Product Quantization for Nearest Neighbor Search
A parallel aproach

Marcelo de Araújo [1], **André Fernandes** [1]

[1]Departamento de Ciência da Computação - Universidade de Brasília(UNB)

Resumo. *O artigo baseia-se na ideia proposta por [Herve Jegou], onde o espaço é decomposto em vários subespaços de um produto cartesiano, produzindo vetores menores, que serão aproximados separadamente, e usados para a criação de uma lista invertida junto com uma base de dados contendo os códigos referentes a cada vetor da base, onde toda busca será feita por meio da lista invertida.*
Também será apresentada uma proposta de paralelização no ambiente distribuído, com o foco na parte de busca.

Introdução

Dados um vetor x, e um conjunto de vetores $Y \subset R^n$, queremos achar o vetor y do conjunto Y que mais se aproxima de x, chamando de $NN(x)$ o vizinho mais próximo e definido como:

$$NN(x) = \arg\min d(x, y)\ ,\ y \in Y \tag{1}$$

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2}$$

Onde $d(x, y)$ é a distância euclidiana entre x e y. Porém para conjuntos Y grandes seria muito custoso a busca exaustiva. Por isso a estratégia adotada em [1], tenta aproximar os vetores da base Y em outro conjunto de vetores, chamados centroides($c_i \in C$) aproximados com o algoritmo K-means a partir de um conjunto de treino.

Com os centroides conhecidos podemos definir formalmente como $q(.)$ a função que mapeia um vetor arbitrário $x \in R^n$ em $q(x) \in C = \{c_i\ ;\ i \in I\}$, onde I é um intervalo finito, $I = \{0, \cdots, k-1\}$ e c_i são centroides.

$$q(x) = \arg\min d(x, c_i)\ ,\ c_i \in C \tag{3}$$

Além de aproximar os vetores y da base em seus centroides mais próximos, centroides são criados a partir de subvetores, e assim vetores y são divididos em partes de dimensão $d = \frac{n}{m}$ e assinalada a cada subdimensão do centroide.

$$\begin{gathered} y = \{y_1, y_1, \cdots, y_n\},\ \text{seus respectivos subvetores } u_i \\ u_1 = \{y_1, y_2, \cdots, y_d\},\ u_2 = \{y_{d+1}, y_{d+2}, \cdots, y_{2d}\} \\ u_m = \{y_{n-d}, y_{n-d+1}, \cdots, y_n\},\ u_i \in R^d \end{gathered} \tag{4}$$

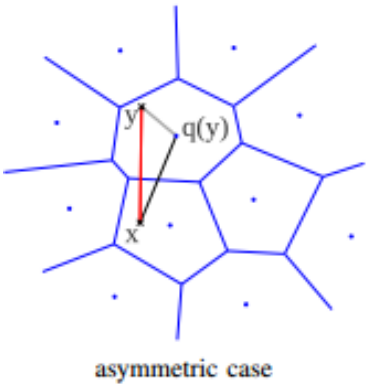

Figure 1. Centroides e vetores.

E seus respectivos centroides de seus subespaços:

$$q(y) = \{q(u_1), q(u_2), \cdots, q(u_m)\},\ q(u_i) \in C \tag{5}$$

Lista Invertida

Com a finalidade de tornar a busca mais eficiente uma estrutura de lista invertida foi utilizada por [Herve Jegou].

Para montar a lista são usados dois conjuntos de centróides C_1 e C_2, onde C_1 representa os centroides assinalados a base de treino T, chamados em [Herve Jegou] por *coarse centroids*, e após conhecidos, C_2 é calculado e são os centróides assinalados ao resto , $r(t)$, dos vetores de treino com cada um de seus centróides.

$$\begin{gathered} q(t) \in C_1 \\ r(t) = y - q(t),\ y \in T \\ q(r(t)) \in C_2 \end{gathered} \tag{6}$$

Com os conjuntos C_1 e C_2 conhecidos, podemos montar a estrutura da lista em si, indexando os vetores de uma base Y na lista, da seguinte forma:

Cada entrada da lista representa um centróide de C_1 e cada entrada da lista contida representa o centroide de C_2 possuindo os identificadores dos vetores y da base que possuem aquele centróide como o mais próximo.

