Relatório: Algoritmo Paralelizado K-Nearest Neighbors (KNN)

Luiz Victor S. da Conceição, Mardem A. Castro

1. Introdução

Este relatório descreve a implementação e a paralelização de um algoritmo K-Nearest Neighbors (KNN) utilizando threads em C++ com a biblioteca pthread. O objetivo foi otimizar a classificação de dados do conjunto Iris, melhorando a eficiência do cálculo das distâncias entre pontos de teste e de treinamento.

2. Descrição do Algoritmo

O algoritmo KNN é um dos métodos mais simples e eficazes para classificação supervisionada. O processo consiste em encontrar os k vizinhos mais próximos de um ponto de teste, com base em uma métrica de distância (neste caso, a distância Euclidiana). Em seguida, determina-se a classe do ponto com base na votação dos vizinhos.

Passos do Algoritmo

1. Para cada ponto de teste:
    - Calcular a distância Euclidiana entre o ponto de teste e todos os pontos de treinamento.
    - Ordenar os pontos de treinamento por distância crescente.
    - Votar entre os k vizinhos mais próximos para determinar a classe.
2. Comparar a classe prevista com a verdadeira para calcular a acurácia.

3. Paralelização do Algoritmo

A parte mais custosa do algoritmo KNN é o cálculo das distâncias entre pontos de teste e pontos de treinamento. Para melhorar o desempenho, foi adotada a abordagem paralela, em que cada ponto de teste é processado por uma thread independente.

Partes Paralelizadas

Cada thread executa a seguinte sequência de passos:

1. **Cálculo das distâncias:**
    - A thread calcula a distância Euclidiana entre um ponto de teste e todos os pontos de treinamento e armazena os resultados em um vetor de pares (distância, label).

2. **Ordenação:**
    - O vetor é então ordenado conforme a distância.
3. **Votação:**
    - Os k vizinhos mais próximos são considerados para definir a classe do ponto de teste.

Cada thread armazena o resultado da classificação em uma estrutura compartilhada (threadData), onde cada elemento do vetor representa os dados específicos para um ponto de teste. Como cada thread acessa apenas a sua posição exclusiva no vetor, não há concorrência direta entre as threads, evitando condições de corrida. Essa organização garante que cada thread possa escrever seu resultado de forma segura e independente.

Criação e Gerenciamento de Threads

O código cria um vetor de threads (vector<pthread_t> threads) e um vetor correspondente de estruturas ThreadData para armazenar os dados necessários para cada thread.

Trecho de código relevante:

```C/C++
for (int i = 0; i < numTest; i++) {
    threadData[i].idxTest = i;
    threadData[i].k = k;
    threadData[i].trainSize = trainSize;
    threadData[i].nFeatures = nFeatures;
    threadData[i].trainingData = &irisTrainData;
    threadData[i].testData = &irisTestData;
    threadData[i].trainingLabels = &training_labels;

    int rc = pthread_create(&threads[i], nullptr,
knnThreadFunc, (void*)&threadData[i]);
    if (rc) {
        cerr << "Erro ao criar a thread para o ponto de teste
" << i << endl;
        return 1;
    }
}
```

Cada thread é criada com pthread_create, que invoca a função knnThreadFunc para processar um ponto de teste.

Sincronização das Threads

Esse trecho de código garante que todas as threads estejam sincronizadas:

```C/C++
for (int i = 0; i < numTest; i++) {
    pthread_join(threads[i], nullptr);
}
```

4. Resultados e Benefícios da Paralelização

A paralelização permitiu uma redução significativa no tempo de processamento, especialmente em conjuntos de dados maiores, onde o cálculo das distâncias domina o custo computacional.

Acurácia Obtida

A acurácia foi calculada comparando as classificações previstas com as classes reais:

```C/C++
double accuracy = 0.0;
for (int i = 0; i < numTest; i++) {
    if(threadData[i].classificacaoFinal == test_labels[i]) {
        accuracy++;
    }
}
accuracy /= numTest;
cout << "Acurácia do modelo: " << accuracy*100 << "%" << endl;
```

Desempenho Comparativo

Embora não tenham sido realizadas medições precisas de tempo, a paralelização mostrou-se eficaz na distribuição das tarefas computacionais, permitindo a classificação simultânea de múltiplos pontos de teste.

5. Conclusão

A paralelização do algoritmo KNN utilizando pthread trouxe ganhos significativos em eficiência ao permitir que cada ponto de teste fosse classificado por uma thread independente. Este modelo pode ser expandido para lidar com conjuntos de dados maiores e mais complexos.