Paralelização e Otimizações do Algoritmo de Indexação de Dados Multimídia baseado em Quantização

André Fernandes[1], **George L. M. Teodoro**[1,2]

[1]Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Brasília – DF – Brasil

[2]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

andreff@aluno.unb.br, george@dcc.ufmg.br

Abstract. *In this paper is presented an efficient parallelization of the similarity search algorithm Product Quantization Approximate Nearest Neighbor Search (PQANNS). This method is capable of answering queries with a reduced memory demand and, coupled with a distributed memory parallelization proposed here, can efficiently handle very large datasets. The execution using 128 nodes/3584 CPU cores has attained a parallel efficiency of 0.97 with a dataset of 256 billion SIFT vectors.*

Resumo. *Nesse artigo é apresentada uma paralelização eficiente do algoritmo de busca por similaridade Product Quantization Approximate Nearest Neighbor Search (PQANNS). Esse método pode responder consultas com uma demanda reduzida de memória e, juntamente com a paralelização proposta, pode lidar de forma eficiente com grandes bases de dados. A execução utilizando 128 nós/3584 núcleos de CPU foi capaz de atingir uma eficiência do paralelismo de 0.97 em uma base de dados contendo 256 bilhões de descritores SIFT.*

1. Introdução

A busca por similaridade é uma operação crucial em diversas aplicações em recuperação de dados multimídia [Jain 2014, Böhm et al. 2001] e consiste em encontrar os objetos mais similares de um banco de dados a uma consulta. Para sua realização, os dados são representados por vetores de alta dimensionalidade que descrevem computacionalmente seu conteúdo. Assim, vetores mais próximos entre si descrevem dados mais semelhantes. No entanto, o aumento no volume dos dados e a alta dimensionalidade dos mesmos torna inviável a realização de uma busca exaustiva. Com isso tornou-se necessária a utilização de algoritmos de indexação e estruturas de dados que reduzam o espaço de busca.

Uma solução comum é o particionamento dos objetos da base espacialmente, seguido pela busca em trechos que efetivamente contém objetos mais similares. Em cima disso são utilizados diferentes métodos de indexação, como *kd-trees* [Friedman et al. 1977], *k-means trees* [Muja and Lowe 2009], dentre outros.

Já para endereçar o problema da alta dimensionalidade é utilizada busca aproximada (*Approximate Nearest Neighbors* - ANN), que tem como ideia central encontrar os vizinhos mais próximos com grande probabilidade ao invés da busca exata. Diversos algoritmos utilizam essa abordagem, como o *Fast Library for Approximate Nearest Neighbors* (FLANN) [Muja and Lowe 2009], o *Local Sensitive Hashing* (LSH) [Gionis et al. 1999] e o *Product Quantization for Approximate Nearest Neighbors Search* [Jegou et al. 2011].

Apesar dessas propostas fornecerem bons compromissos entre o desempenho na execução e a qualidade da busca, elas focam na execução sequencial que por sua vez não consegue atender aplicações que trabalham com grandes bases de dados. Por isso é importante o uso de uma arquitetura que permita o processamento paralelo das consultas em diversas máquinas. Além disso, processadores modernos contam com diversos núcleos de processamento, o que torna interessante uma implementação que faça uso de todos os núcleos de forma concorrente.

Dentre os algoritmos aproximados estudados, o PQANNS obteve compromissos melhores entre tempos de execução, uso de memória e qualidade de busca. Por isso a paralelização hierárquica desse algoritmo para sistemas distribuídos. A paralelização e resultados apresentados nesse trabalho foram desenvolvidos exclusivamente pelo aluno André Fernandes. Esses resultados foram publicados no artigo [Andrade et al. 2019] com outros coautores, os quais adicionaram contribuições diversas das discutidas aqui, como o ajuste automático de paralelismo.