Algoritmo

Aprendizagem

Primeiramente o algoritmo necessita aprender os centróides c_i dos dois conjuntos C_1 e C_2, para sabermos a função $q(.)$, e realiza isto na parte de aprendizagem, onde a partir de uma base de treino T os conjuntos são aprendidos com o algoritmo K-means.

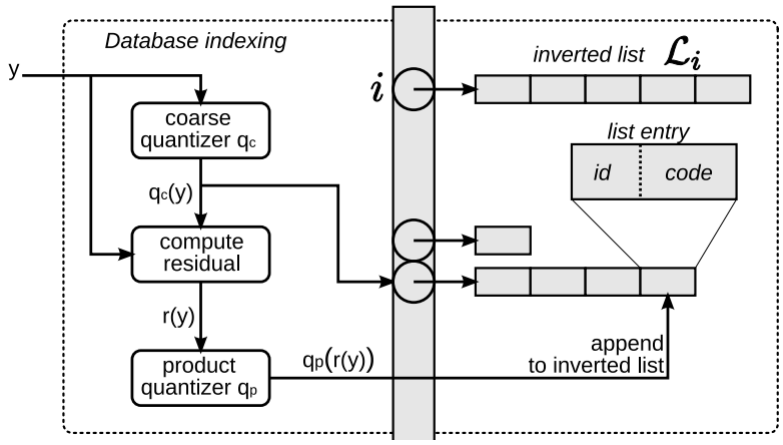

Figure 2. Processo de indexação.

Indexação

A figura 2 representa o processo de indexação de uma base de dados Y, onde e feito da seguinte forma:

- Para cada vetor $y_i \in Y$ calculamos seu centroide mais próximo $c_i \in C_1$, assim sabemos a entrada da lista principal;
- Calculamos $r(y_i)$ conforme (7) e calculamos o centroide mais próximo $q(r(y_i)) = c_j \in C_2$, para cada subdimensão;
- Agora que temos o código para cada $c_j \in C_2$, guardamos na entrada correspondente junto com o identificador do vetor.

Busca

Durante a busca, como dito na secção 1, queremos buscar o vizinho mais próximo de um determinado vetor x, ou k vizinhos mais próximos dele.

- Procuramos o centroide $c_i \in C_1$ mais próximo de x, agora sabemos qual entrada da lista possui vetores associados ao mesmo centroide;
- Calculamos o $r(x)$ e usamos para calcular a $d(r(x), c_j),\ c_j \in C_2$, para cada subdimensão;
- Somamos as distâncias das subdimensões de interesse, aquelas cujos c_j se encontram na entrada da lista descoberta no primeiro passo;
- Com as distâncias podemos procurar as k distância mínimas, gerando uma lista L a de possíveis canditados da base Y próximos a x, que são encontrados pelos seus identificadores presentes nas entradas de cada lista.

Solução Paralela

A abordagem paralela adotada se baseia em uma fila de execução para sistemas distribuídos, decompondo PQNNS em quatro estágios (figura 4).

Os estágios de leitura da base e de recebimento da *query* são responsáveis pela criação dos centroides e assinalam os vetores da *query* aos centroides correspondentes, criando a lista invertida. Enquanto os estágios de busca dos vizinhos mais próximos e

