Relatório EP2 OAC2

Considerações Iniciais

Especificação de Tamanho: nas linhas iniciais, estão definidos os valores das constantes, eles devem ser ajustados de acordo com os testes executados. Alguns exemplos de valores que utilizamos estão comentados (a partir da linha 10 no código paralelo e da linha 9 no sequencial).

Quantidade de threads: dentro da função *knn* em *EP2_Paralelizado.c*, definimos o número de threads a serem criadas (linhas 69 a 71). Neste trabalho, avaliamos os tempos para 4, 8 e 12 threads.

Especificação de Caminho: na função principal, os arquivos de teste, treino e saída são abertos, por isso é necessário especificar o caminho de diretórios até os arquivos desejados. Alguns arquivos de exemplo estão comentados (a partir da linha 111 no código paralelo e 105 no sequencial).

Valor de k: para os valores de tempos aqui registrados, utilizamos os valores de k=1, k=3, k=7 e k=10, mas é possível entrar com o valor desejado.

EP2_Sequencial.c

Implementação em C do algoritmo knn. Ele lê os valores das coordenadas de cada linha de um arquivo de treinamento (tal que a quantidade de linhas corresponde à quantidade de pontos de treinamento) , armazenando-os em um vetor de floats *xtrain*; da mesma forma, salva os valores das classes de cada um desses pontos em um vetor *ytrain* e os pontos de teste em um vetor *xtest*. Segue abaixo o que cada função do arquivo faz (ordem de aparição):

- ***void merge(int *sortedIndices, float *distances, int low, int mid, int high):*** ordena e une partes de um vetor através da lógica do algoritmo de mergesort.

- ***void mergeSort(int *sortedIndices, float *distances, int low, int high):*** implementa o algoritmo mergesort para ordenar um vetor de índices com base na distância em relação a *xtest*, dividindo recursivamente o vetor e chamando a função *merge* para união.
- ***int knn(int k, float *xtrain, float *ytrain, float *xtest):*** analisa os k vizinhos mais próximos do ponto *xtest*. Ela calcula as distâncias, ordena os vizinhos e determina a classe do ponto de teste com base na proximidade e contagem de classes.
- ***int main():*** executa o programa principal. Solicita o valor de `k` ao usuário, abre arquivos de entrada (*xtrain, ytrain, xtest*) e saída (*ytest*), aloca memória dinâmica para dados de treinamento e lê os dados de treinamento e teste dos arquivos correspondentes para então chamar a função *knn* para cada ponto de teste. Uma vez que todos os pontos foram analisados, ela escreve as classes no arquivo de saída. Por fim, mede o tempo de execução, libera a memória alocada e fecha os arquivos.

EP2_Paralelizado.c

A principal diferença entre os dois arquivos é que *EP2_Paralelizado.c* faz uso da biblioteca *omp.h*, que permite o uso de diretivas para a criação de threads e consequente paralelização do código. Assim, utilizamos as diretivas:

- *omp_set_num_threads(4), omp_set_num_threads(8)* e *omp_set_num_threads(12);* ao entrar na função *knn*, deixamos comentadas as diretivas que utilizamos para definir a quantidade de threads a serem criadas.
- *#pragma omp task firstprivate*: no algoritmo de *mergeSort*, cada divisão do vetor de índices é realizada em uma thread separada, tomando o cuidado de privar as variáveis utilizadas para cada chamada.

- *#pragma omp taskwait*: aparece em *mergeSort* para garantir que a união das partes (*merge*) ocorrerá apenas após o término da ordenação (tarefas paralelas).
- *#pragma omp parallel for*: paralelização dos loops de cálculo de distância em *knn*, já que cada ponto é independente do outro, as distâncias podem ser calculadas separadamente. Além disso, em *main*, separa os pontos de teste e executa *knn* para cada um deles em uma thread diferente.
- *#pragma omp parallel*: paraleliza um trecho determinado do código. No caso, paraleliza a execução do *mergeSort* em *knn*.
- *#pragma omp single*: garante a execução única de *mergeSort* em *knn*, evitando ordenar as distâncias mais de uma vez em relação a um mesmo ponto de teste.
- *#pragma omp atomic*: utilizada em *knn* para que a contagem de classes não sofra interferência de demais threads, evitando contagens repetidas.