2. Referencial Teórico

É muito comum a adoção de duas abordagens para minimizar o custo da busca por similaridade: (i) o uso de algoritmos de indexação e estruturas de dados que reduzam o espaço de busca, particionando os objetos espacialmente e (ii) a busca aproximada de vizinhos mais próximos (ANN), que substitui o cálculo exato por uma busca pelos vizinhos provavelmente mais próximos.

Diversos métodos foram empregados para reduzir o espaço de busca na recuperação de imagens. Algumas das abordagens da área utilizam árvores k-d [Friedman et al. 1977, Silpa-Anan and Hartley 2008], árvores *k-means* [Muja and Lowe 2009], árvores de cobertura [Beygelzimer et al. 2006], dentre outros. Nesses casos, as estruturas tentam agrupar nos nós folhas os vetores de características próximos no espaço e então avaliar a semelhança apenas em um conjunto de nós folhas.

Dentre o grupo de algoritmo que utilizam busca aproximada como forma de lidar com a dimensionalidade, podemos destacar o FLANN [Muja and Lowe 2009], LSH [Gionis et al. 1999], *Multicurves* [Valle et al. 2008] e PQANNS [Jegou et al. 2011]. Todos esses algoritmos apresentam bons compromissos entre a qualidade da resposta e velocidade da busca. O LSH utiliza funções de *hash* para mapear os vetores da base para chaves de *hash* de menor dimensionalidade, o *Multicurves* utiliza múltiplas curvas de preenchimento para projetar subespaços dos dados em pontos unidimensionais, reduzindo assim a dimensionalidade dos dados, o FLANN seleciona automaticamente o melhor entre diferentes algoritmos de busca e o PQANNS reduz a dimensionalidade dos dados de entrada através de quantização. Por ser nosso alvo para paralelização, esse algoritmo é explicado em detalhes na Seção 3.

Trabalhos recentes propuseram a paralelização de alguns desses métodos de indexação, dentre eles pode-se destacar as paralelizações do LSH utilizando o MapReduce [Stupar et al. 2010, Bahmani et al. 2012]. A formulação em MapReduce do LSH no trabalho de Stupar *et al.* [Stupar et al. 2010] tem: (i) uma fase de mapeamento que visita independentemente partições dos dados com valor de hash iguais aos da consulta de entrada, e, (ii) uma fase de redução para agregar os resultados de todos as partições visitadas. Como reportado pelos autores, as combinações de parâmetros LSH podem criar uma grande quantidade de arquivos e reduzir o desempenho geral do sistema. Além disso, essa

solução armazena o conteúdo de cada tabela de *hash* usada, ao invés de ponteiros como no algoritmo original. Com isso a base de dados inteira é replicada inúmeras vezes e uma configuração eficiente do LSH pode necessitar de um número proibitivo de tabelas.

Bahmani *et al.* [Bahmani et al. 2012] implementou outra variante do algoritmo LSH baseado em MapReduce, o *Layered* LSH. Eles propuseram limites teóricos de tráfego assumindo que uma única tabela de *hash* é usada, sendo que o uso de diversas tabelas tornaria o particionamento dos dados mais complexo. Dessa forma, nenhuma das paralelizações do LSH baseadas em MapReduce resolvem o problema de construir uma solução de indexação em larga escala que minimiza a comunicação e evita replicação de dados, enquanto preserva o comportamento do algoritmo original. Como apresentado na Seção 4, a paralelização proposta por nós endereça essas limitações.

3. *Product Quantization for Approximate Nearest Neighbor Search* - PQANNS

O cálculo de distâncias euclidianas entre vetores de alta dimensão é fundamental na busca por vizinhos mais próximos, pois é a principal métrica utilizada para definir a semelhança entre vetores. No entanto, esse processo é encarecido com o aumento da dimensionalidade. Para contornar esse problema é possível fazer um cálculo aproximado das distâncias pela quantização dos mesmos.

