ACH 2055 - OAC II
4º semestre de 2023.

EACH/USP

RELATÓRIO SOBRE A IMPLEMENTAÇÃO DO ALGORITMO KNN UTILIZANDO PARALELISMO EM C

NOME(s) alunos:

Eloisa Antero Guisse - 13781924
Rafael Varago de Castro - 13837428

PERÍODO NOTURNO

Disciplina: Organização e Arquitetura de Computadores II
Profº Clodoaldo Aparecido de Moraes Lima

4º semestre/2023

1 Estruturas para o cálculo dos pontos mais próximos

Para armazenar a informação dos pontos mais próximos, a estrutura escolhida foi uma lista ligada. A decisão foi tomada considerando a maior facilidade em realizar inserções, exclusões, e ordenação no geral.

Foi criada uma estrutura `Node`, que armazena as seguintes informações sobre um ponto: a distância dele ao ponto de `xtest` sendo calculado, a sua classe (0 ou 1) e um ponteiro para o próximo ponto mais perto do de `xtest`.

```
typedef struct aux {
    double distancia;
    double classe;
    struct aux * prox;
} Node;
```

Então, foi a criada a estrutura que armazenará a lista de pontos mais próximos. Ela contém o ponteiro para o primeiro ponto da lista (o mais distante dentre os K mais próximos), um inteiro que indica o seu tamanho atual e outro que indica o seu tamanho máximo (no caso, o valor de K escolhido).

```
typedef struct {
    Node * primeiro;
    int tam_atual;
    int tam_max;
} Pontos_Proximos;
```

Foi definida uma função para inserir um ponto na lista já na posição correta baseada em sua distância do ponto de `xtest`.

```
void insert(Pontos_Proximos * lista, double distancia, double classe, int index) {
    lista->tam_atual++;

    // Cria novo elemento
    Node * new_node = (Node *)malloc(sizeof(Node));
    new_node->distancia = distancia;
    new_node->classe = classe;
    new_node->prox = NULL;
```

```
    // Insere
    if (lista->primeiro == NULL) {    // Caso a lista ainda esteja vazia
        lista->primeiro = new_node;
    } else {                          // Caso a lista já tenha elementos
        Node * atual = lista->primeiro;
        Node * anterior = NULL;

        // Itera sobre a lista até achar a posição correta
        while (atual != NULL && atual->distancia > distancia) {
            anterior = atual;
            atual = atual->prox;
        }
        // Caso o novo elemento deve ser inserido no início da lista
        if (anterior == NULL) {
            new_node->prox = lista->primeiro;
            lista->primeiro = new_node;
        // Caso o novo elemento deve ser inserido no meio ou no final
        } else {
            new_node->prox = atual;
            anterior->prox = new_node;
        }
    }

    // Checa se a lista excedeu seu tamanho máximo
    if (lista->tam_atual > lista->tam_max) {
        Node * to_remove = lista->primeiro;
        lista->primeiro = lista->primeiro->prox;
        free(to_remove);
        lista->tam_atual--;
    }
}
```

2 Implementação do KNN

Para o cálculo do KNN, foi implementada uma função que calcula a distância entre dois pontos, recebendo como parâmetros seus respectivos endereços na matriz e um inteiro representando o tamanho da linha. O valor de retorno é a distância euclidiana quadrática entre os dois pontos.

```
double calc_dist(double * line_train, double * line_test, int len_line) {
    double x, y, aux, dist = 0.0;
    for (int i = 0; i < len_line; i++) {
        x = line_train[i];
        y = line_test[i];
        aux = x - y;
        dist += aux * aux;
    }
    return dist;
}
```

Também foi implementada uma função que calcula a classificação de um ponto dada a lista dos pontos mais próximos dele. Ela conta o número de instâncias da classe 0 e da classe 1, e devolve a classe que houver o maior número de aparições. Caso haja um empate, ela devolve a classe do ponto com a menor distância.

```
double calc_classe(Pontos_Proximos * lista) {
    int count_zero = 0, count_one = 0;

    Node * atual = lista->primeiro;
    Node * ant;
    while (atual != NULL) {
        // Compara a classe com 0.5 para evitar erros de precisão
        if (atual->classe < 0.5) count_zero++;
        else count_one++;
        ant = atual;
        atual = atual->prox;
    }

    // Se o contador de ambos for igual, retorna a classe do número
    // na última posição da lista
    if (count_zero == count_one) return ant->classe;
```

```
    // Caso contrário, retorna a classe com maior número de incidência
    else return (count_zero > count_one) ? 0.0 : 1.0;
}
```

