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] 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 *Locality-Sensitive Hashing*

(LSH) [Gionis et al. 1999, Durmaz and Bilge 2019] e o *Product Quantization for Approximate Nearest Neighbors Search* [Jegou et al. 2011, Johnson et al. 2019].

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 utilizam árvores k-d [Friedman et al. 1977, Beis and Lowe 1997], árvores *k-means* [Fukunaga and Narendra 1975, Muja and Lowe 2009], árvores de cobertura [Beygelzimer et al. 2006], dentre outros.

O trabalho de Friedman *et al.* introduz uma forma de utilizar árvores k-d em buscas por vizinhos mais próximos [Friedman et al. 1977]. A árvore k-d é binária, em que cada nó representa uma partição de um arquivo e a raiz representa o arquivo inteiro. Excetuando-se as folhas, cada nó tem dois filhos, que representam dois subarquivos definidos pelo particionamento do arquivo do nó pai. As folhas representam pequenos subconjuntos mutualmente exclusivos que coletivamente formam o arquivo completo. Esses subconjuntos são denominados baldes.

Em bases de dados, a árvore k-d pode ser utilizada para organizar os dados, os distribuindo entre os baldes. Os dados de uma base de dimensão k são representados por k chaves, o particionamento nos nós da árvore k-d utiliza uma dessas chaves para selecionar os dados, chamado de descriminador. Além disso, é preciso determinar um valor de particionamento, que será utilizado para agregar os dados com a chave menor que seu valor em um nó e os com a chave maior em outro. Friedman propõe que o descriminador de cada nó não-folha seja a chave com maior propagação entre os dados daquele nó e que o valor de particionamento seja a mediana dos descriminadores.

Essa estrutura permite que a busca seja realizada em alguns baldes, ao invés de ser exaustiva. Isso reduz o custo computacional médio para $kNlogN$, onde k são as

dimensões do vetor e N o número de vetores na base. Beis e Lowe expandem a utilização de árvores k-d para o domínio de buscas aproximadas [Beis and Lowe 1997] e propõem uma estratégia de busca dos melhores conjuntos da árvore binária.

Outro método é o uso de árvores *k-means*, proposto por Fukunaga e Narendra [Fukunaga and Narendra 1975]. Assim como nas árvores k-d, essa estrutura tem como objetivo particionar a base de dados de forma em que os nós folhas contenham pequenos grupos de dados disjuntos que representam a base toda. No caso das árvores *k-means*, cada nó é dividido em l nós filhos. O particionamento então é feito utilizando o algoritmo de clusterização *k-means*, de forma em que cada nó contenha os dados de um cluster.

A agregação dos dados em clusters permite que a busca seja feita em parte da base, reduzindo o custo em comparação à busca exautiva. Posteriormente Muja e Lowe propuseram um algoritmo [Muja and Lowe 2009] que, de forma semelhante ao proposto em [Beis and Lowe 1997], implementa uma estratégia de busca que prioriza a busca nos melhores conjuntos da árvore.

No entanto a "maldição da dimensionalidade" ainda é um problema, já que a redução do espaço de busca só minimizou o problema com bases grandes. Diversos trabalhos propuseram a realização de busca aproximada associada ao uso de estruturas de dados, como em FLANN [Muja and Lowe 2009, Muja and Lowe 2012, Muja and Lowe 2014, Muja 2013], 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 o alvo de paralelização desse trabalho, esse algoritmo é explicado em detalhes na Seção 3.

2.1. Soluções Distribuídas para Indexação e Busca Aproximada

A busca por similaridade deve ser capaz de trabalhar com bases de dados que crescem rápido conforme o tempo passa, oferencendo tempos de resposta baixos para o usuário final. Essas demandas incentivaram o desenvolvimento de métodos que utilizem técnicas de alto desempenho, bem como implementações escaláveis em ambientes distribuídos [Muja and Lowe 2014, Stupar et al. 2010, Bahmani et al. 2012, Johnson et al. 2017, Moise et al. 2013, Andrade et al. 2017, Wieschollek et al. 2016, Teixeira et al. 2013, Silva et al. 2014].

Dentre os trabalhos propostos, pode-se destacar as paralelizações do LSH utilizando o MapReduce [Stupar et al. 2010, Bahmani et al. 2012] e a versão paralela do FLANN em cima do *Message Passing Interface* (MPI) [Muja and Lowe 2014, Gropp et al. 1999].