Comparador.py

- ***ler_arquivo(nome_arquivo)***: essa função recebe o nome de um arquivo como parâmetro, tenta abrir o arquivo, lê todas as linhas, converte cada linha para um número de ponto flutuante e retorna uma lista desses valores. Se o arquivo não for encontrado, trata a exceção *FileNotFoundError* informando que o arquivo não foi encontrado e retorna uma lista vazia.
- ***calcular_porcentagem_igualdade(arquivo1, arquivo2)***: recebe duas listas de valores (presumivelmente vindas de arquivos lidos pela função *ler_arquivo*). Ela verifica se as listas têm o mesmo comprimento. Se não tiverem, a função retorna 0.0, indicando que os arquivos têm tamanhos diferentes e, portanto, a porcentagem de igualdade é 0.

Se as listas têm o mesmo comprimento, a função calcula o número de elementos iguais nas posições correspondentes das duas listas usando a função zip e sum. Em seguida, calcula a porcentagem de igualdade em relação ao total de elementos e retorna esse valor multiplicado por 100.

No final do código, há a execução do processo principal: duas variáveis, arquivo1 e arquivo2, são definidas chamando a função ler_arquivo para ler os conteúdos de dois arquivos diferentes. Em seguida, verifica-se se ambas as variáveis arquivo1 e arquivo2 têm conteúdo (ou seja, não são listas vazias ou o arquivo não foi encontrado). Se ambos os arquivos existirem, a função calcular_porcentagem_igualdade é chamada com esses dois arquivos como argumentos, e o resultado é armazenado na variável porcentagem. Finalmente, imprime a porcentagem de igualdade entre os dois arquivos, formatada com duas casas decimais.