O quantizador é uma função que mapeia um vetor x de dimensão D, tal que $x \in \mathbb{R}^D$ para um vetor $q(x) \in C = c_i; i \in I$, onde o conjunto de índices I é finito: $I = 0...k-1$, os valores c_i são os centroides e o grupo de centroides C é o *codebook* de tamanho k. O *codebook* C compõe um diagrama de Voronoi, de forma em que cada centroide c_i pertence a uma célula do diagrama. Sendo assim, cada vetor do conjunto V é reconstruído a partir do centroide que representa a célula a que ele pertence.

Considerando a quantização de um espaço contendo vetores de 128 dimensões, como o descritor *Scale Invariant Feature Transform* (SIFT), um quantizador produzindo códigos de 64 bits, contém 2^{64} centroides. Dessa forma, não é viável utilizar o *k-means* para fazer a clusterização, dada a quantidade de amostras e a complexidade de aprender o quantizador. Para contornar esse problema, a quantização do produto divide o vetor x, de dimensão D, em m subespaços que são quantizados separadamente (3). Cada subvetor u_j, $1 \leq j \leq m$ com $D^* = D/m$ dimensões. Assim a quantização de x pode ser representada por:

$$\underbrace{x_1, ..., x_{D^*}}_{u_1(x)}, ..., \underbrace{x_{D-D^*+1}, ..., x_D}_{u_m(x)} \quad \rightarrow \quad q_1(u_1(x)), ..., q_m(u_m(x)) \quad (1)$$

Tal que o resultado desse processo é o produto cartesiano de seus subvetores quantizados: $q(y) = q_1(u_1) \times q_2(u_2) \times ... \times q_m(u_m)$. Assim, a busca pelos vizinhos mais próximos é executada no espaço quantizado usando os índices do *codebook*, calculando a distância entre o vetor de consulta e os vetores da base de dados. Os autores de [Jegou et al. 2011] propõem dois métodos para aproximar a distância entre o vetor de consulta x e os valores quantizados da base ($q(y)$), um simétrico e um assimétrico.

No cálculo de distância simétrica (SDC), ambos os vetores x e y são representados por seus respectivos centroides $q(x)$ e $q(y)$ e a distância aproximada é então obtida

por: $\hat{d}(x, y) = d(q(x), q(y))$. Já o cálculo da distância assimétrica (ADC) utiliza o valor quantizado dos vetores da base ($q(y)$), mas o vetor de consulta x não é quantizado. A distância aproximada dada por: $\tilde{d}(x, y) = d(q(x), q(y))$. ADC busca melhorar a qualidade da aproximação ao usar x ao invés de $q(x)$ para calcular as distâncias. A busca e representação dos dados quantizados reduz drasticamente os requisitos de memória.

Para evitar a busca exaustiva, Jégou *et al.* propõem um método que combina um sistema de lista invertida com o cálculo assimétrico de distância, IVFADC. Nesse esquema, uma lista invertida agrupa os descritores que são similares em uma mesma entrada. As entradas da lista invertida são representadas por *coarse centroids*, que também são aprendidos usando o algoritmo de clusterização *k-means* em uma base de treinamento.

Cada entrada da lista invertida é associada a um *coarse centroid*, que armazena vetores da base que são os mais próximos a aquele centroide que qualquer outro. Durante a busca, a consulta de entrada é comparada aos coarse centroids e as listas dos centroides mais próximo são visitadas para computar ADC e selecionar os resultados. A fase de busca dos vizinhos mais próximos de x é executada nos seguintes passos:

1. O vetor x é quantizado para os w vizinhos mais próximos do conjunto de *coarse centroids* (q_c). O algoritmo usa w elementos para permitir a busca em diversas entradas da lista invertida, o que pode ser necessário caso uma entrada não consiga atingir a qualidade necessária. Os próximos passos são repetidos para cada uma das w entradas da lista invertida;
2. computa a distância entre cada subquantizador j e o centroide associado;
3. calcula a distância entre $r(x)$ e os elementos contidos naquela entrada da lista invertida;
4. recupera os k vizinhos mais próximos de x baseado nas distâncias calculadas no passo anterior.

