UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES II

DOCENTE: CLODOALDO APARECIDO DE MORAES LIMA

RELATÓRIO: SEGUNDO EXERCÍCIO-PROGRAMA
PARALELIZAÇÃO COM OPENMP

Diego Soares de Carvalho Junior N° USP:14562571 Turma: 94

Felipe Bavuso Fraga Nº USP:14569290 Turma: 94

Thiago Tanaka Padrão N° USP:14609492 Turma: 94

SÃO PAULO 2024

1. INTRODUÇÃO

O avanço das tecnologias de hardware e a crescente demanda por soluções computacionais eficientes impulsionaram o desenvolvimento de algoritmos otimizados para diferentes arquiteturas de sistemas. Nesse contexto, o K-Nearest Neighbors (KNN) destaca-se como um dos métodos mais populares para classificação e regressão devido à sua simplicidade e eficácia. Apesar de ser fácil de implementar, o KNN é um algoritmo computacionalmente intensivo, principalmente em aplicações que envolvem grandes volumes de dados, devido à necessidade de calcular distâncias para cada amostra do conjunto de treinamento.

Neste trabalho, foi realizado o estudo e a implementação do algoritmo KNN em linguagem C, com o objetivo de comparar o desempenho entre uma versão sequencial e outra paralelizada utilizando a biblioteca OpenMP. A abordagem sequencial serve como base para a avaliação da eficiência e do ganho de desempenho da versão paralela, que explora o processamento simultâneo em múltiplos núcleos da CPU.

O relatório apresenta os principais aspectos relacionados ao desenvolvimento das duas implementações, incluindo a análise do algoritmo, as estratégias de paralelização empregadas, os resultados obtidos e as considerações finais sobre o impacto do paralelismo no desempenho do KNN. Este estudo reforça a importância da otimização de algoritmos em cenários computacionalmente intensivos, destacando os benefícios das técnicas de programação paralela.

2. METODOLOGIA

2.1. Código sequencial

O programa implementa o algoritmo K-Nearest Neighbors (KNN) para um conjunto de dados de treinamento e um conjunto de dados de teste, ambos fornecidos em arquivos separados. A ideia central é, dado um vetor de entrada (Xtest), encontrar os k vizinhos mais próximos no conjunto de treinamento (Xtrain) e, a partir de seus valores de saída (Ytrain), prever um valor médio (no caso de k > 1) ou simplesmente retornar o valor do vizinho mais próximo (se k = 1). A

implementação é feita em C, de forma sequencial, e faz uso de algumas funções auxiliares para:

1. Ler dados de arquivos de texto.
2. Montar as matrizes de atributos (X) e vetores de saída (Y) a partir dos dados.
3. Calcular as distâncias entre instâncias de teste e instâncias de treinamento.
4. Selecionar os k vizinhos mais próximos.
5. Gerar as predições para o conjunto de teste e salvar os resultados em arquivos.

2.1.1. Constantes e variáveis globais

São definidas algumas constantes essenciais para a execução do programa, tanto para montagem dos conjuntos X e Y quanto para cálculo das distâncias, definindo como os dados são estruturados e calculados. São essas:

- w: tamanho da janela (número de atributos consecutivos considerados para cada ponto de dados).
- h: passo ou horizonte (quantidade de passos à frente no vetor original que se deseja prever).
- k: número de vizinhos a serem considerados no KNN.

Além disso, são definidas três variável globais ‘numLinhas’, ‘numLinhasTrain’ e ‘numLinhasTest’ para armazenar o número total de valores lidos dos arquivos, permitindo calcular o tamanho das matrizes e vetores posteriormente.

2.1.2. Função ‘lerArquivo’

Objetivo: Ler dados de um arquivo texto e armazená-los em um array unidimensional.

Processo:

1. Abre o arquivo em modo leitura.
2. Lê valores do arquivo, um a um, inserindo-os no array de doubles.
3. Armazena em `numLinhas` a quantidade total de valores lidos.
4. Fecha o arquivo.

Se ocorrer erro na abertura, o programa imprime uma mensagem de erro.

2.1.3. Função ‘montaX’

Objetivo: Dado um array de dados, monta a matriz X considerando a janela ‘w’ e o deslocamento ‘h’.

Processo:

1. Para cada posição possível no vetor original (do início até ‘numLinhas - w - h + 1’), extrai-se um conjunto de `w` valores consecutivos, formando um "vetor de atributos".
2. Esses valores são organizados por linhas em uma matriz ‘X’. Cada linha da matriz ‘X’ representa um conjunto de atributos que serão utilizados para prever a saída correspondente.

