UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES 2

DOCENTE: CLODOALDO APARECIDO

RELATÓRIO: EXERCÍCIO PROGRAMA

KNN PARALELIZADO COM A BIBLIOTECA OMP.H

Tiago Weizmann Hudler N° USP: 11204641

Júlia Du Bois Araújo Silva N° USP: 14584360

Gabriela Carvalho Vitorino N° USP:14749616

SÃO PAULO 2024

SUMÁRIO

1. IMPLEMENTAÇÃO

O principal objetivo do programa é executar o algoritmo do KNN (*K-nearest neighbours*) de forma sequencial e de forma paralela utilizando a biblioteca OpenMP, marcando o tempo de execução do algoritmo para análise posterior. Para esse propósito, a linguagem de programação escolhida foi C.

O programa final utiliza as seguintes bibliotecas:

- `stdio.h`: utilizada para executar entradas e saídas do programa, como a definição da quantidade de iterações para teste e o retorno do tempo de execução
- `stdlib.h`: utilizada para execução de alocação dinâmica de memória
- `time.h`: utilizada para marcação e cálculo do tempo de execução do programa
- `omp.h`: utilizada para paralelização do código

Para organização do código, foram criados 6 arquivos de código (.c) e 3 arquivos *header* (.h). Esses arquivos são os seguintes:

1. `readAndConvert.c`
    a. `readAndConvert.h`
2. `matrizes.c`
    a. `matrizes.h`
3. `knn.c`
    a. `knn.h`
4. `fullCodeNoParallel.c`
5. `fullCodeParallel.c`
6. `main.c`

Além desses arquivos, para facilitar as compilações e execuções do código, foi criado um arquivo *Makefile*. A forma de executar os arquivos será expandida posteriormente.

1.2 Arquivos do programa e suas funções

1.2.1 `readAndConvert.c` e `readAndConvert.h`

Esse arquivo foca na conversão de arquivo .txt para array e na conversão de array para arquivo .txt. Ele utiliza as bibliotecas stdio.h, stdlib.h e locale.h. Ele possui 3 funções: `int getNumberOfLines`, `double* readFileAndConvertToArray` e `void readArrayAndConvertToFile`.

A função `getNumberOfLines` recebe um char* representando o nome de um arquivo e calcula a quantidade de linhas do arquivo. Essa função só é utilizada internamente (ou seja, só é utilizada dentro desse arquivo).

A função `readFileAndConvertToArray` recebe um char* representando o nome de um arquivo e converte esse arquivo em um array de doubles cujo número de elementos é igual ao número de linhas do arquivo. Essa conversão é realizada por meio da função `atof()` da biblioteca `locale.h`. Ao final, retorna o array preenchido. Essa função é utilizada para o código sequencial e para o código paralelo.

Por último, a função `readArrayAndConvertToFile` recebe um array de doubles, o tamanho desse array e um char* que indica o arquivo onde este array deve ser escrito. Em seguida, ela escreve os doubles no arquivo por meio da função `fprintf()` com uma máscara para double com até 6 dígitos decimais. Essa função também é utilizada no código sequencial e paralelo.

1.2.2 `matrizes.c` e `matrizes.h`

Esse arquivo tem como objetivo a construção das matrizes de treino e teste sequencial e paralelo e o array de treino utilizados para o cálculo das distâncias na aplicação do knn. Ele utilizam as bibliotecas `stdio.h`, `stdlib.h` e `omp.h`, e possui 3 funções: `void createMatriz`, `void createMatrizParallel` e `void createYtrain`.

O arquivo .c possui o código das funções e o arquivo .h possui as assinaturas delas para serem usadas nos códigos completos do knn.

A função `void createMatriz` tem como input `n`, `h`, `w`, `array` e `result` que são, respectivamente, o número de elementos do arquivo de entrada, um parâmetro de previsão do knn, um parâmetro que indica a quantidade de linhas da matriz, o array feito na leitura dos arquivos e a estrutura da matriz de saída com w colunas.

A construção das matrizes é realizada com duas estruturas de loops for aninhadas, a primeira itera pelo array de entrada até um índice limite calculado por $n - w - h + 2$ e a cada elemento iterado o loop de dentro aloca esse elemento e os próximos $w - 1$ seguintes na respectiva linha da matriz de saída. Essa função é utilizada no código sequencial.