Devido as limitações de espeço, para mais detalhes sobre o algoritmo, recomendamos a consulta ao trabalho original do PQANNS [Jegou et al. 2011].

4. Paralelização do PQANNS

Essa seção descreve a paralelização do algoritmo PQANNS para máquinas com memória distribuída e equipadas com CPUs multicore.

A estratégia de paralelização adotada consiste no particionamento uniforme da base de dados entre as máquinas. Cada nó mantém uma lista invertida que indexa um trecho da base de dados e as consultas são enviadas para todos os nós de busca, que executam a consulta localmente e enviam os resultados para os nós responsáveis por sua agregação e pelo retorno dos resultados globais. A distribuição dos dados é feita num estilo *round-robin*, o que evita o desbalanceamento de carga e permite trabalhar com uma grande quantidade de dados.

Nossa paralelização foi desenvolvida em cima do MPI, utilizando a versão 3.1 da implementação da Intel. A aplicação foi decomposta em estágios baseados em fluxo de dados, de forma que esses estágios se comuniquem por fluxos direcionados. Os quatro estágios criados são: Leitura da Entrada, responsável pela leitura da base de dados e dos centroides, além de fazer o particionamento dos dados. Entrada de Consultas, recebe o fluxo de consultas, os processa e as direciona para a busca. Busca no Índice, faz a indexação e busca dos dados locais de cada cópia, calculando os resultados parciais de

cada consulta. Agregação, recebe os vizinhos mais próximos locais de cada cópia de Busca no Índice e retorna o resultado global. Cada estágio pode ser replicado em um sistema de memória distribuída (Figura 1) e são distribuídos em dois fluxos de execução, que fazem a construção do índice e a busca na base.

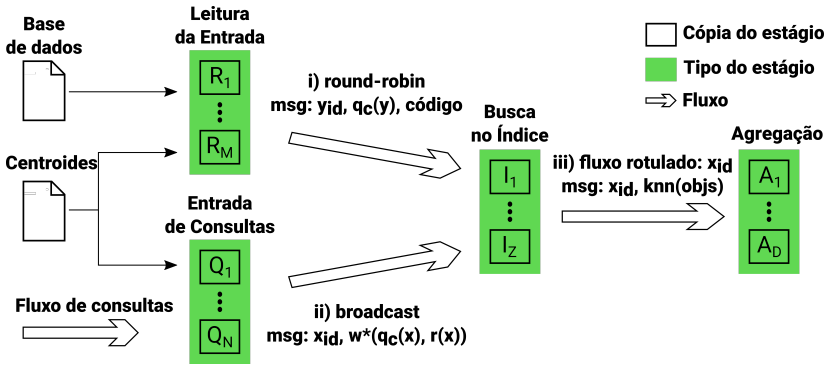


Figura 1. Arquitetura da paralelização.

O fluxo de construção do índice envolve os estágios de Leitura da Entrada e de Busca no Índice. Nessa fase, as cópias de Leitura de Entrada fazem a leitura da base de dados e dos centroides e a quantização de cada vetor y a ser indexado. O ID do vetor ($y_i d$), seu código quantizado ($q(r(y))$) e o *coarse centroid* ($q_c(y)$) assinalado a ele são enviados para o estágio de Busca no Índice. O envio segue um estilo *round-robin*, distribuindo igualmente os dados entre as máquinas.

Cada cópia do estágio de Busca no Índice recebe os dados correspondentes ao trecho da base que vai indexar e constrói uma lista invertida contendo o código quantizado dos vetores da base e seu id, recebidos do estágio anterior. Os *coarse centroids* são utilizados na inserção dos elementos na lista invertida, de forma que os dados de um objeto sejam incluídos na entrada da lista invertida que corresponde ao seu centroide associado.