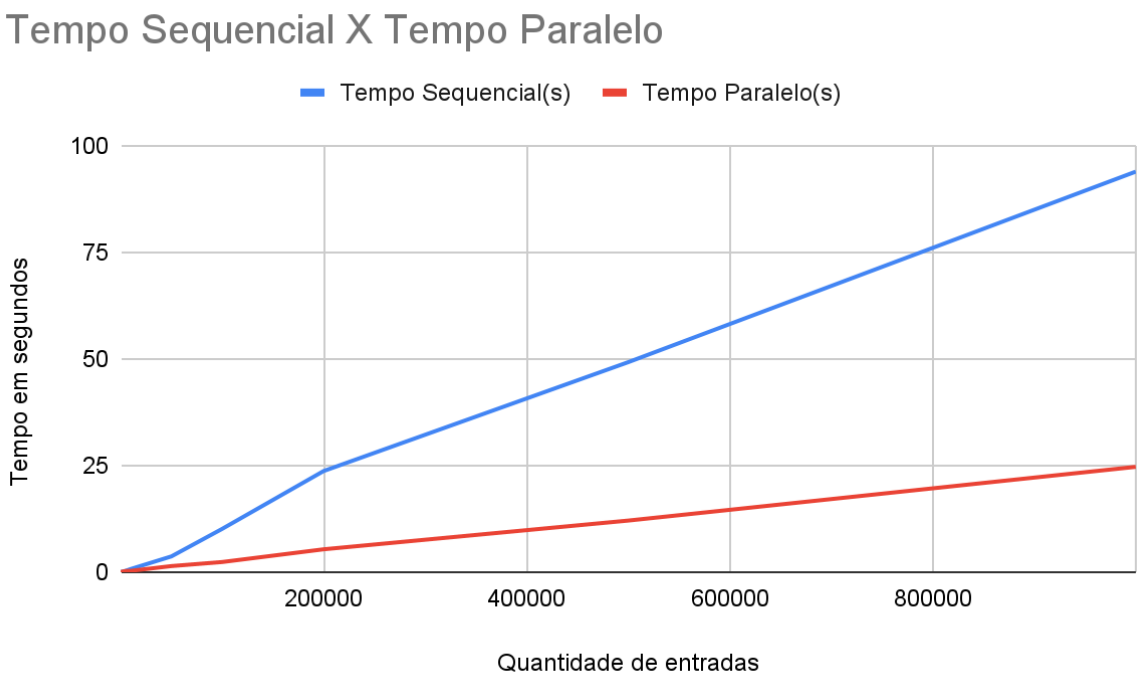
Resultados

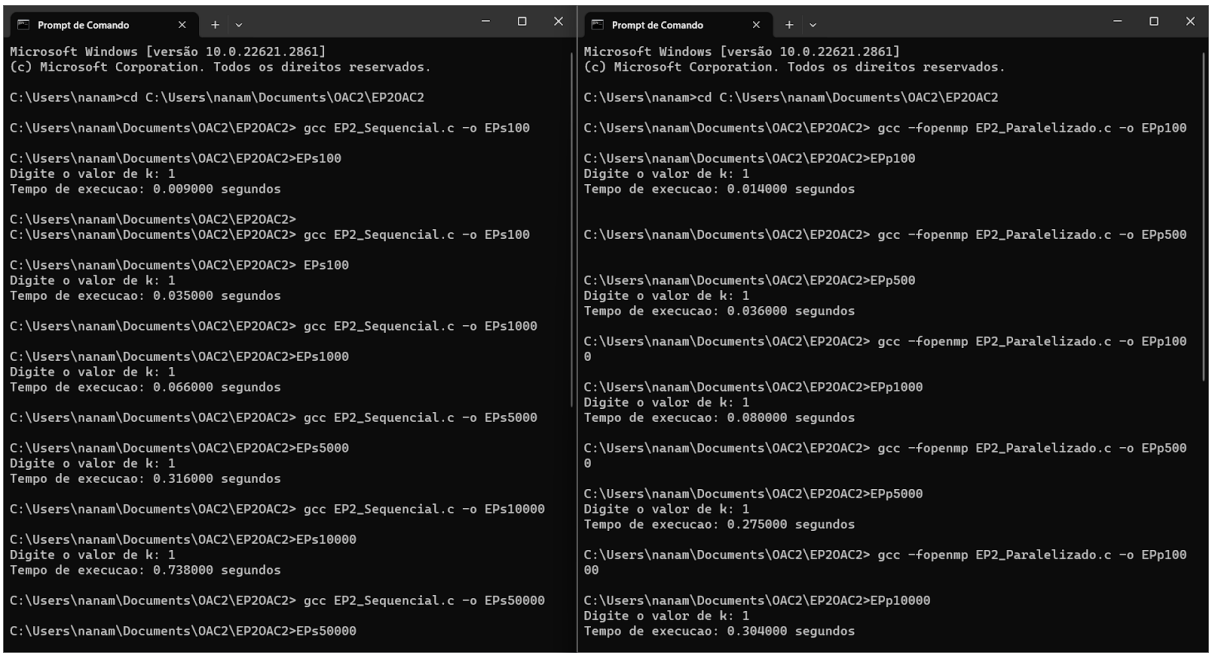
Abaixo, segue uma tabela e o gráfico correspondente para os tempos de cada um dos códigos (sequencial e paralelo) de acordo com a quantidade de pontos analisados:

