UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

Organização e Arquitetura de Computadores 2

ANDREAS HUKUHARA CHRISTE - 13673110

NICOLAS PEREIRA RISSO VIEIRA - 13672262

PEDRO PALAZZI DE SOUZA - 13748012

Exercício Programa 2

São Paulo

2023

Introdução

O exercício programa 2 tem como objetivo a realização de um knn, em C, no qual são lidos os arquivos “xtrain.txt”, “ytrain.txt” e “xtest.txt” e será gerado uma saída com a devida classificação no arquivo “ytest.txt”. Além disso, neste projeto usamos uma abordagem utilizando threads para identificar o ganho do tempo de execução nas comparações entre os elementos de xtest e todos os pontos de train. Para finalizar, foram analisados os resultados obtidos e representados graficamente.

Como Compilar

Primeiramente entre no diretório ‘KNN_C’, logo após, há duas possibilidades para compilar, sendo:

1- compilar e executar com make file:

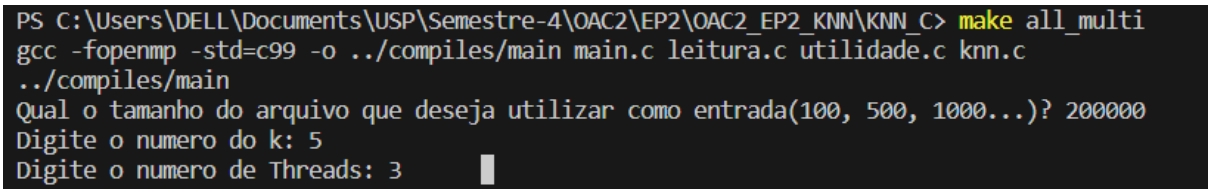
make all_multi

2 - compilar e executar manualmente:

gcc -fopenmp -std=c99 -o ../compiles/main main.c leitura.c utilidade.c knn.c

../compiles/main

Para facilitar o entendimento segue a ilustração que mostra o código sendo executado:



Uma vez executado, primeiramente lhe será questionado qual o tamanho da entrada que deseja utilizar (indo da entrada de 100 até a de 500.000), então, o programa lhe perguntará o valor que deseja utilizar na variável k do KNN. Logo em seguida será possível passar quantas threads deseja que sejam utilizadas na execução.

Como saída para esse caso lhe será mostrado o resultado do KNN, o arquivo” ytest” será salvo na pasta output e lhe será fornecido o tempo de execução.

Caso o número de threads seja menor ou igual a 0 (zero), o programa assumirá que você quer fazer uma análise de múltiplas threads diferentes, então, lhe perguntará qual será a thread máxima que você deseja executar o programa e o passo que aumentará as threads, iniciando em 1 (um) por padrão.

Como saída, lhe será fornecido apenas o tempo de execução do programa para cada N threads. Já que a ideia dessa execução é facilitar a coleta de dados do tempo necessário para o programa rodar.

Observações: *A entrada de 1.000.000 (Um Milhão) não foi adicionada ao programa pela falta do arquivo ytrain1000000.txt, tornando, assim, impossível executar a sua análise.*

Main.C

Primeiramente carrega-se os conteúdo dos arquivos “xtrain.txt”, “xtest.txt” e “ytrain.txt e seus respectivos números de linhas e de colunas (Arquivos utilizados dependem da entrada do usuário), após isso, chama a função ”KNN”, caso seja uma execução única, ou “chamaknn”, para o caso de análise de diversas threads diferentes ao mesmo tempo, para iniciar o processo de análise de cada um dos vetores presentes em “xtest.txt”. Após isso , chama-se a função “saída” para salvar o conteúdo da saída em “ytest.txt”.

Vale ressaltar que nesta classe só é possível utilizar os arquivos já predeterminados pelo código, ou seja, caso o usuário queira adicionar um novo arquivo para execução, de maneira simples, ele deverá nomear seu arquivo como “xtrainXXX.txt” ou “ytrainXXX.txt” em que XXX representa o número que identifica esse arquivo. Após isso, ele deverá adicioná-lo como um dos arquivos aceitos no 14 do main.

Abaixo está um exemplo de como está a estrutura de nomeação do código:

```
char xtrainFileName[] = "../data/xtrainXXX.txt";
char ytrainFileName[] = "../data/ytrainXXX.txt";
```

Struct.H

Criamos estruturas para auxiliar no armazenamento dos dados. A primeira estrutura ‘Ponto’ armazena as coordenadas do ponto a classe e a linha correspondente (utilizada para debugar o código), já a DistanciaPonto armazena a distância a classe e a linha correspondente, como ilustrado na figura a seguir:

```
struct Ponto {
    float *x;  //x, y, z, w, u, v, s, t
    float classe;
    int id;
};
typedef struct Ponto Ponto;

struct DistanciaPonto
{
    float distancia;
    float classe;
    int id;
};
typedef struct DistanciaPonto DistanciaPonto;
```

KNN.C

Abaixo serão explicadas algumas das funções utilizadas pelo código para gerar o knn. O knn.h é apenas o header deste código, tendo declarado suas funções e importações.

