtest_1000000
dados_xtest_10000000
Arquivos


Paralelização sem 'k-menores' e 'calcular_distancias' - 32 threads
Paralelizado
Sequencial
Tempo (segundos) - Log
dados_xtest_10
dados_xtest_30
dados_xtest_50
dados_xtest_100
dados_xtest_1000
dados_xtest_100000
dados_xtest_1000000
dados_xtest_10000000
Arquivos

Além de gráficos, foi criada uma tabela comparativa dos tempos de execução do código atual para diferentes quantidades de threads. Com ela, fica claro que o cenário testado mais rápido, para esse código, ocorre com 24 threads.

	Tempo (s) para Arquivo 100.000	Tempo (s) para Arquivo 1.000.000	Tempo (s) para Arquivo 10.000.000
04 threads	0.19	2.30	25.63
08 threads	0.15	1.52	18.02
12 threads	0.16	1.55	16.27
16 threads	0.16	1.55	16.44
24 threads	0.15	1.51	15.76
32 threads	0.16	1.50	16.98

A tabela seguinte apresenta o speedup no tempo de execução do código atual para cada quantidade de threads testada, em relação ao código sequencial. O speedup mais próximo do ideal é satisfatório (consideravelmente próximo) e ocorre, neste código, com 04 threads.

	Tempo (s) para Arquivo 100.000	Tempo (s) para Arquivo 1.000.000	Tempo (s) para Arquivo 10.000.000
04 threads	4.16x (+0.16x do ideal)	3.44x (-0.56x do ideal)	3.66x (-0.34 do ideal)
08 threads	5.27x (-2.73x do ideal)	5.20x (-2.80x do ideal)	5.20x (-2.80x do ideal)
12 threads	4.94x (-7.06x do ideal)	5.10x (-6.90x do ideal)	5.76x (-6.24x do ideal)
16 threads	4.94x (-11.06x do ideal)	5.10x (-10.9x do ideal)	5.70x (-10.3x do ideal)
24 threads	5.27x (-18.73x do ideal)	5.24x (-18.76x do ideal)	5.95x (-18.05x do ideal)
32 threads	4.94x (-27.06x do ideal)	5.27 (-26.73x do ideal)	5.52x (-26.48x do ideal)

A comparação dos gráficos e tabelas desta sessão com os da sessão anterior permite verificar, ainda, que a paralelização no formato em que ocorre na tentativa

atual (paralelização de tudo o que é plausível, com exceção do cálculo das distâncias e do encontro dos k menores valores) é mais vantajosa do que a paralelização da seção anterior.

Para comprovar, segue tabela comparativa dos melhores tempos para o código atual e para o código da seção anterior. Ela evidencia ganho de 70%, 71% e 70%, respectivamente, em relação ao código da seção 3.2.

	Tempo (s) para Arquivo 100.000	Tempo (s) para Arquivo 1.000.000	Tempo (s) para Arquivo 10.000.000
Código 3.2 - 08 threads	0.48	5.17	52.55
Código 3.3 - 24 threads	0.15	1.51	15.76

Além disso, segue tabela comparativa dos melhores speedups para o código atual e para o código da seção anterior. Ela evidencia que o código atual atinge speedup mais próximo do ideal do que o código da seção 3.2.

Código 3.2 - 04 threads	1.41x (-2.59x do ideal)	1.37x (-2.63x do ideal)	1.29x (-2.71x do ideal)
Código 3.3 - 04 threads	4.16x (+0.16x do ideal)	3.44x (-0.56x do ideal)	3.66x (-0.34 do ideal)

3.4 Terceira tentativa

Para a terceira tentativa, calculou-se a fração de tempo do KNN ocupada por cada uma das principais tarefas, no código sequencial. Visa-se, dessa forma, descobrir a(s) parte(s) com maior participação no tempo, paralelizando apenas estes trechos e evitando overheads com pouco impacto positivo.