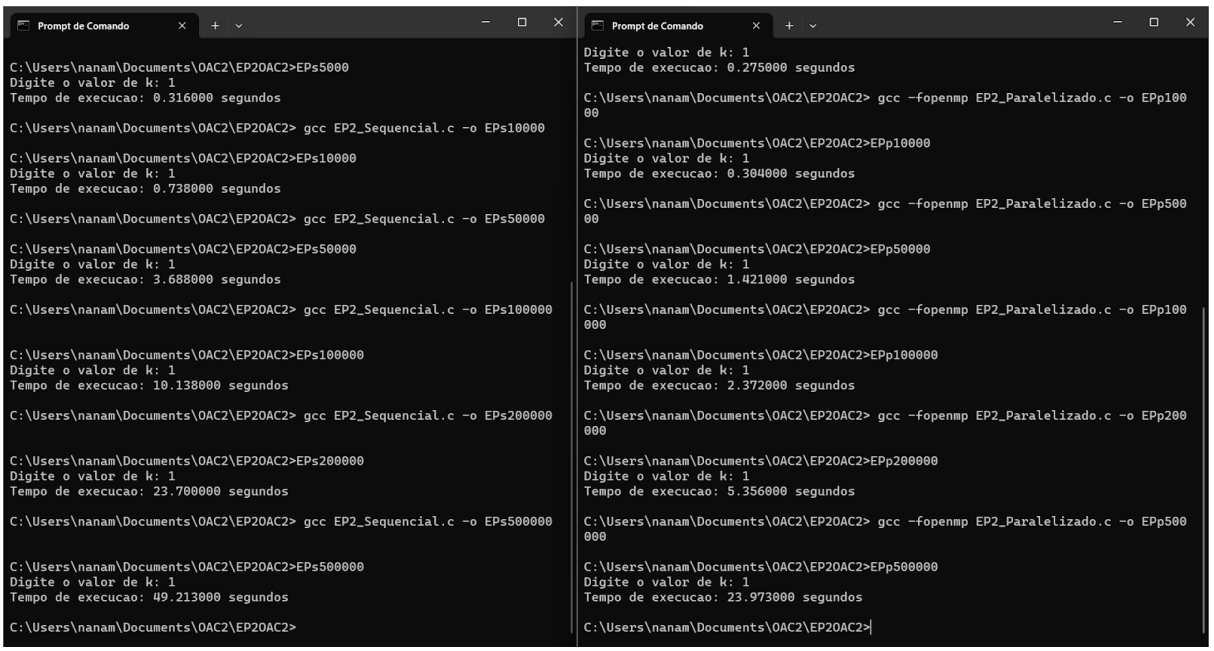
k=1 (1 ªcompilação + execução)

Quantidade de entradas (xtrain/ytrain)	Tempo Sequencial(s)	Tempo Paralelo(s)
100	0,009	0,014
500	0,035	0,036
1000	0,066	0,080
5000	0,316	0,275
10000	0,738	0,304
50000	3,688	1,421
100000	10,138	2,372
200000	23,700	5,356
500000	49,213	12,085