A função atual retorna `0` explicitamente, mas seu objetivo principal é preencher a matriz `X` com os valores adequados.

2.1.4. Função ‘alocaMatriz’

Objetivo: Alocar dinamicamente memória para a matriz X.

Processo:

1. Calcula o número de linhas como `(numLinhas - w - h + 1)`.
2. Aloca um ponteiro para um array de ponteiros do tipo `double`, sendo cada ponteiro uma linha.
3. Aloca, para cada linha, espaço para `w` colunas.

Essa função facilita a criação de matrizes para treinamento e teste, dado o número de linhas e colunas definido pelo tamanho dos dados e pelas constantes `w` e `h`.

2.1.5. Função ‘montaY’

Objetivo: Extrair o vetor Y (valores de saída) a partir do vetor original.

Processo:

1. Aloca um vetor de tamanho `(numLinhas - w - h + 1)`.
2. Cada elemento de `Y` corresponde ao valor no vetor original deslocado por `w + h - 1` posições. Isso significa que, dada a janela de

`w` pontos de entrada, o valor de saída é o valor presente `h` passos à frente no vetor original.

Assim, Y é construído de modo a corresponder diretamente à cada linha de X montada anteriormente.

2.1.6. Função ‘calculaDistancia’

Objetivo: Calcular a distância entre uma dada linha do conjunto de teste (`xtest[linhaAtual]`) e todas as linhas do conjunto de treinamento (`xtrain`).

Processo:

1. Aloca um vetor de distâncias.
2. Para cada linha do conjunto de treinamento:
    - Calcula a distância Euclidiana entre `xtest[linhaAtual]` e `xtrain[l]`, somando o quadrado das diferenças entre cada atributo.
    - Armazena o valor resultante no vetor de distâncias.

Essa etapa é crucial no KNN, pois determina quais pontos do conjunto de treinamento estão mais próximos do ponto de teste em questão.

2.1.7. Função ‘k_menores_indices’

Objetivo: A partir do vetor de distâncias calculadas, encontrar os índices correspondentes aos k menores valores.

Processo:

1. Inicializa vetores auxiliares para manter rastreio dos menores valores e seus índices.
2. Percorre o vetor de distâncias:
    - Se a distância atual for menor que o maior entre os k menores já encontrados, insere-a no lugar correto, deslocando os valores existentes para manter ordenação.

Ao final, ‘indices’ conterá os índices das linhas de treinamento que representam os k vizinhos mais próximos do ponto de teste atual.

2.1.8. Função ‘knn’

Objetivo: Executar o KNN para cada ponto de teste.

Processo:

1. Aloca um vetor `vetorytest` para armazenar as previsões do KNN.
2. Para cada linha do conjunto de teste:
    - Chama `calculaDistancia` para obter as distâncias.
    - Chama `k_menores_indices` para encontrar os k vizinhos mais próximos.
    - Se `k == 1`, a predição é simplesmente ‘ytrain[índice do vizinho mais próximo]’.
    - Se `k > 1`, calcula a média dos valores Y correspondentes aos k vizinhos.
3. Armazena a predição resultante em `vetorytest`.

Note que há uma chamada a `Sleep(100)`. Isso introduz uma pausa de 100 milissegundos por iteração. Isso é feito para simular um cenário mais custoso em tempo ou tornar a execução mais longa para fins de medição e comparação. Em um cenário real, isso não é necessário e prejudica o tempo de execução.

2.1.9. Função ‘salvarArrayEmArquivo’

Objetivo: Escrever um vetor de doubles em um arquivo texto.

Processo:

1. Abre o arquivo para escrita.
2. Percorre o array, escrevendo um valor por linha.
3. Fecha o arquivo.

Essa função é utilizada para salvar `Ytrain` e `Ytest` em arquivos de saída (`Ytrain.txt` e `Ytest.txt`), permitindo análise posterior dos resultados.

2.1.10. Função ‘main’

Objetivo: Executar o fluxo completo do programa.

Processo:

1. Aloca memória para um grande array que armazenará os dados lidos do arquivo.

2. Lê `Xtrain.txt` e monta `Xtrain` e `Ytrain`.
3. Lê `Xtest.txt` e monta `Xtest`\
4. Marca o tempo de início (`start`) para medir o desempenho.
5. Executa o KNN para gerar `Ytest`.
6. Marca o tempo de término e calcula o tempo total de execução.
7. Salva `Ytest` em arquivos.
8. Imprime o tempo de execução na tela.

