# Paralelização de Algoritmo - KNN

**Bryan de Lima Naneti Barbosa**[1], **Heitor Rodrigues Sabino**[2], **Ranulfo Mascari Neto**[3]

[1] Departamento de Ciência da Computação – Universidade Federal de Lavras (UFLA) – Lavras – MG – Brazil

{bryan.barbosa, heitor.sabino, ranulfo.neto}@estudante.ufla.br

***Abstract.*** *This project explores the parallelization of the K-Nearest Neighbors (KNN) algorithm using parallel processing techniques to accelerate the classification of large datasets. The implementation was done in Python, using the MPI for Python (`mpi4py`) library to distribute the processing across multiple cores. The results showed that the parallelized version of KNN significantly improved execution time compared to the sequential version, particularly with large-scale datasets.*

***Resumo.*** *Este projeto explora a paralelização do algoritmo K-Nearest Neighbors (KNN) utilizando técnicas de processamento paralelo para acelerar a classificação de grandes volumes de dados. A implementação foi realizada em Python, empregando a biblioteca MPI for Python (`mpi4py`) para distribuir o processamento entre múltiplos núcleos. Os resultados demonstraram que a versão paralelizada do KNN apresentou melhorias significativas em termos de tempo de execução quando comparada à versão sequencial, especialmente em datasets de grande escala.*

## 1. Introdução

O K-Nearest Neighbors (KNN) é um dos algoritmos de aprendizado supervisionado mais simples e populares, utilizado para tarefas de classificação e regressão. Apesar de sua simplicidade, o KNN pode se tornar computacionalmente custoso em datasets grandes, já que a classificação de uma nova amostra requer a comparação com todas as amostras presentes no dataset de treinamento. Este projeto visa explorar a paralelização do KNN para melhorar seu desempenho em termos de tempo de execução, especialmente em contextos onde a eficiência computacional é crucial.

### 1.1. Definição do Problema

O principal desafio abordado neste projeto é a alta complexidade computacional do KNN em grandes datasets. O algoritmo exige a comparação de cada nova amostra com todas as amostras do dataset de treinamento, resultando em uma complexidade de tempo de O(n * m), onde n é o número de amostras no dataset de treinamento e m é o número de amostras a serem classificadas. A paralelização é uma abordagem promissora para mitigar este custo computacional, permitindo que as operações sejam distribuídas entre vários processadores, reduzindo assim o tempo total de execução.

### 1.2. Motivação

A crescente disponibilidade de dados e o aumento na complexidade dos modelos de aprendizado de máquina exigem soluções mais eficientes para processamento de dados em larga

escala. A paralelização do KNN é particularmente relevante em cenários onde a precisão do algoritmo é desejável, mas o tempo de execução se torna um fator limitante. Implementar uma versão paralela do KNN pode permitir a utilização do algoritmo em contextos onde, de outra forma, ele seria inviável devido a restrições de tempo.

### 1.3. Objetivo

O objetivo deste projeto é desenvolver e avaliar uma implementação paralelizada do algoritmo KNN, utilizando a biblioteca `mpi4py`. Pretende-se comparar o desempenho da versão paralela com a versão sequencial, medindo o tempo de execução e a escalabilidade em diferentes configurações de hardware e tamanhos de datasets.

## 2. Referencial Teórico

Durante a revisão da literatura, encontramos poucos estudos que abordam diretamente a paralelização do algoritmo K-Nearest Neighbors (KNN). No entanto, um dos principais trabalhos relevantes é o artigo "Parallel K-Nearest Neighbor Search Using Synchronous Pruning" (Zhang et al., 2013), que se concentra em otimizar o processo de busca de vizinhos mais próximos em ambientes paralelos. Esse estudo propõe o uso de uma técnica de poda síncrona, que permite reduzir o número de cálculos de distância necessários ao eliminar candidatos que não podem ser vizinhos mais próximos com base em um critério de poda antecipada.