Finalmente, foi implementada a KNN. Ela itera sobre `xtest`, e a cada iteração, itera também sobre `xtrain`, calculando a distância entre o ponto de `xtest` para todos os pontos de `xtrain`. Caso a distância seja menor do que a do primeiro ponto da lista dos K mais próximos (sendo a lista decrescente, ou seja, o ponto mais próximo está na última posição), ele faz a inserção através da função `insert`. Após a lista ser construída, a classe é calculada através da função `calc_classe`, e o valor é armazenado no vetor de `ytest`.

```
void knn_no_threads(int k, double **xtrain, double **ytrain, double **xtest,
                    double *ytest, int n_xtrain, int n_xtest, int num_cols) {
    // Calcula para cada ponto de xtest
    for (int i = 0; i < n_xtest; i++) {
        Pontos_Proximos *lista = init(k);

        // Calcula distância de cada ponto de xtrain
        for (int j = 0; j < n_xtrain; j++) {
            double dist = calc_dist(xtrain[j], xtest[i], num_cols);
            if (lista->tam_atual < k || dist < lista->primeiro->distancia) {
                // Se o ponto for mais próximo que o primeiro da lista, insere
                insert(lista, dist, ytrain[j][0]);
            }
        }

        // Calcula classe
        double classe = calc_classe(lista);

        // Armazena em ytest
        ytest[i] = classe;

        // Libera memória
        free(lista);
    }
}
```

Também foi implementada uma segunda versão da KNN, que utiliza threads para tornar o calculo da classe de cada ponto de `xtest` paralelizado e aumentar a performance.

Esta foi a seção escolhida para a paralelização por não apresentar necessidade de implementação de regiões críticas para lidar com acesso concorrente; cada *thread* tem sua própria lista de pontos próximos e nenhuma acessa o mesmo endereço de memória em nenhum momento.

```
void knn_threads(int k, double **xtrain, double **ytrain, double **xtest,
    double *ytest, int n_xtrain, int n_xtest, int num_cols) {
    // Calcula para cada ponto de xtest (paralelizado)
    #pragma omp parallel for
    for (int i = 0; i < n_xtest; i++) {
        // Inicializa lista de pontos próximos
        Pontos_Proximos *lista = init(k);

        // Calcula distância de cada ponto de xtrain
        for (int j = 0; j < n_xtrain; j++) {
            double dist = calc_dist(xtrain[j], xtest[i], num_cols);
            if (lista->tam_atual < k || dist < lista->primeiro->distancia) {
                // Se o ponto for mais próximo que o primeiro da lista, insere
                    insert(lista, dist, ytrain[j][0]);
            }
        }

        // Calcula classe
        double classe = calc_classe(lista);

        // Armazena em ytest
        ytest[i] = classe;

        // Libera memória alocada para a lista de pontos próximos
        free(lista);
    }
}
```

3 Main

A função `main` do código irá receber o input do valor de K pelo usuário e criar um vetor com todos os tamanhos possíveis de arquivo (referente ao número de linhas), para rodar o KNN para todos eles.

```
int main() {
    // Armazenando input de k
    int k;
    printf("Insira o valor de k desejado: ");
    scanf("%d", &k);
    printf("\n");

    // Tamanhos de arquivos
    int sizes[] = {100, 500, 1000, 5000, 10000, 20000,
                   50000, 100000, 200000, 500000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
```

Após isso, é feito um loop `for` para cada arquivo. No loop, os arquivos são lidos e transformados e matrizes, e as duas implementações do KNN são rodadas. Após isso, é exibido o tempo de execução de cada implementação em milissegundos. Finalmente, o vetor `ytest` é escrito num arquivo.

```
// Rodando KNN sem threads
start = clock();

knn_no_threads(k, xtrain, ytrain, xtest, ytest, n_xtrain, n_xtest, num_cols);

end = clock();

cpu_time_used = (((double) (end - start)) / CLOCKS_PER_SEC) * 1000;
printf("O programa com %d linhas levou %f ms para ser executado SEM THREADS.\n",
                                                    sizes[s], cpu_time_used);

// Rodando KNN com threads
start = clock();

knn_threads(k, xtrain, ytrain, xtest, ytest, n_xtrain, n_xtest, num_cols);
```

```
end = clock();

cpu_time_used = (((double) (end - start)) / CLOCKS_PER_SEC) * 1000;
printf("O programa com %d linhas levou %f ms para ser executado COM THREADS.\n",
                                                        sizes[s], cpu_time_used);
```

4 Performance

O programa foi testado com números variados de *threads*. Por limitações de *hardware*, o maior número testado foi seis. O seguinte gráfico foi gerado relacionando o tempo de execução (em milissegundos) com o tamanho do arquivo processado para cada número de *threads*. O valor de K usado nos testes foi 5.