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] implementam outra versão baseada em MapReduce, chamada de *Layered* LSH. Foram implementadas duas versões do LSH utilizando: 1) Hadoop para armazenamento baseado em sistema de arquivos e 2) *Distributed Hash Tables* (DHT) ativa para o armazenamento em memória. São propostos limites teóricos para o tráfego da rede assumindo que uma única tabela de *hash* LSH é usada. Essa suposição simplifica a análise e implementação do algoritmo, mas pode ser não-realista já que o LSH geralmente atinge alta eficiência com o uso de múltiplas tabelas *hash* [Gionis et al. 1999]. Se forem utilizadas múltiplas tabelas *hash*, o mesmo objeto é indexado por múltiplos baldes de diferentes tabelas e, como consequência, a partição dos dados não seria tão simples quanto com apenas uma tabela.

Dessa forma, nenhuma das paralelizações utilizando *MapReduce* resolvem o problema de construir um índice de busca de larga escala que minimize a comunicação e evite replicação de dados, enquanto preserva o comportamento do algoritmo sequencial e provê baixo tempo de resposta às consultas.

Já Muja *et al.* propõem uma versão paralela do FLANN em cima do MPI [Muja and Lowe 2014]. Sua abordagem segue uma estratégia semelhante ao *MapReduce*, os dados são distribuídos entre múltiplas máquinas de um cluster e a busca por vizinhos mais próxima é executada paralelamente. Os dados são distribuídos igualmente entre as máquinas, de forma que em um cluster com N máquinas, cada um deles tenha que indexar e buscar em apenas $1/N$ dos dados. O resultado final é obtido pela agregação dos resultados parciais de cada máquina, uma vez que ela tenha terminado.

Cada processo MPI executa em paralelo e lê de um sistema de arquivos distribuído uma fração da base de dados. Todos os processos constroem seu índice paralelamente, usando sua fração dos dados. Para realizar a busca no índice distribuído, a consulta é enviada de um cliente para um dos computadores do cluster, chamado de servidor mestre.

O nó mestre faz um broadcast da consulta para todos os processos do cluster e cada nó pode realizar a busca paralelamente em sua fração da base. Quando a busca é terminada, uma operação de redução é utilizada para agregar o resultado final no nó mestre. A redução é realizada por pares num sistema hierárquico até retornar ao nó mestre, o que distribui a computação entre as máquinas de forma eficiente.

Essa implementação exige uma grande largura de banda de memória devido aos algoritmos empregados no FLANN, o que facilmente satura as demandas de largura de banda de memória e limita sua escalabilidade.

Implementações do PQANNS baseadas em GPU foram propostas [Johnson et al. 2017, Wakatani and Murakami 2014], mas são limitadas a execução em uma máquina e não conseguem lidar com grandes bases de dados. Por fim, o trabalho de Moise *at. al* [Moise et al. 2013] apresenta uma estratégia de indexação baseada em árvores para indexar os dados, capaz de realizar buscas em bases de dados com mais de 30 bilhões de descritores. Uma comparação das abordagens paralelas disponíveis está sumarizada na tabela 1, primeiro apresentada em [Andrade et al. 2019].

Tabela 1. Visão geral dos métodos de busca aproximada de vizinhos mais próximos em memória distríbuída discutidos nessa seção. Nesse trabalho foi utilizada a maior base de dados em que os autores reportaram o tempo de execução.

Trabalho	Algoritmo empregado	Tamanho da base de dados	# de nós	Tempo de execução da consulta
[Stupar et al. 2010]	LSH	100K	1	65s
[Bahmani et al. 2012]	LSH	1M	16	15ms
[Muja and Lowe 2014]	FLANN	80M	4	N/A
[Teodoro et al. 2011]	Multicurves	130M	8	2ms
[Moise et al. 2013]	Index Tree	7.8B	100	134ms
Este	PQANNS	256B	128	7ms

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:

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

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 (Figura 1), 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.