A função `void createMatrizParallel` recebe, além dos inputs da função descrita acima, o input nThreads para determinar o número de threads utilizado e tem a mesma lógica de cálculo das matrizes, porém ela possui as duas estruturas de loop for paralelizadas pela biblioteca `omp.h`. Na paralelização do código são usadas as diretrizes `#pragma omp parallel` para indicar que o trecho de código a seguir vai ser paralelizado e `#pragma omp for` utilizado para paralelizar estruturas de loop `for`, onde cada iteração pode ser realizada por uma thread, dependendo da disponibilidade de threads da máquina. A paralelização foi realizada no `for` externo da criação das matrizes, dividindo o parâmetro `i` no número desejado de threads. Essa função é utilizada no código paralelo.

A função `void createYtrain` recebe os parâmetros `n`, `h`, `w`, `ytrainSize`, `xtrain`, `ytrain`, os primeiros três parâmetros são os mesmos utilizados nas funções anteriores, yTrainSize é um valor calculado para ser o tamanho do array de saída, que é alocado no input ytrain, e xtrain é a matriz calculada anteriormente. Essa função é utilizada tanto no código sequencial quanto no paralelo.

1.2.3 `knn.c` e `knn.h`

Esse arquivo tem como objetivo a execução do algoritmo knn, tanto sequencialmente quanto com paralelização, recebendo como parâmetros a dimensão dos vetores (`w`), `xtrain`, `xtest`, e `ytrain`, os tamanhos de xtrain e xtest e, no caso da função com paralelização, o número de threads. As bibliotecas utilizadas são `stdio.h`, `stdlib.h` e `omp.h`.

Primeiramente temos a função `double* knn`, que é executada sequencialmente. A primeira linha já é a alocação do array de saída, o `ytest`. Essa alocação é feita dinamicamente, com base no tamanho de xtest. Podemos, então, dar início ao algoritmo.

O primeiro for-loop itera por xtest, usando o índice de iteração `i`. Portanto, `i` é o índice atrelado a `xtest`. Então, é alocado o array de distâncias (`dist`), do qual serão extraídas as k menores. Esse array é então populado com as distâncias entre o elemento atual de `xtest` e todos os elementos de `xtrain`. Portanto, o elemento 0 de dist é a distância entre o elemento `i` de `xtest` e o elemento 0 de `xtrain`, o elemento 1 de `dist` é a distância entre o elemento `i` de `xtest` e o elemento 1 de `xtrain`, e assim por diante.

O próximo passo é alocar o array de inteiros para guardar os índices de `xtrain` aos quais se relacionam as distâncias de `dist` (`idx`), pois estes valores serão ordenados. Após populado `idx`, é ordenado o array `dist`, e também `idx` de acordo com a ordenação de `dist` (lembrando que cada índice em `idx` relaciona uma distância de `dist` com um elemento de `xtrain`).

Por fim, verificamos os k primeiros índices de `idx`, pois estes estão relacionados às menores distâncias. Somamos então os valores de `ytrain` dos índices verificados em uma variável `sum` inicializada com zero, e depois calculamos a média com (`sum / k`). O valor resultado é colocado no índice `i` (índice atual de `xtest`) do `ytest`.

Todos esses passos se repetem para todos os elementos de `xtest`.

A função `double* knnParallel` segue a mesma lógica da primeira função deste arquivo, porém paraleliza a iteração por `xtest` em `nThreads`, recebendo o número de threads em sua chamada, utilizando `#pragma omp for`.

1.2.4 `fullCodeNoParallel.c`

Esse arquivo junta todos os códigos necessários para realizar o knn sequencial. Ele funciona apenas com variáveis definidas internamente, dentro do próprio código, e foi utilizado para os testes principais de tempo. Esse código importa `readAndConvert.h`, `matrizes.h` e `knn.h`, e utiliza as bibliotecas `stdio.h`, `stdlib.h` e `time.h`.