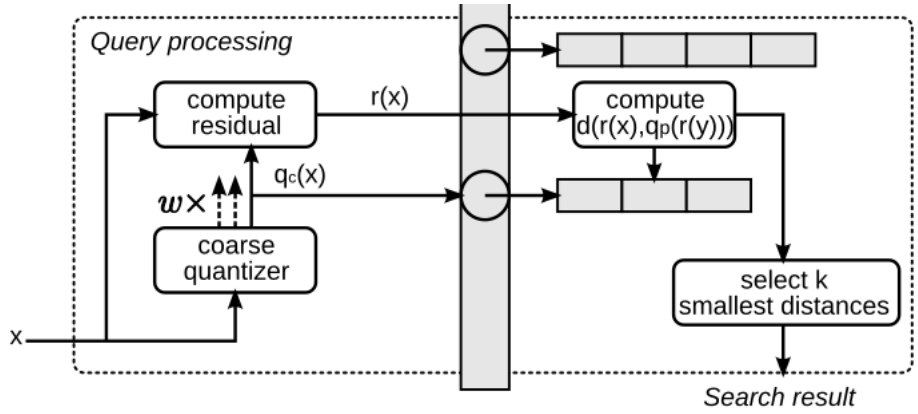

Figure 3. Processo de busca.

de agregação encontram os vetores correspondentes na base e agregam os resultados da busca.

O gerenciamento dos processos e a comunicação entre os estágios é feita pela ferramenta MPI, que rotula as mensagens baseado em seus destinos e as distribui.

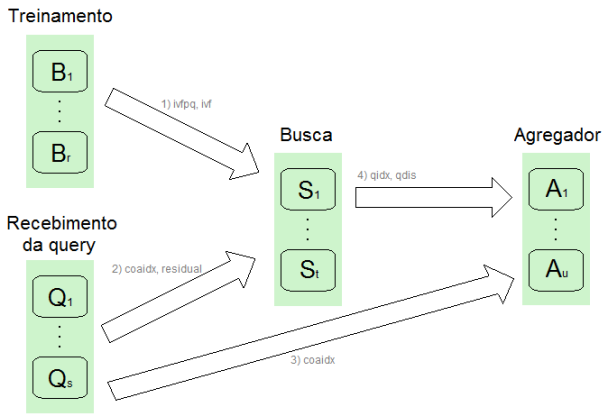

Figure 4. Fila de execução.

Treinamento

O estágio de treinamento lê as bases de vetores e aprende os centroides dos conjuntos C_1 e C_2. Os centroides encontrados são usados para criar a lista invertida que assinala os vetores da base.

Os dados criados são enviados para os próximos estágios, os centroides são enviados para o estágio de recebimento da query e para o estágio de busca. A lista invertida é enviada ao estágio de busca, no qual cada processo recebe um trecho da lista.

Recebimento de query

O estágio de recebimento da query faz a leitura dos vetores y_i da query e os assinala aos centroides mais próximos no conjunto C_1. É responsável também pelo cálculo dos resíduos $r(y_i)$ conforme (6).

Os índices e resíduos de cada vetor da query são enviados para o estágio de busca, onde cada processo recebe dados que estejam assinalados aos centroides pelos qual ele é responsável.

Os dados de cada vetor da query serão enviados para os processos responsáveis pelos w centroides mais próximos.

Busca

O estágio de busca procura os vetores mais próximos de um determinado vetor. Cada processo S_h $(0 < h \leq u)$ é responsável por um conjunto C_3 de centroides que é determinado por meio de uma função de hash.

$$\begin{gathered} c_1 \in C_1 \\ c_l = c_1 \mod u \\ c_l \in C_3 \end{gathered} \tag{7}$$

Ao receber os resíduos de um vetor da query, o processo procura os k vetores mais próximos assinalados pelo centroide correspondente. Os índices dos vetores mais próximos e as distâncias até o vetor da query são enviados para o estágio de agregação.

Agregação

O estágio de agregação recebe os resultados para cada vetor da query, que correspondem a wk vetores da base mais próximos. São deterinados os k vetores mais próximos à query e os resultados de todas as consultas são agregados.

Resultados experimentais

Vamos avaliar os resultados da implementação paralela proposta na seção anterior.

Os testes foram executados em uma máquina ...

Foram utilizadas dois conjuntos de vetores nos testes, disponibilizados por [Herve Jegou]. Ambos contém vetores obtidos pelo algoritmo SIFT.