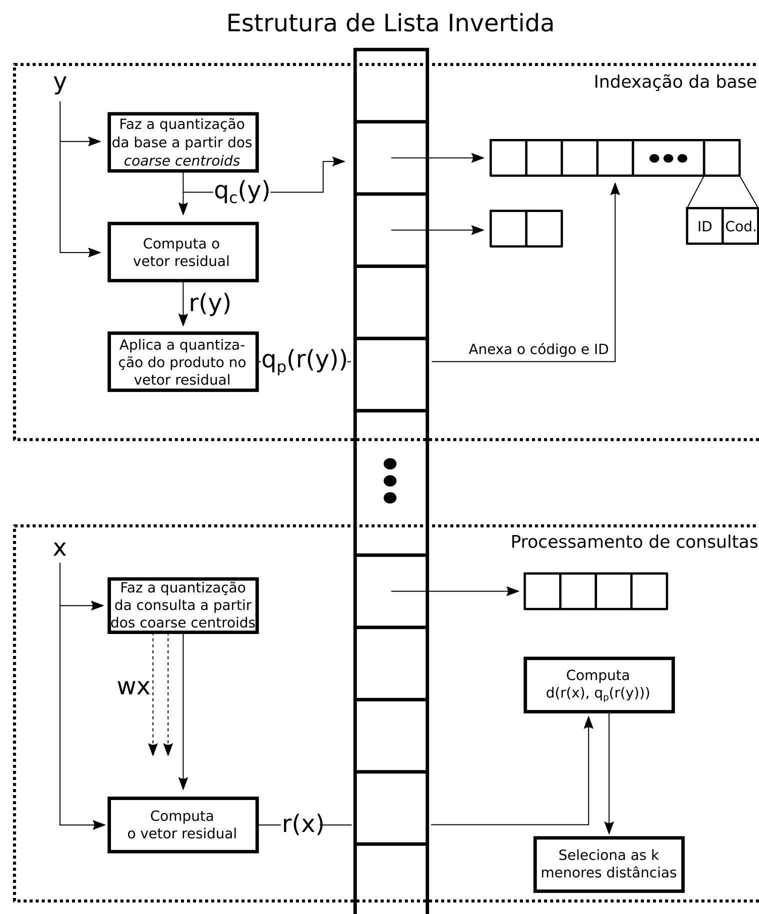

Figura 1. Visão geral do sistema de lista invertida com cálculo assimétrico de distância.

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. Dado um vetor y da base de dados, o algoritmo computa os seguintes passos:

1. quantiza $y : q_c(y)$;
2. computa o vetor residual $(r(y))$ a partir do vetor quantizado $(q_c(y))$ e o descritor $r(y) = y - q_c(y)$;

3. quantiza $r(y)$ para $q_p(r(y))$;
4. insere o novo item na entrada da lista invertida correspondente.

A fase de busca dos vizinhos mais próximos de x é executada a partir dos 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.

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 2) e são distribuídos em dois fluxos de execução, que fazem a construção do índice e a busca na base.

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

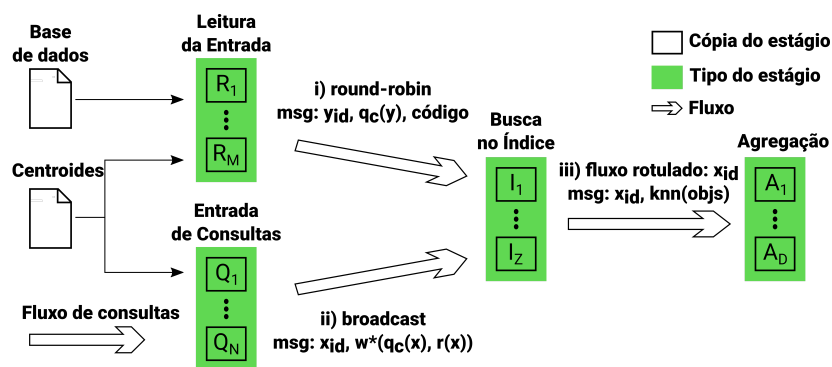

Figura 2. Arquitetura da paralelização.

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_id), 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 Figura 2).

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 [Chandra et al. 2001], versão 3.1. Para explorar paralelismo nesse nível, dividimos o processamento dos vetores de consulta entre as threads

disponíveis para execução.

4.1. Paralelização em Máquina Multicore

No estágio de Busca no Índice é feita a construção dos índices locais, bem como é executada a busca em cada nó. Ambos os processos são caros computacionalmente e por isso exigem a adoção de alguma estratégia de paralelização interna que permita a execução concorrente entre os núcleos da CPU.