No início do código, são definidos os arquivos que serão utilizados. Os arquivos de xtrain, xtest e ytest devem ser definidos nas variáveis string de mesmo nome. Em seguida, em arrays de doubles chamados `arrayXtrain` e `arrayXtest` são colocados, respectivamente, todos os valores de xtrain e xtest. Depois, deve ser determinado pelo usuário no terminal em que o código está rodando a quantidade de

iterações que devem ocorrer. O código guarda o valor de cada uma dessas iterações e, ao final, calcula a média do tempo levado.

Após a determinação de iterações, é iniciado o loop que efetivamente executa o algoritmo do knn e o momento do início é marcado utilizando a função `clock()` da biblioteca `time.h`. É determinado o número de linhas de xtrain e xtest utilizando a fórmula $número\ de\ elementos\ do\ array - w - h + 2$ e é alocado o espaço para uma matriz para cada de tamanho `[númeroDeLinhas][w]`. Em seguida, é utilizada a função `createMatriz()` de `matrizes.h` para xtrain e xtest.

Para criação do ytrain, seu tamanho é calculado utilizando a fórmula $nXtrain - w - h + 1$ e é alocado um array desse tamanho para que seja inserido utilizando a função `createYtrain()` de `matrizes.h`.

O array ytest é gerado utilizando a função `knn()` de `knn.h`. Após a geração desse array, o tempo final é marcado utilizando a função `clock()`. Para marcar o tempo da iteração, o Δt é calculado por meio de $(t_1 - t_0) \div CLOCKS\_PER\_SEC$, onde `CLOCKS_PER_SEC` é uma constante utilizada para converter um valor de ticks de clock para segundos. Por último, o array ytest é colocado em um arquivo .txt determinado no início do código. Depois disso, as outras iterações ocorrem da mesma forma descrita.

Ao final das iterações, o tempo médio é calculado por meio da média do tempo de cada uma das iterações. O tempo de cada iteração e o tempo médio são impressos no terminal utilizado para rodar o código.

1.2.5 `fullCodeParallel.c`

Esse arquivo junta todos os códigos necessários para realizar o knn com paralelização. Ele funciona apenas com variáveis definidas internamente, dentro do próprio código, e foi utilizado para os testes principais de tempo. Esse código importa `readAndConvert.h`, `matrizes.h` e `knn.h`, e utiliza as bibliotecas `stdio.h`, `stdlib.h` e `omp.h`.

Esse código segue, em sua maior parte, a mesma lógica de fullCodeNoParallel.c, porém, no lugar das funções `createMatriz()` e `knn()`, usa, respectivamente, as funções `createMatrizParallel()` e `knnParallel()`, das mesmas bibliotecas das funções anteriores. Além disso, para marcar o tempo de cada iteração, não foi possível utilizar a função `clock()` pois ela contava os ticks de todos

os processadores, resultando em um tempo quase quadruplicado em relação ao tempo real de execução. Para substituir essa função, foi utilizada a função `omp_get_wtime()` da biblioteca `omp.h`.

1.2.6 `main.c`

Esse código foi feito para os testes executados pelo professor. Ele roda os códigos de `fullCodeNoParallel.c` e `fullCodeParallel.c`, um depois do outro, e imprime o tempo de execução de cada forma de realizar o knn. Ele recebe como parâmetros no comando de execução do programa na seguinte ordem:

```
xtrain.txt xtest.txt ytest.txt k h w numeroDeThreads
```

Os parâmetros utilizados nos testes foram:

- k = 3
- h = 1
- w = 3
- numeroDeThreads = 4

1.3 Como compilar e executar o código

Foi anexado juntamente com os códigos um arquivo Makefile que deve ser utilizado para a compilação. Para cada um dos arquivos, segue o comando que deve ser utilizado e o arquivo que é gerado por meio desse comando.

Programa	Make	Arquivo de execução
`readAndConvert.c`	`make readAndConvert`	`readAndConvert.o`
`matrizes.c`	`make matrizes`	`matrizes.o`
`knn.c`	`make knn`	`knn.o`
`fullCodeNoParallel.c`	`make normalEP`	`ep`
`fullCodeParallel.c`	`make parallelEP`	`ep`
`main.c`	`make ep`	`ep`
`clean`	`make clean`	-