Após a conclusão, o usuário terá os valores preditos pelo KNN para o conjunto de teste e poderá avaliar a qualidade das previsões e o desempenho do algoritmo (tempo de execução).

2.1.11. Comentários Adicionais

- Uso de memória: O programa utiliza alocação dinâmica de memória tanto para arrays como para matrizes, garantindo flexibilidade quanto ao tamanho dos dados.
- Preparação para Paralelização: A estrutura atual do código é propícia para paralelização, já que o cálculo das distâncias e a determinação dos k vizinhos mais próximos pode ser feita de forma independente para cada linha de teste.
- Parâmetros: A configuração do KNN depende das constantes `w`, `h` e `k`. Ajustar esses valores mudará a forma como os dados são dispostos e como a previsão é feita.

2.2. Código paralelizado

A seguir, apresenta-se apenas as diferenças introduzidas na versão paralelizada do código, em comparação à versão sequencial já descrita anteriormente. A estrutura geral do algoritmo, bem como a lógica de funcionamento do KNN, permanece inalterada. O foco desta seção é destacar os trechos modificados pela inclusão da biblioteca OpenMP e justificar a escolha dessas regiões como pontos estratégicos para paralelização.

2.2.1. Definição do número de Threads

```
omp_set_num_threads(512);
```

Antes de iniciar a execução do programa (na main), é definida a quantidade de threads que o OpenMP utilizará. No exemplo, foram selecionadas 512 threads. O objetivo dessa escolha é permitir que, durante a execução das partes paralelizadas, o trabalho seja distribuído entre múltiplos núcleos de processamento, aumentando o potencial de ganho de desempenho. Esse número pode ser ajustado.

2.2.2. Paralelização do Cálculo de Distâncias

```
double* calculaDistancia(double **xtrain,  double **xtest, int linhaAtual){
    double *distancias = malloc((numLinhas - w - h + 1) * sizeof(double));
    double d = 0.0;

    #pragma omp parallel for private(d)
    for(int l = 0; l < (numLinhas - w - h + 1); l++){
        d = 0.0;
        for(int c = 0; c < w; c++){
            d += pow(xtrain[l][c] - xtest[linhaAtual][c], 2);
        }
        distancias[l] = d;
    }

    return distancias;
}
```

Alteração Introduzida: A diretiva *"#pragma omp parallel for"* foi adicionada antes do laço que calcula as distâncias entre uma instância de teste e todas as instâncias de treinamento.

Motivação: O cálculo das distâncias é uma das etapas mais custosas do KNN, pois para cada ponto de teste percorre-se todo o conjunto de treinamento. Ao paralelizar esse laço, cada thread fica responsável por um subconjunto de linhas do conjunto de treinamento, permitindo que várias distâncias sejam calculadas simultaneamente, reduzindo assim o tempo total dessa operação.

Uso de *"private(d)"*: A variável d é declarada como private para garantir que cada thread tenha sua própria cópia, evitando condições de corrida durante o cálculo das distâncias.

2.2.3. Paralelização do Loop Principal do KNN

```
double* knn (double **xtrain, double *ytrain, double **xtest){
    double *vetorytest = malloc((numLinhas - w - h + 1) * sizeof(double));

    #pragma omp parallel for
    for(int x = 0; x < (numLinhas - w - h + 1); x++){
        Sleep(100);
        double *vetorDistancias = calculaDistancia(xtrain, xtest, x);
        int *indices = malloc(k * sizeof(int));
        k_menores_indices(vetorDistancias, indices);

        if(k == 1) {
            vetorytest[x] = ytrain[indices[0]];
        }

        if(k > 1){
            double media = 0.0;
            for(int t = 0; t < k; t++){
                media += ytrain[indices[t]];
            }
            media /= k;
            vetorytest[x] = media;
        }

    }

    return vetorytest;
}
```

Alteração Introduzida: A diretiva *"#pragma omp parallel for"* foi aplicada ao laço principal do KNN, responsável por iterar sobre cada linha do conjunto de teste.

Motivação: A paralelização desse loop é uma das chaves para o ganho de desempenho do KNN. Como cada ponto de teste é independente dos demais em termos de cálculo de distâncias e escolha dos vizinhos, é possível processar diversos pontos de teste simultaneamente. Cada thread executa o mesmo conjunto de operações para um subconjunto das linhas de teste, resultando em um aumento significativo de throughput.