O conjunto denominado por *siftsmall* contém uma base de dados menor e é consultado menos vezes que a base *sift* (tabela 1).

Table 1. Conjuntos de vetores

Conjunto	Dimensão	Tamanho da base	Tamanho da query	Tamanho da base de treinamento
Siftsmall	128	10000	100	25000
Sift	128	1000000	10000	100000

Os estágios da implementação paralela permitem o uso de vários processos para serem executados, mas nos limitamos a variar o número de processos do estágio de busca.

Os testes buscaram mostrar o impacto da implementação paralela no estágio de busca, observar suas limitações e buscar melhorias.

Impacto do número de processos no tempo de busca

O primeiro teste foi feito usando a base *siftsmall* e foram criados entre 4 e 8 processos, sendo que três dos processos são dedicados ao treinamento da base, ao recebimento da *query* e ao agregador e os processos restantes são dedicados à busca.

Table 2. Tempo de execução e *speedup* - *siftsmall*

Processos	Tempo (ms)	Speedup
4	33.017	1
5	16.763	1.969
6	12.909	2.558
7	11.343	2.911
8	8.946	3,691

A tabela 2 mostra um ganho em desempenho com o aumento do número de processos. Considerando que a execução com 4 processos é executada sequencialmente, percebe-se um *speedup* de 3,691 na execução com 8 processos, ou seja, não se assemelha ao esperado (5).

A partir do gráfico da figura 5 pode-se notar que a variação no desempenho não é linear e isso em parte se deve à base pequena utilizada, justificando os testes com a base maior.

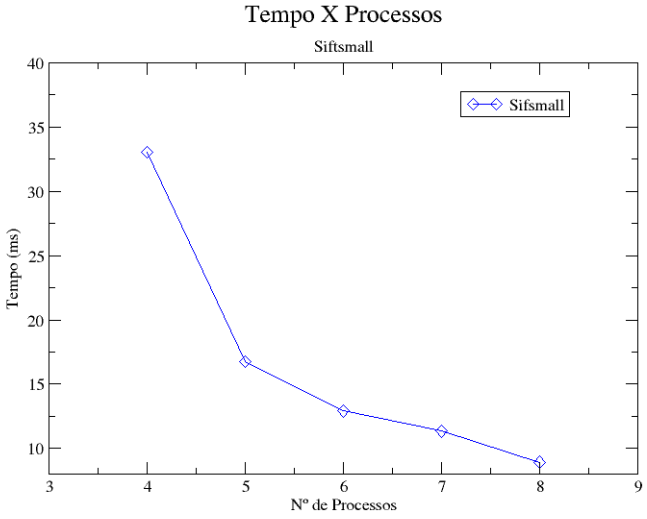

Figure 5. Impacto da variação do número de processos no tempo da busca na base *siftsmall*

Os mesmos testes foram executados com o conjunto de dados *sift*, que contém uma base maior. A tabela 3 e o gráfico da imagem 6 mostram um comportamento mais linear no desempenho do algoritmo, apesar de não atingir um *speedup* linear.

Variação no número de centroides

A variação no número de processos mostra um ganho considerável em desempenho e justifica o uso de uma execução do algoritmo em um ambiente distribuído. No entanto

Table 3. Tempo de execução e *speedup* - sift

Processos	Tempo (s)	Speedup
4	12.505	1
5	6.583	1.899
6	4.705	2.657
7	3.706	3.374
8	2.965	4.217