`pedantic readAndConvert.c`	`make readAndConvert_pedantic`	`readAndConvert.o`
`pedantic matrizes.c`	`make matrizes_pedantic`	`matrizes.o`
`pedantic knn.c`	`make knn_pedantic`	`knn.o`
`pedantic fullCodeNoParallel.c`	`make pedantic_normal`	`ep`
`pedantic fullCodeParallel.c`	`make pedantic_parallel`	`ep`
`pedantic main.c`	`make pedantic_ep`	`ep`

É recomendado, para os arquivos de 1 milhão e 10 milhões de linhas, aumentar o stack size para acomodar as matrizes por meio do terminal. O comando, no Linux, para cada um desses arquivos, na utilização com os parâmetros de teste, é:

Número de linhas	Arquivo
1 milhão	`ulimit -s 65536`
10 milhões	`ulimit -s 524288`

Esse valor foi calculado com base no tamanho necessário para XTrain e XTest, com a seguinte fórmula, arredondada para o número de MB para cima da conversão (teto):

$$número\ de\ linhas\ da\ matriz\ \bullet\ w\ \bullet\ sizeof(double)\ \bullet\ 2$$

2. ANÁLISE DO DESEMPENHO

A análise do desempenho dos códigos sequencial e paralelo foi realizada em 3 máquinas com diferentes processadores.

O processamento dos códigos com mais de uma iteração foram limitados pelos arquivos com grandes números de linhas.

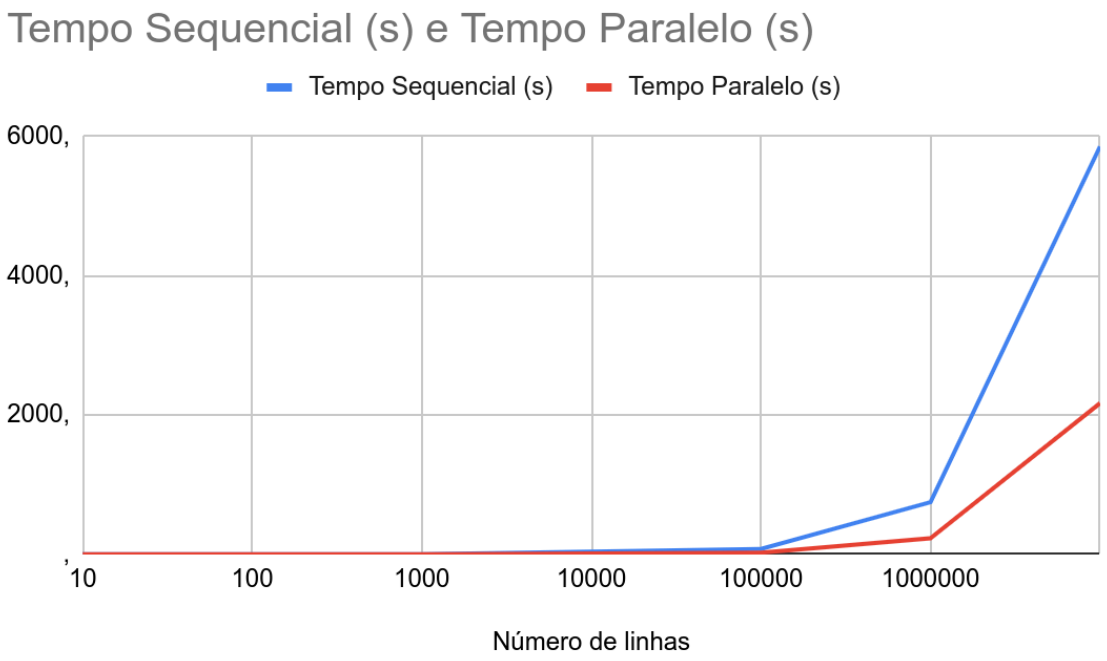
As respectivas análises são mostradas a seguir:

2.1 Máquina 1

Tabela de tempos e ganho:

8th Gen Intel® Core™ i5-8250U CPU @ 1.60GHz x 8				
Número de linhas	**Tempo Sequencial (s)**	**Tempo Paralelo (s)**	**num iterações**	**ganho percentual**
10	,014319	,005484	5	61,70%
30	,02955	,013728	5	53,54%
50	,042975	,018897	5	56,03%
100	,084083	,030243	5	64,03%
1000	,778883	,187372	5	75,94%
100000	75,806044	24,447389	5	67,75%
1000000	749,115898	230,766294	1	69,19%
10000000	5854,74	2169,34	1	62,95%

Gráfico comparativo:



De forma comparativa, é possível observar, desde os primeiros arquivos, que o código paralelo possui melhor desempenho do que o código sequencial.