A fase de busca do PQANNS passa por três estágios: Entrada de Consultas, Busca no Índice e Agregação. No estágio de Entrada de Consultas é feita a leitura do fluxo de vetores de consulta, para cada vetor recebido é feita a quantização para o w centroides mais próximos, e o resultado da quantização é enviado para o estágio de Busca no Índice por *broadcast*. O *broadcast* causa pouco impacto na escalabilidade do fluxo de dados, pois o gasto computacional é dominado pela busca no índice, e a comunicação é feita em segundo plano por *threads* de comunicação.

Após o recebimento da mensagem, cada cópia do estágio de Busca no Índice calcula as distâncias entre o código associado ao vetor de consulta e os códigos contidos nas w entradas da lista invertida. As distâncias e os índices dos vetores são organizado em ordem crescente de distância e os k vetores mais próximos são enviados para o estágio de Agregação, bem como o índice do vetor de consulta ao qual estão associados.

O estágio de Agregação recebe os dados de todas as cópias do estágio de Busca no Índice por um "fluxo rotulado". Essa política de comunicação associa um rótulo ou uma tag às mensagens (em nosso caso $x_i d$), que é utilizado para encaminhar todas as mensagens com a mesma tag para uma mesma cópia de Agregação. Esse mapeamento é feito por uma função de *hash* que utiliza a tag como parâmetro de entrada. Essa função retorna um valor que corresponde ao identificador das cópias de Agregação (um valor entre 1 e D, veja a 1).

Por fim, o estágio de Agregação reúne os resultados locais, os organiza em ordem crescente de distância e os reduz para os k vetores mais próximos do vetor de consulta. O uso do "fluxo rotulado" para a troca de mensagens permite a redução paralela dos resultados da busca calculado pelas cópias do Busca no Índice, já que várias cópias de Agregação podem ser executados no ambiente.

Dentre esses estágios, o de Busca no Índice é o mais caro computacionalmente, logo também paralelizamos ele internamente para executar em múltiplos núcleos em cada nós de computação. Isso também reduz a quantidade de cópias desse estágio no ambiente de execução, que por sua vez diminui o tráfego de dados na rede. Esse implementação utilizou a biblioteca OpenMP, versão 3.1. Para explorar paralelismo nesse nível, dividimos o processamento dos vetores de consulta entre as threads disponíveis para execução.

5. Avaliação Experimental

Os experimentos foram executados em uma máquina de memória distribuída com 128 nós interconectados via um switch FDR Infiniband. Cada nó contém 2 processadores Intel Haswell E5-2695 v3 CPU, 128 GB de RAM e executa Linux. Primeiramente comparamos o PQANNS com outros trabalhos na literatura utilizando uma base de 1 milhão de vetores SIFT. Posteriormente, avaliamos a escalabilidade da nossa paralelização com uma base de até 256 bilhões de vetores SIFT.

5.1. Comparação com o Estado da Arte: FLANN

Uma diferença essencial entre o FLANN e o IVFADC/PQANNS é que o FLANN mantém todos os vetores na memória RAM, pois ele executa uma fase de rerranqueamento que computa a distância real entre o vetor de consulta e os candidatos a vizinhos mais próximos. Enquanto no PQANNS são mantidos apenas os valores quantizados, reduzindo significantemente a demanda de memória.

Nossa avaliação apresenta o resultado para 1-recall@1, isto é, a proporção média de vizinhos mais próximos nos vetores de retorno [Muja and Lowe 2009]. Em ambos os algoritmos, PQANNS e FLANN, foram executados 10 mil consultas em uma base contendo 1 milhão de vetores.

Os algoritmos foram configurados para comparar o tempo de execução da busca para resultados em diferentes níveis de qualidade. Os parâmetros do FLANN são escolhidos automaticamente pela ferramenta, dado um nível de precisão esperado. Para o IVFADC foi variado o w (número de entradas da lista invertida verificadas na busca) e o número de coarse centroids.