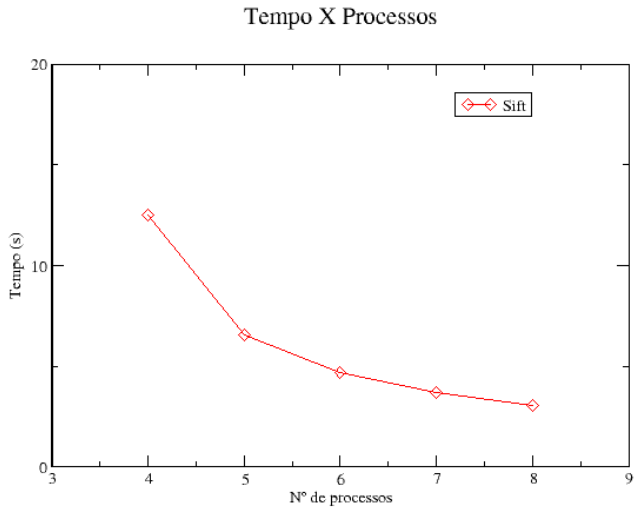

Figure 6. Impacto da variação do número de processos no tempo da busca na base *sift*

foram encontradas algumas limitações na aplicação do algoritmo, todos os testes foram executados com 256 *coarse centroids*, o que limita o uso da aplicação a 256 processos de busca.

Para expandir o uso do algoritmo, foram feitos testes no conjunto de vetores *sift*, variou-se o número de *coarse centroids*, de centroides e o de *coarse centroids* a serem visitados em cada busca.

O gráfico da imagem 7 a perda de precisão na busca com o aumento do número de *coarse centroids* e mostra também que a variação no número de centroides não afeta tanto.

Uma solução para reduzir a perda de precisão é o aumento no campo de busca, ao se dobrar o número de *coarse centroids* a serem usados na busca (w) o desempenho com o quádruplo de *coarse centroids* ainda é ligeiramente melhor, compensando totalmente a perda causada pelo aumento.

O gráfico da imagem 8 mostra que o aumento de w impacta linearmente no tempo de execução.

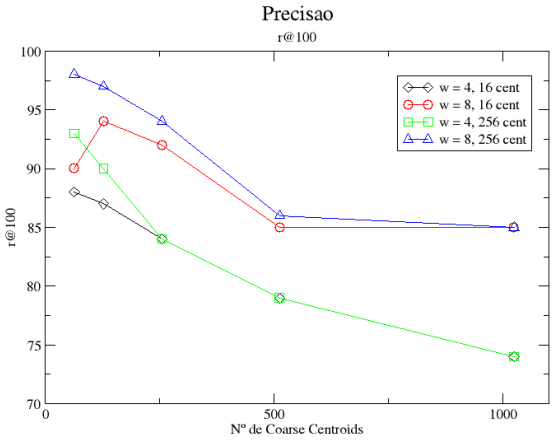

Figure 7. Impacto da variação do número de *coarse centroids* na performance da busca com um recall@100.

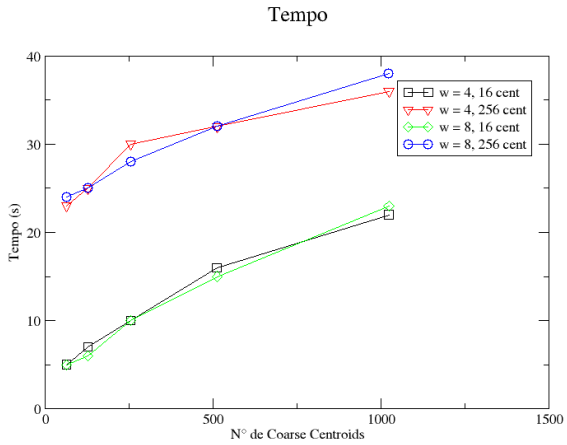

Figure 8. Impacto da variação do número de *coarse centroids* no tempo de execução.

Conclusão

Nós apresentamos uma abordagem paralela do algoritmo *Product Quantization for Neares Neighbor Search* para sistemas distribuídos. A proposta atinge bons *speedups* na busca. Apesar das limitações que podem ser geradas pelo número de *coarse centroids*, elas podem ser resolvidas com o ajuste de alguns parâmetros e assim seu uso ainda é justificado. A nossa abordagem replica os resultados atingidos no algoritmo sequencial e é uma solução escalável.

References

Herve Jegou, Matthijs Douze, C. S. Product quantization for nearest neighbor search. 33(1):117–128.

[Herve Jegou]