A partir do teste com o arquivo de 100 linhas, o ganho do código paralelo começa a se tornar maior do que o dos arquivos anteriores, com exceção do arquivo de

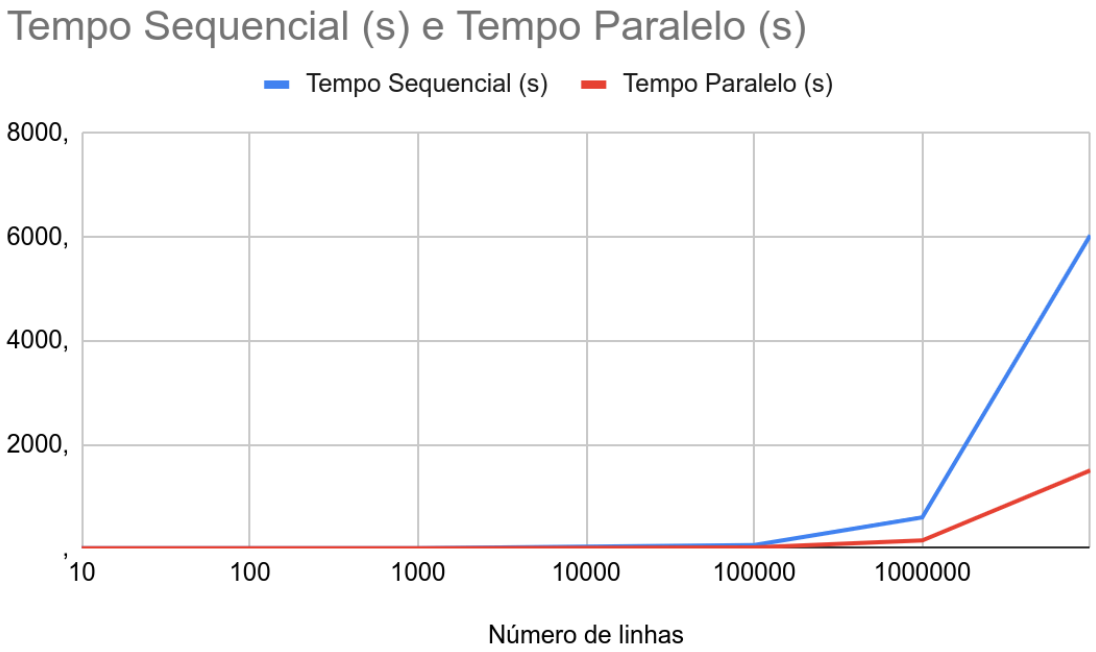
10 linhas. O código paralelo com maior ganho percentual foi o que possui arquivo de 1000 linhas, com ganho de 75%.

2.2 Máquina 2

Tabela de tempos e ganho:

11th Gen Intel(R) Core(TM) i7-11390H @ 3.40GHz x 8				
Número de linhas	**Tempo Sequencial (s)**	**Tempo Paralelo (s)**	**Num Iterações**	**Ganho Percentual**
10	,006807	,004003		41,19%
30	,01704	,0064		62,44%
50	,02774	,009452		65,93%
100	,05724	,016563		71,06%
1000	,54604	,153667		71,86%
100000	59,951697	15,040223		74,91%
1000000	595,163965	149,931997		74,81%
10000000	6034,801041	1499,690603		75,15%

Gráfico comparativo:



Na análise comparativa acima observou-se um comportamento semelhante ao da máquina 1, o qual há um melhor desempenho do código paralizado em todos os

casos teste com um aumento de desempenho dos códigos paralelos a partir do teste com arquivo de 100 linhas, além disso o arquivo com melhor desempenho foi o de 1 milhão de linhas com melhora de 75,15%.