2.2.4. Resumo

A escolha de paralelizar esses trechos específicos (o cálculo de distâncias e o loop principal do KNN) é baseada na análise do custo computacional do algoritmo. As principais operações do KNN—cálculo de distâncias e obtenção dos vizinhos mais próximos—podem ser realizadas de forma independente para cada ponto de teste. Ao distribuí-las entre múltiplas threads, obtemos um uso mais eficiente dos recursos de hardware (múltiplos núcleos), reduzindo o tempo de execução total. Essas áreas do código eram os “hotspots” da versão sequencial, ou seja, as partes onde o programa passava mais tempo computando, e portanto, são as mais indicadas para a paralelização.

2.3. Ambiente de desenvolvimento

O projeto foi desenvolvido e executado em um ambiente configurado para atender aos requisitos técnicos necessários. O hardware utilizado conta com um processador Intel Core i5 12400F, que possui 12 núcleos. Além disso, o sistema é equipado com 16 GB de memória RAM, proporcionando um ambiente robusto e eficiente para as etapas de compilação e execução.

O sistema operacional utilizado foi o Windows 11 Home, na versão de 64 bits. O compilador utilizado foi o MinGW-w64 (https://www.mingw-w64.org/), amplamente adotado em ambientes Windows para desenvolvimento em C e C++. A versão do GCC instalada foi a 14.2.0, assegurando suporte a recursos modernos de compilação e otimização.

2.4. Como executar as aplicações

As instruções a seguir descrevem o processo de compilação e execução do projeto, diferenciando as versões do código:

Versão sequencial:

Para compilar a versão sequencial do código, utilize o comando:

“gcc -o <nomeexecutavel> <nomeprograma>.c”

-versão paralelizada:

Para compilar a versão paralelizada com OpenMP, utilize o comando:
“gcc -o <nomeexecutavel> <nomeprograma>.c -fopenmp -lm”

As aplicações foram desenvolvidas para aceitar os parâmetros necessários diretamente pela linha de comando durante a execução. Os parâmetros esperados são:

- k: número de vizinhos mais próximos.
- w: janela de observações
- h: horizonte de previsão
- arquivo_Xtrain: nome do arquivo contendo os dados de treinamento.
- arquivo_Xtest: nome do arquivo contendo os dados de teste.

O comando para executar o programa, tanto sequencial quando paralelizado, segue o formato abaixo:
“./<nomeexecutavel> <k> <w> <h> <arquivo_Xtrain> <arquivo_Xtest>”

Certifique-se de que os arquivos de entrada estão no mesmo diretório do executável ou forneça o caminho completo.

2.5. Configurações dos experimentos

2.5.1. Tamanho dos Conjuntos de Dados e Parâmetros do KNN

Os experimentos foram realizados utilizando conjuntos de dados com diferentes quantidades de linhas, variando entre 10, 30, 50, 100, 1.000, 100.000, 1.000.000 e 10.000.000. Esses tamanhos foram selecionados com o objetivo de avaliar o comportamento do algoritmo em cenários que vão desde um volume muito pequeno até um volume extremamente grande de informações. Em todos os casos, os parâmetros do KNN permaneceram fixos, com a janela (w) igual a 3, o número de vizinhos (k) definido como 2 e o horizonte de previsão (h) igual a 1. Esses valores foram indicados pelo professor junto aos dados, os quais foram fornecidos pelo próprio docente, garantindo assim a padronização e a integridade do conjunto experimental.

2.5.2. Número de Execuções e Estatísticas de Avaliação

Para cada combinação de tamanho do conjunto de dados e configuração do KNN, foram realizadas 5 execuções independentes. Essa abordagem busca minimizar a influência de fatores externos e garantir maior robustez na análise dos resultados. Ao término das execuções, os tempos obtidos foram agregados por meio da média, proporcionando um valor representativo do desempenho em cada cenário testado. Essa estratégia permite uma análise mais confiável, reduzindo o impacto de flutuações ocasionais no tempo de processamento.

2.5.3. Métrica de Avaliação (Tempo de Execução e Speedup)

A métrica principal utilizada para avaliar o desempenho foi o tempo total de execução da função knn, considerando tanto a versão sequencial quanto a versão paralelizada. A partir desses tempos, calculou-se o speedup, definido como a razão entre o tempo da versão sequencial e o tempo da versão paralelizada. Além disso, foram analisados diferentes números de threads na versão paralela para verificar o impacto da utilização de múltiplos núcleos de processamento. Dessa forma, foi possível avaliar não apenas o desempenho bruto, mas também a eficácia da paralelização, fornecendo uma visão mais completa sobre a escalabilidade do algoritmo KNN para diferentes tamanhos de dados.