Arquivo	Tempo (s) para popular Matriz Teste	Tempo (s) para Calcular Distâncias	Tempo (s) para Encontrar K Menores	Tempo (s) para Calcular Valores YTeste
10	0.3%	14.3%	4.0%	0.2%
30	0.5%	37.4%	11.0%	0.4%
50	0.5%	41.8%	12.6%	0.5%
100	0.4%	50.3%	18.6%	0.5%

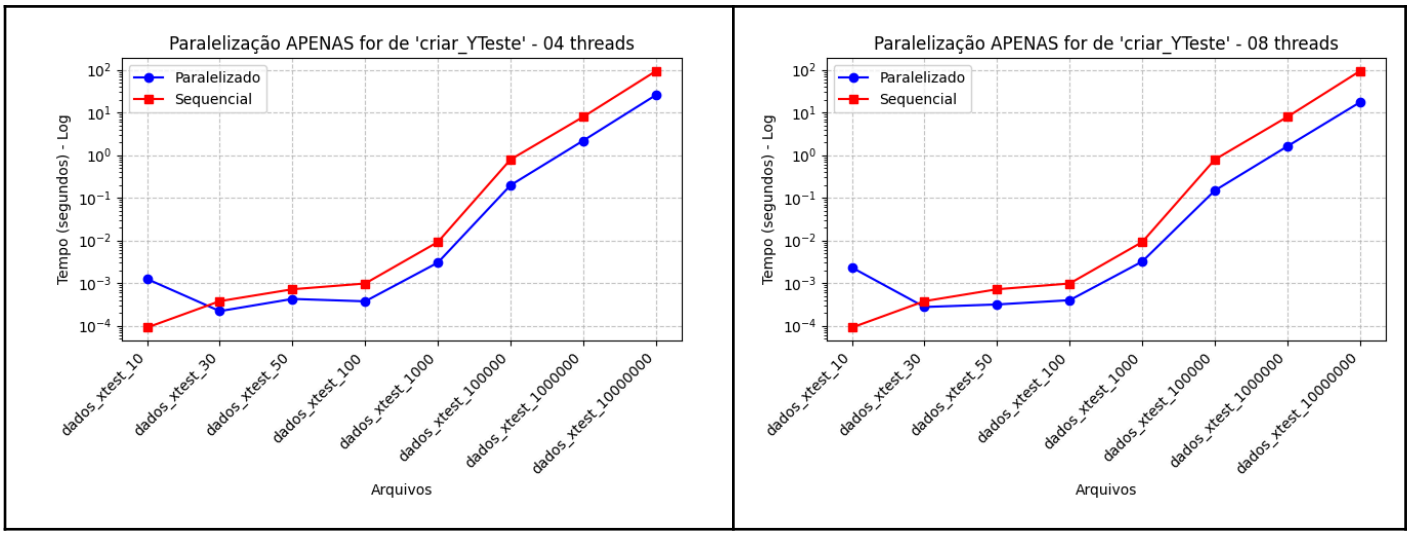
1000	0.6%	62.4%	19.3%	0.6%
100000	0.6%	70.2%	24.0%	0.6%
1000000	0.7%	69.2%	25.1%	0.6%
10000000	0.9%	69.1%	24.9%	0.7%