Outro trabalho relevante é o "Efficient Parallel Algorithms for K-Nearest Neighbor Classification" (Karypis e Kumar, 1998), onde os autores propõem uma abordagem para dividir o conjunto de dados entre múltiplos processadores de forma equilibrada, minimizando o tempo de comunicação entre os processos. Esta divisão otimizada é fundamental para garantir que todos os processadores estejam igualmente carregados, evitando gargalos que poderiam comprometer a eficiência do algoritmo paralelo. Essas abordagens, embora focadas em diferentes aspectos da paralelização do KNN, destacam a importância de balancear a carga entre os processadores e de minimizar a comunicação desnecessária. Ambas as técnicas foram consideradas e adaptadas ao desenvolvimento do KNN paralelizado neste projeto, utilizando a biblioteca `mpi4py` para implementar a distribuição das tarefas de cálculo de distâncias.

### 2.1. Explicação do KNN

O *K-Nearest Neighbors* (KNN) é um algoritmo de aprendizado supervisionado amplamente utilizado para tarefas de classificação e regressão. A ideia central do KNN é baseada na premissa de que objetos semelhantes tendem a estar próximos uns dos outros no espaço de características. Em outras palavras, a classe ou valor de uma amostra desconhecida pode ser determinada observando as classes ou valores das amostras mais próximas a ela.

O funcionamento básico do KNN pode ser descrito pelos seguintes passos:

- **Escolha de** $k$: Inicialmente, é necessário escolher um valor para *k*, que representa o número de vizinhos mais próximos que serão considerados na determinação da classe ou valor da amostra desconhecida.

- **Cálculo da Distância**: Para determinar os vizinhos mais próximos, é calculada a distância entre a amostra desconhecida e todas as amostras do conjunto de treinamento. A métrica de distância mais comum utilizada é a distância Euclidiana, que é calculada pela fórmula:

$$d(p, q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

  Onde $d(p, q)$ é a distância entre os pontos $p$ e $q$, e $p_i$ e $q_i$ são as coordenadas dos pontos em cada dimensão.
- **Identificação dos Vizinhos Mais Próximos**: Após calcular a distância entre a amostra desconhecida e todas as amostras do conjunto de treinamento, são selecionados os *k* vizinhos mais próximos, ou seja, aqueles com as menores distâncias.
- **Classificação ou Regressão**:
  - **Classificação**: No caso de uma tarefa de classificação, a amostra desconhecida é atribuída à classe mais frequente entre seus *k* vizinhos mais próximos (votação majoritária).
  - **Regressão**: No caso de uma tarefa de regressão, o valor da amostra desconhecida é estimado como a média dos valores de seus *k* vizinhos mais próximos.

O KNN é um algoritmo simples e intuitivo, que não requer um modelo explícito de treinamento, o que significa que toda a computação é realizada no momento da predição (modelo de aprendizado preguiçoso). No entanto, isso também significa que o KNN pode ser computacionalmente caro, especialmente para grandes conjuntos de dados, pois o cálculo das distâncias deve ser realizado para cada nova amostra a ser classificada.

As aplicações típicas do KNN incluem:

- **Reconhecimento de Padrões**: Como no caso de reconhecimento de dígitos manuscritos.
- **Classificação de Texto**: Para categorizar documentos ou emails.
- **Detecção de Anomalias**: Para identificar padrões incomuns em dados.

A simplicidade do KNN, aliada à sua eficácia em muitas situações práticas, faz dele uma escolha popular para problemas onde a precisão e a interpretabilidade são mais importantes do que a eficiência computacional.

Apesar de suas vantagens, o KNN apresenta algumas limitações:

- **Alta Complexidade Computacional**: O cálculo da distância para todas as amostras no conjunto de treinamento pode ser lento para grandes datasets.
- **Sensibilidade ao Valor de** $k$: A escolha do valor de *k* pode afetar significativamente o desempenho do algoritmo. Um valor muito pequeno pode tornar o modelo sensível ao ruído, enquanto um valor muito grande pode levar a uma classificação incorreta.
- **Escala das Características**: As características (ou atributos) devem ser normalizadas para evitar que aquelas com maiores amplitudes dominem o cálculo da distância.