3. ANÁLISE DE DESEMPENHO

3.1. Resultados

3.1.1. Variando tamanho da entrada, mantendo o número de Threads

A fim de avaliar o desempenho da solução paralelizada, foram realizados testes variando o número de linhas de dados de entrada, mantendo-se fixo o número de 512 threads para todas as execuções. A tabela a seguir apresenta os

tempos de execução obtidos nas versões sequencial e paralelizada, bem como o speedup calculado.

tamanho (linhas)	sequencial (segundos)	paralelizado (segundos)	SpeedUp (T_seq/T_par)
10	0,000	0,011	0,00
30	0,000	0,012	0,00
50	0,003	0,017	0,18
100	0,004	0,010	0,40
1.000	0,027	0,018	1,50
100.000	2,266	0,476	4,76
1.000.000	22,768	4,463	5,10
10.000.000	217,383	42,760	5,09

Observa-se que, para quantidades relativamente pequenas de dados, o overhead da paralelização resulta em speedups menores que 1, indicando que o esforço de coordenação das threads não é compensado pelo ganho de desempenho. Entretanto, à medida que o tamanho do conjunto de dados cresce, o speedup aumenta expressivamente, demonstrando a eficácia da abordagem paralela. Desse modo, para conjuntos muito maiores, a versão paralela entrega um ganho substancial, reforçando a relevância da utilização de técnicas de paralelismo em algoritmos intensivos em processamento.

3.1.2. Variando o número de Threads, mantendo o tamanho da entrada

A fim de avaliar o desempenho da solução paralelizada, foram realizados testes variando o número de Threads, mantendo-se fixo o tamanho da entrada de dados em 100000 linhas para todas as execuções. A tabela a seguir apresenta os tempos de execução obtidos.

Número de Threads	Tempo de Execução (segundos)

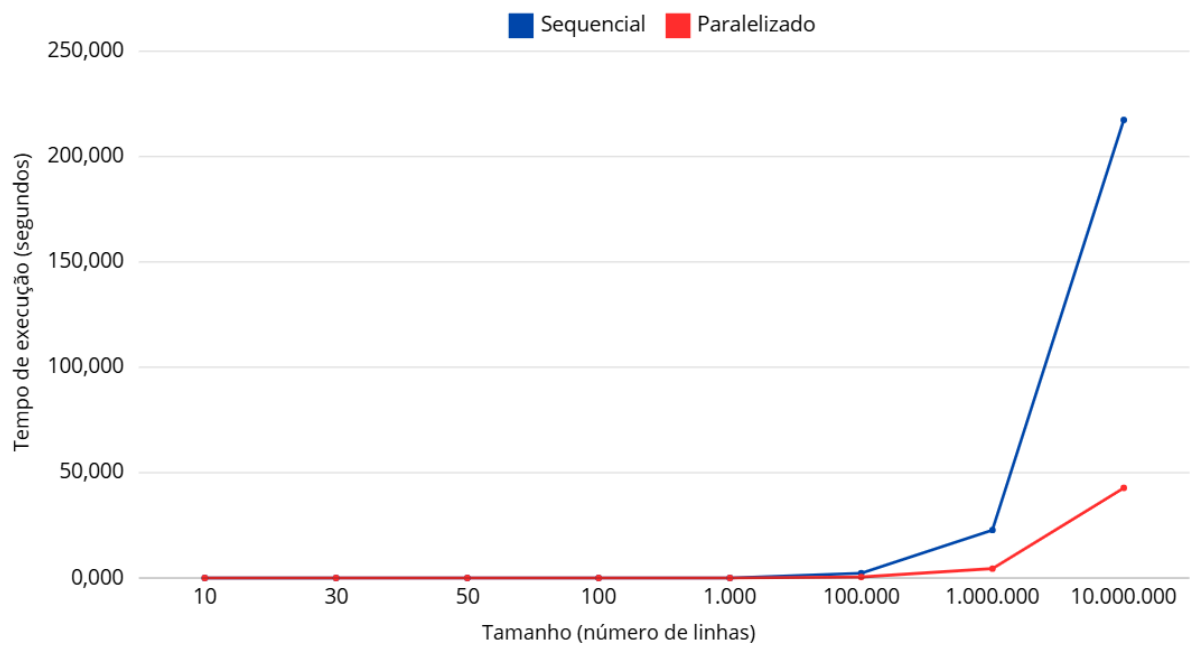
1	2.272
2	1.315
4	0.806
8	0.498
16	0.453
32	0.430
64	0.427
128	0.428
256	0.433
512	0.442
1024	0.463