O resultado experimental comparando os tempos de busca do IVFADC/PQANNS e do FLANN enquanto a precisão é variada é mostrado na Figura 2. Com exceção da execução utilizando 1024 centroides e $w = 4$, o PQANNS é mais eficiente em todos os casos, obtendo menores tempos de execuções para a mesma precisão. Além disso, PQANNS utiliza cerca de 25 MB de RAM para realizar buscas, enquanto o FLANN requer mais de 600 MB. O uso reduzido de memória e os bons tempos de execução fazem do PQANNS a melhor opção para aplicações com limitações de memória e que trabalham com grande quantidade de dados. Quando comparados à busca exata, ambos os métodos aproximados obtêm melhorias significantes em desempenho. Enquanto o tempo de execução da busca exata, utilizando a biblioteca Yael [Douze and Jégou 2014], é de 212 segundos, para uma precisão de 98% o PQANNS leva 31 segundos.

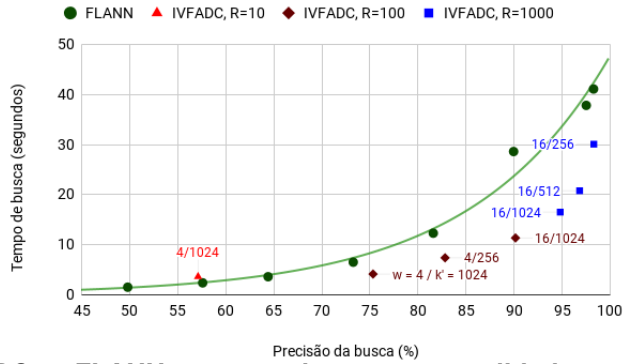
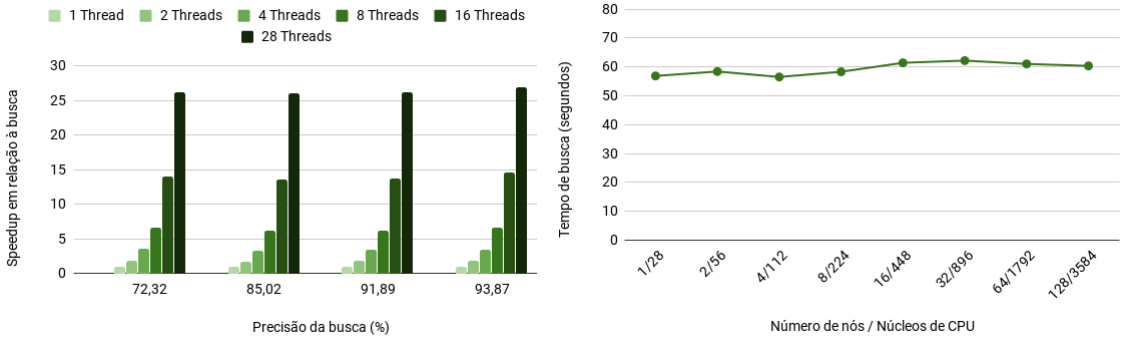


Figura 2. IVFADC vs FLANN: compromissos entre qualidade e tempo de busca.

5.2. Avaliação de Escalabilidade em Ambientes Multicore e de Memória Distribuída

A avaliação da nossa implementação em máquina multicore consiste na execução de testes variando o número de *threads* e a precisão da busca para se verificar o impacto no tempo de execução. Para isso foram realizadas 10 mil consultas em uma base de dados contendo 1 milhão de vetores SIFT, utilizando uma única máquina contendo 28 núcleos. A Figura 3a mostra a variação do *speedup* sobre o algoritmo sequencial atingido pela aplicação em diversos níveis de precisão, enquanto é variado o número de *threads*. Em todos os casos nota-se um ganho próximo do linear.



(a) Escalabilidade multicore. **(b)** Escalabilidade mem. distribuída.

Figura 3. Escalabilidade da paralelização. Em memória distribuída, o tamanho da base de dados é proporcional a quantidade de nós (*weakscaling*).