DistanciaPonto Distancia(Ponto ponto1, Ponto ponto2)

Tal função calcula a distância euclidiana em relação a dois pontos passados e já preenche a estrutura DistanciaPonto com a distancia calculada, a linha correspondente ao xtrain passado e a classe correspondente e retorna essa estrutura da DistanciaPonto temporária criada.

float verificaClasse(DistanciaPonto distanciasPontos[],int k)

Tal função tem como principal objetivo identificar a classe do ponto que está sendo testado, para isso ela recebe as distâncias já ordenadas e o valor de k que será avaliado, caso o valor de k seja ímpar ela retorna a classe mais presente dentre as k menores distâncias. Para os k pares é retornada a classe cujo a distância é a menor (primeira menor distância).

double KNN(Ponto pontos[], Ponto testes[], int k, int tamanhoPontos, int tamanhoTestes, int nthreads)

Essa função recebe o array de pontos e o Array de teste sendo que ambos são da estrutura 'Pontos' , o valor de k observação que serão avaliadas , o número de linhas do train, o número de linha dos testes e o número de threads que será usado para comparar os pontos de teste com todos os pontos de train.

Tal função realiza o algoritmo de knn em si e retorna o tempo gasto na comparação entre os pontos, de acordo com o número de threads definido.

A imagem abaixo ilustra em " omp_set_num_threads(nthreads);" o número de threads criadas e em " #pragma omp parallel for schedule(dynamic)" que o primeiro for na sequencia é executado de maneira paralelizada, ou seja, cada uma das threads criada trata um intervalo de teste e ao término une o resultado de maneira dinâmica a variável compartilhada 'distanciasPonto' e retorna o tempo que levou para executar..

```
//atualize aqui o num de threads
omp_set_num_threads(nthreads);


clock_t start_time = clock();

#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < tamanhoTestes; i++) {
    // Calculate distances for each test point
    for (int j = 0; j < tamanhoPontos; j++) {
        distanciasPontos[i][j] = Distancia(testes[i], pontos[j]);
    }
}

clock_t end_time = clock();
```

*int ordena(int k, float *xtrain, float *ytrain, float *xtest)*

Ordena a entrada com um quicksort.

*double Mini_KNN(Ponto pontos[], Ponto testes[], int k, int tamanhoPontos, int tamanhoTestes, int nthreads, DistanciaPonto **distanciasPontos)*

O mini KNN tem esse nome devido ao seu tamanho reduzido. Como comentado anteriormente, a ideia dele é de testar diversas combinações de threads diferentes ao mesmo

tempo. Ou seja, com apenas um comando do usuário, é possível medir o tempo para 2, 4, 6, 8, 10, 12, 14 e 16 threads em uma utilização apenas do programa.

void Chama_KNN(Ponto pontos[], Ponto testes[], int k, int tamanhoPontos, int tamanhoTestes, int nthreads, int max_threads, int passo)

O mini KNN não trabalha sozinho, na verdade ele só faz um KNN de maneira mais eficiente (sem utilizar as partes que o tempo não é medido). O Chama KNN é responsável por chamar diversos mini KNNs para medir seus tempos e imprimi-los no terminal. Enquanto aumenta a quantidade a partir de um passo fornecido pelo usuário.

Leitura.C

Abaixo serão explicadas algumas das funções utilizadas pelo código para ler os arquivos de entrada. O leitura.h é apenas o header deste código, tendo declarado suas funções e importações.

void LeituraTrain(Ponto matrizPonto[], char xtrainFileName[], char ytrainFileName[])

Carrega os valores de Xtrain e Ytrain para a memória principal

void LeituraTest(Ponto matrizPonto[], char xtestFileName[])

Carrega os valores de Xtest para a memória principal

void EscreverY(Ponto matrizPonto[], FILE *file, int maxLinhas)

Escreve a classe dos Pontos dentro deles, utilizando como base o arquivo de entrada ytrain fornecido.

void EscreverX(Ponto matrizPonto[], FILE *file, int maxLinhas, int numColunas)

Escreve os valores do array “x” da estrutura Ponto. Ou seja, escreve os valores de entrada do arquivo xtrain dentro da estrutura.