Em busca de reduzir o consumo de memória e permitir a execução concorrente da construção do índice. O algoritmo de "Construção do Índice em Máquina Multicore", divide o trecho da base a ser indexado pelo estágio de Busca no Índice em trechos menores e as *threads* fazem a construção simultânea do índice de trechos da base assinalada para aquele processo, o armazenando em uma estrutura de lista invertida local (linha 4). Ao fim da construção de cada índice local, a *thread* responsável por sua criação o adiciona na lista invertida global da máquina (linha 6).

Algoritmo 1: CONSTRUÇÃO DO ÍNDICE EM MÁQUINA MULTICORE

Entrada: conjunto de dados D, quantidade de threads $threads$
Saída: lista invertida L

1 $L \leftarrow$ lista invertida vazia
2 $\#pragma$ omp parallel for num_threads(threads) schedule(dynamic)
3 **para** *cada trecho* $D' \in D$ **faça**
4 $\quad L' \leftarrow indexa(D')$
5 $\quad \#pragma$ omp critical
6 $\quad L \leftarrow concatena(L')$
7 **fim**
8 **retorna** L

Essa abordagem permite um uso melhor dos recursos computacionais, pois distribui a execução entre os núcleos da CPU assinalada para esse estágio, fazendo uso de todo o poder de processamento disponível. Além disso, a divisão do trecho da base em "pedaços menores" permite que ao fim da inclusão de cada pedaço no índice, os vetores originais possam ser descartados da memória. Isso permite a redução no consumo de memória de $d * n$ para $d * n' * n_{threads}$, em que d é a dimensão dos vetores da base, n é o tamanho do trecho da base, n' é o tamanho do pedaço menor da base e $n_{threads}$ é o número de *threads* executando concorrentemente.

Na realização da busca foi adotada uma estratégia um pouco diferente. No algoritmo de "Busca em Máquina Multicore", uma *thread* fica responsável por fazer a comunicação com o estágio de Agregação, enquanto o restante realiza a busca. Cada *thread* de busca assume a computação de um vetor de consulta e insere o resultado em um buffer (linhas 14 e 15), quando o buffer utrapassa uma quantidade de vetores processados ou quando não há mais consultas a serem processadas, a *thread* de comunicação envia os resultados para a cópia de Agregação correspondente àquela consulta (linha 6).

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

Algoritmo 2: BUSCA EM MÁQUINA MULTICORE

```
    Entrada: consultas, número de consultas consultas_num, número de
             threads threads_num
 1  buffer ← ∅
 2  #pragma omp parallel num_threads(threads_num)
 3  se thread_id == threads_num − 1 então
 4  |   enquanto 1 faça
 5  |   |   se consultas == ∅ || buffer está cheio então
 6  |   |   |   enviar_agregador(buffer)
 7  |   |   |   se consultas == ∅ então
 8  |   |   |   |   break
 9  |   |   |   fim
10  |   |   fim
11  |   fim
12  senão
13  |   para i = thread_id; i < consultas_num; i+ = threads_num − 1 faça
14  |   |   knn_local ← PQANNS(consultas[i])
15  |   |   buffer ← adicione(knn_local)
16  |   fim
17  fim
```

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.

Foram realizados três testes utilizando o algoritmo proposto neste trabalho. Nesta seção serão apresentados os resultados obtidos e uma discussão acerca da análise dos mesmos.

O primeiro teste busca avaliar a qualidade da nossa implementação sequencial do PQANNS em comparação com a implementação sequencial do FLANN, um dos principais métodos da literatura. Utilizando uma base de 1 milhão de vetores SIFT, ambos os algoritmos foram executados com as melhores configurações para diversos níveis de precisão. Com isso pode-se comparar o tempo de execução e uso de memória de ambos na realização de uma busca de mesma qualidade.

O segundo teste consiste na avaliação da escalabilidade em processadores multicore, usando uma base de 1 milhão de vetores SIFT e variando o número de núcleos de processamento até atingir 28 nós. A partir desse teste é possivel avaliar o impacto da paralelização interna no tempo de execução do algoritmo.

Por fim foram executados testes de escalabilidade horizontal da nossa implementação em memória distribuída. Variando o tamanho da base proporcionalmente ao número de nós de busca, foram realizadas 10 mil consultas utilizando até 128 nós em uma base 256 bilhões de vetores SIFT. A análise do tempo de busca nessas condições permite a avaliação da escalabilidade fraca do algoritmo.

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.