A tabela mostra a influência do número de threads no tempo de execução. Observa-se uma queda acentuada no tempo à medida que se aumenta o número de threads de 1 para 8, passando de 2,272 segundos para 0,498 segundos, indicando um ganho significativo de desempenho com a adição de mais núcleos de processamento. A partir de 16 threads, o tempo continua a diminuir levemente, chegando a aproximadamente 0,427 segundos com 64 threads. No entanto, ao ultrapassar esse patamar, o tempo de execução praticamente se estabiliza ou até aumenta ligeiramente (por exemplo, em 1024 threads, sobe para 0,463 segundos), sugerindo um ponto de saturação.

Esse comportamento indica que, após certo número de threads, o overhead da gestão do paralelismo supera os benefícios do paralelismo adicional, resultando em ganhos marginais ou inexistentes. Assim, há um número ótimo de threads próximo a 64, além do qual não há melhora significativa no desempenho.

3.2. Gráficos

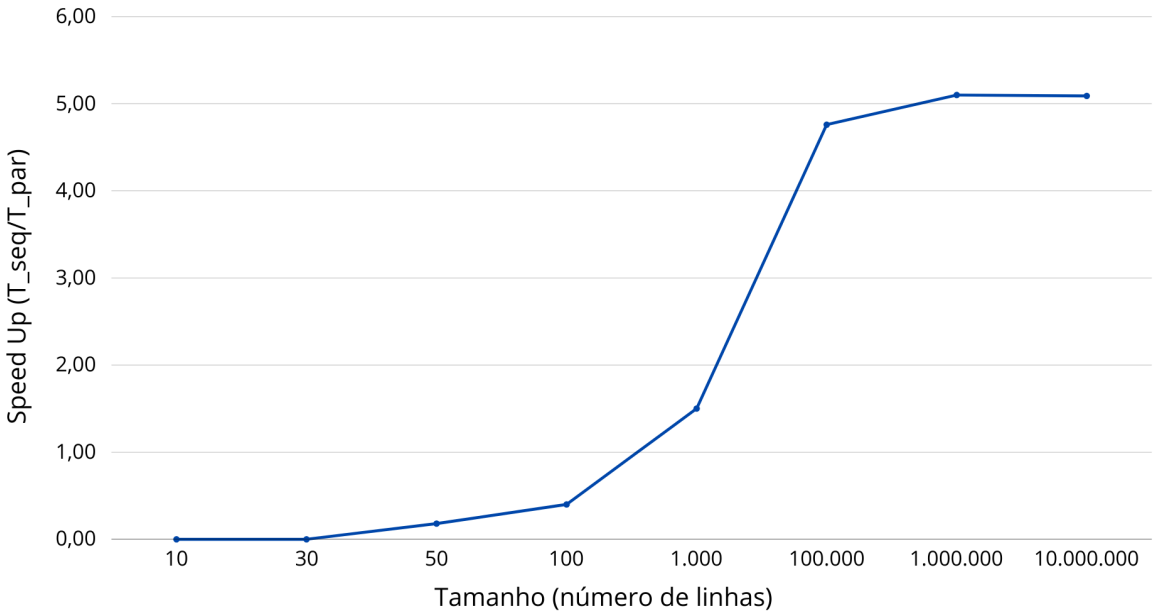
3.2.1. Tempo de execução vs. Tamanho do conjunto de dados



O gráfico de linhas elaborado a partir dos dados de tempo de execução apresenta, no eixo horizontal (eixo X), o aumento gradativo do tamanho do conjunto de dados (em número de linhas), enquanto no eixo vertical (eixo Y) é exibido o tempo de processamento em segundos. Foram plotadas duas linhas: uma representando o tempo de execução da versão sequencial e outra da versão paralelizada. Observa-se que, para tamanhos de dados reduzidos, ambas as linhas apresentam valores próximos, com a versão paralelizada ligeiramente mais lenta devido ao overhead.

Porém, conforme o número de linhas cresce, a linha que representa a versão paralelizada se mantém substancialmente abaixo daquela da versão sequencial, evidenciando a vantagem do paralelismo na redução do tempo de execução para grandes quantidades de dados.