Utilidade.C

Abaixo serão explicadas algumas das funções utilizadas pelo código para as mais diversas tarefas. O utilidades.h é apenas o header deste código, tendo declarado suas funções e importações.

void PrintArray(Ponto lista[], int max)

Imprime o array no terminal.

int CountFileLines(char filePath[])

Conta quantas linhas tem em um arquivo a partir de seu nome.

int ContarCol(char nomeArquivo[])

Conta quantas colunas tem um arquivo a partir de seu nome.

int Lenght(float array[])

Calcula o tamanho de uma array de pontos flutuantes.

int LenghtPonto(Ponto array[])

Calcula o tamanho de uma array da estrutura Ponto.

void alocarEspaco(Ponto *ponto, int numColunas)

Aloca memória.

void trocar(DistanciaPonto *a, DistanciaPonto *b)

Realiza a troca de dois DistanciaPonto de lugar.

int compararDistancias(const void *a, const void *b)

Compara duas distâncias e retorna 1 ou 0 dependendo de qual está mais próxima.

void SaveYTest(Ponto testes[])

Salva o resultado do KNN num arquivo chamado “ytest” localizado no diretório “output”.

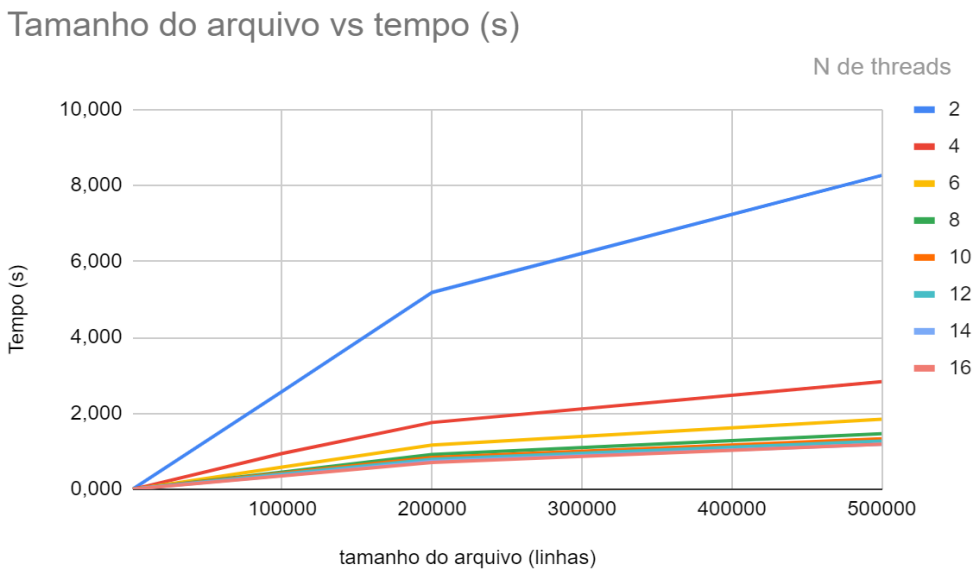
ANÁLISE

Para o projeto foi analisado o tempo de execução das comparações das distâncias. Vale ressaltar que o que foi paralelizado foram os cálculos das distâncias, a ordenação não está inclusa nesse trecho de código paralelizado. Para a análise foi utilizado threads iniciando em 2 e com passo de 2 como ilustrado na tabela abaixo:

Tam. Arq. \ N° threads	2	4	6	8	10	12	14	16
100	0,003	0,001	0,000	0,000	0,001	0,001	0,000	0,001
576	0,012	0,005	0,004	0,001	0,002	0,001	0,003	0,001
1.000	0,026	0,010	0,006	0,006	0,004	0,004	0,004	0,003
10.000	0,255	0,088	0,054	0,047	0,046	0,040	0,040	0,038
100.000	2,579	0,942	0,583	0,452	0,424	0,400	0,365	0,353
200.000	5,186	1,759	1,165	0,916	0,836	0,788	0,736	0,708
500.000	8,268	2,836	1,843	1,462	1,331	1,272	1,188	1,187

Tendo os tempos gastos em cada thread de acordo com o tamanho dos arquivos, pode-se plotar um gráfico para facilitar a análise do desempenho obtido em cada thread.

O computador utilizado para a execução possuía uma Ryzen 7 5800X, processador com 8 núcleos, 16 threads, velocidade base de 3,4 GHz e velocidade máxima de 4,7 GHz.



Tendo em vista o gráfico acima é possível notar um aumento de desempenho com o aumento no número de threads, contudo ao utilizar um valor de threads relativamente alto como, por exemplo, a partir de 8 os valores de tempo são bem semelhantes , ou seja, apresenta um ganho não muito significativo de tempo de execução.

Dificuldades

Comparado com o Exercício-Programa 1 da disciplina de Arquitetura e Organização de Computadores 2, esse código foi muito mais prático e simples de ser feito. Devido ao maior nível da linguagem utilizada (C) se comparado ao EP1 (Assembly MIPS). Porém, algumas dificuldades surgiram no decorrer desse trabalho:

- Testar diversas entradas ao mesmo tempo, sem reiniciar a execução. Devido ao alto tamanho das entradas, muitas vezes o programa dava segmentation fault, devido à falta de RAM.
- Foi mais difícil caçar erros do código devido ao multithreading.