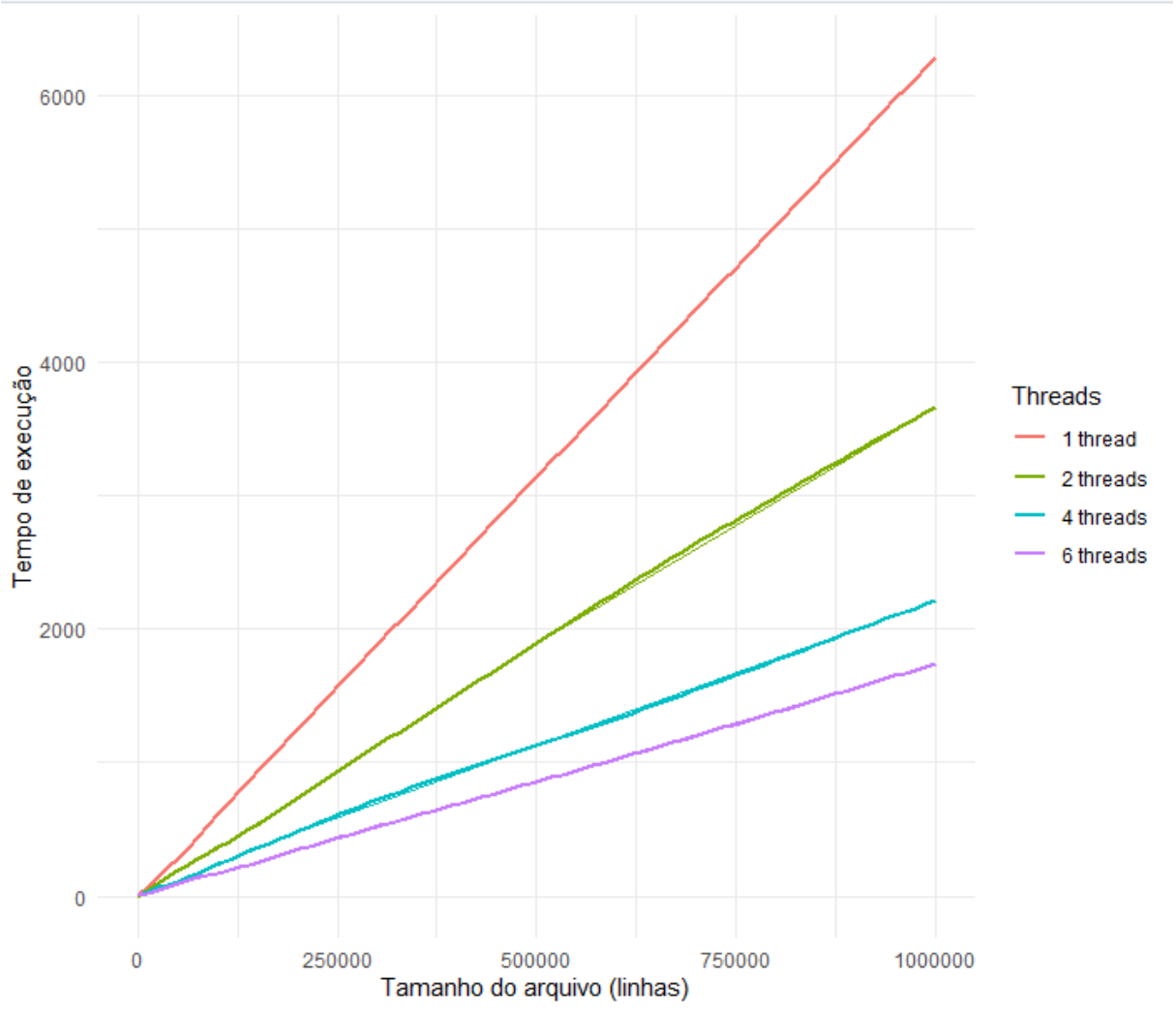

Figura 1 – Gráfico com o tempo de execução para cada número de threads em relação ao tamanho do arquivo.

O gráfico claramente demonstra uma relação quase que completamente proporcional entre o número de *threads* e o tempo de execução; ao dobrar-se o número de *threads*, o tempo de execução cai pela metade.