3.2.2. SpeedUp vs. Tamanho do conjunto de dados

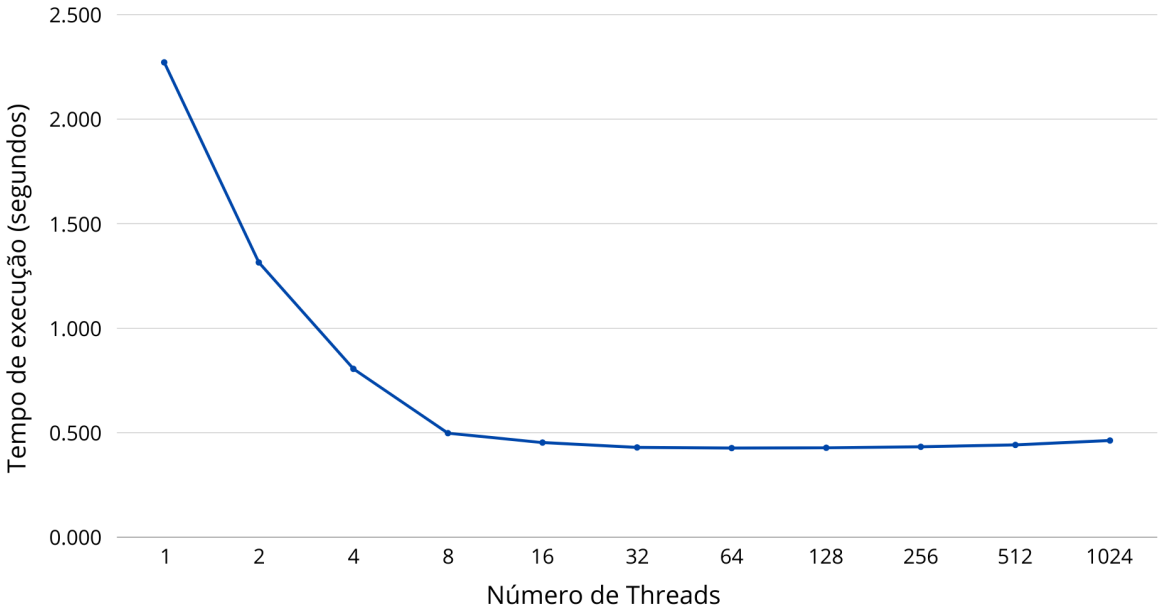


O gráfico de linhas elaborado para o speedup apresenta, no eixo horizontal (X), o tamanho do conjunto de dados (em número de linhas), enquanto o eixo vertical (Y) exibe os valores do speedup calculado. Cada ponto no gráfico representa um determinado tamanho de entrada com o respectivo ganho de desempenho da versão paralelizada em relação à sequencial.

O gráfico revela um comportamento significativo no ganho de desempenho à medida que o tamanho do conjunto de dados aumenta. Em volumes de dados pequenos (entre 10 e 100 linhas), o speedup é praticamente nulo ou muito baixo, indicando que o overhead introduzido pela paralelização supera os benefícios do processamento paralelo.

Entretanto, conforme o número de linhas cresce, há uma virada acentuada na curva: a partir de 1.000 linhas, o speedup passa a aumentar rapidamente, atingindo valores superiores a 5 com 1.000.000 e 10.000.000 de linhas. Esse padrão evidencia que a paralelização torna-se extremamente eficaz em cenários de grande escala, onde as operações custosas do KNN podem ser divididas entre múltiplas threads, reduzindo drasticamente o tempo total de execução.

3.2.3. Tempo de execução vs. Número de Threads



O gráfico apresenta o tempo total de execução em função do número de threads utilizadas no processamento. No eixo horizontal (X), encontra-se o número de threads, variando de 1 até 1024, enquanto o eixo vertical (Y) exibe o tempo de execução em segundos. Cada ponto da linha representa uma configuração diferente de paralelismo, evidenciando como o tempo de processamento se comporta à medida que o paralelismo é ampliado.

Na análise dos resultados, nota-se uma queda muito expressiva no tempo de execução ao passar de 1 para 8 threads, demonstrando ganhos significativos de desempenho devido à divisão eficiente do trabalho entre múltiplos núcleos. Essa redução continua, porém de forma mais sutil, conforme se aumenta o número de threads até atingir valores em torno de 64 threads. Nesse intervalo, o paralelismo é explorado de maneira bastante produtiva, minimizando significativamente o tempo necessário para concluir a tarefa.