1000000	93,876	24,680

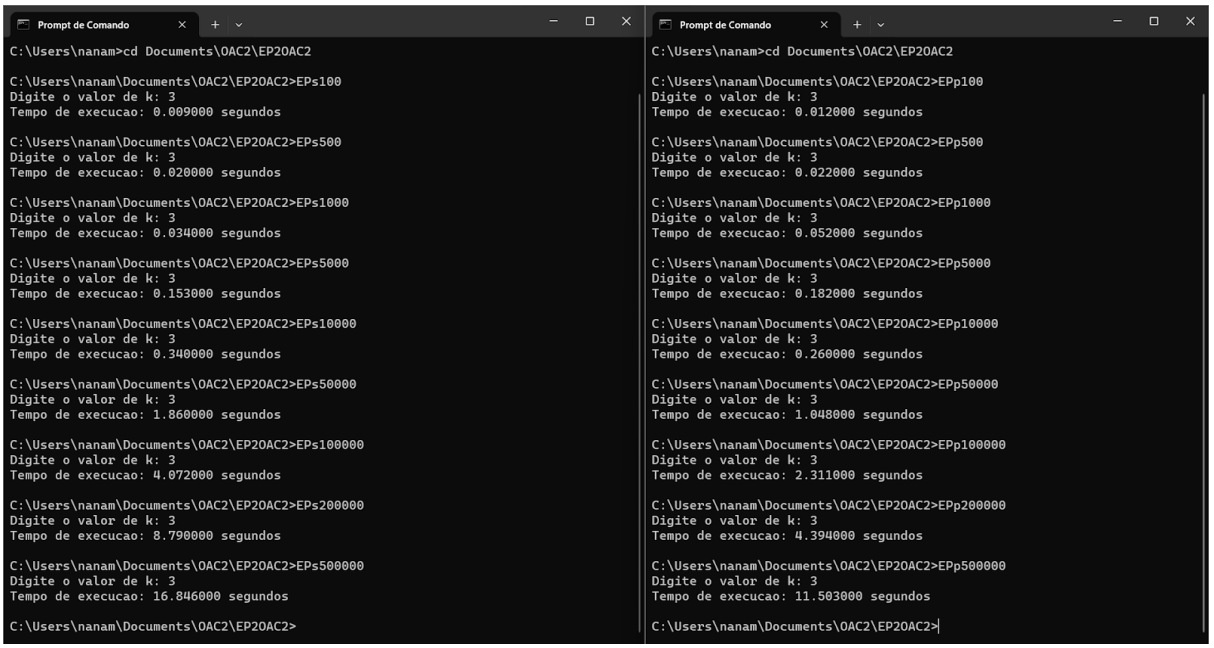






k=3

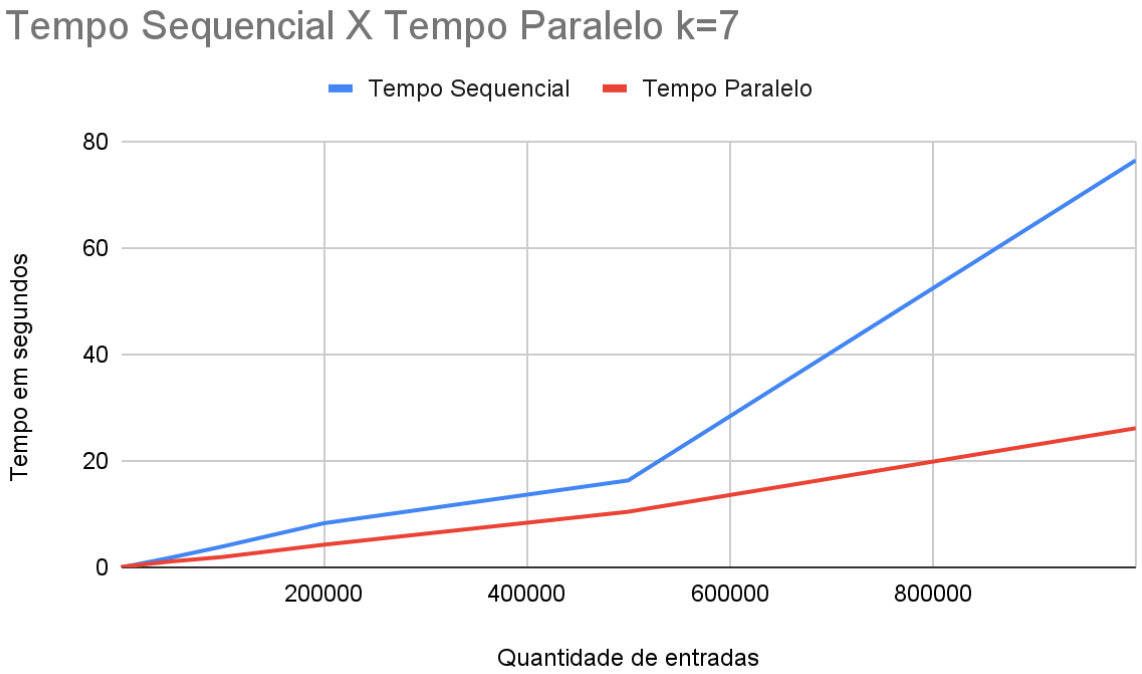
Quantidade de entradas (xtrain/ytrain)	Tempo Sequencial (s)	Tempo Paralelo (s)
100	0,009	0,012
500	0,020	0,022
1000	0,034	0,052
5000	0,153	0,182
10000	0,340	0,260
50000	1,860	1,048
100000	4,072	2,311
200000	8,790	4,394
500000	16,846	11,503
1000000	96,972	24,593