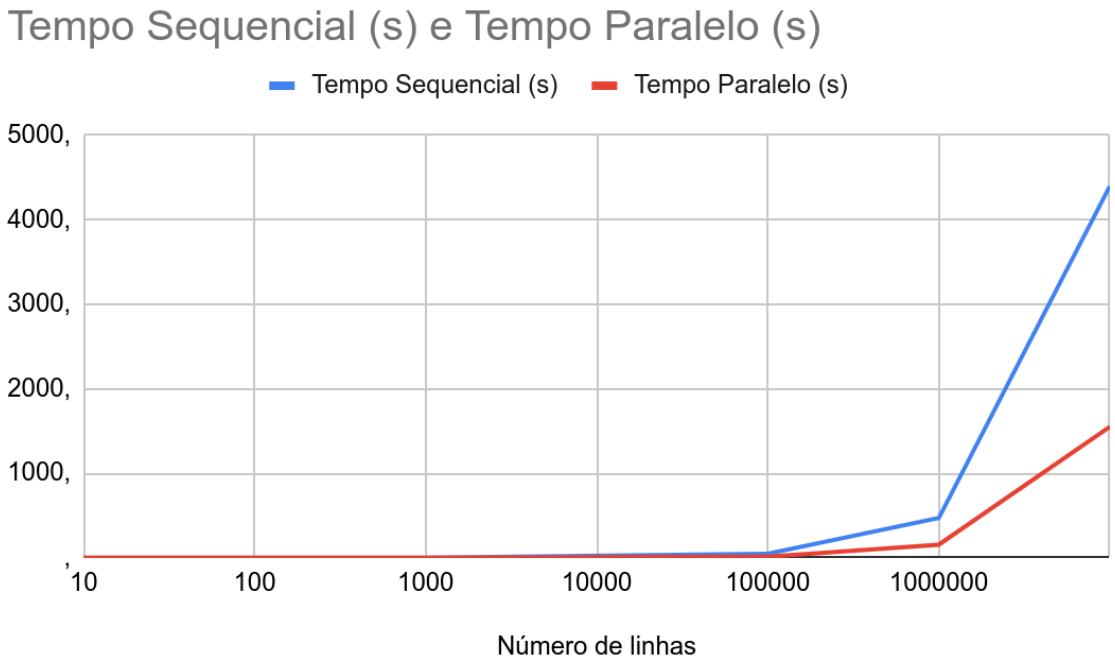
As diferenças entre o desempenho da máquina 1 e 2 foram o tempo do primeiro caso teste, o qual a máquina 2 teve o menor desempenho percentual, enquanto que na máquina 1 esse comportamento não foi observado, ademais as duas máquinas tiveram o melhor desempenho de códigos paralelos em testes de arquivos diferentes.

2.3 Máquina 3

Tabela de tempos e ganho:

11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz x 8				
Número de linhas	**Tempo Sequencial (s)**	**Tempo Paralelo (s)**	**Num Iterações**	**Ganho Percentual**
10	,004978	,01285	5	-58,14%
30	,013362	,012142	5	9,13%
50	,024166	,012904	5	46,60%
100	,044757	,016226	5	63,75%
1000	,465351	,13036	5	71,99%
100000	47,642631	14,603291	5	69,35%
1000000	470,823721	154,303864	1	67,23%
10000000	4396,430343	1549,730781	1	64,75%

Gráfico comparativo:



Por fim, a máquina 3 apresentou um comportamento geral semelhante aos demais, porém com algumas diferenças.

Primeiramente foi observado melhor desempenho no código paralelizado, com exceção do caso teste com o arquivo de 10 linhas, o qual foi observado um melhor desempenho, de 58%, no código sequencial. Outro ponto observado foi a melhora de desempenho do caso teste no código paralelizado de 30 linhas, com um percentual de melhora bem abaixo das demais máquinas no mesmo caso, ademais houve uma melhora de desempenho do código paralelo nos casos dos arquivos maiores, porém essa porcentagem foi menor do que nas máquinas 1 e 2.

Por fim, como na máquina 1, foi observado o melhor desempenho no caso paralelizado de 1000 linhas com 71,9% de melhora, enquanto que na máquina 3 o melhor caso foi no teste de 1 milhão de linhas.

Essa curva de melhora, que tem seu pico em 1000 linhas, é possivelmente causada por conta da forma como o computador está alocando espaço para arquivos maiores (possivelmente passando a utilizar a memória RAM em conjunto com os caches, o que é mais custoso em questão de tempo).