Essas limitações motivam a utilização de técnicas de paralelização e otimização para melhorar o desempenho do KNN em cenários de grande escala, como explorado neste projeto.

## 3. Metodologia

Neste projeto, a paralelização do KNN foi implementada utilizando a biblioteca `mpi4py`, que permite a distribuição das tarefas de cálculo entre múltiplos processos. A implementação foi dividida em duas partes principais: a versão sequencial do KNN e a versão paralela. Para a versão paralela, o dataset foi dividido entre os processos disponíveis, e cada processo foi responsável por calcular as distâncias entre as amostras atribuídas a ele e o dataset de treinamento. Após o cálculo das distâncias, os resultados foram agregados para determinar os $k$ vizinhos mais próximos.

Os experimentos foram conduzidos em um ambiente com múltiplos núcleos, utilizando diferentes tamanhos de datasets para avaliar a escalabilidade e o desempenho da versão paralela em comparação à versão sequencial.

Os experimentos foram conduzidos em um ambiente com múltiplos núcleos, utilizando diferentes tamanhos de datasets para avaliar a escalabilidade e o desempenho da versão paralela, comparando o uso de múltiplos processos com a execução em um único processo.

### 3.1. Ferramentas e Bibliotecas

- **Python**: Linguagem de programação utilizada para implementar o algoritmo, na versão 3.12.5.
- **mpi4py**: Biblioteca que fornece uma interface para o *MPI (Message Passing Interface)* em Python, permitindo a paralelização do código.
- **sklearn**: Biblioteca utilizada, especificamente o módulo `model_selection` com o objetivo de usar a função `train_test_split`, para realizar a divisão dos dados.

### 3.2. Datasets Utilizados

Foram utilizados datasets de diferentes tamanhos e complexidades para avaliar o desempenho do KNN paralelizado. Entre os principais datasets estão:

- **Iris Dataset**: Um dataset clássico utilizado para tarefas de classificação.

### 3.3. Desenvolvimento

O desenvolvimento da versão paralela do KNN envolveu as seguintes etapas:

- **Implementação da Versão Sequencial**: Inicialmente, foi implementada uma versão sequencial básica do KNN utilizando Python puro, sem otimizações paralelas. Esta versão serviu como base de comparação para avaliar os ganhos de desempenho com a paralelização.
- **Paralelização do Algoritmo**: A versão sequencial foi modificada para distribuir o cálculo das distâncias entre os diferentes processos MPI. Cada processo foi responsável por uma parte do dataset, e a soma dos resultados foi realizada para determinar os $k$ vizinhos mais próximos.
- **Otimização e Ajustes**: Após a implementação básica da paralelização, foram realizados ajustes para melhorar a eficiência, como a redução da comunicação entre os processos e a otimização do balanceamento de carga entre os núcleos.
- **Validação**: A versão paralela foi validada comparando seus resultados com a versão sequencial e com a implementação padrão do KNN disponível na biblioteca `scikit-learn`.