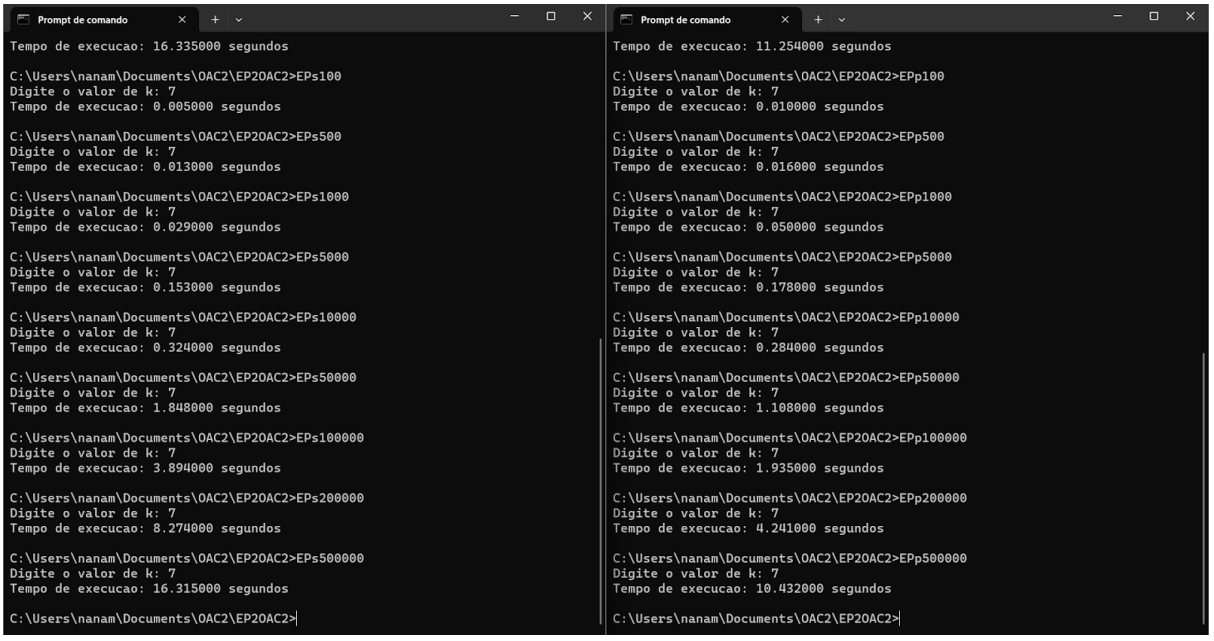


k=7

Quantidade de entradas (xtrain/ytrain)	Tempo Sequencial (s)	Tempo Paralelo (s)
100	0,005	0,010
500	0,013	0,016
1000	0,029	0,050
5000	0,153	0,178
10000	0,324	0,284

50000	1,848	1,108
100000	3,894	1,935
200000	8,274	4,241
500000	16,315	10,432
1000000	97,9931	26,121

