Funcionamento do KNN

No algoritmo KNN, calcula-se a distância entre cada ponto no conjunto de dados de treinamento dado previamente e o elemento novo que será rotulado. Em seguida, as distâncias calculadas são ordenadas em ordem crescente, mantendo as classes dos pontos associadas a essas distâncias. O algoritmo seleciona os vizinhos mais próximos e escolhe um valor K, que será a quantidade de vizinhos que será levado em consideração para rotular o novo elemento de entrada. Por fim, as classes dos K vizinhos mais próximos determinaram qual será seu rótulo de acordo com a frequência que aparecem, o rótulo que tiver maior quantidade determinará o tipo de elemento.

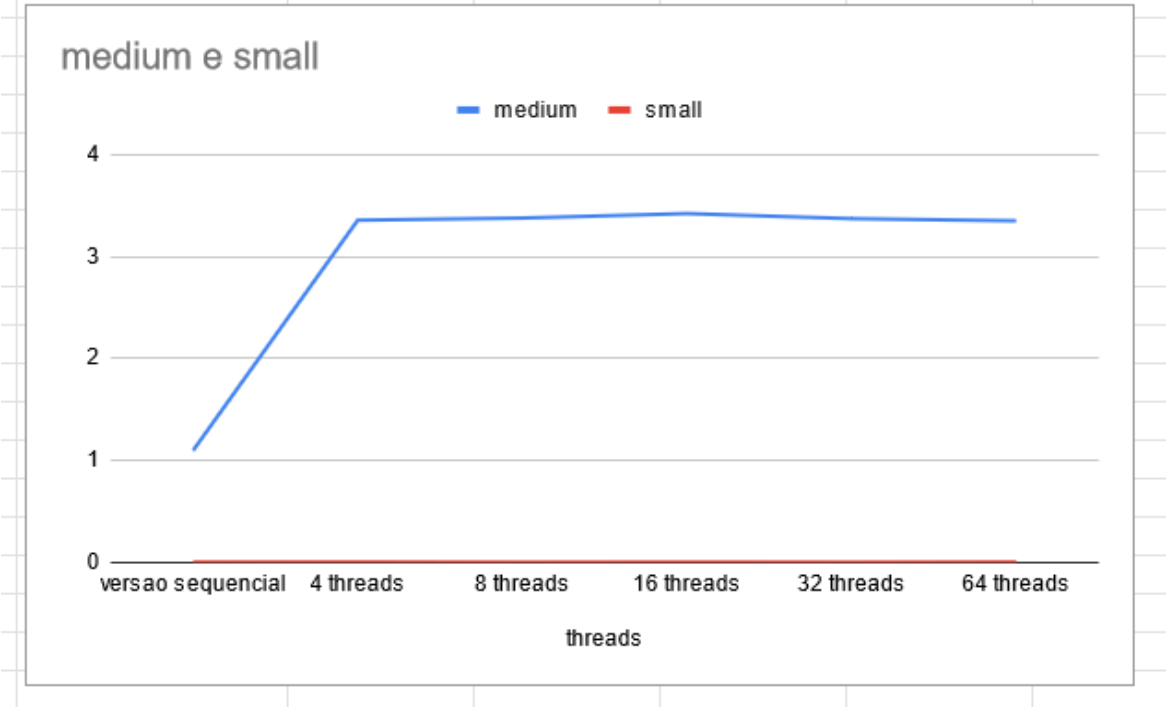
Paralelização do KNN

O algoritmo pega as amostras já classificadas (neste projeto, os arquivos de entrada são `medium.in` e `small.in`) e calcula a distância de cada um dos pontos em relação à nova amostra, onde tentamos paralelizar para que o cálculo das distâncias não precise ser feito sequencialmente. As distâncias são ordenadas em ordem crescente, e as amostras mais próximas são selecionadas. Este processo pode ser computacionalmente caro, especialmente para grandes conjuntos de dados. Para melhorar o desempenho, paralelizamos a parte mais intensiva do cálculo de distância, pois os cálculos são feitos de maneira independente com as amostras já classificadas.

Testes

Realizamos testes com 4, 8, 16, 32 e 64 threads para avaliar o desempenho e comparar a versão sequencial com a paralela. Observamos que a versão sequencial está mais performática que a versão paralela para os exemplos abaixo. Contudo, conforme aumentamos o número de threads e a entrada de dados (como no caso do `medium.in`), a versão paralela mostrou um pequeno ganho de desempenho.

threads	medium	small
versao sequencial	1,098	0,003
4 threads	3,356	0,004
8 threads	3,377	0,002
16 threads	3,421	0,003
32 threads	3,371	0,002
64 threads	3,35	0,004



https://docs.google.com/spreadsheets/d/1D3jYKMhjpFvfTgYAPZ9GU0r_HR5hpaw0qnAQbNrIAvs/edit#gid=0

Solução

O cálculo das distâncias da nova amostra para cada um dos pontos das amostras já classificadas em cada grupo, foi realizado paralelamente. Utilizou-se `#pragma omp parallel for` para paralelizar o loop que percorre todos os grupos e pontos. Dentro da seção paralela, foi utilizado `#pragma omp critical` para ter o acesso de apenas uma thread por vez e evitar condições de corrida ao atualizar as variáveis com memória compartilhada "`distances`" e "`labels`", que armazenam as distâncias mais curtas e seus respectivos rótulos. Após fazer o cálculo de todas as distâncias, os rótulos dos K vizinhos mais próximos são ordenados e assim é determinada a classe mais frequente entre os K vizinhos mais próximos para rotular a nova amostra.

Conclusão

Para as entradas fornecidas pelo desafio, o desempenho do modelo sequencial foi superior. No entanto, ao aumentar as entradas e o número de threads, houve um ganho pequeno, mas significativo na performance do algoritmo paralelo. A exclusão da diretiva `omp critical` resultou em um ganho significativo na performance, tornando-o alguns segundos mais rápido que a versão sequencial. Contudo, esta não é uma opção viável devido ao risco de condições de corrida, já que as variáveis `distances` e `labels` são compartilhadas.