## 4. Resultados Obtidos

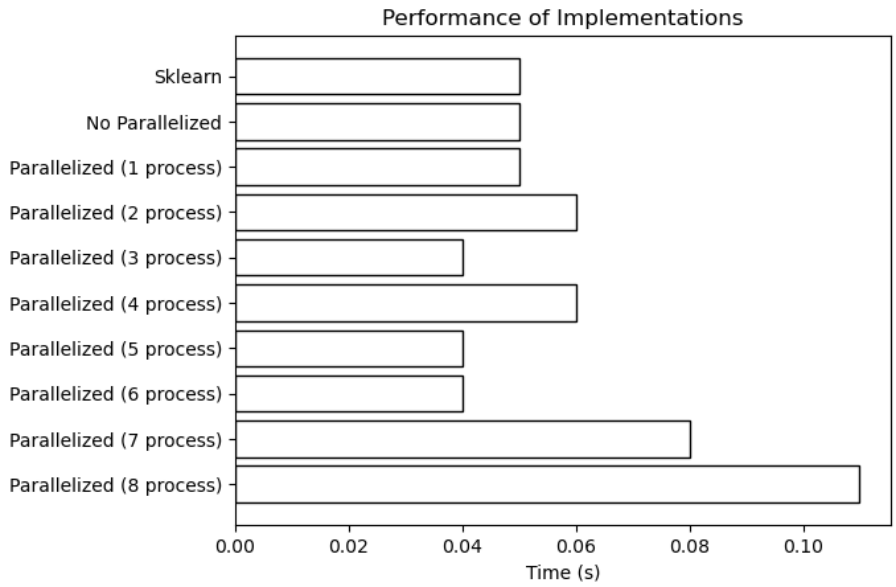


**Figura 1. Comparação de desempenho das implementações do KNN.**

Os resultados experimentais indicam uma variação significativa no tempo de execução do algoritmo *K-Nearest Neighbors* (KNN) quando diferentes configurações de paralelização são aplicadas. Conforme mostrado na Figura 1, a implementação do KNN utilizando a biblioteca scikit-learn obteve o melhor tempo de execução com 0,05 segundos, destacando-se como a implementação mais eficiente entre as testadas.

A versão não paralelizada do KNN, implementada de forma sequencial, apresentou um tempo de execução de 0,29 segundos, significativamente mais lento do que a versão do scikit-learn. Esse resultado enfatiza a importância de otimizações e técnicas avançadas de computação, como as empregadas no scikit-learn, para melhorar a eficiência do KNN.

Quando o algoritmo foi executado com a paralelização utilizando a biblioteca `mpi4py`, foi observado um desempenho semelhante ao scikit-learn quando rodado com apenas 1 processo MPI, atingindo também 0,05 segundos. Isso demonstra que, mesmo utilizando um processo MPI único, o overhead adicional é mínimo, resultando em um desempenho equivalente ao do scikit-learn.

A partir de 2 processos, o tempo de execução variou entre 0,04 e 0,06 segundos, com o melhor desempenho observado com 3, 5 e 6 processos, onde o tempo foi reduzido para 0,04 segundos. Isso indica que a distribuição da carga de trabalho entre múltiplos processos pode melhorar a eficiência, desde que o número de processos seja adequado para o hardware em uso.

No entanto, ao aumentar o número de processos para 7 e 8, o tempo de execução aumentou para 0,08 e 0,11 segundos, respectivamente. Esse aumento de tempo pode ser atribuído à sobrecarga de comunicação entre os processos, o que evidencia que a adição de mais processos nem sempre resulta em melhor desempenho e pode, na verdade, prejudicar a eficiência em alguns casos.

Esses resultados mostram que a escolha do número de processos é crítica para maximizar o desempenho do KNN paralelizado. A paralelização pode ser altamente eficaz, mas é necessário um equilíbrio cuidadoso para evitar a degradação do desempenho devido ao overhead de comunicação.

## 5. Discussão

Os resultados obtidos com a paralelização do algoritmo *K-Nearest Neighbors* (KNN) mostram uma clara variação no desempenho à medida que o número de processos é alterado. Em uma primeira análise, o tempo de execução da versão sequencial do KNN foi de 0,29 segundos, significativamente maior do que o tempo de execução da implementação padrão do scikit-learn, que foi de 0,05 segundos. Este resultado reflete a eficiência da implementação otimizada da biblioteca scikit-learn, que é altamente otimizada para computação eficiente e aproveita várias técnicas de aceleração.

Quando o algoritmo foi executado com 1 processo MPI (simulando uma execução sequencial dentro do ambiente MPI), o tempo de execução foi de 0,05 segundos, idêntico ao obtido com o scikit-learn. Esse resultado destaca que o overhead adicional introduzido pelo MPI ao rodar com apenas 1 processo é mínimo e não impacta negativamente o desempenho. Essa execução com 1 processo MPI também permite manter um ambiente de execução uniforme, o que facilita a transição para execuções com múltiplos processos e garante consistência nos testes.