Esta subseção avalia a escalabilidade da nossa implementação paralela em memória distribuída do PQANNS. O experimento foi executado utilizando nossa maior base de dados, que contém 256 bilhões de descritores SIFT. O algoritmo foi configurado para utilizar 8192 coarse centroids e w = 4, a base de dados de treinamento contém 50 milhões de descritores e foram realizadas 10 mil consultas. Com essa configuração, os resultados obtidos atingem cerca de 80% de precisão.

O experimento foi realizado utilizando uma avaliação de escalabilidade fraca, isto é, o tamanho da base de dados e o número de nós são incrementados na mesma taxa. Dessa forma, cada nó guarda 2 bilhões de descritores SIFT e 256 bilhões de vetores são utilizados no experimento com 128 nós. Nesse domínio, a avaliação a partir da escalabilidade fraca é mais apropriada que o típico experimento de escalabilidade forte, devido à quantidade massiva e crescente de dados que a indexação deve conseguir lidar.

A Figura 3b apresenta os tempos de execução da busca com o crescimento do ta-

manho da base de dados e o número de nós. A aplicação escalou muito bem, obtendo uma eficiência de cerca de 0.97 (97%) com 128 nós se comparado com a execução utilizando apenas uma máquina. Diferentemente de outras paralelizações, como as do algoritmo LSH [Stupar et al. 2010, Bahmani et al. 2012], a nossa preserva o comportamento do algoritmo sequencial, sem fazer replicações da base ou impor limitações de comunicação. O baixo tráfego de rede (menos de 2.5MB/s com 128 nós) indica que o algoritmo continuará escalável se uma quantidade muito maior de nós for utilizada.

6. Conclusão

Esse projeto endereça o problema da busca em vizinhos mais próximos em espaços de alta dimensionalidade, um problema central em algoritmos de visão computacional e inteligência artificial, sendo muitas vezes a parte mais cara desses algoritmos. O PQANNS consegue melhorar o desempenho e reduzir o consumo de memória em relação à busca exata, no entanto sua abordagem sequencial não consegue lidar com grandes bases de dados e um volume realista de consultas. Por isso, nós propusemos uma paralelização do algoritmo, que distribui a computação entre diversas máquinas e permite concorrência interna. A paralelização do algoritmo permite a realização de buscas em grandes bases, sem as limitações de memória de apenas uma máquina. As avaliações da Seção 6 mostram a alta escalabilidade do algoritmo, que realiza a busca em uma base contendo 256 bilhões de vetores e mantém as características do algoritmo sequencial, sem sobrecarregar o tráfego da rede pela troca de mensagens.

Referências

Andrade, G., Fernandes, A., Gomes, J. M., Ferreira, R., and Teodoro, G. (2019). Large-scale parallel similarity search with product quantization for online multimedia services. *J. Parallel Distrib. Comput.*, 125:81–92.

Bahmani, B., Goel, A., and Shinde, R. (2012). Efficient distributed locality sensitive hashing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2174–2178. ACM.

Beygelzimer, A., Kakade, S., and Langford, J. (2006). Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM.

Böhm, C., Berchtold, S., and Keim, D. A. (2001). Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)*, 33(3):322–373.

Douze, M. and Jégou, H. (2014). The yael library. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 687–690. ACM.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226.

Gionis, A., Indyk, P., Motwani, R., et al. (1999). Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529.

Jain, M. (2014). *Enhanced image and video representation for visual recognition*. PhD thesis, Université Rennes 1.

Jegou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128.

Muja, M. and Lowe, D. G. (2009). Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2.

Silpa-Anan, C. and Hartley, R. (2008). Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE.

Stupar, A., Michel, S., and Schenkel, R. (2010). Rankreduce-processing k-nearest neighbor queries on top of mapreduce. *Large-Scale Distributed Systems for Information Retrieval*, 15.

Valle, E., Cord, M., and Philipp-Foliguet, S. (2008). High-dimensional descriptor indexing for large multimedia databases. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 739–748. ACM.