Contudo, a partir do momento em que se ultrapassa a faixa de aproximadamente 64 threads, o tempo de execução deixa de diminuir de forma relevante, mantendo-se praticamente estável ou até mesmo aumentando levemente com o uso de 512 ou 1024 threads. Essa estabilização sugere que, a partir de certo ponto, o overhead gerado pela coordenação das threads e comunicação entre elas passa a anular os benefícios do paralelismo adicional.

Assim, conclui-se que existe um número ótimo de threads após o qual não se obtém vantagens de desempenho significativas, evidenciando um limite prático na escalabilidade do algoritmo nesse ambiente.

3.3. Discussão

A análise dos gráficos evidencia um padrão claro: conforme o tamanho dos dados aumenta, a versão paralelizada do KNN apresenta ganhos de desempenho cada vez mais significativos. Nos experimentos iniciais, com poucos dados, o overhead da paralelização é predominante, resultando em speedups próximos de zero. No entanto, à medida que o volume de dados cresce, o tempo de execução da versão paralelizada se reduz drasticamente em relação ao sequencial, alcançando um speedup superior a 5 para cenários de milhões de linhas. Esse comportamento reflete a tendência do paralelismo em se tornar mais eficaz à medida que a carga computacional aumenta, permitindo um melhor aproveitamento dos recursos disponíveis.

Em termos de impacto do paralelismo, observou-se que, para tamanhos de dados reduzidos, o processamento simultâneo não compensa o custo adicional de coordenação entre as threads. Entretanto, quando se atinge um certo limiar de dados, o paralelismo passa a fornecer ganhos expressivos, reduzindo o tempo total de execução. Dessa forma, o paralelismo mostra-se especialmente vantajoso em aplicações de grande escala, onde o tempo de cálculo se torna o fator limitante e o esforço de sincronização e comunicação entre as threads é diluído diante do volume de trabalho.

Por fim, algumas limitações foram identificadas. Embora o aumento do número de threads resulte em ganhos iniciais significativos, existe um ponto a partir do qual o tempo de execução deixa de diminuir e pode até aumentar. Esse fenômeno indica a presença de limites de escalabilidade, influenciados tanto pelo overhead de criação e gerenciamento de threads quanto pela concorrência no acesso à memória. Além disso, em conjuntos de dados muito pequenos, o custo associado à paralelização não é compensado pela redução no tempo de cálculo, evidenciando que a eficácia do paralelismo depende diretamente da escala do problema abordado.

4. CONCLUSÃO

O presente trabalho demonstrou a importância e o potencial do uso do paralelismo no algoritmo KNN, um dos métodos mais simples e efetivos em tarefas de classificação e regressão. A comparação entre a versão sequencial e a versão paralelizada utilizando OpenMP evidenciou que, embora o overhead inicial da coordenação entre as threads possa não se justificar em casos com conjuntos de dados pequenos, a abordagem paralela torna-se extremamente vantajosa à medida que o volume de dados aumenta significativamente.

Os resultados mostraram que, para grandes quantidades de dados, o tempo de execução da versão paralelizada foi substancialmente menor do que o da versão sequencial, alcançando speedups superiores a 5. Esse ganho de desempenho ressalta a escalabilidade do algoritmo e a capacidade de explorar múltiplos núcleos de processamento de maneira eficiente. Ao mesmo tempo, a análise também permitiu identificar limites práticos da escalabilidade: além de um determinado número de threads, o tempo de execução não diminui mais e pode até mesmo apresentar um leve aumento, resultado do maior custo de sincronização e gerenciamento de recursos.

Em suma, a conclusão principal é que o paralelismo compensa para cargas de trabalho elevadas, tornando o KNN mais eficiente em contextos de grande escala. Esses resultados reforçam a relevância da programação paralela como estratégia para lidar com o crescimento contínuo de dados e demandas computacionais, indicando que a escolha dos parâmetros, como o número de threads, deve ser feita de forma criteriosa para equilibrar o ganho de desempenho e o overhead inerente ao paralelismo.

5. REFERÊNCIAS BIBLIOGRÁFICAS

1. LIMA, Clodoaldo. *Programação Paralela*. Organização e Arquitetura de Computadores Ⅱ, Universidade de São Paulo, 2014.

2. PATTERSON, David A.; HENNESSY, John L. *Organização e projeto de computadores: A interface Hardware/Software*. Tradução da 5. ed. Elsevier, 2014.
3. OPENMP REFERENCE GUIDE. *OpenMP Application Programming Interface*. Version 5.2. [PDF]. Disponível em: https://www.openmp.org/resources/refguides/ .