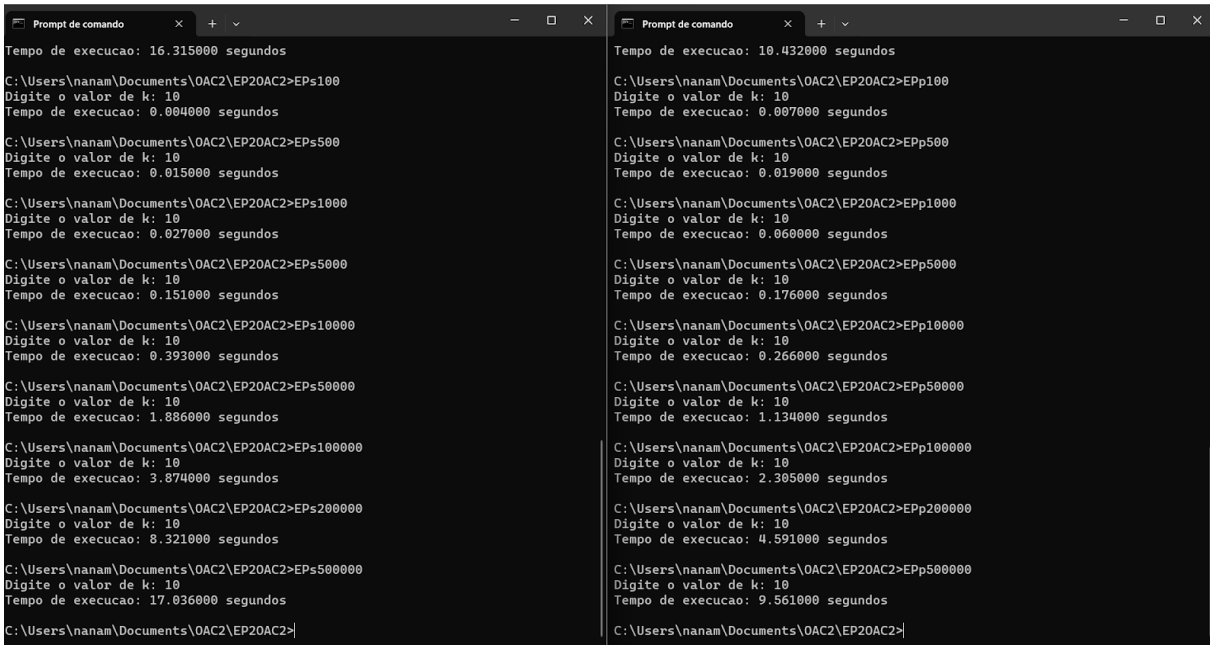




k=10

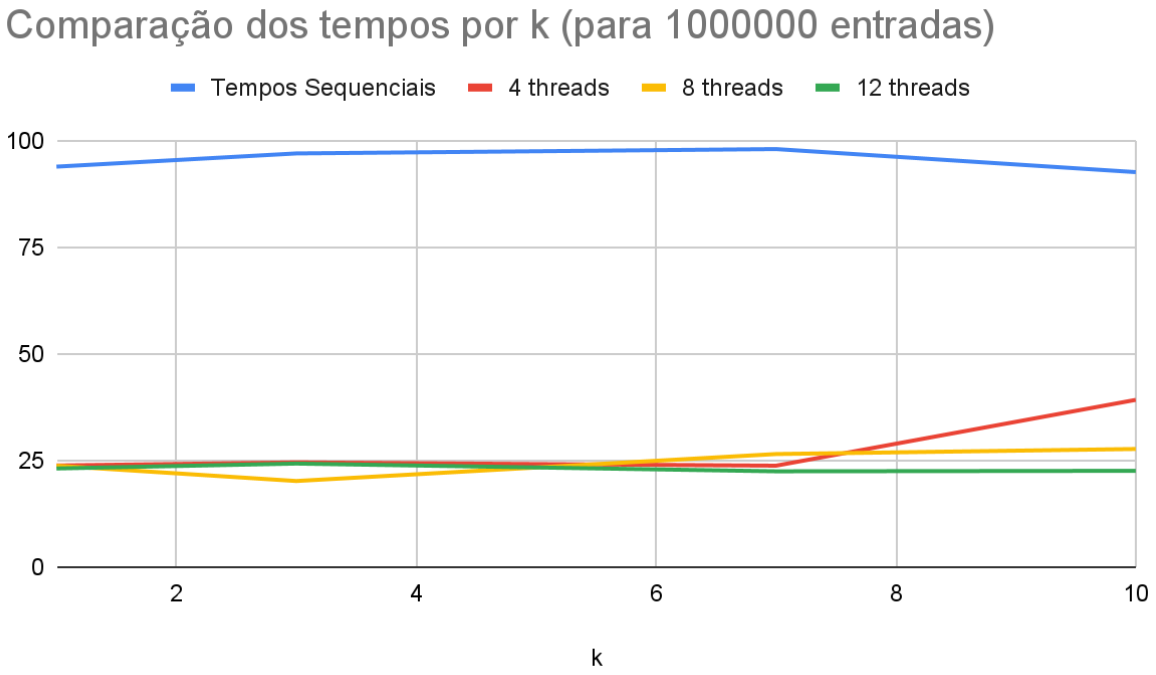
Quantidade de entradas (xtrain/ytrain)	Tempo Sequencial (s)	Tempo Paralelo (s)
100	0,004	0,007
500	0,015	0,190
1000	0,027	0,060
5000	0,151	0,176
10000	0,393	0,266
50000	1,886	1,134
100000	3,874	2,236
200000	8,321	4,591
500000	17,036	9,561
1000000	92,591	25,047

2



Tempos para 500000 entradas

k	Tempos Sequenciais	4 threads	8 threads	12 threads
1	93,876	23,824	23,78	23,128
3	96,972	24,549	20,188	24,275
7	97,993	23,781	26,509	22,462
10	92,591	39,209	27,694	22,576



Vale ressaltar que os dados de saída foram comparados utilizando o arquivo *Comparador.py,* que confirmou os resultados obtidos. Por extensão, validou que *EP2_Paralelizado.c* e *EP2_Sequencial.c* produzem o mesmo resultado.

Dos resultados, podemos notar que quanto mais entradas, melhor era o desempenho de *EP2_Paralelizado.c* em relação a *EP2_Sequencial.c*, tal que a ultrapassagem ocorre entre 5 e 10 mil entradas em geral. Além disso, aparentemente, quanto maior o k, mais tempo *EP2_Paralelizado.c* demora para passar *EP2_Sequencial.c* (como visto de k=1 para k=3, em que a intersecção entre os tempos de execução passa de 5000 para 10000 entradas).

Com os dados que nos foram fornecidos, os gráficos resultantes revelam a tendência de aumento do intervalo entre tempos de execução, ou seja, o tempo de execução de *EP2_Sequencial.c* tende a consumir cada vez mais tempo que *EP2_Paralelizado.c*, o que torna o crescimento de sua curva mais acentuado. Mesmo assim, possivelmente, se houver cada vez mais entradas, os gráficos convergiriam novamente.

Da quantidade de threads, podemos extrair que a utilização se torna vantajosa a partir de 8 delas, visto que, nesses casos, os códigos tendem a um tempo mais uniforme de execução ou, pelo menos, não perdem tanto desempenho para k's maiores. Com 4 threads a perda de desempenho ocorre rapidamente, de forma que, com o aumento de k, aparentemente se aproxima do tempo de execução do código sequencial.

Em resumo, podemos afirmar que a paralelização do código causa grande impacto na velocidade com que o programa é executado. Conforme as quantidades de entradas e comparações (k) aumentam, mais perceptível isso se faz. Ainda assim, deve-se ter em mente que esse ganho é limitado tanto pelo volume de dados quanto pelo número de threads.