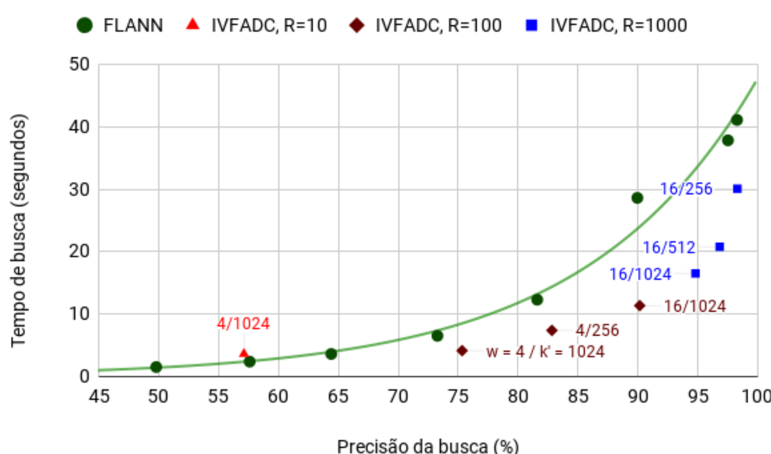

Figura 3. IVFADC vs FLANN: compromissos entre qualidade e tempo de busca.

O resultado experimental comparando os tempos de busca do IVFADC/PQANNS e do FLANN enquanto a precisão é variada é mostrado na Figura 3. 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.

5.2. Avaliação de Escalabilidade em Ambientes Multicore

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 4 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.

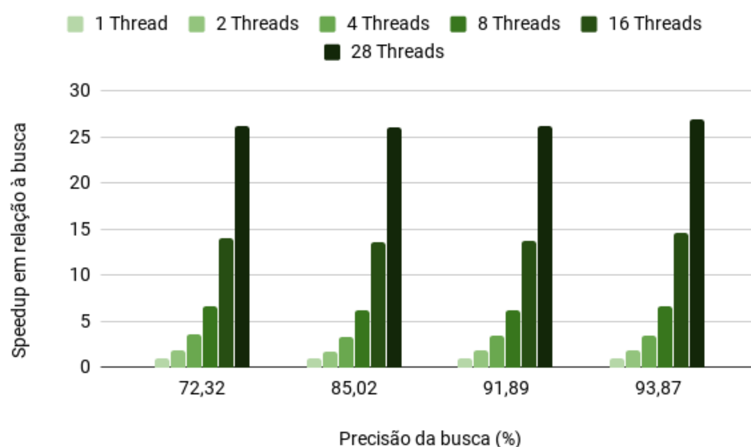

Figura 4. Escalabilidade em ambiente multicore.

5.3. Avaliação de Escalabilidade em Ambientes de Memória Distribuída

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 5 apresenta os tempos de execução da busca com o crescimento do tamanho 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 (Figura 6)

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

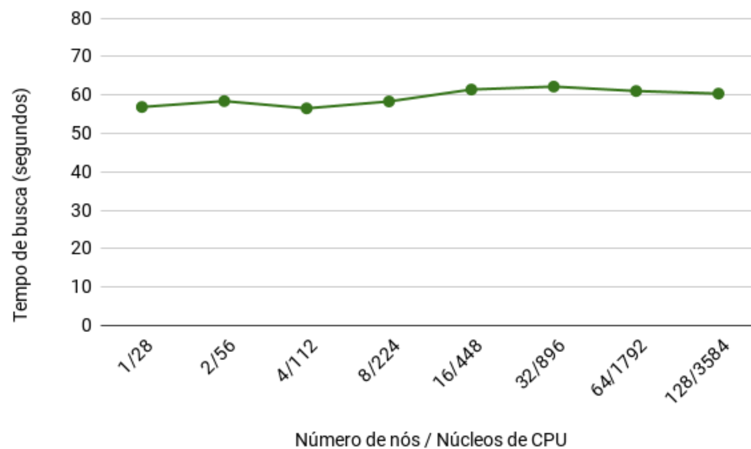

Figura 5. Escalabilidade em memória distribuída: tamanho da base de dados é proporcional a quantidade de nós ($weakscaling$).