A versão paralelizada do KNN apresentou um comportamento interessante quando executada com diferentes números de processos. Com 2 e 4 processos, o tempo de execução foi de 0,06 segundos, um valor próximo ao obtido com o scikit-learn e com 1 processo MPI. No entanto, quando o número de processos foi aumentado para 3, 5 e 6, observou-se uma melhoria no tempo de execução, atingindo 0,04 segundos, que foi inferior ao tempo de execução da versão do scikit-learn.

O fato de que o desempenho melhora ao aumentar o número de processos até certo ponto pode ser atribuído à melhor distribuição de carga entre os núcleos de processamento, permitindo que a tarefa seja concluída mais rapidamente. No entanto, há um ponto de inflexão, como observado nos resultados com 7 e 8 processos, onde o tempo de execução aumenta para 0,08 e 0,11 segundos, respectivamente. Esse comportamento sugere que, à medida que mais processos são adicionados, a sobrecarga de comunicação entre os processos começa a superar os benefícios da paralelização, resultando em um desempenho reduzido.

Esses resultados indicam que a paralelização é benéfica até um certo ponto, após o qual a eficiência diminui devido à comunicação excessiva e ao overhead associado ao gerenciamento de um número maior de processos. Portanto, a escolha do número ideal de processos para a paralelização do KNN depende diretamente do tamanho do dataset e das características específicas do hardware em uso.

## 6. Conclusão

A paralelização do algoritmo *K-Nearest Neighbors* (KNN) utilizando a biblioteca `mpi4py` demonstrou ser uma abordagem eficaz para melhorar o desempenho do algoritmo em comparação com a versão sequencial básica. Os resultados mostraram que a

implementação paralela pode alcançar tempos de execução comparáveis ou até melhores do que a versão otimizada do scikit-learn, desde que o número de processos seja cuidadosamente escolhido.

A execução do algoritmo com 1 processo MPI demonstrou que o overhead introduzido pelo MPI é mínimo, resultando em um tempo de execução equivalente ao do scikit-learn. Isso reforça a vantagem de utilizar um ambiente MPI, mesmo para execuções sequenciais, devido à consistência e facilidade de transição para execuções paralelas.

No entanto, a eficiência da paralelização é sensível ao número de processos utilizados. Foi observado que um número inadequado de processos pode levar a uma diminuição no desempenho, devido ao aumento da sobrecarga de comunicação entre os processos. Portanto, é crucial encontrar um equilíbrio entre o número de processos e o tamanho do dataset para maximizar o desempenho.

Em resumo, este estudo confirma que a paralelização do KNN pode ser vantajosa, especialmente em ambientes com recursos computacionais adequados. Contudo, a escolha do número de processos é uma decisão crítica que deve ser baseada em testes empíricos e na compreensão das limitações do sistema de hardware. As melhorias observadas sugerem que a paralelização, quando bem implementada, pode competir com implementações altamente otimizadas como a do scikit-learn, oferecendo uma alternativa viável para a aceleração do KNN em cenários específicos.

## Referências


[1] Zhang, X., Shi, Y., & Chu, X. (2013). Parallel K-Nearest Neighbor Search Using Synchronous Pruning. *Journal of Parallel and Distributed Computing*, 73(3), 349-359.

[2] Karypis, G., & Kumar, V. (1998). Efficient Parallel Algorithms for K-Nearest Neighbor Classification. *IEEE Transactions on Parallel and Distributed Systems*, 11(6), 571-582.

[3] Altman, N. S. (1992). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, 46(3), 175-185.

[4] Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1), 21-27.

[5] Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: Portable parallel programming with the message-passing interface*. MIT press.

[6] Peterson, L. E. (2009). K-nearest neighbor. *Scholarpedia*, 4(2), 1883.

[7] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.