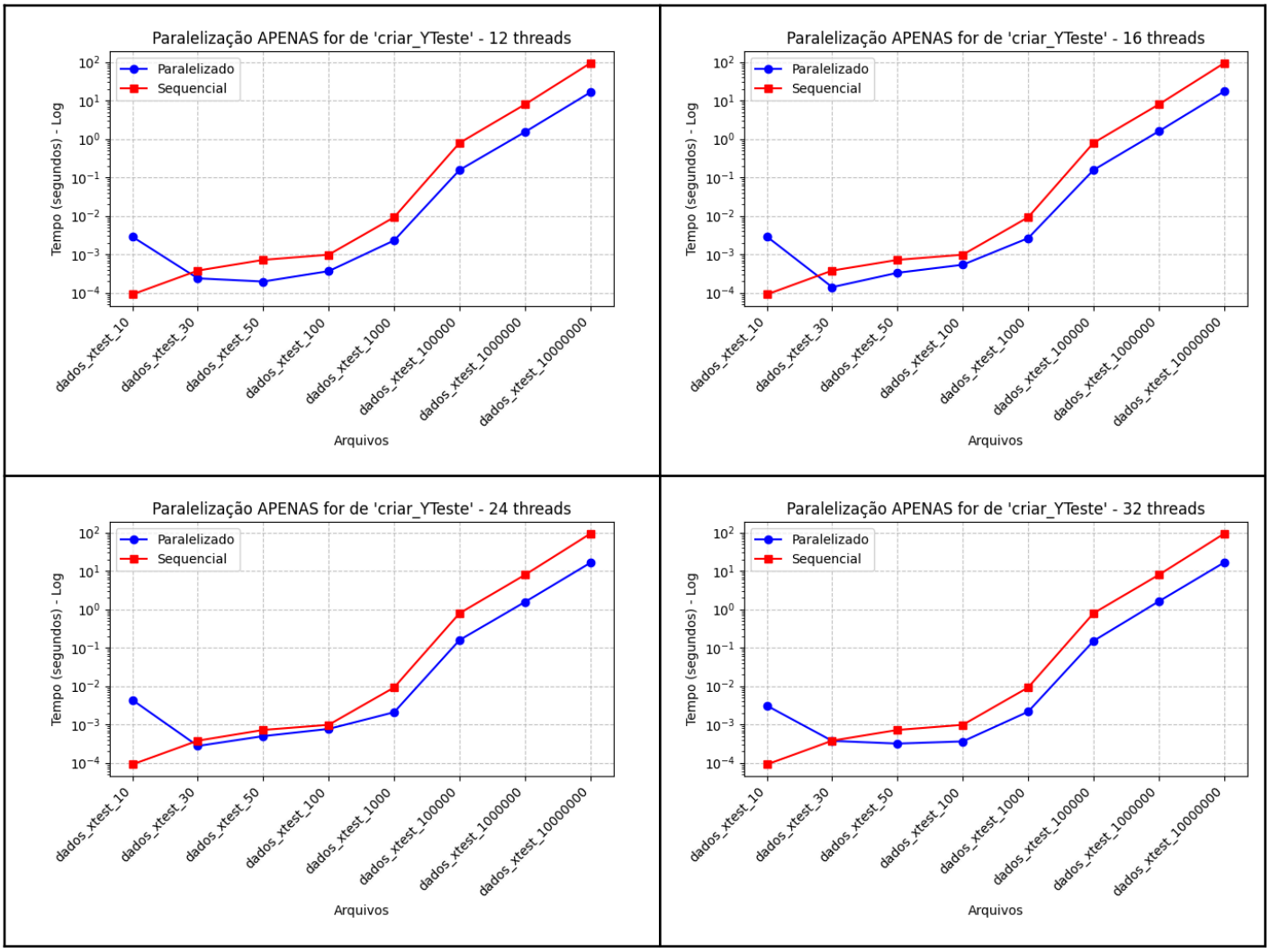
Fica evidente que o cálculo das distâncias é a tarefa mais custosa (em tempo) do KNN sequencial, seguida pela busca dos K menores valores.

Vale ressaltar que, para arquivos pequenos, essas tarefas também são as mais custosas proporcionalmente, porém não representam fração tão elevada do tempo total do KNN sequencial. Atribui-se essa situação à quantidade relativamente pequena de cálculos e buscas por menores valores, dado o tamanho do arquivo, o que faz com que elementos não analisados na tabela (como atribuição de valores a variáveis) ocupem proporções mais significativas do tempo total do que no caso dos arquivos maiores.

A função “criar_YTest” possui um laço que chama ambas as funções mencionadas (“calcular_distancias” e “encontrar_k_menores”), tarefa cuja execução ocupa a maior parte do laço. Esta subseção analisará o código que paraleliza exclusivamente este laço.

Seguem os gráficos dos tempos de execução do código desta subseção para variadas quantidades de threads.