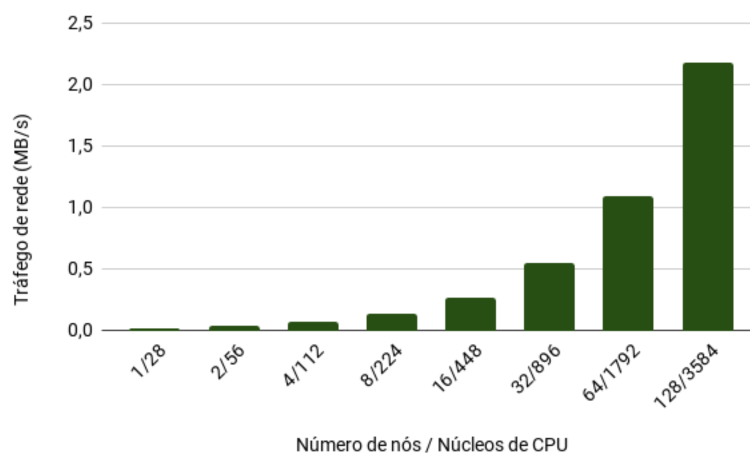

Figura 6. Tráfego de rede (MB/s) em relação à variação do número de nós em um experimento de escalabilidade fraca e uma base de dados contendo 256 bilhões de descritores SIFT na configuração com 128 nós.

exata e, se comparado ao estado da arte (FLANN), tem desempenho melhor em quase todas as configurações, utilizando cerca de 20 vezes menos memória para armazenar sua estrutura de indexação.

No entanto a implementação sequencial do PQANNS não consegue lidar com grandes bases de dados e um volume realista de consultas. Por isso, nesse trabalho foi proposta uma paralelização do algoritmo, que distribui a computação entre diversas máquinas e permite concorrência interna.

A avaliação dessa abordagem em um cluster com 128 mostrou sua ótima escalabilidade, com uma eficiência de cerca de 0,97, sendo capaz de indexar uma base de dados de 256 bilhões de vetores. Essa base de dados é cerca de 8,5 vezes maior que a maior base de dados empregada em trabalhos relacionados [Moise et al. 2013] (30 bilhões de descritores), e foi capaz de responder a consultas em 7ms nessa configuração.

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.

Andrade, G., Teodoro, G., and Ferreira, R. (2017). Online Multimedia Similarity Search with Response Time-Aware Parallelism and Task Granularity Auto-Tuning. In *29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017*, pages 153–160.

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.

Beis, J. S. and Lowe, D. G. (1997). Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *cvpr*, page 1000. IEEE.

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.

Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

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

Durmaz, O. and Bilge, H. S. (2019). Fast image similarity search by distributed locality sensitive hashing. *Pattern Recognition Letters*, 128:361–369.

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.

Fukunaga, K. and Narendra, P. M. (1975). A branch and bound algorithm for computing k-nearest neighbors. *IEEE transactions on computers*, 100(7):750–753.

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

Gropp, W., Gropp, W. D., Lusk, E., Lusk, A. D. F. E. E., and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press.

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.

Johnson, J., Douze, M., and Jégou, H. (2017). Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*.

Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*.

Moise, D., Shestakov, D., Gudmundsson, G., and Amsaleg, L. (2013). Indexing and searching 100m images with map-reduce. In *Proceedings of the 3rd ACM conference on International conference on multimedia retrieval*, pages 17–24. ACM.

Muja, M. (2013). *Scalable nearest neighbour methods for high dimensional data*. PhD thesis, University of British Columbia.

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

Muja, M. and Lowe, D. G. (2012). Fast matching of binary features. In *Computer and Robot Vision (CRV), 2012 Ninth Conference on*, pages 404–410. IEEE.

Muja, M. and Lowe, D. G. (2014). Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (11):2227–2240.

Silva, E., Teixeira, T., Teodoro, G., and Valle, E. (2014). Large-scale distributed locality-sensitive hashing for general metric data. In Traina, A. J. M., Traina, C., and Cordeiro, R. L. F., editors, *Similarity Search and Applications*, pages 82–93. Springer International Publishing.

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.

Teixeira, T. S. F. X., Teodoro, G., Valle, E., and Saltz, J. H. (2013). Scalable Locality-Sensitive Hashing for Similarity Search in High-Dimensional, Large-Scale Multimedia Datasets. *CoRR*, abs/1310.4136:1–20.

Teodoro, G., Valle, E., Mariano, N., Torres, R., and Meira Jr, W. (2011). Adaptive parallel approximate similarity search for responsive multimedia retrieval. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 495–504.

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.

Wakatani, A. and Murakami, A. (2014). Gpgpu implementation of nearest neighbor search with product quantization. In *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 248–253. IEEE.

Wieschollek, P., Wang, O., Sorkine-Hornung, A., and Lensch, H. (2016). Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027–2035.