Além de gráficos, foi criada uma tabela comparativa dos tempos de execução do código atual para diferentes quantidades de threads. Com ela, fica claro que o cenário testado mais rápido, para esse código, ocorre com 12 threads.

	Tempo (s) para Arquivo 100.000	Tempo (s) para Arquivo 1.000.000	Tempo (s) para Arquivo 10.000.000
04 threads	0.2	2.2	25.27
08 threads	0.15	1.61	17.36
12 threads	0.16	1.52	16.27
16 threads	0.16	1.58	17.24
24 threads	0.16	1.55	16.3
32 threads	0.15	1.61	16.56

O speedup é relativamente satisfatório para os arquivos de 100.000 linhas e de 1.000.000 de linhas, porém insatisfatório para o arquivo de 10.000.000 de linhas (maior arquivo). Dadas as 12 threads, esperavam-se tempos de 0.07s, 0.66s e 7.81s para estes arquivos. Desta forma, o código atual exigiu apenas 0.09s a mais para 100.000 linhas e 0.86s a mais para 1.000.000 linhas (valores que podem ser atribuídos à não exclusividade do computador para executar o código). Já para o maior arquivo, exigiu 8.46s a mais do que o ideal.

Comparativamente à solução da seção 3.3, o código atual apresenta-se levemente inferior.

	Tempo (s) para Arquivo 100.000	Tempo (s) para Arquivo 1.000.000	Tempo (s) para Arquivo 10.000.000
Código 3.3 - 24 threads	0.15	1.51	15.76
Código 3.4 - 12 threads	0.16	1.52	16.27

6. CONCLUSÕES

A partir dos testes e análises apresentados na seção anterior, conclui-se que o senso comum de que deve-se paralelizar o máximo de tarefas possível para obter o maior ganho de desempenho é, de fato, equivocado. Isto pode ser observado pela tentativa 01 (seção 3.2), que, apesar de utilizar código com máxima paralelização plausível (de forma a não prejudicar os valores resultantes ou gerar conflitos), obteve o pior desempenho entre os testes.

As tentativas mais bem sucedidas foram 02 e 03, com resultados aproximadamente iguais em tempo de execução (apesar de melhor speedup na solução 02).

A segunda tentativa (seção 3.3) paraleliza tudo o que pode ser paralelizado, com exceção dos laços dentro das funções “calcular_distancias” e “encontrar_k_menores” (portanto, também paraleliza a criação da matriz de Teste). Vale destacar que, ainda assim, o laço de “criar_YTeste”, responsável principalmente por chamar as funções “calcular_distancias” e “encontrar_k_menores”, é paralelizado.

Já a terceira tentativa (seção 3.4) paraleliza exclusivamente o laço de “criar_YTeste”, responsável principalmente por chamar as funções “calcular_distancias” e “encontrar_k_menores”. O laço de dentro destas funções, assim como a criação da matriz de Teste, não foram paralelizados nesse caso.

A partir deste resumo e da análise de participação de cada tarefa no tempo sequencial, fica claro que faz sentido que as tentativas 02 e 03 sejam as melhores, sendo a 02 levemente mais rápida.

Isso ocorre uma vez que possuem overhead de criação de threads apenas nos pontos onde elas de fato terão impacto expressivo (tarefas que, quando não paralelizadas, ocupam grande proporção do tempo total).

A tentativa 02 é levemente mais rápida porque paraleliza a criação da matriz de Teste, que, no maior arquivo, representa apenas 01% do tempo total do KNN sequencial.

Conclui-se, ainda, que a tentativa 01 é insatisfatória por duas razões principais:

- Gera gasto de tempo para criação e gerenciamento de threads em tarefas que não se beneficiarão expressivamente delas, o que gera mais prejuízos do que benefícios;
- Paraleliza o laço que chama “calcular_distancias” e “encontrar_k_menores” e, também, os laços dentro destas funções, o que gerou ineficiência.

7. ANEXOS

Código Python para Gráficos. Disponível em: https://colab.research.google.com/drive/15huBo5ajrx3F92XcmFFJHuFvSCEBm-uC?usp=sharing